Ontology-based Service Discovery in a Globally Distributed Network

Knarig Arabshian
Advisor Henning Schulzrinne

Submitted in partial fulfillment of the
Requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2008

Abstract

Ontology-based Service Discovery in a Globally Distributed Network

Knarig Arabshian

In order to accomplish context-aware service discovery, static or dynamic service information must be made available globally and represented in such a way that registering and querying for these services can be more complicated than what is available to us now. Imagine a service discovery system where a user can enter the following searches: ”Find me a nearby Italian restaurant that has available seating for 5 people” or ”Find me a Chinese restaurant or something similar to it that has a nearby movie theatre playing an action movie”. This requires efficiently distributing dynamic service information which require frequent updates as well as enhancing querying and registration beyond text and key word search so that specific services as well as logically similar services can be searched for.

This dissertation introduces GloServ, an ontology-based global service discovery system. Two of its main contributions are its scalable network architecture and intelligent querying and registration of services. GloServ aggregates different types of services in a globally distributed network. A partial list of such services include real-time event-based services, location-based services, communication, e-commerce or web services. GloServ is a service discovery architecture that uses a description logic ontology, such as the Web Ontology Language Description Language (OWL DL), for classifying service information and mapping them onto a physical hierarchical peer-to-peer network. GloServ operates on a wide as well as local area network and supports the discovery of all types of services that are described in

an ontology. This classification ontology defines service classes and their relationships with other services and properties. Thus, Gloserv aggregates and classifies service information in a distributed network.

Due to the use of description logic ontologies, GloServ can perform semantic matching of queries to return results that are logically related to the user's request, rather than searching only for exact information using attribute-value matching as current service discovery protocols do. Initially, the GloServ query matchmaking engine represents the query as a temporary class within the service ontology. This class is essentially a first order logic statement which has a number of its properties restricted. The matching engine then uses an ontology reasoner to classify the temporary query class within the ontology. Service information within equivalent classes and subclasses are considered matches. Similar matches are considered to be non-disjoint siblings of the object property values within the query.

Additionally, GloServ further refines the results obtained by combining both ontology-based querying with keyword matching. Since the service ontology may not capture all parts of a particular service, it allows each ontology to have a keyWords property which service providers can populate with keyword terms that describe the specifics of their service. Furthermore, GloServ allows querying of more than one service in a single query by supporting subquerying between its servers, which allows services that share common properties to be composed into a single query. This enhancement causes fairly complicated queries to be issued in a single search such as the one given in the above example of searching for a restaurant and a nearby theatre.

We have built a prototype implementation of GloServ and the results have shown that the underlying distributed architecture improves the query latency and load distribution of query and registration messages. Also, because GloServ uses an ontology to map a network, different service classes are added to the network quite easily as only an ontology configuration file is needed for a service class to be generated and mapped to a network of servers. Furthermore, we have built a web-based front-end for a restaurant and theater service search which demonstrates the different querying methods GloServ allows.

Contents

List of Figures

I dedicate this thesis to my father,

Hovanes Arapshian,

who gave up his dream of pursuing

a career in Computer Science to pave

the way for me to fulfill the same dream.

Dad, thank you. This one's for you.

Acknowledgements

First and foremost, I would like to thank my advisor, Professor Henning Schulzrinne. This work would not have been possible without his mentoring, direction, and support. Professor Schulzrinne always provided the right dose of guidance and freedom, which continually motivated me to explore new grounds. His broad vision and common-sense pragmatism has been one of the greatest influences in my research work. He has taught me, by his own example, that making an impact within the research community entails a precise mixture of creativity and practicality. I also thank him for his devotion to all of his students, carving out weekly meetings, always availing himself whenever necessary, and providing moral support during our personal struggles.

I also acknowledge my proposal and thesis committee members: Luis Gravano, Gail Kaiser, Peter Patel-Schneider, Doree Seligmann, and Ted Shortliffe for their invaluable advice and help in this thesis. I'm also very grateful to Nokia Research for supporting my research work–especially my collaborators, Dirk Trossen and Dana Pavel, who were instrumental in advising me during the early stages of GloServ's development. I thank Michael Wessel from the Racer support team for giving me helpful feedback whenever needed. I thank all of my colleagues at the IRT lab, in particular, Maria Papadopouli, who motivated me to pursue a PhD and Christian Dickmann, my project student who helped launch the GloServ prototype, through his excellent work. I also appreciate all of my friends at Columbia, who were always ready to lend a listening ear and provide encouragement throughout the years.

I would never have completed this degree without the support of my family and friends. I'm forever grateful to my mother, Dzovinar, for her infinite, steadfast, love and devotion throughout my life and especially for believing in me as I trekked through the PhD. I'm also deeply indebted to my father, Hovanes, whose creative and inventive nature influenced me to delve into research in the first place. My heart-felt thanks goes to my sister, Alice, my hero, who lives life passionately and valiantly despite many obstacles and inspires me to do the same. I also thank the rest of my (very large) extended family for always rooting for me through the years. In particular, Aunty Maro, for being a second mother to me with her endless generosity and love; Aunty Elmon, for being my spiritual mentor and confidante; Uncle Apar, who has always encouraged me to pursue greater heights; my cousin, Siran, herself a PhD, who pioneered the path of academe in our family and encouraged me to follow suit; my cousin, Kapo, whose unquenchable thirst for knowledge continually piqued my intellectual curiosity; my grandparents, Knarig, Bartukh, Aghavni and Mardiros, genocide survivors, brilliant, self-taught individuals, who have passed down the love of knowledge to their descendants. I am profoundly blessed by my two dear friends, Kundan and Heather, who always cheered me on and encouraged me to seek excellence in my work. I am also very grateful for my church community at Redeemer, my pastors, Tim Keller and Terry Gyger, as well as my fellowship groups 108HFG, AFG and AEYF+ for their spiritual and moral support as well as their wisdom.

Part I

Introduction and Background

Chapter 1

Introduction

1.1 Overview

The goal of *Ubiquitous Computing* systems is to disappear into the fabric of our everyday lives [125] [115]. Ubiquitous computing was coined by Mark Weiser who envisioned different devices interacting with each other in ways that would not be noticeable to the average person, but would enhance people's lives. An example is when sensor devices in a home detect a user's location in the home and control lighting or temperature of the room accordingly. A key component of ubiquitous computing is context-aware service discovery where systems detect people's contexts and discover services matching the person's context. For example, if someone is looking for a restaurant, the system should return a list of restaurants based on the person's location. There are many different types of services that can be made available for automatic discovery. A partial list of such services include real-time event-based services, location-based services, communication, e-commerce or web services. This thesis addresses the problem of context-aware service discovery.

Two aspects of context-aware service discovery need to be addressed in order for it to function in a ubiquitous computing environment. First, information must be made available globally. The underlying architecture should be designed in such a way that information is retrieved and updated efficiently. We envision including not only static services, but dynamic in nature. For instance, a restaurant may want to display the number of people it can seat during its peak hours of operation. These values can change every few minutes. For example, a tourist who is searching for an Italian restaurant in the busy Times Square district right before or after a Broadway show will have a hard time finding a place without a reservation. She will send out a query for a restaurant available to seat five people. As the restaurants automatically update their information, when one becomes available near her location, she will be immediately notified.

Second, information needs to be represented with richer semantics so that registering and querying for services can be more feature-rich than what is available to us now. This means enhancing querying and registration beyond even attribute-value pair or text search so that specific services as well as logically similar services can be searched for. Google, for example, works quite well for simple text searches such as finding an *Italian restaurant in the Upper West Side neighborhood of New York City*. The first few results match the text query as seen in Figure 1.1. On the other hand, if we complicate the search to include a price range in the query, where we want to search for a restaurant that offers a meal under thirty dollars, the results are not as accurate. As can be seen in Figure 1.2, the first few links are a result of searching for text terms within the query and do not make as much sense.

Aside from searching for specific items, users may want to search for items that are logically similar to what they specify. Taking the above example, if one is searching for an Italian restaurant within the thirty dollar price range, a similar restaurant may be a restaurant serving European or Mediterranean cuisines.

The recent emergence of the Semantic Web [93] has also contributed to enhancing the goal of ubiquitous computing and, in particular, context-aware service discovery. The Semantic Web originated with Tim Berners Lee, the inventor of the World Wide Web and Hyper Text Transfer Protocol (HTTP) [79]. The Semantic Web envisions connecting all types of data sources via Uniform Resource Identifiers (URI) [60]. A URI is "a compact

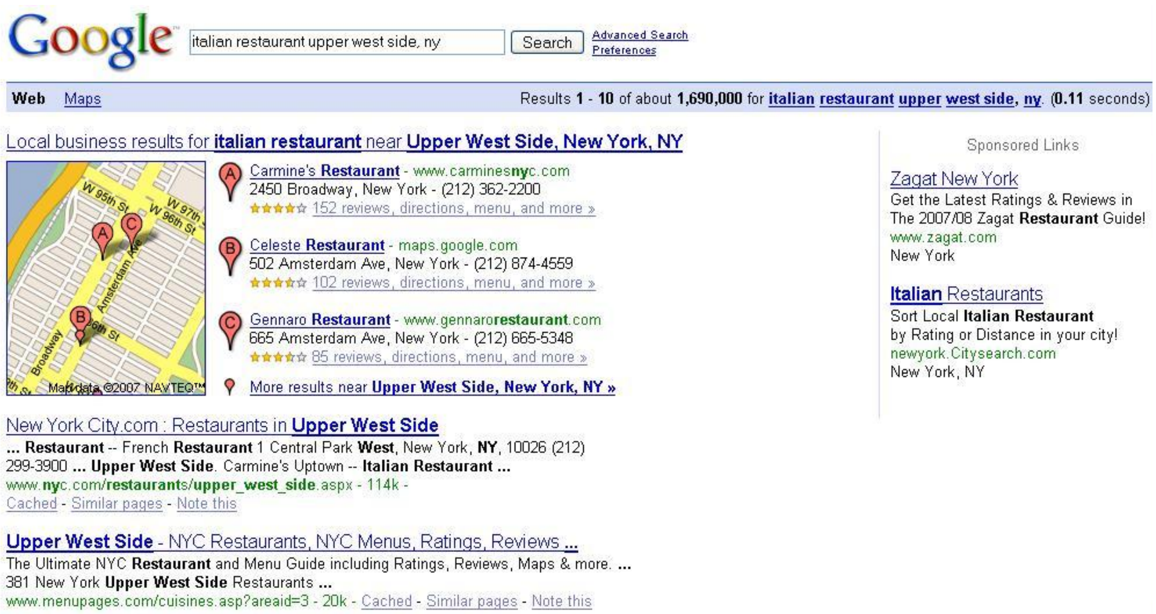

Figure 1.1: Searching for Italian restaurants in Upper West Side, NY using Google

string of characters for identifying an abstract or physical resource”. Data is used by web applications themselves but direct access to ”deep web” data may not be possible. The deep web is content that is not part of the surface Web indexed by search engines [92]. Examples of this type of structured data are online services such as social networking, e-commerce, or location-based services. The semantic web seeks to create a web of data that connects this raw structured data within databases so that independent applications can access them automatically. Thus, whereas the WWW connects web pages together in mostly unstructured ways using information retrieval techniques to index web pages, the semantic web seeks to connect data within databases in structured ways by using semantic data for information sharing. The World Wide Web Consortium (W3C) [41] defines the semantic web as:

It is about common formats for integration and combination of data drawn from diverse sources, where the original Web mainly concentrated on the interchange of documents. It is also about language for recording how the data relates to real world objects. That allows a person, or a machine, to start off in one database, and then move through an unending set

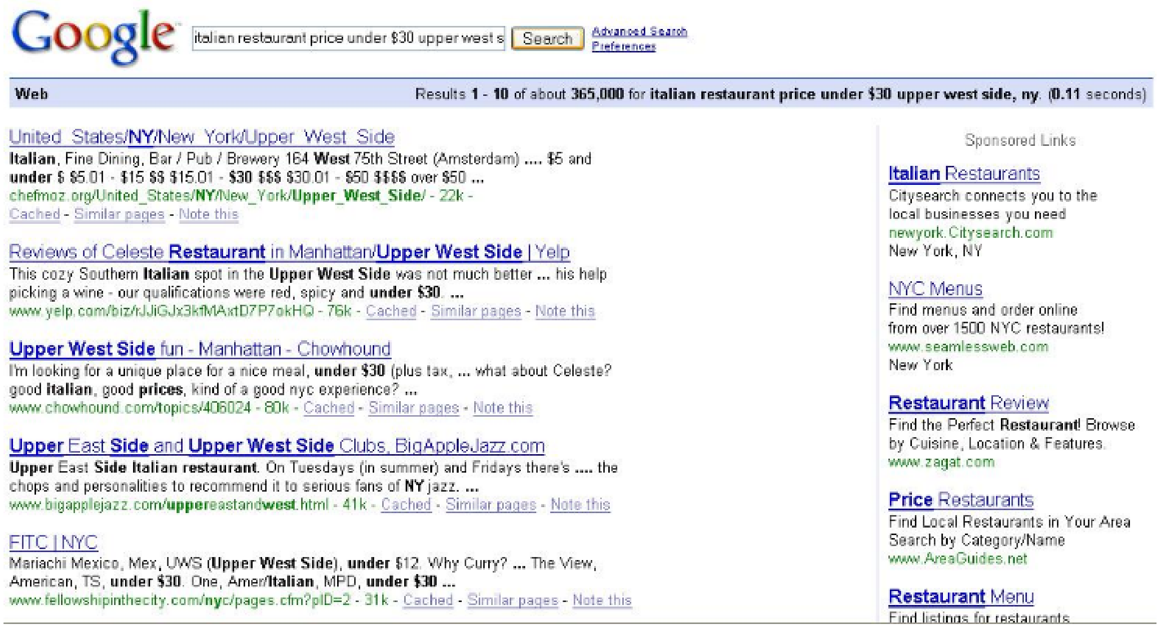

Figure 1.2: Searching for Italian restaurants in a certain price range using Google

of databases which are connected not by wires but by being about the same thing. [62] Thus, the Semantic Web seeks to integrate data across different domains, a key component in creating a ubiquitous computing environment.

The problem with integrating data today is that databases are defined differently across different domains. If a program wants to compare or combine information across databases, it needs to know which identifiers, such as database field names, mean the same thing even if they are defined differently. For example, an *SMS* and a *text message* have the same meaning but these are defined with different database field names in different sites.

Data integration can be accomplished by using ontologies. An *ontology* is a term that originated in philosophy which describes the nature of existence [87]. This term was adopted within the field of knowledge representation, which is a sub-field of artificial intelligence, to represent knowledge by formally defining relations among terms. Practically, this means defining a classification of concepts (classes) and the relations among them (properties). For example, within the food domain, a class *Cuisine* is defined. *Cuisine* may have as its subclasses regional foods (*Italian* or *Chinese*) or types of ingredients (*Vegetarian* or *Spicy*).

A class can be defined by restricting a number of its properties to a set of values. Inference rules within an ontology can give further reasoning power. For example, if a cuisine class such as *SpicyVegetarian* is defined to have its properties restricted to spicy and vegetarian, it is automatically classified under the *Spicy* and *Vegetarian* cuisine classes. Furthermore, when systems use ontologies to represent data, queries can return not only exact results but similar ones. Thus, answering a query such as ”Find me things that are similar to A” reduces to finding classes that are related to ’A’ either by parent, child or sibling relationships. Chapter 2 goes into greater detail of how ontologies are constructed and reasoned with.

1.2 Motivation

Traditional service discovery systems are deficient in two main areas: network architecture and service descriptions. Service discovery systems such as UDDI [45], Jini [101] or SLP [89] are limited in network scalability because they rely on a centralized client-server model or pure hierarchical architecture. Because of this, data cannot be distributed on a global scale or be updated frequently, in case of dynamic services. These systems also describe services by simple attribute-value pairs, which limit the types of searches to exact matches and also prohibits the integration of data across different domains. Other service discovery systems such as Google Local [18], Yelp [43], or Yahoo Yellow Pages [42] also have limitations in that the queries are limited to text searches. Below, I discuss a few service discovery applications which motivate the need for a service discovery system which scales globally as well as represents data in a richer format.

1.2.1 Location-Based Services

Location-Based Services (LBS) ”are business and consumer services that give users a set of services starting from the geographic location of the client” [31]. Consider location-based service discovery for restaurants, theaters, and traffic reports. Imagine a couple is planning an evening out in New York City who would like to find a seafood restaurant near the waterfront in Manhattan that also has a theater nearby playing an action movie. In order to find the quickest route to the restaurant, the closest restaurant that has the least congested route needs to be found. The restaurant should also be located near a movie theater which is featuring action movies with showtimes that are some time after dinner is over.

In order to search for the scenario above, location data must be combined across the different domains. First, seafood restaurants are searched for in New York City which have an additional attribute of being located near a waterfront. The location of these restaurants are then fed into a traffic service network which returns the best route to the restaurant. These locations of these restaurants are chosen and then fed into the theater service network along with attributes for movie and showtimes. The results shown will be all the restaurants and nearby theaters which are located in places that offer the least congested route.

Currently, in order to perform such a search, a user must issue multiple queries to restaurant, theater and traffic sites in order to determine the best restaurant and theater to go to. Since data for each of these services is represented in different formats, it is difficult to integrate the data while composing services. Thus, the use of ontologies would greatly aid in performing these types of complicated queries. Furthermore, if we take into account all types of location-based services, global distribution of data becomes necessary.

1.2.2 Social Networking Services

The second scenario involves extending the definition of a service so that a person is represented as a service. For example, social networking applications such as MySpace [26], Friendster [17], LinkedIn [23] and Facebook [12] have attracted millions of users and are becoming a communication tool that is as prevalent as e-mail or text messaging. All provide information about users such as their profile, pictures or blogs. However, each site targets a different niche. MySpace users are mostly teenagers; Facebook began with college students and has quickly expanded to the young adult community; and LinkedIn focuses on professionals. Furthermore, there are many users who are registered in more than one of these networks.

Searching for people within these sites is limited to text or attribute-value searches. But there are many types of searches that can still be done which are not implemented. For example, there is no way to search for users who have similar musical taste, work histories, or travel experiences. Even more interesting would be searching for users across the different social networks. Google's OpenSocial API [19] defines a common API for social applications across multiple websites. However, search is limited to what is provided by the social network sites themselves. As long as data is represented purely in attribute-value fashion, the types of searches that can be performed are limited. However, if a social networking ontology is created which creates a set of common classes across the different networks, then data can be easily integrated and further reasoning can be done over the queries.

Furthermore, imagine if these social networks or other context-aware systems record a user's current physical location and include this as a searchable attribute within the user's profile. If an application on a handheld device automatically updates a person's location every five minutes, suddenly the number of updates necessary for millions of users becomes extremely large and thus a distributed network architecture is needed which can handle large numbers of reads and writes. This is just one example of how a person can be represented as a location-based service.

1.3 Problem Statement

It can be seen from the applications listed above that current service discovery systems do not suffice. Thus, the problem explored in this thesis is of creating a distributed ontology-based service discovery system. The main challenges are the following: 1) designing a system that scales globally and distributes service information in an efficient and fault-tolerant manner; 2) handling frequent updates of dynamic services efficiently; 3) establishing a generic framework which adapts easily to the addition and removal of new service classes; 4) performing intelligent querying such as ontology-based queries, combination of ontology and key word searches, and combining more than one service in a single query.

1.4 Hypotheses

- Classifying and representing services in an ontology results in:
  - greater reasoning power when querying for services such as finding similarity matches in addition to exact matches and combining key word searches with ontology querying as described in Chapters 6 and 10, respectively;
  - simplified data sharing across different domains especially when performing service composition as described in Chapter 9.
- Distributing these ontologies by using the logical classification information within them across a hierarchical peer-to-peer network achieves:
  - global availability of service information in a generic framework that aggregates service information, as described in the architectural design in Chapter 4;
  - efficient access and updates to dynamic service information, as described in the system evaluation in Chapter 8.

1.5 Solutions

I have designed *GloServ* [50], [53], [51], an ontology-based global service discovery architecture. GloServ uses a description logic ontology, such as the Web Ontology Language Description Language (OWL DL) [30], for classifying service information and mapping it onto a physical hierarchical peer-to-peer network. GloServ operates on a wide as well as local area network and supports the discovery of all types of services as long as they are described in an ontology.

1.5.1 Requirements

Ontology

Directories such as yellowpages or business to business directories classify services on a high-level. GloServ also classifies services in this way using a high-level service ontology. Additionally, each service class has its own ontology. We envision that a separate classification system similar to North American Industry Classification System (NAICS) [1] classifies the hierarchy of services and establishes OWL ontologies that describe each type of service. Experts within a domain construct ontologies for each domain and an authority such as ICANN [2] delegates the top level services and events to individual high-level servers.

With this design, GloServ is able to aggregate and classify service information across all classes. Examples of a high-level service classification ontology and an ontology for a given service class such as the *Restaurant* service class are seen in Figures 1.3 and 1.4. The high-level classification arranges service classes in a pure hierarchy where each service class is different than its sibling class. Thus, a service residing in one of the branches will not reside in the other. The *Restaurant* service class has its own ontology with subclasses restricted by a location. It also has three properties, namely, hasPrice, hasNeighborhood and hasCuisine each mapping to the *Price*, *Neighborhood* and *Cuisine* classes, respectively.

Distributed Architecture

GloServ achieves distribution of service information by mapping the service ontologies onto a hierarchical peer-to-peer network. The high-level service classification ontology, also called a *primitive skeleton*, defines each class to be disjoint from its siblings. The ontology contains information for each host and a hierarchical network is formed by connecting nodes

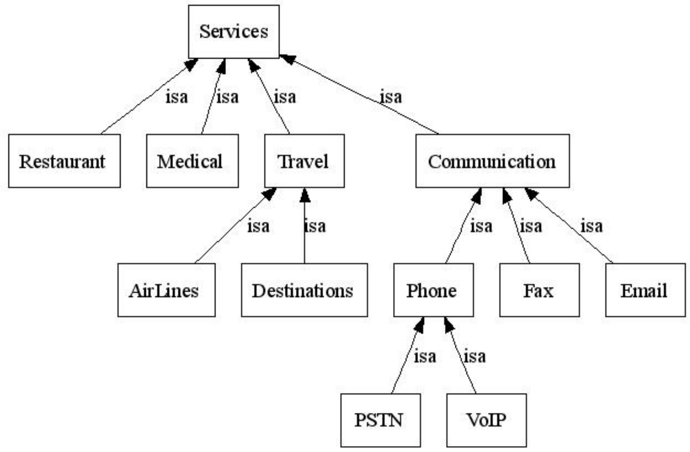

Figure 1.3: A high-level service classification

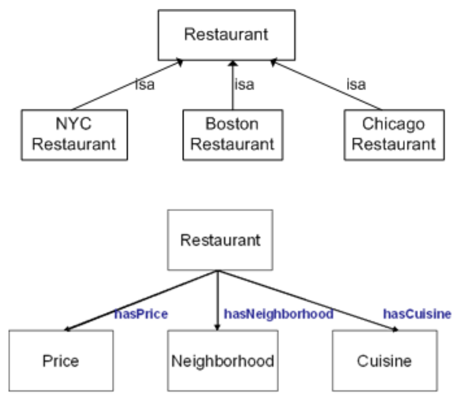

Figure 1.4: A classification for the *Restaurant* service class

which represent each high-level service class to each other in the same way the ontology is constructed.

A peer-to-peer network is formed between subclasses within the high-level service class via a Content Addressable Network (CAN) [110]. In this way, classes that are related to each other are put in a peer-to-peer network. Figure 2.7 gives an overview of the network architecture. Due to the hierarchical peer-to-peer network architecture, dynamic service information can be distributed on a global scale while still balancing the load of frequent queries and registrations, which supports the scenario of updating a person's location described in Section 1.2.2.

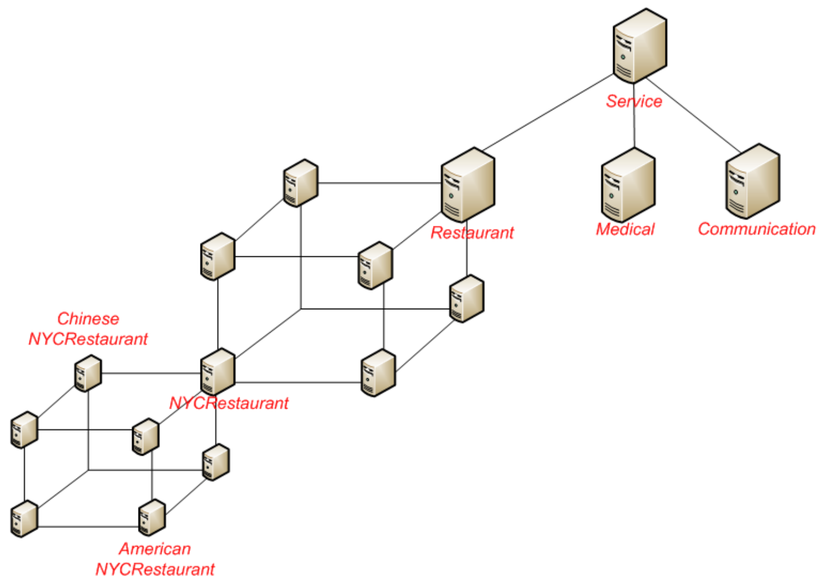

Figure 1.5: GloServ hierarchical peer-to-peer architecture

1.5.2 Contributions

Ontology Querying

Due to the use of description logic ontologies, GloServ can perform semantic matching of queries to return results that are logically related to the user's request, rather than searching only for exact information using attribute-value matching as current service discovery protocols do. Initially, the GloServ query matchmaking engine represents the query as a temporary class within the service ontology. This class is essentially a first order logic statement which has a number of its properties restricted. The matching engine then uses an ontology reasoner to classify the temporary query class within the ontology. Service information within equivalent classes and subclasses is considered a match. For similarity matches, the query engine looks into service information within the non-disjoint sibling classes.

Ontology and Keyword Querying

Additionally, the query engine further refines the results obtained by combining ontology-based querying with key word matching [52]. Since the service ontology may not capture all parts of a particular service, we allow each ontology to have a keyWords property which service providers can populate with keyword terms that describe the specifics of their service. Thus, when a user queries for a service, it initially provides an ontology-based first order predicate logic query statement of the service's properties as well as a set of additional keywords. The ontology-based query is executed and the remaining results are refined by matching the keywords entered.

Service Composition

Furthermore, GloServ allows service composition on a global scale. Service composition aggregates or combines small services into larger services [49]. In many cases, a single service will act as a front-end to many small services. There are a variety of methods for combining services, including: The main limitation of current web services is that they exist as separate entities on the web because data is not easily shared across different domains in the Internet. Every site has its databases modeled in a specific way. Semantically equivalent properties are defined differently. Because of this, querying for a combination of services over a shared property is not easy to accomplish. GloServ supports subquerying between its servers in order to allow services which share common properties to be composed into a single query. A shared property can be defined differently in each domain but mapped to each other as equivalent properties due to the use of ontologies. For example one service domain may define location as a region and another service domain may define it as geographical coordinates. With shared ontologies, these two values can be mapped to one common value using an ontology mapping algorithm. In this case, the geographical coordinates can map to a certain region. This enhancement allows fairly complicated queries to be issued in a single search such as the one given in the earlier example of searching for a restaurant and a nearby theater.

Dynamic Service Updates

Due to GloServ's underlying CAN peer-to-peer architecture, service data is not replicated across every server. The CAN routes messages in approximately $O(\log_2 n)$ hops where n is the number of nodes. This is much more efficient when compared to $O(n)$ node hops. Also, since CAN is a peer-to-peer architecture, it provides load-balancing and fault-tolerance. Thus, frequent service registrations and updates are performed efficiently within the CAN. The CAN architecture is described in greater detail in Chapter 3.

Generic Framework

Since the GloServ back-end provides a generic framework where services can be aggregated into one system, different front-end systems can interact with it. We have built a web front-end which demonstrates the different types of location-based services that can be queried for in GloServ. As future work, we would like to incorporate a user's context into the queries and build a context-aware front-end. Context-aware agents record users' context and store the GloServ queries that they issued in order to recommend future services to the user based on previous choices. Figure 1.6 shows the overall architecture of GloServ with the different front-end systems interacting with it.

1.6 Novel Components

The novel components of this thesis are:

- Contributions of the network architecture
  - using a service ontology classification to map a hierarchical peer-to-peer network architecture;

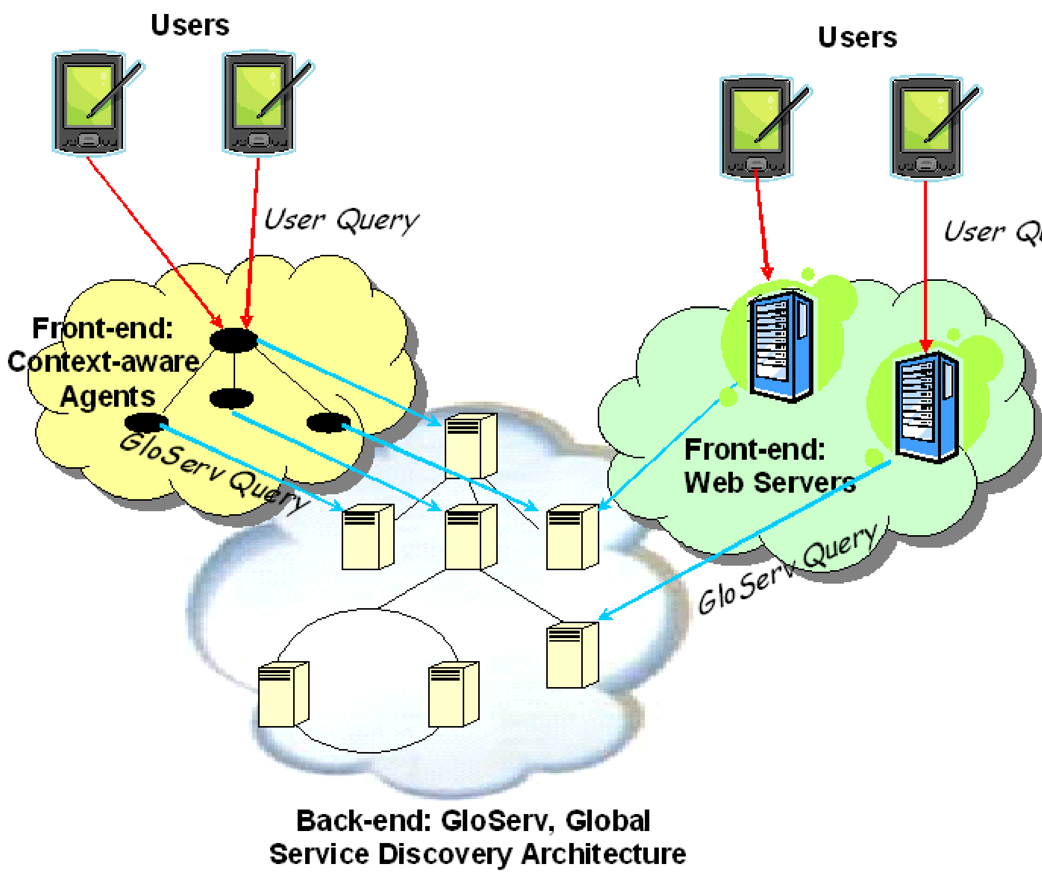

Figure 1.6: GloServ interacting with different front-end systems

- globally distributing services which may also be dynamic in nature, because an underlying peer-to-peer network is used;
- forming a generalized service discovery architecture which integrates different services and allows new service classes to be added easily into the network, given a service classification ontology.

• Querying

- performing semantic ontology-based queries;
- combining ontology-based queries with key word search;
- querying for more than one service in a single search;

1.7 Thesis Outline

The rest of the thesis is organized as follows:

- *Part 1* introduces the thesis and gives background information on peer-to-peer distributed hash tables and ontologies.

- *Part 2* describes the back-end GloServ service discovery system which includes the network architecture, service registration and ontology querying.

- *Part3* discusses query extensions to GloServ such as combining multiple services in a single search as well as allowing ontology and keyword search.

- *Part 4* describes the front-end web-based interface built on top of GloServ's back-end and the design of a context-aware architecture which is ongoing and future work.

- *Part 5* discusses future work and the conclusions to the thesis.

- *Part 6* includes an appendix which has a glossary of commonly used terms and the GloServ library classes.

Chapter 2

Ontologies

2.1 Introduction

An *ontology* is a vocabulary that describes objects and the relations between them in a formal way. It is a specification of a conceptualization [88]. Ontologies describe *classes*, which are sets or collections of certain concepts; *attributes*, which are the properties or features of classes; *relations* that allow classes to relate to one another; *individuals* or *instances*, which are actual instances or objects of a certain class type. The words class, attribute, and individual are frequently interchanged with the words concept, property and instance, respectively. Ontologies are used for knowledge sharing and reuse and allow the ability to reason over the classes and their relationships.

The World Wide Web Consortium [41] has recently approved the Web Ontology Language (OWL) [30] as a standard for representing semantic data in the Semantic Web [62]. The vision of the Semantic Web is to bring structure to web content by creating an environment where software agents roaming from page to page can carry out sophisticated tasks for users. Due to this vision, work has been done in standardizing ontologies which add semantics to traditional web data. This chapter provides background on commonly used ontologies and delves deeply into OWL DL, which is the ontology language that GloServ uses for classification and reasoning.

2.2 RDF and RDF Schema

OWL builds on the Resource Description Framework (RDF) [34] and RDF Schema (RDFS) [33]. RDF provides a metadata model. It has an XML-based [11] syntax and is a language for representing metadata about resources in the World Wide Web and provides a common framework so that applications can process and exchange the information automatically. RDF Schema is RDF's description language. It is also expressed in XML and provides a simple ontology of RDF class and property definitions.

2.2.1 Model

RDF identifies objects using Uniform Resource Identifiers (URIs) [61]. A URI is a compact sequence of characters that identifies an abstract or physical resource. It is used widely as a way to identify resources on the web such as web pages. RDF describes resources in terms of simple properties and property values. Thus, statements in RDF are represented as triples of *(subject, predicate, object)*. The subject denotes the resource; the predicate denotes the relationship between the subject and the object, namely the property; and the object can itself be a resource or a string literal which represents a basic datatype such as integer, string, or boolean values. An example of an RDF triple is:

(http://example.org/index.html, http://example.org/terms/author, "John Doe").

RDF Schema is a semantic extension of RDF which provides a way to describe how resources in RDF are related to each other. The schema divides resources into groups or *classes* and provides a simple hierarchical classification structure which relates these classes to one another through properties. For example, an *author* property relates the *Document* class to the *Person* class.

2.2.2 Examples

RDF and RDFS can have different syntactic forms. The same information can be expressed in more than one way where one expresses greater detail and the other abbreviates. I will show one example of an RDF schema and its corresponding RDF description below.

```
<?xml version="1.0"?>
 <rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">

  <rdfs:Class rdf:ID="Document">
   <rdfs:comment>Document Class</rdfs:comment>
   <rdfs:subClassOf
    rdf:resource=
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Resource"/>
  </rdfs:Class>

  <rdfs:Class rdf:ID="Person">
   <rdfs:comment>Person Class</rdfs:comment>
   <rdfs:subClassOf
    rdf:resource=
    "http://www.w3.org/1999/02/22-rdf-syntax-ns#Resource"/>
  </rdfs:Class>

  <rdf:Property rdf:ID="author">
   <rdfs:comment>Creator of a document</rdfs:comment>
   <rdfs:domain rdf:resource="#Document"/>
   <rdfs:range rdf:resource="#Person"/>
  </rdf:Property>
 </rdf:RDF>
```

Figure 2.1: RDF description

The RDF schema above declares the XML namespace as an xmlns attribute of the *rdf:RDF* start-tag. This specifies that all tags prefixed with *rdf:* are part of the namespace identified by the URI reference, *http://www.w3.org/1999/02/22-rdf-syntax-ns#*, which hold terms from the RDF vocabulary. The second XML namespace declaration, *rdfs:*, specifies that the namespace referenced by the URI, *http://www.w3.org/2000/01/rdf-schema#*, holds terms from the standard RDF schema vocabulary. Here, the schema defines the resources *Document* and *Person* as classes. It also defines the *author* property and assigns its domain to *Document* and range to *Person* which specifies a string or integer value. The domain indicates that the property *author* applies to the *Document* class and the range indicates that the values of the *author* property are instances of the *Person* class.

```
<?xml version="1.0"?>
 <rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:exterms="http://www.example.org/terms/">
  <rdf:Description rdf:about="http://www.example.org/index.html">
   <exterms:author>John Doe</exterms:author>
  </rdf:Description>
 </rdf:RDF>
```

Figure 2.2: RDF Schema

The RDF document above defines the same XML namespace *rdf* as defined in the schema. The second namespace declaration, *exterms* specifies that the namespace referenced by the URI *http://www.example.org/terms/* is to be associated with the *exterms* prefix, which holds terms within an RDF schema developed by the example organization, example.org. The *rdf:Description* start-tag indicates the start of a description of a resource, and goes on to identify the resource the statement is about (the subject of the statement) using the *rdf:about* attribute to specify the *URIref* of the subject resource. The predicate or property element is identified by the *exterms:author* tag, to represent the predicate and object of the statement. Its content, *John Doe*, is the object of the statement, which is a plain literal. These triples can also be represented as a graph where the nodes are resources (subject and object) and the arcs are the properties (predicates).

2.3 OWL

An OWL ontology is also an RDF graph and is represented by a set of RDF triples. As with any RDF graph, an OWL ontology graph also has different syntactic forms. OWL extends

RDF and RDF schema ontologies by adding more vocabulary for describing properties and classes such as relations between classes, cardinality, equality, richer typing of properties, characteristics of properties, and enumerated classes. Thus, the ontology is not limited to defining pure hierarchical class relationships and hence further inferences can be made about class relationships such as equivalence or disjointness. Below we give an overview of the sublanguages of OWL and the characteristics of OWL classes and properties.

2.3.1 Model

Every OWL ontology has a root class called *owl:Thing*. Every other class defined with the ontology is a subclass of *owl:Thing*. OWL supports set operators on classes such as union, intersection and complement. It also allows class enumeration and disjointness. There are two types of simple properties, namely *datatype* and *object* properties. Datatype properties are relations between instances of classes and RDF literals or XML schema datatypes. Object properties are relations between instances of two classes. Properties can also have logical connectivity such as being transitive, symmetric, inverse and functional. As in RDFS, an OWL class contains individuals, which are instances of the class, and other subclasses. Instances are RDF descriptions of a class where properties are populated with certain values.

OWL allows a class to be defined with logical restrictions on certain properties. Classes can be restricted by existential or universal quantifiers. For example, the class *Restaurant* may have subclasses defined with existential quantifiers on the hasCuisine property such as:

hasCuisine **some** *Chinese*.

Thus, a restricted subclass of the *Restaurant* class can be defined as the *ChineseRestaurant* class which contain all restaurants that have *Chinese* as a cuisine. This amounts to all the instances that have the hasCuisine property assigned to the *Chinese* cuisine. In addition, these properties can also have cardinality restrictions. For example, a restaurant must have at least one type of cuisine but may have more than one. Thus, the hasCuisine property can be restricted to:

hasCuisine ≥ 1

and a restaurant instance can have multiple hasCuisine properties. For example, a restaurant which serves both Chinese food and Korean food will have two hasCuisine properties each assigned to *Chinese* and *Korean* respectively.

2.3.2 Sublanguages

There are three sublanguages in OWL: OWL Lite, OWL DL and OWL Full, all described in [30]. OWL Lite is the least expressive of the three sublanguages. OWL Lite is a bit more expressive than RDFS because in addition to supporting a classification hierarchy, it also provides simple constraints of classes and properties. OWL DL is modeled after description logics and supports maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs. OWL Full is the most expressive of the three sublanguages. The main difference between OWL DL and OWL Full is that in OWL DL, a class is only expressed as a collection of individuals and cannot be regarded as an object in and of itself. However, in OWL Full, a class can be treated simultaneously as a collection of individuals and as an individual in its own right. Due to this difference, OWL Full cannot be checked for soundness using OWL description logic reasoners to check for soundness. We have chosen to use OWL DL for two reasons: a service class will only represent a collection of individuals and does not need to be an individual in its own right and we would like to use OWL DL reasoners such as RacerPro [90] to check for the soundness of OWL documents. Thus, due to the heavy use of reasoners within GloServ, our system operates within OWL DL, whereas classifying an ontology in a finite amount of time is not guaranteed to be successful in OWL Full.

2.3.3 OWL Querying

There are a number of ways to query an OWL ontology. One way is to use a reasoner to create a class with certain restrictions and classify this class within the ontology to see which classes it relates to. The query processor can then look at the equivalent classes, super classes and subclasses to see the different relationships of the query class.

Another way is to use the reasoner to reason over the OWL instances. The query processor converts the query to an instance of an OWL class where all the property values are the same as that of the query. The reasoner then finds all the classes which have this instance as an inferred instance. An inferred instance is one that has not been explicitly instantiated within a class but is inferred to be part of a class because of its properties. The instance is then matched to a number of classes.

Aside from these methods, there are a few OWL query languages which have been developed. The OWL Query Language (OWL-QL) [80] is a formal language and protocol that queries an OWL ontology by finding class relationships. It also allows querying and answering agents to conduct a query-answering dialogue in ontologies represented by OWL. SPARQ-DL [118] is another OWL DL query language. It extends SPARQL [38], which is a query language for RDF, to also query OWL description logics by searching for complex class relationships.

GloServ uses the first method of creating a class and finding class relationships in order to query the ontology. Reasoning only over the classes is much faster than reasoning over instances and also allows complex queries to be formulated. Since GloServ implements its own query-answering dialogue, it does not use OWL-QL.

2.3.4 OWL Examples

OWL Syntax

```
<?xml version="1.0"?>
 <rdf:RDF xmlns="http://www.owl-ontologies.com/unnamed.owl#"
        xml:base="http://www.owl-ontologies.com/unnamed.owl"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:owl="http://www.w3.org/2002/07/owl#">
```

Figure 2.3: OWL XML heading

As in RDF, an OWL ontology begins with an XML heading and defines a number of namespaces.

OWL Class

```
<owl:Class rdf:ID="Restaurant">
  <description_us rdf:datatype="&xsd;string">The restaurant service
    class represents restaurants, cafes, bars etc. The major properties
    of each restaurant is its  location (i.e. neighborhood) and the cuisin
    it offers.
  </description_us>
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasCuisine"/>
      <owl:minCardinality rdf:datatype="&xsd;int">1</owl:minCardinality
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Figure 2.4: OWL class description

An OWL class description includes an *owl:Class* label and an *rdf:ID* or *rdf:about* property which provides the OWL class with a name. This example above defines a *Restaurant* class. The *Restaurant* class has an *annotation property* description_us. An annotation property is used to annotate a class or property for documentation purposes. It is not used by the reasoner but only for human readability. Thus, here, the property description_us describes the *Restaurant* class in English.

The *Restaurant* class is set equivalent to an anonymous class which has a cardinality restriction on the hasCuisine property. The restriction sets the hasCuisine property to have a minimum cardinality of 1. This means that all instances within the *Restaurant* class must have at least one cuisine property defined.

OWL Subclasses

```
<owl:Class rdf:ID="ChineseRestaurant">
  <rdfs:subClassOf rdf:resource="#Restaurant"/>
    <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasCuisine"/>
      <owl:someValuesFrom rdf:resource="#Chinese"/>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="KoreanRestaurant">
  <rdfs:subClassOf rdf:resource="#Restaurant"/>
  <owl:equivalentClass>
  <owl:Restriction>
    <owl:onProperty rdf:resource="#hasCuisine"/>
    <owl:someValuesFrom rdf:resource="#Korean"/>
  </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Figure 2.5: OWL restricted subclass description

In this section, two subclasses of the *Restaurant* class are defined, *ChineseRestaurant* and *KoreanRestaurant*. These classes are equivalent to anonymous classes which have restrictions on the hasCuisine property. The *ChineseRestaurant* class has an existential restriction on the hasCuisine property of at least one value from the *Chinese* cuisine class. The *KoreanRestaurant* class has an existential restriction on the hasCuisine property of at least one value from the *Korean* cuisine class.

```
<Chinese rdf:ID="ChineseCuisine"/>
<Korean rdf:ID="KoreanCuisine"/>

<Restaurant rdf:ID="ChineseKorean">
  <hasCuisine rdf:resource="#ChineseCuisine"/>
  <hasCuisine rdf:resource="#KoreanCuisine"/>
</Restaurant>
```

Figure 2.6: OWL instance description

OWL Instances

The above description shows an instance of the *Restaurant* class, *ChineseKorean*. It is described by the label *Restaurant* and its *rdf:ID* labels the instance name to *ChineseKorean*. This instance has two hasCuisine values: *ChineseCuisine* and *KoreanCuisine*. These cuisine values are both instances of the *Chinese* and *Korean* cuisine classes. The ontology reasoner can compute the inferred instances of an ontology and in this case, it will infer that this instance is also part of the *ChineseRestaurant* and them *KoreanRestaurant* classes.

OWL Querying

```
<owl:Class rdf:ID="queryClass">
  <rdfs:subClassOf rdf:resource="#Restaurant"/>
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasCuisine"/>
        <owl:someValuesFrom>
          <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
              <owl:Class rdf:about="#Chinese"/>
              <owl:Class rdf:about="#Korean"/>
            </owl:unionOf>
          </owl:Class>
        </owl:someValuesFrom>
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

Figure 2.7: OWL query class

One way to query an OWL ontology is by creating a temporary class with a number of restrictions and classifying this class. Here, a *queryClass* is defined to be a subclass of the *Restaurant* class. The restrictions on this class are on the hasCuisine property. This class is equivalent to the restricted class which has cuisines *Chinese* or *Korean*. When this ontology is classified, the *ChineseRestaurant* and the *KoreanRestaurant* classes will be classified as subclasses of the *queryClass*.

2.4 Constructing Ontologies

There are many ways to construct ontologies. Horridge, et al. [94] and Rector [112] describe a few. Rector describes an ontological engineering method that provides a modularized approach to classification [112]. Modularization or normalization occurs during the ontology construction. High-level classes that are disjoint from each other are created and put into a pure hierarchical tree structure. This creates a *primitive skeleton*. Disjointness means that no class is logically related to a sibling class within the disjoint hierarchy, hence individuals will only be classified within one of the branches. The main goal of normalization is to allow portions of ontologies to be re-used and separated from the whole and to evolve independently of each other. These characteristics are necessary in any ontology-based system. We have chosen this method for service classification and describe the details of how services are classified below.

As described in [112], the best approach to modularize an ontology, by identifying portions of an ontology that describe a certain domain, is to first create a primitive skeleton which is a hierarchical tree of primitive or high-level concepts. Since these concepts represent high-level classes, the primitive skeleton resides in the upper level of the ontology. The upper level is constructed in such a way that each concept has only one parent and disjoint siblings. Once this primitive skeleton has been formed, descriptions and definitions are created to express the relations between those primitives. It is then easy to identify each branch of the ontology as one “module”.

Primitive skeletons distinguish two types of concepts, *self-standing concepts* and *partitioning concepts*. Self-standing concepts include “things” that are part of the physical world such as “animals” or “organizations”. Partitioning concepts, on the other hand, are values that partition self-standing concepts such as “small, medium, large”. These two types of concepts are called *Self-Standing* and *Refiner* entities. These two entities can be used in order to construct a modularized ontology that can be shared and re-used as well as reasoned over automatically.

For example, Figure 2.8 shows a classification ontology which is not modularized and its corresponding modularized primitive skeleton. In the original classification, the ontology explicitly defines the class *RedApple* as a subclass of *Apple* and *Red*. This is a poor construction because the *RedApple* concept is an apple that *has* a color red and thus should not be classified under the color red but be *described* by it. However, in the corresponding primitive skeleton, the main *Entity* class is broken down into two subclasses which represent: the *Self-Standing Entity* and *Refiner* classes. The *RedApple* class is defined as a restricted subclass of the *Apple* class where its hasColor property is restricted to *Red*. Now, when a user searches for red apples, she can query for the following:

Apple and hasColor **someValuesFrom** *Red*

The same idea applies for the class *BigApple* which is defined as a subclass of *Apple* and has a hasSize property restricted to *Big* from the *Refiner Skeleton*.

Thus, normalizing an ontology provides better modularization and allows a reasoner to do the classification automatically. The separation of the self-standing and refiner ontologies allows other systems to re-use parts of these ontologies. Such classifications will not change frequently over time and thus can be distributed and cached periodically. This model fits well with the GloServ network described in this thesis.

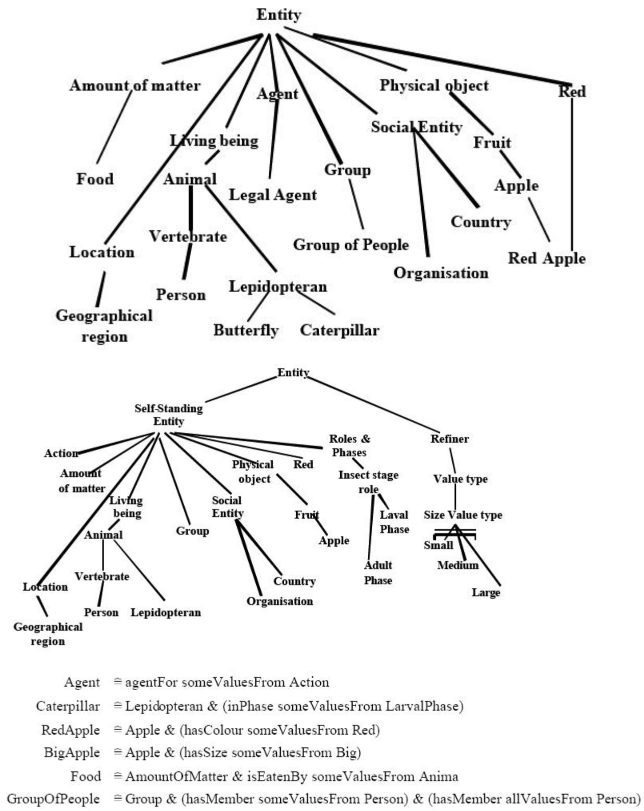

Figure 2.8: Original classification ontology converted to corresponding primitive skeleton [112]

2.5 Ontology Sharing

Certain classes can be shared and re-used by other ontologies. For example a *GeographicLocation* class can classify cities of the world by continent, country, county or state and city. For example *New York* would be classified under $NorthAmerica-\rangle UnitedStates-\rangle NewYork$. Because the location ontology is a generic definition, this can reside in a centralized OWL repository and downloaded by systems which need a location ontology. Another example of a generic classification that can be shared is the *Cuisine* class. Here, classes are distributed by ethnic regions such as $Asian-\rangle Chinese$. GloServ relies on this type of ontology sharing within its system. This is described in the subsequent chapters.

2.6 Difference between Ontologies and Databases

A relational database [70] is defined by a logical model of entities and relationships (ER model) in a domain. The ER model is a simple ontology because it does not define relationship types and class expressions and thus cannot be classified using an ontology reasoner. Also, an ER model is used for translating to physical tables and thus knowledge about the relationships is captured mainly in the documentation versus within the ontology itself. This schema is normally specific to the needs of the given application and thus cannot be shared easily across different domains.

The motivation behind using ontologies for representing services rather than using simple attribute-value representations of data, such as in traditional databases, is mainly due to the reasoning power behind ontologies. An example of a query which can be done using an ontology which is difficult to do using an SQL query is: "Given a service class, find all logically related matches to my query". Service search is evolving to cater to individual users and their context and thus reasoning capabilities such as these are essential for performing context-aware searches. SQL also does not support abstract data types, thus making it difficult to determine if a certain property value belongs in a number of different classes or types. Ontologies can also be shared, re-used and changed flexibly. For example, when new relationships are established within the ontology because of ontology migration or the addition of new classes, determining new relationships within the ontology simply reduces to running a reasoner on the ontology in order to reclassify the classes.

The main drawback to using an ontology is that classification is expensive. As ontologies grow large and especially when instances of classes are stored in the ontology, reasoning becomes a bottleneck. We tackle this problem by storing instances in a database back-end instead of the ontology itself and using only class relationships for determining which classes a query belongs in. This speeds up the classification process considerably. Also, a positive side-effect of the distributed architecture of GloServ allows each server to handle an ontology for one service class. Thus, the size of the ontology always remains manageable for classification.

Chapter 3

Peer-to-Peer Systems

3.1 Introduction

In peer-to-peer systems nodes are symmetric and can act both as a client and a server, unlike in the traditional client-server model. A subset of peer-to-peer systems are file sharing applications, which allow Internet users to share files, such as music or videos, directly with other Internet users. In 1999, Napster [27] introduced a peer-to-peer communication model where users were able to download and share music files with others. However, although Napster uses a peer-to-peer model, it still relies on centralized servers to maintain an index of files being shared. Since then, even more popular commercialized peer-to-peer systems have been developed such as Skype [36], which enables Voice over IP phone calls over a peer-to-peer network, and Bittorrent [5], which is a peer-to-peer file sharing protocol that allows large amounts of data to be transferred between peers by reducing the load on an individual node.

There have also been a number of peer-to-peer architectures designed within the academic community. A few well-known systems are the Content Addressable Network (CAN) [110], Chord [105], Kademlia [100] and Pastry [114]. These systems provide load balancing, are fault-tolerant and scalable. Peer nodes distribute files by hashing either a file name to a key which maps to a specific node in the network. Each node maintains a hash table that contains a mapping of keys to node IP addresses. The whole system is thus a Distributed Hash Table (DHT). Each of these architectures uses a routing algorithm that forwards a query to a peer node that is closer to the node containing the file that is being searched. This chapter discusses these distributed hash tables in greater detail and motivates the use of the CAN DHT for distributing service information in GloServ.

3.2 Overview of Peer-to-Peer Architectures

Structured peer-to-peer (P2P) architectures share a few main components [56]. At the heart of these systems is the Distributed Hash Table (DHT) whose main function is to map keys to nodes in the network. A *key* is the numeric representation of the file that is being searched for. It is usually created by hashing the name of a file by using a hashing algorithm, such as SHA-1 [74].

Generally, keys and nodes are represented as m-bit identifiers (*IDs*), and the keys reside in nodes whose IDs are ”closest” to it. The closeness function is defined differently in each P2P DHT. For example, in Chord and CAN it is the numeric difference between the two IDs, in Pastry it is the number of bit positions with equal value and in Kademlia it is the bit-wise XOR of the two IDs. The DHT maintains a list of neighboring peer nodes that is used for routing an incoming query to the closest node. These tables adapt to nodes joining and leaving the network.

The main differences between these architectures are the data structures used for implementing the DHT. CAN implements its DHT as a d-dimensional Cartesian-coordinate space; Chord implements a finger table for look up, which is a routing table with a number of entries containing information about successor nodes; and Pastry and Kademlia have tree data structures.

3.3 Content Addressable Network

CAN is modeled around a d-dimensional Cartesian-coordinate space on a d-torus. The coordinate space is partitioned into hyper-rectangles, or zones. Each node in the system is responsible for a zone, and a node ID is represented by its zone boundaries. A key is represented by a point in the coordinate space and is forwarded to and stored in a node whose zone contains this point in its space. Thus, each key will be represented as a $\langle dimension, value\rangle$ pair. Taking a 2-dimensional CAN as an example, a node may reside in a zone whose keys are in values 0-5 in dimension 0 and 0-5 in dimension 1. Thus, a key with ID $\langle 1, 4\rangle$ will be forwarded and stored to this node. Figure 3.1 shows how this message is routed in the CAN.

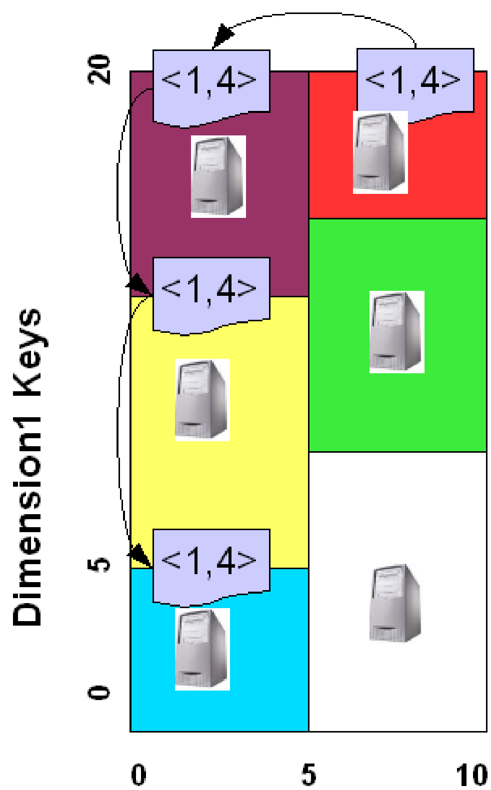

Figure 3.1: Example of look-up in a 2-dimensional CAN

Each peer in the CAN maintains a routing table that holds the IP address and zone of its neighbors. When a query arrives at a CAN node in the form of a $\langle dimension, value\rangle$ key, the node checks to see if it holds information for that dimension-value pair. If it does not, it routes the query to the neighboring node which is closest to the destination coordinates.

The worst case number of message hops in an N-node CAN is from one end of the torus space to the other, which evaluates to $O(N^{1/d})$ hops. If the number of dimensions is fixed to be $(\log_2 N)/2$ then the run-time reduces to $O(\log_2 N)$.

3.4 Chord

The Chord [105] topology assembles nodes in a one-dimensional ring. In an N-node network, each node maintains information about $\log_2 N$ other nodes in a finger table. Thus, each node maintains a DHT that is connected to nodes half way across the network, a quarter way across the network, an eighth of the way, etc. It uses a consistent hash function to assign keys to nodes. Key k is assigned to the first node whose identifier is greater than or equal to k, namely, the first node clockwise from k within the ring. Thus, closeness is defined as the difference between the node ID and the key. A request is routed to the nearest node to k. If the ring has 2^m nodes with identifiers and keys ranging from 0 to $2^m - 1$, then lookup requires O($\log_2 N$) messages. Figure 3.2 displays a Chord ring and demonstrates how a query is routed within the network.

3.5 Pastry

Pastry [114] is a peer-to-peer DHT that is similar to Chord in that key space is circular. It determines closeness between a key and a node ID by comparing the prefix of a number of bits of a node identifier and the key. When a message key enters node n, it is matched to a node in the routing table whose ID shares a prefix with the key that is at least a digit (or b bits) longer than the prefix that the key shares with the present node n's ID, If such a node exists, the request is sent to that node. If such a node does not exist, then node n forwards the message to another node that shares a prefix with the key which is at least

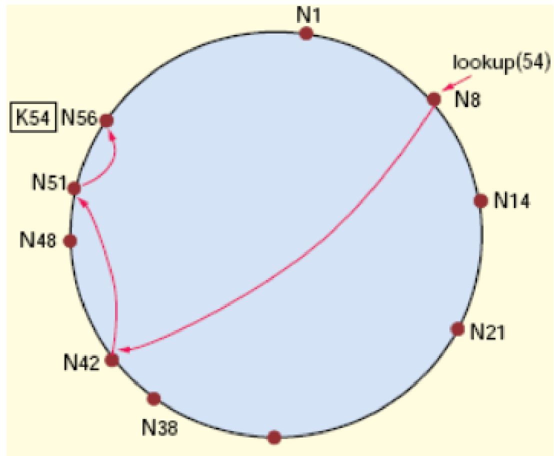

Figure 3.2: Example of look-up for key value 54 in Chord [56]

the same prefix length as node n's, but is numerically closer to the key than the present n's ID. The routing table is separated by address blocks which are formed by dividing the key into b-bit digits. Due to this, a look up requires $O(\log_2^b N)$ messages for an N-node network. Figure 3.3 illustrates the architecture and lookup of a request. The keys are represented as hexadecimal values.

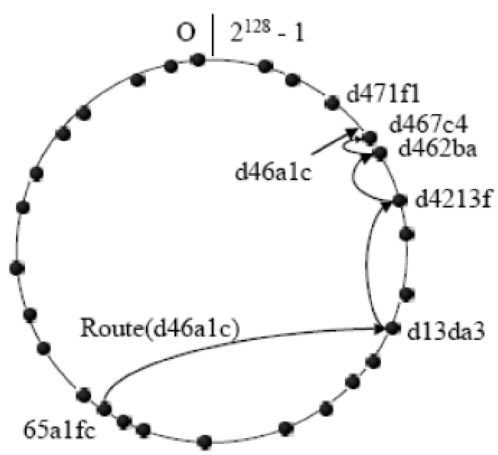

Figure 3.3: Example of look-up key value d46a1c in Pastry [114]

3.6 Kademlia

Kademlia [100] is a peer-to-peer DHT similar to Pastry and Chord except that it uses an exclusive-or (XOR) metric to compute the distance between two nodes. Each node has an m bit identifier. Each bit points to a routing table so, for an m-bit identifier, there are m routing tables, one for each bit. Every list corresponds to a specific distance from the node where the nth list holds information about nodes that match the first $n-1$ bits of the node and differs in the nth bit. A key is matched to its node identifier, the nth routing table is found and a request is routed to the nodes within that table. Since Kademlia uses an XOR metric to define distance and matches two IDs by an m-bit prefix, this reduces look-up messages to $O(\log_2 N)$. Figure 3.4 shows how node n, with prefix 0011 and represented by the black dot, has neighboring nodes represented by gray dots. Node n finds the node prefixed with 1110 by querying closer and closer nodes.

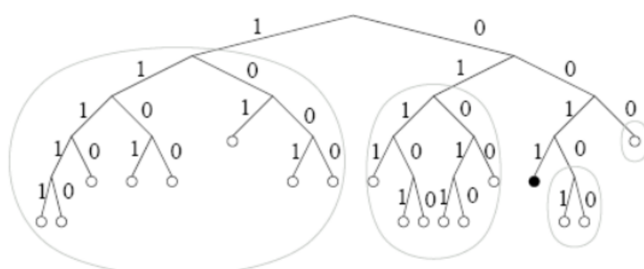

Figure 3.4: Example of look-up of key 1110 in Kademlia [100]

3.7 Use of CAN in GloServ vs. Other DHTs

GloServ uses the CAN DHT as its underlying peer-to-peer architecture. The main motivation for choosing the CAN over the other DHTs is its multi-dimensional structure. GloServ maps an ontology onto a CAN by assigning each class to a $\langle dimension, value \rangle$ key. DHTs usually assign keys by hashing the string that is being searched for with well-known hashing algorithms such as SHA-1. However, in the case of the GloServ CAN DHTs, we leverage information within the ontology and therefore distribute the instances in a unique way, which is by the classification information in the ontology.

Each class, which is restricted on a set of properties, is classified and hashed to a numeric $\langle dimension, value\rangle$ key. Since ontology classification causes similar classes to cluster together, similar classes will be assigned $\langle dimension, value\rangle$ keys that are numerically close and thus will reside in either the same CAN zone or nearby zones. Thus, a server which handles values on a certain number of keys will hold similar content within its node. This is helpful because CAN permits proximity routing and thus when one searches for a service within a certain class, similarity searches will minimize query latency because similar classes will reside in either the same node or nearby nodes.

However, by default, Chord, Kademlia, and Pastry do not take advantage of the content to distribute data due to their 1-dimensional structure. Chord uses the SHA-1 hash of the content which gives some randomly distributed number and removes the closeness property of the values. One alternative is to enforce closeness in Chord, which can be done in either of two ways: 1) The attributes can be ordered such that all the values with a given attribute are grouped together. Then within this group, all the values with another attribute are grouped together and so on. This is space efficient, but if the search is done for the second attribute, then all the groups need to be searched which is inefficient. 2) Group each attribute so if, for example, there are 100 instances and 10 attributes, assuming that most instances have all the attributes populated, then the first attribute will contain most of the 100 instances, and similarly for all the other attribute-groups. Thus, there will be 1000 instances which is not space efficient. Secondly, if we use a non-uniform hash in Chord, such as the attribute-value hash we describe in Section 4.3.3, then the logarithmic bounds of Chord are no longer valid, since it assumes a uniform distribution of keys.

Other single-dimension DHTs, such as Kademlia and Pastry, also suffer from the same problems. For example, in Pastry, we can map the attributes onto the identifier digits by designating a certain number of digits to one attribute. But the problem here is that two instances that may have logically similar attributes may not be neighbors in the identifier space, if the attribute was mapped to the most significant digits. Kademlia uses the same mechanism as Pastry for the first half of its look up algorithm, thus it too is not an appropriate DHT to use for GloServ.

We use CAN overlay networks where every class of siblings within an ontology lie on the same CAN. If a class does not have any subclasses, it creates a CAN based on its property values. Classes that are similar have keys in a nearby range and thus similar content is clustered together in the network. It would be advantageous to use a one-dimensional DHT if all the sibling classes are disjoint from each other, since there is no benefit in clustering information at that point. However, since this case will be rare in the lower rungs of the ontology classification and we do not want to complicate the architecture by implementing different DHTs, we implement the whole system in CAN. The next chapter describes, in detail, how the CAN is used within GloServ.

Part II

GloServ Back-End Service Discovery Engine

Chapter 4

Network Architecture

4.1 Introduction

The GloServ back-end service discovery architecture consists of servers, named *GloServers*, connected as a hybrid network of hierarchical and peer-to-peer nodes. Our motivation for using a hybrid network is due to the way the service classification ontology has been engineered. Initially, we define high-level service classes, such as *Restaurant, Travel* and *Communication* and arrange them in a pure hierarchical ontology where sibling classes are logically disjoint from each other, ie, completely unrelated. This represents a primitive skeleton ontology. As service classes become more specific, they begin to have greater interconnections between other classes and a pure hierarchy is no longer maintained. Figure 4.1, in the Introduction, illustrates this architecture.

Due to this ontology construction, I have elected to design a hybrid network that mirrors the ontology classification. High-level service classes are mapped onto servers that are arranged in a hierarchical network. These servers are used for maintaining a snapshot of the high-level service ontology, which contains each high-level server's hostname and other types of information. They also route GloServ messages to the lower levels of the network. Lower-level service classes are mapped onto a peer-to-peer network of servers where similar

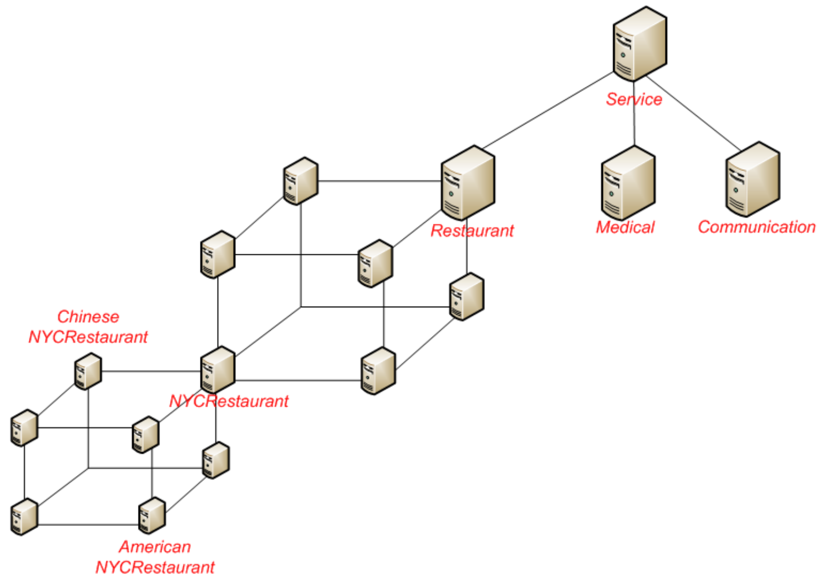

Figure 4.1: GloServ hierarchical peer-to-peer architecture

service classes reside. These servers hold the service data within GloServ.

The hierarchical network of servers will not hold service data because the data will instead be registered within the low-level peer-to-peer servers. Each high-level node contains a snapshot of the entire high-level ontology and thus a query can contact any node in the high-level network before entering into the peer-to-peer layers. There is little need to worry about load distribution and query scaling in the high level network since the purpose of these servers is to route messages to the lower-level. Thus, a simple hierarchical network structure suffices. By using the primitive ontology model, a server is able to get to its children as well as to those servers that are disjoint from it very quickly. Furthermore, front-end systems regularly interacting with the GloServ back-end will not need to go through the hierarchical network each time they issue a query because they will have cached the host information of the peer-to-peer nodes.

Besides organizing high-level servers, we need to establish a network for servers that contain closely-related information. These servers are connected to each other in peer-to-peer overlay networks. The motivation for distributing data using a peer-to-peer network, instead of using replicated servers such as in distributed databases, is because we anticipate both a large number of updates and queries in the system. For the application scenarios described in the Introduction, replicating data across the servers would not scale for large number of frequent updates. Thus, in order to achieve load distribution, fast query and update processing time, while maintaining reliability, I have elected to use a peer-to-peer network. Below, I discuss the GloServ architecture in detail and use the *Restaurant* service class defined in the ontology in Figure 4.6 for the examples given.

4.2 Hierarchical Network

4.2.1 Elements within a GloServer in the hierarchical network

Gloservers in the hierarchical part of the GloServ network contain two types of information: a *primitive skeleton* representing a high-level service classification ontology and a thesaurus ontology. An example of a high-level service classification ontology can be seen in Figure 4.2. As mentioned above, this classification is not likely to change frequently and thus can be distributed and cached across the GloServ hierarchical network. Each high-level service class has a set of properties. Subclasses of the high-level service classes inherit these properties. As the subclasses are constructed, the properties become specific to the particular service type.

The second piece of information within a GloServer is a thesaurus ontology. The thesaurus ontology maps synonymous words to each of the service terms in the service classification ontology. This results in greater accuracy in finding the correct server and information for registration and querying. Synonyms of every class within the service classification hierarchy are stored. Figure 4.3 gives an example of the partial graph of the synonyms of *Restaurant* within the thesaurus ontology. The thesaurus, too, will not change often and thus can be distributed and remain in each of the servers.

Since GloServ is a global system, it will be used in many different languages. The use of thesaurus ontologies enables multilingual synonyms to be defined as well. However, in

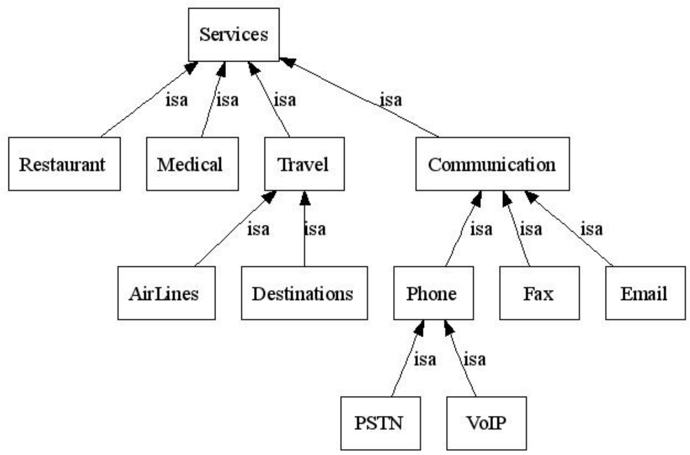

Figure 4.2: A high-level service classification

this case every single ontology defined in the whole GloServ network requires a thesaurus to translate foreign language queries to the native language that the ontology is written in. This is not difficult to accomplish considering that ontologies are easily shared. One can imagine a thesaurus repository providing multilingual thesaurus ontologies for service classes and properties. Thus, as an example, for GloServers that are located in France, service ontologies may be defined in French but an English speaker can easily access this information because of the multilingual thesaurus ontology. Another way of implementing this besides using thesaurus ontologies is to use labels or language tags within the service ontology itself. However, in order to avoid operating with large ontologies, I have elected to create a separate thesaurus ontology for each service class.

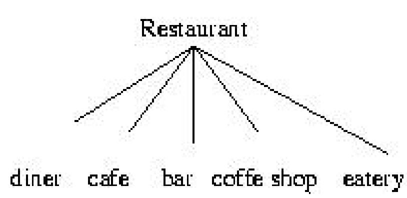

Figure 4.3: Partial view of a thesaurus ontology containing synonyms of “Restaurant”

4.2.2 Generating the Hierarchical Network

The hierarchical network is mapped from the high-level service classification ontology. The GloServ architecture is similar to the Domain Name System (DNS) [104] in that the high-level servers are like *root name servers* and *authoritative name servers* that manage the information within their domain. We envision that a separate classification system similar to North American Industry Classification System (NAICS) [1] classifies the hierarchy of services and establishes OWL ontologies that describe each type of service. Experts within a domain construct ontologies for each domain and an authority such as ICANN [2] delegates the top level services and events to individual root and authoritative name servers. The service categorization is similar to a yellow pages directory. Although the categories may change from time to time, they are not expected to change drastically or frequently, which gives us the ability to pre-define the service hierarchy and implement caching within the back-end GloServers and front-end servers.

The hierarchical network is generated automatically given a high-level service classification ontology (primitive skeleton) and a set of servers. First a root server is turned on which represents the root service class *Service*. The server is identified by a Uniform Resource Name (URN) [124]. A URN is a Uniform Resource Identifier (URI) [61] which is used for identifying resources on the Internet. In GloServ, the root server has the URN urn:gloserv:Service. It enters its hostname in the ontology and then for each subclass of the *Service* class, it contacts a pool of servers and assigns each server to one of the child classes in the ontology. It also assigns the URNs of each of these servers according to their location in the hierarchy, enters it into the ontology. Once the network is completed, the root node distributes a snapshot of the whole ontology to each high-level server.

4.2.3 Server Bootstrapping

Bootstrapping is the process of locating a server within the GloServ network, given a URN. Bootstrapping into the GloServ system is similar to a DNS lookup. DNS is a hierarchical system that translates hostnames to IP addresses and has *authoritative name servers* which publish information about their own domain as well as child domains. For example, the server which handles *cs.columbia.edu* is an authoritative name server which knows information about all name servers in child domains such as *clic.cs.columbia.edu* and *cluster.cs.columbia.edu* name server. In this way, if a message originates from a different domain, it can go up the branches of the DNS hierarchy and then traverse the path of the hostname in order to find the IP address of the node that needs to be contacted.

Similarly, in GloServ, each hierarchical server represents a service class and its hostname is determined by looking at the primitive skeleton ontology. A server's URN will follow the hierarchical format. For instance, given the example of the service classification ontology in Figure 4.2, the *Restaurant* class's URN [103] will be urn:gloserv:Service.Restaurant whereas the *Destination* class's URN will be urn:gloserv:Service.Travel.Destination. As a server is assigned to a hierarchical network, it updates the ontology to include its server information.

Figure 4.4 outlines the steps high-level GloServers take in order to find the main high-level server for a given query. Initially, a user contacts a front-end system which connects to the GloServ network. The GloServ front-end is described in greater detail in Chapter 12. For now, we will allude to it briefly. The user then enters the word *cafe*. If the front-end server has access to the thesaurus ontology it maps the word *cafe* to the *Restaurant* class and contacts the high-level server for *Restaurant* immediately. However, if it does not have a thesaurus ontology, but has the high-level service classification ontology, it sends the query to any of the high-lever servers. If it does not have this either, then it sends it to the root node in the GloServ network which has a well-known URN, urn:gloserv:Service. In either case, one of the high-level servers receives the query, maps it to the *Restaurant* service class, locates the *Restaurant* class in the primitive service classification ontology and determines the host name which is stored in the ontology itself.

High-level servers can communicate in two ways. The first way is to store a snapshot of the whole primitive classification in every server. This classification not only provides information about the relationships of each of the service classes, but also holds the domain name information of the main high-level server to contact. Thus, the high-level servers can connect to each other in one hop. This method is plausible only because we expect the number of service types to be in the order of 100s. This expectation comes from realizing that at a high-level, the number of service classes are limited. Also, the average number of words known by a human is around 20,000 words which causes us to conclude that the number of words within a high-level classification is much less than 20,000. The other possibility is that servers store information about the root, disjoint sibling classes, parent class and child classes. Using this method, there is always a way of getting to another node within the classification ontology.

Both these methods have benefits and drawbacks. The main benefit of the first method is that the look-up time is $O(1)$, since there will be a limited number of high-level classes within the primitive skeleton. The drawback, however, is that every time a server's domain name is changed, the other nodes need to be notified. Although this may pose a problem, it can be solved by allowing each server to periodically cache a new snapshot from a central ontology repository, rather than have a node notify all other nodes of its updates. The second method solves the problem of updating domain names during changes in the network. However, since the domain names of servers are not expected to change frequently, caching is a viable solution in order to save in lookup time.

Continuing with the example in Figure 4.4, a high-level server routes the query to the servers handling the *Restaurant* class. Since the *Restaurant* class is distributed in a peer-to-peer network, any *supernode* can be contacted. A supernode is a node within the peer-to-peer network that handles incoming queries from users. The *Restaurant* CAN network will have service data registered within its nodes. These are actual instances of the *Restaurant* class. Many instances of restaurants are stored here and thus the information will have to be distributed across other restaurant servers that are connected to each other in a peer-to-peer fashion. This is where CAN is used.

The main high-level *Restaurant* server that is initially contacted responds to the query by returning its restaurant ontology. If the front-end is a web-based system, it constructs a form from the ontology for the user to fill out. If the front-end is not user-centric, but automated, it automatically constructs a query using information from the ontology.

Queries are constructed by using information from the class properties. Some restaurant properties are hasCuisine, hasNeighborhood, hasRating and hasPriceRange. The user fills out her preferences for each property. Since there are many restaurant servers that store similar information, there are two possible ways of issuing the query. One way is for the query to be sent to all of the peer nodes. This is inefficient considering some nodes may not contain any of this information and thus sending it to those servers is futile. A better way is to convert the query data to a key and search for the server within a CAN network. We adopt the latter approach and discuss it below.

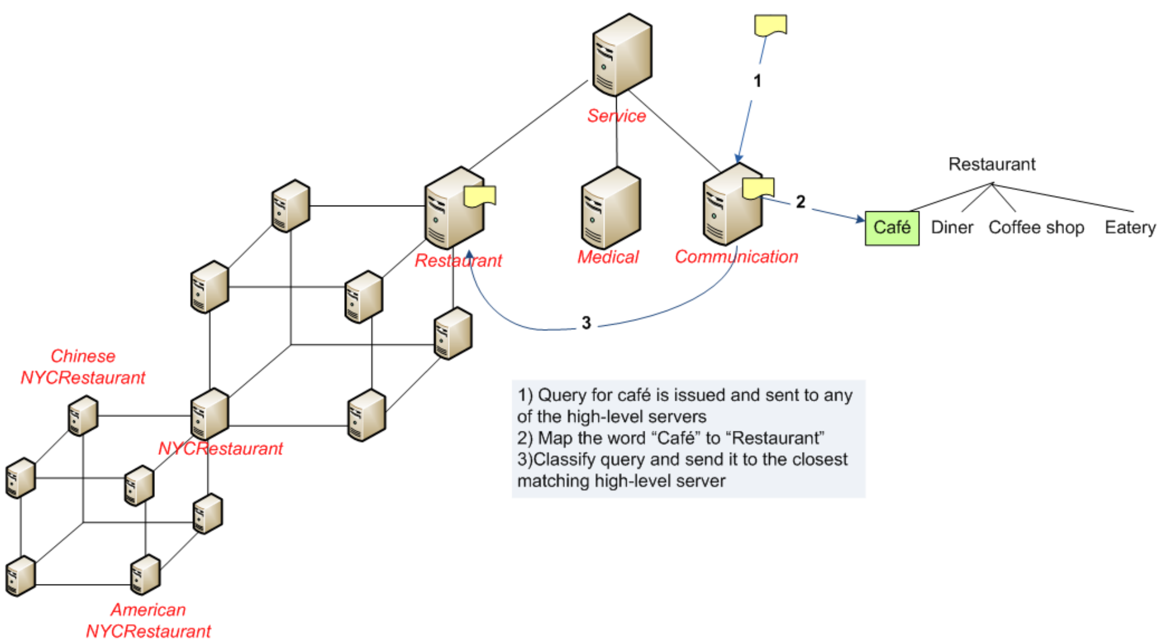

Figure 4.4: Finding servers in GloServ

4.3 Peer-to-Peer Network

4.3.1 Elements within a P2P Gloserver

Every P2P GloServer contains three main elements: a service ontology, a CAN lookup table and service data. First, each P2P network represents a certain service class. Thus, P2P GloServers share the same service ontology as their peers. For example, if a P2P network exists for the *Restaurant* class, then each node in the network will contain an ontology for the *Restaurant* class. Second, each node has a CAN lookup table which is constructed according to the service ontology. The CAN table connects servers of the same type to each other in a peer-to-peer network. We use a novel mapping algorithm that combines the benefits of OWL and CAN to map content of service instances to nodes in a peer-to-peer network. We describe this further in Section 4.3.3. Third, P2P nodes may also contain service registrations. When services register within GloServ, they are routed to the P2P network which handles that service class and the instances are stored in the P2P nodes.

4.3.2 OWL Ontology Construction

Each service is defined with an OWL-DL ontology. The ontology has a main service class which is defined by other classes within the ontology. I will use the *Restaurant* service class as the running example. We used the restaurant classification specified on the http://www.menupages.com website [24]. Restaurants are classified by neighborhood, cuisine, price range and rating. Thus, I defined a *Restaurant* class which has properties such as hasNeighborhood, hasCuisine and hasPrice. These properties are object properties which have values ranging to other OWL classes. For example, the hasCuisine property is an object property which has the *Cuisine* class set as its range. Thus, values of the hasCuisine property are all the subclasses of the *Cuisine* class. Similarly there must be *Rating* and *PriceRange* classes for the hasRating and hasPriceRange object properties. These classes can be imported from OWL ontology repositories and can be shared and reused by other ontologies. Since there are multiple classes within a given service ontology, one of them needs to be designated the main service class where service providers register. For this example, it is the *Restaurant* class. Figure 4.5 illustrates how the *Restaurant* class relates to the other classes via its object properties. A complete view of the restaurant ontology and the imported classes is shown in Figure 4.6. Here, the *Neighborhood*, *Cuisine*, *Price* and *Rating* classes are imported. The first level of the network separates the *Restaurant* by location. Below, I discuss details on how service data is distributed within the CAN.

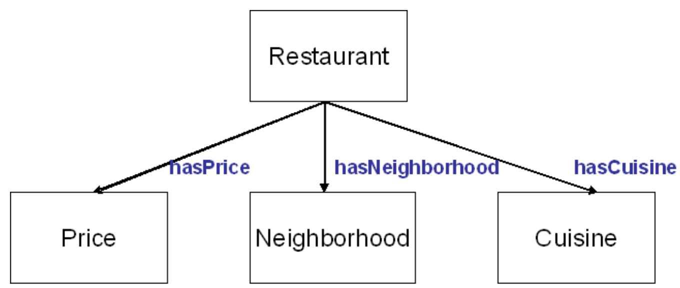

Figure 4.5: Class relationships within the Restaurant ontology

Annotation Properties

OWL uses annotation properties to indicate information about a class useful for documentation and other extra-ontology purposes rather than for classification. In this case, GloServ service ontologies define a number of annotation properties which describe how the ontology is to be parsed and interpreted for distribution in the CAN. Every service ontology in GloServ must contain these annotation properties. The annotation properties and their definitions are listed below:

- mainClass: an annotation property of the *owl:Thing* class located at the root. It specifies which class is the main service class in the ontology in order to distinguish from other (possibly imported) classes within the ontology. For the restaurant ontology, the main service class is the *Restaurant* class.

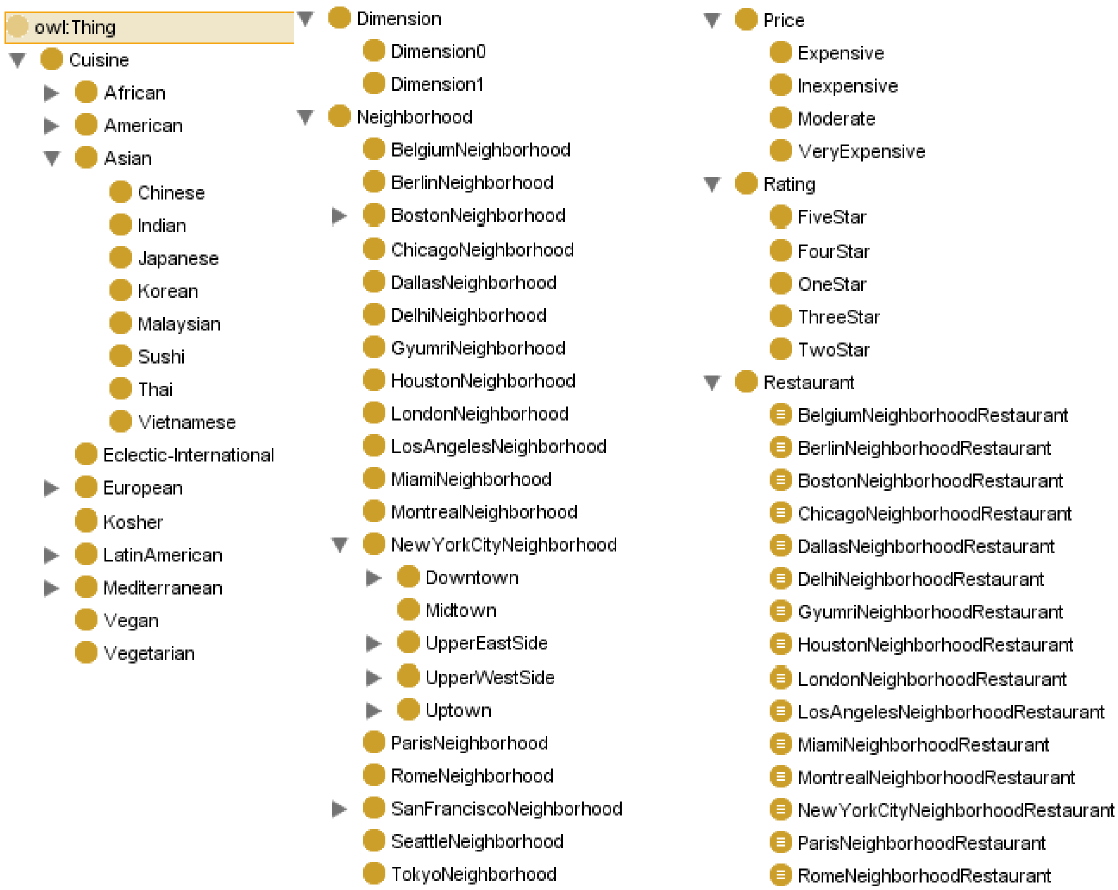

Figure 4.6: Complete view of the restaurant ontology and imported classes

- hostName: an annotation property of the *owl:Thing* class and specifies the host name of the P2P node.

- hasNetworkLevel: an annotation property of the *owl:Thing* class which specifies the subclass level to where the CAN distribution takes place. For example the ontology may be nested with many class levels. The hasNetworkLevel property identifies which class level to distribute the CAN by. In Section 4.3.3, I describe the CAN overlay network distribution in greater detail.

- isSuperNode: an annotation property of the *owl:Thing* class which specifies whether or not this node is a super node within the CAN.

- numDimensions: an annotation property of the *owl:Thing* class which specifies the number of dimensions the CAN has. CAN dimensions were discussed in Chapter 3.
- hasDimension: an annotation property of the main class and inherited by the main class's restricted subclasses. It specifies which dimension the restricted subclasses belongs in. Every restricted subclass of the main class must belong to a dimension within the CAN.
- hasKey: an annotation property of the main class and inherited by the main class's restricted subclasses. It specifies the key value of the restricted subclass within the CAN dimension it is located in.

Given this ontology construct, a CAN network can be generated automatically.

4.3.3 Generating CAN Overlay Networks

Overview

The CAN architecture is generated as a network of n-level overlays, where n is the number of subclasses nested within the main class. Let us take the *Restaurant* class and its subclasses as our example. The ontology classification and the CAN overlay network generated is shown in Figure 4.7. The first CAN overlay is a d-dimensional network which has the first level of subclasses of the *Restaurant* class. The number of dimensions is determined by the number of nodes contained within the CAN. As stated in [110], in order to achieve a query hit within $O(\log_2 n)$ number of hops, d must be $(\log_2 n)/2$. Thus, we approximate the number of nodes necessary for the network and calculate the dimension accordingly. The number of nodes is dependant on the estimated load of the network, given a service class. Since each CAN handles a certain service class and the number of service classes per ontology is limited to thousands or even hundreds of classes, this limits the number of nodes from the order of millions to thousands or even hundreds of nodes. However, if a CAN needs millions of nodes, we simply fix the dimension to a reasonable size and the worst case number of hops will then be $O(n^{1/d})$.

Each node will hold instances of a set of classes. Service registration and query processors route messages by creating a temporary *query* class within the service ontology and classifying it in order to find out which classes it belongs to. This is described in greater detail in Chapters 5 and 6. The query class gets routed down the CAN overlay networks. Continuing with the restaurant example above, a restaurant query is routed to the *Restaurant* server. The *Restaurant* server's query processor creates a temporary query class within the *Restaurant* ontology and classifies it. If the query message contains a search for restaurants in New York City, it will be classified under the *NYCRestaurant* class and sent to the nodes that handle information about New York City restaurants. The *NYCRestaurant* servers may either store instances of New York City restaurants within its servers or distribute this information further in a *subnetwork* of CAN GloServers. A subnetwork distributes the subclasses of the main class in the ontology in another CAN overlay network.

CAN Generation with Restricted Subclass Dimensions

As in the hierarchical network generation, a CAN is generated by a pool of servers and a service classification ontology. The initial server represents the main class in the ontology and peer nodes represent the restricted subclasses of the main class. We continue with the restaurant example to explain the generation of CAN networks.

The CAN network initially starts with a node that represents the *Restaurant* class. The *Restaurant* class will have a set of restricted subclasses. These are classified with a reasoner, such as RacerPro. The classification will produce a new ontology which will cause related siblings to form relationships with each other. These relationships include superclass, subclass or equivalence relations. Once the siblings are classified, they are assigned to a $\langle dimension, value \rangle$ numeric key. This is done by dividing the classes into the

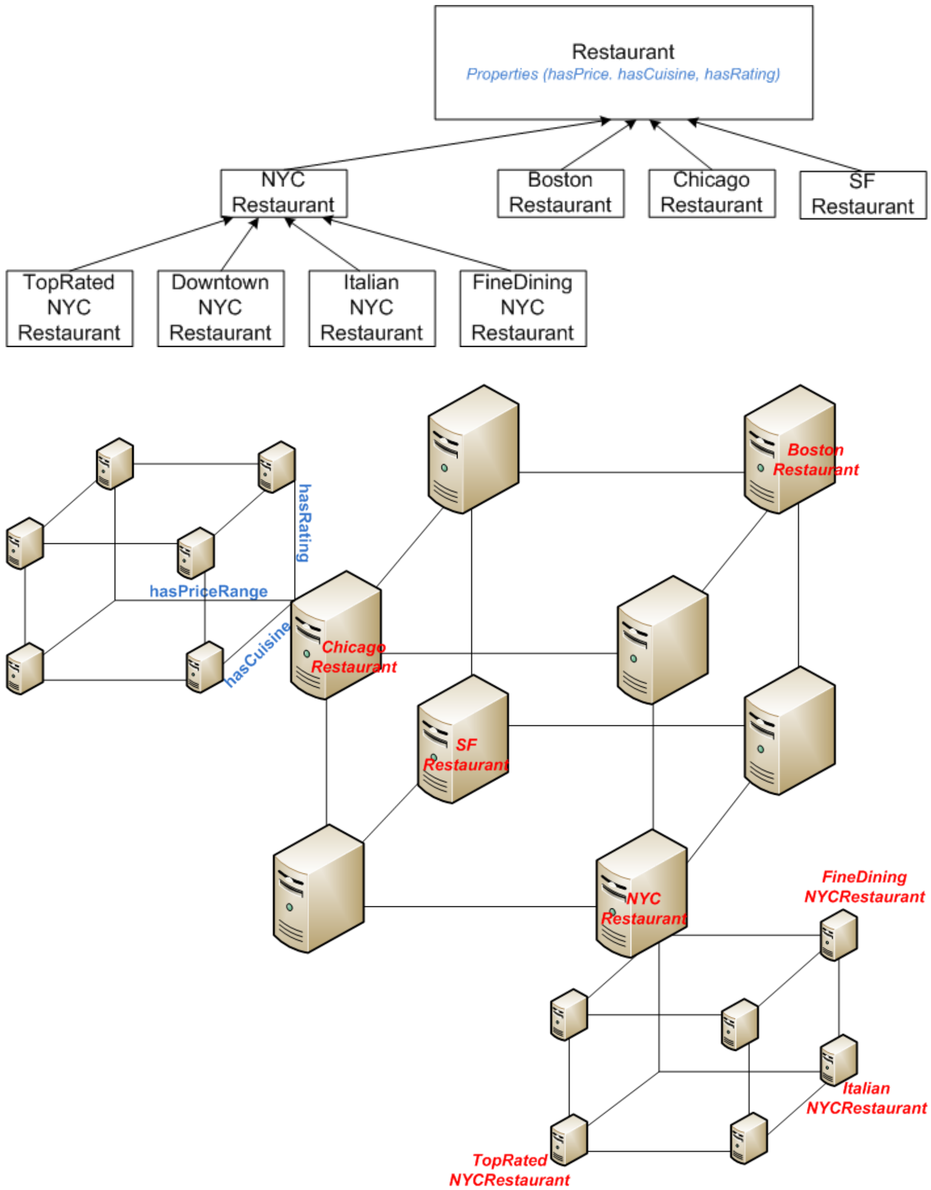

Figure 4.7: CAN overlay network

number of dimensions set in the CAN and assigning each class within a dimension a unique numeric key. Figure 4.8 illustrates how a list of subclasses under the *NYCRestaurant* class is converted to a $\langle dimension, key \rangle$ pair. It must be noted that there may not be a need to further distribute the *NYCRestaurant* class as one or two servers for this location may suffice. However, for the purpose of illustrating the CAN network examples, let us assume that it is.

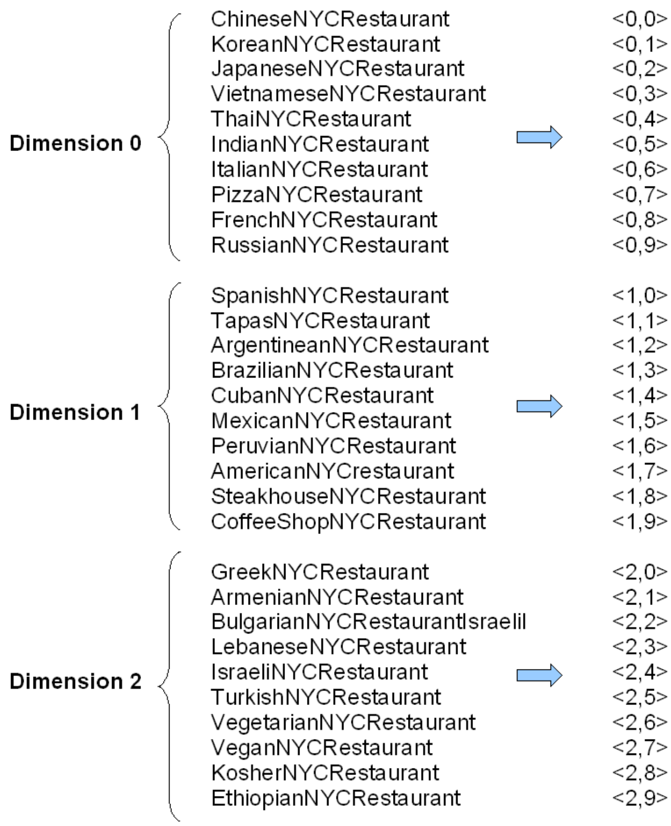

Figure 4.8: Conversion of classes to $\langle dimension, key \rangle$ pairs

As services register within the node and instances are created, they are classified into the subclasses of *Restaurant*. When a new node joins the network, one of the CAN dimensions is split into two and data is transferred to the new node. If there are c classes and d dimensions, classes are separated into d parts where each part contains c/d classes. When a new node joins the network, the CAN network generator chooses the dimension with the largest number of keys, which is essentially the longest dimension, and splits this into two. Thus, if the initial node has 3 dimensions with 10 classes in each dimension, then the range of each dimension is $[0-9], [0-9], [0-9]$. When a new node joins the network, one of the dimensions with the greatest number of key values is split and the resulting two nodes will have the following range of values: $[0-4], [0-9], [0-9]$ and $[5-9], [0-9], [0-9]$. Figure 4.9 illustrates the splitting into four CAN nodes in the network.

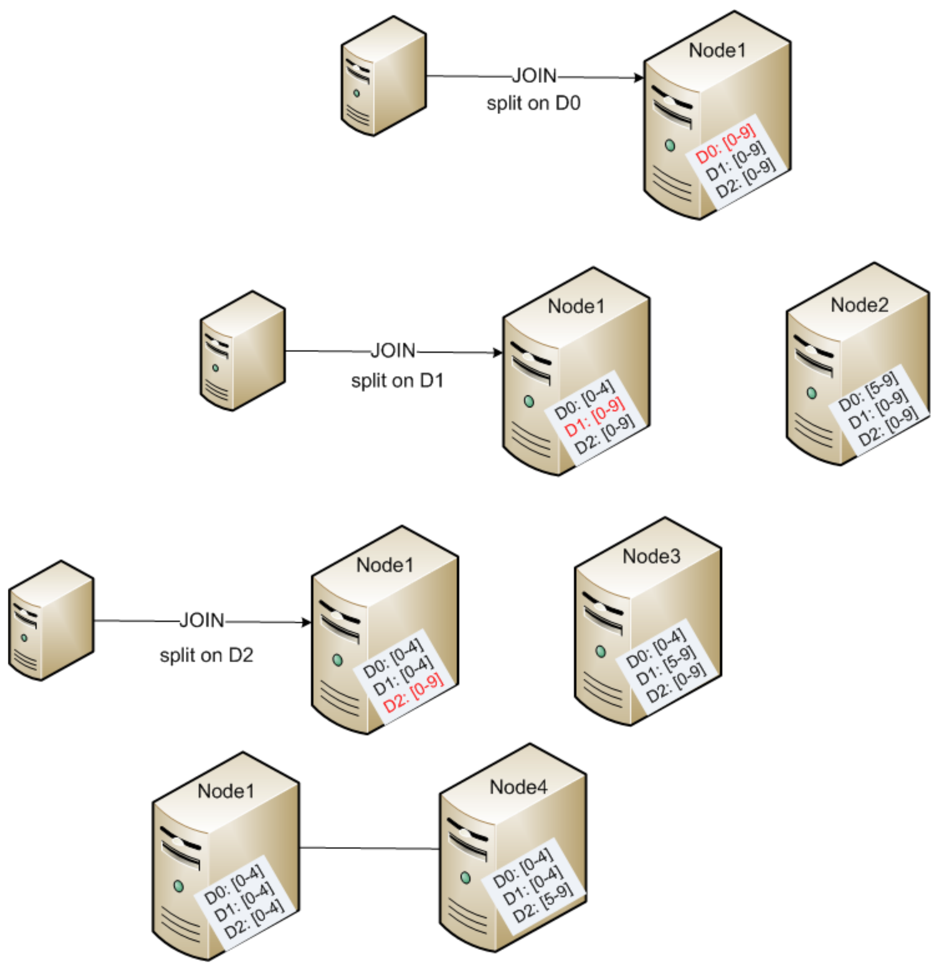

Figure 4.9: CAN node splitting into two nodes

Generally, in P2P DHTs, a node joins the system at a random point. This is also true for CAN. However, GloServ can improve load balancing and routing within the CAN by

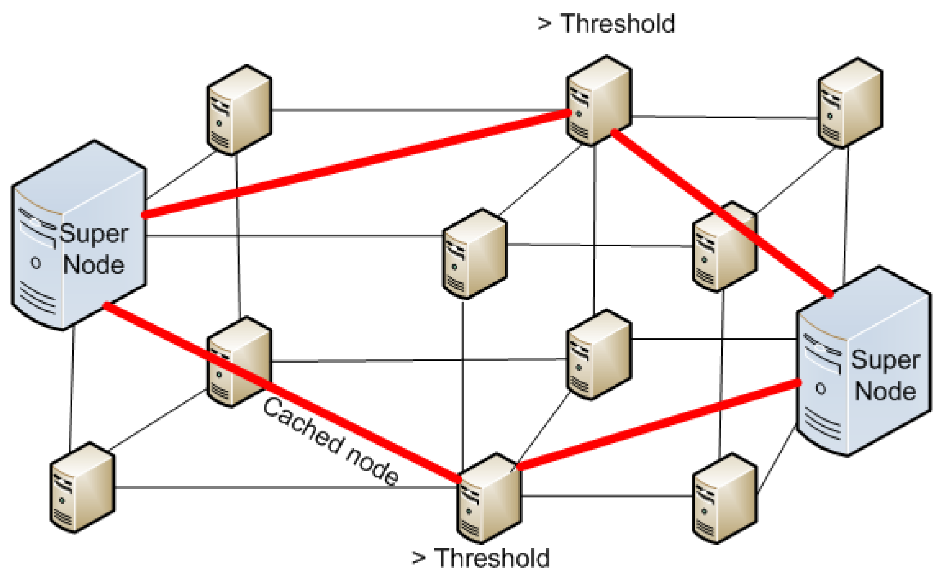

Figure 4.10: Supernodes pointing to frequently visited nodes

analyzing the query and registration messages. Instead of choosing a random point for a node join, it can be sent to an overloaded node to alleviate the number of requests made to that node which results in a load balanced network. A set of nodes are designated as *supernodes* of the CAN network. These nodes are the initial contact points of the CAN during a node join, query and registration. Query and registration messages pass through the supernodes. Thus, the supernodes can monitor the rate of queries and registrations routed to each subclass. This information will be shared among all the supernodes periodically so that all of them will have the same view of the network state. Those subclasses which exceed a certain threshold value of queries and registrations are considered overloaded nodes. Thus, the new node entering into the system will be routed to the node which is the most overloaded.

Additionally, GloServ can achieve faster routing of messages by again using the query and registration information. Similar to the case when a new node enters into the system, the supernodes monitor the number of queries and registrations that each subclass receives. The number of requests per class are periodically refreshed because there are different services queried for in various periods of time. For example, in the case of the *Travel* service, each season during the year will produce a surge in specific types of destinations. The supernodes check the subclasses that exceed a certain threshold value and cache the URNs of the nodes containing these subclasses. In this way, nodes that are queried most will always be one hop away from the supernodes. Figure 4.10 gives an overview of this.

CAN Generation with Property Dimensions

When a class does not have any restricted subclasses, we can distribute the service instances by generating a CAN where each dimension represents a property. We analyze the properties and their values such that instances that contain similar information will migrate together. There are two basic types of properties in OWL: object properties and datatype properties. Object properties have ranges that are other classes. Thus, the object property creates a mapping between objects. A datatype property on the other hand, maps classes to traditional datatypes such as strings and integers.

First we deal with object properties. These properties are separated into mandatory and optional categories by labeling it with an annotation property which specifies whether or not the property is optional or mandatory. If a property is mandatory, an instance of this class must have this property populated. Otherwise, this property may or may not be populated. Since mandatory properties will always have a value, we know that the only distinguishing characteristic of the keys generated with these properties is the value of the property. Optional object properties may or may not be populated which gives an added distinction to the property characteristic.

Next, we analyze the datatype properties. Since datatype properties can have any RDFLiteral value (such as ”integer” or ”string”), they result in an unbounded number of keys that can be generated. Thus, the only way we include datatype properties in the key generation is to see if an optional datatype property is populated. The presence or absence of this datatype property can be part of a CAN key.

The number of possible values of mandatory object properties is the product of their cardinalities. For optional object properties, the cardinality is incremented by one due to the possible null value. This means that the property may not have a value and thus is assigned to a null value. However, datatype properties are optional and do not range to a concrete number of classes but can have any number of values. Thus, the only information included in the CAN distribution for datatype properties is whether or not a property value is present. Thus, if we let p_i be the number of possible values for the i^{th} property and there are l mandatory object properties, m optional object properties and n optional datatype properties, then the total number of combinations of property values is:

$$\prod_{i=1}^{l} p_i \cdot \prod_{i=1}^{m} (p_i + 1) \cdot 2^n \ .$$

We recommend that classes that have CAN networks generated via their property values be tightly restricted classes. Tightly restricted classes have many of their properties restricted to a set of values already, thus leaving only a small number of properties that are not restricted. This results in a smaller subset of properties assigned to the CAN dimensions because at this level only a few properties are left unrestricted. As a result, exponential runtime when calculating the query key combinations is avoided. Taking Figure 4.7 as an example, the class *TopRatedNYCRestaurant* will already have its hasRating property restricted. Thus, this leaves the properties hasPriceRange and hasCuisine which are not restricted. Therefore, the CAN has two dimensions where each dimension is assigned one of the properties. If a class has no restricted subclasses and many properties, the ontology should be redefined where class restrictions can be formed in order to create an efficient CAN network. Figure 4.11 shows how servers are distributed in a CAN for the *TopRatedNYCRestaurant* class using the generated keys. We make a CAN using the example above and focus on a 2-property class for simplicity. The grid is partitioned into various spaces where each server handles a particular property combination.

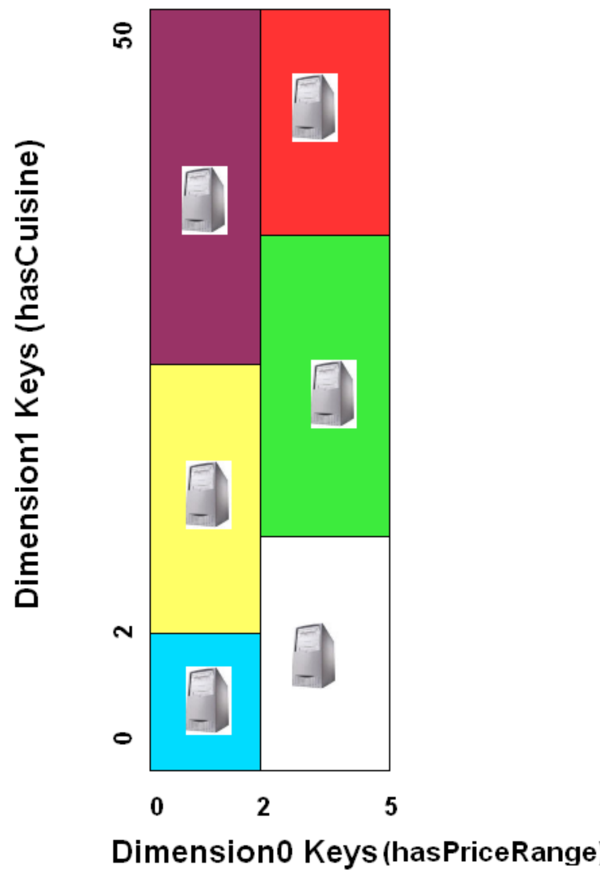

Figure 4.11: GloServers in a CAN

4.4 Implementation

The implementation of the GloServ library is in Java. The library uses two major software components, namely the Protege OWL API [83], which provides functionality for basic ontology operations such as creating classes and properties and the Racer Pro reasoner which is the ontology reasoning engine that runs on every GloServer. In the sections below, I discuss the implementation of the ontology and CAN network generation.

4.4.1 Ontology Generation

Generating a New Service Ontology

Since the use of ontologies is fairly new in the service discovery community, we cannot rely on the fact that ontologies for certain service domains already exist. Therefore, as new service classes arise, experts within a domain will engineer ontologies for each domain. I built a tool that generates an OWL ontology given a set of configuration files. This ontology is then converted to a CAN ontology which is used to map the CAN network. The classes *OWLOntologyGenerator* and *CANOWLOntologyGenerator* in the GloServ API handle this functionality, delegating to the Java Protege-OWL API for basic ontology operations such as creating classes and properties. The configuration file has a number of key words which indicate what to add within the ontology. Below I describe the various ways these are used in order to construct an ontology.

- Creating Classes: Classes are created by the *SuperClass* key word. The following format creates a class named A and a list of subclasses of *A*, named *B* and *C*:

  ```
  SuperClass=A
  B
  C
  ```

- Creating Properties: There are three main types of OWL properties: Datatype, Object and Annotation properties. Below is an example of a dataType property which is named `hasFreeSeats`. The domain of this property is the *Restaurant* class and the range is an integer. The maximum cardinality is set to 1. This means that this property can only have at most one value. Further, the property `hasFreeSeats` has two annotation properties that label it in various languages such as English or German. This provides a more user friendly label such as ”Free Seats” when displaying this property to a user in a form or other venue, rather than displaying the property name hasFreeSeats directly.

```
Property=DataTypeProperty
Name=hasFreeSeats
Domain=Restaurant
Range=http://www.w3.org/2001/XMLSchema#int
MaxCardinality=1
label.us=Free seats
label.de=Freie Plaetze
```

An example of defining an object property is seen below. The format is essentially the same except that the range is now another class within the ontology. Thus, this is defining an object property named hasCuisine which has as its domain the class *Restaurant* and range *Cuisine*.

```
Property=object property
Name=hasCuisine
Domain=Restaurant
Range=Cuisine
MaxCardinality=1
label.us=Cuisine
label.de=Kueche
```

- Restricted classes: A restricted class is one that has a property restriction on it. The example below creates a number of classes that have a restriction on the hasNeighborhood object property. The restricted class is a subclass of the class *Restaurant* class and the restriction is on every subclass of the *Neighborhood* class. The name of the class is the concatenation of the name of the superclass and the name of the class to restrict on. For example, if the *Neighborhood* class had two subclasses, *BostonNeighborhood* and *NYCNeighborhood*, then there would be two restricted classes created as subclasses of *Restaurant* named *BostonNeighborhoodRestaurant* and *NYCNeighborhoodRestaurant*. The quantifier specifies the type of restriction such as the existential or universal quantifiers. In this case, it is an existential quantifier that restricts the class by the property specified which is hasNeighborhood. Thus, for the examples above, the restriction for *BostonNeighborhoodRestaurant* is ”hasNeighborhood some BostonNeighborhood” and *NYCNeighborhoodRestaurant* is ”hasNeighborhood some NYCNeighborhood”. The action specifies whether or not to create a new class or to change the restriction of a given class.

```
RestrictedClass
Property=hasNeighborhood
Superclass=Class-Restaurant
Restriction=Subclass-Neighborhood
Name=Restriction+Superclass
Quantifier=some
Action=create
```

Converting to CAN Ontology

Once a service ontology is created, it is converted to be used by the CAN. The main service class's subclasses are analyzed, assigned numeric keys, and separated into the given number of dimensions. Another configuration file specifies information for the network level of the CAN. As mentioned in Section 4.3.3, the CAN networks can be separated into overlays based on the service class's construction. If the main class is dense with many subclass levels, the network construction can be separated into more than one level. The information following CAN information is contained in a configuration file:

- Import: imports the ontology to convert. This can either be a locally saved file or located on the Web.
- NetworkLevel: specifies the number of overlay networks to generate given the main class and its subclasses.
- Server: indicates the server's host name and is assigned as an annotation property in the owl:Thing class.

4.4.2 CAN Network Generation

A service class is handled by a GloServ CAN Network. The CAN is a P2P network that distributes the instances of the service class. Every node inside this GloServ CAN Network is a *GloServer*. As mentioned above, some of the GloServers are super nodes within the CAN network. Every GloServer is part of a CAN network which understands a set of messages that relate to building and maintaining the CAN as well as updating and querying for the instances. Key information which indicates the part of the CAN network a node handles is stored within the ontology file of the node. Other means of storing this information can also be implemented. Each GloServer also maintains an OWL ontology file which describes the service classes the node handles and holds the restricted classes that are used for classification of queries. Below, I describe the implementation of how a CAN network is constructed. Further information on the GloServ library classes are listed in the Appendix 17.

Implementation

Setting up a new CAN network is done in the *CANNetwork::generateCANNetwork()* method. The node running *CANNetwork::generateCANNetwork()* will be the first super node of the network. It will send remote "Request Join" messages to a set of nodes that are supposed to become the super nodes of the newly created CAN network. Afterwards, *CANNetwork::generateCANNetwork()* assigns nodes randomly to super nodes by sending the new nodes remote "Request Join" messages which instructs them to contact the randomly assigned super node. Once all nodes have been requested to join the network, the network is fully operational.

The code walk below is based on the scenario given in Figure 4.12. The scenario describes the following: Node1 is a super node that requests Node2 to join the CAN network. Node3 is Node1's current upper neighbor within its largest dimension.

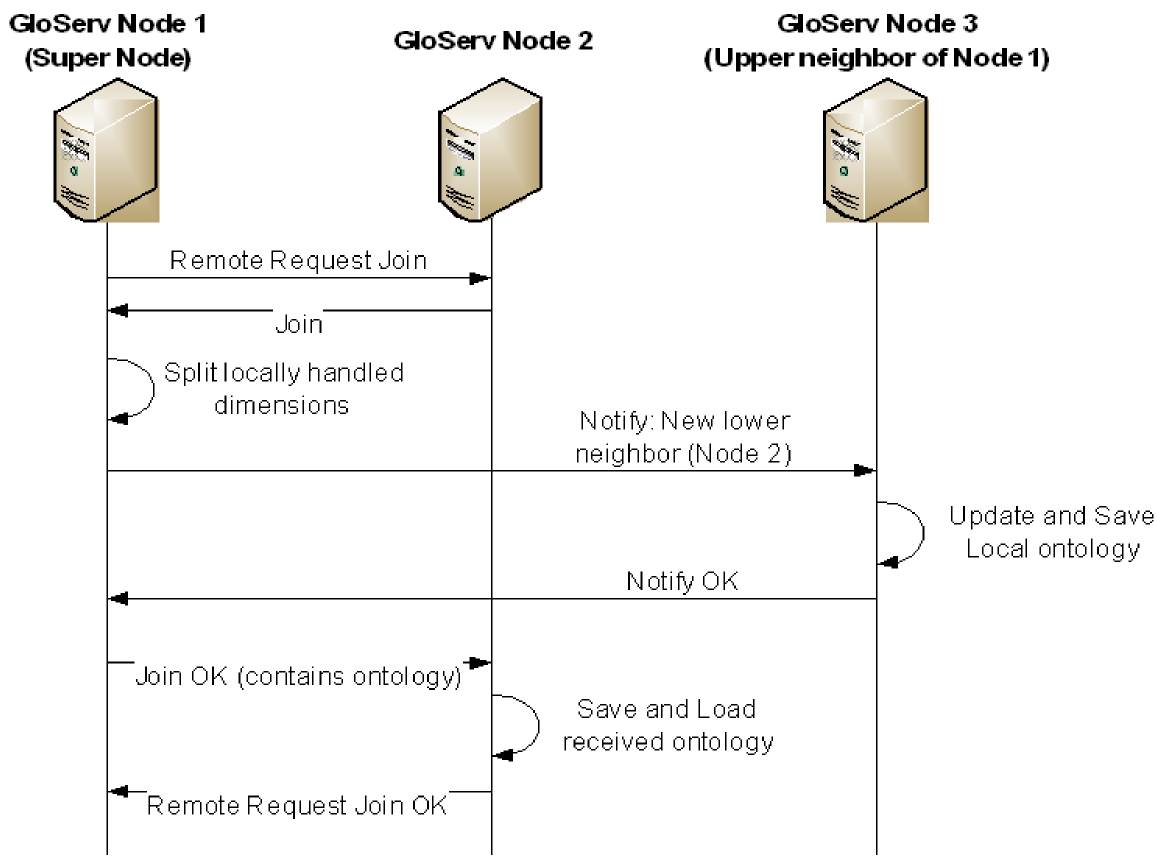

Figure 4.12: The code walk of a node joining the CAN

- Node1 issues a remote "Request Join" message to Node2.

- Node2 receives the remote "Request Join" message in *CANNode::processInput()*, which then calls *CANNode::incomingRemoteRequestJoin()*, which in turn calls *CANNode::outgoingJoin()*.

- Node2 composes a "Join" message and sends it to Node1 in *CANNode::outgoingJoin()*.

- Node 1 receives the "Join" message in *CANNode::processInput()* and calls *CANNode::incomingJoin()*.

- Node1 splits the keys it handles in the CAN space, stores half and transfers the other half to Node2 as *CANNode::incomingJoin()* calls *CANNode::insertNeighbor()*.

- *CANNode::insertNeighbor()* duplicates the local ontology and splits the dimension that has the most keys handled by the local node. If the local node has an upper neighbor within the split dimension, the upper neighbor is notified about the change of its lower neighbor. In this example, Node1's *CANNode::insertNeighbor()* sends a "Notify" message to Node3, its upper neighbor.

- Node3 receives the "Notify" message in *CANNode::processInput()* and calls *CANNode::incomingNotify()*. *CANNode::incomingNotify()* updates the local ontology and sends back an "OK" message.

- After receiving the "OK" message for the "Notify" message, *CANNode::insertNeighbor()* in Node1 continues operation. When the splitting is completed, Node1 sends a "Join OK" message back to Node 2. This "JoinOK" message contains the generated ontology file that Node2 is supposed to handle.

- Node 2 receives the "JoinOK" message and continues operation in *CANNode::outgoingJoin()*. It calls *CANNode::incomingOkJoin()*, which stores the received ontology and configures *CANNode* to handle it.

- Node 2 then sends an ”OK” message for the ”Remote Join” message, which ends the code walk.

4.5 Contributions

4.5.1 Generic Network Architecture

GloServ provides a generic network architecture where new service classes and subnetworks can easily be added into the GloServ architecture. In order to map an ontology to a network, GloServ uses data within an ontology. The ontology can either be read from a central repository or be generated from scratch and then stored in the repository. One of the benefits of using ontologies is the fact that they are easily shared. Thus, a central ontology repository may hold a number of ontologies for various service classes and thus can be accessed directly. The automation of the network creation allows a new service class to be inserted into the GloServ architecture fairly easily. The only requirements are a set of configuration files which GloServ uses to either read an already existing ontology or create a new ontology, convert the ontology to a CAN DHT and map the network and insert it into either an existing branch within the GloServ hierarchical network or create a new hierarchical branch.

This flexibility adds great benefit to service discovery because we anticipate new service classes forming over time. As new service classes arise, adding these for discovery within GloServ reduces to experts, within a domain, engineering an ontology and inserting this ontology into the GloServ system, which automatically generates a network architecture and inserts it into GloServ.

Furthermore, we also anticipate that these service ontologies may change over time. There is active research within the knowledge representation community that looks into how ontologies can automatically evolve according to how they are being used [107], [97], [109], [48]. Thus, one can foresee new subclasses or attributes being added or further refined within a given service ontology. Although we do not expect these changes to happen frequently, they may still occur. Given this predicament, the network also needs to be remapped from the ontology information and GloServ provides a simple way of doing this.

4.5.2 Distribution of Service Data

The GloServ architecture allows distribution of service data on a global scale because of its hybrid hierarchical peer-to-peer architecture. Distribution provides many benefits to an ontology-based service discovery system.

The first obvious benefit is that load is distributed across nodes in the network. Thus, a node is unlikely to be overloaded as a given service class is distributed in a CAN. Therefore, it is now possible to extend service discovery to dynamic services besides static ones. Although we use supernodes for initial query routing, this is still an improvement over a pure hierarchical architecture because many nodes can be assigned as supernodes.

Second, reasoning over very large ontologies can be a slow process. There are a handful of ontology reasoners, such as FaCT++ [121], Pellet [119] and Racer [90], that are available for use. Thus, distributing these ontologies across different service domains limits the number of classes within an ontology and thus not only provides the benefits of load distribution but allows for faster reasoning as well.

4.6 Conclusion

This chapter has introduced the design and implementation of the GloServ architecture which is an ontology-based global service discovery system. It uses a high-level service classification ontology to map a hierarchical network and individual service classification ontologies for mapping a peer-to-peer network of similar services. The next few chapters discuss how service registration, querying and instance processing are performed in GloServ.

Chapter 5

Service Registration

5.1 Introduction

A service registration in GloServ is represented by stored service data. Specifically, a service registration is an instance of a service class. For example, according to the ontology example given in Figure 4.6, when an Italian restaurant in New York City registers itself within the GloServ network, it becomes an instance of the *NYCRestaurant* class and is stored in the server, which handles the set of services within the *NYCRestaurant* class. The data is registered in a database where its attributes for location and cuisine are set to New York City and Italian, respectively.

There are two essential models of service registration, *manual* or *fully automated*. In a manual model, a service provider registers a service instance via a web-based form which is generated from the service ontology. This registration is then converted to an XML file which describes the registration and is sent over to a GloServer. In a fully automated registration model, a service automatically registers by inputting the attributes of the service ontology within an XML file and sending this to the GloServer. The metadata of the service is then routed within the GloServ network and stored in the servers which handle the service class that the registration belongs to. In this chapter, I discuss the details of the back-end service registration process and allude to the front-end interface in general terms, as details of the front-end interface are described in Part IV.

There are two types of registration messages: *user* registration and *internal* registration. A user service registration refers to a registration message that is received from the front-end user interface and is propagated down the CAN overlay networks. An internal registration refers to the registration messages that are propagated internally within a single CAN. Below, I describe how both types of registration messages are processed and routed throughout the network.

User service registration and querying use the same mechanisms to find the correct node for messages coming in from the front-end. However, internal registration and querying differ in the propagation of the request within the CAN. A registration message is routed to every node which handles the service classes it matches. This node then propagates the internal query until it hits a matching node, which is a node that handles that service class. However, since we are using a DHT, not every node within the system needs to be updated. Thus, this system is robust enough to handle dynamic services that require frequent updates.

5.2 User Service Registration

User service registration refers to registration messages that enter in through a front-end user interface. These registration messages are routed through the hierarchical network and down each level of the CAN overlay networks. As mentioned above, I describe a generic front-end interface and go into further details of the web-based front end we have built in Part IV.

5.2.1 Routing

Initially, a user contacts a GloServ front-end user agent and enters a service name. The initial GloServer is found by following steps one to three in Figure 5.1. Once the correct

GloServer is contacted, the front-end user agent obtains the ontology pertaining to that service class. The ontology is parsed and displayed to the user as a form to fill out. The ontology itself will have annotation properties that indicate which class is the main service class as well as other properties that indicate how to display the form.

The form represents the main service class and displays all the properties of this class; in the example of the *Restaurant* service class, the main class is *Restaurant*. When the user fills out this form, the values are parsed and converted to a first-order predicate logic query statement. The front-end user agent then sends this query statement to the back-end *Restaurant* GloServer. A sample registration contains restrictions on various properties such as:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Korean`) **or** (*hasCuisine* **some** `Chinese`)

Classification is the process of determining which restricted classes a query belongs to. The *Restaurant* GloServer creates a *query* class with this restriction and classifies it in its ontology. The *query* class is a temporary class which represents the first-order predicate logic statement that contained within the registration message. Thus, the input is a query string and the output is a list of OWL classes. Since the subclasses of the *Restaurant* class are restricted by location, the *query* class gets classified as a subclass of the *NYCRestaurant* class. The GloServer forwards the registration message to the nodes that handle *NYCRestaurant* classes. If the *NYCRestaurant* server does not have a sub-CAN network, then the registration is processed in that node by inserting the registration in its database. If the *NYCRestaurant* server has a sub-CAN network, then a *query* class is created in the ontology of the *NYCRestaurant* server and classified again. Since the *NYCRestaurant* class has subclasses that have cuisine restrictions, it may have another sub-CAN network which distributes restaurants by cuisines. Thus, the *query* class is classified under the *ChineseNYCRestaurant* and *KoreanNYCRestaurant* classes and sent to the servers that handle these classes within the sub-CAN network. This process repeats until there are

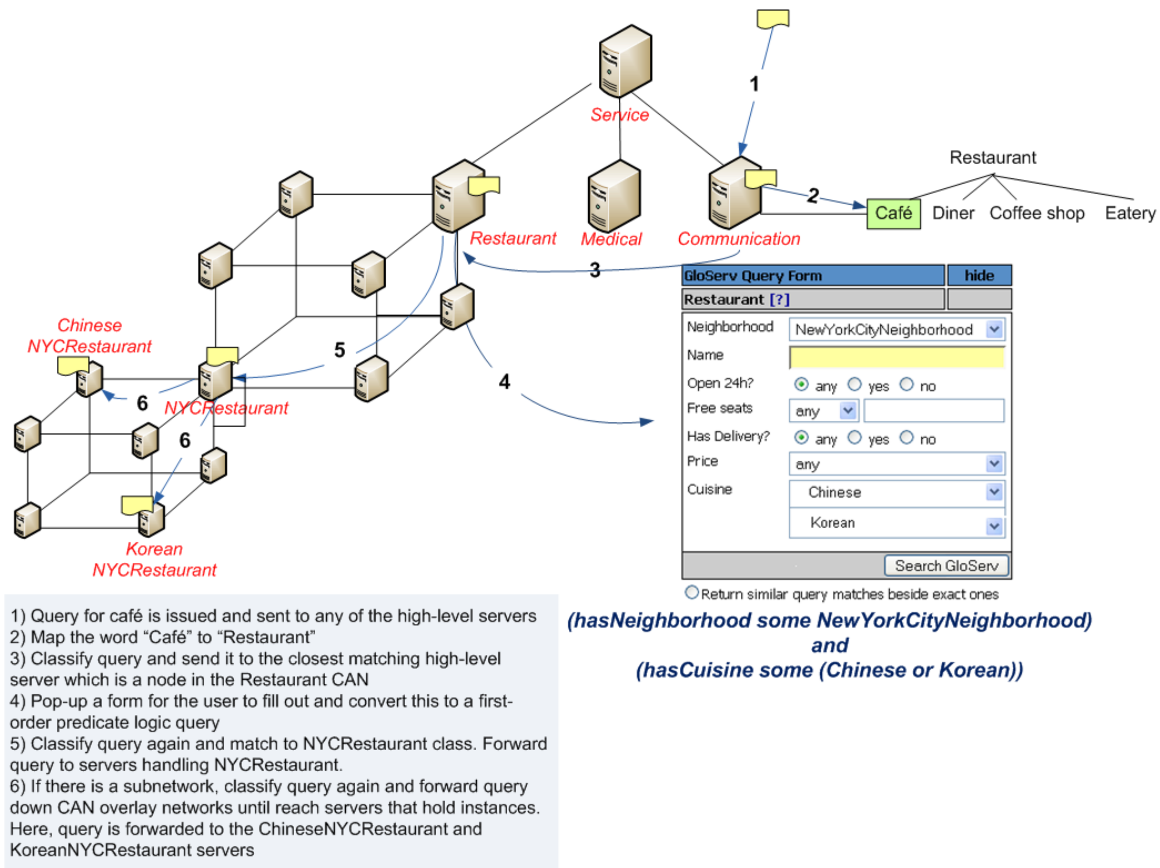

Figure 5.1: Example query form for Restaurant service class

no more overlay networks to send the query to. Figure 5.1 illustrates how a user service registration routes through the CAN overlay networks.

5.2.2 User Service Registration Message

GloServ messages are similar to HTTP [79] and SIP [113] messages. It is a text-based message which has a number of headers. The first line of the message contains a method which describes the type of message it is (JOIN, REGISTER or QUERY). Subsequent headers contain information used for message processing.

The headers in a user service registration message contain the following information:

- REGISTER: method for a registration message

- regType: indicates whether it is a *user* or *internal* registration message
- mainClassName: specifies the main service class
- query: contains the first-order predicate logic query
- authToken: contains an authorization token that a service provider can use in order to verify itself and update its registration

An example of a user service registration message is the following:

REGISTER

regType: userRegister

mainClassName:Restaurant

query:hasLocation some NYC and hasCuisine some Chinese or hasCuisine some Korean and hasName has ”EastMeetsWest” and isOpen24hours has false and hasDelivery has true and hasLongitude has ”-73.969471” and hasStreetAddress has ”875 3rd Ave” and hasLatitude has ”40.757602”

authToken:myToken

5.3 Internal Service Registration

Internal service registration refers to the registration messages that are routed only within a single CAN network. These messages do not require classification since the classes they match to are hashed to their numeric keys. They are routed to and registered in all the servers that handle the classes they are matched to. This is because it is anticipated that there will be many more queries than registration messages, even for dynamic service registrations. Thus, when a query is issued, the first hit results in all the matches for that query preventing it from propagating further through the CAN. Below I describe the details of how internal registration messages are routed.

5.3.1 Routing

CAN With Restricted Subclass Dimensions

Only one of the supernodes within a CAN overlay network which receive a user service registration message create a *query* class and classify it within the ontology. Once the initial node determines the classes the *query* class belongs to, after the initial classification, it hashes these classes to their $\langle dimension, key \rangle$ values, determined during the CAN generation described in Section 4.3.3. It then inserts these values in an internal registration message and routes the message to the CAN neighbors. Converting the ontology query to the CAN $\langle dimension, key \rangle$ pairs avoids multiple runs of the ontology reasoner which saves system resources within the node, consequently lowering the latency of the registration propagation as well as increasing the message load an super node can handle.

If a node receiving this internal registration message is a match, the node checks to see whether instances are stored within its own node and whether or not there are further subclasses which are distributed in another overlay. In the former case, the registration message is processed and stored in a database. In the latter case, the same steps are taken as mentioned above where the first-order predicate logic query within the registration message is classified, related classes hashed to numeric $\langle dimension, key \rangle$ pairs and routed within the sub-CAN network.

Let us look at an example of how an internal registration message is processed. For the sake of simplicity, I will use a 2-dimensional CAN to illustrate the registration routing. Assume there is a *NYCRestaurant* ontology that has 20 subclasses, separated into 2 dimensions with 10 subclasses in each dimension, as illustrated in the first two dimensions of Figure 4.8. Here, the *ChineseNYCRestaurant* subclass is assigned to dimension 0 with key 0 and *KoreanNYCRestaurant* to dimension 0 with key 1. Let us say a restaurant offers

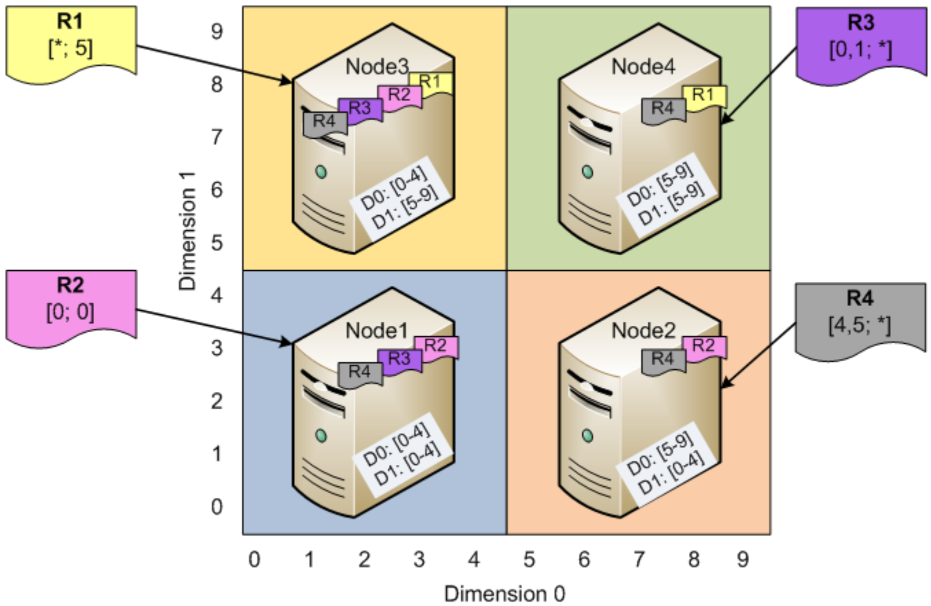

Figure 5.2: Registration propagation in the GloServ CAN

both Chinese and Korean foods. Thus, a service registration comes in with the restriction:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Korean`) **or** (*hasCuisine* **some** `Chinese`).

The service instance is registered in all the nodes that handle *ChineseNYCRestaurant* and *KoreanNYCRestaurant* classes. The format of a registration consists of values for each dimension, $[d_0; d_1]$, where d_i represents the key value at the *ith* dimension. If one of the values is a $*$, it is a wild card which means that dimension must be traversed to see if the other dimensions match the registration. The registration message for *ChineseNYCRestaurant* and *KoreanNYCRestaurant* in a 2-D CAN is $[0, 1; *]$.

Figure 5.2 illustrates different internal registration messages and how they propagate within the CAN. The registration message $R1$ enters into Node3 with key $[*; 5]$ which specifies that it is a class with key 5 in $Dimension1$. Since Node3 handles this class, it registers this instance in its database. It then sends this registration message down its other dimensions. In this case, it will send the registration down *Dimension*0 to its neighbor Node4. Since Node4 also has a match, it registers this in its node. The message is not propagated further since Node4 handles the maximum key in *Dimension*0.

*R*2 illustrates another registration message, [0; 0]. Initially this registration is sent to Node1 and registered there. Since the classes found are in both dimensions, it propagates this message down *Dimension*0 and *Dimension*1. Node2 and Node3 handle class 0 in *Dimension*1 and *Dimension*0 respectively. Thus, they register this instance in their databases as well. They do not propagate this message further since they handle the maximum key in that dimension.

*R*3 demonstrates a case where a service registration is classified under classes within the same dimension, such as in *Dimension*0 with keys 0 and 1. The registration message then includes both values in the following format: [0,1; *]. Here, the message enters into Node4. Since Node4 does not handle these classes, it sends it to its neighbor Node3 in *Dimension*0. Node3 registers this instance in its node and forwards it to its neighbor Node1 in the other dimension, *Dimension*1. Node1 also registers this in its node and since it handles the minimum key in *Dimension*1, it does not forward the registration message further.

Finally, *R*4 demonstrates a similar scenario to *R*3 where two classes in the same dimension are matched. However, in this case, since the nodes split the dimension where key 4 is in one node and key 5 is in another, the registration is propagated to all the nodes.

CAN with Property Dimensions

The same rules apply if the CAN is separated by property dimensions, as described in Section 4.3.3, rather than separated by restricted classes. As the classes become more restricted, it may not be necessary to restrict further properties within the class. But as the registration and query load grows within these servers, a sub-network may still be generated by assigning object properties not used in the class restriction to dimensions. For this case, since the ontology does not have anymore restricted subclasses to classify the *query* class in, if there is a sub-network distributed by properties then the registration processor hashes the remaining properties in the query statement that are not restricted to their $\langle dimension, key \rangle$ values.

From the previous example, the query class lands in the nodes that contain the *ChineseNYCRestaurant* and *KoreanNYCRestaurant* classes. If these classes are not broken down further into subclasses, then the remaining unrestricted properties are hasRating and hasPriceRange properties. Thus, the 2-dimensional CAN has dimensions representing each property. The hasRating dimension has five values, [OneStar, TwoStar, ThreeStar, FourStar, FiveStar], and hasPriceRange has four values [InExpensive, Moderate, Expensive, VeryExpensive].

Registration routing in this level follows the same method as routing within a CAN with restricted subclasses. The only difference is that the hashing of the $\langle dimension, key \rangle$ values come from the property values. For the registration example

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Korean`) **or** (*hasCuisine* **some** `Chinese`),

since a price range and rating is not specified, the instances are stored in the main *ChineseNYCRestaurant* and *KoreanNYCRestaurant* servers. But if a price range and rating is specified such as:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Korean`) **or** (*hasCuisine* **some** `Chinese`) **and** (*hasRating* **some** (`FourStar`) **and** (*hasPriceRange* **some** `InExpensive`)

then the registration initially gets mapped to *ChineseNYCRestaurant* and *KoreanNYCRestaurant* and then if there is another CAN network beneath these classes, the remaining properties that the CAN is distributed by are looked at and if they match the properties specified in the registration, the keys for these two properties are generated. In this case, the key generated is

[3;0] which indicate that `FourStar` maps to key 3 in $Dimension0$ and `Inexpensive` maps to key 0 in $Dimension1$. The registration is routed to all nodes that handle those values and the instance is registered within these.

From these examples, it can be seen that registrations are replicated in those servers that handle the same keys (or service classes). As more servers enter the network, the classes are further distributed and replicated less and less. Thus, for dynamic services, frequent updates can be issued efficiently without having to distribute the registration message to every node. However, depending on the type of registration and the way the CAN is distributed, the registration messages may be replicated across the servers that handle the same service classes. The ontology mapping algorithm assigns each class to a $\langle dimension, key\rangle$ value. Since it classifies these classes before the assignments are made, similar classes cluster together and are assigned keys that are closer to one another. Thus, as seen in the example in registration $R3$, the *ChineseNYCRestaurant* and *KoreanNYCRestaurant* classes reside in the same dimension with nearby keys so that a registration or query that has these values is limited in the number of nodes it is routed to. However, cases like $R2$ or $R4$ may exist where the classes are either in different dimensions or exist in a single dimension and thus are routed to more servers.

5.3.2 Internal Service Registration Message

An internal registration message has the same elements as the user service registration message except the following headers are added:

- hash: the conversion from the first-order predicate logic query to $< dimension, value >$ keys.
- route: specifies where the registration message must be routed to prevent each node from recalculating the message route.

An example of an internal registration message is shown below. Since the initial query is classified to the *ChineseNYCRestaurant* and *KoreanNYCRestaurant* classes, this is hashed to the value [0,1;*], which means that the classes have keys 0 and 1 in *Dimension*0 and is inserted into the hash header. The route header gives information on where to route the message. When a registration message entering a node is a match, then the node inserts a route header which indicates which $\langle dimension, key \rangle$ values to propagate the register message to. Since a registration must be registered in all matching nodes, then all other dimensions beside the key dimension are traversed. In this example, since the registration is hashed to [0,1;*], this means that nodes in *Dimension*1 must be traversed to find nodes that handle the keys 0 and 1 in *Dimension*0. The corresponding example is illustrated as $R3$ in Figure 5.2. Since $R3$ initially enters into Node4, Node4 does not handle this key so it routes the message to its lower neighbor Node3. Node3 is a hit so it registers the message in its server and inserts a route header in the internal query message. The route header, $[][0,1,2,3,4]$, means that the message needs to be routed to all other nodes in *Dimension*1 which handle keys 0-4 since Node3 handles keys 5-9 itself and already registered this message in its server.

REGISTER

regType: internalRegister

mainClassName:Restaurant

query:hasLocation some NYC and hasCuisine some Chinese or hasCuisine some Korean and hasName has ”EastMeetsWest” and isOpen24hours has false and hasDelivery has true and hasLongitude has ”-73.969471” and hasStreetAddress has ”875 3rd Ave” and hasLatitude has ”40.757602”

authToken:myToken

hash:[*;5]

route:[][0,1,2,3,4]

5.4 Implementation

This section describes how the code is excuted when $R3$ is routed through Node4, Node3 and Node1. Initially, *CANNode::processInput()* receives all incoming messages. The registration message is then passed to *CANNode::incomingUserRegister()*. The registration classification is done by the *RacerQueryClassifier* class, which uses the Racer Pro reasoning engine, running in a separate process, to classify the registered query. Although Racer itself employs very good caching techniques, we have added our own cache in *RacerQueryClassifier* which maps query strings to OWL class lists in order to avoid running the query through a classifier and save system resources. This improves the classification speed considerably, as discussed in the evaluation in Chapter 8.

- **Node4**
  - *CANNode::processInput(message)*: Node4 receives the registration message $R3$ in CANNode::processInput() which then calls CANNode::incomingUserRegister().
  - *CANNode::incomingUserRegister(message)*: Processes a user register coming directly from a user. It calls the following functions:
    - *CANNode::processUserRegister(message)*: classifies the query, determines all CAN coordinates the registration should go to, as well as the neighboring hosts that are responsible for those coordinates and stores instances in these nodes.
      - *QueryParser::parse(message.getQuery())*: parses the query statement from the query header of the registration message and assigns it to the variable *statement* of type *QueryStatement*. Query parsing is described in greater detail in Section 6.4.
      - *RacerQueryComposer::composeStatement(statement)*: converts the query statement into a Racer query statement and returns the string *racerQueryStatement*.

  · *RacerQueryClassifier::findQueryMatches(racerQueryStatement)*: runs the query through the Racer reasoner and obtains a list of classes which match the query and stores it in the vector *queryKeys*.

  · *GloServOWLModel::findUserRegisterNeighborHosts(queryKeys, queryHostKeyMap, statement, message.getMainClassName())*: Calls the function GloServOWLModel::findAllCombinations() which converts the classes in *queryKeys* to $\langle dimension, key \rangle$ values and returns this in the Vector *canCoordList*. It then calls GloServOWLModel:: findRegisterNeighborHosts() which inserts these into the hash map *queryHostKeyMap* that maps local or neighboring hosts to these $\langle dimension, key \rangle$ values. If the host is the same as local host, it saves the instance of this registration in its own database. In this case, Node1 is mapped to the values [0,1;*] and nothing is saved in Node4.

  ∗ *CANNode::routeRegister(message, queryHostKeyMap)*: Loops through each host in *queryHostKeyMap*, if the host is different than the local host then it creates an internal register message and sends it to the registered host. Here, Node4 will send the message to Node3. If the register message is the same as the local host then it creates the route header by finding all the neighboring nodes in the other dimensions and sends this message to the neighbors in the other dimensions.

- **Node3**

  – *CANNode::processInput(message)*: Node3 receives the internal registration message $R3$ in CANNode:processInput() which then calls CANNode::incomingInternalRegister(message).

  – *CANNode:incomingInternalRegister(message)*: Similar to *CANNode::incomingUserRegister()* except that it will not re-classify the query statement. It calls the following functions:

  * *GloServOWLModel::findRegisterNeighborHosts(message.getCanCoordList(), queryHostKeyMap, statement, message.getMainClassName())*: Same as the above call, however, in this case since Node3 is a hit, it saves the register message in its database.
  * *CANNode::routeRegister(message, queryHostKeyMap)*: Since *queryHostKeyMap* contains the local host, all the neighbors in the other dimensions are also inserted into the *queryHostKeyMap* and the register message is routed to these nodes as well.

- **Node1**
  - *CANNode::processInput(message)*: Node1 receives the internal registration message $R3$ in CANNode:processInput() which then calls CANNode::incomingInternalRegister().
  - *CANNode:incomingInternalRegister(message)*: as in the previous case, calls the functions:
    * *GloServOWLModel::findRegisterNeighborHosts(message.getCanCoordList(), queryHostKeyMap, statement, message.getMainClassName())*: since Node1 is a hit, it saves the register message in its database
    * *CANNode::routeRegister(message, queryHostKeyMap)*: There are no more nodes to route this message to so the function just returns.

5.5 Conclusion

This chapter has described how services are processed and routed within the GloServ network. Due to the underlying CAN architecture, frequent registration updates are more efficient than in replicated networks. The evaluation of the registration is given in greater detail in Chapter 8.

Chapter 6

Service Ontology Querying

6.1 Introduction

A GloServ service ontology query takes as input a service class's property values and outputs service instance data that matches the service properties. As in the registration, the query is represented as a first-order predicate logic query statement. The data returned to the user can include the service's property information as well as a reference to where the service actually resides, such as a hyperlink. Querying is very similar to the registration mechanism discussed in the previous chapter and can either be manual or automated.

As in the registration mechanism, there are also two types of query messages: *user* queries and *internal* queries. These play the same roles as in the registration where the *user* query enters through the front-end and is propagated down the CAN overlay networks and the *internal* query is routed within a single CAN. However, the query not only needs to propagate forward to the correct node but also return the results to the user in a manner that the user will understand. Furthermore, the user query may go through extra processing in the CAN super nodes depending on whether the user wants related matches in addition to exact ones. This chapter describes the core algorithms for ontology query routing and processing. I discuss extensions to ontology querying, such as combining services in a single search and combining key word search with ontology querying, in Chapters 9 and 10.

6.2 User Querying

User query routing follows the same exact steps as user registration routing described in the previous chapter. The main difference between these is in the classification process. A user query can either be an *exact* or *similar* query. The processing of these queries is described below.

6.2.1 Exact User Query

When a user issues a query from the front-end, she will receive exact matches, unless otherwise specified. This exact query is routed through the GloServ network and once it reaches one of the super nodes of the CAN network, the first-order predicate logic statement within the query message is turned into a restricted *query* class. The query processor within the GloServer classifies the *query* class in the ontology and matches it to classes that are equivalent classes or classes that are its children (subclasses). For example, suppose the query that enters the system is the following:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`)

If the ontology's main class, *Restaurant*, has restrictions by location within that CAN level, then the query matches to the *NYCRestaurant* class. If the *NYC* location class is broken down to neighborhoods such as *UpperEastSide* or *UpperWestSide* and the CAN includes nodes at this level, then the query is matched not just to *NYCRestaurant* but *UpperEastSideRestaurant* and *UpperWestSideRestaurant* classes as well, since these classes are also *NYCRestaurant* classes, but subclasses of it. In this manner, the query is propagated to all nodes that handle restaurants in New York City.

6.2.2 User Query with Similarity Matches

When a user issues a query but wants a broader range of results, she can specify that she wants similar matches as well as exact matches when issuing the query. The query form has a "similar results" button next to the service properties that a user can check off, indicating that the properties can be relaxed. For the following query:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`) and (*hasRating* **some** `FiveStar`)

if the user indicates that it wants similar cuisine matches in addition to *Chinese* restaurants in NYC then the query processing will also return a variety of Asian cuisine restaurants such as Korean or Japanese.

When a GloServer receives a query, it parses the headers which indicate the query type (exact or similar) and the properties to issue similarity matches on. The similar query matching algorithm finds the classes that are related to the *query* class's properties and looks into the non-disjoint siblings that have these property restrictions. Each property has a domain class and a range class. In order to find a related property, the range is classified and the sibling classes are considered. For example, the *Cuisine* class has the subclass *Asian* which has subclasses *Chinese*, *Korean*, and *Japanese*. When a query is issued for a Chinese restaurant that has a five-star rating in NYC, the query class will have the following restriction:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`) and (*hasRating* **some** `FiveStar`)

This query class is classified according to how the ontology is constructed. In our continuing example, it first is classified under the *NYCRestaurant* class. Since the query type is a similarity query on the cuisine property, the non-disjoint sibling classes of the class *Chinese* are analyzed and found to be *Korean* and *Japanese*. These classes are added to the original query in a disjunctive string converting the original query to:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese` or `Korean` or `Japanese`) and (*hasRating* **some** `FiveStar`)

The query is then classified and matched to the *ChineseNYCRestaurant*, *KoreanNYCRestaurant* and *JapaneseNYCRestaurant* classes and routed to the nodes handling these classes.

6.2.3 User Query Message

The headers in a user query message contain the following information:

- QUERY: label for a query message
- queryType: indicates whether it is a *user* or *internal* query message
- from: the sender's IP address
- mainClassName: specifies the main service class
- userQuery: contains the first-order predicate logic query
- queryMatch: can be ”exact” or ”similar” depending on the user's preference
- propertyMatch: if the queryMatch value is ”similar” then this header is inserted, indicating which properties to include in the similarity match
- queryId: unique query ID number
- parts: displays the percentage of results obtained
- cardinalityResults: displays the number of instances obtained from a node

QUERY

queryType:UserQuery

from:127.0.0.1:5501

mainClassName:Restaurant

userQuery: hasLocation some NYC and hasCuisine some Chinese or hasCuisine some Korean and hasName has ”EastMeetsWest” and isOpen24hours has no and hasDelivery has true and hasLongitude has ”-73.969471” and hasStreetAddress has ”875 3rd Ave” and hasLatitude has ”40.757602”

queryMatch:exact

queryid:354026

parts:0;1

cardinalityResults:0

6.3 Internal Querying

Internal query routing in GloServ is similar to registration routing except for two points. First, when the query is matched to a list of classes after classification and routed to the CAN servers handling those nodes, the query does not continue propagating once a match occurs. This is because a register message is routed to every server that handles that registration class, which prevents the query from being forwarded to all the nodes in the network that handle that service class. Second, unlike in the case of registration, the initial node issuing a query in the CAN must keep track of the query results and return these to the user in a list that is easy to understand. The interface for this is described in greater detail in Chapter 12. Below, I describe how both of these scenarios play out when a user issues a query in GloServ.

6.3.1 Internal Query Routing

CAN with Restricted Subclass Dimensions

The user query processor either leaves the query the same or changes it to accommodate logically similar results and classifies it in the ontology. It then passes the list of matching classes to the internal query processor which converts these classes to $\langle dimesnion, key\rangle$ values. If an exact user query is

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`),

the query message is hashed to $[0; *]$ as described in the previous Chapter in Section 5.3.2. If this is a similar user query then it is changed to include *Korean* and *Japanese* cuisines in addition to *Chinese* and thus the key becomes $[0, 1, 2; *]$.

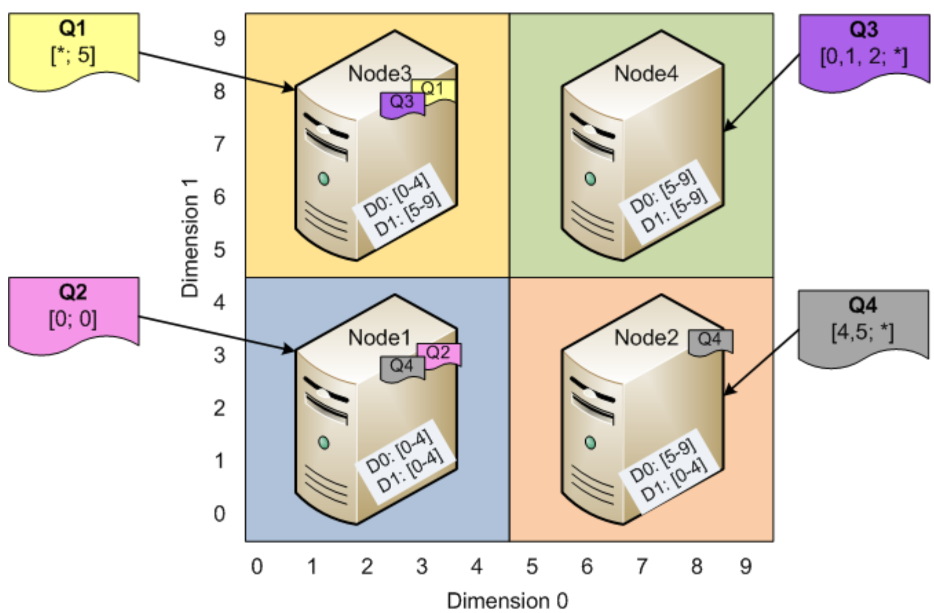

Figure 6.1: Query propagation in the GloServ CAN

Figure 6.1 illustrates this query message as $Q3$. Node4 receives $Q3$ and since it does not hold any of the classes in the dimensions specified, it forwards the query message to its lower neighbor on *Dimension*0 which is Node3. Node3 handles the first five keys in *Dimension*0 and thus processes the query and returns the results to the user.

The remaining query examples in Figure 6.1 are the same as those in the registration examples. $Q1$ and $Q2$ are immediate matches and thus not propagated further. $Q4$, $[4, 5; *]$, is a query which has values that are split within a dimension. It enters into Node2 which handles keys 5-9 in *Dimension*0 and forwards the query to its lower neighbor, Node1, which handles the remaining lower keys in *Dimension*0. The results are then sent back to the user.

CAN with Property Dimensions

As mentioned in Section 4.3.3, if there is an additional sub-network underneath the *ChineseNYCRestaurant* class, and it does not have any restricted subclasses, then they are distributed by the remaining object properties such as `hasRating` and `hasPriceRange`. Thus, the query is first processed in the *ChineseNYCRestaurant* server and then sent out further down to the sub-network. For the query example:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`)

since the property `hasRating` or `hasPriceRange` is not specified, then there are a total of $5x4 = 20$ possible query combinations to issue in the CAN sub-network. This is because hasRating property has five values, [OneStar, TwoStar, ThreeStar, FourStar, FiveStar], and hasPriceRange has four values [InExpensive, Moderate, Expensive, VeryExpensive] and each of these are assigned to a dimension. If the query was more specific, such as:

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`) and (*hasRating* **some** (`FourStar` or `FiveStar`)) and (*hasPriceRange* **some** `InExpensive`) where the rating and price range was specified, then the query is hashed to the key $[3,4;0]$ and sent out as an internal query.

6.3.2 Internal Query Message

The internal query message contains the same headers as the user query message except for the additional header, `hash`, that contains the conversion from the original first-order predicate logic query to a $< dimension, value >$ key. Unlike the register message, a query message does not need a `route` header because the query is routed only down a single dimension.

6.4 Query Parsing

GloServ uses the query syntax offered by Protege. The smallest building block of queries are statements like:

A **some** `B`

where *A* is an object property and `B` is a class within the range of the property *A* is pointing to or

A **has** `B`

where *A* is a datatype property and `B` has a datatype value such as a string, integer or boolean. These can be composed into more complex queries by using logic operators such as OR and AND. Queries can also be nested into each other. For example, a query could be:

A **some** (`B` and (`C` **has** `D`)).

A user query is normalized and then simplified. Normalization transforms a query to an OR-ed list of AND-ed statements. For example the following query is converted from

(*A* **some** `B` or *C* **some** `D`) and
(*E* **some** `F` or *G* **some** `H`)

to

(*A* **some** `B` and *E* **some** `F`) or
(*A* **some** `B` and *G* **some** `H`) or
(*C* **some** `D` and *E* **some** `F`) or
(*C* **some** `D` and *G* **some** `H`).

Simplification puts the boolean expressions separated by an OR into a Java Vector list. This way, each expression becomes a single query to issue.

In addition to the query syntax offered by Protege, GloServ adds some features to the query syntax in order to support the sub-query mechanism. Chapter 9 discusses this in greater detail.

6.5 Query Results

When an internal query propagates through the CAN and retrieves instances, it returns the instances in a reply message back to the user who issued the query specified in the from header in the query message. Since the GloServ CAN network is a ”black box” to the user, the user has no knowledge of the number of results that will be returned to her. Similarly, GloServers do not know how many other nodes a query will traverse once it has left its node.

Thus GloServ implements a simple mechanism to keep track of answers.

Every query has a value which could be read as *the percentage of the overall results this query covers*. Every GloServ node that forwards a query message changes this percentage. For example, if the initial query is answered by the first node and sent to three neighboring nodes in the CAN, the initial query answers are calculated to be 25% of the overall result set and every single query sent to neighboring nodes amounts to 25% of the overall answers. If one of these queries is then split into five queries, each of these new queries will amount to 5% of the overall result set.

Using this mechanism, the user can roughly approximate how much of the overall results she received at any point in time. It must be noted that this is not an accurate metric in the sense that a value of 50% does not necessarily mean that the user received 50% of the query results. However, when the value reaches 100%, the user can be completely certain that she received all the results. This also allows the front-end systems to load query results in batches where the first set of query results are shown to the user and if the user is satisfied, the query is not propagated further. If the user clicks on ”next”, the next batch is loaded.

6.6 Implementation

6.6.1 Query Routing

This section describes the code walk of the query message $Q4$ seen in Figure 6.1. The initial steps of receiving a user query message, parsing and classifying it are the same as that in the user registration message described in Section 5.4. Similar to registration processing, *CANNode::processInput()* receives all incoming query messages. It passes a user query message to *CANNode::incomingUserQuery()* and an internal query message to *CANNode::incomingInternalQuery()*. The query classification is done by the *Racer-QueryClassifier* class, which uses the Racer Pro reasoning engine, running in a separate process, to classify the register query.

- **Node2**
  - *CANNode::processInput(message)*: Node2 receives the user query message $Q4$ in CANNode:processInput() which then calls CANNode:incomingUserQuery().
  - *CANNode::incomingUserQuery(message)*: normalizes the query to an OR-ed list of AND-ed statements. It then splits the query so that no single query contains any OR operators except for nested subqueries. It assigns the number of query parts in the query header and passes each query part to *CANNode::processUserQuery()*. It then routes the queries which are mapped to neighboring nodes by calling *CANNode::routeQuery()*.
    - *CANNode::processUserQuery(message)*: similar to CANNode::processUserRegister(). It classifies the query, hashes it into its $\langle dimension, key \rangle$ values and determines all CAN coordinates the query should go to, as well as the neighboring hosts that are responsible for those coordinates. Here, classification also includes checking to see if the query is an exact match or a similar match. In addition, if the local node stores any instances, the instances are retrieved and sent back to the user. In this case, since the query is $[4, 5; *]$ and Node2 handles key 5 in $Dimension0$, it retrieves instances matching this class and sends this to the user.
    - *CANNode::routeQuery(message neighborHostMap*, *userQuery*, *userQueryNormalized*, *instanceList)*: Based on the CAN routing decision made earlier, an internal query message is created and forwarded to $Node1$. Also, the instances that were found in the previous step are sent back to the user.
- **Node1**
  - *CANNode:processInput(message)*: Node1 receives the internal query message and passes it to CANNode::incomingInternalQuery().

- *CANNode::incomingInternalQuery(message)*: similar to CANNode:: incomingUserQuery(), but does not need to classify the query again. Instead, it looks at the $\langle dimension, key\rangle$ values and sees that key 4 in $Dimension0$ is a match so it retrieves the local instances and calls CANNode::routeQuery.
  - *CANNode::routeQuery(message, neighborHostMap, userQuery, userQueryNormalized, instanceList)*: since this node covers remaining class, it does not route the query further and returns instances to the user.

6.6.2 Query Parsing

The query parser is implemented using JFlex [21] and CUP [6]. JFlex is a lexer that offers a language to describe the different kinds of tokens. For example, GloServ defines string and literal tokens, tokens for operators like **has**, **some** and **or** and tokens for items like brackets. CUP is a parser and offers a language to describe a grammar. The grammar uses the tokens defined with JFlex. GloServ defines basic statements like *A* `some` `B`, complex compositions like (*A* **some** `B`) or (*C* **some** `D`) and so on. In the language CUP offers, Java code is inserted to express what the parser should do when it finds the constructs defined by the grammar. Therefore, GloServ defines classes that represented all kinds of language constructs used in queries such as SimpleIdentifier, SomeStatement, HasStatement, OrExpression. Both, JFlex and CUP generate Java code. In the QueryParser class, GloServ calls the methods offered by the generated code to parse queries. The generated code then creates a Java representation of the query based on the language construct classes described earlier. The QueryParser class defines a number of methods to perform common tasks on query statements which are described in greater detail in the Appendix.

6.7 Conclusion

This chapter has described how GloServ ontology queries are processed and routed through the GloServ network. GloServ provides richer service descriptions than traditional attribute-value pair service descriptions or text-based service descriptions due to its ontology representation. Thus, GloServ allows services to be queried for beyond attribute-value searches to also include searching for services that are logically similar to what is being searched for. The evaluation of GloServ querying is described in detail in Chapter 8.

Chapter 7

Service Instances

7.1 Introduction

Services in GloServ are represented as instances of OWL classes. These instances can either be stored in the OWL ontology itself or stored in a database back-end. Instance retrieval using ontologies is an expensive procedure. Since there could be thousands of instances in one class, the ontology will become very large and classification will become very slow. Thus, GloServ creates a mapping of ontology classes to database tables so that instances can be stored in a database back-end and instances are retrieved quickly. GloServ implements both ontology and database instance retrieval methods. This chapter discusses how instances are registered and queried for using both models.

7.2 Ontology Model

7.2.1 Registering Instances

Chapter 5 discussed how a registration message is processed and routed through the GloServ network. Once the registration processor determines the nodes the registration belongs to, it instantiates the registration in those nodes.

For an ontology instantiation, the processor converts the query to an instance of the main service class within a service classification ontology. Continuing with the examples given in the previous chapters, the CAN is distributed by cuisine and the main class is restricted by location. Thus, the service classification ontologies in the CAN GloServers have the main class assigned to *NYCRestaurant*. When the instance registration message

(*hasNeighborhood* **some** `NYCNeighborhood`) and (hasCuisine **some** `Chinese`) or (*hasCuisine* **some** `Korean`) and (*hasName* **has** `"EastMeetsWest"`) and (*isOpen24hours* **has** `false`) and (*hasDelivery* **has** `true`) and (*hasStreetAddress* has `"875 3rd Ave"`) and (*hasLongitude* **has** `"-73.969471"`) and (*hasLatitude* **has** `"40.757602"`)

reaches a super node, a registration processor classifies it under the *ChineseNYCRestaurant* and *KoreanNYCRestaurant* classes. It then creates an instance in one of the classes and assigns all the property values as well as a unique ID and instance name to distinguish it from other service instances. The ID is a number and the instance name is a URN. The URN represents the location of the instance in the classification hierarchy. In this case it is *urn:gloserv.Service.Restaurant.NYCRestaurant.1*.

If the instance is created in the *ChineseNYCRestaurant* class, this class becomes an *asserted type* of that instance, meaning the class actually holds this instance and the instance belongs to that class. OWL allows an instance to belong to more than one class by assigning matching classes to a single instance. Since the registration message belongs to the *KoreanNYCRestaurant* class as well, the registration processor also adds the *KoreanNYCRestaurant* class as an asserted type to the instance. The processor creates instances and assigns its values using the Protege-OWL API, which provides basic OWL functionality such as creating classes, properties and instances. As described in the ontology example in Chapter 2, the instance looks like the ontology shown in Figure 7.1.

```
<ChineseNYCNeighborhoodRestaurant rdf:ID=
  "urn:gloserv/Restaurant/NYCRestaurant/1">
  <rdf:type rdf:resource="#KoreanNYCrestaurant"/>
  <hasDelivery rdf:datatype="&xsd;boolean">true</hasDelivery>
  <hasName rdf:datatype="&xsd;string">"EastMeetsWest"</hasName
  <hasStreetAddress rdf:datatype="&xsd;string">875 3rd Ave
  </hasStreetAddress>
  <hasLongitude rdf:datatype="&xsd;float">-73.969475</hasLongitude
  <hasLatitude rdf:datatype="&xsd;float">40.757603</hasLatitude>
  <isOpen24hours rdf:datatype="&xsd;boolean">false</isOpen24hour
  <hasCuisine rdf:resource="#ChineseCuisine"/>
  <hasCuisine rdf:resource="#KoreanCuisine"/>
  <hasNeighborhood rdf:resource="#NYCNeighborhood"/>
</ChineseNYCNeighborhoodRestaurant>
```

Figure 7.1: Service Ontology Instance

7.2.2 Querying Instances

The query processor creates a *query* class and sets restrictions on it which match that of the user query. For the query

($hasNeighborhood$ **some** `NYCNeighborhood`) and (hasCuisine **some** `Chinese`) or ($hasCuisine$ **some** `Korean`) and ($hasDelivery$ **has** `true`) and ($isOpen24hours$ **has** `true`),

the query processor creates a *query* class equivalent to the one given in the ontology and runs the reasoner to obtain all *inferred* instances of that class. An inferred instance is an instance which has been explicitly instantiated in a certain class, say *X*, but because its attributes match class *Y*'s property restrictions, the reasoner logically concludes or *infers* that it can belong within *X* as well, given that classes *X* and *Y* are related somehow (e.g., equivalent, parent-child, sibling). These instances will have matching values for the $hasNeighborhood$, $hasCuisine$, $hasDelivery$ and $isOpen24hours$ properties. The processor then returns the set of instances to the user which include all the property values of each instance.

7.3 Database Model

7.3.1 Ontology to Database Mapping

The ontology-to-database mapping occurs when the CAN is created during the network generation. Each GloServer holds a service classification ontology which has a main service class. The main service class is represented by a table. Each row in the table is a service instance. Every instance has a unique ID and instance name, which is a URN representing the hierarchical classification of the instance. Also, since an instance can belong to a number of classes, this is also represented in the database by assigning a child table for the class column and allowing an instance to have a number of classes assigned to it.

OWL allows classes to set cardinality restrictions on a given property. Thus, properties are either represented as columns within the service class table or their own tables depending on their cardinality restrictions. We define these two properties as: *Class-A* properties and *Class-B* properties.

Class-A properties are all properties of the service class that have a cardinality restriction of exactly one. This means that an instance can only have one value assigned to this property. Class-B properties are those properties which are **NOT Class-A. Thus, these properties have cardinality greater than 1 or no cardinality restriction at all.**

Class-A properties are represented as columns in the service class table. Class-B properties are represented in their own table. This table is comprised of two columns: the instance ID (foreign key to the service class table) and the property value. This way it is possible to assign no value, one value or any number of values to a single property of a single service instance.

In the running example, the main class of a CAN GloServer is the *NYCRestaurant* service class. It has several Class-A properties like instanceName, *isOpen24hours*, hasDelivery or hasFreeSeats. They are all Class-A because it does not make sense to assign more than one value to these properties (e.g., *hasDelivery* can not be both yes and no). The Class-B properties could be hasCuisine or hasClassName. The hasCuisine property is expressed in the registration message since a restaurant can offer more than one cuisine such as *Chinese* and *Korean*. The hasClassName property is added in the database to indicate which classes this instance belongs; in this case it belongs to classes *ChineseNYCRestaurant* and *KoreanNYCRestaurant*. An entity relationship model of a Class-B property table is shown in Figure 7.2.

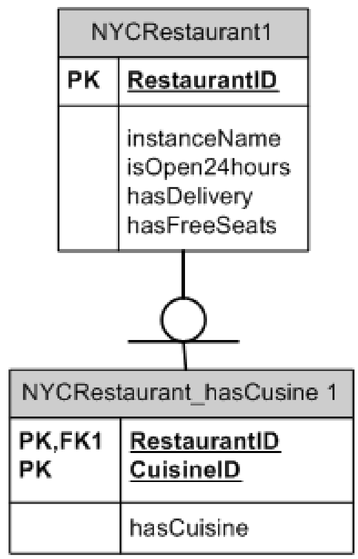

Figure 7.2: Entity relationship notation

7.3.2 Registering Instances

A service is registered by inserting a row in the service class table. Class-A properties in the registration message are just inserted into the main table. Class-B properties are inserted in child tables. In this example, for the following registration message

(*hasNeighborhood* **some** `NYCNeighborhood`) and (hasCuisine **some** `Chinese`) or (*hasCuisine* **some** `Korean`) and (*hasName* **has** `"EastMeetsWest"`) and (*isOpen24hours* **has** `false`) and (*hasDelivery* **has** `true`) and (*hasStreetAddress* has "`875 3rd Ave`") and (*hasLongitude* **has** "`-73.969471`") and (*hasLatitude* **has** "`40.757602`")

the Class-B properties are hasCuisine and the additional hasClassName.

Initially, the registration processor inserts the Class-A property values and then iterates through all the Class-B properties and inserts these into their corresponding child tables. As in the ontology registration case, a unique ID and instance name is assigned to the instance. The corresponding SQL query represents the service registration

```
INSERT INTO NYCRestaurant SET instanceName =
'urn:gloserv/Restaurant/NYCRestaurant/1', hasNeighborhood
='NYCNeighborhood', hasName = 'EastMeetsWest', isOpen24hours =
'false', hasDelivery = 'yes' , hasLongitude = '-73.969471',
hasStreetAddress = '875 3rd Ave' , hasLatitude = '40.757602'

REPLACE INTO NYCRestaurant_hasCuisine SET NYCRestaurantID = '1',
hasCuisine = 'Chinese'

REPLACE INTO NYCRestaurant_hasCuisine SET NYCRestaurantID = '1',
hasCuisine = 'Korean'

REPLACE INTO NYCRestaurant_hasClassName SET NYCRestaurantID = '1',
hasClassName = 'ChineseNYCRestaurant'

REPLACE INTO NYCRestaurant_hasClassName SET NYCRestaurantID = '1',
hasClassName = 'KoreanNYCRestaurant'
```

7.3.3 Querying Instances

The following query only contains Class-A properties.

(*hasNeighborhood* **some** `NYCNeighborhood`) and *hasDelivery* **has** `true` and *isOpen24hours* **has** `true`

Thus, the SQL statement covers only the main service class table and none of the separate property tables

```
SELECT * FROM NYCRestaurant WHERE isOpen24hours = 'true' AND
hasDelivery = 'true'
```

In a query where Class-B properties exist in addition to Class-A properties, the Class-B properties are joined in the SQL statement. For the following query:

(*hasNeighborhood* **some** `NYCNeighborhood`) and (hasCuisine **some** `Chinese`) or (*hasCuisine* **some** `Korean`) and *hasDelivery* **has** `true` and *isOpen24hours* **has** `true`

the SQL statement is:

```
SELECT * FROM NYCRestaurant as t1, NYCRestaurant_hasCuisine as t2
WHERE t1.isOpen24hours = 'true' AND t2.hasCuisine IN ('Chinese',
'Korean') AND t1.ID = t2.NYCRestaurantID
```

Once the database has been queried, the result set needs to be converted back to an ontology form. For example, when a property has two values, the joining will create two rows in the result set covering the same service instance. All values will be duplicated except for the property that has multiple values. However, from the ontology perspective there is only one instance with just two values for a property. The results are returned to the front-end in the form of an ontology.

7.4 Implementation

GloServ uses the MySQL database server for the database instance processing. The classes defined for service instance processing are:

- The *DBUtil* class contains the methods for creating the tables during the ontology to database mapping. It also contains utility methods for connecting to the database and executing queries.
- The *IInstanceProcessor* interface contains the methods *queryInstances* and *registerInstances* which queries and writes to the database.
- The *RacerInstanceProcessor* and *DBInstanceProcessor* implement *IInstanceProcessor* and perform the ontology and database instance processing respectively.

7.5 Conclusion

This chapter has described how service instances in GloServ are stored within the CAN GloServers. GloServ performs an ontology-to-database mapping to store query instances in a back-end database. It implements both an ontology model and a database model for storing instances, but uses the database model because ontology processing becomes expensive as ontologies grow large. This is evaluated in the next chapter.

Chapter 8

Evaluation

8.1 Introduction

The GloServ architecture has two main contributions, namely, a scalable network architecture for distributing services and the ability to perform intelligent querying of services. These are attributed to the underlying peer-to-peer CAN architecture and the use of description logic ontologies for classifying services. Because of this, the evaluation of GloServ depends on both the architectural component and the ontology component.

The architecture should be evaluated using three main criteria: the time it takes for the network to generate given an ontology, the query latency and the amount of load one GloServer takes in terms of service registrations, updates and queries. Since there are no other comparable architectures to GloServ and the main point of difference in GloServ, compared to other service discovery systems, is its network architecture and use of ontologies, I evaluated the performance of GloServ within the CAN networks and used the results obtained in the original CAN paper to calculate the overall query latency of the system. This helps in evaluating the benefits and drawbacks of using ontologies for query processing.

Additionally, the type of ontology used affects the results of the evaluation. The main drawback of ontologies is that classification is expensive. As ontologies grow large, reasoning becomes a bottleneck. However, the number of classes for a given service ontology can be anticipated to be a few hundred. For example, for location-based services such as restaurants or theaters, the services may be distributed by location and then by cuisine, or movie genre. Given that there are around 600 cities [3] with population greater than 50,000 in the United States and around 1300 urban areas in the world [10], even if all the locations are put into one ontology, the number of classes will still be manageable. However, this is highly unlikely given that services are distributed in GloServ and each subnetwork handles a subset of a high-level service class. Thus, the number of restricted subclasses of the main class reduces to a few hundred within a given service classification ontology. GloServ also tackles the problem of ontology size by storing instances in a database back-end instead of the ontology itself and using only class relationships for determining which classes a query belongs in. This speeds up the classification process considerably.

This chapter describes the evaluation of GloServ. In Sections 8.2 and 8.3 I describe how the system is set up and formalize notation. Section 8.4 evaluates two common description logic reasoners to determine which reasoner is best used in GloServ. The query latency, CAN network generation, and the server load within a CAN are evaluated in Sections 8.5, 8.6 and 8.8 respectively and compared to the bare bones CAN network described in [110].

8.2 Setup of GloServ

8.2.1 Ontology

In order to use a real-world example, the GloServ evaluation uses a *Restaurant* ontology based on the classification provided on the menupages.com website for evaluation purposes. Since the number of locations are limited in the classification, the size of the ontology is not very large. Thus, I modified the ontology to increase the number of classes and the number of restrictions per class to see how well it performs in the average and worst case scenarios.

8.2.2 System

The GloServ library contains approximately 13,500 lines of code in Java. It also uses the Protege-OWL API as well as a description logic reasoner. The evaluation tests were run on an IBM Lenovo notebook which has an Intel core duo processor (2 GHz each), 1 GB RAM running Windows XP.

8.3 Ontology and Query Notation

8.3.1 Ontology Type

The notation for the ontology type is O_c_r where c is the number of classes and r is the number of property restrictions for each subclass of the main class. If the ontology is labeled O_c_5, for example, this means the number of classes (c) is varied and number of restrictions (r) are kept to a constant of 5.

8.3.2 Query Type

Below, the query notation is expressed in BNF notation as:

```
SOME_EXPR = property, "some", value


AND_EXPR_n = SOME_EXPR, {and, SOME_EXPR}
```

where n signifies the number of $SOME_EXPR$s joined together in a conjunction. For example, AND_EXPR_3 evaluates to:

$SOME\_EXPR$ and $SOME\_EXPR$ and $SOME\_EXPR$

8.4 Description Logic Reasoner Type

The first testing was done to select the best reasoner for GloServ. Benchmarking tests [82] have been performed on three currently available description logic reasoners, RacerPro [90], FaCT++ [121] and Pellet [119], using the top 10 most difficult ontologies as determined in [82]. The benchmark tests involved finding how long it takes for these ontologies to be classified. Classification involves finding all class relationships within an ontology by forming equivalent, superclass and subclass relationships between them. Each ontology has a different set of constraints on the number of classes and individuals. The closest to our model is a description logic ontology which has around 1500 classes and 0 instances. Such an ontology is classified in about 1 second with the RacerPro reasoning engine. It also shows that RacerPro performs the best for this type of ontology. In order to verify these numbers, I tested the different ontologies with Racer and Pellet to determine which reasoner to use for the GloServ evaluations. Both of these have Java APIs which fit nicely with the GloServ API. I tested a simple ontology classification. Classification involves finding equivalent, parent and subclass relationships for *every* class in the ontology. The initial classification is much slower than subsequent classifications because there are no cached structures in the ontology reasoners. However, after the first classification, ontology reasoners cache the relationship structures and thus subsequent runs of the reasoner are much faster. Thus, before querying an ontology, it should be classified once.

8.4.1 Benchmark Results

The results in Figure 8.1 show the first two classification times for each reasoner. RacerPro performs quite well for both. On the lower end, an ontology with 250 classes takes around 3 seconds to classify and on the high end a 3000-class ontology takes a little over a minute. The classification time improves the second time the reasoner classifies the ontologies: the 250-class ontology takes 1 second and the 3000-class ontology takes 45 seconds. On

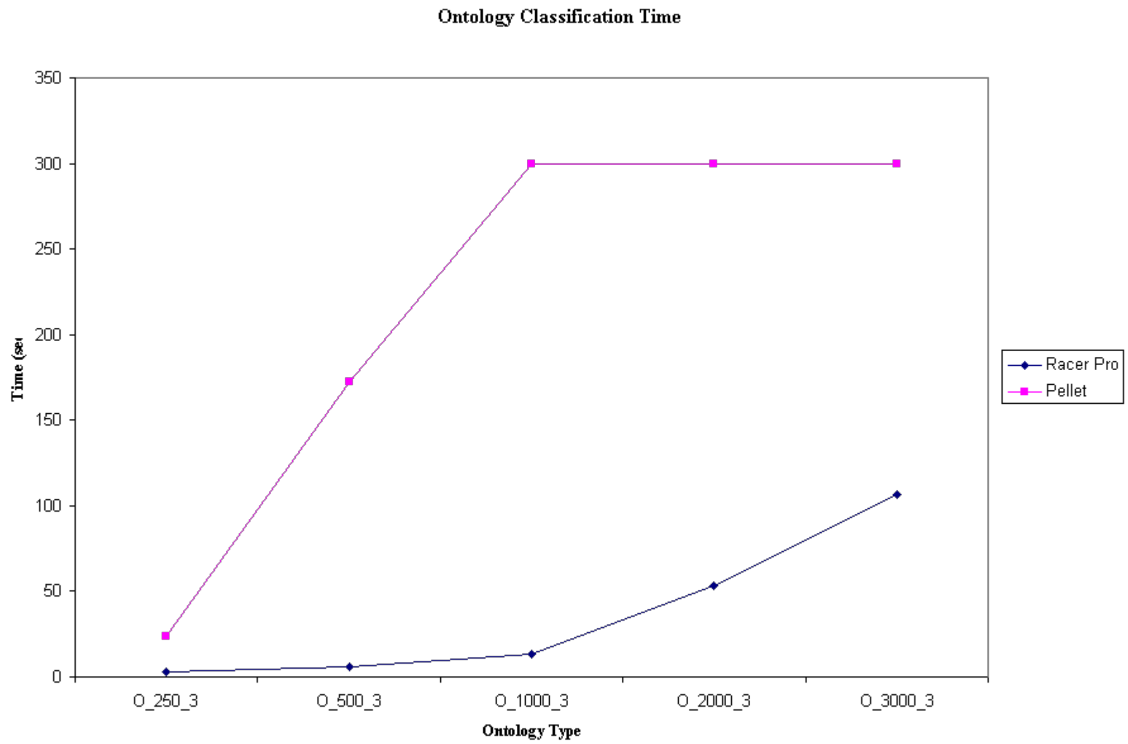

Figure 8.1: Ontology classification time using RacerPro and Pellet reasoners

average, the ontology size will be around 500 classes and this takes around 5 seconds for the first run and 2 seconds for the second run.

Pellet, however, performs poorly as the ontology grows. For the first run, a 250-class ontology takes 24 seconds and a 3000-class ontology hangs as it goes over the threshold of 5 minutes. On the second run, the 250-class ontology performs dramatically better, classifying it in less than a second. However, it is difficult to tell what the classification time of the second run is for the large ontologies since the first run was never processed completely. As mentioned earlier, the average-sized ontology is 500 classes for location-based services and this takes around 2 seconds to process.

8.4.2 Conclusion

These results show that RacerPro outperforms Pellet as the number of classes in the ontology grows. On the other hand, preliminary results for Pellet show that it has powerful caching algorithms and improves the overall classification time dramatically after the first run. However, since RacerPro is a fully-deployed description logic reasoner which also has strong caching algorithms [90] and can handle large ontologies with greater ease, it is the best choice as a description logic reasoner for GloServ.

8.5 Query Classification

Before a GloServer uses an ontology, it first classifies the complete ontology once. This causes the reasoner to establish all relationships between the ontology classes and allows the reasoner to cache these structures so that subsequent query classifications will occur faster. Query classification involves classifying a single query within an already classified ontology. This process is much faster than classifying the complete ontology where every class is classified. A query is classified either as a parent class, an equivalent class or subclass of the ontology. Each of these are regarded as matches. Because of RacerPro's caching mechanisms, this works very well with previously seen queries. However, all these tests are done with unique queries first so that the difference in performance can be analyzed.

8.5.1 Ontology Type

Since GloServ relies heavily on ontologies for network generation, registration and querying, I tested it with a number of different types of ontologies. Query classification time depends on the number of classes as well as the number of property restrictions per class. With more classes and restrictions, the classification time increases dramatically as ontology reasoning has an exponential runtime [55]. GloServ is evaluated by varying these two aspects of an ontology. Ontologies with 250, 500, 1000, 2000 and 4000 classes were used.

As mentioned earlier, the average number of restricted subclasses of the *main* class will be approximately 500 with another 500 imported classes which are used for the property restrictions. For example, a *Restaurant* class has object properties which point to imported classes, such as *Neighborhood* and *Cuisine*. These classes are also included within the overall ontology. If a restaurant is separated by location, it will have the same number of (restricted) subclasses as the *Neighborhood* subclasses, which doubles the ontology size.

However, it should be noted that these *refiner* ontologies are not needed in full for every CAN subnetwork. For example, in the initial *Restaurant* CAN, each class is restricted by neighborhood, mapping to another CAN; for each of these CAN networks, say the *NYCRestaurant* CAN, the location is already restricted to NYC and thus the *Neighborhood* ontology may have the *NYC* subclass as well as subclasses of *NYC*, such as *UpperEastSide* or *UpperWestSide*, but does not need the full *Neighborhood* ontology from the higher level network which consists of other locations. However, for the purpose of testing GloServ's performance, I use larger ontologies which have a maximum of 4000 classes.

For each of these ontologies, the number of property restrictions within the main class are either 1, 3 or 5. For example, if the main class of the *Restaurant* ontology is the class *Restaurant*, then one property restriction means that each of the subclasses of *Restaurant* are restricted by one property such as the `hasLocation` or `hasCuisine`. For an ontology that has three property restrictions, the subclasses are restricted by three properties such as `hasLocation`, `hasCuisine` and `hasPriceRange`. The average number of restrictions per class will probably be no more than three. This is again due to the fact that the ontologies are distributed and thus for one CAN level, the restrictions are not going to be that many since at each level a few properties are restricted. However, for the purpose of testing the limitations of the system, I increase the restrictions per class to five.

The first test was to determine if the number of classes in an ontology affects the query classification time. The ontology type used was O_c_3 where the number of classes were varied and the number of restrictions kept to a constant of 3. For each ontology type, a unique *AND_EXPR_5* query was issued which matched around five classes. The results are shown in Figure 8.2. As expected, as the number of ontology classes grew from 250 to 4,000, the query classification increased exponentially. However, the results are encouraging for an ontology size of 1,000 to 2,000 classes. The classification performance is around 30 ms for both. It should be noted that these results pertain to only unique queries. If this test is run a second time, the query classification time drops drastically. Query classification for ontologies with less than 2,000 classes takes around 7 ms and a 4,000 class ontology 30 ms due to Racer's caching mechanisms. Additionally, GloServers have their own cache. Ontology queries are mapped to service classes in a hash table. If a given query is not unique or new, it is not processed through the reasoner and thus query processing time drops to 1 or 2 ms.

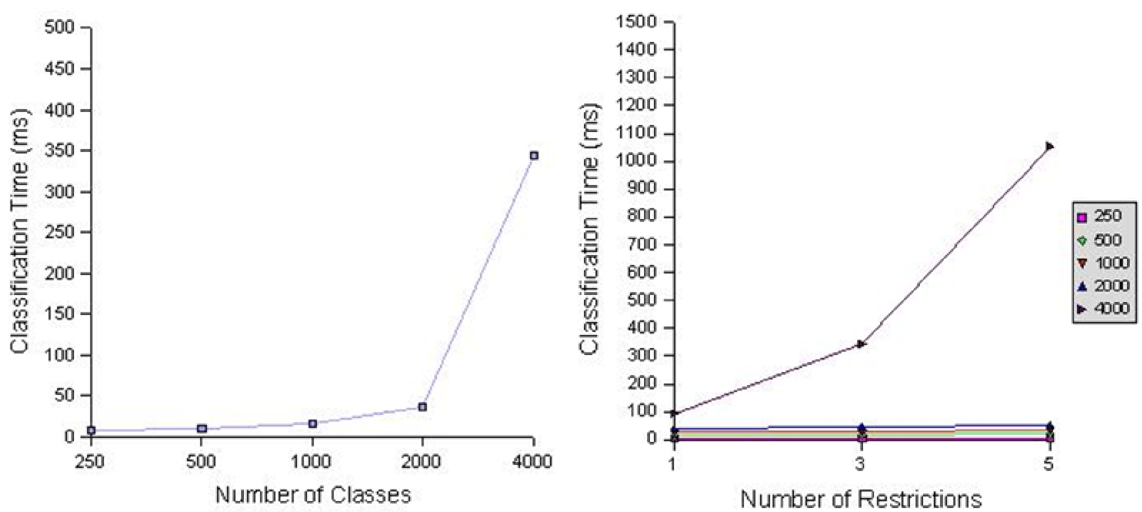

Figure 8.2: Query classification time with varying number of ontology classes and restrictions

In order to see if the number of restrictions per class affects the query classification time as well, the number of classes should be kept constant while increasing the number of restrictions per class. Ontologies ranging from 1 to 5 restrictions per class were tested on 250-class to 4000-class ontologies. The graph shows that for ontologies with classes ranging from 250 to 2000 classes, query classification remains below 50 ms. However, once the number of classes increases to 4000 classes, the query classification time increases to a second.

As mentioned earlier, the number of ontologies used in a given CAN is not expected to exceed a thousand classes averaging 3 to 5 restrictions per class. Thus, the results are promising for the ontology type O_1000_n and this ontology type is the one used for the remaining evaluation of GloServ.

8.5.2 Matched Classes

Besides trying to determine the optimal ontology to use for GloServ by testing it with different ontologies, the number of classes a query is matched to may affect the classification time as well. The ontology to network mapping is structured in such a way that a generic query generally does not match more than a few tens of classes. For example, in the *NYCRestaurant* ontology, classes are separated by location and cuisine. Thus, when a user searches for a restaurant with a specific cuisine it will get only one or a few class matches. However, if the user does not put in a cuisine restriction, then every subclass is matched to the query. This is expected to be around 50 classes since there are around that many cuisines defined in the ontology. Since the whole point in using the CAN is to distribute service data and let each node handle a set of similar service classes, as ontologies are distributed in the CAN, they become more and more specific and hence the number of classes matched becomes less and less.

In order to determine query classification time with respect to number of matches, I set a simple $SOME\_EXPR$ query that matched 50, 100, 200 and 500 of the restricted subclasses of the main class in a O_1000_3 ontology. The results once again show exponential growth in query classifications as the number of classes matched increases. If the query matches all 500 subclasses, it takes a little over a second to classify. However, for 50 matches, the classification time takes around 15 ms and for the expected 5 matches, the time is 2 ms.

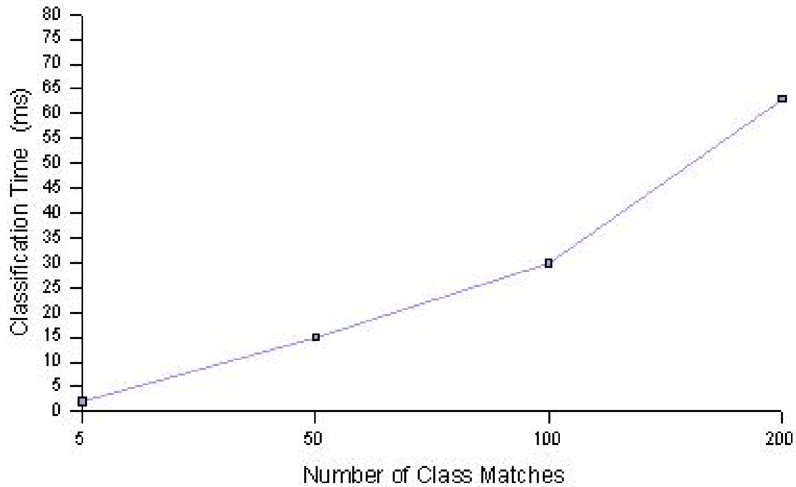

Figure 8.3: Query classification time with varying number of class matches

8.5.3 Conclusion

These tests conclude that ontology query classification time is not much of a bottleneck for GloServ because of its distributed design. Ontology size remains manageable as well as the number of matched classes. This gives the benefits of using ontologies as well as reasonable query latency.

8.6 CAN Network Generation

We tested the CAN network generation both on multiple hosts as well as a single host. The results were similar for both. For an O_1000_3 ontology, it takes around 5 seconds for a node to start up and join the CAN network. The bulk of the time spent during a node insertion is in loading the OWL file into the reasoner and classifying it before it starts. This takes approximately 3 seconds. Every node that wants to join the network downloads this file, stores it in the reasoner and classifies it. The remaining time is taken up by the steps shown in Figure 4.12.

Since each CAN handles a specific service class and the ontology size is manageable, the number of nodes will scale on the order of hundreds or thousands at most. For example, given the *Restaurant* example, on the first level, restaurants are distributed by location. On the second level, they are distributed by cuisine. However, on average, we expect there to be at most 500 nodes per CAN because each CAN handles a subset of the service class distributed by certain properties such as location.

8.7 Query Latency

Since we determined that the query processing time for an O_1000_3 ontology query is approximately 30 ms, the query latency can now be estimated for the complete hierarchical peer-to-peer network to

$$q_h h + \sum_{i=1}^{c}(q_i + l_i)$$

where q_h is query processing time for a hierarchical node, h is number of hierarchical nodes a query traverses, q_i is query processing time for a CAN supernode, c is number of CAN overlay networks traversed, l_i is lookup latency for a single CAN network.

8.7.1 New Queries

As explained in Chapter 4, h is a constant equaling 2 as long as every hierarchical node holds a snapshot of the ontology. This is because a query comes into one of the hierarchical nodes, is classified and matched to a CAN supernode. The primitive skeleton ontology is expected to be around a few hundred classes and will also have a few restrictions. Thus, we approximate the query processing time to be around 30 ms as shown in the O_1000_3 ontology.

Each CAN network will process an ontology query once within its supernode, hash it to $\langle dimension, value\rangle$ keys and route the query. The query processing time, q_i, for every

CAN subnetwork, i, will decrease as it traverses down the CAN overlay networks because each network will have a smaller ontology than its parent network. Thus, the ontology classification time will be at most 30 ms for any given CAN which holds an O_1000_3 ontology.

Because each CAN is limited to a single service class, such as *Restaurant* each service class will probably have a few subclass levels because the assumption is that a given service class will have the number of properties in the order of tens. Thus when classifying the main class into subclasses, each subclass level will restrict a few properties which limits the subclass nesting to a few levels. Let us say there are 3 subclass levels, this means that if CANs are created for each subclass level, on average, there will be around 3 CAN overlay networks for a give service class. Also, lookup latency, l_i, within a single CAN has been evaluated in [110]. The number of nodes are varied from 256 to 256,000 and values range from 0.5 ms to 16 s, assuming that each node is run on a separate machine. Since the CAN networks in GloServ will contain at most 500 nodes, due to the ontology distribution, the lookup latency of each CAN is 1 second.

I assume the worst case where every CAN overlay network has the same number of nodes, the same query processing time for each supernode (q_i) and the same latency for each CAN (l_i) which means the overall query latency evaluates to $q_h h + c(q + l)$, where q and l are the query processing time and latency, respectively.

Given this case, the following values hold for our example, 3 CAN overlay networks for a given service class with 500 nodes in each CAN and an ontology type of O_1000_3; hierarchical nodes that contain snapshots of the full primitive skeleton ontology which means that queries traverse 2 hierarchical nodes. The variables are set to the following values:

q_h = 30 ms, $h = 2$, q = 30 ms, $c = 3$, l = 1000 ms.

The full query latency evaluates to 3.2 seconds.

8.7.2 Cached Queries

As expected, GloServ performs much faster for queries that are cached aside from description logic reasoner caching. The queries are stored in hash tables and mapped to corresponding classes. If a query is found in the hash table, the description logic reasoner is not used to process the query. Thus, query processing time is around 1 or 2 ms. Substituting 2 ms for the variables q_h and q_i and leaving the rest of the variables the same for the equation above, gives an overall latency evaluates to 3 seconds. This shows that the bulk of the time of the overall latency is due to the CAN routing.

8.7.3 Queries in Optimized CAN

As mentioned in Chapter 4, supernodes cache not only the classes that the query maps to, but also the IP address of the GloServer handling that class. This speeds up the query routing to $O(1)$ hop. Thus, for cached queries which also contain host information, the latency reduces to the query processing time of one node and the CAN latency is not included in the calculation. With this scenario, the GloServ query latency becomes approximately 25 ms.

8.8 Server Load

The RacerPro reasoner uses most of the resources in a GloServer. However, the reasoners reside only within the supernodes which process the initial ontology query and convert it to an internal query. The subsequent GloServers in a CAN either route the message to a peer if the message's $\langle dimension, value \rangle$ key does not match or store it in their database if it is a match. Thus, the load on the system for the internal messages is negligible. In order to analyze how many unique ontology registration and query messages a given supernode can handle, the query classification had to be looked at once again.

The Racer Query Language (RQL) [90] is a querying language that was recently developed for RacerPro that allows concurrent queries to be processed at once whereas issuing queries in the regular Racer syntax causes the reasoner to block until each query is processed. Thus, I issued concurrent RQL queries, using the same matching criteria of super class, equivalent class and subclass, to Racer which held the average O_1000_3 ontology. The threshold time was capped to 1 minute and the results showed that 10,000 unique queries could be issued concurrently per minute. The other nodes which handle internal query and registration messages are used for routing or storing information. These nodes processed requests in a negligible amount of time of at most 1 or 2 ms. Thus, I use the supernode load to evaluate the number of servers necessary in a given CAN network. It must be noted that this pertains only to new queries which require reasoning. Otherwise, queries are cached locally and mapped to the service classes and do not need the reasoner which removes the extra load.

Let us look at a concrete example. One of the main motivations to use a P2P DHT instead of a network with replicated servers in GloServ is for dynamic service updates. In a replicated network, every node must process a service update which increases the load of the system. In P2P networks, only those nodes which handle that specific service will be updated, thus reducing the overall load of the system. Consider the example given in the introduction where people can be represented as services. Imagine an application that queries and records the longitude/latitude location of people at given time intervals and uses this information in a context-aware computing application. New York City alone has 8,000,000 inhabitants. Assume that this application issues queries and updates for the location of 1 million of these people every minute and these requests are routed evenly throughout the CAN (no ”hotspots”). This results in twice the number of requests, 2 million requests a minute. Given that a supernode handles 10,000 requests a minute, the CAN requires at least 200 servers for this service class in New York City alone. Let us set this to a 250-node CAN network.

Now, let us look at how many of these nodes must update their information for a give service request. If this 250-node CAN uses the O_1000_3 ontology, half of these classes will be auxiliary classes and half will be part of the main service class, as described above. Thus, it will have 500 restricted subclasses for the main class. It separates these classes into four dimensions where each dimension handles 125 classes. For a 250-node CAN, each node handles around 2 classes and each dimension has around $n^{1/d}$ nodes where n is the number of nodes and d is the number of dimensions. Thus, in a 250-CAN network separated into 4 dimensions, there are $250^{1/4}$ nodes per dimension. With this architecture, certain classes of people can be queried for and updated at a given time. For example, a 256-node CAN separated into 4 dimensions needs approximately $256^{1/4}$ or 4 hops. A service update will first query for a node in one dimension and then update nodes in the other dimensions. This evaluates to $d(n^{1/d})$. Thus, for a 250-CAN network, the number of nodes that need updating is approximately 16 nodes. If this was a replicated network that did not distribute information by domain, every server would have to be updated for that service domain. This is quite an improvement to a replicated network of 250 nodes. Even if the nodes were replicated in such a way where database tables were distributed in a network, the advantage of GloServ's automatic network generation from ontology information does not exist in traditional replicated networks which. Thus, GloServ allows automated creation and distribution of an overlay network and service information.

It must be noted that although CAN distributes load evenly, it still suffers from the ”hotspot” problem where many queries are routed to the same node in a short period of time. There are a number of ways to handle this as described in [110]. One obvious way is to replicate hotspot nodes and then synchronize the data between them. As future work, I would like to take a look at how ontologies can be used for network management purposes in order to alleviate these problems.

8.9 Conclusion

This chapter has evaluated the various components of the GloServ back-end service discovery system. The evaluation demonstrates that GloServ functions quite well under the following conditions: a service ontology that has a few thousand classes and several property restrictions per class; a CAN network with several hundred nodes per service class. Since the GloServ architecture is distributed, these conditions are upheld. Additionally, if the caching and routing optimizations are implemented for queries, the latency becomes quite low. The calculations above also show that the underlying peer-to-peer architecture of GloServ allows frequent updates of services to be performed very efficiently. This allows many different types of context-aware service discovery applications to interact with GloServ.

Part III

GloServ Query Extensions

Chapter 9

Combining Multiple Service Queries in a Single Search

9.1 Introduction

This chapter describes an enhancement to GloServ which allows different types of services to be queried for in a single search, in a *combined query*, is described. In general, service composition is done mainly within one service class such as travel or restaurant. For example, travel sites such as expedia.com and priceline.com allow one to search for a combination of travel services such as tickets, hotels and car rentals. The menupages.com site gives information on restaurants in certain major cities in the United States which include menus, price ranges and ratings for each restaurant. Restaurants can be searched for either by location, cuisine or text. Additionally, the seamlessweb.com site allows one to place an order at a restaurant for delivery. When searching for movies on sites such as movies.com or fandango.com, a list of movies in a certain location along with ratings and a link to purchase tickets are given. Thus, when a user wants to search for a restaurant and a nearby movie theater, she will need to first go to the restaurant site and find the restaurant. Then, she would browse the movie site and search for the theater near the restaurant's location that is playing a movie that interests her. To address the shortcomings, we have extended GloServ to allow multiple service queries in a single search.

9.2 Motivation

The main limitation of current web services, as defined in the previous chapter, is that they exist as separate entities on the web because data is not easily shared across different domains in the Internet. Every site has its databases modeled in a specific way. Semantically equivalent properties are defined differently. Because of this, querying for a combination of services over a shared property is not easy to accomplish.

9.2.1 Location-based Services

In the context of location-based services, location may be defined by region such as $Country-\rangle State-\rangle City-\rangle StreetAddress$, a zipcode, geographical coordinates, or a combination of all three. Given this predicament, we need a way to combine these equivalent properties so that when one wants to search for a service given geographical coordinates of longitude/latitude values, it can also find services which have its locations defined by region.

An ontology is useful in mapping semantically equivalent concepts to one another. Thus, it can define the relationships between the different location classes. A *Location* ontology can be defined and shared across all the different domains. Even though location may not be defined uniformly in each service class, interoperability is still possible because the ontologies map the different meanings into common classes.

For example, the *Location* ontology defines the *Region* and *ZipCode* classes. Each region includes hasZipCode, hasLongitude and hasLatitude properties. The class *UpperWestSide* is classified in the $USA-\rangle NewYork-\rangle NewYorkCity-\rangle UpperWestSide$ hierarchy and has the restriction:

hasZipCode **some** ('10023' or '10024' or '10025).

A location instance is classified under *UpperWestSide* if it has one of the above zip codes. The *ZipCode* class can have a set of restricted subclasses which define the perimeter of each zipcode boundary by its geographical coordinates. As a result, the location ontology defines regions in terms of zipcodes and zipcodes in terms of geographical coordinates. Hence, attributes that are defined by either region, zipcode or geographical coordinates, can be easily converted to the other.

If the location property is shared across different location-based service classes, combined queries can be performed over the different services. An example would be searching for a Chinese restaurant in New York City which also has a nearby theater playing an action movie. The nearby theaters are those that are located in the same place that the restaurants are. The results will show all Chinese restaurants and their corresponding movie theaters.

9.2.2 Service Tagging

Another example of combining queries in a single search would be querying for a service and its tags. Currently, tagging is becoming a common phenomenon in various web services such as Flickr [15] and de.licio.us [9]. We define a *tag* as an attribute of a service that is not part of the service description itself. For example, many users like to read and give feedback pertaining to services they will use or have used. This review system can be deemed as a *Tag* service class which has any number of different rating services such as Zagat [44] for restaurants, Better Business Bureau for businesses, or regular user reviews. Given this type of service composition, one can search for any service which has a particular rating and give feedback for this service as well. A *Tag* service class is defined with its own service classification ontology and inserted into the GloServ network. In order to motivate this, let us look at the following scenarios:

- A user searches for a restaurant in the Upper East Side neighborhood of Manhattan that serves Italian cuisine, which has a certain *Zagat* rating. Zagat is a restaurant guide and is independent of the restaurants on the Upper East Side, but provides ratings by tagging restaurant services.

- A tourist searches for a sightseeing venue in Manhattan, but is not able to provide the current weather situation. Thus, he wants to find places that are rated good to visit today by any tagging provider.

- A user searches for all tags the Michelin Guide [25] has for the Morton's of Chicago steakhouse in Midtown Manhattan.

Tags can be categorized and described just like services using classes, subclasses and instances. Just as there is a *Restaurant* class, there also is a *Tag* class described in an ontology. This categorizes the class of tags in a coherent manner. It also means that there is no special architecture required in order to store tags and query for tags directly because GloServ is able to register, update and query for tags in the same way as it does these jobs for services. Thus, issuing the query, *all rating tags for restaurant X* becomes possible.

The scenarios listed above are combined service queries in GloServ where the shared property between these two services is the service instance of the primary query in question. For example, the restaurant instance URNs of the *Restaurant* query are matched with all the tag instances which have the same URN and satisfy the query condition. For the first scenario, all the Italian restaurants in the Upper East Side are found and the service URNs of these instances are substituted in the *Tag* query which then searches for all the restaurants with these tags which also has a certain Zagat rating.

With the above scenarios, multiple entities will provide tags for the same service and may even contradict each other. One might say *recommended* another might state *do not visit*. This allows the user to specify the author of the tags since they may tend to trust one author over another. For example, an avid diner trusts the New York Times [28] rating for restaurants over Zagat. However, in some situations the user does not know about authors yet and thus has no preference. She may want to see an overview of the tags provided for a special service.

The relationships between GloServ service classes and their corresponding tag services can be determined using an *ontology mapper*. An ontology mapper maps related properties to one another and determines equivalent terms in different ontologies. For example, tags from the *Zagat* and *NY Times* classes match the *Restaurant* service class for restaurant ratings and the *NY Times* class also matches the *Theater* class for movie ratings. *Flickr* and *Weather* tag classes match the *SightSeeing* service class because each of these can tag various travel sites such as travel brochures or tourist spots. The front-end interface establishes the relationship table of a given service class and its corresponding tagging service by using an ontology mapper.

Tags are aggregated in such a way where the user is able to see or search for specific authors of tags. As the tags are categorized and described just like services, the requirements for an architecture that supports discovery of tags are very similar to the ones identified for service discovery with GloServ. Tags are supplied and managed by tagging providers, who are independent from each other and manage their tags individually. An architecture supporting tag improved service discovery should honor the independent nature of tagging providers.

Figure 9.1 shows the relationships between *Tag* services and *Location-based Services*. Front-end web services handling sightseeing or restaurant services know about the tagging providers that contribute tags to their services via this relationship ontology. In most cases, these tags will be contributed by third party entities. Examples of tags include simple strings, photos or more complex data, like the weather at this service at the moment. Tag providers are either individuals or organizations such as Zagat which contribute tags on various services. Companies are envisioned, which either provide the annotations themselves or provide a nice interface to users (Flickr could allow linking user photos to GloServ services). The tag providers are envisioned to be independent of the services.

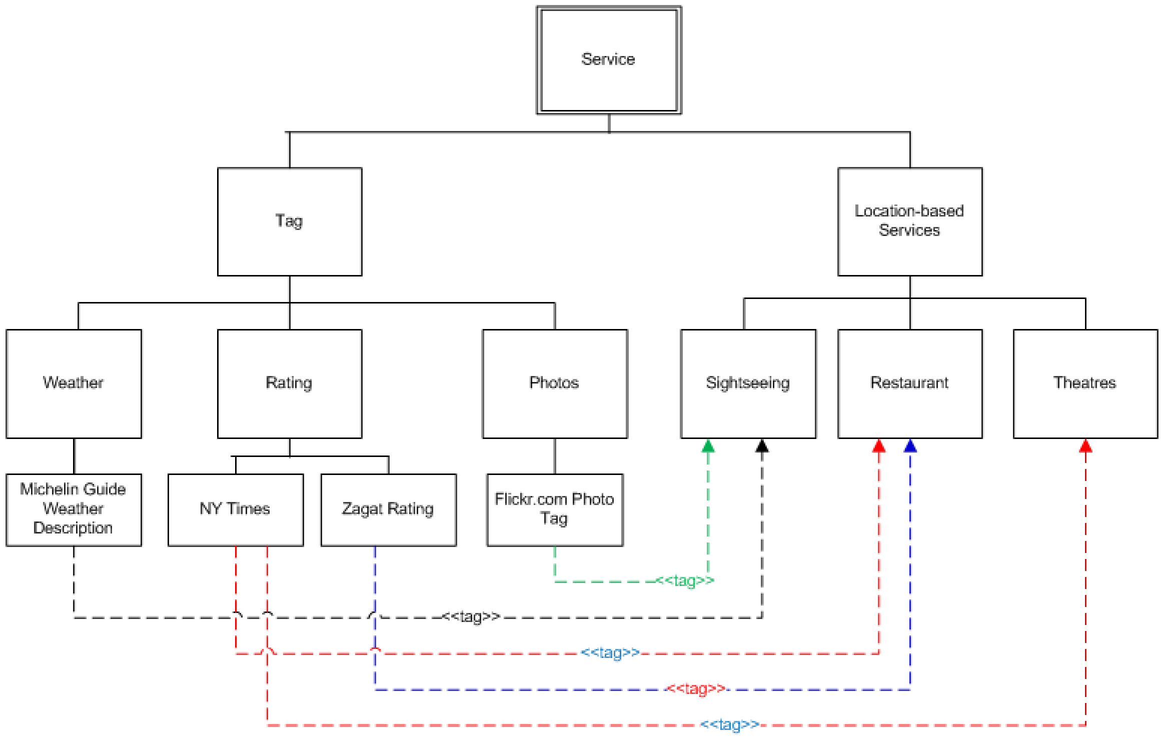

Figure 9.1: Relationship ontology of the tag class and the location-based services class

9.3 Problem of Combined Service Queries

We have enhanced GloServ to support subquerying between its servers in order to allow services that share common properties to be composed into a single query. With this querying extension, fairly complicated queries can be issued in a single search.

Since GloServ aggregates and distributes different classes of service data using ontologies and also provides quick access to this data, combining more than one service in a given search becomes possible. A few challenges exist in accomplishing this: creating a query language which allows CAN servers to distinguish different parts of the query, namely, the part belonging to its own service class and the parts belonging to other service classes; routing the query to the CAN GloServers handling the other service types; mapping shared properties to semantically equivalent values; efficiently joining the result sets of the different query parts; displaying the results to the user in an easy-to-understand graphical user interface, which is further described in Part IV: Front-End Systems. We describe the design of our solution below.

9.4 Query Language for Combined Service Queries

A combined query consists of three parts: *primary query*, *nested subquery* and *shared property* which we define below. The query is initially routed to the servers handling the *primary* query's service class. We have extended the basic ontology query language to allow this combination to be expressed.

The primary query part is a regular GloServ query. If the user is searching for an Chinese Restaurant in New York City, the primary query might be

(*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`).

Each *nested* query itself is a regular GloServ query, except that all property names are qualified with the service class name the sub-query is referring to. If the user is searching for a nearby theater playing an action movie, the nested query could look like:

Theatre.hasMovieGenre **some** `Action`.

Besides the primary query and the nested subquery, a shared property is required. To express this relationship we added the equals operator to the querying language. Following the given example, if the user is searching on a shared location property (i.e., query for a restaurant with a nearby movie theater), the equals expression would be:

Restaurant.hasLocation **equals** *Theatre.hasLocation* and the overall resulting GloServ query would be:

((*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`)) and (*Theatre.hasMovieGenre* **some** `Action`) and (*Restaurant.hasLocation* **equals** *Theatre.hasLocation*).

As mentioned earlier, one query might contain multiple subqueries. All of them use the same syntax and can be distinguished by the qualifier that is used as a prefix to the property names. The order in which the equals expressions describe relationships between the service classes determine in which order the sub-queries will be processed by GloServ.

9.5 Routing of Combined Service Queries

9.5.1 Hierarchical Routing

A combined service query is routed through the hierarchy similar to a single service search. However, instead of the query containing one service class name, it contains multiple service classes. When a user is searching for a restaurant and movie theater, she enters the words ”dinner” and ”movies”. These words are mapped to the *Restaurant* and *Theater* service classes and get routed through the hierarchical network to one of the CAN supernodes for these classes. The *Restaurant* and *Theater* CAN supernodes in turn send their ontologies to the user including their server IP addresses. The front-end caches this information so it can contact these servers directly if the ontologies need to be refreshed.

The front-end will then determine the shared properties by passing them through the ontology mapper which will return the shared properties of these two ontologies and formulate the query accordingly. The ontology mapper can be a separate plug-in to GloServ. This is discussed further in Section 9.6. Once the front-end determines the shared properties, it converts the ontologies to a form where only one field for the shared property is visible for multiple service classes. It may also use the ontologies directly to issue queries. The front-end creates a combined ontology query matching the user's request. The combined query consists of a *primary* query and a *nested* query. One of the service classes will be the primary query and the other will be nested. Thus, if one is searching for a restaurant with a nearby movie theater, the restaurant will be the primary query and the theater query will be nested. The reverse can also be supported, but one of the services will be assigned as the primary query.

9.5.2 CAN Routing

The combined query is routed directly to the GloServer handling the primary query's service class. The routing depends on how the front-end constructed the combined query. If the *Theater* query is nested in the *Restaurant* query, then the *Restaurant* CAN receives the combined query, otherwise the *Theater* CAN receives the query. For this example, we assume that the *Restaurant* CAN receives the combined query first.

Query Processing

The *Restaurant* server's query processor analyzes the incoming query in a fashion similar to how a user query is processed in terms of classification and routing. However, since the *queryType* header indicates that it is a combined query, the processor handles the query slightly differently. The query processor first strips away the primary query from the combined query and inserts the primary query in the query message's *primaryQuery* header. Then, it classifies the primary query matching it to the *NYCRestaurant* class and routes this query to the *NYCRestaurant* server. The query is routed down the CAN overlay networks in the usual manner until the query hits a node with instances. In our running example, if the *NYCRestaurant* class has a subnetwork that is separated by cuisine, then the query is routed to the *ChineseNYCRestaurant* server.

Once the query hits a node with instances, the query processor then analyzes the nested query and sends it to the *Theater* CAN. The query processor parses the nested query by removing the qualifier prefixes and transforming it into a valid primary query. It also resolves the equals expression by creating an expression for every found value of the shared property. This set of expressions is then "OR"-ed and attached to the new query.

Continuing with the example introduced above and assuming that Restaurants in *UpperWestSide* and *UpperEastSide* were found, the new query for Theater becomes:

(*hasMovieGenre* **some** `Action`) and ((*hasLocation* **some** `UpperWestSide`) or (*hasLocation* **some** `UpperEastSide`)).

As we can see, this new query is a regular GloServ user query without nested queries. The GloServers in the *Theater* CAN process the query as a user query. However, because the query type is a combined query, servers that contain instances do not send the instances directly to the user but process the instances further by joining it with the instances from the *Restaurant* CAN. This is described in detail in Section 9.7.

Query Routing

Since the primary query is just like any other GloServ query, it is routed and processed like a regular query with one exception. Instead of routing the query result with matching instances directly to the user, the nested query is sent to the other service CAN. Instances of both service classes are joined and then routed back to the user either from the *Restaurant* CAN or the *Theater* CAN depending on the join order. If the primary query searches for a Chinese restaurant in NYC, it will be routed down the following CAN supernodes: $Restaurant {-}\rangle NYCRestaurant{-}\rangle ChineseNYCRestaurant$. The *ChineseNYCRestaurant* server inserts its own IP address in the *CANServer* header and routes this query to the *Theater* CAN.

The *Theater* CAN is found in various ways. First, during the initial hierarchical routing

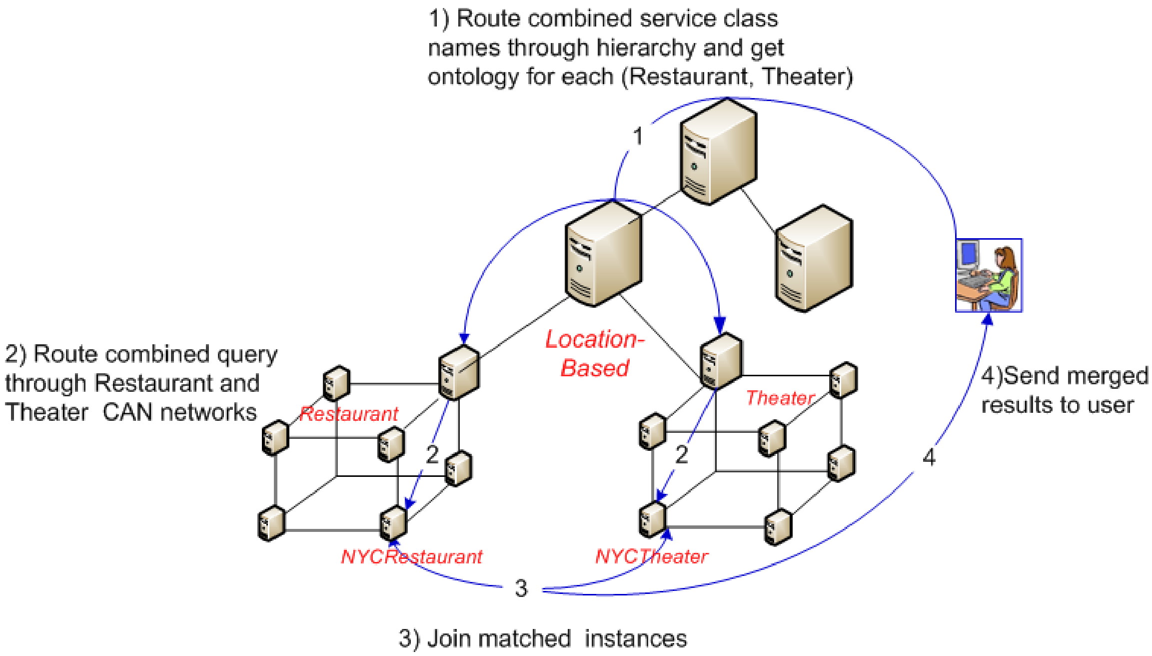

Figure 9.2: Sub-query routing in GloServ

of the combined query, the IP addresses of the *Restaurant* and *Theater* CANs are returned to the front-end and thus can be inserted into the user query message's *CANServer* header. This permits the *ChineseNYCRestaurant* server to send the *Theater* query directly to the *Theater* CAN. Second, if the front-end does not have the header included in the query message, the query processor determines the service class of the nested query from the qualifier in the nested query. Thus, the query processor can route this as a hierarchical query back into the GloServ network and find out the IP address of the *Theater* CAN supernodes. Third, if two services are so closely related that users frequently search for them in a combined query, the routing table of the *Restaurant* servers could have direct pointers to the servers in the *Theater* CAN. Since the latter two methods require greater overhead in routing and maintaining server information, we utilize the first method in our system. Figure 9.2 illustrates how the combined query is routed through the GloServ network.

9.5.3 Query Message

The query message of the combined query is similar to a user query message. Thus, we only list the header values that distinguish the two:

- CANServer: service class and IP addresses of the CAN servers which are communicating with one another. Format is:

  serviceClass1:IPAddr1;serviceClass2:IPAddr2.

- combinedQuery: contains the combined query.

The query message looks like the following:

QUERY

queryType:CombinedQuery

from:127.0.0.1:5501

mainClassName:Restaurant

CANServer:Theater:128.43.23.2;ChineseNYCRestaurant:127.23.54.1

combinedQuery:

((hasLocation some NYC) and (hasCuisine some Chinese)) and (Theatre.hasMovieGenre some Action) and (Restaurant.hasLocation equals Theatre.hasLocation)

queryMatch:exact

queryid:354026

9.6 Shared Properties

Extensive work has been done in the area of ontology mapping [75]. An *ontology mapper* maps classes from different ontologies which have similar semantic meaning to one another. Both the back-end and front-end systems to GloServ access the ontology mapper during a combined query to determine the shared properties. We use a simple method to map shared properties to one another for the purpose of service composition.

As mentioned in Chapter 2, there are two types of properties: object and datatype properties. Object properties map to other ontology classes while datatype properties map to RDF literals such as string or integer data types. For object properties, if two properties have the same class as their range, then they can be considered related properties. For example, the properties *hasLocation* and *isLocated* can be defined in different ontologies where both ranges are equal to the *Location* class. These two properties are considered equivalent. Thus, in a combined query, one method of determining shared properties is by looking at the range of each object property in the two service class ontologies.

A more complex method that provides greater semantic matching is to search for the shared ontologies across the service classes and deduce equivalent meanings of each property value. For the class of *location-based* services, every service is expected to have some kind of location. Thus the service classes within the class of *location-based* services all import the *Location* ontology, which have *Region* and *ZipCode* classes defined. The *Region* class will have zipCode, longitude and latitude properties for each instance. Instances of the *ZipCode* class can have a hasRegion property and four geographical long/lat points pertaining to the boundary of the zipcode. The longitude and latitude properties are datatype properties of type float. Although not all zipcodes are rectangular, a rough mapping of zipcodes to geographical coordinates can still be possible.

For example, let us assume the *Restaurant* class defines its location property range as neighborhoods (which is a subclass of the *Region* class) and the *Theater* class defines its location property range as zipcodes. Continuing with the given query example, since the primary query is the restaurant query, the front-end determines that the shared property will be defined as a neighborhood property. The query is forwarded to the *ChineseNYCRestaurant* CAN server which contains instances for the query. The query processor retrieves the instances and inserts the neighborhood property values in the nested theater query and sends these over to the *Theater* CAN. The *Theater* CAN server processes the query by mapping the shared property values of neighborhood to its own definition of location which is zipcode and queries for instances with these values. The two CAN servers then exchange a set of messages to merge the result set and return it to the user.

Thus, these two ontology mapping techniques create versatile types of combined queries which can be issued within GloServ. We have implemented and evaluated the first method but the ontology mapper can be easily extended to include the second type of mapping.

9.7 Matching services

In order to match the set of services from both queries, a join has to be performed on the shared property. This is similar to a traditional database join operation. The join operation in relational databases allows two sets of results from different tables to be joined over one or more shared properties. It is essentially the Cartesian product of the two result sets. Join is a difficult operation to implement efficiently as the runtime is $O(n^2)$. However, there are a number of query optimization algorithms that make the join operation more efficient [102].

We use the simplest method of optimizing a join by comparing the cardinalities of each of the result sets and determine the join order based on this. The cardinality of each result set in location-based services will be at most in the order of 100s due to the distribution of the service data. For example, the number of Chinese restaurants in NYC is estimated to be a few hundred. However, a *bloomjoin* [99] can be performed if the cardinality of the result sets are large or the join is performed over multiple properties. We describe how both of these methods can be used in GloServ and have implemented the first one.

9.7.1 Cardinality comparison

Using the running query example, after the *ChineseNYCRestaurant* CAN server retrieves a list of instances and determines the shared properties, it sends the nested subquery to the *Theater* CAN. It also includes the cardinality of its result set, CardR in the subquery message. The *Theater* CAN executes its query and retrieves a list of instances. It compares the cardinality of its own result set, CardT, with CardR. If its own cardinality set is smaller, it sends its result set to the *ChineseNYCRestaurant* server. Otherwise, it requests the result set from the *ChineseNYCRestaurant* server. The results are matched in one of the servers and sent directly to the user. Figure 9.3 illustrates the message exchange within the CAN servers to determine the join order.

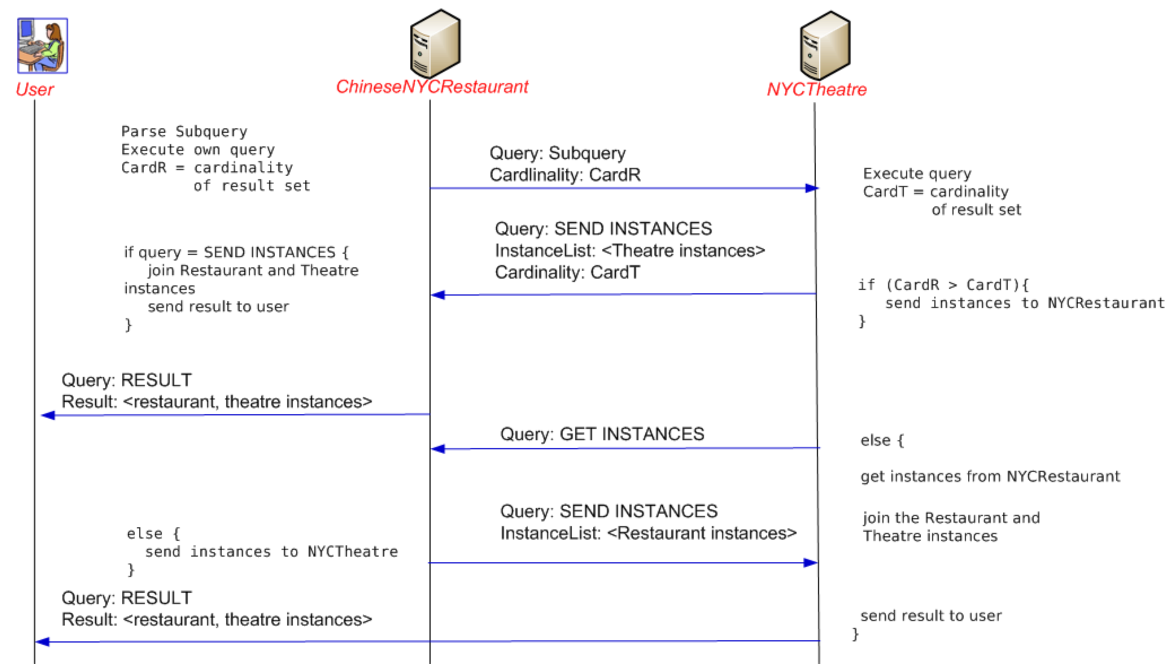

Figure 9.3: Message exchange between Restaurant and Theater CAN servers

9.7.2 Bloomjoin

The bloomjoin utilizes a Bloom filter [63]. A Bloom filter is a vector of bits that are initially set to zero. The vector index represents a hashed value. If the element within the index is set to 1, then the value is present; otherwise it is absent. This is used with a join operation as follows.

The set of property values are hashed to one of the index values of the bloom filter. If the result set contains this property value, its corresponding element in the bloom filter is set to one. The bloom filter shows which properties exist in the result set. The filter should be constructed from the result set which has lower cardinality. Thus, the join reduces to searching for entries in the other result set that match the property value. The same set of messages are exchanged in a bloomjoin as shown in Figure 9.3 except the server that sends the instances also includes its bloom filter.

9.8 Implementation

The implementation of the combined service query components of GloServ is, for the most part, the same as that for single queries. The only additional methods called are the following:

- *QueryParser.findNestedQueries()*: extracts the nested queries from the combined query message.
- *CANNode::issueSubQueries()*: handles the message exchanges between CAN servers.
- *CANNode:: computeCombinedResultList()*: joins the two sets of instances for each service query

9.9 Performance Evaluation of Combined Service Queries

Given the single service query latency for one CAN traversal, a combined service query latency increases linearly as the number of service classes are increased in the combined query. The main point of difference between a single ontology query and a combined service query is in the joining of instances.

Thus the formula to calculate query latency for a combined service query becomes:

$$s(q_h h + \sum_{i=1}^{c}(q_i + l_i))$$

where s is the number of service classes in the combined query. Thus, when searching for a restaurant and a nearby theater, the approximate time will be double that of the unique single query, around 6.4 seconds. For the optimized CAN, discovered in Section 8.7.3, it will be around 50 ms.

9.10 Conclusion

This chapter has described an extension to GloServ which allows combined service queries to be issued to the GloServ network. With this enhancement, multiple service classes can be searched for in a single search. Additionally, the main goals of data sharing within the Semantic Web are realized by the use of ontologies for mapping shared properties between different service classes.

Chapter 10

Combining Key Word Search with Ontology Querying

10.1 Introduction

GloServ has been extended to perform keyword search in addition to ontology querying. Currently, searching for location-based services on the Internet is limited to either text search as in Google Local or Yahoo Yellowpages or attribute-value pair searches as in SLP and Jini. A combination of structured and unstructured searching currently does not exist in service discovery. Thus, I extended GloServ to include keyword searching in addition to ontology querying. Service providers registering their services can add keywords to their service registrations in addition to the ontology description. Users can then query for services using a combined ontology and keyword search. The query will be routed and processed first as an ontology query. Keyword searching will further refine the results by matching certain terms within a service instance's properties.

10.2 Motivation

Combining ontology and keyword queries gives more flexibility and accuracy when obtaining query results. Performing pure ontology queries limits a service description and query to the service ontology definition, but it gives inferencing capabilities which allow for more sophisticated queries. On the other hand, when a service is described purely with text, although all types of descriptions can be added, an obvious drawback is that the semantic correctness of the results can only be approximated and may or may not be what the user is looking for. The examples using Google Local in the Introduction show the inaccuracy of text search when searching for specific attributes within a service. Thus, combining structured and unstructured querying reaps the benefits of both mechanisms by first performing an ontology query in order to route the query to the correct servers, honing in on the correct set of service instances, and then further refining this set with keyword matching on a set of keywords.

10.3 Problem of Combining Key Word Search with Ontology Querying

The combination of ontology querying with keyword search solves the problem of allowing structured and unstructured searches in service discovery. The following challenges exist in accomplishing this: modifying the ontology to indicate which properties have keyword search enabled; changing the query message to include a header for the keyword query; storing and retrieving instances with keywords. Below, I describe the design and implementation on how to incorporate keywords in service registrations and queries.

10.4 Ontology Modification

Keywords are described either as keywords of certain properties or they are keywords of the overall service instance. For example, the *Restaurant* class can have keywords set for the hasCuisine property. A service provider registering its Chinese restaurant can also add menu items such as *Roast Duck* as keywords of the *Chinese* cuisine. The keywords of the overall instance are those that may not be described by any particular property but pertain to the service itself. For example, a restaurant might want to advertise that it has garden seating. Thus, keywords need to be added to the instance itself.

In order to process keywords, the ontology must indicate which of its properties have keyword matching enabled. OWL DL allows properties to be annotated with properties themselves. Thus, in our system, if a property is annotated with a *keyWordMatch*=`true` annotation, then keyword matches are allowed for those properties. It makes sense to enable keywords only for object properties since string datatype properties are already matched with text. Object properties that are to be displayed in the ontology form, in the front-end, are displayed as drop-down menu forms where each value in the menu is a value within the property's range. For example, the hasCuisine property has values such as *Chinese*, *Japanese*, *Italian*. This property will also have a text box displayed underneath the drop-down menu form. In addition, key words for the overall service instance are entered in yet another text box labeled *General KeyWords*.

The main motivation for processing keywords separately for each property and service instance is to be able to classify the key words in some way. Currently, there are a number of applications that allow keyword tagging, like Flickr [15] and del.iciou.us [9]. But there is no way to classify these tags. With this mechanism, machine learning algorithms may be able to process the list of keywords for a given property and over time, with human intervention, these words may be added onto the ontology as classes or properties. For example, as users enter *Szechuan* or *Cantonese* keywords in addition to choosing a *Chinese* cuisine, if this is a keyword that is commonly searched within Chinese restaurants, these words may be added as subclasses of the *Chinese* class. Thus, this allows the system to analyze the social usage of the service class and augment or modify its ontology definition if need be.

10.5 Query Message for Combined Key Word Search with Ontology Querying

A query message which includes keywords is constructed and routed like a regular ontology query except for an additional header which contains the keyword query. The keyword query begins with the main service class name to indicate which service class it belongs to in case of a combined query search. The keywords for each property are named by the original property name itself concatenated with the value of the property. This indicates that the keyword belongs to the property-value pair. Similarly, the key words for overall service instance are indicated by the main service class name concatenated with the word *keyWord* to indicate that this is a keyword for the overall instance.

Let us say the user searches for a Chinese restaurant in NYC which also has the keywords ”Szechuan” or ”Cantonese” for the hasCuisine property and keywords ”garden seating” for the overall service instance. The following query message is sent to GloServ:

QUERY

queryType:UserQuery

from:128.58.18.132:5501

mainClassName:Restaurant

userQuery:(hasLocation some NYC) and (hasCuisine some Chinese)

keyWordQuery:Restaurant-((hasCuisine.Chinese has ”Szechuan”) or (hasCuisine.Chinese has ”Cantonese”)) and (Restaurant.keyWord has ”garden seating”)

queryMatch:exact

queryid:354026

If the user does not specify the logical operator between the keywords, the default is a conjunction of all the keywords. Here, the user has entered "Szechuan or Cantonese" for the Chinese cuisine and hence it is a disjunctive statement.

If the query was a combined query with multiple service classes, the keyWordQuery header would include each set of key word queries separated by a ';' in the following manner:

keyWordQuery:Restaurant-((hasCuisine.Chinese has "Szechuan") or (hasCuisine.Chinese has "Cantonese")) and (Restaurant.keyWord has "garden seating");Theater-(Theater.keyWord has "Woody Allen").

The keyword query treats each property as a datatype property which uses the *has* qualifier to indicate that it is a string. Below we describe how these are stored within the GloServers.

10.6 Processing Service Instances with KeyWords

Registration and query routing for combined ontology and key word messages works exactly the same way as a regular user query as described in Chapter 7. Thus, I will only describe how the messages are stored and accessed within the server because this is the only point of difference due to the addition of keywords.

10.6.1 Database Model

The GloServ implementation uses a database model for querying with keywords because a single GloServer may have thousands of keywords. Say there are 5000 instances in a given GloServer and each instance has on average five unique keywords, then there will be a total of 25,000 keywords stored. When using an ontology model to store instances, this number is too high for there to be an efficient way of querying for keywords. Thus, storing the keywords in a database back-end results in an efficient implementation.

The main point of difference for enabling keyword search is adding database tables for the keywords of each property. Every property which has keyword matching enabled also has a database table with its property name concatenated with the hasKeyWord tag. As described in Chapter 7, the object properties which may have more than one value are stored in child tables. Thus, keywords for these properties will be stored in child tables of these property tables. The keywords for the overall service instance are stored in a direct child table of the service class. These tables are created during server startup. Figure 10.1 describes the entity relation model.

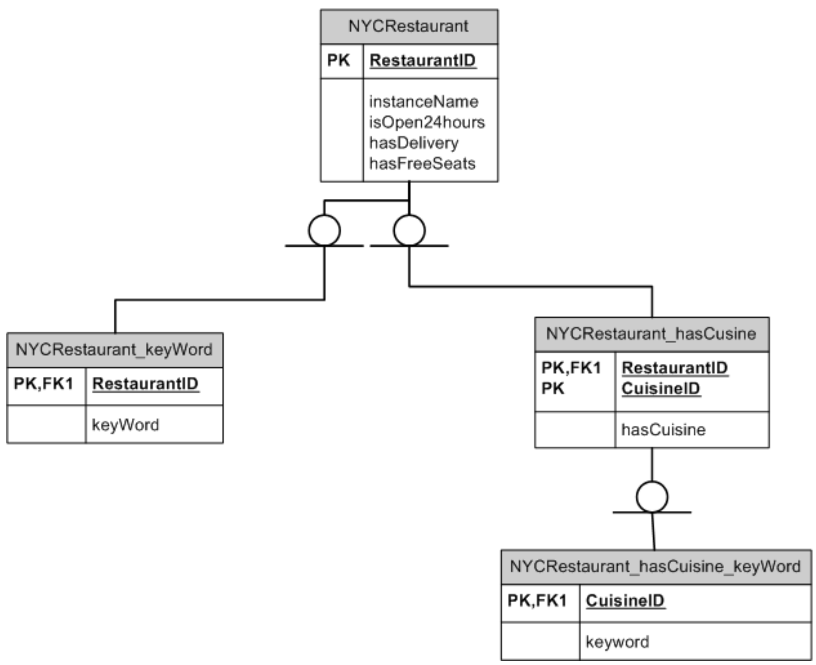

Figure 10.1: Keyword instance table

10.6.2 Registration Processing

The registration processor first checks if the keyWordQuery header exists in the registration message. If it does, it then parses the keyword query and puts it in a hash table where each property-value pair points to a list of key words. When storing instances, this hash table is used to insert all the keywords that do not already exist for that property-value pair. For example, for the query

userQuery: (*hasLocation* **some** `NYC`) and (*hasCuisine* **some** `Chinese`)
keyWordQuery :Restaurant-((*hasCuisine.Chinese* **has** `"Szechuan"`) or (*hasCuisine.Chinese* **has** `"Cantonese"`)) and (*Restaurant.keyWord* **has** `"garden seating"`),

the keywords ”Szechuan” and ”Cantonese” are inserted in the *NYCRestaurant_hasCuisine_keyWord* where the *CuisineID* matches the *CuisineID* of the parent *NYCRestaurant_hasCuisine* table. The same applies to the service instance key word where the term ”garden seating” is inserted into the *NYCRestaurant_keyWord* table where *RestaurantID* matches the *RestaurantID* of the parent *NYCRestaurant* table.

10.6.3 Query Processing

Querying for service instances with keywords involves the same steps as a user query but in addition to this, there is an extra join on the keyword tables to query for the keywords as shown below:

```
 SELECT * from NYCRestaurant as T1, NYCRestaurant\_hasCuisine as T2,
NYCRestaurant\_hasCuisine\_keyword as T3 where T2.hasCuisine =
'Chinese' and (T3.keyWord = 'Szechuan' or T3.keyWord  = 'Cantonese')
and T1.RestaurantID = T2.RestaurantID and T2.CuisineID =
T3.CuisineID.
```

This query matches the keywords exactly. However, in order to be more flexible in the keyword search, the query processor matches using a similarity comparison as shown below:

```
SELECT * from NYCRestaurant as T1, NYCRestaurant\_hasCuisine as T2,
NYCRestaurant\_hasCuisine\_keyword as T3 where T2.hasCuisine =
```

```
'Chinese' and (T3.keyWord like '\%Szechuan\%' and T3.keyWord like
'\%Cantonese\%') and T1.RestaurantID = T2.RestaurantID and
T2.CuisineID = T3.CuisineID.
```

Although this method is less efficient, it does not affect the performance of the overall system as much because the number of instances stored in a given server is expected to be in the order of thousands. Below, we evaluate both methods.

As mentioned above, the default logical operator connecting the keywords is a conjunction. However, if the number of instances retrieved is less than 5 then the processor relaxes the query to be a disjunction and performs the query again returning the results to the user in a list ranging from most accurate to least accurate instances. Accuracy is defined by finding the instances that have the most matched keywords.

10.7 Implementation

Since the main point of difference of keyword queries is the additional keyword processing, the *QueryParser* and classes handling GloServ messages were augmented to include the keyword headers. Also, the *DBInstanceProcessor* class was extended to include storing and retrieving keywords within the database.

10.8 Performance Evaluation

10.8.1 SQL Query Time

In order to get an idea of how long it takes for keywords to be retrieved, I tested the following types of queries: exact keyword matches of 3, 5 and 10 keywords; similar keyword matches of 3, 5, and 10 keywords. I downloaded a list of 58,000 words and inserted it into a keyword table for the cuisine property. The test function randomly chooses 3, 5 and 10 keywords from the list and issues exact and similar matching SQL queries on those keywords. This test was done on both conjunctive and disjunctive queries. The notation AND_n and OR_n signifies the number of key words that are joined together in a conjunctive or disjunctive query equals n.

The exact matches have an expected runtime of $O(1)$, since the table is indexed by the keywords themselves. The runtime for similarity matches is $O(n)$ where n is the number of words in the table, since every keyword is being compared to the one given. Since each keyword table represents a given property in the service class, the number of keywords per table is expected to be around a few thousand since there are a limited number of ways one can express a given property.

Thus, for the first test, I inserted a list of 2,000 keywords in a table and issued exact and similar SQL queries on them. The exact matches took a negligible amount of time of 1 or 2 ms. This was also the case for similarity matches where the worst case of an OR_{10} query took 8 ms and the average OR_5 took around 4. Figure 10.2 shows these results.

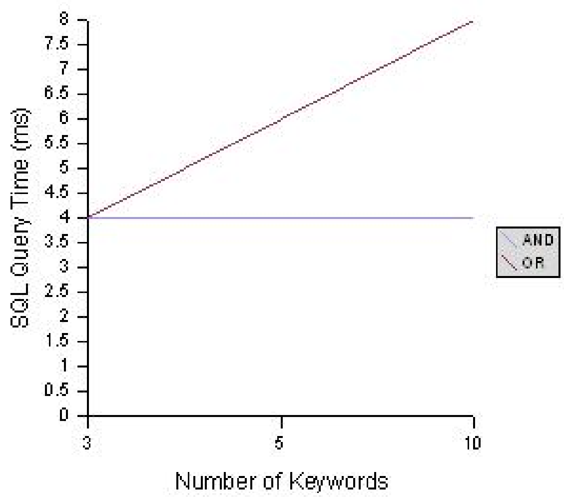

Figure 10.2: Query Latency of KeyWord retrieval for 2000-keyword table

The number of keywords per property table is not expected to exceed a few thousand, however, in order to test how the system performs for a very large data set, I inserted a list of 58,000 words in a table and performed the same tests on these. The resulting graph in Figure 10.3 shows that the similarity matches for conjunctive queries have an average time of 90 ms for all AND_n queries. However, the disjunctive queries take longer as the number of keywords increase. For an OR_3 query the time is 112 ms and for an OR_{10} query, 177 ms. $O(n)$. Exact matches are once a negligible 1 to 5 ms. Thus, these results show that although

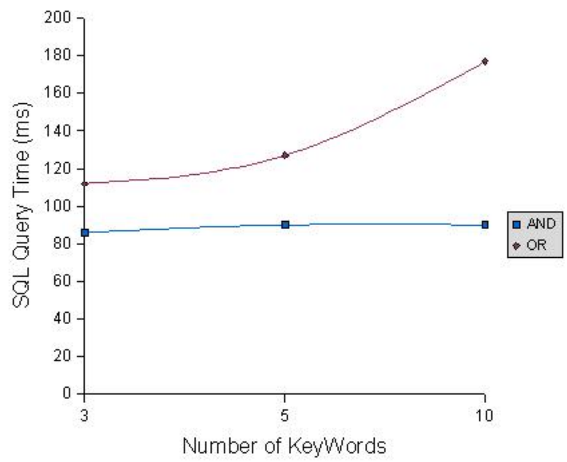

Figure 10.3: Query Latency of KeyWord retrieval for 58,000 keyword table

similarity matches are more expensive for large tables, it is definitely a good choice to use for our system because of the limited number of instances in a given table.

10.8.2 Overall GloServ Query Latency

The main part to evaluate for this contribution is the time it takes to do keyword retrieval in addition to the ontology querying. Thus, the query latency for a given combined ontology and keyword query amounts to the latency of the ontology query as well as the keyword query which is the following:

$$q_h h + \sum_{i=1}^{c}(q_i + l_i + p(k_i))$$

where k_i is the sql query time for a given property and p is the number of properties keywords are inserted for, since there will be a database query for each property.

Taking the results for the average case, the following conditions are given: keyword property table of 2,000 instances; $p = 5$ where 5 properties are populated with keywords; around 5 keywords per property which results an SQL query to be an AND_5 or OR_5 where $k_i = 4$ ms. Thus, the time it takes to retrieve keywords amounts to 20 ms. When inserting this value in the original equation from the ontology querying section, we get the overall GloServ query latency to be 3.4 seconds.

For the worst case, the following conditions are given: keyword property table of 58,000 instances; $p = 5$; 3)OR_{10} where $k_i = 177$ ms. The time it takes to retrieve keywords amounts to 885 ms. Inserting this value in the equation gives 5.8 seconds.

10.9 Conclusion

This chapter has described the design and evaluation of a GloServ query extension to include keyword searches in addition to ontology querying. This work shows that search need not be limited to fully structured or unstructured formats. Ontologies provide a basic structure where query matches are logically equivalent or similar to a user's query and the addition of keyword matches filters out the results to be those of the user's preference. Thus, systems which use tagging for labeling data may add structure to their system by inserting these tags in ontologies, enhancing the querying mechanism of these systems.

The evaluation results shows that the addition of keyword matching does not change the overall query latency of a given query by much. An ontology query takes 3.2 seconds. With the addition of keywords, on average it will take 3.4 seconds and in the worst case 5.8 seconds. With these promising results, one can imagine services uploading their webpages in the system and allowing a parser to extract common keyword terms from the webpage in order to store these keywords along with the ontology instance. This shows that the GloServ architecture can be a good medium for Semantic Web-driven applications where webpages are classified in a structured format as envisioned in the Semantic Web.

Chapter 11

Related Work

11.1 Service Discovery Protocols

There are a few service discovery protocols in use today. Most service discovery mechanisms operate on local-area networks and use attribute-value pairs for service descriptions. Below we describe each of these and compare them to GloServ.

The Service Location Protocol (SLP) [89] has three main components: *User Agents (UA)* which perform service discovery on behalf of a client, *Service Agents (SA)* which advertise location and characteristics of the service on behalf of the service, and *Directory Agents (DA)*, which are optional, record available services and also respond to service requests from UAs. In SLP, there are two modes of operation; one includes the DA and the other does not. When a DA exists, the UAs learn about services by unicasting messages to the DA. Otherwise, UAs send multicast messages to the SAs and learn about available services. SLP can also operate without DAs and in this case agents multicast messages to one another by flooding the network. This also does not scale to large networks. The mesh-enhanced Service Location Protocol (mSLP) [127] was designed to enhance the scalability of SLP. It has a fully-meshed peering DA architecture with one or more scopes in common, which allows the network to scale at a wider level, such as in enterprise networks.

Jini [101] is built on top of the Java object and RMI system. Service registries, similar to SLP's DAs, are used to register service proxy objects and act as lookup services. Through a discovery process, a client downloads the service proxy and invokes it to access the service. The Java class hierarchy defines services and their attributes.

UPnP [81] differs from SLP and Jini in that it doesn't have a central service registry but services just multicast their announcements to control points that are listening to these messages. Control points can also multicast discovery messages and search for devices within the system. The eXtensible Markup Language (XML) [11] describes the services in greater detail. UPnP is appropriate for home or small office networks.

The Universal Description, Discovery and Integration (UDDI) [45] specification is used to build discovery services on the Internet. UDDI provides a publishing interface and allows programmatic discovery of services. Services are described in XML and published using a publisher's API. Consumers access services by using the programmer's API built on top of SOAP [35]. Services in UDDI are stored in a centralized business registry. The main drawback of UDDI is that it has a centralized architecture and thus does not extend to a global area.

The Lightweight Directory Access Protocol (LDAP) [123] is a protocol for querying and modifying directory services running in the Internet and intranet. The directories arrange information with similar attributes in a hierarchical manner. LDAP directories are optimized for read performance because it is not built for real-time data which requires frequent updates. Information in LDAP is also represented in a series of attribute-value pairs.

Recently there have been developments in wide-area service discovery. INS/Twine [57] and Ninja [85] [71] describe two such systems. Both systems use attribute-value pair XML messages to describe services. However, INS/Twine extracts each unique subsequence of attributes and values, which they label strands, to a structured peer-to-peer system such as Chord. Ninja, on the other hand, organizes servers dynamically into hierarchies and issues upward queries using Bloom filters.

Traditional service discovery systems are also being enhanced to perform semantic matching and increase network scalability. DReggie [65] enhances the Jini matching mechanism by implementing a Prolog reasoning engine used for matching services. Services register with the DARPA Agent Markup Language (DAML) [7] ontology which provides richer service descriptions.

GloServ differs from all of these systems in that it is globally scalable by incorporating a hybrid hierarchical and structured peer-to-peer architecture. It also has greater logical capabilities in its use of OWL-DL for its architectural design and service descriptions. The main difference between using OWL and any other attribute-value or XML description mechanism is that OWL not only classifies services hierarchically but also allows logical restrictions on class relationships. By using OWL, the relationships of the services to each other are known. According to these classifications, the service discovery architecture is constructed. The logical capabilities of OWL aid in finding the appropriate service classes within the system as well as in content distribution and query propagation. Also, since the service discovery architecture is a peer-to-peer system, frequent reads and writes for dynamic services are enabled.

11.2 Agent-based Service Discovery Systems

Agent-based systems apply concepts from artificial intelligence in order to distribute object technology [16]. An agent is a software component that is autonomous, proactive and social. Autonomous agents control their own actions and are able to take decisions for themselves. Proactive agents not only react to external events, such as remote procedure calls, but also take initiative where appropriate. Social agents interact with other agents in order to accomplish their task and achieve the complete goal of the system. Agent-based systems are peer-to-peer and each agent can initiate communication with any other agent.

The Foundation for Intelligent Physical Agents (FIPA) [16] was formed, which is an industry-led standards body in the agents area. FIPA has designed an agent platform [14] for service discovery which consists of three main components: a directory facilitator that is a yellowpages directory where agents providing services can register; an agent communication channel, which allows agents to communicate with each other; an agent management system that manages the agent life cycle, such as starting, deleting, or accessing an agent. Agents communicate with each other using the Agent Communication Language (ACL) [13].

Agent-based service discovery systems utilize the FIPA agent platform. Two such systems, among many others, are JADE (Java Agent Development Framework) [58] and its extension, LEAP (Lightweight and Extensible Agent Platform) [59]. Both of these java-enabled systems provide a software framework which allows to implement a multi-agent service discovery system. JADE operates on a structured robust platform where machines have higher processing power and more memory, whereas LEAP operates on lightweight devices. The Ronin agent framework [68] is built on top of Jini. It has an agent-oriented architecture where agents communicate with each other using ACL. An *agent deputy* is located in the Jini lookup service and mediates agent communication as well as represents the owner of the agent.

The main difference between GloServ and these agent-based service discovery systems is again the issue of global scalability of nodes within a network. The agent platform which incorporates the different components of a multi-agent system, such as, the directory facilitator and agent management system, does not account for global scaling of agents. One of GloServ's main objectives, however, is to provide a service discovery system that scales globally.

11.3 Schema-based Peer-to-Peer Systems

The existing work closest to our research use schemas or ontologies to create a mapping of the network [98], [47], [122].

Aberer et al., in [47] outlines a semantic gossiping framework that exchanges XML schema information within a peer-to-peer network. It uses Gnutella [84] as its underlying peer-to-peer structure. The main problem with this system is that it uses flooding to broadcast its queries and thus reduces the scalability of the network.

Loser [98] proposes the HyperCuP, a 2-tier peer-to-peer hierarchy which uses indices within a super-peer topology. The indices are built using RDF schema information from associated peers. Super-peers are connected to each other in a hypercube [117] and represent disjoint concepts. Underlying peers of a super-peer contain information regarding that particular concept. A search is done via broadcast with a time-to-live. Due to the hypercube structure, the complexity is $O(\log_2 n)$, assuming the graph structure is balanced and optimizations have been implemented. This structure is very similar to the CAN except that the number of dimensions are limited to 3 due to its hypercube structure.

Meteor-S [122] [120] provides a peer-to-peer infrastructure of registries for semantic registration and querying of web services. It organizes the architecture in three layers: 1) the *data layer* contains UDDI registries; 2) the *communications layer* organizes *operator peers*, of these UDDI registries, in a peer-to-peer network using the JXTA [22] platform; 3) the *operator services layer* maintains all the service information provided by the operator peers for semantic discovery.

GloServ is similar to these systems in that a service class represents a sub-network. However, the underlying service discovery architecture differs from these systems. HyperCuP and the semantic gossiping framework use broadcasting to send queries which can not scale globally. Meteor-S uses a centralized and replicated UDDI service registry which, again, does not scale globally or allow dynamic service registrations and updates. GloServ, on the other hand, implements the service registries within CAN peer nodes, and thus is a fully distributed system.

Additionally, GloServ organizes disjoint concepts in a hierarchical network whereas concepts that are similar to each other are organized in a CAN. Indices within the CAN are formulated according to the ontological content of each node whereas in HyperCuP, indices refer to whole peers.

Finally, the HyperCuP and semantic gossiping framework describe the peer-to-peer network formation, namely, dealing with nodes entering and leaving the system and Meteor-S describes an overall architecture. However, they do not completely describe how the data is distributed and queried for. GloServ, on the other hand, implements the formation of the network as well as defines algorithms that distribute and query the data by mapping the ontology onto a CAN network.

11.4 Web Services

I would like to clearly identify the differences between the Web Services Architecture and GloServ in this section. The Web Services Architecture [39] was designed by the World Wide Web Consortium. It defines web services as a software system designed to support interoperable machine-to-machine interaction over a network. A *Web service* is an abstract notion that must be implemented by a concrete agent. Web services are described in a machine-processable language, such as the Web Service Description Language (WSDL) [40]. The Simple Object Access Protocol (SOAP) [35] is used to send messages indicating how other systems interact with the Web service, and are typically transported using the Hyper Text Transfer Protocol (HTTP) [79]. The service discovery phase is not specified in the Web Services architecture spec [39]. It proposes three essential ways of discovering a service: centralized registry system such as UDDI, a text-based search engine such as Google, or peer-to-peer architecture. Figure 11.1 illustrates the Web Services Architecture. GloServ concentrates on service description, distribution and discovery. Thus, web services could be one type of service registered within GloServ. Alternately, in the Web Services Architecture, GloServ can be incorporated as the service discovery entity.

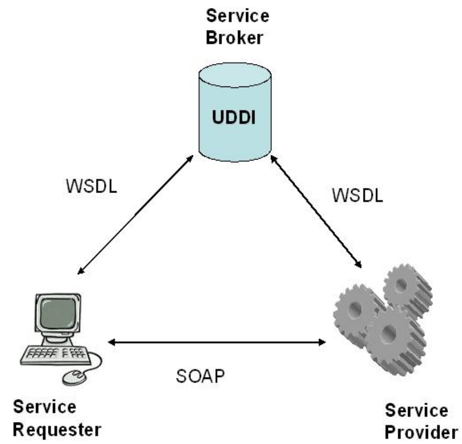

Figure 11.1: Web Services Architecture

11.5 Service Composition

As described in Section 11.4, traditional web service composition uses the Web Services Description Language (WSDL) to represent service descriptions and UDDI to store and discover services. They also use flow specification languages such as Business Process Execution Language for Web Services (BPEL4WS) [77] and Web Service Choreography Interface (WSCI) [76] to design the execution flow of the composed services.

Since these systems are limited in the way services are described, using WSDL, there is a large body of work that has been done in enhancing web service composition to allow greater flexibility in service matching by using ontologies, such as OWL, DAML, and their corresponding web service versions, OWL-S [29] and DAML-S [8], for describing services. A few of these systems are described in [64], [78], [108]. Although these systems provide greater semantic service matching with the use of ontologies, they still use the UDDI service discovery registry which limits the scalability of the system.

In the realm of finding tagged services, Cooltown [96], Splendor [128] and Agents2Go [111] are systems which allow services to be tagged. Cooltown describes a system where every service is connected to a web server and a tag. Users walk in and receive a URL from tag beacons and can then connect and invoke the service. Resources are managed and discovered in the *place manager*. Splendor is a tag-based location aware service discovery system. Tags label location and people. Proxies provide service registration and querying. Agents2Go implement localized brokers that store services in a certain location. Users querying for services are first mapped to a given location and their queries are issued to the nearest broker.

These systems allow searching for tagged services, but in a local environment. GloServ, on the other hand, allows tagging and search of tagged services on a global scale.

11.6 Key Word and Ontology Search

General research has focused on combining ontology-based search with information retrieval techniques, especially within medical informatics community where ontologies are used to structure documents. The main goal of this research area is to add semantic meaning to documents. The key words are already defined within these documents and they are then mapped into the ontology and classified within certain domains. A few of these systems, among many others, are described in [106], [46] and [95].

GloServ addresses the reverse problem. Its goal is not to bring structure to unstructured text documents, but to allow already-structured service instances greater flexibility in its descriptions. Since the service classification ontology is used to distribute data in peer-to-peer overlay networks, the set of key words generated, as services register, will belong to a certain number of service classes handled by that server. Thus, key word generation and search is dynamic and can apply to all service domains. Unlike in information retrieval, the query results will be logically perfect matches because of the ontology query and keyword search only causes the results to be further refined to the user's preference.

Part IV

GloServ Front-End Interface

Chapter 12

Web-based Front End Interface

12.1 Introduction

This chapter describes the various front-end applications of GloServ. One of the contributions of GloServ is that it provides a generic back-end service discovery framework for different front-end systems to interact with it. We have built a web-based front-end which demonstrates how GloServ can be used for different types of web services such as: location-based services, tagged services and collaborative search with other users. I have designed but not yet implemented a context-aware agent architecture on top of GloServ that classifies the services according to context, as future work. These agents receive context information, map them to the appropriate GloServ service classes, and translate them to GloServ queries. This chapter and the next describe the web-based and the context-aware agent front-ends, respectively.

12.2 Motivation

The motivation to create a web-based front-end to GloServ is to demonstrate that GloServ can be easily plugged into a web service architecture. As described in Section 11.4, a *Web service* is an abstract notion that must be implemented by a concrete agent [39]. The Web

Service Architecture may have different types of service discovery systems such as centralized registries (UDDI), text-based search engines (Google) or peer-to-peer architectures (GloServ). Using GloServ for web service discovery shows that service discovery can be done using a richer description language as well as within a distributed architecture. We demonstrate that a web service front-end can be implemented which discovers services using GloServ. However, we have not built a front-end which interacts with the service providers directly using SOAP and HTTP as this is not applicable to the service discovery phase and is beyond the scope of this thesis.

12.3 Problem of Designing a Web-based Front-end to GloServ

In order to build a web-based front-end to GloServ, the following problems need to be addressed: generating a search form from the service class ontology files; generating cascading forms for combined service queries; composing a valid GloServ query based on the user selection in the search form; displaying the results to the user in a coherent manner. Below, I describe the solution of the design and implementation of the web-based front-end system in Section 12.4 and then delve into a number of use cases that the front-end has been used for in Section 12.5.

12.4 Web-based Front-end

The GloServ front-end is a web server that runs Apache and PHP [32] (version 5.1.x). It allows users to register and query for location-based services. Currently, the service classes that are supported are for *Restaurant*, *Theater*, *Weather* and *Tagging* services. The interface provides links for each service class. When the user clicks on a service class name, a form is generated for that service class. The query results are displayed in a Google map as well as in a list. The overall GloServ front-end can be seen in Figure 12.1. This section describes how the front-end operates in order to perform location-based service discovery using GloServ.

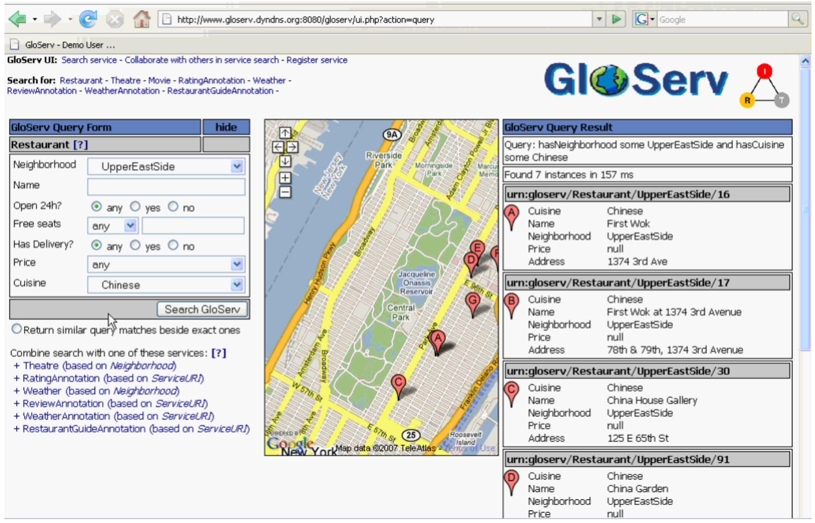

Figure 12.1: GloServ Front-End

12.4.1 Generating a Search Form

The first step that a front-end needs to accomplish is to download the correct service ontology files from the GloServ back-end. Depending on what the web service is, it contacts the GloServ back-end by submitting the service class name. This query is routed through the GloServ hierarchy as described in Chapter 6 and the ontology is returned to the front-end server. The front-end caches these ontology files as well as the hostnames for the CAN super nodes of the service classes and refreshes these periodically. Thus, the query does not have to route through the hierarchical servers but can go directly to the CAN level.

The front-end parses the ontology and displays each property as a field in the form. As described in Chapter 4, each property is annotated with a label property which is used for the graphical user interface. The ontology parser parses out each property's label and displays it as the field's label in the form. The form field is determined by the property's range. If the range is an object property, all the classes in the range are parsed and displayed as a drop-down menu. For datatype properties, the field is a regular text box. For example, the hasCuisine property has annotated properties label.us=`Cuisine` and label.de=`Kueche`. These two labels annotate the display labels for the hasCuisine field in the English and German languages. The form field is a drop down menu of the *Cuisine* class and displays all the subclasses of *Cuisine*. Similarly, the hasName datatype property is labeled `Name` and its field is a text box. Figure 12.2 illustrates this concept.

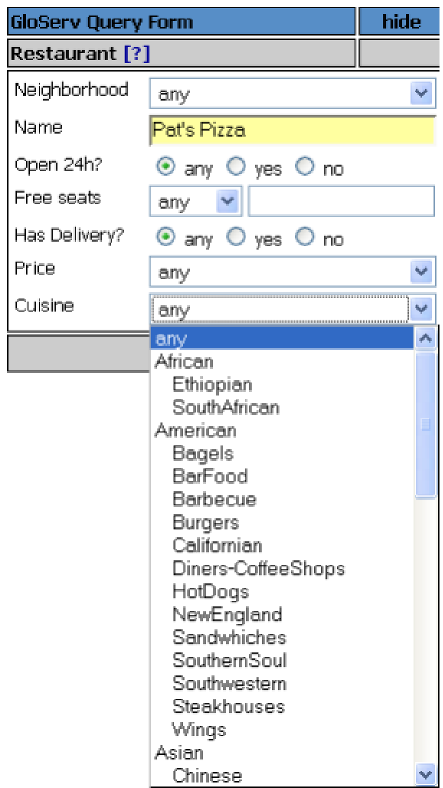

Figure 12.2: Ontology Form

12.4.2 Cascading Forms for Combined Service Queries

We have built a front-end that supports combined querying of services. In order to accomplish this, the user interface needs to be able to display more than one service form at a time. The front-end server downloads a number of ontologies for each service class. In order to allow multiple services to be searched for in a single query, the relationship between these service classes needs to be established. As mentioned in Chapter 9, this is accomplished by passing the ontologies through an ontology mapper which establishes the relationship between the ontologies. We implement a simple ontology mapping tool by storing the relationship between services in a relationship table, implemented as a PHP array. This also includes the corresponding matching properties.

The front-end interface displays the combined service classes as links. When users click on these links, cascading forms are displayed. The matching property or properties are only displayed once and inserted as a shared property in the combined query. Figure 12.3 below shows a combination of *Restaurant* and *Theater* service forms displayed with one *Neighborhood* field which is the common property.

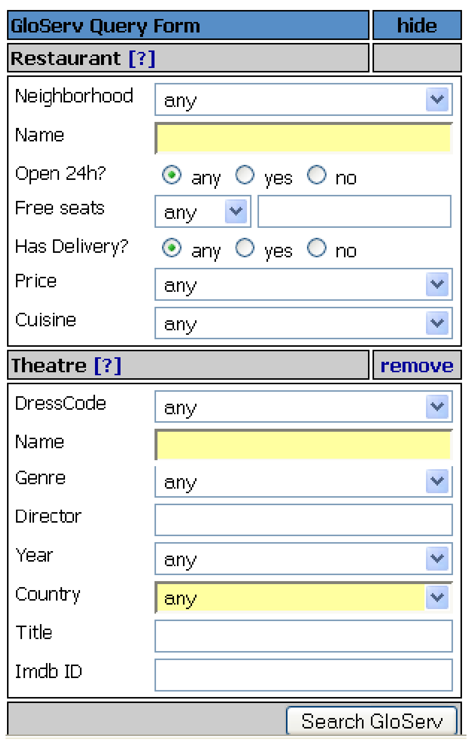

Figure 12.3: Cascading service forms

12.4.3 Creating GloServ Queries

Single service ontology queries in GloServ are constructed with *AND_EXPR*s. Although more complicated queries can be formed (i.e., disjunctive queries, cardinality assignments, and set equivalences), a conjunctive query suffices for the purpose of the demonstrative use cases. Besides creating simple ontology queries, the front-end also checks to see if there are combined queries or keywords entered and creates the query accordingly.

The information entered in by the user is converted to a GloServ query. Each object property and its value becomes a *SOME_EXPR*, such as hasCuisine **some** `Italian`. A datatype property and its value becomes a *HAS_EXPR*, such as hasName **has** `"Pat's Pizza"`. These expressions are joined together in a conjunction.

For combined queries, the front-end creates a *primary* query out of the first service class and *nested* queries from subsequent service classes. In the example above, the *Restaurant* service query will be the primary one and the *Theater* query, the nested one. If there were more services cascaded in the form, then these would be nested within each other.

For keyword queries, the list of keywords for each property is inserted in the query message header as described in Chapter 10. Keywords are entered in a text box underneath each property field.

The query is put in an XML message which indicates what type of query it is (service registration or query). It is then passed onto the front-end server which constructs the appropriate GloServ message and issues the query to the GloServ backend.

12.4.4 Displaying Results

For single service query results, the results are displayed in a list where each instance and its properties are displayed. In combined queries, results are displayed using an instance tree that shows the relationship between each service instance. An example of this is the combined query of *Restaurant* and *Theater*, where the tree is formed such that instances of the *Restaurant* class are parents of matching *Theater*.

For location-based services, the results are also displayed in a Google map. Each service instance is labeled with a pin on the Google map. For combined service queries, the map shows the primary query tagged in red and the nested query tagged in green. When clicking on either the red or green tags, the user can choose to show only corresponding services. This grays out all the services except for those services which match the current one that has been clicked. In this way, users can easily navigate through the map and can determine what the matching services are. Figure 12.4 shows how the query results are displayed for a combined restaurant and movie search.

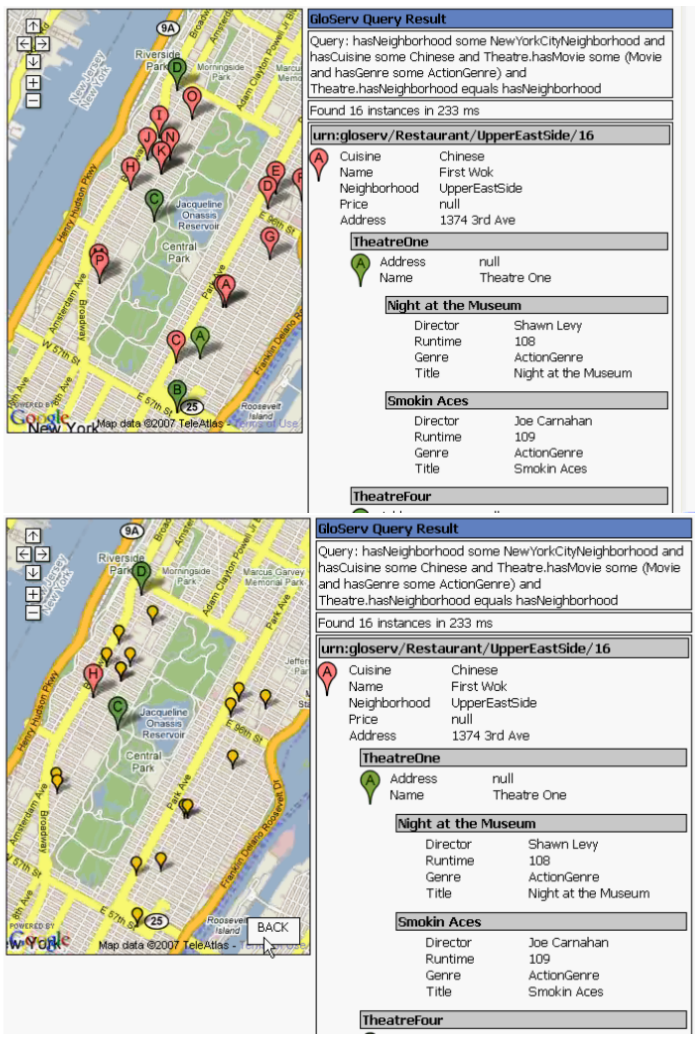

Figure 12.4: Google map results for combined service queries

12.5 Use Cases

GloServ provides a very good framework for discovering location-based services. As discussed earlier in Chapter 9, even though services may define location in various ways, it does not pose a problem in GloServ due to its use of ontologies. Additionally, since GloServ aggregates service data into a distributed network, it is possible to search for different service classes. This section describes various location-based service discovery applications which use web-based front-end systems as described above.

12.5.1 Location-based services

The use case mentioned in the Introduction chapter, in Section 1.2.1, is applicable with this web-based front-end implementation. To review, the use case describes the following scenario: a user searching for the quickest route to a seafood restaurant, near the waterfront in Manhattan, which also has a theater nearby playing an action movie. In this case, three service classes are queried: *Restaurant*, *Theater*, and *Traffic*. In order to accomplish this type of service discovery, the front-end downloads ontologies for these service classes and allows the user to search for a combination of these by providing links to the service forms. There are three cascading forms that appear to the user, one for each service class with the shared location field appearing once. The results are displayed in a Google map as seen in Figure 12.4.

12.5.2 Service Tagging

As described in Section 9.2.2, searching for restaurants and theaters with specific reviews becomes simple with this front-end. The user queries for *Restaurant* and *Rating* by choosing these service links and entering data into the restaurant and rating forms. Users can also add tags to a service by providing the service URN and the reviews in the rating form.

12.6 Collaborative Search

An interesting application of GloServ is letting users collaborate in a single search. Imagine two mobile users who decide on a last minute dinner meeting. They would like to perform the search together but one of them is sitting in a meeting and is unable to talk. Thus, her friend invites her to collaborate with her on a service discovery search through GloServ. The invitation is sent via email or text message which includes a link to the GloServ collaborative interface. As *user A* clicks on her preferences, the values change on *user B*'s front-end as well, by synchronizing the interfaces. The values entered by both users is converted to a regular GloServ query and issued to GloServ.

12.7 Implementation

12.7.1 Front-End

The front-end implementation is done mostly in PHP. For the collaborative interface, functionality in JavaScript [20] has also been defined.

- *GloServ::constructQuery(), GloServ::constructQueryPart()* issues the query to the GloServ backend interface.
- *GloServ::getInstancesFromXml();GloServ::seperateInstancesByServiceClass()* interprets XML results from the GloServ back-end.
- *ServiceManager::calculateRelationships()* implements a simple ontology mapping tool by storing the relationship table of the different service classes.
- *GloServ::constructInstanceTree()* constructs an instance tree from the XML results for combined query results.
- *GoogleMaps::assignMarkerBasicInformation()* marks all the instances found on the Google map by coloring each service class with a specific marker; displaying results

when clicking on a marker; grouping combined service query results and graying out the rest.

- *Smarty* is the front-end template engine that generates HTML code for PHP [37]. Smarty facilitates a manageable way to separate application logic and content from its presentation.

- *GlServAjax.js, GloServDB.php* implement the collaborative search interface, which requires user view synchronization and a central data storage. The Asynchronous JavaScript and XML (AJAX) [4] framework is used to communicate with the client side and web server. AJAX provides a framework for creating efficient and interactive web applications.

12.7.2 Front-end to Back-end Message Exchange

GloServ nodes communicate with each other using a form of property-value-pair encoding. The implementation separates the message format and encoding from the logic. All messages are represented as classes derived from the *IGloServMessage* interface. The interface offers two methods, one for parsing and another for composing of messages.

- *FrontendServerInputHandler* receives the XML message from the browser and passes this onto the ExternalAPI class.

- *ExternalApi::executeRpcCall()* reads the XML message and calls the following methods based on the type of message (registration or query):

  - Query: is handled in the *SyncQueryApi::issueSyncQuery()* method which constructs a GloServ query message and issues it to the GloServ back-end by calling *QueryAPI::issueQuery()*. It then waits until it is notified by the *QueryResultServerThread* that the query answers have been received.

– Registration: is handled in the *QueryAPI::registerInstance()* method. It constructs a GloServ registration message and issues it to the GloServ back-end.

12.7.3 Code Walk

The code walk in Figure 12.5 below shows how the messages are exchanged between the user's browser, the front-end server, the Smarty template engine and the GloServ back-end interface. The user calls the main PHP script, *ui.php*, twice: the first time to get the service search form; the second to issue the query. The codewalk assumes that the front-end server already has the ontology file in the cache.

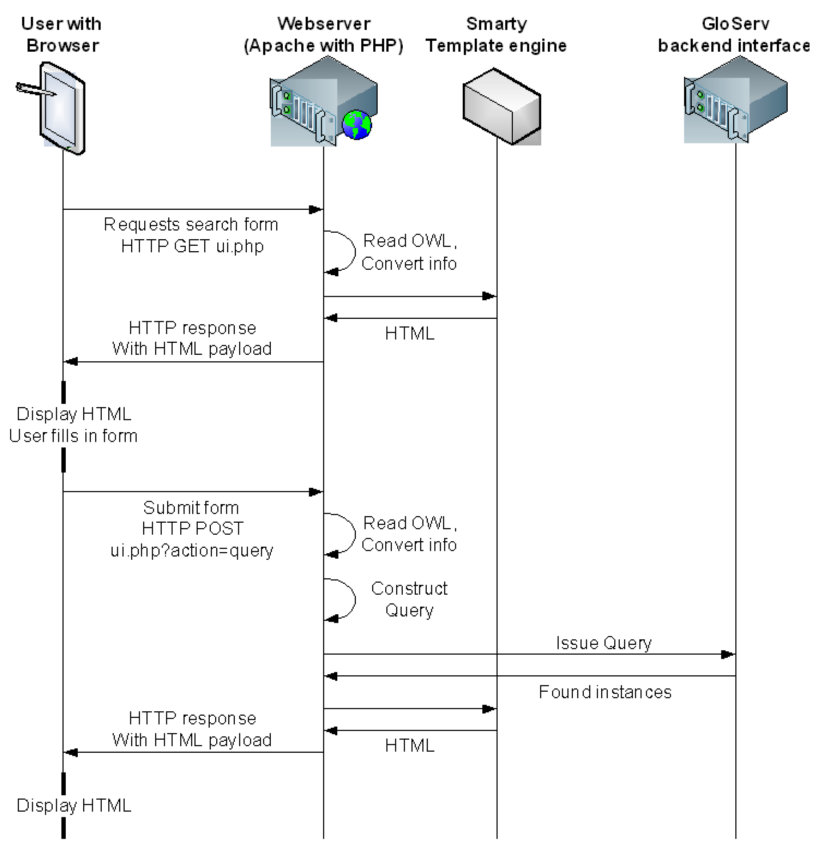

Figure 12.5: Query code walk

12.8 Conclusion

GloServ provides ontology service descriptions as well as a framework for different service classes to be aggregated in a single network. Due to these attributes, we have built a web-based front-end to demonstrate the interesting use cases for GloServ. These include searching for a combination of location-based services, tagged services as well as collaborating with other users on a single search.

Chapter 13

Future Work: Distributed Context Aware Front-End

13.1 Introduction

Currently, context-aware systems, such as [67], [91], are limited to local area networks which have traditional client/server architecture that prevent global scaling. The goal of these architectures is to aggregate context from different sources within a local environment in order to determine the appropriate service for the user. Context-aware systems that are distributed use simple attribute-value descriptions of context and services, which results in only exact query matches without considering related ones. I improve on current systems by providing a distributed context-aware agent architecture which receives context information from various sources and translates them into specific GloServ queries. As a part of my ongoing work, I have designed an agent architecture on top of GloServ that classifies the services according to context [54]. These agents receive context information, map them to the appropriate GloServ service classes, and translate them to GloServ queries.

Extensive work has been done in defining and categorizing context [72] [73] [126] [116]. I adopt the definition and categorization of context specified in [72], which is defined as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves." Furthermore, it is categorized into three main classes:

- Computing environment: available processors, devices accessible for user input and display, network capacity, connectivity, and costs of computing.
- User environment: location, collection of nearby people, and social situation or activity.
- Physical environment: lighting and noise level.

A service context ontology classifies GloServ service classes according to their context attributes. Agents, distributed similarly to GloServ and connected to the GloServers, hold this context information. The service context ontology has two main purposes: it describes the context attributes of each of the high-level service classes within GloServ; it maps a distributed network topology.

The context ontology classifies GloServ service classes within the three main context classes defined above. Although many services can be classified into all three context categories, one of the categories will be of highest priority. Service attributes are ranked in a partial order within GloServ in order to prioritize the importance of each attribute. The service class is categorized in the context ontology according to these priorities.

13.1.1 Motivation

To motivate the necessity for a distributed context-aware service discovery architecture, let us look at the following scenarios that include user-environment contexts. The first one is of a traveler who already knows what type of services he wants based on his context. Suppose he has a rule set up that when his context information consists of time, location and activity: *day time*; *Paris*; *tourism* and *walking*, then he must query for a Restaurant or Museum service. His device automatically sends this information to a context-aware agent, along with his preferences, when the context conditions are satisfied. The agent routes this information to the correct agent that handles this service context class. That agent then stores the context information, processes it, and presents the user with a specific GloServ query. The traveler stores this query and the next time this context condition is true, a direct query is issued to GloServ. The second scenario is when the traveler does not know what service he needs but knows his given context. In this case, it sends the context information to the agent and the agent searches the context ontology and matches the given context combination to a set of service contexts. The user receives a list of possible services that are appropriate to his situation and indicates which ones he is interested in, along with his preferences, and the agent again provides him with specific GloServ queries.

Since it is expected that these users have thin clients, they have limited processing capability. Thus, maintaining large context ontologies on their systems is not wise. Instead, a generic context ontology is distributed across context-aware agents that keep track of context history and find appropriate services for the users. This asserts that distributing context is necessary. Below I describe the architecture and algorithms that allow these types of service requests to be issued.

13.2 Context-aware Agent Architecture

13.2.1 Context Ontology

The context-aware agent architecture is based on an OWL context ontology classification. The context ontology represents service context and is formed with the three categorizations defined above: *computing environment, user environment, physical environment*. Furthermore, context attributes are defined for each of these classes. For example, the *user environment* class has a subset of attributes such as location, activity, time. The attributes are OWL object properties, which have as their range a set of OWL classes. The location attribute maps to the *Location* class which classifies global regions into country-state/county-city, zip code, or longitude-latitude coordinates. The *ComputingEnvironmentContext* class may also have these context attributes, but it will have a higher rank for attributes networkCapacity and connectivity. The *Telecommunication* service class within the GloServ service ontology ranks its properties in a full or partial order. It has networkCapacity, connectivity and may also have location properties, among others. The properties are ranked during the construction of the ontology. A telecommunication service relies heavily on the network capacity and thus this property is ranked highest. It is then matched to all the properties within the service context ontology and grouped within the class that has this property as the highest rank. Since the highest ranking properties for the *ComputingEnvironmentContext* are networkCapacity and connectivity, the telecommunication service context is classified under this context class.

The three context environment classes are disjoint from each other because they each have different priority orders assigned to each of the context attributes. Even if they share a certain attribute, they will not share the priority level of that attribute. This way, when a combination of contexts is classified, the priorities are taken into consideration and this causes it to be classified under one of the three context classes.

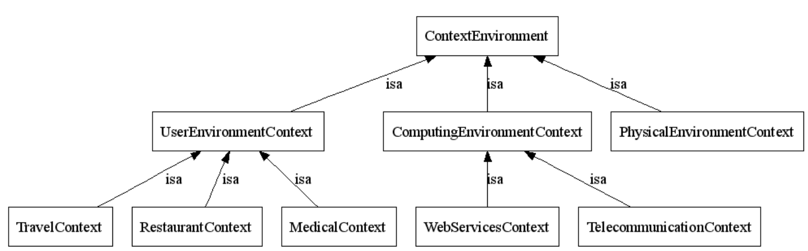

Figure 13.1: Context ontology of GloServ services

13.2.2 Agent Architecture

The context-aware agent architecture sits on top of the GloServ architecture. This means that context-aware user agents are connected with corresponding GloServers for a given service class. Since each agent handles a specific service class, it is aware of the GloServer hosts that maintain those services. In case a GloServer goes down, the agents find the new one by querying for the service class once again. Figure 13.2 gives an overview of the context-aware agent architecture.

For the case of classifying services within a context ontology, working with the high-level services is sufficient. The main purpose of this ontology is to map various context attributes to service classes. GloServ already implements a robust hierarchical peer-to-peer network architecture which handles service classification of both high-level services as well as similar services. Since it is not necessary to replicate this within the context-aware agent architecture, the context-aware agent architecture is a purely hierarchical structure and is created for the purpose of mapping context attributes to high-level service classes within GloServ.

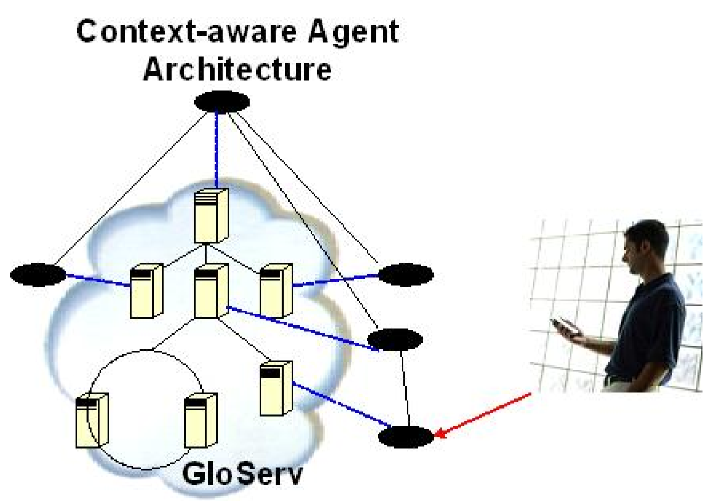

Figure 13.2: Context-Aware agent architecture connected to GloServ

Each agent holds context ontologies of its parent and children in order to navigate the requests to the appropriate agent. It also contains two types of thesaurus ontologies: one mapping synonymous service names to the actual service classes within the context ontology; the other mapping synonymous attributes to actual attributes within the context ontology. So the words ”cafe” and ”diner” map to the *RestaurantContext* service class; the words ”place” and ”street” map to the location attribute.

Bootstrapping agents into the context-aware agent architecture is also accomplished similarly to GloServ as seen in Figure 13.2, except it uses the context ontology for determining the host. Each agent represents a context service class and its hostname is determined by looking at the primitive skeleton ontology. Hostnames will follow the hierarchical format. For instance, as seen in the context ontology in Figure 13.1 the *Restaurant* class’s URN will be urn:gloserv:Context.RestaurantContext.UserEnvironmentContext.Context. As an agent is assigned to a hierarchical network, it updates the ontology to include its host information. Users access a well-known URI which maps to a certain number of random high-level context-aware user agents. If the user knows the service classes it is searching for, the agent uses a thesaurus ontology to match the service class names to the actual class names within the context ontology. The initial agent contacted will forward the user’s request to the agent that handles the service context class the user is searching for. Thus, if initially the *TelecommunicationContext* class is contacted and the user is searching for a ”cafe”, the agent maps the word ”cafe” to the *RestaurantContext* class and checks to determine the hostname of the agent handling that class. If it does not know this, it finds the nearest ancestor to the *RestaurantContext* class that it does know the hostname of and forwards the user’s request to that agent. This process continues until the correct agent is contacted.

13.3 Mapping Context Attributes to Specific GloServ Queries

I describe different protocols for ways various users can connect to the context-aware agents and search for services. There are two different context-aware query cases, as I mentioned in Section 13.1. In the first case, the user has a set of rules which translate to a set of services. When the rule conditions are met, it sends this rule to the context-aware agent and after a few more message exchanges, receives a specific GloServ query. The second case assumes that the user has context information but does not know what type of service he needs. The context-aware agents must reason what the best service for this user is by matching the user's context attributes to the service context attributes within the context ontology. Below I discuss these two cases.

13.3.1 Creating GloServ Queries With Known Services

In the first case, the user is aware of the services it needs and wants to set up a set of rules to initiate the service requests. In order to do this, a user contacts a context-aware agent and downloads the context attributes of the service class he wants to set rules for.

To illustrate this with a concrete example, I continue with the tourism example. If the tourist already knows that he wants to be notified of *Restaurant*, *Museum* or *Boutique* services while on vacation, he first contacts a context-aware agent and submits these service classes to it in the following message:

```
SERVICE
Restaurant
Museum
Boutique
```

The first agent contacted finds the appropriate agents that handle these service contexts with the methods mentioned in section 13.2.2. The agents handling the services restaurant, museum and boutique, collect the context information from the user. They also query GloServ for the actual service attributes. Thus if the user submits the service *Restaurant*, the agent handling the restaurant service will find the following attributes:

Context Attributes: location, time, activity

Service Attributes: hasCuisine, hasLocation, hasRating

The agents then present the user with a form that allows him to fill out the appropriate values of each set of attributes.

The agent receives this information, creates a restricted subclass with the specified context attributes and classifies this within its ontology. This allows the agent to keep track of context history of a particular service class since its restricted subclasses represent context combinations that future users can benefit from. The agent then creates a GloServ query with the restrictions specified by the user, instantiates it within the restricted subclass created, and returns the query to the user. The restricted subclass is formed in OWL as follows:

```
<owl:Class rdf:ID="contextClass1">
<rdfs:subClassOf rdf:resource="#RestaurantContext"/>
<owl:equivalentClass>
 <owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
   <owl:Restriction>
    <owl:onProperty rdf:resource="#activity"/>
    <owl:someValuesFrom>
     <owl:Class>
      <owl:intersectionOf rdf:parseType="Collection">
       <owl:Class rdf:about="#Vacation"/>
       <owl:Class rdf:about="#Walking"/>
      </owl:intersectionOf>
     </owl:Class>
    </owl:someValuesFrom>
   </owl:Restriction>
   <owl:Restriction>
    <owl:onProperty rdf:resource="#location"/>
    <owl:someValuesFrom rdf:resource="#NewYorkCity"/>
   </owl:Restriction>
```

```
    <owl:Restriction>
     <owl:onProperty rdf:resource="#time"/>
     <owl:someValuesFrom rdf:resource="#OneAM"/>
    </owl:Restriction>
   </owl:intersectionOf>
  </owl:Class>
 </owl:equivalentClass>
</owl:Class>
```

The user then modifies his rule so that when his context conditions are met, a query is sent directly to GloServ. We represent the rule in a generic form which represents a case-based reasoning:

$$location(?x, ?y) \wedge time(?x, day) \wedge activity(?x, walking) \wedge activity(?x, tourism)$$
$$\Rightarrow Query(Restaurant, \text{“}(hasCuisine\ some\ Italian) \wedge (hasLocation\ some\ ?y)'')$$
$$location(?x, ?y) \wedge time(?x, day) \wedge activity(?x, walking) \wedge activity(?x, tourism)$$
$$\Rightarrow Query(Museum, \text{“}(hasStyle\ some\ Art) \wedge (hasLocation\ some\ ?y)'')$$
$$location(?x, ?y) \wedge time(?x, day) \wedge activity(?x, walking) \wedge activity(?x, tourism)$$
$$\Rightarrow Query(Boutique, \text{“}(hasItems\ some\ Clothing) \wedge (hasLocation\ some\ ?y)'')$$

Figure 13.3 illustrates the exchange of messages between the user and the agent.

13.3.2 Creating GloServ Queries With Unknown Services

As the agents instantiate restricted context classes, they collect past user experiences for future users who do not know what services to search for but know their context information. In the case of the tourist, a user may not know what type of services are available to him, but he is aware of certain context conditions and wants to know the service classes pertaining to him.

The user sends the agent a message similar to the one above, except instead of putting in service classes, it specifies a list of context conditions such as: city=Paris, weather=warm,

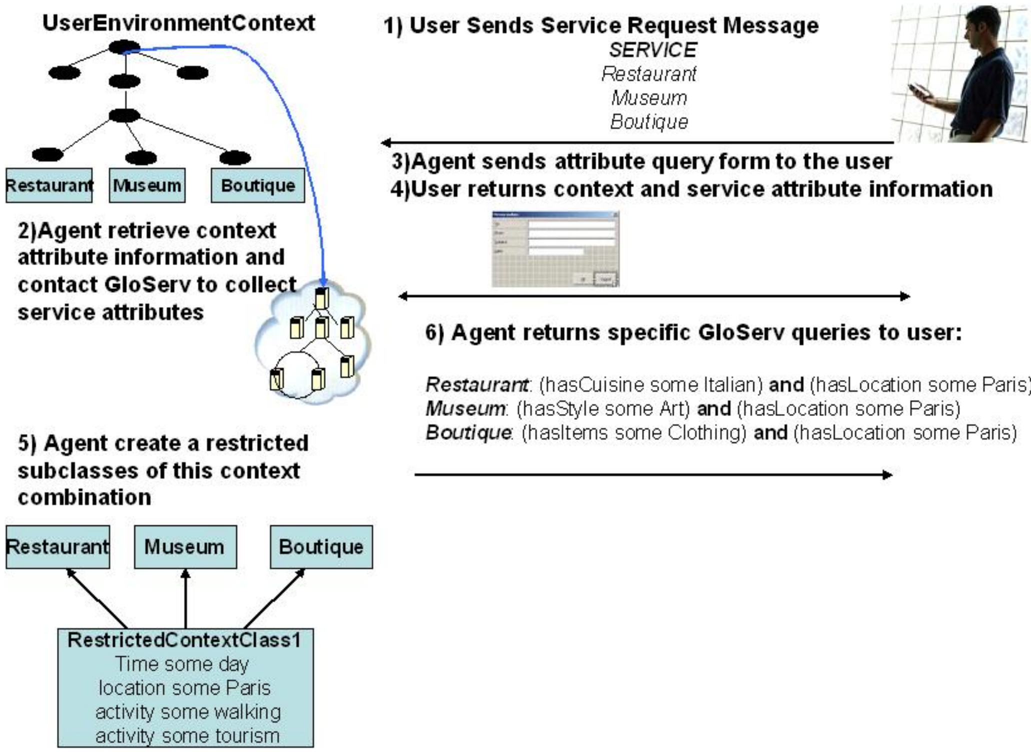

Figure 13.3: One scenario of context-aware service discovery

time=noon, and ranks these attributes by ordering them. The agent uses the attribute thesaurus ontology to match these context attributes to the actual ones in the system. The agent then creates a restricted subclass and since the first level in the ontology is restricted by the rank of each attribute, the restricted class gets classified under one of the three subclasses of the *Context* class. The agent then sends the user's information to these agents and the message continues propagating to the agents below until there is a match. A hit occurs within an agent when the restricted class is classified as either an equivalent class or superclass of one or more of the classes within the agent's ontology; this signifies that there is a context combination that is a subclass or equivalent class to the user's context combination. Once a hit occurs, the agent knows the class of services that the user may be interested in and responds to the user with a query form specific to the service class. The steps following are similar to the previous scenario's.

13.4 Related Work for Context-Aware Service Discovery

Current work done in context-aware service discovery concentrates either on local area networks or uses simple representations of context and services. Cobra [67] uses OWL to represent context and collects context from different sources and reasons with rules and policies for user preferences. However, it is not a distributed system and the example scenarios are for home environments.

Distributed context-aware systems developed recently, such as [86] and [66], lack in semantically rich representation of data. Grossmann describes a scalable system that collects context information from different sources in [86]. However, it does not describe the distributed nature of the architecture in much detail. It also uses object oriented or an attribute-value model for service description rather than ontologies. Chen describes a system that collects context information in a distributed environment using INS [57] as its service discovery architecture in [66] . However, the INS system also uses simple attribute-value representation of services using XML. The context-aware discovery system presented in this work differs from all of these in that it both distributes context data globally as well as classifies them using ontologies. It also uses GloServ for service discovery which also uses ontologies to describe services and is distributed globally.

13.5 Conclusion and Future Work

In conclusion, I have presented a distributed context-aware agent architecture for global service discovery. Unique characteristics of this architecture is that it aggregates context data in a distributed system using the OWL ontology for richer semantics, in order to learn context history over time. It also uses GloServ to map context attributes to services classes so that users can issue specific context-aware queries.

The design of the distributed context-aware agent architecture is part of our ongoing work. For future work, I plan to implement and evaluate this system by conducting user studies to see how this additional context information benefits the user experience during service discovery.

Part V

Conclusion and Future Work

Chapter 14

Conclusion

14.1 Conclusion

This thesis has described the design, implementation and evaluation of GloServ, an ontology-based global service discovery system. GloServ uses the OWL DL ontology, for classifying service information and mapping it onto a physical hierarchical peer-to-peer network. GloServ operates on a wide as well as local area network and supports the discovery of all types of services as long as they are described in an ontology. Below, I summarize the contributions of this work.

14.1.1 Thesis Contributions

Two aspects of service discovery have been enhanced by GloServ: overlay network architecture and service querying and registration.

Overlay Network Architecture

GloServ has three unique characteristic in its overlay network architecture. First, it uses a service classification ontology to map a hierarchical peer-to-peer network architecture. The architecture is designed in such a way where services of a given class reside in closer logical proximity to one another, resulting in efficient message routing. For location-based services, if the ontology classifies services by domain and by location, this closeness also applies to the physical proximity meaning servers are located near the services they describe.

Second, because GloServ has an underlying peer-to-peer architecture, dynamic services can be handled. As the evaluation showed in Chapter 8, for applications which may require a few million updates per minute in a given location, a 250-node CAN network requires approximately 16 node updates. This is an improvement over replicated networks because GloServ has the added benefit of automatically generating the network.

Third, GloServ provides a generic framework for service aggregation. Since it uses ontologies to represent service data, adding new service classes to the network is simplified. The new service class is classified within the overall hierarchical ontology and the CAN network is generated automatically given that service class's ontology. Thus, with a service classification ontology and a set of servers, a new service domain is added to GloServ.

Querying

GloServ has also improved on how queries are performed for service discovery. Current service discovery systems such as SLP, Jini, or UDDI find exact services matches but are not flexible in finding logically similar matches for services. Also, service search using well-known search engines such as Google Local or Yahoo is limited to only text search which has the opposite problem where a user is not able to query for specifics. Thus, GloServ allows a combination of these two types of searches, which is currently lacking in service discovery systems.

GloServ uses OWL DL to define a service classification ontology. Queries are created as temporary classes within the ontology and mapped to related classes. Equivalent classes and subclasses of the query class are considered *exact* matches and hence service instances in these classes are searched first. For a broader search, a *similar* match is performed which maps the query class to its nondisjoint sibling classes. With this type of service matching, a user is presented with more options to choose from. Furthermore, GloServ combines keyword search with ontology querying. This provides greater flexibility to both service providers and users who need to search for specific terms beyond what the ontology allows.

GloServ also allows service composition across different domains which share a common property. Due to the use of ontologies, properties that have the same semantics but are defined differently can be mapped to each other automatically and be combined in a single query. As the use cases in Chapter 9 shows, a location-based service can represent its location as a region and still be able to interact with other location-based services that define their location as zip codes. This is due to a common location ontology which defines location in terms of region and zip codes and is shared across the different domains. Additionally, service composition can now be performed in a global network and thus improves on current web service composition systems which do not scale because of the architectural limitations of UDDI.

The results in Chapter 8 have shown that the bulk of the query processing time for a given GloServ query lies in ontology reasoning. However, this applies only to the GloServ supernodes that process the ontology query. It also applies to new queries because common queries are cached and thus do not require ontology reasoning. For the average case, a unique ontology query takes 30 ms to process. However, with caching optimization, the reasoner is not run and query processing time drops to a few milliseconds. The performance for cached queries is very fast, around 1 or 2 ms.

The bulk of the time for the overall query latency comes from the CAN routing. For a GloServ query which has three 500-node CAN overlay networks amounts to approximately 3 seconds because a 500-node CAN has a query latency of 1 second. However, for an optimized CAN where common queries and their matching servers are cached in the supernodes, the query latency reduces to O(1) hop and the overall latency reduces to 25 ms.

14.2 Implications of this Research

The evaluation of GloServ shows that, although ontology processing is expensive, it does not affect the overall performance for a globalized service discovery system. This is mainly due to the fact that when representing services with ontologies, the number of classes remain limited to a few thousand, as discussed in Chapter 8. Additionally, because the overlay network's distributed architecture, each of the underlying CAN networks remain a manageable size of a few hundred. This thesis has shown that service discovery can be improved by the use of ontologies in a distributed architecture.

14.3 Future Work

The GloServ prototype has been evaluated with test data from the restaurant, movie and weather services which we imported from crawling the Internet. I also randomly generated large amounts of ontology and service data in order to test the performance of GloServ. Although this sufficed in the overall evaluation of GloServ, I would like to deploy GloServ to be used as a real system on the web. In order to accomplish this, real services need to be found and updated. One way of accomplishing this is by crawling the Internet periodically for service data and updating this within GloServ. I would like to start by deploying GloServ for restaurant and movie service discovery and launch it for the broad Internet community to use.

Additionally, I would like to look at the various context-aware applications that can be built on top of GloServ. As discussed in Chapter 13, context-aware agents can interact with GloServ and recommend services to users. Furthermore, I would like to implement and evaluate the design of the context-aware architecture to see how service discovery can be improved by incorporating a user's previous context history.

Finally, there are a number of ways GloServ can be optimized. Since ontologies are used for mapping the network, this information can also be leveraged to manage the network.

Since the ontology shows which servers a given GloServ message is routed to and also has a history of past GloServ messages, there could be a number of ways to improve message routing for future messages. The Knowledge Plane [69] explores the use of schemas for network management. I would like to see how network management techniques can be improved with the use of ontologies.

Chapter 15

Bibliography

[1] North American Industry Classification System http://www.naics.com.

[2] Internet Corporation for Assigned Names and Numbers http://www.icann.org.

[3] 2000 census: US municipalities over 50,000: Alphabetical. http://www.demographia.com/db-2000city50k.htm.

[4] Asp.net ajax. http://www.asp.net/ajax/.

[5] Bittorrent. http://www.bittorrent.com.

[6] CUP LALR parser generator for java. http://www.cs.princeton.edu/ appel/modern/java/CUP/.

[7] DAML: DARPA agent markup language. http://www.daml.org/.

[8] DAML: DARPA agent markup language for web services. http://www.daml.org/services/daml-s/0.7/.

[9] del.icio.us. http://del.icio.us/.

[10] Demographia world urban areas. http://www.demographia.com/db-worldua.pdf.

[11] Extensible Markup Language (XML). http://www.w3.org/XML/.

[12] Facebook. http://www.facebook.com.

[13] FIPA ACL message structure specification. http://www.fipa.org/specs/fipa0061.

[14] FIPA agent management specification. http://www.fipa.org/specs/fipa00023/.

[15] Flickr. http://www.flickr.com.

[16] Foundation of intelligent physical agents. http://www.fipa.org/.

[17] Friendster. http://www.friendster.com.

[18] Google local. http://local.google.com.

[19] Google opensocial. http://code.google.com/apis/opensocial/.

[20] Javascript. http://www.javascript.com/.

[21] Jflex. http://jflex.de/.

[22] JXTA community project. https://jxta.dev.java.net/.

[23] Linkedin. http://www.linkedin.com.

[24] Menu pages. http://www.menupages.com.

[25] Michelin travel guide. http://www.michelintravel.com/.

[26] Myspace. http://www.myspace.com.

[27] Napster. http://www.napster.com.

[28] New york times. http://www.nytimes.com.

[29] OWL-S semantic markup for web services. http://www.w3.org/Submission/OWL-S/.

[30] Owl: Web ontology language. http://www.w3.org/2004/OWL/.

[31] Permanent reference document se.23: Location based services. http://www.gsmworld.com/documents/lbs/se23.pdf.

[32] PHPs hypertext processor. http://www.php.net/.

[33] RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/.

[34] Resource Description Framework (RDF): W3C semantic web activity. http://www.w3.org/RDF.

[35] Simple object access protcol. http://www.w3.org/TR/soap/.

[36] Skype. http://www.skype.com.

[37] Smarty template engine. http://www.smarty.net.

[38] Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/.

[39] Web services architecture. http://www.w3.org/TR/ws-arch/.

[40] Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

[41] World wide web consortium (w3c). http://www.w3.org/.

[42] Yahoo! yellow pages. http://yp.yahoo.com.

[43] Yelp. http://www.yelp.com.

[44] Zagat survey. http://www.zagat.com/.

[45] UDDI technical white paper. white paper, uddi (universal description, discovery and integration). Technical report, OASIS, September 2000.

[46] J. Abasolo and M. Gomez. MELISA: An ontology-based agent for information retrieval in medicine. *Proceedings of ECDL 2000 Workshop on the Semantic Web*, September 2000.

[47] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth. A framework for semantic gossiping. In *ACM SIGMOD: Special Interest Group on Management of Data*, Madison, Wisconsin, 2002.

[48] G. Antoniou and D. Plexousakis G. Flouris. Evolving ontology evolution. In *SOFSEM 2006: Theory and Practice of Computer Science*, pages 14–29. Springer Berlin / Heidelberg, 2006.

[49] K. Arabshian, C. Dickmann, and H. Schulzrinne. Service composition in an ontology-based global service discovery system. Technical report, Columbia University, New York, NY, September 2007.

[50] K. Arabshian and H. Schulzrinne. An ontology-based hierarchical peer-to-peer global service discovery system. *Journal of Ubiquitous Computing and Intelligence (JUCI)*, 2:133–144, December.

[51] K. Arabshian and H. Schulzrinne. Gloserv: global service discovery architecture. *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, pages 319–325, Aug. 2004.

[52] K. Arabshian and H. Schulzrinne. Combining ontology queries with key word search in service discovery. In *ACM/IFIP/USENIX 8th International Middleware Conference*, Newport Beach, California, 2007.

[53] K. Arabshian, H. Schulzrinne, D. Trossen, and D. Pavel. Gloserv: Global service discovery using the owl web ontology language. In *The IEE International Workshop on Intelligent Environments*, University of Essex, Colchester, UK, June 2005.

[54] Knarig Arabshian and Henning Schulzrinne. Distributed context-aware agent architecture for global service discovery. In *The Second International Workshop on Semantic Web Technology For Ubiquitous and Mobile Applications (SWUMA)*, Trentino, Italy, August 2006.

[55] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[56] H. Balakrishnan, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Looking up data in p2p systems. *Commun. ACM*, 46(2):43–48, 2003.

[57] M. Balazinska, H. Balakrishnan, and D. Karger. INS/Twine: A Scalable Peer-to-Peer Architecture for Intentional Resource Discovery. In *International Conference on Pervasive Computing*, Zurich, Switzerland, August 2002.

[58] F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA-compliant agent framework. In *Proceedings of Practical Application of Intelligent Agents and MultiAgents*, London, UK, 1999.

[59] F. Bergenti and A. Poggi. Leap: a FIPA platform for handheld and mobile devices. In *Agent Theories, Architectures, and Languages (ATAL)*, Seattle, Washington, 2001.

[60] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): generic syntax. RFC 2396, Internet Engineering Task Force, August 1998.

[61] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): generic syntax. RFC 3986, Internet Engineering Task Force, January 2005.

[62] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[63] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[64] C. Bussler, D. Fensel, and A. Maedche. A conceptual architecture for semantic web enabled web services. *ACM SIGMOD Rec.*, 31(4):24–29, 2002.

[65] D. Chakraborty, F. Perich, S. Avancha, and A. Joshi. Dreggie: Semantic service discovery for m-commerce applications. In *Workshop on Reliable and Secure Applications in Mobile Environment, 20th Symposiom on Reliable Distributed Systems*, 2001.

[66] G. Chen and D. Kotz. Context-sensitive resource discovery. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, Fort Worth, Texas, 2003.

[67] H. Chen, T. Finin, and A. Joshi. Semantic Web in in the Context Broker Architecture. In *Proceedings of the Second Annual IEEE International Conference on Pervasive Computer and Communications*, Orlando, Florida, 2004.

[68] H. Chen, A. Joshi, and T. Finin. Dynamic service discovery for mobile computing: Intelligent agents meet Jini in the aether. *Cluster Computing*, 4(4):343–354, 2001.

[69] D. Clark, C. Partridge, J. Ramming, and J. Wroclawski. A knowledge plane for the internet. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–10, 2003.

[70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 26(1):64–69, 1983.

[71] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Mobile Computing and Networking*, pages 24–35, 1999.

[72] A. Dey. Understanding and using context. In *Personal and Ubiquitous Computing, Vol. 5, No. 1*, 2001.

[73] A. Dey, G.D. Abowd, and D. Salber. A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. In *Human-Computer Interaction, 16(2-4):97-166*, 2001.

[74] D. Eastlake and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[75] M. Ehrig and Y. Sure. Ontology mapping - an integrated approach. In *Proceedings of the First European Semantic Web Symposium, ESWS 2004*, Heraklion, Greece, 2004.

[76] A. Arkin et al. Web services choreography interface WSCI, 2002.

[77] F. Curbera et al. Business process execution language for web services, 2002.

[78] D. Fensel and C. Bussler. The web service modeling framework WSMF. *Electronic Commerce Research and Applications*, 1(2):113–137, 2002.

[79] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[80] R. Fikes, P. J. Hayes, and Ian Horrocks. Owl-ql - a language for deductive query answering on the semantic web. *J. Web Sem.*, 2(1):19–29, 2004.

[81] UPnP Forum. UPnP hdevice architecture 1.0. Technical report, December 2003.

[82] T. Gardiner, I. Horrocks, and D. Tsarkov. Automated benchmarking of description logic reasoners. In *Proc. of the 2006 Description Logic Workshop (DL 2006)*, volume 189, 2006.

[83] J. Gennari and et al. Evolution of protege: An environment for knowledge-based systems development. Tehcnical report, Stanford University, 2002.

[84] Gnutella. http://gnutella.wego.com.

[85] S. Gribble, M. Welsh, J. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. Katz, Z. Morley Mao, S. Ross, and B. Zhao. The Ninja architecture for robust internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[86] M. Grossmann, M. Bauer, N. Honle, U. Kappeler, D. Nicklas, and T. Schwarz. Efficiently managing context information for large-scale scenarios. In *Third IEEE International Conference on Pervasive Computing and Communications*, Kauai Island, Hawaii, 2005.

[87] T. Gruber. Ontology. *Encyclopedia of Database Systems*, 2001.

[88] T. R. Gruber. A translation approach to portable ontology specifications. In *Knowledge Acquisition*, volume 5, pages 199–220, 1993.

[89] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.

[90] V. Haarslev and R. Moller. Racer user's guide and reference manual version 1.7.19. Technical report, 2004. http://www.sts.tu-harburg.de/ r.f.moeller/racer/racer-manual-1-7-19.pdf.

[91] M. Hattori, K. Cho, A. Ohsuga, M. Isshiki, and S. Honiden. Context-aware agent platform in ubiquitous environments and its verification tests. *Systems and Computers*, 35(7):13–23, 2004.

[92] Bin He, Mitesh Patel, Zhen Zhang, and Kevin Chen-Chuan Chang. Accessing the deep web. *Commun. ACM*, 50(5):94–101, 2007.

[93] T. Berners-Lee J. Hendler and O. Lassila. The semantic web. 284(5):34–43, May 2001.

[94] M. Horridge, A. Rector, N. Drummond, H. Knublauch, and H. Wang. A user oriented owl development environment designed to implement common patterns and minimise common errors. In *3rd International Semantic Web Conference (ISWC2004)*, Hiroshima, Japan, Nov 2004.

[95] Latifur Khan, Dennis McLeod, and Eduard Hovy. Retrieval effectiveness of an ontology-based model for information selection. *The VLDB Journal*, 13(1):71–85, 2004.

[96] T. Kindberg and J. Barton. A Web-based nomadic computing system. *Computer Networks (Amsterdam, Netherlands: 1999)*, 35(4):443–456, 2001.

[97] M. Klein and N. Noy. A component-based framework for ontology evolution. Technical report, Department of Computer Science, Vrije Universiteit Amsterdam, March 2003.

[98] A. Loser, W. Siberski, M. Wolpers, and W. Nejdl. Information integration in schema-based peer-to-peer networks. In *Proceedings of the Conference on Advanced Information Systems Engineering*, Klagenfurt, Austria, June 2003.

[99] L. Mackert and G. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, pages 149–159, Kyoto, Japan, 1986.

[100] P. Maymounkov and D. Mazi. Kademlia: A peer-to-peer information system based on the xor metric. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, London, UK, 2002.

[101] Sun Microsystems. Jini architectural overview. Technical report, 1999.

[102] P. Mishra and M. Eich. Join processing in relational databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.

[103] R. Moats. URN syntax. RFC 2141, Internet Engineering Task Force, May 1997.

[104] P. Mockapetris. Domain names - implementation and specification. RFC 1035, Internet Engineering Task Force, November 1987.

[105] R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *ACM SIGCOMM*, San Diego, CA, 2001.

[106] Hans-Michael Muller, Eimear E. Kenny, and Paul W. Sternberg. Textpresso: An ontology-based information retrieval and extraction system for biological literature. *PLoS Biology*, 2, 2004.

[107] N. F. Noy, A. Chugh, W. Liu, and M. A. Musen. A framework for ontology evolution in collaborative environments. In *International Semantic Web Conference*, pages 544–558, 2006.

[108] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic matching of web services capabilities. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333–347, London, UK, 2002. Springer-Verlag.

[109] P. Plessers, O. Troyer, and S. Casteleyn. Understanding ontology evolution: A change detection approach. *Web Semantics*, 5(1):39–49, 2007.

[110] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, San Diego, CA, 2001.

[111] O. Ratsimor, V. Korolev, A. Joshi, and T. Finin. Agents2go: an infrastructure for location-dependent service discovery in the mobile electronic commerce environment. In *Workshop Mobile Commerce*, pages 31–37, Rome, Italy, 2001.

[112] A. Rector. Modularisation of domain ontologies implemented in description logics and related formalisms including OWL. In *2nd International Conference on Knowledge Capture (K-CAP)*, Sanibel Island, FL, 2003.

[113] J. Rosenberg, Henning Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: session initiation protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[114] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, 2001.

[115] M. Satyanarayanan. Pervasive computing: Vision and challenges, August 2001.

[116] B. Schilit and M. Theimer. Disseminating active map information to mobile hosts. In *IEEE Network, 8(5):22-32*, 1994.

[117] M. Schlosser, M. Sintek, S. Decker, and W. Nejdl. Hypercup – hypercubes, ontologies and efficient search on p2p networks. In *Workshop on Agents and P2P Computing*, Bologna, Italy, 2002.

[118] E. Sirin and B. Parsia. Sparql-dl: Sparql query for owl-dl. Technical report, 2007. http://pellet.owldl.com/papers/sirin07sparqldl.pdf.

[119] E. Sirin, B. Parsia, B. Cuenca Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semant.*, 5(2):51–53, 2007.

[120] K. Sivashanmugam, K. Verma, and A. Sheth. Discovery of web services in a federated registry environment. In *ICWS*, pages 270–278, 2004.

[121] D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[122] K. Verma, K. Sivashanmugam, A. Sheth, A. Patil, S. Oundhakar, and J. Miller. Meteor-s wsdi: A scalable p2p infrastructure of registries for semantic publication and discovery of web services. *Inf. Technol. and Management*, 6(1):17–39, 2005.

[123] M. Wahl, T. Howes, and S. Kille. Lightweight directory access protocol (v3). RFC 2251, Internet Engineering Task Force, December 1997.

[124] C. Weider and P. Deutsch. A vision of an integrated internet information service. RFC 1727, Internet Engineering Task Force, December 1994.

[125] M. Weiser. The computer for the 21st century. In *Human-computer interaction: toward the year 2000*, pages 933–940, San Francisco, CA, USA, 1995.

[126] T. Winograd. Architectures for context. In *Human-Computer Interaction 16(2-4):401-419*, 2001.

[127] Weibin Zhao, Henning Schulzrinne, and Erik Guttman. Mesh-enhanced service location protocol (mSLP). RFC 3528, Internet Engineering Task Force, April 2003.

[128] F. Zhu, M. Mutka, and L. Ni. Splendor: A secure, private, and location-aware service discovery protocol supporting mobile services. In *PERCOM '03: Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 235, Dallas-Fort Worth, TX, USA, 2003. IEEE Computer Society.

Part VI

Appendix

Chapter 16

Glossary Of Terms

combined ontology query - an ontology query that includes more than one service. It contains three parts: *primary query*, *nested subquery* and *shared property*.

GloServ - an ontology-based global service discovery system

keyWord ontology query - an ontology query that also includes a list of key words for each property. A match occurs when the ontology query is matched to a list of classes and then results are further filtered by key word matching for each property stated.

nested subquery - the second part of a combined ontology query. This query is issued from a CAN GloServer.

ontology - a vocabulary that describes objects and the relations between them in a formal way. It is a specification of a conceptualization.

ontology query - a first-order predicate logic query statement that is matched to a set of OWL classes and routed through the network.

primary query - the initial query in a combined query where the query is initially routed to.

primitive skeleton - a pure hierarchical ontology that defines each class to be disjoint from its siblings.

property - attribute or feature of a class.

registration - service data stored within the GloServ network. A registration is an instantiation of an ontology class and stored in a database back-end. It is represented as a first-order predicate logic query statement.

related ontology query - a first-order predicate logic query that obtains not only exact matches to the query but similar ones as well. These include the non-disjoint *sibling* classes of the matched classes.

shared property - the third part of a combined ontology query which indicates the common property between the service classes. A join is performed on this property.

Chapter 17

GloServ Library

This appendix gives an overview of the main classes of the GloServ library. This is in no way a complete list. However, throughout the thesis, I have referred to these classes and I list them here in an organized fashion.

17.1 Back-end GloServ Library

The GloServ Java library for the back-end service discovery system.

17.1.1 package edu.cs.columbia.gloserv.can

The classes below describe CAN related functionality.

CANNetwork: The CANNetwork class handles a set of methods which instantiates the CAN network for the first time. It invokes methods that creates the service classification OWL file and parses it in order to create the dimensions for the CAN. It also reads from a file, a list of available CAN servers for that service class and invokes remote joins into a newly created CAN network. The initial construction of the CAN network creates a balanced network where each node is separated into equal number of keys for each dimension.

CANNode: The CANNode class handles the CAN logic for a single node. Functionality includes methods for joining or leaving the CAN, processing and routing queries and registration messages.

CANServer: The CANServer class implements a listening thread for all incoming CAN messages

17.1.2 package edu.cs.columbia.gloserv.ontology

The classes below describe the ontology related functionality.

GloServOWLModel: The GloServOWLModel class handles all functionality related to the service classification OWL ontology. The ontology describes the service classes the node handles as well as the neighboring host information for each dimension. The GloServOWLModel class includes the methods that converts classes to $\langle dimension, key \rangle$ values and finds neighboring hosts or finding the dimension to split. It also has routine OWL functions such as getting or setting properties, creating classes.

OWLOntologyGenerator: The OWLOntologyGenerator class handles functionality for creating an OWL ontology file given a set of configuration files.

CANOWLOntologyGenerator: The CANOWLOntologyGenerator class converts a given ontology file into a CAN ontology file which includes a *Dimension* class which assigns each restricted class in the main service class to $\langle dimension, key \rangle$ values.

17.1.3 package edu.cs.columbia.gloserv.queryparser

The classes below describe GloServ query parsing functionality.

QueryParser: The QueryParser class handles all types of query parsing, including single ontology query, combined query and keyword query.

QueryStatement: The QueryStatement class is an interface for different types of query statements such as *SomeStatement*, *AndStatement*.

17.1.4 package edu.cs.columbia.gloserv.racer

The classes below describe functionality for accessing the Racer description logic reasoner:

RacerQueryClassifier: The RacerQueryClassifier class defines functionality for issuing an ontology query to Racer.

RacerClient: The RacerClient class defines a number of Racer commands such as ontology classification, and retrieving equivalent, subclass and superclass relationships of a given ontology query.

17.1.5 package edu.cs.columbia.gloserv.messages

The classes below describe GloServ message handling functionality.

IGloServMessage: The IGloServMessage class is an interface for all GloServ messages.

RemoteRequestJoinMessage: The RemoteRequestJoinMessage class defines the RemoteRequestJoin message during CAN network generation.

RegisterMessage, *QueryMessage*: The RegisterMessage class and QueryMessage class define the GloServ ontology query and registration messages.

InternalRegisterMessage, *InternalQueryMessage*: The InternalRegisterMessage and InternalQueryMessage class define an GloServ internal query and registration message

17.1.6 package edu.cs.columbia.gloserv.instance

The classes below describe functionality for service instance processing.

DBUtil: The DBUtil class contains the methods for creating the tables during the ontology to database mapping. It also contains utility methods for connecting to the database and executing queries.

IInstanceProcessor: The IInstanceProcessor class interface contains the methods *queryInstances* and *registerInstances*.

RacerInstanceProcessor, *DBInstanceProcessor*: The RacerInstanceProcessor and DBInstanceProcessor classes implement *IInstanceProcessor* and perform the ontology and database instance processing respectively.

17.1.7 package edu.cs.columbia.gloserv.frontend

The classes below describe the frontend interface classes to GloServ.

QueryAPI: The QueryAPI class issues queries to GloServ. The query is sent to a super node of the GloServ CAN Network, which handles the desired service class GloServ will route the queries and the GloServ nodes contacted will send individual answers.

SyncQueryAPI: The SyncQueryAPI class extends the QueryApi to collect the individual answers and deliver one coherent answer Provides a synchronous interface. The call to issueQuery() is blocked until all parts of the answer are received.

QueryResultServerThread: The QueryResultServerThread class is a listening thread which collects all the query results and returns it to the user.

ExternalAPI, FrontendServerThread, *FrontendServerConnectionThread*: The ExternalAPI, FrontendServerThread and FrontendServerConnectionThread classes provide an XML interface users can query for any kind of service class Users can retrieve ontologies for service classes ExternalApi uses SyncQueryApi internally to perform its tasks

17.2 PHP Web-based Front-end

This section describes the PHP classes which implement the Web-based front-end interface.

GloServ : The GloServ class represents one service that is read from an OWL file. The class provides methods to read the ontology information, as well as methods that construct GloServ compatible query strings. It also provides methods to process instance lists received from the GloServ backend.

GloServAPI: The GloServAPI class defines a low level API to the GloServ backend. It communicates with GloServ through a XML over socket interface. It can be used for issuing query and registration messages.

GoogleMaps: The GoogleMaps class provides helper functions for displaying Google markers for location-based service query results. It also defines functionality for displaying combined services displaying markers for corresponding services and graying out the rest.

ServiceManager: The ServiceManager class defines a simple ontology mapping tool. It is a wrapper class for managing multiple services that are chained together. It initializes the GloServ class instances for all the services by parsing their ontology files. It then determines the relevant relationships between the services based on the hard coded relationship table.

GloServAjax: The GloServAjax class defines methods for a collaborative user interface for performing collaborative search between two or more users.

GloServDB: The GloServDB class defines methods for storing the query information of the collaborative users.