Service Composition in a Global Service Discovery System

Knarig Arabshian, Christian Dickmann and Henning Schulzrinne
Department of Computer Science,
Columbia University,
New York NY 10027, USA
knarig@cs.columbia.edu,mail@christian-dickmann.de,hgs@cs.columbia.edu

***Abstract*—GloServ is a global service discovery system which aggregates information about different types of services in a globally distributed network. GloServ classifies services in an ontology and maps knowledge obtained by the ontology onto a scalable hierarchical peer-to-peer network. Since services are described in greater detail, due to the ontology representation, queries are matched semantically. In this paper, we describe an enhancement to the GloServ querying mechanism which allows GloServ servers to process and issue subqueries between servers of different classes. Thus, information about different service classes may be queried for in a single query, creating an extensible platform for service composition. The results are then aggregated and presented to the user such that services which share an attribute are categorized together. We have built and evaluated a location-based web service discovery prototype which demonstrates the flexibility of service composition in GloServ and discuss the design and evaluation of this system. Keywords: service discovery, ontologies, OWL, CAN, peer-to-peer, web service composition**

I. Introduction

As more services become available and context-aware applications and ubiquitous computing becomes commonplace, service discovery in a wider area network is necessary and network scaling becomes an issue. The proliferation of services also creates the problem of performing more sophisticated query matching such as giving one the option of searching for exact and similar matches to a query or allowing one to query for a combination of different services in a single search. Additionally, services may be dynamic in nature which requires the service discovery system to handle frequent service updates. Currently, service discovery systems do not scale well and are limited to local area networks. They also use simple attribute-value pair matching in order to discover services, which limits the results only to exact matches.

In order to address these problems, we have developed GloServ [7], a global service discovery system, which uses the Web Ontology Language Description Logic (OWL DL) [3] to classify services in an ontology and map knowledge obtained by the ontology onto a hierarchical peer-to-peer network. It operates in wide as well as local area networks and supports a large range of services that are aggregated and classified in ontologies. A partial list of these services include events-based, physical location-based, communication, e-commerce or web services. Organizing services in an ontology and searching within that ontology allows searching for general categories of services and then specializing to specific services.

These attributes make GloServ a very good candidate for context-aware applications. Such context-aware applications need to have access to both ubiquitous as well as pervasive information, thus revealing the need for a globally scalable system. The underlying peer-to-peer architecture provides an efficient system for global distribution of services that may also be dynamic in nature. We envision all types of services to be handled within GloServ, including those that need real-time service description updates. For example, restaurants may want to update their available seating every 15 minutes during peak hours. Hence, a peer-to-peer system provides an efficient means of performing frequent writes in addition to reads.

Furthermore, GloServ aggregates all types of services into one system. Since the high-level service classes are organized in an ontology, adding a new service class into the system simply reduces to adding a class to the ontology and deploying the servers necessary for handling the given service class. The network is then automatically generated via the information within the ontology. This attribute of GloServ provides great deal of generality and ease of construction of services within the network, allowing third parties who want to add their service class to GloServ the ability to do so by submitting an ontology for the service class and deploying a pool of servers. Ontologies for a given service class will be designed by experts within that domain. Since these ontologies represent high-level service classes, all services within a given class will share common attributes. For example, the restaurant service class shares the location, cuisine and price range attribute across the globe. However, if service providers need to specify further information, we have extended GloServ so that key words can be inserted within the ontology and a combination of text and ontology querying can be done [8].

The main contribution of this paper is describing an enhancement to GloServ which allows different types of services to be queried for in a single search. Service composition involves, especially for web services, involves consolidating multiple services into a single composite service. Currently service composition is done by issuing multiple queries to various entities. In order to compose several services in one query, systems exist which handle a limited set of closely related service classes. For example, when searching for travel

services, sites such as Expedia or Priceline provide flight, hotel and car rental information in one query. However, if one wants to search for classes that are indirectly related to travel, such as restaurants or theatres in the cities they are traveling to, this is not as easy to accomplish automatically in one query.

We have enhanced GloServ to support subquerying between its servers in order to allow services which share common properties to be composed into a single query. With this enhancement, fairly complicated queries can be issued in a single search. An example using location-based services would be searching for a Chinese restaurant in the Upper West Side neighborhood of New York City which also has a nearby theatre playing an action movie. The results will show all Chinese restaurants and their corresponding movie theatres. If one wants to broaden the search to include similar results such as similar restaurants to Chinese, reasoning using the ontology produces results for Korean and Japanese restaurants as well since these are all categorized under Asian cuisine. Another example would be querying for a service and its annotations. 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 an *Annotation* service class which has any number of different rating services such as Zagat 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. Thus, a search can be issued for an Italian restaurant in New York which has been rated excellent by Zagat and by regular users.

Below, we describe the subquerying mechanism in detail and discuss the results of our initial prototype for location-based services. Section II gives background information on ontologies and the GloServ system. The design of the subquerying mechanism is found in Section III. Sections IV and V discusses the implementation and evaluation of the prototype system. Related work is discussed in Section VI and we conclude in Section VII.

II. Background

A. Ontologies

An ontology is defined as a formal, explicit specification of a shared conceptualization [11]. A conceptualization refers to an abstract model of how people think about things in the world. The concepts and relationships are given explicit names and definitions. These are formalized into a specification which is encoded in a logic-based language. Ontologies are meant to be shared across different applications and communities.

An ontology specification includes a number of classes which represent various concepts, similar to how object orientation or the entity-relation model consider classes. Classes have object or datatype properties which can be restricted using existential or universal quantifiers. Classes contain instances or individuals which represent actual data. Description logic ontologies, such as OWL DL, allow class relationships to be inferred based on established relationships within the ontology. Relationships can also be established between classes via logical connectives such as intersection, union, or complement.

The motivation behind using ontologies for service discovery 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 would be something like: “Given a service class, find exact and similar matches to my query”. Service search is evolving to cater to particular 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. Finally, ontologies can 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.

B. GloServ

We give an overview of GloServ’s design in this section but encourage the reader to refer to [7] and [6] for greater detail. The remaining sections of this paper concentrate on the subquerying enhancements made to GloServ which improves upon service composition.

One of the main components of GloServ is the service classification ontology. Although there are many ways to engineer ontologies, we have adopted the modularization approach specified in [14] and [17]. Modularizing ontologies into separate domains allows ontologies to be re-used, maintained and to evolve with flexibility. Modularization is achieved by putting general classes within an ontology in a pure hierarchy where siblings are disjoint from each other. This creates a *primitive skeleton*. Hence, service instances will only be classified within one of the branches. At the lower levels of the ontology, classes may have relationships with other classes and a pure hierarchy is not maintained. The upper hierarchical ontology which defines high-level services is mapped onto a hierarchical network and the low-level ontologies are mapped to a peer-to-peer network.

Another component of GloServ is the back-end hybrid hierarchical peer-to-peer service discovery network. A GloServ server (GloServer) in the hierarchical network represents a

high-level service class within the pure hierarchy of the ontology, described above. The high-level network works similar to DNS except that the network exploits the knowledge obtained by the service classification ontology to establish the hierarchy and route the messages to the correct peer-to-peer network. A benefit of this design is that since the high-level service classes are disjoint, a query will be routed to one of the hierarchical branches, reducing the number of hops a query needs to propagate through.

The lower levels of the network architecture are organized in a peer-to-peer Content Addressable Network (CAN) and also represent the class relationships within the ontology. We describe a novel mapping algorithm in [7] that combines the benefits of OWL DL and CAN to map content of service instances to nodes in a peer-to-peer network. Although there are other types of structured peer-to-peer networks such as Pastry [18] and Chord [19], we have elected to use CAN because it is easily constructed given a service ontology. Organizing similar classes within a peer-to-peer network causes them to lie in closer proximity to each other within the network which makes the similarity searches defined above faster. Thus, in order to achieve load distribution, fast query and update processing time, while maintaining reliability, we have elected to use a hierarchical peer-to-peer network as the underlying service discovery architecture.

GloServers maintain three types of information: a service classification ontology, a thesaurus ontology and, if part of a peer-to-peer network, a CAN lookup table. The high-level service classification ontology is not prone to frequent changes and thus can be distributed and cached across the GloServ hierarchical network. Each high-level service will have a set of properties that are inherited by all of its children. As the subclasses are constructed, the properties become specific to the particular service type. The thesaurus ontology maps synonymous terms of each service to the actual service term within the system. Figure 1 gives an overview of how servers are found in GloServ. Services are represented as instances

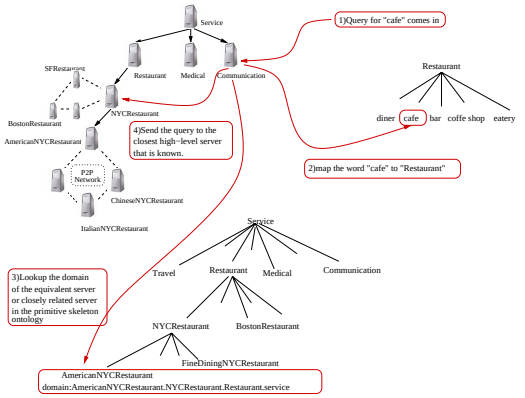

Fig. 1. Finding servers in GloServ

of the service classes and usually reside in the more specific, lower levels of the ontology. Each service instance has a set of properties that are populated. According to the service's attributes, it is classified in a set of related classes within the ontology. Services are registered and queried for either in a user-centric way through a web-based form or in an automated fashion by issuing a first-order predicate logic query. We have implemented the GloServ front-end as a web-based form. The form generated reflects the ontology of the service class. Service providers and users register or query for services by filling in values for each of the attributes of the service. Details of the implementation are given in IV.

At the lower levels, maintaining a purely hierarchical ontology structure becomes difficult as classes tend to overlap. Thus, in order to efficiently distribute service instances according to similar content, servers that hold information on similar classes are distributed in a peer-to-peer network. We employ a CAN peer-to-peer architecture to distribute classes with similar content. The CAN architecture is generated as a network of n-level overlays, where n is the number of subclasses nested within the main class. An example of an ontology classification using the *Restaurant* class and the CAN overlay network generated is seen in Figure 2. 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 maximum number of nodes which can be added into the CAN. This is estimated to be $(\log_2 n)/2$ where n is the number of nodes in the network, to ensure that the number of query hops are $O(\log_2 n)$.

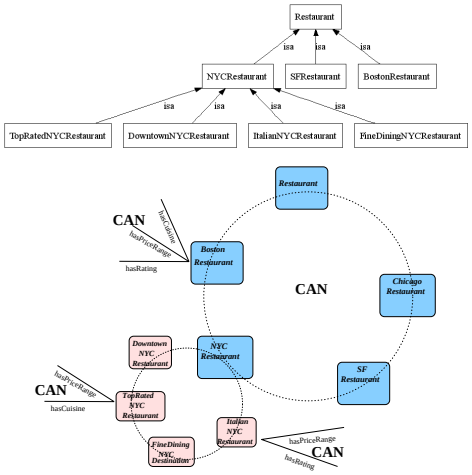

Fig. 2. CAN overlay network

Continuing with the restaurant example, as services register within CAN nodes and instances are created, they are classified into the subclasses of *Restaurant*. When a new CAN node joins the network, one of the CAN dimensions is split into two and data is transfered over 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. According to some criteria, one of these dimensions is chosen and split into two. The current criteria we use is choosing the dimension with

the largest number of keys. However, in the future we will implement network management techniques which keep track of the overloaded servers and split the dimension which has the greatest number of overloaded nodes. 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 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 3 illustrates the joining of four CAN nodes in the network.

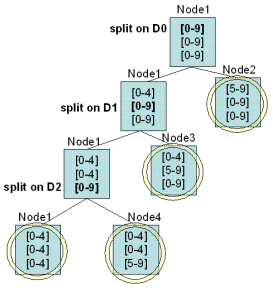

Fig. 3. CAN node splitting into two nodes

A service registration instance is a unique URI which refers to the service's description. Service instances are distributed to all CAN nodes that handle their service classes and stored in a database. Since we are using a CAN distributed hash table, not every node within the system needs to be updated. For queries, when a query is matched exactly, the first matching node will have the complete data set for that particular query restriction and thus further nodes need not be traversed. For a related match query, only the servers that hold logically similar information will be searched. Figure 4 gives a graphical overview of the query propagation in the CAN. We explain the details of ontology querying by looking at the *Restaurant* ontology as our running example.

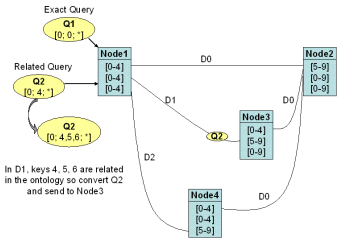

Fig. 4. Query propagation in the GloServ CAN

When a user initially contacts a GloServ user agent and enters a service name, the initial GloServer is found after following the steps outlined in Figure 1. Since each hierarchical node handles a class which is disjoint from its siblings, the query is routed down only one branch reducing the query hops considerably. Once the correct GloServer is contacted, the user agent obtains the ontology pertaining to that service class. The interface to the user can either be human-centric or automated, depending on the implementation. In either case, a query is formed and sent to the GloServer. The query is a first order predicate logic statement that contains restrictions on various properties such as: ($hasLocation$ **some** `NYC`) and ($hasCuisine$ **some** (`Korean` **or** `Chinese`)). The server handling that service class creates a class with this query restriction and classifies it in its ontology. Since the subclasses of the *Restaurant* class are restricted by location, the query class gets classified as a subclass of the *NYCRestaurant* class. The query is then forwarded to the nodes that handle *NYCRestaurant* classes. When a node is found, the query class is classified again. Since the *NYCRestaurant* class has subclasses that have cuisine restrictions, when the reasoner classifies the query class, it becomes a subclass of *NYCRestaurant* and a superclass of *KoreanNYCRestaurant* and *ChineseNYCRestaurant* classes. The classification indicates that the query must be routed to the servers handling Chinese and Korean restaurants. In order to route the query within a CAN, the query needs to reduce to a dimension and key. We use the dimension and key values assigned to each of these classes during the CAN network generation to convert the ontology class to a $< dimension, key >$ pair

We illustrate ontology querying with the following example. Let us assume we have a *NYCRestaurant* ontology that has 30 subclasses, separated into 3 dimensions with 10 subclasses in each dimension. Each subclass is assigned a unique key. Thus, keys are separated into [0-9], [10-19] and [20-29] for dimensions 1, 2 and 3 respectively and a class is represented by its $< dimension, key >$ pair. Furthermore, the *ChineseNYCRestaurant* subclass is assigned to dimension 0 with key 0 and *KoreanNYCRestaurant* to dimension 1 with key 0. If a user queries for: ($hasLocation$ **some** `NYC`) and ($hasCuisine$ **some** (`Korean` **or** `Chinese`)) the query message is $[0; 0; *]$. As seen in Figure 4, Node1 receives the query and stops propagating it because it handles these classes.

If a user relaxes her query requirements to not only include equivalent, superclass or subclass relationships but sibling relationships as well, Node1 looks at the sibling classes and issues query messages for each. For example, if the query message $[*; 4; *]$ comes into Node1 where semantic matches to the query are classes that are numbered 4, 5 and 6 in dimension 1, then the query message is converted to $[*; 4, 5, 6; *]$, processed in Node1 and propagated to Node3. A query continues to propagate until the original node is reached. Since a dimension is circular, it is guaranteed that the query will return back to its original position with at most $O(n^{1/d})$ hops. For example, each of these siblings have certain restrictions on various properties. The related query matching algorithm finds properties that are related to the *query* class's properties and looks into the 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 equivalent classes and subclasses of the range are looked into. For instance, the *Cuisine* class has the subclass *Italian* which has subclasses *Pizza* and *Pasta*. When a query comes in for a pizzeria with a five star rating in NYC, the query class will have the following restriction: ($hasLocation$ **some** `NYC`) and ($hasCuisine$ **some** `Pizza`) and ($hasRating$ **some** `FiveStar`). This query class is classified according to how the ontology is constructed. In our ontology, it first gets classified under the *ItalianNYCRestaurant* class. If there are no instances within this class that have a *Pizza* cuisine and *FiveStar* rating, then the related classes of the the *Pizza* class are analyzed. Since the *Pasta* class is related to the *Pizza* class, the query is reformulated to include *Pasta* as the cuisine. Alternately, a user may choose to have related information in the query even if exact ones exist in which case the results given are both exact and related matches.

III. Service Composition in GloServ

A. Motivation

Currently, outside of GloServ, service composition is done mainly within one service class. 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 also include menus, price ranges and ratings for each restaurant and restaurants can be searched for either by location or cuisine. Additionally, the seamlessweb.com site also allows one to place an order at a restaurant for delivery. When searching for movies, sites such as movies.com or fandango.com give a list of movies in a certain location along with ratings and a link to purchase tickets.

There are many other examples of web services, such as these, which allow one to search and invoke a number of different services. However, the main drawback to these systems is that they exist as separate entities on the web. Thus, when one wants to search for a restaurant and a nearby movie theatre, one will need to first go to the restaurant site, find the restaurant and then go to the movie site and search for the theatre near the restaurant's location that is playing a movie that interests him.

Additionally, when searching for a service using Google, the type of search is limited to key words which does not allow specificity or greater reasoning as mentioned above. Thus, besides not being able to search for a combination of services at once, it is also not possible to reason over the data in order to search for similar services. So a restaurant search in local.google.com is limited to search terms such as "chinese restaurant" or "pizza" which returns results with these terms in them, but can not extend the search to include price range, cuisine, or ratings.

GloServ's subquerying extension solves the above problems. Since service information from different classes are aggregated, categorized in an ontology and distributed across a global network, multiple services can be searched for in a single query. The search for services can also be quite sophisticated where one can search for details given the ontology of each service class which can result in exact and similar matches. Thus, a search for Chinese Restaurants can also produce results for Asian restaurants such as Japanese or Korean. We have also added the capability of combining ontology querying with text search [8]. Thus, service providers are not limited to the ontology description but can add their own set of key words as well. Users can then add key words in addition to filling out the ontology form.

Finally, because all these services are distributed in a global network and can be accessed rather quickly, combining more than one service in a given search becomes possible. A few challenges exist in accomplishing this: 1) creating a query language which allows servers to distinguish different parts of the query, namely, the part belonging to its own service class, the parts belonging to other service classes and shared properties between each part; 2) routing the query to the other servers, collecting all the results from the various servers and processing them such that those with matching attributes are displayed to the user in a easy to understand graphical user interface. We describe the design of our solution below.

B. Matching services

Two services can be matched only if they share a common property. Combining two services in one search only makes sense if there is a relationship between them. As all service classes are described in ontologies within GloServ, a relationship between two services means that they share a property. For example, the *Theatre*, *Restaurant* and *Weather* classes share the Location property. Thus, a search can be issued for an outdoor restaurant which has a nearby theatre playing an action movie in a region of NYC which does not show rain in the forecast.

Another example of matching properties in a combined search occurs when searching for annotations for a given service, such as reviews. The matching property will be the InstanceID of a given service. Every service has a unique ID which distinguishes it from others. The *Annotation* class has a service instance ID property as well in order to identify the service being annotated. Thus, a search for a "Japanese restaurant in NYC which has an excellent rating by Zagat" reduces to matching the service instance IDs of the restaurants obtained to the service instance IDs of the annotations.

Additionally, the shared property might have different meanings in the two service classes and the semantic relationship may often be invisible to the ontologies. For example, the Name property of a person could be matched to the Author property of a restaurant review.

To match two services based on a shared property, only the type of the properties in two services classes have to match. For our prototype, we use the Location property and the InstanceID property as the shared properties since these are used to search for location-based services as well as finding annotations on services such as reviews. The Location property is an object property that has the *Location* class as its range. The InstanceID property is a datatype property which

is a string datatype. Because matching is done based on the type of a property, this not only requires, but also promotes one of the fundamental ideas of ontologies, namely that ontologies can be reused and become common "vocabulary". To adhere to our example, the *Location* class is envisioned to be engineered once and then reused by every service class that needs to express a physical location. We have designed this class to be categorized by country, county/state, city. Additionally, there attributes for street, zip, longitude and latitude. While one could argue that this type constraint limits the ability to freely combine service classes, we believe that it has the opposite effect. By providing an incentive to reuse ontologies (i.e., combining services into one query), we promote this very desirable concept.

C. Query Language

Thus far, we discussed the general concept of when service classes can be combined into one query. In the next step, we will elaborate how we extended our basic query language to allow combining services. The query language used is the same as that of Protege's [9], which is an open source development environment for ontologies and knowledge-based systems and this is what we have extended.

The first and most important design decision was to organize the service classes in a chain. Every query has a main service class that it is searching for. In addition it may contain one or more subqueries that relate to a different service class. The main query part is a regular GloServ query. If the user is searching for an Italian Restaurant in New York City, the main query might be: ($hasLocation$ **some** `NYC`) and ($hasCuisine$ **some** `Italian`). Each subquery itself is a regular GloServ query, except that all property names are qualified with the service class the subquery is referring to. If the user is searching for a nearby theatre which is THX certified, the subquery could look like: $Theatre.hasTHX$ **has** `True`.

Besides the main query and the subquery a third part is required that identifies the shared property. To express this relationship we added the equals operator to the querying language which is similar to an SQL join operation. Following the given example, if the user wants the Theatre to be near the Restaurant, the equals expression would be: $Restaurant.hasLocation$ **equals** $Theatre.hasLocation$ and the overall resulting GloServ query would be:
(($hasLocation$ **some** `NYC`) and ($hasCuisine$ **some** `Italian`)) and ($Theatre.hasTHX$ **has** `true`) 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 told apart 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 subqueries will be processed by GloServ.

D. Request Routing

The GloServ message format is similar to HTTP and SIP. Each message contains a list of headers which indicate the method, JOIN (for node joins), QUERY, REGISTER, as well as a payload. A query that is sent to a a CAN network for the first time is labeled as a *userquery* and contains the full logical query statement. This initial *userquery* is classified within the ontology and is then converted into an *internalquery*. The *internalquery* consists of a numeric representation of the matching classes ($< dimension, key >$ pairs described in section II-B) along with the logical query statement. The *internalquery* is routed in the CAN internally so that further classification is avoided.

When combining services, each query part is sent to its corresponding CAN servers. Since the primary query of the combined query is just like any other GloServ query, it is routed and processed like a regular query. From the example above, the query is initially routed to the GloServers which handle the *Restaurant* service class.

Once a GloServer is found which has matching instances, these instances are retrieved but not sent to the user directly. Instead, a new query is constructed, which is related to the service class within the subquery. This new query is derived from the old query in three steps. First, the main query is stripped away. Second, the qualifier prefixes of the subquery are removed, transforming this into a valid main query. Third, and most importantly, the equals expression is resolved 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 Manhattan and Queens were found, the new query for Theatres would look like this: ($hasTHX$ **has** `true`) and (($hasLocation$ **some** `Manhattan`) or ($hasLocation$ **some** `Queens`)) As seen above, this new query is a regular GloServ query without subqueries. The new Theatre query is issued to the GloServ architecture just like any other query, but contains the Restaurant instances in its payload. Thus, it is routed through the Theatre CAN network until Theatre instances are found. Once this happens, a final answer containing both, the Restaurant and matching Theatre instances is sent to the searching user. Figure 5 illustrates how the query is routed across the different CAN networks. If two services are so closely related that users frequently search for them in a combined query, there are two ways to optimize routing in the GloServ architecture. In a first step, the routing table of the Restaurant servers could have direct pointers to the servers in the Theatre network. While this would reduce delays and bandwidth requirements, it would also require greater overhead. If two services are so closely related that the users often combine them and only in rare cases query for just one of them, the two services can be integrated into a single CAN network. While this approach is totally different from the approach presented thus far and is beyond the scope of this document, we implemented it in our prototype to demonstrate the feasibility.

Since the GloServ CAN network is a black box to the user, there is no way of knowing how many answers will be returned from a query. Similarly no GloServ node knows how many

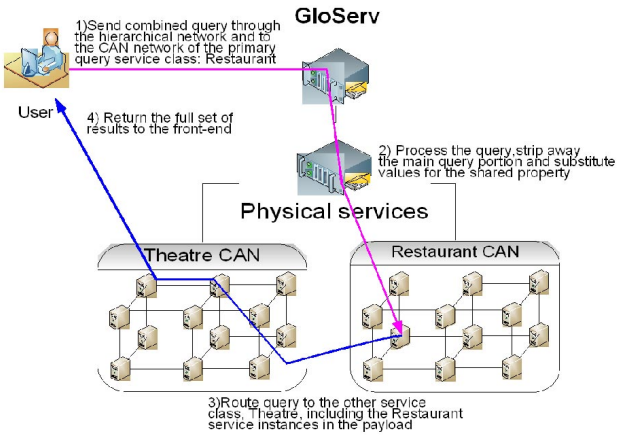

Fig. 5. Subquery routing in GloServ

other nodes a query will traverse once it has left the node. Thus, there is a need for a mechanism to keep track of answers. GloServ implements a simple mechanism to do that job. Every query has a value which could be read as "what percentage of the overall answer does this query cover". The initial query sent by the user amounts for 100% of all answers. Every GloServ node that forwards queries changes this percentage. For example, if the initial query is answered by the first node (i.e., it stores instances) and sent to 4 neighboring nodes in the CAN, the sent answer amounts for 25% of the overall answer and every single query sent to neighboring nodes amounts for 25% of the overall answer. If one of these queries is then split into five 5 queries, each of these new queries will amount for 5% of the overall answer. Using this mechanism, the user can measure "how much" of the overall answer he received at any point in time. This is not an accurate metric in the sense that a value of 50% does not really mean that the user received 50% of the answer messages or even 50% of the instances. However, when the value reaches 100% the user can be completely certain that he has received all answers.

IV. Implementation

We have implemented a prototype of GloServ using Protege and Racer-Pro [13]. Protege is an open-source development environment for ontologies and knowledge-based systems. In order to follow a real-world classification, we have written tools to automatically generate ontologies pertaining to the restaurant classification in *http://www.menupages.com*. The *Restaurant* ontology is modified to represent the CAN lookup table. The subclasses within *Restaurant* are assigned to a unique $\langle dimension, key \rangle$ pair. When a node joins a server, the server's ontology is split across a dimension and transfered over to the new node.

We have built a front-end user interface to GloServ written in PHP. The interface uses Google Maps to display location-based services. The front-end parses the OWL ontology using PHP and automatically generates a form based on the service description given in the ontology. Service providers and users register or query via this form. The ontology also has annotation properties which guides the front-end server in populating the fields within the ontology. Thus, for properties that may also have key word search enabled as mentioned in I, the interface displays a text box underneath the form for the user to enter key words in. The interface is seen in Figure 6 and can be accessed on the web at *http://gloserv.dyndns.org:8080*

The query parser is implemented using JFlex and CUP. JFlex is a lexical analyzer and 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 (Identifier, SomeStatement, Has-Statement, OrExpression, etc.).

Both JFlex and CUP generate Java code. We have defined a QueryParser class which 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 class QueryParser defines a number of methods to perform common tasks on query statements, e.g., Normalization (query is transformed to an OR-ed list of AND-ed statements) or finding nested queries. It also provides methods to extract subqueries and transform them in a way that the subquery can be issued individually. Once the query reaches a CAN GloServer, it is classified within the ontology. For query classification, the query needs to be converted in the Racer syntax. If the query matches the service class of that server, then for instance retrieval, since we store the instances in a database, the query also needs to be converted into an SQL query. We have also implemented a cache in order to avoid the overhead of classification in case the query has already been seen.

V. Evaluation

We have evaluated the querying latency of GloServ for different types of ontologies. Since the ontology is the principal bottleneck of the system, it is essential to determine the ideal ontology size and type needed for optimal performance. The worst case is expected for unique queries where the query needs to be classified using Racer. The best case is when the query has already seen and found in the cache.

One GloServer was run on an IBM Lenovo machine which has an Intel core duo processor (2 GHz each), 1 GB RAM running Windows XP. In order to measure the query latency for different types of ontologies, we modified two aspects of

Fig. 6. GloServ Front-End

the ontology which we suspect will be the two things most modified. First, the size of the ontology was varied. For the Restaurant service class, the shared classes were the *Location*, *Cuisine* and *PriceRange* classes. The *Cuisine* and *PriceRange* classes had a total of 50 classes. We adopted the classification of restaurants used in the menupages.com site and categorized the *Cuisine* and *PriceRange* classes accordingly.

Currently, there are around 600 cities [1] with population greater than 50,000 in the United States and around 1,300 urban areas in the world [2]. Since services are distributed in GloServ where each subnetwork handles a subset of a high-level service class, if we distribute these services by location, then we anticipate that each CAN will handle 100 to 200 locations within a given region. Because of this, a location-based service class will have this many subclasses as well. For example, each subclass of the restaurant class will be restricted by a location, thus doubling the number of location classes. In addition to this, there are auxiliary classes, such as cuisine and price range. Thus, in total, we expect the number of classes to equal $2C_l + C_a$, where C_l is the number of location classes and C_a is the number of auxiliary classes. So in total, the number of classes are expected to be between 250 and 450. We tested the system on 250, 350 and 450 class ontologies.

Additionally, we varied the number of properties that are restricted per subclass within the main class. For example, if each subclass of the restaurant class has a restriction on the location, then this signifies one restriction (O1). However, the restaurant classes may be restricted in more than one property, for example, location and cuisine restrictions (ie., ChineseNYCRestaurant has a restriction for Chinese cuisine and NYC location). Thus, we varied the number of restrictions per class from 1 to 10 restrictions. The query issued for one service class was in the form (A some B or C some D or E some F).

The results in Figures 7 shows that for the simplest ontology, which has 250 classes and one restriction per subclass, O1, query processing took 150 ms and grew sublinearly as the number of classes grew to 450. On the other extreme, for an ontology that had 10 restrictions per subclass, query classification became very slow and grew at a faster rate. Since we expect that ontologies will range in size from 250 to 350 and have between 1 to 3 restrictions, we believe querying time will range from 150 ms to 550 ms. For cached queries and internal queries, processing was much faster at 30 ms for all ontology types since classification was not done in these cases.

From our evaluation, we can see that increasing the ontology class size does not affect the latency as much as raising the number of restrictions per class. Thus, if an ontology is going to be more complicated, requiring greater logical restrictions, then the number of classes should be kept at a minimum. However, if the ontology has one or two restrictions per class, the number of classes can get larger. For optimal query time, it is best to keep both at a manageable level. However, since query classification is done once per CAN traversal, and since a CAN is localized, the query cache will be used often thus decreasing the query latency considerably. Additionally, since these ontologies represent high-level service classes and mimic categorizations seen in yellow-page directories, it is highly unlikely that the number of restrictions per property will become very complicated. Thus, it is safe to assume that the restrictions will average around 3 restrictions per class.

For combined service queries, the value of the the number of service classes combined in the query will be multiplied by the latency of an initial user query. Thus, for a typical restaurant and movie search where each of these ontologies have approximately 350 classes each with 1 to 3 restrictions, the latency will be twice that of a single service query which ranges from 500 ms to 1s. Again, with the use of caching, this value will drop considerably.

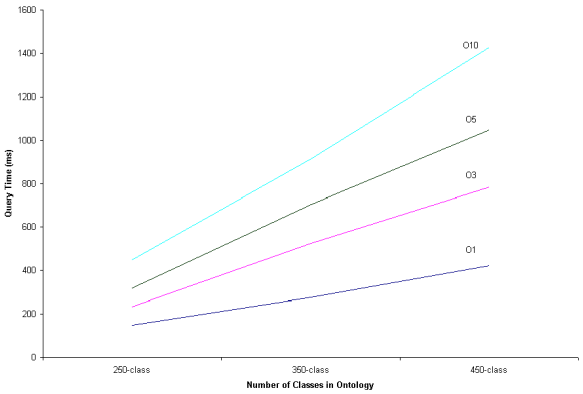

Fig. 7. Query Latency

We will continue to evaluate the system to test the throughput of each node. Currently, Racer blocks for each query. Although it can handle up to 1,500 simultaneous queries, it becomes very slow. We will tweak the system a bit to bypass the blocking in Racer and test the time it takes for all these queries in order to better assess the number of servers necessary for a CAN network to function.

VI. Related Work

A. Service Discovery

Current approaches to service discovery are either centralized or decentralized. Centralized service discovery systems include SLP [12], Jini [16], and UDDI [4]. These systems are limited in network scalability as well as in representing service data.

The existing work closest to our research use schemas to map onto a network [15], [5], [20]. These systems are similar to GloServ in that they use data from a schema or ontology to map onto a network. [5] outlines a semantic gossiping framework that exchanges ontology information within a peer-to-peer network. It uses Gnutella [10] 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.

[15] proposes the HyperCuP, a 2-tier peer-to-peer hierarchy which uses indices within a super-peer topology. The indices are built using schema information from associated peers. Super-peers are connected to each other in a hypercube or Cayley graph and represent disjoint concepts. Underlying peers of a super-peer contain information regarding that particular concept. [20] describes Meteor-S which is also similar to GloServ in that it separates peer-to-peer networks into different domains and represents these services using an ontology. However, the JXTA framework's Peer Discovery Protocol is used which employs flooding techniques for disseminating information and hence does not scale globally.

GloServ differs from these systems by using an ontology to map a multi-tier hierarchical peer-to-peer network. It is similar to [15] and [20] in that a concept represents a subnetwork. However, unlike [15], concepts that are disjoint from each other are hierarchically organized 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 [15], indices refer to whole peers. Also, because of the CAN DHT, GloServ generates $O(\log_2 n)$ query messages whereas in [20] flooding is used which generates $O(n)$ messages which is inefficient on a global scale. Additionally, [5] and [15] describe the peer-to-peer network formation, namely, dealing with nodes entering and leaving the system and concentrate less on describing how the data is disseminated. We, on the other hand, describe the formation of the network as well as describe algorithms that distribute and query the data by mapping the ontology onto a CAN network.

VII. Conclusion

We have discussed the design, implementation and evaluation of service combination in GloServ, an ontology-based hybrid hierarchical peer-to-peer global service discovery mechanism. GloServ is able to register and query for services semantically using a service classification ontology. Since these services are aggregated in a global network, combining different services in a single search becomes possible. Our results show that this is indeed possible as long as the number of classes within the service ontology remains at a manageable size.

References

[1] 2000 census: US municipalities over 50,000: Alphabetical. http://www.demographia.com/db-2000city50k.htm.

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

[3] Owl web ontology language. OWL http://www.w3.org/2004/OWL/.

[4] UDDI technical white paper. white paper, UDDI (universal description, discovery and integration), September 2000. http://www.uddi.org/pubs/.

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

[6] Knarig Arabshian and Henning Schulzrinne. Distributed context-aware agent architecture for global service discovery. *The Second International Workshop on Semantic Web Technology For Ubiquitous and Mobile Applications (SWUMA'06)*, 2006.

[7] Knarig Arabshian and Henning Schulzrinne. An ontology-based hierarchical Peer-to-Peer global service discovery system. *Journal of Ubiquitous Computing and Intelligence (JUCI)*, 2006.

[8] Knarig Arabshian and Henning Schulzrinne. Combining ontology queries with text search in service discovery. Technical report, Columbia University Technical Report CUCS-006-07, January 2007.

[9] J. Gennari, Mark A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubézy, H. Eriksson, N. F. Noy, and S.-C. Tu. Evolution of protégé: An environment for knowledge-based systems development. Technical report, Stanford University, 2002.

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

[11] T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993.

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

[13] Volker Haarslev and Ralph Moller. Racer user's guide and reference manual version 1.7.19. Technical report, Technical University of Hamburg-Harburg, University of Hamburg, 2004.

[14] Matthew Horridge, Alan Rector, Nick Drummond, Holger Knublauch, and Hai Wang. A user oriented owl development environment designed to implement common patterns and minimise common errors. In *3rd International Semantic Web C3onference (ISWC2004)*, Hiroshima, Japan, Nov 2004.

[15] Alexander Loser, Wolf Siberski, Martin Wolpers, and Wolfgang Nejdl. Information integration in schema-based peer-to-peer networks. In *Proceedings of the Conference on Advanced Information Systems Engineering*, June 2003.

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

[17] Alan 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.

[18] Antony Rowstron and Peter 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)*, pages 329–350, Heidelberg, Germany, November 2001.

[19] Ion Stoica, Robert Morris, David R. Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. San Diego, CA, USA, August 2001. ACM.

[20] Kunal Verma, Kaarthik Sivashanmugam, Amit Sheth, Abhijit Patil, Swapna Oundhakar, and John Miller. METEOR-S WSDI: A scalable p2p infrastructure of registries for semantic publication and discovery of web services. *Journal of Information Technology and Management. Special Issue on Universal Global Integration*, 6(1):17–39, 2005.