Combining Ontology Queries with Key Word Search in the GloServ Service Discovery System

Knarig Arabshian
Computer Science Department
Columbia University
New York, NY 10027
knarig@cs.columbia.edu

Henning Schulzrinne
Computer Science Department
Columbia University
New York, NY 10027
hgs@cs.columbia.edu

ABSTRACT

GloServ is a global service discovery system which aggregates different types of services in a globally distributed network. It improves on current service discovery systems by scaling across a globally distributed network and allowing intelligent querying and registration of services. It uses the OWL DL sublanguage of the Web Ontology Language to classify services in an ontology and map knowledge obtained by the ontology onto a scalable hierarchical peer-to-peer network. We present an enhanced novel querying mechanism for service discovery which combines ontology queries with key word search.

Keywords

service discovery, ontologies, OWL, CAN, peer-to-peer, key word search, ontology queries

1. INTRODUCTION

Current service discovery systems use simple attribute-value pair matching in order to discover services, which limits the results only to exact matches. They also do not scale well and are limited to local area networks. However, as more services become available, service discovery in a wider area network is necessary and network scaling becomes an issue. The proliferation of services also creates the problem of finding different relationships between services and performing intelligent query matching. Services may also be dynamic in nature which requires the system to handle frequent service updates.

In order to address these problems, we have developed GloServ [5], a global service discovery system, which uses the Web Ontology Language Description Logic (OWL DL) [1] to classify services in an ontology and map knowledge obtained by the ontology onto a hybrid 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: event-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.

We have elected to use ontologies for service discovery 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 all related matches to my query". Ontologies can also be shared, re-used and changed flexibly so that new relationships between classes can be established easily.

The motivation behind designing a hybrid hierarchical peer-to-peer architecture is to provide an efficient architecture for global distribution of services that may also be dynamic in nature. If data is replicated across servers, large number of frequent updates of all types of services will not scale. We describe a novel mapping algorithm in [5] that combines the benefits of OWL DL and the Content Addressable Network (CAN) DHT [15] 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 [16] and Chord [17], we have elected to use CAN because it is easily constructed given a service ontology.

The main contribution of this paper is allowing the combination of ontology queries with key word search. Currently, querying is limited to either simple attribute-value pair searches, structured ontology queries or key word and text searches. Combining ontology and key word 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 correctness of the results can only be approximated and may or may not be what the user is looking for. Thus, combining these two types of queries allows us to reap the benefits of both mechanisms.

Below, we describe the combined ontology and key word matching mechanism. Section 2 gives an overview of GloServ. Sections 3, 4 and 5 describe the ontology querying, key word matching and implementation respectively. Related work is discussed in Section 6. Finally, we conclude in Section 7.

2. OVERVIEW OF GLOSERV

The GloServ service discovery system achieves large-scale distribution of semantic service data that is queried for with specificity and efficiency, due to its hybrid hierarchical peer-to-peer architecture and ontology service descriptions. These attributes make GloServ a very good candidate for context-aware applications. We give an overview of the design in this section but encourage the reader to refer to [5] for greater detail as the remaining sections concentrate on the querying enhancements made to GloServ.

An ontology is defined as a formal, explicit specification of a shared conceptualization [7]. A conceptualization refers to an abstract model of how people think about things in the world. Thus, an ontology defines a number of classes, properties and the relationships between them. Ontologies are meant to be shared across different applications and communities. There are many ways to engineer ontologies. We have adopted the modularization approach specified in [10]. 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*. At the lower levels of the ontology, classes may have relationships with other classes and a pure hierarchy is not maintained.

The GloServ architecture is designed as a hierarchical peer-to-peer architecture. We use a CAN for the peer-to-peer network. A CAN is a fault-tolerant, scalable and self-organizing distributed peer-to-peer network, formed as a d-dimensional torus separated into a certain number of zones. The coordinate space is dynamically partitioned among all the peers such that every peer possesses its individual, distinct zone. If the number of dimensions is fixed to be $(log_2 n)/2$ then the worst-case number of message hops is $O(log_2 n)$. We have employed this in our algorithm to assure scalable routing.

A GloServ server (GloServer) in the hierarchical network represents a high-level service class within the primitive skeleton the service ontology, which is the pure hierarchy described above. Since the high-level service classes are disjoint, a query will be routed to one of the hierarchical branches, largely 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 network and also represents the class relationships within the ontology.

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. The thesaurus ontology maps synonymous terms of each service to the actual service term within the system. Figure 1 illustrates how servers are found in GloServ. Services are represented as instances

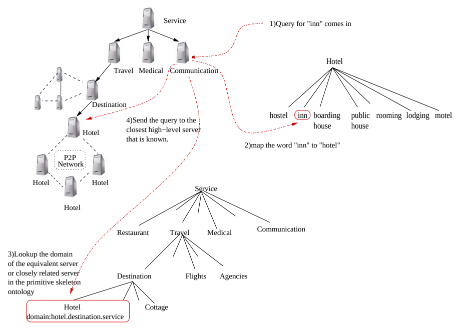

Service
1)Query for "inn" comes in
Travel
Medical
Communication
Destination
Hotel
4)Send the query to the closest high−level server that is known.
Hotel
hostel
inn
boarding house
public house
rooming
lodging
motel
2)map the word "inn" to "hotel"
P2P Network
Hotel
Hotel
Hotel
Service
Restaurant
Travel
Medical
Communication
3)Lookup the domain of the equivalent server or closely related server in the primitive skeleton ontology
Destination
Flights
Agencies
Hotel
domain:hotel.destination.service
Cottage

Figure 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 either in a user-centric way through a web-based form or in an automated fashion by issuing a first-order predicate logic query. 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. Using the Restaurant class as our example, as services

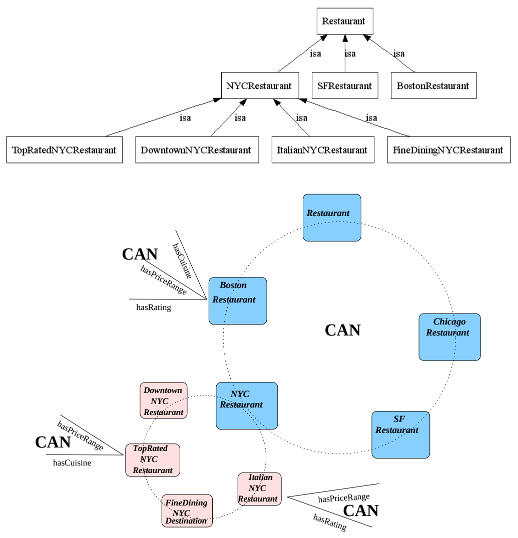


Figure 2: CAN overlay network

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. Details of how the ontology is mapped to a CAN can be read in [5]

Establishing a global service discovery architecture such as GloServ allows us to use ontologies to build a scalable, logically connected network. Ontology queries are then issued in GloServ and are efficiently routed to the correct server. Below we describe the ontology querying mechanism and the recent enhancement of combining the ontology queries with key word search.

3. ONTOLOGY QUERYING

A service registration instance is distributed to all CAN nodes that handle the service classes it belongs to. 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 3 gives a graphical overview of the query propagation in the CAN. We explain the details of ontology querying below by looking at the *Restaurant* ontology as our running example.

3.1 Querying with Restricted Class Dimensions

A user initially contacts a GloServ user agent and enters a service name. 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

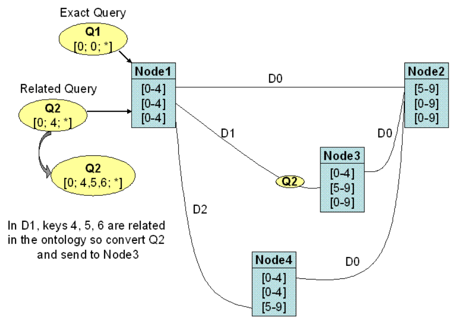


Figure 3: Query propagation in the GloServ CAN

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 restaurant server 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. 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 3, 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.

3.2 Query Matching of Related Information

GloServers receive queries in first order predicate logic statements. The GloServer creates a *query* class with the logical restriction specified in the query and classifies this *query* class within the ontology. The *query* class's superclasses, equivalent classes and subclasses are analyzed. For exact query matching, the equivalent classes and the subclasses are looked into. For related query matching, the superclass's children (which are the restricted class's siblings) are analyzed. 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 example, 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 *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.

3.3 Querying a CAN with Property Dimensions

In the previous example, we looked at queries that were mapped to classes which had restricted subclasses mapped to a CAN. As the class restriction narrows, it may not be necessary to further restrict classes. But as the registration and query load grows within these servers, it is best to distribute the data where each dimension is a property type. For this case, querying is a bit different. Since we do not have subclasses to classify the query class by, we must look at the query class itself and generate keys to distribute within the CAN.

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, a 2-dimensional CAN is generated where each dimension represents a property. If the hasRating property has five values, [OneStar, TwoStar, ThreeStar, FourStar, FiveStar], and hasPriceRange has four values [InExpensive, Moderate, Expensive, VeryExpensive], then there are a total of $5x4 = 20$ possible query combinations to issue. If the query was more specific, where a price range was specified, then the hasPriceRange property value is fixed and only five queries are issued.

Once the query is routed to the correct nodes, it is classified and all the inferred instances are obtained which match this query. The instances are then further analyzed by doing key word matching as we describe below.

4. KEY WORD MATCHING

Thus far, we have described ontology-based queries for query propagation and matching of service instances. However, services may also want to describe themselves with free text, such as with key words that are not already defined in the ontology. In order to be able to handle this case there needs to be a way for the ontology to handle key words and concepts. Below we describe an algorithm on how to incorporate key words in service registrations and queries.

4.1 Service Registration with Key Words

When a service registers within a given service class in GloServ, an instance is created, properties are populated and the instance is classified under a number of restricted classes in the service ontology. Restricted classes are normally restricted by object properties because the range of these properties are predetermined classes. The service instance is then routed to the servers which hold information on these restricted classes. We discuss service registration with key words for object properties and continue to look at an example using the *Restaurant* class.

For the Restaurant ontology, we restrict the classes by the *Neighborhood* and *Cuisine* classes. Thus, if this service provider is a Chinese restaurant in NYC, it is classified under a class which has as its restriction:
($hasNeighborhood$ **some** `NYC`) and ($hasCuisine$ **some** `Chinese`)
The hasCuisine property has its range set to the *Cuisine* class and the hasNeighborhood property has its range set to the *Neighborhood* class. These classes can then have a set of nested subclasses. For example, the *Cuisine* class can have the subclass *Asian* which can then have the subclasses *Chinese*, *Japanese* and *Korean*. Although the *Cuisine* class can be constructed to be very rich in its subclass definitions, this is not always guaranteed. Thus, we would like to allow services to provide extra information when registering to include specific key words. For example, when a Chinese restaurant is registering, it sets its hasCuisine property to *Chinese* and then should have the option of adding extra keywords which may include their most popular menu items, daily specials, etc. Although all object type properties can have key words added to them, this may not be necessary. For example, the Restaurant ontology has a number of object properties such as $hasNeighborhood$, $hasCuisine$, $hasRating$, $hasPriceRange$. The $hasRating$ or $hasNeighborhood$ properties do not vary much in terms of expressivity. Thus it is redundant to add key words to these properties. Since OWL DL allows properties to be annotated with properties themselves, if a property is annotated with a $keyWordMatch$=`true` annotation, then key word matches are allowed for those properties.

In order to accomplish this, we create a *KeyWord* class which holds a list of key words that services have created while registering. The service provider fills out the object properties and is then given the option of creating key words for each of these properties. Thus, if the service provider sets the hasCuisine restaurant value to *Chinese*, it is prompted with a list of key words from the *KeyWord* class which it can choose from and tag onto the hasCuisine property. It is also given the choice of creating new key words, which are added as new terms within the *KeyWord* class.

The property hasKeyWord is an object property which points to the *KeyWord* class. Every class in the ontology, which can be tagged with key words, has the hasKeyWord property as one of its properties. When a service chooses the *Chinese* class as its cuisine, an instance of the *Chinese* class is assigned to that service's hasCuisine property. If the service provider wants to add extra key words describing specifics to the Chinese cuisine, it is given a list of already generated key words it can choose from. The service provider chooses from this list and adds these key words to the instance of the *Chinese* cuisine class by adding multiple hasKeyWord statements for the instance. If the service provider wants to add new key words, these get added to the *KeyWord* class and are tagged onto the *Chinese* cuisine instance.

Additionally, the service provider can add a set of key words which are not tied to properties but generically describe its own service In this case it will populate the `hasKeyWord` property for its own service instance.

4.2 Querying for Services with Key Words

A query which includes key words is constructed in two parts. The first part is the ontology-based query. The second part holds key word information for each property. This is then converted to a complete ontology query.

If the front-end to the system is a graphical user interface, a user fills out the ontology form and can then enter key words next to each of the properties which has key word matching enabled. This query is sent to the back-end in two parts: ontology-based query and key word query. The ontology query is classified and sent to the appropriate GloServers. A new ontology query is then constructed to also include the key word queries.

This ontology query is constructed by first matching the key words to the key words in the *KeyWord* class. This can either be done by asking the user to choose a list of key words directly from the *KeyWord* class or have the user enter random key words. For the first case, the ontology query must be issued first in order for it to be routed to the appropriate server. Then a list of terms from the *KeyWord* class within that server is returned to the user so that she can choose those that match her request. For the second case, a user can add the key word terms along with the ontology query because the key words are matched to the terms in the *KeyWord* class using a text matching tool. Our implementation handles the first case, but can easily be extended to handle the second one by adding a text matching engine to match the key words to the key words already in the *KeyWord* class. This is a desirable feature because it extends to different types of front-ends which may not be user centric.

Once the key words are determined, an ontology query is built for each of the properties. For example, let us say the user entered the key words `Schezuan` and `Cantonese` as additional key words for the hasCuisine property. A restricted subclass is formed under the *Cuisine*/*Chinese* class with the restriction:
($hasKeyWord$ **has** `Szechuan`) and ($hasKeyWord$ **has** `Cantonese`)
The ontology is classified and a list of inferred instances of the *Chinese* class which have these key words are classified under the query class. The name of an instance is usually the name of its class with a unique number to distinguish it from other instances in that class. Thus, a list of instances returned could be: `Chinese_1` and `Chinese_2`. Once these instances are obtained, the original ontology query is changed to include these specific instances. Thus, the original query:
($hasCuisine$ **some** `Chinese`) is replaced with:
($hasCuisine$ **has** `Chinese_1`) or ($hasCuisine$ **has** `Chinese_2`)
A restricted class is created under the *Restaurant* class with this condition and the reasoner is run on the ontology to obtain a list of inferred instances which match this restriction. These instances are returned to the user. Additionally, since the key word queries match the *Chinese* class, related query matching can also be done as described above.

Besides entering key words for specific properties, the user may enter key words which give generic descriptions of the service. For these generic key words, the original ontology query is extended to include a condition for the hasKeyWord property. If the user enters key words such as: `rotating` and `view`, the ontology query

example above is changed to:
($hasCuisine$ **some** `Chinese`) and
(($hasKeyWord$ **has** `rotating`) or ($hasKeyWord$ **has** `view`))
A query class is created under the *Restaurant* class with this restriction and the ontology is classified to obtain all the instances which have these key words.

If users enter key words that do not match key words already in the ontology, then the instances which match the ontology part of the query are returned to the user and the key words that do not have a match are discarded.

4.3 Discussion

We try to limit the size of each service ontology so that classification will be done in a reasonable amount of time. When adding key words to an ontology, this can potentially cause the ontology to become very large. Below we discuss how to solve this problem.

First, periodic text matching is done between key words so that similar key words are merged into one term. Second, key words that are queried for often are included in the ontology and those that do not appear as often are stored only in the database instances. Thus, storing key words in both the ontology directly and the database, gives us the benefit of doing reasoning with key words, but also limiting the size of the ontology. Third, we store the *KeyWord* class apart from the main service ontology and merge it with the service ontology only when a key word search is being performed. Implementing these optimizations are part of our future work.

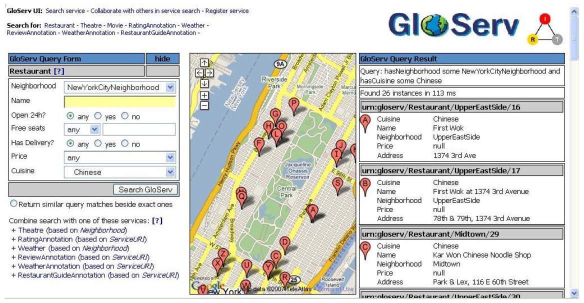
5. IMPLEMENTATION

We have implemented a prototype of GloServ using Protege [6] and Racer-Pro [9]. 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.

As the CAN is generated, nodes enter the system and are assigned to a zone. Each server initially holds many classes but as the number of nodes increase, the servers hold one class per dimension. Once a class exceeds a threshold for registration or querying, it checks with other servers that handle the same class to see if a CAN subnetwork has already been formed. If it has, it caches the subnetwork's supernode information and transfers its data to this subnetwork. Subsequent registrations and queries are sent to this subnetwork. Otherwise, if there is no subnetwork, it processes its preconfigured ontology to generate the CAN subnetwork, and transfers data there. When subclasses do not exist, it parses the unrestricted properties and generates a CAN with property dimensions. We have built an ontology generating tool which automatically generates an ontology given a configuration file. A full CAN network is generated automatically when given the following input: an ontology configuration file, number of nodes in the CAN, list of server host names or IP addresses, list of super nodes in the CAN. The CAN is generated by the main server invoking remote "JOIN" messages from the list of listening servers.

We have built a front-end user interface to GloServ written in PHP. The interface uses Google Maps which displays 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. Services 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, the interface displays a text box underneath the form for the user to enter key words in. The interface is seen in Figure 4 and can be accessed on the web at *http://gloserv.dyndns.org:8080*

Tests for the ontology querying have produced promising results. For an ontology with 100 classes, the time it took to classify ranged from 0 to 15 ms. Query propagation across the CAN for 10 nodes averaged 80 ms. We estimate the CAN networks to have approximately 1000 nodes each since they represent a subdomain of the full ontology, following from the location-based example, a CAN network for *NYCRestaurant* classes. Thus, the maximum delay would average 8 seconds. The full query including reading 10,000 instances in the database took around 150 ms, using MySQL as the back-end database. Since key word search requires another clas-



GloServ UI: Search service - Collaborate with others in service search - Register service

Search for: Restaurant - Theatre - Movie - RatingAnnotation - Weather - ReviewAnnotation - WeatherAnnotation - RestaurantGuideAnnotation -

GloServ Query Form | hide

Restaurant [?]

Neighborhood	NewYorkCityNeighborhood
Name	
Open 24h?	[✓ any] yes no
Free seats	any
Has Delivery?	[✓ any] yes no
Price	any
Cuisine	Chinese

Search GloServ

Return similar query matches beside exact ones

Combine search with one of these services: [?]
+ Theatre (based on *Neighborhood*)
+ RatingAnnotation (based on *ServiceURI*)
+ Weather (based on *Neighborhood*)
+ ReviewAnnotation (based on *ServiceURI*)
+ WeatherAnnotation (based on *ServiceURI*)
+ RestaurantGuideAnnotation (based on *ServiceURI*)

GloServ Query Result

Query: hasNeighborhood some NewYorkCityNeighborhood and hasCuisine some Chinese

Found 26 instances in 113 ms

urn:gloserv/Restaurant/UpperEastSide/16

A	Cuisine	Chinese
	Name	First Wok
	Neighborhood	UpperEastSide
	Price	null
	Address	1374 3rd Ave

urn:gloserv/Restaurant/UpperEastSide/17

B	Cuisine	Chinese
	Name	First Wok at 1374 3rd Avenue
	Neighborhood	UpperEastSide
	Price	null
	Address	78th & 79th, 1374 3rd Avenue

urn:gloserv/Restaurant/Midtown/29

C	Cuisine	Chinese
	Name	Kar Won Chinese Noodle Shop
	Neighborhood	Midtown
	Price	null
	Address	Park & Lex, 116 E 60th Street

urn:gloserv/Restaurant/UpperEastSide/30

Figure 4: GloServ Front-End

sification using the reasoner, this adds another 0 to 15 ms to the query time. Thus, ontology querying with key word search results in double the amount of time of a regular ontology query, namely 0 to 30 ms.

6. RELATED WORK

6.1 Service Discovery

Current approaches to service discovery are either centralized or decentralized. Centralized service discovery systems include: SLP [8], Jini [13], and UDDI [2]. 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 [12], [4], [18]. These systems are similar to GloServ in that they use data from a schema or ontology to map onto a network. GloServ differs from these systems by using an ontology to map a multi-tier hierarchical peer-to-peer network. It is similar to [12] and [18] in that a concept represents a subnetwork. However, unlike [12], 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 [12], indices refer to whole peers. Also, because of the CAN DHT, GloServ generates $O(log_2 n)$ query messages whereas in [18] flooding is used which generates $O(n)$ messages which is inefficient on a global scale.

6.2 Ontology-based Information Retrieval

Currently, work done in combining ontology-based search with information retrieval focuses on adding 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 [14], [3] and [11].

GloServ addresses a different problem. It seeks to represent and discover services using ontology queries and key word search. Service data is already represented as ontology instances. Thus, a set of key words does not exist initially, but the ontology is modified as services register and add their own key words to the ontology. 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.

7. CONCLUSION

GloServ is a hierarchical peer-to-peer global service discovery system using OWL DL. GloServ functions both on a wide area as well as a local area network. A broad range of services are defined flexibly using OWL ontologies. We have described a recent enhancement to GloServ which combines ontology querying with key word search. The ontology query is used to route the query to the servers that hold information on these services and to find a list of matching and related instances. Key words are then matched to the properties of these instances. Combining ontology querying with text searching enhances the description and discovery of services.

8. ACKNOWLEDGEMENT

We would like to thank Peter F. Patel-Schneider of Bell Labs Research for his contribution to this work.

9. REFERENCES

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

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

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

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

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

[6] 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.

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

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

[9] 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.

[10] 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 Conference (ISWC2004)*, Hiroshima, Japan, Nov 2004.

[11] 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.

[12] 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.

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

[14] 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.

[15] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. San Diego, CA, USA, August 2001. ACM.

[16] 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.

[17] 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.

[18] 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.