Combining Ontology Queries with Text Search in Service Discovery

Knarig Arabshian and Henning Schulzrinne

Department of Computer Science
Columbia University, New York NY 10027, USA
{knarig,hgs}@cs.columbia.edu

Abstract. We present a querying mechanism for service discovery which combines ontology queries with text search. The underlying service discovery architecture used is GloServ. GloServ uses the Web Ontology Language (OWL) to classify services in an ontology and map knowledge obtained by the ontology onto a hierarchical peer-to-peer network. Initially, an ontology-based first order predicate logic query is issued in order to route the query to the appropriate server and to obtain exact and related service data. Text search further enhances querying by allowing services to be described not only with ontology attributes, but with plain text so that users can query for them using key words. Currently, querying is limited to either simple attribute-value pair searches, ontology queries or text search. Combining ontology queries with text search enhances current service discovery mechanisms.
Keywords: service discovery, ontologies, OWL, CAN, peer-to-peer, text search, 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 but 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 relations to services besides exact matches.

In order to address these problems, we have developed GloServ [6], a global service discovery system, which uses the Web Ontology Language (OWL) [1] to classify services in an ontology and map knowledge obtained by the ontology onto a hierarchical peer-to-peer network. It operates on 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. Initially, an ontology-based first order predicate logic query is routed to the servers which handle that service class. The query is then processed within the servers in order to obtain service instances that not only include exact matches, but logically related ones as well.

As part of our ongoing research, we have enhanced GloServ to also perform text search. Text searching further refines the results by querying for certain terms within a service instance's property that holds free text, such as key words. Additionally, we provide a way of performing text search on the ontology constructs themselves by allowing service providers to add key words to classes within the ontology.

Combining ontology queries with text search gives more flexibility and accuracy when obtaining query results. Performing pure ontology queries limits a service description and query to the the service ontology definition. However, when a service is described purely with text, only an approximated set of results can be obtained which may or may not be what the user is looking for. Thus, when combining these methods together, we reap the benefits of both by first classifying services and performing an ontology query in order to hone in on the correct set of services and then further refining this set with text matching on a set of key words.

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

2 Overview of GloServ

GloServ classifies services in an OWL DL ontology. This classification defines service classes and their relationships with other services and properties. There are many ways to compose ontologies. We have adopted the modularization approach specified in [11] 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 in an ontology in disjoint hierarchical trees, which creates a *primitive skeleton*. Hence, individuals 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

Scaling is achieved by mapping the ontology onto a hierarchical peer-to-peer network of services. This network exploits the knowledge obtained by the service classification ontology as well as the content of specific service registra-

tions. The hierarchical network is formed by connecting the nodes between the high-level, disjoint service classes within the service classification. The peer-to-peer network is formed between the service classes in the lower levels of the ontology, which may have relationships with other classes.

We use a Content Addressable Network (CAN) [16] as our distributed hash table to form a 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 within the overall space. Each peer in the CAN maintains a routing table that holds the IP address and virtual coordinate zone of its neighbors. When a query comes into a CAN node in the form of a dimension-key value, the node checks to see if it holds information for that dimension-key pair. If it does not, it routes the query to the neighboring node which is closest to the destination coordinates.

GloServ servers (GloServers) have 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 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. Registration can be done either in a user-centric way through a web-based form or in an automated fashion by issuing a first-order predicate logic query.

At the lower levels, maintaining a purely hierarchical ontology structure becomes difficult as there are many overlaps between classes. 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 Content Addressable Network 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 *Restau-*

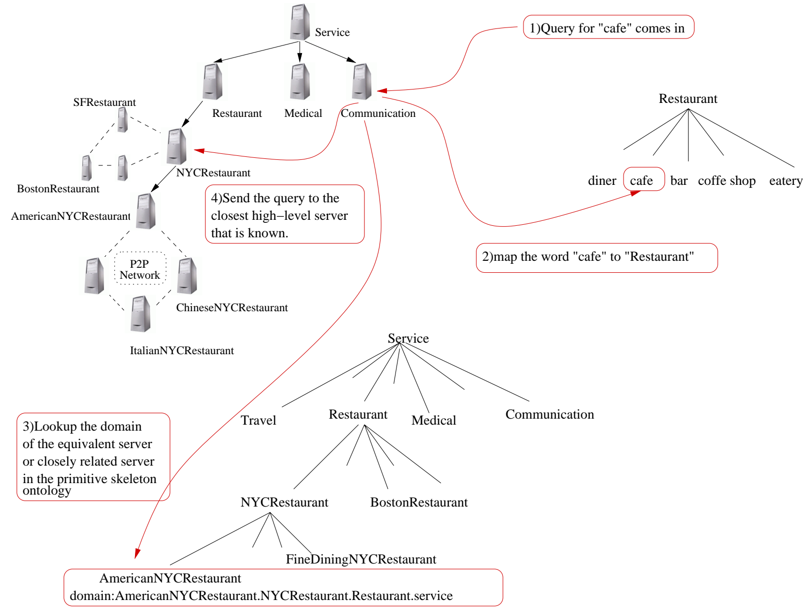

Fig. 1. Finding servers in GloServ

rant class. The number of dimensions is determined by the number of nodes contained within the CAN.

As services register within CAN nodes 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 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. 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.

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 text search.

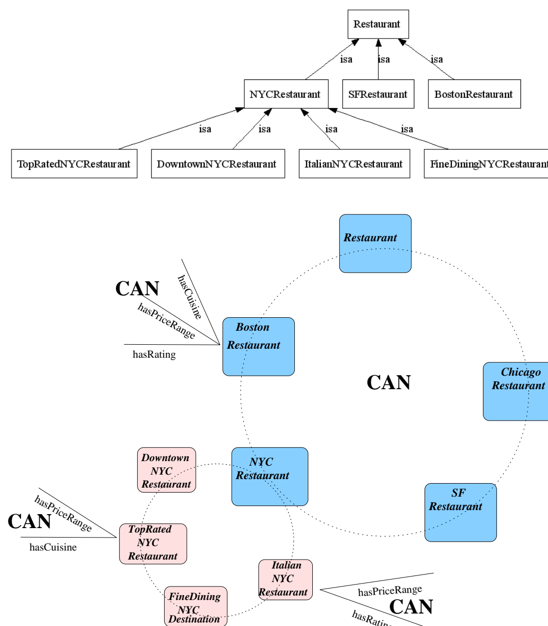

Fig. 2. CAN overlay network

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 distributed hash table such as CAN, 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 below by looking at the *Restaurant* ontology.

3.1 Querying with Restricted Class Dimensions

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. When 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

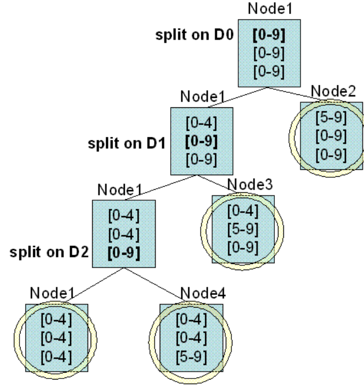

Fig. 3. CAN node splitting into two nodes

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, the query class is classified under the *KoreanNYCRestaurant* and *ChineseNYCRestaurant* classes. This process repeats until there are no more subnetworks to send the query to. In order to implement this using 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.

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`))

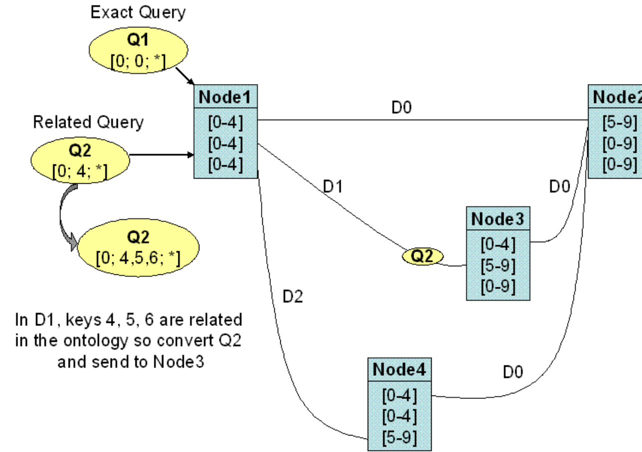

Fig. 4. Query propagation in the GloServ CAN

the query message is $[0; 0; *]$. As seen in Figure 4, Node1 receives the query and stops propagating it because it handles these classes. When querying for related classes besides the one specified, Node1 looks at related keys to the ones specified and issues query messages for each. For example, if a the query message $[*; 4; *]$ comes into Node1 and dimension 1 has related keys 4, 5 and 6, 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. Query languages such as RQL [12] or SPARQL [3] are not used because our query matching algorithm does not just look for exact matches. Rather, as mentioned above, 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 re-

lated 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 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.

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 in, 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 insances are then further analyzed using text-based search as we describe below.

4 Text Search

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 to 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.

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 propety 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 inserting multiple hasKeyWord properties to 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

When a user queries for services, it first fills out the ontology form which is converted to a first-order predicate logic query. The user is then given an option to enter additional key words for each of the properties. The key words in the query are matched 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. 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 also be extended to handle the second one.

Once the key words are set, 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 numeric 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.

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.

5 Implementation

Currently, we are implementing a prototype of GloServ using Protege [8] and Racer [10]. Protege is an open-source development environment for ontologies and knowledge-based systems. The OWL Plugin is an extension of Protege that supports OWL. The Protege OWL Plugin provides a user-friendly environment to edit and visualize OWL classes and properties. It also has a graphical user interface that allows users to define logical class characteristics in OWL and execute description logic reasoners such as Racer. Protege's flexible architecture makes it easy to configure and extend the tool. Protege has an open-source Java API for the development of custom-tailored user interface components or arbitrary Semantic Web services.

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 is 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.

The service provider registers through a graphical user interface by choosing various property values. This is converted to a restricted query class and propagated across the CAN. The service is registered when it is instantiated within the matching nodes and classified appropriately. To generate many service registrations, we have automated the creation of instances throughout the network and distributed them throughout the nodes. Registration and querying are then done with the algorithms described in Sections 3 and 4

We have implemented the CAN network generation using information from the service ontology classification and the ontology querying scheme. Currently, we are working on implementing the text searching extension. We will test the scalability of the queries by running experiments to examine the latency of the query routing. We will then compare results from pure ontology queries to those that have been enhanced with text searches to see if these yield more accurate results.

6 Related Work

6.1 Service Discovery

Service discovery protocols in use today include SLP [9], standardized by the IETF, Sun Microsystem's Jini [14], Microsoft's UPnP (Universal Plug and Play) [7] and UDDI (Universal Description, Discovery and Integration [4].

SLP and Jini have centralized service registries which store service information in attribute-value pair descriptions. Users discover services by querying the registries for services. SLP and Jini funciton in local area networks but do not scale to wide area networks.

UPnP 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. UPnP is also limited in terms of service description and network scaling.

UDDI 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 [2]. Services in UDDI are stored in a centralized business registry.

GloServ differs from all of these systems in that it is globally scalable because it is built on 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.

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 [15], [5] and [13].

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. Broad range of services are defined flexibly using OWL ontologies. The Gloserv architecture achieves large-scale distribution of semantic data that is queried for with specificity and efficiency. The ability to reason in OWL DL promotes intelligent distribution of service content across nodes connected in a CAN peer-to-peer network.

We have described a recent enhancement to GloServ which combines ontology querying with text 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. Text search is used to match key words 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.

References

1. Owl web ontology language. OWL http://www.w3.org/2004/OWL/.
2. Simple object access protcol. http://www.w3.org/TR/soap/.
3. Sparql query language for rdf. http://www.w3.org/TR/rdf-sparql-query/.
4. Uddi technical white paper. white paper, uddi (universal description, discovery and integration), September 2000. http://www.uddi.org/pubs/.
5. 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*, 2000.
6. Knarig Arabshian and Henning Schulzrinne. An ontology-based hierarchical peer-to-peer global service discovery system. *Journal of Ubiquitous Computing and Intelligence (JUCI)*, 2006.
7. Upnp Forum. Upnp device architecture 1.0. Technical report, December 2003.
8. 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.
9. E. Guttman, C. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.
10. Volker Haarslev and Ralph Moller. Racer user's guide and reference manual version 1.7.19. Concordia University, Tehcnical Universityh of Hamburg-Harburg, University of Hamburg, 2004.
11. 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 Prince Hotel, Hiroshima, Japan, Nov 2004.
12. Gregory Karvounarakis, Sofia Alexaki, Vassilis Christophides, Dimitris Plexousakis, and Michel Scholl. RQL: a declarative query language for RDF. In *Proceedings of the 11th International World Wide Web Conference*, pages 592–603, 2002.
13. 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.
14. Sun Microsystems. Jini architectural overview. Technical report, 1999.
15. 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.

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