An Ontology-based Hierarchical Peer-to-Peer Global Service Discovery System

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

***Abstract*— Current service discovery systems fail to span across the globe and they use simple attribute-value pair or interface matching for service description and querying. We propose a global service discovery system, GloServ, that uses the Web Ontology Language (OWL) for service classification and the dynamic formation of the network architecture. The GloServ architecture spans both local and wide area networks. It maps knowledge obtained by the service classification ontology to a structured peer-to-peer network such as a Content Addressable Network (CAN). GloServ also performs automated and intelligent registration and querying by exploiting the logical relationships within the service ontologies.**

Keywords: service discovery, ontologies, OWL, CAN, peer-to-peer, ubiquitous and pervasive computing, context-aware

I. Introduction

GloServ [1] [2] is a global service discovery architecture that is used for ubiquitous and pervasive computing. 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. Unfortunately, current service discovery systems have a limited capability in network scaling and service descriptions. GloServ solves these problems by providing network scalability, logical service descriptions, and intelligent service registration and querying.

GloServ classifies services in an ontology, such as OWL DL [3]. This classification defines service classes and their relationships with other services and properties. 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 registrations. The hierarchical network is formed by connecting the nodes between the high-level, disjoint services within the service classification. The peer-to-peer network is formed, between equivalent or related services via a Content Addressable Network (CAN) [4]. Furthermore, due to the use of ontologies, services are described with richer semantics. This results in automated and intelligent registration and querying,which we provide algorithms for. Thus, the GloServ architecture provides a foundation where the primary challenges in ubiquitous and pervasive computing technologies are addressed.

In [1] we gave an initial outline of GloServ and in [2] we described how services are classified using OWL. Details of the network architecture, registration and querying are fine tuned in this paper. Below we present our global service discovery system and discuss each of the problems and their corresponding solutions. Section II motivates the need for a global service discovery architecture for ubiquitous and pervasive computing environments. An overview of current service discovery systems is discussed in Section III. We describe the ontological engineering approach used in Section IV. Sections V, VI and VII discuss the architecture, CAN generation, querying and registration mechanisms of GloServ respectively. The implementation of the system is described in Section VIII and finally we conclude in Section IX.

II. Motivation

The Gloserv architecture achieves large-scale distribution of semantic data that is queried for with specificity and efficiency. These attributes make it a very good candidate for context-aware applications. Such context-aware applications need to have access to both ubiquitous as well as pervasive information. The design of these applications is not discussed in this paper, but regarded as future work. However, it is useful to discuss this in order to motivate the creation of global ontology-based service discovery architecture.

We discuss two types of context-aware applications that can be built upon the underlying GloServ architecture. Imagine a driver who wants to be directed to a nearest gas station, within a certain price range, when the car's gas begins to run low. Another scenario is of a traveler, carrying a PDA with her daily itinerary, who is notified when walking nearby the services within her itinerary. In these cases, a device is preconfigured to search for a specific service.

A GloServ user agent runs on a certain monitoring device. A user may configure her device by connecting to GloServ, downloading a query form of preferred services and entering attributes. Thus, in the case of the car directing the user to the gas station, the user will have preconfigured a device that monitors certain dynamic conditions of the car, such as the gas level. The user downloads a query form for the "GasStation" service and enters the price range and other attributes it prefers. Attributes dynamically added upon the service query, such as location information, is provided by the monitoring device. Thus, the device will know that when the car is running out of gas, it must check its location via a GPS receiver, enter that location information in the query form and issue the query. The driver is alerted of the low gas situation and the query results are shown to her.

For the case of the traveler, she will have a list of things to do in her PDA before she starts her day. She will indicate her preferred services to be notified of in the configuration. A GloServ user agent running on the PDA queries for these services and notifies her when she is near a particular service. If she is taking various tours from city to city, she is notified of budget hotels within her area at a particular time of day.

Thus, GloServ assists in automated context-aware service discovery when a certain query needs to be issued repeatedly for a specific context. The device is preconfigured by contacting GloServ, issuing a query form of that service and filling out the necessary attributes. Additional attributes that vary are entered at the time of issuing a query. Preconfiguration is unavoidable as the context needs to be determined beforehand in order to issue a query with specific attributes.

GloServ is also useful in event monitoring. In this case, a dynamic service registers with GloServ and periodically updates its attributes. For instance, a restaurant service may want to indicate how many people it can seat every half hour. A user will request notification when a nearby Italian restaurant has availability for a party of five during the evening. As she is walking with her friends, her device will notify her of restaurants that come up. In this case, a service provider will automate updates of its service at certain time intervals. Every time a service is updated, a device that is subscribed to this service is notified and the query is re-issued. Traditional publish/subscribe methods can be used to issue the query subscriptions and event notifications.

In conclusion, the use of ontologies for service description and querying accomplishes specific context-aware service discovery. By scaling the service discovery system globally, services are ubiquitously available. Furthermore, using service classification ontologies results in supporting a broad range of service types. All these properties are useful for the next wave of applications in the area of ubiquitous and pervasive computing.

III. Related Work

A. Service Discovery

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

SLP [5] and Jini [6] are both similar in that they have agents that manage services, users and directories of services. Agents advertise each others' presence to each other using either multicast or unicast. In SLP, service registration and queries are broadcast to the directory agents or directly between the service and user agents depending on if the directory agents are present. In Jini, however, a client downloads the service proxy and invokes through Java RMI in order to access the service through a discovery process. Service descriptions in SLP are done in simple attribute-value pairs whereas Jini matches interfaces. SLP is mainly used in local area networks. Jini and a scalable version of SLP, Mesh-enhanced SLP (mSLP) [7], can span to a larger enterprise networks.

UPnP [8] 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. XML describes the services in greater detail. UPnP is appropriate for home or small office networks. Unlike SLP and Jini, UPnP provides more descriptive queries through XML.

The Universal Description, Discovery and Integration (UDDI) [9] specification is used to build discovery services on the Internet. UDDI provides a consistent 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. Services in UDDI are stored in a centralized business registry. The main drawback of UDDI is that it has a centralized architecture and does not span to a global area.

Recently there have been developments in wide area service discovery. INS/Twine [10] and Ninja [11] describe two such systems. Both systems use XML to describe services. However, INS/Twine maps strands of hierarchically partitioned data to a structured peer-to-peer system such as Chord. Ninja, on the other hand, organizes servers dynamically into hierarchies and issues upward queries using Bloom filters.

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

B. Schema-based Peer-to-Peer Systems

The existing work closest to our research use schemas to map onto a network [12], [13]. These systems are similar to GloServ in that they use data from a schema to map onto a network. [13] outlines a semantic gossiping framework that exchanges ontology information within a peer-to-peer network. It uses Gnutella [14] 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. [12] 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.

GloServ differs from these systems by using an ontology to map a multi-tier hierarchical peer-to-peer network. It is similar to [12] in that a concept represents a sub-network. 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. Section V describes GloServ architecture in greater detail.

Also, [13] and [12] 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.

IV. Ontologies

A. OWL Overview

The World Wide Web Consortium has recently approved OWL [3] as a standard for the Semantic Web. OWL builds on Resource Description Framework (RDF) [15] and RDF Schema [16] and adds more vocabulary for describing properties and classes such as: relations between classes, cardinality, equality, richer typing of properties, characteristics of properties, and enumerated classes. Below we give an overview of the sublanguages of OWL and the characteristics of OWL Classes and Properties.

There are three sublanguages in OWL: OWL Lite, OWL DL and OWL Full. OWL Lite is the least expressive of the three sublanguages. Although it is a bit more expressive than RDFS that in addition to supporting a classification hierarchy, it also provides simple constraints of classes and properties. OWL DL is modeled after description logics and supports maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computable) and decidability (all computations will finish in finite time). OWL DL includes all OWL language constructs. OWL Full is the most expressive of the three sublanguages. The main difference between OWL DL and OWL Full is that in OWL DL, a class is only expressed as a collection of individuals and can not be regarded as an object in and of itself. However, in OWL Full, a class can be treated simultaneously as a collection of individuals and as an individual in its own right. Due to this difference, OWL Full can not be completely supported by OWL Description Logic Reasoners to check for soundness. We have chosen to use OWL DL for two reasons: 1)a service class will only represent a collection of individuals and does not need to be an individual in its own right and 2)we would like to use OWL DL reasoners such as Racer [17] to check for the soundness of OWL documents. Thus, due to the heavy use of reasoners within GloServ, our system operates correctly within OWL DL and the use of the other OWL sublanguages are not guaranteed to be successful.

B. Constructing Ontologies

There are many ways to compose ontologies. [18] and [13] describe a few. [13] describes an ontological engineering method that provides a modularized approach to classification and allows the most flexibility when combining various ontologies. It uses an analogous method of database normalization in order to normalize ontologies. General domains (or classes), within in an ontology, are put in disjoint hierarchical trees, which creates a *primitive skeleton*. The main goals of normalization are to allow modules to be re-used and separated from the whole and to evolve independently of each other. These characteristics are necessary in any ontological-based system. We have chosen this method for service classification and describe the details of how services are classified below.

The main features of OWL DL include primitive and defined classes (or concepts), properties, restrictions and axioms. A primitive class is one that is described by necessary conditions whereas a defined class is described by necessary and sufficient conditions. Practically, this means that primitive classes are in a hierarchical class/subclass relationship, while defined classes describe equivalences. Properties relate classes to each other and can themselves be hierarchical as well. Restrictions quantify the property-class pair and axioms declare classes disjoint or imply other classes.

These features can be used in a variety of ways in order to produce meaningful ontologies. However, as described in [13], the best approach to identify modules is to first create a primitive tree which is a hierarchical tree of primitive concepts. The primitive skeleton resides on the top level of the ontology and is constructed in such a way that each concept has only one parent and disjoint siblings. Once this primitive skeleton has been formed, descriptions and definitions are created to express the relations between those primitives.

Primitive skeletons should also distinguish two types of concepts: *Self-standing concepts* and *Partitioning concepts*. Self-standing concepts include “things” that are part of the physical world such as “animals” or “organizations”. Partitioning concepts, on the other hand, are values that partition self-standing concepts such as “small, medium, large”. By using primitive skeletons, the evolution, sharing and re-use of ontologies is greatly simplified.

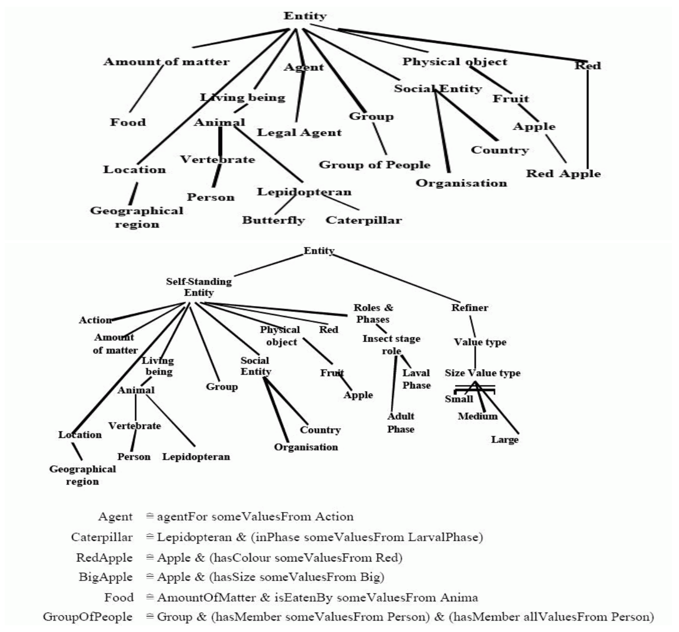
Figure 9 shows an example of a classification ontology that has been converted to a primitive skeleton. In the original hierarchy there are certain concepts such as, *RedApple* that is both a child of *Apple* and *Red*. However, in the normalized skeleton, there are two skeletons: a *Self-Standing Entity* and a *Refiner*. In this way, *RedApple* is split so that *Apple* is a child of *Fruit* and a subclass of *Apple* is created with a hasColor property of *Red*. Further, *BigApple* is refined to be a subclass of *Apple* with a hasSize property of *Big* from the *Refiner Skeleton*.

Thus, normalizing an ontology provides better modularization. The separation of the self-standing and refiner ontologies allows other systems to re-use parts of these ontologies. Such classifications will not change frequently over time and thus can be distributed and cached periodically across many servers within the GloServ network.

V. GloServ Architecture

A. Motivation

The GloServ architecture consists of servers connected as a hybrid network of hierarchical and peer-to-peer nodes. Our



Agent	≘ agentFor someValuesFrom Action
Caterpillar	≘ Lepidopteran & (inPhase someValuesFrom LarvalPhase)
RedApple	≘ Apple & (hasColour someValuesFrom Red)
BigApple	≘ Apple & (hasSize someValuesFrom Big)
Food	≘ AmountOfMatter & isEatenBy someValuesFrom Anima
GroupOfPeople	≘ Group & (hasMember someValuesFrom Person) & (hasMember allValuesFrom Person)

Fig. 1. Original classification ontology converted to corresponding primitive skeleton

motivation for using a hybrid network, verses a homogeneous one, is due to the nature of the ontology classification. If an ontology is richly defined with many restricted subclasses, an instance of a class will be classified automatically within the deeper part of the classification tree. Thus, it is expected that the higher-level services will not contain as much data as those in the lower part of the classification. Therefore, we form the network as a hierarchy on top and peer-to-peer networks in the bottom levels.

The hierarchical primitive skeleton ontology is used for separating these high-level services. As mentioned above, we anticipate that these servers will not hold as much data as the low-level servers. Therefore, we do not need to worry about load distribution and query scaling since the main purpose of these servers is to route messages to the lower-level. By using the primitive ontology model, any server is able to get to its children as well as to those servers that are disjoint from it very quickly. Disjoint servers are those that handle service classes that are completely unrelated to each other, namely the sibling servers.

Besides organizing high-level servers, we need to establish a network for servers that contain closely related information. These servers are connected to each other in peer-to-peer overlay networks. The motivation for using a peer-to-peer network is to achieve load distribution and fast query processing time, while maintaining reliability. Since we use a CAN for data distribution, querying is faster as data is distributed according to content and each server handles a set of information. In general, there are several ways to do load distribution. If registrations are replicated across all servers, there is no need for a structured peer-to-peer system. However, this causes the servers to hold large amounts of data which need to be processed for every query, resulting in slower query processing time. Thus, a peer-to-peer network establishes a robust and scalable environment for service registrations and queries. Figure 2 shows a partial view of the overall GloServ architecture.

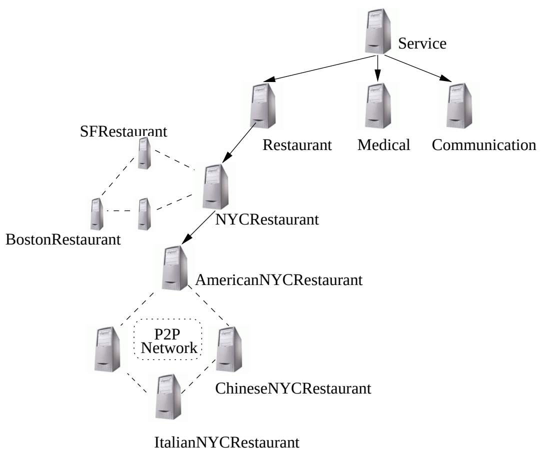


Fig. 2. GloServ architecture

B. Elements within a Gloserver

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. An example of a high-level service classification ontology can be seen in Figure 3. As mentioned above, this classification 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

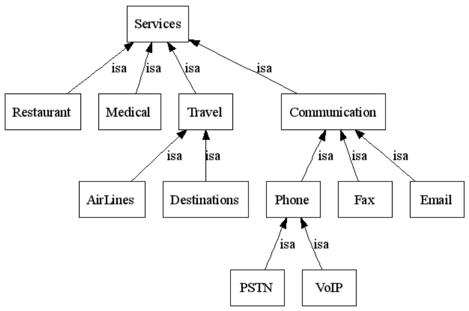


Fig. 3. A high-level service classification

The second piece of information is a thesaurus ontology. The thesaurus ontology maps synonymous words to each of the service terms in the service classification ontology. This results in a greater degree of accuracy in finding the correct server and information for registration and querying.

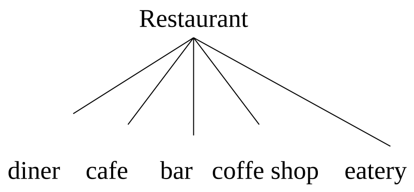


Fig. 4. Partial view of a thesaurus graph containing synonyms of "Restaurant"

Synonyms of every class within the service classification hierarchy are stored. Figure 4 gives an example of the partial graph of the synonyms of *Restaurant* within the thesaurus ontology. This, too, will not change often and thus can be distributed and remain in each of the servers.

The third component within each Gloserver is a CAN lookup table which is constructed according to the ontology structure. The CAN table connects servers of related classes to each other in a peer-to-peer network. We use a novel mapping algorithm that combines the benefits of OWL and CAN to map content of service instances to nodes in a peer-to-peer network. Although there are other types of structured peer-to-peer networks such as Pastry [19] and Chord [20], we have elected to use CAN because it distributes data according to content, which fits best with our ontology-based service discovery model. The CAN structure, is a d-dimensional torus that is separated into a certain number of zones. Each zone handles a subset of keys pertaining to the dimensions. Section VI describes how we use an OWL ontology to generate keys in order to map the data onto a CAN network so that related service classes reside in adjacent CAN zones. This limits the query propagation while guaranteeing accurate results.

C. Server Bootstrapping in the Hierarchical Network

High-level services that are represented in the primitive ontology, are stored in a hierarchical network. The servers of these high-level services are bootstrapped using a DNS lookup. Each server represents a service class and its hostname is determined by looking at the primitive skeleton ontology. Server hostnames will follow the hierarchical format. For instance, as seen in the service classification ontology in Figure 3 the *Restaurant* class's URI will be Restaurant.Service whereas the *Destination* class's URI will be Destination.Travel.Service. As a server is assigned to a hierarchical network, it updates the ontology to include its server information.

Figure 5 outlines the steps taken to find the main high-level server when a query is issued. Initially, the appropriate server is found by querying for a service type. Let's assume a user is querying. When the user enters the word *cafe*, the initial server processing the query will first map the word *cafe* to a synonymous term within the thesaurus ontology. In this case, it is mapped to the *Restaurant* class. The server locates the class in the primitive service classification ontology and determines the host name in either of two ways described below.

The first way would be to store a snapshot of the whole primitive classification in every server. This classification not only gives the relationships of each of the service classes, but also holds the domain name information of the main high-level server to contact. This method is plausible only because we expect the order of service types to be in the 100s. This expectation comes from realizing that the average number of words known by a human is around 20,000 words which causes us to conclude that the number of words within the classification is much less than 20,000. The other possibility is that servers only have information about their disjoint siblings, a parent and child. Using this method, there is always a way of getting to another node within the classification ontology.

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

Once the restaurant node's domain name is determined, it is contacted with the user's query. The restaurant node will have service registrations stored in it. These are actual instances of the *Restaurant* class. Many instances of restaurants are stored here and thus the information will have to be distributed across other restaurant servers that are connected to each other in a peer-to-peer fashion. This is where CAN is used.

The main high-level restaurant server that is initially contacted (which is a supernode in the peer network of restaurant servers) will present the user with information from the restaurant ontology. One way of interfacing to the user is through a web-based form; however, direct queries can also be issued if the system is aware of the OWL ontology beforehand, as described in Section II. A query is constructed using information from the class properties. Some restaurant properties are: hasCuisine, hasNeighborhood, hasRating and hasPriceRange. The user fills out its preferences for each property. Since there are many restaurant servers that store similar information, there are two possible ways of issuing the query. One way is for the query to be sent to all of the peer nodes. This is inefficient considering some nodes may not contain any of this information and thus sending it to those servers is futile. A better way is to convert the query data to a key and search for the server within a CAN network. We adopt the latter approach and discuss it below.

VI. Generating CAN overlay networks

A. Overview

The CAN architecture is generated as a network of n-level overlays, where n is the number of subclasses nested

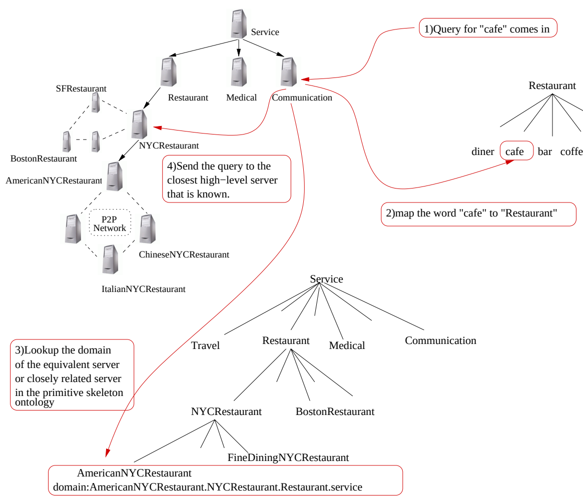


Fig. 5. Finding servers in GloServ

within the main class. Let us take the *Restaurant* class and its subclasses as our example. The ontology classification and the CAN overlay network generated is seen in Figure 6. The first CAN overlay is a d-dimensional network which has the first level of subclasses of the *Restaurant* class. The number of dimensions is determined by the number of nodes contained within the CAN. [4] states that in order to achieve a query hit within $O(\log_2 n)$ number of hops , $d = (\log_2 n)/2$. Thus, we approximate the number of nodes necessary for the network and calculate the dimension accordingly. This can be done since each CAN handles a certain service class which limits the number of nodes from millions to thousands or even hundreds of nodes. However, if a CAN needs millions of nodes, we simply fix the dimension to a reasonable size and the worst case number of hops will then be $O(n^{1/d})$.

Each node will hold instances of a set of classes. During service registration or querying, a *Restaurant* query class is created and classified. If it is classified under *NYCRestaurant*, then the instance will be sent to the node that contains *NYCRestaurant* instances. Once it reaches this node, it looks at the ontology of *NYCRestaurant* and classifies it further. If the query class is classified under the *NYCRestaurant* subclasses, it is routed to the subnetwork of *NYCRestaurant*, otherwise, it remains in the *NYCRestaurant* server.

B. CAN Generation with Restricted Subclass Dimensions

The CAN network initially starts with a node that handles all *Restaurant* instances. The *Restaurant* class will have a set of restricted subclasses. These are classified with a reasoner, such as Racer. The classification will produce a new ontology which will cause related siblings to form relationships with each other. These relationships include superclass, subclass or equivalence relations. Once the siblings are classified, they are assigned consecutive numerical keys. This insures that related siblings will have keys assigned close to each other. The result is that nodes resid in adjacent zones within the CAN and cluster together.

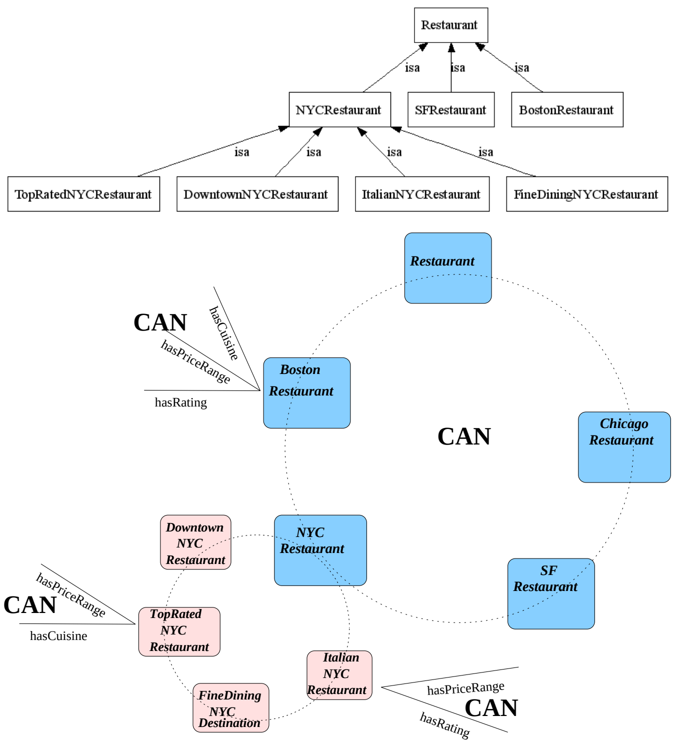


Fig. 6. CAN overlay network

As services register within the node and instances are created, they are classified into the subclasses of *Restaurant*. When a new node joins the network, one of the CAN dimensions is split into two and data is 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 its range 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: $[0-4]$, $[0-9]$, $[0-9]$.

We can improve the load balance and routing within the CAN by paying attention to the query and registration messages. For better load balance, instead of choosing a random point to route to within the CAN, as stated in [4], we can send it to an overloaded node. A subset of nodes are designated supernodes of the CAN network so that one of these can be contacted initially when a new node enters into the system. Query and registration messages pass through the supernodes. Thus, the supernodes can monitor the amount of queries and registrations routed to each subclass. This information will be shared among all the supernodes so that all the supernodes will have the same view of the network state. Those subclasses which exceed a certain threshold value of queries and registrations are considered overloaded nodes. Thus, the new node entering into the system will be routed to the node which is the most overloaded.

We can achieve faster routing of messages by again using the query and registration information. Similar to when a new node enters into the system, the supernodes monitor the

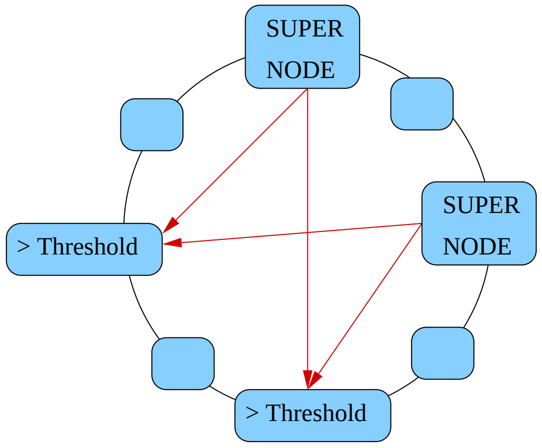


Fig. 7. Supernodes pointing to frequently visited nodes

number of queries and registrations that each subclass receives. These values are periodically refreshed because there are different services queried for in various periods of time. For example, in the case of the *Travel* service, each season during the year will produce a surge in specific types of destinations. The supernodes check the subclasses that exceed a certain threshold value and cache the URIs of the nodes containing these subclasses. In this way, nodes that are queried most will always be one hop away from the supernodes, resulting in query hits of minimal time. Figure 7 gives an overview of this.

C. *CAN Generation with Property Dimensions*

When a class does not have any restricted subclasses, we can distribute the instances by generating a CAN where each dimension represents a property. Similar to INS/Twine [10], we manipulate the property-value pairs of the instances. However, INS/Twine hashes simple XML attribute-value pairs onto a Chord ring whereas GloServ exploits the logical benefits of OWL DL in converting the data to a key within CAN. We analyze the properties and their values so that instances that contain similar information will migrate together. There are two basic types of properties in OWL: object properties and datatype properties. Object properties have ranges that are other classes. Thus the object property maps two classes together either unidirectionally or bidirectionally depending on the property. A datatype property on the other hand, maps classes to traditional datatypes such as strings and integers.

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

Next, we analyze the datatype properties. Datatype properties are used in a limited way. Since datatype properties can have any value, it results in an unbounded limit to the number of keys that can be generated. Thus, the only way we include datatype properties in the key generation is to see if an optional datatype property is populated. The presence or absence of this datatype property can be part of the key value.

The number of possible values of mandatory object properties is the product of their cardinalities. For optional object properties, the cardinality is incremented by one due to the possible blank value. However, for the optional datatype properties, since there is no concrete value of cardinality, the only part that counts is whether or not it is present. Thus, if we let p_i be the number of possible values for the i^{th} property and there are *l* mandatory object properties, *m* optional object properties and *n* optional datatype properties, then the total number of combinations of property values is:

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

It is recommended that classes that have CAN networks generated via their property values be tighly restricted classes. Therefore they only have a small subset of properties assigned to the CAN dimensions. This avoids exponential runtime when calculating the query key combinations. Taking Figure 6 as an example, the class *TopRatedNYCRestaurant* will already have its hasRating property restricted. Thus, this leaves the properties hasPriceRange and hasCuisine which are not restricted. Therefore, the CAN has 2-dimensions where each dimension is assigned one of the properties. If a class has no restricted subclasses and many properties, it is advised that the ontology be redefined where class restrictions can be formed in order to create an efficient CAN network. Figure 8 shows how servers are distributed in a CAN for the *TopRatedNYCRestaurant* class using the generated keys. We make a CAN using the example above and focus on a 2-property class for simplicity. The grid is partitioned into various spaces where each server handles a particular property combination.

VII. GloServ Registration and Querying

Service registration and querying use very similar mechanisms to find the correct node. The main difference between the two is in the propagation of the request within the CAN. Since we anticipate that there will be many more queries issued verses service registrations, we limit the number of hops necessary for queries by registering services within all matching nodes. This has a two-fold benefit: it insures that data is appropriately replicated across the network, and it allows for far fewer query hops. When a query is issued, we know that the first matching node will have the complete data set for that particular query restriction and thus further nodes need not be traversed. Below we explain how registration and querying is done in the two types of CAN networks.

A. *Querying a CAN 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 5. When the correct Gloserver

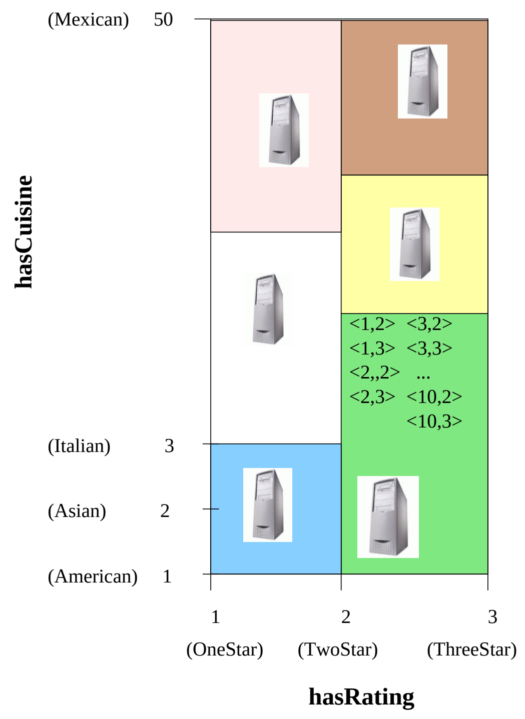


Fig. 8. Gloservers in a CAN

is contacted, the user agent obtains the ontology pertaining to that service class. As previously mentioned in Section II, 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 logical statement that contains restrictions on various properties. Below is an example query:

($hasLocation$ **some** NYC) and ($hasCuisine$ **some** (Italian **or** Chinese))

Continuing with our *Restaurant* class example, 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 *ItalianNYCRestaurant* 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 generation described in Section VI-B.

When registering a service with this restriction, all the nodes containing the *ChineseNYCRestaurant* and *ItalianNYCRestaurant* classes are searched and the service is registered in all the nodes. For example, let us say that the *NYCRestaurant* ontology has 30 subclasses; it is separated into 3 dimensions with 10 subclasses in each dimension. Furthermore, let us assign *ChineseNYCRestaurant* to dimension 0 with key 0, and *ItalianNYCRestaurant* to dimension 1 with key 1. A query will consist of 3 values for each dimension, $[d_0, d_1, d_2]$, where d_i represents the key value at the ith dimension. If one of the values is a * it is a wild card which means the whole dimension must be traversed. The queries for *ChineseNYCRestaurant* and *ItalianNYCRestaurant* are $[0, *, *]$ and $[*, 1, *]$ respectively. Thus, once we find the first node that contains the key 0 in dimension 0, for *ChineseNYCRestaurant*, we send the query to all the neighbors in the other two dimensions and search for key $[0, *, *]$. The nodes that contain this key, will have this service registered at that node. Otherwise, the 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.

B. 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 within a CAN 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 *ItalianNYCRestaurant* and *ChineseNYCRestaurant* 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.

Registration follows a similar manner. However, for the above example, since a price range and rating is not specified, the instances are stored in the main *ItalianNYCRestaurant* and *ChineseNYCRestaurant* servers. But it is recommended that these propeties be populated mandatorily. In that case, when a price range and rating is specified, the keys for these two properties are generated, the node handling that key is found and the instance is registered within it.

VIII. Implementation and Future Work

Currently, we are implementing a prototype of GloServ using Protege [21] and Racer [17]. 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 ontolgies 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 —¡—dimension, key—¿— 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 assign themselves to a zone. Each server initially holds many classes but as the number of nodes increase, the number of classes dwindle to one per dimension. Once a class exceeds its threshold registration, it checks with other servers that handle the same class to see if a CAN subnetwork has already been formed. If it has, it transfers its data to this subnetwork. Otherwise it processes its pre-configured ontology to generate a CAN subnetwork, transfering 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. The query follows the same steps except that it stops once it finds the first node that holds the information it is looking for, as it does not need to look further. Figure 8 shows screens shots of the GloServ registration process.

We plan on designing the second phase of the GloServ. Our focus is on creating context-aware and event-based applications. We are looking at ways to automatically issue queries in context-aware environments. Additionally, we are exploring ways to incorporate event-based services in a dynamically changing service network.

IX. Conclusion

We have described a hybrid hierarchical and 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.

X. Acknowledgment

This work is supported by a grant from Nokia Research Center. We would also like to acknowledge the contributions of Dirk Trossen and Dana Pavel from Nokia Research.

XI. Bio

Knarig Arabshian, is a PhD Candidate in the Department of Computer Science at Columbia University, New York. She currently works in the Internet Real Time Laboratory with Prof. Henning Schulzrinne. Her research interests include service discovery, ubiquitous computing, context-aware systems, semantic web technologies, and distributed computing.

Prof. Henning Schulzrinne received his Ph.D. from the University of Massachusetts in Amherst, Massachusetts. He was a member of technical staff at AT&T Bell Laboratories, Murray Hill and an associate department head at GMD-Fokus (Berlin), before joining the Computer Science and Electrical Engineering departments at Columbia University, New York. He is currently chair of the Department of Computer Science. Protocols co-developed by him, such as RTP, RTSP and SIP, are now Internet standards, used by almost all Internet telephony and multimedia applications. His research interests include Internet multimedia systems, ubiquitous computing, mobile systems, quality of service, and performance evaluation. He is a Fellow of the IEEE.

References

[1] K. Arabshian and H. Schulzrinne, "Gloserv: Global service discovery architecture," in *First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (Mobiquitous)*, Aug. 2004. [Online]. Available: http://www.cs.columbia.edu/ knarig/gloserv.pdf

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

[3] OWL http://www.w3.org/2004/OWL/.

[4] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network." San Diego, CA, USA: ACM, Aug. 2001.

[5] E. Guttman, C. E. Perkins, J. Veizades, and M. Day, "Service location protocol, version 2," Internet Engineering Task Force, RFC 2608, June 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2608.txt

[6] S. Microsystems, "Jini architectural overview," Tech. Rep., 1999.

[7] W. Zhao and H. Schulzrinne, "mSLP - mesh-enhanced service location protocol," Columbia University, New York, Technical Report CUCS-013-00, May 2000. [Online]. Available: http://www.cs.columbia.edu/

[8] U. Forum, "UPnP device architecture 1.0," Tech. Rep., Dec. 2003. [Online]. Available: http://www.upnp.org/resources/documents/CleanUPnPDA101-20031202s.pdf

[9] U. D. DI, "UDDI technical white paper," UDDI (Universal Description, Discovery and Integration)," White Paper, Sept. 2000. [Online]. Available: http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.PDF

[10] M. Balazinska, H. Balakrishnan, and D. Karger, "Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery," 2002. [Online]. Available: citeseer.ist.psu.edu/701773.html

[11] S. D. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, and R. Gummadi, "The ninja architecture for robust internet-scale systems and services," vol. 35, no. 4, pp. 473–497, Mar. 2001. [Online]. Available: http://www.elsevier.com/locate/comnet

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

[13] K. Aberer, P. Cudre-Mauroux, and M. Hauswirth, "A framework for semantic gossiping," 2002.

[14] Gnutella, http://gnutella.wego.com.

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

[16] D. Brickly and R. Guha, "Rdf vocabulary description language 1.0: Rdf schema," World Wide Web Consortium," W3C Proposed Recommendation, Feb. 2004.

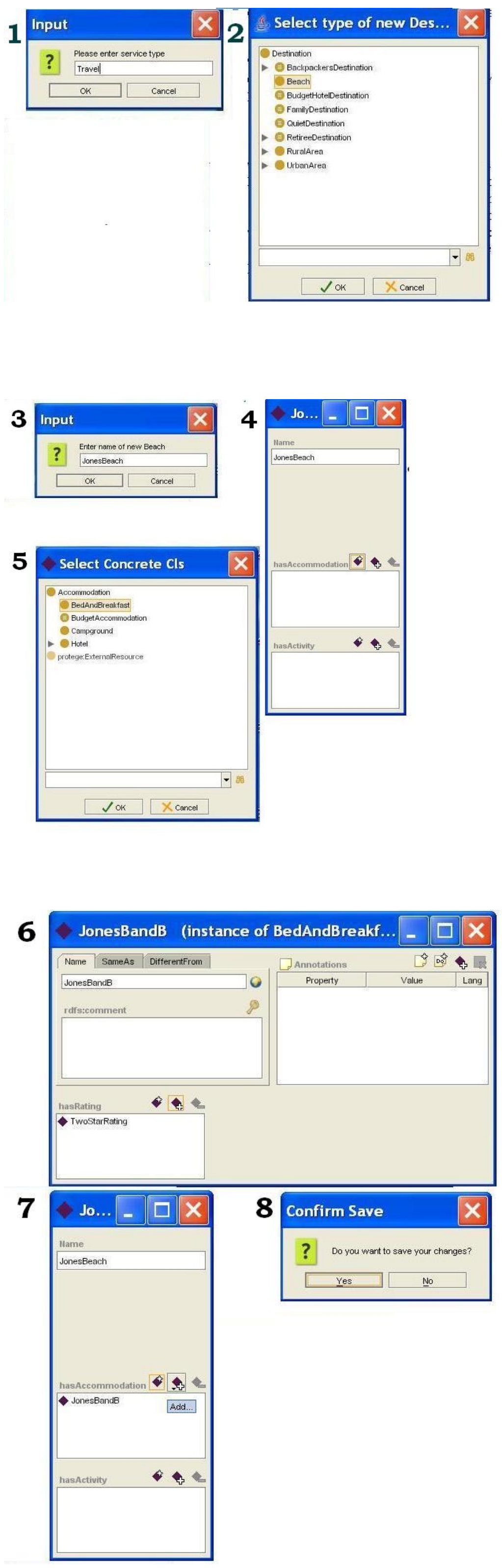
[17] V. Haarslev and R. Moller, "Racer user's guide and reference manual version 1.7.19," 2004, concordia University, Tehcnical Universityh of Hamburg-Harburg, University of Hamburg. [Online]. Available: http://www.sts.tu-harburg.de/ r.f.moeller/racer/racer-manual-1-7-19.pdf

[18] M. Horridge, A. Rector, N. Drummond, H. Knublauch, and H. Wang, "A user oriented owl development environment designed to implement common patterns and minimise common errors," in *3rd International Semantic Web C3onference (ISWC2004)*, Hiroshima Prince Hotel, Hiroshima, Japan, Nov 2004.

[19] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001, pp. 329–350. [Online]. Available: http://research.microsoft.com/ antr/PAST/pastry.pdf

[20] I. Stoica, R. Morris, D. R. Karger, F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for Internet applications." San Diego, CA, USA: ACM, Aug. 2001.

[21] J. Gennari, M. 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," Stanford University, Tech. Rep., 2002.



Please enter service type
Travel

Enter name of new Beach
JonesBeach

Name: JonesBeach

Name: JonesBandB

hasRating: TwoStarRating

Name: JonesBeach

hasAccommodation: JonesBandB

Do you want to save your changes?

Fig. 9. GloServ Registration Screen Shots