A Hybrid Hierarchical and Peer-to-Peer Ontology-based Global Service Discovery System

Knarig Arabshian and Henning Schulzrinne

Department of Computer Science
Columbia University, New York NY 10027, USA

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 description logic Web Ontology Language (OWL DL). 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,

1 Introduction

GloServ [3] [4] is a global service discovery architecture that addresses the need for wide area service discovery for ubiquitous and pervasive computing. It operates on wide area as well as local area networks. There are many types of services that can be registered within GloServ. A partial list of these include events-based, physical location-based, communication, e-commerce or web services. The main motivation for designing a global service discovery architecture using a description logic ontology such as OWL DL [1] is to provide scalability and logical service descriptions. Services, should be searchable from any location and should also be described in enough detail so that queries can be specific to a user's context. This is the primary challenge in pervasive and ubiquitous computing technologies and GloServ provides the means to accomplish these tasks.

The service discovery problem can be split into three main components: server bootstrapping, automated and intelligent routing and matching of service registrations and queries. These need to occur in a robust and scalable environment. Services can also be expressed in greater detail using OWL DL, which is a description logic ontology that uses first order predicate logic that describes relationships between services. GloServ addresses these issues by using the benefits of these logical descriptions, within the service classification ontology, to solve the three main service discovery problems.

In [3] we gave an initial outline of GloServ and in [4] we described how services are described using OWL in greater detail. The details of the solutions to the system are fine tuned in this paper. GloServ classifies services using OWL DL which defines classes of

services and their relationships with other services and properties. We have designed a hybrid hierarchical and peer-to-peer network that exploits the knowledge obtained by this classification ontology as well as the content of specific service registrations. The hierarchical network is formed by connecting the nodes between the high-level services within the service classification. On the other hand, the peer-to-peer network is created between equivalent or related services by analyzing the content of registered services and mapping combinations of properties and values to keys which identify nodes within a Content Addressable Network (CAN) [16]. We also provide algorithms for automated and intelligent routing and matching of service registrations and queries by exploiting the logical relationships between concepts within the ontologies. Current service discovery systems fail to solve all of the above problems due to limited scalability and lack of detailed service descriptions. We address these issues in this paper.

Below we present our global service discovery system and discuss each of the problems and their corresponding solutions. Section 2 gives an overview of current service discovery systems. We describe the ontological engineering approach used in Section 3. Sections 4, 5 and 6 discuss the architecture, querying and registration mechanisms of GloServ respectively. The implementation of the system is described in Section 7 and finally we conclude in Section 8.

2 Background and Related Work

2.1 Overview of OWL

The World Wide Web Consortium has recently approved OWL [1] as a standard for the Semantic Web. OWL builds on Resource Description Framework (RDF) [2] and RDF Schema [6] 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 to check for the soundness of OWL documents.

2.2 Related Work

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 [12] and Jini [15] 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) [20], can span to a larger enterprise networks.

UPnP [9] 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) [7] 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 [5] 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.

3 Ontological Data

There are many ways to compose ontologies. [14] and [17] describe a few. [17] 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 [17], 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 1 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.

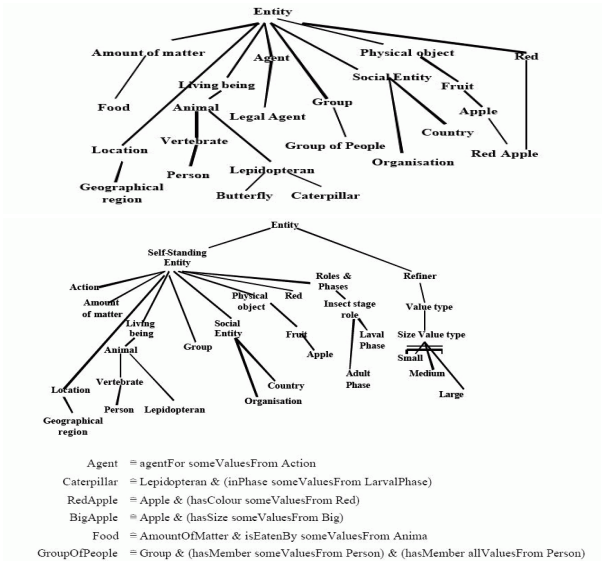

Fig. 1. Original classification ontology converted to corresponding primitive skeleton

4 GloServ Architecture

4.1 Motivation

The GloServ architecture consists of servers connected as a hybrid network of hierarchical and peer-to-peer nodes. The high-level services are established in a hierarchical format. The hierarchical primitive skeleton ontology is used for separating these high-level services. Since it is expected that there will be a limited number of high-level servers, each will know about other high-level servers. By using the primitive ontology model, any server will be 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 a service classes that are completely unrelated to each other.

Besides organizing high-level servers, we need to establish servers that are related to each other. Servers who hold information about the same or equivalent service class, will be connected to each other in a peer-to-peer network. The motivation for using a peer-to-peer network is for load distribution during query processing. Querying is faster when the 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 you replicate the registrations across all servers, there is no need for a structured peer-to-peer system. However, the servers will hold so much content that query processing will be slow for a single server. Thus, a peer-to-peer network establishes a robust and scalable environment for querying. Figure 2 shows a partial view of the GloServ architecture.

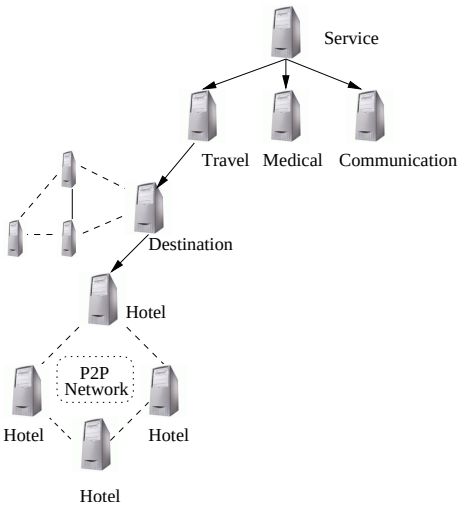

Fig. 2. GloServ architecture

4.2 Elements within a gloserver

Gloservers within the network have three types of information: a service classification ontology, a thesaurus ontology and a CAN lookup table. The service classification ontology is similar to the one seen in Figure 1. As mentioned above, this classification is not prone to frequent changes and thus can be distributed and cached across the GloServ network. Each high level service will have a set of properties that will be inherited by all of its children. As the subclasses are constructed, the properties become specific to the particular service type

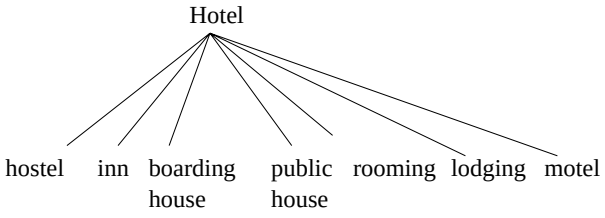

Fig. 3. Partial view of a thesaurus graph containing synonyms of “hotel”

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. Synonyms of every class within the service classification hierarchy will be stored. Figure 4 gives an example of the partial graph of the synonyms of *Hotel* 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 data within each class. In this case, data is an actual instance of a class. Each instance represents a registered service. The CAN table connects servers of the same type to each other in a peer-to-peer network. We use a novel mapping algorithm that combines the benefits of OWL and CAN to map content of service instances to nodes in a peer-to-peer network. Although there are other types of structured peer-to-peer networks such as Pastry [18] and Chord [19], we have elected to use CAN because it fits best with our ontology-based service discovery model. Section 4 explains why we have elected to use CAN over the other structured peer-to-peer networks.

4.3 Server Bootstrapping

When service providers register or users query for services, a gloserver is found using DNS lookup and contacted. The first piece of information a user needs to provide the appropriate gloserver is the type of service it is looking for either in registration or querying. Let's assume a user is querying. When the user enters the word *inn*, the initial server processing the query will first map the word inn to a synonymous term within the service classification ontology. In this case, it is mapped to *Hotel*. The server locates hotel in the primitive service classification ontology and determines the domain 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 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 hotel node's domain name is determined, it is contacted with the user's query. The hotel node will have service registrations stored in it. These are actual instances of the hotel class. Many instances of hotels are stored here and thus the information will have to be distributed across other hotel servers that are connected to each other in a peer-to-peer fashion. This is where CAN is used.

The main high-level hotel server that is initially contacted (which is the supernode in the peer network of hotel servers) will present the user with the hotel ontology skeleton. Using a web-based form is one method, however, the form is converted to an OWL file in the end. Thus, any automated program can query by requesting the OWL template from a server and automatically creating an OWL file and sending it in for querying or registration. This is constructed using information from its class properties. Some hotel properties properties may be, hasLocation, hasAccommodation, hasActivity. The user fills out the mandatory property values (if there are any) and possibly other values. At this point, since there are many hotel servers that store similar information, again there are a 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 look up the server within a CAN network. We adopt the latter approach and discuss it below. Figure 6 gives an overview of the steps to find the main high-level server.

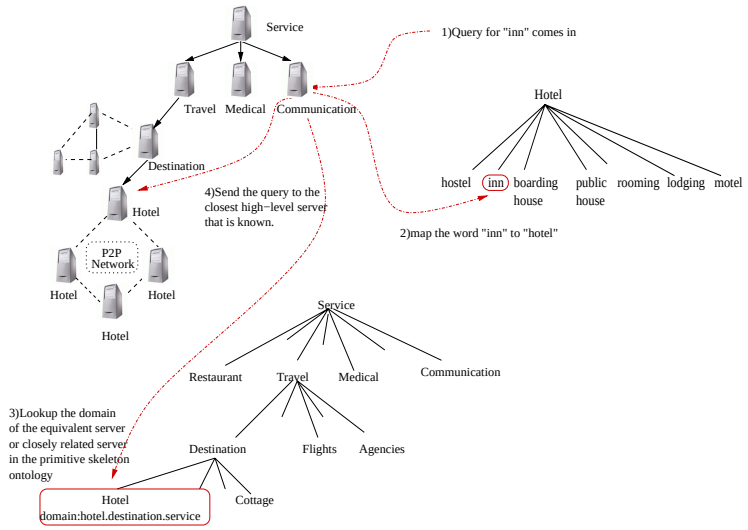

Fig. 4. Finding servers in GloServ

4.4 Analyzing OWL Instances

We have developed an algorithm for converting the OWL instance data to keys that map to servers in CAN. Similar to INS/Twine [5], 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. We do not include mandatory datatype properties since these will always be present and thus there is no need to include this in the key generation process. Including the unstructured values of datatype properties in the key generation is an area of future research we are looking into.

We analyze all the possible combinations of the three types of properties: mandatory object properties, optional object properties and optional datatype properties. 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:

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

These values give us a way to organize the data in such a way that keys will be generated for every possible combination and mapped to a server in CAN. The fact that the data distribution is based on content is an added benefit since the propagation of the query becomes limited to a cluster of servers that know about each other. The next two sections describes how these keys are devised and mapped onto CAN.

4.5 Converting OWL Instances to Vector Keys

As discussed above, there are a set number of property combinations the OWL instance can have. Every data object property will have a class as its range. The range of values can be enumerated into number values. By enumerating the range of values, the combinations of values entered in a query or registration can be converted to a vector key. For instance, if the property is hasActivity and the class Activity has the following subclasses: Sports, Adventure, Hiking, Sightseeing, then the ontology will specify a numeric value for all of these as well via a hasKey property. The following OWL code shows this.

```
<owl:Class rdf:ID="Sports">
  <rdfs:subClassOf rdf:resource="#Activity"/>
  <hasKey>1</hasKey>
</owl:Class>

<owl:Class rdf:ID="Adventure">
  <rdfs:subClassOf rdf:resource="#Activity"/>
  <hasKey>2</hasKey>
</owl:Class>

<owl:Class rdf:ID="SightSeeing">
  <rdfs:subClassOf rdf:resource="#Activity"/>
  <hasKey>3</hasKey>
</owl:Class>
```

This will be done for all the object properties in the class. For example, let us assume we are dealing with mandatory object properties, then if the hasActivity object property op_1 has three possible values and the hasState object property op_2 has fifty possible values then you can generate the vector: <op_1, op_2> where op_i represents the key of that particular property. For this case, we get $50*3 = 150$ possible values. Figure 5 gives an overview of the whole key generation process. For optional object properties, a 0 key is added to represent the blank value which results in $51 * 4 = 204$ keys and the vector set includes the additional values of: <0,0>, <0,1>, <0,2>, <0,3>,<1,0>,<2,0>,...,<50,0> .

4.6 Mapping Vector Keys to CAN

We found that CAN was the most appropriate peer-to-peer network to use for our system. Both exact and approximate matching are possible by using CAN. The generated vector keys, are distributed in a CAN. However, instead of using random keys for each dimension, we use the generated keys by using a property per dimension for the d-dimension key, where d defines the CAN structure.

When vector keys are mapped to CAN, both approximate and partial querying is simplified. For example, in a d-dimensional CAN network, if a user enters in the following keys: x=1, z=17 and y=blank where x, y and z represent object properties, the node at x=1 and z=17 is located and the query then propagates to all nodes in the y dimension. If y is a datatype property, then all the data within the node where x=1 and

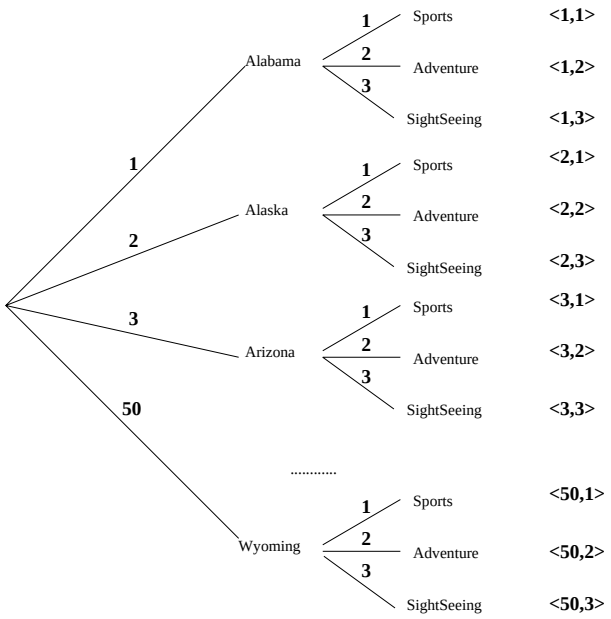

Fig. 5. Keys generated by properties

z=17 is queried for with the condition that x=1 and z=17. If there is no result from this search, then classes that are ontologically related to 1 and 17, which may be outside of the CAN, are searched. If x and z are both datatype properties then all the nodes within CAN are searched up to a threshold value.

We expect that the number of servers for every service type will be at most in the 100s since the peer-to-peer network of servers is handling one service type. CAN has a runtime of $O(n^{1/d})$ which works well with our model considering that each class will have the number of properties in the order of 10s and maximum 100s. Figure 6 shows how servers are distributed in a CAN 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.

5 GloServ Querying

When a user issues a query in GloServ, the query will be first handled by any gloserver. As mentioned above, the user will first choose the type of service it is looking for. Once the server that is most likely to have the service the user is searching for is contacted, the query is distributed among the peer servers of that service type according to the key values. Below we describe the query propagation in greater detail.

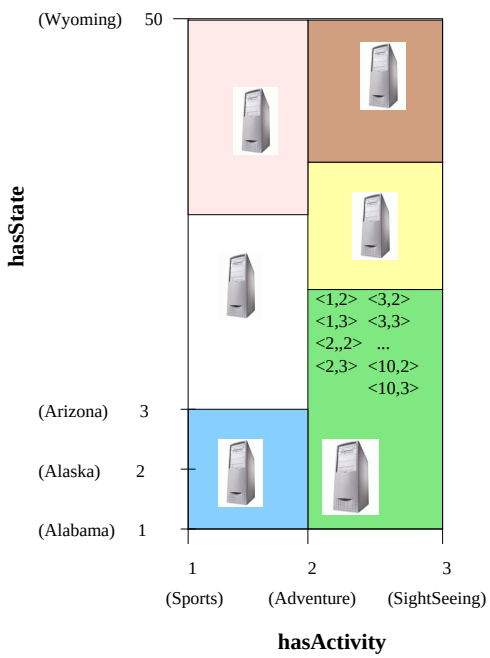

Fig. 6. Gloservers in a CAN

5.1 Query Propagation

When the correct gloserver is contacted, it obtains the user query. Query input is guided by the service class's OWL ontology. As mentioned before, query input can be done either through a user form, or by automatically filling out an OWL ontology skeleton. We anticipate that GloServ will be used in context-aware and pervasive computing environments where a user's preferences can be detected and user input can be relied on in order to get an accurate query. Thus, we employ a similar method to OWL-QL [8] where the user can indicate when it is satisfied with the query results. If it is satisfied, the query propagation terminates. Otherwise, it continues sending either the same or modified query until all results are found. If the query is not human-centric, then query propagation automatically travels to all possible routes up to a threshold value.

If the user was querying for hotels, the hotel server would send a form of all its properties so that the user could specify values it is querying for. If the hotel server is a leaf node within the service classification hierarchy, then it is apparent that the query will only remain in its immediate peer network of other hotel servers. However, if it has further children or related siblings that hold similar information to it, the query is sent to these related nodes as well. This is described in greater detail in Section 6.1 where service registration is discussed.

The user will fill out any number of fields in the form. The general fields that are filled out are object properties such as the location or the activity. As long as users specify a few object properties, then the query will be directed to the correct server in the CAN peer-to-peer network. Otherwise, if only datatype properties are issued, then all servers within the CAN network will be contacted.

When the hotel server receives the data, it analyzes the property-value pairs that have been filled out and generates a set of keys that evaluate to the query's combination. It then looks up the keys and maps them in CAN to find the servers that may be able to handle this query or have information on another server.

5.2 Query Matching

Once the appropriate server for the query is found, the query is analyzed by first looking at all the properties that have been populated. The exact query combination is generated as well as all possible combinations. For instance if a user is querying for *Sports* activity in *Arizona* then the following is generated: <3,1>. This key is mapped to the server that will handle these properties. If a result isn't found, then approximate matching is done. Keys of geographically nearby locations and related activities to sports are generated. All combinations of these are queried for to give the user an approximate result. Related keys are obtained by finding all classes that are related to the property value being queried for. This is described in greater detail in Section 6.1.

The algorithm first performs an exact matching and then further filters this by the matching the datatype values in the query by using text matching. It presents the data to the user in the order of most to least accurate. If the user is still not satisfied, the query continues to propagate to all related servers until either the number of servers are exhausted or the user is satisfied. As mentioned above, if the mode of operation is not in a human-centric interface, then the query propagation will end after a threshold value of n servers have been searched. The pseudocode below goes through these steps:

Query Matching Algorithm

```
Store the populated properties in a list of <Name,Value> pairs
Generate keys for the property-value combinations
/* Exact Matching */
Send the query to the server that contains an exact key matching
for all instances that match exactly with the object properties do
   Filter further by text-matching the data type properties
end for
Prioritize query results in order of most to least accurate
Send results to user
if  user satisfied then
   end query propagation
else
   /*Approximate matching */
   Evaluate all combinations of keys of related values to the query
   while (user is not satisfied) and (there are keys to process) do
```

Send query to a server that matches the key
for all instances that match exactly with the object properties **do**
Filter by text-matching the data type properties
end for
Prioritize query results in order of most to least accurate
Send results to user and obtain response
end while
end if

6 GloServ Registration

Registration of services is similar to the processes mentioned above. The first problem is finding the appropriate server to register in. The user contacts the nearest gloserver and as mentioned above, enters a type of service it wants to register for. The term is mapped to the synonymous term in the network and the domain name of the high-level server of that service term is obtained and contacted. If the node contacted is a leaf node, the service will only be registered at that node. Otherwise, the user will be presented with the option of registering simultaneously in nodes that are similar to the initial node contacted.

6.1 Determining class of servers

For registration in a leaf node, the process of registering a service is very similar in the query propagation section. The user's registration form is processed and keys are generated according to the user's input. This information is distributed to the nodes that carry related information. If registration is done in a node that is not a leaf node within the network, this means that a reference to the registration instance may need to propagate to other related servers such as the node's children or related siblings.

If the service class within the ontology has children, then it may also have gloservers that represent some of the children. Each class of servers has a specific ontology that specifies the domain names of its relatives: related siblings and children. Child servers represent subclasses within the ontology and they inherit all the properties from the parent server. They may also have restrictions on the inherited properties or have additional properties of their own. In order to determine if the registered service belongs in a subclass or a sibling class server, the values of the properties are analyzed to see if they match any of the restrictions of these nodes. If they do then a reference to that service is sent to these nodes. This is determined through the following algorithm:

Registration Matching Algorithm

for all populated properties in the registration form **do**
if the value entered is within a restricted property range of a particular subclass **then**
send a link of this service to that subclass
else if the value matches a sibling with a similar property range **then**

send a link of this service to the sibling
end if
end for

Example We will discuss an example shown through Venn Diagrams in Figure 5. The class *Destination* specifies possible travel destinations. It has the subclasses: *Back-*

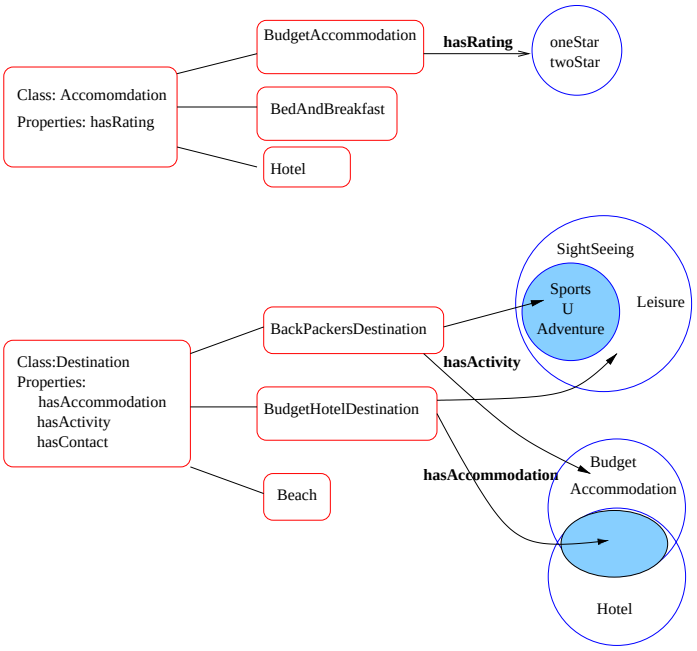

Fig. 7. Related classes and their properties

packersDestination and *BudgetHotelDestination*. The asserted necessary and sufficient conditions of the *BackpackersDestination* class are the following:

$$Destination \equiv BackpackersDestination$$
$$\exists hasAccommodation(BudgetAccommodation)$$
$$\exists hasActivity(Sports \cup Adventure)$$

Similarly, the necessary and sufficient conditions of *BudgetHotelDestination* are:

$$Destination \equiv BudgetHotelDestination$$
$$\exists hasAccommodation(BudgetAccommodation \cap Hotel)$$

It can be seen that the hasAccommodation property values of BackpackersDestination may contain some elements from the property values of *BudgetHotelDestination* since there may be budget hotels in a backpacker destination. Also, since there is no restriction on the hasActivity property in the *BudgetHotelDestination* class, there may be some budget hotels that also offer sports and adventure activities. In this case, these instances will also be registered under *BackpackersDestination*. Thus, services that intersect these two classes will be registered at both *BackpackersDestination* and the *BudgetHotelDestination* servers. The benefits of using the logical information of the ontology are that service instances are classified more accurately and query propagation time is saved due to this accuracy.

6.2 Instantiating a Registration

A service registration is an actual instance of a particular OWL class. The instance need not have all the properties populated. Thus, when determining how to instantiate the registration, as in the case above, a few methods may be used. One method is storing the instances in both servers. This is not efficient because if the service description changes, all servers holding an instance of that service need to be contacted. A better way of instantiating the services is to analyze the service instance and store it in the server whose class is the most restrictive. This server holds a copy of the service instance and the one that has least restriction will have a pointer to this service instance.

Example We will continue looking at the example in Figure 7. When a travel destination service registers, it is presented with all of these properties. The values of each of these are then analyzed and instantiated accordingly. Let us assume a service registration yields the following properties:

```
<hasAccommodation>
  <Hotel rdf:ID="BudgetHotel">
    <hasRating rdf:resource="#OneStarRating"/>
  </Hotel>
</hasAccommodation>
<hasActivity rdf:resource="#hiking"/>
```

We see that this service is part of the class *Hotel* and it has the accommodation rating set to one star. Since the *BudgetAccommodation* class is restricted to accommodations that have one or two star ratings, this hotel is also a part of *BudgetAccommodation*. Next, we notice that it also has an activity that is listed under the class *Sports*. Since this service satisfies both *BudgetHotelDestination* and *BackPackersDestination* classes, it will be listed under both. However, we will instantiate it in the *BackPackersDestination* server and have the *BudgetHotelDestination* reference this instance instead of storing the complete OWL file. Thus, logically distributing the data creates an efficient and automated environment.

7 Implementation and Future Work

Currently, we are implementing a prototype of GloServ using Protege [10] and Racer [13]. 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.

The registration component of GloServ has been completed. We have created our own service classification ontology, but are using a sample travel ontology provided, by the Protege group, to register travel services. We have created a primitive skeleton service classification ontology as well as a thesaurus ontology to map equivalences to the high-level services. Equivalence ontologies map various words to the main high-level servers accurately and services are registered under the correct class according to their registration description. We plan on completing the implementation on service querying in order to better test how accurate services are registered and queried for in GloServ.

Once the first phase of the GloServ prototype is completed, we plan on designing the second phase of the GloServ. Our focus will be on creating extensions for accessing services, having a service rating system, policy establishment, and security guidelines.

8 Conclusion

We have described a hybrid hierarchical and peer-to-peer global service discovery system using OWL DL. GloServ functions both on an wide area as well as a local area network. It applies to a broad range of services that are defined flexibly using OWL ontologies. Logic capabilities in OWL are used to distribute service content across nodes connected in a CAN peer-to-peer network. Service registration and querying are also done with greater speed and accuracy.

9 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.

References

1. OWL http://www.w3.org/2004/OWL/.
2. Resource Description Framework (RDF): W3C semantic web activity. http://www.w3.org/RDF.
3. 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.
4. 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.
5. M. Balazinska, H. Balakrishnan, and D. Karger. Ins/twine: A scalable peer-to-peer architecture for intentional resource discovery, 2002.
6. D. Brickly and R. Guha. Rdf vocabulary description language 1.0: Rdf schema. W3c proposed recommendation, World Wide Web Consortium, Feb. 2004.
7. U. D. DI. UDDI technical white paper. White paper, UDDI (Universal Description, Discovery and Integration), Sept. 2000.
8. R. Fikes, P. Hayes, and I. Horrocks. OWL-QL: A Language for Deductive Query Answering on the Semantic Web. Technical report.
9. U. Forum. UPnP device architecture 1.0. Technical report, Dec. 2003.
10. J. Gennari, M. A. Musen, R. W. Fergerson, W. E. Grosso, M. Crubzy, H. Eriksson, N. F. Noy, and S.-C. Tu. Evolution of protg: An environment for knowledge-based systems development. Technical report, Stanford University, 2002.
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. 35(4):473–497, Mar. 2001.
12. E. Guttman, C. E. Perkins, J. Veizades, and M. Day. Service location protocol, version 2. RFC 2608, Internet Engineering Task Force, June 1999.
13. V. Haarslev and R. Moller. Racer user's guide and reference manual version 1.7.19. Concordia University, Tehcnical Universityh of Hamburg-Harburg, University of Hamburg, 2004.
14. M. Horridge, A. Rector, N. Drummond, H. Knublauch, and H. Wang. A user oriented owl development environment designed to implement common patterns and minimise common errors. In *3rd International Semantic Web Conference (ISWC2004)*, Hiroshima Prince Hotel, Hiroshima, Japan, Nov 2004.
15. S. Microsystems. Jini architectural overview. Technical report, 1999.
16. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. San Diego, CA, USA, Aug. 2001. ACM.
17. A. Rector. Modularisation of domain ontologies implemented in description logics and related formalisms including owl. In *2nd International Conference on Knowledge Capture (K-CAP)*, Sanibel Island, FL, 2003.
18. 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)*, pages 329–350, Heidelberg, Germany, Nov. 2001.
19. 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, Aug. 2001. ACM.
20. W. Zhao and H. Schulzrinne. mSLP - mesh-enhanced service location protocol. Technical Report CUCS-013-00, Columbia University, New York, May 2000.