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
February 21, 2006

1 Introduction

GloServ is a global service discovery architecture that uses ontologies to create logically constructed service classes as well as automatically generates an artificially intelligent hierarchical-p2p network. It operates on wide area as well as local area networks. Any type of service can be defined within GloServ using the OWL ontology. Examples of services include events-based, physical location-based, communication, e-commerce or web services. Below we describe how GloServ works using the travel ontology as an example.

2 OWL Ontology Construct

Each service is defined within an OWL ontology. The ontology has a main service class which is defined by other classes within the ontology. For example, in the travel ontology, the main service class is the Destination class that has the properties hasActivity and hasAccommodation. These properties map the Destination class to the Activity and Accommodation classes. Thus the ontology contains the classes Destination, Accommodation, Activity, and others. However, only one class is designated the main service class where service providers will register to. OWL allows classes to be imported and resused. Thus, since the Accommodation and Activity classes may be used in other types of service descriptions, they do not necessarily have to be reside in the travel servers but can be imported. Figure 1 illustrates how the various classes within one ontology can relate to each other.

OWL uses annotated properties to indicate information about a class useful for documentation or processing purposes rather than for classification. In this case, we create annotated properties which describe how the class is to be parsed and interpreted for distribution in the CAN. Every service ontology in GloServ must contain these annotated properties. The annotated properties and their definitions are listed below:
mainClass: is an annotated property of the owl:Thing class which is the root of every OWL ontology. It specifies which class is the main service class in the ontology. In the case of the travel destination ontology, its main service class is the Destination class, which means a service that registers within this service class, registers as a type of destination.

distributeBy: is an annotated property of the mainClass class, which in this case is the Destination class. GloServ uses a CAN to distribute data. Every service instance belongs to a class and has a set of property-value pairs that it is described by. A node within a CAN holds a set of service instances which map to a certain class and subset of property-value pairs. As seen in Figure 1, the main class Destination maps to Accommodation and Activity via the hasAccommodation and hasActivity properties. This means that a service instance within the Destination class is described by accommodation and activity properties. Thus, we set two CAN dimensions to Accommodation and Activity and indicate this by setting the distributeBy property to these two classes. However, the Accommodation and Activity classes themselves have a number of properties. For instance, they have the hasLocation property which ranges to the City, State, and Country classes. A destination instance can be distributed by location as well as by accommodation and activity types. Therefore the distributeBy property has additional assignments to City, State, and Country classes. Figure 2 shows the Destination class with its properties.

subclassLevel: is an annotated property that gives an extra level of detail on the dimensions of the CAN. This property is set within all the distributeBy classes. Hence, since we have determined that the Accommodation, Activity, City, State, and Country classes are used for distribution, and since these classes may have nested subclasses and instances, the subclassLevel property defines which class level to go down to for distribution. If the value for subclassLevel is 0, then the dimension values are assigned to the instances of that class. If it is greater than 0, then the values are assigned to the classes within the level indicated
by `subclassLevel`. For example, Figure 3 illustrates class relationships of the `Activity` class. It has 2 nested subclasses. Since the `subclassLevel` property is set to 2, then the classes used for the CAN dimension values are the `BunjeeJumping`, `Safari`, `Museums`, `Yoga`, `Sunbathing`, `Surfing`, `Hiking` classes.

- `hasKey`: is an annotated property that assigns keys to the classes or instances that are actual points within a CAN dimension. CAN nodes handle a range of keys within a dimension. Traditionally, in CAN, keys are generated by hashing the string that is being searched for using a well-known hashing algorithm such as SHA-1. However, in the case of GloServ, we maximize the usage of logical information within an ontology and therefore distribute the instances in a unique way, which is by their classification information. However, we need to assign numerical values for each point within a dimension. Since the dimension consists of class names, we convert each class name to a numerical value.

Figure 2: Destination class with its properties

Figure 3: Examples of how Annotation properties are used
Thus, if a user searches for a destination which has an activity of Museums, the term is converted to the numerical key value that the Museums class is assigned to and searched for within the CAN using that number. Figure 3 shows how the hasKey property value is assigned in the designated subclass level of the Activity class.

- totalKeys: is an annotated property within the distributeBy classes to keep track of the number of keys each dimension contains. In Figure 3, the Activity class has a totalKeys assigned to 7 because there are 7 classes that will be assigned to the Activities dimension.

3 GloServ Network

GloServ combines both hierarchical and peer-to-peer architectures. The network architecture is shaped by service ontologies that are created by ontology engineers. The ontology engineers are experts of the particular service class and are aware of how best to classify and define properties of that service. However, as users begin to query and register for services within GloServ, the ontologies may evolve to reflect the social perceptions of a particular concept. Additionally, the queries issued are analyzed to see how the network can dynamically reshape itself to produce the results with greater speed and accuracy of specific queries. We discuss the GloServ network generation below.

![Figure 4: Primitive skeleton of high-level services](image)

3.1 Hierarchical Network

The hierarchical network is shaped by the high-level services within the whole service ontology. The high-level services represent basic primitive concepts that are arranged...
in a hierarchical tree where siblings are disjoint from each other. For example, the Travel class is disjoint from the Technology class. Further down the hierarchy, underneath the Travel class, the Destination class is disjoint from the Airlines class. However, these classes do share a set of “partitioning” concepts. For example, the Location class can be a partitioning concept for both the Destination and Airlines classes. Thus, it is necessary to have a set of ontologies that partition primitive concepts. Figure 4 shows an example of a primitive skeleton ontology.

Servers get bootstrapped into the network by accessing the primitive skeleton ontology which contains the hostname of each of the servers. Server hostnames will follow the hierarchical format. For instance, the Destination class’s URL will be Destination.Tavel.GlobalServices. As a server is assigned to a hierarchical network, it updates the ontology to include its server information.

3.2 Ontology-based CAN

3.2.1 Classifying Instances

Each service class will be connected with nodes that are similar to it in a peer-to-peer fashion. Thus, for the Destination class, when a service provider registers, it will be distributed within the Destination CAN according to how it is classified in the ontology. Let us take a closer look at the ontology describing the Destination class. As mentioned above, the Destination class has properties that indicate what kind of accommodation and activity it provides. Further, an ontology engineer will create many different subclasses underneath the Destination class which describe different types of destinations. Each subclass will be defined with a set of restrictions which is useful for classifying the destination instances. Figure 5 shows possible subclasses of the Destination class. Instances will be classified under the subclasses according to their restrictions. For example, the UrbanDestination and BeachDestination classes have the following restrictions:

\begin{itemize}
  \item UrbanDestination:
    \begin{itemize}
      \item hasActivity some Artsy
      \item hasActivity some NightLife
      \item hasAccommodation some not Campground
    \end{itemize}
  \item BeachDestination:
    \begin{itemize}
      \item hasActivity some Beaches
      \item hasActivity some Surfing
      \item hasActivity some Sunbathing
    \end{itemize}
\end{itemize}

When a service is registered within the Destination service class, an instance of Destination is created. The service instance has values for the hasActivity and hasAccommodation properties. If the activities fall under Artsy and Nightlife and the accommodation is not a Campground, then this instance is classified under the UrbanDestination class. In this way, all registered services will be classified under a certain
Figure 5: Destination class with its subclasses

subclass of Destination. If the instance can not be classified under any subclass, it remains under the Destination class.

Figure 6 illustrates how a Destination instance is classified under the UrbanDestination class due to its property values. In the diagram, nodes whose edges are labeled \( \text{isa} \) are “instances of” the class they are pointing to.

3.2.2 Generating CAN overlay networks

The CAN architecture is generated as a network of n-level overlays, where n is the number of subclasses nested within the Destination class. The Destination class and its subclasses is depicted as a CAN overlay network in Figure 7. The first CAN overlay is a \( d \)-dimensional network which has the first level of subclasses of the Destination class. The number of dimensions is determined by the number of subclasses the main class has. \[ \text{log}_2 n \] states that in order to achieve \( O(\text{log}_2 n) \) runtime, \( d = \text{log}_2 n \). Thus, if the number of nodes \( n \) equals the number of subclasses, then the number of dimensions \( d \) within the CAN will be calculated according to that formula. Each node will hold instances of a particular subclass. During service registration or querying, a Destination instance is created and classified. If a destination is classified under BeachDestination, then the instance will be sent to the BeachDestination node. If we use the UrbanDestination instance, in Figure 6, we see that it will be classified under Destination - UrbanDestination - NewYorkDestination. Thus, it will first be sent to the UrbanDestination node and the UrbanDestination node will then classify it under its NewYorkDestination node. If a class doesn’t have subclasses then its CAN network will be a \( p \)-dimensional CAN where \( p \) is the number of distributeBy properties annotated in the ontology.

The CAN network initially starts with a node that handles all Destination instances. As services register within the node, they are classified into the subclasses of Destination. The subclasses are assigned hasDestination and hasKey properties. If there are 3 dimensions, then the subclasses are separated into 3 parts where each part is assigned a destination 0, 1, 2 and subclasses within each part is assigned a consecutively numbered key. A certain number 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. When a new node contacts one of these nodes, a random key is generated for the new node and it is routed to the CAN node which holds that key. The node that holds this
Figure 6: Graphical view of a destination instance classified under *UrbanDestination*

key splits the dimensions and transfers instances of half of the subclasses to the new node. Initially we generate a CAN with a certain number of nodes. Once a certain size is reached, the supernodes are assigned to the nodes that are in the center of the CAN so that query or registration routing is done with greater speed. As the CAN continues to evolve, the supernodes are periodically reassigned to insure that they are in the center of the CAN.

Figure 8 gives a detailed view of the data each type of node in GloServ holds. The *Destination* node has a *Child Dimension* structure which holds node information about its subclass neighbors. The *UrbanDestination* node has information about its sibling neighbors and its subclass neighbors. Thus, it has *Sibling Dimension* and *Child Dimension* data structures. Since the *BeachDestination* node does not have any subclasses, it distributes its information within a CAN where each dimension is a property type. Therefore, it holds information about its siblings and neighbors of every property type within the *Sibling Dimension* and *Property Dimension* data structures. The *Property Dimension* structure holds node information for every dis-
tributeBy property in the ontology. In this case, there are 5 different types of properties City, State, Country, Activity, Accommodation. Each one can have many different range of values. For example, [1-1000, 1-500, 1-200, 1-100, 1-50] indicates that there are 1000 possible cities, 500 states, and so on. Every node handles a range of values for each property. Thus, a certain node within BeachDestination may handle the range [1-15, 20-50, 180-200, 1-10, 40-50]. This is indicated in the Property Dimension data structure for each property type. Further, a node will have left and right neighbors for every dimension. Thus, whenever a registration or a query instance is routed...
to the \textit{BeachDestination} subnetwork, the instance is routed within the \textit{BeachDestination} property-distributed CAN. Those nodes that handle the properties specified in the instance will process the registration or query instance.

### 3.2.3 Future work

For future work, 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 within the CAN to route the new node to, we can send it to an overloaded node. 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 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, those nodes queried for the most will always be one hop away from the supernodes and will return a fast result. Figure 9 gives an overview of this.

4 Registration and Query Routing

Service registration and querying use the same mechanism in order to find the correct node to register in. Thus, we will refer to both as a querying mechanism. Initially a user searches for a service term which is mapped to a class within the primitive skeleton ontology. We describe a user-centric query mechanism, but this can be automated as well. A form is generated from the specific ontology of that class which lets the user enter in values for the properties of that class. An instance is created and classified. As in the case of Figure 6, an instance is classified under the UrbanDestination subclass. Thus, this instance is forwarded to the UrbanDestination node. The UrbanDestination server classifies this instance again and sees that it belongs in the NewYorkDestination node and thus forwards it down once again. Since the NewYorkDestination node is a leaf node, its CAN is separated by property dimension. We already know that all instances within the NewYorkDestination node has the values City = NewYork, State = NewYork, Country = UnitedStates, thus the network is separated into its remaining properties: Activity and Accommodation. The values entered for these properties are converted to their numeric keys. Continuing with our example, query is further routed within the NewYorkDestination CAN subnetwork which is distributed by its properties.
The UrbanDestination instance has as its activities NightLife and Museums and the accommodation Hotel. If the keys for the activities are 0 and 6 respectively and the key for Hotel is 0, then the following query key strings are formed: [0, 0], [6, 0] which represents the combination: [NightLife, Hotel], [Museums, Hotel]. The key string is matched to the nodes that handle those keys. For instance, if Node1 handles an activity range [0-5] and accommodation [0-10] and Node2 handles activity range [6-10] and accommodation [0-30] then both nodes are considered hits and searched. The actual query is then administered in those nodes. If the query is

\[ \text{hasActivity some (NightLife or Museums)} \]
\[ \text{hasAccommodation some (Hotel)} \]

the query is reduced to:

\[ (\text{hasActivity Nightlife and hasAccommodation Hotel}) \text{ OR (hasActivity Museums and hasAccommodation Hotel).} \]

Thus, it is necessary to search for both Node1 and Node2 for instances that have either NightLife or Museum activities and Hotel accommodations in order to get all possible results.

Since calculating the query key string combinations has an exponential runtime, it is recommended that a leaf node handles the least number of properties as possible. Currently, we assume that the ontology is defined well enough where this will be avoided and each leaf node will handle a small number of properties so that calculating and issuing the query key strings will not result in exponential runtimes. However, for future work, we will look into how we can detect classes that need to be further broken down into subclasses. For example, if a node handles many properties and sees that this is causing many queries to be issued, then it will detect similar instances within the network and group it underneath a class. The criteria for evaluating “similarity” is unique to each class. An example of similarity is sharing a property value. If one type of property value is queried for often, then it makes sense to create a subclass containing a restriction with this property value and group all instances with this property value within that subclass. Any time a query comes in with that property value set, it is classified underneath that subclass and forwarded to that node.