Semantics-Based Resource Discovery in Global-Scale Grids

by

Juan Li

B.Sc., Northern Jiaotong University, 1997
M.Sc., Chinese Academy of Sciences, 2001
M.Sc., University of British Columbia, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

(Vancouver)

June 2008

Abstract

Grid computing is a virtualized distributed computing environment aimed at enabling the sharing of geographically distributed resources. Grid resources have traditionally consisted of dedicated super-computers, clusters, or storage units. With the present ubiquitous network connections and the growing computational and storage capabilities of modern everyday-use computers, more resources such as PCs, devices (e.g., PDAs and sensors), applications, and services are on grid networks. Grid is expected to evolve from a computing and data management facility to a pervasive, world-wide resource-sharing infrastructure. To fully utilize the wide range of grid resources, effective resource discovery mechanisms are required. However, resource discovery in a global-scale grid is challenging due to the considerable diversity, large number, dynamic behavior, and geographical distribution of the resources. The resource discovery technology required to achieve the ambitious global grid vision is still in its infancy, and existing applications have difficulties in achieving both rich searchability and good scalability. In this thesis, we investigate the resource discovery problem for open-networked global-scale grids. In particular, we propose a distributed semantics-based discovery framework. We show how this framework can be used to address the discovery problem in such grids and improve three aspects of performance: expressiveness, scalability, and efficiency.

Expressiveness is the first characteristic that a grid resource-searching mechanism should have. Most existing search systems use simple keyword-based lookups, which limit the searchability of the system. Our framework improves search expressiveness from two directions: First, it uses a semantic metadata scheme to provide users with a rich and flexible representation mechanism, to enable effective descriptions of desired resource properties and query requirements. Second, we employ ontological domain knowledge to assist in the search process. The system is thus able to understand the semantics of query requests according to their meanings in a specific domain; this procedure helps the system to locate only semantically related results.

The more expressive the resource description and query request, however, the more difficult it is to design a scalable and efficient search mechanism. We ensure scalability by reconfiguring the network with respect to shared ontologies. This reconfiguration partitions the large unorganized search space into multiple well-organized semantically related sub-spaces that we call semantic virtual organizations. Semantic virtual organizations help to discriminatively distribute resource information and queries to related nodes, thus reducing the search space and improving scalability. To further improve the efficiency of searching the virtual organizations, we propose two semantics-based resource-integrating and searching systems: GONID and OntoSum. These two systems address searching problems for applications based on different network topologies: structured and unstructured peer-to-peer overlay networks. Queries in the search systems are processed in a transparent way, so that users accessing the data can be insulated from the fact that the information is distributed across different sources and

represented with different formats. In both systems, ontological knowledge is decomposed into different coarse-grained elements, and then these elements are indexed with different schemes to fit the requirements of different applications. Resource metadata reasoning, integrating, and searching are based on the index. A complex query can be evaluated by performing relational operations such as *select, project,* and *join* on combinations of the indexing elements.

We evaluate the performance of our system with extensive simulation experiments, the results of which confirm the effectiveness of the design. In addition, we implement a prototype that incorporates our ontology-based virtual organization formation and semantics-based query mechanisms. Our deployment of the prototype verifies the system's feasibility and its applicability to real-world applications.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Dr. Son Vuong, for six years of wonderful guidance, inspiration, and support. He always encouraged me to follow my own interests and gave me the maximum freedom in my research. He helped me to grow as an independent researcher. Without his support, this work would not have been possible.

I am indebted to Dr. Norm Hutchinson for his invaluable technical, editorial, and moral support. His patience, kindness, and considerations for students have helped me through the most difficult time of my graduate study. He showed me that the best way to inspire and teach a student is to set a good role model.

I must add my appreciation for Dr. Vincent Wong, Dr. Charles Krasic, and Dr. Mike Feeley who kindly agreed to be member of my exam committee. Their comments are valuable to make this thesis better.

I would also like to thank Ms. Hermie Lam, Ms. Holly Kwan, and Ms. Joyce Poon, for all the help they have provided.

I would like to extend my thanks to all members of the NIC lab for the useful feedback and fun times. Many thanks to Ying, Wei, Billy, Stanley, Shahed, Nguyet, Iulian, Ricky, Anthony, and Xin. They have made the lab more enjoyable.

My graduate study would not have been so pleasant and memorable without the following friends, Qi, Wendy, Jane, Shaofeng, Kan, Fahong, Zhimin, Xiaodong, Peng, and Xiaojuan.

Most of all, I would like to thank my family for their unconditional love and support.

Chapter 1

Introduction

1.1 The resource discovery problem in global-scale grids

A grid is a hardware and software infrastructure that provides users with transparent access to the entire set of resources available on the system. Early grid efforts linked supercomputing sites in order to provide computational resources to a range of high-performance applications. The grid offers a model for solving massive computational problems by making use of the unused resources (mainly CPU cycles and/or disk storage) of large number of computers. The focus of grid computing on supporting computation across administrative domains sets it apart from traditional computer clusters and traditional distributed computing.

With the prevalence of PCs as well as high-speed network technologies, more and more inexpensive PCs, and various devices such as PDAs and sensors at the periphery of networks are joining the grid. Grid research is therefore shifting its focus from scientific computing to a pervasive, world-wide resource-sharing infrastructure. The "grid problem" is defined by Forster as [32]: "Flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources." This view emphasizes the importance of information aspects, essential for resource discovery and interoperability. The new generation of grids enables the sharing of a wide variety of resources, including hardware, software packages, knowledge information, licenses, specialized devices, and other grid services. These resources are geographically distributed and owned by different organizations. The fact that users typically have little or no knowledge of the resources contributed by other participants in the grid poses a significant obstacle to their use. For this reason, resource discovery is a vital part of a grid system, and an efficient resource discovery infrastructure is crucial to make the distributed resource information available to users in a timely and reliable manner. However, resource discovery in large-scale grids is very challenging due to the potential large number of resources, and their diverse, distributed, and dynamic nature. In addition, it is equally difficult to integrate the information sources with a heterogeneous representation format.

The provision of an *information service*, as currently envisaged by the grid community, is a first step towards the discovery of distributed resources. However, a large part of these efforts have been focused on "getting it to work," without directly addressing issues of scalability, reliability, and information quality [17]. For example, classical grids always use centralized or static hierarchical models to discover resources. The Globus Toolkit [128] is a famous example. Globus users can get a node's resource information by directly querying a server application running on that node, or querying dedicated information servers that retrieve and publish the resource information of the organization. Although

interactions between these information servers are supported, the general-purpose decentralized service discovery mechanism is still absent. To discover resources in a more dynamic, large-scale, and distributed environment, peer-to-peer (P2P) techniques have been used in resent research (e.g., [15, 46]). P2P systems offer many benefits, such as adaptation, self-organization, fault-tolerance, and load-balancing, but they also present several challenges that remain obstacles to their widespread acceptance and usage in grids: First, current P2P systems offer limited data management facilities; in most cases, searching information relies on simple identifiers or Information Retrieval (IR)-style string matching. This limitation is acceptable for file-sharing applications, but in order to support complex resource discovery in grids we need richer facilities for exchanging, querying and integrating structured and semi-structured data. Second, most P2P systems specialize in a single functionality, for example, music sharing. More work needs to be done to support the sharing of varieties of resources in grids. Moreover, designing a good search mechanism is difficult in P2P systems because of the scale of the system and the unreliability of individual peers.

This thesis seeks to provide a general solution to the above-mentioned problem. It proposes a framework to share and discover resources on an unprecedented scale, and for geographically distributed groups to work together in ways that were previously impossible. In the remainder of this chapter we introduce the resource discovery problem, identify a set of requirements for resource discovery solutions, summarize our thesis contributions, and present the roadmap of this thesis.

1.2 Requirements for resource discovery in large-scale grids

The recent development of Internet-scale grids poses significant and novel research challenges for resource discovery. An effective discovery solution for this environment should meet the following requirements:

Scalability: Because of the participation of PCs and devices such as sensors at the periphery of the Internet, the number of nodes and resources involved in a grid may grow from hundreds to millions. The increased size of grids raises the problem of potential performance degradation. Consequently, applications must be scalable in handling the vast number of nodes and resources involved in grid deployment. The system therefore must be capable of spreading load among the participants themselves, serving large number of users at a very low cost.

Expressiveness: Grid resources may be highly diverse: hardware (processor, memory, storage, and instruments), software, network bandwidth, data, knowledge, services, etc. Each of these resources may have many attributes; for example, a computer resource may have attributes such as operating systems, number of CPUs, and speed. Different resources belong to different organizations and have different usage policies. Appropriate data schemas must be defined for both resource providers and resource requesters. Resource providers need to describe not only the complex attributes but also the usage policies of the resources in grids, so that the resource information can be encoded, stored, and searched in

an efficient manner. Resource requesters need richer query interfaces to accurately describe what they want to discover.

Recall and precision: Recall is a measure of the ability of the system to locate all relevant resources, and precision is a measure of the ability of a system to present only relevant resources. In a global-scale grid, resources are represented with heterogeneous metadata. The same resource in different nodes can be defined differently. For example, one node may use the term *processor* to represent a computing resource, while another node may use *CPU* to describe it. On the other hand, the same term can also be used to represent different resources. For example, the term *mouse* might represent a computer mouse in one node, and rodent in another. There are other relationships between concepts that also need to be considered. For example, a query related to operating systems should be able to understand that *Windows*, *UNIX*, and *Macintosh* are different types of operating systems. It is important for the discovery scheme to determine the relationships among concepts and locate all related resources, however they are represented. The system therefore requires a highly expressive logic with sophisticated reasoning capabilities.

Robustness: PCs in the system are not as stable as servers in classical grids; with so many resources in a grid, the probability of some resource failing is high. Therefore, node and resource failures are the rule rather than the exception. In addition, resources and their status change frequently. The system should tailor its behavior dynamically and be able to provide reliable services in a highly dynamic environment, locating resources when they are present and meeting the requirements of the application.

Responsiveness/ freshness of information: Grid resources change much more frequently than web pages. Therefore, a grid resource discovery system should update the status of resources much more frequently than web search engines do, in order to guarantee the freshness of the resource information.

1.3 Our solutions and contributions

In this thesis, we present the design, implementation and evaluation of an effective framework that enables flexible resource discovery in large-scale grid environments. The discovery framework tries to meet all of the requirements outlined above. Here we briefly present the key techniques we employ to realize our proposed system.

We propose a semantic model that adds semantics to the discovery framework. In the semantic model, ontologies are employed to provide a formal conceptualization of domains shared by users. Our goal is that queries can be properly interpreted according to their meanings in a specific domain by considering relationships between concepts. To address the issues of using domain knowledge in the search process, the ontology model includes: (1) an expressive ontological representation to encode the resource metadata, (2) an effective mapping formalism along with corresponding reasoning algorithms to integrate heterogeneous ontology representations, and (3) a comprehensive semantic query evaluation scheme to process complex SQL-like queries. Using an ontology for vocabularies and schemas allows abstraction

over resources and other concepts and provides a very rich querying and discovery mechanism. Based on domain knowledge, an ontology-based search understands the *meaning* of a user's search request; therefore, it overcomes the complexities inherent in natural languages, such as synonyms and homonyms/polysemes, and guarantees the precision and recall of the search results. This semantic model appeared first in Li *et al.* [63].

However, the more expressive and detailed a query request is, the more difficult it is for the related search to be scalable and efficient. We investigate the applicability of a P2P system for discovering grid resources. Our discovery system treats each node as a peer providing a distributed lookup service that allows other participants to discover resources and maintain resource states. We improve the performance of this P2P-based search technique by effective topology formation, efficient indexing methods, scalable searching algorithms, and reasonable load-balancing schemes.

Without a crawler and a centralized index engine, finding particular resources in an Internet-scale system is like looking for a needle in a haystack – there is too much to explore. We make this problem much more manageable by partitioning the large unorganized search space into multiple well-organized sub-spaces, in which resources are semantically related. Then queries are executed only within semantically related sub-spaces. We call the sub-space as a semantic Virtual Organization (VO). We have proposed a hierarchical ontological model – the ontology directory – to facilitate VO formation. The ontology directory defines a hierarchy of categories of a domain of interest, which are used to represent the semantics of resources. The ontology directory is used as a yellow page or a "rendezvous" for nodes to find contacts with similar interests. The domain ontology of the directory does not need to be predefined; it spontaneously grows as interest in the network evolves. We implement the hierarchical ontology directory with a flat Distributed Hash Table (DHT) overlay, which transforms the ontology category entries into a set of numeric keys. It does so in a way that preserves the expressiveness of semi-structured data, facilitates an even data distribution through the network, and enables efficient location of ontology domains. This ontology directory idea was first propounded in Li *et al.* [64, 68].

To overcome the inherent load-unbalancing problem experienced by the directory overlay, we propose a comprehensive load-balancing algorithm that includes an adaptive load redistribution scheme as well as a dynamic routing-table reconfiguring scheme. It achieves load balancing by dynamically balancing the query-routing load and query-answering load respectively. This balancing algorithm originally appeared in Li *et al.* [70].

To efficiently discover resources inside VOs, we propose an ontological resource indexing, integrating, and searching framework: GONID. Existing DHT overlays [121, 90, 102, 88] and the maturing of Semantic Web technologies [8, 140, 137] form the foundation of GONID. GONID realizes the ontological model we propose through an efficient indexing scheme. It decomposes ontological knowledge into atomic elements and then indexes them with DHTs. GONID adds semantics to DHT indexing and builds complex query facilities on top of DHTs, while still maintaining the scalability of the DHT infrastructure. Ontological reasoning, integrating, and searching are all based on this index. A

complex query can be evaluated by performing relational operations such as *select, project,* and *join* on combinations of the indexed atoms. A key advantage of this ontological indexing scheme is its ability to index in different granularities, as we distinguish knowledge at different levels of abstractions. The resulting prototype system, GONID-toolkit, verifies the viability of this indexing and searching infrastructure. Results related to the GONID system originally appeared in Li *et al.* [63, 61].

Because GONID is constructed on top of a structured DHT overlay, it inherits some inherent DHT problems, such as sensitivity to churn, an inability to control the index location, and difficulty in finding ontologically related peers. To overcome these problems, we investigate a design alternative and propose another search framework, OntoSum. OntoSum utilizes the small-world research results and tries to construct small-worlds inside grids based on semantics. In OntoSum, nodes are loosely structured, and each of them keeps track of a set of neighbors and organizes them into a multi-resolution neighborhood according to their semantic distance. A query is matched against the relevant nodes in the neighborhood. Intuitively, the query forwarding "zooms in" towards those regions with minimal ontological distance from the query, handing off the query to a node that has better information and is thus better prepared to answer the question. This architecture combines the efficiency and scalability of structured overlay networks with the connection flexibility of unstructured networks. It achieves full distribution, high scalability, robustness, and has been published as Li *et al.* [69, 62].

To further improve the search efficiency inside an ontologically homogeneous cluster, we propose a semantics-based query forwarding strategy – the Resource Distance Vector (RDV) routing algorithm. The basic idea is to extract the building blocks from the metadata instance and then summarize them to form a compact structure. Based on this summarization, we create a routing table to guide the query forwarding. Compared with unstructured P2P applications, which are oblivious of the resource location, this routing strategy reduces both the query overhead and query latency, and guarantees a higher query hit ratio. Compared with DHTs, our approach inherently supports rich queries, and requires no explicit control over the network topology or placement of data. The RDV algorithm has been published as Li *et al.* [65, 66, 67].

In summary, the primary contributions of this thesis are as follows:

- We propose a semantic model that adds semantics to the discovery system, thus improving the expressiveness and interoperability of the system.
- We propose effective topology adaptation schemes that reorganize the network according to semantic properties of participating nodes, in order to improve system scalability.
- We design novel indexing and searching algorithms for efficient discovery in networks with different underlying structures.
- We propose a comprehensive load-balancing scheme that effectively balances the system load and significantly improves performance.
- We conduct comprehensive simulation experiments to test all aspects of the designed system.
- We implement and deploy a prototype system that demonstrates the feasibility of the proposed system.

1.4 Thesis organization

The remainder of this thesis is organized as follows: Chapter 2 provides a general survey of related work in resource discovery, and background knowledge of the enabling technologies required for our work.

Chapter 3 describes our approach to the construction of semantic virtual organizations (VOs). We start by introducing the concepts of semantic virtual organizations and ontological directories. We then describe a DHT implementation of the ontological directory. Then, we present our distributed load-balancing mechanism, which addresses one of the major problems of the directory overlay.

In Chapter 4, we present GONID – a framework for semantic resource integration and discovery in virtual organizations. We describe our techniques for representing the resource metadata, mapping metadata with different representations, and evaluating complex queries. We then present a coarse-grained indexing scheme to efficiently store and retrieve resource metadata in a fully distributed DHT overlay. Finally, we describe a prototype implementation of this system.

An alternative discovery framework, OntoSum, is detailed in Chapter 5. We first explain our motivation for proposing such an alternative to the GONID system, then give an overview of the basic behind it – creating semantic small-worlds. We develop algorithms to compute the semantic distance between nodes, so that nodes can be reorganized to form semantic small-worlds according to their semantic distance from one another. Then, we present our searching algorithm applied to these small-worlds, followed by a distance-vector-based semantic query routing algorithm, RDV.

Finally, Chapter 6 presents our conclusions, a discussion of limitations, and suggestions for future work.

Chapter 2
Related Work
and Background Knowledge

In this chapter, we discuss related work on cross-domain resource discovery, as well as the enabling technologies required for our proposed framework. We also cite and discuss the research most relevant to our work in subsequent chapters.

2.1 Existing discovery technologies

To design a resource discovery system for global-scale grids, we examine existing resource discovery approaches, to see if any of these can provide a solution, or generate valuable insights into our discovery problem.

2.1.1 Grid information service

Infrastructure targeting grid resource information is often referred to as a Grid Information Service (GIS) [24], and provides information about resources and services to users.

Traditionally, a Grid Information Service is mainly based on a centralized or hierarchical model. In the Globus Toolkit 2 [31], the Monitoring and Discovery Service (MDS) [24, 29] provides access to static and dynamic information about resources. MDS is based on the Lightweight Directory Access Protocol (LDAP) [91, 58, 120], and consists of two components: Grid Index Information Services (GIIS) and Grid Resource Information Service (GRIS). The resource information is obtained by the information provider and is passed on to GRIS. GRIS registers its local information with the GIIS, which registers with another GIIS, and so on. MDS clients can get the resource information directly from GRIS (for local resources) and/or a GIIS (for grid-wide resources). The MDS hierarchy mechanism is similar to DNS. GRIS and GIIS, at lower layers of the hierarchy, register with the GIIS at upper layers, realizing the global indexing and discovery.

Globus Toolkit versions 3 and 4 [128] provides a service-oriented information service, i.e., the Index Service. The Index Service leverages service data defined in the Open Grid Services Architecture (OGSA) [30] specification to provide services. All services are described in a standardized XML schema, called Elements of Service Data (SDEs). The Index Service provides high-level API functionalities to register, aggregate, and query SDEs. Users can get a node's resource information by either directly querying a server application running on that node, or querying dedicated information servers that

retrieve and publish the resource information of the organization. Techniques for associating information servers, and to construct an efficient, scalable network of directory servers, are left unspecified.

Condor's Matchmaker [87] uses a centralized mechanism to locate desirable resources. Each node in the Condor system advertises its resources and reports resource status to a central manager. The central manager then matches resource requesters' queries with resource providers' advertisements.

Legion [20] takes an object-oriented approach to resource management. It uses *Collections* to search and locate resources in the grid. When a user requests a resource, Legion will query resource information in multiple Collections; if it finds several such resources, Legion's resource scheduler will randomly choose one of them.

For small-to-medium scale grids, these centralized or static hierarchical solutions work fine. However, for large, up to global-scale grids, these approaches are not efficient and do not scale. Additionally, even for smaller grids, a centralized solution will always be a performance bottleneck and a single point of failure. Presently, grids have moved from the obscurely academic to the highly popular. As the size of the grid grows from tens to thousands or even millions of nodes, the traditional server-based grid information service will not scale well. As a remedy, some researchers (e.g., [15] and [46]) advocate the use of P2P techniques for implementing scalable grid systems. We now review the state of the art in P2P discovery technologies.

2.1.2 P2P discovery systems

P2P has recently been recognized as a more natural and flexible approach to sharing resources than the traditional client-server model. P2P introduces a paradigm of decentralization and self-organization of highly autonomous peers. This new paradigm allows the system to scale to a very high number of participating nodes, while ensuring fault-tolerance.

2.1.2.1 Structured and unstructured routing

Existing P2P routing algorithms have been classified as "structured" or "unstructured". Each category has a few variations where certain features are enhanced to satisfy different application requirements.

Early versions of Gnutella [129] were unstructured, with queries being flooded throughout the network. This method, though simple, does not scale well in terms of message overhead. In addition, it is effective in locating popular content, but performs poorly for more rare content. There have been numerous attempts to enhance its scalability. For example, Chawathe *et al.* [21] improved the efficiency of searches in unstructured P2P networks by topology adaptation, replication, and flow control. Yang and Garcia-Molina [118] presented several strategies, such as iterative deepening and the DBF algorithm, to reduce the overhead of searching. Another algorithm, called probabilistic flooding [4], was modeled using percolation theory. Random-Walks [73, 1] proposed an alternative to flooding for unstructured searches.

They reduce the amount of network traffic at the cost of search speed. Random-Walks usually employ previous experience to help forward the query.

Structured systems, also called Distributed Hash Tables (DHTs), include Pastry [121], Tapestry [90], Chord [102], and the Content Addressable Network (CAN) [88]. They map a content identifier to a key, and guarantee that content can be located within a bounded number of hops. These systems have been shown to be scalable and efficient, however, there have been two main criticisms of structured systems. First, they do not support keyword searches and complex queries as well as unstructured systems. The second main criticism of structured systems is their tight control of both data placement and network topology, which also makes them more sensitive to failures.

Recently, hierarchical super-peer systems [119] have been proposed to improve search efficiency. They utilize the heterogeneity existing in P2P networks and adopt hierarchy in the form of Ultrapeers (Gnutella [129]) or Super-nodes (FastTrack [33). These powerful nodes maintain the indices for other nodes, allowing searching to be carried out only among these more powerful nodes. The introduction of a new level of hierarchy in the system increases the scale and speed of a query lookup.

2.1.2.2 Index and lookup

P2P discovery systems can be classified into the following four categories, according to index and lookup differences.

Exact key match. This is the simplest index and lookup method. The early file-sharing systems belong to this category. These systems index file names; to search a file, the user has to provide the exact file names as the search key. Because of its limited searchability, few systems continue to use this method.

Keyword lookup. The most widely referenced P2P systems use this lookup method. For example, Gnutella queries contain a string of keywords. Gnutella peers answer when they have files whose names contain all the keywords. A few studies extend DHTs to support keyword lookup as well. For example, in their paper [93], Reynolds *et al.* proposed mapping each keyword to a key and publishing links to objects based on these keys. A query with multiple keywords has to look up each keyword and return the intersection. All related peers have to exchange large amount of data to locate the intersection. Systems such as eSearch [105] avoid this multiple lookup and intersection by storing the complete keyword list of objects on each node, but this may incur more overhead on publishing and storing the keywords.

Peer information retrieval (IR). The taxonomy of peer IR adopts information retrieval techniques to P2P networks to search large text collections. Most peer IR systems adopt the Vector Space Model (VSM) [94]. The VSM model is a way of representing documents and queries through the words (terms) that they contain. Each term is assigned a weight reflecting its importance in the document. One popular weighting scheme is TFIDF weighting [95], which includes the collection of weighting calculations that incorporate term frequency (TF) and inverse document frequency (IDF). The similarity of the document

and query vectors gives an indication of how well a document matches a particular query. P2P IR systems such as PlanetP [23] and eSearch [105] leverage the VSM by indexing the vector by distributed peers. Latent Semantic Indexing (LSI) [34] tries to overcome the lexical matching problems of VSM by using statistically derived conceptual indices instead of individual words for retrieval. LSI assumes that there is some underlying structure in word usage that is partially obscured by variability in word choice. A truncated singular value decomposition (SVD) [25] is used to estimate the structure in word usage across documents. Systems such as pSearch [106] incorporate LSI with the P2P overlay to benefit peer IR. Like other information retrieval systems, peer IR is mainly used for full-text (e.g., HTML, PDF) search, but is not applicable to searching other resources.

Peer data management. Another growing class of P2P systems, called Peer Data Management Systems (PDMS) [107], have been motivated by P2P technologies and distributed database systems. In a PDMS, every peer maintains a local database associated with a schema. The relationships between the data of peers are maintained between pairs of peers. By traversing paths of schema mappings, a query (usually in SQL format) over one peer can obtain relevant data from any reachable peer in the network. PeerDB [84] is reportedly the first PDMS implementation. PeerDB can be viewed as a network of local databases on peers. It allows data sharing without a global schema by using meta-data for each relation and attributes of relation entities. To process queries, relations that match the user's search are returned by searching on neighbors. After the user selects the desired relations, queries are directed to nodes containing the selected relations. Recently, PDMSs using semi-structured data, such as eXtensible Markup Language (XML) and Resource Description Framework (RDF) [57, 124], have appeared. For example, Piazza [42, 108] answers XQuery-based queries [10] through chained directional XML mappings. Edutella [82, 83] provides RDF-based metadata management on a JXTA framework [132].

Most PDMSs concentrate on data management such as schema mapping and query processing, and do not emphasize the underlying P2P network structure and routing algorithms. For example, PeerDB and Piazza do not provide network topology or query routing schemes; they simply assume that queries can be answered by forwarding to neighbors. Edutella uses JXTA to broadcast queries to a HyperCup topology. The simple P2P structure and routing algorithms used by PDMSs make these systems difficult to scale to very large networks. Although research such as RDFPeer [16] provides a DHT-based routing scheme to route RDF data, other data management problems such as information integration are not addressed. Efficient architectures and routing schemes that can improve the communication efficiency of PDMS have got to be developed.

P2P systems offer many benefits over the traditional client-server model, including better scalability, automatic management, fault-tolerance, and load-balancing. Therefore, we use P2P as our underlying communication structure. At the same time, P2P systems also present several challenges that preclude their widespread acceptance and usage in grids. For example, current P2P systems often lack the ability to deploy production-quality services, persistent and multipurpose service infrastructure, complex services, robustness, performance, and security. Thus, one of the tasks of this thesis is to overcome these problems and make P2P systems better serve grid needs.

2.1.3 Other discovery protocols

Domain Name System (DNS) [125] is the most successful wide-area service for location of resources based on names. DNS allows nodes on the public Internet to be assigned both an IP address and a corresponding domain name. For DNS to work as designed, these names must be globally unique. DNS is a hierarchical system; it organizes all registered names in a tree structure. DNS was not designed to work with dynamic addressing such as that supported by DHCP [126]. DNS requires that fixed (static) addresses be maintained in the database. Its hierarchical organization and caching strategies take advantage of the rather static nature of the information to be managed. DNS name resolution and grid resource discovery have similar features: both are used to locate resources by giving a description of the resources. DNS use names as its search criteria, while resource discovery allows for attribute matching and service browsing. In addition, DNS was designed to locate rather static information, while grid services are more dynamic.

Universal Description Discovery and Integration (UDDI) [139] is an open industry initiative, enabling businesses to publish service listings and discover each other, and define how services or software applications interact over the Internet. It consists of a standards-based set of specifications that allow service description and discovery. Businesses populate the registry with descriptions of the services they support. UDDI assigns a unique universal identifier (UUID) to each registration and stores them in the registry; this can be a dynamic process allowing search and discovery to automatically adapt to available services. The current search facilities offered by the latest version of UDDI do not offer any special features for finding Web service registries. As a result, it is assumed that Web service clients have prior knowledge of the location of the registries.

Search engines are an important tool for information foraging on the Web. Search engines are suites of computer programs that automatically find and download web pages, storing them in a database. They include programs that link the database to a user interface, normally in the form of a web page, so that it can be interrogated through the Internet, often by a keyword search. The query capability of Web search engines is limited to keyword search. Given a set of keywords, a Web search engine returns a ranked list of pages that contain the keywords. Since Web data is essentially text, it does not have the rich semantics of structured data. As a result, more expressive, database-style queries cannot be supported. A search engine has to crawl from the Web periodically in order to maintain an up-to-date index. Most Internet search-type lookup services fail to be responsive to updates, for example, Google measures its response time to dynamic changes in days.

Traditional search engines do not consider domain knowledge; they do not understand the meaning of a user's search request and the inherent relationships among the terms that a Web document contains. This limitation severely curtails their ability to perform a content-based search. Due to the complexity of natural language, which contains synonyms and homonyms/polysemes, the precision and recall of the search results can hardly be guaranteed. Although present search engines are trying to make up for this limitation by means of query expansion or Latent Semantic Indexing (LSI) [34], the core of the problem

is not being addressed. Employing domain knowledge to assist in the search process is the key to semantic search.

2.1.4 Summary of existing discovery technologies

In summary, there are many discovery-related technologies, addressing different aspects of resource location in networks. Each model has its own advantages and disadvantages. Classical grid information service and Web services use centralized or hierarchical solutions, which work fine for small- or medium- sized networks, but are not scalable for large-scale networks. DNS requires a global naming strategy and was designed to locate rather static information. Search engines are powerful but are unable to answer complex queries, respond slowly to new updates, and are not applicable to searching intra-organization resources. P2P schemes always provide single functionality and rarely achieve both scalability and rich searchablility. In view of these defects of existing systems, a new architecture is clearly needed. In this thesis, we address some of the major issues of resource discovery in a large-scale grid and attempt to provide a solution to these problems.

2.2 Concepts and enabling technologies

In this section, we discuss two key enabling technologies required for our proposed discovery framework, namely the Semantic Web and P2P overlay networks.

2.2.1 Semantic Web and ontologies

In our work, we employ techniques from the Semantic Web to make searching more intelligent. The Semantic Web is an evolving extension of the World Wide Web (WWW) in which Web content can be expressed not only in natural language, but also in a format that can be read and used by software agents, thus permitting them to find, share and integrate information more easily. Proposed by Tim Berners-Lee, inventor of the Web and HTML, the Semantic Web is his vision of the future of the WWW. The development of the Semantic Web proceeds in steps, each step building a layer on top of another. Figure 2.1, which is adapted from a book by Antoniou *et al.* [2], shows the main layers of the Semantic Web design and vision. The bottom layer is XML, a language used for writing structured Web documents with a user-defined vocabulary. XML is particularly suitable for sending documents across the Web. RDF is a basic data model, like the entity-relationship model, for writing simple statements about Web objects (resources). The RDF data model does not rely on XML, but RDF has an XML-based syntax. Therefore, in Figure 2.1, it is located on top of the XML layer. The RDF schema is based on RDF and provides modeling primitives for organizing Web objects into hierarchies. Its key primitives are classes and properties, subclass and sub-property relationships, and domain and range restrictions. The RDF schema can be viewed as a primitive language for writing ontologies. The next layer represents more powerful ontology languages such as OWL [134] that expand the RDF schema and allow representations of more complex relationships between Web objects. The logic layer is used to enhance the ontology language

further and to allow the writing of application-specific declarative knowledge. The proof layer involves the deductive process as well as the representation of proofs in Web language and proof validation. Finally, the Trust layer will emerge through the use of digital signatures and other kinds of knowledge, based on recommendations by trusted agents.

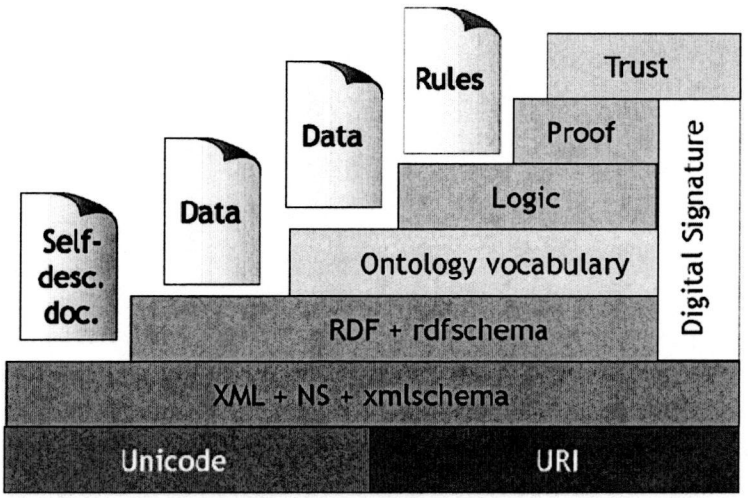


Figure 2.1 A layered approach to the Semantic Web (Adapted from Antoniou *et al.* [2])

Ontology

The Semantic Web relies on ontologies that structure underlying data for the purpose of comprehensive and transportable machine understanding. Ontology is defined as "an explicit specification of a conceptualization" [39]. Ontologies are (meta)data schemas, providing a controlled vocabulary of concepts, each with an explicitly defined and machine-processable semantics. By defining shared and common domain theories, ontologies help both people and machines to communicate concisely, supporting the exchange of semantics and not only syntax. In general, an ontology of a domain consists of the four major components listed below [112]:

- Concepts: Concepts of a domain are abstractions of concrete entities derived from specific instances or occurrences.
- Attributes: Attributes are characteristics of the concepts that may or may not be concepts by themselves.
- Taxonomy: Taxonomy provides the hierarchical relationships between the concepts.
- Non-taxonomic relationships: Non-taxonomic relationships specify non-hierarchical semantic relationships between the concepts.

Along with the above four components, ontologies may also consist of instances of each of the concepts, and inference rules of the domain. Figure 2.2 illustrates a portion of the ontology for a Computer Science Department. It indicates some of the major concepts and their hierarchical relationships only. In general, the ontology includes non-taxonomic relationships as well, for example: *Student – take – Course*, *Professor – teach –Course*, *Worker – work – Department*

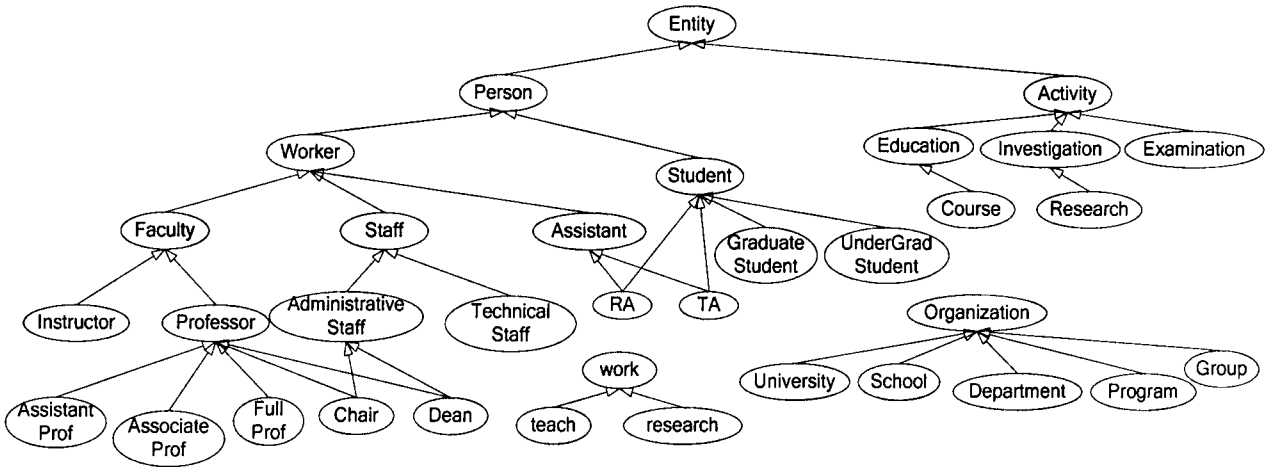


Figure 2.2 An example of an ontology snippet

Various meta-languages such as XML, RDF, and OWL have been developed for encoding the ontologies of a domain. Several algorithms or techniques for merging or querying ontologies are undergoing development. Tools such as Protégé [85] and OntoEdit [104] have been developed for the construction and management of ontologies. Moreover, a wide variety of communities have developed either general purpose or domain-specific ontologies for various domains. For example, one part of our framework, the OntoSum sub-system, uses a general purpose ontology - WordNet [76] to facilitate computation of the semantic distance between different ontologies.

Resource Description Framework (RDF)

The Resource Description Framework (RDF) [57, 124] is a World Wide Web Consortium (W3C) recommendation for describing Web resources. RDF is a basic data model, like the entity-relationship model, for writing simple statements about Web objects (resources). The fundamental concepts of RDF are resources, properties, and statements. We can think of a resource as an object, a "thing" we want to talk about. Resources may be authors, books, places, rooms, search queries, and so on. Every resource has a Uniform Resource Identifier (URI). A URI can be a Uniform Resource Locator (URL) or some other kind of unique identifier. Note that an identifier does not necessarily enable access to a resource. In general, we assume that a URI is the identifier of a Web resource. Properties are a special kind of resource; they describe relationships between resources, for example, "written by", "age", "title" and so on. Properties in RDF are also identified by URIs. Statements assert the properties of resources. A statement is an object-attribute-value triple, consisting of a resource, a property, and a value. Values can be either resources or literals.

RDF can represent simple statements about resources as a graph of nodes and arcs representing the resources, and their properties and values. For example, the group of statements, "there is a person identified by http://someURI/contact#me, whose name is Juan Li, whose email address is juali@cs.ubc.ca, and who is the creator of the web page http://cs.ubc.ca/~juanli" could be represented as the RDF graph in Figure 2.3.

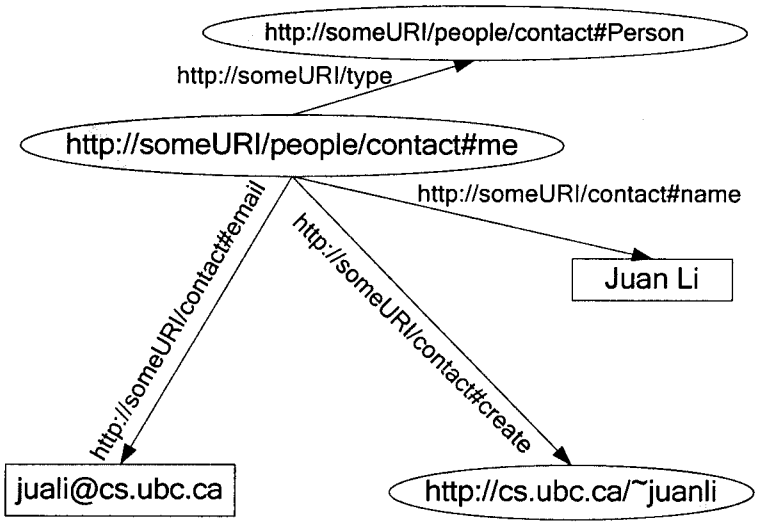


Figure 2.3 An example RDF graph

Figure 2.3 illustrates that RDF uses URIs to identify:

- Individuals, e.g., Juan Li, identified by http://someURI/People/contact#me
- Kinds of things, e.g., Person, identified by http://someURI/people/contact#Person
- Properties of those things, e.g., email, identified by http://someURI/contact#email
- Values of those properties, e.g., character string "Juan Li" as the value of the name property (RDF also uses values from other data types such as integers and dates, as the values of properties)

RDF also provides an XML-based syntax (called RDF/XML) for recording and exchanging these graphs. Figure 2.4 shows a small chunk of RDF in RDF/XML corresponding to the graph in Figure 2.3.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:contact="http://someURI/contact#">

 <contact:Person rdf:about="http://someURI/people/contact#me">
  <contact:name>Juan Li</contact:name>
  <contact:create rdf:resource="http//cs.ubc.ca/~juanli">
  <contact:email>juanli@cs.ubc.ca</contact:email>
 </contact:Person>
</rdf:RDF>
```

Figure 2.4 An example RDF/XML description

Web Ontology Language (OWL)

RDF only provides users with metadata for describing resources, using named properties and values. It does not make any commitment to a conceptual structure or a set of relations to be used. An RDF Schema (RDFS) [135] defines a simple structure by introducing relationships like inheritance and

instantiation, standard resources for classes, as well as a small set of restrictions on objects in a relationship. An RDFS can be viewed as a primitive language for writing ontologies. It is roughly limited to a subclass hierarchy and a property hierarchy, with domain and range definitions of these properties. Since our system needs to represent more complex relationships between resources and perform useful reasoning tasks, we use the OWL language to represent shared models of the resources.

Web Ontology Language (OWL) [134], a W3C recommendation, is an RDF-based language that introduces special language primitives for defining classes and relationships as well as necessary conditions (e.g., every human has exactly one mother) and sufficient conditions (e.g., every woman who has a child is a mother) for class membership, as well as general constraints on the interpretation of a domain. RDF data can be linked to OWL models by the use of classes and relationships in the metadata descriptions. The additional definitions in the corresponding OWL model impose further restrictions on the validity and interpretation of the metadata. For example, in OWL, we can say that *AssociateProfessor* and *AssistantProfessor* are disjoint classes: no individual can be both an associate professor and an assistant professor. *Faculty* as a class can be defined as the equivalence class of *Academic Staff*. It is possible to talk about Boolean combinations (union, intersection, complement) of classes in OWL. Similarly, OWL can express more complex properties. It provides more property restrictions by <owl:Restriction>. In RDFS, properties can be related via a property hierarchy. OWL extends this feature by allowing properties to be denoted as transitive, symmetric or functional, and allows one property to be declared to be the inverse of another. OWL also makes a distinction between properties that have data-values as their range, and those whose range is other individuals.

There are three different sub-languages of OWL: OWL-Full, OWL-DL and OWL-Lite, each geared toward fulfilling different application requirements. We have adopted OWL-DL as our ontology language, because it supports automated reasoning, and, in this regard, has a formal semantics based on Description Logic (DL) [5]. DLs are typically a decidable subset of First Order Logic, and are tailored towards Knowledge Representation (KR) [81]. They are suitable for representing structured information about concepts, concept hierarchies and relationships between concepts. The decidability of the logic ensures that sound and complete DL reasoners can be built to check the consistency of an OWL ontology, i.e., verify whether there are any logical contradictions in the ontology axioms. Furthermore, reasoners can be used to derive inferences from the asserted information, e.g., infer whether a particular concept in an ontology is a sub-concept of another, or whether a particular individual in an ontology belongs to a specific class. Popular existing DL reasoners in the OWL community include Pellet [99], FaCT [44] and RACER [41].

All varieties of OWL use RDF for their syntax. Instances are declared as in RDF, using RDF descriptions and typing information. OWL constructors such as owl:Class, owl:DatatypeProperty, and owl:ObjectProperty are specialized instance of their RDFS counterparts.

2.2.2 P2P overlay networks

Because of its attractive properties, we employ P2P overlay network as the networking foundation of our searching system. In the context of this thesis, P2P is a communications model, in which each peer can play dynamic roles: resource requester, resource provider, or query-forwarding router, based on the required functionalities among peers. In our work, both of the P2P overlay categories, namely unstructured and structured overlays, are used to construct virtual networks on top of the existing network. Overlay network technologies facilitate network functionalities and enable P2P applications and services to perform their intended operations, independent of their physical locations and logical network domains. The basic idea of a P2P overlay was presented in Section 2.1.2. We discuss the most relevant P2P overlay techniques in the context of our work in subsequent chapters.

2.2.3 Summary of enabling technologies

In summary, Semantic Web and P2P are the two main enabling technologies used in our grid resource discovery framework. The Semantic Web addresses the requirement to model, manipulate and query information at the conceptual level. It applies ontologies to information to improve the quality of information management. Ontologies are used to explicitly represent semantics of resource metadata, and enable sophisticated automatic support for acquiring, maintaining and accessing information. P2P technologies aim at abandoning centralized control in favor of decentralized organizational principles, allowing new degrees of freedom for changing information architectures and exchanging information between different nodes in a network. Together, the Semantic Web and P2P allow for combined flexibility at the level of information structuring and distribution. In this thesis, we investigate how to benefit from this combined flexibility by discussing an ontologically driven search and exploration of inter-ontological relationships over a P2P infrastructure. Methodology and tools are developed for intelligent access to large quantities of resource information in intra- and extra-virtual organizations, as well as Internet-based environments, to employ the full power of ontologies in supporting of resource searching and sharing, from both the resource consumer perspective and that of the resource provider.

Chapter 3

Virtual Organization Formation with an Ontological Model

If not properly organized, searching an Internet-scale grid for quality resources is like looking for a needle in a haystack – we have too large a space to explore. Therefore, as the first step of our discovery scheme, we organize and reduce the huge chaotic search space to multiple semantics-based sub-spaces. Participants in each sub-space share similar semantic interests, forming semantics-based Virtual Organizations (VO). Searching can then be performed on VOs, and queries can be quickly propagated to many appropriate members in the VO. This procedure results in a higher precision and recall of search results.

We propose an ontology directory model, the Distributed Ontology Directory (DOD), to help nodes locate their VOs of interest. The ontology directory defines a hierarchy of categories of a domain of interest that are used to represent the semantics of resources. It is like a yellow page or a "rendezvous" point, allowing nodes to find contacts with similar interests. The directory does not need to be predefined; it grows spontaneously as network interest evolves. To implement the directory, the system relies on an efficient DHT process as a building block. A hierarchy path identifies relations of categories in the path. The system transforms the ontology directory path into a set of numeric keys. It does so in a way that preserves the expressiveness of the semi-structured data, facilitates an even data distribution through the network, enables efficient query resolution, and provides a flexible lookup interface.

One major problem of this directory overlay is the load unbalance caused by the inherently uneven hierarchical namespace and highly variable directory popularity. To solve this problem, we propose an effective load-balancing solution, which takes the node heterogeneity and access popularity into account to determine the load distribution. Our algorithm achieves load balancing by dynamically balancing the query routing load and query answering load respectively.

The main contributions of this chapter are as follows:

1. We propose an effective model – the ontology directory – to help cluster grid nodes into VOs in order to reduce the search space and thereby improve the search efficiency.
2. We design an effective strategy to index the hierarchical ontology directory with a flat DHT structure.
3. We solve a major problem of the DHT ontology indexing: unbalanced loads caused by skewed directory popularity.

Portions of this chapter originally appeared in Li *et al.* [64, 68, 70].

In the rest of this chapter, we introduce the notion of ontological directories in Section 3.1. Then in Section 3.2 we describe how to index and search in the directory overlay. In Section 3.3, we explain a comprehensive scheme for solving the load unbalancing problem of the directory overlay. Finally, we validate our models using simulation in Section 3.4.

3.1 Concept of ontological directory

An open and dynamic grid should allow users to use the grid structure and resources for a wide variety of purposes. Grids will be fully exploited only when people can quickly and conveniently build virtual organizations (VOs), which are defined as a group of individuals or institutions who share the grid resources for a common goal [32]. In order to organize different interests and facilitate the construction of VOs, we propose an abstract generic ontological model that guides users in determining the desired ontological properties and choosing the "right" VOs to join. The ontology model defines most general categories of existence (e.g., existing item, spatial region, dependent part), which essentially form a hierarchy where each entry corresponds to a categorical domain. Here we provide a formal definition of this ontology model, which we call the ontology directory.

DEFINITION 3.1: An ontology directory is a system $D=(L,H,r)$, which consists of:
- *A lexicon: The lexicon L contains a set of natural language terms.*
- *A hierarchy H: Terms in L are taxonomically related by the directed, acyclic, transitive, reflexive relation H.* ($H \subset L \times L$);
- *A root term* $r \in L$. *For all* $l \in L$, *it holds:* $H(l,R)$.

The ontology directory essentially defines a hierarchy where each node corresponds to a lexicon or a categorical term. It is almost a rooted tree structure, with rare nodes having multiple parents. The subordination relationship between nodes is interpreted as the involvement (topic/subtopic) relationship, while the layers of nodes correspond to intuitively perceived levels of abstractness of topics. Each node is described by primitives which are generic concepts that include other concepts. An example of a primitive is *computer* that includes *software*, *hardware*, *networks*, and so forth. The hierarchical relationship, also called the IS-A relationship, is transitive, i.e., whatever holds for a more general concept also holds for a more specific concept, e.g., *music* is a type of *art*. Figure 3.1 shows a fragment of an example ontology directory.

It is beneficial to have a common ontology module, because having some common ground for the high-level general concepts could alleviate the problem of semantic heterogeneity and provide an aggregated view of the network. The ontology model allows users to choose the right VO to join, detect new trends, or find useful information they did not realize was available. In fact, in the World Wide Web, the global directory of web sites, e.g., Google directory, Yahoo directory, and DMOZ [123], has been widely adopted. The directory helps the system to create a well-knit ontology structure and makes ontology location and browsing efficient. Our ontology directory is different from those global web directories, because it is not predefined, but created and extended automatically with network growth and the

evolution of the ontology. Moreover, the ontology directory loosely defines domain categories; it does not expect different communities of users to conform to the same ontology to describe their resources and interests. Therefore, it is based on multiple ontologies as opposed to a global ontology.

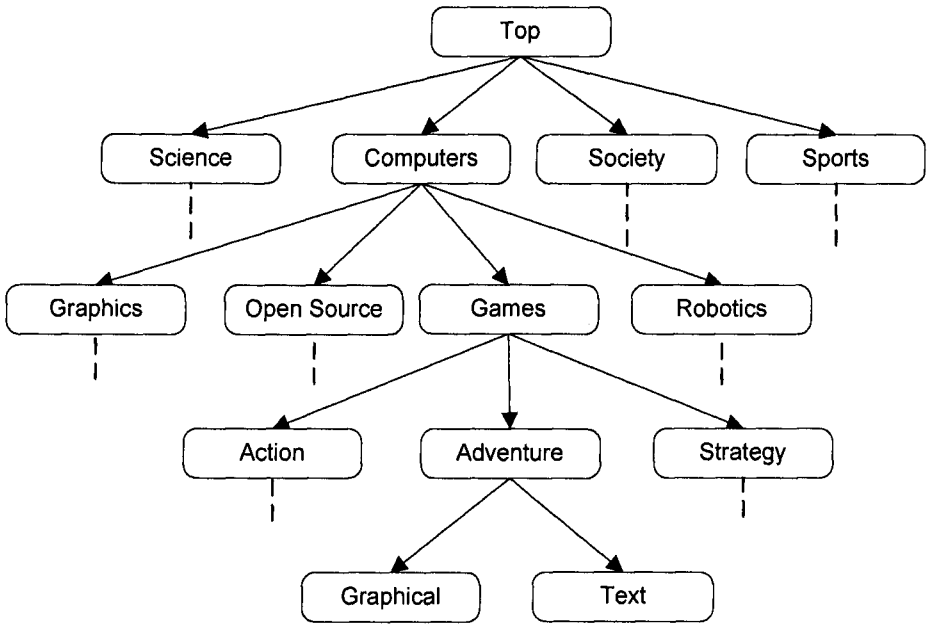


Figure 3.1 Fragment of an example ontology directory

To implement the ontology directory in a decentralized manner, we need an efficient and scalable structure to index and lookup the hierarchical taxonomy. Many applications, such as Canon [35] or HIERAS [117], use hierarchical DHTs to index hierarchical data; however, they are not applicable to our system. Specifically, in Canon, information is only stored in real nodes (leaf nodes), while all internal nodes are purely virtual; however, directory information in our system is stored in real (non-virtual) internal nodes. In addition, it is difficult to implement directory browsing inside the Canon network. HIERAS constructs the multi-level hierarchy by creating multiple DHT overlays for each subdirectory. As mentioned by Artigas *et al.* [3], "DHTs are mainly designed for very big networks, and the creation of several DHTs assigned to sub-domains or layers can represent a burden for the problem that they aim to solve". The major problem of the multi-level DHT architecture is that maintaining each level of the hierarchy brings extra overhead; it is a waste on subdirectories not large enough. In our work, we propose a simple flat DHT structure to index the hierarchical directory. This structure enables economical, flexible, and efficient lookup services.

3.2 DHT-based directory indexing

In this section, we describe techniques for indexing a hierarchical ontology directory in a flat DHT overlay. Our system provides multiple interfaces permitting users to access the directory in different ways.

3.2.1 DHT indexing background

Distributed hash tables (DHTs) are a class of decentralized distributed systems that partition ownership of a set of keys among participating nodes, and can efficiently route messages to the unique owner of any given key. A partitioning scheme splits ownership of the key-space among the participating nodes. Each node maintains a set of links to others; together these form the overlay network. For any key k, a node either owns k or has a link to a node that is closer to k in terms of the key-space distance. Routing of a message to the owner of any key k can use a greedy algorithm: at each step, forward the message to the neighbor whose ID is closest to k. When there is no such neighbor, then it must have arrived at the closest node, which is the owner of k. A typical use of the DHT for storage and retrieval might proceed as follows. Suppose the key-space is the set of 160-bit strings. To store a file with a given filename and data in the DHT, the SHA1 hash of filename is computed, producing a 160-bit key k, and a message *put (k, data)* is sent to any node participating in the DHT. The message is forwarded from node to node through the overlay network until it reaches the single node responsible for key k. Any other client can then retrieve the contents of the file by again hashing the filename to produce k and asking any DHT node to find the data associated with k with a message *get(k)*. While any of the structured DHTs can be used for the purpose of our scheme, we use Pastry [90] and Chord [102] as examples in this thesis. In the following, we briefly explain the DHT mapping in Pastry and Chord.

Pastry is a structured P2P routing protocol that implements the DHT abstraction. In a Pastry network, each node has a unique, uniform, randomly assigned nodeId in a circular 128-bit identifier space. Given a message and an associated 128-bit key, Pastry reliably routes the message to the live node whose nodeId is numerically closest to the key. In a Pastry network consisting of N nodes, a message can be routed to any node in less than $log\ 2^b N$ steps on average, where b is a configuration parameter. Each node stores only *O(logN)* entries, where each entry maps a nodeId to the associated node's IP address. Specifically, a Pastry node's Routing Table is organized into $\lceil \log 2^b N \rceil$ rows with (2^b-1) entries each. Each of the (2^b-1) entries at the row n of the Routing Table refers to a node whose nodeId shares the first n digits with the present node's nodeId, but whose $(n+1)th$ digit has one of the (2^b-1) possible values other than the $(n+1)th$ digit in the present node's nodeId. Pastry stores multiple candidates per Routing Table entry to increase availability. In addition to a Routing Table, each node maintains a Leaf Set, consisting of $L/2$ nodes with the numerically closest larger nodeIds and $L/2$ nodes with the numerically closest smaller nodeIds, relative to the present node's nodeId. L is another configuration parameter. In each routing step, the current node forwards a message to a node whose nodeId shares with the message key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the current nodeId. If no such node is found in the Routing Table, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node but is numerically closer to the key than the current nodeId. Such a node must exist in the Leaf Set unless the nodeId of the current node or its immediate neighbor is numerically closest to the key. A detailed description of locality-aware routing in Pastry can be found in [90].

Chord system is another popular DHT algorithm. Nodes in Chord are placed on a Ring. Both node IDs and keys are placed on the same ring. The hash function produces an m-bit identifier for both nodes and keys for this purpose. Each node has a successor and predecessor . Insertion of a new node between two older nodes involves the update of successor of one of those node and predecessor of the other. A key-value pair is assigned to the first node whose identifier is equal or follows the identifier of the key. For the efficiency of searching, Chord use finger table to partition the virtual cycle into $1+logN$ segments. This finger table let each machine use $O(log(N))$ memory to maintain the topology information. When node searches a resource, it first calculates the hash value of the resource name and then determines the position of the resource in the virtual cycle of node.

3.2.2 Ontology directory indexing

To efficiently index and retrieve the hierarchical ontology directory with a flat DHT structure, we need to extend the basic DHT API. The directory path starting from the root is used to represent the ontology domain (e.g., */computer science/systems/network*). One domain corresponding to a particular VO should include contact information for peers in this VO. A direct indexing scheme is to index the full directory path as a key, and users can locate a VO by providing the full directory path. However, like navigating in a UNIX file system, users rarely input an absolute directory path, but rather browse directories level by level and select the more interesting one at each level. Therefore, it is necessary to provide users an ontology browsing interface. Moreover, to automatically locate related VOs for nodes, we extract key concepts from the joining nodes' ontology and then use them as keys to locate the right directory domain. Therefore, we should also provide a keyword-based lookup interface.

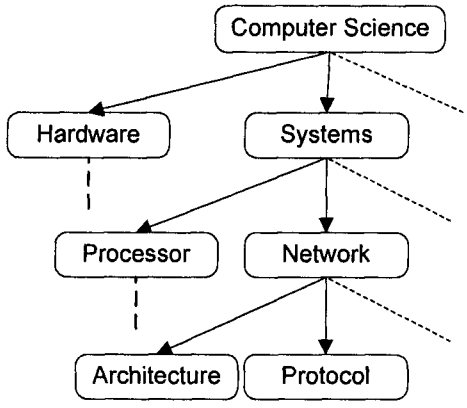


Figure 3.2 Fragment of an ontological directory model

Consider the ontology model in Figure 3.2. It consists of taxonomy paths:

/computer science
/computer science/systems
/computer science/hardware
/computer science/systems/network

/computer science/systems/processor
/computer science/systems/network/architecture
/computer science/systems/network/protocol

Some domains may relate to keywords, for example:
Keywords: *cluster, grid, P2P,* are related to taxonomy */computer science/system/network/architecture*
Keywords: *protocol, TCP, IP* are related to taxonomy */computer science/system/network/protocol*

For each path and keyword, a hash value (key) is computed in Pastry using an SHA-1 algorithm. Table 3.1 shows keys for taxonomy paths and keywords of the model. To make the example simple, we use a 4-digits (8 bits) identifier space; in reality a much larger identifier space is used, such as 160 or 128 bits. Each key is assigned to a node, which is the nearest node to the key in the key-space. For example, as listed in Table 3.1, the hashed key of directory path */computer science/system* is *0230,* and the key is stored at node *0213* as shown in Figure 3.3, since node *0213*'s id is closest to the key. Each owner node of a directory key maintains a Least Recently Used (LRU) cache storing contact information of peers that are interested in this directory. To implement the directory browser's functionality, an overlay node that is in charge of a directory entry also stores information about that directory's direct children. When the user chooses one directory, Pastry routes to that directory entry and retrieves child directory information, allowing the directory to be extended dynamically while browsing. An overlay node also stores keywords that are hashed to it and links the keywords with related ontology domains. Figure 3.3 shows how the directory model above is stored into an example Pastry network.

Table 3.1 Hash keys of models in Figure 3.2 in a sample 4-digit identifier space

Hash key	**Directory path**
1211	/computer science
0230	/computer science/systems
3211	/computer science/hardware
2011	/computer science/systems/network
1000	/computer science/systems/processor
1013	/computer science/systems/network/architecture
0012	/computer science/systems/network/protocol
2111	Protocol
0211	TCP
1201	IP
2003	Cluster
0012	Grid
0032	P2P

Since nodes might fail and network connections might break, the ontology model stored on its corresponding overlay nodes are replicated on its neighbors in the Pastry identifier space. This can be done by setting the replica factor *f*. Whenever a node receives a directory storing request, it will not only store the directory locally but also store it to its *f* immediate leaf nodes. If any node fails or its connection breaks, its leaf neighbors will detect it by using the keep-alive messages.

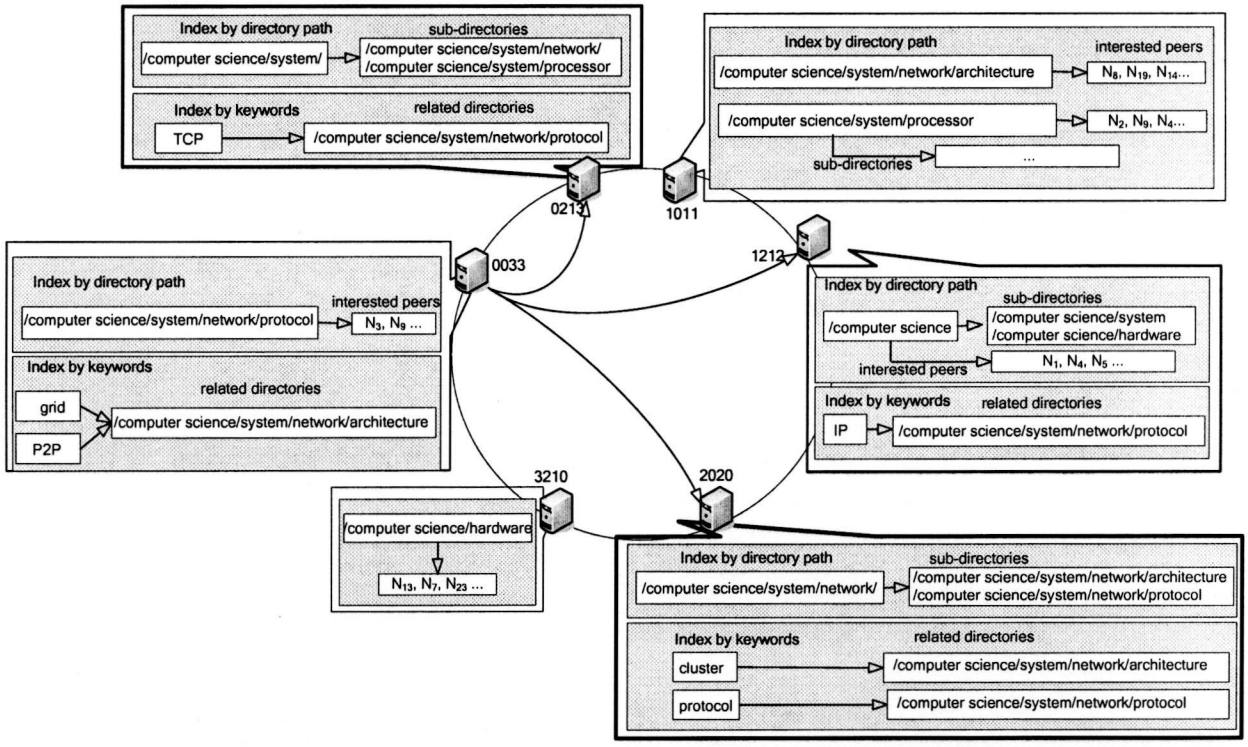


Figure 3.3 Storing the ontology model into a Pastry network

of 6 nodes in an example 8-bit identifier space

3.2.3 Ontology directory lookup and VO register

We provide three kinds of lookup interfaces for users: exact lookups, browser-based lookups, and keyword-based lookups. A node can use these three interfaces to locate VOs they are interested in and join these VOs.

Exact lookups: This is the simplest form of a lookup. This type of query contains the complete directory path of the interest domain, for example *"/computer science/system/network/architecture"*. This complete directory path is hashed to a key and then a corresponding lookup of the hashed key on the Pastry overlay is executed.

Browser-based lookups: In this case, users do not need to remember the directory path to locate the directory domain of interest. Instead, they can navigate from the root of each hierarchy down to the leaves to reach the directory of interest. A user first uses the root Id as the key to locate the node storing the root of the ontology model. Since a node storing a directory entry also storing the next level children, then users can dynamically expand a directory tree node to browse its child branches. After the user chooses an interested branch, the directory path of that branch is used as a key to lookup the next level directory. In this way, the tree is expanded until users find the desired directory entries. In reality, the root and top level categories are widely cached in most of the nodes in the network; therefore they can be quickly located without going through the overlay network.

Keyword-based lookups: Users can also specify one or more key concepts of their local ontology and use the concepts as keys to lookup the corresponding directory in the overlay. Since overlay nodes in charge of the keywords keep links to the corresponding directory entries, a keyword-based lookup can be converted into an exact lookup. When a user provides multiple keywords, each of them may correspond to multiple directories, the intersection (or union) of all directories related to these keywords is returned to the user. Domain ontologies and/or external generic ontologies like WordNet [79, 28] can be used for keyword semantic query expansion or keyword conceptual indexing in order to improve retrieval performance.

VO register: Since an overlay node in charge of an ontology directory also keeps a cache storing information about nodes interested in that ontology directory, a querying node can get contacts of others sharing the same interest through this overlay node. The new node can then join the VO by connecting to those contacts. At the same time, if its ontology matches the ontology of the VO, this new node can register with the VO by adding itself to the cache of the directory overlay node; therefore, in the future, others can find it. A node with multiple interests can register with multiple VOs. There are several special cases for a node's registration: (1) If a new registering node cannot get enough contacts from the interested domain (i.e., the VO is very small), it explicitly routes to the upper- and/or lower- level categories to register and get more contacts. (2) If a node cannot find suitable categories satisfying its interest (i.e., it is the first node registering this interest), it will try to add this category by applying from an authoritative organization.

Directory overlay maintenance: The directory overlay nodes are also user nodes. We utilize the heterogeneity of grid nodes, and promote those stable and powerful ones to join the directory overlay. Excluding ephemeral nodes from the directory overlay avoids unnecessary maintenance costs. The maintenance of the directory overlay mainly includes adding new directory entries. We assume deleting and updating do not occur frequently. When a new joining node cannot find its category of interest, it may try to apply to create a new category. If the application is approved by the authoritative organizations in the grid, the node will create this category by hashing the directory path to an overlay node and informing the parent node to add this entry. Then it hashes each of its main key concepts in the ontology to the overlay network. A node joins the directory overlay only when three conditions are satisfied: (1) It satisfies the capacity requirements, i.e., it is powerful enough. (2) It is stable for a

threshold time period. (3) The directory load balancing algorithm (which will be explained in next section) requires it to do so.

3.3 Directory overlay load balancing

The directory overlay uses a DHT to distribute directories randomly among peers, but the consistent hash which used by DHT may cause some peers to have O(log*N*) times as many objects as the average peer [102]. In addition, peer capacity, such as computational power, storage capacity and network bandwidth are quite different among peers. Even with a uniform workload distribution, serious load imbalance problems may occur. Further imbalance may happen due to the non-uniform distribution of directory entries in the identifier space. The situation is even worse for hierarchical systems such as our ontological directory hierarchy: servers hosting nodes at the top of the hierarchy will incur exponentially disproportionately more load than servers hosting leaf nodes. Last but not least, directory queries tend to be skewed, i.e., certain directories are quite popular as compared to the others. Heavy lookup traffic load is experienced at the peers responsible for popular objects, as well as at the intermediary nodes on the lookup paths to those peers. When subsequent tasks are then obliviously assigned to the already overloaded node, the average response time consequently increases drastically. We aim at balancing the highly unbalanced load caused by skewed directory distribution through the use of a comprehensive balancing mechanism, which includes an adaptive load redistribution scheme as well as a dynamic routing table reconfiguration scheme.

3.3.1 Existing balancing strategies

There have been many load balancing schemes proposed for DHT-based systems. Roughly, we divide them into four categories:

The **virtual server** approach [36, 50, 51] focuses on the imbalance of the key distribution due to the hash function. Each physical node instantiates O(log*N*) virtual servers with random IDs that act as peers in the DHT, which reduces the load imbalance to a constant factor. To address peer heterogeneity, each node chooses to create a number of virtual servers proportional to its capacity. Unfortunately, the usage of virtual servers greatly increases the amount of routing metadata needed on each peer and causes more maintenance overhead. In addition, the number of hops per lookup (and the corresponding latency) increases as well. Moreover, it does not take object popularity into account.

Unlike the virtual server approach, the **dynamic ID** approach uses just a single ID per node [80, 74, 52, 116]. The load of a peer can be adjusted by choosing a suitable ID in the namespace. However, all such solutions requires IDs to be reassigned to maintain load balance as nodes dynamically join and leave the system, resulting in a high overhead because it involves transferring objects and updating overlay links.

The third class of approaches uses **multiple hash functions** to balance the load. The *power of two choices* [14] uses two or more hash functions to map a key to multiple nodes and store the key on the

peer that is the least loaded. In the *k-choice* [59] load balancing algorithm, the node uses multiple hashes to generate a set of IDs and at join time selects an ID in a way to minimize the discrepancies between capacity and load for itself and the nodes that will be affected by its joining. While such a strategy is simple and efficient, it increases the computational overhead for publishing and retrieving content, since multiple hash functions have to be computed each time; in addition, it is a static allocation, and does not change in the case that the workload distribution shifts.

The last category of balancing schemes is by **caching and replication** [90][102][37]. Hotspots and dynamic streams are handled by using caches to store popular objects in the network, and lookups are considered resolved whenever cache hits occur along the path. Pastry [90] and Chord [102] replicate an object on the k servers whose identifiers are closest to the object key in the namespace to improve the availability, but it also help balance the load of a popular topic. Unfortunately, the last few hops of a lookup are precisely the ones that can least be optimized [33]. Moreover, since the query load is dynamic, a fixed number of replicas do not work well; if the number is chosen too high, then resources may be wasted, and if it is set too low, then these replicas may not be enough to support a high query load.

The most significant load-unbalancing problem of our directory overlay is caused by skewed directory popularity. Therefore, we focus on this unbalance problem. Our load-balancing solution partially belongs to the last category of the aforementioned schemes. It replicates and dynamically reconfigures the routing table to balance the heterogeneous request load – the most significant problem of our directory overlay. The existing approaches, especially caching, are orthogonal to our solution.

3.3.2 Adaptive load balancing scheme

In this section, we detail our load balancing scheme, focusing on the imbalance caused by heterogeneous directory popularity. We propose a comprehensive load balancing strategy, which address this problem by dynamically re-distributing the load of hot spots to other "cold spots". Particularly, we distinguish two types of load: query answering load and query forwarding load (query load and routing load for short). Aiming at balancing these two kinds of load, we propose three balancing strategies: (1) adaptive object replication, which targets balancing the query load, and (2) adaptive routing replication and (3) dynamic routing table reconfiguration, both aimed at balancing the system's routing load. Each node analyzes the main cause of its overloading and uses a particular balancing algorithm to correct its situation. This scheme is generic enough to resolve the load balancing problem of general DHT applications. In the rest of this section, we use terms *peer* and *node* interchangeably to represent the node of a directory overlay.

3.3.2.1 Load metric

Our load balancing scheme involves a load metric to gauge the activity of each peer node and make the necessary adjustments. Each peer p in the network has a capacity C for serving requests, which corresponds to the maximum amount of load that it can support. In this thesis, this is derived from the

maximum number of queries that can be routed, answered, or queued per second by the peer. It is assumed that any arriving traffic that cannot be either processed or queued by the peer is dropped. It is also assumed that nodes will be able to define their capacity consistently via a globally ratified/used metrics scale.

At any given time, the load of peer p is defined as the number of requests received per unit of time. We focus on two kinds of requests: query routing requests and query answering requests. Upon receiving a routing request, the peer checks its routing table and forwards the query to next hop. If it receives a query answering request (meaning that it stores the hashed key locally), it serves that request according to the application's needs, for example, for our directory overlay, this may include retrieving sub-directories or registering peers to the VOs. For other applications, this may include answering a complex query, or transferring a file, and so on. In this thesis, the current load value L of a node is defined in Equation (1) as the sum of its current routing load and its current query load:

$$L=L_r + L_q \qquad (3.1)$$

$$L=(a\times\sum q_i + b\times\sum r_i) \times l \qquad (3.2)$$

Both the routing and query load can be represented by the number of requests received in unit time. Assuming that the unit load is l, and each routing request creates a unit load while each query request creates b unit load, then (1) can be converted to (2), in which $\sum q_i$ is the number of query requests in unit time, and $\sum r_i$ is the number of routing requests in unit time.

For any given peer p, we also define an overloading threshold value, T_o, which represents the point after which additional workload placed on the peer will induce overloading, and trigger load redistribution for p. This value can be represented as a portion of the peer's capacity, e.g., $T_o = 0.8C$, which means that p is considered overloaded when it reaches 80% of its capacity. We also introduce another load threshold value, T_s, that represents the "safe" workload capacity for a peer. A peer will agree to accept redistributed load from an overloaded peer only when its load is below T_s, e.g., $T_s=0.6C$. The goal of load redistribution is to make the workload on all participating peers fall below their respective Ts in order to guarantee that none of them will again be overloaded soon after the redistribution.

3.3.2.2 The adaptive object replication algorithm

Nodes storing very popular objects are susceptible to becoming overwhelmed due to external requests for those objects. In this case, attempting to redistribute the load via shedding objects and keys to other nodes does not guarantee any noticeable improvement, since even one very popular key could overload a node. Therefore, we suggest a replication-based method to relieve the load of overwhelmed nodes. By replicating the popular keys of overloaded nodes to lightly loaded nodes, we help to balance the network load. While this idea of balancing by replication is by itself not new, the *when*, *where*, and *how* we propose are. Specifically: When does replication occurs? Where do we locate the candidates to help out

an encumbered node? And how do the consequences of the redistribution get announced to the rest of the system?

When: Each peer periodically checks its current load via the previously mentioned load metrics. If it is above the overloading threshold (i.e., $L > T_o$), and this overloading is caused mainly by query loads, it will pick a lightly loaded node to replicate its keys thus sharing the load. When more than one peer is responsible for a popular key, each responsible peer only manages part of the load, thus reducing the chance of overloading.

Where: Upon detecting that it has crossed the "overload" threshold, a node will issue a replica discovery query to the network, broadcasted (with limited steps) down the DHT broadcast tree with the querying node as the root. Any lightly loaded node (defined previously as nodes with current load $L<T_s$) in the path of the tree will reply with its load information. Once enough responses have been received, the overloaded node begins transferring its keys and objects to these candidates, creating replica nodes of itself.

How: Once replicas are created, dissemination of information about the existence of these new replica must occur. For prefix-based DHTs like Pastry or Tapestry, the replica information is updated at all the peers in the original peer's neighborhood set, leaf set, and routing table. Those nodes in turn update their own state based on the information received. Similar to the node joining process, the total cost for the replica update in terms of the number of messages exchanged is $O(log_2{}^b N)$. Similarly, for Chord-based DHTs, the replica info is updated at the fingers and predecessors of the related nodes to reflect the addition of this replica, requiring $O(log^2 N)$ messages. This process can be carried out asynchronously, since the peers in the routing table already have a pointer to the original peers and asynchronous updates will not negatively affect the correctness of the system. When a query needs to be forwarded to a popular key, neighboring nodes can now pick peers in a round-robin fashion from the list of available peers holding the key. Thus, the queries for the hot key are now partitioned among the multiple peers storing the key. When a popular key later becomes unpopular, the replica nodes can just get rid of the replicated keys, using access history to gauge the popularity of the replica.

3.3.2.3 Adaptive routing replication algorithm

Replicating popular keys relieves the query answering load of nodes responsible for these keys. However, another major source of workload in DHT overlays is caused by relaying queries among nodes. A node may be overwhelmed simply by the traffic of forwarding incoming routing queries. For example, the last hop neighbors of a popular key can be overloaded by forwarding queries to the popular node. While this problem can be partially solved by the aforementioned replication of popular keys to disperse the traffic, it cannot completely alleviate the problem since certain nodes in the system might still be functioning effectively as traffic hubs for popular sections of the network. To address this problem, we propose a balancing scheme which actively redistributes the routing load of an overloaded node by duplicating its routing table to other nodes, thereby sharing its routing load. When a node is

overloaded by routing loads, it will pick a lightly loaded node to replicate its routing table, so that the replica node can share its routing load.

```
// n is the original node, r is the replica node
// this function propagates the replica info by updating related nodes' finger tables
n.propagateReplica(r)
{
  for i=1 to m        //m is the number of entries in n's routing table
     //find last node p whoes i^th finger might be n
     p=find_predecessor(n-2^(i-1));
     p.update_finger_table(n,r,i);
}

//ask node n to find id's predecessor
n.find_predecessor(id)
{
  n'=n;
  while (id∉ (n',n'.successor])
     for i=m down to 1
        if (n'.finger[i].node ∈(n',id)
          n'=n'.finger[i].node;
}

// if n is the i^th finger of p, update p's finger table by adding replica r to n's entry
p.update_finger_table(n,r,i)
{
   if (n=finger [i].node])
      finger[i].node.addEntry(r);
      x=predecessor; //get first node preceding p
      x.update_finger_table(n,r,i);
}
```

Figure 3.4 Algorithm of replica propagation in Chord

To let replicas share the responsibility of routing, their information must be propagated to other related nodes in the network. For Chord-based DHTs, the replica info is updated at the fingers and predecessors of the related nodes to reflect the addition of this replica. Figure 3.4 shows the pseudocode of the replica propagation algorithm. The total cost for the replica update in terms of the number of messages exchanged is $O(\log^2 N)$. Similarly, for prefix-based DHTs like Pastry or Tapestry, the replica info is updated at all the peers in the original peer's leaf set and routing table. Those nodes in turn update their respective routing tables by adding a replica entry to the entry of the original node so that future queries can be routed to either the original node or the new node, all the while maintaining system network correctness. This process requiring $O(log_{2^b}N)$messages exchanged and can be carried out asynchronously, since the peers in the routing table already have a pointer to the original peers and asynchronous updates will not negatively affect the correctness of the system. Besides load balancing, this replication approach can also improve the routing resilience in the face of network failures.

Figure 3.5 shows an example of the Pastry structure with the replication of routing tables. The query for item ID *0221*, which is actually served by node *0222*, is initiated at node *3012*. According to its routing table, node *3012* chooses *0021* as the next hop. Node *0021* determines that node *0200* should be the right node to forward the query. Since node *0200* has a replica at node *1102*, node *0021* may choose *1102* as the next hop. When the query is sent to *1102*, it uses the duplicated routing table for *0200* to serve the query and send the query to the destination node *0222*. When node *0200* is exposed to a high load, the replicas will share some of the traffic, preventing overload.

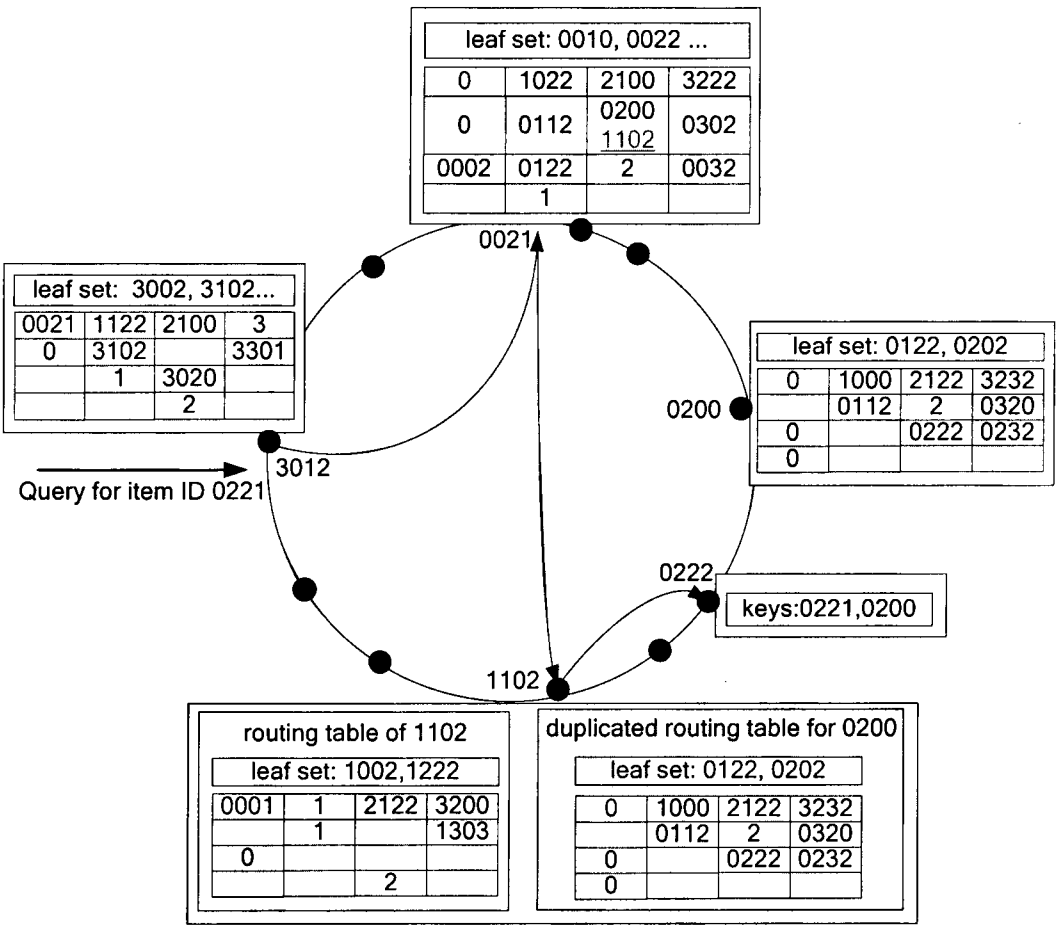


Figure 3.5 An example of adaptive routing replication algorithm

3.3.2.4 Dynamic routing load adjusting algorithm

In addition to the use of replication, another scheme to balance the routing load is by dynamically reconfiguring the routing table. In the previously mentioned methods, an overloaded node actively redistributes its own load, but in cases where external policies or the network environment prevents the redistribution effort, replacing routing table content can help relieve highly loaded nodes.

This algorithm is tailored specifically for DHTs like Pastry or Tapestry. In those systems, many nodes with the same prefix can be potentially filled in a node's routing table; the one in the table is the one that

is "closest" to this node according to the topological proximity. We propose changing the strategy of choosing nodes in the routing table to balance routing load (especially to relieve heavily loaded nodes). In lieu of simply choosing according to a proximity metric, we choose according to routing load instead. When an overloaded node receives a querying message from its neighbor, it will reply with a message indicating its overloaded status. This neighbor, receiving the message, will, at the earliest opportunity possible, replace the entry of the overloaded node in its routing table with another node with the same prefix. The light-loaded candidate nodes can be learned from forwarded query messages which include Ids of passed nodes, or by broadcasting a candidate-discovery query as the aforementioned replicating schemes did. By doing so, traffic is shed from the overloaded load as long as it is not the actual "end target" of the query request, as the replacement node will be able to direct any queries the original node could have, and forwarding traffic is spread out more evenly.

Continuing from our example in Figure 3.5, in node *1102*'s routing table, let us assume that a neighbor node, *3200*, (1st row 4th column) is heavily-loaded. When a query passes through node *3012* to *0021* and then comes to node *1102*, since *3012* shares the identical first digit prefix (3) with the overloaded neighbor *3200* in 1102's routing table, the entry of *3200* will be replaced with *3012*. This way, the traffic to the more heavily loaded *3200* will be redirected to the more free *3012*.

3.3.2.5 Dynamic load splitting algorithm and caching

It is possible that the whole directory overlay is full – most nodes are experiencing high loads. In practice, this is detected when overloaded nodes cannot find a replicating node easily. When this happens, trying to use the previously mentioned algorithms to relieve an overloaded node is in vain. Instead, we need to add new nodes to the overlay to take some load. When a new node registers to an overloaded overlay node in charge of a domain, the overloaded node will split its load and let the new node join the overlay network and share part of its load. The sharing can be implemented by letting the new node choose a suitable ID (close to the overloaded node) to take some directories or replicate the directories to the new node, depending on the nature of the overloading.

Caching popular directories is another effective solution for alleviating the load of a hot-spot. Caching provides a way to speed the performance of domain resolution for subsequent queries of popular directories, while substantially reducing query traffic on the network. Caching also helps improve the availability of the system by the ability to jump over namespace partitions induced by network failure. Our directory overlay uses recursive queries and allows en-route caching of records. After a query has been resolved, all the intermediate nodes that forward the query back to the querying node can store a local copy. Thus, subsequent queries for the same content that cross any of the nodes with cached copies can be answered immediately. As a result, the number of hops needed to resolve a query is decreased. Caching is orthogonal to our load balancing scheme; it cannot replace our routing scheme, because caching is random and ad hoc, it helps redistribute the load of popular node, but it cannot guarantee to avoid overloading.

3.4 Experiments

In this section, we examine the performance of our proposed directory overlay with simulation experiments. Since our directory location problem is similar to key search in a DHT, we do not list the general locating performance here, rather, we focus on evaluating how our load balancing algorithm and caching improves system performance.

3.4.1 Methodology

Data set: We base our simulation framework on a data set of the RDF dump of the open DMOZ directory [133], since it consists of realistic data about the content distribution within a large community. DMOZ is an open directory project (ODP) maintained by a large community of volunteers on the Web. Participants of the open directory project manually categorize web pages of general interest into a topic hierarchy. Editors contribute links to web pages, define subtopics and associate related topics to the DMOZ topic pages. This kind of metadata is one of the first metadata available on the Web in significant quantities and it is useful to provide hierarchically structured access to high-quality content on the Web. It is one of the largest efforts to manually annotate web pages, exporting all this metadata information in RDF format. Over 65,000 editors are busy keeping the directory reasonably up-to-date, and the ODP now provides access to over 4 million web pages in the ODP catalogue. The DMOZ data is available as two big RDF dumps, one for category hierarchy information (structure.rdf.u8.gz) and one for links within each category (content.rdf.u8.gz). We use the category hierarchy file in this experiment to simulate the ontology directory. For the topic distribution we select topics in the first four levels of the DMOZ hierarchy. According to a previous research effort [72], the hierarchy topics are distributed with a heavily tailed Zipf popularity.

Query: Since both keyword-based queries and browser-based queries are eventually converted to directory-path/sub-path queries, in this experiment, we only generate directory-path queries. Queries are generated by instantiating the topics chosen from the set of DMOZ topics. We use both uniform query distribution and Zipf distribution to simulate the query requests.

Topology: The directory overlay is built on Pastry. Each peer in Pastry is assigned a 160-bit identifier. The unique key of the directory is generated using SHA-1 [24] secure hashes. For a network of N peers, Pastry routes to the numerically closest peer to a given key in less than $log_{2^b}N$steps, where the identifiers use a sequence of digits with base 2^b. In our simulation, the value of base b is 2.

Other parameters: Each node is randomly assigned a value C representing its capacity ($C=2^i$, $i \in \{0, 1, 2, 3, 4\}$). A node's current load is represented by the number of query forwarding requests and query answering requests it receives per unit time. The load caused by the two kinds of requests has different weight to simulate the different costs they would incur. In our experiment, the query load is a simple question answering procedure, such that we can set the ratio of the weight of query answering load vs. query routing load to *5* (i.e., $a:b = 5:1$ in Equation 3.2). Given the lightness of the directory locating

process in the current experiment, this would be a reasonable projection. In the case of more significant operations, such as file transfers, the ratio will be larger by several orders of magnitude.

The simulation is carried out on an overlay network with 10^3 nodes and the directory topics randomly distributed throughout the nodes. Queries are issued with different frequencies and distributions (random distribution and Zipf distributions with different α value, which represents how skewed the distribution is, with a larger α value indicating greater levels of skew). For the purpose of our experiments, the T_o (overload) threshold for each node was set at 0.8, and the T_s (safety) threshold at 0.6 of its maximum capacity. Each experiment is run ten times with different random seeds, and the results are the average of these ten sets of results.

Four different load balancing strategies were evaluated and analyzed: (1) simple Pastry: this is the basic Pastry system with no load balancing strategy used (represented by *None* in the following figures), (2) reconfiguring routing table (*RR*), (3) duplicating objects/directories (*DO*), (4) duplicating the routing table (*DR*), (5) integrating all of the previous three balancing schemes (*All*). The performance metric we used is the *load/capacity* ratio.

3.4.2 Results

Effect of query distribution

Figure 3.6 shows the effect of query distribution on a node's load burden (without any balancing mechanism used), indicating the mean, 1st and 99th percentiles of the peer workload/capacity ratio. This percentile represents the workload variances on the peers, such that the greater the difference, the less evenly the load is being distributed. In the experiment, we increase the skew degree of the query distribution from random to Zipf with α=1.25. We can see that query distribution has a significant impact on peer load. The more skewed the query distribution, the more unevenly distributed the load becomes, causing some nodes to suffer from a very high load when the query is sufficiently skewed.

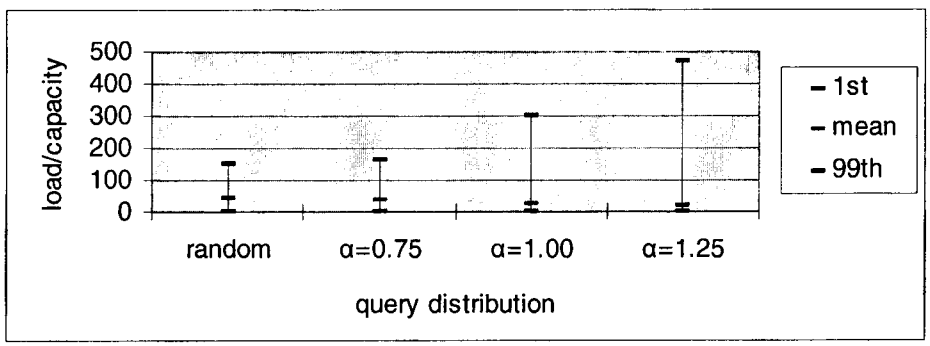


Figure 3.6 Percentiles of the ratio of load/capacity under different query distributions on a network without any balancing algorithms used (None)

Performance of load balancing schemes under different query distributions

Overloading a node can induce an overflow to its request queue, causing new incoming queries to be dropped, which in turn deteriorates the system performance. Figure 3.7 shows an overview of the fraction of dropped queries under different query distributions and with each of our load balancing schemes. We can clearly see that each of our load balancing algorithms reduce the fraction of dropped queries, thus improving the system performance. Specifically, algorithm *All*, which integrates all of the other algorithms we presented earlier, experiences the best performance in terms of minimizing the query drop rate even under a highly skewed query distribution.

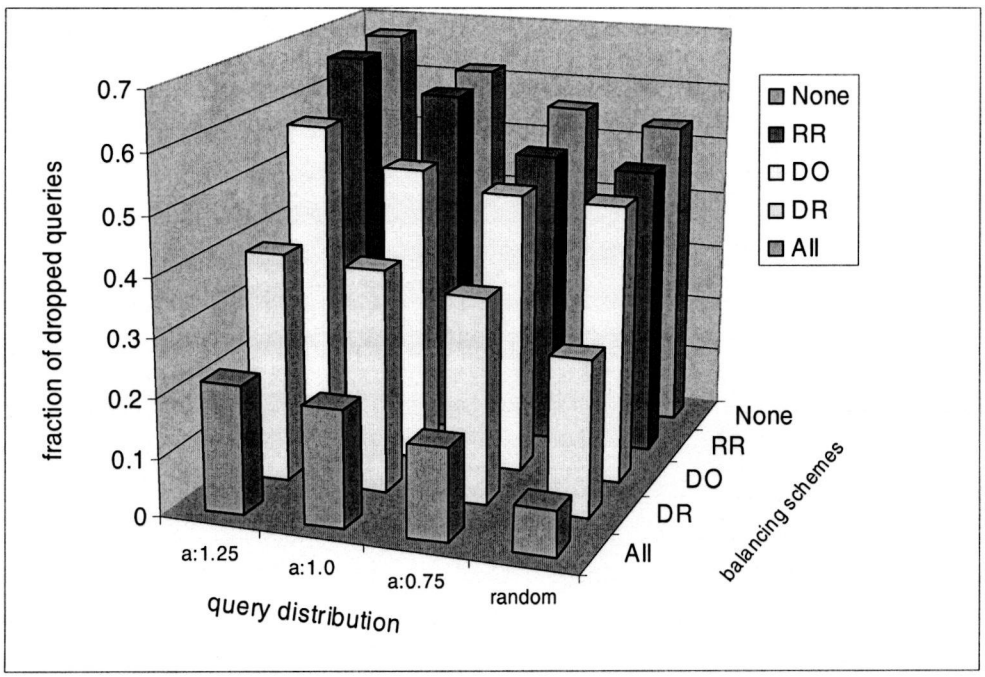


Figure 3.7 Fraction of dropped queries under different query distributions and load balancing schemes

A caveat worth mentioning is that in Figure 3.7, we can see that duplicating routing table entries reduces the number of dropped queries more than duplicating objects does. Note that this is dependent on the parameters we set, particularly the query load to routing load ratio (*a:b=5:1*). If the ratio is larger, it means that the query answering is more complex compared to the query forwarding, thereby accounting for more of the total load. From the figure, we see clear indication of the effectiveness of our proposed algorithms. The following is a more in-depth examination of the results of each of our balancing schemes:

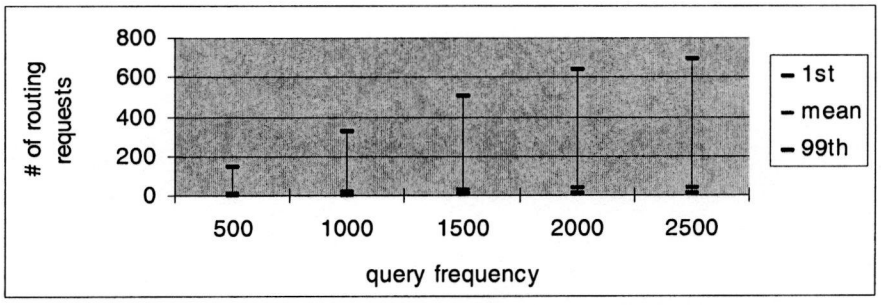
Balancing of routing load

Figure 3.8 illustrates the performance of each query routing-related balancing algorithm relative to the query insertion rate. The network size is 10^3 and the query distribution is Zipf (α =1). The figure shows the percentile of the routing load in terms of query forwarding requests received. As mentioned, the smaller the difference, the better the load balancing performs. As we increase the query frequency, the variance for all the algorithms becomes larger. This is because query distribution is skewed, so

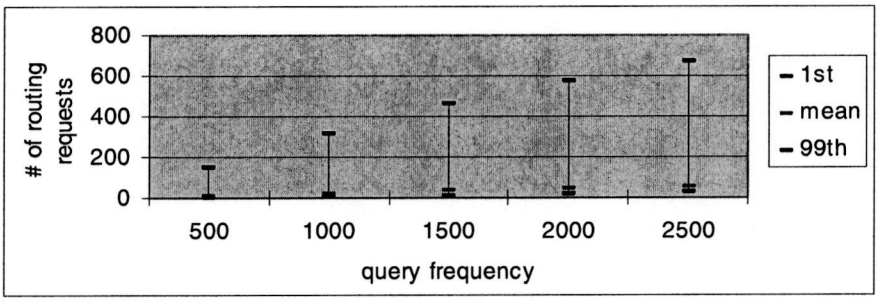
increasing the query frequency will result in more unbalanced requests, exacerbating the existing imbalance problem.

While the majority of the experimental results are as we expected, the re-configuring routing table scheme contributes surprisingly little to performance gain. We attribute this observation due to the following: (1) Prefix requirements for the bottom rows of a node's routing table are more stringent, therefore candidates for the replacement nodes of these rows are more difficult to find, resulting in the algorithm being unable to efficiently adjust this part of the routing; (2) consequently, the last-hops-neighbor of a node cannot find replacements to route to that node. This means that neighbors (in Id space) of a popular node can not be effectively relieved.

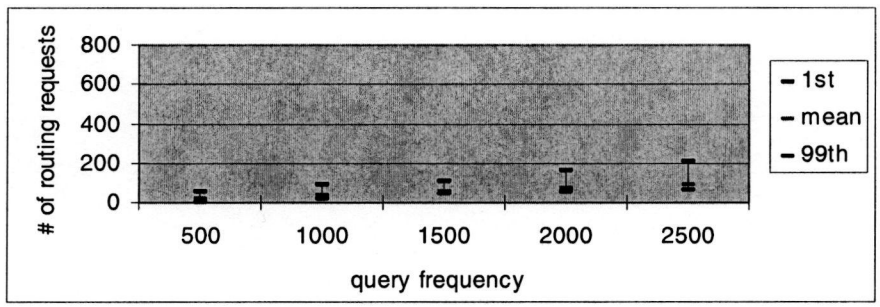
We can also observe from Figure 3.8 (d) that by integrating all of the schemes together, we are able to achieve performance beyond the sum of the benefits from all these algorithms. We surmise that this is due to the fact that although duplicating-objects does not balancing routing loads directly, it redistributes the load of hot spots, helping to relieve the traffic towards the hot spots and thus avoiding overloading the neighborhood with forwarding requests.



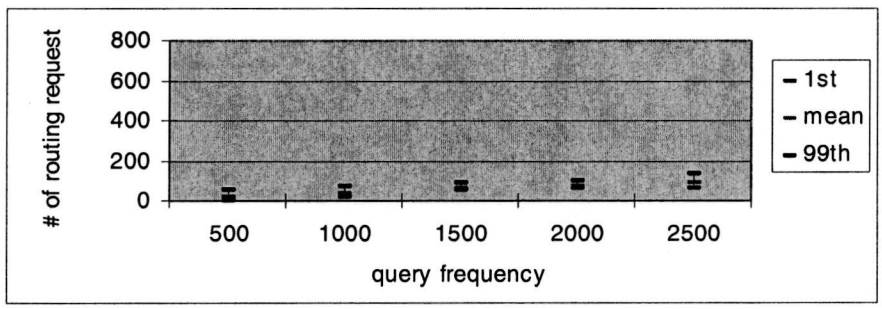
(a) Pastry, without any balancing adjustments



(b) Balancing by dynamic re-configuring routing tables



(c) Balancing by replicating routing tables

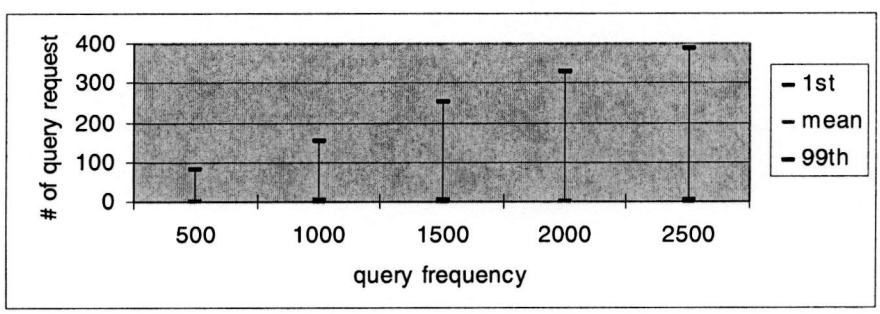


(d) Balancing by combining all schemes

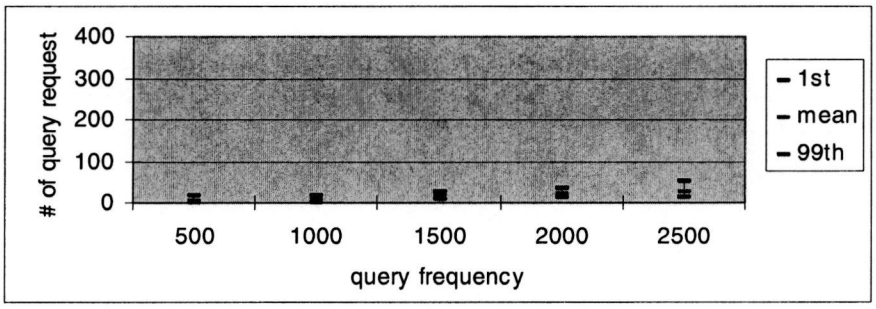
Figure 3.8 Percentiles of the routing load (in terms of number of routing requests) under different query frequencies

Balancing of query answering load

Figure 3.9 shows the performance of the adaptive object replication algorithm. We can see that the algorithm effectively relieves the overloaded nodes and balances the load because hot items are quickly replicated in other nodes in the network.



(a) Pastry, without any balancing adjustments

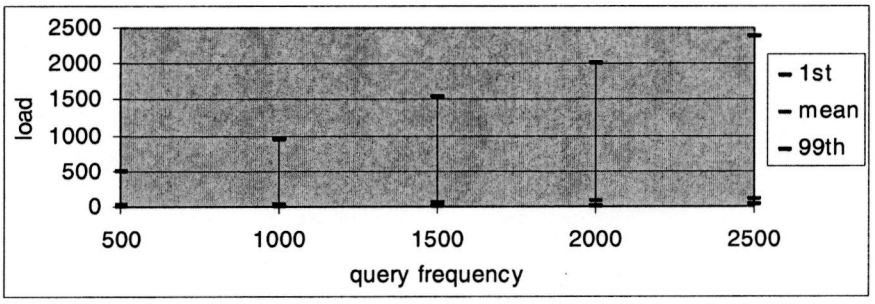


(b) Balancing by replicating objects

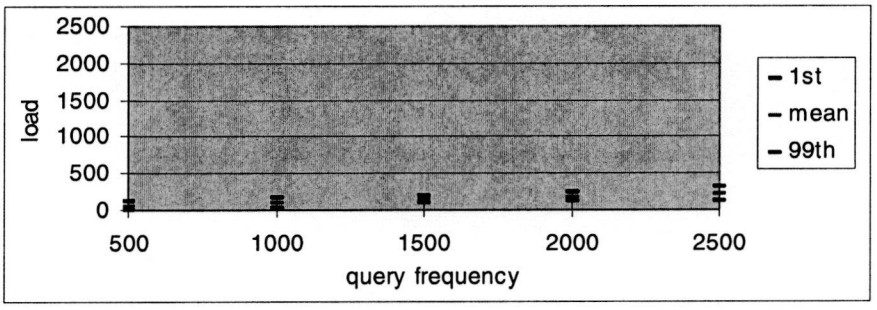
Figure 3.9 Percentiles of the query load (in terms of number of query answering requests) under different query frequencies

Balancing of the whole system load

Figure 3.10 shows the results of the combined algorithm in balancing system load. The results of the experiment clearly indicate a significant and drastic effect on the system load balance.



(a) Pastry, without any balancing adjustments

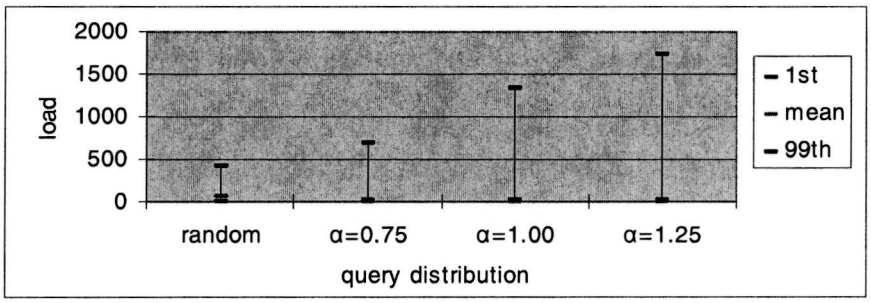


(b) Balancing by combining all schemes

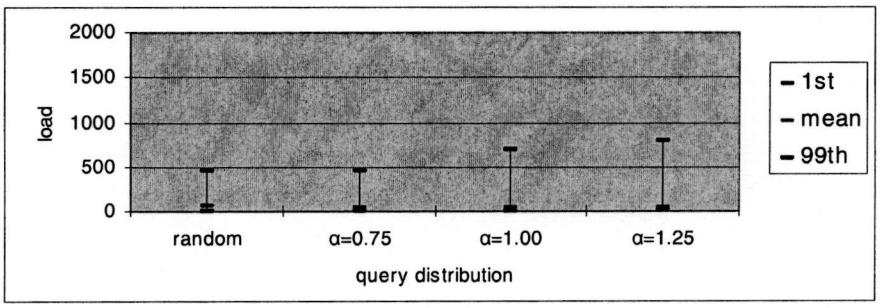
Figure 3.10 Percentiles of the system load under different query frequencies

Effect of caching

Cache can relieve the load of hot spots, thus improving load balancing. Figure 3.11 shows the load balancing effect of caching on the system. The experiment runs in two different modes: (1) caching-disabled, where every query traverses the whole routing path to reach the destination node which replies back with an authoritative answer, and (2) caching-enabled, where intermediate nodes can use previously cached records to speed up lookups. The experiment is performed without using any other load balancing schemes. We see that caching can significantly improve load balancing.



(a) Without cache



(b) With cache

Figure 3.11 Effect of caching on load-balancing

As shown in Figure 3.12, caching can also reduce the latency of searching by reducing the number of hops to the destination.

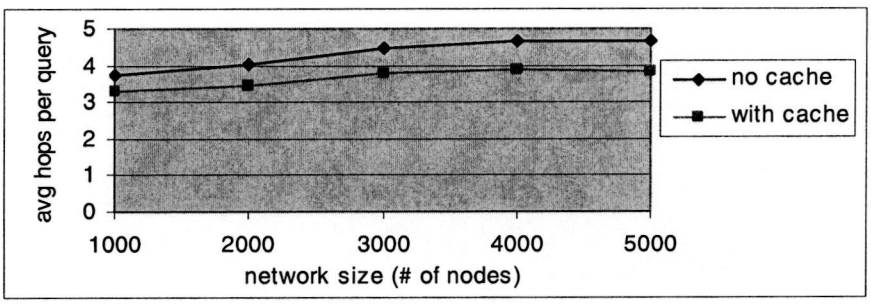


Figure 3.12 Effect of caching on query latency

3.5 Conclusion

In this chapter, we have presented an ontology-based directory overlay that helps reconfigure the network topology to form semantic virtual organizations, so that queries can be focused in semantically related organizations only. The ontology hierarchy is indexed in a flat DHT overlay, providing nodes with flexible interfaces to locate domains of interest efficiently in a decentralized fashion. To overcome the major problem of the DHT-based directory overlay – load unbalance – we have proposed an effective scheme to balance load of the directory overlay. This scheme enables the system to achieve good load balance even when demand is heavily skewed. Extensive simulation results indicate significant improvements in maintaining a more balanced system, leading to improved scalability and performance.

Chapter 4
Semantics-based Resource Discovery in Virtual Organizations

In the previous chapter, we presented an ontology-based model to facilitate nodes in forming virtual organizations (VOs). The next task is to efficiently share and search inside VOs. Searching and sharing within VOs is still very challenging, since the heterogeneous, distributed, dynamic, and large-scale properties of the problem still exist. This chapter proposes an infrastructure named Grid Ontology Integration and Discovery system (GONID) for efficiently sharing and discovering resources inside VOs. P2P and Semantic Web technologies form the foundation of GONID. We propose an ontological framework for describing the structure and semantics of resource properties, in order to increase the system's expressiveness and interoperability. Specifically, the ontology infrastructure includes (1) an expressive metadata model to represent the resource information of VO members, (2) an effective mapping formalism along with corresponding reasoning algorithms to integrate heterogeneous metadata representations, (3) a comprehensive semantic query evaluation scheme to process complex SQL-like queries. The ontological framework is based on an efficient P2P indexing system that indexes the dispersed resource ontology knowledge with a decentralized DHT overlay. Ontological knowledge is decomposed into atomic elements and then indexed with DHTs. Ontology reasoning, integration, and searching are all based on the index. A complex query can be evaluated by performing relational operations such as *select*, *project*, and *join* on combinations of the atoms. A key advantage of this ontological indexing scheme is its ability to index in different granularities, as we distinguish knowledge in different levels of abstraction. The resulting prototype system, GONID-toolkit, verifies the viability of this indexing and searching infrastructure, and our evaluation using simulations demonstrates its good performance.

The main contributions of this chapter are as follows:

1. We propose an ontological model in which resource knowledge can be effectively expressed, integrated, and queried.
2. We design a DHT-based indexing strategy to register and retrieve resource knowledge with different degrees of abstraction.
3. We implement the proposed architecture with a prototype system, the GONID toolkit.
4. We evaluate the performance of the system via simulation.

Portions of this chapter have been published as Li *et al.* [63, 61].

In the rest of this chapter, we introduce the semantic building blocks of the system in Section 4.1, including the resource metadata representation, integration, and reasoning. In Section 4.2, we describe

how to publish the metadata information in different granularities on a DHT overlay. In Section 4.3, we illustrate how to solve complex SQL-like queries with examples. We explain the prototype implementation, deployment, and evaluation in Section 4.4. Finally, we present an evaluation of the system's performance and properties in Section 4.5.

4.1 Semantic building blocks

A major focus of our discovery solution is to provide intelligent semantic search to overcome the problem of traditional keyword-based search. In a traditional search process, a user provides one or more keywords, and then the system locates documents/files containing these keywords. The traditional searching system does not deal with the domain knowledge; therefore it has difficulty to understand the meaning of a user's search request. This severely limits its searchabilities and hardly guarantees the precision and recall of the search process. Although recent search engines try to make up this by means of query expansion or Latent Semantic Analysis (LSA) [34], the fundamental problem is not addressed, because these approaches do not deal with the meaning of the keywords that a user provides, and they do not understand the semantic relations among the terms that a document contains with respect to a specific domain.

To overcome these shortcomings of keyword-based search, we employ ontology domain knowledge to assist in the search process, so that queries can be properly interpreted according to their meanings in a specific domain with the inherent relations between concepts also being considered. In our view, four problems must be addressed to use domain knowledge in the VO search: how to represent resource metadata, how to mediate schemas between heterogeneous metadata, how to efficiently index and retrieve metadata, and how to evaluate complex queries based on the metadata. In the rest of this chapter, we will present our solutions to these four problems.

4.1.1 Ontology-based metadata representation

Metadata, the data about data, is a crucial element of our VO discovery infrastructure. An effective way of locating resources of interest within large-scale resource intensive environments is providing and managing metadata about resources. Therefore, the goal in devising a successful metadata description scheme is to make it detailed enough and structured enough that users can flexibly customize their queries while ensuring that the system is still capable of efficiently locating the related resources. More important, metadata should be able to express the meaning of resource information. But meanings can be considered as a "locally constructed" artifact, as described by Brasethvik [12], so that some form of agreement is required to maintain a common space of understanding. In consequence, our metadata requires shared representations of knowledge as the basic vocabulary from which metadata statements can be asserted. An ontology, "a shared and common understanding of a domain that can be communicated between people and application systems", as considered in modern knowledge engineering [38] is precisely intended to convey that kind of shared understanding. An ontological

representation defines concepts and relationships. It sets the vocabulary, properties, and relationships for concepts. The elements accumulate more meaning by the relationships they hold and the potential inferences that can be made by those relationships. This capability of formal ontologies to convey relationships and axioms make them ideal vehicles for describing the vocabulary for metadata statements, providing a rich formal semantic structure for their interpretation. Therefore, we use ontologies to represent resource metadata semantics. The combination of metadata description and ontology engineering forms the foundation of GONID's semantic architecture.

To cope with the openness and extensibility requirements, we adopt two W3C recommendations: the Resource Description Framework (RDF) and the Web Ontology Language (OWL) as our ontology language. As an overview of RDF and OWL appears in Chapter 2, here we briefly emphasize the key concepts we use.

We concentrate on RDF's property of making statements about resources in the form of *subject-predicate-object* expressions, called *triples* in RDF terminology. The *subject* denotes the resource which has a Universal Resource Identifier (URI). The *predicate* denotes traits or aspects of the resource and expresses a relationship between the *subject* and the *object. Predicates* in RDF are also identified by URIs. The *object* is the actual value, which can either be a resource or a literal. The concept of triple is very important in our work, because our metadata indexing scheme is based on this triple representation. Applications can use tools such as Protégé [85] and OntoEdit [103] to construct and manage ontologies in RDF format. In fact, RDF is a natural way to index existing documents, such as HTML files, documents, PDF, etc. There is existing research on how to extract RDF triples from documents [40, 6, 26]; in this thesis, we assume users can use the research result to extract triples from existing documents.

We use OWL to process the content of information instead of just presenting information. OWL can be used to explicitly represent the meaning of terms in vocabularies and the relationships between those terms, i.e., an ontology. Our ontological metadata reasoning and mapping are based on OWL-DL. We use an example to illustrate an ontology that describes printers. Figure 4.1 shows the basic classes and their subclass relationships. Note that the subclass information is only part of the information included in the ontology. The entire graph is much larger. Figure 4.2 shows the part of the corresponding OWL statements in RDF syntax.

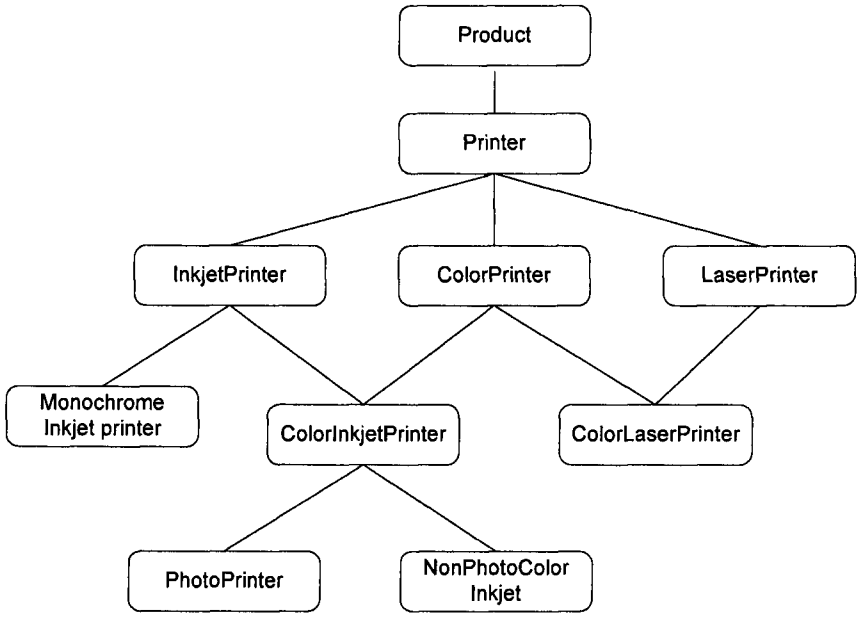


Figure 4.1 Classes and subclass-relationships of the printer ontology

```
<!DOCTYPE owl [
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema#" > ]>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns:owl ="http://www.w3.org/2002/07/owl#"
  xmlns="http://www.cs.us.ca/~juanli/thesis/printer.owl#">

<owl:Ontology rdf:about="">
 <rdfs:comment>An OWL ontology example (OWL DL)</rdfs:comment>
  <rdfs:label> Printer Ontology</rdfs:label>
</owl:Ontology>

<owl:Class rdf:ID="product">
 <rdfs:comment>Products form a class.</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="printer">
 <rdfs:comment>
  Printers are printing and digital imaging devices.
 </rdfs:comment>
 <rdfs:subClassOf rdf:resource="Product"/>
</owl:Class>

<owl:Class rdf:ID="laserPrinter">
 <rdfs:comment>
  Laserjet printers are those printers that use laser jet printing technology.
 </rdfs:comment>
```

```
<owl:intersectionOf rdf:parseType="Collection">
 <owl:Class rdf:about="#printer"/>
 <owl:Restriction>
  <owl:onProperty rdf:resource="#printingTechnology"/>
  <owl:hasValue rdf:datatype="&xsd;string">
   laserjet
  </owl:hasValue>
 </owl:Restriction>
</owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="ColorLaserPrinter">
 <rdfs:comment>
  Color laser jet printers are color laser jet printers.
 </rdfs:comment>
 <rdfs:subClassOf rdf:resource="#LaserPrinter"/>
 <rdfs:subClassOf rdf:resource="#ColorPrinter"/>
</owl:Class>
...
</rdf:RDF>
```

Figure 4.2 A printer ontology in OWL

4.1.2 Ontology mapping

For many reasons, different people and organizations tend to use different ontologies. Therefore, VOs have to deal with situations where various local ontologies developed independently are required to be integrated as a means for extracting information from the local ones. Mapping provides a common layer from which several ontologies could be accessed and hence information could be exchanged in a semantically sound manner. In VOs, the task of ontology mapping is to relate the vocabulary of two ontologies that share the same domain of discourse in such a way that the logical structure and the intended interpretations of the ontologies are respected.

4.1.2.1 Discovery of mapping candidates

Before a node can create mappings between its local ontology and other ontologies, it has to first find semantically close ontologies; these "close" ontologies have either significant overlap or strong relationships with the local ontology. The ontological directory described in the previous chapter helps narrow down the candidates to one VO, but nodes in the same VO may not be close enough to establish direct mapping relationships. The process of discovering semantically close ontologies is similar to the process of discovering other resources, which is explained in Section 4.3.3. The basic idea is to use the main conceptual structure of a node's ontology metadata, i.e., its ontology profile, as the discovery criteria to search all related ontologies. Then the users can choose the mapping ontologies from the results.

4.1.2.2 Mapping definition

Based on major OWL ontology elements, we propose two categories of ontology mappings as follows:

- Class mappings: $C_1 \xrightarrow{Mc} C_2 \quad M_c \in \{=_c, \subseteq_c, \supseteq_c, \approx_c\}$

Mc: class mapping between class C_1 and C_2

$=_c$: *equivalentClass* mapping

$\subseteq_c$: *subClass* mapping

$\supseteq_c$: *superClass* mapping

$\approx_c$: *referentialClass* mapping

In order to represent mappings between the classes of source and target ontologies, we have defined four mapping patterns: *equivalentClass, subClass, superClass,* and *referentialClass.* Two identical classes are mapped through *equivalentClass.* The *subClass* represents one class in an ontology is a more specific form of a class in another ontology, while the *superClass* means one class is a more general form of another class. The *subClass* and the *superClass* imply each other: a class A is *subClass* of a class B, then class B is *superClass* of class A. The *equivalentClass* also implies a *subClass* relationship and is always stated between two classes: A is the *equivalentClass* of B, then A is *subClass* of B, and B is *subClass* of A. When a source ontology class is *subClass* of a target ontology class, all the instances of the source ontology qualify as the instances of the target ontology. The *referentialClass* pattern implies that the involved classes have overlapping content. How these two classes are related is determined through further mapping of their datatype property or object property.

Property mappings: $P_1 \xrightarrow{Mp} P_2 \quad M_p \in \{\subseteq_p, \supseteq_p, =_p, \leftrightarrow_p\}$

Mp: property mapping between property P_1 and P_2

$=_p$: *equivalentProperty* mapping

$\subseteq_p$: *subProperty* mapping

$\supseteq_p$: *superProperty* mapping

$\leftrightarrow_p$: *inverseProperty* mapping

Property mapping helps to transform properties of a class in the source ontology to corresponding target ontology properties. We define the following property mappings: *subProperty, superProperty, equivalentProperty,* and *inverseProperty.* Similar to the class mappings, the property mappings between different ontologies either refer to the same relation or one is a special (or general) case of the other. The *inverseProperty* mapping allows relate "inverse properties"; for example, the *isTaughtBy* property in one ontology is *inverseProperty* of the *teaches* property in another ontology.

As mappings define relationships between classes and properties, they are in fact Obejct Property in OWL. The mapping patterns defined can be represented through an OWL ontology shown in Figure 4.3.

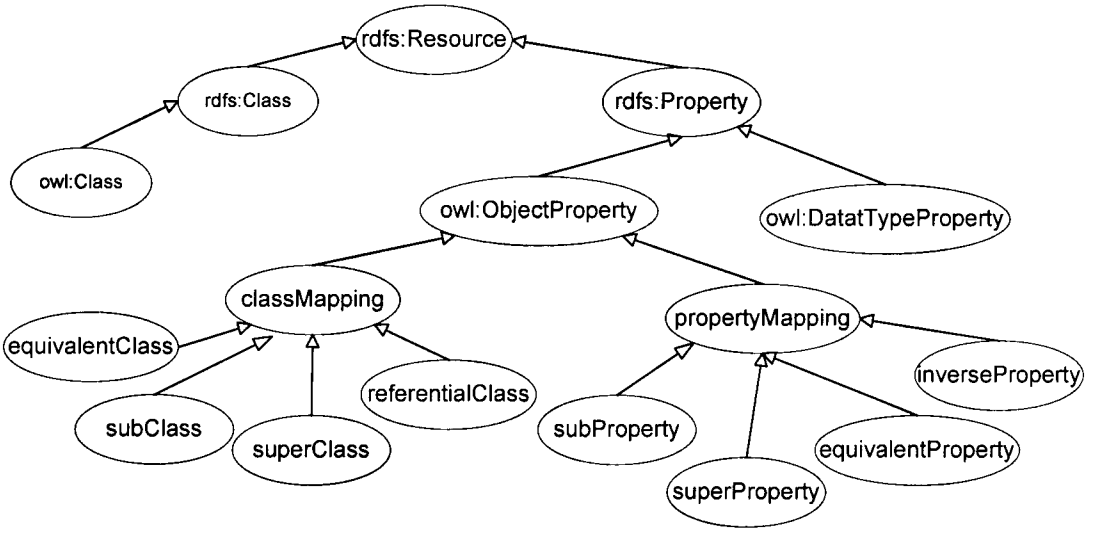


Figure 4.3 OWL-based mapping schema

An example illustrating the class mapping among four nodes with heterogeneous ontologies is shown in Figure 4.4. *Node A* and *Node C* have an *equivalentClass* mapping, stating that the *"CPU"* and *"Processor"* refer the same concept. *Node B* and *Node D* relate through *subclass* relation, allowing *Node D* to match its instances when the *"OS"* concept is queried. A *referentialClass* relation is created between the *Node A "PC"* concept and *Node B "Computer"* concept. This states that the classes describe different aspects of the same higher level concept, and their instances may overlap. They are integrated based on the property *"name"*; thus instances of these different classes can be aggregated if they have the same *"name"*.

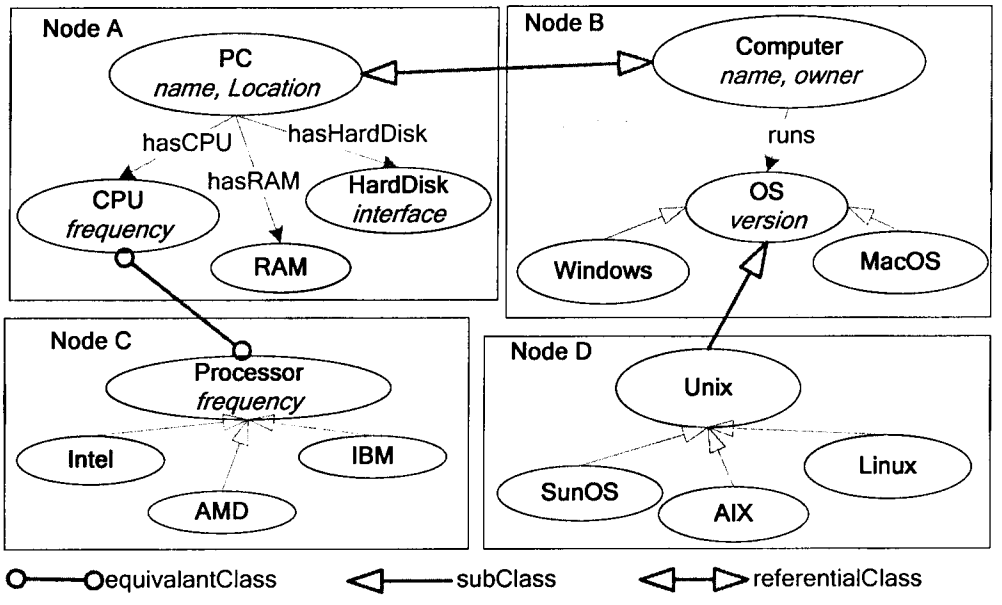


Figure 4.4 An example of ontology mapping

We propose the mapping definition, while the mapping process is beyond the scope of this thesis. Generally, it requires manual intervention. Several popular semi-automatic tools such as Protégé-PROMPT [87] and Chimaera [20] are available for eased ontology mappings. These approaches are similar to approaches to mapping XML schemas or other structured data but tend to rely more heavily on features of concept definitions or on explicit semantics of these definitions. They comprise heuristics-based or machine learning techniques that use various characteristics of ontologies, such as ontology structure, lexical components, definitions of concepts, and instances of classes, to give mapping suggestions.

4.1.2.3 Mapping indexing

The mappings should be stored in the network so that they can be used by inference engines to reformulate queries. Since our inter-ontology mappings are also represented as ontologies, they can be indexed the same as other ontologies in the network. This allows all participants of the VO to access the mapping information. The indexing process is explained in Section 4.2.

4.1.3 Reasoning

As mentioned, we use OWL-DL to represent local resource knowledge and mapping. OWL-DL can be translated into a Description Logic representation. Therefore, it is possible to perform automated reasoning using a Description Logic reasoner (DIG reasoner) on OWL-DL. Our primary objective of using the DIG reasoner is to derive additional assertions which are entailed from the base local ontology together with any inter-ontology mappings, external ontology information like WordNet, and the axioms and rules associated with the reasoner. We can perform various inferences, such as computing the inferred super-classes of a class, deciding whether or not one class is subsumed by another, determining whether or not a class is consistent (a class is inconsistent if it cannot possibly have any instances), etc. For example, if there are two assertions: "UNIX *subClassOf* OperatingSystem" and "AIX *subClassOf* UNIX" in the original knowledgebase; then we can infer "AIX *subClassOf* OperatingSystem" and add this new assertion into the knowledgebase.

The following example shows how to use mapping and reasoning to bridge the ontology differences between different users. In the example scenario, one user is trying to find "*a camera with a 75-300mm lens-size, a resolution not less than 5MP, and a cost ranging from 500CAD to 1000CAD.*" The search agent looks for resources that can fulfill this request. Assume that there exists a mapping ontology, which the search process can consult. Suppose a search agent finds the metadata document in Figure 4.5 at a peer.

To determine if there is a match between the query and the metadata, the following questions must be answered:

1. What's the relationship between "DC" and "Camera"?

2. What's the relationship between "focal-length" and "lens-size"?
3. What's the relationship between "megapixels" and "resolution"?
4. What's the relationship between "price" and "cost"?

```
<DigitalCameraStore rdf:ID="DcShop"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
   <location>4299 No.3 road</location>
   <phone>604-426-9936</phone>
   <catalog rdf:parseType="Collection">
     <DC rdf:ID="Canon-EOS-Rebel-XTi"
              xmlns=http://www.cs.ubc.ca/~juanli/camera #>
        <lens>
             <focal-length>50-400mm</focal-length>
             <megapixels>10.1 MP</megapixels>
        </lens>
        <body>
          <Body>
             <optical-zoom>3X</ optical-zoom>
            <lcd-monitor-size>2.5in</lcd-monitor-size>
             <shutterSpeed rdf:parseType="Resource">
                <min>0.003</min>
                <max>1.4</max>
                <units>seconds</units>
             </shutterSpeed>
          </Body>
        </body>
        <price rdf:parseType="Resource">
          <rdf:value>749.99</rdf:value>
          <currency>CAD</currency>
        </price>
     </DC>
  </catalog>
</DigitalCameraStore>
```

Figure 4.5 Metadata document in a peer

The search process consults the OWL mapping ontology. This OWL statement tells the search agent that a DC is a type of Camera:

```
<owl:Class rdf:ID="DC">
        <rdfs:subClassOf rdf:resource="#Camera"/>
</owl:Class>
```

This statement says that focal-length is equivalent to lens-size:

```
<owl:DatatypePeroperty rdf:ID="focal-length">
        <owl:equivalentProperty rdf:resource="#lens-size"/>
        <rdfs:domain rdf:resource="#lens"/>
        <rdfs:range rdf:resource="&xsd;#string"/>
</owl:DatatypeProperty>
```

This one means that megapixels is a type of resolution:

```
<owl:DatatypePeroperty rdf:ID=" megapixels">
        <owl:subProperty rdf:resource="#resolution"/>
        <rdfs:domain rdf:resource="#lens"/>
        <rdfs:range rdf:resource="&xsd;#decimal"/>
</owl:DatatypeProperty>
```

And the following statement tells the agent that price is equivalent to cost:

```
<owl:Class rdf:resource="price"/>
        <rdfs:equvalentClassOf rdf:resource="#cost"/>
</owl:Class>
```

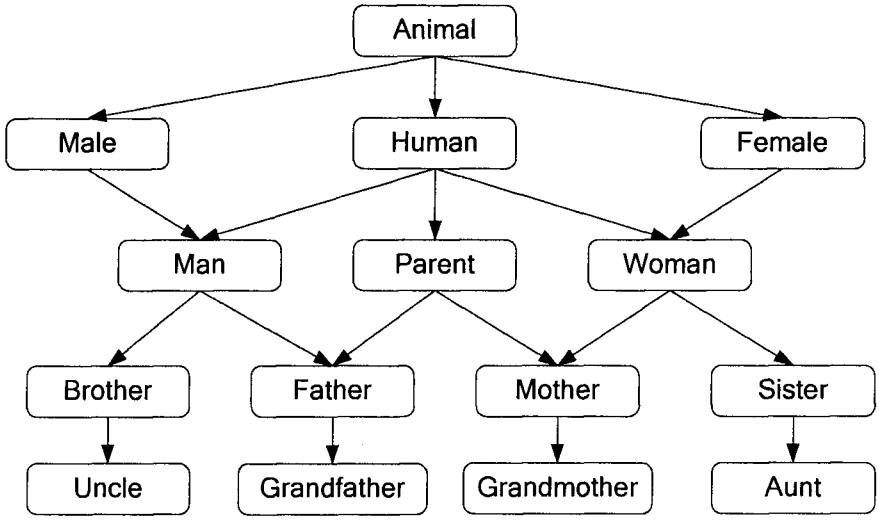
With the above mapping ontology, the search agent now recognizes that the resource metadata it found is talking about cameras, and it does show the lens-size, the resolution, the cost for the camera, and the values for lens-size, resolution, and cost are met. Thus the search agent knows that the metadata document is a match.

4.2 Metadata indexing

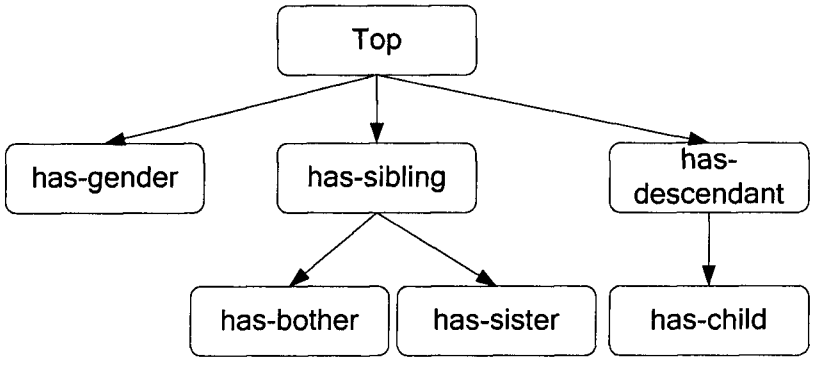
4.2.1 Peer local knowledge repository

In order to be shared and reused, the ontologies that we use including original metadata knowledge, inferred knowledge, and mappings have to be published in repositories for later reference by system designers and query agent applications. In our system the ontology knowledge is represented by OWL-DL and is separated into two parts: the terminological box (T-Box) and the assertion box (A-Box) as defined in the description logic terminology. The purpose of distinguishing between the T-Box and A-Box is to enable different coarse-grained indexing based on these two cases. The T-Box is a finite set of terminological axioms, which includes all axioms for concept definition and descriptions of domain structure, for example a set of classes and properties. The A-Box is a finite set of assertional axioms, which includes a set of axioms for the descriptions of concrete data and relations, for example, the instances of the classes defined in the T-Box. Figure 4.6 shows the T-Box and A-Box graph of a Family ontology. Examples of A-Box statements are "Betty is a Person", "Betty has-child Doris". This should be contrasted with T-Box statements about terminology such as: "All Mothers are subclasses of Woman" or

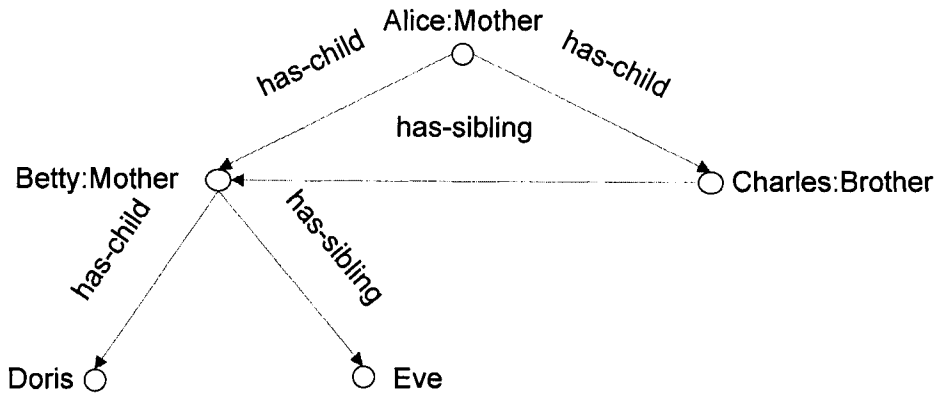
"There are different types of Human: Man and Woman". T-Box statements tend to be more permanent within a knowledge repository. In contrast, A-Box statements are much more dynamic in nature. Generally speaking, there are many more A-Box instances than T-Box concepts. Separating the T-Box and A-Box enables different coarse-grained knowledge indexing, thus increasing the scalability of the system. Both T-Box knowledge and A-Box knowledge are wrapped as RDF statements and stored in the RDF knowledge repository. The repository can store the RDF data inside a relational database or text file.



(a) Class hierarchy for the Family T-Box



(b) Property hierarchy for the Family T-Box



(c) Depiction of the Family A-Box

Figure 4.6 T-Box and A-Box graph

4.2.2 Indexing

Each peer's local ontology repository makes flexible statements about resources. However, putting an ontology document in a peer's repository does not mean that others can find it. The system needs a searching scheme to locate desirable resources from distributed repositories. For this purpose, we use a DHT-based P2P network which implements a distributed ontology repository for storing, indexing and querying resource ontology knowledge. The main purpose of an index is to reduce the number of direct accesses to the data while searching. Given a large-scale distributed repository, it is infeasible to do a thorough search of the entire component repositories. The indexing on the distributed repositories speeds up the searching process by only pushing down queries to information sources we can expect to contain an answer.

The index uses information extracted from the data to facilitate access to this data. The index should correspond, in some way, to the queries used to retrieve the data. Therefore, the main issue in choosing an indexing scheme is to decide which information to use in an index and how to organize it. Since our OWL ontology knowledge is represented with RDF syntax, knowledge indexing is based on indexing RDF statements. As mentioned, complex structures can be easily encoded in a set of RDF triples. Therefore, our indexing is based on RDF triples in the format of (*spo*), where *s* is the *subject*, *p* is the *predicate* and *o* is the *object*. This indexing scheme is consistent with RDFPeer's RDF data indexing [16].

The key advantage of our ontological indexing is its ability to handle different granularities. We distinguish T-Box knowledge and A-Box knowledge in each peer's local repository as distinguishing between schema information and the data themselves. In this way, indices can be created based on these two types of knowledge. By the combination of these two indexing schemes an application on top can choose which scheme fits the needs of the system best. The system will be able to scale to hundreds of thousands of nodes and to large amounts of ontology data and queries.

4.2.2.1 A-Box indexing

The purpose of A-Box indexing is to index individual resource information so that the right resources can be efficiently located. The basic idea is to divide a resource's RDF description into triples and index the triples in a DHT overlay. We store each triple three times by applying a hash function to its subject, predicate, and object. In this way, a query providing partial information of a triple can be handled. The insertion operation of a triple *t* is performed as follows:

Insert(t)≡Insert(SHA1Hash(t.subject), t), Insert(SHA1Hash(t.predicate), t), Insert(SHA1Hash(t.object), t)

For example, the statement *t*: {< *Billy*>, <*teaches*>, < *cs213* >} is first indexed by *subject*, and sends the following message to the overlay:

Insert {key, {("subject", <Billy>),
("predicate", <teaches >),
("object", < cs213>)}}
where key=SHA1Hash("< Billy>")

In the message, the first attribute-value pair ("*subject*", < *Billy*>) is the routing key pair, and key is the SHA1 hash value of the subject value. Similarly, the triple is indexed by predicate and object as well. The target DHT node stores the assertion and possibly generates new assertions by applying the entailment rules. These new assertions have to be sent out to other nodes. For example, transitive properties, such as *ancestorsOf*, will have a chaining effect. Thus, after finishing this process, the entire set of A-Box knowledge is accessible in a well-defined way over the community overlay. Figure 4.7 illustrates how the triple is stored into a Chord overlay. Table 4.1 shows an example of indexes stored in a node, e.g., node N_{11}.

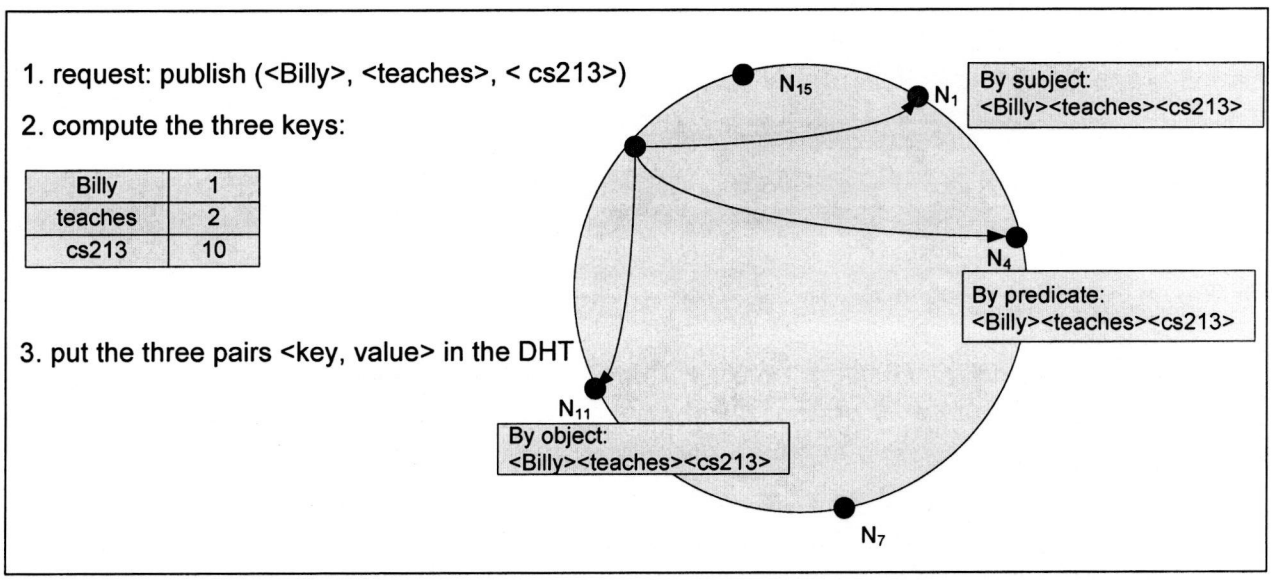


Figure 4.7 Storing a triple into a Chord overlay

Table 4.1 A-Box indexes stored at node N_{11}

subject	related triple
<comp1>	<comp1><runs ><winXP>
	<comp1><hasMem > 256M
<grp1>	< grp1><owns><comp1>

predicate	related triple
<owns>	<Mike><owns><comp1>

object	related triple
<cs213>	< Billy><teaches><cs213>
<comp1>	< grp1><owns><comp1>

With this indexing scheme, triples can be retrieved from the DHT by fixing one part of the triple and using this part as a retrieval key. The crucial problem of the triple indexing is the violation of the assumption that keys will be uniformly distributed over the network. Some elements can be so popular that nodes in charge of them get a tremendous load and it may become impossible for any single node in the network to store this key. To avoid this kind of situation, we choose not to index those overly popular elements. A node must then find an alternative way of resolving the query by navigation to the target triples through other parts of the triple. Further load imbalance may be caused by the transitive relation, in which the head of the transitive chain may get more load because of the entailment rule applied to it. Our load-balancing algorithms proposed in Chapter 3 can be adopted to solve the load-unbalancing problem.

A-Box indexing keeps each instance triple, thus queries can be accurately forwarded to the instance level. Applications with large storage requiring fast query responses would consider using A-Box indexing. The downside of indexing A-Box information is that the oversized indices of individual instances may cause large maintenance overhead, thus making the system hard to scale. Moreover, in many cases it would not even be applicable to index A-Box knowledge, e.g., when sources do not allow replication of their data (which is what instance indices essentially do). To solve this problem, we also provide another indexing scheme: T-Box indexing.

4.2.2.2 T-Box indexing

Similar to a database schema, a node's T-Box knowledge is more abstract, describing the node's high-level concepts and their relationships. Basically, the T-Box knowledge includes class elements and property elements. It also adheres to the triple (*spo*) format, while here the subject *s* is the class (or property) in question, *p* is the predefined OWL predicates describing the attribute of this class (property), and *o* is the value of the attribute of related class (property). Below is an example of a simple T-Box ontology describing a simple teaching relationship in the triple format.

@prefix univ: <http:/www.cs.ubc.ca/~juanli/univ#>
< univ:Teacher>,< rdf:type>,<owl: class>

< univ:Teacher>, <rdfs:subClassOf>, < univ:People>

< univ:Course>, < rdf:type>, <owl:class>

< univ:teach>, < rdf:type>, <owl:InverseFunctionalProperty >
< univ:teach>, <rdfs: domain>, < univ:Teacher">
< univ:teach>, <rdfs:range>,< univ:Course">
< univ:teach>,<owl:inverseOf>,< univ:isTaughtBy">

< univ:isTaughtBy>, < rdf:type>, <owl:InverseFunctionalProperty >
< univ:isTaughtBy>, <rdfs: domain>, < univ:Course>
< univ:isTaughtBy>, <rdfs:range>, < univ:Teacher>
< univ:isTaughtBy>, <owl:inverseOf>, < univ:teach>

The T-Box definition is indexed in the triple format as well. Classes and properties of the T-Box are indexed separately. The indexing process is the same as A-Box indexing – storing each triple three times by the *subject*, *predicate*, and *object* respectively. The three parts of the T-Box triple are uneven: a T-Box has only a limited number of predefined predicates, but many more objects and subjects. For example, many classes have a subclass property; each is encoded as a triple with predicate *rdf:subClass*. When indexing by the predicate, all these triples are mapped to the same key and therefore to the same peer in the network. This causes overloading of the peer in charge of the key. This problem can be solved by simply not indexing the overly popular keys; the query can be resolved by using other information of the triple.

Storing the T-Box definition is only part of the indexing task. In a VO, many nodes may use the existing T-Box instead of defining their own. Therefore, another task of T-Box indexing is to link the T-Box triples with nodes using them. This is done by extracting the T-Box concepts from a node's ontology, and then using them as the key and the node's Id as the value to do indexing. Table 4.2 shows an example T-Box index table maintained by a peer.

Table 4.2 An example T-Box index table stored in a node

concept	**peers involved**	**related T-Box triple**
<OS>	n_2, n_{14}, n_5 n_{31}, n_1, ...	<OS><superClasss><UNIX>
		<OS><equivalentClass><OperatingSystem>
<PC>	n_7, ...	<PC><referentialClass><Computer>
<UNIX>	n_5, n_2	<OS><superClasss><UNIX>
<CPU >	n_{14}, n_5	<Processor>< equivalentClass ><CPU>
<own>	n_{11}, n_{53}	<ownedBy><inverseProperty><own>
<run>	n_2, n_{14}, n_5 n_{12}, n_{23}, ...	<run><equivalentProperty><execute>
		<run><domain><Computer>
		<run><range><OS>

T-Box indexing only stores the schema information but ignores the individual instances. It has two functionalities: it helps answering knowledge schema queries; it also helps filtering the candidate result set for individual instance queries. Compared to the instance-level A-Box indexing, T-Box indexing does not require creating and maintaining oversized indices since there are far fewer concepts than instances. The down side of keeping only the schema information is that query answering without the index support at the instance level is much more computationally intensive. Obviously, there is a tradeoff between query overhead and indexing overhead. When the system has a high requirement for fast and efficient query answering, it has to pay more for the indexing. On the other hand, if the system does not index the detailed knowledge, it has to explore more nodes in searching for query results. An application should determine the right indexing granularity that can trade off the cost of maintaining the index against the benefit that the index offers for queries.

4.3 Query evaluation

Having introduced the metadata indexing scheme, we now turn our attention to how to utilize the index. Particularly this section describes how to combine lookup operations from different indexes to process queries such as finding mapping ontologies, locating certain resource providers, or answering complex questions. During the query processing, users do not need to know where a particular piece of information resides. The system behaves as if all the information is available from a single source. The query answering system can locate relevant information, retrieve it, and combine the individual answers.

4.3.1 Overview

The GONID system uses SPARQL [24] as the query language, but the query evaluation approach is not limited to a specific query language. SPARQL is a recursive acronym standing for SPARQL Protocol and RDF Query Language. Most uses of the SPARQL acronym refer to the RDF query language. The query evaluation process begins with the parsing of a user's query to SPARQL format. Then the query in terms of relations in the user's local ontology will be translated into sub-queries using the semantic mapping axioms indexed into the overlay. Then each of the sub-queries can be executed at different sources in parallel and the query engine can collect returned answers from the sources and combine them as the answer to the query. This process is illustrated in Figure 4.8. All the steps are straightforward except for matching the queries using the distributed index. The next section will explain the matching process in detail, and we assume the underlining indexing is based on A-Box indexing. Searching based on T-Box indexing is similar and is studied in Section 4.3.3. The system supports two categories of queries, T-Box queries and A-Box queries, for querying abstract structural knowledge and concrete instance knowledge. Because both T-Box and A-Box knowledge are indexed with RDF, technically speaking, processing these two kinds of queries are the same, therefore we do not need to particularly distinguish them. In addition, we assume reasoners have extended the source RDF triples with inferred triples according to the entailment rules, thus we do not need to worry about reasoning when processing queries.

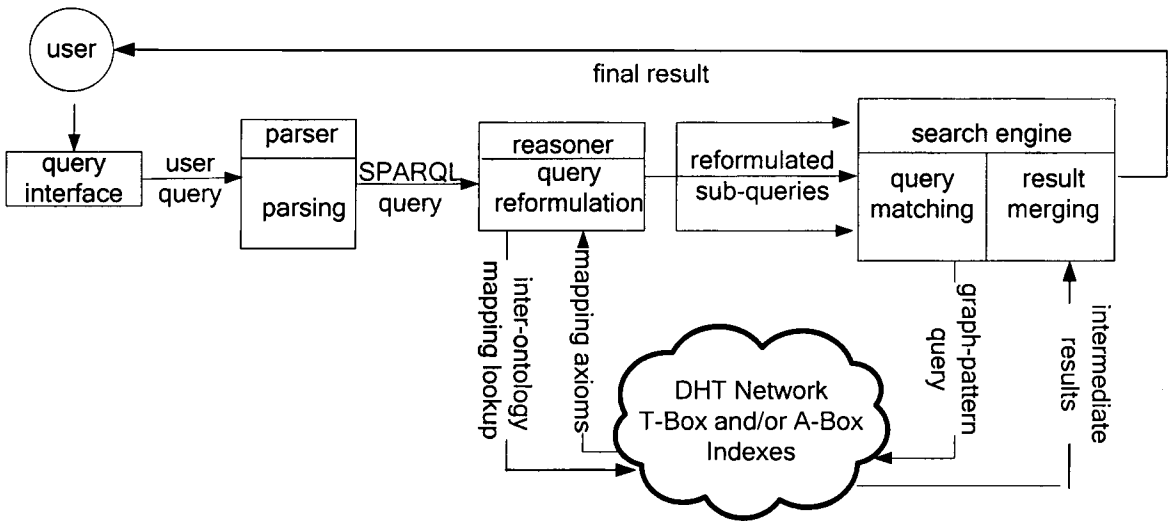


Figure 4.8 Query processing

4.3.2 Processing SPARQL queries

4.3.2.1 SPARQL query and graph patterns

The W3C recommendation SPARQL [136] is a query language developed primarily to query RDF graphs. The building block for SPARQL queries is graph patterns which contain triple patterns. Triple patterns are RDF triples, but with the option of query variables in place of RDF terms in the subject, predicate, or object. A solution to a SPARQL graph pattern with respect to a source RDF graph *G* is a mapping from the variables in the query to RDF terms such that the substitution of variables in the graph pattern yields a sub-graph of *G* [130]. More complex SPARQL queries are constructed by using *projection* (SELECT operator), *left join* (OPTIONAL operator), *union* (UNION operator), and *constraints* (FILTER operator) [98]. The semantics for these operations are defined as algebraic operations over the solutions of graph patterns [86]. Figure 4.9 shows an example RDF graph structure.

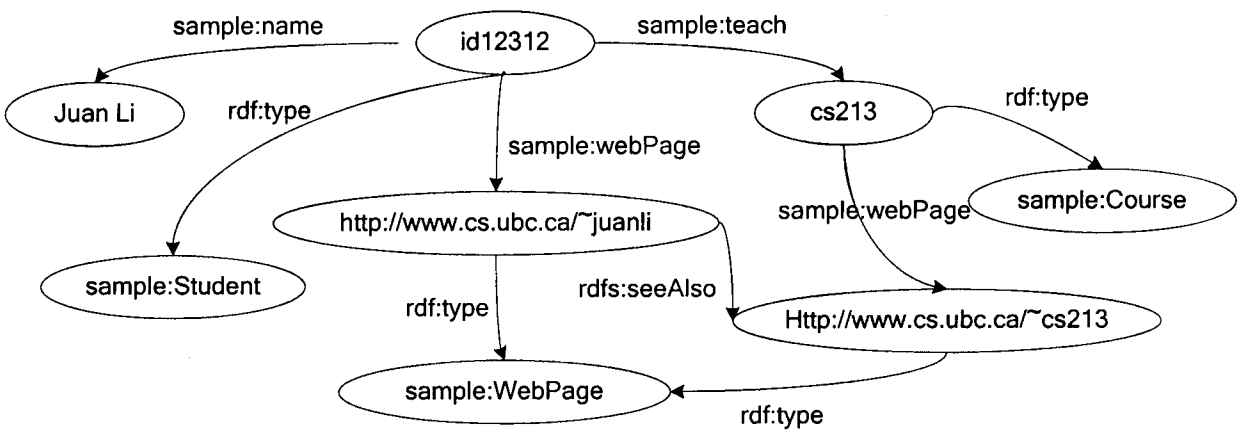


Figure 4.9 A sample RDF graph structure

A typical SPARQL query to find the courses taught by Juan Li over the graph is listed below.

```
PREFIX sample:<http://www.cs.ubc.ca/~juanli/#sample >
SELECT ?course
FROM     <example.rdf>
WHERE  {
    ?person sample:name Juan Li .
    ?person sample:teach ?course
}
```

The query attempts to match the triples of the graph pattern to the model shown in Figure 4.9. Each matching binding of the graph pattern's variables to the model's nodes becomes a query solution, and the values of the variables named in the SELECT clause become part of the query results. Next we will explain how to solve different SPARQL queries based on our index.

4.3.2.2 Single triple pattern

The simplest query is to ask for resources matching one triple pattern. To illustrate how to perform this kind of simple SPARQL queries, imagine a query to discover the person who teaches course cs213. In SPARQL this query could be written as:

```
PREFIX
sample:<http://www.cs.ubc.ca/~juanli/#sample >
SELECT ?person
WHERE  {
    ?person sample:teach sample:cs213
}
```

In this query pattern, there is only one triple pattern and at least one part of the triple is a constant. Since we store each triple three times based on its hashed subject, predicate, and object values, we can resolve the query by routing it to the node responsible for storing that constant. Then the responsible node matches this triple against the patterns stored locally and returns results to the requesting node. In this example, there are two constants in the triple pattern; the query processor can use either of them as the DHT lookup key. For example, we can hash on the object: *sample:cs213*, then use it as the key to route the query. The node in charge of this key in the DHT overlay matches triples indexed locally using this pattern, and sends back the matched triples.

4.3.2.3 Conjunctive patterns

When the graph pattern is more complex containing multiple triples, or the query contains a group graph pattern, then each triple pattern will be evaluated by one or two different nodes. These nodes form a processing chain for the query. The first triple pattern is evaluated at the first node, the result is then sent to the next node for further processing. This process continues until the last triple pattern is processed. An alternative approach is to process patterns in parallel, and all results are sent to one node to do the final processing. A system should choose the appropriate approach according to its application. In our work, we use the sequential approach since sequentially joining intermediate results saves the traffic for

transferring large amounts of unrelated data. The sequence to evaluate the triple patterns is crucial. Many database researchers have worked on it [48, 96]. Here, for simplicity, we assume that we evaluate the query with the original triple pattern order, in which adjacent triple patterns share at least one common variable.

For a query q that has k conjunctive triple patterns $(t_1, t_2, \ldots t_k)$, the query evaluation proceeds as follows: First, t_1 is evaluated using the single triple pattern processing method mentioned previously. The result is projected on the variables with values that are needed in the next query evaluation. Then the query together with the next triple sequence number and the intermediate result is sent to the node responsible for the next triple pattern. When a node n_i receives the query request, n_i evaluates the *i-th* triple pattern t_i of the query using its local triple index and the intermediate result from previous nodes. Then n_i computes the intermediate result and projects the result on columns that are needed in the rest of the query evaluation (i.e., variables appearing in the triple pattern t_{i+1} of q). This is a nested loop join on the common column for the inner relation. The process recursively repeats until the last triple pattern t_k of q is evaluated. Then, the last node n_k simply returns the result back to the querying node. We use an example to explain this process. The query to find authors who write papers in the field of P2P is listed below:

```
SELECT ?author
WHERE  {
    ?author :create ?paper .
   ?paper :category ?cat .
   ?cat :label P2P
}
```

The query evaluation process is illustrated in Figure 4.10. Each event in this figure represents an event in the network, i.e., the arrival of a new query request. The query request consists of three parts: (1) the original query, (2) the triple pattern to be processed in this node, represented with that triple's sequence number in the original query's triple lists, (3) the intermediate result from previous nodes. Initially, the intermediate result is empty (Ø).

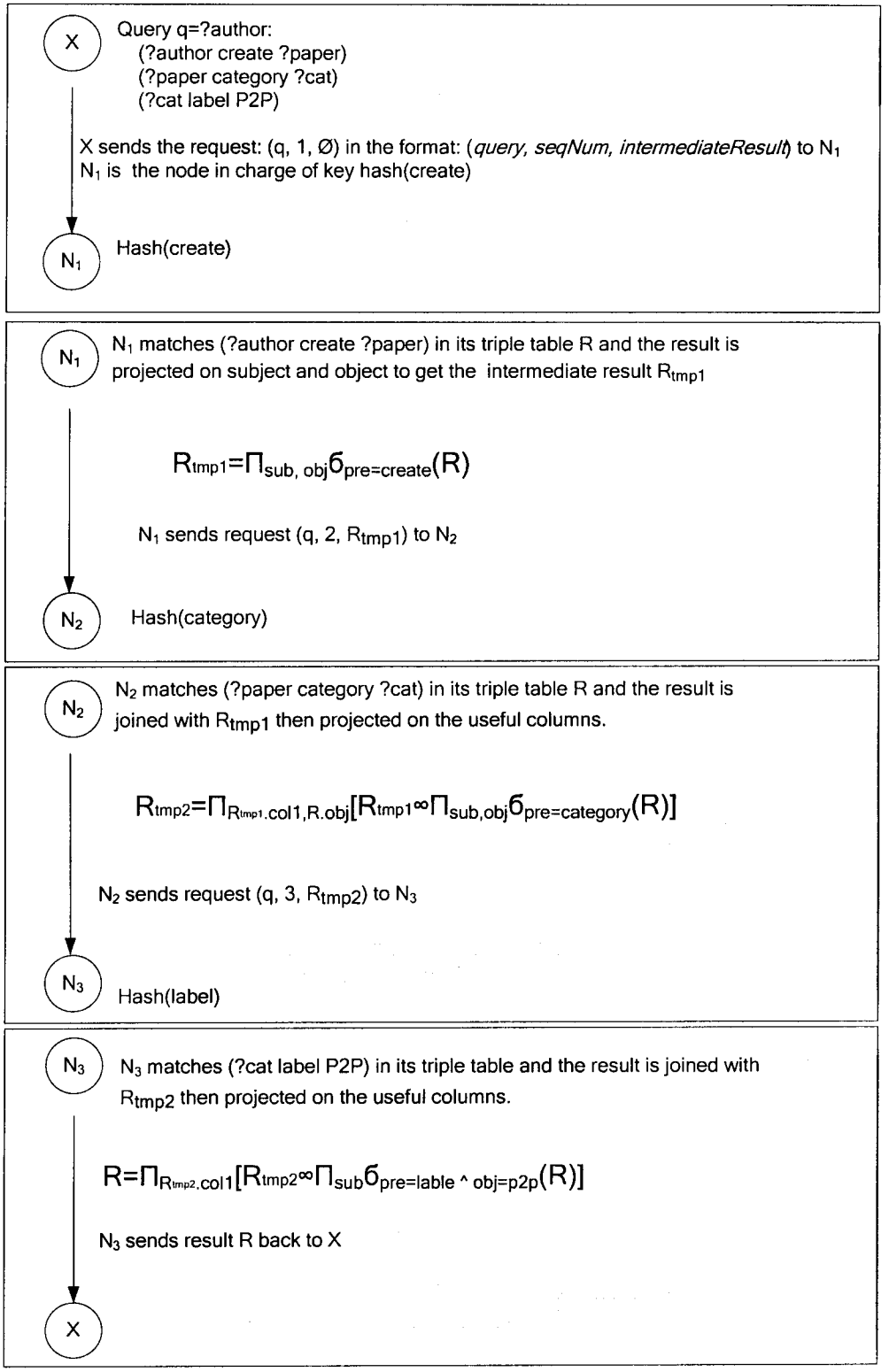


Figure 4.10 Processing a query with a conjunctive pattern

(Results are represented as relational algebras. Π:Projection, б:Selection, ∞:join)

4.3.2.4 Value constraints

A constraint, expressed by the keyword FILTER, is a restriction on solutions over the graph pattern group in which the filter appears. In the simplest case, the value constraint refers only to variables that are bound in the current group and the constraint can be mapped into an equivalent relational expression. In this case the constraint may be applied simply by selecting on the appropriate column. For example, if we have (*?x :age ?y . FILTER(?y > 30)*) we need only to select *?y* with value greater than *30*.

Sometimes constraints are placed in optional patterns (explained in next section) with variables that do not appear in that block. In this case, since the bindings for that variable are not available at the time the intermediate result is selected, the constraint can be transferred to the results processing step. Alternatively, if available, the bindings for the variables in question can be joined to the intermediate results in which they appear. Delaying the selection of the FILTER constraints is undesirable as it increases the size of the intermediate results.

4.3.2.5 Optional patterns

SPARQL's OPTIONAL operator as defined by Harris [141] is used to signify a subset of the query that should not cause the result to fail if it cannot be satisfied. It is roughly analogous to the left outer join of relational algebra. When processing queries with optional patterns, the intermediate results are produced for each pattern as before, but in the case of an optional pattern, columns that allow joining onto the required pattern must also be projected.

A query with an optional pattern is shown below. The processing of the query is similar to processing other queries mentioned. A node processes the first pattern, the required pattern, and gets the intermediate result $R_{tmp.}$(Equation 4.1). The query together with the intermediate result is sent to another node responsible for the optional pattern, where the optional pattern will be matched with the local triples and the result R_{opt} is outer joined with R_{tmp}. The intermediate result of the optional pattern R_{opt} must be outer joined, as the bindings may be empty from failed matches in the optional result. All simple legal optional expressions may be transformed in this way, though a more sophisticated algorithm is required to express nested optional graph patterns.

```
SELECT ?name ?homepage
WHERE { ?person :name ?name .
OPTIONAL { ?person :homepage ?homepage . }
}
```

$$R_{tmp} = \pi_{sub}\sigma_{pre=name}(T_1) \quad (4.1)$$

$$R_{opt} = \pi_{sub,obj}\sigma_{pre=\text{homepage}}(T_2) \quad (4.2)$$

$$R = \pi_{col1,col2}(R_{tmp} \overset{left}{\infty} R_{opt}) \quad (4.3)$$

4.3.2.6 Disjunctive patterns

SPARQL provides a means of combining graph patterns so that one of several alternative graph patterns may match. If more than one of the alternatives matches, all the possible pattern solutions are found. Pattern alternatives are syntactically specified with the UNION keyword. Obviously, this kind of disjunctive query could simply be resolved by evaluating each sub-query and then computing the union of the results. For example, for the query listed below, the two sub-queries are sent to different responsible nodes, which then calculate and return the intermediate results to the querying node, where the final result is merged.

```
SELECT ?name ?mbox
WHERE {
  ?person :name ?name .
  {
     ?person :mbox ?mbox } UNION { ?person :mbox_sha1sum ?mbox }
   }
}
```

4.3.2.7 Query optimization and relaxation

Query optimization should be performed in the query evaluation process to improve the performance. For example, according to the existing research [56], we can rely on algebraic equivalences (e.g., distribution of joins and unions) to order the evaluation sequence. We may want to separate the *unions* early to parallelize the execution of the union in several peers. Additionally, *selects* and *projects* should be pushed down to the lowest possible places, while *joins* should be evaluated closer to the intermediate peers to reduce the size of the result set as early as possible. Furthermore, statistics about the communication cost between peers and the size of expected intermediary query results can be used to decide which peer and in what order will undertake the execution of each query operator. There has been extensive work in query optimization [48, 53, 56]; we can utilize their results in our system.

The matching manager has the task of finding candidate instances that match the specific query constraints, in particular to take into account the concepts, attributes and relationships. It is possible that the descriptions of different ontologies referring to the same real-world object can be significantly different. As a consequence, real-world objects that are meant to be an answer to a query are not returned because their description does not match the query due to insufficient mappings. If a query cannot get enough results because of this high heterogeneity, the matching manager can relax the query constraints by partially matching the query. At the same time, nodes can extend their mappings based on user feedback on the partial results of matching.

4.3.3 Query processing based on T-Box indexing

In our previous description of query evaluation, we assume the overlay maintains A-Box indexing. In that scenario, instance triple patterns are indexed in the network, and queries for instances can be

accurately forwarded to the right peers in charge of the triples. If an application only maintains T-Box indices, the evaluation process is different.

For schema (T-Box) queries on T-Box indexing, the evaluation process is similar to the query evaluation process we just explained, because T-Box indexing is detailed enough to answer the schema query. For example, consider the query pattern:

```
SELECT ?class
WHERE {
?class rdfs:subClassOf ?someClass
:teach rdfs:domain ?someClass
}
```

It asks for the subclasses of a class which forms the domain of a property *teach*. The processing of this query is the same as the evaluation of conjunctive queries as discussed in Section 4.3.2.3. The query will first be hashed on property *teach* to find its domain class, which will then be used to resolve the next sub-query.

T-Box indexing cannot be used directly to evaluate queries at the instance level, but it can restrict the query to a small set of nodes which are ontologically related to the query. These nodes have the T-Box knowledge to understand the query, thus are capable of answering the query. When a node issues an instance-level query, the T-Box concepts related to the query are extracted in the form of a keyword list, and these keywords are used as parameters to retrieve the relevant peers. We use an example to explain this process. The query is shown below:

```
SELECT ?author
WHERE  {
    ?author rdf:type :Person .
    ?author :name "Juan Li" .
}
```

First, the query processor uses the concepts *Person* and *name* as keys to locate all nodes related to these concepts. Then the query is sent to these nodes for further evaluation. This way, the search scope is limited to a number of nodes whose schemas are related to the query, although not all of them can answer the query.

4.4 Prototype implementation

We evaluate the operability of the presented architecture by implementing a prototype system. In this section we discuss the experiences gained and the lessons learned while developing and implementing the prototype, GONID toolkit, which realizes the ontology directory service proposed in Chapter 3, and the resource integration and discovery framework, GONID, presented in this chapter.

4.4.1 System architecture

The architecture of the GONID toolkit is divided into three layers, namely the communication layer, the semantic layer, and the GUI layer, as shown in Figure 4.11. The communication layer is dedicated to managing the underlying P2P overlay communication, specifically, the directory overlay and the VO overlays. We use FreePastry [127], an open source implementation of Pastry, as our underlying P2P infrastructure. The directory overlay and the VO overlay are implemented in the same Pastry network. The semantic layer manages local ontology knowledge. Its functions include knowledge storing, reasoning, mapping, querying, and indexing. The third layer of the system provides a user-friendly graphic interface, through which users can browse the existing ontology directories in the network, join interested VOs, create and edit ontology metadata, map ontologies, and issue complex queries. The implementation of the semantic layer and the GUI layer is built on top of Protégé [85] – a free, open source ontology editor and knowledge-base framework. Protégé is written in Java, and provides a plug-and-play environment that makes it a flexible base for rapid prototyping and application development. We implement our functionalities as Protégé plug-ins and all components of the system are integrated in the Protégé framework. We choose Java as our development language because both FreePastry and Protégé are written in Java. Next we will describe the implementation of the main components of the prototype system.

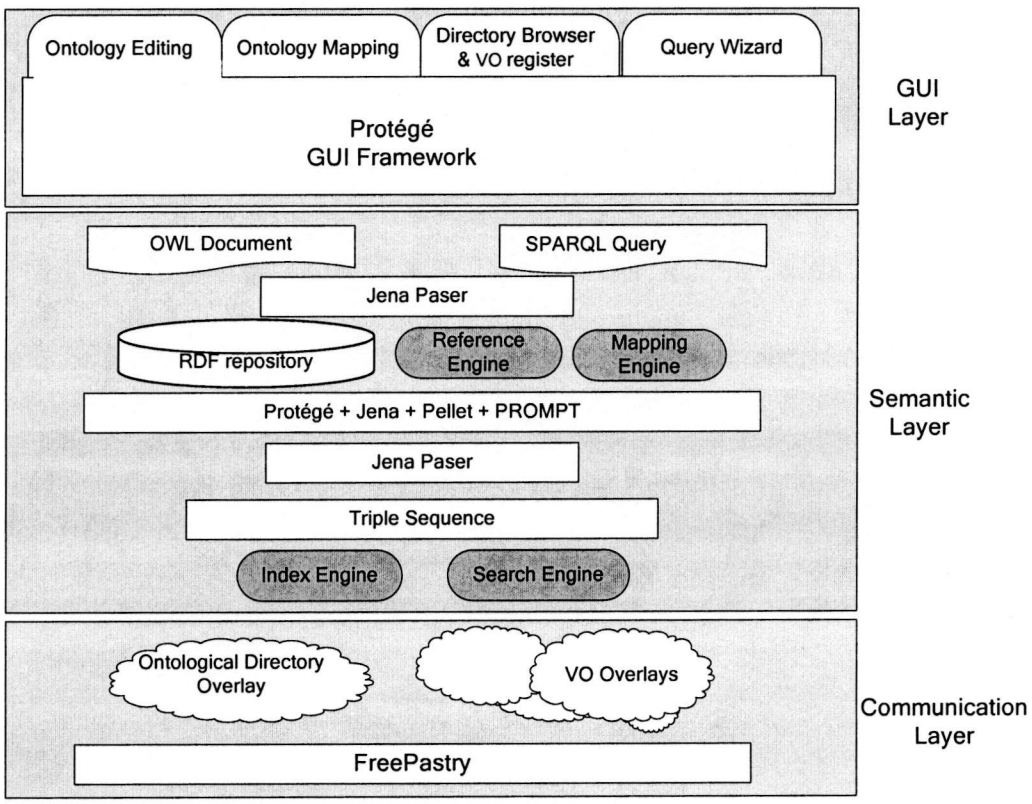


Figure 4.11 System architecture of the GONID toolkit

4.4.2 Main components

4.4.2.1 Pastry overlay

We have layered our implementation of the ontology directory overlay and VO overlay on top of FreePastry. Our implementation is structured as a transparent layer on top of PAST – FreePastry's DHT part. It is customized to efficiently support the insert/modify/delete/query DHT interface for our application, and requires no modifications to the underlying Pastry. An object, either a directory or an RDF triple, has to be wrapped into a *GonidContent* class before being stored in the network. *GonidContent* extends PAST *ContentHashPastContent*, so that PAST can index and retrieve the object. To insert or lookup an object, the hash ID (key) of the object should be provided. A hashed ID (key) of an object can be obtained by calling *PastryIdFactory.buildId()*. Then the object can be stored in or retrieved from the overlay by calling PAST *insert ()* or *lookup()*.

4.4.2.2 User interface

The system's graphical user interface builds on top of the available plug-ins of the Protégé user interface and provides additional graphical components for managing the distributed resource metadata. The GONID toolkit extends Protégé by adding two plug-in tabs: (1) the *ontology directory browser & VO register tab* and (2) the *VO ontological query tab*. Figure 4.12 shows a screenshot of the current state of the *directory browser &VO register* plugin. The main functionality of the ontology browser is to let the user browse the existing ontology hierarchy graphically. There are three panels from left to the right in the tab: the domain ontology browsing panel, class browsing panel, and node browsing panel. In the domain ontology browsing panel, when a user selects a domain/category, sub-domains within the selected domain are listed in an alphabetical order. The class browser shows all the classes and class hierarchies defined in a particular ontology domain. When clicking a class, its detailed definition is listed in a pop-up window. Corresponding to each class, the node browser lists nodes using that class. A user can choose one or more categories and join their corresponding virtual organizations. The *query tab* shown in Figure 4.13 provides a query interface to support sharing and discovering knowledge in VOs. In the query panel on the left, users can enter queries in the SPARQL syntax. After a user presses the *Execute Query* button, the query results will be shown on the right panel. Double-clicking on a result entry will navigate to the particular individual in the *Individuals* tab. A more user-friendly query interface (e.g., a query wizard) is part of our ongoing work.

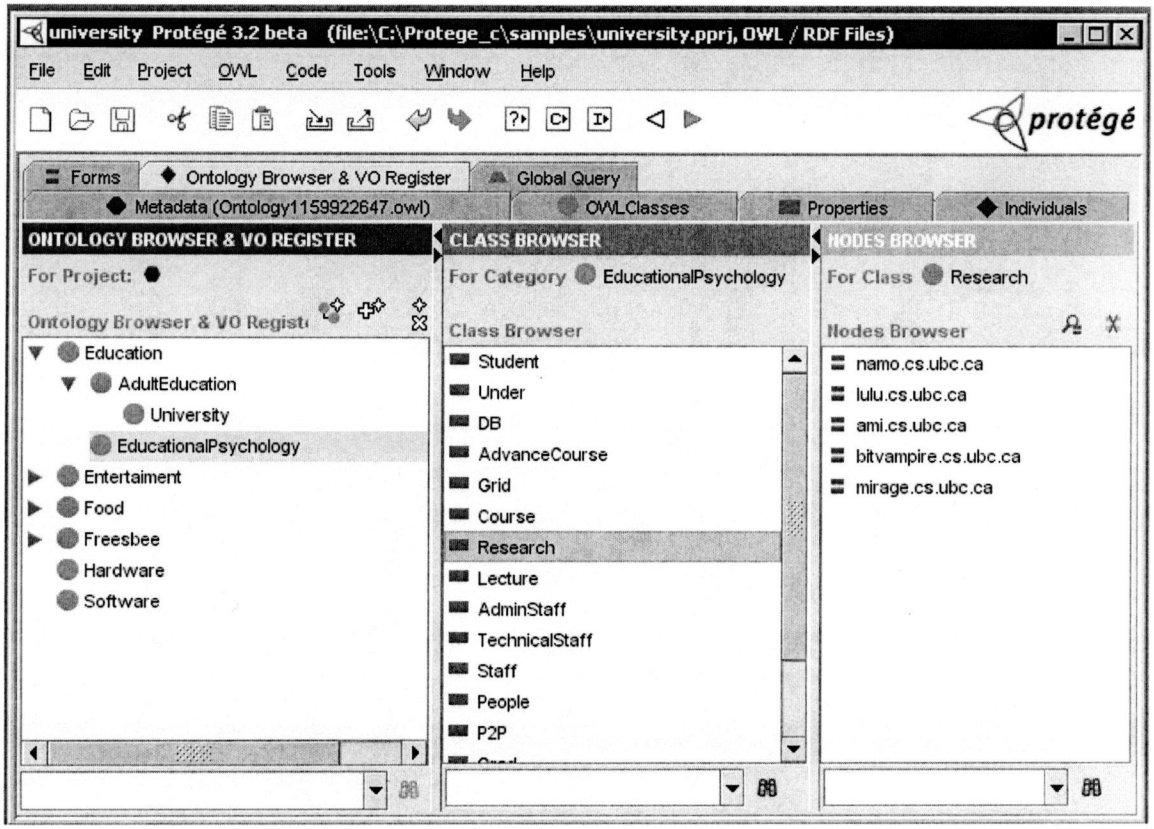


Figure 4.12 A screenshot of the directory browser & VO register tab

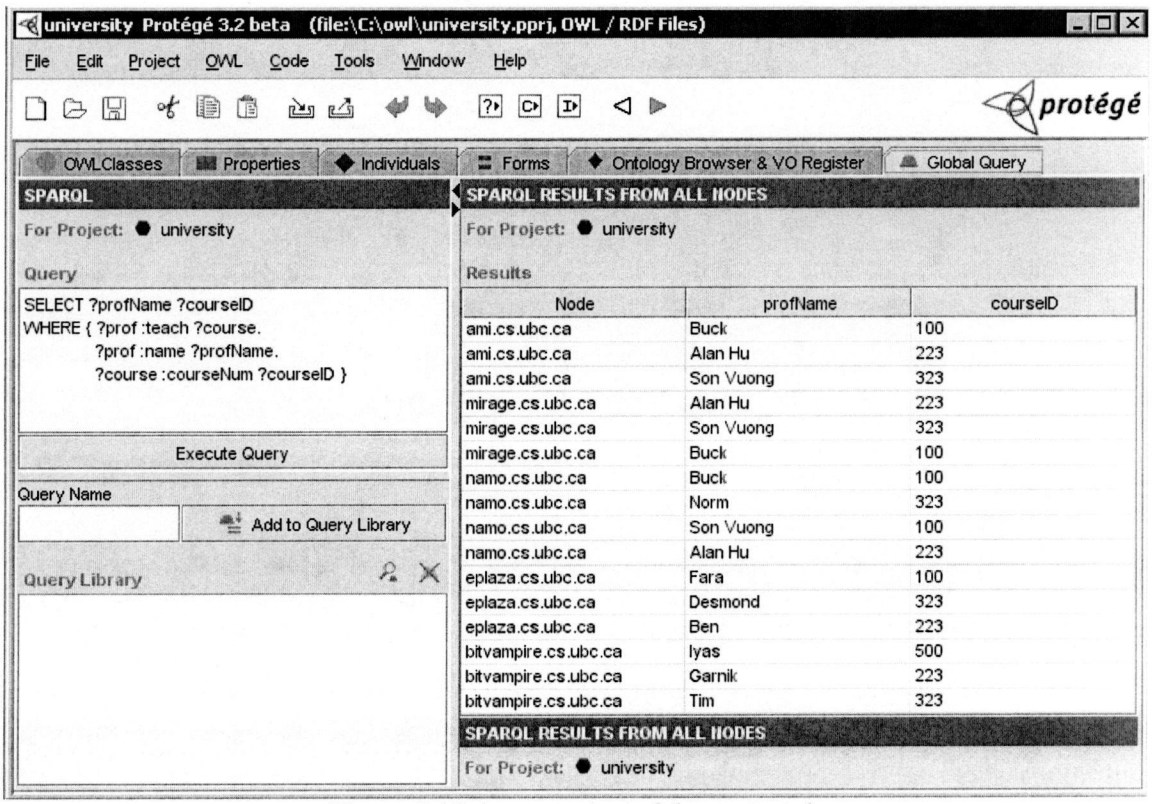


Figure 4.13 A screenshot of the query tab

4.4.3 Ontology management

Processing local ontology knowledge at the private workspace is carried out as users are working in a stand-alone fashion in Protégé. For example, users can use the class editor, property editor, or individual editor to edit their local ontologies. They can use reasoners like Racer [41] or Pellet [99] to compute the inferred ontology class hierarchy and check consistency. They may use PROMPT to do mappings with other ontologies. All these local operations can be carried out with the existing Protégé plug-ins. The knowledge can be saved in all kinds of formats, and the user can use it locally.

More importantly, in our system, users can publish their metadata knowledge so that others in the system can share it. When a user publishes his/her ontology to the VO, the ontology file is saved into a particular directory of the local repository. Then the ontology information is passed to the index engine to index in the network. The current implementation supports ontologies in OWL/RDF format.

4.4.3.1 Parser and index engine

The function of the indexing engine is to index the local ontology knowledge to the Pastry overlay network to be accessed by remote peers. Before indexing, the local ontology file in the OWL/RDF format is first passed through an OWL parser which parses the OWL file into triples by using Jena [75, 131]. The following code snippet illustrates the basic idea of the parsing process.

```
// create a Model
Model model = ModelFactory.createDefaultModel();

// list the statements in the Model
StmtIterator iter = model.listStatements();

// get the subject, predicate, and object of each statement
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();  // get next statement
    Resource  subject   = stmt.getSubject();     // get the subject
    Property  predicate = stmt.getPredicate();   // get the predicate
    RDFNode   object    = stmt.getObject();      // get the object
}
```

The output of the parser – triples are then passed as the input to the index engine. The triple has to be wrapped in the *GonidContent* class, and then it will be indexed three times by hashing the *subject, predicate,* and *object* respectively as the key.

4.4.3.2 Distributed semantic search engine

Through the query tab, a user can submit SPARQL queries to the system. A query is matched with local ontology by a local query processor, which is implemented by Jena's ARQ engine. A simple query matching example is illustrated with following code:

```
import com.hp.hpl.jena.query.* ;

  Model model = ... ;
  String queryString = " .... " ;
  Query query = QueryFactory.create(queryString) ;
  QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
  try {
    ResultSet results = qexec.execSelect() ;
    for ( ; results.hasNext() ; )
    {
      QuerySolution soln = results.nextSolution() ;
      RDFNode x = soln.get("varName") ;       //Get a result variable
      Resource r = soln.getResource("VarR") ; //Get a result variable
      Literal l = soln.getLiteral("VarL") ;   //Get a result variable
    }
  } finally { qexec.close() ; }
```

At the same time, the query will be parsed to triples by Jena's ARQ parser, and then the triples will be sent to the index engine to match on the Pastry overlay. After finding matches from the Pastry overlay, the original SPARQL query will be forwarded to the matching nodes to do the local matching. Results are returned back directly to the requester. Currently, we do not implement the ranking of the results.

4.4.4 GONID toolkit deployment and evaluation

We installed the GONID Toolkit software on six WinXP computers and six Linux SUSE 10 computers at the computer science department of the University of British Columbia. Each physical node runs three copies of the software and simulates three virtual nodes, therefore we have in total thirty-six nodes in the system.

We create an experimental scenario to show that GONID does improve the performance of searching. We use two Information Retrieval (IR) standards: *precision* and *recall* as the performance metrics. *Precision* is defined in Equation 4.4. It measures the purity of the search results, or how well a search avoids returning results that are not relevant. The "document" in the IR definition represents a resource in our experiment. *Recall* refers to completeness of retrieval of relevant items, as defined in Equation 4.5.

$$precision = \frac{|relevantDocuments \cap retrievedDocuments|}{|retrievedDocuments|} \quad (4.4)$$

$$recall = \frac{|relevantDocuments \cap retrievedDocuments|}{|relevantDocuments|} \quad (4.5)$$

Our experiments try to justify two hypotheses: (1) grouping semantically related nodes into VOs and using domain ontologies helps to eliminate the semantic ambiguity, thus improving the search precision; (2) mappings between parties with different ontologies and reasoning help to extend a concept's semantic meaning, thus improving the search recall.

To make the experiment easy to control, we simplify the ontology data: we use a small-sized vocabulary set to generate the ontology data; we fix the mapping relation to the *equivalentClass* relation and ignore all other mapping relations. Specifically, the data is generated as follows:

- We generate a dictionary D containing N words. It provides all the vocabulary for the system's ontologies.
- In dictionary D, we create a set of semantically equivalent categories, C. In each category, we have c keywords which are assumed to represent the same semantic meaning, i.e., any two keywords refer to the same meaning. In addition, we randomly pick s words, representing polysemy or homonymy (words with multiple meanings); if these words appear in different VOs, they represent different meanings.
- We created V ($V=3$) VOs. Each VO has O ($O=10$) ontologies, and each ontology includes k ($k=5$) classes. The ontologies are created with the following process:

```
for class number i=1 to k
    for ontology num j=1 to O
        ontology onto_j set its class class_i as a keyword randomly picked from a category C_i
    end for
end for
```

  This procedure creates O semantically related ontologies which can be mapped mutually, because each class in an ontology can find mappings from other ontologies. In addition, each class has i ($i=6$) instances.

- Each node joins to *1-3* VOs and keeps one ontology for each VO; it also maintains m ($1 \le m \le 3$) mapping neighbors, i.e., neighbors that map corresponding equivalent classes. The equivalent mapping is published so that the query engine can use the mapping to reformulate the query.

The following simple query form has been used in our experiment.

```
SELECT ?x
WHERE ?x rdfs:type ClassX;
```

We compare the GONID ontology-based search with semantics-free exact-match-based search. For the exact-match-based search, a query only matches a keyword without caring about the keyword's specific meaning in a VO and the mappings between nodes. For GONID searching, we vary the number of mapping neighbors (m) a node maintains from 1 to 3. For all the results returned, we compute the *precision* and *recall*. The results are shown in Table 4.3.

Table 4.3 Performance comparison of GONID search and exact-match search

	Exact-Match	**GONID (m=1)**	**GONID (m=2)**	**GONID (m=3)**
Precision	57%	100%	100%	100%
Recall	33%	64%	89%	100%

As shown in Table 4.3, GONID dramatically outperforms exact-match in both precision and recall. Because GONID search is executed in semantic VOs, it eliminates the semantic ambiguity problem such as polysemy and homonymy. Therefore, all the results returned by GONID search are relevant and the precision is 100%. When the number of mappings increases, the recall also increases, as most of the

relevant relations can be identified. In fact, when each node maintains about 3 mapping neighbors, GONID search achieves a 100% recall rate. The result shows that the proposed GONID strategy is effective in improving the quality of search. Because of the limited experimental environment: with only 12 physical nodes in a LAN, we leave the scalability evaluation to simulations presented in the next section.

4.5 Simulation experiments

We have demonstrated GONID's improved searchability in terms of its expressive query language, semantic reasoning, and integration, by theoretical analysis, examples, and a prototype implementation. Owing to the lack of access to the semantic environment with many nodes, our system performance evaluation falls back to simulations. In this experiment part, we focus on evaluating the performance that can be quantitatively measured by simulation. We first describe the experimental setup, and then analyze the simulation results.

4.5.1 Experimental setup

As it is difficult to find representative real world ontology data, we have chosen to generate test data artificially. Our data does not claim to model real data, but shall rather provide reasonable approximation to evaluate the performance of the system. Ontology data can be characterized by many factors such as the number of classes, properties, and individuals; thus we have generated the test data in multiple steps. The algorithm starts with generating the ontology schema (T-Box). Each schema includes the definition of a number of classes and properties. The classes and properties may form a multilevel hierarchy. Then the classes are instantiated by creating a number of individuals of the classes. To generate an RDF instance triple t, we first randomly choose an instance of a class C among the classes to be the subject: *sub(t)*. A property p of C is chosen as the predicate *pre(t)*, and a value from the range of p to be the object: *obj(t)*. If the range of the selected property p are instances of a class C', then *obj(t)* is a resource; otherwise, it is a literal.

The queries are generated by randomly replacing parts of the created triples with variables. For our experiments, we use single-triple-queries and conjunctive-triple-queries. To create the conjunctive-queries, we randomly choose a property p_1 of class C_1. Property p_1 leads us to a class C_2 which is the range of p_1. Then we randomly choose a property p_2 of class C_2. This procedure is repeated until the range or the property is a literal value or we have created n ($n \leq 3$) triple patterns.

Our dataset uses the following parameters: The total number of distinguished ontologies is 100. We assume each node uses 1 to 3 ontologies. Each ontology includes at most 10 classes. The number of properties that each class has is at most k=3. The number of instances of each class at each peer is less than 10. Finally, the number of triple patterns in each query we create is either 1 or 3. In our experiment,

we do not do knowledge reasoning. In other words, we do not augment the RDF graph by inference (forward chaining).

We implement a simulator of Pastry in Java on top of which we developed our indexing and routing algorithms. Each peer is assigned a 160-bit identifier, representing 80 digits (each digit uses 2 bits) with base b=2. After the network topology has been established, nodes publish their data on the overlay network. Then nodes are randomly picked to issue queries. Each experiment is run ten times with different random seeds, and the results are the average of these ten sets of results.

4.5.2 Experimental results

The system's ability to integrate heterogeneous ontologies, infer new knowledge, and answer all kinds of complex queries are illustrated with examples and the prototype toolkit. Here, we are mainly interested in three different questions, related to three different aspects of the indexing and searching scheme. First, we want to verify the efficiency of answering typical lookup requests. Second, we need to compare the overhead of indexing T-box and A-box as well as the overhead of searching based on these two indexing schemes. Last, we try to examine one important factor that affects the decision of choosing the index granularity. Next we list the major simulation results and provide a brief analysis.

The first experiment answers the first question showing the number of routing hops as a function of the size of the Pastry network. We vary the number of Pastry nodes in the network from 2^9 to 2^{14}. We run two trials of experiments: one trial issues only single-triple-queries, while the other trial issues conjunctive-triple-queries. Figure 4.14 shows the average number of routing hops taken as a function of the network size for both query patterns. $\log_{2^b}N$ is the expected maximum number of hops required to route a key in a network containing N nodes (In our experiment b=2), therefore, in the figure "$\log_4N$" is included for comparison. The results show that the number of route hops scales with the size of the network as predicted: for the single triple query, the route length is below $\log_4N$; for conjunctive queries, the routing hops is below $3\log_4N$ as expected.

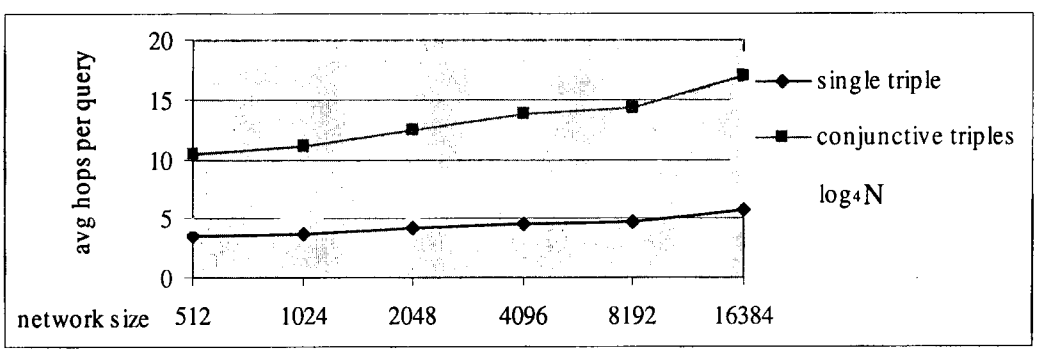


Figure 4.14 Query lookup efficiency

The next experiment compares the performance of the T-Box and the A-Box indexing in terms of indexing overhead and query overhead. Each node may randomly choose n (n<3) ontologies from 100 distinguished ontologies, and instantiate each class with m (m<10) instances. For simplicity, each query uses the simple single triple pattern. With this configuration, we see from Table 4.4 that A-Box indexing incurs much more overhead than T-Box indexing, and the discrepancy increases as the network size increases, for example, A-Box indexing causes several orders of magnitude higher overhead than what TBox indexing creates when the network size is 4096. On the other hand, if the system can afford the cost of maintaining the large index, A-Box indexing can improve searching efficiency. Table 4.5 shows the query overhead in terms of cumulative query messages. It is obvious that with A-Box indexing, processing a query requires much less message forwarding overhead than that based only on T-Box indexing.

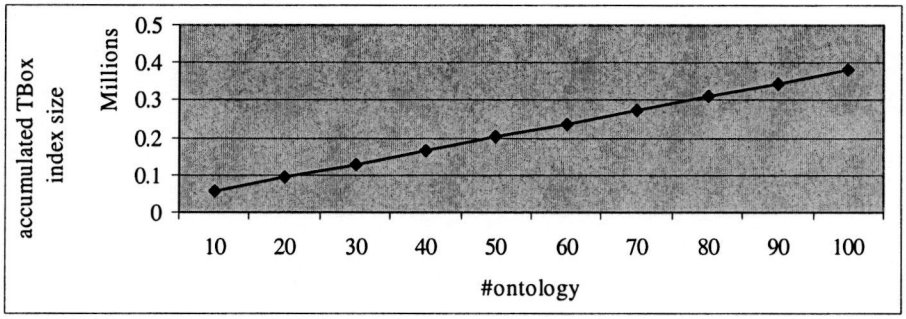
Table 4.4: Cumulative indexing storage load of T-Box indexing and A-Box indexing

Network Size	256	512	1024	2048	4096
Cumulative A-Box index (bytes)	10472400	20732040	39491280	81690660	1.63E+08
Cumulative T-Box index (bytes)	365497	370939	381086	403625	446060

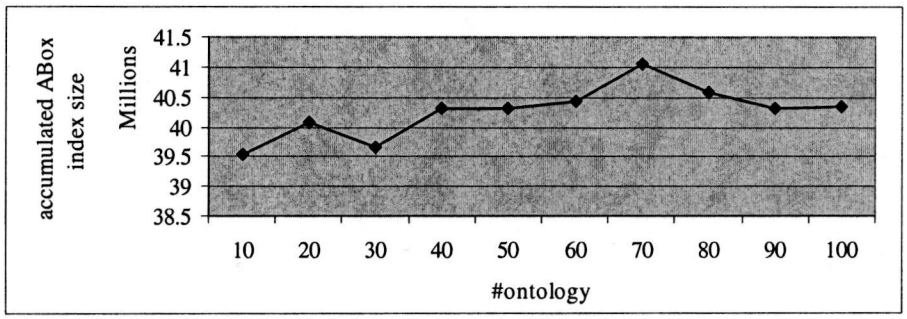
Table 4.5: Cumulative query overhead based on T-Box index and A-Box index

Network Size	256	512	1024	2048	4096
Cumulative Query messages on A-Box index	17880	22080	21840	24780	26880
Cumulative Query messages on T-Box index	66120	105360	170880	302940	573960

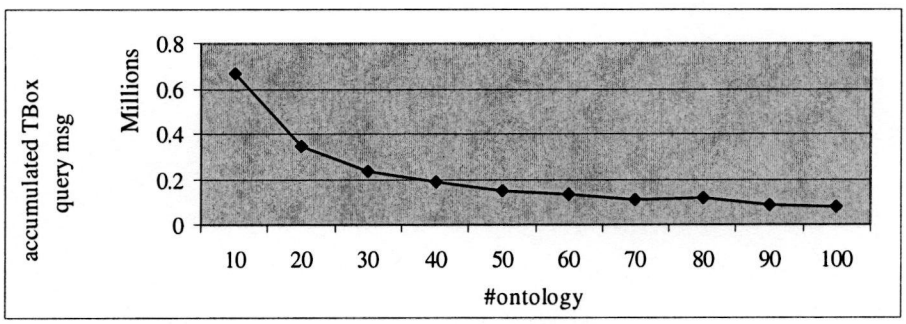
We have seen the differences between T-Box indexing and A-Box indexing. An important question is how to choose the right indexing scheme for a system. There are many factors to consider, for example, the storage capacity of the participating nodes, the nature of the major queries, and even the organizations' policy. Another important factor is the degree of heterogeneity of the system's ontology. We performed a set of experiments to examine how the ontology variety affects the indexing performance. In the experiment, we fixed the network size at 1024. Initially, there are 10 distinguished schemas for the participating nodes to choose from, then we increase the number of ontology choices. We then examine the indexing and query overhead of both T-Box indexing and A-Box indexing. The results are illustrated in Figure 4.15. We notice that the ontology variety does not significantly influence the indexing and query overhead based on A-Box indexing, but does have an impact on T-Box indexing. For a fixed sized network, the more heterogeneous the ontology, the more effective the T-Box indexing becomes. As shown in Figure 4.15 (c), when the system has 100 ontologies to choose from, queries based on T-Box indexing cause just a small message load comparable to A-Box indexing in Figure 4.15(d). This is easy to understand: when nodes have homogeneous ontologies, most nodes have the same T-Box knowledge; then indexing T-Box cannot effectively distinguish nodes and guide query forwarding. When the system has highly heterogeneous ontologies, T-Box indexing can distinguish nodes' ontologies well; therefore query routing is more efficient.



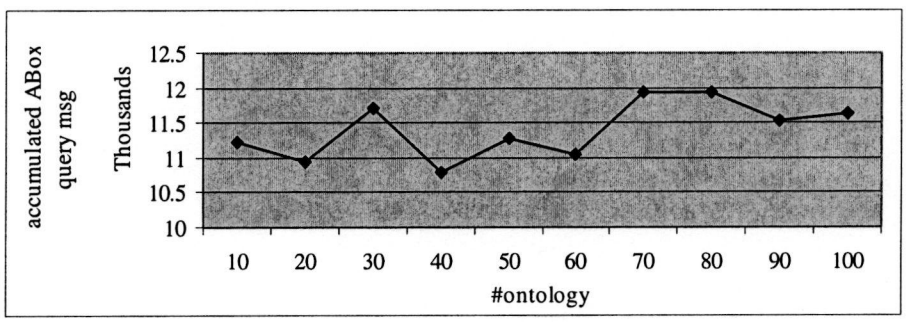
(a) T-Box indexing overhead with increasing ontological heterogeneity



(b) A-Box indexing overhead with increasing ontological heterogeneity



(c) T-Box query overhead with increasing ontological heterogeneity



(d) A-Box query overhead with increasing ontological heterogeneity

Figure 4.15 Effect of ontological heterogeneity on the indexing scheme

4.6 Summary

In this chapter, we have presented GONID, an ontological framework for resource integration and discovery in semantic virtual organizations. One focus of the GONID system is to overcome the shortcomings of keyword-based searching, such as limited searchability and low recall and precision. To achieve this goal, we have proposed an ontological model that employs ontological domain knowledge to assist in the search process. The model provides efficient solutions to resource representation, integration, reasoning, indexing, and complex query evaluation. Queries are processed in a distributed and transparent fashion, so that the fact that the information is distributed across different sources and represented with different formats can be hidden from the users. The resulting prototype system, GONID toolkit, verifies the viability of this indexing and searching infrastructure. Simulation experiments have demonstrated its performance.

Chapter 5
OntoSum -
An Alternative Discovery Scheme

Chapter 4 describes GONID, a VO discovery scheme based on a structured DHT overlay. The ontology-based DHT indexing used by GONID allows scalable and efficient lookup, but it also has some shortcomings: (1) it is sensitive to churn because of the large overhead incurred in recovering the neighbor-relationship; (2) there is no control over where the metadata index is stored; (3) it has limited support for richer queries, such as wildcard queries, fuzzy queries, and proximity queries; and (4) since each node's ontology is decomposed and dispersed to the network, nodes may not have a complete view of others' ontologies. This limitation makes it difficult for nodes to find semantically related nodes in order to do ontology mappings. Therefore, if a system is highly dynamic, or needs to provide participating nodes full control of the storage of their resources (or the resource index), or has to support arbitrary complex queries, or the initial ontology mapping candidates cannot be discovered by using only dispersed linguistic features, then the DHT structure may not be an ideal choice.

In this chapter, we propose a new search architecture, OntoSum, that attempts to overcome the above-mentioned problems by the use of "semantic small-worlds". OntoSum is based on the observation that query transferring in social networks is made possible by locally available knowledge about acquaintances. Because of the similarity between grid networks and social networks and the fact that human users of grid networks direct grid nodes' links, we argue that grid networks can also utilize this phenomenon to discover resources. Peers in OntoSum use their ontology summary to represent their expertise; they learn and store knowledge about other peers with a view to their potential for answering prospective queries. This way, the network topology is reconfigured with respect to peers' semantic properties, and peers with similar ontologies are close to each other. Resources can then be located through nodes' current neighbors, rather than by contacting some central hubs (or virtual central hubs, such as DHTs). Like the GONID system, OntoSum addresses the routing issues of expressive queries in an ontologically heterogeneous environment. But unlike GONID's dependence on a well-organized DHT overlay, OntoSum adopts an unstructured overlay; it does not need to maintain a strict network topology, thus it is resilient to churn. Additionally, nodes have full control over where their resources (or the index of the resources) are stored. Moreover, in theory, it places no constraints on the query format; it can deal with any complex and flexible queries. Lastly, since each node indexes its metadata locally, it is possible to summarize a node's semantic interest or property from its indices; this facilitates a node finding its semantically related peers and performing ontology mapping with them.

The contributions of this chapter are as follows:

1. We propose a small-world model based on semantic similarity to facilitate resource discovery and sharing.
2. We propose a novel structure, the Ontology Signature Set (OSS), as a concise summary of nodes' ontology schema. Based on OSSs, we design a method to compute the semantic distance (similarity) between different nodes.
3. We propose a topology adaptation algorithm to form semantic small-worlds according to nodes' semantic similarity.
4. We design an efficient semantics-based routing algorithm, RDV, which can further improve the performance of searching inside a cluster of the small-world system.
5. We perform extensive simulation to evaluate system performance.

Portions of this chapter appeared first in Li *et al.* [62, 65, 66].

The remainder of this chapter is organized as follows: Section 5.1 introduces the concept of semantic small-worlds and their advantageous properties. Section 5.2 presents a novel method of computing the semantic similarity between different peers. Section 5.3 gives an overview of our semantic small-world architecture – OntoSum. Section 5.4 describes the algorithm of organizing nodes to construct the OntoSum architecture according to the semantic similarity between nodes. Section 5.5 explains how resource discovery is performed in OntoSum. Section 5.6 presents a comprehensive semantics-based query routing algorithm, RDV, which works as an improved routing algorithm to forward queries inside OntoSum clusters. Simulation experimental results are given in Section 5.7.

5.1 The concept of a semantic small-world

A widely-held belief pertaining to social networks is that any two people in the world are connected via a chain of six acquaintances (*six-degrees of separation*) [7]. The quantitative study of the phenomenon started with Milgram's experiments [77] in 1960's, in which people were asked to send letters to unfamiliar targets only through acquaintances. Milgram's experiments illustrated that individuals with only local knowledge of the network (i.e., their immediate acquaintances) may successfully construct acquaintance chains of short length, leading to networks with "small-world" characteristics. In such a network, a query is forwarded along outgoing links taking it closer to the destination. Randomized network constructions that model the small-world phenomenon have recently received considerable attention. To model the routing aspects of the small-world phenomenon, Kleinberg constructed a family of random graphs [55]. The graphs not only have small diameter (to model the "six degrees of separation") but also allow short routes to be discovered on the basis of local information alone (to model Milgram's observation that messages can be "routed to unknown individuals efficiently"). In particular, Kleinberg considered a 2D $n \times n$ grid with n^2 nodes. Each node is equipped with a small set of "local" contacts and one "long-range" contact drawn from a harmonic distribution. With greedy routing, the path-length between any pair of nodes is $O(log^2 n)$ hops, with high probability.

Small-world networks exhibit special properties, namely, a small average diameter and a high degree of clustering. A small diameter corresponds to a small separation between peers, while a high clustering signals tight communities. Small world graphs contain inherent community structure, where similar nodes are grouped together in some meaningful way. Intuitively, a network satisfying the small-world properties would allow peers to reach each other via short paths while maximizing the efficiency of communication within the clustered communities.

We draw inspiration from small-world networks and organize nodes in our system to form a small-world topology, particularly from a semantic perspective. Our objective is to make the system's dynamic topology match the semantic clustering of peers, i.e., there is a high degree of semantic similarity between peers within the clustered community; this would allow queries to quickly propagate among relevant peers as soon as one of them is reached. To construct the semantic small world network depicted above, we follow the idea of the Kleinberg experiment: each node keeps many close neighbors (short-range contacts), as well as a small number of distant neighbors (long-range contacts). The distance metric in our system is determined by nodes' semantic similarity. With the semantics-based small-world constructed, a query can be efficiently resolved in the semantic cluster neighborhood through short semantic paths.

The preliminary semantic building blocks of OntoSum are similar to those defined in the GONID system presented in Chapter 4. We apply ontology to resource descriptions, and adopt RDF/OWL as the ontology language. Inference engines are used to derive additional facts from existing knowledge. Four class mapping patterns: *equivalentClass, subClass, superClass, referentialClass,* and four property mapping patterns: *equivalentProperty, subProperty, superProperty, inverseProperty* are defined to represent mappings between ontologies. We distinguish T-Box knowledge and A-Box knowledge in each peer's local ontology repository as well. Since all these semantic components have been explained in Chapter 4, we do not repeat the description here. Rather, we will present how to determine the similarity between ontologies, so that nodes can reconfigure the topology accordingly.

5.2 Semantic similarity

Computing the semantic similarity between two peers is very difficult and related research is still in its initial stage. It may need technologies from natural language processing, information integration, graph matching, etc. It involves measuring the similarity of the syntactical, structural, and semantic aspects of the ontology data. There has been extensive research [49, 60, 89] focusing on measuring the semantic similarity between two objects in the field of information retrieval and information integration. However their methods are very comprehensive and computationally intensive. In this thesis work, we propose a simple method to compute the semantic similarity between two peers; this can easily be replaced with other advanced functions for a complex system.

5.2.1 Ontology signature set (OSS)

To measure the semantic similarity between peers, we need to extract each peer's semantic characteristics. The representation of these characteristics should be light-weight, so that they can be efficiently exchanged between peers and the similarity based on these characteristics can be easily computed. As elaborated in Section 4.2.2, the T-Box part of an ontology defines high-level concepts and their relationships like the schema of a database. It is a good abstraction of the ontology's semantics and structure. Therefore, our semantic property representation is based on T-Box knowledge.

5.2.1.1 Primitive ontology signature set

A naïve approach is to use keywords of a node's T-Box ontology as its ontology summary. For each node, we extract the class and property labels from its T-Box ontology, and put them into a set. This set is called this node's Ontology Signature Set (OSS). We can measure the similarity of two ontologies by comparing the elements of their OSSs. To reduce the size of an OSS, it can be compressed to a compact structure: a Bloom filter [9]. With the OSS, we summarize a node's ontology properties as a set of keywords. This summarization is simple and concise, but on the other hand, it is not precise; it ignores the inherent relationships between T-Box concepts and thus damages the semantic meaning of each concept. Next, we present two methods to improve the semantic precision of the OSS.

5.2.1.2 Extended ontology signature set

A node's primitive OSS contains only local ontology concept labels. However, a semantic meaning may be represented by different labels in different ontologies, while it is also possible that the same literal label in different ontologies means totally different things. Therefore, two semantically equivalent ontologies may have totally different OSSs, while two similar OSSs may represent two completely different ontologies. Ontology comparison based on primitive OSSs may not yield satisfying results. One improvement is to extend each concept with its semantic meanings, so that semantically related concepts would have overlaps. Based on this intuition, we use the lexical database, WorldNet [78], to extend the OSS to include words which are semantically related to the concepts from the original set. In our work, WordNet is interpreted and used as a lexical ontology which extends the semantic meaning of the class and property labels in question.

WordNet is the product of a research project at Princeton University. It was conceived as a machine-readable dictionary, following psycholinguistic principles [28, 78]. Unlike standard alphabetical dictionaries which organize vocabularies using morphological similarities, WorldNet structures lexical information in terms of word meanings. WordNet maps word forms in word senses using the syntactic category as a parameter. Words of the same syntactic category that can be used to express the same meaning are grouped into a single synonym set, called *synset*. For example, the noun "computer" has a synset: {computer, data processor, electronic computer, information processing system}. An intuitive idea of extending an OSS is to extend each concept with its synset, i.e., its synonyms. Given a primitive

OSS consisting of a number of ontology concept labels, we lookup each concept in the WordNet lexicon and extend each concept with its synonyms in the synset. In this way, two semantically related ontologies would have common WordNet terms in their extended OSSs. Besides synonyms, WordNet also includes other lexical semantic relations, such as *is-a*, *kind-of*, *part-of*. Among these relations, *is-a* (represented by hyponym/hypernym in WordNet) is the most important relationship; it explains a concept by a more general concept. Therefore, we also extend OSS concepts with their hypernyms.

5.2.1.3 Refined ontology signature set

After extension, an OSS may get a large number of synonyms for each concept. However, not all of these synonyms should be included in the set, because each concept may have many senses (meanings), and not all of them are related to the ontology context. For example, consider the noun "computer", it has two senses defined in WordNet, hence two synsets, {computer, data processor, electronic computer, information processing system} and {calculator, reckoner, figurer, estimator, computer}. Having unrelated senses in the OSS will diminish the accuracy of measuring the ontology difference and incur higher computation cost for set operations. Therefore, we have to prune the expanded OSS to exclude those unrelated terms.

A problem causing the ambiguity of concepts in OSS is that the extension does not make use of any relations in the ontology. Relations between concepts are important clues to infer the semantic meanings of concepts, and they should be considered when creating the OSS. Therefore, we utilize relations between the concepts in an ontology to further refine the semantic meaning of a particular concept. Only words with the most appropriate senses are added to the OSS. Since the dominant semantic relation in an ontology is the subsumption relation (super-class, the converse of is-a, is-subtype-of, or is-subclass-of), in this development phase of our system, we use the subsumption relation and the sense disambiguation information provided by WordNet to refine OSSs. It is based on a principle that a concept's semantic meaning should be consistent with its super-class's meaning. We use this principle to remove those inconsistent meanings. The refined algorithm to generate the OSS is illustrated with the pseudocode in Figure 5.1.

```
/* This algorithm generates the refined Ontology Signature Set
OSS for an ontology O */

createOss(Ontology O)
{
   OSS={};
   for each c ∈{concepts of ontology O}
      p_c is parent concept of c
      add c, p_c to OSS
      for each S_c∈{senses of c}
          H_c={hypernyms of S_c}
          for each Sp_c∈{senses of p_c)
            if H_c ∩ Sp_c !=null
               add S_c,Sp_c to OSS
```

Figure 5.1 A refined algorithm to generate the Ontology Signature Set

The algorithm in Figure 5.1 creates the refined OSS by adding the appropriate sense set of each ontology concept based on the *sub-class/super-class* relationships between the parent concepts and child concepts. For every concept in an ontology, we check each of its senses; if a sense's hypernym has an overlap with this concept's parent's senses, then we add this sense and the overlapped parent's sense to the OSS set. In this way, we can refine the OSS and reduce imprecision.

Possible improvements could be obtained by using other relations in the ontology, such as the *meronymy* relation, written as *part-of*, representing how objects combine together to form composite objects. Besides the *is-a* and *part-of* relations, ontologies often include additional types of domain-specific relationships that further refine the semantics they model. Using complex relations to refine the meaning of the ontology concept is difficult to be performed automatically because too little information is available and exploring all the possibilities will greatly slow down the process. The system may provide an interface to users to select the right meaning from the possible ones found in WordNet; this is beyond our current thesis work.

5.2.2 Peer semantic similarity

To compare two ontologies, we define an ontology similarity function based on the refined OSS. The definition is based on Tversky's "Ratio Model" [111], which is evaluated by set operations and is in agreement with an information-theoretic definition of similarity [71]. Our similarity function is based on the normalization of Tversky's model to give a numeric measurement of ontology similarity.

Definition 5.1: Assume A and B are two peers, and their extended Ontology Signature Sets are S(A) and S(B) respectively. The semantic similarity between peer A and peer B is defined as:

$$sim(A,B)=\frac{|S(A)\cap S(B)|}{|S(A)\cap S(B)|+\alpha|S(A)-S(B)|+\beta|S(B)-S(A)|} \qquad (5.1)$$

In the above equations, "$\cap$" denotes set intersection, "–" is set difference, while "| |" represents set cardinality, "α" and "β" are parameters that provide for differences in focus on the different components. The similarity *sim*, between A and B, is defined in terms of the semantic concepts common to OSS of A and B: $S(A)\cap S(B)$, the concepts that are distinctive to A: $S(A)–S(B)$, and the features that are distinctive to B: $S(B) – S(A)$. The parameters α and β are non-negative, determining the relative weights of these two components. The similarity depends not only on the proportion of features common to the two ontologies but also on their unique features and the relative importance varies with the parameters α and β. These parameters allow the model some flexibility, because it can decide whether common or distinctive features have more influence. Note that with this definition, similarity is not a symmetric relation, i.e., "how similar is A to B" may give a different answer than "how similar is B to A". Employing such an asymmetric measurement reflects human judgment: sometimes, we say one object is similar to another one, but not vice versa. With the similarity measure specified, we have the following definition:

Definition 5.2: Two nodes, node A and node B are said to be semantically equivalent if their semantic similarity measure, sim(A,B) equals to 1 (implying sim(B,A)=1 as well). Node A is said to be semantically related to node B, if sim(A,B) exceeds the user-defined similarity threshold t (0<t≤1). Node A is semantically unrelated to node B if sim(A,B)<t.

5.2.3 An example

We use an example to further illustrate how to use the refined OSS and similarity function to measure the semantic similarity between two peers. Figure 5.2 shows two partial ontology definitions about automobiles. Detailed ontology definitions are omitted here. Table 5.1 and 5.2 list the ontology concepts and their synonyms and hypernyms from all senses extracted from WordNet.

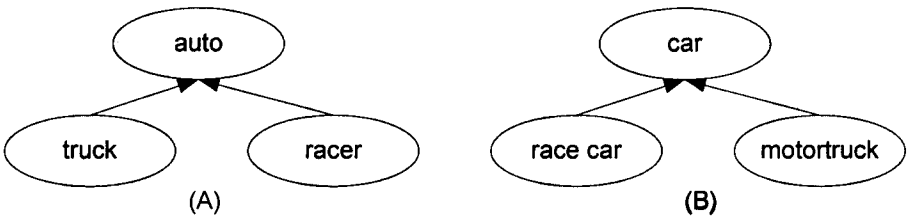


Figure 5.2 Parts of two ontologies

The primitive OSSs of these two ontologies are:

S_A = *{auto, truck, racer}*
S_B = *{car, race car, motortruck}*

These two sets share no common terms, and literally they are totally different. If the similarity function is applied to these two sets, the result is 0, meaning they are totally unrelated. Table 5.1 and Table 5.2 illustrate how to extend the OSSs with right WordNet senses by applying the algorithm in Figure 5.1.

Table 5.1 WordNet senses and hypernyms for ontology A

Concept	Parent-concept	WordNet senses/synset	Hypernyms of senses in WordNet	Right sense?
auto		car, auto, automobile, machine, motorcar	motor vehicle, automotive vehicle	yes
truck	auto	truck, motortruck	motor vehicle, automotive vehicle	yes
		hand truck, truck	handcart, pushcart, cart, go-cart	no
racer	auto	race driver, automobile driver	driver	no
		racer, race car, racing car	car, auto, automobile, machine, motorcar	yes
		racer (an animal that races)	animal, animate being, beast, brute, creature, fauna	no
		racer (slender fast-moving North American snakes)	colubrid snake, colubrid	no

Table 5.2 WordNet senses and hypernyms for ontology B

Concept	Parent-concept	WordNet senses/synset	Hypernym of senses WordNet	Right sense?
car		auto, automobile, machine, motorcar	motor vehicle, automotive vehicle	yes
		railcar, railway car, railroad car	wheeled vehicle	no
		gondola	compartment	no
		cable car, car	compartment	no
race car	car	racer, race car, racing car	car, auto, automobile, machine, motorcar	yes
motortruck	car	truck, motortruck	motor vehicle, automotive vehicle	yes

By extending the two primitive OSSs: S_A *and* S_B, we get the extended OSSs: S_A' and S_B':

S_A' = *{auto, car, automobile, machine, motorcar, truck, motortruck, racer, race car, racing car}*
S_B' = *{car, auto, automobile, machine, motorcar, racer, race car, racing car, truck, motortruck}*

Now we can see that these two sets share exactly the same semantic concepts! The similarity function based on the extended OSSs are:

$$sim(A, B) = \frac{|S(A) \cap S(B)|}{|S(A) \cap S(B)| + \alpha |S(A) - S(B)| + \beta |S(B) - S(A)|} = \frac{11}{11 + 0\alpha + 0\beta} = 1$$

$$sim(B, A) = \frac{|S(A) \cap S(B)|}{|S(A) \cap S(B)| + \alpha |S(B) - S(A)| + \beta |S(A) - S(B)|} = \frac{11}{11 + 0\alpha + 0\beta} = 1$$

This means ontology A and ontology B are semantically equivalent. Note: the equivalent is independent of α and β. With the semantic similarity function defined, we can measure the semantic distance between nodes and reconfigure the network topology accordingly to form semantic small-worlds. The next section gives a brief overview of our semantic small-world topology.

5.3 Small-world topology adaptation

5.3.1 Topology overview

In Kleinberg's experiment [55], to form a network with small-world characteristics nodes keep many "local" contacts and one "remote" contact. Our semantic topology construction is based on this idea. In our system, a node distinguishes three kinds of neighbors based on their semantic similarity. A peer A's neighbor, B, can be one of these three types: (1) zero-distance neighbor (or semantically equivalent neighbor), if $sim(A,B)=1$, (2) short-distance neighbor (or semantically related neighbor) if $sim(A,B) \geq t$ ($0<t<1$ is A's semantic threshold), (3) long-distance neighbor (or semantically unrelated neighbor) if $sim(A,B)<t$. A node always tries to find as many close neighbors as possible, but it also keeps some long distance neighbors to reach out to other ontological clusters.

Nodes in the system randomly connect to each other through these three types of neighbor links. They produce a semantically clustered small-world topology. The cluster structure is not flat but multi-layered; nodes with similar ontological topics (short-distance neighbors) form a domain; inside the domain, nodes may create smaller clusters if they share the same ontology schema. Figure 5.3 shows a high level view of a sample network topology. All peers in the medical domain are interested in information related to medicine. They may be interested in different aspects of the medical resources, and they may use different ontologies to describe their resources. They connect with each other through short-distance links. Inside the medical domain, nodes further organize themselves to finer-grained clusters based on their ontologies. For example, nodes N_1, N_2, N_5, and N_8 use the same ontology, *onto*$_1$ (e.g., a medical ontology, SNOMED-RT [122]), thus they are zero-distance neighbors and form the same-ontology cluster. In the rest of this chapter, we use the term "domain" to represent a group of clusters sharing similar ontological topics, and use the term "cluster" to denote the ontologically equivalent cluster. Clusters and domains do not have fixed boundaries; they are formed by randomly connecting relevant nodes.

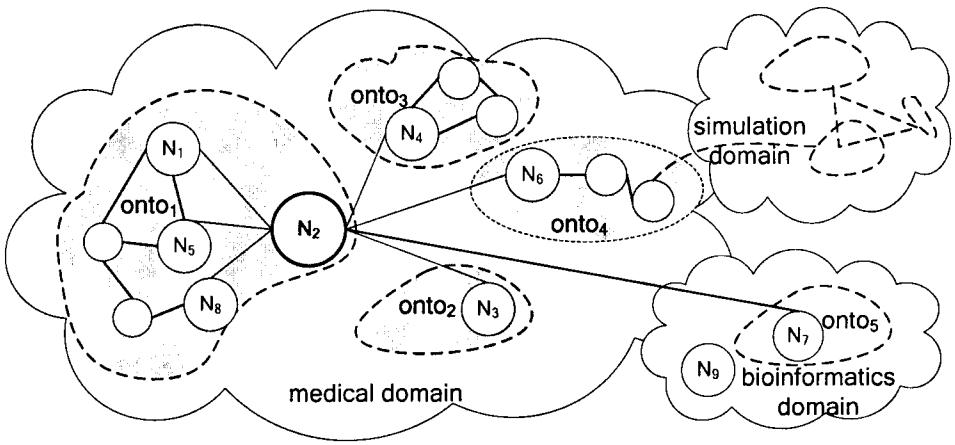


Figure 5.3 A sample network topology

Peers in our system may pose two kinds of queries, neighbor-discovery queries and resource-discovery queries. The neighbor-discovery query is used to construct the semantic small-world topology. When a new node joins the network, it issues neighbor-discovery queries to find semantically related neighbors, so that it can join their domains and clusters by connecting to them. The resource-discovery query is used to locate desirable resources in the network. Once the semantic topology has been created, resource discovery can be performed inside local clusters and domains. To efficiently resolve both queries, each node maintains finer-grained knowledge of neighbors semantically closer to it, but coarser-grained knowledge of neighbors further from it. This reflects the characteristic of our routing strategy, in which the query first walks around the network, and once it reaches the target cluster, it zooms in on that cluster and investigates its detailed ontology properties.

5.3.2 Inter-cluster routing table

The construction of an ontology-based topology is a process of finding semantically related neighbors. A node joins the network by connecting to one or more bootstrapping neighbors. Then the joining node issues a neighbor-discovery query, and forwards the query to the network through its bootstrapping neighbors. The neighbor-discovery query routing is in fact a process of inter-cluster routing and is based on the inter-cluster routing table.

A node's inter-cluster routing table stores the abstract semantic knowledge of its neighboring clusters. Specifically, it keeps contacts to those clusters – its short-distance and long-distance neighbors, their semantic similarities to this node, and their OSS mapped in a compressed Bloom filter. To reconcile the semantic differences between clusters, inter-ontology mappings are also stored in the inter-cluster routing table. A query can then be forwarded to a neighbor after being translated according to the inter-ontology mapping. A neighbor-discovery query is mainly routed over clusters to quickly locate related clusters. A resource-discovery query is always forwarded inside clusters because of the topology's semantic locality property.

Table 5.3 shows the inter-cluster routing table of N_2, a node in Figure 5.3. N_3, N_4, and N_6 are short-distance neighbors of N_2 (assume the similarity threshold is 0.6). N_7 is a long-distance neighbor which links N_2 to a semantically unrelated domain. The Ontology Signature Sets of N_2's neighbors are compressed into a Bloom filter, thus they are sequences of 0s and 1s. The last column of the table stores the inter-ontology mappings between N_2 and other semantically related neighbors. For example, the last column of the first row stores ontology mappings between N_2 and N_3, which includes equivalent class mapping C_a=C_a' and equivalent property mapping P_a=P_a'.

Table 5.3 Inter-cluster routing table of node N_2

Neighbor	Semantic similarity	Compressed OSS	Inter-ontology mappings
N_3	0.8	$onto_2$ [10011010...]	C_a=C_a', P_a=P_a'...
N_4	0.7	$onto_3$ [01101010...]	C_m⊃C_m', P_2⊃P_2'
N_6	0.6	$onto_4$ [11100010...]	C_t⊂C_t' ...
N_7	0	$onto_5$ [00001010...]	none

To control the overhead of routing table maintenance, a soft-state update mechanism is used to keep the routing information up-to-date; nodes periodically probe their neighbors and propagate updated ontology information to them. At any given time, the resource routing information may potentially be stale or inconsistent, but in the long run, they are good enough to direct query forwarding to the right peers.

5.3.3 Neighbor discovery query

A neighbor-discovery query message includes several parts: (1) the querying node's compressed OSSs, (2) a similarity threshold which is a criterion to determine if a node is semantically related to the query (optional), (3) a query Time To Live (TTL) to gauge how far the query should be propagated, (4) a list of clusters (represented by the ontology namespace of the cluster) the query has passed through, so that the query will not be forwarded to the same cluster again and again.

When a node N receives a neighbor-discovery query Q which tries to find neighbors for a new joining node X, N computes the semantic similarity between X and itself. If N is semantically related to X, N will send a *Neighbor Found* reply to X. If the query's TTL has not expired, N computes the semantic similarity between X and each of its neighbors, and forwards the query to semantically related neighbors. If no semantically related neighbors are found, the query will be forwarded to N's long-distance neighbors. The detailed query processing algorithm is illustrated in Figure 5.4.

```
/* When a node N receives a neighbor-discovery query Q issued by a
new joining node X, N calls this function to process the query*/

process_neighbor_discovery_query (query Q)
{
1.  if Q has been received before, discard it, return
2.  compute the semantic similarity between X and N, sim(X,N)
3.  if (sim (X,N) =1)
4.      send a reply indicating N is X's zero-distance neighbor
        the reply also contains N's zero-distance neighbours
5.  if (threshold ≤ sim(X,N) < 1)
6.      send a reply indicating N is X's short-distance neighbor
7.  if (TTL does not expire)
8.      for each neighbor Nj in N's inter-cluster table
9.          compute the semantic similarity sim(X, Nj)
10.         if (sim(X, Nj) ≥ threshold)
11.             forward Q to Nj
12.     if no Nj found
13.       forward Q to N's long distance neighbors
}
```

Figure 5.4 The algorithm of neighbor-discovery query

A neighbor discovery query aims to locate short-distance and zero-distance neighbors for the querying node. Bootstrapping neighbors can be candidates for long-distance neighbors if they are not semantically related to the querying node. Information of short-distance and long-distance neighbors is used to construct a node's inter-cluster routing table. After a node finds its short-distance neighbors, it will contact them to map ontologies with them. Unlike the GONID system in which mappings are globally accessible to the network, mappings here are only between the two neighbors, and queries are translated whenever passing along short-distance links.

5.4 Resource discovery in OntoSum

With the semantic small-world topology constructed, resource discovery can be efficiently performed. In most cases, a resource discovery query can be answered within the querying node's local domain, because queries reflect the querying node's ontology interest, and semantically related nodes are within the neighborhood of the querying node. When a node issues (or receives) a query, it first chooses its zero-distance neighbors to forward the query inside the local cluster. Since they use the same ontology, the zero-distance neighbors are the best candidates to forward the query to. Another important step in query processing is to reformulate a peer's query over other peers on the available semantic paths. Starting from the querying peer, the query is reformulated over the querying peer's short-distance neighbors, then over their short-distance neighbors, and so on until the query TTL expires. Because of the small-world property, the query can get enough answers within a small number of hops with high probability. The query reformulation is according to the inter-ontology mappings. Since the ontology mapping between two clusters rarely maps all concepts in one cluster to all concepts in the other, mappings typically lose some information and can be partial or incomplete; the reformulated query may deviate from the original query's intention, and the query result should be evaluated at the querying node. Feedback on query results can be used to improve the quality of inter-ontology mappings. Moreover, nodes can learn from query results to update their neighbors. Therefore, when a node updates its semantic interests, the system is able to adjust that node's links accordingly.

Sometimes, users may want to locate resources in other semantic domains. In this case, they would first locate the related domain using the inter-cluster routing algorithm; then they can follow procedures just mentioned to process the query in that domain.

The semantic domains and clusters reduce the search time and decrease the network traffic by minimizing the number of messages circulating among domains and clusters. Inside the cluster, nodes randomly connect with their zero-distance neighbors sharing the same ontology schema. Queries looking for particular resources can be routed inside the cluster using flooding- or random-walk- based simple forwarding algorithms. To further improve the performance of intra-cluster searching, we propose an efficient intra-cluster routing algorithm which is presented in the next section.

5.5 RDV routing

To efficiently forward resource discovery queries inside the cluster, we propose the Resource Distance Vector (RDV) routing algorithm. The main idea of this algorithm is to build and integrate each node's ontological instance summaries. When processing a query, the summaries are used in a pre-processing step to find peers that are likely to provide relevant answers to the query. The RDV algorithm can be used independently as a semantics-based routing algorithm in a network with a fixed ontology schema.

5.5.1 Index summarization: triple-filters

Compared with the whole network, the size of a cluster is relatively small. Therefore, it is possible to index more detailed ontology information into the intra-cluster routing table. Unlike the inter-cluster routing tables which store abstract T-box knowledge, the intra-cluster routing table records detailed A-Box knowledge from neighbors inside the same cluster (i.e., zero-distance neighbor). In the rest of this section, we use the term "neighbor" to represent zero-distance neighbor. Every peer maintains a resource index table, and peers exchange their indices. Queries can then be distributed by relaying based on these indices. However, the instance-level indexing can be expensive due to the large number of instances. To reduce the overhead of propagating the index information, we propose a lightweight indexing summarization scheme based on a concise data structure, the *triple filter*, which extends the Bloom filter. Since many parts of this thesis use the Bloom filter technology, we provide a brief introduction about Bloom filters.

5.5.1.1 Bloom filters

A Bloom filter is a compact randomized data structure for representing a set in order to support membership queries. The basic idea is illustrated in Figure 5.5. For a set A composed of n elements: $\{a_1, a_2,\ldots,a_n\}$, a vector v of m bits, initially all set to 0, is allocated to it. Then k independent hash functions, $h_1, h_2,\ldots, h_k$, each with range $\{1,\ldots m\}$ are applied to every element of the set. For each element a in A, the bits at positions $h_1(a), h_2(a),\ldots,h_k(a)$ in v are set to 1. A particular bit may be set to 1 multiple times. To determine if an element b is in the set A, we check the bits at positions $h_1(b), h_2(b),\ldots, h_k(b)$. If any of them is 0, then b is certainly not in the set A. Otherwise we conjecture that b is in the set although there is a certain probability that we are wrong. This is called a "false positive". The parameter k and m should be chosen such that the probability of a false positive is acceptable. There is a clear tradeoff between m and the probability of a false positive rate, which can be estimated by: $(1- e^{-kn/m})^k$.

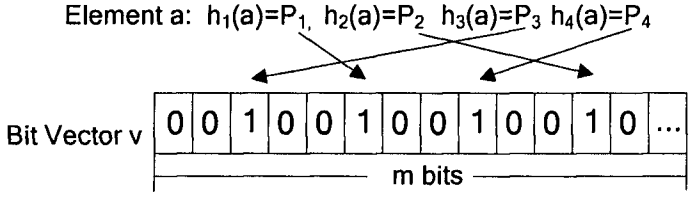


Figure 5.5 Storing an element a into a Bloom filter with k=4

5.5.1.2 Triple filters

A classical Bloom filter is a simple randomized data structure for representing a set and supporting membership queries. It is unable to represent structured data like RDF. To address this problem, we extend the classical Bloom filter to a structure called a triple filter. As mentioned, the building block of

RDF statements is a triple including a *subject*, a *predicate*, and an *object*. Any RDF statement can be represented by a sequence of triples. A triple filter includes three different Bloom filters: the *subject filter*, the *predicate filter*, and the *object filter*. These three filters work together to represent the RDF triples and answer triple membership queries. To store an A-Box RDF statement, the statement is first decomposed to sequence of triples and these triples in turn can be mapped to the corresponding triple filters. For example, in Figure 5.6 an RDF triple: (:JavaProgramming, :creator, Ken Arnold) is mapped to the triple filter. In this example, each filter's size is 16 bits, and 3 hash functions (h_1, h_2, h_3) are used to map an element to the filter. In reality, the sizes of the subject filter, the predicate filter, and the object filter are different, so are the number of hash functions used on these filters. Normally, the object filter has larger size and uses more hash functions, while the predicate filter has smaller size and uses fewer hashes, because a particular ontology usually has more distinct objects than distinct predicates. To identify the existence of a triple, three parts of the triple are mapped to the corresponding filters. If all of them are found in the triple fitter, we conjecture that the queried triple exists. However, this conjecture may be false, because even when all parts of a triple are found in the filter, these parts may belong to different resource instances. Luckily, this false positive does not affect the system fidelity, but relaxes the filtering requirements; the query has to be evaluated by the real data source anyhow.

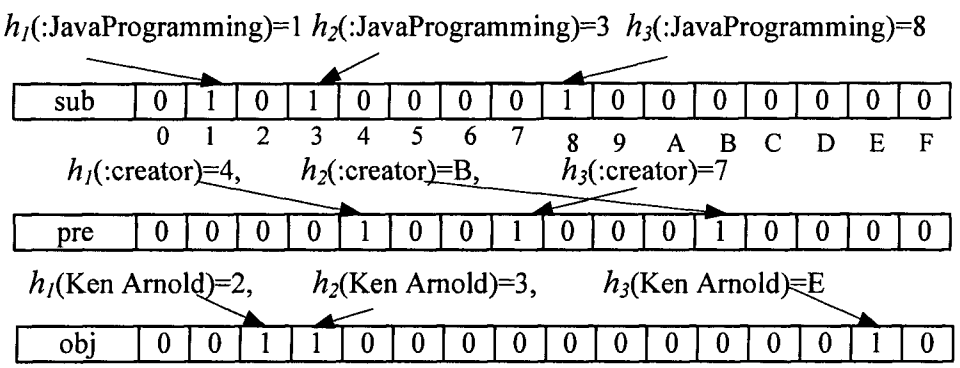


Figure 5.6 A triple filter example. The triple <*JavaProgramming, :creator, Ken Arnold*> is mapped to a triple filter by 3 hash functions: h_1, h_2, and h_3

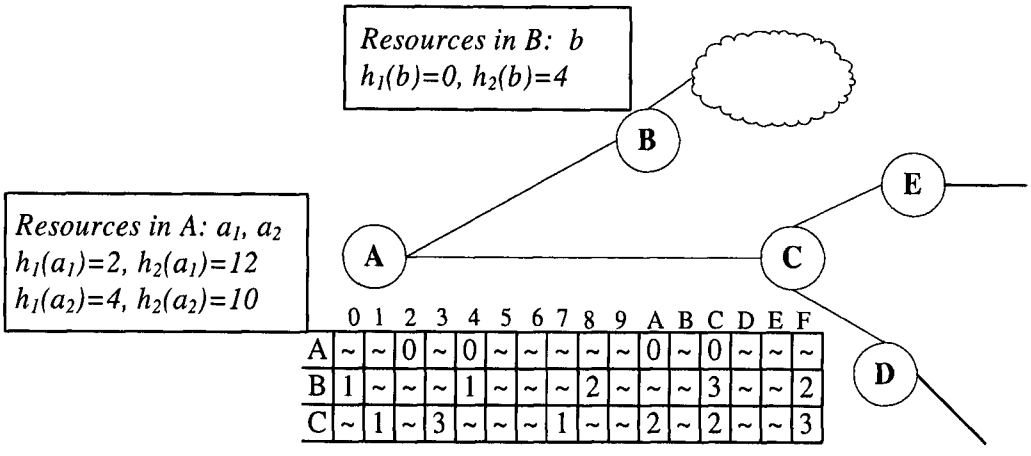
5.5.2 Routing table

5.5.2.1 From triple filter to RDV routing table

The triple filter is a compact summary of a node's local resource information. If a node knew every other nodes' resource summary, then it could accurately send a query to exactly the right nodes. Although transferring the summary instead of real data can significantly reduce the network traffic, it is still prohibitive for every node to know each other in a large system. To solve this problem, we propose a distance-vector-based query routing algorithm, RDV, in which a node can make routing decisions by knowing only its immediate neighbors and limited resource information. We innovatively construct the

RDV routing table (RDVT) based on the triple filters. Specifically, each node in the network keeps a modified triple filter for every neighbor (adjacent node) in the overlay topology. A neighbor filter is created by merging filters of all nodes *d* hops away from that neighbor; therefore it keeps track of resources reachable via *d* hops through the overlay network starting with that neighbor. We add distance information to the triple filter, so that we can not only know how far away the resource is located, but also control how far a node can "see" its neighborhood; together with the neighbor summaries, we can determine where to forward a particular query. In a routing table, each entry in the triple filter is not a single bit but rather a small counter. Initially, all entries are set to *infinity* (represented by a special number). When a local resource is inserted, the corresponding counters are set to *0*, meaning the distance is *0*, representing a local resource. When the summary is propagated to another node, the counters corresponding to each resource are incremented. To control the false positives caused by Bloom filter aggregation, we set the maximum value of the counter, which we call the *radius*. The *radius* limits the number of hops the resource information can be propagated. After a series of propagations, if a resource is propagated to a node which is more than *radius* away, then its entries in the RDVT are set to *infinity* (not available). Because of the small-world theory, nodes are connected with a small number of hops. Therefore, a small *radius* works for our system. As revealed by our experiments, 3 bits per counter should suffice.

The RDV routing table contains both local and neighbouring filters. The first row of the table is the local filter containing the index of local resources. The rest of the rows represent resources accessible from neighbors but not just the resources of immediate neighbors. Each element in the filter is associated with a distance number representing the minimum distance to a related resource (~ represents *infinity*). Figure 5.7 shows part of a network and a particular node *A*'s routing table. For brevity, only one of the three filters is shown here. The size of the filter vector is two bytes and two hash functions are used to map the key. In A's routing table, the first row contains two local resources: a_1 and a_2. Resource a_1 is mapped to indices *(2,12)* and a_2 is hashed to indices *(4,10)* of the filter, thus those indices are set to *0* in the filter. The second row of *A*'s routing table contains resources that can be reached through neighbor *B*. *B* has a local resource *b* which is hashed to *0* and *4*, therefore positions *0* and *4* are set to *1* in the second row of *A*'s routing table; this means *A* can access resource *b* with *1* hop; also in the same row, we can see *2* and *3* in some positions, meaning *A* can reach these resources by *2* hops or *3* hops through node *B*. These resources are not located in node B itself, but in *B*'s neighborhood. Similarly, the third row of *A*'s RDVT records resource information reachable from neighbor *C*.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A	~	~	0	~	0	~	~	~	~	~	0	~	0	~	~	~
B	1	~	~	~	1	~	~	~	2	~	~	~	3	~	~	2
C	~	1	~	3	~	~	~	1	~	~	2	~	2	~	~	3

Figure 5.7 Example routing table of node *A*

5.5.2.2 Merging of the routing table

Nodes should be able to exchange their resource information efficiently, so that they can construct their routing tables and keep them up-to-date. To save bandwidth, a node merges its local and neighbouring filter vectors to one single vector to send to its neighbours, instead of sending the whole table. Since we changed the filter from a bitmap to a vector of numbers, we cannot merge them by *bitwise or* as we would merge the standard Bloom filter. Merging of filters must include all resources in each of the participating filters, and also keep the right distance information. With this requirement in mind, we implement the merge operation by obtaining the smallest counter values from participating filters in the corresponding positions. The rationale behind this operation is that if filters representing different distances to the same resource are merged, the shortest distance should be recorded. However, different resources may overlap on some hash positions. After merging, positions related to a resource may hold different distance values, since some positions may overlap with other resources and be changed to a smaller value by the merging process. It is not difficult to see that, in the merged filter, a resource's distance value would be the largest value among all values of its corresponding hash positions.

Figure 5.8 shows how node *A* merges its local and neighbour filter vectors to one vector *A'*. It is clear that each element of *A'* is the minimum of all corresponding elements of *A*, *B*, and *C*. Now let us look at a resource *c* corresponding to positions *3* and *7* in *C*'s vector. Its distance values in *C*'s vector is 3 *(C(3)=3, C(7)=3)*. Because resource *c* has a hash position *7* which overlaps with a resource in vector *B*, position *7* is updated to a smaller value, *1*, after merging. In the merged vector *A'*, *A'(3)=3, A'(7)=1*, we can see that the larger distance value: *3* in position *3* represents the actual distance to resource *c*.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
A	~	~	0	~	0	~	~	~	~	~	0	~	0	~	~	~
B	1	~	~	~	1	~	~	1	2	~	~	~	3	~	~	2
C	~	1	~	3	~	~	~	3	~	~	2	~	2	~	~	1
A'	1	1	0	3	0	~	~	1	2	~	0	~	0	~	~	1

Figure 5.8 Merging of three filter vectors: *A*, *B*, *C* to one filter vector *A'*

The merging process guarantees that the nearer resource information always gets higher priority in the filter, because a position occupied by a nearer resource would never be overwritten by a further resource. This property in turn guarantees that increasing *radius* will not bring more false positives for a limited sized triple filter, which will be proved by our experiments as well.

A node accumulates its local and neighboring resource filter vectors into one vector and exchanges the merged vector with its neighbors. By exchanging the merged vectors, we reduce both the amount of information transmitted and the storage used. When a merged vector arrives at the next hop, all of its distance counters are increased by *1*. As mentioned, the *radius* limits how far the resource information can travel. In the merged vector, if an element's value equals *radius*, we reset the value to *infinity* ("~" in the figure), representing "not available". Since our network possesses the small-world property, nodes are connected with a small number of hops; a small *radius* value is sufficient to gather the neighbourhood resource information. This has been verified by our experiments. Using the *radius* to limit the propagation of information also reduces false positives caused by resource information aggregation. By accumulating and exchanging routing tables, eventually all resources within the range determined by *radius* are known.

5.5.2.3 Construction of the routing table

When a node joins the cluster, it should construct its routing table, RDVT. Neighbours of this new node should update their RDVTs to reflect the joining of this new node. Figure 5.9 illustrates the RDVT updating process when a new node *C* joins the network. Node *C* joins the network by connecting to an existing node *A* in the network. After the connection is established, node *C* sends its resource indices to *A*. Similarly, *A* should inform *C* of all the resources *A* has knowledge of. Specifically, *A* merges its local and neighbor vectors into one vector and sends it to *C*. The merged vector of *A* represents resources accessible from *A* and their shortest distances to *A*. *A* does not need to send more information as *C* does not need to know the precise location of these resources, but only that they can be accessed through *A*. After *C* receives the merged vector from *A*, it adds *1* hop to each element of the vector, and adds an additional row in its RDVT (as shown in Figure 5.9 (b)). After *A* receives *C*'s resource information and updates its routing table, it informs its neighbors (in this case, node *B*) of the update. In this way, nodes can construct and update their RDVTs.

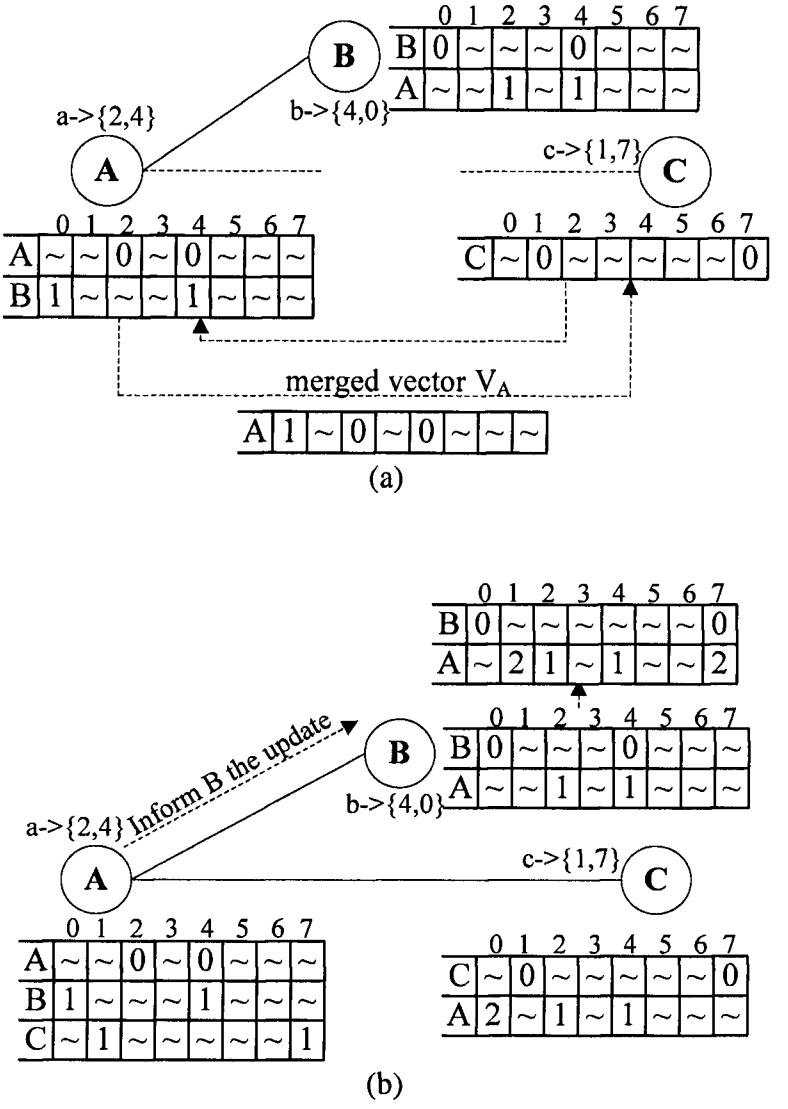


Figure 5.9 Construction of the routing table

5.5.2.4 Updating the routing table

A node's routing table should be updated when the resource information changes. When new resources are added to a node, this node calculates the changed positions in its own filter (1st row in its routing table) and the merged filter. It then sends these positions out to each neighbor. On receiving such a message, each neighbor changes these positions in the row corresponding to that node and computes the changes it will make in its merged filter. These changes are sent out as well. The deleting process is more complex. Because of the overlapping of different resources, deleting cannot be performed by simply setting the related hash positions to *infinity*. We can solve the problem by using the counting Bloom filter proposed by Fan *et al.* [27], or using the timing-based deletion approach [11]. A resource update can be implemented as a deletion followed by an addition. Each node sends updates to and receives updates

from its directly connected neighbors. Nodes also periodically "ping" their neighbors to make sure that they are still alive. To reduce the overhead of transmitting routing information, a soft-state update mechanism is used, in which routing information is exchanged periodically. At any given time, the resource routing information may potentially be stale or inconsistent.

5.5.3 Query forwarding

Based on the routing table RDVT, we propose a so-called resource-distance-vector (RDV) routing algorithm. It uses a distance vector approach to route the query to the nearest matching nodes. The traditional distance vector approach is not scalable for locating unique nodes in a large network, but this modified version is extremely well suited for our resource discovery problem.

When a node receives a query, it converts the query into a triple sequence and matches the sequence in the RDVT. Besides matching local resources, the query is also forwarded to the "right" neighbors. A query may be transferred several hops until arriving at the matching node or the query TTL expires. Figure 5.10 illustrates a query routing example. We only show one of the three triple vectors. For simplicity, the query has only one constraint. The *radius* is set to *3*, so nodes are only aware of resources within *3* hops. In this example, node *A* receives a query for resource *e* (which is mapped to two positions: *3* and *6* in the filter). It checks its routing table and finds two matches: through *C* with *2* hops (C_3=2, C_6=2) and through *D* with *3* hops (D_3=3, D_6=3). Since the shortest distance to the resource is *2* through neighbor *C*, the query is forwarded to *C*. Similarly, *C* forwards the query to *E*. *E* finds a match in its local vector, and then it checks the RDF database against the original query.

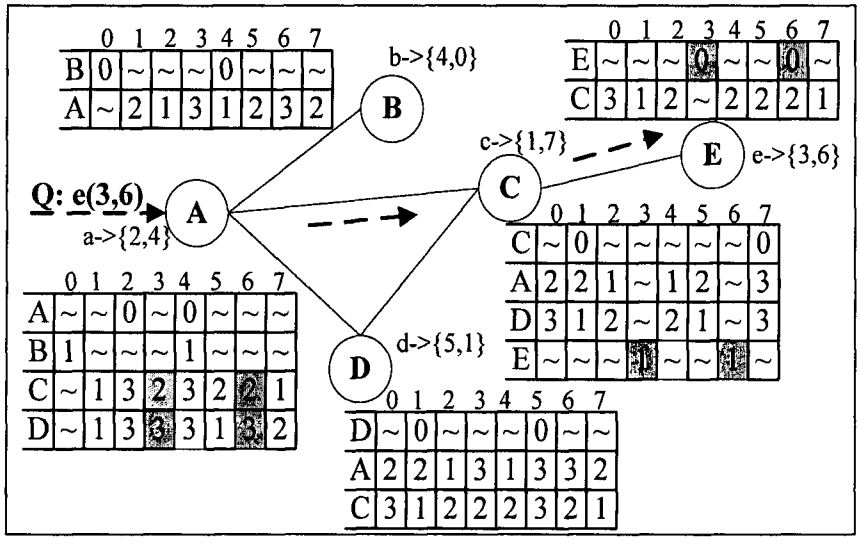


Figure 5.10 RDV query routing

Our routing algorithm works fine with networks containing cycles. Because of cycles, a node may receive a query multiple times. To avoid processing queries more than once, every query has a unique query ID and every node keeps a list of recently received query IDs. If a query has been received before,

it will be discarded. Another benefit of recording the query is that it ensures the query does not hit the same false positive twice.

5.5.4 Heuristic jump and caching

By setting a *radius*, we limit the distance a node's resource information can travel. This reduces false positives, but at the same time, this causes a node not to have global knowledge of the network but have only a local view of the neighborhood. Because of this, a node may not find enough matches from its RDVT to forward queries. A naïve solution is to forward the query to some random neighbors even if they have no match hoping that these neighbors can find matches from their neighborhood. This method is inefficient since a node's neighbor has a neighborhood which largely overlaps its own. If the requested resources are scarce in the local area, forwarding the query to another neighbor in this area will not substantially increase the chance of resolving a query. To address this problem, we introduce a forwarding method called the "heuristic jump."

This method allows the system to keep additional long-distance links as an addendum to the RDVT. When the RDVT cannot resolve the query, the query will "jump" to remote nodes the links point to. To discover those long-distance links, the system employs an aggressive caching technique. After finding the result of a query, the result travels along the reverse path to the requester. Whenever it is passed through a node, it is cached in that location. Every internal node caches the query, the destination node, and the distance to that node. We use caching to not only eliminate the need to forward a query which may be resolved locally, but also use this cached information as links for future long-distance jumps. During the query-forwarding process, when a node cannot find enough matches in its routing table, it chooses appropriate long-distance links from its cache and forwards the query accordingly. This expedites the searching process by jumping over barren areas. Candidate long-distance nodes should be located outside the neighborhood area; i.e., the distance should be greater than *radius*. In our heuristic, we also consider other metrics. For example, the query might "jump" to nodes that answered more previous queries, or to nodes that answered similar queries. Our experiment in Section 5.7 shows that forwarding by "heuristic jump" improves search efficiency.

5.5.5 Query evaluation

Like the GONID system, OntoSum supports SPARQL queries as well. But unlike the GONID system which distributes triples of a single resource to different DHT nodes, OntoSum stores a resource's description in its original publisher's site. Therefore, the query evaluation in OntoSum may avoid the many joins required in GONID. A query is broken up into triple patterns, and then all the triple patterns will be hashed and matched with neighboring triple filters. The query is forwarded to nodes whose triple filters match the hashed query. At the destination node, the query will be further matched with the detailed metadata repository. How to resolve SPARQL queries with triple patterns has been explained in the previous chapter. Since we use a similar idea, we do not repeat the explanation. We use one example

to illustrate the query evaluation process. In this example, a conjunctive query Q is issued to locate printers with particular properties:

Q: {(?printer :printMethod "Thermal Inkjet"), (?printer :connectivity :USB)}.

The two triples in the query are hashed and matched with triple filters. Specifically, *:printMethod* and *:connectivity* are hashed and matched with the *predicate filter*; *Thermal Inkjet* and *:USB* are hashed and matched with the *object filter*. Then the hashed query will be forwarded to nodes, both of whose filters match the query. When the query arrives at the destination, it is matched with the local metadata repository. If results located are not enough, the query can be relaxed by removing some requirements.

The above mentioned query processing requires that the queried resource be described and published by a single provider. This is the most common case for describing a physical resource such as software resources and hardware resources. In practice, most of the queries belong to this case. However, there are exceptional cases, in which annotations for one and the same resource may be distributed. This may be applicable for describing virtual resources such as information and knowledge. With the evolution of their knowledge, participants of the system may add new knowledge to complement the existing knowledge base. Thus, one node might store metadata which includes some properties for specific resources using an ontology; other nodes could hold metadata that provides different properties for the same resources, probably using another ontology. The inter-ontology mapping *referentialClass* defined previously is used to connect ontologies that complementarily describe the same resource. A query searching for such resources using multiple dispersed ontologies cannot be processed by simply matching neighboring triple filters, because triple filters summarize resources defined with the same ontologies. To process such queries, we break the query into multiple sub-queries, and use local ontologies to answer the first sub-query, then forward the other sub-queries to nodes which have *referencialClass* mapping with the local ontologies. All intermediate results can be returned to the querying node and operations such as *join, union,* or *conjunction* can be performed on intermediate results to get the final result.

5.6 Experiment

The semantic building blocks of OntoSum are the same as that of GONID. For example, they use the same metadata representation and the same mapping and reasoning strategies. The only difference between these two systems is the searching and indexing method. Therefore, our simulation experiment in this part focuses on evaluating the performance of searching. In the rest of this section, we will explain the experiment setup, and then present the simulation results.

5.6.1 Setup

The test data is artificially generated and the procedure follows that of the GONID experiment described in Section 4.5.1. In short, the ontology schemas are generated first, and then individuals are created by

instantiating classes. We assume for simulation purposes that ontologies and queries are associated with a specific domain, and all ontologies in the same domain have ontology mappings defined in advance. Queries were generated by randomly replacing parts of the created triples with variables. Single triple queries and conjunctive triple queries are used as the representative query format in this experiment.

The simulation is initialized by injecting nodes one by one into the network until a certain network size has been reached. The network topology created this way has power-law properties; nodes inserted earlier have more links than those inserted later. This property is consistent with the real world situation, in which nodes with longer session time have more neighbors. After the initial topology is created, a mixture of joins, leaves, and queries are injected into the network based on certain ratios. The proportion of join to leave operations is kept the same to maintain the network at approximately the same size. Inserted nodes start functioning without any prior knowledge.

For comparisons, we simulate our searching scheme OntoSum in conjunction with the learning-based ShortCut scheme [109] and a random-walk based simple Gnutella scheme [129]. The ShortCut approach is chosen as one comparison reference since it is simple yet effective, and many popular applications (e.g., [109], [100], [18], [19]) use this approach as their basic routing scheme. Moreover, it is comparable to our approach in the sense that it creates clusters on top of the unstructured network. The ShortCut approach relies on the presence of interest-based locality to create "shortcuts". Each peer builds a shortcut list of nodes that answered previous queries. To find content, a peer first queries the nodes on its shortcut list and only if unsuccessful, floods the query. This approach presents a promising reorganization method within unstructured P2P networks. Flooding-based Gnutella was chosen as another reference approach for its simplicity and prevalence, which, in fact, made it a widely used baseline for many previous research efforts. We tested two versions of OntoSum, OntoSum_0 and OntoSum_1. The former has no intra-cluster RDV routing table and uses random-walk to forward queries inside the cluster; while the latter uses the RDV routing scheme (with *radius 1*) to forward queries inside a cluster. The reason to use *1* as the value of *radius* in this experiment is to save memory storage to support large-scale test. When *radius* is *1*, the RDV table does not need to maintain distance information and it is simplified as a Bloom filter bitmap. How *radius* affects the system performance is evaluated with a separate RDV performance experiment presented in Section 5.6.2.4.

The resource-discovery query is propagated exponentially, i.e., each node chooses a certain number of neighbors (called walkers) to forward the query. The neighbor-discovery query (for OntoSum only) is propagated linearly, i.e., only the node that issues the query forwards the query to a certain number of walkers, while all other nodes only forward the query to one neighbor. In the rest of the paper, we use the term "query" to refer to resource-discovery query.

The simulation parameters and their default values are listed in Table 5.4.

Table 5.4 Parameters used in the simulations

Parameter	Range and default value
network size	2^9~2^{15} default: 10,000
initial neighbors (node degree)	5
maximum neighbors	30
average node degree	14
TTL	1~20 default 9
resource-discovery query walkers	3 (propagate exponentially)
neighbor-discovery query walkers	2 (propagate lineally)
ontology domains	1~10 default: 8
ontology schemas per domain	1~10 default:8
distinct resources per domain	100
resources per node	1~10
RDV table radius	1
die/leave probability per time slice per node	0-21%, 3% default
resource change probability per time slice per node	20%instance update, 2% schema update
query probability per time slice per node	5%
RDVT update frequency	every 5 time slices
sample of nodes to compute diameter	5%

5.6.2 Results

In this part, we present the experimental results which demonstrate the performance of our searching scheme.

5.6.2.1 Emergence of the small-world

As discussed, the topology of the peer network is a crucial factor determining the efficiency of the search system. We expect that the OntoSum semantic neighbor discovery scheme will transform the topology into a small-world network. To verify this transformation, we examine two network statistics, the *clustering coefficient* and the *average network path length*, as indicators of how closely the topology has approached a "small-world" topology.

The *clustering coefficient* (*CC*) is a measure of how well connected a node's neighbors are with each other. According to one commonly used formula for computing the *clustering coefficient* of a graph (Eq. 5.2), the *clustering coefficient* of a node is the ratio of the number of existing edges and the maximum number of possible edges connecting its neighbors. The average over all |V| nodes gives the *cluttering coefficient* of a graph (Eq. 5.3).

$$CCv = \frac{\text{\# of edges between v's neighbors}}{\text{maximum \# of possible edges between v's neighbors}} \tag{5.2}$$

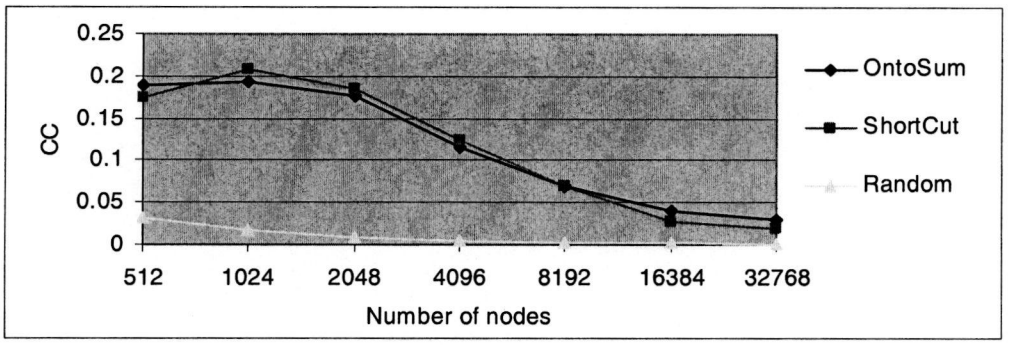
$$CC = \frac{1}{|V|}\sum_{v} CCv \tag{5.3}$$

The *average path length (APL)* is defined as the average shortest path across all pairs of nodes (Eq. 5.4). The *APL* corresponds to the degree of separation between peers. For a large graph, measuring distances between all node pairs is computationally expensive; therefore an accepted procedure is to measure it over a random sample of nodes [113]. In our experiment, we use a random sample of certain percent of the graph nodes. We use Dijkstra's algorithm to compute the shortest distance between pairs of nodes. In our simulated topology we intentionally make the network strongly connected, so that any pairs of nodes have a directed path.

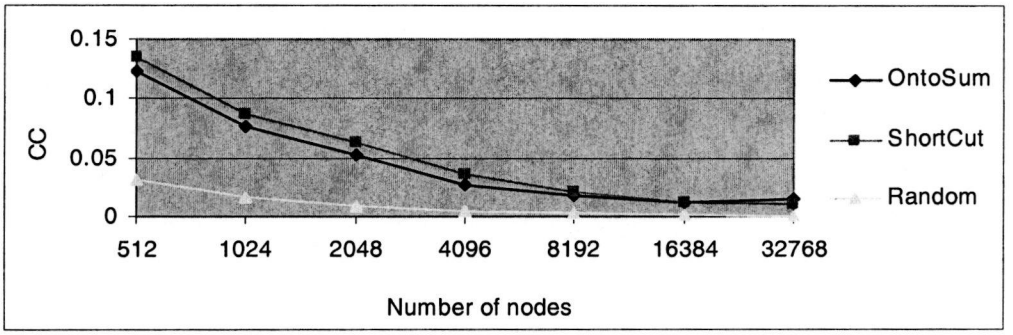
$$APL = \frac{\sum_{ij} l_{i,j}}{|V| \cdot (|V| - 1)} \tag{5.4}$$

We performed experiments to measure OntoSum's *cluster coefficient* (*CC*) and *average path length (APL)*. An interest-based ShortCut topology and a random power-law topology with the same average node degree are used as reference topologies. The former has been proved to be a small-world system [47]. For the ShortCut scheme, test results are collected after the system has had an extensive training process, i.e., nodes have learned as many ShortCuts as possible through query results and the system topology has become stable.

Figures 5.11 and Figure 5.12 show plots of the *clustering coefficient* and the *average path length* as a function of the number of nodes in the network. The system has two configurations; in Figure 5.11 (a) and 5.12 (a), nodes have more ontologies to choose from, while in Figure 5.11 (b) and 5.12 (b), nodes have fewer ontological domains. We observe that both the *clustering coefficient* and the *average path length* of OntoSum are very similar to those of ShortCut. The *clustering coefficients* of OntoSum and ShortCut are much larger than that of the random power-law network, while the *average path length* of OntoSum and ShortCut are almost the same as that of the random network. This indicates the emergence of a small-world network topology [113]. Note: Because all of the three topologies are created by inserting nodes to the existing system, all topologies show the power-law property to some extent, and thus the *average path length* of all three topologies are smaller than a random network. This set of experiments verifies that firstly, well connected clusters exist in the OntoSum system; due to the semantic similarity definition, these clusters correspond to groups of users with shared ontological interests. Secondly, there is, on average, a short path between any two nodes in the system topology graph; therefore, queries with relatively small TTL would cover most of the network. Our later simulation experiments will verify this.

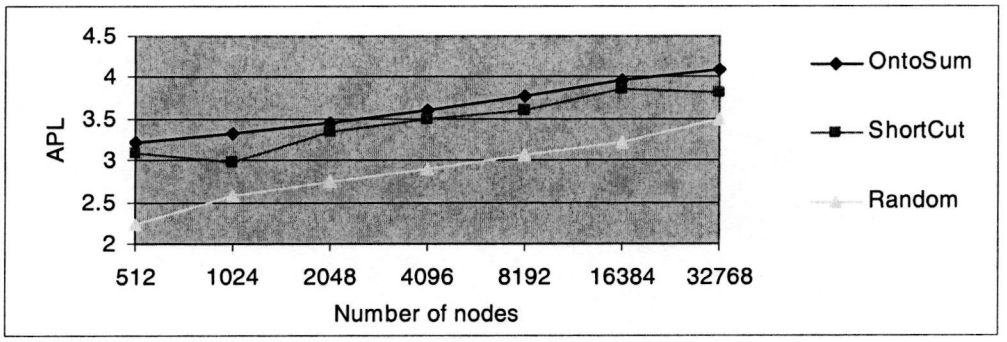


(a) With 10 ontological domains, and 5-10 ontologies per domain

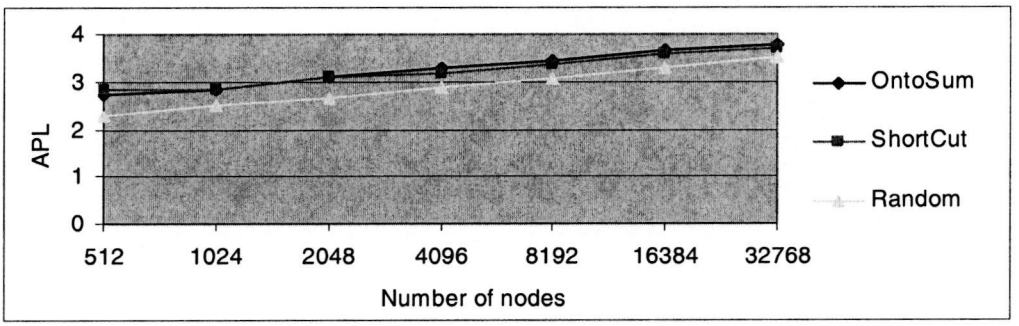


(b) With 4 ontological domains, and 2-4 ontologies per domain

Figure 5.11 Comparison of clustering coefficient



(a) With 10 ontological domains, and 5-10 ontologies per domain



(b) With 4 ontological domains, and 2-4 ontologies per domain

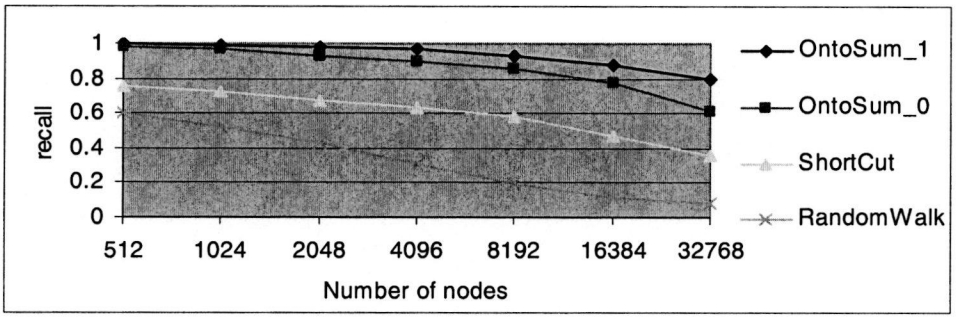
Figure 5.12 Comparison of average path length

5.6.2.2 Scalability and efficiency

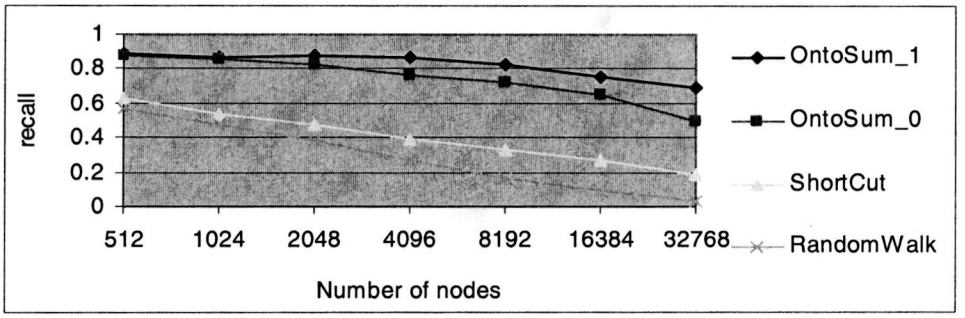
We examine the system performance in three different aspects, namely routing scalability, efficiency, and accuracy by executing the experiment in different network configurations. The performance is measured using the metric of recall rate, which is defined as the number of results returned divided by the number of results actually available in the network, as shown in Equation 5.5.

$$R = \frac{\text{results retrived}}{\text{results available}} \tag{5.5}$$

For comparison, we also implement the learning-based ShortCut algorithm and random-walk based Gnutella algorithm. For the ShortCut approach, we collect query results after sufficient learning has been done. To simulate dynamic factors, in each time slice every node has a *5* percent probability to issue a query, and a *2* percent probability to leave the system. The probability of new nodes with new resources joining the system is the same as the probability of a node leaving. First, we vary the number of nodes from 2^9 to 2^{15} to test the scalability of the routing scheme. The results are listed in Figure 5.13. As we expected, both versions of OntoSum get higher recall in all these different sized networks and in both static and dynamic environments. In addition, OntoSum's recall decreases less with the increase in network size. Figure 5.14 illustrates the system efficiency by showing the relationship between query recall rate and query TTL. With a small TTL, OntoSum gets a higher recall rate than the other two algorithms. This means that OntoSum resolves queries faster than the others. In Figure 5.15 we show the effect of dispatching a different number of walkers to search the network. We can see that with the same TTL, OntoSum locates more results with fewer walkers. For example, OntoSum_1 gets a recall rate of up to 50% with only a single walker in the static network. This indicates that OntoSum routing is more accurate and can always find the right node to forward the query to. From Figures 5.13, 5.14 and 5.15, we also notice that OntoSum performs better than ShortCut and random-walk in both static and moderately dynamic environments. How the system dynamics affect the system performance is further evaluated in the next section.

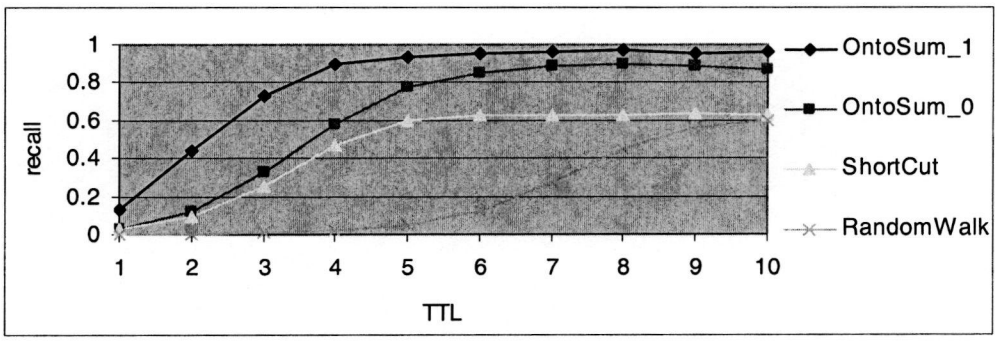


(a) Static environment

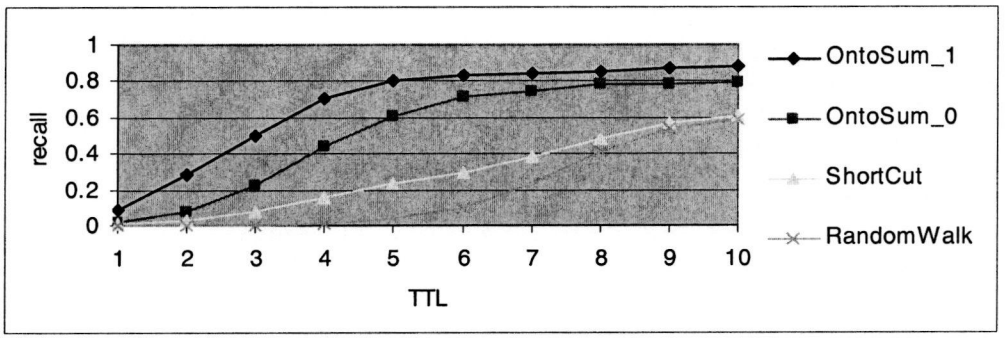


(b) Dynamic environment

Figure 5.13 Recall rate vs. network size

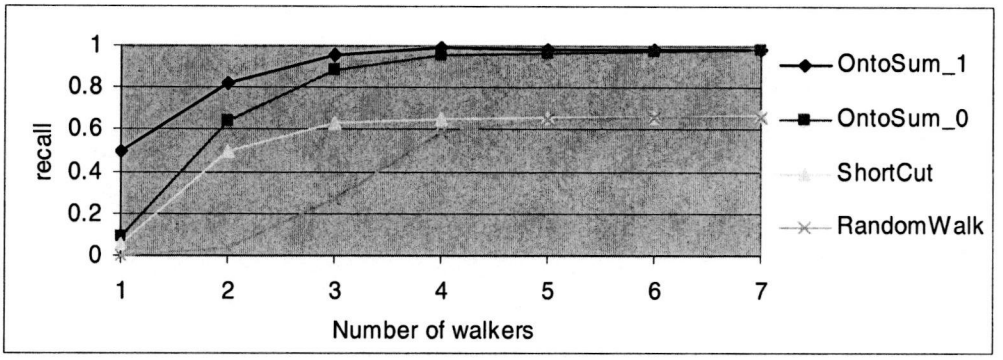


(a) Static environment

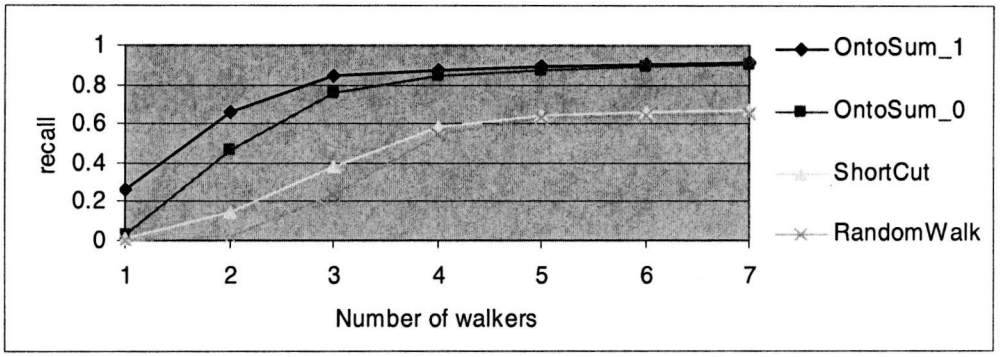


(b) Dynamic environment

Figure 5.14 Recall rate vs. TTL (with # walkers=3)



(a) Static environment



(b) Dynamic environment

Figure 5.15 Recall rate vs. walkers (with TTL=5)

As expected, our OntoSum searching scheme performs well as measured by recall rate in both static and dynamic networks. OntoSum's small-world topology effectively reduces the search space, and its ontology summary guides the query in the right direction. Therefore, OntoSum can locate results faster and more accurately. This explains why OntoSum scales to large network size and why it achieves higher recall with shorter TTL and fewer walkers. Besides all these reasons, another factor contributing OntoSum's overall better recall rate is that OntoSum is able to locate semantically related results that cannot be located by the ShortCut and random-walk. Because of the semantic heterogeneity of our experimental setup, relevant resources may be represented with different ontologies. OntoSum may use its ontology signature set to find semantically related nodes and use the mapping defined to translate the query. Therefore, it can locate most of the relevant results. However, for ShortCut and random-walk, they have no way to find semantically related resources. Therefore, they can only locate resources represented in the same ontology as the ontology of the querying node.

5.6.2.3 Overhead and adaptability to dynamics

The good recall performance of OntoSum does not come for free. Generally speaking, the more efficient the query searching is, the more the system has to pay for maintaining the system structure or indexing the resource information, i.e., there is a tradeoff between query efficiency and maintenance overhead. Unlike ShortCut and random-walk approaches, which only create query propagating overhead, OntoSum

also creates overhead for maintaining the inter-cluster and intra-cluster routing table. We expect the extra overhead is reasonable and the saving from query cost exceeds the extra maintenance cost. To verify this, we examine the system's overhead in terms of accumulated bandwidth and compare it with that of ShortCut and random-walk. System overhead has a close relation with the system dynamics, as a system must maintain consistent information about peers in the system in order to operate most effectively. Therefore, we measure the system dynamics together with the overhead. To evaluate the adaptability to different levels of dynamics, we measure the system overhead under different levels of peer "churn rate" and "update rate", referring to the rate of peers leaving/joining the system and the rate of resource updates. Experiments in this section are performed on a 10,000-node network. The churn rate is represented as the probability for a node to die/leave the system in unit time slice; to maintain the constant number of network size we also insert an equal number of new nodes into the system. The update rate is the probability for a node to update its resource information in a time slice.

The experiment shown in Figure 5.16 gives an overview of how dynamics affect the system performance. Specifically, it shows the query recall rate under different dynamic configurations. In the experiment, we increase the dynamics by increasing the churn rate. From the figure, we find that OntoSum performs similarly to the ShortCut algorithm which is proved to be resilient to churn [109]. When peers join or leave frequently, the performance of ShortCut and OntoSum deteriorate gracefully. Churn does not affect the two schemes dramatically because both algorithms do not depend on a strict structure to perform routing as DHTs do. Their unstructured random topologies provide multiple routes to a destination thus increasing the system resilience. In the worst case, they degrade to random-walk. Another observation is that when the system is more dynamic, OntoSum_1 degrades to OntoSum_0. This is easy to understand because when the system is more dynamic, the resource information in the RDV table is not accurate. In the worst case, using the RDV table to forward the query is like randomly choosing a neighbor to forward to.

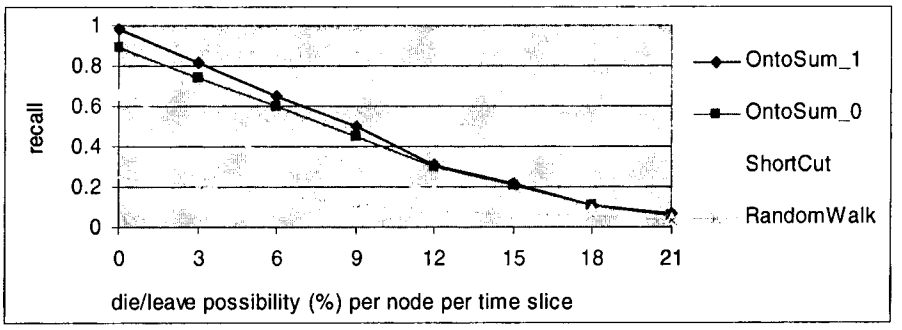


Figure 5.16 Recall vs. churn rate

Figure 5.17 shows the accumulated bandwidth overhead of finding 10000 results under different churn rates. We use a soft state approach to update the routing table: the routing table is updated periodically instead of in real time. From the figure, we can see that in most situations OntoSum produces much less overhead then the other two methods, and that OntoSum_1 is even better than OntoSum_0. But when the system is very dynamic, such as when the dying probability is beyond 20%, OntoSum produces much

more overhead. When the system is very dynamic, the neighborhood relationship changes frequently, and OntoSum creates great amounts of overhead maintaining its routing table. Even worse, the overwhelming maintenance overhead does not bring much benefit in this situation, because the newly constructed topology will change quickly. Luckily, churn of the nature described above rarely happens in reality [101], and we can see from Figure 5.17 that with this churn rate, ShortCut degrades to random-walk. The high overhead problem of OntoSum in very dynamic environments can be solved by a simple solution: when the network is very dynamic, the system can give up the ontology-based topology construction and routing and resort to basic Gnutella random-walk as the solution.

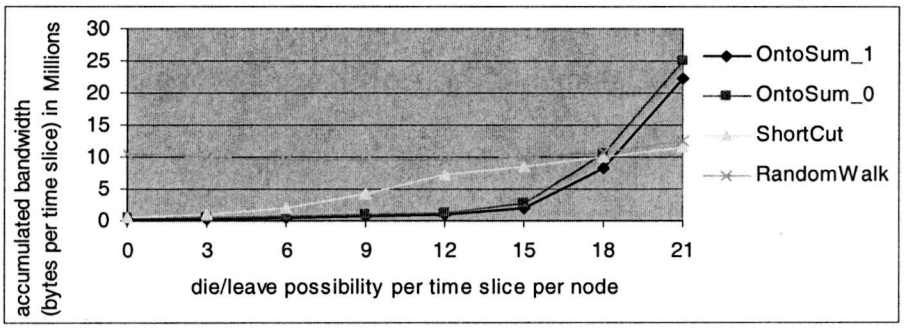
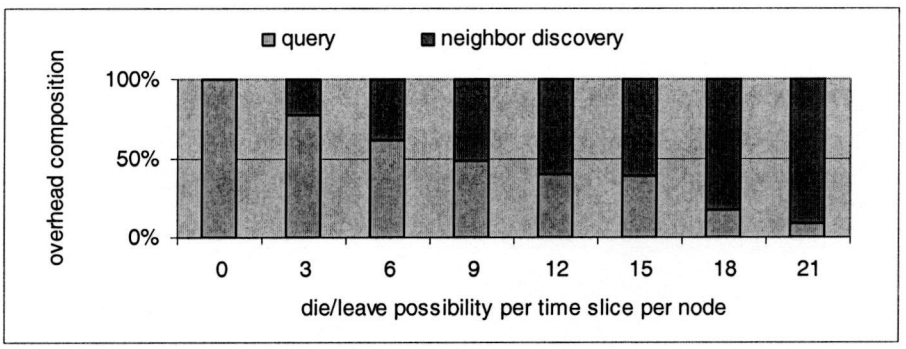
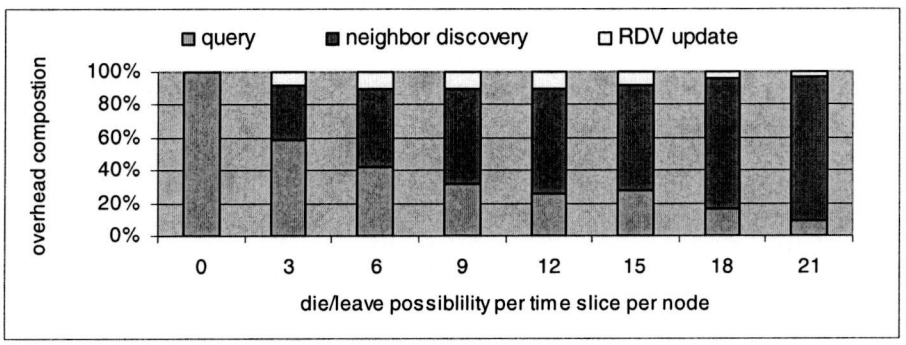


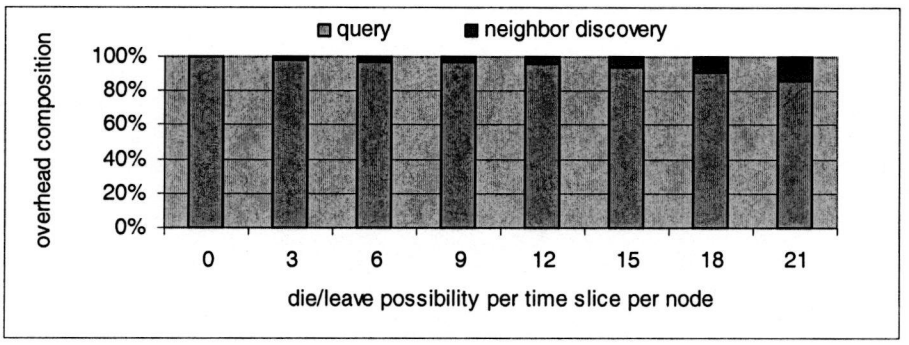
Figure 5.17 System overhead vs. churn rate



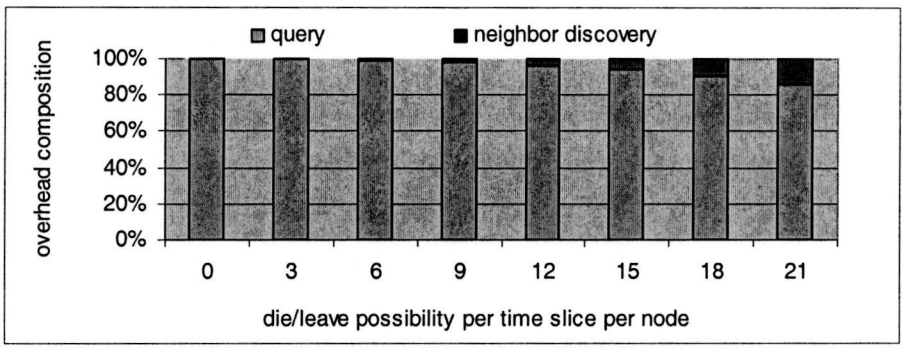
(a) OntoSum_0 overhead composition



(b) OntoSum_1 overhead composition



(c) Shortcut overhead composition



(d) Random-walk overhead composition

Figure 5.18 Overhead composition vs. churn rate

With the same configuration as the experiment in Figure 5.17, Figure 5.18 illustrates the overhead composition of each routing approach. Most of the overhead of ShortCut and random-walk is caused by query forwarding, and a little overhead is caused by finding neighbors when new nodes are inserted into the system. For OntoSum, the neighbor discovery overhead accounts for a higher proportion of the overhead when the system is more dynamic.

We also performed a set of experiments to evaluate the system overhead under various resource update rates. The resource update rate is represented by the probability of a resource change per node per time slice. There are two types of updates: one is called instance update and the other is called the ontology (schema) update. In an instance update, a node keeps its original ontology schema and only updates instances of that ontology. In this case, each node changes 20% of its resources per update. An ontology update, on the other hand, changes the ontology schema and of course all the related instances. This kind of change is a dramatic change: it means the node totally changes its interest, and the practical effect is the same as inserting a new node. The rate of these two types of changes is set to 10:1. The RDV routing table is updated periodically every five time slices in the simulation.

Figure 5.19 illustrates the relationship between system overhead and resource update frequency. It is clear that the rate of resource updates has a bigger impact on OntoSum_1 than on the other algorithms. Because OntoSum_1 has to update its RDV routing table to reflect the changing resources, this unavoidably causes more overhead when resource update is frequent. When the resource update is so

frequent that the routing table update cannot catch up to the resource update, the information in the routing table cannot represent the real resource distribution, and maintaining the routing table becomes useless. Therefore, in a very dynamic environment, we recommend the system stop intra-cluster routing table updates, and turn to OntoSum_0 as the routing scheme. Because most of the resource updates are instance-level updates, OntoSum_0 and ShortCut do not need to change their neighborhood too much, and consequently they do not see much overhead. If there are more ontology-level updates, it is like inserting more new nodes, and the result would be similar to the result shown in Figure 5.17. Figure 5.20 illustrates the overhead composition of OntoSum_1 and Onto_Sum_0. Because resource changes do not affect Short-cut and random-walk much, we do not plot their results.

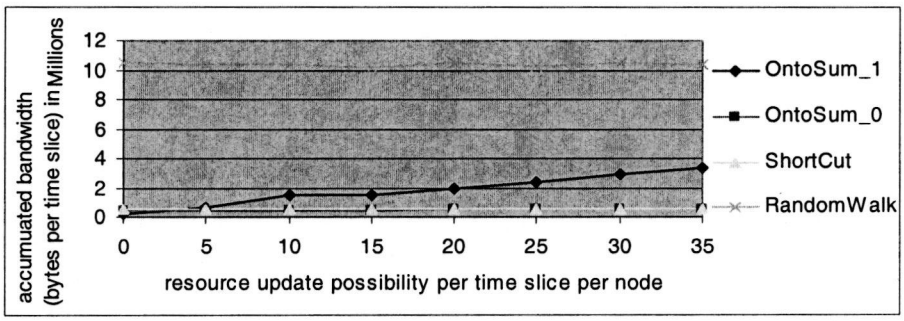
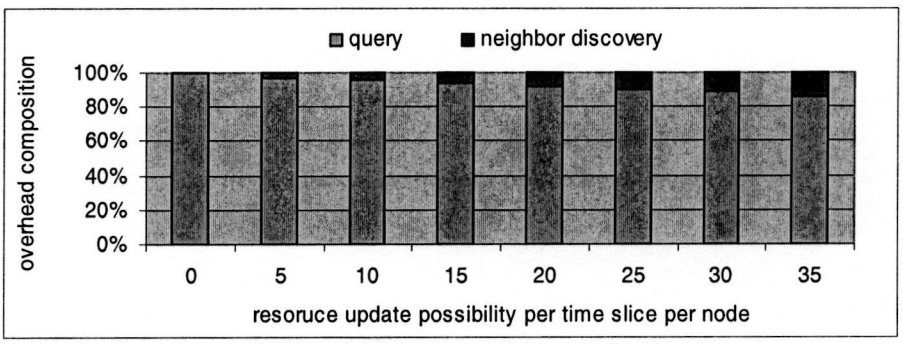
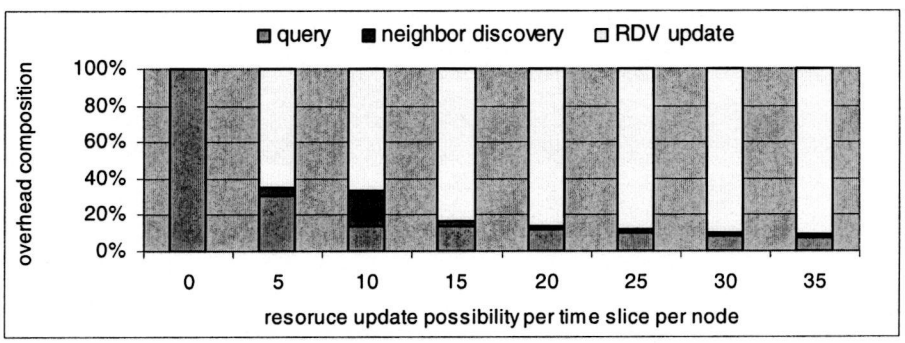


Figure 5.19 System overhead vs. resource update rate



(a) OntoSum_0 overhead composition



(b) OntoSum_1 overhead composition

Figure 5.20 Overhead composition vs. resource update rate

We should understand that the overhead is application dependent. It depends on the quantity of resources, the routing table update rate, as well as factors like the compression rate of RDV routing table. In our experiments our resource size is between 10 to 200 instances per node. We set OntoSum's RDV routing *radius* as *1*, which means that inter-cluster routing tables are only exchanged between direct neighbors. We do not compress the RDV table during transferring. If a system has more resources than this configuration, the system will see more overhead and vice versa. We can see there is a tradeoff between query efficiency and indexing overhead. The application should choose a suitable OntoSum version for its particular purpose.

5.6.2.4 Evaluation of RDV routing

Since RDV can be used as an independent routing algorithm to deal with semantic queries, we perform a separate simulation experiment to better examine its performance. In this part of the evaluation, nodes share the same ontology definition but with different instances indexed by the RDV table. Three versions of RDV with *radius* range from *1* to *3* are tested together with a reference random-walk routing algorithm. We evaluated the performance of the RDV routing algorithm under both static and moderately dynamic environments. Dynamics includes resource change and nodes joining/leaving the system. For resource updates, we only add new resources, and do not consider deleting resources for this simulation.

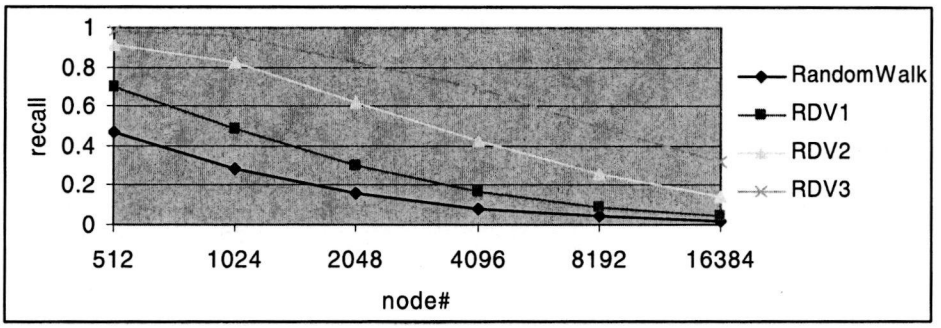
Table 5.5 Parameters used in the RDV simulation

Parameter	**Range and default value**
network size	2^9~2^{14} default: 1000
average node degree	10
TTL	1~7 default 5
walkers	3
resources per node	10
number of hash functions used for RDV filter (k)	4
RDV filter size (m)	5KB~2MB
RDV table radius	1~6
die/leave probability per time slice per node	0~27%, default:3%
resource change probability per time slice per node	0~27%, default: 3%
query probability per time slice per node	5%
RDVT update frequency	every 5 time slice

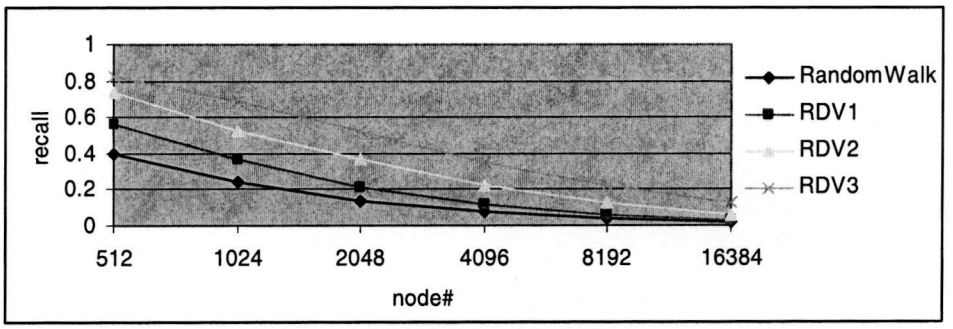
General performance

Figure 5.21 illustrates the query recall rate in different sized networks. We see that the RDV routing algorithm dramatically outperforms random-walk routing in both static and dynamic environments, and RDV with a larger radius performs better compared with RDV with a smaller radius. We notice that

dynamics will cause a decrease in the RDV recall rate, especially when the radius is large. Fortunately, the drop of the quality is relatively small in moderately dynamic environments. Figure 5.22 shows the relationship between recall rate and query TTL. It is clear that RDV achieves better recall rate with fewer routing hops. When TTL is smaller than 3, all three versions of RDV get a similar recall rate. This tells us an obvious rule: do not set a *radius* beyond TTL, because it is useless to index resource information beyond the searching scope.

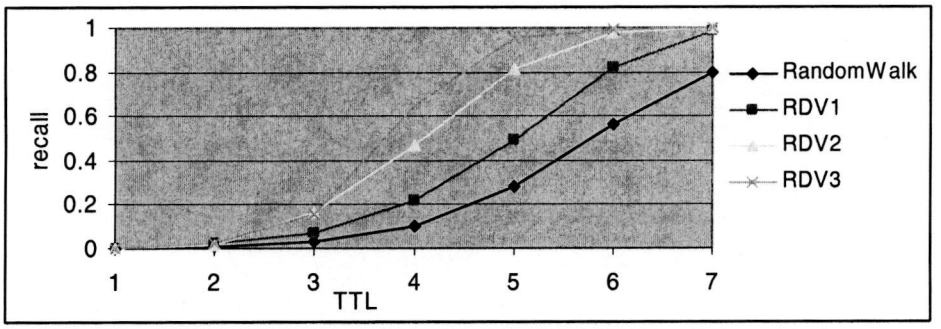


(a) Static environment

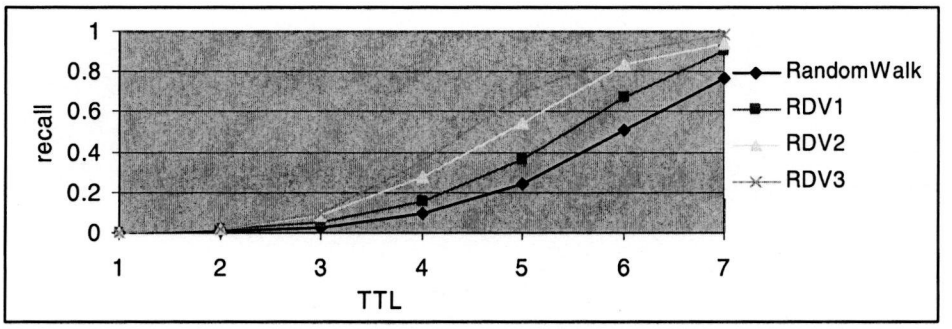


(b) Dynamic environment

Figure 5.21 Recall vs. network size



(a) Static environment



(b) Dynamic environment

Figure 5.22 Recall vs. TTL

Adaptability to dynamics and overhead

Experiments in this section evaluate how dynamics affects RDV's routing performance. The experiments are performed in a network with 1000 nodes, and results are collected after finding 100,000 results. It can be seen from Figure 5.23 that the performance of RDV degrades as the system becomes more dynamic, especially when the *radius* is large. When resources change or nodes join or leave the network, it takes some time to propagate the update to other nodes a few hops away. Our soft-state update mechanism increases the latency even longer. Therefore, when the system is very dynamic, the resource information in the RDV table may be out-of-date and cannot accurately route queries.

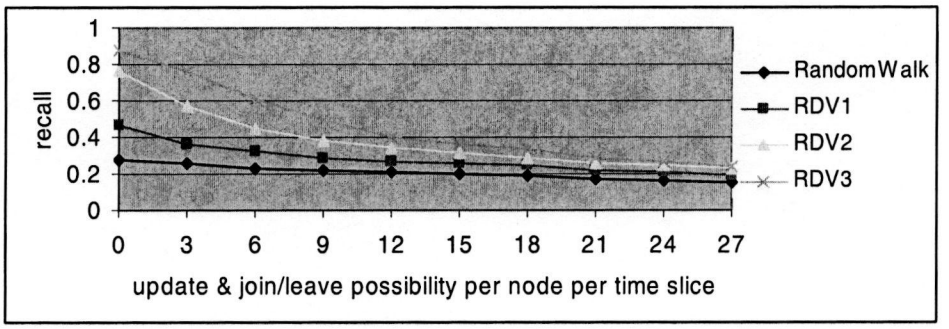


Figure 5.23 Recall under different dynamics

The overhead in terms of accumulated bandwidth is shown in Figure 5.24. The overhead includes bandwidth used for forwarding queries and propagating routing table updates. According to the figure, the overhead of RDV increases as the system becomes more dynamic, but the overall overhead of all three RDV is not high because we fix the routing table update frequency. In most cases, nodes do not need to transfer their whole RDV table to neighbors to broadcast the update, but only the hash positions related to the updated resources. Therefore, the bandwidth consumption is relatively small. The full table update overhead is higher, but it happens infrequently, and with additional compression techniques, the overhead can be lowered as well. The overhead is related to the routing table update frequency. In this experiment, we set the update frequency to 10 time-slices. When nodes update their routing table more

frequently, the routing information would be more accurate, and thus the recall would be higher, but the overhead would also be higher.

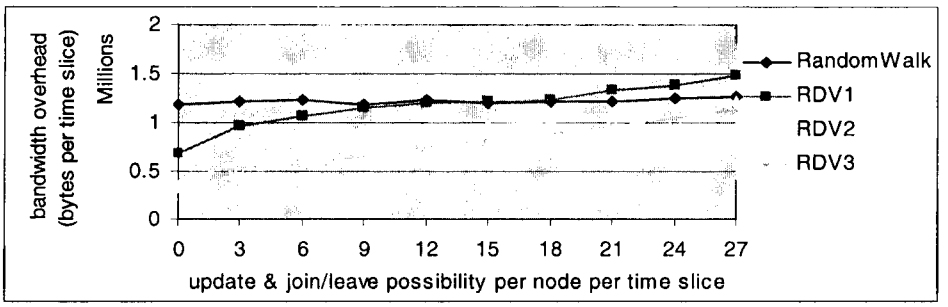
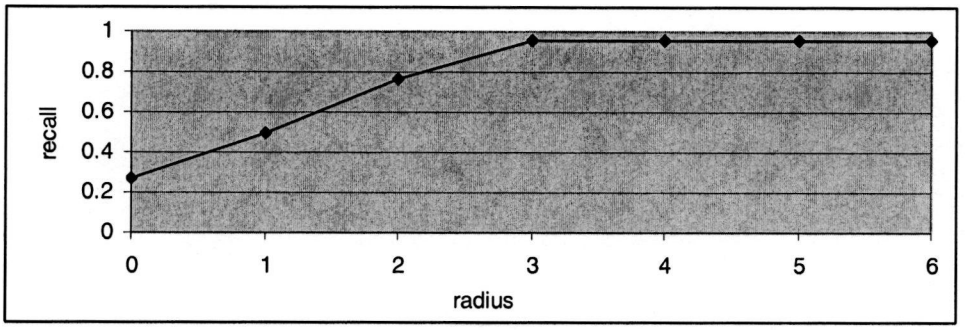


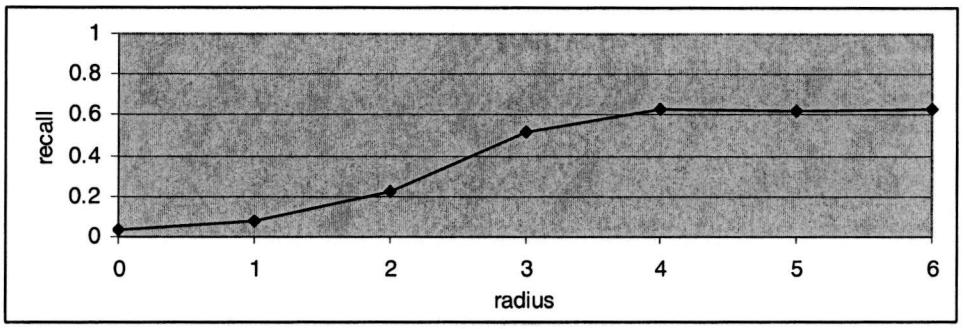
Figure 5.24 Accumulated bandwidth overhead under different dynamics

Radius

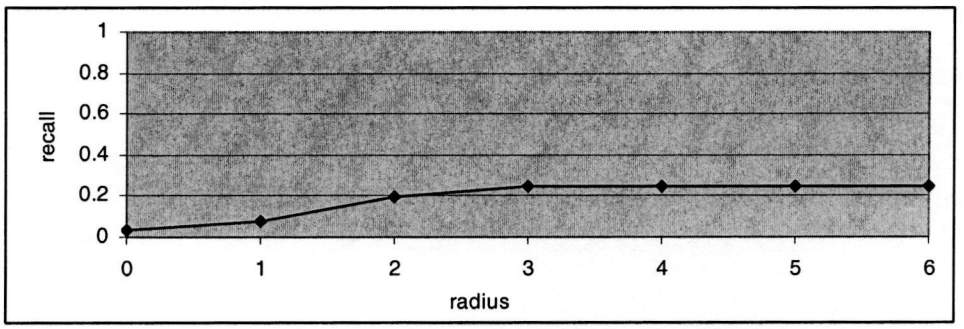
The *radius* is an important property affecting the performance of the system. It determines how far resource information can be propagated and thus how much a node learns about the network. When *radius=0*, the RDV algorithm degrades to a random-walk algorithm. When *radius=1*, the RDV routing becomes very simple: nodes only exchange their local resource index with their neighbors and they do not need to maintain the distance information; therefore, the index can be stored in a bit map instead of a numeric vector in which each entry needs several bits. Figure 5.23 (a) and (b) illustrate the influence of the *radius* on query recall in different sized networks. In this experiment, the average node degree is *10*. For both figures, initially increasing the *radius* increases nodes' knowledge of the network, thus improving the recall rate. When *radius* grows to three or four, further increasing the radius does not result in higher recall rate. As we mentioned, our network exhibits a small-world property; nodes are connected with short distances. Therefore, a small *radius* is enough for most networks to get good performance. Besides network size and environment dynamics, another concern for choosing *radius* value is the RDV filter size. When the filter reaches its capacity, increasing *radius* is a waste. One advantage of our RDV filter is that even when the filter has reached its capacity, adding resources from longer distance will not introduce more conflicts to the existing resource information, as nearer resource information has higher priority in the RDV table. Figure 5.23 (c) shows this situation. In this example, we use a small-sized RDV filter (10k) which can only hold up to *radius* = *2* resources. From the figure, we see increasing radius will not increase the recall rate, but on the other hand, neither will it add more errors.



(a) network size:1000, filter size: 100KB



(b) Network size: 10000, filter size: 1MB



(c) Network size is 10000, filter size is 30KB

Figure 5.25 Recall vs. radius under different filter size

RDV filter size

From Figure 5.25(c), we have seen that the filter size has an influence on recall rate. As mentioned, Bloom filters have false positives. The number of hash function k and the filter size m should be chosen such that the probability of a false positive is acceptable. In this experiment, we set k to 4. We found there is a tradeoff between the filter size m and the number of false positives, which is reflected by the recall rate in our experiment. We examine the influence of the filter size with respect to recall rate. Some critical parameters in this experiment are set as follows: network size: 1000, average resource number (triple number) per node: 20, RDV *radius*: 3. The results in Figure 5.26 show that increasing filter size does increase the recall rate to some extent. When the filter has enough space to hold the resource information, further increasing the size will not improve the recall. Unlike a Bloom filter's bit map, a RDV filter uses a distance vector, and each distance value is represented by 2 bits in this experiment.

Although we double the space, resources with different distance values can exploit the extra space, thus reducing false positives.

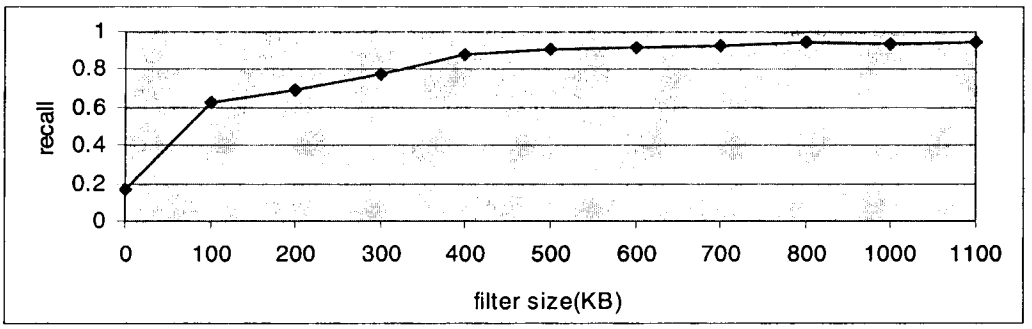


Figure 5.26 Influence of RDV filter size

5.7 Summary

In this chapter, we have presented an efficient model for sharing and searching semantically heterogeneous resources based on an unstructured P2P overlay. In particular, we have proposed a semantics-aware topology construction method to group nodes sharing similar semantics together to create small-worlds. For this purpose, we have designed an algorithm to extract a node's ontology summary and use that summary to compute the semantic similarity between nodes. With this semantic similarity defined, nodes are grouped accordingly, forming semantic virtual domains and clusters. Resource information integration and searching can be efficiently performed on top of this topology. In addition, we have proposed an intelligent query routing scheme, RDV, which uses a succinct structure to summarize each node's resource instance information. The routing scheme can route the query accurately and quickly. The proposed system was evaluated by simulation, which indicated that the system is scalable and efficient.

Chapter 6

Conclusions

This thesis focuses on resource discovery in a global-scale grid environment. We have demonstrated that it is possible to meet both the scalability and searchability challenges faced by the resource discovery problem in this target environment. Towards this end, we have designed, implemented, and evaluated distributed discovery systems that are fully decentralized, scalable to the number of users and resources involved, adaptive to heterogeneous resource representations, and capable of handling complex queries.

6.1 Contributions

The main contributions of this thesis are as follows:

- **Expressive semantic model for efficient resource discovery**

  An important characteristic of our search system is the ability to perform effective resource retrieval by taking into account the semantic properties of what is being searched for. We have realized this intelligent search system by adding the necessary semantic building blocks into the system. We examined the employment of ontological domain knowledge to assist in the search process. With this knowledge, queries can be properly interpreted according to their meanings in a specific domain; the inherent relationships among the concepts are also considered. The semantic model includes (1) an expressive ontological representation to encode the resource metadata, (2) an effective mapping formalism along with corresponding reasoning algorithms to integrate heterogeneous ontological representations, and (3) a comprehensive semantic query evaluation scheme to process complex SQL-like queries.

- **Effective ontological topology adaptation schemes**

  We reorganized the network topology according to participating nodes' ontological properties, so that semantically related nodes are topologically close and queries can be efficiently propagated. In our system, we always try to optimize the network topology according to this principle. We have proposed a hierarchical ontology model – the ontology directory – to facilitate partitioning the large unorganized search space into multiple well-organized sub-spaces, which we call semantic virtual organizations. In addition, in the OntoSum searching framework, we have designed an algorithm to compute the semantic similarity between nodes. This algorithm allows nodes to discover semantically related neighbors according to their semantic similarity to construct small-worlds.

- **Scalable semantic resource indexing**

  Indexing is at the heart of our search system, and determines searching scalability and efficiency. We have indexed ontological knowledge to add semantics to the metadata to support reasoning and improve query recall and precision. For different application requirements and network topologies,

we have proposed two different indexing infrastructures: GONID and OntoSum, based on structured DHT and unstructured P2P overlays, respectively. GONID builds complex query facilities on top of DHTs, while maintaining the scalability of the DHT infrastructure. A key advantage of the ontological indexing scheme in GONID is its ability to index at different granularities to adapt to different system properties. In OntoSum, nodes are loosely structured, forming small-world networks. Each node keeps track of a set of neighbors and organizes these neighbors into a multi-resolution neighborhood according to their semantic similarities. A query is matched against the relevant nodes in the neighborhood. This architecture combines the efficiency and scalability of structured overlay networks with the connection flexibility of unstructured networks.

- **Efficient complex query resolution**

  Based on our semantic indexing and ontology mappings, we can efficiently process SQL-like queries, particularly SPARQL queries, in our current implementation. A query in terms of relations in the user's local ontology is translated into sub-queries using semantic mapping axioms. Then each of the sub-queries can be executed at different sources in parallel. A complex query can be evaluated by performing relational operations such as select, project, and join on combinations of the indexing triple pattern. The query request is navigated through the network in a reasonable pattern sequence. Each pattern can be mapped to an indexing node where the intermediate result can be obtained. The intermediate results, together with the rest of the query patterns, are passed on to the next node. In this way, a complex query can be processed though this chain.

- **Distributed load balancing**

  We studied the problem of load imbalance resulting from skewed access distribution, and proposed an effective load balancing solution that takes peer heterogeneity and access popularity into account to determine the load distribution. Our algorithm manages to achieve load balancing by dynamically balancing the query routing load and query answering load.

6.2 Limitations and future work

Important problems in large-scale resource discovery remain to be solved. We identify several limitations of our work and research directions for future work. Some research directions are a natural continuation of this thesis; others are more general problems in resource discovery.

Our search system focuses on relatively static resource information; however, sometimes we need to consider very dynamic information. For example, in computational grids, a scheduler may need to find available computational resources with both relatively static requirements, such as system architecture, OS version, and access policy, and more dynamic requirements, such as instantaneous load and predictions of future availability. In order to deal with dynamic information, we need to add monitoring components to our system. Resource monitoring is important for a variety of tasks, such as fault detection, performance analysis, performance tuning, performance prediction, and scheduling. Unlike our more static forms of metadata, most monitoring data may go stale quickly, and are typically updated more frequently than they are read. Our work is optimized for query and not update, therefore it is

potentially unsuitable for dynamic information storage. The data management system must minimize the elapsed time associated with storage and retrieval. There have been many attempts [110, 115, 92, 114, 24] to address this problem, however, it is still an open problem in large scale grids. Our existing architectures and algorithms would be helpful to make the monitoring system scalable across wide-area networks and adaptable to a wide range of heterogeneous resources.

Mapping and integrating heterogeneous information sources is an integral part of a search system. To find semantically related nodes to map ontologies in the GONID system, users use their T-Box ontology as keys to find other nodes having overlapping ontologies. They can then pick candidates from those nodes to do the mapping. This process requires lots of human work. In the future, we plan to make it more automatic, and provide more hints or suggestions to users to help them make these decisions. In OntoSum, finding semantically related neighbors is accomplished according to their semantic similarity, which is defined by comparing the extended Ontology Signature Set. This simple similarity can be improved by considering other factors such as nodes' ontological structure, definitions of concepts, and instances of classes.

In our current system, query results are returned to requesters without using any ranking mechanisms. There are many techniques for ranking entities on the Web, for example PageRank [13] and HITS [54], on XML documents [45] [22], and on the Semantic Web [97]. However, these techniques cannot be used directly to rank our search results because of the different problem nature. We plan to investigate the result-ranking problem, so that query results can be ordered based on relevance and importance for users. The ranking problem involves a rich blend of semantic and information-theoretic techniques. The ordering of the results should be able to vary according to user need.

Our work aims at processing ontological metadata in the format of OWL/RDF. We plan to extend it to metadata in other rich formats, for example multi-attribute multimedia data. There are many other potential areas that can benefit from our semantics-based search design. One such area is Information Retrieval which includes keyword-based search for text documents, as well as semi-structured XML and RDF documents. The Web Service-based distributed computational model can also benefit from semantic search over P2P systems to locate Web Services and compose them dynamically to provide more complex services. These designs can also be used for providing discovery services for publish/subscribe systems.

Security is another important issue. For sensitive information, it is necessary to control the access to resources or resource metadata. In GONID, resource metadata information is indexed on distributed overlay nodes. Unlike local repositories, it is very difficult to enforce access restrictions on distributed nodes. Distributed P2P-based access control is another future research topic. A secure system should be able to index information with different access restrictions. The owners should be able to specify fine-grained restrictions on who can access their resources, and which users can access what part of the data.

Finally, it requires a tremendous amount of effort to evaluate a large scale grid resource discovery system in a realistic setting. Such an evaluation necessitates not only a large number of nodes but also a huge amount of realistic and representative semantic content. So far, we have evaluated our design by simulating a large number of networked nodes and experimenting with artificial resource data. These evaluations have been very constrained. We also implemented a prototype that we deployed and tested on a small network. However, this environment is obviously quite limited. Our long-term goal is to release our results to the open-source community and foster a user community, eventually letting the users be the final judge of our systems.

Bibliography

[1] L. Adamic, B. Huberman, R. Lukose, A. Puniyani: "Search in power law networks". Physical Review, 2001.

[2] G. Antoniou, F. V. Harmelen, "A Semantic Web Primer", MIT Press, 2004.

[3] M. S. Artigas, P. G. López, J. P. Ahulló, and A. F. G. Skarmeta, "Cyclone: A Novel Design Schema for Hierarchical DHTs", in Proceedings of the IEEE P2P 2005.

[4] F. Banaei-Kashani, and C. Shahabi. "Criticality-based analysis and design of unstructured peer-to-peer networks as complex systems". Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 351-358, 2003.

[5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, P. F. Patel-Schneider: "The Description Logic Handbook: Theory, Implementation, Applications." Cambridge University Press, Cambridge, UK, 2003.

[6] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. Open information extraction from the web. In Twentieth International Joint Conference on Artificial Intelligence, 2007.

[7] A. L. Barabási, "Linked: How Everything is Connected to Everything Else and What It Means for Business, Science, and Everyday Life." New York, Plume, 2003.

[8] T. Berners-Lee, J. Hendler and O. Lassila (2001) "The semantic web", Scientific American, 284(5):34–43, 2001.

[9] B. Bloom. "Space/time tradeoffs in hash coding with allowable errors". Communications of the ACM, pages 13(7):422-426, July 1970.

[10] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie and J. Simeon. "XQuery 1.0: An XML Query Language." W3C Working Draft, 2003.

[11] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. "Beyond Bloom filters: From approximate membership checks to approximate state machines", in Proceeding of SIGCOMM, 2006.

[12] T. Brasethvik and J. A. Gulla. "Natural language analysis for semantic document modeling." Data & knowledge Engineering, 2001.

[13] S. Brin, L. Page, "The anatomy of a large-scale hypertextual Web search engine." In Proceeding Of WWW1998, pages 107-117. Brisbane, Australia, 1998.

[14] J. Byers, J. Considine, and M. Mitzenmacher, "Simple load balancing for distributed hash tables," in Proceedings of 2nd International Workshop on Peer-to-Peer Systems, pp. 80–87. 2003.

[15] M. Cai, M. Frank, J. Chen and P. Szekely, "MAAN: A Multi-Attribute Addressable Network for Grid Information Services." The 4th International Workshop on Grid Computing, 2003.

[16] M. Cai, M. Frank, "RDFPeers: A scalable distributed RDF repository based on a structured peer-to-peer network", in proc of WWW conference, NewYork, USA, pp 650-657, May 2004.

[17] H. Casanova, "Distributed Computing Research Issues in Grid", Computing, typescript, Univ. of California, San Diego, 2002.

[18] S. Castano, A. Ferrara, S. Montanelli, and D. Zucchelli. "Helios: a general framework for ontology-based knowledge sharing and evolution in P2P systems." In Proceeding of IEEE DEXA WEBS 2003 Workshop, Prague, Czech Republic, September 2003.

[19] A. Castano, S. Ferrara, S. Montanelli, E. Pagani, G. Rossi, "Ontology addressable contents in P2P networks." in Proceedings of the WWW'03 Workshop on Semantics in Peer-to-Peer and Grid Computing, 2003.

[20] S. J. Chapin, D. Katramatos, J. Karpovich, and A. Grimshaw. "The Legion resource management system." In Proceedings of the 5th Workshop on Job Scheduling Strategies for Parallel Processing, San Juan, Puerto Rico, 1999.

[21] Y. Chawathe, S. Ratnasam, L. Breslau, N. Lanhan, S. Shenker, "Making Gnutella-like P2P Systems Scalable", In Proceedings of ACM SIGCOMM'03, 2003.

[22] S. Cohen, J. Mamou, Y. Kanza, Y. Sagiv, "XSEarch: A Semantic Search Engine for XML", in Proceedings of the VLDB, 2003.

[23] F. Cuenca-Acuna, C. Peery, R. Martin, T. Nguyen, "PlanetP: using gossiping to build content addressable peer-to-peer information sharing communities," in Proceedings of the 12th International Symposium on High Performance Distributed Computing (HPDC), 2002.

[24] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. "Grid Information Services for Distributed Resource Sharing." In Proceedings 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10'01), pages 181–194, 2001.

[25] S. Deerwester, S. T. Dumais, G. W. Furnas, K. L. Thomas, R. Harshman, "Indexing by latent semantic analysis," Journal of the American Society for Information Science, pp. 391-407, 1990.

[26] X. Dong and A. Halevy. "A platform for personal information management and integration." In Proceedings of the CIDR, 2005.

[27] L. Fan, P. Cao, J. Almeida, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", IEEE/ACM Transactions on Networking 8 (3): 281–293, 2000.

[28] C. Fellbaum. "WordNet: An Electronic Lexical Database," In Fellbaum, Christiane, MIT Press, pp. 1-19, 1998.

[29] S. Fitzgerald, I. Foster, C. Kesselman, G. von Laszewski, W. Smith, and S. Tuecke. "A Directory Service for Configuring High-Performance Distributed Computations." In Proceedings of the 6th IEEE Symp. on High-Performance Distributed Computing, pages 365–375. IEEE Computer Society, 1997.

[30] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke. "The Physiology of the Grid. An Open Grid Services Architecture for Distributed Systems Integration." Technical report, Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.

[31] I. Foster and C. Kesselman. "The Grid: Bluepring for a Future Computing Infrastructure," chapter "Globus: A Toolkit-based Grid Architecture," pages 259–278. Morgan-Kaufmann, 1999.

[32] I. Foster, C. Kesselman, S. Tuecke "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International Journal of Supercomputer Applications and High Performance Computing, 2001.

[33] M. Freedman, D. Mazieres, "Sloppy hashing and self-organizing clusters" in Proceedings of 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), Berkeley, CA, USA, 2003.

[34] G. W. Furnas, S. Deerwester, S. T. Dumais, T. K. Landauer, R. A. Harshman, L. A. Streeter, K. E. Lochbaum, "Information retrieval using a singular value decomposition model of latent semantic structure", in Proceedings of 11th Annual Int'l ACM SIGIR Conference on Research and Development in Information Retrieval, 1988.

[35] P. Ganesan, K. Gummadi and H. Garcia-Molina, "Canon in G major designing DHTs with hierarchical structure", in Proceedings of the 24th ICDCS, Tokyo,2004.

[36] B. Godfrey, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in dynamic structured P2P systems," in Proceedings of the 23rd Conference of the IEEE Communications Society Infocom, 2004.

[37] V. Gopalakrishnan, B. Silaghi, B. Bhattacharjee, and P. Keleher. "Adaptive replication in peer-to-peer systems." In Proceedings of 24th International Conference on Distributed Computing Systems (ICDCS), 2004.

[38] W. A. Gruver, and J. C. Boudreaux "Intelligent Manufacturing: programming environments for CIM," Springer-Verlag, London, 1993.

[39] T. R. Gruber, "Principles for the Design of Ontologies Used for Knowledge Sharing." International Journal Human-Computer Studies, 43(5-6):907-928, 1995.

[40] M. Gubanov and P. A. Berstein. "Structured text search and comparison using automatically extracted schema." In webDB, 2006.

[41] V. Haarslev and R. Moeller. "Racer system description." In Proceeding of the Joint Conf. on Automated Reasoning. Volume 2083 of Lecture Notes in Artificial Intelligence, pages 701-705, 2001.

[42] A. Y. Halevy, Z. Ives, P. Mork and I. Tatarinov. "Piazza: Data management infrastructure for semantic web applications." In Proceeding of the Twelfth International conference on World Wide Web, Budapest, Hungary, 2003.

[43] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. "SkipNet:A Scalable Overlay Network with Practical Locality Properties". In Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS '03), 2003.

[44] I. Horrocks. "FaCT and iFaCT." In Description Logics, 1999.

[45] V. Hristidis, Y. Papakonstantinou, A. Balmin, "Keyword Proximity Search on XML Graphs." In Proceedings of the IEEE ICDE, 2003.

[46] A. Iamnitchi, I. Foster, "On Fully Decentralized Resource Discovery in Grid Environments," in Proceeding of the 2nd IEEE/ACM International Workshop on Grid Computing 2001, Denver, 2001.

[47] A. Iamnitchi, M. Ripeanu, and I. Foster. Small-world filesharing communities. In Proceedings of the Infocom, Hong Kong, China, 2004.

[48] M. JARKR, AND J. KOCH, "Query optimization in database systems." ACM Comput. Surv. 1984.

[49] J. Jiang and D. Conrath, "Semantic Similarity Based on Corpus Statistics and Lexical Taxonomy," in Proceeding of the Int'l Conf. Computational Linguistics (ROCLING X), 1997.

[50] D. Karger and M. Ruhl, "Simple efficient load balancing algorithms for peer-to-peer systems," in Proceedings of 16th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 36–43. 2004.

[51] A. R. Karthik, K. Lakshminarayanan, S. Surana, R. Karp, and I. Stoica, "Load balancing in structured P2P systems," in Proceedings of 2ndInternational Workshop on Peer-to-Peer Systems, pp. 68–79, 2003.

[52] D. Karger and M. Ruhl. "Simple Efficient Load Balancing Algorithms for Peer-to-Peer Systems." In Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithms and Architectures, 2004.

[53] J. KING. "QUIST: A system for semantic query optimization in relational databases." In Proceedings 7th International Conference on Very Large Data Bases (Cannes, France). VLDB Endowment, 510–517. 1981.

[54] J. Kleinberg, "Athorative sources in a hyperlinked environment." J. ACM, 48:604-632, 1999.

[55] J. Kleinberg, "Navigation in a small world," Nature, no. 406, p. 845, 2000.

[56] G. Kokkinidis, L. Sidirourgos, and V. Christophides. "Query Processing in RDF/S-Based P2P Database Systems." In Steffen Staab and Heiner Stuckenschmidt, editors, Semantic Web and Peer-to-Peer. Springer-Verlag, Berlin Heidelberg, 2006.

[57] O. Lassila and Ralph R. Swick, "W3C Resource Description framework (RDF) Model and Syntax Specification", World Wide Web Consortium, 1999.

[58] G. V. Laszewski and I. Foster. Usage of LDAP in Globus.
http://www.globus.org/mds/globus in ldap.html, 2002.

[59] J. Ledlie and M. Seltzer, "Distributed, secure load balancing with skew, heterogeneity, and churn," in Proceedings of the 24th IEEE Infocom, 2005.

[60] J. Lee, M. Kim, and Y. Lee, "Information Retrieval Based on Conceptual Distance in IS-A Hierarchies," J. Documentation, vol. 49, pp. 188-207, 1993.

[61] J. Li, S. Vuong, "An Ontological Framework for Large-Scale Grid Resource Discovery", in Proceedings of the IEEE Symposium on Computers and Communications (ISCC'07), Aveiro, Portugal, July 2007.

[62] J. Li, S. Vuong, "OntSum: A Semantic Query Routing Scheme in P2P Networks Based on Concise Ontology Indexing", in Proceedings of the 21st IEEE International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada, May 2007.

[63] J. Li, I. Radu, S. Vuong, "GODIS: Ontology-Based Resource Discovery and Integration in Grids", in Proceedings of the 18th IASTED International Conference: Parallel and Distributed Computing Systems, Dallas, USA, November 2006.

[64] J. Li, S. Vuong, "Grid Resource Discovery Based on Semantic P2P Communities", in Proceedings of the 21st Annual ACM Symposium on Applied Computing, (SAC-06) Dijon, France, April 2006.

[65] J. Li, S. Vuong, "A Semantics-based Routing Scheme for Grid Resource Discovery", in Proceedings of the 1st IEEE International Conference on e-Science and Grid Computing, (eScience2005), Melbourne, Australia, December 2005.

[66] J. Li, S. Vuong, "Semantic Overlay Network for Grid Resource Discovery", in Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing (Grid05), Seattle, USA, November 2005.

[67] J. Li, S. Vuong, "A Scalable Semantic Routing Architecture for Grid Resource Discovery", in Proceedings of the the 11th IEEE International Conference on Parallel and Distributed Systems (ICPADS05), Fukuoka, Japan, July 2005.

[68] J. Li, S. Vuong, "Ontology-Based Clustering and Routing in Peer-to-Peer Networks", in Proceedings of the 6th International Conference on Parallel and Distributed Computing, Applications and Technologies, Dalian, China, December 2005.

[69] J. Li, S. Vuong, "Grid Resource Discovery Using Semantic Communities", in Proceedings of the 4th International Conference on Grid and Cooperative Computing, Beijing, China, November 2005.

[70] J. Li, B. Cheung, S. Vuong, "A Scheme for Balancing Heterogeneous Request Load in DHT-based P2P Systems", in Proceedings of the Fourth International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks (QShine 2007), Vancouver, British Columbia, August 2007.

[71] D. Lin. "An information-theoretic definition of similarity." In Proceeding of the 15th International Conf. on Machine Learning, pages 296–304. San Francisco, CA, 1998.

[72] A. Löser, S. Staab, and C. Tempich, "Semantic Methods for P2P Query Routing," MATES, pp. 15-26, 2005.

[73] C. Lv, P. Cao, E. Cohen, K. Li, S. Shenker, "Search and replication in unstructured peer-to-peer networks". In Proceeding of the ACM, SIGMETRICS, 2002.

[74] G. Manku. "Balanced binary trees for ID management and load balance in distributed hash tables." In Proceedings of Twenty-Third Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC 2004), 2004.

[75] B. McBride. "Jena: Implementing the RDF model and syntax specification." Technical report, Hewlett Packard Laboratories, Bristol, UK, 2000.

[76] G. Miller, "WordNet: A lexical database for English", Communications of the ACM, vol. 38, no. 11, 1995.

[77] S. Milgram, "The small world problem," Psychology Today, vol. 67, no. 1, 1967.

[78] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross and K. J. Miller 1990, Introduction to WordNet: an on-line lexical database." In: International Journal of Lexicography 3 (4), pp. 235 – 244, 1990.

[79] G. A. Miller, R. Beckwith, C. Fellbaum, D. Gross and K. J. Miller. "Introduction to WordNet: an on-line lexical database." International Journal of Lexicography, 3(4):235-312. 1993.

[80] M. Naor and U. Wieder. "Novel architectures for P2P applications: the continuous-discrete approach." In Proceedings of the Fifteenth ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2003), June 2003.

[81] B. Nebel, "Artificial intelligence: A computational perspective." In G. Brewka, Editor, Principles of Knowledge Representation (2nd ed.), Studies in Logic, Language and Information, CSLI Publications, Stanford, 1996.

[82] W. Nejdl, B. Wolf, C. Qu, S. Decker, M.SIntek, A. Naeve, M. Nilsson, M. Palmer and T. Risch. "Edutella: A P2P Networking Infrastructure Based on RDF." In Proceedings of the WWW2002, May 7-11, Honolulu, Hawaii, USA. 2002.

[83] W. Nejdl, W. Siberski an M. Sintek, "Design Issues and Challenges for RDF an schema-based peer-to-peer systems." In Proceedings of the ACM SIGMOD Record 32(3):41-46, 2003.

[84] W.S. Ng, B.C. Ooi, K.L. Tan and A. Zhou . "PeerDB: A p2p based system for distributed data sharing." In Proceedings of the 19th International Conference on Data Engineering (ICDE), Bangalore, India, March 2003.

[85] N. F. Noy, Sintek, M., Decker, S., Crubézy, M., Fergerson, R.W., & Musen, M.A. "Creating Semantic Web Contents with Protégé-2000." In Proceedings of the IEEE Intelligent Systems 16(2), 60-71. 2001.

[86] J. Perez, M. Arenas, C. Gutierrez: "The semantics and complexity of SPARQL." In Proceedings of the 5th International Semantic Web Conference, 2006.

[87] R. Raman, M. Livny, M. Solomon, "Matchmaking: Distributed Resource Management for High Throughput Computing". In Proceeding of IEEE Intel. Symp. On High Performance Distributed Computing, Chicago, USA, 1998.

[88] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. "A Scalable Content-Addressable Network," In Proceedings of the ACM SIGCOMM, August, pp. 161-172. 2001.

[89] M. A. Rodriguez, M. J. Egenhofer, "Determining Semantic Similarity Among Entity Classes from Different Ontologies". IEEE Transactions on Knowledge and Data Engineering, VOL. 15, NO. 2, MARCH/APRIL, 2003.

[90] A. Rowstron and P. Druschel. "Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems," In Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms, Middleware, 2001.

[91] W. Smith and D. Gunter. "Simple LDAP Schemas for Grid Monitoring." Global Grid Forum, GWD-Perf-13-1, June 2001.

[92] L. R. Randy, J. S. Vetter, H. Simitci, and D. A. Reed, "Autopilot: Adaptive Control of Distributed Applications," In Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, 1998.

[93] P. Reynolds and A. Vahdat. "Efficient Peer-to-Peer Keyword Searching". In Proceedings of the ACM/IFIP/USENIX Middleware, 2003.

[94] G. Salton, A. Wong and C.S. Yang. "A vector space model for automatic indexing. " Communications of the ACM 18(11):613-620. November, 1975.

[95] G. Salton and C. Buckley "Term-weighting approaches in automatic text retrieval." Information Processing & Management 24(5): 513–523. 1988.

[96] T. Sellis, "Multiple-Query Optimization", ACM Transactions on Database Systems, 12(1), pp. 23-52, June 1990.

[97] A. Sheth, Aleman-Meza, B., Arpinar, I. B., Halaschek, C., Ramakrishnan, C., Bertram, C., Warke, Y., Avant, D., Arpinar, F. S., Anyanwu, K., Kochut, K. "Semantic Association Identification and Knowledge Discovery for National Security Applications." Journal of Database Management, 16 (1), pp. 33-53, Jan-Mar 2005.

[98] E. Sirin and B. Parsia. "SPARQL-DL: SPARQL Query for OWL-DL." In Proceedings of the 3rd OWL Experiences and Directions Workshop, 2007.

[99] E. Sirin and B. Parsia. "Pellet: An owl dl reasoner." In Description Logics, 2004.

[100] K. Sripanidkulchai, B. Maggs, and H. Zhang. "Efficient content location using interest-based locality in peer-to-peer systems," In Proceedings of the INFOCOM'03, 2003.

[101] D. Stutzbach and R. Rejaie. "Understanding churn in peer-to-peer networks." In Proceeding of the Internet Measurement Conference (IMC), Oct. 2006.

[102] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H.Balakrishnan. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," In Proceedings of the ACM SIGCOMM, pp. 149-160. August 2001.

[103] Y. Sure, M. Erdmann, J. Angele, S. Staab, R. Studer, and D. Wenke. "Ontoedit: Collaborative ontology development for the semantic web." In Proceedings of the 1st International Semantic Web Conference, Sardinia, Italia, LNCS 2342, pages 221–235. Springer, 2002.

[104] Y. Sure, M. Erdmann, J. Angele, S. Staab, S. Studer, and D. Wenke. "OntoEdit: Collaborative ontology engineering for the semantic web. In Proceedings of the 1st International Semantic Web Conference, Italy. 2002.

[105] C. Tang and S. Dwarkadas. "Hybrid Gloablal-Local Indexing for Efficient Peer-to-Peer Information Retrieval". In Proceedings of USENIX NSDI, March 2004.

[106] C. Tang, Z. Xu, S. Dwarkadas, "Peer-to-peer information retrieval using self-organizing semantic overlay networks," in Proceedings of 2003 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications, pp. 175–186,2003.

[107] I. Tatarinov and A. Halevy. "Efficient query reformulation in peer data management systems." In Proceedings of the 2004 ACM SIGMOD International Conference on the management of data. Paris, France, 2004.

[108] I. Tatarinov, P. Mork, Z. Ives, J. Madhavan, A. Halevy, D. Suciu, N.Dalvi, X. Dong, Y. Kadiyska and G. Miklau. "The Piazza per data management project." in Proceedings of the ACM SIGMOD Record 32(3):47-52, 2003.

[109] X. Tempich, S. Staab, A. Wranik, "REMINDIN: semantic query routing in peer-to-peer networks based on social metaphors" in Proceedings of the International World Wide Web Conference (WWW), New York, USA, 2004.

[110] B. Tierney, B. Crowley, D. Gunter, M. Holding, J. Lee, M. Thompson. "A Monitoring Sensor Management System for Grid Environments." In Proceedings of the IEEE High Performance Distributed Computing conference (HPDC-9), LBNL-45260. August 2000.

[111] A. Tversky. "Features of similarity." Psychological Review, 84(4):327–352, 1977.

[112] M. Uschold and M. Gr¨uniger. "Ontologies: Principles, methods and applications." Knowledge Engineering Review, 11(2):93–155, 1996.

[113] D. Watts and S. Strogatz. "Collective dynamics of "small-world" networks." - Nature, 1998

[114] A. Waheed, W. Smith, J. George, J. Yan. "An Infrastructure for Monitoring and Management in Computational Grids." In Proceedings of the Conference on Languages, Compilers, and Runtime Systems. 2000.

[115] R. Wolski, N. Spring, J. Hayes, "The Network Weather Services: A Distributed Resource Performance Forecasting Service for Metacomputing," Future Generation Computing Systems, 1999.

[116] Z. Xu, and A. Bhuyan, "Effective Load Balancing in P2P Systems," in Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006.

[117] Z. Xu, R. Min, Y. Hu. "HIERAS: A DHT Based Hierarchical P2P Routing Algorithm." in Proceedings of the ICPP, 2003.

[118] B. Yang, H. Garcia-Molina, "Efficient search in peer-to-peer networks", in Proceeding of the CDCS'02, Vienna, Austria, July 2002.

[119] B.Yang and H.Garcia-Molina, "Designing a Super-Peer Network", in Proceeding of the 19th Int'l Conf. Data Engineering,, March 2003.

[120] W. Yeong, T. Howes, and S. Kille. "Lightweight Directory Access Protocol." IETF, RFC 1777, 1995.

[121] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. "Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing," Technical Report, UCB/CSD-01-1141, April 2000.

[122] College of American Pathologists. "SNOMED RT - Systematized Nomenclature of Medicine Reference Terminology," VERSION 1.1, USER GUIDE, 2001.

[123] DMOZ Open Directory Project. http://www.dmoz.org/

[124] Dan Brickley and R.V.Guha. "W3C Resource Description Framework (RDF) Schema Specification". http://www.w3.org/TR/1998/WD-rdf-schema/

[125] DNS related RFCs. http://www.dns.net/dnsrd/rfc/

[126] DHCP RFC2131. http://www.ietf.org/rfc/rfc2131.txt

[127] FreePastry. http://freepastry.org/FreePastry/

[128] Globus Toolkit: http://www.globus.org/toolkit/.

[129] Gnutella website. http://gnutella.wego.com/

[130] P. Hayes: RDF semantics. W3C Recommendation http://www.w3.org/TR/owl-semantics/ (2004)

[131] Jena – A Semantic Web Framework for Java. http://jena.sourceforge.net/

[132] JXTA website: http://www.sun.com/software/jxta/

[133] Open Directory RDF Dump. http://rdf.dmoz.org/

[134] OWL Web Ontology Language Overview. W3C Recommendation 10 February 2004. http://www.w3.org/TR/owl-features/

[135] RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3.org/TR/rdf-schema/

[136] SPARQL Query Language for RDF W3C Recommendation. 15 January 2008. http://www.w3.org/TR/rdf-sparql-query/

[137] T. Berners-Lee. "Semantic Web Road Map". http://www.w3.org/DesignIssues/Semantic.html

[138] The FastTrack website. http://www.fasttrack.nu/

[139] UDDI specification. http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm

[140] W3C Semantic Web Activity. http://www.w3.org/2001/sw/

[141] S. Harris, and N. Shadbolt, "SPARQL Query Processing with Conventional Relational Database Systems," WISE 2005 International Workshops, New York, NY, USA, pp. 235–244. 2005.