A scalable semantic-based resource discovery service for Grids

YEOU YANG

Master of Science Thesis
Stockholm, Sweden 2007

ICT/ ECS -2007-10

A scalable semantic-based resource discovery service for Grids

Master Thesis of
Software Engineering of Distributed System

KTH, ICT, ECS, Kista
2007

YEOU YANG
yeou@kth.se

Thesis Examiner: Vladimir Vlassov
Thesis Supervisor: Konstantin Popov

Abstract

This thesis presents a design and a prototype implementation of a scalable semantic-based resource discovery system for Grids. Our thesis work gives one possible solution for semantic web service discovery in grid environment. In this paper we describe two resource discovery architectures, two layers architecture and one layer architecture. One layer architecture is the simplified model of two layers architecture. In our design, we use OWL-S service description to describe semantic web service. Because using OWL-S service ontology enables automatic web service discovery which can fulfill a specific need within some quality constraints, without the need for human intervention. Distribute K-ary system is used to construct a scalable overlay network in order to provide low level transportation such as broadcast request message. We also use the Monitoring & Discovery System (MDS) in Globus Toolkit to manage web service description resources.

This grid system provides six functionalities: 1. It provides registration function based on MDS aggregator framework. It can register three kinds of customization resources to the MDS Index Service, the content of OWL-S web service description, the URI of OWL-S service description and DKS node reference. 2. It provides download function which downloads registered information from MDS Index Service and Internet, and then save this information to the local disk. 3. It also implements the matchmaking algorithm to decide the matchmaking degree of two semantic web services description. 4. It uses DKS broadcast feature to query the P2P overlay network in order to complete resource discovery. 5. It implements subscription/notification mechanism based on WS-Topics specification of GT4 WS-Notification. 6. It gives an automatic Index Service query function which user doesn't need to type command to communicate with MDS. User can configure and run our system through our GUI. Before executing query over the network, user generates his OWL-S web service description which defines what kind of web service he wants to find and then submit it to our system. The system can search the whole overlay network and run the matchmaking and return results to the requester.

Key words: Globus Toolkit 4, structured overlay network, Grid computing, broadcast, MDS4, index service, matchmaking, OWL-S

Acknowledgements

I am very grateful for the opportunity given to me by Swedish Institute of Computer Science in Kista, Stockholm to write my master thesis at their Distributed System Lab. My gratitude goes to Researcher Mr. Konstantin Popov from Swedish Institute of Computer Science (SICS), who was the supervisor, Professor Vladimir Vlassov from at the Royal Institute of Technology who examiner for this Master Thesis. They guided me and provided invaluable comments on my work. Thank you for your endless patience, for your guidance, and advice along the way! I also thank for Systems Administrator Mikael Nehlsen at SICS. He gave me a lot of help with using the facility at SICS.

Table of Content

1 Introduction

1.1 Thesis goal

The goal of the thesis is to carry research on scalable semantic-based resource discovery for Grids and design and develop a grid resource discovery system built on top of overlay peer-to-peer network. This thesis work should give a resource discovery architecture in Grid environment and implement this architecture by using of Globus Toolkit 4.0.4, peer-to-peer middleware Distributed K-ary (DKS). This system should be a scalable, fault tolerant and can fulfill registration, downloading, matchmaking, index query and notification functions. The resource discovery mechanism should be able to broadcast the client’ s request, compare the request with the advertised service descriptions and then return a set of resources.

1.2 Structure of the Thesis

The second chapter gives a brief background of Grid computing, P2P computing, Globus toolkit 4, semantics, and the relationship between OGSA, WSRF and GT4. The third chapter summarized related works including tools and algorithm for semantic-based resource discovery. The fourth chapter is mainly about a primary design to realize all proposed features of scalable semantic based resource discovery. The fifth chapter describes the prototype implemented. The sixth chapter focuses on time anatomy of register, download, query and overview of different phases. The seventh chapter is about conclusion of the thesis work and future work. In the end are list of abbreviations, reference and appendix.

2 Background

2.1 Grid computing

”Grid computing enables the virtualization of distributed computing and data resources such as processing, network bandwidth and storage capacity to create a single system image, granting users and applications seamless access to vast IT capabilities. [1]” For example, companies are using grid computing to accelerate the pace of drug development, process complex financial models, and animate movies. Linking geographically dispersed computer systems can lead to staggering gains in computing power, speed, and productivity. Without Grid computing, an organization is stuck with using only the resources it has direct control over. Using Grid computing, resources from several different organizations are dynamically pooled into virtual organizations (or VO) [2] to solve specific problems.

In order to assure interoperability on heterogeneous systems so that different types of resources can communicate and share information, the Open Grid Services Architecture (OGSA) describes architecture for a service-oriented grid computing environment for business and scientific use. OGSA has been adopted as a grid architecture by a number of grid projects including the Globus Alliance [3]. Globus Toolkit [4], a software toolkit used for building grids, is being developed by the Globus Alliance and many others all over the world. In our thesis work we use Globus Toolkit 4 as grid system development kit.

Grid computing can be used in a variety of ways to address various kinds of application requirements. Often, grids are categorized by the type of solutions that they best address. The three primary types are *Computational grid, Scavenging grid* and *Data grid* [5]. A *computational grid* is focused on setting aside resources specifically for computing power. In this type of grid, most of the machines are high-performance servers. A *scavenging grid* is most commonly used with large numbers of desktop machines. Machines are scavenged for available CPU cycles and other resources. A *data grid* is responsible for housing and providing access to data across multiple organizations. Users are not concerned with where the data is located as long as they have access to the data. There are no hard boundaries between these grid types and often grids may be a combination of two or more of these. Another common distributed computing model that is often associated with or confused with grid computing is peer-to-peer computing. We will discuss peer-to-peer computing in the next section.

2.2 Peer-to-Peer Computing

So far, there is no agreement on a succinct definition of P2P. Because this new computing model is rapidly evolving, it is healthy and desirable not to be locked down by a rigid definition. In book [6], the Intel P2P working group defines definition of P2P computing,”Peer-to-peer computing is a network-based computing model for applications where computers share resources via direct exchanges between the participating computers." Alex Weytsel of Aberdeen defines P2P as "the use of devices on the internet periphery in a succinct capacity" [7]. Clay Shirky of O'Reilly and Associate uses the following definition: "P2P is a class of applications that takes advantage of resources – storage, cycles, content, human presence – available at the edges of the Internet. Because accessing these decentralized resources means operating in an environment of unstable connectivity and unpredictable IP addresses, P2P nodes must operate outside the DNS system and have significant or total autonomy from central servers" [8].

The P2P framework allows peers to interact with each other directly which makes computing environment becomes decentralized. On the Internet, Napster [9] and Gnutella [10] are examples of this kind of peer-to-peer software. These applications deal with sharing different types of files and textual exchanges among the participants. Other projects entail other type of P2P computing. Computer on the net can share more than copies of file or space of message text. They can share their processing power by working together to process data and solve computational tasks of great complexity and magnitude. This is called "cycle sharing". Cycle sharing is used to share CPU resources across a network so that all machines function as one large supercomputer. A good example of cycle sharing is SETI@HOME project (http://setiathome.ssl.berkeley.edu/). 2.4 million users come from over 200 countries contribute their time and internet connected computer to help scientific experiment in the Search for Extraterrestrial Intelligence. This project involves a high degree of collaboration without direct communication between participants. However SETI is not a "true" P2P, the participants work towards a common goal, but have no communication among them, and the computations are controlled by a central server.

P2P computing is different from P2P network. A P2P network allows every computer in the network to act as a server to every other user on the network. But in P2P computing the relationship between users is negotiated in some manner. The participating peers, the computers that can be servers to others, may be a part of a P2P network, or may belong to networks with different characteristics. P2P network implies P2P communication between computers in the network. But P2P communication can also occur between two computers in a network that is not P2P.

P2P and Grid computing are both concerned with enabling resource sharing within distributed communities. In paper [11], the authors compare P2P computing and Grid computing in different respects, target communities, resources, scale, applications, and technologies. P2P systems have focused on resource sharing in environments characterized by potentially millions of users, most with homogenous desktop systems and low bandwidth, intermittent connections to the Internet. As such, the emphasis has been on global fault-tolerance and massive scalability. Grid systems have arisen from collaboration between generally smaller, better-connected groups of users with more diverse resources to share. The hallmark of a P2P system is that it lacks a central point of management; this makes it ideal for providing anonymity and offers some protection from being traced. Grid environments, on the other hand, usually have some form of centralized management and security (for instance, in resource management or workload scheduling).

The state in [12] is said this lack of centralization in P2P environments carries two important consequences: first, P2P systems are generally far more scalable than grid computing systems. Grid computing systems are inherently not as scalable as P2P systems. Second, P2P systems are generally more tolerant of single-point failures than grid computing systems. Although grids are much more resilient than tightly coupled distributed systems, a grid inevitably includes some key elements that can become single points of failure. This means that the key to build grid computing systems is finding a balance between decentralization and manageability.

Based on the mutual benefits that grid and P2P systems seem to offer to each other, the authors of paper [11] expect that the two approaches will eventually converge, especially when grids reach the "inter-grid" stage of development in which they essentially become public utilities. In [13], the author points out considerable potential for a synthesis between the two approaches. As depicted in Figure 2-1, a P2P Grid computer could combine the varied resources, services, and power of Grid computing with the global-scale, resilient, and self-organizing properties of P2P systems. A P2P substrate provides lower-level services on which to build a globally distributed Grid services infrastructure. The authors call this kind of system P2P GRID system. In P2P GRID system, every super grid peer is a grid system. Our thesis work use Distributed K-ary System (DKS), a structured P2P system in a Grid system to achieve this synthesis.

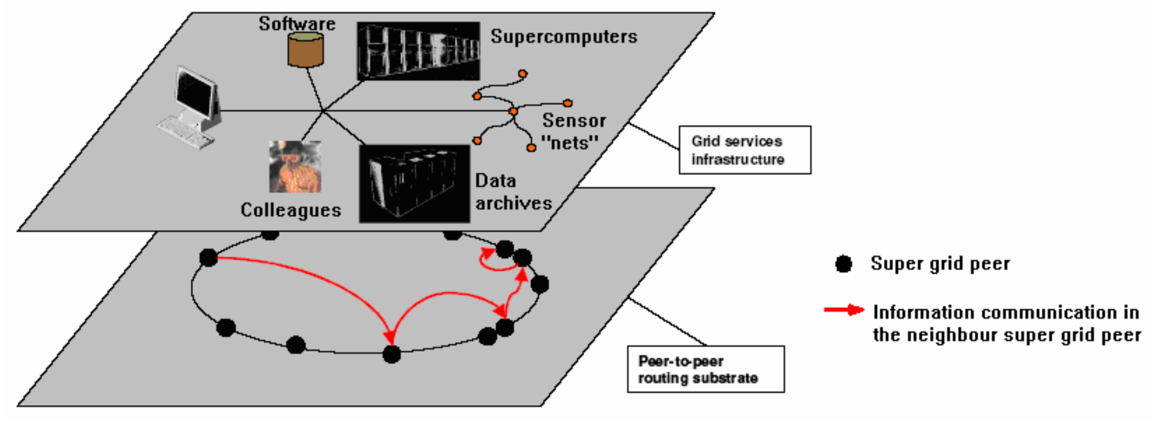

Software
Supercomputers
Sensor "nets"
Data archives
Colleagues
Grid services infrastructure
Peer-to-peer routing substrate
Super grid peer
Information communication in the neighbour super grid peer

Figure 2-1 A P2P GRID system based on MDS-Index services [13]

2.2.1 Unstructured P2P and structured P2P

A pure peer-to-peer network does not have the notion of clients or servers, but only equal peer nodes that simultaneously function as both "clients" and "servers" to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server. A typical example for a non peer-to-peer file transfer is an FTP server where the client and server programs are quite distinct, and the clients initiate the download/uploads and the servers react to and satisfy these requests.

A peer-to-peer computer network is a network that relies primarily on the computing power and bandwidth of the participants in the network rather than concentrating it in a relatively low number of servers. P2P networks are typically used for connecting nodes via largely ad hoc connections. Such networks are useful for many purposes. Sharing content files containing audio, video, data or anything in digital format is very common, and real-time data, such as telephony traffic, is also passed using P2P technology.

Based on the difference of information stored in the peer and search method, decentralized P2P systems are typically classified into two categories: unstructured P2P systems and structured P2P systems.

Unstructured P2P

In an unstructured P2P network, if a peer wants to find a desired piece of data in the network, the query is flooded through the network in order to find as many peers as possible that share the data. Before starting the query, the peer doesn't know where the data stored. So the flooding in unstructured P2P is blind. Although people set Time To Live (TTL) value to limit message life time, finding a appropriate TTL is not easy. Most of the popular P2P networks such as Napster [9], Gnutella [10] and KaZaA [14] are unstructured. Gnutella uses this kind of flooding search. Each node visited during a flood evaluates the query locally on the data items that it stores. This approach supports arbitrarily complex queries and it does not impose any constraints on the node graph or on data placement. For example, each node chooses any other node as its neighbor in the overlay and it can store the data it owns.

Flooding is a fundamental search method in unstructured P2P systems.However main disadvantage with such networks is that the queries may not always be resolved. A popular content is likely to be available at several peers and any peer searching for it is likely to find the same, but, if a peer is looking for a rare or not-so-popular data shared by only a few other peers, then it is highly unlikely that search be successful. Since there is no correlation between a peer and the content managed by it, there is no guarantee that flooding will find a peer that has the desired data. Flooding also causes a high amount of signaling traffic in the network and hence such networks typically have a very poor search efficiency.

Researchers have proposed some solutions to a problem mentioned above. The solutions are Random walk [15] and Dynamic Query [16]. For example, for searching, random walks achieve improvement over flooding in the case of clustered overlay topologies and in the case of re-issuing the same request several times.

Although unstructured P2P systems have the shortcoming mentioned above, the advantages of unstructured P2P networks are obvious: they incur almost no maintenance traffic. Node leaves are treated optimistically and node joins are very cheap since it is sufficient to know only one other node of the system in order to participate within the network.

Structured P2P

Structured P2P is developed to improve the performance of data discovery. It organizes peers so that any node can be reached in a bounded number of hops, typically logarithmic in the size of the network. In order to accomplish this, each node must hold additional status information, called "finger tables" to other nodes of the network. Here, the size of the status tables is O(log(n)). Each data item is identified by a key and nodes are organized into a structured graph that maps each key to a responsible node. The data or a pointer to the data is stored at the node responsible for its key. These constraints provide efficient support for exact match queries. Some well known structured P2P networks are: Chord [17], DKS [18], Pastry [19], Tapestry [20], CAN [21],Tulip [22]. Outside academia, DHT technology has been adopted as a component of BitTorrent [23] and in the Coral Content Distribution Network [24].

The first generation of peer-to-peer applications, including Napster and Gnutella, had restricting limitations such as a central directory for Napster and scoped broadcast queries for Gnutella limiting scalability. To address these problems a second generation of p2p applications, which are structured P2P, were developed including Tapestry, Chord, Pastry, DKS and CAN. These overlays implement a basic key-based routing mechanism. This allows for deterministic routing of messages and adaptation to node failures in the overlay network.

The advantage of Structured P2P networks is their ability to efficiently and reliably lookup objects in the network (IF the object exists, it WILL be found). Their two disadvantages are (1) only simple lookup queries are supported and (2) maintenance of the network, i.e., nodes joining and leaving the network, is much more expensive than in unstructured networks.

2.2.2 Distributed K-ary System (DKS)

DKS [25] is a peer-to-peer middleware developed at KTH/Royal Institute of Technology and the Swedish Institute of Computer Science (SICS) in the context of the european project PEPITO. It is entirely written in JAVA. Supports scalable Internet-scale Multicast, Broadcast, Name-based Routing, and provides a simple Distributed Hash Table abstraction.

Distributed hash table (DHT)

DHT[26] is a class of decentralized distributed systems that provide a lookup service similar to a hash table. A hash table is a data structure that associates *keys* with *values*.

The primary operation it supports efficiently is a *lookup*: given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number). It works by transforming the key using a hash function into a *hash*, a number that is used to index into an array to locate the desired location ("bucket") where the values should be.

DHT has key features: decentralization, scalability, fault tolerance. Decentralization means the nodes collectively form the system without any central coordination. Scalability means the system should function efficiently even with thousands or millions of nodes. Fault tolerance implies that lookups should be possible even if some nodes fail and the system should be reliable even with nodes continuously joining, leaving, and failing.

The structure of DHT can be decomposed into several main components [27] [28]. The foundation is an abstract keyspace, such as the set of 160-bit strings. A keyspace partitioning scheme splits ownership of this keyspace among the participating nodes. Each node maintains a set of links to other nodes (its *neighbors* or routing table). Together these links form the overlay network. A node picks its neighbors according to a certain structure, called the network's topology. Guarantee that the maximum number of hops in any route (route length) is low, so that requests complete quickly; and that the maximum number of neighbors of any node is low, so that maintenance overhead is not excessive are two key constrains on ontology.The overlay network then connects the nodes, allowing them to find the owner of any given key in the keyspace.

DHT use consistent hashing [29] to map keys to nodes. Consistent hashing is a scheme that provides hash table functionality in a way that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. This technique employs a function $\delta(k_1,k_2)$ which defines an abstract notion of the *distance* from key k_1 to key k_2, which is unrelated to physical distance or network latency. Each node is assigned a single key called its *identifier* (ID). A node with ID i owns all the keys for which i is the closest ID, measured according to δ.

Once these components are in place, a typical use of the DHT for storage and retrieval might proceed as follows. Suppose the keyspace is the set of 160-bit strings. To store a file with given *filename* and *data* in the DHT, the SHA1 hash of *filename* is found, 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* as specified by the keyspace partitioning, where the pair (*k*,*data*) is stored. Any other client can then retrieve the contents of the file by again hashing *filename* to produce *k* and asking any DHT node to find the data associated with *k* with a message *get*(*k*). The message will again be routed through the overlay to the node responsible for *k*, which will reply with the stored *data*.

DKS

Distributed K-ary System is scalable and fault-tolerance peer-to-peer middleware. It supports scalable self-managing without unnecessary bandwidth consumption. It uses symmetric replication [30] to enable information backup and concurrent requests. It also supports multicast [31] and efficient broadcast [32] [33] [34] group communication and guarantees consistent lookup results in the presence of nodes joining and leaving. Each lookup request in DKS is resolved in at most logk(N) overlay hops under normal operations. Each node maintains only (k − 1) $\log_k(N) + 1$ addresses of other nodes for routing purposes. PhD paper [31] gives all the algorithms which are implemented by DKS. For instance, it gives a simple broadcast algorithm, the time complexity of the simple broadcast algorithm is $\log_k(n)$ and the message complexity is n, because all nodes receive the message and no node receives the message more than once. Every node delegates non-overlapping intervals to all its children, so this algorithm is a no-redundancy algorithm. In the same paper, authors also give an extension algorithm called Simple Broadcast with Feedback which efficiently collects responses from all nodes after broadcasting.

One challenge in peer-to-peer is to maintain routing information in the presence of nodes joining, leaving, or failing. Most of the existing structured P2P systems use costly periodic stabilization protocols to ensure that the routing information is up-to-date [35][36][37]. The main disadvantage of this is that it includes a high bandwidth consumption. Indeed, at steady periods when the dynamism in the system is low, unnecessary bandwidth is consumed by periodic stabilization. In [38] the authors of the paper proposed a technique called correction-on-use that embeds parameters in routing messages such that incorrect routing information is corrected on-the-fly without the use of periodic stabilization. The outdated routing entries are corrected only when they are used. As long as the ratio of lookups to joins, leaves, and failures is high, the routing information is eventually corrected. Though correction-on-use consumes less bandwidth than periodic stabilization it assumes that the ratio between the number of routing messages to the dynamism in the system is high enough such that there are enough routing messages to correct the routing information that is invalidated as the result of dynamism. Consequently, routing information will become outdated if this ratio is low. In [18] the author proposed a novel technique called correction-on-change, which allows the system to automatically adapt to the dynamism while avoiding unnecessary bandwidth consumption. This technique achieves this goal without any assumptions on the amount of routing message in the system. Instead of correcting outdated routing information lazily, correction-on-change only updates outdated entries of all nodes eagerly whenever a change is detected. Effective failure handling is simplified as the detection of a failure triggers a correction-on-change which updates all the nodes that have a pointer to the failed node. The resulting system has increased robustness as nodes with stale routing information are immediately updated.

2.3 OGSA, WSRF, and GT4

The OGSA Grid Services Architecture (OGSA), developed by The Global Grid Forum[39], of which Globus Alliance is a leading member, aims to define a common, standard, and open architecture for grid-based applications. In OGSA, everything is service. Therefore Grid is an aggregation of extendable grid services. The objectives of OGSA are to:

- Manage resources across distributed heterogeneous platforms.
- Deliver seamless quality of service (QoS).
- Provide a common base for autonomic management solutions.
- Define open, published interfaces. For interoperability of diverse resources, grids must be built on standard interfaces and protocols.
- Exploit industry standard integration technologies.

Four main layers comprise the OGSA architecture: See Figure 2-2. Starting from the bottom, they are:

- Resources layer. Resources are physical resources and logical resources. Physical resources include servers, storage, and network. Above the physical resources are logical resources. They provide additional function by virtualizing and aggregating the resources in the physical layer.
- Web services, plus the OGSI extensions that define grid services. All grid resources, both logical and physical are modeled as services. OGSI exploits the mechanisms of Web services like XML and WSDL to specify standard interfaces, behaviors, and interaction for all grid resources. OGSI extends the definition of Web services to provide capabilities for dynamic, stateful, and manageable Web services that are required to model the resources of the grid. OGSA architected services
- OGSA architected grid services layer. The Global Grid Forum is currently working to define many of these architected grid services in areas like program execution, data services, and core services. As more implementations of grid services appear, OGSA will become a more useful Service-Oriented Architecture (SOA).
- Grid applications. Grid applications that use one or more grid architected services.

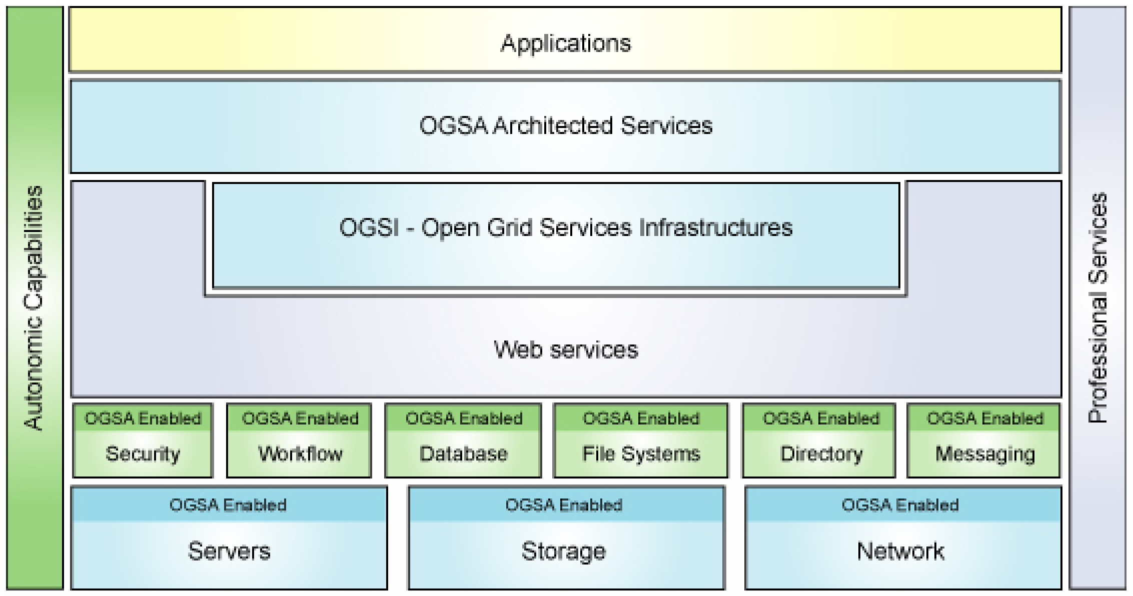


Figure 2-2 OGSA main architecture [40]

The Web Service Resource Framework (WSRF) is a specification developed by OASIS [41]. WSRF is only a small part of the whole GT4 Architecture. WSRF specifies how we can make our Web Services stateful. WSRF provides the stateful services that OGSA needs. Stateful service means that the Web service can "remember" information, or keep state, from one invocation to another. Instead of putting the state in the Web service WSRF keeps it in a separate entity called a resource, which will store all the state information. In WSRF, there is a formula: Web service + Resource= Web Resource. Endpoint reference is used to address the particular Web Resource. The following are the WSRF specifications:

- WS-ResourceProperties. It describes an interface to associate a set of typed values with a WS-Resource that may be read and manipulated in a standard way.
  It specifies how resource properties are defined and accessed. These properties are defined in the Web service's WSDL interface description.
- WS-ResourceLifetime. It supplies some basic mechanisms to manage the lifecycle of our resources.
- WS-ServiceGroup. It specifies how exactly we should go about grouping services or WS-Resources together. It is the base of more powerful discovery services (such as GT4's IndexService[42]) which allow us to group different services together and access them through a single point of entry (the service group).
- WS-BaseFaults. This specification aims to provide a standard way of reporting faults when something goes wrong during a WS-Service invocation.

Related specifications:

- WS-Notification. It is not a part of WSRF, is closely related to it. This specification allows a Web service to be configured as a notification producer, and certain clients to be notification consumers (or subscribers).

- WS-Addressing. We can use WS-Addressing to address a Web service + resource pair (a WS-Resource).

GT4 includes quite a few high-level services that we can use to build Grid applications. The relationship between OGSA, GT4, WSRF, and Web Services is shown in Figure 2-3

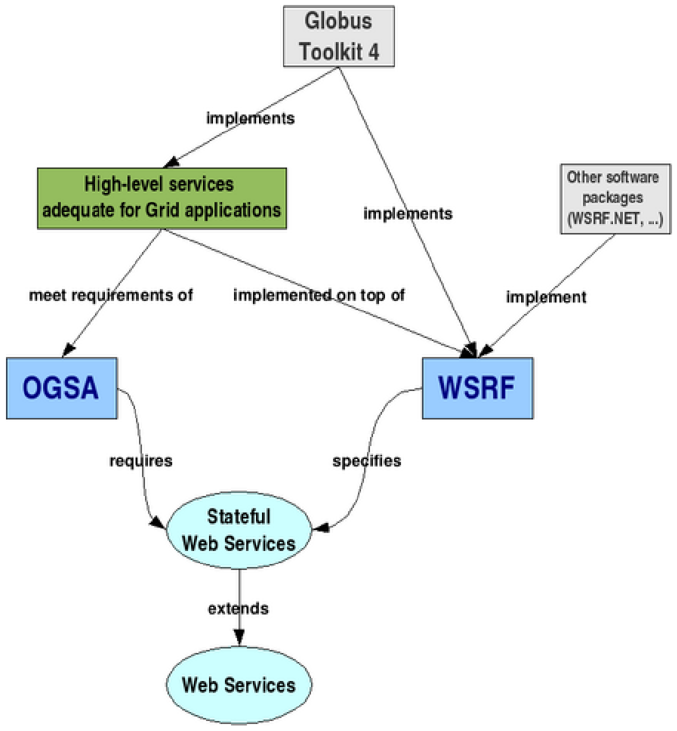


Figure 2-3. The relationship between OGSA, GT4, WSRF, and Web Services. [43]

2.4 Globus Toolkit 4

The Globus Toolkit is a software toolkit, developed by The Globus Alliance [4], which can be used to create Grid systems. The Globus Toolkit includes a resource monitoring and discovery service, a job submission infrastructure, security infrastructure, and data management service.

The Globus Toolkit's Monitoring and Discovery System (MDS) implements a standard Web Services interface to a variety of local monitoring tools and other information sources. MDS4 builds on query, subscription and notification protocols and interfaces defined by the WS Resource Framework (WSRF) and WS-Notification families of specifications and implemented by the GT4 Web Services Core. It provides a range of *information providers* which are used to collect information from specific sources. These components often interface to other tools and systems, such as the Ganglia cluster monitor and the PBS and Condor schedulers. MDS4 also provides two higher-level services: an *Index* service, which collects and publishes aggregated information about information sources, and a *Trigger* service, which collects resource information and performs actions when certain conditions are triggered. These services are built upon a common *Aggregation Framework* infrastructure that provides common interfaces and mechanisms for working with data sources.

Grid Resource Allocation Manager (GRAM) component is a core set of services that help perform the actual work of launching a job on a particular resource, checking status and retrieving results. GRAM provides job and execution management services to submit, monitor, and control jobs, but relies on supporting services for transferring files and managing credentials. File services are provided by GridFTP to assist GRAM with staging input and output files. Credential management handles delegation of credentials to other services and to the required distributed grid resources.

Grid Security Infrastructure (GSI) which enables grid entities to use authentication, authorization, and secure communication over open networks is the security component in Globus Toolkit 4. GSI uses *public key* cryptography (also known as asymmetric cryptography) as the basis for its functionality. GSI offers programmers five features:

- Transport-level and message-level security. The difference between these two level is that Transport-level security encrypt all the information exchanged between the client and the server, however the latter only encrypt the content of the SOAP message. GSI offers two message-level protection schemes which are GSI Secure Message and GSI Secure Conversation, and one transport-level scheme, GSI Transport.
- Three authentication methods. The first method is X.509 certificates, all three protection schemes are used along with X.509 certificated to provide strong authentication. The second is Username and password and the third method is anonymous authentication.
- Several authorization schemes. GSI supports authorization in both the server-side and the client-side. The server will decide if it accepts or declines an incoming request depending on the authorization it chooses. The client figure out when it will allow a service to be invoked.
- Credential delegation, single sign-on and Proxy Certificates. GSI provides a delegation capability: an extension of the standard SSL protocol which reduces the number of times the user must enter his passphrase. If a Grid computation requires that several Grid resources be used (each requiring mutual authentication), or if there is a need to have agents (local or remote) requesting services on behalf of a user, the need to re-enter the user's passphrase can be avoided by creating a *proxy*. Using proxy Certificate, the user only has to sign in once to create the proxy certificate which then is used for all subsequent authentications.
- Different levels of security: container, service, and resource. We can configure security and set different authorization mechanisms for each level.

The Globus Toolkit provides a number of components for doing data management. The components available for data management fall into two basic categories: data movement and data replication. There are two components related to data movement in the Globus

Toolkit: the Globus GridFTP tools and the Globus Reliable File Transfer (RFT) service. The Replica Location Service (RLS) is one component of data management services for Grid environments. RLS is a tool that provides the ability keep track of one or more copies, or replicas, of files in a Grid environment.

2.5 Semantics

The Grid is frequently heralded as the next generation of the Internet. The Semantic Web is proposed as the (or at least a) future of the Web [44]. The Semantic Web is a vision for the future of the Web in which information is given explicit meaning, making it easier for machines to automatically process and integrate information available on the Web. The semantic web comprises the standards and tools of XML, XML Schema, RDF, RDF Schema and OWL. XML provides an elemental syntax for content structure within documents, yet associates no semantics with the meaning of the content contained within. XML Schema is a language for providing and restricting the structure and content of elements contained within XML documents. RDF is a simple language for expressing data models, which refer to objects ("resources") and their relationships. RDF Schema is a vocabulary for describing properties and classes of RDF-based resources, with semantics for generalized-hierarchies of such properties and classes. In automatic Web service discovery application the OWL-S service ontology [45] is used to provide the vocabulary for service advertisements and users’ requirements and the OWL ontologies [46] are used to describe domain knowledge. Based on these descriptions a prototype of web service automatic discovery, where machines can flexibly and automatically search for services according to users’ requirements, is implemented.

2.5.1 RDF

RDF is a general framework for describing a Web site's metadata, or the information about the information on the site. It describes resources in terms of simple properties and property values. The subject of an RDF statement is a resource, possibly as named by a Uniform Resource Identifier (URI). It does not represent a tangible, network-accessible resource. Such a URI could denote the abstract notion of world peace. RDF is intended for situations where information should be processed by applications.

One RDF statement has only three parts:

- Subject – identifies the thing the statement is about
- Predicate – the part that identifies properties of the subject
- Object – the part that specifies the value of the property

Let's take RDF statement, “http://www.example.org/artical.html has a creator Yeou Yang” as an example. “http://www.example.org/artical.html” is a subject, “http://purl.org/dc/element/1.1/creator” is a predicate and “http://www.example.org/staffname/YeouYang” is the object. The graph is presented below, Figure 2-4:

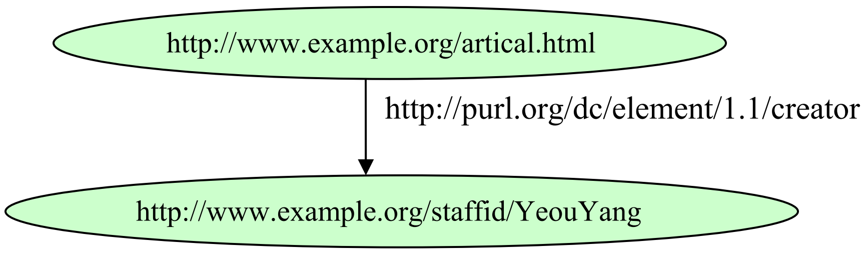

http://www.example.org/artical.html
http://purl.org/dc/element/1.1/creator
http://www.example.org/staffid/YeouYang

Figure 2-4 An example of RDF model

We put RDF description in the XML file as following:

```
<?xml version="1.0"?>
    <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
                xmlns:exterms=" http://purl.org/dc/element/1.1/">
        <rdf:Description rdf:about="http://www.example.org/index.html">
                <exterms:creator
                        rdf:resource="http://www.example.org/staffid/YeouYang" />
        </rdf:Description>
    </rdf:RDF>
```

Figure 2-5 XML serialization of the RDF model

2.5.2 RDF Schema (RDFS)

RDF describes resources with classes, properties, and values. In addition, RDF also needs a way to define application-specific classes and properties. Application-specific classes and properties must be defined using extensions to RDF. One such extension is RDF Schema. RDF Schema does not provide actual application-specific classes and properties. Instead RDF Schema provides the framework to describe application-specific classes and properties. Classes in RDF Schema are much like classes in object oriented programming languages. This allows resources to be defined as instances of classes, and subclasses of classes. Let's take another example, "horse is a subclass of animal". The RDF schema is shown in Figure 2-6

```
<?xml version="1.0"?>
   <rdf:RDF
      xmlns:rdf= "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xml:base=  "http://www.animals.fake/animals#">
        <rdfs:Class rdf:ID="animal" />
        <rdfs:Class rdf:ID="horse">
                <rdfs:subClassOf rdf:resource="#animal"/>
       </rdfs:Class>
   </rdf:RDF>
```

Figure 2-6 An example of RDF schema

2.5.3 OWL

This language is used as a standard by the W3C. OWL is a set of XML elements and attributes, with standardized meaning, that are used to define terms and their relationships. A logical reasoning can be applied to the ontology. The logic which is used is description logic (some other is FLogic). Ontology can also be expressed as binary relations.

The OWL has the same features found in other languages used for ontology, such as: DAML-OIL (DARPA Agent Markup Language - Ontology Inference Layer), RDF (Resource Description Framework) and RDF-S (RDF Schema). OWL is designed for use by applications that need to process the content of information instead of only presenting information to humans. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF-S by providing additional vocabulary along with a formal semantics. OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full [46].

OWL is Different from RDF. OWL and RDF are much of the same thing, but OWL is a stronger language with greater machine interpretability than RDF. OWL extends RDF Schema. OWL comes with a larger vocabulary and stronger syntax than RDF. OWL adds more vocabulary for describing properties and classes: among others, relations between classes (e.g. disjointness), cardinality (e.g. "exactly one“ instance), equality, richer typing of properties, characteristics of properties (e.g. symmetry), and enumerated classes.Different types of constraints can be expressed: equivalentProperty , inverseOf, TransitiveProperty, SymmetricProperty etc [47].

We give an example of using OWL to define two terms, “Camera”, “SLR” and their relationship. State that SLR(Single Lens Reflex) is a type of camera.

```
<owl:Class rdf:ID="Camera"/>

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

Figure 2-7 An example of using OWL to define terms and their relationship

2.5.4 OWL-S

OWL-S [45] supplies web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-interpretable form. The current version of OWL-S builds on the Ontology Web Language (OWL). OWL-S of Web services is intended to facilitate the automation of Web service tasks including automated Web service discovery, execution, interoperation, composition and execution monitoring. In OWL-S, service descriptions are structured into three essential types of knowledge, shown in Figure 2-8: a ServiceProfile, a ServiceModel (which describes the ServiceProfile), and a ServiceGrounding. Services can be matched by either their OWL-S profiles [48] or OWL-S models [49]. Using semantic-based description to describe service, we can retrieve and process service easily and improve the efficiency of using the network.

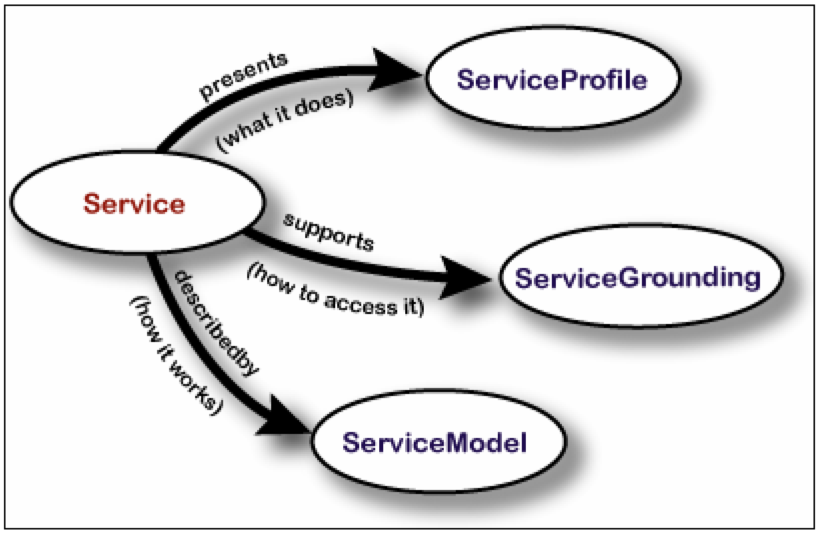


Figure 2-8 Top level of the service ontology [50]

- The **service profile** presents “what the service does” with necessary functional information: input, output, preconditions, and the effect of the service. It is used for advertising and discovering services. Figure 2-9 shows the Properties of the Profile.

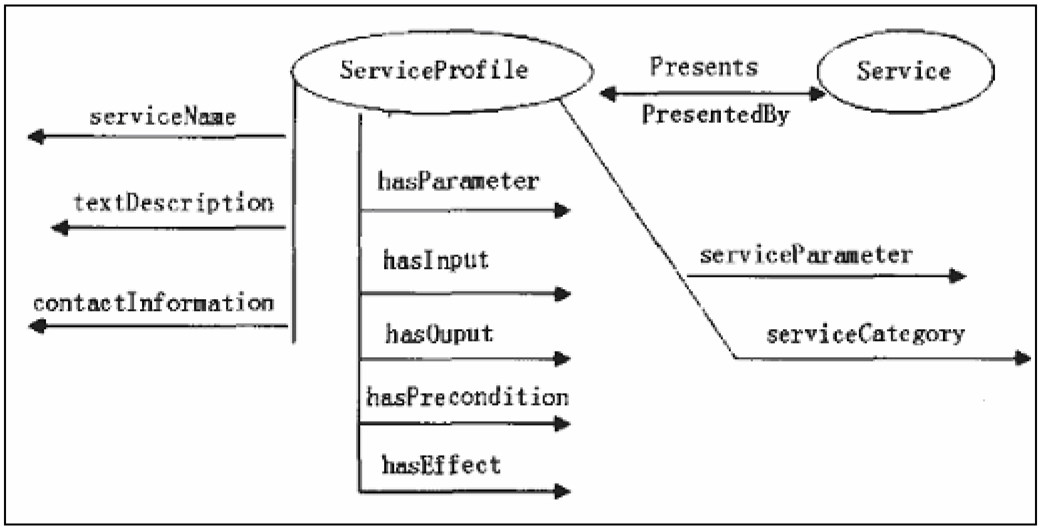


Figure 2-9 Properties of the Profile [51]

Service Profile: The class ServiceProfile provides a superclass of every type of high-level description of the service. ServiceProfile does not mandate any representation of services, but it mandates the basic information to link any instance of profile with an instance of service. There is a two-way relation between a service and a profile, so that a service can be related to a profile and a profile to a service. These relations are expressed by the properties presents and presentedBy.

Service Name, Contacts and Description: Some properties of the profile provide human-readable information that is unlikely to be automatically processed. These properties include *serviceName*, *textDescription* and *contactInformation.* A profile may hame at most one service name, and text description, but many items of contact information as the provider wants to offer.

Functionality Description: An essential component of the profile is the specification of what functionality the service provides and the specification of the conditions that must be satisfied for a successful result. In addition, the profile specifies what conditions result from the service, including the expected and unexpected results of the service activity. The OWL-S Profile represents two aspects of the functionality of the service: the information transformation (represented by inputs and outputs) and the state change produced by the execution of the service (represented by preconditions and effects).

Profile Attributes: Besides functional description of services, there are additional attributes include the quality guarantees that are provided by the service, possible classification of the service, and additional parameters that the service may want to specify. *serviceParameter* is an expandable list of properties that may accompany a profile description. The value of the property is an instance of the class *ServiceParameter*. *serviceCategory* refers to an entry in some ontology or taxonomy of services. The value of the property is an instance of the class ServiceCategory.

- The **service model** describes "how the service works", that is all the processes the service is composed of, how these processes are executed, and under which conditions they are executed. It gives a detailed description of a service's operation.
- The **service grounding** describes "How is it used". It provides details on how to interoperate with a service, via messages.

2.5.5 Reasoning

- Jena[52]

Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL, SPARQL and includes a rule-based inference engine.
The Jena Framework includes:

- A RDF API
- Reading and writing RDF in RDF/XML, N3 and N-Triples
- An OWL API
- In-memory and persistent storage
- SPARQL query engine

Jena only supports OWL Lite. It has a number of predefined reasoners. They are transitive reasoner, RDFS rule reasoner, OWL, OWL Mini, OWL micro reasoners, DAML micro reasoner, Generic rule reasoner. The default OWL reasoner included in Jena is rather limited and incomplete hence the need for a fuller reasoner to be plugged on Jena. The Jena2 inference subsystem is designed to allow a range of inference engines or reasoners to be plugged into Jena. The primary use of this mechanism is to support the use of languages such as RDFS and OWL which allow additional facts to be inferred from instance data and class descriptions.

- Pellet [53]

Pellet is an open source, OWL DL reasoner in Java. It can be used in conjunction with both Jena and OWL API libraries; it can also be downloaded and be included in other applications. Based on the tableaux algorithms developed for expressive Description Logics (DL). It has many features:

- Standard Reasoning Services
- Multiple Interfaces to the Reasoner [54]
- Datatype Reasoning
- Conjunctive Query Answering
- Rules Support
- Ontology Analysis and Repair
- Ontology Debugging
- Incremental Reasoning

Pellet provides all the standard inference services that are traditionally provided by DL reasoners:

- Consistency checking, which ensure an ontology doesn’t contain any contradictory facts.
- Concept satisfiability, which determines whether it’s possible for a class to have any instances.
- Classification, which computes the subclass relations between every named class to create the complete class hierarchy. The class hierarchy can be used to answer queries such as getting all or only the direct subclasses of a class.
- Realization, which finds the most specific classes that an individual belongs to; or in other words, computes the direct types for each of the individuals.

3 Related works in Semantic-Based Resource Discovery

3.1 Monitoring and Discovery System (MDS4)

Monitoring and Discovery System (MDS4) [55] is a suite of web services to monitor and discover resources and services on Grids. It is the Globus Toolkit's information services component. MDS4 provides query and subscription interfaces to arbitrarily detailed resource data and a trigger interface that can be configured to take action when pre-configured trouble conditions are met.

Monitoring and discovery mechanisms can help us observing resources or services and finding a suitable resource to perform a task. Take finding a compute host on which to run a job for instance. This process may involve both finding which resources have the correct CPU architecture and choosing a suitable member with the shortest submission queue. The motivation for collecting information is to enable discovery of services or resources and enable monitoring of system status.

In the following sections (3.1.1- 3.1.4), we'll take a closer look at the MDS4 and know more about how to use MDS4 in a grid system.

3.1.1 Three types of Aggregator

Before we introduce this section, we need to import an important concept that is *aggregator service*. MDS4 provides *aggregator services* that collect recent state information from registered information sources. It provides some user interfaces like browser based interfaces, command line tools, and Web service interfaces that allow users to query and access the collected information.

MDS4 provides three different aggregator services with different interfaces and behaviors: *MDS-Index*, which supports Xpath queries on the latest values obtained from the information sources; *MDS-Trigger*, which performs user-specified actions (such as send email) whenever collected information matches user determined criteria; and *MDS-Archiver*, which stores information source values in a persistent database that a client can then query for historical information. It also implemented a range of *information providers* used to collect information from specific sources. We will discuss *information providers* in section 3.1.4.

The *MDS-Index* service makes data collected from information sources available as XML documents. More specifically, the data is maintained as WSRF resource properties. There are three ways to retrieve this data. The first one is write your own application which collect information using standard Web service interface, WSRF get-property and WS-Notification operations. The second method is using command line tool *wsrf-get-property* to retrieve resource properties, with the desired resource property specified via an XPath expression. The third method is using a tool WebMDS. Standard transformations included in GT4 provide an interface that displays overview information, with hyperlinks giving the ability to view more detailed information about each monitored resource.

The *MDS-Trigger* service defines a Web service interface that allows a client to register an Xpath query and a program to be executed whenever a new value matches a user-supplied matching rule. It compares the data against a set of conditions defined in a configuration file. When a condition is met, or triggered, an action takes place.

The *MDS-Archive* service stores all values received from information sources in persistent storage. Client requests can then specify a time range for which data values are required.

MDS4 makes heavy use of XML and Web service interfaces to simplify the tasks of registering information sources and locating and accessing information of interest. In particular, all information collected by aggregator services is maintained as XML, and can be queried via Xpath queries. Therefore we decide to use Xpath language as client's query language to query the response XML from MDS-Index service.

3.1.2 MDS Aggregator Framework

MDS Aggregator Framework provides common VO-level functionality, such as registration management, collection of information about Grid resource. It also allows developer to plug in their specialized functionalities, for example Index Service and Trigger Service. There are two important concepts *Aggregator sources* and *Aggregator sink* in MDS Aggregator Framework. *Aggregator sources* collect information from WS-Resources and feed that information to *Aggregator sinks* (such as the Index Service and Trigger Service). The following graphic, Figure 3-1, describes the basic information flow including the three standard aggregator sources: Query Aggregator Source, Subscription Aggregator Source and Execution Source.

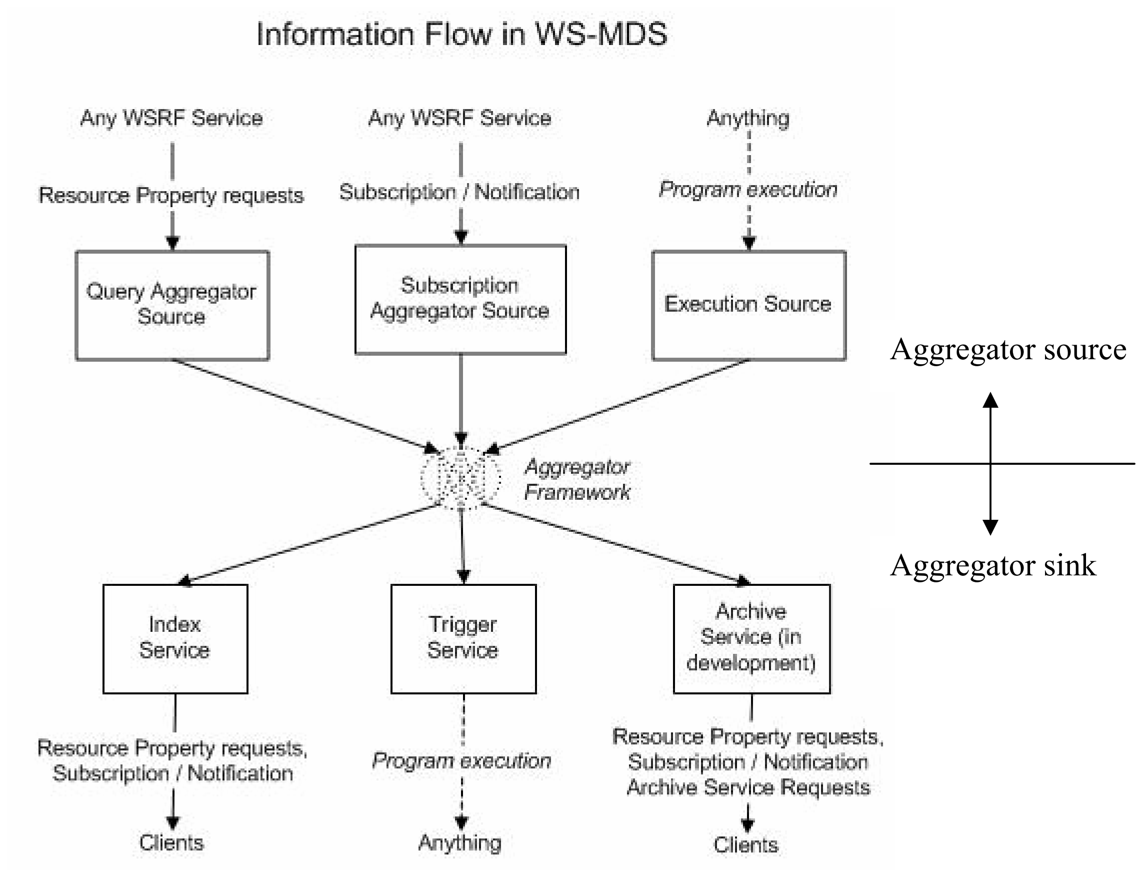

Information Flow in WS-MDS
Any WSRF Service
Resource Property requests
Any WSRF Service
Subscription / Notification
Anything
Program execution
Query Aggregator Source
Subscription Aggregator Source
Execution Source
Aggregator source
Aggregator Framework
Aggregator sink
Index Service
Trigger Service
Archive Service (in development)
Resource Property requests, Subscription / Notification
Clients
Program execution
Anything
Resource Property requests, Subscription / Notification Archive Service Requests
Clients

Figure 3-1 Information flow in MDS4 [56]

The basic ideas of aggregator-information source framework as follows:

- Information sources for which discovery or access is required are explicitly registered with an aggregator service. These information sources could be a file, a program, a Web service, or another network-enabled service.
- Registrations have a lifetime, if not renewed periodically, they expire. Information sources must be registered periodically with any aggregator service that is to provide access to its data values. In our thesis work, we use MDS-Index as aggregator service. Registration is performed via a Web service (WS-ServiceGroup) Add operation. Information resources are registered using tools like mds-servicegroup-add [57]. One way of registration is to use an aggregator registration file defines service registrations. Each registration specifies a grid resource, a service group the resource should register with, and service configuration parameters. This file is used with the mds-servicegroup-add command to maintain registrations between grid resources and the Index Service. The file defines the location of the Index Service referred to as the default service group end point reference.
- The aggregator periodically collects up-to-date state or status information from all registered information sources.
- The aggregator then makes all information obtained from registered information sources available via an aggregator-specific Web services interface.

MDS4 aggregators are distinguished from a traditional static registry such as X.500, LDAP and UDDI by their soft-state registration of information sources and periodic refresh of the information source values that they store. The authors of paper [60] explained that in the case of X.500 and LDAP, there is an assumption of a well-defined hierarchical organization, and current implementations tend not to support dynamic data well. This dynamic behavior provided by MDS4 enables scalable discovery, by allowing users to access "recent" information without accessing the information sources directly. However X.500, LDAP and UDDI do not address explicitly the dynamic addition and deletion of information sources. So MDS4 aggregators are more flexible.

3.1.3 Information Providers

An Aggregator Source is used to collect XML-formatted data, this data is provided by external software component, we call it Information Providers. These components often interface to other tools and systems, such as the Ganglia cluster monitor and Condor schedulers and WS GRAM (see Table 3-1 for a current list).

Table 3-1: Information Providers

Source	Information
WS GRAM	The job submission service component of GT4. This WSRF service publishes information about the local scheduler, including: query information, number of CPUs available and free, job count information, some memory staticts.
Reliable File Transfer Service (RFT)	The file transfer service component of GT4. This WSRF service publishes: status data of the server, transfer status for a file or set of files, number of active transfers, and some status information about the resource running the service.
Community Authorization Service (CAS)	This WSRF service publishes information identifying the VO that it serves. Such as ServerDN, VODescription.
Ganglia information Provider	It gathers cluster data from resources running Ganglia using the XML mapping of the GLUE schema [61] and reports it to a WS GRAM service, which publishes it as resource properties. This information includes: basic host data (name, ID), memory size, OS name and version, file system data, processor load data and other basic cluster data.
Hawkeye information Provider	It gathers Hawkeye data about Condor pool resources using the XML mapping of the GLUE schema and reports it to a WS GRAM service, which publishes it as resource properties. This information includes: basic host data (name, ID), processor information, memory size, OS name and version, file system data, processor load data and other basic Condor host data.
Any other WSRF service	Publish resource properties.

The GLUE resource property (as used by GRAM) collects information from two sources: the scheduler and the cluster information system (for example Ganglia or Hawkeye). These are merged to form a single output resource property in the GLUE schema.

Because WS GRAM, RFT and CAS have already registered to DefaultIndexService by default, if we want to collect data from two cluster monitoring systems Ganglia or Hawkeye, we need to make sure that Ganglia or Hawkeye is configured and running properly to view cluster information in the Index Service. Document [62] gives us more information about configuration and how to write a new provider.

3.2 The Java XPath API

MDS4 has similar feature to previous versions MDS2 and MDS3. One important difference is a more powerful query language (XPath instead of LDAP). Because the query response returned from MDS4 Index Service is in form of XML file which includes the look up result, we use XPath[63] in our thesis work.

“XPath 2.0 is an expression language that allows the processing of values conforming to the data model defined in [XQuery/XPath Data Model (XDM)]. [63]” “XPath 2.0 is a superset of XPath 1.0, with the added capability to support a richer set of data types, and to take advantage of the type information that becomes available when documents are validated using XML Schema. [63]” It is backwards compatible with XPath 1.0.

In XPath, there are four kinds of data type: node-set, boolean, string, number. It has seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document (root) nodes. XPath uses path expressions to select nodes or node-sets in an XML document. The most useful path expressions are listed in the Table 3-2.

Table 3-2: useful path expressions

Expression	Description
nodename	Selects all child nodes of the node
/	Selects from the root node
//	Selects nodes in the document from the current node that match the selection no matter where they are
.	Selects the current node
..	Selects the parent of the current node
@	Selects attributes

We can also use predicates to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets. For example, we select all the title elements that have an attribute named lang with a value of 'eng'. We use expression like //title[@lang='eng']. Another example, /bookstore/book[@price>35.00]/title means that selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00. XPath wildcards can be used to select unknown XML elements. For example, we use //title[@*] to select all title elements which have any attribute.

XPath includes over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, Boolean values, and more.

3.3 Matchmaking

Discovering a service which satisfies a request sufficiently is a major issue in any application process. As a result, the motivation to develop a powerful and customizable matchmaking engine becomes a very important criterion. The Matchmaker serves as a "yellow pages" of service capabilities. The Matchmaker allows users and/or software agents to find each other by providing a mechanism for registering service capabilities. Matchmaking agents would, upon receiving a request from a consumer of a web service, search their database of advertisements to come up with a set of advertisements that best meet the requested requirements. In section 3.4.1 we introduce a traditional resource matchmaker Condor Matchmaker in Grid, section 3.4.2 we will see the profile and the model can be used for matchmaking.

3.3.1 Condor Matchmaker

Existing resource description and resource selection in the Grid is highly constrained. Traditional resource matching, as exemplified by the Condor Matchmaker, is done based on symmetric, attribute-based matching. In these systems, the values of attributes advertised by resources are compared with those required by jobs. For the comparison to be meaningful and effective, the resource providers and consumers have to agree upon attribute names and values. The exact matching and coordination between providers and consumers make such systems inflexible and difficult to extend to new characteristics or concepts. Moreover, in a heterogeneous multi-institutional environment such as the Grid, it is difficult to enforce the syntax and semantics of resource descriptions.

Condor uses matchmaking to bridge the gap between planning and scheduling. Matchmaking creates opportunities for planners and schedulers to work together while still respecting their essential independence. The ClassAd mechanism in Condor provides an extremely flexible and expressive framework for matching resource requests (e.g. jobs) with resource offers (e.g. machines). ClassAds allow Condor to adopt to nearly any desired resource utilization policy and to adopt a planning approach when incorporating Grid resources.

In book [64], authors describe the steps for matchmaking, shown in Figure 3-2. In the first step, agents and resources advertise their characteristics and requirements in classified advertisements (ClassAds). In the second step, a matchmaker scans the known ClassAds and creates pairs that satisfy each other's constraints and preferences. In the third step, both parties of the match are informed by matchmaker. In the final step, claiming, the matched agent and the resource establish contact, possibly negotiate further terms, and then cooperate to execute a job.

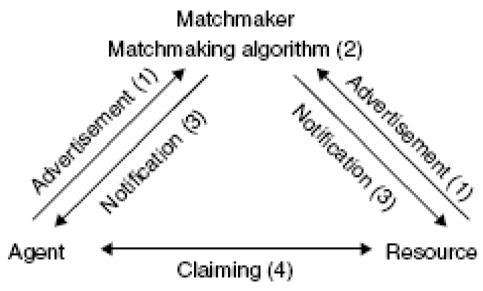


Figure 3-2 Condor matchmaking

3.3.2 Semantic web service matchmaking algorithms

Before we introduce semantic based web service matchmaking, let us see an online car shop example. Figure 3-3 gives a simple example of web service matchmaking. In figure 3-3, a provider describes advertised services using semantic web service description. A requester specifies what kind of service he wants to find. Given the automobile name, the service should return the price of it. It means the service should have one input and one out put. Repository matches the input and output of request and advertisements web service separately and returns matched advertisements in relevance order. A provider advertises automobile selling services, whereas a requester is looking for a service selling sedan. We know that sedan is an automobile having two or four doors and a front and rear seat. From figure 3-4 we find out the sedan is subsumed by Family car. The output of the request and advertisement are exactly the same.

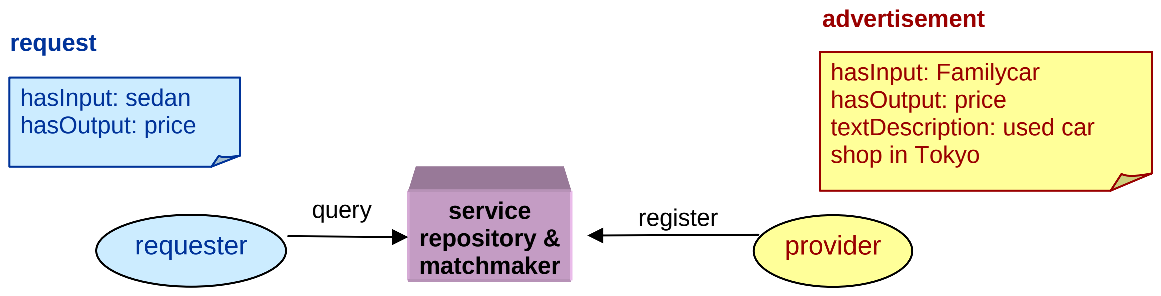


Figure 3-3 Semantic based web service matchmaking

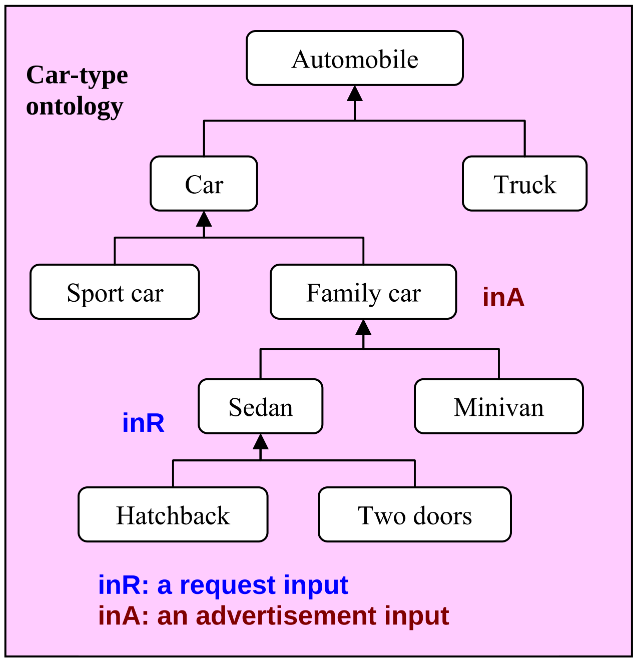


Figure 3-4 Car-type ontology

Based on the semantic description according to OWL-S, there are already some approaches available for matching of service requirements with service advertisements according to such ontology.

The idea beneath the matching methods in [65] is that two services do not necessarily need to be exactly equal to match; the only thing we need is to let services "sufficiently" similar. A match between an advertisement and a request consists of the match of (1) all request outputs are matched by advertisement outputs and (2) all advertisement inputs are matched by request inputs (Figure 3-5). Guarantees that the matched service provides all outputs requested by the requester, and that the requester provides all input required for correct operation to the matched service. In [65], the algorithm for output matching is described in detail in Figure 3-6. The degree of success depends on the degree of match detected. If one of the request's output is not matched by any of the advertisement's output the match fails. The matching between inputs is computed following the same algorithm, but with the order of the request and the advertisement reversed (Figure 3-7).

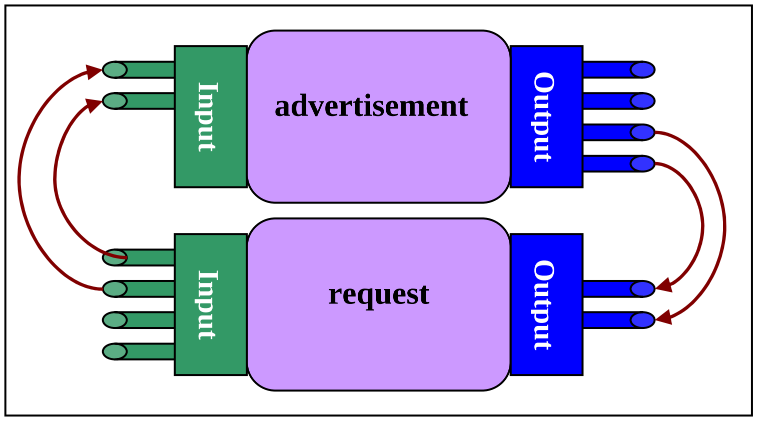


Figure 3-5 Basic principle of matching

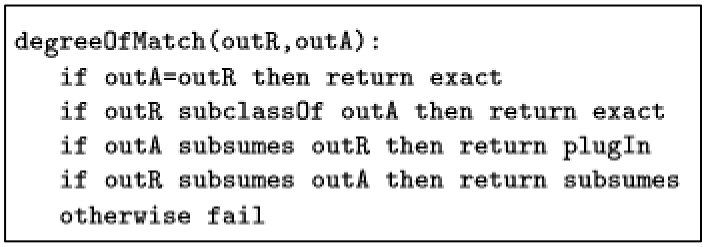
```
outputMatch(outputsRequest, outputsAdvertisement) {
   globalDegreeMatch= Exact
   forall outR in outputsRequest do {
      find outA in outputsAdvertisement such that
         degreeMatch= maxDegreeMatch(outR,outA)
         if (degreeMatch=fail) return fail
         if (degreeMatch<globalDegreeMatch)
            globalDegreeMatch= degreeMatch
   return sort(recordMatch);}
```

Figure 3-6 Algorithm for output matching

```
inputMatch(inputsRequest, inputsAdvertisement){
        globalDegreeMatch=Exsact
            forall inA in inputsAdvertisement do{
                find inR in inputsRequest such that
               degreeMatch= degreeOfMatch(inR, inA)
                if(degreeMatch=fail) return fail
                if(degreeMatch<globalDegreeMatch)
                         globalDgreeMatch=degreeMatch
               return sort(recordMatch);
        }
}
```

Figure 3-7 Algorithm for input matchmaking

Degrees of match are organized along a discrete scale in which exact matches are of course preferable to any another; plugIn matches are the next best level, because the output returned can probably be used instead of what the requester expects. Subsumes is the third best level since the requirements of the requester are only partially satisfied; the advertised service can provide only some specific cases of what the requester desires. Fail is the lower level and it represents an unacceptable result [65]. The maxDegreeMatch(outR,outA) in output matching algorithm and the degreeOfMatch(inR,inA) in the input matchmaking can be the reuse the same algorithm in Figure 3-8.



```
degreeOfMatch(outR,outA):
   if outA=outR then return exact
   if outR subclassOf outA then return exact
   if outA subsumes outR then return plugIn
   if outR subsumes outA then return subsumes
   otherwise fail
```

Figure 3-8 Rules for the degree of match assignment

- There is another extension of the algorithm [66] and [67].

Different matching degrees are achieved based on the matching degrees of the input and output types for requested and advertised services. Furthermore, additional elements of the service description, such as the service category, are either covered by reasoning processes. “DAML-S is a DAML-based Web service ontology, which supplies Web service providers with a core set of markup language constructs for describing the properties and capabilities of their Web services in unambiguous, computer-intepretable form. DAML-S markup of Web services will facilitate the automation of Web service tasks, including automated Web service discovery, execution, composition and interoperation.” “Everything said about a DAML-S file (or ontology element) is also true of the equivalent OWL-S file (or ontology element). [68]” OWL-S is built upon OWL and DAML-S is built upon DAML-OIL which is predecessor of OWL. So matchmaking algorithm for daml-s description in [67] can be used for OWL-S service description matchmaking.

The matching algorithm uses propertyMatch and conceptMatch to classify the different relations between properties and concepts when comparing two concepts or two properties. In Description Logics [69], nodes are mainly referred to as concepts. In general, the elements of a network are nodes and links. Figure 3-9 describes a simple network. Typically nodes are used to characterize sets of classes of individuals, such as Mother in the network, and links are used to characterize relationships among them, such as the link between the concepts Mother and Female states that a woman is a female. Such a relationship is often termed a "IS-A" relationship. This relationship defines a hierarchy over the concepts, i.e. Female is a more general concept than Mother. The more general concept is termed the superconcept, whereas the more specific concept is called the subconcept. The "IS-A" relationship also provides the basis for the inheritance of properties; when a concept is more specific than another concept, it inherits the properties of the more general one. For example, if the concept Person has the property age then the concept Woman also has the property age.

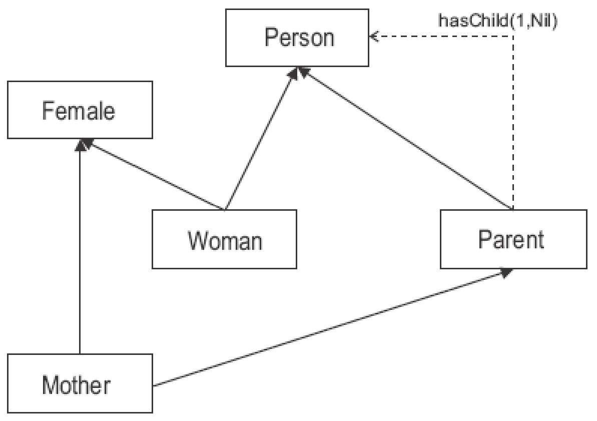


Figure 3-9 A simple example of network [69]

DAML-S service profiles are defined as subclass of the Profile class, but can also be indirect subclasses of Profile, this way it is possible to build a service hierarchy [70] and relationships between two profiles can be found with reasoning on subsumption. Moreover, IOPEs (Inputs-Outputs-Preconditions-Effects) can also be classified the same way, by defining an IOPE parameter as a subproperty of another IOPE parameter. Thanks to these classifications, a distance between two profiles or two parameters can be computed.

[67] defines a ranking for two parameters, shown in Table 3-3. The algorithm distinguishes up to 9 different degrees for the matching of parameters. These 9 different degrees are justified by the classification of parameters and service profiles. It gives two types of matchmaking, propertyMatch and conceptMatch. The propertyMatch determines *Equivalent, Subproperty and Fail* degrees for two given properties. The conceptMatch determines the *Equivalent, Subsumes, inverted subsumption and Fail* degrees for two given concepts. The priority of propertyMatch is higher than typeMatch, as a classification of a parameter gives more insight to the purpose of the parameter than its type definition. The types in the Table 3-3 can be concepts, so the degrees of a type-match are the same as the degrees of the concpetMatch. The subsumption pattern is very important for reasoning. It can be seen as the determination of subconcept and superconcept relationships between concepts of a given terminology. The subproperty relationship is similar to subsumption, the only difference is the relationship exists between properties.

Rank	property-match result	type-match result
0	Fail	Any
	Any	Fail
1	Unclassified	Invert Subsumes
2		Subsumes
3		Equivalent
4	Subproperty	Invert Subsumes
5		Subsumes
6		Equivalent
7	Equivalent	Invert Subsumes
8		Subsumes
9		Equivalent

Table 3-3 Rankings for the matching of two parameters

The final matching result for two considered services is composed of four types of matching, input, output, profile, user-defined matching. In addition to specifying the requested service and the advertised service, the user of the matching algorithm specifies lower bounds on the matching degrees for the input parameter, output parameter and profile matching see Table 3-4. Each partial matching result has to satisfy its minimal requirement for the matching algorithm to succeed. In addition, if the user-defined matchmaking returns FAIL, the final matching result will also return FAIL, regardless of the results of the other three partial matching results. If it returns MATCH, the matching result is only based on the other three partial matching results.

Input	Output	Profile
FAIL	FAIL	FAIL
UNCLASSIFIED	PARTIAL_ FAIL	CLASSIFIED
SUBPROPERTY	UNCLASSIFIED	SUBSUMS
TYPE_INVERT	SUBPROPERTY	MATCH
TYPE_SUBSUMS	TYPE_INVERT	
MATCH	TYPE_SUBSUMS	
	MATCH	

Table 3-4 Matching degree for input, output and profile matching

The following (from Figure 3-10 and Figure 3-12) are the matching algorithms [67] based on DAML-S description. Each output, input matching algorithm provides a function called *rankForParameters* which determines the rank of two parameters according to Table 3-3 and which calls the functions propertyMatch and conceptMatch.

```
public int match(Vector reqOutputsList, Vector advOutputsList, Reasoner reasoner) {
  if (reqOutputsList==null) {
    return MATCH;
  } else {
    if (advOutputsList==null) {
      return PARTIAL_FAIL;
    }
  }
  boolean atLeastOneFail = false;
  int minOverallRank  = 9;
  for (int i=0; i<reqOutputsList.size();i++) {
    Output reqOutput = (Output)reqOutputsList.elementAt(i);
    Output bestMatch = null; int maxRank = 0;
    for (int j=0; j<advOutputsList.size(); j++) {
      Output tempAdvOutput = (Output)advOutputsList.elementAt(j);
      int rank = reasoner.rankForParameters(tempAdvOutput, reqOutput);
      if (rank>maxRank) {
        maxRank = rank;
        bestMatch = tempAdvOutput;
      }
    }
    if (bestMatch==null) {
      atLeastOneFail = true;
    }
    if (maxRank<minOverallRank) {
      minOverallRank = maxRank;
    }
  }
  if (atLeastOneFail) {
    return PARTIAL_FAIL;
  }
  if (minOverallRank==1 || minOverallRank==2 || minOverallRank==3) {
    return UNCLASSIFIED;
  } else if (minOverallRank==4 || minOverallRank==5 || minOverallRank==6) {
    return SUBPROPERTY;
  } else if (minOverallRank==7) {
    return TYPE_INVERT;
  } else if (minOverallRank==8) {
    return TYPE_SUBSUME;
  } else if (minOverallRank==9) {
    return MATCH;
  } else {
    return FAIL;
  }
}
```

Figure 3-10 DAML-S output parameter matching algorithm

```
public int match(Vector reqInputsList, Vector advInputsList,
                 Reasoner reasoner) {
  if (advInputsList==null) {
    return MATCH;
  }
  int minOverallRank = 9;
  for (int i=0; i<advInputsList.size(); i++) {
    Input advInput = (Input)advInputsList.elementAt(i);
    Input bestMatch = null;
    int maxRank = 0;
    if (reqInputsList!=null) {
      for (int j = 0; j < reqInputsList.size(); j++) {
        Input tempReqInput = (Input) reqInputsList.elementAt(j);
        int rank = reasoner.rankForParameters(tempReqInput, advInput);
        if (rank > maxRank) {
          maxRank = rank;
          bestMatch = tempReqInput;
        }
      }
    }
    if (bestMatch==null) {
      return FAIL;
    }
    if (maxRank<minOverallRank) {
      minOverallRank = maxRank;
    }
  }
  if (minOverallRank==1 || minOverallRank==2 || minOverallRank==3) {
    return UNCLASSIFIED;
  } else if (minOverallRank==4 || minOverallRank==5 || minOverallRank==6) {
    return SUBPROPERTY;
  } else if (minOverallRank==7) {
    return TYPE_INVERT;
  } else if (minOverallRank==8) {
    return TYPE_SUBSUME;
  } else {
    return MATCH;
  }
}
```

Figure 3-11 DAML-S input matching algorithm

```
public int match(String reqServiceCat , String advServiceCat , Reasoner reasoner) {
  if (reqServiceCat==null || advServiceCat==null) {
    return UNCLASSIFIED;
  }
  if (reqServiceCat.equals(DamlsParser.profileID) ||
      advServiceCat.equals(DamlsParser.profileID)) {
    return UNCLASSIFIED;
  } else {
    int match = reasoner.conceptMatch(reqServiceCat , advServiceCat);
    if (match == Reasoner.EQUIVALENT) {
      return MATCH;
    }
    if (match == Reasoner.SUBSUMES) {
      return SUBSUMES;
    }
    else {
      return FAIL;
    }
  }
}
```

Figure 3-12 DAML-S profile matching algorithm

3.4 Grid security infrastructure

In this section we introduce the Grid Security Infrastructure (GSI), the security component in Globus Toolkit 4 (GT4). The basic design and architecture of GSI enables grid entities to use authentication, authorization, and secure communication over open networks.

In any networked environment, security is a paramount concern. The system must protect itself from outside threats. Because this requirement is especially true for a grid environment where clients can be geographically and organizationally diverse, GT4 meets this need. Basic requirements and motivations for security in any grid environment include, but are not restricted to:

- Secure and tamper-proof communication between grid entities such as users, resources, and programs
- The ability for grid users to use single sign-on capabilities across multiple resources
- Privilege delegation from one entity to another for proxy-like operations
- Interoperability with security mechanisms in place at the participating organizations

GSI is the GT4 component that addresses all these requirements and allows for privacy, integrity, and replay protection for grid communication (to eliminate sniffing and man-in-the-middle attacks), as well as single sign-on and delegation abilities for grid users. It also includes facilities for verifying the identity of a grid entity (authentication) and, based on that, determining the actions the entity is allowed to perform (authorization).

In our system we take advantage of simpleCA [58] in the Globus toolkit 4 (GT4) [4] to set up Globus Grid Security Infrastructure (GSI) [59].

4 Designs

This section gives an overview of the design of our resource discovery service for Grids system. We discuss the main components of our system and identify the functionalities and interdependencies of the components. Here the “component” refers to a function unit, such as matchmaking component.

4.1 System Needs

With more and more development of semantics-based web services on the Internet, the resource discovery system should support semantic description and discovery of grid resources. The functions of “managing” could be registering the service description, monitoring those registered service descriptions, getting service description by given the URI or name of the semantic web service description. “Discovery of grid resources” means by given the search requirement the system has the ability to find the suitable semantics-based web service automatically over the overlay network.

The grid system we design and develop should be a scalable semantic-based resource discovery system. It should have ability to construct a scalable P2P overlay network. Each P2P node holds its own resources and should be able to accept registration of the semantic based resource, monitor these resources. In addition, it can accept user’s request and search the suitable resource over the overlay network, in the end notify the request user automatically.

4.2 Two layers architecture of resource discovery

In our thesis work, the automatic resource discovery is divided into querying, matchmaking, broadcasting and downloading in our thesis work. Querying in our thesis work means to find out what registered descriptions the local node has. After local querying and matchmaking, if there is no suitable result found, the system should broadcast the request and extend resource discovering on whole overlay network. Downloading function used in our thesis work is used to update local repository for matchmaking function. Local repository is a local file directory which stores all the semantic descriptions registered on that node. These descriptions are dynamically registered resources, so we use downloading to reload these resources. When node receives the broadcast message which includes the request message, node compares request and its registered description in local repository and sends back the result to the client. Besides above functions, the system also uses some secure communication, such as only the certain user can use the registration function not all the users. In our thesis work, we use DKS middleware to construct overlay network and provide low level communication.

Based on the system needs, we design two-layer architecture for resource discovery system. Figure 4-1 illustrates the two layers architecture. The super nodes are on upper level and the local nodes are on lower level. Each local node holds its own resources which are semantic descriptions, OWL-Ss and it doesn't know other resources held by other local node. In order to let local node's resources found by other local node, we need to make these resources become public. So we introduce super node. The super node is very important, in that it implements many functions, such as communicating with other super nodes, accepting registrations which come from local node, discovery the resource and monitoring the registered resources. One super node can manage one or more local nodes. Each local node can only registered its resource to one super node. When number of local nodes is one, we can simplify two layers architecture. Removing the local node layer, resource still can be registered to super node directly. By introducing super node, we construct the connection between super nodes and enable resource discovery.

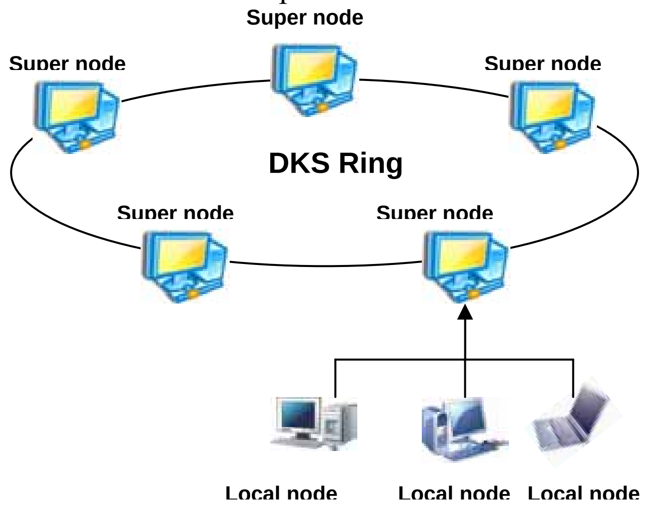


Figure 4-1 Two layers architecture of our resource discovery system

4.3 Terminology

Here are presented the most recurring terms used in this thesis, along with their signification.

Table 4-1 Terminology of Super node

Term	Signification
SIS (Super Index Service)	This service runs in the super node's GT4 container by default, called "Default Index Service". We use this service to monitoring our registered resources. The entire content of the SIS can be seen by executing command "wsrf-query –a –z none –s http://127.0.0.1:8080/wsrf/services/DefaultIndexService" or by executing program. In our thesis work, we integrate this query function into program.
DKSB (DKS broadcast)	This Super node's component is used to construct overlay network which is DKS ring and uses broadcast function of DKS to spread the client's request to the network. It is also responsible for sending the

	matchmaking result to the requester, which is end user.
Repository	A file directory stores the downloaded semantic descriptions which originally registered in SIS. In our program, we name it “downloaded_files”. In addition, “downloaded_files” has another directory named “request file” which is used to store the downloaded request file from SIS or LIS, see Table 4-2 “Register”.
Download	Download semantic description from SIS or Internet to the Repository. We specify description’s name or URI when we download semantic description.
Matchmaker	Given the client’s requirement which is a semantic description, Matchmaker look up Repository and compute the matchmaking degree between client’s requirement and Repository’s semantic description.
NSS (Notification Service for Super node)	This only exists in our simplify resource discovery model, please see section 4.4. It used to locate in local node see also NSL in Table 4-2.
SVL (Super node Value Listener)	This only exists in our simplify resource discovery model, please see section 4.4. SVL is used to listen the notification sent by NSS.

Table 4-2 Terminology of Local node

Term	Signification
LIS (Local Index Service)	LIS is a Default Index Service running in GT4 Container on local node’s side. It is responsible for local resources registration. LIS and SIS can construct an index service hierarchy.
NSL (Notification Service for Local node)	This is a web service we deployed in the GT4 Container. It provides notification service for user. When its status changes, it will contact LVL. It is always running and waiting for matchmaking results from DKSBs.
LVL (Local Value Listener)	End user who starts the request runs LVL to receive the return result from NSL.
Register	Register the local resource to the LIS. Resource is the semantics-based description which wants to be shared on the network. It can also register the end user’s request file to LIS.
Requester	It encapsulates address of NSL, name of the semantic description of client’s requirement and addresses of LIS (optional) which has the registration information of end user’s request file in a message and sends it to its Super node’s DKSB. Address of LIS is optional because if LIS and SIS construct index service hierarchy, super node’s DKSB can get end user’s request file from SIS directly, so it doesn’t need to know the address of LIS.

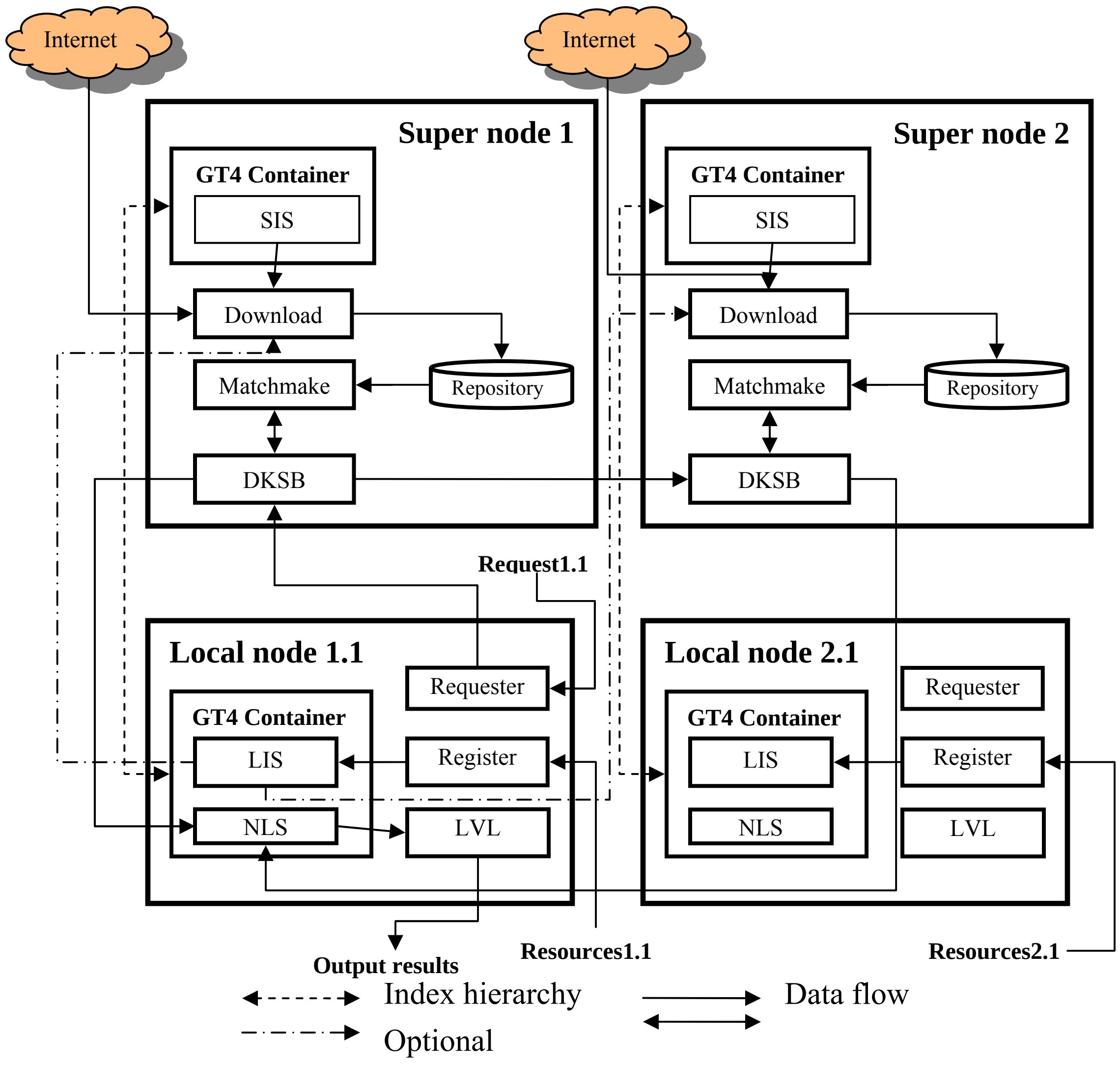


Figure 4-2 Data flow between super node and local node.

Now we give an example to describe how the system works. In the Figure 4-2, we give two super nodes 1 and 2. Each super node has its own local node 1.1 and 2.1 respectively. Local node 1.1 registers local resource to super node 1 and local node 2.1 registers its resource to super node 2. The system will help end user to find semantic descriptions which match end user's requirement on the overlay network. End user now uses Local node 1.1 to start his query. His requirement is registered to LIS1.1 by Requester1.1. Local node 1.1 runs a NSL1.1 in order to receive the matchmaking results from the overlay network and notify LVL1.1 which displays the result to end user. Local node 1.1 starts a query procedure. Its requester encapsulates name of the requirement, address of NSL1.1 and address of LIS1.1 (this is optional) in a message1.1 and sends it to DKSB1. Here we use address of LIS1.1. We say LIS1.1 is optional because SIS and LIS can construct a hierarchy index service. So address of SIS1 can be used instead of LIS in the broadcast message parameters. After parsing out message1.1, DKSB1 knows where to get the end user's request file in order to fulfill matchmaking. DKSB1 goes to address LIS1.1 and downloads the request file to "request file" directory in Repository1 and download registered semantic description from SIS to Repository1. And then Super node1 starts matchmaking and asks DKSB1 to send the matchmaking result of Super node1 to Local node1.1's NLS1.1. In the mean time DKSB1 broadcast this message1.1 to Super nodes in the overlay network. In this example, there are two super nodes, so Super node2 gets the broadcast message by DKSB2. And then DKSB2 does the same things as DKSB1 did before in the end send the matchmaking result to NLS1.1. On local node1.1, its NLS1.1 keeps running and receiving return message from Super node. Here the message is from Super node 1 and 2. NLS1.1 notifies the end user's application LVL1.1 to display all the return results. Through LVL1.1, end user of local node1.1 gets the query result.

4.4 One layer resource discovery model

In section 4.2, we defined that if number of local node is one, two layer architecture becomes one layer. All the functions of local node move to super node. These functions are register, requester, LIS, NSL and LVL. The complete model which is described in section 4.3 LIS and SIS can construct index service hierarchy. So in our simplification model we only use SIS instead of LIS and SIS hierarchy. So all the functions based on LIS move to super node, like register and requester. To simplify the implementation, we move the NSL in local node GT4 container to super node's container and name it NSS (Notification Service for Super node) and the LVL based on NSL becomes SVL (Super node Value Listener). In Figure 4-3, we give an example of query process using simplify model.

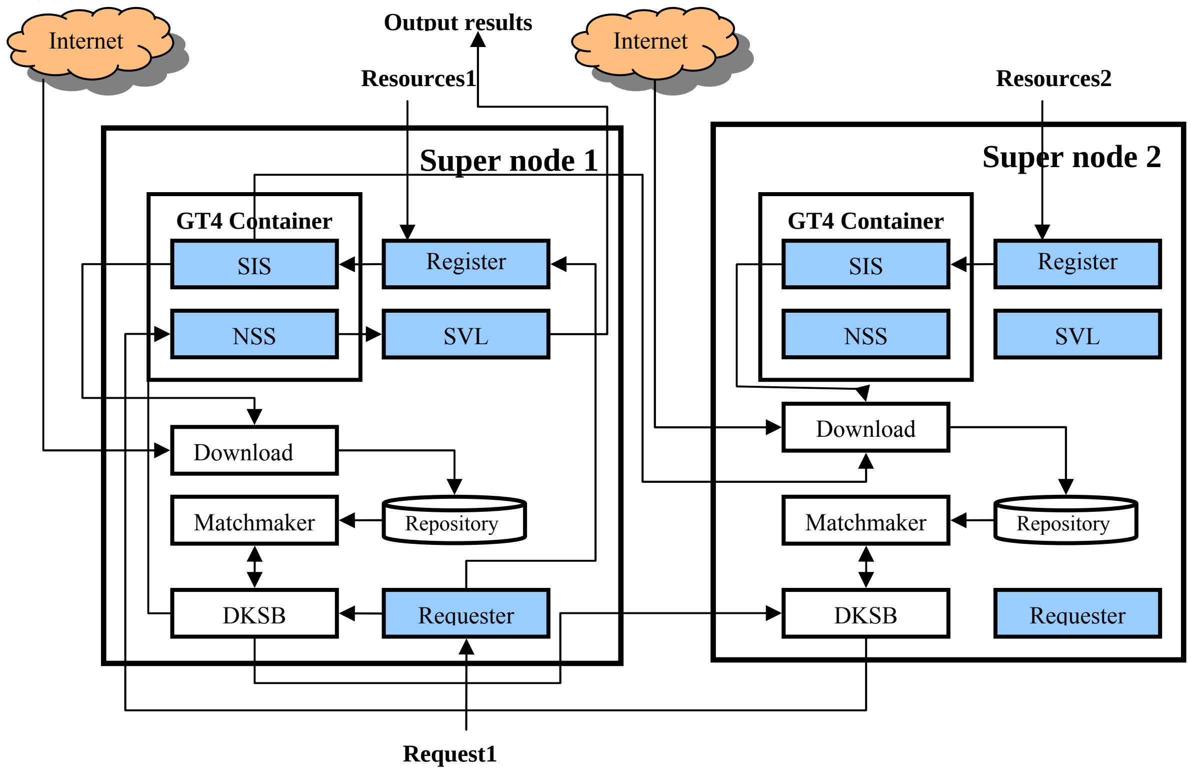


Figure 4-3 Data flow of simplify model

In this example, the blue squares are the new components we add to super node. We remove local node in our new model. Super node takes more responsibilities which used to be the ones taken by local node. End user of Super node1 can register resources to SIS1 and submit query requirement to DKSB1 to super node1. End user1 waits for the query result by running SVL. Super node1 register the end user's requirement to SIS1 in order to let other super node, for example super node2, download it to Repository2's "request file" directory and complete matchmaking on super node2. Request1 has three parameters address of SIS, name of request file and address of NSS. After receiving broadcast message Reques1, DKSB2 in super node2 parse out this Request1 message, and then goes to address of SIS1 to download the request file to its local repository's "request file" directory, start matchmaking and send the result to the address of NSS1. Because SVL1 is the client end of NSS1, so SVL1 receive the change of status in NSS1. End user1 then can see the query results by running SVL1.

Actually we found that when we remove local node, and move the function to super node, we only need to run one GT4 container and the whole system becomes one-layer architecture, see Figure 4-4.

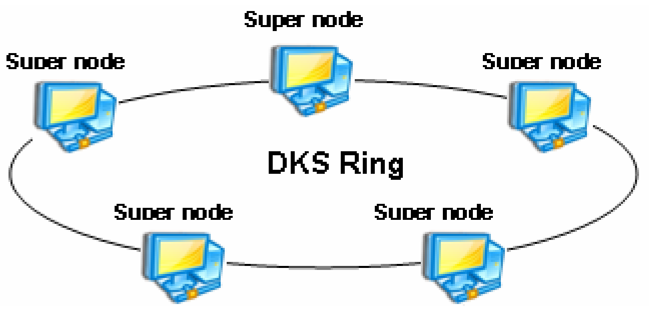


Figure 4-4 One-layer architecture of simple resource discovery model

The system which runs on each super node can be divided in three logistic layers, shown in Figure 4-5. The transportation layer, DKSB belongs to this layer; the service layer which includes NSS, SIS and Register service; application layer include Download, Requester, and Matchmaker. In service layer, Register can be divided into DKS Nodes Register (DKSNR), Request Register (RR), File Provider (FP) and URI Provider (URIP). Services in service layer not only can accept the registration, it can also provide Index Query (IQ) function. Requester of application layer is an abstract notation. It actually integrate many functions together, such as register the end user's semantic description requirement file to SIS, see Figure 4-3, encapsulate address of NSS, address of SIS and name of end user's requirement file in the message and send it to DKSB. We use a graphical user interface (GUI) to manage these functions, and simplify the usage of our system.

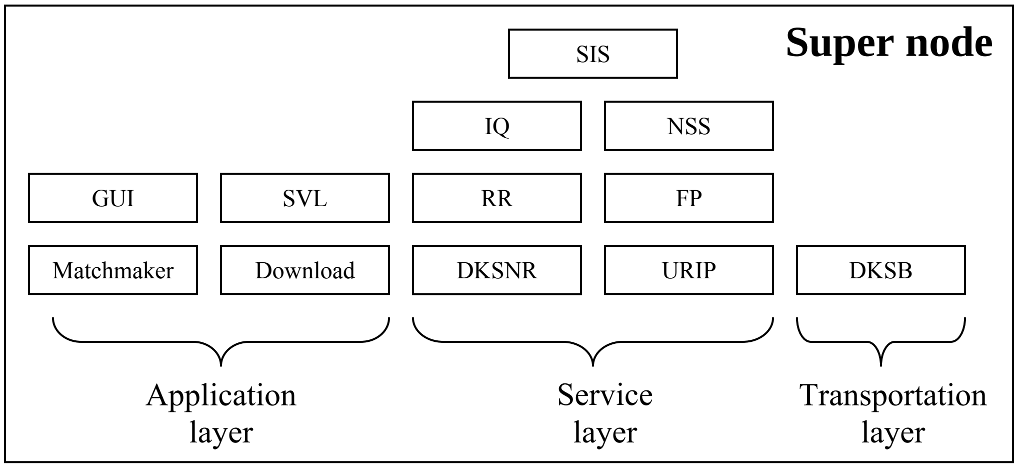


Figure 4-5 Logistic layer of simplify resource discovery model

Here we only give brief explanations the purpose of each component.

1 DKSB on Transportation layer uses DKS middleware to construct overlay network and broadcast request message to the network. This layer receives the message from application layer and then broadcast it, it also receives message from P2P network and sends it to NSS in service layer.

2 Application layer is an application layer. It accepts users input and system configuration, such as the address of the NSS and SIS, the local path of the request file. It also receives the message sent by transportation layer, and then invokes the service layer, for example, after receiving the request message the matchmaker will invoke IQ to know what are the resources registered in SIS. After contacting SIS, matchmaker starts Download procedure. SVL is the end user's client end to get the notification message from NSS. Application layer also provides an easy use user graphical interface which can simplify the usage of our system.

3 Service layer is GT4 Web Services Container. It provides registry, IQ and NSS to the upper layer. This layer is used to monitor all the local registered resources and give an interface to query the status or the value of those resources. Registered resource could be local resources or end user's requirement file, in our work is semantic description. IQ is used to contact with SIS, query the registered resource in SIS. RR (Request Register) can help end user register his requirement semantic description to SIS. FP (File Provider) means if the resource that we want to share stores on local file system, we need such service to make this file searchable. FP registers these local files which is semantic descriptions to SIS. URIP (URI Provider) register the resource which in the form of URI to SIS. DKSNR (DKS Node Register) registers DKS node's reference parameter to one SIS and then if a new super node wants to join overlay network, it lookups one live super node from the SIS, which is running on only one super node. We define the index service which registered all live DKS nodes' reference DKSSIS. DKSSIS can provide live nodes lookup service for other super nodes when they want to join the DKS overlay network. Note that this super node must run before

any other super node in order to receive the registration and lookup join point to join the overlay network.

4.5 Registration using Index Service

In our system the resource which we are looking for is semantic based service description, not the workstation server, high performance computing cluster, OS, CPU, memory or LAN. That means we can use this resource discovery system to register our own semantic description, which can exist in different form like file or the URI. Here in the system we have a function for DKSNRP (DKS Node Reference Provider), because we need to provide a node location service which other super nodes can find the join point to join to the P2P network, here is DKS overlay network.

The default GT4 Information Service (MDS) only provide Hawkeye Information provider, Ganglia Information provider, WS GRAM, Reliable File Transfer Service (RFT), Community Authorization Service (CAS) and any other WSRF service that publishes resource properties. These MDS information providers can only provide us system information. However our system embodies a new feature that is the user not only can register the URI of semantic description, but also the content of the semantic description. Figure 4-6 shows the sequence diagram for this registration.

DKSNRP, FP and URIP will cooperate with SIS to provide resource registration function. The diagram for the service removing action is not presented but is the trivial symmetric of the previous diagram.

Fortunately, we do not need to implement an index service ourselves, as GT4 already includes an index service implementation. In Grid deployments a local index service is often used to maintain a registry of interesting services and/or resources running in that container. In our scenario, a user’s local resources of interest include the content files, URIs and reference of DKS node.

Furthermore, GT4 Index services can also be configured to register with an upstream index which automatically aggregates the data of all downstream indexes. From figure 4-2, we can see that the super node manages some local nodes. Each local node has some resources which want to be discovered. All the nodes, no matter super or local, they all run their own GT4 Index service. Therefore we can construct an index service hierarchy, figure 4-7, which the super node runs SIS, and local node runs LIS. This hierarchy is not included in our simplify resource model.

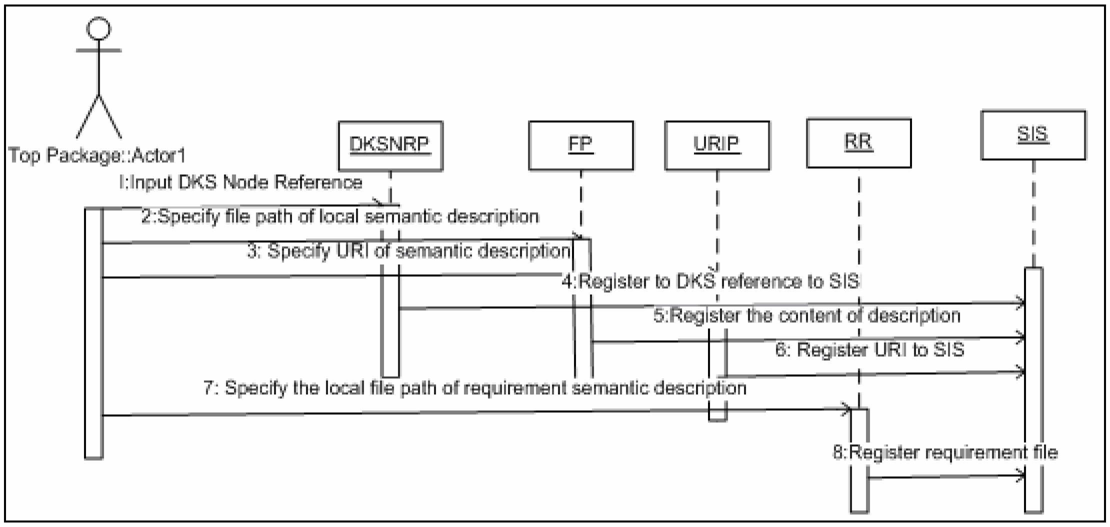

Top Package::Actor1
DKSNRP
FP
URIP
RR
SIS
I:Input DKS Node Reference
2:Specify file path of local semantic description
3: Specify URI of semantic description
4:Register to DKS reference to SIS
5:Register the content of description
6: Register URI to SIS
7: Specify the local file path of requirement semantic description
8:Register requirement file

Figure 4-6 Registering DKS node, file, URI to the index service

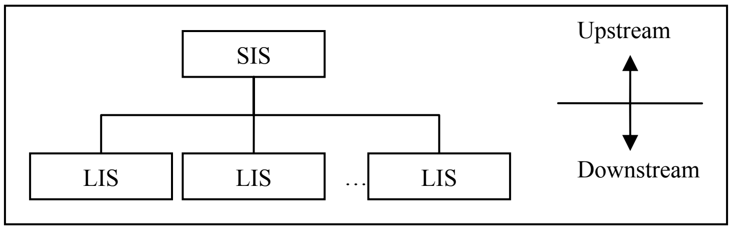


Figure 4-7 Index service hierarchy

4.6 DKSB design: Join DKS and DKS Broadcast

If end user wants to find the semantic description over the network, the precondition is we should construct an overlay network before client sends the query message to the system. This overlay network is implemented by DKS middleware.

These resources are managed by different super node. If the end user wants to find some web services which semantically match his requirement, it not possible to let end user asks the super nodes one by one to find out the matched service if there are so many super nodes over the network. In fact, DKS middleware will take care of all these searching on the network. The only thing the client needs to do is generating a request file which describes what kind of web service he wants to find, sending the request to super node and waiting for the results coming from the entire network. Our solution is using the DKS which is a structured P2P middleware to organize all the super nodes. Super nodes which are DKS's peers can communicate with each other. In our system we use DKS broadcast function, because we can use these function to broadcast the client's requirement to the entire P2P network.

Figure 4-8 shows the progress of joining the DKS ring. As an example, in our sequence diagram, we only create two super nodes in the DKS ring. One super node runs DKSSIS before any other super node is created. The rest nodes look up this DKSSIS to find out the join point of DKS ring.

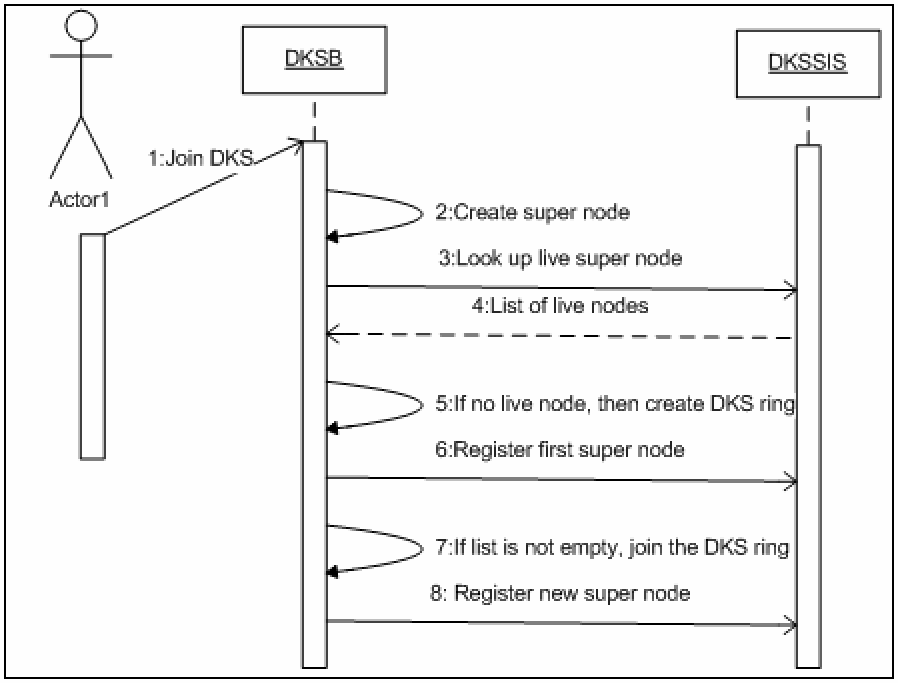

DKSB
DKSSIS
Actor1
1:Join DKS
2:Create super node
3:Look up live super node
4:List of live nodes
5:If no live node, then create DKS ring
6:Register first super node
7:If list is not empty, join the DKS ring
8: Register new super node

Figure 4-8 Join DKS ring sequence diagram

After joining the DKS, the system is ready to broadcast end user's query message. The discovering function can take care of the searching semantic descriptions over the network and return the results to the client. Figure 4-9 shows the progress of discovering. When query start, the system first registers the semantic description, which presents the end user's query requirement to the SIS. The reason we register end user's semantic description to SIS is the system doesn't send the content of end user's semantic description. It only sends the reference parameters of this description. The link means the name of the description, and the location of SIS which holds the content of end user's semantic description. SIS can make the local end user created semantic description become public, and can be found by another super node's DKSB in order to fulfill matchmaking. This progress doesn't show in Figure 4-9.

Figure 4-9 shows the broadcast progress of our resource discovery system. End user first start a SVL1 on Super node1. Then he invoke query function, the broadcast message send to DKSB1, DKSB2 and to until DKSB n. Every super node then starts a matchmaking to compute the degree between the end user's requirement semantic description and descriptions in its local repository. After computing the degree, DKSBi sends back the result to NSS1. NSS1 then notify the SVL1 to display all the returned result from all the DKSBs.

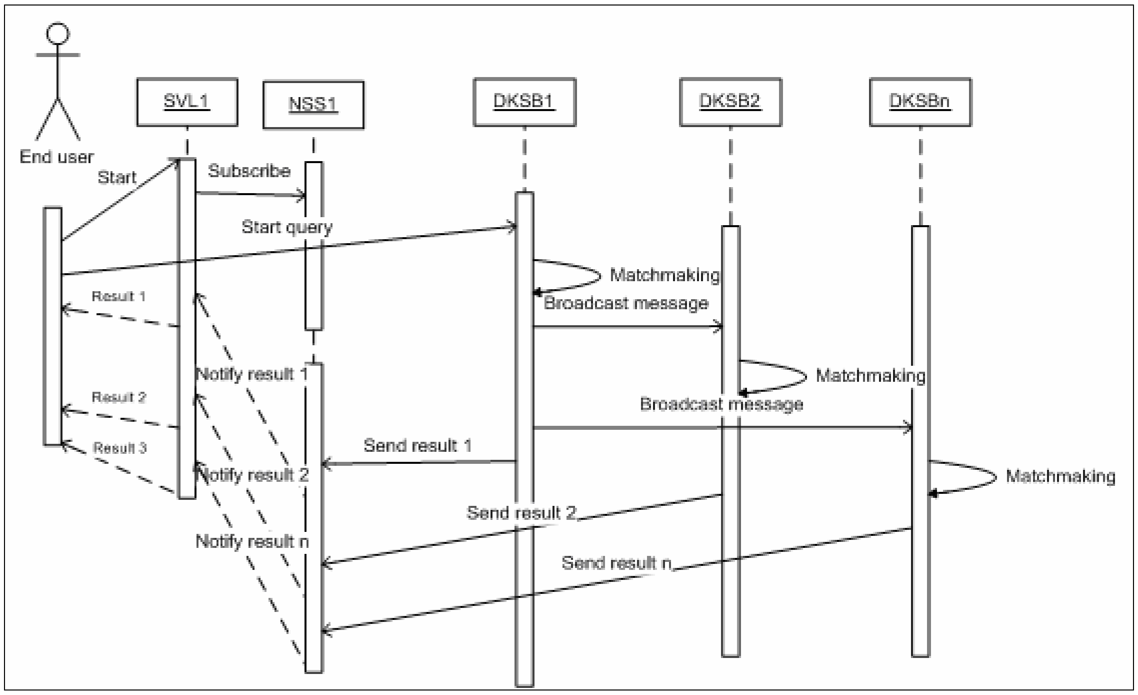


Figure 4-9 DKS broadcast sequence diagram

4.7 Download and update design

In section 4.6, DKSB invokes matchmaking function. Matchmaking function computes the match degree of request resource and advertised resources. The request resource is semantic description generated end user's who uses this description to describe what kind of web service he wants to find. Advertised resources are semantic descriptions registered on super node. Each super node holds its own resource. Super node's SIS is used to monitoring and discovery these semantic descriptions. So if the system wants to complete matchmaking, the first thing that needs to be done is getting these descriptions. In this section we describe the download mechanism.

Figure 4-10 shows how to use download to fulfill matchmaking for DKSB. Number 1 in the figure means the components are running on super node 1. Number 2 means this component is running on super node 2. In this example, end user of super node 2 already registered request resource in SIS2. Super node1 wants to complete matchmaking. So super node 1 use its IQ 1 to get request resource from SIS2 and save it in its Repository "request_file" and downloads advertised resources from SIS1 and Internet and save then in its Repository "downloaded_files". Because in the progress of register advertised resources, end user can choose the resources which already exists in his local disk or choose the resource which is directed by URI from the Internet.

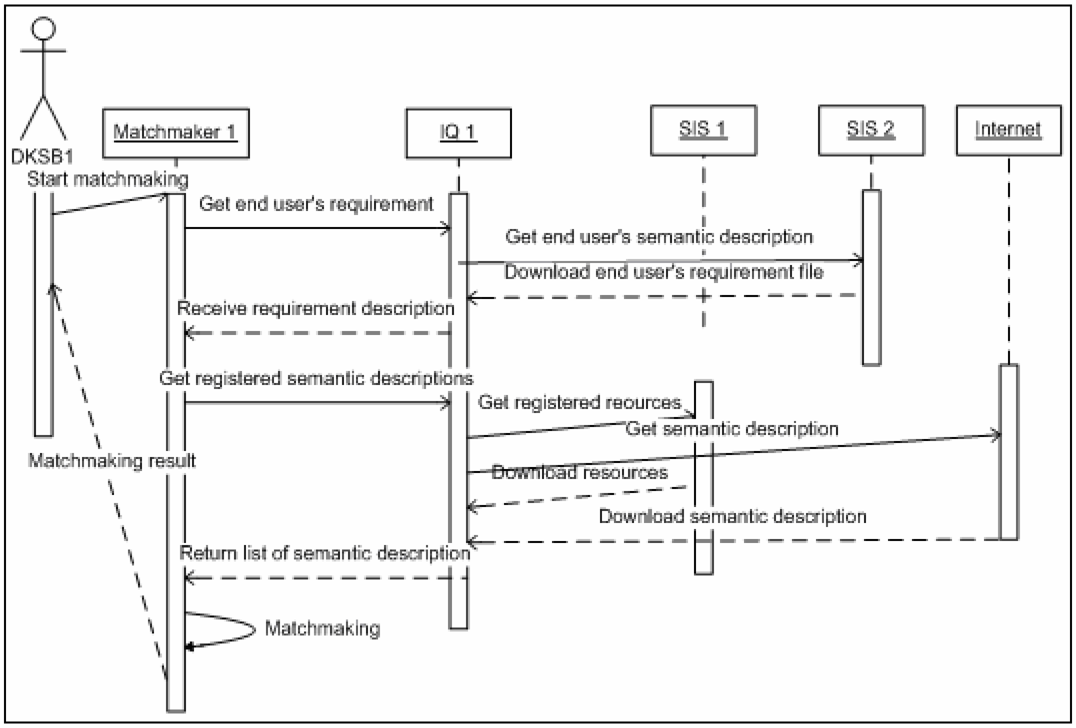

DKSB1
Matchmaker 1
IQ 1
SIS 1
SIS 2
Internet
Start matchmaking
Get end user's requirement
Get end user's semantic description
Download end user's requirement file
Receive requirement description
Get registered semantic descriptions
Get registered reources
Get semantic description
Download resources
Download semantic description
Matchmaking result
Return list of semantic description
Matchmaking

Figure 4-10 Downloading Sequence diagram for matchmaking

SIS is use to monitoring the registered resources dynamically. All the resource can be registered or deregistered in the SIS. End user can execute Update function manually to update its local "downloaded_files" directory.

4.8 Matchmaking Algorithm

The matchmaking algorithm use profile matchmaking described in section 3.3.2. After computing matching degree of input, output and profile of request service and advertised service, we can get scores separately. In order to compute the global final score of matching between request and advertised service, we assign different weight value to the different matching method (Figure 4-11). The final matchmaking result for two considered services is composed of three types of matching, input, output, profile matching. Then compute the final score. If this final matchmaking score is bigger than the MIN_SCORE, which means the result is better than the worst expectation of the end user we save this advertised service and return this service.

```
List match(requestedService, provideServices){
   List matching_Services:={}

   for each providedService in providedServices do
          int currScore:=0;

          currScore:+=weightInput*
              matchInput(requestedService,providedService)
          currScore:+=weithtOutput*
               matchOutput(requestedService,providedService)
          currScore:+=weightProfile*
              matchProfile(requestedService,providedService)

        if currScore>MIN_SCORE then
              matching_Services:=
                     matching_Services.providedServices
        end if
  end for

  return matching_Services
```

Figure 4-11 Service matching algorithm

4.9 Registration design

From Information flow in MDS4 in Figure 3-1, there are an execution source in the *Aggregator source* and an Index Service in the *Aggregator sink*. In Figure 4-4, DKSNR, RR, FP and URIP need registration to register certain information to SIS. In this section, we introduce a registration method provided by GT 4.0 [62]. GT 4.0 use external software component execution source to provide us information [62]. There is a current method in GT 4.0 combines execution source and Index Service together to provide us a non web-service based information service. Using this method we can generate our own information provider to provide resource in order to let Index Service monitor and discovery.

The registration method for DKS Node Reference and URIP are the almost the same. Here URIP provides the link of semantic description that the end user wants to be shared. Registration method can use current GT 4.0 [62] method to register it in SIS. However the way we register the whole content of semantic description to SIS is a little bit different. We will introduce this very useful registration method provided by GT 4.0 in the following.

All the information that we want to register in SIS is not the certain type of the information, such as CPU load, free space. We register DKS node's reference or URI of the semantic description to the SIS. The URI of the service description is as following:

http://www.mindswap.org/2004/owl-s/1.1/GetWeather.owl

In our work, we use Execution Aggregator Source as our Aggregator source. This aggregator source is used for gathering arbitrary XML information about a registered resource by execution an external script. This is mostly useful for scenarios where the user would like to publish information into the MDS4 from a non web-service based information source.

Now we give an example to show how to register a URI of the semantic description, OWL-S document to the SIS. The registration for URIP (URI Provider) is described below:

1. Decide which information you would like to have published. In registration method of [62], this information should be in XML format. Figure 4-12.

```
<OWLURIs>
   http://www.mindswap.org/2004/owl-s/1.1/GetWeather.owl
</OWLURIs>
```

Figure 4-12 Simple schema for publishing

The information between the <OWLURIs> and </OWLURIs> is URI which we want to publish. The tag <OWLURIs> and tag </OWLURIs> is defined by developer and it will become useful when it comes to retrieve.

2. Write a script that gathers and formats the resource that we want to register to SIS. This can be C code, shell script, perl code, etc, and it doesn't matter what kind of methods it uses behind the scenes, so long as it produces well formatted XML data. If we wanted to publish a list of URIs into the SIS, we could write a simple shell script (Figure 4-13) to retrieve it and format it into our chosen XML schema.

```
#!/bin/bash
#OWL-Provider.sh

TMPFILE="/tmp/myowls/ServiceList.txt"

echo "<OWLURIs>"
if test -f $TMPFILE; then
    URIs_DATA=`cat $TMPFILE`
    echo $URIs_DATA
fi
echo "</OWLURIs>"
```

Figure 4-13 Simple shell script

TMPFILE variable defined by developer points to a file which records list of URIs that we want to publish. The code between the "if" and "fi" is used to read the content of file and print it out. The content is the information we want to publish. For example, we write the URIs into file directed by TMPFILE in advance, and then when the shell script is executed it can read out URIs and register it into SIS. If the content of file under TMPFILE is the content of semantic description, we can also register it into SIS. So we can put any information into this part except the XML format information. If we want to publish XML format information, we need to change the format before we register it into Index Service. We will introduce this method later.

3. Establish mapping of your information provider in jndi-config.xml. To establish the mapping of our provider, we need to edit the $GLOBUS_LOCATION/etc/globus_wsrf_mds_index/jndi-configure.xml file. To add our *OWL-Provider.sh* file, we call it *owlPorvider* as the mapped name. Our entry would then look like this:

```
owlProvider=OWL-Provider.sh
```

We need to add above sentence into the executable Mappings section that looks like this, shown in Figure 4-14:

```
<parameter>
  <name> executableMappings</name>
     <value>
        aggr-test=aggregator-exec-test.sh,
         pingexec=example-ping-exec,
         owlProvider=OWL_Provider.sh
     </value>
</parameter>
```

Figure 4-14 Partial code of Executable mapping section in jndi-config.xml

4. Copy information provider to $GLOBUS_LOCATION/libexec/aggrexec directory. Make sure the shell file in this directory has proper executable permissions, for example, the permission of this shell file should look like this “-rwxr-xr-x”.

5. Configure the registration file. This step tells us how to make the registration to the SIS. To do this, we’ll need a registration file. Registration file is in the appendix A. In this file, we define the location of the index service that we want to make the registration to, the time interval that refresh the registration, the time interval that we execute the specified provider and probe name that is where the executable mapping is put to use. In registration file, time interval for refreshing the registration means how long the SIS runs the registration. If using “mds-servicegourp-add” utility, the system refreshes the registration automatically. In this example probe name is ”owlProvider”. Time interval that refresh the registration, the time interval that execute the specified provider and probe name are specified in registration file, we can use the default setting to set our own value.

6. Register with SIS: run mds-servicegrout-add. To make the registration of our Execution Aggregator provider URIP to the SIS, we should run the mds-servicegroup-add program in a similar manner:

```
 $GLOBUS_LOCATION/bin/mds-servicegroup-add –s \
https://127.0.0.1:8443/wsrf/services/DefaultIndexService \
/home/yeou/workspace/YangYeou_SourceCode/registration-schemas/owls-provider-
registration.xml
```

7. Query. There are two ways of query. One is using the command line like:

```
yeou@yeou:~$ ./wsrf-query –s https://127.0.0.1:8443/wsrf/services/DefaultIndexService \
"//*[local-name()='OWLURIs']"
```

The other method is using the program. For example, the Appendix B shows the code of getAliveNodeList method. The following code is one method of QueryIndexService.java. It is used to get the all the alive nodes reference from Index Service. The getting information is stored in the list allREFs.

Now we already use above 7 steps to register the URI of service description and the reference of the DKS node. Now we explain how we register the content of semantic description into SIS and download it into the local disk. The procedure of publishing the file to the Index Service is almost the same as the previous steps. The only difference is the format of the content in the file pointed by TMPFILE variable. Before the format is plain text "http://www.mindswap.org/2004/owl-s/1.1/GetWeather.owl", now we want to change it into the content of OWL-S semantic description. Here there is one thing we should notice. We can not write the whole OWL-S file into the TMPFILE, because format conflict. Every time the system use IQ to query the SIS, the query result is in XML format. The IQ will get a XML file which includes the registered resources' information. So we can not make our own information be in XML format. Because both of the OWL-S and the returned xml file have the root elements which are <?xml version="1.0">, this can cause registration fail. "mds-servicegroup-add" can return an error, in other words the content of OWL-S can not be registered into SIS. So we must do some preparation for our registration. We need to change the format of the file. The way we do is like this, we remove the root element of the OWL-S file that we are going to register, and write this changed file into TMPFILE, then use the previous steps to register. And then we run step 6, the content of file in TMPFILE can be registered to SIS. The content is the OWL-S content without root element. When we need to download that, we can get this content which without the root element, and then add the root element in front of the content. We use file name to tag our OWL-S content. By doing so, we can retrieve the file by name. So we add a FileUnite class into our Registration package to let it modify all these files.

Depending on our need, we design four registrations. They are registration for the content of local advertised semantic descriptions (FP), registration for URI links of advertised semantic description (URIP), registration for end user's requirement semantic description (RR) and registration for DKS node reference (DKSNR). Each of four registration needs to use the above seven steps to fulfill registration. As long as we start these registrations, here we mean running the “mds-servicegroup-add” four times, the only difference is the parameter after this command, the SIS can monitoring these content automatically.

We can see that if end user has many semantic descriptions of web services which want to be registered and we create a TMPFILE to store the content of one description for every description and then run the whole seven steps to register the content into SIS that will reduce the running efficiency. For example if we have ten local semantic description resources wants to be registered, we have to create a TMPFILE for each resource, write new shell script, move the script to certain GT4 directory…. We have to do these ten times. All the end user can not accept this. It’s too complicated and will waste our time. So we introduce File uniting function to help us turn many times of repeated operation into one. We do like this: put all the content of semantic descriptions into one TMPFILE. Here we use a little trick, we have described before. Now TMPFILE has all the content of semantic descriptions. The content in the TMPFILE is divided by a name tags which specify which content belongs to which file. Now we are ready to start our seven steps registration procedure. We only need to do one time registration; in addition, it’s very easy to fulfill the dynamic resource registration. The only thing we need to do is rewrite the content of TMPFILE after we starting our registration. All these are done by Registration class and FileUnite class. We will describe the methods of these two classes in section 5.3.

5 Prototype implementation

Our goal in this chapter is to explain some details of the implementation. We will organize our implementation around the various Java packages:

- DKS_broadcast: It includes a class which is used to construct a P2P overlay network, send and listen to the broadcast message. This class can also invoke the matchmaking function and send the result to the client by the notification.
- MDS: This package includes one class called QueryIndexService which is used to contact with GT4 Index Service and query the registered information.
- Registration: It includes two classes FileUnite and Register, which are used to register URI, service description, DKS node reference and request file to Index Service
- Matcher: The classes in this package work together to provide matchmaking function.
- Download: It includes one user defined class which is used for downloading the specified service description from Index Service or Internet.
- Notification: This is a custom GT4 implementation of WS-Notification. On the server side, a notification service can be run in the GT4 container. On the client side, a client runs a value listener and subscribe to a notification service topic. When a certain resource property is modified, a notification is triggered.
- GUI: The class in this package creates a GUI platform with Swing.

In matcher package, for reasoning purposes, have been used: Pellet 1.2, Jena 2.2; for read and write service, have been used: OWL-S1.1 API which is given by Maryland Information and Network Dynamics Lab Semantic Web Agents Project , for xml parsing purposes: jdom 1.0. For DKS broadcast purpose, have been used DKS, a peer-to-peer middleware developed at KTH/Royal Institute of Technology and the Swedish Institute of Computer Science (SICS). MDS, Registration and Download packages are based on the GT4 Information service MDS4. GT4 also provides us a set of standard interface to use the notification design pattern with Web Service.

5.1 DKS_broadcast package

The DKS_broadcast package holds the modules in charge of broadcast service. In order to broadcast client's requirement to the P2P network, we need to construct the P2P network, send the message to all the nodes in the network and every node should listen to the network in order to receive the broadcast message. Thanks to the DKS middleware, it provides us a DKS API which has implemented all these functions. The My_DKS provides the following functionalities:

● *joinDKS*. DKSB uses this method creates and registers a DKS node, and then joins this super node to the DKS ring automatically. After joining the network, DKSB can listen to the broadcast message coming from the client and parsing the request message. It calls other methods in order to complete matching and notification function. For example, DKSB receive the query message which includes name of the end user's requirement semantic description's name and address of the SIS which holds the content of requirement semantic description, then goes to that SIS to download the requester's semantic description in order to fulfill the local matchmaking. After completing matchmaking, it will contact the notification service to notify the matchmaking result. Appendix C shows the part of the source code of joinDKS(). Please Class Register and ClientAdd in the code will be described later.

● *my_broadcast.* DKSB use this method to broadcast query message. This method has tree parameters. The first one is the name of the request file. This request file is created by end user and it describes the desired semantic web service he wants to find. The second address of SIS which DKSB wants end user's request file to be registered to. The last parameter is the address of the notification service (NSS) that the super node which the end user is using. The format of the broadcast message is as following:

```
String queryMsg=filename+"&"+AddressOfSIS+"@"+NotificationService
```

Later after receiving the query message the DKS node can parse this message in order to complete download request file and notify the client.

5.2 MDS package

This component is responsible for getting the information which has been registered in the SIS. Based on the information it gets, it can fulfill download and matchmaking. This information includes the super nodes list, get the list of URI of service description and file names. This package also invokes other package to implement download and matchmaking functions. It can not only query SIS in order to get the name or URI of the registered resource from SIS, but also can download the content of registered resource. It can download the content of registered semantic description, such as the content of advertised semantic description and the end user's requirement semantic description. It also provides us a file transfer function for transferring from the URI of semantic description to object OWLOntology.

Table 5.1 QueryIndexService class

QueryIndexService class
Constructor
`QueryIndexService(String service, String xPathForURI, String xPathForFileList)`
Methods
public void contactIndex()
public List<String> getAliveNodeList(String remoteIndexAddress)
public List getAllRegServices()
public Vector <OWLOntology> toOWLS(List<String> allURIs)
public OWLOntology toOWL(String URI)

public void downloadAllOntologies(String remoteIndexSite)
public void downloadFile(String remoteIndexSite,String fileName)
public OWLOntology downloadRequestFile(String remoteIndexSite,String fileName)
public Vector update(String remoteIndexSite)
public Vector match(OWLOntology request)
public Vector match(String request)
public Vector matchDownloadedRequest()

Constructor:

- *QueryIndexService(String service, String xPathForURI, String xPathForFileList)*. The first parameter *service* specifies the address of the SIS. *XPathForURI* and *xPathForFileList* are the XPath statements used to query the returned XML file from Default Index. *XPathForURI* is used to retrieve the registered URIs of the service description from the Index Service. *xPathForFileLis*t is used to get the list of registered file in the Index Service.

Methods:

- *contactIndex()*. Get the list of URI of service descriptions and file name list of the service descriptions.
- *getAliveNodeList(String remoteIndexAddress)*. Download the alive DKS nodes' references from DKSSIS.
- *getAllRegServices()*. Get all the URIs of registered services.
- *toOWLS(List<String> allURIs)*. Given a list of URIs of the service descriptions, the function can transfer them to corresponding service descriptions. This method transfers the URIs to reader created by OWL-S API `OWLFactory.createReader` and generages OWLOntology.
- *toOWL(String URI)*. Given the service URI, the function can return a corresponding OWLS-S description.
- *downloadAllOntologies(String remoteIndexSite)*. Download all the content of semantic description from SIS and Internet, save them on the local Repository and transform these description to OWLOntology type objects. *remoteIndexSite* means the address of SIS. In our work we define Ontology is the semantic description.
- *downloadFile(String remoteIndexSite,String fileName)*. Download the specified file by giving the name of the file. This method downloads the file which registered in SIS. The file could be the end user's requirement semantic description or the advertised semantic description registered in SIS. This function will use Download class's getFile method.
- *downloadRequestFile(String remoteIndexSite,String fileName)*. Download the specified request file by giving the name of the file. This function will call Download class's getRequestFile method. *remoteIndexSite* is the address of the requester's SIS. End user has already registered his request file into this SIS. So if we want to complete the matchmaking, we need to download the requested service description.
- *update(String remoteIndexSite)*. The registered resources held by Super node are dynamically changing. Update method contact the SIS to get the fresh registered resources; here resources are semantic descriptions of the web services. Clear the local Repository "downloaded_files" directory and then download the registered service description again. It invokes some methods of download class.

- *match (OWLOntology request)*. The end user's requirement semantic description has been transformed into OWLOntology object before we invoke this method. It will invoke the ProfileMatcher class's match (request, allOWLS) method and return match result. Given the requested service description, super node can compare it with the local registered service.
- *match (String request)*. End user publishes his requirement on the Internet and give this resource a URI link. Given the URI of the request service description, it can also complete matchmaking and return match result.
- *matchDownloadedRequest()*. If we have already gotten the request file by invoking *downloadRequestFile* method, we can call this *matchDownloadedRequest()* method and then return match result.

5.3 Registration package

Registration package is used to prepare the TMPFILE which is described in Section 4.7. The content of file pointed by TMPFILE will be registered to SIS. Each information provider, DKSNR, RR, URIP and FP has their own TMPFILE. Registration package also has a Storage directory which is used to store the local semantic description files. All the files in this Storage directory will be first modified and unite in one TMPFILE, and then the content of TMPFILE is registered to the SIS. Registration means we can register the URI, advertised semantic description and client's requirement semantic description file to the SIS.

Table 5-2 Registration class

Registration class	
Constructor	
`Register()`	
Methods	
public void register(Collection services)	private void writeFile()
public void register(String service)	public static void copyFile(String srcFile)
public void regDKS(String ref)	public boolean isItInStorage(String service)
public void regRequest(String localPath)	public Vector<String> getServices()
public void unregister(Collection services)	private Collection removeServiceFromVector(Collection regServices,String service)
public void unregister(String service)	

- *register(Collection services)*. Register the java collection of advertised services. The element of the services collection is in String type. This String of service can present two meaning URI and local path of the advertised service description. It calls two other methods register (String service) and writeFile().
- *register(String service)*. It is called by *register(Collection services)* method.
- *regDKS(String ref)*. This method writes the reference of DKS node to the file MY_DKS_REF. This file is directed by variable TMPFILE. This file is the local file

which records all the live nodes' references. Then DKSB can use this file to register the reference of its super node to SIS.

- regRequest(String localPath). Register the local request file to end user's SIS.
- *unregister(Collection services)*. Deregister the collection of the semantic descriptions. It invoke unregister(String service) method. It rewrite file directed by TMPFILE.
- *unregister(String service)*. This method removes the content of the semantic description from TMPFILE. The parameter service is the local file path of semantic description.
- *wrieFile()*. This method is used to rewrite the content of file which specified by TMPFILE. It invokes copyFile(String path) and unite() of class FileUnite.
- *copyFile(String srcFile)*. Copy the file from path srcFile to the Storage directory.
- *isItInStorage(String service)*. Return true if file is belonged to Storage. Otherwise return false.
- *getServices()*. Returns a list of registered service description links, such as URI and local path of the file.
- *Collection removeServiceFromVector(Collection regServices,String service)*. Collection stores all the registered semantic description's file path and URI. Remove the service from the collection of services.

Table 5-3 FileUnite class

FileUnite class
Constructor
FileUnite(String fileName,String filterName)
Methods
public void unite()
public void FileUniteModify(Vector<String> absoluteFiles)
public void requestFileModify(String reqFile)

Constructor:

- *FileUnite(String fileName,String filterName)*. The first parameter is the destination file which unites all the local advertised files stored in Storage directory. The second parameter *filterName* is the file type. It specifies the certain type of files can be united, for example ".owl". The files are stored in Storage directory by default.

Methods:

- *unite()*. Unite all the files in the Storage directory into one file. This file will be used by FileUniteModify(Vector<String> absoluteFiles). After invoking this method, it generates a new output file which includes the whole content of the files, here we call it FILE_UNITE. The file in the Storage is written to the FILE_UNITE one by one.
- *FileUniteModify(Vector<String> absoluteFiles)*. Modifies the format of the file FILE_UNITE and generate a new file named *owlModify.txt* in order to register this file into SIS. We add some tags and remove the XML declaration of each file in the FILE_UNITE. So we can retrieve each file when we query the Default Index Service.
- *requestFileModify(String reqFile)*. Modifies the format of the end user's requirement semantic description. *reqFile* is the file path of the request file. In this method, we

generate a new file called *RequestOWL.txt*. It is directed by TMPFILE which is used for registering the request file to the SIS. All the content will be registered into that service. The way we change our request file is the same as the way we use in *FileUniteModify* method.

5.4 Matcher package

The matcher package holds the modules in charge of matchmaking service. It also includes some other classes which are used by the service matcher. The matching method uses Profile matcher method. The implementation of the profile matcher follows carefully the algorithms specified in section 3.3.2 and section 4.6.

Table 5-4 ProfileMatcher class

ProfileMatcher class
Constructor
ProfileMatcher()
Methods
public void addServices(Collection services)
public void clearKb()
public Vector match(OWLOntology request, Collection advertisements)
public Vector match(OWLOntology request, Collection advertisements,int minScore)
public int match(Profile request, Profile advertisement)
public Vector matchCategory(OWLOntology request, Collection advertisements)
public int inputMatch(Profile request, Profile advertisement)
public int outputMatch(Profile request, Profile advertisement)
public int profileMatch(Profile request, Profile advertisement)

- *public void addServices(Collection services)*. Add a collection of services to the knowledge base
- *public void clearKb()*. Clears the knowledge base
- *public Vector match(OWLOntology request, Collection advertisements)*. Returns a collection of OWLOntology type objects contained in *advertisements*, which these objects match request with a score greater or equal than a default score specified in the configuration file.
- *public Vector match(OWLOntology request, Collection advertisements,int minScore)*. Returns a collection of OWLOntology type objects contained in *advertisements*, which match request with a score greater or equal than *minScore*.
- *public int match(Profile request, Profile advertisement)*. Match request against advertisement, and return the score obtained.
- *public Vector matchCategory(OWLOntology request, Collection advertisements)*. Returns a collection of services contained in *advertisements*, which category match *request*'s category.

- *public int inputMatch(Profile request, Profile advertisement)*. It compares the request profile's input with the input of the advertisement's profile, and the returns the score of input matching.
- *public int outputMatch(Profile request, Profile advertisement)*. It compares the request profile's output with the output of the advertisement's profile, and the returns the score of output matching.
- *public int profileMatch(Profile request, Profile advertisement)*. It compares the request profile with the advertisement profile, and the returns the score of profile matching.

Table 5-5 Machmaking class

Matchmaking class
Constructor
Matching()
Methods
public void addService(URI service)
public void addService(OWLOntology service)
public void clearKb()
public int conceptMatch(URI conceptA, URI conceptB)
public int propertyMatch(URI propertyA, URI propertyB)
public int scoreMatch(Parameter req, Parameter adv)

`Constructor`

- Matching(). Create a new reasoner.

`Methods`

- *public void addService(URI service)*. Add a service to the knowledgebase.
- *public void addService(OWLOntology service)*. Add a service to the knowledge base.
- *clearKb()***.** Clears the knowledgebase.
- *public int conceptMatch(URI conceptA, URI conceptB)*. Returns the score of a concept match between the two concepts.
- *public int propertyMatch(URI propertyA, URI propertyB)*. Returns the score of a property match between the two properties.
- *public int scoreMatch(Parameter req, Parameter adv)*. Computes the score obtained when matching the two parameters.

The matcher package also contains static functions used to read and write OWL-S files. The reading functions return OWLOntology object from an input URI where a service is located or from a String containing an OWL Service. The writing functions write OWLOntology objects (OWL Services in the OWL-S API) into Strings (basically so that the service can be sent on the wire).

5.5 Download package

Download package includes a Download class and "*downloaded_files*" directory. "*downloaded_files*" stores the complete OWL-S files which have been downloaded from SIS and has two subdirectories called *request_file* and *temp*. *request_file* stores the request file downloaded from the service Default Index Service which saves the client's request file. *temp* directory stores the downloaded files. Here all the files in the *temp* are files without XML declaration, we will modify these files to the final OWL-S files and save them in the *downloaded_files* directory.

Table 5-6 Download class

Download class
Constructor
public Download(String remoteIndexSite,Vector<String> list)
public Download(String remoteIndexSite,String fileName)
Methods
public void getFile(String file_name)
public OWLOntology getRequestFile(String remoteSite, String file_name)
public void getAllOntologies(Vector<String> serviceList)
public Vector<OWLOntology> getOntologyStorage()
public void update(Vector<String> serviceList)

Constructor

- *public Download(String remoteIndexSite,Vector<String> list).* The parameter *list* stores the names of the files that we want to download. *remoteIndexSite* is from where we want to download the *list* of files.
- public Download(String remoteIndexSite,String fileName). Initialize the file name and address of the SIS.

Methods

- *public void getFile(String file_name).* Download the file from the Default Index Service to the local disk. Save the file in the *donwloaded_files* directory.
- *public OWLOntology getRequestFile(String remoteSite, String file_name)*. Download the end user's request service description from the end user's SIS which its address is specified by *remoteSite* parameter and then returns the object of that request file. This object is an OWLOntology type of OWL-S document.
- *public void getAllOntologies(Vector<String> serviceList).* Download the list of the files to the "*donwloaded_files*" directory.
- *public Vector<OWLOntology> getOntologyStorage()*. Return the list of the downloaded OWLOntology.
- *public void update(Vector<String> serviceList).* Clear the DOWNLOAD_DIRECTORY and FINAL_DOWNLOAD directory, and then download the service description. The *serviceList* has the links of the service descriptions. This method will download them again from the Default Index Service. This method is called when the client starts matchmaking or press the update button.

5.6 Notification package

This package provides us a notification service and value listener application. The notification service is deployed in the GT4 container, and on the value listener subscribes a topic and start listing to the change of this topic. The WSDL file for notification service is in the appendix D. If on the server side the value of this topic changes, the server will notify the client automatically. The WS-Notification interaction is shown in the Figure 5-1.

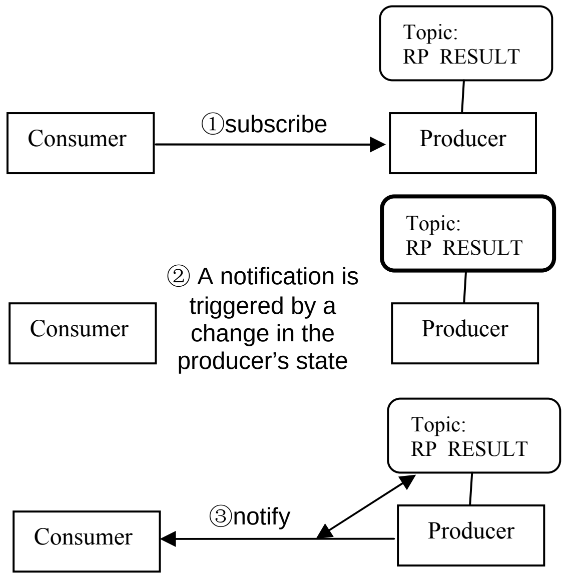


Figure 5-1 A typical WS-Notification interaction

We need two clients to fulfill notification. The first client is in charge of *listening* for notifications. The second client is used to add the result to the Notification Service. The second client is integrated in the DKS_broadcast package. It uses this client to notify the NSS running on the end user's super node.

The listener client is composed of two important parts:

1. Subscription: This block of code is in charge of setting up the subscription with the Result RP. Once the subscription is set up, this block of code simply loops indefinitely until we press a key.
2. Delivery: Once the subscription has been set up, and the main thread of the program is looping infinitely, the delivery code gets invoked any time a notification arrives at the client.

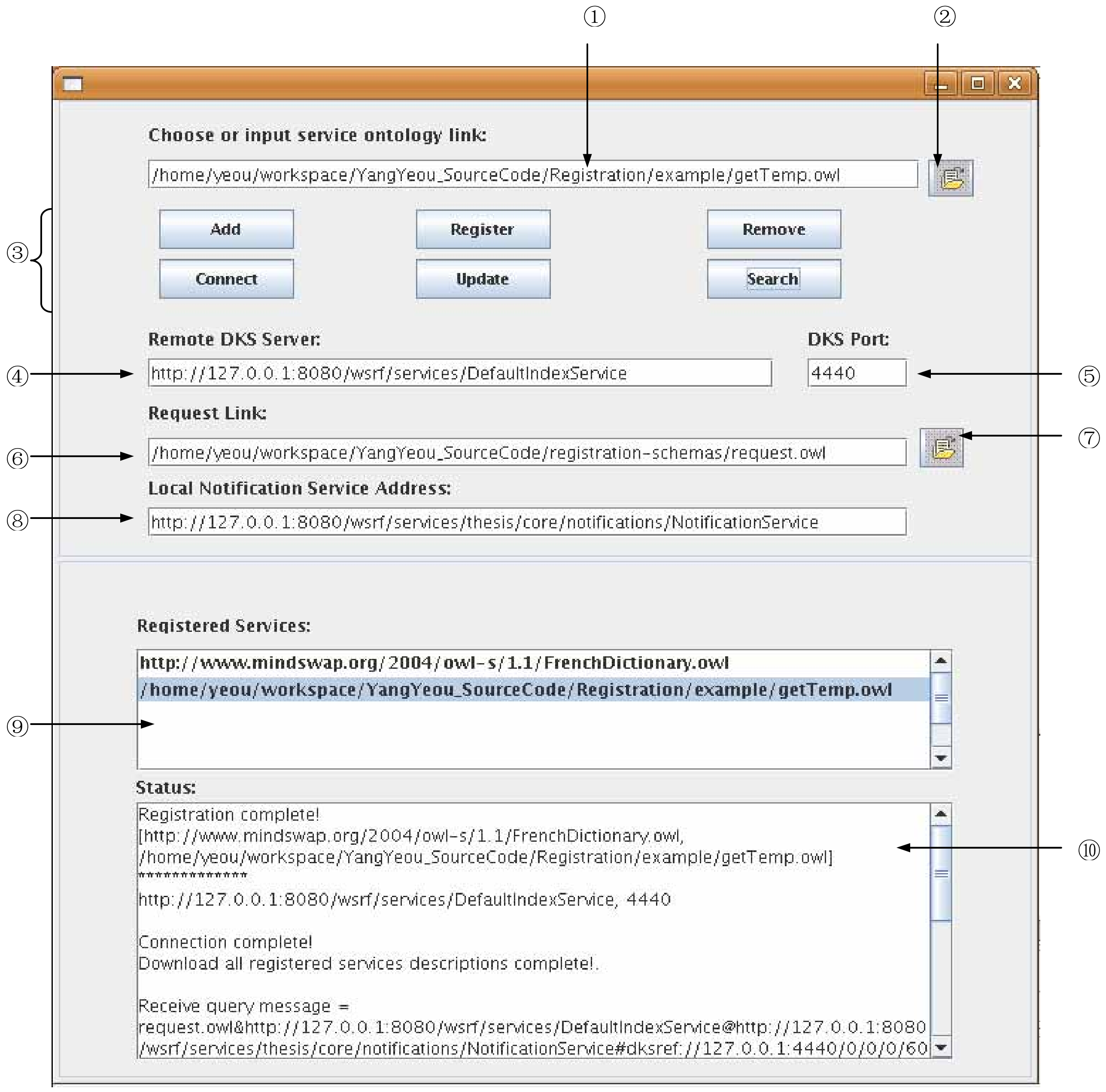
5.7 GUI package

Our resource discovery grid system is a P2P application, and runs under Linux operating system. Each node is both server and client. Figure 5-2, is the snapshot of our system. The client end for notification service uses command line interface.

Explained snapshot of resource discovery system GUI:

① Path of the advertised service description. This path can be the local file path or the URI of the service description on the Internet.

② Open a file explorer to select the advertised service description file.



```
/home/yeou/workspace/YangYeou_SourceCode/Registration/example/getTemp.owl
http://127.0.0.1:8080/wsrf/services/DefaultIndexService
4440
/home/yeou/workspace/YangYeou_SourceCode/registration-schemas/request.owl
http://127.0.0.1:8080/wsrf/services/thesis/core/notifications/NotificationService
```

```
http://www.mindswap.org/2004/owl-s/1.1/FrenchDictionary.owl
/home/yeou/workspace/YangYeou_SourceCode/Registration/example/getTemp.owl
```

```
Registration complete!
[http://www.mindswap.org/2004/owl-s/1.1/FrenchDictionary.owl,
/home/yeou/workspace/YangYeou_SourceCode/Registration/example/getTemp.owl]
**************
http://127.0.0.1:8080/wsrf/services/DefaultIndexService, 4440

Connection complete!
Download all registered services descriptions complete!.

Receive query message =
request.owl&http://127.0.0.1:8080/wsrf/services/DefaultIndexService@http://127.0.0.1:8080
/wsrf/services/thesis/core/notifications/NotificationService#dksref://127.0.0.1:4440/0/0/0/60
```

Figure 5-2 GUI

③ Functions of buttons:

Add button. Add ① to the ⑨. It generates a registered service descriptions list which will be registered to the Index Service.

Register button. Register the list of service descriptions stored in ⑨ to the Default Index Service. The address of this service is specified in the registration file. So we didn't specify in our interface.

Remove button. Remove the registered service descriptions which are chosen in ⑨ by mouse and then reregister the left service to the Default Index Service.

Connect button. When it is clicked, the application downloads the registered service descriptions from Default Index Service to the local disk first. Then it creates a DKS node and joins itself to the DKS ring and waits for the broadcast message sending from other node.

Update button. Update the local OWL-S files directory which is used to store the downloaded file from the Default Index Service or from Internet. It first clears the local download file directory, and then downloads the registered service description again.

Search button. Register client's request file to the Default Index Service, and then broadcast the request message to all the nodes in the DKS ring. Here the request file is specified in ⑥.

④ Specify the address of the remote DKS server. This server stores all the living nodes' references.

⑤ Each node in the DKS ring should have a unique port number; otherwise the node cannot be registered into the DKS ring.

⑥ Path of the request owl document.

⑦ Open a file explorer to select the request file.

⑧ The address of the notification service. This service is used to notify the client the result after client sending the request. Each node in the DKS ring will send back the matching result by contacting this notification service. So the client can receive all returned messages from the DKS ring.

⑨ List of the provided service.

⑩ This window displays the state of our system.

6 Profiling of Prototype

In this chapter we will evaluate the performances of the application. We divide our test into different phase, registration, joining DKS, download, matchmaking, notification. We will do the performance evaluation through each phase, and in the end we will give a time consumption distribution of different phases.

In our evaluation, the Grid site runs a GT4 container as well as a notification service, matchmaking, DKS middleware and Default Index Service. The grid site is Intel Centrino Duo T2300@1.66GHz, 512 MB of RAM, Intel PRO/1000 PL Network Connection.

6.1 Time anatomy of registration

In our work, we write our own execution aggregator information provider. Each information provider has a registration file which records address of default service group EPR and default registrant EPR, refresh interval seconds for renew registration, poll interval millisecond for running our script. We set both of these intervals 10 seconds. After we run command *mds-servicegroup-add*, as figure 6-1, the registration complete immediately. Then the registration renew each 10 seconds.

```
yeou@yeou:~$ mds-servicegroup-add –s \
http://127.0.0.1:8080/wsrf/services/DefaultIndexService \
/home/yeou/workspace/YangYeou_SourceCode/registration-schemas/owls-provider- \
registration.xml
Processing configuration file...
Processed 1 registration entries
Successfully registered http://127.0.0.1:8080/wsrf/services/owlsProvider to
servicegroup at http://127.0.0.1:8080/wsrf/services/DefaultIndexService
```

Figure 6-1 Register with Index Service

6.2 Time anatomy of download

- **Download file from Default Index Service**

We divide our download into three phases, which are query, modify and generate ontology. The test case for downloading is that we have already registered some files to the Default Index Service. All the test files have been download form http://www.mindswap.org/2004/owl-s/1.1. We test time consuming in three phase for each file downloading from Default Index Service. The times obtained are presented in Table 6-1.

Table 6-1 Time consuming of downloading file from Default Index Service

Size(Kb)	File name	Query(ms)	Modify(ms)	toOntology(ms)	Total(ms)
4.2	getCurrentStorms.owl	118	14	2233	2363
4.7	getTemp.owl	113	15	2347	2475
5.5	GetWeather.owl	115	29	2366	2510
7.9	NDFDgenByDay.owl	127	19	2231	2377
8.5	NDFDgen.owl	127	25	2488	2640
8.7	FrenchDictionary.owl	130	27	9639	9736

As we can see that generate Ontology from a file takes the most of the time. It takes more than 93% of the time. And query the file by giving the name of the file and return the content of that file takes about 4.6% of the total time, except the time for querying FrenchDictionary.owl file, it takes 1.3% of the total time. Modify the downloaded file takes the least of the time. In general, there is an upward trend in the number of total downloading time.

- **Download service description and save it to the local disk**

We test two types of download, shown in Figure 6-2. The one is that by giving the URI of the description compute the time that downloads that file from the internet to the local disk. The other is by giving the name of the description compute the time that downloads that file from Default Index Service to the local disk.

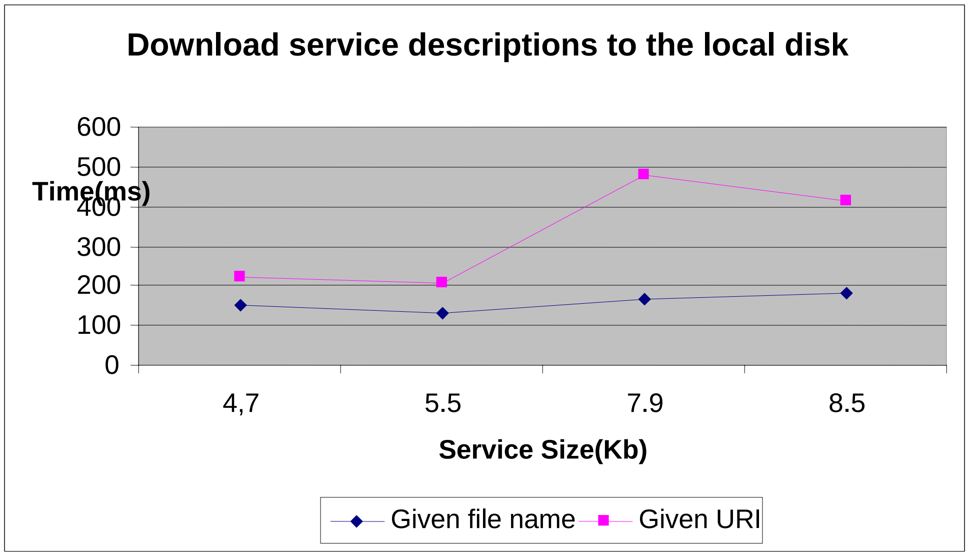


Figure 6-2 Downloading service description from Internet and Index Service time chart

As can be seen from the graph, the two curves show the fluctuation of different download methods. Given the URI of the service description, download this description to local disk needs more time compare with download the same description from Default Index Service.

- **Reading services at a given URI (i.e. Reader.read(uri)) compares with reading services at a given InputStream (i.e. Reader.read(InputStream, FileURI))**

The times obtained are represented in the chart in Figure 6-3 below

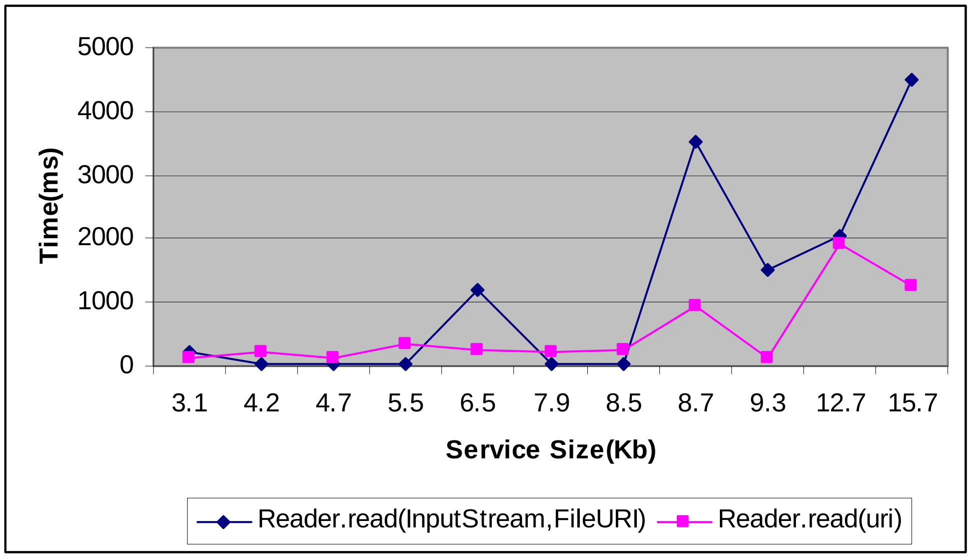


Figure 6-3 Services parsing time chart

As we can see, the time needed to parse services increases exponentially with the ontologies size. When the size is small, from 3.1kb to 8.5kb, the Reader.read(uri) needs slightly more time compares with transform a file to an Ontology. When the size increases, parsing the file needs much more time than parsing the URI to generate Ontology. For example, when the file size is 15.7kb, the blue point reach 4487ms, however parsing URI only needs 1260ms.

6.3 Query service description URI

We register some URI to the Default Index Service in advance, then we compute the time spending on the getting the list of the URIs. The time of getting the file list is the same as the time of getting the URIs list. We set the number of registered URIs from 0-2000; we obtained the times shown in Figure 6-4 below:

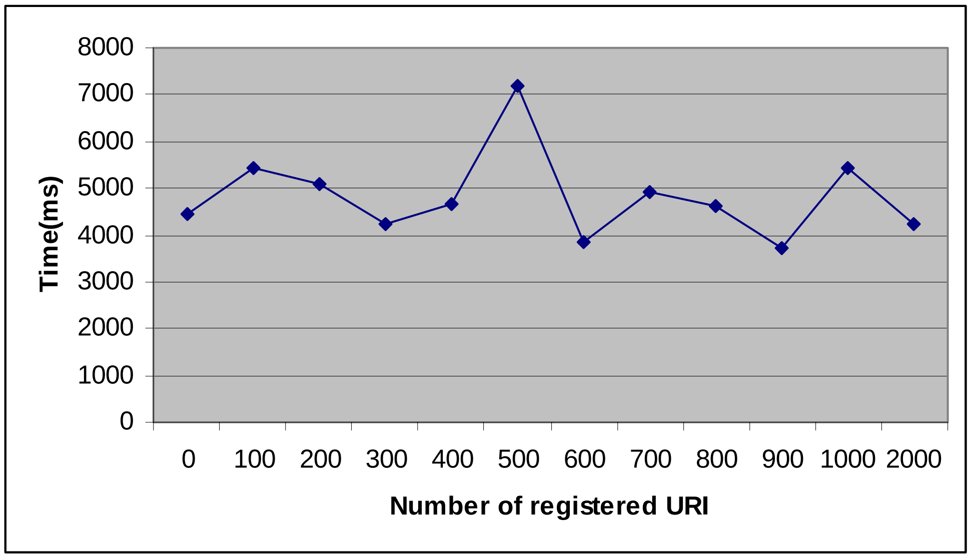


Figure 6-4 Query list of URIs time chart

As we can see from Figure5-7, even though the number of registered URI is 2000, the query time is still the same as registered URI is 0. The query time remains steady from the number of registration information is 0 to 2000.

6.4 Time consumption distribution of different phases

In this section we will see the time consumption distribution of different phases. In order to test this, we registered two services description into Default Index Service in advance, file *getTemp.owl* and URI *http://www.mindswap.org/2004/owl-s/1.1/FrenchDictionary.owl*. Here we only create one DKS node, and using broadcast in this DKS network. The Figure 6-5 shows the time consumption distribution of different phrases.

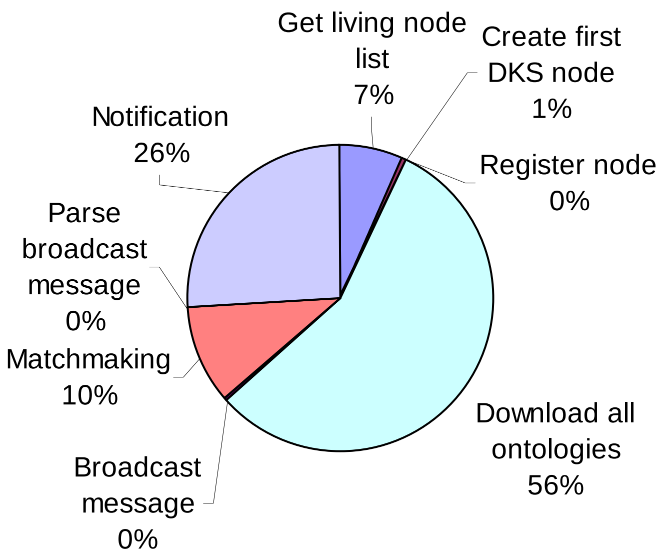


Figure 6-5 Time consumption distribution of different phrases

With no surprise, the Download phase takes the most of the time. Because download need to cooperate with other sites quite often to get information, which is time-consuming. The site might be Default Index Service or the URI on the Internet. If the number of Ontoloties increases, there is usually an increase in time consumption. Matchmaking phase also play important roles in time consumption. If the number of local registered service descriptions increase, matchmaking will take more time. So Download and Matchmaking phases consume more than 60% of the time. Get living node phase is related to query Default Index Service, so it also takes a lot of time. A notification phase is related to the operation on the web service resources, which may account for the large time consumption.

7 Conclusion and Future works

In this chapter a conclusion of the thesis work will be given first, which includes a contribution of our semantic based resource discovery service system. In addition, future work is pointed out and expected to be carried on later.

7.1 Summary

In this thesis we have studied the problem of resource discovery in Grids by means of P2P technology. We considered system where super nodes hold a set of resource. System user can use our grid system to register and deregister their resources. Users can locate resource by performing a desired web service query. Our system can help user to search the web services which match user's requirement and then notify that user. We have given the design and implementation of our system and performance evaluation.

We described architecture for resource discovery that adopts a Distributed K-ary System which is a P2P approach to extend the model of the GT4 information service (MDS4). In our work we defined resource as OWL-S1.1, which is a description of semantic based web service. OWL-S markup of Web services can facilitate the automation of Web service tasks, including automated Web service discovery, execution, composition and interoperation. We publish these OWL-Ss on the GT4 Default Index Service, in order to let other user find it. The matchmaker then use Profile matchmaking algorithm to decide the degree of matching of two OWL-Ss. After matching on each super node, the grid system uses WS-Notification mechanism to notify the request sender about the matching result.

7.2 Conclusions

As a result of the work carried out on this thesis project, the following conclusions could be highlighted:

- MDS4 Services

  Index Service, which is one of the WSRF-based services in MDS4, collects data from various sources and provides a query/subscription interface to that data. Each Globus container that has MDS4 has a Default Index Service by default. If we want to collect our own data by querying the Default Index Service, we need to register that service, which is used to collect information, into Default Index Service. By default, information providers in MDS4 can only provide information, such as host data, memory size, OS name and version, number of CPUs available and free, status data of the server. In our thesis work, we extend the usage of MDS4. We use execution aggregator source which executes an administrator-supplied program to collect information and make it available to the Index Service. In our system, we use execution aggregator source to register living DKS node reference, content of advertised service description and request service description, URI links of advertised service description to the Default Index Service.

- Download service description from Index Service

Our grid system provides a download function which can download the registered service descriptions to the local disk, and transform this file into Ontology type, in order to complete matching. We have developed an API of download function. We only need to specify the name or the URI of that service description and if it is the file name, we also need to specify the address of the Index Service which stores that file, and then the system can download that file or list of file from Internet or Index Service.

- Matchmaking

When a search request is received at a Super Node, it extracts all its advertised services from its local storage and matches them against the requested one, using a matching algorithm. We use profile matching algorithm described in [66] [67]. The best matching services (i.e. obtaining a score greater or equal than the score specified by the requester) are returned to the requester.

- Grid system and Distributed K-ary System

We implement resource discovery using techniques from P2P systems and achieve full distribution, high-performance, scalability, resilience to failures, robustness and adaptivity. Notice that MDS4 implementations are centralized or hierarchical and will never achieve the performance and scalability typically associated with P2P networks. In our thesis work we use DKS middleware to construct P2P overlay network, and then use DKS broadcast function [27] to broadcast request message.

- Notification

Globus Toolkit 4 provides us a very useful notification pattern. We use notification to notify the requester the matchmaking results. In our thesis work, we deployed a notification web service in GT4 container. This service is used to receive the subscription and its state can be changed, and then the notification is triggered to notify the subscription sender, which is the web service query requester. Each web service query requester has one notification service and one value listener which is used to receive the result notification. Each notification service can be invoked by many clients. These clients are actually super nodes. After complete matchmaking, each of these super nodes will contact the requester’s notification service to tell it matchmaking result.

- Security configuration

In any networked environment, security is a paramount concern. We must protect ourselves from outside threats. Because this requirement is especially true for a grid environment where clients can be geographically and organizationally diverse, GT4 meets this need. After installation of GT4, we take advantage of simpleCA [58] in order to obtain certificates and perform the certificate setup for Globus Grid Security Infrastructure (GSI). Only we have installed a trusted CA, we are allowed to use the

GT4 system. Another security configuration we use is that we used a security descriptor to configure our notification service's security and we use a secure client to invoke the notification. The client uses GSI Secure Conversation which is message level security with encryption and no client-side authorization.

- GUI

Our system is an integrator of GT4, DKS, OWL Reasoner Jena and Pellet, OWL-S1.1 and some other technologies. The system is running under Linux. Generally, most of the operations under Grid environment are using command line, like query Index Service, we use wsrf-query command. Our grid system performs background processing and hides all these complexities operation. User can use our GUI to set all configurations and connect to the network, register and deregister advertised service descriptions, update and search.

7.3 Future works

We can extend our matchmaking algorithm. There is another algorithm for matchmaking that is used to match service models. The authors of the method in [60] present their algorithm as an extension to profile matchmaking as they take into account the detailed process description of services, the service model.

We hope that the system presented, still very effective in favorable conditions, will be useful for resource discovery in Grid environment and it also extends the usage of MDS4, such as register the whole file to Index Service and query and download the file to the local disk.

8 List of Abbreviations

GT4	**Globus Toolkit 4**
DKS	**Distributed K-ary System**
MDS4	**Monitoring and Discovery System 4**
RDF	**The Resource Description Framework**
OWL-S	**Ontology Web Language**
P2P	**Peer-to-Peer**
SIS	**Super node’s Index Service**
LIS	**Local node’s Index Service**
NSS	**Notification service for super node**
DKSB	**DKS broadcast**
SVL	**Super node’s value listener**
LVL	**Local node’s value listener**
IQ	**Index service Query**
RR	**Request register**
FP	**File provider**
URIP	**URI provider**
DKSNR	**DKS Node Register**

9 References

[1]. IBM Grid computing http://www-03.ibm.com/grid/about_grid/what_is.shtml.
[2]. Ian Foster, Carl Kesselman. The Grid: Blueprint for a New Computing Infrastructure, Second Edition. Page 40
[3]. http://www.globus.org/alliance/
[4]. Globus Toolkit 4. URL: http://www.globus.org/toolkit/
[5]. http://www-128.ibm.com/developerworks/library/gr-overview/index.html
[6]. David Barkai, "Peer-to-Peer Computing Technologies for Sharing and Collaborating on the Net" Page 13, Intel Press; 1st edition (March 18, 2002), ISBN-10: 0970284675
[7]. VEYTSEL, A. 2001. There is no P-to-P Market... But There is a Market for P-to-P. Aberdeen Group Presentation at the P2PWG, May 2001.
[8]. SHIRKY, C. 2001.What is P2P... and what Isn't. An article published on O'Reilly Network. www.openp2p.com/lpt/a//p2p/2000/11/24/shirky1-whatisp2p.html.
[9]. Napster. Napster media sharing system. http://www.napster.com/.
[10]. Gnutella. http://rfc-gnutella.sourceforge.net
[11].Foster, I., and Iamnitchi, A., On death, taxes, and the convergence of peer-to-peer and Grid computing, in 2nd International Workshop on Peer-to-Peer Systems, Berkeley, CA. LNCS, Springer-Verlag, Heidelberg, 2003
[12]. http://www.ibm.com/developerworks/grid/library/gr-heritage/
[13]. Ian Foster, Carl Kesselman "The Grid: Blueprint for a New Computing Infrastructure, Second Edition" Page 622. ISBN 7-111-16054-1
[14]. KaZaA URL: http://www.kazaa.com/us/index.htm
[15]. Random work C. Gkantsidis, et.al,*Random Walks in Peer-to-Peer Networks*, INFOCOM 2004
[16]. Dynamic Query URL: http://www.the-gdf.org/index.php?title=Dynamic_Querying
[17]. 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' 01 Conference, San Diego, California, August 2001.
[18]. Ali Ghodsi, Luc Onana Alima, Seif Haridi. Low-Bandwidth Topology Maintenance for Robustness in Structured Overlay Networks, In the 38th International HICSS Conference, Springer-Verlag, January, 2005, Best Paper Award in the Software Track.
[19]. A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In International Conference on Distributed Systems Platforms (Middleware), Nov. 2001.
[20]. [20]. B.Y.Zhao, J.D. Kubiatowicz, and A.D. Joseph. Tapestry: An infrastructure for fault-resilient wide-area location and routing. Technical Report UCB//CSD-01-1141, University of California-Berkeley, April 2001.
[21]. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, Scott Shenker. A Scalable Content-Addressable Network. In Proceedings of the ACM SIGCOMM, 2001.
[22]. URL: http://en.wikipedia.org/wiki/Tulip_Overlay
[23]. URL: http://www.bittorrent.com/
[24]. URL: http://en.wikipedia.org/wiki/Coral_Content_Distribution_Network
[25]. L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications.

In *Proceedings of the 3rd International Workshop on Global and Peer-To-Peer Computing on Large Scale Distributed Systems (CCGRID'03)*, pages 344–350, Tokyo, Japan, May 2003. IEEE Computer Society.
[26]. http://en.wikipedia.org/wiki/Distributed_hash_table
[27]. Moni Naor and Udi Wieder. Novel Architectures for P2P Applications: the Continuous-Discrete Approach. Proc. SPAA, 2003.
[28].Gurmeet Singh Manku. Dipsea: A Modular Distributed Hash Table. Ph. D. Thesis (Stanford University), August 2004
[29]. David Karger, Eric Lehman, Tom Leighton, Matthew Levine, Daniel Lewin, and Rina Panigrahy. Consistent Hashing and Random Trees: Tools for Relieving Hot Spots on the World Wide Web. STOC 1997.
[30]. Ali Ghodsi, Luc Onana Alima, Seif Haridi. Symmetric Replication for Structured Peer-to-Peer Systems. In The 3rd International Workshop on Databases, Information Systems and Peer-to-Peer Computing, August 28-29, 2005, Trondheim, Norway
[31]. Ali Ghodsi. Distributed k-ary System: Algorithms for Distributed Hash Tables, PhD dissertation, KTH-Royal Institute of Technology, October 2006
[32]. Luc Onana Alima, Ali Ghodsi, Per Brand, Seif Haridi. Multicast in DKS(N, k, f) Overlay Networks, In Proceedings of the 7th International Conference on Principles of Distributed Systems, Springer-Verlag, Berlin, 2004
[33]. Ali Ghodsi, Luc Onana Alima, Sameh el-Ansary, Per Brand, Seif Haridi. Self-Correcting Broadcast in Distributed Hash Tables. In Series on Parallel and Distributed Computing and Systems , ACTA Press, Calgary, 2003
[34]. Sameh El-Ansary, Luc Onana Alima, Per Brand and Seif Haridi, Efficient Broadcast in Structured P2P Networks. In the 2nd International Workshop On Peer-To-Peer Systems, (Berkeley, CA, USA), February 2003
[35]. F. Kaashoek and D. R. Karger. Koorde: A simple degree-optimal distributed hash table. In Proceedings of the Second International Workshop on Peer-to-Peer Systems, IPTPS, 2003.
[36]. B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.
[37]. A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. Lecture Notes in Computer Science, 2218, 2001.
[38]. L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi. DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In The 3rd International workshop on Global and Peer-To-Peer Computing on large scale distributed systems - CCGRID2003, Tokyo, Japan, May 2003.
[39]. Global Grid Forum URL: http://www.ggf.org
[40]. http://www.ibm.com/developerworks/grid/library/gr-visual/
[41]. OASIS URL: http://www.oasis-open.org
[42]. Globus Index services http://www.globus.org/toolkit/docs/4.0/info/index/
[43]. http://gdp.globus.org/gt4-tutorial/multiplehtml/ch01s01.html
[44]. Berners-Lee,T., Hendler,J. and Lassila ,O.. "The Semantic Web",Scientific American, May 2001.
[45].OWL Services Coalition,OWL-S:Semantic Markup for Web Services, http://www.daml.org/services/owl-s/1.1/overview/

[46]. Deborah L. McGuinness, Frank van Harmelen. OWL Web Ontology Language Overview, http://www.w3.org/TR/owl-features/

[47]. http://www.w3.org/TR/owl-features/

[48]. M. Paolucci, T. Kawamura, T.R. Payne, and K.P. Sycara. Semantic matching of web services capabilities. In *ISWC '02: Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 333.347, London, UK, 2002. Springer-Verlag.

[49]. T. Payne, M. Paolucci, and K. Sycara. Advertising and matching DAML-S service descriptions. In Position papers of the first Semantic Web Working Symposium (SWWS'2001), pages 76.78, Stanford, USA, July 2001.

[50]. http://www.w3.org/Submission/OWL-S/

[51]. Wang Xin, Zhang Xiao lin, Realizing Semantic Web Services Description With OWL-S Ontology, Library of Chinese Academy of Sciences，Beijing1 00080，China

[52]. http://jena.sourceforge.net/

[53]. http://pellet.owldl.com/

[54]. http://www.mindswap.org/2003/pellet/faq#jena-1

[55]. MDS http://www.globus.org/toolkit/mds/

[56].http://globus.org/toolkit/docs/development/4.1.1/info/aggregator/developer/index.html#aggregator-developer-archdes

[57].mds-servicegroup-add URL:http://www.globus.org/toolkit/docs/4.0/info/aggregator/rn01re02.html

[58]. SimpleCA. http://www.globus.org/toolkit/docs/4.0/security/simpleca/

[59]. I. Fonster, C. Kessleman, G. Tsudik and S. Tuecke. A security architecture for computational grids. In ACM Conference on Computer and Communications Security Conference, 1998.

[60]. Sharad Bansal, Jose M. Vidal "Matchmaking of web services based on the DAML-S service model" AAMAS 2003: 926-927

[61]. GLUE resource property URL: http://www.globus.org/toolkit/docs/4.0/info/key/gluerp.html

[62].http://www.globus.org/toolkit/docs/4.0/info/index/WS_MDS_Index_HOWTO_Execution_Aggregator.html

[63]. Xpath 2.0. URL: http://www.w3.org/TR/xpath20/

[64]. Douglas Thain, Todd Tannenbaum, Miron Livny, Grid Computing: Making the Global Infrastructure a Reality Chapter 11, ISBN: 9780470853191

[65]. Paolucci, Kawamura, Paine, Sycara,"Semantic matching of Web-Services capabilities", Int. Semantic Web Conference 2001.

[66]. Jaeger,Rojec-Goldmann, Liebetruth, Kurt Geihs "Ranked Matching for Service Descriptions using OWL-S" KiVS 2005:91-102

[67]. Stefan Tang,"Matching of web service specifications using daml-s descriptions" Thesis, 19 Mar-2004

[68].http://www.daml.org/services/daml-s/0.9/

[69]. Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter Patel-Schneider. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, January 2003.

[70].http://www.daml.org/services/owl-s/1.1/ProfileHierarchy.html. Profile-based Class Hierarchies

Appendix A. Registration file

An example of registration files for register URI of the service description.

owls-provider-registration.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<ServiceGroupRegistrations
xmlns="http://mds.globus.org/servicegroup/client"
xmlns:sgc="http://mds.globus.org/servicegroup/client"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
xmlns:agg="http://mds.globus.org/aggregator/types">

  <defaultServiceGroupEPR>
    <wsa:Address>
http://127.0.0.1:8080/wsrf/services/DefaultIndexService</wsa:Address>
  </defaultServiceGroupEPR>

  <!-- The defaultRegistrantEPR defines the grid resource that will be
       registered, unless overridden in the
ServiceGroupRegistrationParameters
  -->
  <defaultRegistrantEPR>

<wsa:Address>http://127.0.0.1:8080/wsrf/services/owlsProvider</wsa:Addr
ess>
  </defaultRegistrantEPR>
<defaultSecurityDecriptorFile>some/other/sec/file.xml</defaultSecurityD
ecriptorFile>
  <ServiceGroupRegistrationParameters
xmlns="http://mds.globus.org/servicegroup/client">

    <!-- Renew this registration every 15 seconds (15) -->
    <RefreshIntervalSecs>15</RefreshIntervalSecs>
    <Content xsi:type="agg:AggregatorContent"
xmlns:agg="http://mds.globus.org/aggregator/types">
      <agg:AggregatorConfig xsi:type="agg:AggregatorConfig">
        <agg:ExecutionPollType>
          <!-- Run our script every 15000 milliseconds (15 seconds) -->
          <agg:PollIntervalMillis>15000</agg:PollIntervalMillis>
          <!-- Specify our mapped ProbeName registered in the
               $GLOBUS_LOCATION/etc/globus_wsrf_mds_index/jndi-
config.xml
               file -->
          <agg:ProbeName>owlsProvider</agg:ProbeName>
        </agg:ExecutionPollType>
      </agg:AggregatorConfig>
      <agg:AggregatorData/>
    </Content>
  </ServiceGroupRegistrationParameters>

</ServiceGroupRegistrations>
```

Appendix B. Code of getAliveNodeList (String remoteIndexAddress)

```
public List<String> getAliveNodeList(String remoteIndexAddress) throws Exception{
   EndpointReferenceType endpointRT = new EndpointReferenceType();
   endpointRT.setAddress(new AttributedURI(remoteIndexAddress));
   WSResourcePropertiesServiceAddressingLocator locator =
      new WSResourcePropertiesServiceAddressingLocator();
   QueryResourceProperties_PortType port;
   try {
      port = locator.getQueryResourcePropertiesPort(endpointRT);
      ((Stub)port)._setProperty(Constants.GSI_ANONYMOUS, Boolean.TRUE);
      String queryString ="//*[local-name()='JOIN_DKS_POSITION']";
      QueryExpressionType query = new QueryExpressionType();
      try {
         query.setDialect(WSRFConstants.XPATH_1_DIALECT);
      } catch (MalformedURIException e) {// Do something.}
    try {
       query.setValue(queryString);
       QueryResourceProperties_Element request = new QueryResourceProperties_Element();
       request.setQueryExpression(query);
       QueryResourcePropertiesResponse response;
       response = port.queryResourceProperties(request);
       MessageElement[] entries = response.get_any();
       if(response == null ||response.get_any() == null||response.get_any().length ==0) {
          System.out.println("Query did not return any results.");
          allREFs.clear();
          allREFs.add("none");
          for(int i=0;i<allREFs.size();i++)
               System.out.println("all ref ===="+allREFs.get(i));
       } else {
          for (int i = 0; entries != null && i < entries.length; i++) {
               String resultStr=AnyHelper.toSingleString(response);
               int startpoint = resultStr.indexOf(">") + 1;
               int endpoint = resultStr.indexOf("<", startpoint);
               String ref = resultStr.substring(startpoint, endpoint);
               if(ref.length()==2){
                  allREFs.add("none");
                 }
              // Parse out the uris
               else{
                   StringTokenizer st = new StringTokenizer(ref);
                   while (st.hasMoreTokens()) {
                       String ref_temp=st.nextToken();
                       if(!ref_temp.equals(""))
                           allREFs.add(ref_temp);
                  }
                }
            }//for
         }//else
    } catch (RemoteException e) {// Do something.}
} catch (ServiceException e) {// Do something.}
               return allREFs;
}
```

Appendix C. JoinDKS() source code in MyDKS.java

```
public void joinDKS(){
…
if(aliveList.get(i).equals("none")){
   ConnectionManager cm ConnectionManager.getInstance(this.dks_port);
   DKSImpl node;
   DKSOverlayAddress oa = new DKSOverlayAddress("dksoverlay://0/"+0+"/0");
   node = new DKSImpl(cm, oa);
   Register reg_client=new Register();
   reg_client.regDKS(node.getDKSRef().toString());
   node.create();
   mynode=node;
   setStatus("First node created with DKSURL " +node.getDKSURL());

   DefaultAppHandler ah = new DefaultAppHandler() {
        // broadcastCallback : This callback routine is called when //the
        node recieves a   broadcast message.
       public void broadcastCallback(DKSObject payload){
             Vector matchResult=new Vector();
            String queryMsg="";
             String senderIP=""; //sender’s notification service address
             DKSNode n = (DKSNode) dks;

            if (payload.getData().length!=0){
                    byte[] recMsg=payload.getData();
                    String recStr=new String(recMsg);
                    setStatus("receive queryMsg = "+recStr);
                  //Parse the request message
                  //Download the request file, local registered files
                    //…
                  //Matchmaking
                    //…

                  // Contact requester
                    ClientAdd client_noti=new
                         ClientAdd(senderIP,matchResult.toString());
                    client_noti.contactNotificationService();
           }else{
                    setStatus("WRONG DATA "+n.getDKSRef());
             }
      } //broadcastCallback
      };
             DKSCallbackInterface c = node.setCallbackHandler( ah );
             ah.setDKSCallbackInterface(c);
             node.logLevel(2);
}
//…
// if the node is not the first node in the DKS, join the DKS and //start
listening
}
```

Appendix D. WSDL file

Notification.wsdl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="NotificationService"

targetNamespace="http://www.globus.org/namespaces/thesis/Notification"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://www.globus.org/namespaces/thesis/Notification"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    xmlns:wsntw="http://docs.oasis-open.org/wsn/2004/06/wsn-WS-
BaseNotification-1.2-draft-01.wsdl"
    xmlns:wsrlw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceLifetime-1.2-draft-01.wsdl"

xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor
"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-
ResourceProperties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-
1.2-draft-01.wsdl"
    location="../../wsrf/notification/WS-BaseN.wsdl"/>

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-
1.2-draft-01.wsdl"
    location="../../wsrf/lifetime/WS-ResourceLifetime.wsdl" />

<types>
<xsd:schema
targetNamespace="http://www.globus.org/namespaces/thesis/Notification"
    xmlns:tns="http://www.globus.org/namespaces/thesis/Notification"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">


      <!-- Requests and responses -->

        <xsd:element name="addResult" type="xsd:string"/>
      <xsd:element name="addResultResponse">
            <xsd:complexType/>
      </xsd:element>
      <!-- Resource properties -->
      <xsd:element name="Result" type="xsd:string"/>

      <xsd:element name="ResDisResourceProperties">
      <xsd:complexType>
            <xsd:sequence>
                  <xsd:element ref="tns:Result" minOccurs="1"
maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>
      </xsd:element>

      <!-- Custom Notification Messages -->
      <xsd:element name="BigValueChangeNotificationMessage"
type="tns:BigValueChangeNotificationMessageType"/>

      <xsd:element name="Result" type="xsd:string"/>

      <xsd:complexType name="BigValueChangeNotificationMessageType">
            <xsd:sequence>
                  <xsd:element ref="tns:Result" minOccurs="1"
maxOccurs="1"/>
            </xsd:sequence>
      </xsd:complexType>

      <xsd:complexType
name="BigValueChangeNotificationMessageWrapperType">
            <xsd:sequence>
                  <xsd:element
ref="tns:BigValueChangeNotificationMessage"/>
            </xsd:sequence>
      </xsd:complexType>


</xsd:schema>
</types>

<message name="AddResultInputMessage">
      <part name="parameters" element="tns:addResult"/>
</message>
<message name="AddResultOutputMessage">
      <part name="parameters" element="tns:addResultResponse"/>
</message>

<portType name="NotificationPortType"
    wsdlpp:extends="wsntw:NotificationProducer
wsntw:SubscriptionManager"
    wsrp:ResourceProperties="tns:ResDisResourceProperties">


       <operation name="addResult">
            <input message="tns:AddResultInputMessage"/>
            <output message="tns:AddResultOutputMessage"/>
      </operation>
</portType>
```

```
</definitions>
```

Appendix E. Mapping for information provider

$GLOBUS_LOCATION/etc/globus_wsrf_mds_index/jndi-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">
    <service name="IndexService">

        <resource
            name="home"
            type="org.globus.mds.index.impl.IndexHome">
            <resourceParams>
                <parameter>
                    <name>
                        factory
                    </name>
                    <value>
                        org.globus.wsrf.jndi.BeanFactory
                    </value>
                </parameter>
                <parameter>
                    <name>resourceClass</name>

<value>org.globus.mds.index.impl.IndexResource</value>
                </parameter>
                <parameter>
                    <name>resourceKeyName</name>

<value>{http://mds.globus.org/inmemoryservicegroup}ServiceGroupKey</val
ue>
                </parameter>

            </resourceParams>
        </resource>

        <resource name="configuration"

type="org.globus.mds.aggregator.impl.AggregatorConfiguration">
          <resourceParams>
             <parameter>
                    <name>
                        factory
                    </name>
                    <value>
                        org.globus.wsrf.jndi.BeanFactory
                    </value>
</parameter>
             <parameter>
                <name>aggregatorSources</name>

<value>org.globus.mds.aggregator.impl.QueryAggregatorSource
```

```
org.globus.mds.aggregator.impl.SubscriptionAggregatorSource
org.globus.mds.aggregator.impl.ExecutionAggregatorSource</value>
             </parameter>
             <!--<parameter>
                <name>executableMappings</name>
                <value>aggr-test=aggregator-exec-test.sh,
pingexec=example-ping-exec,owlsProvider=OWL-Provider.sh</value>
             </parameter>-->
             <parameter>
                <name>executableMappings</name>
                <value>aggr-test=aggregator-exec-test.sh,
pingexec=example-ping-exec,owlsProvider=OWL-
Provider.sh,owlFileProvider=OWLFileProvider.sh,joinDKSPosition=JoinDKSP
osition.sh,requestDeploy=RequestDeploy.sh</value>
             </parameter>
          </resourceParams>
       </resource>

       <resourceLink name="groupHome"
target="java:comp/env/services/IndexService/home"/>
       <resourceLink name="entryHome"
target="java:comp/env/services/IndexServiceEntry/home"/>
   </service>

    <service name="IndexServiceEntry">
       <resource name="home"
            type="org.globus.mds.index.impl.IndexEntryHome">
            <resourceParams>
                <parameter>
                    <name>
                        factory
                    </name>
                    <value>
                        org.globus.wsrf.jndi.BeanFactory
                    </value>
                </parameter>
            </resourceParams>
</resource>

       <resourceLink name="groupHome"
target="java:comp/env/services/IndexService/home"/>
       <resourceLink name="entryHome"
target="java:comp/env/services/IndexServiceEntry/home"/>
       <resourceLink name="configuration"
target="java:comp/env/services/IndexService/configuration"/>
    </service>

    <service name="IndexFactoryService">
       <resourceLink name="entryHome"
target="java:comp/env/services/IndexServiceEntry/home"/>
       <resourceLink name="groupHome"
target="java:comp/env/services/IndexService/home"/>
       <resourceLink name="configuration"
target="java:comp/env/services/IndexService/configuration"/>

        <resource
            name="home"
            type="org.globus.wsrf.impl.ServiceResourceHome">
            <resourceParams>
                <parameter>
                    <name>
                        factory
                    </name>
                    <value>
                        org.globus.wsrf.jndi.BeanFactory
                    </value>
                </parameter>
            </resourceParams>
        </resource>
    </service>

   <service name="DefaultIndexService">
  <resource
            name="home"

type="org.globus.mds.aggregator.impl.SingletonAggregatorHome">
            <resourceParams>
                <parameter>
                    <name>
                        factory
                    </name>
 <value>
                        org.globus.wsrf.jndi.BeanFactory
                    </value>
                </parameter>
                <parameter>
                    <name>resourceClass</name>

<value>org.globus.mds.index.impl.IndexResource</value>
                </parameter>
            </resourceParams>
        </resource>

       <resourceLink name="groupHome"
target="java:comp/env/services/DefaultIndexService/home"/>
       <resourceLink name="entryHome"
target="java:comp/env/services/DefaultIndexServiceEntry/home"/>
       <resourceLink name="configuration"
target="java:comp/env/services/IndexService/configuration"/>
   </service>

    <service name="DefaultIndexServiceEntry">

         <resource
            name="home"
            type="org.globus.mds.index.impl.IndexEntryHome">
            <resourceParams>
                <parameter>
                    <name>
                        factory
                    </name>
                    <value>
                        org.globus.wsrf.jndi.BeanFactory
                    </value>
                </parameter>
            </resourceParams>
        </resource>

       <resourceLink name="groupHome"
target="java:comp/env/services/DefaultIndexService/home"/>
       <resourceLink name="entryHome"
target="java:comp/env/services/DefaultIndexServiceEntry/home"/>
       <resourceLink name="configuration"
target="java:comp/env/services/IndexService/configuration"/>
    </service>
</jndiConfig>
```

Appendix F. User Guides

There are four steps of running our system. The precondition of running our program is having Globus Toolkit 4 environment, Java environment.

1. Re-build the notification service.
   We have to re-build the notification service instead of using the Gar file in the source code directory. Our notification service java files are in /org/globus/thesis/services/core/notifications/impl directory.
   As a normal user, the corresponding command is:
   yeou@yeou:~$ ./globus-build-service.sh notifications
   After that the program will generate a org_globus_thesis_services_core_notifications.gar file in the current directory
2. Deploy the service
   This deployment command must be run with a user that has write permissions in $GLOBUS_LOCATION. For example, we use user "globus" as our GT4 runner. So open another terminal, change the user to globus, and then run:
   globus-deploy-gar \
   $GAR_Directory/org_globus_thesis_services_core_notifications.gar
3. Run globus container as a *globus* user:
   `globus-start-container -nosec`
4. Run shell scrip to create document for registration
   Open another terminal, go to the shell script directory and run command:
   . ~/create.sh
1. Move the shells in the Shells_for_registration directory to the $GLOBUS_LOCATION/libexec/aggrexec/. Make sure your file resides in this directory with proper executable permissions.
2. Establish mapping of our information provider.
   To establish the mapping of your provider, you need to edit the $GLOBUS_LOCATION/etc/globus_wsrf_mds_index/jndi-config.xml. There is an example of jndi-config.xm in the Appendix E.
   The only place we need to modified is in executableMappings section:

```
<parameter>
        <name>executableMappings</name>
        <value>aggr-test=aggregator-exec-test.sh, pingexec=example-
        ping-exec,owlsProvider=OWL-
        Provider.sh,owlFileProvider=OWLFileProvider.sh,joinDKSPositio
        n=JoinDKSPosition.sh,requestDeploy=RequestDeploy.sh</value>
</parameter>
```

3. Run mds-servicegroup-add
   Each of the following should be run under different terminal.

```
 yeou@yeou:~$ mds-servicegroup-add –s
http://127.0.0.1:8080/wsrf/services/DefaultIndexService
/home/yeou/workspace/YangYeou_SourceCode/registration-schemas/owls-
provider-registration.xml
```

```
yeou@yeou:~$  mds-servicegroup-add -s
http://127.0.0.1:8080/wsrf/services/DefaultIndexService
/home/yeou/workspace/YangYeou_SourceCode/registration-
schemas/owlFile-provider-registration.xml

yeou@yeou:~$  mds-servicegroup-add -s
http://127.0.0.1:8080/wsrf/services/DefaultIndexService
/home/yeou/workspace/YangYeou_SourceCode/registration-
schemas/joinDKSPosition-registration.xml

yeou@yeou:~$  mds-servicegroup-add -s
http://127.0.0.1:8080/wsrf/services/DefaultIndexService
/home/yeou/workspace/YangYeou_SourceCode/registration-
schemas/requestDeploy-registration.xml
```

4. Start client value listener
   1) `Start a new terminal and run:`
      `javac -classpath $CLASSPATH:build/stubs/classes/ org/globus/thesis/clients/Notification/ValueListener.java`
   2) `source $GLOBUS_LOCATION/etc/globus-user-env.sh`
   3) `grid-proxy-init`
      `Here the system will ask you to enter your passphrase. This is the passphrase which you specified when you created your CA certificate in set up certificates progress for GT4.`
   4) `java -DGLOBUS_LOCATION=$GLOBUS_LOCATION -classpath $CLASSPATH:build/stubs/classes/ org/globus/thesis/clients/Notification/ValueListener http://127.0.0.1:8080/wsrf/services/thesis/core/notifications/NotificationService`

9. Start GUI.
   Start a new terminal and run the shell script in the shell directory. You need to modify the location of the jar files in this set.sh file. Change the path to the `pellet-1.4-RC1/lib/`, `jdom-1.0/lib`, `owl-s-1.1.0-beta/lib/` and `dks.jar` `to your own library. The command is:`

   ```
   yeou@yeou:~$ . ~/set.sh
   ```
10. Set the port number and then press connect button.
11. Choose the local OWL-S file or type in the URI of the OWL-S which you want to register. You can add more than one file or URI, just press add button after each.
12. Press register button.
13. If this client wants to start query, he can specify the request link such as URI or choose the prepared request file from local disk.
14. Press query button. Wait for a couple of seconds, the client can see the state in the status window, and receive the result from the value listener.