ENHANCEMENT OF THE
UNIFRAME RESOURCE DISCOVERY SERVICE

A Thesis

Submitted to the Faculty

of

Purdue University

by

Barun Dev Devaraju

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2005

To Mom, Dad, and Arun

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my professor and advisor, Dr. Rajeev R. Raje, for giving me an opportunity to be part of the UniFrame project. His support and advice throughout the course of my graduate study and research work, along with his invaluable guidance, have helped me to successfully complete this thesis. I am grateful to him for his constant encouragement to accomplish this research work to the best of my abilities. I am positive that the motivation and knowledge gained from this experience will have a tremendous impact on my career.

I would like to thank Dr. Andrew M. Olson for his valuable suggestions throughout the research period, and for reviewing my thesis. I would also like to thank Dr. Xukai Zou for being part of my graduate committee, and for his guidance in a few of my course works.

I sincerely acknowledge the U.S. Department of Defense and the U.S. Office of Naval Research for supporting this research project with their grant under the award number N00014-01-1-0746. I would like to thank my colleagues in the UniFrame research project for their assistance thru interesting and helpful discussions. I would also like to thank the staff of the Department of Computer and Information Science for their assistance during my entire graduate study.

Finally, I extend my thanks to my family and friends for their encouragement and support.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Devaraju, Barun Dev. M.S., Purdue University, May 2005. Enhancement of the UniFrame Resource Discovery Service. Major Professor: Dr. Rajeev R. Raje.

Unlike the traditional way of software development, the software realization of large scale Distributed Computing Systems (DCS) could be achieved through the Component Based Software Development (CBSD) approach. To develop DCS using the CBSD approach, appropriate software components are to be discovered from the network and these components need to be integrated. UniFrame is a research project that provides a unified framework to create spontaneous and high-confidence distributed computing systems using distributed heterogeneous components. UniFrame uses the UniFrame Resource Discovery Service (URDS) for the discovery of components that are deployed on the network. As the discovered components are utilized to form DCS, the component discovery service should be comprehensive and sophisticated to discover the appropriate components. The research of this thesis enhances the existing URDS architecture, in terms of the performance of the component discovery process and the quality of the components discovered. It also customizes the component discovery process by the incorporation of profiling at different levels of the URDS architecture. A prototype reflecting various enhancements is implemented and experimented with. The experimental analysis using the prototype indicates the benefits of the enhancements carried out in this thesis.

CHAPTER 1. INTRODUCTION

Distributed Computing Systems (DCS) have been designed to use a large number of computers dispersed over a large geographical area. Although, DCS have improved in the recent years due to the advancement in the computing architectures and better networking technologies, creating efficient and robust software for such systems is a challenging task. A few of the reasons for this are attributed to the differences in processing speeds between the distributed nodes, heterogeneity in both the hardware and software, achieving synchronization between the distributed entities, and a lack of adherence to proven software development standards.

Recently, there has been a shift in the distributed computing paradigm from the traditional ad-hoc methods of building software systems to Component-based Software Development (CBSD). CBSD is defined as the process of building large and complex software systems by integrating previously developed and deployed software components. CBSD shifts the development emphasis from programming software to composing software systems. At the foundation of this approach is the assumption that certain parts of large software systems reappear with sufficient regularity that these common parts should be written once, rather than many times, and that systems should be assembled through reuse rather than rewriting them. The CBSD approach can potentially be used to increase the automation and productivity in software development, enable quality and reliability improvements in software, reduce software development cost, reduce the time to develop and test software, reduce the spiraling maintenance burden associated with the support and upgrade of large software systems, and cost amortization through a software component reuse. The challenges in CBSD are discussed in [CRN01a, CRN01b].

The CBSD approach of software development could be applied to build complex DCS i.e., DCS could be assembled using previously developed and deployed software components that are distributed across the network. These software components, although scattered across the network, work collectively to achieve a common task. These software components are independently developed and deployed by possibly different developers without a prior knowledge about the system that they may be a part of and other software components with which they may interact. The software components that are integrated to form DCS could belong to different distributed computing models such as Java Remote Method Invocation (RMI), Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA), and .NET. Apart from the heterogeneity in the computing model, there could be other forms of heterogeneity between the software components. These types of heterogeneity should be taken care of, so that components could be integrated together to form a software system.

An important step in building DCS using CBSD approach is the discovery of independently developed and deployed software components. This is then followed by the integration of those components. The component discovery process should be sophisticated to support the discovery of appropriate software components that precisely match the requirements of the desired system. Another challenging issue lies in tackling the quality aspect of a software component. In order for the CBSD approach to result in software systems with a predictable quality, the individual software components should offer a guaranteed level of quality of the functionality that they support. So, each component must define the quality of the services (QoS) offered by the components, such as the turn-around time, throughput, and precision. The process of integrating the software components must be done through a well-defined approach that should provide the binding that allows the generation of a software system from the disparate components. So, there is a necessity for a framework that can aid in the component discovery process and the component integration process. This framework should also incorporate QoS as an inherent part of software components and should offer objective means to quantify, validate and specify the QoS of software components.

UniFrame [RAJ01] is one such approach that provides a comprehensive framework to construct high-confidence DCS using distributed heterogeneous components that belong to existing and emerging distributed computing models. The resource discovery aspect of UniFrame, called the UniFrame Resource Discovery Service (URDS) [NAN02], aids in discovering components available over a network. The URDS architecture provides services for an automated discovery and selection of components that meet the necessary criteria specified in the requirements of the desired software system. This thesis contributes to the UniFrame research project by providing mechanisms to enhance the URDS architecture.

1.1. Problem Definition and Motivation

The component discovery process is more complicated than document discovery, which follows pattern matching. Each software component is associated with a specification which formally describes the component's properties, such as what services the component provides, how the component should be used and how it interacts with other components when brought together to form a distributed system. So, discovering a software component consists of component specification matching, which means matching of the component's specification against the requirements of the desired DCS. The component specification matching involves matching of the functional and non-functional attributes, i.e., QoS, of a component. The functional matching of a component is divided into different levels of matching such as syntax/signature level, semantic level, and synchronization level. The non-functional part of matching includes matching of different QoS parameters, such as the turn around time, the throughput, and the cost of the software components.

After the component specification matching process, the discovered components also have to undergo an integration process whose difficulty is based on the interoperability and heterogeneity among the discovered components. The more comprehensive the component matching criteria are, the more closely the discovered components match the requirements specification. Because, the discovered components are used to build DCS, the component discovery should be comprehensive. So, there is a need for a sophisticated discovery service architecture, which can discover better components. Here ‘better’ is defined as, how closely the functional and non-functional attributes of the component will match the component requirement specifications, and how easily those components can be integrated to form DCS.

The existing URDS architecture [NAN02] provides an infrastructure wherein new services are dynamically discovered meeting the type and the necessary QoS requirements specified by a component developer. Improvements could be made on this existing URDS architecture in terms of the performance of the component discovery process, and the quality of the discovered components. This thesis focuses on different ways on enhancing the URDS architecture.

1.2. Objectives

The overall objectives of this thesis are:

- To improve the performance of the existing URDS architecture and to provide support for an efficient component discovery process.
- To customize the component discovery process based on the preferences of the system developers.
- To improve the query propagation techniques involved in the component discovery process and, thereby, improve the quality of the discovered components.
- To implement the URDS architecture onto handheld devices and study the issues and challenges related to that adaptation.

The entities that form the URDS architecture are single threaded. So, each entity can only process the queries sequentially, i.e., there is no overlapping between the processing of different queries in an entity. Also, the query propagation between different entities follows a synchronous communication pattern, i.e., the entity that propagated a query is idle just waiting for the results, thus wasting a lot of processing power. These possibly could be improved by making the entities that form the URDS be multi-threaded. This might improve the performance of the URDS in terms of the overall time taken for the component discovery process.

The URDS architecture incorporates query propagation techniques, which are used to pass the queries among different nodes that form the URDS. These queries will carry the component requirement specifications, which are compared against the specification of the deployed software components, during the component discovery process. The query propagation technique incorporated in the existing URDS architecture is exhaustive i.e., all the entities that form the URDS process the query. Also, the selection of entities for the propagation of queries is random. These could be enhanced by incorporating efficient query propagation techniques, wherein the amount of processing done for discovering components can be reduced by reducing the number of nodes that process a query. This performance improvement should not compromise the quality of the discovered components, i.e., quality of the components discovered by this restricted processing, should be comparable to the quality of the components discovered in the case of an exhaustive search. So, the subset of nodes for propagating a query should be chosen such that the probability of finding the best matching components is more than a random case. This improvement could depend on the prediction of the nodes that have matching components based on the past performance. The reinforcement learning techniques discussed in [KAE96, MUK02] form as a basis for this improvement. This improvement in the query propagation techniques is helpful when there is a time restriction for the component discovery process and the quality of the components to be discovered cannot be compromised.

Other ways of enhancing the URDS are by improving the quality of the software components that match the given query. To improve the quality of the discovered components, the quality of components that are available for discovery should be good, i.e., the components available for discovery in the component pool need to be periodically refreshed, and the components that do not perform as defined in the component requirement specification need to be eliminated. Also, the quality of the matching components could be measured, which will aid the system developers (who use the component discovery service for developing DCS) in selecting the components for building DCS.

The URDS comprises several entities distributed across the network. These entities could be hosted in varies types of machines, having different capabilities and processing powers. The entities of the existing URDS architecture will be implemented into handheld devices having restrictions in the processing speed and memory, and the challenges related to that will be studied.

1.3. Contribution

The contributions of this thesis are:

- Improving the performance of the URDS architecture.
- Incorporation of reinforcement learning algorithms and techniques in the URDS architecture and the analysis of the incorporation of heuristic techniques in a component discovery service.
- Addition of various levels of profiling in the URDS architecture and customization of the component discovery process based on the profile information.
- Inclusion of the handheld devices for hosting a few of the entities that form the URDS architecture.

1.4. Thesis Organization

This thesis is organized into six chapters. An introduction, along with the problem definition and motivation, objectives, and the contributions are provided in this chapter. Chapter 2 discusses the background and related work of this thesis. Chapter 3 describes the existing architecture of the URDS, its features, and the possible enhancements that can be made. Chapter 4 gives the design and implementation details of the different entities that form the URDS architecture after incorporating a few of the enhancements. The experimental analysis of different enhancements and the comparison of those using graphs are discussed in Chapter 5. Chapter 6 provides the conclusion of this research work, the possible future enhancements, and the summary of this thesis.

CHAPTER 2. BACKGROUND AND RELATED WORK

This chapter provides an introduction to prominent research efforts related to the objective of this thesis i.e., the enhancement of the UniFrame Resource Discovery Service (URDS) [NAN02]. The first section of this chapter explains about an ideal component discovery service and the prominent features of those. The following sections discuss about the background work, related to achieve this ideal discovery service. A few of the general discovery services are discussed in section 2.2. Section 2.3 discusses about the component discovery services, which also includes the existing URDS. The foundation of this thesis work is based on the URDS architecture. Section 2.4 discusses about the Internet search engines, followed by profiling activities in resource discovery services in section 2.5.

2.1. Ideal Component Discovery Service

A component discovery service is viewed as any other discovery service, with the resources discovered being software components. The component discovery needs a comprehensive framework to specify the specifications of the components, and rules to match the query with the components. More care should be taken in the matching process, as the discovered software components will be used in developing distributed computing systems (DCS). This section discusses the features an ideal component discovery service architecture should posses.

▪ Heterogeneous – The component discovery service should support discovery of heterogeneous components. The component developed and deployed in the network could be Heterogeneous i.e., they belong to different distributed computing models such as: Java RMI, .NET, and CORBA. Restricting the component model to one of the distributed computing models will curtail the component list or impose restrictions on the component developers to stick to a particular model. So, the component discovery service should be generalized to support all models.

- Activeness – The discovery service should possess '*Activeness*', which means that the client looking for a resource will initiate the request, for which the resource provider responds. The resource providers should be actively listening for the clients' request.
- Scalability – Scalability is the ability to continue to function well when it is changed in size or volume. With more and more components being developed and deployed in the network, the component discovery service architecture should scale, so that the component discovery is done efficiently. The distributed discovery service architecture due to its hierarchical nature, aids in improving the scalability.
- Multi-level matching – There should be a support for matching components at multiple levels such as syntactic, semantic, protocol, and QoS. This multiple level matching helps to improve the matching of the query i.e., it improves the closeness in which the software component specifications match with the given requirement specification, thus facilitating to build a better distributed computing system.
- Reliability – Reliability is defined as the ability of a system or component to perform its required functions under stated conditions for a specified period of time. The component discovery service should be reliable so that it consistently discovers the best components that match the given query. The discovery service should neither fail nor compromise the quality of the component matching process.
- Profiling – Profiling could be added to the component discovery service architecture to enhance the query propagation techniques, and to improve the quality of the components discovered, within the distributed component discovery architecture. This could be achieved by storing the history of the previous component discoveries.
- Security – The component discovery architecture should be secure. This is mainly because the entities that form the component discovery architecture are in an open environment (i.e., network subject to intrusion) and they frequently communicate as part of the component discovery process.

URDS, discussed in the later part of this chapter supports a few of these features. This thesis work is based on the enhancement of the URDS; in an attempt at create an ideal component discovery service.

2.2. Resource Discovery Service

A Resource Discovery Service gathers contents from diverse distributed sources using various approaches and provides a subset of these to incoming requests. The resource discovery protocols could be directory-based or discovery-based. The basic difference between them is that the discovery service is considered active, while the directory service is considered passive. In active discovery, the entity seeking for the resource initiates the search by sending a discovery request, for which the resource provider responds. In passive discovery, the entity seeking for the resource listens to the discovery providers i.e., the discovery providers initiate the communication by sending out the resource information to the resource seekers.

2.2.1. Directory Service

A directory service is basically a lookup service, where the services are registered in a registry. This service could be compared with the yellow pages. A few of the discovery protocols that come under the directory service are discussed below.

2.2.1.1. Universal Description, Discovery and Integration (UDDI)

Universal Description, Discovery and Integration (UDDI) [UDD00] is a specification for distributed, Web-based information registries of Web services. A Web service is a software component or an application that exposes its functionality programmatically over the Internet or Intranets. UDDI creates a standard interoperable platform that enables applications to quickly and dynamically find and use Web services over the Internet. The three main components of the UDDI architecture are: web service providers, service requesters and the service brokers. To have a communication between the services in a language independent manner, the UDDI architecture uses the Simple Object Access Protocol (SOAP) built on top of XML. The service requester uses Web Service Definition Language (WSDL) to search for services in a common UDDI service registry. The matching in UDDI is carried out by matching the keyValue, keyName and tModelKey. The keyValue contains the searchable property, the keyName is just for human use, and the tModelKey is used to know the categorization scheme.

2.2.1.2. CORBA Trader Service

The CORBA trader service [OBJ00] is a lookup service for both the service providers and the service requesters. The three main components of the CORBA trader service are: the Exporters, Importers and Traders. Exporters are the service provider objects, which register or publish their services with the Traders. The Importers, which are consumers, gets the list of services from the local Traders. The local Traders, apart from searching in its local directory, also propagate the query to other Traders. The Traders are defined as CORBA Interfaces, defined by the Interface Definition Language (IDL). The matching process is carried out when the Importer is in need of a service. An Exporter will first register its service with the Trading Service, with relevant information such as an object reference, the service name, and the properties of the service. An Importer makes a request for a specific service type. Based on the properties of the services, the trading service performs a matching algorithm to return the result to the Importers.

2.2.1.3. Light Weight Directory Access Protocol (LDAP)

The Light Weight Directory Access Protocol (LDAP) [LDA01] is a light weight version of the Directory Access Protocol. It is also part of X.500, a standard for network directory services. LDAP aids in locating organizations, individuals and other resources such as devices and files across the network. The query model of LDAP allows for search and retrieval of entries stored in the LDAP directory service, which is organized as a hierarchical tree structure. The complexity of LDAP makes it difficult to specify spontaneous discovery protocols. A lack of a built-in security model and lack of replication standard (i.e., the process of copying information between geographically distributed servers) could be considered as other deficiencies for LDAP.

2.2.1.4. Global Name Service (GNS) and Domain Name Service (DNS)

The Global Name Service [LAM86] is a name service which acts as a basis for resource location, mail addressing and authentication in a distributed computing system. GNS addresses the problems of high availability, large size, continuing evolution, fault isolation and lack of global trust. A database composed of a tree of directories, holding names and values, is managed by this naming service.

A Domain Name System [MOU87] is an Internet-based service used to translate domain names in to IP addresses. A hierarchically partitioned static database of name-address mappings is used to achieve this. The types of queries supported by DNS include host name resolution and reverse resolution, mail host location, host information and well-known services information.

This section discussed a few directory services. Although the directory services are used to match the resource requestors and the resource providers, they are not active and not fit for the discovery of components where more complex matching features are required.

2.2.2. Discovery Service

A discovery service allows a process to spontaneously discover other resources and presenting itself to other resources. A discovery service adopts a more active nature, as opposed to the passive directory services. Here '*Activeness*' means that the resource seekers initiate the resource discovery while the resource providers periodically listen to the resource seekers. A few of the prominent discovery services are discussed below.

2.2.2.1. Jini

Jini [SUN01] is a Java-based discovery service, which uses Java RMI to achieve communication among service providers and service requestors. The main components involved in the Jini discovery service are the Service, Client and the Lookup service. Here a Service, which could by a software program, a hardware device or a combination of these, gets registered in the Lookup service. The Lookup service acts as a directory service of all the available Service and is well known to the Clients. The client looking for a Service, contacts the Lookup service to get the list of services. Although Jini is a prominent discovery service, one of the shortcomings of Jini is that it is homogeneous.

2.2.2.2. Service Location Protocol (SLP)

SLP [GUT99, PER99] is an Internet Engineering Task Force (IETF) standard framework used for dynamic resource discovery. The SLP architecture comprises User Agents (UA), Service Agents (SA) and the Discovery Agents (DA). SAs register the location and characteristics of the services in the DAs, which act as a directory by maintaining a list of all the services. UAs are responsible for discovering the resources on behalf of the clients that request for any service. SLP could be implemented in various configurations namely Active Discovery, Passive Discovery and Dynamic Host Configuration Protocol (DHCP).

2.2.2.3. Secure Service Discovery Service (SSDS): Ninja Project

The SSDS [CZE99] is a part of the Ninja research project [NIN02] at the University of California, Berkley. The SSDS is a scalable, fault-tolerant, reliable, and secure information repository, providing clients with directory style access to all available services by storing two types of information: descriptions of the services that are available for execution, and the services that are already running at a specific location. The SDS also supports both push-based (passive discovery) and pull-based access (query-based model).

The main components of SSDS architecture are Service Discovery Service (SDS) servers, services, capability managers, certificate authorities and clients. Services continuously listen for SDS server messages to determine the appropriate SDS server for its service descriptions. Clients listen to a well-known location to identify SDS servers related to their domains and submit a query in the form of an XML based service description. Although this supports complex queries to be matched against cached service descriptions, there is neither heterogeneity nor different levels of matching involved in this discovery service, which are necessary for a component discovery.

2.2.2.4. Salutation

Salutation [SAL99] is an open standard service discovery and session management protocol. The architecture provides a standard method for applications, services, and devices; and also enables them to search for other applications, services and devices for a particular capability. The main components of this architecture are the server, client and the Salutation Lookup Manager (SLM). The Salutation Manager manages all the communication, by serving as a broker for the services in the network. It also classifies the services into a collection of Functional Units (FU), where each functional unit represents one or more essential features. The query for a particular service from the client reaches the local SLM directory associated with it for a list of services. Based on the required service type, the query is propagated to other SLMs.

2.2.2.5. Universal Plug and Play (UPnP)

The Universal Plug and Play [REK99, MIC00] is a spontaneous service from the Microsoft Corporation, which uses Simple Service Discovery Protocol (SSDP) to discover services in the network. The architecture of UPnP is similar to that of Salutation and SLP. A service sends a multicast message to the lookup service, to advertise its services. XML is used to describe the service features. A client has an option to either contact the service directly from the URL specified in the lookup service or to multicast the query to the lookup service and wait for a reply from the lookup service or the actual service.

2.2.2.6. Bluetooth Service Discovery Protocol

The Bluetooth Service Discovery Protocol [GOL99, MIL99] is a short range wireless transmission protocol. There is a Bluetooth protocol stack, which has the Service Discovery Protocol (SDP) used to locate the services provided by the Bluetooth device. The SDP block generates and receives Bluetooth SDP commands and responses from the lower layers of the Bluetooth stack. All the information about a service is maintained by an SDP within a single service record. These service records consist of a list of service attributes, which describes a single characteristic of a service. SDP uses a request/response model where each transaction consists of one request protocol data unit (PDU) and one response PDU. The Bluetooth service discovery protocol allows the devices to exchange real time data and provides very efficient service discovery on resource-constrained devices.

2.2.2.7. Grid Discovery Service (Globus Toolkit MDS)

The Grid Discovery Service [OGS01] incorporates a Monitoring and Discovery Service (MDS), which consists of two components namely the Grid Information Resource Service (GRIS) and the Grid Index Information Service (GIIS). The GRIS is an information provider framework for specific information sources. The GIIS is a user accessible directory server that aggregates the information from child GIIS and GRIS instances. In the GRIS and GIIS hierarchies, all the instances immediately below its instance are considered as the child instance. This discovery service is mainly employed for computational resources. The performance of a query in this Grid Discovery Service cannot be predicted, because the predictability of the performance decreases with the complexity of the GRIS and the GIIS hierarchy.

All the above-discussed discovery services focus on general resources which could vary from hardware resources to documents resources. There is no discovery service specific to software components, supporting heterogeneity and various levels of component matching. The next section discusses the discovery services specific to software components.

2.3. Component Discovery Service

2.3.1. Agora

Agora [SEA98] is a software prototype, developed at the Software Engineering Institute (SEI). The objective of this work is to create an automatically generated, indexed, worldwide database of software products classified by a component type. Agora is developed for COTS based systems, to search for software components. Agora combines introspection with web search engines to reduce the costs of finding the components in the software marketplace.

According to [SEA98], a step towards integrating component technology and web search can have an impact on the emergence of an online component marketplace by:

- Providing developers with a worldwide distribution channel for software components.
- Providing consumers with a flexible search capability over a large base of available components.
- Providing a basis for the emergence of value-added component qualification services, within and across specific business sectors.

The location and indexing of components and the search and retrieval of a component are the two basic processes supported by Agora. Although the location and indexing of components is primarily an automated background task, there is a human intervention in search and retrieval.

Components are introspected during the indexing phase to discover their interfaces. Introspection is primarily associated with JavaBeans and is described by the ability of JavaBeans to provide information about their own interfaces. CORBA provides a similar capability of divulging information about interfaces, although these data are maintained external to the CORBA server in an implementation repository.

Component information can be differentiated into different fields such as component name, type, methods, attributes, and events. The component search in Agora starts by specifying the query terms, which is searched against the index collected by the search agents. The result set is inspected by the user and based on the number and quality of matches, the search could be broadened or refined. Agora also supports certain advanced search criteria by allowing Boolean logic operators.

Although this work is similar to the intent of this thesis work, Agora does not consider various levels of component matching. Also the search process does not consider user profiling and peer profiling, where the components searched depend on the user interest and also on the performance during the previous queries.

2.3.2. UniFrame Resource Discovery Service

UniFrame Resource Discovery Service (URDS) [NAN02] is a component resource discovery service, which is part of the UniFrame research project [RAJ01]. The URDS architecture provides the necessary infrastructure for an automated discovery and the selection of components meeting the necessary criteria. The components could belong to different component models such as Java RMI, .NET, and CORBA. These components are developed and deployed in the network by the component developers. URDS is designed as a discovery service wherein new services are dynamically discovered while providing the system developers with a directory style access to components. The motivation was to automate the process of assembling a DCS, for which components are discovered using URDS. The URDS infrastructure comprises of Internet Component Broker (ICB), Headhunter and Active Registry. The URDS is discussed in detail in

Chapter 3. This thesis work is the enhancement of the URDS with more advanced features and searching capabilities such as user profiling, enhancement in the query propagation techniques, and improvement in the quality of the discovered components.

2.4. Search Engines

Search Engines such as Google, Lycos, AltaVista, WebCrawler, Infoseek, and A9, enable users to search for and locate information on the web. Search engines use indexes that are automatically compiled by computer programs (such as robots and spiders) and that go out over the Internet to discover and collect Internet resources. A few of the prominent search engines are discussed below.

2.4.1. Lycos

Lycos [MAU97] is a search engine used for collecting, storing, and retrieving information about pages on the web. Spiders are the key for the search engines, which crawl in the web to search for relevant documents. One major difference between Lycos and Spiders is in the way the searching is done. Some Spiders explore the most recently found page first, resulting in a depth-first search of the web. In this case, the load on the target servers is high, sometimes causing the servers to crash. Lycos uses best-first search, which tends to find home pages rather than subsidiary pages, and so the Lycos catalog is biased toward more popular and more useful web pages. Lycos pioneered the use of automated abstracts. This means that Lycos identifies the 100 most important terms and creates an abstract that is about one-fourth of the original document (for all the documents that are available for matching). These abstracts are used during the search to find the relevant documents. They are also displayed along with the list of relevant links, allowing the users to quickly determine the relevant documents. For matching the query with the relevant documents, three basic schemes have been used: Boolean keyword query, regular expression matching, and vector space (statistical) retrieval.

2.4.2. Google

Google [BRI98] is designed to search for the resources in the Internet by crawling and indexing the web efficiently. Google ranks the search results, which is determined by several factors, including the patented PageRank algorithm. The PageRank algorithm relies on the democratic nature of the web by using its vast link structure as an indicator of an individual page’s value. A link from page A to page B is interpreted as a vote, by page A, for page B. Google also analyzes the page that casts the vote; the more important the page that cast the vote is, the more important is the weight of the vote. Apart from the vote, Google also remembers each time it conducts a search, which aids important and high-quality sites to receive a higher PageRank. Google combines PageRank with sophisticated text-matching techniques to find pages that are both important and relevant to the given search. Google goes far beyond the number of times a term appears on a page and examines all aspects of the page's content, along with the content of the pages linking to it, to determine if it's a good match for your query. Google also supports obtaining personalized search results that depend on profile settings, where the user selects the categories of topic (such as computers, science, recreation, music, sports, etc.) that interests them.

2.4.3. A9

A9 [AMA04], a new search engine from Amazon.com, is mainly created to be used as a search engine for e-commerce. A9 offers users search results from five powerful information sources: web and image search provided by Google, book text from Amazon’s Search, movie information from the Internet movie database, and reference information (encyclopedia, dictionary, etc.) through GuruNet.com. A9 offers a set of additional features, apart from the regular search capabilities. The concept of saving search results is hardly new and has been incorporated in A9 search engine, i.e., A9.com is a search engine with a memory, as it returns results from the user’s information. So, with every search, users will see results from their own history, bookmarks, and diary. This adds user customization to the search results.

2.5. Profiling in a Discovery Service

2.5.1. User Profiling

One of the major challenges faced in the distributed network resource discovery is the plethora of information matching the given query. The result set should be restricted depending on the preference of the user. A few of the information retrieval projects, which focus on customizing the query results based on the user profiles are discussed below.

2.5.1.1. GESTALT Resource Discovery Service

GESTALT RDS [DON01] was designed to improve the resource discovery over the Internet, to help customers locate educational courses and resources. This discovery service offers facilities for searching, previewing, and ordering educational courses. It offers three different types of services to the users, namely text-based search, structured search and complex search. The complex search is an extension of the structured search which uses the profile information.

According to [DON01], the GESTALT project defines the following set of requirements on the search to meet the end-user search requirements:

- The search utility should be fast and easy to use.
- The search result should provide the customer with all the information he needs about a product.
- The search engine should confirm the authenticity and quality of the product and supplier.
- The search/locate facility should tell the customer if the product is available.

Along with these requirements, this resource discovery service maintains the user profile information. The users are allowed to enter and manage information about themselves and their service requirements, which is used by the resource discovery service to aid in improving the search results. Defining standards for the user profiles is one of the important factors in the successful specification of the customizable resource discovery service. The resource discovery service interacts with the LDAP directory service to manage profiles. In GESTALT, a CORBA-based profile management service offers the ability to manage, store and retrieve user profiles data. This profile manager handles the operations between the client, server, the LDAP directory server and the XML implementation. XML was chosen as the binding for the profile information and the data exchanged during a search. Based on the Document Type Definition (DTD) defined for the user profile, the client and server can easily exchange data. An XML string representing the user profile is created and sent by the client to the LDAP server via a CORBA IDL interface. The LDAP API used by the resource discovery service to enable profile storage and retrieval is based on Netscape directory SDK 3.0 for C, which requires conversion of the data structure (MFC based data structure extracted from the XML document) to a C based data structure for LDAP and visa-versa. Wrapper classes are created to handle this profile information based on the DTD and LDAP, which ensures the generic nature of this resource discovery service.

2.5.1.2. GUARDIANS

The Information Society Technologies (IST) project GUARDIANS [ROU01] has come up with a more complex resource discovery service than GESTALT. It describes models for meta-data, which provides information about the user as well as the content. The argument made in [ROU01] is that it is important to gather information on preferences, activities and characteristics of users as they facilitate personalized content delivery. The issue of user profiling is quite complex because of the necessity of the profile information to be generic. A complex, extensible, generic user profile called the Generic User Profile (GUP) is developed, based on the IMS Learning Information Package (LIP) specification [JON03]. The information in this profile is grouped into sections such as accessibility, affiliation, usage history, relationship, interest, contact details, qualifications and goals. The main extension of GUP from that of the IMS LIP is the usage history, where all the previous searches are stored and used during future queries.

The GUARDIANS also focus on the Information Service Provider (ISP) profile. By this, the user could select the service provider, depending on user's requirements. The ISP profile stores both the domain specific metadata and the generic metadata. It also defines an aggregated view of the information stored in its repository, and also enables operating across multiple domains. Examples of the information maintained in the ISP profile are service name, description, contact details, metametadata, legal constraints, payment information, security information, relationship, classification, taxonpaths, and keywords. Among all this, the important element would be the classification i.e., the categories covered by the service provider. Depending on this information, the user could select the information service provider. In addition, some ISPs may wish to extend their profiles from the defined standard structure. Apart from the user profile and the ISP profile, it also maintains storage and retrieval profiles. The success of the query depends on the way these profiles are mapped and the profile retrieval performance.

2.5.1.3. Cheshire project

The Cheshire project [LAR99], also known as the Cross-Domain Resource Discovery: Integrated Discovery and use of Textual, Numeric and Spatial data, is aimed to develop a search and retrieval system that is capable of searching cross-domain resources held in multiple locations. Cross-domain means that the resources discovered belong to different domains, such as textual, numeric, and spatial. The Cheshire project will allow users to locate and retrieve information about collections that are organized hierarchically and distributed across servers. It also supports user profiling and authentication. The basic architecture is of three tiers, namely the Client, the application tier and the repository. The data delivery format is XML, while the data manipulation and navigation is through CORBA-enabled Java applets. One of the main advantages of this Cheshire project is the support for cross-domain resource discovery. This project also specifically addresses the critical issue of vocabulary control by supporting the probabilistic best match ranked searching and support for Entry Vocabulary Modules (EVM) that provide a mapping between a searcher's natural language and controlled vocabularies used in the description of digital objects and collections. Although this project deals with probabilistic search, the resources are restricted to textual, numeric and spatial data.

2.5.1.4. Z39.50 Information Retrieval Protocol

The Z39.50 Information Retrieval Protocol [INF01] aids in the electronic discovery of resources in the library community. According to [INF01], an effective access to electronic information in the Digital Library community must solve problems of Information space complexity and Resource and information source heterogeneity. Here the protocol addresses a few solutions to the problem of context at both the resource and user levels. These solutions could be: having structured text, Metadata, controlled vocabularies, query by example and user profiling. By maintaining the profiles of the users, the results of the search are being customized to the preference of the users. The profile of the user could be based on the history of the responses or demographics. Metadata has been defined as being structured information about information. In a large distributed network, where metadata could be defined by many different people, it is essential that there be some standardization of metadata to enable semantic interoperability. The techniques of fragmenting the information space into community specific servers and using meta-searching to re-unify the information space are important for controlling complexity and providing scalable discovery. Although [WAR01] tries to solve the problem of Information space complexity, information source heterogeneity and customization of user results by profiles, the resources are restricted to documents in the library community.

2.5.1.5. Resource Discovery Network (RDN)

The RDN-I [RDN01] search is an Internet service dedicated to providing an effective access to high quality Internet resources for learning, teaching and resource community. This service is primarily aimed at Internet users doing higher education. RDN provides an access to a series of Internet resources of high quality, selected and described by specialists from within UK academia and affiliated organizations. It has value-added services such as interactive web tutorials and user profiles. RDN Subject Portals Project (SPP) [RDN02] is an aggregated cross search tool that presents the users with a tailored view of the web within a particular subject area. The result of the information retrieval depends on the user profiling data.

There are other information retrieval projects which uses user profiles to customize the search results according to the preference of the user. Mostly in these kinds of information retrieval systems, the result set is of a wide range and user profiling helps to narrow down the result set depending on the preference of the user. But, all the research and project work done in user profiling is restricted to resource discovery and document discovery. By applying the profiling in component discovery, the expectation is that the discovered components will be customized to the system developers interested in developing DCS.

2.5.2. Peer profiling

Although the user profiling customizes the search results based on the preference of the user, there is no profiling activity involved in the resource discovery process. The increment in the available resources results in an increment in the time and complexity to search for the required resource. To have an efficient search mechanism, the query should be propagated to a better subset of nodes that is more likely to return matching resources. So, profiling of the peer nodes involved in the search mechanism could help achieve this task. Some of the work done in peer profiling is discussed below.

2.5.2.1. Decentralized Discovery

[DRD01] discusses resource discovery in peer-based networks. It argues that, for a scalable, large peer-based system, an adaptive social discovery mechanism is more appropriate and will work efficiently. Here, each peer directly controls the peers it communicates with, the way bandwidth is consumed and the network is used. [DRD01] stresses maintaining a history of the peer nodes, which could aid in the selection of the peers. Communication with a peer node requires a direct connection to that peer and the longevity in this connection helps the peers to maintain the history of the connection, which in turn is used for reputation management and optimization of the discovery service.

2.5.2.2. Ontology-Driven Peer profiling

The Ontology-Driven Peer profiling in Peer-to-Peer (P2P) enabled semantic web [PAR01] delves into ontology-driven peer discovery. The proposed ontology-based peer profile will enhance the peer communication mechanism with its metadata representation providing a bridge for the semantic search and discovery of information and services hosted on the peers. With this peer profiling, depending on the response to the queries, the profiles are updated, which aids in maintaining up-to-date knowledge about each peer. Unlike the publish-and-subscribe method used in conventional peer management, here a query is sent to a peer based on the contents and resources of that peer. This kind of profile management can eliminate some major issues persistent in current P2P networks such as security, resource aggregation and group management. The diversity of the peers could affect the discovery of the peers. Examples of this diversity could be due to the peers being wireless or mobile devices, which have their own range of communication.

2.5.2.3. Query Processing in Self-Profiling

The Query Processing in Self-Profiling composable P2P Mediator database [KAT01], aims to develop a query compilation and execution techniques that will allow for many autonomous and distributed sources to be integrated and queried effectively through a mediator system. This system tries to fulfill the challenges such as autonomous and distributed nature of the participating entities and integration of complex and diverse data which requires a lot of domain knowledge. The mediators must adapt to the changes in the environment by measuring and storing various parameters. The intent is to integrate the database profiling system with the query processor of each mediator peer by measuring system performance and managing various environment data. By accumulating the measured information, the query processor of the mediator could use that for making better future decisions. A few of the major challenges faced are minimizing the performance penalty of profiling, ensuring that that the necessary profiling data can be accessed very quickly and dynamically controlling the parameters being measured.

Peer profiling is implemented in many P2P systems. In URDS, the Headhunters could be considered as peer nodes. The selection of headhunters for the query propagation is crucial, particularly when the response time requested for the component discovery is small. So, the implementation of peer profiling in the URDS will help in propagating the query to a better sub-set of headhunters for component discovery, as opposed to random selection of headhunter for the query propagation. This profiling could also be extended to the Active Registry, to have a better search of the components.

This chapter provided a discussion of the background of the thesis and brief descriptions about a few of the related works. It also discussed a couple of existing component discovery service architectures. This thesis is based on the URDS architecture and the enhancement of the same. The next chapter discusses the existing URDS in detail. It also discusses the enhancements possible to the URDS architecture.

CHAPTER 3. THE URDS ARCHITECTURE

Chapter 2 provided the background and related work in various topics such as resource discovery services, component discovery service, search engines, and profiling in discovery service. This chapter presents a detailed discussion on the existing UniFrame Resource Discovery Service (URDS) architecture. It also discusses about the possible enhancements of the URDS architecture.

3.1. Architecture of URDS

The URDS architecture [NAN02] provides the necessary infrastructure for discovering the heterogeneous components across the network, matching a given query. These heterogeneous components are combined to form a distributed computing system (DCS). The quality of this DCS would depend on the quality of the discovered components and the compatibility between them. The URDS architecture is organized as a federated hierarchy in order to achieve scalability. The discovery of components in URDS is domain based i.e., the location of services is within an administratively defined logical domain, which refers to industry specific markets such as Document management services, Financial services, Health care services, etc. This section discusses about the URDS architecture in detail.

Figure 3.1 shows the URDS architecture and the sub-components that comprises the URDS such as Active Registry, Headhunter and the Internet Component Broker. These sub-components of the URDS are discussed in detail.

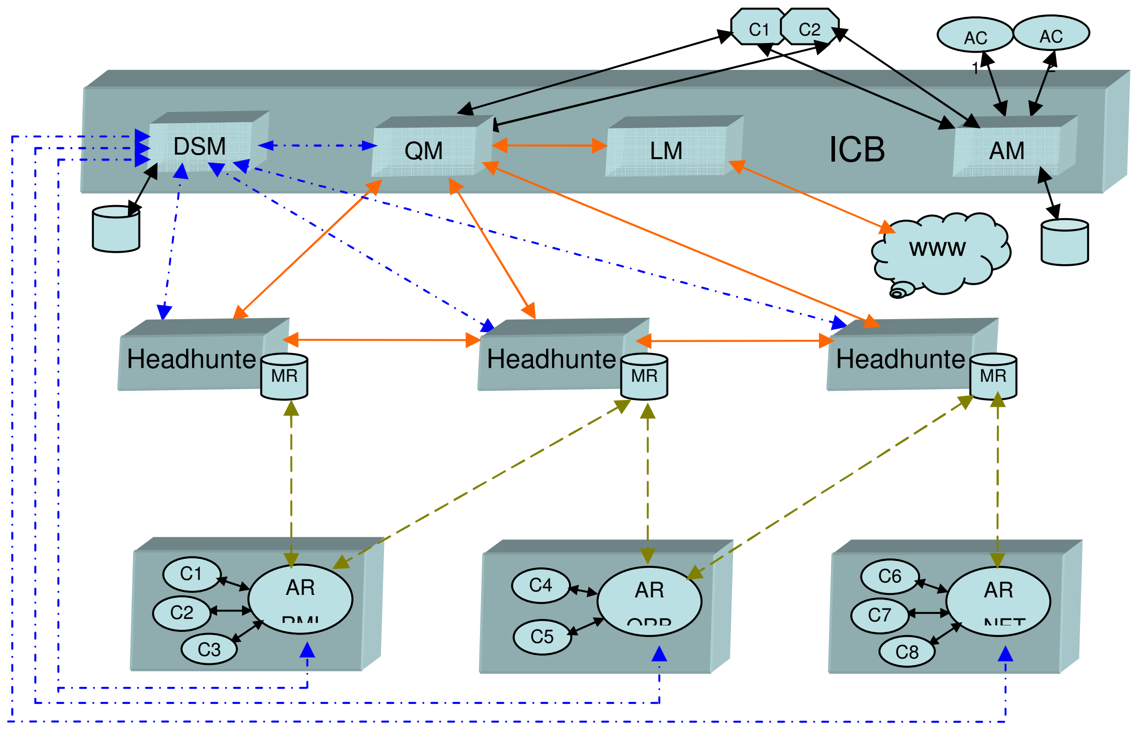


Figure 3.1 URDS Architecture

Legend:

◄·-·-·► Domain Security Manager authentication messages

◄- - -► Meta-repository component update messages

◄——► Query propagation and result returning messages

ICB: Internet Component Broker

DSM: Domain Security Manager

QM: Query Manager

AM: Adapter Manager

LM: Link Manager

MR: Meta-repository

AR: Active Registry

C1, C2…: Components

AC1, AC2…: Adapter Components

3.1.1. Active Registry

The components are developed by the component developers and are registered in the Active Registry. Each Active Registry belongs to a particular domain and only components belonging to that domain are registered with the registry. Each registered component could belong to one of the distributed models such as Java RMI, .NET, or CORBA.

3.1.2. Headhunter

A Headhunter is responsible for returning the components which satisfy the given query. A Headhunter belongs to a particular domain and only queries belonging to that domain are forwarded to that Headhunter. Each Headhunter maintains a list of components in its meta-repository, which are actively updated from one or more Active Registries of the same domain. The communication between a Headhunter and an Active Registry is by multicasting, where the Active Registries listen to the multicast messages from the Headhunters. Each Headhunter receives the queries, matches the queries with the components present in its meta-repository and returns the components that match the query to the requestor (node which requested for the components).

3.1.3. Internet Component Broker

The Internet Component Broker (ICB) encompasses the communication infrastructure necessary to identify and locate services and to enforce security. It consists of the Domain Security Manager, Query Manager, Link Manager, and the Adapter Manager. All these services will be located at well-known addresses. There will be one or more ICBs deployed in the network.

3.1.3.1. Domain Security Manager

The Domain Security Manager (DSM) authenticates the Query Managers, Headhunters and the Active Registries. It serves as an authorized third party that handles the secretkey generation and distribution. DSM also enforces group memberships and access controls to multicast resources through authentication.

3.1.3.2. Query Manager

The query from the system developer is dispatched to the appropriate Headhunters by the Query Manager. The Headhunters belonging to the same domain as that of the query are selected for the query propagation. The Headhunters will return the component list matching the given criteria to the Query Manager. Then the Query Manager sends the list of components from all the Headhunters to the system developer for component selection, and the selected components are used to create the system. In addition to that, the query may be propagated to other linked ICBs through the Link Manager.

3.1.3.3. Link Manager

The Link Manager serves as a link for connecting one or more ICBs, for the purpose of federation. The query from the Query Manager is sent to the other ICBs by the Link Manager i.e., the query is sent to the Link Manager, which contacts the Link Manager of the other ICBs in the network.

3.1.3.4. Adapter Manager

The Adapter Manager serves as a registry or lookup service for the adapter components. The Adapter components are used as bridges to combine the heterogeneous components to form a DCS. The specialization of each adapter component is also indicated, i.e., the component model that it can bridge efficiently.

3.1.4. Working of URDS

The working of the URDS is explained by two activities, one of which is updating the component information in the meta-repository and the other is the discovery of components for the given query. Updating the meta-repository component information happens periodically, while the component discovery happens whenever a query is given to the URDS.

The URDS meta-repository refresh activity is based on periodic multicasting. Initially, the DSM authenticates the Headhunters and the Active Registries. The Active Registries listen in a particular location for the messages from the Headhunters. The Headhunters will multicast their presence to the Active Registries of the same domain. Each Headhunter then obtains the component information from one or more Active Registries and updates its meta-repository. This update of components happens periodically so that the new components that are registered with the Active Registry are reflected in the meta-repository of the Headhunter that communicated with this Active Registry.

The component discovery process happens whenever a query is available to the URDS. The system developer queries the Query Manager for discovering the components to create a DCS. The Query Manager gets the list of Headhunters belonging to the same domain as that of the query from the DSM and propagates the query to those Headhunters. Each Headhunter match the propagated query with the components in its meta-repository. This matching could involve matching of both the functional and non-functional attributes. The matched components are returned to the system developer for selecting the components to form the system. Once the components are selected, appropriate adapter components are selected through the Adapter Manager. These adapter components are used to bridge the matched components in forming the DCS, incase of the matching components being heterogeneous. Apart from the Query Manager propagating the query to the Headhunters, it also propagates the query to the Link Manager through which the query is sent to other ICBs. Once all the components required to form the system are obtained, a DCS is formed by combining those components with the help of the selected adapter components.

3.1.5. Features of the existing URDS architecture

The existing URDS architecture can be summarized by elucidating the following features.

- The discovery of components is based on the specification matching between the requirements specified in the query and the components available in the Headhunters' meta-repository through the UniFrame Resource Discovery Architecture.
- The query is propagated all the available Headhunters belonging to the same domain as that of the query.
- The query propagation hierarchy is formed by the random selection of the Headhunters.
- The query propagation between Headhunters is based on synchronous function calls, i.e., the control in a Headhunter that propagated a query will wait until the query-recipient Headhunter matches for the components and returns the component results to the requestor.
- The components in the meta-repository of the Headhunters are updated frequently from the Active Registries that the Headhunters communicate.
- All the matched components for the given query are returned to the Query Manager without any differentiation in the level of match.
- Choice is given to the system developer in selecting the components to form a system from the components that are matched and returned by the Headhunters.
- Adapter components are selected to efficiently bridge the components to form a system.

3.2. Enhancement of the URDS architecture

The existing URDS architecture provides basic framework for the discovery of components to form DCS. This architecture could be enhanced to provide an efficient component discovery service to the system developers. The following sections discuss the possible enhancements, the reasons for providing those enhancements, the underlying challenges faced due to those enhancements, and the benefits of those enhancements.

3.2.1. Reasons for the enhancement of URDS

The existing URDS architecture propagates the query to all the available Headhunters. So, there is no control over the response time of the discovery of the components as the component discovery time depends on the number of Headhunters available for that domain. The query propagation techniques in the URDS could be enhanced by selecting a subset of Headhunters for the propagation of a query. The assumption that the behavior of the Headhunters for the future queries can be predicted based on their past behaviors can be used to select this subset of Headhunters for the query propagation based, as opposed to a random and exhaustive selection. Here behavior could be measured by the Headhunters' capability to return components that closely match the given query. This will improve the response time of discovering the components compared to exhaustive search, still maintaining comparable quality of the matching components for a query. This requires the maintenance of the behavior of the Headhunters for the past queries.

The propagation of the query to the Headhunters is implemented through synchronous function calls in the existing URSD, i.e., if the Query Manager propagates a query to a Headhunter, the Query Manager is blocked until the Headhunter responds for the propagated query. Similarly, if a Headhunter propagates a query to, say three other Headhunters, the propagating Headhunter gets blocked for each of the propagations. So the propagation to more than one Headhunter happens sequentially, thus wasting time simple waiting for the results. Since all the propagations are implemented through synchronous function calls, the URDS can process only one query at any given point in time, thus making all the other queries wait until the previous queries are processed.

The existing URDS architecture does not support ranking of the matching components, i.e., the components are not differentiated based on the different levels of match with the given query. Having different levels of matching and assigning rank based on this match will aid the system developers in selecting the potential components that could form an efficient and effective DCS with a good Quality of Service (QoS). The discovered components could be tested dynamically for the functionality of the components and the compatibility of each component with other components that form a DCS. This will assure the functioning of the individual components selected, and thus the functioning of the DCS formed from those individual components. Also, the testing of the individual components is essential to make sure the components that form the DCS have the functionality that they claim to have. The next section lists the possible enhancements that could be provided for the existing URDS architecture.

3.2.2. Possible enhancements of URDS

The existing URDS architecture could be enhanced by adding the following features to facilitate an efficient and effective discovery of components. These enhancements together can achieve a few of the features of the ideal component discovery service (discussed in the previous chapter).

- Improving the performance of the URDS by various means, such as overlapping the component discovery process for the queries given by different system developers by handling the query propagation asynchronously and improving the time taken for the component discovery process.
- Profiling at different levels in the URDS hierarchy – This feature is used for discovering the components based on user customization by maintaining the history of component discovery, which could aid in improving the future component discoveries.
- Ranking of the discovered components – This enhancement is to aid the system developers' component selection process for creating a reliable DCS.
- Supporting various levels of component matching such as exact match and relaxed match – This feature is to give more flexibility to the system developer in expanding or pruning the discovered components that match the given query.
- Efficient query propagation techniques for component discovery – This enhancement is to decrease the response time for the component discovery by propagating the query to only a subset of Headhunters, thus maintaining comparable quality of components that would have resulted in a comprehensive component search.

- Dynamic component testing capability to test for the functionality of the components – This is used to test the discovered components dynamically for their functionality and compatibility, before using them as part of the DCS.
- Considering only the new components when the same query is re-propagated – A query could be re-propagated (maybe with slight modification), either due to dissatisfaction in the discovered components or to expand/prune the component set discovered. This feature is used during the query re-propagation to avoid discovering the same components that the system developer already came across and not is interested in them.

3.2.3. Underlying challenges faced

There are a few challenges while incorporating these enhancements into URDS, which are discussed below.

- Each Headhunter could have its own criteria for matching the query and ranking the components based on the confidence level it calculates for the components. These criteria for the confidence calculation and hence, the ranking of components in the Headhunters could differ, thus making the comparing and matching of the component results a difficult task.
- Profiling on Headhunters, at the level of the Query Manager and the Headhunter, aids in minimizing the number of Headhunters to which the query will be propagated. But, at the same time, the quality of the components returned should be comparable to the ones obtained as if the query had been propagated to all available Headhunters belonging to same domain as that of the query. Selecting a subset of appropriate Headhunters is difficult, particularly when there is a possibility that the preference (such as QoS, response time, cost, etc.) of the system developers could vary for each query.
- Each entity that forms the URDS could maintain a trust/confidence value about other entities. This confidence value plays a role in the selection of a subset of Headhunters for the query propagation and updating the meta-repository components. So, a miscalculation in these confidence values could lead to inefficient query propagation and selection of components that do not closely match the given query. Also, these

confidence values need to be updated periodically based on the entity's (i.e., the entities that form the URDS) interaction with them. Maintenance of these confidence values and periodically updating them are complex and challenging.

3.3. Features of the enhanced URDS

This section discusses in detail, the features of each of the enhancement discussed in the earlier sections.

3.3.1. Profiling

Although there are a lot of definitions for profiling, in this context it could be summarized as a technique that develops a pattern based on the history of events and applies a selection criteria for the future events, thereby improving efficiency. The use of profiling is quite complex, as the profile information should be very generic, with a compromise between storing too little and too much information. Profiling could be based on reinforcement learning [KAE96], which is defined as a problem faced by an entity that must learn behavior through trial-and-error interactions with a dynamic environment. The learning process is aided by assigning a reward/penalty for the trial-and-error actions performed. The reward or penalty depends on the users' feedback for the performed action. The history of these rewards and penalties are used to decide the next action. A survey on the reinforcement learning techniques is discussed in [KAE96].

Profiling is done by maintaining a history about the queries propagated and the responses obtained for those queries. The maintenance of the profile information could be done at various levels in the URDS hierarchy. This profile information aids the system developer and the entities of the URDS in taking better decisions in terms of discovering relevant components matching the given query. The basics of profiling and the reinforcement learning are adapted from [MUK02, THA85, THA00, and OOM90].

3.3.1.1. Levels of profiling for URDS

Profiling could be done at various levels in the URDS hierarchy. The reasons for maintaining profile information at different levels of the hierarchy are discussed below.

(1) Query Manager

a. Headhunter profile

As indicated earlier, a Query Manager propagates the query to the Headhunters for them to return the matching components. One possibility to reduce the response time associated with the component discovery process is to propagate the query to a few selected Headhunters, as against to propagating the query to all available Headhunters. The query propagation to this subset of Headhunters should not compromise the quality of components matched and returned. So, the selection of this subset of Headhunters should be in such a way that these Headhunters manage to discover components having quality comparable to that in the case of propagating the query to all the available Headhunters. So, the query manager maintains the profile information of Headhunters, which aids in selecting a better subset of Headhunters for the query propagation.

(2) Headhunter

a. Query Manager profile

A Headhunter responds to a query from the Query Manager by returning a list of components that matched the query. A Query Manager could always ignore the components from a Headhunter due to various reasons such as lack of trust, or lack of satisfaction in the Headhunters component matching process. The Headhunters need not bother to match the components for the queries from such Query Managers. So, the response of a Headhunter for the component matching process could vary to different Query Managers. So, the Headhunter maintains profile information about the Query Managers, which is used to decide on this query response.

b. Headhunter profile

Whenever a Headhunter receives a query, apart from matching the query against the components in its meta-repository, it also propagates the query to one or more other Headhunters. The selection of Headhunters for this propagation would affect the quality of components returned by this Headhunter to its requestor (node that requested components by propagating a query), thus affecting the rating of this Headhunter with respect to the Query Manager or other Headhunters. So, returning better quality components to the requestor (i.e., a Query Manager or another Headhunter) would improve its chances of being contacted more often for the future queries.

c. Active Registry profile

The components in the meta-repository of the Headhunters are updated from the Active Registry. A lack of good quality components from the Active Registry could be a reason for a Headhunter to avoid adding component information from that Active Registry into its meta-repository. Also, the components from an Active Registry that fail to have the features claimed in its component specifications could be avoided. So, the Headhunter maintains the profile information of each Active Registry, which aids in selecting appropriate Active Registries for updating the component information in its meta-repository.

(3) Active Registry

a. Headhunter profile

Usually a Headhunter recommends the components obtained from the Active Registries, to its requestor (i.e., a Query Manager or another Headhunter). A Headhunter might not recommend the components from an Active Registry due to various reasons, such as lack of trust and bad experience during the past interactions. In this case, the Active Registries could avoid the hassle of sending the components to such Headhunters. So, the Active Registry could maintain a profile of the Headhunters, so that it has trust values on the Headhunters. These trust values could help the Active Registries to accept or deny the registration of a component to a Headhunter. A few other reasons for denying a component to a Headhunter could be due to the Headhunter charging more for the use of the components than authorized, leaking component information to unauthentic requestors, etc. Also, this profile is used to make a decision on the level of importance to be given to the queries and feedbacks obtained from the Headhunters.

b. Component Developer profile

The components are registered in the Active Registry by the component developers. Since an Active Registry is being profiled by the Headhunters, the Active Registry should give high-quality components to the Headhunters for it to be contacted more often. So, the rating of an Active registry with respect to a Headhunter would depend on the combined quality of the components that the Headhunter updated from this Active Registry. This in turn depends on the combined quality of the components from the component developers registered in this Active Registry. So, the Active Registries maintain component developers' profile information, which could avoid the component developers whose components cannot be trusted. A few of the reasons for avoiding a component from a component developer could be because of the lack of appropriate QoS or the component features not matching the features claimed in the component specification.

Any kind of the profile information needs to be updated constantly, i.e., updated for each query propagated. A few ways to update the profile information could be user feedback on components, insight of a node on other nodes, and responses for the previous queries.

3.3.1.2. Challenges in profiling

Although profiling has been used in many domains, applying profiling in the component based systems for the discovery of components has a few challenges. Maintaining the profile information in all the levels of the URDS hierarchy provides a lot of options in manipulating this profile information. This profile information is mostly based on the responses to the previous queries. There are a lot of choices and alternatives at each stage in deciding the rating for the responses, which could be a reward or a penalty. Sometimes the responses could just have inaction, which means there is neither a reward nor a penalty for a response. Also, discounting could be applied to the profiling. Discounting could be defined as weighted average for the responses, thus giving more importance to a few responses (say recent responses) and relatively less importance to a few other responses (say old responses).

A few of the factors for profiling are based on the users' interests, the domains to which profiling is applied, and the performance of profiling in these domains. The hierarchy in the component discovery architecture provides more options in the way the profile information could be maintained, thus intensifying the challenges in profiling.

3.3.2. Component ranking

The ranking of components aids the system developers to select the components for developing a system. The confidence level of the components matching the query will reflect the ranking of the components. The Headhunters could provide this ranking service to the component requestors. The factors that are considered for coming up with a confidence measure for a component are based on the component specification matching done at different levels such as syntactic, semantic, synchronization, and QoS. The importance given to these factors in deciding the confidence level is discussed in [KUM04], and this thesis uses them without further investigation. The confidence measure will also depend on the rating of the Active Registry with which the component is registered, and the rating of the component developer who developed those components.

3.3.3. Levels of Component matching

Two component specifications can be matched for different criteria such as substitutability and compatibility matching, generic and specialized matching, and exact and relaxed matching. This has been discussed in detail in [KUM04], and this research uses those principles without further investigating them.

3.3.4. Query re-propagation

The system developer propagates the query and gets back the list of matching components from the Query Manager. The system developer selects the components from the component list and forms the required system. The system developer, after obtaining the results for the propagated query, can modify and re-propagate the query. This re-propagation could be done for one or more reasons. One reason could be to prune the component list returned, as the list could be too long. One another reason could be that the system developer is not satisfied with the returned components. So, the system developer should be given an option during the re-propagation whether or not to request again the components that are already selected and returned.

3.4. Enhancements considered in the scope of this thesis

Although there are many enhancements that could be applied to the existing URDS architecture, only a subset of the enhancements is considered in the scope of this thesis. The enhancements considered are: increasing the robustness of the URDS architecture, improving parallelism by making the query propagation as asynchronous function calls, decreasing the response time for the component discovery process by increasing the parallelism, improving the query propagation techniques by implementing reinforcement learning algorithms, adding profile information at different levels in the URDS architecture, and ranking of the components that match the query. This subset of enhancements is incorporated in the enhanced URDS and they are tested experimentally. The reason for considering these enhancements is to improve the quality of the components discovered in a restricted time.

This chapter provided an overview of the existing URDS architecture and the possible enhancements of the same. The benefits, challenges, and features of the enhancements are also discussed in this chapter along with describing a subset of the enhancements that will be considered in the scope of this thesis. The next chapter will discuss the design details in incorporating this subset of enhancements into the existing URDS architecture.

Different versions of URDS are formed and all those architectures are compared by different experiments and validated for the enhancements incorporated in them.

CHAPTER 4. DESIGN AND IMPLEMENTATION OF THE URDS ENHANCEMENTS

Chapter 3 provided the detailed explanation of the existing URDS architecture and the possible enhancements of the same. It also discussed the underlying challenges to achieve these enhancements. This chapter discusses the enhancements achieved due to the performance improvement (measured in average response time for a query) and the incorporation of profiling in the URDS architecture. These two enhancements are the contribution of this research.

4.1. Performance Improvement

The performance of the existing URDS architecture can be improved by improving the time taken for the component discovery process without compromising for the quality of the discovered components. This involves improving the query propagation techniques (described below) incorporated in the URDS architecture.

The Query Manager and the Headhunter are the two entities that are involved in the component discovery process. The system developer interested in developing a system, submits a query to the Query Manager. The Query Manager propagates this query to one or more Headhunters. These Headhunters further propagate the query to a few other Headhunters as needed. Apart from these propagations of the queries to the Headhunters, each Headhunter which received the query searches for the components in its meta-repository. It then combines the matched meta-repository components with the components received from the Headhunters to whom it propagated the query and returns the combined results back to its requestor (the node from which the query was received).

The next section discusses the design and implementation changes made to the existing URDS architecture to achieve this performance improvement.

As discussed in the previous chapter, both the Query Manager and the Headhunters are single threaded and use synchronized communication in the existing URDS architecture. So, when a query is propagated from a Query Manager to a Headhunter, the control in the Query Manager is blocked until the Headhunter that received the query processes the query by matching for the components in its meta-repository and returns back the results. Because the Headhunters further propagate the queries to other Headhunters, the control in the Query Manager is blocked until each of the available Headhunters processes the given query. Also, the processing of the query in each of the Headhunters happens in a sequential fashion, as the Headhunters are also blocked during the propagation of the query to other Headhunters. The time taken for the component discovery process is a factor of the number of Headhunters that are available in the URDS, as there is no overlapping in the matching process between different Headhunters. During most of the time taken in the component discovery process, the Query Manager and the Headhunters are idle in this single threaded URDS version.

The performance improvement in the URDS architecture is done in terms of the time taken for the component discovery process. This improvement is achieved by overlapping the component matching process that happens in different Headhunters. This is attained by making the Query Manger and the Headhunters multi-threaded. The propagation of a query from one entity to another will follow asynchronous communication, so that the control is not blocked during the query propagation. This will improve the performance, as the query requestor will process other queries while waiting for the component results from the Headhunters to which query was propagated. The next subsection discusses in detail the design and implementation details of the multi-threaded version of URDS.

4.1.1. Design and Implementation details

4.1.1.1. Query Manager

The multi-threaded Query Manager implements a query queue and a result queue. New queries to be processed are placed in the query queue and the component results, from the Headhunters to which the query is propagated, are collected in the result queue. There is no restriction imposed on the size of the query queue, so that, no queries are ignored. A separate thread is allocated the task of monitoring the query queue, i.e., to process the queries that arrive in the query queue in the order of their arrival. This thread selects the first query in the query queue, and propagates it to one or more Headhunters. Once the selected query is propagated, this thread checks the query queue for more queries and repeats the task of propagating the available queries to the Headhunters. So, the processing of a query does not wait for all the previous queries in the query queue to be processed, i.e., the control is not blocked for each of the query propagations and the Query Manager in not idle unless there are no more queries in the query queue to be processed. The details of the Headhunters to which the queries are propagated are stored, so that the results can be collected from them. Another thread is assigned the task of monitoring the component results that arrive in the result queue. This thread checks whether the Headhunters which received the queries have processed and returned the results. Once all the results are collected for a particular query, they are returned to the system developer. So, by having different threads in the Query Manager to do different tasks, the query propagation process is overlapped between different queries, thus, reducing the query wait time and improving the component discovery process.

4.1.1.2. Headhunter

The multi-threaded Headhunters also implement query queues and result queues. Similar to the Query Manager, threads are spawned for monitoring the query queue and the result queue. When a Query Manager propagates a query to a Headhunter, the query arrives in the Headhunter's query queue. The thread in the Headhunter that monitors its query queue selects a few new Headhunters and propagates the query to them. Apart from this task, this thread also accesses its meta-repository for the matching components. This is because the task of selecting the new Headhunters for query propagation is trivial compared to the component discovery process, and takes less time. Also, the alternative of having a separate thread for the meta-repository access needs a separate queue to be implemented, thus increasing the overhead associated in storing the query information in that queue. So, the same thread does the job of propagating the query to the new Headhunters and accessing the Headhunter's meta-repository. To further improve the time taken for the component discovery, more than one thread could be allocated to do this task (i.e., propagating the query to the new Headhunters and accessing the meta-repository for the component search). This is done to overlap the processing of different queries in a Headhunter, thus reducing the query wait time in the query queue, and reducing the time taken for the component discovery process. The number of threads allocated for this is made dynamic, based on the query arrival rate. The thread that monitors the result queue collects and returns the results to its query requestor (the node from which it obtained the query). Apart from these processes, there is another thread, which is assigned the task of updating the components in the meta-repository from the Active Registry. This updating of the meta-repository contents is done periodically.

4.1.1.3. Query object

Each query placed in the query queue is a query object that carries a lot of details necessary for the component discovery process. These details include the following information.

- Query ID – A unique identifier in the URDS, assigned by the Query Manager.
- Abstract component – This is an important part of the query object, which carries the details of the component to be discovered. It has many attributes, each of which specifies either a functional or non-functional aspect of the component to be discovered. Examples of the information are the domain and the system for which the component is being searched, the desired quality of the component, the interface details of the component stating the desired functionality, the price willing to offer for the component,

etc. These attributes are matched against specifications of the stored components. These component specifications are called the Unified Meta-component Model (UMM) specifications [RAJ00].

- Response time – This attribute of the query object will specify the response time allowed for the component discovery process.
- Query source – The name and location of the query requestor i.e., the node from which the query was obtained. This is necessary to return the query results back to the requestor.

The multi-threaded version of the URDS improves the performance of the component discovery process. Experimental verification and analysis of this improvement is discussed in the next chapter. The next section discusses the application of reinforcement learning techniques and profiling on this multi-threaded URDS version. It also discusses in detail the algorithms that are incorporated in the Query Manager and the Headhunter as part of this reinforcement learning and profiling.

4.2. Profiling in URDS

The multi-threaded URDS architecture is further enhanced by adding profile information at different levels.

4.2.1. Architecture of URDS with Profiling

The enhancements considered in the profiled URDS architecture are improving the query propagation techniques using efficient query propagation algorithms, improving the quality of the components, and ranking of the discovered components that match the given query. These enhancements are added at different levels of the URDS architecture.

4.2.1.1. Query Manager

The query from the system developer is handled by the Query Manager. So, the Query Manager is an obvious choice for the profiling to be incorporated. The enhanced Query Manager has a separate account maintained for each system developer which is used to store his interests and preferences. This account information is updated after processing every query and obtaining the feedback from the system developer about the results of his query. This account information is mainly used to customize the component discovery process based on the preferences of the system developers. For each account, the Query Manager maintains a list of the Headhunters and their confidence levels. The confidence levels of the Headhunters are initially equal and are updated for every query based on the number of the components discovered by those Headhunters and their level of match. Each Headhunter assigns a confidence value to the matching components for a query (discussed in the next section) based on the level of match. The Headhunter confidence information is used to select a subset of the Headhunters for the query propagation.

An enhanced Query Manager includes the following features:

- Profile information

Profile information maintains the history of the component searches and the feedback for those component searches. The performance of the Headhunters for each query propagated from the system developers for the component search forms this profile information. This profile information is maintained separately for each system developer, so that the interests of one system developer do not interfere with the interests of the other system developers.

- Ranked Headhunter list

This list is used to improve the query propagation. Using this list, a subset of Headhunters to which the query is to be propagated is selected. This list is maintained for each system developer, so that it is customized to the system developers' profile information.

- Combine the resultant lists

The Query Manager propagates the query to one or more Headhunters. In the case of the Query Manager propagating the query to more than one Headhunter, it should have the capability to combine each component list from all the Headhunters before presenting the combined component list to the system developer. This combining of components should be based on the level of match of the component with that of the query.

- Query re-propagation

The Query Manager also supports query re-propagation. When the components matching the query are returned by the Query Manager, the system developer selects a few of the components to create the system. If the system developer is not satisfied with the component list returned or the component list needs to be pruned, then the query could be modified by the system developer. The modified query is re-propagated, during which an option of ignoring already returned components could be selected by the system developer.

- Feedback propagation

After the system is formed, the developer's feedback is propagated to all the nodes of the URDS. This helps in updating the Headhunters, Active Registries and component developer ratings based on their contribution in creating the DCS for which the query was propagated.

- Optional features for the system developers

There are a few optional features provided to the system developers. These optional features aid in customizing the query results to the interests of the system developers. A few of the optional features include:

  - Matching criteria (such as exact match, relaxed match). This feature will specify the level of the match required by a system developer for building a system.
  - An option for giving weighted/equal importance to the history of queries and their feedbacks. Equal importance could be given to all the feedbacks or a discounted importance could be given to the feedbacks. Discounted importance

means that giving more importance to the recent queries and feedbacks, compared to the relatively older queries and feedbacks.

Among the above-mentioned features, the profile information, the ranked Headhunter list, the combined resultant list, and the feedback propagation are implemented in the Query Manager. Details of this implementation are discussed in the later part of this chapter.

4.2.1.2. Headhunter

Each Headhunter maintains an account to store separate profile information, one for each preference types. The preference type is also sent along with the query, which is used to select the profile information to be used. This profile information aids in selecting a subset of other Headhunters for the query propagation and also in combining the component confidence lists returned from these Headhunters. An enhanced Headhunter has the following features:

▪ Maintains different Headhunter-ranked lists, one for each preference type. A preference type could be based on the importance given to the QoS (such as cost or response time) while searching for components. The Headhunter-ranked list is selected depending on the preference type requested by the entity that propagated the query.

▪ Active Registry profile information – This is used to select the Active Registries for updating the meta-repository. The functionality of the components might not reflect the properties mentioned in the component specification. The Headhunters should avoid maintaining such components in their meta-repository as they will mislead the component matching process. The Headhunters maintain profile information for the Active Registries so that the Active Registries hosting such components are ignored when updating the component information.

▪ Combine the component lists – A Query Manager propagates a query to the Headhunters. Each Headhunter could in turn propagate the query to other Headhunters. So, the results of all the sub-queried Headhunters should be combined before the Headhunter sends its ranked list of components to the Query Manager. Hence, each

Headhunter should have a capability to combine the resultant component lists from other Headhunters and also combine the matching components obtained from its meta-repository based on the levels of match, thus creating a single component-ranked list.

- Component confidence – Each Headhunter should have a feature to evaluate the confidence of the components with respect to the given query. The confidence level will be proportional to the level of match of the query with that of the component, i.e., the confidence level will give the relevance of the component to the given query. The different levels of match are discussed in detail in [KUM04]. Each query has a preference associated with it, which indicates the attribute of the components most desired by the system developer. The confidence of the components is also based on this preference. For example, a system developer might be interested in developing a system using components of a cheaper cost. So, the confidence of the matched components is assigned based on the cost of the components. This confidence level determines the rank the components.

All the above-mentioned features are implemented in the Headhunter, except for the component confidence where not all the levels of matching are considered in the implementation. The confidence is assigned to a component based on the basic match and the preference of the query. The details of the implementation are discussed in section 4.2.3.2.

4.2.1.3. Active Registry

The Active Registries will maintain the profile information about the component developers. The feedback obtained from the Headhunters for each query response is used to update this profile information. An enhanced Active Registry has the following features.

- A list of the components registered with the Active Registry.
- Headhunter profile – An Active Registry updates the components it has with the Headhunters. So, the profile information about the Headhunter helps to identify the Headhunter with which the Active Registry has to update the components.

- Component developer profile – The components, along with the component developer responsible for registering that component, are stored in the Active Registry. The functionality of the components developed by a component developer might not reflect the properties mentioned in the component specification. The components from such component developers should be avoided. So, the Active Registry maintains profile information about the component developer. This profile information is updated based on the feedback obtained from the Headhunters.

The above-mentioned features are considered in the implementation. The enhancements based on profiling are implemented only in the Query Manager and the Headhunters.

4.2.2. Design of the profiled URDS

This section discusses about the design details of each entity that forms the URDS, such as the Query Manager, the Headhunters and the Active Registries. The algorithms and techniques used in each of these entities are also discussed. Before going into the design details, the steps involved in the propagation of a query from the system developer, discovering the components that match the given query, presenting the discovered components to the system developer, and propagating the feedback for the whole discovery process from the system developer, need to be discussed.

The overview of the steps involved in the propagation of query in the profiled URDS is as follows:

- A system developer logins into the Query Manager.
- The profile information is maintained for the Headhunters, based on the response of the Headhunters for the previous queries. One or more Headhunters are selected based on this profile information and the query is propagated to those Headhunters.
- The Headhunters that received the query further propagate it to zero or more Headhunters. The selection of Headhunters for this propagation is based on the profile information maintained by the Headhunter, and the time given for the component discovery process.

- The Headhunters that received the query match the components for the query and return back the resultant components to its requestor. If a Headhunter propagated the query to other Headhunters, then it will combine all the results obtained before returning the component results.
- The Query Manager combines all the components obtained from the query-propagated Headhunters and will give it to the system developer.
- The system developer selects the components to be used for the system from the component list returned by the Query Manager. Then feedback for that component discovery process is sent to the Query Manager by the system developer.
- The Query Manager updates profile information based on this feedback information. Then the feedback is propagated to the Headhunters from which the components were obtained.
- The Headhunters updates this profile information based on the feedback information. They in turn propagate the feedback to other Headhunters, similar to the query propagation.

4.2.2.1. Query Manager

The Query Manager incorporates a few algorithms for the improvement of the URDS using profiling. This subsection explains the steps followed in the Query Manager for the component discovery process, and the algorithms used in each of the steps. The diagrammatic representation of the flow of activities in the Query Manager is shown in Figure 4.1.

- The profile information is selected based on the system developer who gives the query.
- A few Headhunters are selected for query propagation using the algorithm *HeadhunterSelect* (discussed below). This algorithm is applied on the selected profile information for the system developer. Also, the number of Headhunters to which the Query is propagated depends on the system developer's choice of the component discovery time given for the component discovery process.

- The query is propagated to the selected Headhunters (represented as HH1, HH2 and HH3 in Figure 4.1).

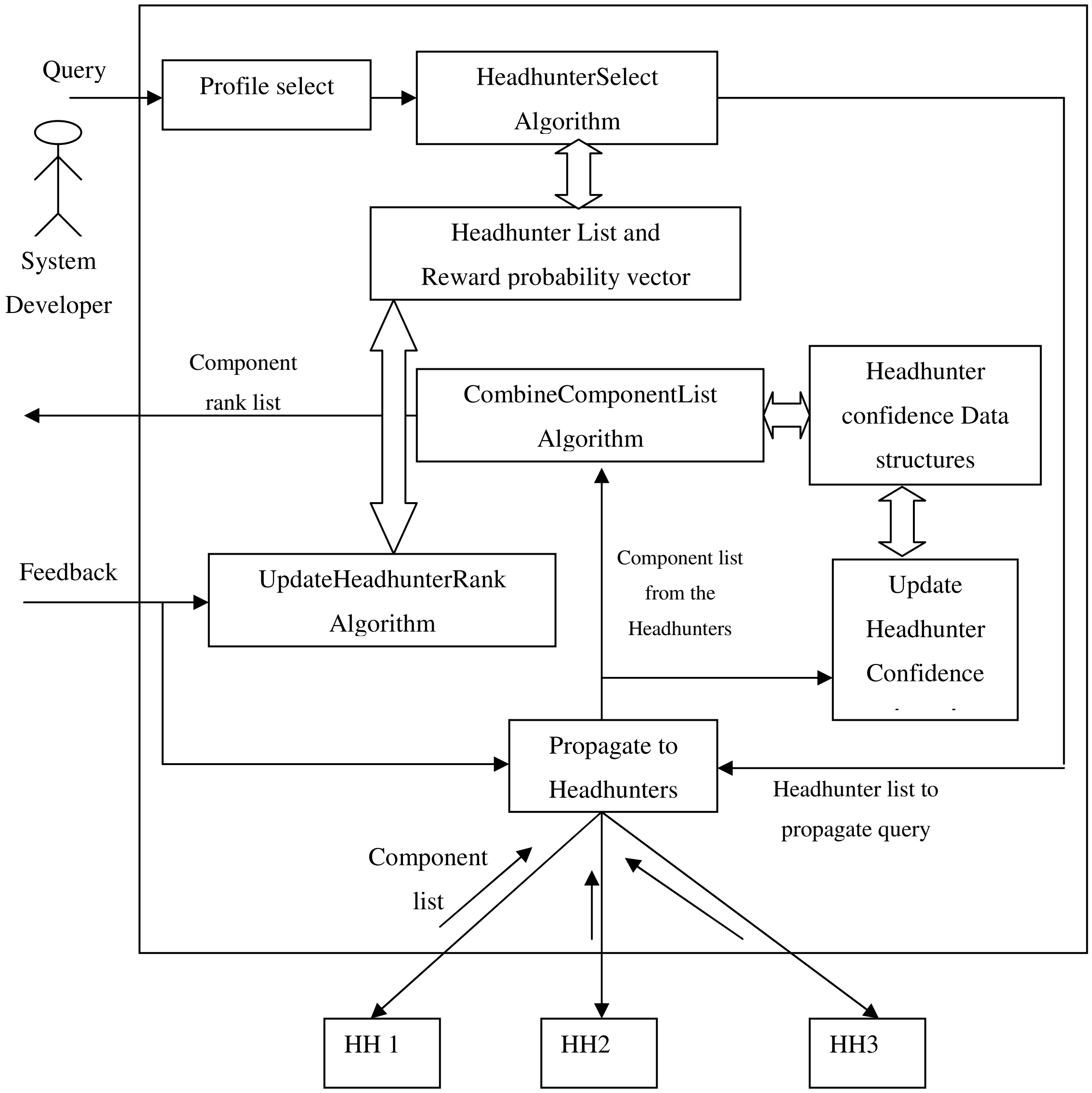

Query
System Developer
Profile select
HeadhunterSelect Algorithm
Headhunter List and Reward probability vector
Component rank list
CombineComponentList Algorithm
Headhunter confidence Data structures
Feedback
UpdateHeadhunterRank Algorithm
Component list from the Headhunters
Update Headhunter Confidence
Propagate to Headhunters
Headhunter list to propagate query
Component list
HH 1
HH2
HH3

Figure 4.1 Flow of activities in the Query Manager

- Each Headhunter (to which the query is propagated) returns to the Query Manager a list of components along with the confidence level of the components. Each headhunter could propagate the query to one or more Headhunters, and combine the components returned from them with the matching components in its meta-repository, before sending its matched component list to the Query Manager.

- The Query Manager combines the component lists from all the Headhunters using the algorithm *CombineComponentList* (discussed below). The difference in the component confidence level algorithm in the Headhunters is adjusted by having separate confidence levels for the Headhunters, which is used by the *CombineComponentList* algorithm.
- The combined component list is ranked and returned to the system developer for the selection of components to form a system.
- Based on the resultant component list, the system developer sends a feedback to the Query Manager. This feedback information contains the reward for the Headhunters which returned the best components for that query. The *UpdateHeadhunterRank* algorithm (discussed below) uses this feedback information to update the Headhunter rank list by updating the reward probability vector.

The following subsections discuss each of the algorithms incorporated in the Query Manager in detail.

4.2.2.1.1. HeadhunterSelect

The *HeadhunterSelect* algorithm selects the Headhunters for the propagation of a query. When a system developer gives a query to the Query Manager for the component discovery, this algorithm is used to select the list of Headhunters to which the query needs to be propagated. This selection of the Headhunters for query propagation is based on the performance of the Headhunters for the previous queries and the feedback they obtained for matching the previous queries. This is decided by having a confidence value for the Headhunters, which are updated by the *UpdateHeadhunterRank* algorithm (discussed in the next section).

The *HeadhunterSelect* algorithm is based on the reinforcement algorithm discussed in [MUK02]. Since this algorithm is based on the past performance, the selection of Headhunters for the initial queries cannot be done based on this algorithm. So, initially the Headhunters are selected randomly. Hence, there is a fair chance for all the

Headhunters to get selected for the initial queries. After the learning process, if the Headhunters for query propagation are selected only based on the past performance, then there is a possibility that any newly joined Headhunter will not be selected for the propagation of the query. To avoid this scenario, one or more Headhunters could be randomly selected, apart from the Headhunters selected through the past performance. So, there is a chance for the newly joined Headhunters to get selected to receive the query for the component matching process.

Steps involved in the algorithm:

- Select zero or more random Headhunters.
- Select the remaining Headhunters that have the highest confidence values associated with them, using the reward probability vector.

Variable representations:

N – Number of Headhunters selected randomly

H_i – Used to denote Headhunter i

H_u – Total number of Headhunters to which the query is propagated

D – Reward probability vector (initialized to 0.5 for all Headhunters)

HeadhunterSelect

(a) Select zero or more Headhunters (say N) in a random fashion.

(b) Select the remaining H_u–N Headhunters from the H_i vector, where the corresponding D vector has maximum values compared to the other D vector values. Since the Headhunter confidence lies between 0 and 1, the Headhunters, having confidence valued closer to 1 are selected. This list excludes the Headhunters selected randomly in step (a).

(c) These H_u Headhunters selected are the Headhunters to which the query will be propagated.

Initially, the D vector values are initialized to 0.5. So, all the Headhunters have a fair chance of getting selected for the query propagation. The number of Headhunters chosen in a random fashion could vary between the Headhunters. One Headhunter could decide to choose the Headhunters only based on the D vector, where another Headhunter could select a few random Headhunters. Also, the number of Headhunters to which the query is propagated varies depending on various factors such as the response time given for the component discovery process, the number of Headhunters having high confidence values, and the Headhunters' capability.

4.2.2.1.2. UpdateHeadhunterRank

Each Headhunter maintains a confidence level of the other Headhunters. This confidence level is used for the selection of a subset of Headhunters for the propagation of a query. The *UpdateHeadhunterRank* algorithm updates the confidence level of the Headhunters. The *UpdateHeadhunterRank* algorithm is based on the update done in the reinforcement learning algorithms discussed in [THA85, OOM90, and MUK02].

After the component discovery process, the system developer selects a set of components to form the desired system. Then a feedback about the component discovery process is sent to the Query Manager and the Headhunters. The feedback is obtained only for the Headhunters to which the query was propagated. This feedback contains a reward value, which depends on many factors such as the number and the quality of components a Headhunter returned for a query, the level of the match of the components specification with the query, and the rank of the returned components. The confidence level of the Headhunters that did not return any component faces a penalty (by decrementing their confidence values), or inaction. Reward and Inaction in learning automata is discussed in [THA00].

The confidence value of other Headhunters is updated based on the reward value obtained for those Headhunters. This algorithm also maintains the number of queries propagated to each of the Headhunters. This is maintained to calculate the average confidence based on the rewards obtained for all the previous queries.

Steps involved in the algorithm:

- Increment the number of queries propagated to the Headhunters.
- Update the confidence values of the Headhunters based on the reward value obtained.

Variable representations:

β_i – Reward value of Headhunter i

D vector – Reward probability vector

S vector – Number of times each Headhunter is selected for query propagation

RF_i – Reward factor, which is proportional to the number of components returned and the level of match of those components with the query

DF – Discounting factor, a factor to assign weighted reward values, i.e., more weight to the recent query rewards compared to the old query rewards

R_{ij} – Rank of a component j returned by Headhunter i

UpdateHeadhunterRank

(a) Update S vector, which represents the number of times each Headhunter has been selected for query propagation. This vector is used to update the reward probability vector D.

$$S_i\,(K+1) = S_i\,(K) + 1,$$

where K – present state

and 'i' – Headhunters to which the query is propagated

Reward value RF_j is calculated for each Headhunter i. This reward value RF_j depends on the number of matching components returned by the Headhunter j, the rank of the components returned by the Headhunter, the level of match of the components with the query, and the number of components from Headhunter j, used by the system developer in building the system.

(b) Update reward probability vector D using the regular average formula or the discounting formula.

$$D_j\ (K+1) = (\ S(K) * D_j\ (K) + \beta_i\ ) / (\ S_j\ (K) + 1\ ) \text{ or}$$

$$D_j\ (K+1) = (\ D_j\ (K) * DF\ ) + (\ \beta_i * (1 - DF)\ )$$

The main vector updated here is the reward probability vector D, whose values are initialized to 0.5. The D vector value varies between 0 and 1, and this range is maintained while updating those values from the feedback. The S vector values maintained, and the reward value obtained for the Headhunters are used to update the reward probability vector. This reward value is a value between 0 and 1, and it depends on the performance of the Headhunters for the given query. The maximum reward value a Headhunter can get for returning the matching components is 1. The Headhunter which returns the best matching component for a query is assigned the maximum reward value. The formula to update this reward value depends on the preference of the Headhunters in classifying the queries. All the queries could be given equal importance, in which case the D vector value for a Headhunter is the average reward value for that Headhunter for all the queries. This formula used the S vector, which has the number of times a Headhunter has been selected for the query propagation. The other option is to give more weight to the recent queries compared to the old queries i.e., applying the concept of discounting, discussed in the previous chapter. There is a discounting factor, which is a number between 0 and 1. The existing D vector value is multiplied by the discounting factor DF. Then the reward is multiplied by a factor of (1-DF) and added to the discounted D vector value to get the new reward probability vector value. In the prototype implemented, the first case is followed, where all the queries are given equal importance.

4.2.2.1.3. CombineComponentList

Each component is associated with the confidence level, which is proportional to the level of the match of a query with that component. The confidence level of the components is assumed to lie between 0 and 1. This assumption is to have a scale for the component confidence so that the confidence levels of a component from different

Headhunters could be compared. The Headhunters will return lists of matching components and their confidence levels to its requestor. So, a Headhunter which propagated a query to one or more other Headhunters will receive different lists of matching components. The *CombineComponentList* algorithm unites these different component lists. This combining of the component lists also depends on the confidence levels of the Headhunters (stored in W vector) that returned the component lists. All the different components are combined to create a single component list to be given to the system developer.

Steps involved in the algorithm:

- The confidence of the components is multiplied by the confidence of the Headhunter which returned that component.
- A single component confidence list is formed from all the component confidences and they are sorted to form a ranked component list.

Variables representations**:**

W vector – Represents the confidences of the Headhunters

CL_k vector – Represents the confidence level of components from Headhunter K

MC – Matrix for confidence level

CombineComponentList

(a) Let the confidence level of the components from Headhunter k be represented as a vector $CL_{k.}$

(b) Multiply CL_{ki} by W_k, where K represents Headhunters to which the query was propagated.

(c) Form a matrix for confidence level (MC), where horizontal-axis represents the components returned from the headhunters and vertical-axis represents the Headhunters to which the query was propagated. If a component X is not returned by a Headhunter Y, then the value in the corresponding position in MC is considered to be 0.

(d) Get the column-wise addition of the matrix MC, which represents the overall confidence level of each component.

(e) The components are sorted by their confidence levels and they are ranked. (With higher confidence level components getting better ranks).

(f) The ranked component list is returned to the system developer.

Two different Headhunters could assign different confidence levels to the same component. The Headhunter confidence level (in the W vector) is used to adjust this difference. So, a better component ranking could be formed by using this Headhunter confidence level.

4.2.2.1.4. UpdateComponentJoin

The *UpdateComponentJoin* algorithm updates the confidence level of the Headhunters. This confidence level is the measure of the Headhunters' capability to match the components with the given query. In case of two or more Headhunters returning different confidence levels for the same component, the average of all the confidence levels is considered as the confidence level. The confidence level of the Headhunter (in the W vector) is updated by a factor to calculate the average confidence level. A factor matrix (FM) is used to calculate this average confidence level.

Steps involved in the algorithm:

- Find the average confidence level for all the similar components.
- The average of those confidence levels is considered as the actual confidence level, and the Headhunter confidence level is updated based on the difference in the actual confidence level and the confidence assigned by the Headhunter.

Variable representation:

(a) S_c – Step size for updating the confidence level

(b) C_{ij} – Component i returned by Headhunter j

(c) ACL_{ci} – Average Confidence Level for component C_i

(d) FM – Factor Matrix

(e) AF – Average Factor

UpdateComponentJoin

(a) For all components C_i, returned by more than one Headhunter, the average confidence level of the component ACL_{ci} is found.

(b) Component confidence level factors F_{ij} are found for each Headhunter - Component pair.

$F_{ij} = ACL_{ci} / C_{ij}$

where i – Components returned by more than one Headhunter

and j – Headhunters that returned component i

(c) A factor matrix FM is formed, where the horizontal axis represents the Headhunters and the vertical axis represents component confidence level factors $F_{i.}$ Headhunters not having component i in their component list will have a value 0 in the matrix.

(d) Average factor AF_j is taken for each Headhunter j (i.e., along the vertical axis).

(e) Each average factor is multiplied by the step size S_c and added to the corresponding elements of the W vector, which has the confidence levels of the Headhunters.

i.e., $W_j (K+1) = W_j (K) + S_c * AF_j$

There is a step size $S_{c,}$ involved in updating this Headhunter confidence level. This step size determines the learning rate for updating the Headhunter confidence level and could be adjusted depending on the performance of the system.

4.2.2.1.5. HeadhunterListChange

Each Headhunter maintains a list of other Headhunters and a probability value for that. This probability value is used to select a random Headhunter for the propagation of the query. The sum of the probability of the Headhunters is 1, i.e., the elements in the action probability vector add to 1. Whenever a new Headhunter joins, the

HeadhunterListChangeAdd algorithm assigns a probability value to this new Headhunter and updates the probability of the existing Headhunters so that the sum of the probability values is maintained as 1. When a Headhunter leaves the URDS, the *HeadhunterListChangeRemove* algorithm increments the probability of the other remaining Headhunters so that the sum of the probability is maintained. The concept of the *HeadhunterListChangeAdd* and the *HeadhunterListChangeRemove* algorithm is from [MUK02].

The D vector (i.e., the reward probability vector) values are maintained between 0 and 1. For each new Headhunter, its D value is initialized to 0.5. But, a D value of one Headhunter will not affect the D values of other Headhunters. So, a Headhunter joining or leaving will not affect the other D values.

Steps involved in the algorithm:

- In the case of adding a Headhunter, the action probability vector values P of the other Headhunters are reduced, and the sum of the reduced quantities is assigned to the new Headhunter.
- In the case of removing a Headhunter, the P vector value of the Headhunter removed is distributed and assigned equally to all the other Headhunters.

Variable representations:

(a) N – Total number of Headhunters after add/remove is done

(b) P vector – Action Probability Vector

(c) D vector – Estimated Reward Vector

HeadhunterListChangeAdd

(a) P vector – Initialize the P vector value of the new Headhunter to 1 / N.

$$P_i\,(K+1) = P_i\,(K) * ((N-1) / N)$$

(b) D vector – The newly added Headhunter's D value is initialized to 0.5. The D values of all other Headhunters are not altered.

HeadhunterListChangeRemove

(a) P vector – distribute the removed Headhunter's P value across all other P values.

P_R – Removed Headhunter node's P value

$P_i(K+1) = P_i(K) / (1-P_i(K))$ where i not equal to R

(b) D vector – No change in D vector values.

(c) Remove the entries for the removing Headhunter in the P vector and the D vector.

4.2.2.2. Headhunter

This section discusses the design of the Headhunters, which incorporates a few profiling algorithms. The following subsection explains the steps followed in the Headhunter for the component discovery process, and the algorithms used for that. The diagrammatic representation of the flow of activities in the Headhunter is shown in Figure 4.2.

The steps followed in the Headhunter for the query processing are as follows:

- The query is obtained from the requestor (Query Manager or a Headhunter) with a preference type.
- A few Headhunters are selected for query propagation, using the algorithm *HeadhunterSelect*. This algorithm is applied on the selected profile information for the preference type to which the query belongs. The query is propagated to the selected Headhunters (represented by HH1, HH2, and HH3 in Figure 4.2). The number of Headhunters selected is based on the time left for the component discovery and the profile information maintained in the Headhunter.
- The Headhunter then checks for matching components in its meta-repository. It uses the *ComponentConfidence* (discussed below) algorithm to assign confidence levels to the matched components.
- Then the *CombineComponentList* algorithm is used to combine the components matched in its meta-repository and the resultant components obtained from the other query-propagated Headhunters.
- The combined component list is returned back to the requestor.

- Once the feedback for that query is obtained, the *UpdateHeadhunterRank* algorithm is used to update the reward probability vector. The feedback information is also propagated to the other Headhunters.

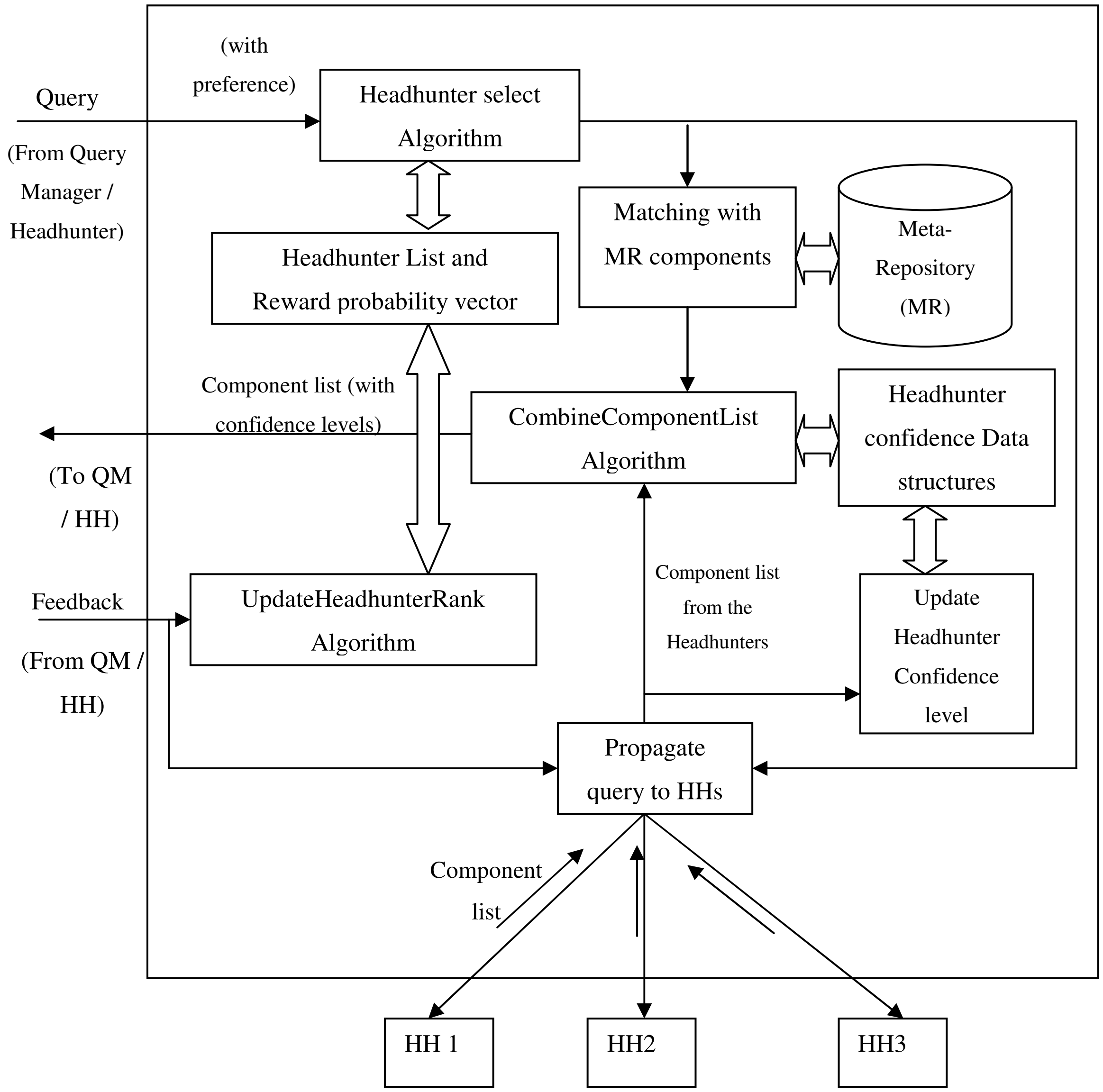

(with preference)
Query
(From Query Manager / Headhunter)
Headhunter select Algorithm
Matching with MR components
Meta-Repository (MR)
Headhunter List and Reward probability vector
Component list (with confidence levels)
(To QM / HH)
CombineComponentList Algorithm
Headhunter confidence Data structures
Component list from the Headhunters
Feedback
(From QM / HH)
UpdateHeadhunterRank Algorithm
Update Headhunter Confidence level
Propagate query to HHs
Component list
HH 1
HH2
HH3

Figure 4.2 Flow of activities in the Headhunter

The algorithms that are incorporated in the Headhunters are discussed in detail in the following subsections.

4.2.2.2.1. HeadhunterSelect

The HeadhunterSelect algorithm selects the Headhunters for the propagation of the query. This selection is based on the cumulative performance of the Headhunters for the previous queries and the feedback they obtained for matching the previous queries. Each Headhunter maintains the profile information about the history of the queries and the feedbacks. This profile information is different for different preference types. The preference types could be QoS, cost, response time or a combination of one or more of these factors in a particular ratio. The preference type is also sent along with the query, which is used to select the appropriate profile for the selection of the Headhunters for the query propagation. The *HeadhunterSelect* algorithm used in the Headhunter is same as the one used in the Query Manager (i.e., described in the previous section).

4.2.2.2.2. UpdateHeadhunterRank

Each Headhunter maintains a confidence level of the other Headhunters. This confidence level is used for the selection of the Headhunters for the propagation of the query. The *UpdateHeadhunterRank* algorithm updates the confidence level of the Headhunters. The *UpdateHeadhutnerRank* algorithm used here is same as the one used in the Query Manager. The feedback to update the Headhunter rank is based on the system developers' selection of components to form the system and is obtained from the entity (Query Manager or Headhunter) that propagated the query.

4.2.2.2.3. CombineComponentList

A Headhunter propagates the query obtained to zero or more other Headhunters, depending on the response time left for the component discovery. The response time left is the difference in the time given for the component discovery process and the elapsed time since the component discovery started. The Headhunters that received a query from this Headhunter will return a list of matching components. The *CombineComponentList* algorithm unites the different components list obtained from different Headhunters and creates a single component list. This combined component list is returned back to the requestor. The concept of the *CombineComponentList* algorithm in a Headhunter is same as the algorithm in the Query Manager (discussed in section 4.2.2.1.3).

4.2.2.2.4. UpdateComponentJoin

The *UpdateComponentJoin* algorithm updates the confidence level of the Headhunters. This confidence level is the measure of the Headhunters' capability to match the components with the given query. This confidence level is used while combining different component confidence lists and needs to be updated after the query propagations and feedbacks. The concept of this algorithm is same as the *UpdateComponentJoin* algorithm in the Query Manager (discussed in section 4.2.2.1.4).

4.2.2.2.5. HeadhunterListChange

Each Headhunter maintains a list of other Headhunters and a probability value for that. This probability value is used to select a random Headhunter for the propagation of the query. The sum of the probability of the Headhunters is 1, i.e., the elements in the action probability vector add to 1. This makes it necessity to update the action probability vector, P, whenever a new Headhunter arrives or an existing Headhunter leaves. The concept of this algorithm is same as the one used in the Query Manager (discussed in section 4.2.2.1.5).

4.2.2.2.6. ComponentConfidence

The *ComponentConfidence* algorithm calculates the confidence level of the components that will reflect the level of match of the component to the query. The confidence level of the component depends on various matches such as syntactic match, semantic match, protocol match, and synchronization match. These matches are discussed in detail in [KUM04] and considered without any further investigation. For the purpose of measuring the component confidence level, the confidence level is assumed to lie between 0 and 1, with 1 being complete match. This assumption is in order to have a single scale of comparison for the component confidence from different Headhunters.

ComponentConfidence

(a) This algorithm takes as input the query, the preference type and other choices of the system developer.

(b) It gives a confidence level of the component, which is a value between 0 and 1.

(c) The factors considered for coming up with the confidence of a component with respect to a given query are security level, QoS, cost, fault tolerance, complexity, design pattern, expected resources, algorithm used and other services offered. For the implementation of the prototype, only the quality level and the cost of the component are considered.

4.2.2.2.7. UpdateARConfidence

The *UpdateARConfidence* algorithm updates the confidence level of the Active Registries based on the feedback obtained from the system developers. The confidence level of the Active Registries depends on the confidence level of the components obtained from the Active Registries for the past queries.

Each component has a component specification, but the component might not behave as specified in the specification, i.e., the functionality of the component might not reflect the properties mentioned in the component specification. The UpdateARConfidence lowers the confidence of the Active Registries hosting such components, so that those components are not updated by the Headhunters.

Each component is associated with a component confidence, which shows the closeness of the behavior of the components as specified in the component specifications. This confidence level of each component is updated from the feedback from the system developer. The confidence level of the Active Registry is calculated as the average confidence of the confidences of the components registered with it. A lower limit is maintained by each Headhunter for the confidence level of the components it will accept. This varies with each of the Headhunters and is approximately 0.5. If the confidence level goes below this lower limit, then those components are discarded from the meta-repository. Also the confidence is reduced for the Active Registries that gave those components to the meta-repositories of the Headhunters. Similarly, a lower limit is maintained for the confidence level of the Active Registries. If the confidence level goes below this lower limit, then the Active Registry is discarded and no more components are accepted from it.

Steps involved in the algorithm:

- The negative feedback value is found, for all the components, from the feedback obtained from the system developer and the negative feedback factor.
- Average negative feedback value is found for each Active Registry from the negative feedback of the components from that registry.
- This average negative feedback is subtracted from the existing feedback of the Active Registry to get the updated confidence value of the Active Registry.

Variable representations:

S_c – Step value for decrementing the confidence level of the components. This will be a value between 0 and 1

NF_i – Negative feedback level on component i. This is a factor to specify the intensity of the negative feedback. This value will be 0 for no negative feedback, 1 for less negative feedback and so on. The maximum value of NF_i depends on the step value S_C, as the total negative feedback value should not exceed 1

NFV_i – Negative feedback value for component i. This is a product of the negative feedback level and the step value

AR_i – Stores the confidence levels of the list of components taken from the Active Registry i

ARC – Stores the confidence levels of the Active Registries

UpdateARConfidence

(a) Let the range of the confidence levels be 0 to 1 and let the confidence level of the components and the Active Registries be initialized to 1, i.e., all the components and the Active Registries have maximum confidence initially.

(b) For each of the negative feedback components

   a. The negative feedback factor NF_i is multiplied by the step value S_C to obtain the negative feedback value NFV_i, i.e.,

   $$NFV_i = NF_i * S_c$$

   b. The negative feedback value NFV_i is subtracted from the existing confidence value of the corresponding component in the AR_i vector, where i represent the Active Registry from which the component was obtained.

(c) ARC vector is updated from the AR_i vectors by the formula:

   $ARC_i = \sum AR_{ij} / N_i$

   where j is the list of all the components belonging to Active Registry i

   and N_i is the total number of components belonging to Active Registry i

(d) ARC now represents the updated confidence levels of the Active Registries.

4.2.2.3. Active Registry

The component developers register their components in the Active Registries. The Headhunters communicate with the Active Registries and update these components into their meta-repositories. The Active Registries, apart from maintaining the component information, will also maintain the information about the component developers. The algorithms incorporated in an Active Registry are discussed below.

4.2.2.3.1. UpdateCDConfidence

The *UpdateCDConfidence* algorithm is used to update the confidence level of the component developers. The confidence of the component developers is based on the combined confidence of the components they develope.

The confidence level of each component is updated from the feedback from the system developer. The confidence level of the component developer is calculated as the average confidence of the components developed by them. A lower limit is maintained for the confidence level of the components. If the confidence level of a component goes below this lower limit, then it is discarded from the Active Registry. Also, the confidence level is reduced for the component developers who registered such components. Similarly, a lower limit is maintained for the confidence level of the component developers. If the confidence level of a component developer goes below this lower limit, then the component developer is discarded and no more components are accepted from it. This *UpdateCDConfidence* algorithm is almost same as the *UpdateARConfidence* algorithm discussed in the previous section, except that the Active Registry confidence values are replaced with the component developer confidence values.

4.2.3. Implementation details of the profiled URDS architecture

The implementation of the profile algorithms is done at the Query Manager and the Headhunter levels. This is because only the algorithms focusing on the improvement of the query propagation techniques are implemented. For both the Query Manager and the Headhunters, their corresponding multi-threaded version (discussed in section 4.1.1) is used as a basis and the profiling algorithms are incorporated in them.

4.2.3.1. Query Manager

The Query Manager maintains the profile information for all the system developers. For each of the system developers, the list of Headhunters and their corresponding reward probability vectors are also maintained. These reward probability vectors are used for selecting the Headhunters for query propagation. The *HeadhunterSelect*, *UpdateHeadhunterRank*, and *CombineComponentList* algorithms are implemented in the Query Manager. Apart from the query queue and the result queue discussed in the multi-threaded Query Manager (discussed in section 4.1.1.1), a separate queue is maintained to store the feedbacks for each of the queries. After the components are matched and returned for a query, a feedback, consisting of reward-penalty values for that component discovery process, is sent to the feedback queue of the Query Manager. A separate feedback thread is used to monitor the feedback queue. On obtaining a feedback from the feedback queue, the feedback thread updates the reward probability vector using the reward and penalty values in the feedback. This thread also propagates the feedback to the Headhunters so that they can update their reward probability vectors. The process of receiving and updating feedback does not affect the query propagation and the component discovery process.

When a Query Manager starts, it spawns the threads for different tasks such as the query propagation, the result collection, and the feedback propagation. These threads monitor their queues for the queries, the results, and the feedbacks respectively. When a query is received from a system developer, the query propagation thread propagates the query to a subset of Headhunters. The results from those Headhunters are monitored in the result queue. Once all the component results for a given query are obtained, or the time allowed for the component discovery has expired, the corresponding components in the Query Manager are combined and sent to the system developer.

The query propagated between different entities of the URDS is a query queue object, which has the details of the query, such as the query bean, the response time allowed for the component discovery, the time when the query was sent by the system developer, and the source location of the query. The query bean carries the attributes (i.e., its UMM specification) of each component to be discovered.

4.2.3.1.1. Steps involved in the query propagation

When a new query (i.e., a query queue object) arrives in the query queue, the following activities happen in the Query Manager:

- The query queue thread retrieves the query queue object. Then it calculates the time the Query Manager needs for processing that query. This includes the time taken for selecting the subset of Headhunters for query propagation, and the time taken for combining the component results from the query-propagated Headhunters.

- The response time allowed for the component discovery process is modified in the query queue object by subtracting the time calculated by the Query Manager for processing the query.
- The source of the query (in the query queue object) is set to the location of the Query Manager.
- The query queue thread selects a subset of Headhunters for propagating the query, using the *HeadhunterSelect* algorithm. The modified query queue object is propagated to the selected Headhunters (i.e., sent to the query queues of those Headhunters), and the number of Headhunters to which the query is propagated, and their locations are noted.
- The result queue thread checks for the results from all the Headhunters to whom the query was propagated. Once all the results are obtained, they are combined using the *CombineComponentList* algorithm, and this combined list is sent to the system developer. This result thread also keeps track of the time allowed for the component discovery process. If the allowed time is about to expire, then the results available at that time are combined and sent to the system developer without waiting for all the Headhunter results. The resultant components are sent along with their confidence levels obtained from the Headhunters.

4.2.3.1.2. Steps involved in the feedback propagation

The feedback, which contains the rewards for the Headhunters, is propagated in the form of a feedback object. When a new feedback (i.e., a feedback object) arrives at the feedback queue, the following activities happen in the Query Manager:

- The feedback thread updates the profile list of the Headhunters maintained by the Query Manager.
- This thread passes the feedback information to the Headhunters which returned those components. Not all feedbacks are sent to all the Headhunters. Only the feedback of the components returned by a Headhunter is sent to that Headhunter.

4.2.3.2. Headhunter

The Headhunter maintains the profile information for all the preference types (i.e., cost, quality) of the queries. For each of the preference types, the list of Headhunters and their corresponding reward probability vectors are also maintained. These reward probability vectors are used for selecting the Headhunters for query propagation. The *HeadhunterSelect*, *UpdateHeadhunterRank*, *CombineComponentList*, and the *ComponentConfidence* algorithms are implemented in the Query Manager. Apart from the query queue and the result queue, a separate queue is maintained to store the feedbacks for each of the queries, as in the case of the Query Manager.

After the components are matched and returned for a query, the feedback for that discovery process is sent to the feedback queue of the Headhunter in the form of a feedback object. A separate feedback thread is used to monitor the feedback queue. On obtaining a feedback object from the feedback queue, the feedback thread updates the reward probability vector using the reward values in the feedback object. It further propagates the feedback information to other Headhunters from which components were obtained for that query, so that they can update their reward probability vectors. This propagation of feedback is similar to the query propagation process and it does not affect the component discovery process.

When a Headhunter starts, it spawns threads for different tasks such as the query propagation, result collection, feedback propagation, and the components update in the meta-repository. The first three threads monitor their queues for the queries, the results, and the feedbacks respectively. The meta-repository update thread communicates with the Active Registries to update its component information in the meta-repository. When a query is received from a requestor (from either a Query Manager or a Headhunter), the query propagation thread propagates the query to a selected subset of Headhunters. The results from those Headhunters are monitored in the result queue. Once all the component results are obtained, or the time allowed for the component discovery has expired, the available components in the result queue are combined and sent to the requestor.

4.2.3.2.1. Steps involved in the query propagation

When a new query (i.e., a query queue object) arrives in the query queue, the following activities happen in the Headhunter. These tasks are similar to the tasks done in the Query Manager:

- The query queue thread retrieves the query queue object. Then it calculates the time the Headhunter needs for processing that query. This includes the time taken for selecting the subset of other Headhunters for query propagation, and the time taken for combining the component results from the query-propagated Headhunters.
- The query queue object is modified with the new response time, which is subtraction of the calculated time by the Headhunter for processing the query, from the given time by the requestor.
- The source of the query (in the query queue object) is set to the location of this Headhunter.
- The query queue thread selects a subset of Headhunters for propagating the query, using the *HeadhunterSelect* algorithm. The modified query queue object is propagated to the selected Headhunters (i.e., sent to the query queues of those Headhunters), and the number of Headhunters to which the query is propagated and the location of those Headhunters are noted.
- Then the query queue thread accesses the meta-repository for finding the matching components for the given query. If any matching components are found, the confidence levels are found using the *ComponentConfidence* algorithm. The confidence value is based on the level of match of the component with the query. The matched components and their confidence levels are added in the result queue.
- The result queue thread checks for the results from all the Headhunters to whom the query was propagated. Once all the results are obtained, they are combined using the *CombineComponentList* algorithm, and this combined list is sent to the requestor. This result thread also keeps track of the time allowed for the component discovery process. If

the allowed time has expired, then the results available at that time are combined and sent to the system developer. The resultant components are sent to the requestor, along with their confidence levels.

4.2.3.2.2. Steps involved in the feedback propagation

The feedback, which contains the rewards for the Headhunters, is propagated in the form of a feedback object. Each feedback is associated for a particular query. When this feedback enters the feedback queue of a Headhunter, the following activities happen in the Headhunter:

- The feedback thread updates the reward probability vector of the Headhunter list maintained.
- This thread passes the feedback information to other Headhunters that returned matching components for that query. Not all feedbacks are sent to all the Headhunters. Only the feedback of the components returned by a Headhunter is sent to that Headhunter.

The class diagrams of the implementation of the Query Manager and the Headhunters are provided in the appendices. This chapter discussed the design and implementation details involved in improving the performance of the existing URDS architecture. It also discussed the addition of the profile information at different levels of the URDS architecture. The next chapter analyses and compares the experimental results of each of this enhanced URDS version.

CHAPTER 5. EXPERIMENTAL ANALYSES OF THE ENHANCED URDS

Chapter 4 provided the design and implementation details about the enhancements that are added to the URDS architecture. It also described the algorithms that are incorporated in each of the nodes that form the URDS architecture. This chapter focuses on the experimental analyses and verification, based on the changes in the architecture. It also discusses the comparisons in performance and quality between the different enhancements implemented. The section 5.1 provides the experimental setup used for all the experiments conducted. The section 5.2 discusses the experiments based on the architectural changes focusing performance of the URDS and the section 5.3 discusses the experiments with the profiled URDS version. Finally, section 5.4 provides the experiments conducted in the handheld devices.

5.1. Experimental setup

The prototype is implemented using the Java 2 Platform, Standard Edition (J2SE) version 1.4 software environment. The core entities (i.e., the Domain Security Manager, Query Manager, Headhunters and the Active Registries) are implemented using Java-RMI technology. The underlying principles of these entities are adapted from [NAN02]. The repository used by the Domain Security Manager for authenticating other entities is database-oriented implementation using Oracle v9.0. The meta-repositories are part of Headhunters, and are implemented as hashtables in Java. The hardware platform used for the experimentation purpose is the Sun Solaris Ultra-250 Sparc machines, hosting Sun OS release 5.8.

5.2. Experiments based on performance improvement

The existing URDS architecture is compared experimentally against different versions of the enhanced URDS architecture. The first version incorporates only the multi-threaded Query Manager, with single-threaded Headhunters. The second version incorporates the multi-threaded Query Manager and multi-threaded Headhunters.

These experiments are used to analyze and compare the performances of different versions of the URDS by measuring the time taken for the component discovery process. The time taken for the discovery of components in the URDS architecture depends on the following factors:

- The number of Headhunters to which the queries are propagated. This is because each of the Headhunters has to perform the matching process of the queries against the components in their respective meta-repositories.
- The amount of query processing overlap among different Headhunters in the component matching process, i.e., if the matching process happens simultaneously across different Headhunters, then the time taken for the component discovery process is reduced.
- The load on the Query Manager and the Headhunters, i.e., the number of queries that the Query Manager and the Headhunters are handling simultaneously.

The load on the URDS is defined as the number of queries that are being processed by the URDS. The rate at which the queries arrive at the Query Manager can be classified in to three categories.

Case 1: The query arrival rate is slower than the rate at which the results are processed in the URDS and returned back to the system developer. In this case, there will be atmost one query being processed in the URDS at any point in time. So, the response time for a query will not keep increasing as the load in the URDS is at most one.

Case 2: The query arrival rate is almost the same rate as the results are processed and returned back to the system developer. In this case, the load of the URDS, i.e., the number of queries that are being processed in the URDS will be almost the same. So, there will not be much variation in the average response time for a large number of queries, for a given set of Headhunters.

Case 3: The query arrival rate is faster than the rate at which the results are processed and returned back to the system developer. In this case, the load of the URDS will keep increasing, and this increase will be proportional to the query arrival rate. Because the time taken for processing a query also depends on the load in the URDS, the average response time for a large number of queries will keep increasing.

The first two cases will not affect the average time taken for the component discovery process for a series of queries. Case 3 will have an impact on the average response time depending on the query arrival rate, as the load in the URDS keeps increasing. Figure 5.1 compares the difference in the average response time for different versions of the URDS architecture for a particular query arrival rate.

In Figure 5.1, the X-axis represents the queries that are given to the Query Manager and the Y-axis represents the average response time (in milliseconds) of a series of queries. The distribution of the components across the Headhunters is decided on a random basis. But this distribution is maintained constant for all the URDS versions compared. The other parameters that are maintained constant across the different versions of URDS compared includes, the number of Headhunters to which the queries are propagated, and the rate at which the queries are submitted to the Query Manager. Also, there is no restriction imposed on the size of the query queue.

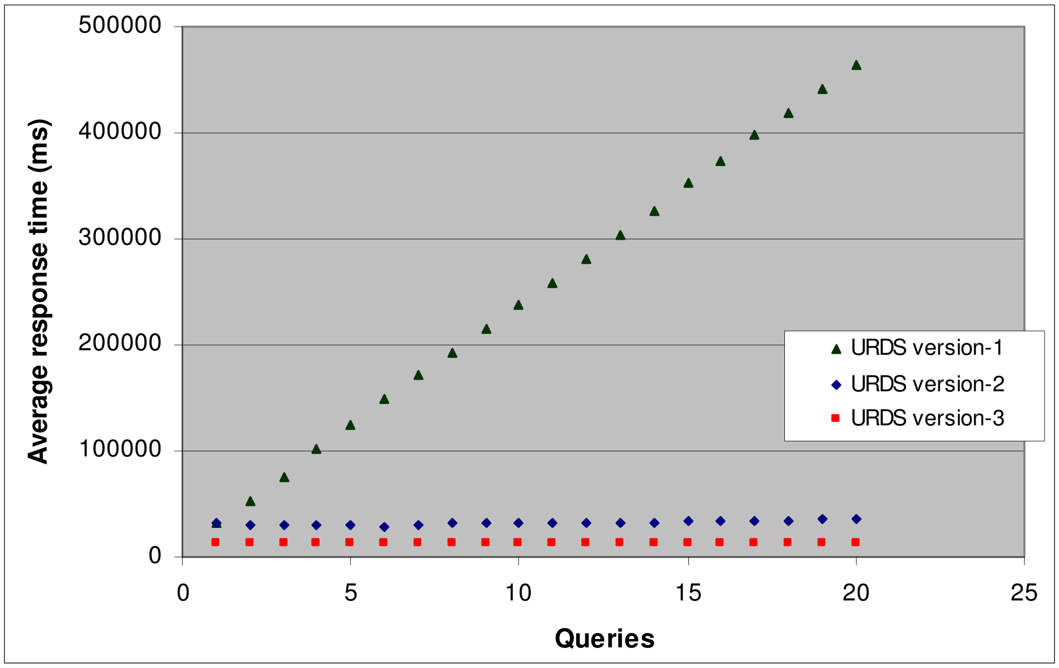


Figure 5.1 Average response time taken in different URDS versions

Legend:

Version 1: Single-threaded Query Manager and the single threaded Headhunters

Version 2: Multi-threaded Query Manager and the single-threaded Headhunters

Version 3: Multi-threaded Query Manager and the multi-threaded Headhunters

The number of Headhunters used in this experiment is 20, and the inter-arrival time of the queries is 2000 milliseconds between each of the queries. For this scenario, the average response time taken for a series of 20 queries is measured. It is observed from Figure 5.1 that the URDS version-2 and version-3 outperform the single threaded URDS version in terms of the average component discovery time. This is due to the multi-threading added in one or more entities of the URDS architecture. Because 20 queries is small, this graph is just to show that there is a considerable difference in the average response time between the URDS version-1 and the other two versions, even for a small number of queries. If the number of queries is increased the difference in the average response time will keep increasing. The next graph, i.e., Figure 5.2, shows the difference between the

URDS version-2 (upper curve) and the version-3 (lower curve) for a larger number of queries.

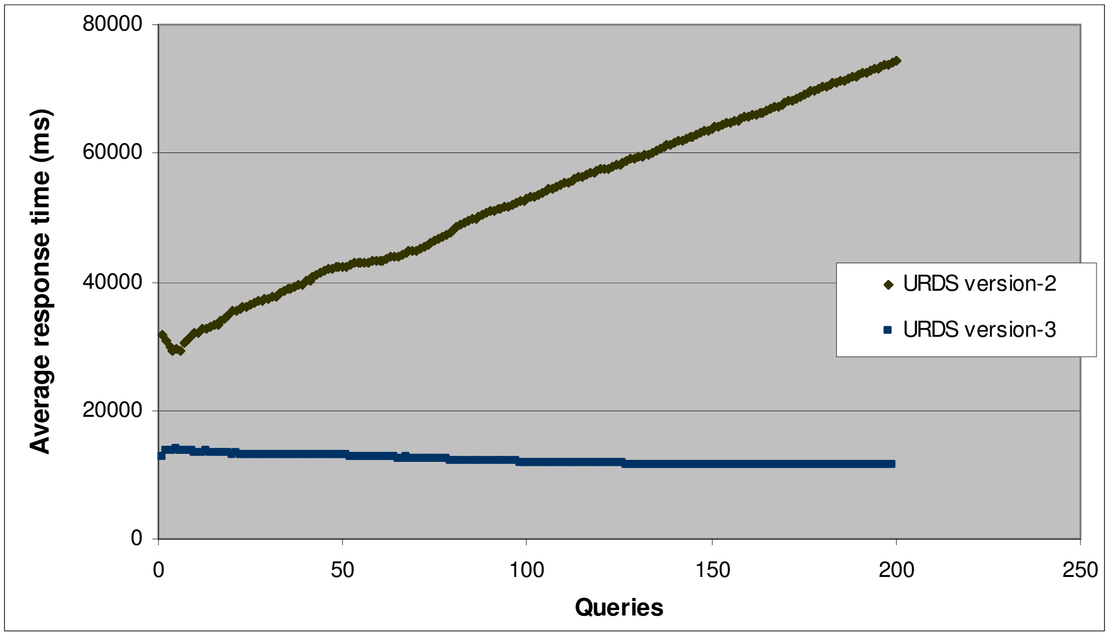


Figure 5.2 Average response time taken in the multi-threaded URDS versions

This comparison is made with 20 Headhunters, for a series of 200 queries. Figure 5.2 shows that the performance of the URDS version-3 improves over that of the URDS version-2 as the load of the URDS increases. This is because; in the URDS version-3, the Headhunters are also multi-threaded, as opposed to the URDS version-2. It can be noted from the graph that the average response time of the URDS version-3 appears to slightly decrease with the increase in the number of queries. The reason for that is as follows. The queries are given to the Query Manager in the specified inter-arrival time, without waiting for the component results for the previously sent queries. The response time taken for the initial queries are more, due to the initialization and thread creation time involved in it. The average response time for a query represents the average time for all the queries processed until this query. Since the initial queries have contributed more component discovery time, and the later queries take relatively less time for processing, the average time for a particular number of queries slightly decreases until it saturates.

Although the Figures 5.1 and 5.2 depict the behavior of the component discovery process for the different versions of the improved URDS, they do not show the difference in average response time of the queries when the load on the URDS changes. Figure 5.3 quantifies this change in the average response time for the change in the load of the URDS. The URDS version with the single threaded Query Manager and the single threaded Headhunters is not compared here, as this version does not support simultaneous processing of the queries. For the versions compared, the number of Headhunters to which the queries are propagated, and the components registered in their meta-repositories are maintained constant.

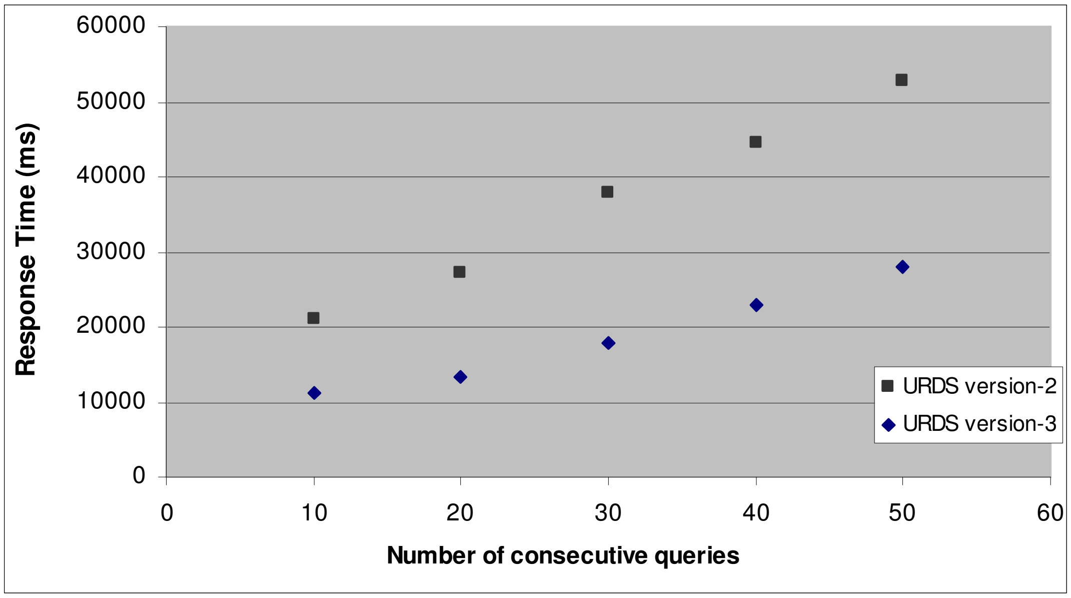


Figure 5.3 Comparison of the average response time for different loads in the URDS

Legend:

Version-2: Multi-threaded Query Manager and the single-threaded Headhunters

Version 3: Multi-threaded Query Manager and multi-threaded Headhunters

In Figure 5.3, the X-axis represents the number of consecutive queries given to the URDS architecture. These queries are given to the Query Manager one after other in a sequence, i.e., with a very small inter-arrival time. These queries are stored in the query queue of the Query Manager and processed in the order of arrival of the queries. There is no restriction imposed on the size of the query queue. The Y-axis represents the average response time (in milliseconds) taken for processing the given queries. For both the versions of the URDS compared, the average response time increases as the load in the URDS increases. It can also be noted that the URDS version with the multi-threaded Query Manager and the multi-threaded Headhunters performs better than the other version. This is because of the increase in the overlap of the component matching in the Headhunters due to the multi-threaded Headhunters and the asynchronous query propagation among the Headhunters.

The above graphs indicate that the multi-threaded version of the URDS improves the time taken for the discovery of the components, thus improving the performance of the component discovery process. The advantages of this enhancement are summarized below.

- The time taken for the component discovery process is significantly reduced. This is achieved by reducing the query wait time in the query queue, overlapping the component matching for different queries in a Headhunter by having multiple threads, and overlapping the component discovery among different entities in the URDS by following asynchronous communication for the propagation of the query.
- The propagation of the queries is implemented by using query queues. So, based on the capability of the Headhunters in processing multiple queries simultaneously, the size of the query queue is varied.

5.3. Experiments based on profiling

This section discusses the experiments carried out after the incorporation of a few of the profiling algorithms at different levels in the URDS architecture. The enhancements made here focus on the improvement of the query propagation techniques in the URDS architecture so as to reduce the time taken for the component discovery process without compromising the quality of the components discovered. This is achieved by choosing a subset of the Headhunters for the propagation of the query, as opposed to an exhaustive search, as the time taken for the component discovery is based on the number of Headhunters to which the query is propagated. This subset of Headhunters should be chosen such that the probability of the selected Headhunters in returning the closely matching components for the query is high. This is achieved by storing the history of the component discovery process about different Headhunters.

The algorithms implemented in the prototype include *HeadhunterSelect*, *UpdateHeadhunterRank*, and *CombineComponentList* (discussed in the previous chapter). These algorithms are chosen, as they improve the query propagation techniques. The *ComponentConfidence* algorithm is used by the Headhunters to assign confidence levels to the components, based on the level of match with that of the query. For the prototype implementation, the component type, the quality level, and the cost of the components are used as the matching factors for the components. Each component is associated with a component type, and in the prototype, 46 component types (chosen randomly) are used. The number of components in the meta-repository of each Headhunter is chosen randomly between 1 and 100, to simulate the real world scenario where the Headhunters have different number of components. Number of quality levels used in the prototype is 10, where each component belongs to one of the quality levels. This is used to differentiate the confidence of a component for the particular query. Each component in the Headhunters belongs to one of the 46 component types, and one of the 10 quality levels. Apart from these two, the components have a specific costs associated with them, which are also chosen randomly. All these factors are chosen on a random basis, so that different combinations of these factors form the components in the meta-repositories. A particular distribution of the components is created in the meta-repositories. For all the experiments conducted and the comparisons made, this component distribution is maintained constant for all the queries. The queries are generated in a random fashion, with each query belonging to one of the 46 component types, and one of the 10 quality levels. Once a random sequence of queries is generated, the queries and the order in which they are sent to the Query Manager are also maintained constant for the different versions of the URDS compared.

Based on the level of match, the components are assigned a value between 0 and 1 by the Headhunters. Most of the experiments measure the quality of the discovered components against the time taken for the component discovery process. The quality here is measured by a combined factor of the number of the components matched by the Headhunters and the confidence values of those components. The confidence values of all the components matched are added up to obtain the quality, because the quality should increase with the number of components matched. Also, a lot of components with smaller confidence values (i.e., close to 0) should not be added up and considered better when compared to one matching component having confidence close to 1. To avoid this scenario, the confidence values of the components are compared against the component with the highest confidence value and the difference is found. If a Headhunter has more than one matching component, then the cumulative confidence value of those is reduced by a factor. This factor is calculated as follows. The difference of each component confidence value from the highest confidence value is multiplied by 0.5. These differences are added up, which gives the subtracting factor. By doing this, the subtracting factor is more if the difference of a confidence value from the highest confidence value is more. So, if there is a greater number of such components (i.e., with less confidence values), then this subtraction reduces the cumulative confidence, thus avoiding giving higher confidence to Headhunters that matched a lot of components with very low confidence values.

5.3.1. Experiments comparing the existing and the profiled URDS architecture

The first set of experiments compares the profiled and the non-profiled URDS versions. Even in the profiled URDS version, differentiation is made with one version implementing the profiling algorithms (i.e., the *HeadhunterSelect*, the *UpdateHeadhunterRank*, and the *CombineComponentList*) only in the Query Manager, and the other version implementing the profiling algorithms in both the Query Manager and the Headhunters.

Figure 5.4 shows the quality of the components discovered with the increase in the time taken for the component discovery process. The quality here is decided by the combined factor (discussed above) of the number of components that match the query and the closeness of the match of those components with the query. In this graph, the X-axis represents the increase in time (in milliseconds) during the component discovery process and the Y-axis represents the quality of the components discovered. The curves for all the three versions compared meet at the same point (not shown in the figure) in case of not restricting the component discovery based on time, which is the case for exhaustive search.

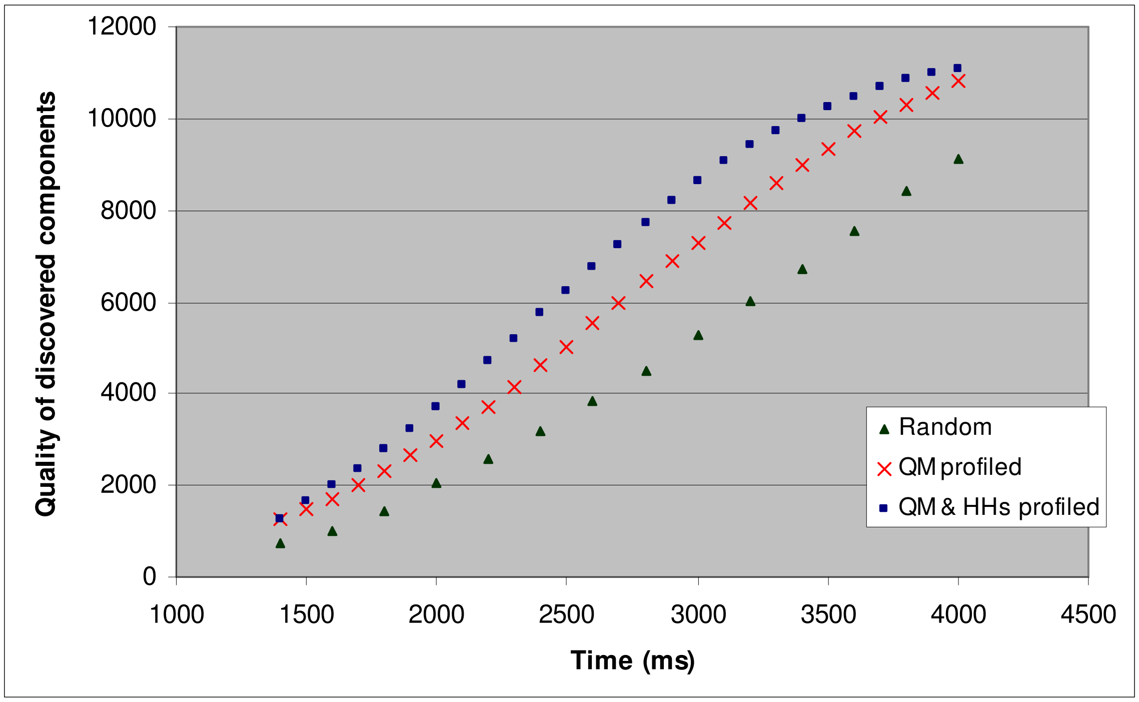


Figure 5.4 Quality of the discovered components

Legend:

Random – This is the multi-threaded URDS version, with no profiling implemented in it. The query propagation used in this version is random

QM profiled – This is the multi-threaded URDS version, with the profiling algorithms implemented in the Query Manager. The query propagation techniques followed by the Query Manager are based on the *HeadhunterSelect* algorithm (discussed in Chapter 4)
QM & HHs profiled – This is the multi-threaded URDS version, with the profiling algorithms implemented in the Query Manager and also in all the Headhunters. The query propagation techniques followed by the Query Manager and the Headhunters are based on the *HeadhunterSelect* algorithm

This experiment is conducted for a series of 400 queries selected on a random basis. For all the cases compared, the number of Headhunters, the distribution of components in each Headhunter, and the number and the order of the queries are maintained as a constant. Each Headhunter that receives the query further propagates it to one, two or three other Headhunters, which is decided on a random basis. Also, all the Headhunters receive the queries only once, i.e., only the Headhunters that did not receive a query are selected for further propagation of the query.

The URDS version without any profiling uses a random strategy for the selection of Headhunters during the query propagation. In all the URDS versions, the quality of the discovered components increases linearly with the increase in time. For the URDS version where profiling in implemented at the Query Manager level, the quality of the components discovered increases at a faster rate than the URDS version with random query propagation. This means that the components with a good quality are discovered more quickly in this version as compared to the one without any profiling. When profiling is added to all the levels of the URDS, the rate of increase in the quality of the discovered components is more compared to the other two versions of the URDS. This shows that, for a given time interval, the components discovered by the profiled URDS are of better quality than the ones discovered by the non-profiled version of the URDS. Thus, the profiled version can provide a better outcome in the search process when a constraint is placed on the time available for the discovery of components.

5.3.2. Experiments comparing the different levels of profiling

The next experiment focuses on implementing query propagation algorithms at different levels of the Headhunters. When a Headhunter receives a query from the Query Manager, it propagates the query to one or more other Headhunters. This propagation of query forms a tree (because of the requirement that a Headhunter not receive a query more than once) among the Headhunters, with Query Manager being the root of the tree. The Headhunters at all the levels implement the query propagation algorithms (i.e., *HeadhunterSelect and UpdateHeadhunterRank*). The implementation of profiling in each of the levels has an overhead associated with it. The Headhunters at every level have to access the profile information and decide on a subset of Headhunters for further query propagation. Care should be taken so that, the overhead due to maintaining the profile information and the associated Headhunter subset selection should not dominate the total time taken for the component discovery process, i.e., the time saved in the component discovery process due to the implementation of profiling, should not be lost to the overhead involved in accessing the profile information.

In the next sets of experiments the time taken for the component discovery process is compared by implementing the query propagation algorithms at each level of the Headhunters. This shows the overhead involved in using the profiled query propagation. This information is necessary to identify the levels of Headhunters until which the profiling needs to be incorporated, so as to achieve the best matching components in the least possible time.

Figure 5.5 shows a comparison in the time taken for the component discovery process for a set of queries by adding query propagation algorithms at different Headhunter levels. This experiment is done to analyze the number of levels in which the query propagation algorithms need to be incorporated to get a better outcome of results. In Figure 5.5, the X-axis represents the increase in time (in milliseconds) during the component discovery process. The Y-axis represents the quality of the discovered components.

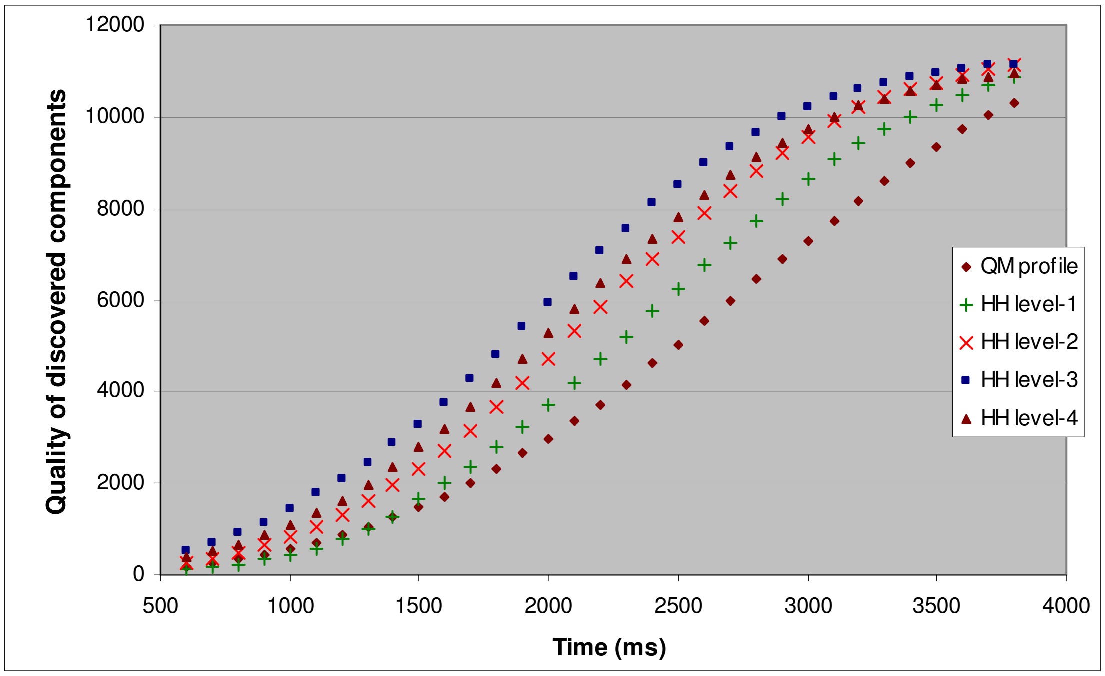


Figure 5.5 Quality of the discovered components for different levels of profiling

Legend:

QM profile – The query propagation algorithms based on profiling are implemented in the Query Manager, but all the Headhunters uses random query propagation

HH level-x – The query propagation algorithms are implemented in the Query Manager and x-level of Headhunters are profiled. The Headhunters from the x+1 level use random query propagation

This experiment uses 40 Headhunters, with each Headhunter having a different number and quality of the components. But, for all the levels of profiling compared, the number of Headhunters, the component distribution in the Headhunters and the order of the queries given to the Query Manager are maintained constant. The quality of the components discovered is found for 400 randomly generated queries. When an experiment is done by implementing the profiling algorithms until a particular Headhunter level, then all the remaining levels follow the random query propagation.

Also, each Headhunter propagates the queries to two other Headhunters, thus forming a binary tree.

The above graph shows that the rate of increase in the quality of the discovered components with the increase in search time. Although, the implementation of profiling algorithms helps in reducing the time taken for the component discovery and improves the quality of the components discovered, implementing profiling algorithms until a certain level seems to perform better than implementing profiling in all the Headhunters. The number of levels formed by the 40 Headhunters is 5, as each Headhunter propagates the query to two other Headhunters. In Figure 5.5, as the number of levels in which the profiling algorithms are implemented is increased, the quality of the component discovery increases, until the profiling in the third level of Headhunters. For the case of implementing profiling until the forth level of Headhunters, the rate of increase in the quality of component discovery reduces compared to the experiment with three levels of profiling. This could be due to the overhead involved in the Headhunter selection based on the profiled information. After the third level of profiling, the overhead due to the implementation of the profiling algorithms could dominate the quality advantage obtained due to the profiling. The results of these experiments depends on the number of levels of Headhunters formed by the queries, i.e., the topology formed by the Headhunters, which in turn depends on the number of Headhunters to which each entity propagates the query. This change after the third level of profiling is for a Headhunter count of 40 and for the binary tree topology. To identify the reason for this change in pattern, and to identify the number of profiling levels, the experiments are carried out for a different number of Headhunters, and with different topologies.

The above experimentation was conducted again for different numbers of Headhunters, i.e., 10, 20, and 40 Headhunters. For all the different numbers of Headhunters, the experiment is conducted with two fixed topologies. The topology for the first experiment is fixed by allowing each Headhunter to propagate the query to exactly two other Headhunters (topology-2 in Table 5.1), and in the second to three other Headhunters (topology-3 in Table 5.1). Table 5.1 shows the number of levels formed by the given Headhunter count and the topology. Table 5.2 shows the number of levels of Headhunters that need to use the profile information for query propagation for the given count of Headhunter and the given topology formed by the queries.

Table 5.1 Number of levels formed by the Headhunters

Headhunter Count	**Topology-2**	**Topology-3**
10	3	2
20	4	3
40	5	4

Table 5.2 Number of levels in which profiling needs to be implemented

Headhunter Count	**Topology-2**	**Topology-3**
10	HH level-1	HH level-0
20	HH level-2	HH level-1
40	HH level-3	HH level-2

Column Headers:

Topology-2 – Each Headhunter propagates the query to two other Headhunters

Topology-3 – Each Headhunter propagates the query to three other Headhunters

For the URDS architecture with 20 Headhunters, and the topology-2, the implementation of profiling algorithms in the second level of Headhunters gives better results compared to the implementation of query propagation in all the levels of the Headhunters. For 40 Headhunters, and the topology-2, profiling is needed until the third level of Headhunters. Based on these results, there appears to be a pattern in the number of levels in which profiling needs to be implemented. Let the number of levels formed by the Headhunters be t (i.e., the height of the tree), for a particular topology. In any propagation tree formed, the leaf nodes will not use the profile information, as they will not pass the query further to other Headhunters. Thus, the maximum level up to which the profiling algorithm can be implemented is t-1. Table 5.2 shows that implementing profiling up to t-2 levels gives a better performance for the configurations experimented with. This is because at t-1 level, there is only one level of Headhunters remaining (i.e., the leaf level Headhunters) and the gain that would be obtained by a selective approach is not substantially more than the one obtained by the random approach. Also, it should be noted that as a result of the profiling process, the Headhunters that have better components (for a given set of queries) will move upwards in the propagation tree. Thus, there is not much of a quality difference among the ones that are available for the selection process at the leaf node levels and there is an added overhead of maintaining the profile information at each Headhunter. Hence, a selective search at that level does not provide a substantial improvement over the random approach.

The comparison between these two cases, one where Headhunters at the t-1 level use the profile information and the other where t-1 level Headhunters uses random selection of Headhunters are analyzed here. The leaf level nodes contribute to a larger percentage of the total Headhunters (almost 50% of Headhunters when the topology is a binary tree) in the URDS. The time taken by each entity to process a query is different, which means that the query will not be obtained by all the leaf level Headhunters at the same time. So, when a query has a restricted component discovery time, there is a possibility that only a few of the leaf nodes obtain the query. In this scenario, if these leaf level Headhunters are selected by the profile information, then the probability of getting good component results is high. Also, the advantage obtained due to saving the overhead time in the last but one level of Headhunters is small. In general, the selection of Headhunters for query propagation using the profile information in all the levels of Headhunters improves the quality of the component discovery process.

5.3.3. Experiments with different component distribution

The next experiment is conducted to check the effect of the component distribution in the Headhunters on the quality of the discovered components using the profile information.

The previous experiments comparing the quality of the discovered components with different levels of profiling are repeated with a different distribution of components in 40 Headhunters. As discussed earlier, the number of components in the Headhunters in the earlier component distribution can vary between 1 and 100. The number of components selected randomly in the Headhunters varies from 4 to 98. Since the difference is high, it is easy to make a differentiation between such Headhunters. In the new component distribution, the difference between the maximum and minimum number of components that the Headhunters can have is reduced. Here the number of components in the meta-repository of the Headhunters varies from 40 to 70, chosen on a random basis. By minimizing this difference, the number of components in each of the 40 Headhunters is relatively close, thus making it difficult to rank the Headhunters. With this new component distribution, comparisons are made between the non-profiled URDS version and the different levels of profiling in the URDS. Although, the component distribution is different, a pattern (i.e., the rate of the quality increase against time) is noticed similar to in the case of the previous, broader component distribution for the given order of queries.

5.3.4. Experiments with different order of queries

The next experiment focuses on the order in which the queries are given to the Query Manager. The Query Manager learns and updates the profile information by obtaining feedback for each of the propagated query. Initially, all the Headhunters are considered equal, and the Headhunters are selected on a random basis. In Figure 5.6, the X-axis represents the time (in milliseconds) and the Y-axis represents the quality of the discovered components.

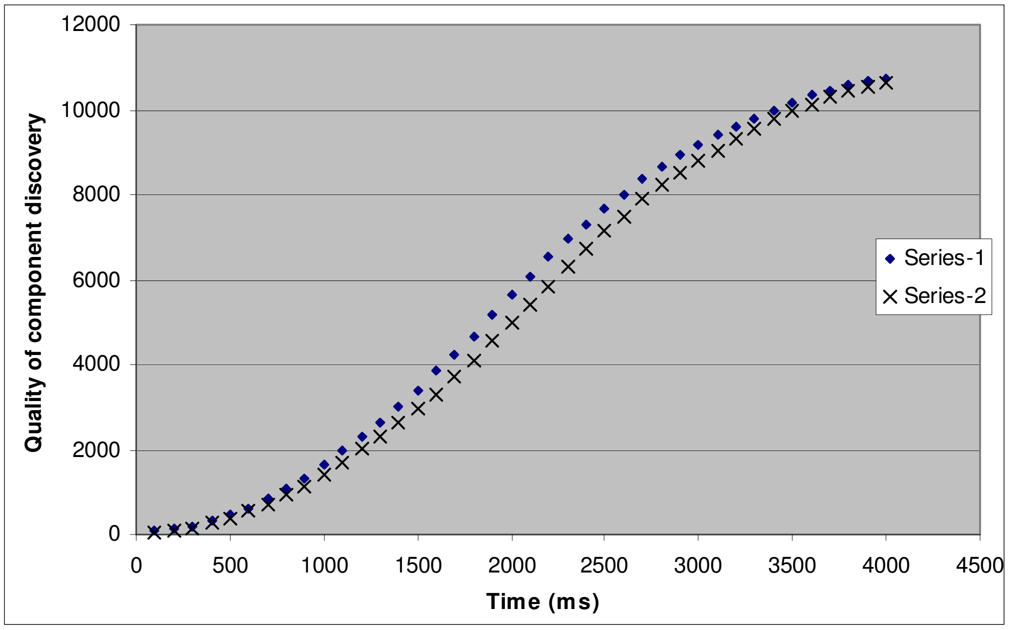


Figure 5.6 Comparison of quality of discovered components for different order of queries

This graph compares two series of queries. Here the number of Headhunters, the distribution of components in the Headhunters, the topology formed by the queries, and the number of queries are same for both the cases compared. Two series of queries (series-1 and series-2) are formed, from a set of 400 queries, by selecting the queries in a random fashion. This graph shows that the order in which the queries are given to the Query Manager also plays a role in the learning process of the entities, thus affecting the discovery of the good quality components. At the end of 400 queries, the Headhunter ranking list formed by the Query Manager will be same for both the cases. But, the process of learning and the selection of the Headhunters for each query depends on the order in which the queries are given to the Query Manager.

5.3.5. Experiment for non-disjoint Headhunter

In all the experiments conducted, the Headhunters are selected in a disjoint fashion. This means that if a Headhunter receives a query from a requestor, it will not receive the same query from a different requestor. This forms a tree among the Headhunters, during the propagation of the queries. The next experiment is conducted with this constraint ignored, i.e., the Headhunters can receive the same query from two different requestors. This forms an arbitrary graph among the Headhunters, unlike a tree in the earlier case. Here the Headhunters will process the query only once, but will send the results to all the requestors from which the query was obtained. The query is not further propagated if a closed loop is formed by the Headhunters due to the query propagation. Feedback is also sent to all the Headhunters to which query is propagated.

In Figure 5.7, X-axis represents the time (in milliseconds) and the Y-axis represents the quality of the discovered components. This graph compares the random query propagation scenario with the profiled query propagation scenario. Although, the quality increases quickly in the profiled URDS until a certain time, the quality saturates after some time. This is because, due to the disjoint constraint being removed, the query is propagated to a few good Headhunters in a closed loop. All the other Headhunters do not receive the query, even if the component discovery process is not restricted by a time constraint.

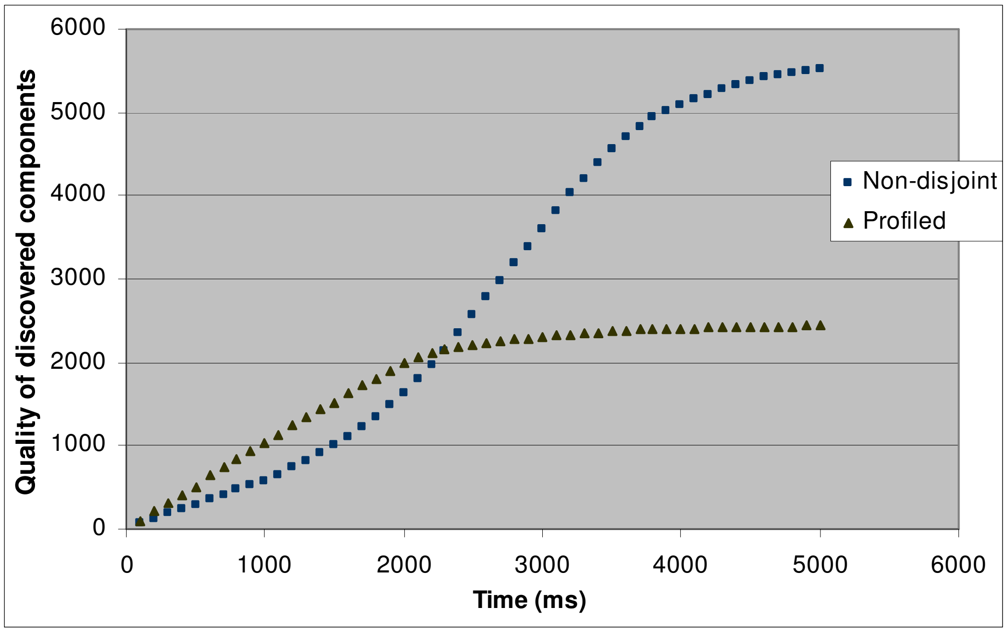


Figure 5.7 Comparison of quality of discovered components

Legend:

Random – Selection of Headhunters is random

Non-Disjoint – Selection of Headhunter is not disjoint

To avoid the above scenario, the disjoint constraint is removed initially, and imposed after a certain level of Headhunters during the query propagation. This improves the quality of the discovered components, as the closed loop is not formed and the query is propagated to a larger subset of Headhunters.

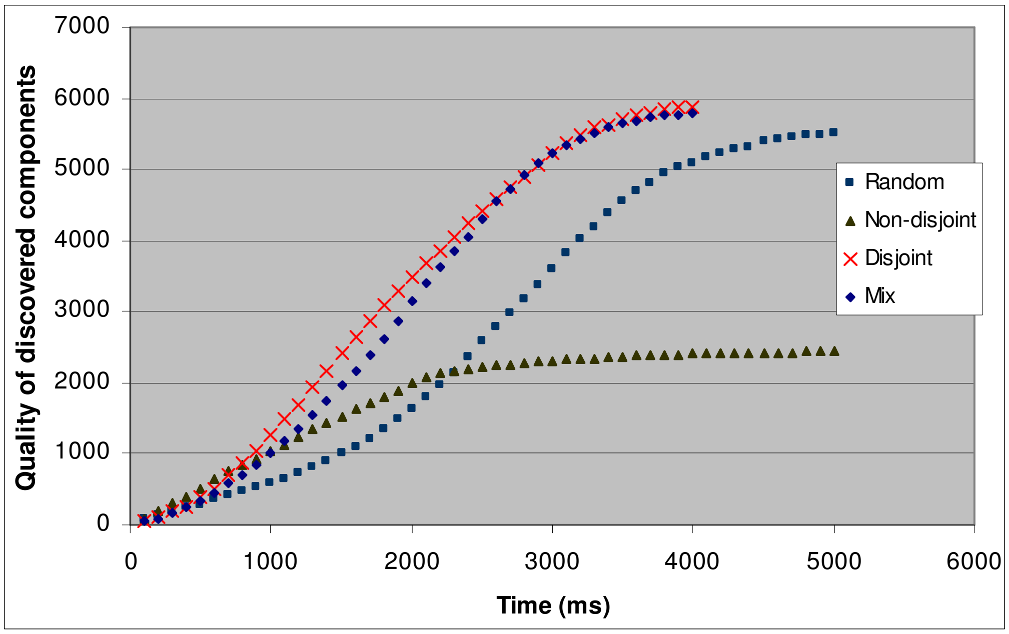


Figure 5.8 Comparison of quality of discovered components

Legend:

Random – Selection of Headhunters is random

Non-Disjoint – Selection of Headhunter is not disjoint

Disjoint – Selection of Headhunters is disjoint

Mix – Selection of Headhunters is not disjoint until 3rd level of Headhunters and disjoint for the rest of the levels

In Figure 5.8, the non-disjoint case is improved by imposing the disjoint Headhunter selection after the third level of Headhunters. This avoids the looping among a small subset of Headhunters, thus allowing the query to be propagated to a larger subset of Headhunters. This scenario is also compared with the case where all the Headhunters are selected in a disjoint fashion. There is a slightly different in time between these two cases. This is because; for the case where the disjoint criteria are removed for the first three levels of Headhunters, there is a possibility of the same Headhunters getting selected more than once. So, the number of Headhunters receiving the query in a particular time is less compared to the case where all the Headhunters are selected in a disjoint fashion. The Headhunters selected once for the query propagation contributes the same number of matching components, even in the case of getting selected multiple times. Hence, the probability of having a matching component is slightly more in the pure disjoint scenario, where the number of Headhunters receiving the query in a particular time is more.

The experiments in this section were conducted to observe the effect of the implementation of the profiling algorithms in the URDS architecture. These experiments concentrated on evaluating the improvement in the performance of the query propagation techniques, which is significant for reducing the time taken for the component discovery process and improving the quality of the discovered components. Different varieties of experiments were conducted to monitor the effect of each parameter in the enhancement process. The improvement in quality is affected by a lot of parameters such as: the time allowed for the component discovery process, the topology formed among the Headhunters due to propagation of the query, the imposition of the disjoint constraint for the selection of Headhunters during the query propagation, the distribution of the components in the Headhunters, the nature and order of the queries given to the Query Manager. Each experiment reflects the change in the factors that affect the quality of the discovered components. The experiments conducted compare the different scenarios to identify the best query propagation techniques. The conclusions drawn from these experiments are as follows:

- Implementation of profiling algorithms improves the quality of the discovered components in the URDS architecture. The components that closely match the query are discovered quickly, as compared to the non-profiled URDS version.
- The rate of the improvement in quality is based on certain factors such as the number of Headhunters in the URDS architecture, the component distribution in the Headhunters, the nature of queries, the number of levels of Headhunters using the profile information, the number of Headhunters selected by each entity for query propagation (which decides the topology formed by the Headhunters), and the criteria for the selection of the Headhunters (such as imposing disjoint Headhunter selection criteria).
- The learning process for the profile information is based on the order in which the queries are obtained.

5.4. Experiments on Handheld device

The following section concentrates on customizing the URDS for handheld devices. These handheld devices have a few challenges, such as restricted processing capabilities, less memory capacity, and less battery life. So, the customization of the URDS should take into account these hardware restrictions. It should provide software solutions so that the whole discovery process is transparent despite these hardware restrictions. The performance and behavior of the URDS architecture is studied by implementing different entities of the URDS onto the handheld devices.

For the experimental setup, a Sharp Zaurus SL 5600 is used hosting the Linux-based embedded OS v2.78 Embedix Qtopia. Both Personal Java and Java 2 Micro Edition (J2ME) are supported in the handheld devices. These Java versions are the restricted versions of J2SE and support only a subset of functionalities offered by J2SE. Personal Java (Jeode) is used in the experimentation, as it is part of the embedded Linux OS. Java RMI is supported as an optional package in Personal Java, and is included so that it can support the architecture of the URDS. The modifications to the entities were made on the desktop machine and the class files are ported to the Zaurus.

The existing URDS (i.e., the single threaded URDS version) is considered for the experimentation purpose. This is because of the memory and processor restrictions. A few changes have been made to customize the URDS architecture into the handheld device. They are as follows:

- The messages between the entities that form the URDS are encrypted in the existing URDS version to ensure security. Encryption and decryption is a costly process, and the Java versions for the handheld devices do not support the security packages. So, the encryption in the messages has been removed for the handheld version of URDS.
- The Active Registry introspects the components registered with it to obtain the functionality of the components. Introspection has been removed. Personal Java supports Introspection, but it still has been removed, as J2ME does not support Introspection. This is to make sure that this URDS version works in the handheld devices hosting J2ME.

The experiment is conducted by deploying different combinations of the URDS entities in the Zaurus. For all these combinations the average query response time is measured.

5.4.1. Experiments with single entities

Due to the number of restrictions imposed by the Zaurus (such as memory constraint, and processor speed), only one instance of each entity (i.e., one Headhunter, one Active Registry and one component) is considered in the first experimentation. All the average times are calculated by taking 10 samples of query response times.

Case 1:

All the entities are on the desktop machine i.e., Domain Security Manager, Headhunter, Active Registry, Query Manager and the components. This experiment is done for comparison purpose.

Average query response time: 1123.2 ms

Case 2:

Zaurus: Active Registry and one component

Desktop: DSM, Headhunter, and Query Manager

Average query response time: 1212.2 ms

This scenario is similar to case 1 in terms of query processing time because during the propagation of the query, only the Query Manager and the Headhunters are involved in processing the query. The Headhunter contacts the Active Registry only to update its meta-repository components. The slight increase in response time is due to the update of components happening between the Headhunter's meta-repository and the Active Registry during the processing of the query.

Case 3:

Zaurus: Query Manager

Desktop: DSM, Headhunter, Active Registry, and Components

Average Query Response Time: 1710 ms

The average query response time is slightly more than the *case 1* and the *case 2* as the Query Manager is in the Zaurus and has to communicate with the Headhunter in the desktop machine.

Case 4:

Zaurus: Headhunter

Desktop: DSM, Active Registry, Components, and Query Manager

Average Query Response Time: 5792 ms

The Average Query Response time is very high here because the Query Manager, which is deployed in the desktop machine, has to contact the Headhunter deployed in the Zaurus to search for the components. Since the Headhunter does the component matching process, the query response time is high for this case.

Case 5:

Zaurus: Headhunter and Query Manager

Desktop: DSM, Active Registry, and Components

Average Query Response Time: 7882 ms

The Average Query Response time is also high in this case as both the Headhunter and the Query Manager are executed in the Zaurus, which has much less memory and processing power.

Case 6:

Zaurus: Headhunter, Active Registry and Components

Desktop: DSM, Query Manager

These entities are not able to be executed in the Zaurus successfully. This is because of the memory and processor constraints. While trying to execute these components, a warning regarding memory insufficiency was given, and the processes were forced to terminate.

Case 7:

Zaurus: Query Manager, Active Registry and Components

Desktop: DSM, Headhunter

This scenario is same as case 6, where memory insufficiency warnings are displayed and the processes are forced to terminate.

The following tables summarizes the different combination of entities experimented with in the Zaurus.

Table 5.3 Comparison of query response time with different combination of components

Entities on the desktop	**Entities on the Sharp Zaurus**	**Average response time (in milliseconds)**
All	-	1123.2
Headhunter, Query Manager	Active Registry and a Component	1212.2
Headhunter, Active Registry, and a component	Query Manager	1710
Query Manager, Active Registry, and a component	Headhunter	5792
Active Registry and a component	Headhunter, Query Manager	7882
Query Manager	Headhunter, Active Registry and Components	Memory Insufficient – Processes are forced to terminate.
Headhunter	Query Manager, Active Registry and Components	Memory Insufficient –Processes are forced to terminate.

5.4.2. Experiments with multiple entities

The next set of experiments is conducted with multiple Headhunters and Active Registries. Here the URDS architecture includes a Domain Security Manager, a Query Manager, five Headhunters, and five Active Registries. The query propagation in this version works as follows. The Query Manager propagates the query received to one primary Headhunter. This primary Headhunter further propagates the query to one or more other Headhunters. The primary Headhunter collects the results from the query-propagated Headhunters and returns them to the Query Manager. This is done to measure the exclusive time taken by the Zaurus to process a query Headhunter is deployed in it is selected for query propagation by a Query Manager and when selected by other

Headhunters. Experiments are done with different combination in the selection of Headhunters for the query propagation.

Case 1:
All the entities are deployed on the desktop machines. This experiment is done for comparison purpose. Average Query Response Time: 24630 ms

Case 2:
Zaurus: One Headhunter
Desktop: Domain Security Manager, four Headhunters, five Active Registries and a Query Manager
The average query response time differs depending on whether or not the Headhunter in the Zaurus is selected as the primary Headhunter by the Query Manager.

Case 2(a): The Query Manager selects a Headhunter deployed on the desktop as the primary Headhunter. Average Query Response Time: 68303 ms

Case 2(b): The Query Manager selects the Headhunter on the Zaurus as the primary Headhunter. Average Query Response Time: 77212 ms

Case 2(b) takes more time than case 2(a) because of the primary Headhunters being in the Zaurus. This is because the primary Headhunter has to do the extra work of combining the components from different Headhunters to which the query was propagated.

5.4.3. Zaurus out-of-range experiments

The handheld devices are mobile and are connected to the network through a wireless hub. For the handheld device to be part of the network, the device has to be within the specified range from the hub. This section addresses the issue of when the Zaurus goes out-of-range from the hub when the URDS entities are communicating with each other.

Case 1: Out-or-range during query propagation

If the Zaurus goes out-of-range when the Query Manager tries to select the Headhunter in the Zaurus for query propagation, then the Query Manager automatically selects any of the other desktop Headhunters. Say, the Zaurus goes out-of-range after the Headhunter deployed in it has obtained the query from the Query Manager. Then the Headhunter in the Zaurus waits to propagate the query to other Headhunters and get back the results to the Query Manager until it is within the range.

Case 2: Out-or-range during updating of components in the meta-repository

For updating the components in the meta-repositories from the Active Registries, each Headhunter keeps sending multicast messages every few milliseconds, and the Active Registrie responds to that. In the case where the Headhunter and the Active Registry are in the Zaurus and the desktop machine respectively, or vice versa, the communication is successful whenever the Zaurus is within the range. Whenever the Zaurus goes out-of-range, the communication stops, and it continues whenever the Zaurus comes within the range.

The above subsections discussed the issues involved in customizing the URDS architecture into the handheld devices. It also discussed the problems faced due to the out-of-range problems in the handheld device. Only the single threaded version was considered in the experimentation as the handheld devices are restricted in processing power and memory, and the multi-threaded Headhunters will take a lot of processing power and memory due to the parallel processing involved.

This chapter discusses the experimental results and verification of the performance improvement made to the URDS architecture by using multiple threads, and the incorporation of the profiling at a few of the levels of the URDS architecture. It provides different graphs and the analysis of those graphs, proving that the implementation enhancements made improve the performance of the URDS architecture. It also exposes the issues involved in customizing the URDS architecture in to handheld devices. The next chapter summarizes this thesis by providing the conclusions of the research work, the contributions made by the thesis, a few of the possible future works, and a summary of the research work.

CHAPTER 6. CONCLUSION AND FUTURE WORK

This thesis presented different enhancements to the existing URDS architecture. The first section in this chapter provides the conclusion of this research work. Section 7.2 describes the contributions of this thesis work followed by section 7.3 which gives the possible future works to this research work. Finally, section 7.4 provides the summary of this thesis.

6.1. Conclusion

This research has incorporated different enhancements to the URDS architecture. These enhancements have been empirically evaluated by experimenting with the prototypes created. By making the Query Manager and the Headhunters multi-threaded, the enhanced architecture is able to process more than one query simultaneously. By implementing asynchronous communication for the propagation of queries, the overlap between the processing of queries in different entities is improved, thus reducing the time taken for the component discovery process. Due to the incorporation of various profiling algorithms at different levels of the URDS architecture, the query propagation techniques are improved so that the quality of the components discovered for a given search time limit is improved. Also, the URDS architecture is customized for handheld devices and the associated challenges are studied. The following are the conclusions drawn from the experimental results shown in the previous chapter:

- The implementation of multi-threading in the Query Manager and the Headhunter and the incorporation of asynchronous communication mechanisms for query propagation improve the performance of the URDS architecture.

- The implementation of the query propagation techniques in the Query Manager and the Headhunters using profile information improves the quality of the component discovery process.
- The learning process of the Query Manager and the Headhunters affects the selection of the Headhunters for future query propagations. This learning process depends on factors such as the number of Headhunters available in the URDS, the distribution of the components in the URDS, the topology formed by the Headhunters during the propagation of the queries, and the order in which the queries arrive.
- The customization of the URDS architecture for handheld devices has restrictions in the processing speed and the memory. Due to this, the handheld devices could support the URDS architecture implemented in the desktop machines, but cannot work independently by themselves.

6.2. Contributions of this thesis

The contribution of the thesis is as follows:

- It provides an enhanced version of the URDS architecture with demonstrated improvements in the performance of the component discovery process.
- It implements various reinforcement learning algorithms in the URDS architecture to improve the quality of the component discovery process.
- It customizes the query propagation by adding profile information at different levels of the URDS architecture to reduce the time taken for the discovering the components and improve the quality of the discovered components.
- In measures the influence of various factors (such as the component distribution, nature of queries, Headhunter topology formed, Headhunter selection criteria for the query propagation) on the implemented learning algorithms.
- It exposes the issues and challenges involved in implementing the entities of the URDS into handheld devices.

6.3. Future work

6.3.1. Future extension to the prototype

The prototype implementation discussed in this thesis includes the implementation of profiling only in the Query Manager and the Headhunter level. This could be extended by incorporating profiling algorithms in the Active Registries. Also, the profiling in the Query Manager and the Headhunter includes only a few of the algorithms described in Chapter 4. These algorithms mainly focus on improving the query propagation techniques. The other algorithms such as the UpdateComponentJoin, HeadhunterListChange, UpdateARConfidence, and UpdateCDConfidence could be added to the prototype. The UpdateComponentJoin updates the confidence level of the Headhunters based on the Headhunters' component matching capability. The HeadhunterListChange algorithm updates the reward probability vector maintained by the Headhunters whenever new Headhunters join the URDS or the existing Headhunters leave the URDS. The UpdateARConfidence and the UpdateCDConfidence maintains the confidence of the Active Registries and the component developers respectively. These two algorithms will aid in improving the quality of the components maintained in the meta-repository.

As discussed earlier, the propagation of the queries between the Query Manager and the Headhunters follows asynchronous communication. This is implemented by having a query queue for the propagation of the query. When a query is received in a query queue, the receiving Headhunter will notify the requestor whether it accepted the query for processing. An enhancement on this could be that the Headhunter could notify the approximate time it would take for the processing of the query based on the position of the query in its query queue and the capacity of the Headhunter in processing the queries. If the requestor is not satisfied with that time, an option could be provided to recall the query back and propagate it to a different Headhunter.

6.3.2. Future extension to the URDS

Several future extensions are possible to this research work, and a few of them are as follows:

- The acceptance of the components by the meta-repository of the Headhunters is exhaustive, i.e., the Headhunters accept all the components obtained from the Active Registries. This could be extended so that the Headhunters specialize in accepting a certain variety (such as only components with very high quality, or only components with a very low cost) of components. Profile information could be maintained accordingly, so that the selection of Headhunters for the query propagation will be based on the variety to which the query belongs. This will improve the selection of Headhunters for the propagation of queries, thus discovering the appropriate components quickly. This would improve the quality of the component discovered process.
- The reinforcement learning algorithms used in the URDS enhancements are based on [MUK02, OOM90, and THA85], where each node maintains profile information about other nodes. If the number of entities is very large, then there is a large overhead involved in maintaining the profile information for all the entities. Also, these profile information needs to be updated often. Since the Headhunters form a tree structure during the query propagation, the hierarchical reinforcement learning techniques discussed in [THA81, PAP94] could be implemented. Here, the profile information is not maintained for all the entities, but in different levels. So, the amount of profile information maintained by each entity is less, thus reducing the overhead involved in updating and processing them. This is useful when the number of entities in the URDS architecture is very large.

6.4. Summary

This thesis has incorporated enhancements to the existing URDS architecture in terms of reducing the time taken for the component discovery process, thus improving the performance of the URDS architecture. It has discussed and implemented various profiling algorithms at different levels of the URDS architecture to improve the query propagation techniques and the quality of the components discovered. This thesis has also provided an experimental analysis and the verification for the enhancements incorporated in the prototype. These experiments demonstrate that the implementation of the reinforcement learning algorithms in the URDS architecture improves the performance and quality of the component discovery process. Thus, the URDS architecture combined with profiling techniques makes it a promising solution for intelligent and sophisticated discovery of the components for building high-confidence DCS.

LIST OF REFERENCES

[AMA04] Amazon’s A9 search engine. http://www.a9.com. Last Referenced: Feb 2005.

[BRI98] Brin, S., Page, L. “The Anatomy of a Large-Scale Hyper-textual Web Search Engine”. Computer Science Department, Stanford University, Stanford, CA, 1998.

[COU01] Coulouris, G., Dollimore, J., Kindberg, T. “Distributed Systems Concepts and Design”, Third Edition, Addison-Wesley, 2001.

[CRN01a] Crnkovic, I. “Component-based Software Engineering - New Paradigm of Software Development”, MIPRO 2001 proceedings Opatija, Croatia, May 2001.

[CRN01b] Crnkovic, I. “Component-based Software Engineering - New Challenges in Software Development”, Malardalen University, Department of Computer Engineering, Sweden, Dec 2001.

[CZE99] Czerwinski, S. E., Zhao, B. Y., Hodes, T. D., Joseph, A. D., Katz, R. H., “An Architecture for a Secure Service Discovery Service”, Proceedings of ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99), 1999, pp. 24-35.

[DON01] Donnelly, M., Nicol, Y. “Customisable Resource Discovery Services”. Telecommunications Software Systems Group, Waterford Institute of Technology, Waterford, Ireland.

[DRD01] “Decentralized Resource Discovery in Large Peer Based Networks”. http://peertech.org/alpine/discovery.html. Last Referenced: Feb 2005.

[GOL99] Goland, Y., Cai, T., Leach P., Gu, Y., Albright, S. “Simple Service Discovery Protocol,” IETF, Draft draft-cai-ssdp-v1-03, Oct 28, 1999.

[GUT99] Guttman, E. “Service Location Protocol: Automatic Discovery of IP Network Services,” IEEE Internet Computing, vol. 3, no. 4, 1999, pp. 71-80.

[INF01] A Draft American National Standard Developed by the National Information Standards Organization. “Information Retrieval (Z39.50): Application Service Definition and Protocol Specification”. National Information Standards Organization Bethesda, Maryland, Jul 1995.

[JON03] Grant, S., Jones, P. R. and Ward, R. “Mapping Personal Development Records to IMS LIP to support Lifelong Learning”, The Center for Educational Technology Interoperability Standards, May 2003.

[KAE96] Kaelbling. L.P., Littman, M.L. and Moore, A.W. “Reinforcement Learning: A Survey”, Journal of Artificial Intelligence Research, 1996, pp. 237-285.

[KAT01] Katchaounov, T. “Query processing in self-profiling composable peer-to-peer mediator database”, EDBT Workshops, 2002, pp. 627-637.

[KUM04] Kumari, A. “Synchronization and Quality of Service Specifications and Matching of Software Components”. M. S. Thesis, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, Dec 2004.

[LAM86] Lampson B. W. “Designing a Global Name Service”, Proceedings of the fifth annual ACM symposium on Principles of distributed computing, Calgary, Alberta, Canada, 1986, pp. 1-10.

[LAR99] Larson, R. R. and Watry, P. B. “Cross-domain Resource Discovery: Integrated Discovery and use of Textual, Numeric, and Spatial Data”, Annual report for JISC/NSF, Jan 1999.

[LDA01] Open LDAP. http://www.openldap.org/jldap/overview.html. Last referenced: Feb 2005.

[MAU97] Maudlin, Michael. “Lycos: Design Choices in an Internet Search Service”, IEEE Expert, Jan 1997.

[MIC00] Microsoft Corporation, “Universal Plug and Play Device Architecture Version 1.0,” June 8, 2000, http://www.upnp.org/download/UPnPDA10_20000613.htm. Last Referenced: Feb 2005.

[MIL99] Miller, B. “Mapping Salutation Architecture APIs to Bluetooth service discovery layer – v1.0”, white paper, Jul 1999.
http://www.salutation.org/whitepaper/BtoothMapping.PDF. Last referenced: Feb 2005.

[MOU87] Mockapetris, P. “Domain Names-Implementation and Specification,” IETF RFC 1035, Oct 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[MUK02] Mukhopadhyay, S., Peng. S., Raje. R., Palakal. M. and Mostafa. J. “Multi-agent Information Classification Using Dynamic Acquaintance Lists”, Journal of the American Society for Information Science and Technology, Aug 2003, pp. 966-975.

[NAN02] Nanditha N. Siram. “An Architecture for the UniFrame Resource Discovery Service”. M. S. Thesis. Department of Computer & Information Science, Indiana University Purdue University Indianapolis, Mar 2002.

[NIN02] Ninja, “The Ninja Project,” http://ninja.cs.berkeley.edu, 2002. Last referenced: Feb 2005.

[OBJ00] Object Management Group. “Trading Object Service Specification”. Object Management Group 2000. ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf. Last referenced: Feb 2005.

[OGS01] Towards Open Grid Services Architecture. http://www.globus.org/ogsa/. Last referenced: Feb 2005.

[OOM90] Oommen, J. and Lanctot, K. “Discretized Pursuit Learning Automata”, IEEE Transactions on systems, Volume 20, Aug 1990.

[PAP94] Papadimitriou, G.I. “Hierarchical Discretized Pursuit Nonlinear Learning Automata with Rapid Convergence and High Accuracy”. IEEE Transactions on Knowledge and Data Engineering, Aug 1994.

[PAR01] Parkhomenko, O., Lee, Y. and Park, E. K. “Ontology-Driven Peer Profiling in Peer-to-Peer Enabled Semantic Web”. Proceedings of the Twelfth International Conference on Information and Knowledge Management, New Orleans, LA, USA, pp. 564-567.

[PER99] Perkins, C., Guttman, E., “DHCP Options for Service Location Protocol,” IETF RFC 2610, Jun 1999.

[RAJ00] Raje, R. “UMM: Unified Meta-object Model fir open distributed systems”, Proceedings of 4th IEEE International Conference on Algorithms and Architecture for Parallel Processing, ICA3PP'2000, Hongkong, 2000, pp: 454-465.

[RAJ01] Raje, R., Auguston, M., Bryant, B. R., Olson, A., Burt, C. “A Unified Approach for the Integration of Distributed Heterogeneous Software Components”. Proceedings of the 2001 Monterey Workshop on Engineering Automation for Software Intensive System Integration, Monterey, CA, 2001, pp. 109-119.

[RDN01] Resource Discovery Network http://www.rdn.ac.uk/publications. Last referenced: Feb 2005.

[RDN02] Resource Discovery Network - Subject Portal Project. http://www.portal.ac.uk/spp/. Last referenced: Feb 2005.

[REK99] Rekesh John, “UPnP, Jini and Salutation - A look at some popular coordination frameworks for future networked devices,” California Software Labs, Jun 17, 1999.

[ROU01] Rousseau, B., Browne, P., Malone, P. and Ofoghlu, M. “Personalized Resource Discovery Searching over multiple repository types - Using user and information provider profiling”. Telecommunications Software Systems Group (TSSG), Waterford Institute of Technology, Waterford, Ireland, 2003.

[SAL99] Salutation Consortium, “Salutation Architecture Specification Version 2.0c - Part 1 and part 2”. The Salutation Consortium, Jun 1, 1999. http://www.salutation.org. Last referenced: Feb 2005.

[SEA98] Seacord, R., Hissam, S. and Wallnau, K. “Agora: A Search Engine for Software Components”, Technical report CMU/SEI-98-TR-011, Aug 1998.

[SUN01] Sun Microsystems, Jini Specifications V2.0, http://java.sun.com/products/jini/. Last referenced: Feb 2005.

[THA00] Thathachar, M.A.L. and Oommen, J. “Discretized Reward-Inaction Learning Automata”, Department of Electrical Engineering, Indian Institute of Science, Bangalore, India.

[THA81] Thathachar, M.A.L. and Ramakrishnan, K.R. “A Hierarchical System of Learning Automata”, IEEE Transactions on Systems, and Cybernetics, March 1981.

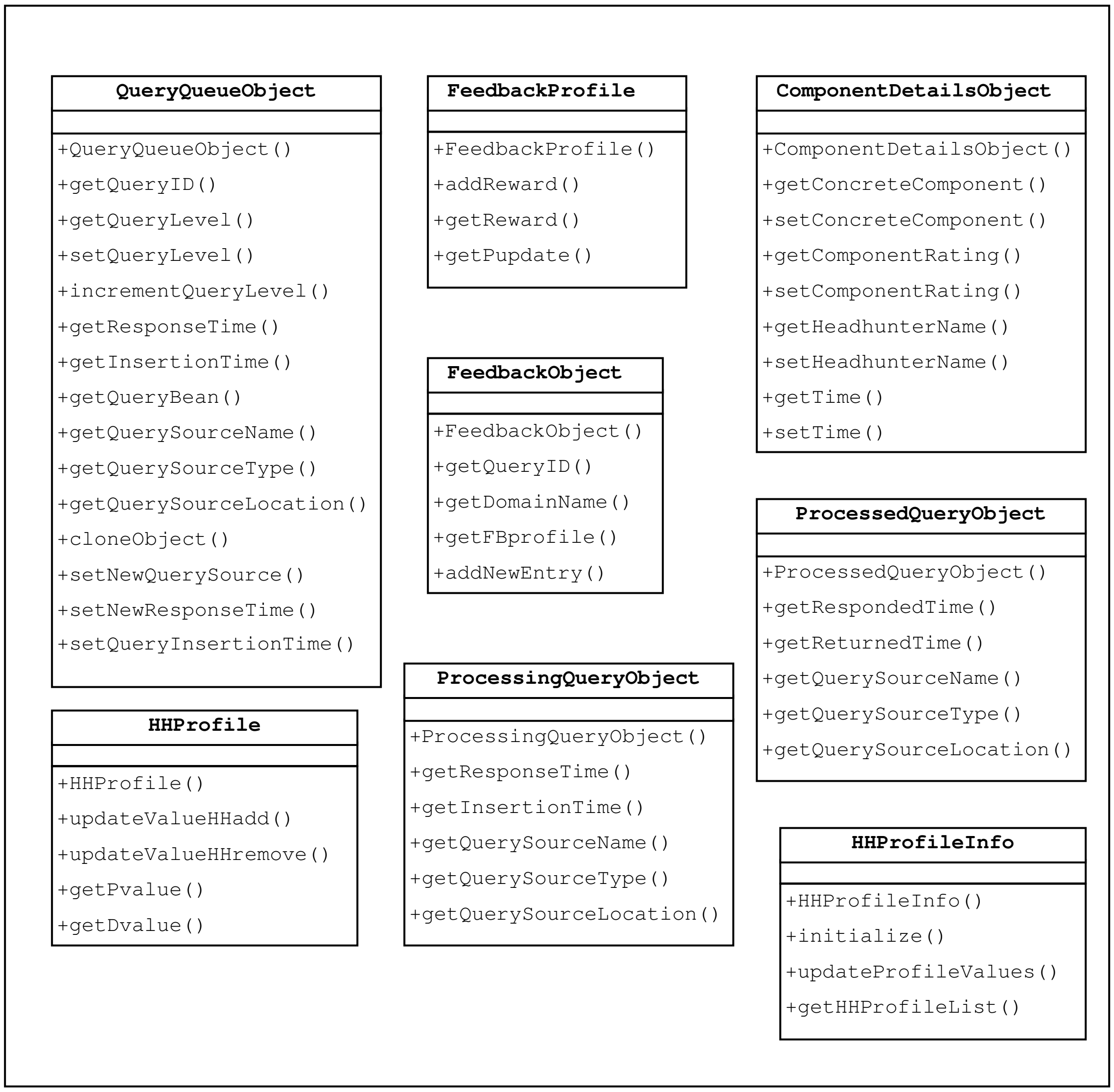
[THA85] Thathachar, M.A.L. and Sastry, P.S. “A new approach to the design of Reinforcement schemes for Learning automata”, IEEE Transactions on systems, Feb 1985.

[UDD00] “UDDI Technical White Paper”, Sep 2000. http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf. Last referenced: Feb 2005.

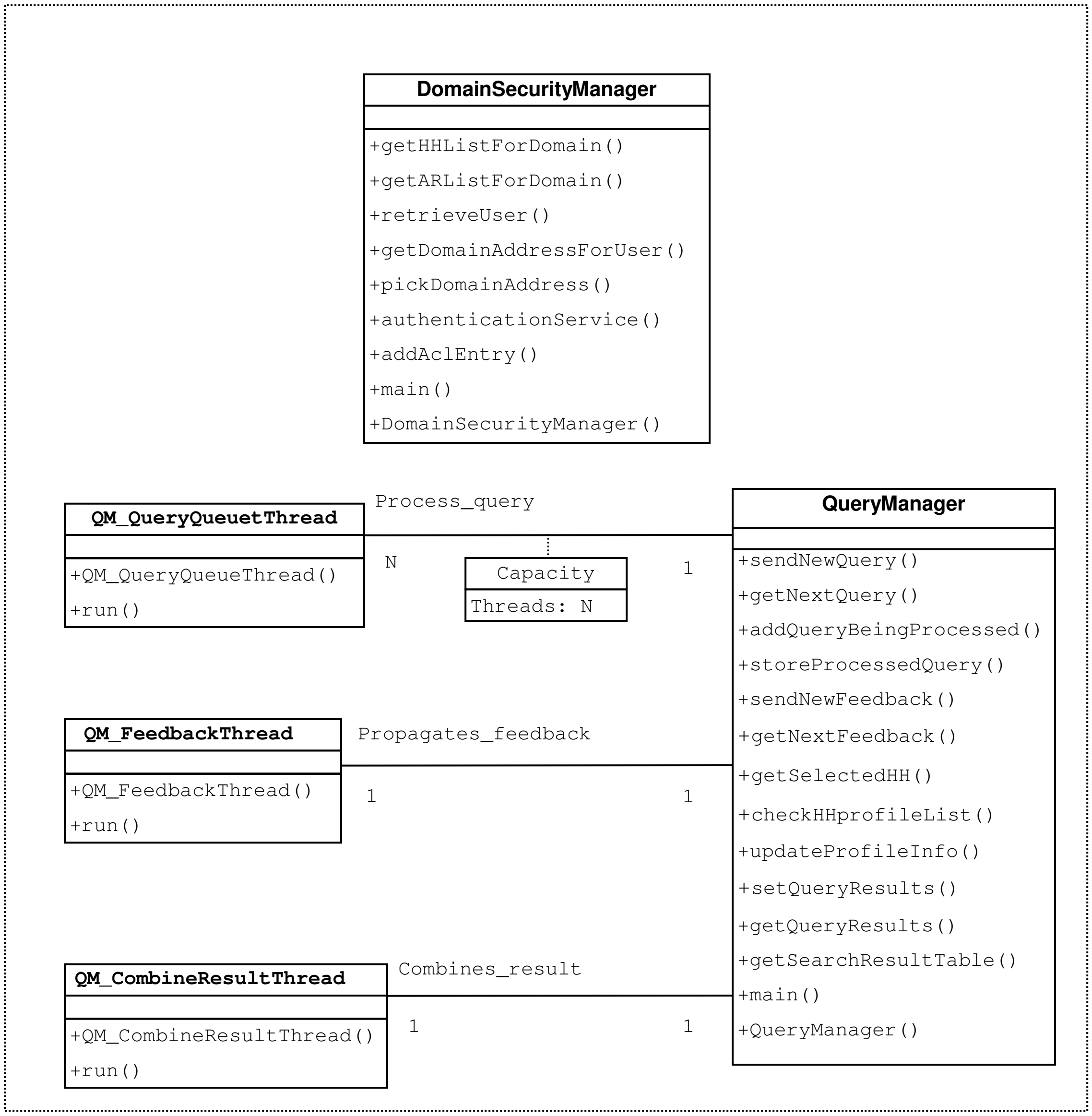
[WAR01] Ward, N. and Wood, A. “Emerging Technologies for Networked Information Discovery: Beyond Z39.50”. CRC for Distributed Systems Technology http://www.dstc.edu.au/RDU/publications/e_vala98/. Last referenced: Feb 2005.

Appendix A. Class Diagrams

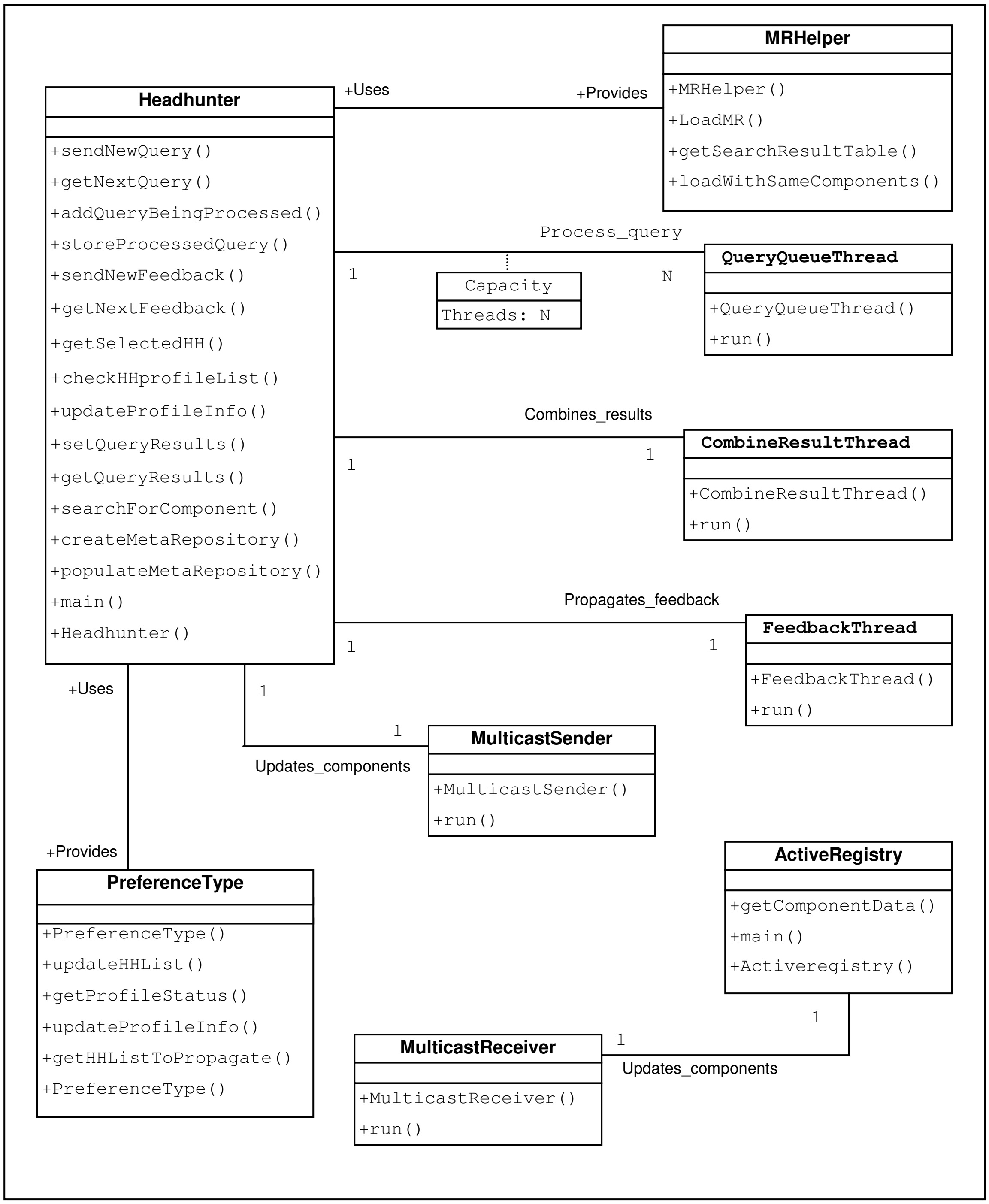
Class Diagrams for the Entity Objects



Class Diagrams for the Service Components



Class Diagrams for the Service Components



Appendix B. Source Code

AbstractComponent.java

```
/**
 * This class represents an Abstract component
 *
 * @author Zhisheng Huang
 * @date January 2003
 */

public class AbstractComponent extends Component
{
}
```

ActiveRegistry.java

```
/**
 * Communicates with Headhunter to update components
 *
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */

import java.rmi.registry.*;
import java.rmi.*;
import java.net.*;
import java.util.*;
import java.rmi.server.*;
import java.lang.Integer;
import java.lang.String;

public class ActiveRegistry extends UnicastRemoteObject implements IActiveRegistry
{
   private int port = 9000;
   private String userType = "Registry";
   private String rmiLocn =null;

   private String[] list(int argPort,String rmiLocn)
   {
             String[] listOfURLS = null;
             try
             {
                listOfURLS = Naming.list("//"+rmiLocn+":" + argPort);
                System.out.println("argPort is ...."+argPort);
             }
             catch (Exception e)
             {
                System.out.println(e.getMessage());
             }
             return listOfURLS;
   }

   public static void main(String[] args)
   {
             int rmiRegistryPort = 0000;
             int mcastPort = 10000;
             String userName = args[5];
             String password = args[6];
             String domain = args[4];

             rmiRegistryPort = Integer.parseInt(args[2].trim());
             System.out.println("Port in use is"+rmiRegistryPort);
             String activeRegistryLocation="//"+args[0]+":"+args[1] + "/ActiveRegistry";
             String dsmLocation ="//magellan.cs.iupui.edu:"+ args[3]+"/DomainSecurityManager";

             try
```

```
            {
               System.setSecurityManager(new RMISecurityManager());
               Naming.rebind( activeRegistryLocation, new ActiveRegistry(
                                          rmiRegistryPort, mcastPort, userName, password,
                                          domain, activeRegistryLocation, dsmLocation) );
               System.out.println("ActiveRegistry is ready.");
            }
            catch (Exception e)
            {
               System.out.println("ActiveRegistry failed: " + e);
            }
}

public ActiveRegistry(int rmiRegistryPort, int mcastPort, String userName,
            String password, String domain, String activeRegistryLocation, String dsmLocation)
            throws RemoteException
{
            try
            {
               System.out.println("String is "+activeRegistryLocation);
               String rmiRegistryLocation = (InetAddress.getLocalHost()).toString();
               port = rmiRegistryPort;
               int j=0;
               for (int i=0;i<activeRegistryLocation.length();i++)
               {
            if(activeRegistryLocation.charAt(i)==':')
          {
            j =i;
            i = activeRegistryLocation.length();
          }
               }
               rmiLocn=activeRegistryLocation.substring(2,j);
               System.out.println("Substring "+rmiLocn);
               LocateRegistry.createRegistry(port);

               rmiRegistryLocation = rmiRegistryLocation + ":" + port;
               System.out.println("\n Active Registry Created RMI Registry At : " + rmiRegistryLocation);
               IDomainSecurityManager dsmanager =
               (IDomainSecurityManager) Naming.lookup(dsmLocation);
               AuthenticatedPacket authpacket = null;

               System.out.println("Active Registry Contacting DSM for Authentication.");
               authpacket = dsmanager.authenticationService(
                                          userType, userName, password,
                                          activeRegistryLocation, domain);
               System.out.println("Active Registry Authenticated by DSM.");

               MulticastReceiver mcastReceiver = new MulticastReceiver(mcastPort,
                                 authpacket, activeRegistryLocation);
               Thread receiverThread = new Thread(mcastReceiver);
               receiverThread.start();
            }
            catch (Exception e)
            {
               System.out.println("what"+e.getMessage());
            }
} //end of constructor

public Hashtable getComponentData() throws RemoteException
{
            Hashtable objectTable = new Hashtable();
            System.out.println("AR contacted by Headhunter to Retrieve Component Data");

            try
            {
               String[] objURL = list(port,rmiLocn);
               for(int i=0;i<objURL.length;i++)
               {
```

```
                              System.out.println("Active Registry gathering component information from : " + objURL[i]);

                              Object  obj = Naming.lookup(objURL[i]);
                              //Obtain the location(URL) of the UMMSpecification for this object by
                              String ummSpecURL = (String) UniFrameIntrospector.getProperty(obj,"ummSpecURL");
               System.out.println("ummSpecRL is "+ummSpecURL);

                              UniFrameSpecificationParser xmlDomParser = new UniFrameSpecificationParser(ummSpecURL);
                              ConcreteComponent component = xmlDomParser.getConcreteComponent();

                              //Add the component to the Hashtable
                              objectTable.put(objURL[i],component);
                   }//end for
                }
                catch(Exception e)
                {
                   System.out.println("Error in getComponent() -- ");
                   e.printStackTrace();
                }

                //Once the Hashtable is filled return the hashtable
                return objectTable;
    }
}
```

CombineResultThread.java

```
/**
 * Combines the results obtained from different HHs and send them to the requestor
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.net.*;
import java.security.*;
import java.util.*;
import java.lang.*;
import java.rmi.*;
import java.rmi.registry.*;

public class CombineResultThread implements Runnable
{
   private Headhunter hh = null;
   private long sleepTime = 400;
   private long thresholdTime = 600;

   public void run()
   {
                Thread CurrentThread = Thread.currentThread();
                Hashtable queryResults;
                while(true)
                {
                   beginThread = System.currentTimeMillis();
                   try
                   {
                              queryResults = hh.getQueryResults();
                              ProcessingQueryObject processingQuery;
                              boolean enterWhileFlag = false;
                Enumeration e = queryResults.keys();
                while(e.hasMoreElements())
                {
                                 enterWhileFlag = true;
                   String queryID = (String) e.nextElement();
                   ResultObject resultObject = (ResultObject) queryResults.get(queryID);
                                 long expiredTime = 0;
                                 long remainingResponseTime = 0;
```

```
                    processingQuery = hh.getProcessingQueryFromHash(queryID);
                    if(processingQuery==null)
          {
            System.out.println("#### Combine: ERROR - Query " +queryID + " NOT found in ProcessingQuery Hashtable");
          }
                    else
                    {
                          expiredTime = System.currentTimeMillis() - processingQuery.getInsertionTime();
                          remainingResponseTime = processingQuery.getResponseTime() - expiredTime;
           resultObject.checkResultsObtained(queryID);
                  if(resultObject.isAllResultsObtained() || remainingResponseTime < thresholdTime)
                  {
                            String sourceLocation = processingQuery.getQuerySourceLocation();
                            String hhName = hh.getHHname();
                            if(sourceLocation.indexOf("HeadHunter") > 0)
                        {
                                  IHeadhunter ihh = null;
                                  try
                                  {
                                    ihh = (IHeadhunter) Naming.lookup(sourceLocation);
                      Hashtable matchedComp =  resultObject.getMatchedComponents();
                    if(matchedComp!=null && matchedComp.size()>0)
                    {
                                  ihh.setQueryResults(queryID, matchedComp, hhName);
                    }
                                    else
                                    {
                                          ihh.setQueryResults(queryID, hhName);
                                    }
                              }
                              catch(Exception ee)
                              {
                                    System.out.println(ee);
                              }
                            }
                            else
                            {
                                  IQueryManager qm = null;
                                  // requestor is QM - look up for object
                                  try
                                  {
                                    qm = (IQueryManager) Naming.lookup(sourceLocation);
                                  }
                                  catch(Exception eee)
                                  {
                                    System.out.println(eee);
                                  }

                              try
                              {
                                    // Check whether components are available
                                    Hashtable results = resultObject.getMatchedComponents();
                                    if(results!=null && results.size()>0)
                                    {
                                  qm.setQueryResults(queryID, results , hhName);
                                    }
                                    else
                                    {
                                  qm.setQueryResults(queryID, hhName);
                                    }
                                  }
                                  catch(Exception eee)
                                  {
                                    System.out.println(eee);
                                  }
                        }
                            long timeToProcess =  System.currentTimeMillis() - processingQuery.getInsertionTime();
                            hh.removeQueryResultEntry(queryID);
```

```
                                        hh.storeProcessedQuery(queryID, timeToProcess);
                            }
                          } // end of ELSE
                            }

                            CurrentThread.sleep(sleepTime);
                  }
                  catch(Exception e)
                  {
                            e.printStackTrace();
                  }
              }
   }

   public CombineResultThread(Headhunter head)
   {
              try
              {
                  hh = head;
              }
              catch (Exception e)
              {
                  e.printStackTrace();
              }
   }
}
```

Component.java

```
/**
 * This class represents a component
 *
 * @author Zhisheng Huang
 * @date January 2003
 * @modified: Barun Devaraju, Nov 2004
 */

import java.util.*;
import java.io.*;

abstract public class Component implements Serializable
{
   private String componentName = "";
   private String subcase = "";
   private String domainName = "";
   private String systemName = "";
   private String description = "";

   private String id = "";   //host id, seems not necessary in abastract component
   private String version = "";
   private String author = "";
   private String date = "";
   private String validity = "";
   private String atomicity = "Yes";
   private String registration = "";
   private String model = "";

   private String purpose = "";    //describe the function of the component
   private String[] algorithms = null;
   private String complexity = "";
   private String[] requiredInterfaces = null;
   private String[] providedInterfaces = null;
   private String[] technologies = null;
   private String[] expectedResources = null;
   private String[] designPatterns = null;
   private String[] knownUsages = null;
   private String[] aliases = null;
private String[] preProcessingCollaborators = null;
private String[] postProcessingCollaborators = null;

private String mobility = "No";
private String security = "";
private String faultTolerance = "";

private String[] qosMetrics = null;
private String qosLevel = "";
private String cost = "";
private String qualityLevel = "";

private ComponentQoS componentQoS;

public void setComponentName (String componentName){this.componentName = componentName;}
public void setSubcase(String subcase){this.subcase = subcase;}
public void setDomainName (String domainName){this.domainName = domainName;}
public void setSystemName(String systemName){this.systemName = systemName;}
public void setDescription (String description){this.description = description;}

public void setID (String id)  {this.id = id;}//host id, seems not necessary in abastract component
public void setVersion (String version){this.version = version;}
public void setAuthor (String author){this.author = author;}
public void setDate (String date){this.date = date;}
public void setValidity (String validity){this.validity =validity;}
public void setAtomicity(String atomicity) {this.atomicity = atomicity;}
public void setRegistration (String registration){this.registration = registration;}
public void setModel (String model){this.model = model;}

public void setPurpose (String purpose) {this.purpose = purpose;} //describe the function of the component
public void setAlgorithms (String[] algorithms) {this.algorithms = algorithms;}
public void setComplexity (String complexity){this.complexity = complexity;}
public void setRequiredInterfaces (String[] interfaces) {this.requiredInterfaces = interfaces;}
public void setProvidedInterfaces (String[] interfaces) {this.providedInterfaces = interfaces;}
public void setTechnologies (String[] technologies){this.technologies = technologies;}
public void setExpectedResources (String[] expectedResources){this.expectedResources = expectedResources;}
public void setDesignPatterns (String[] designPatterns){this.designPatterns = designPatterns;}
public void setKnownUsages (String[] unknownUsages) {this.knownUsages = knownUsages;}
public void setAliases (String[] aliases) {this.aliases = aliases;}

public void setPreProcessingCollaborators (String[] collaborators) {this.preProcessingCollaborators = collaborators;}
public void setPostProcessingCollaborators (String[] collaborators) {this.postProcessingCollaborators = collaborators;}

public void setMobility (String mobility) {this.mobility = mobility;}
public void setSecurity (String security) {this.security = security;}
public void setFaultTolerance (String faultTolerance){this.faultTolerance = faultTolerance;}

public void setQoSMetrics (String[] qosMetrics){this.qosMetrics = qosMetrics;}
public void setQoSLevel (String qosLevel){this.qosLevel = qosLevel;}
public void setCost (String cost){this.cost = cost;}
public void setQualityLevel (String qualityLevel){this.qualityLevel = qualityLevel;}

public void setComponentQoS(ComponentQoS componentQoS){this.componentQoS = componentQoS;}

public String getComponentName(){ return componentName;}
public String getSubcase(){ return subcase;}
public String getDomainName(){ return domainName;}
public String getSystemName() {return systemName;}
public String getDescription(){ return description;}

public String getID(){ return id;}   //host id, seems not necessary in abastract component
public String getVersion(){ return version;}
public String getAuthor(){ return author;}
public String getDate(){ return date;}
public String getValidity(){ return validity;}
public String getAtomicity () {return atomicity;}
public String getRegistration(){ return registration;}
public String getModel(){ return model;}
```

```
   public String getPurpose(){ return purpose;}    //describe the function of the component
   public String[] getAlgorithms(){ return algorithms;}
   public String getComplexity(){ return complexity;}
   public String[] getRequiredInterfaces () {return requiredInterfaces;}
   public String[] getProvidedInterfaces() {return providedInterfaces;}
   public String[] getTechnologies(){ return technologies;}
   public String[] getExpectedResources(){ return expectedResources;}
   public String[] getDesignPatterns(){ return designPatterns;}
   public String[] getKnownUsages(){ return knownUsages;}
   public String[] getAliases(){ return aliases;}

   public String[] getPreProcessingCollaborators() {return preProcessingCollaborators;}
   public String[] getPostProcessingCollaborators() {return postProcessingCollaborators;}

   public String getMobility (){return mobility;}
   public String getSecurity(){ return security;}
   public String getFaultTolerance(){ return faultTolerance;}

   public String[] getQoSMetrics() {return qosMetrics;}
   public String getQoSLevel(){ return qosLevel;}
   public String getCost(){ return cost;}
   public String getQualityLevel(){ return qualityLevel;}

   public ComponentQoS getComponentQoS(){return componentQoS;}

   public String toString()
   {
      return domainName + "/" + systemName + "/" + componentName;
   }
}
```

ComponentDetailsObject.java

```
import java.io.*;

/**
 * Details of a matched component
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

public class ComponentDetailsObject implements Serializable
{
   private ConcreteComponent concreteComponent;
   private float componentRating;
   private String hhName;
   private long timeFound=0;

   public ComponentDetailsObject(ConcreteComponent cc, float rating)
   {
              concreteComponent = cc;
              componentRating = rating;
   }

   public ConcreteComponent getConcreteComponent()
   {
              return concreteComponent;
   }

   public float getComponentRating()
   {
              return componentRating;
   }

   public String getHeadhunterName()
   {
```

```
                return hhName;
    }

    public void setConcreteComponent(ConcreteComponent cc)
    {
                concreteComponent = cc;
    }

    public void setComponentRating(float rating)
    {
                componentRating = rating;
    }

    public void setHeadhunterName(String hh)
    {
                hhName = hh;
    }

    public void setTime(long time)
    {
                timeFound = time;
    }

    public long getTime()
    {
                return timeFound;
    }
}
```

ConcreteComponent.java

```
/**
 * This class represents a concrete component
 *
 * @author Zhisheng Huang
 * @date January 2003
 */

public class ConcreteComponent extends Component
{
  public ConcreteComponent(){}
}
```

DomainSecurityManager.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 * @modified: Barun Devaraju, Nov 2004
 */

import java.net.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.security.Principal;
import java.security.acl.*;
import sun.security.acl.*;

public class DomainSecurityManager extends UnicastRemoteObject implements IDomainSecurityManager
{
            private static String qmLocation = null;
            private static final String securityOwnerString = "DomainSecurityManager";
            private static final String aclIdentifier = "DomainSecurityACL";
            private static final Principal securityOwner = new PrincipalImpl(securityOwnerString);
            private static final Acl domainACL = new AclImpl( securityOwner,aclIdentifier);
            private static int size=0;
```

```
        private static Hashtable userdomainMapping = null;
        private static Hashtable domainList = null;
        private static Hashtable HHAddressAllocTable = new Hashtable();
        private static Hashtable registeredHHTable = new Hashtable();
        private static Hashtable registeredHHTableWithProfile = null;
        private static ArrayList registeredHHname = new ArrayList();
        private static Hashtable registeredARTable = new Hashtable();
        private static Random rand = new Random();

  public DomainSecurityManager() throws RemoteException
  {
        try
        {
          createACLEntries();
        }
        catch (DomainSecurityManagerException e)
        {
          System.out.println(e);
        }
  }

  public static void main(String[] args)
  {
        String dsmLocation="//magellan.cs.iupui.edu:"+args[0]+"/DomainSecurityManager";
        try {
                System.setSecurityManager(new RMISecurityManager());
                Naming.rebind(dsmLocation, new DomainSecurityManager());
                System.out.println("DomainSecurityManager " + dsmLocation + " is ready.");

        } catch (Exception e) {
                System.out.println("DomainSecurityManager failed: " + e);
        }
  }

  public void setQMlocation(String qmLoc) throws RemoteException
  {
        qmLocation = qmLoc;
  }

  public String getQMlocation() throws RemoteException
  {
        return qmLocation;
  }

public java.util.ArrayList getHHListForDomain(String domainName)
        throws RemoteException
 {
        ArrayList hhList = new ArrayList();
        Enumeration e = registeredHHTable.keys();

        while (e.hasMoreElements()) {
                String key = (String) e.nextElement();
    String value = (String) registeredHHTable.get(key);
                if((value).equalsIgnoreCase(domainName))
        {
                        hhList.add(key);
                        System.out.print(key + " ");
                }//end if
        }//end while
        System.out.println();
        return hhList;
  }

public java.util.ArrayList getARListForDomain(String domainName)
        throws RemoteException
 {

        System.out.println("DSM Contacted for AR List for : " + domainName + " Domain");
```

```
            ArrayList arList = new ArrayList();
            Enumeration e = registeredARTable.keys();
            while (e.hasMoreElements()) {
                        String key = (String) e.nextElement();
      String value = (String) registeredARTable.get(key);
                        if((value).equalsIgnoreCase(domainName))
            {
                                    arList.add(key);
                        }//end if
            }//end while
            return arList;
   }


private static Principal retrieveUser( String userType, String userName,
            String password) throws DomainSecurityManagerException
  {
            boolean userExists = DSMRepositoryHelper.authenticateUser(userType, userName, password);
            System.out.println("User Exist: "+userExists);

            // if not found, throw exception
            if (!userExists) {
                        throw new DomainSecurityManagerException(
                                    "User " + userName + " failed authentication##.", null);
            }
            System.out.println("DSM authenticated " + userName);
            // Create the Principal object
            Principal user = new PrincipalImpl(userName);
            return user;
}

private static String getDomainAddressForUser(
            String userType, String userName, String password, String location,
            String domainName) throws DomainSecurityManagerException
   {
            String domainAddress = null;
            Principal user = retrieveUser(userType, userName, password);
            Permission permission = new PermissionImpl(domainName);
            if (isUserAuthenticated(user, permission)) {
                        domainAddress = pickDomainAddress(userType, domainName, location);
            } else {
                        throw new DomainSecurityManagerException( "User " + userName + " is not authorized for the domain " +
domainName, null);
            }
            registeredHHname.add(userName);
            return domainAddress;
}

private static String pickDomainAddress( String userType, String domainName,
            String location) throws DomainSecurityManagerException
   {
      String domainAddress = null;
      if ((domainList != null) && (domainList.containsValue(domainName)))
      {
            if ((userType.equals("Registry"))
                  && (HHAddressAllocTable.containsValue(domainName)) && (!registeredHHTable.isEmpty()))
            {
                  Enumeration e = HHAddressAllocTable.keys();
                  Vector thisVector = new Vector();
                  while (e.hasMoreElements())
                  {
                        String key = (String) e.nextElement();
                        if (((String) HHAddressAllocTable.get(key)).equals(domainName))
                              thisVector.add(key);
                  }

                  size = thisVector.size();
                  if (size > 0)
```

```
                {
                    rand_num_id_AR = (( (rand_num_id_AR+1) % size ));
                    System.out.println("Magic no is (if)" + rand_num_id_AR);
                    domainAddress=(String) thisVector.get(rand_num_id_AR);
                    registeredARTable.put(location,domainName);
                }
            }
            else if (userType.equals("Registry"))
            {
                if (domainAddress != null)
                    registeredARTable.put(location,domainName);
                System.out.println("Registered " + (String) registeredARTable.get(location) +
                    " Active Registry located at " + location);
            }
            else
            {
                if ((domainAddress != null) && (userType.equals("HeadHunter"))
                        && !(HHAddressAllocTable.containsKey(domainAddress)))
                {
                    HHAddressAllocTable.put(domainAddress, domainName);
                }
                registeredHHTable.put(location,domainName);
                System.out.println("Registered " + (String) registeredHHTable.get(location) +
                    " Headhunter located at " + location);

            }
        }
        else
        {
            throw new DomainSecurityManagerException("No Such Domain Exists:  " + domainName, null);
        }
        return domainAddress;
    }

private static void addAclEntry(String userName, String domain)
            throws DomainSecurityManagerException
    {
            // Create the Principal object
            Principal user = new PrincipalImpl(userName);
            // create a new Acl entry for this user
            AclEntry newAclEntry = new AclEntryImpl(user);
            // initialize some temporary variables
            Permission permission = new PermissionImpl(domain);
            // add the permission to the aclEntry
            newAclEntry.addPermission(permission);
            try {
                    // add the aclEntry to the ACL for the securityOwner
                    domainACL.addEntry(securityOwner, newAclEntry);

            } catch (NotOwnerException noE) {
                    throw new DomainSecurityManagerException("In addAclEntry", noE);
            }
    }

    public AuthenticatedPacket authenticationService( String userType, String userName,
            String password, String location, String domain) throws RemoteException
    {
            AuthenticatedPacket authPacket = new AuthenticatedPacket(null);
            try {
                    authPacket.setMCAddress( getDomainAddressForUser(
                                    userType, userName, password, location, domain));
            } catch (Exception e) {
                    System.out.println(e);
            }
            return authPacket;
    }

    private static void createACLEntries() throws DomainSecurityManagerException
```

```
   {
          try {
                 if (userdomainMapping == null) {
                        // initialize the jdbc helper class
                        DSMRepositoryHelper.initialize();
                        userdomainMapping = DSMRepositoryHelper.loadUserDomainMapping();

                        if (userdomainMapping != null) {
                               Enumeration e = userdomainMapping.keys();
                               while (e.hasMoreElements()) {
                                      String key = (String) e.nextElement();
                                      String domainName = (String) userdomainMapping.get(key);
                                      addAclEntry(key, domainName);
                                      System.out.println(
                                             "Added ACLEntry for User = " + key + " Domain = " +
domainName);

                               } //while
                        } //if(userdomainMapping != null)
                 } //if (userdomainMapping==null)

                 if (domainList == null) {
domainList = DSMRepositoryHelper.loadDomainList();
                        System.out.println("\n Loaded DomainList");
                 }

          } catch (Exception e) {
                 throw new DomainSecurityManagerException("Error in init method", e);
          }
   }
}
```

DSMRepositoryHelper.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */

import java.sql.*;
import java.util.*;

public class DSMRepositoryHelper
{
   public static void initialize()
   {
          try {
                 sqlHelper = new SQLHelper();

          } catch (Exception e) {
                 System.out.println(e);
          }
   }

   public static boolean authenticateUser( String sUserType, String sUserName,
          String sPassword)
   {
          boolean isAuthenticated = false;
          try {
                 String sUserQuery =
                        "SELECT UserName From Users "
                               + "WHERE ( ( Users.UserName = '"
                               + sUserName
                               + "') AND ( Users.Password = '"
                               + sPassword
                               + "' ) AND ( Users.UserType = '"
                               + sUserType
                               + "' ) )";
```

```
				// execute the Query
				ResultSet resultSet = sqlHelper.executeQuery(sUserQuery);
				if (!resultSet.next()) {
					return false;
				}
				String sResult = resultSet.getString(1);
				if (sResult != null) {
					isAuthenticated = sUserName.equals(sResult);
				}
			} catch (Exception e) {
				e.printStackTrace();
				return false;
			}
			return isAuthenticated;
	}

	private static Hashtable getFeaturesFromResultSet(ResultSet resultSet)
			throws SQLException
	{
			Hashtable hFeatures = new Hashtable();
			String sFeatureName = null;
			String sFeatureValue = null;
			while (resultSet.next()) {
				sFeatureName = resultSet.getString(1);
				sFeatureValue = resultSet.getString(2);
				hFeatures.put(sFeatureName, sFeatureValue);
			}
			return hFeatures;
	}

	public static Hashtable loadDomainList()
	{
			Hashtable hFeatures = null;
			String sDomainListQuery =
				"SELECT DomainList.DomainAddress, Permissions.PermissionName From DomainList, Permissions  "
					+ " WHERE (DomainList.DomainID = Permissions.PermissionID)";

			try {
				// execute the Query
				ResultSet resultSet = sqlHelper.executeQuery(sDomainListQuery);
				hFeatures = getFeaturesFromResultSet(resultSet);
			} catch (Exception e) {
				e.printStackTrace();
				return null;
			}
			return hFeatures;
	}

	public static Hashtable loadUserDomainMapping()
	{
			Hashtable hFeatures = null;
			String sUserDomainQuery =
				"SELECT Users.UserName, Permissions.PermissionName "
				+ "FROM Users, Permissions, User_Permission_Xref "
				+ " WHERE ( "
				+ "(User_Permission_Xref.PermissionID = Permissions.PermissionID) AND "
				+ "(User_Permission_Xref.UserID = Users.UserID ) )";
			try {
				ResultSet resultSet = sqlHelper.executeQuery(sUserDomainQuery);
				hFeatures = getFeaturesFromResultSet(resultSet);
			} catch (Exception e) {
				e.printStackTrace();
				return null;
			}
			return hFeatures;
	}

			private static SQLHelper sqlHelper = null;
```

```
		public static ArrayList getListOfDomains() {
		String sListofDomainsQuery = "SELECT PermissionName From Permissions";

		try {
			ResultSet resultSet = sqlHelper.executeQuery(sListofDomainsQuery);
			boolean moreRecords = resultSet.next();
			if (!moreRecords) {
				return null;
			} else {
				ArrayList listOfDomains = new ArrayList();
				do {
					String domain = resultSet.getString("PermissionName");
					listOfDomains.add(domain);
				} while (resultSet.next());
				return listOfDomains;
			} //end else
		} catch (Exception e) {
			e.printStackTrace();
			return null;
		}
  }
}
```

FeedbackObject.java

```
/**
 * Object holding details about the feedback
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;
import java.util.*;

public class FeedbackObject implements Serializable
{
   private String queryID;
   private String domainName;
   private Hashtable fbProfile;

   public FeedbackObject(String qID, String domain)
   {
		queryID = qID;
		domainName = domain;
		fbProfile = new Hashtable();
   }

   public FeedbackObject(String qID, String domain, Hashtable profile)
   {
		queryID = qID;
		domainName = domain;
		fbProfile = new Hashtable();
   }

   public String getQueryID()
   {
		return queryID;
   }

   public String getDomainName()
   {
		return domainName;
   }

   public Hashtable getFBprofile()
   {
		return fbProfile;
```

```
    }

    public void addNewEntry(String hhName, FeedbackProfile fbp)
    {
                if(fbProfile.containsKey(hhName))
                {
                   FeedbackProfile alreadyInserted = (FeedbackProfile)  fbProfile.remove(hhName);
                   alreadyInserted.addReward(fbp.getReward());
                   fbProfile.put(hhName, alreadyInserted);
                }
                else
                {
                   fbProfile.put(hhName, fbp);
       }
    }
}
```

FeedbackProfile.java

```
/**
 * Component reward values in the feedback
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;

public class FeedbackProfile implements Serializable
{
    public float reward;
    public float pUpdate;

    public FeedbackProfile()
    {
                pUpdate = 0.0f;
    }

    public FeedbackProfile(float r)
    {
                reward = r;
                pUpdate = 0.0f;
    }

    public void addReward(float r)
    {
                reward+=r;
    }

    public float getReward()
    {
                return reward;
    }

    public float getPupdate()
    {
                return pUpdate;
    }
}
```

FeedbackThread.java

```
/**
 * Thread for propagated the feedback
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */
```

```
import java.net.*;
import java.security.*;
import java.util.*;
import java.lang.*;
import java.rmi.*;
import java.rmi.registry.*;

public class FeedbackThread implements Runnable
{
   private boolean feedbackSend = true;
   private IDomainSecurityManager dsm = null;
   private QueryManager queryManager = null;
   private long sleepTime = 1000;   // Time for which the thread sleeps

   public void run()
   {
              Thread CurrentThread = Thread.currentThread();
              FeedbackObject feedbackObject = null;

              while(true)
              {
                 try
                 {
                             // Process the next query in the queue
                             feedbackObject = queryManager.getNextFeedback();

                             if(feedbackObject == null)
                             {
                                CurrentThread.sleep(sleepTime);
                       }
                             else
                             {
                                String queryID = feedbackObject.getQueryID();
                                System.out.println("FBT: Processing feedback for query - " + queryID);
                                Hashtable fbProfile = feedbackObject.getFBprofile();
                                // Update QMs profile
                                queryManager.updateProfileInfo(fbProfile);

                                if(dsm!=null)
                                {
                                          // get the HH list from DSM
                                          ArrayList hhList = dsm.getHHListForDomain(feedbackObject.getDomainName());

                                          // Update all the HHs profile
                                          IHeadhunter ihh;
                                          String hhLocation = null;

                             for(int i=0; i<hhList.size(); i++)
                             {
                                             //hhLocation = null;
                                hhLocation = (String) hhList.get(i);
                                             if(hhLocation != null)
                                             {
                             try
                             {
                                ihh = (IHeadhunter) Naming.lookup(hhLocation);
                                                     if(ihh!=null)
                                                     {
                                                               ihh.updateProfileInfo(feedbackObject);
                                             }
                                                   }
                             catch(Exception ee)
                             {
                                System.out.println(ee);
                             }
                                }
                                             else
```

```
                                                {
                                                                System.out.println("FBT: hhLocation is NULL");
                                                }
                                        }
                                }
                                else
                                {
                                        System.out.println("#### ERROR in FeedbackThread - DSM is NULL");
                                }
                        }
                }
                catch(Exception e)
                {
                        e.printStackTrace();
                }
        }
    }

    public FeedbackThread(QueryManager qm, String dsmLocation)
    {
            try
            {
               queryManager = qm;
            }
            catch (Exception e)
            {
               System.out.println("#### ERROR in FeedbackThread constructor: " + e);
               e.printStackTrace();
               System.exit(1);
            }
        try
        {
           dsm = (IDomainSecurityManager) Naming.lookup(dsmLocation);
        }
        catch (Exception e)
        {
        }
    }
}
```

Headhunter.java

```
/**
 * Communicated periodically with Active Registry
 * Matches components and propagated the query
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 *
 */

import java.net.*;
import java.util.*;
import java.sql.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.security.*;
import java.io.*;
import java.lang.*;
import mss.ea.core.RandomUtil;

public class Headhunter extends UnicastRemoteObject implements IHeadhunter
{
   // New data structues for all the threads
   private LinkedList queryQueue = new LinkedList();
   private LinkedList feedbackQueue = new LinkedList();
   private Hashtable queryResults = new Hashtable();
   private Hashtable processingQuery = new Hashtable();
   private Hashtable processedQuery = new Hashtable();
   private PreferenceType preferenceType = null;
   private String hhLocation = null;
   private String qmLocation = null;
   private IDomainSecurityManager dsmanager = null;
   private IQueryManager iqm = null;
   private float profileValue;
   private static int hhForProfile = 0;
   private String domainName= null;
   private MRHelper mrHelper = null;
   private Hashtable propagatedhhHashtable = new Hashtable();
   private java.lang.String userType = "Headhunter";
   private Hashtable registryTable = new Hashtable();
   private ArrayList QIDList=new ArrayList();
   private String componentTableName=null;
   private long start_time,end_time;
   private long elapse_time;

   public boolean sendNewQuery(QueryQueueObject queryQueueObject, String querySourceType, String sourceLocation)
   {
            if(QIDList.contains(queryQueueObject.getQueryID()))
            {
               return false;
            }
            else
            {
               QIDList.add(queryQueueObject.getQueryID());
               queryQueue.addFirst(queryQueueObject);
            }
            return true;
   }

   public QueryQueueObject getNextQuery()
   {
            QueryQueueObject queryQueueObject = null;
            try
            {
               queryQueueObject = (QueryQueueObject) queryQueue.removeLast();
            }
      catch(NoSuchElementException nsee)
      {
         // No more elements in the Queue
      }
            return queryQueueObject;
   }


   public void addQueryBeingProcessed(String queryID, ProcessingQueryObject processingQueryObject)
   {
            processingQuery.put(queryID, processingQueryObject);
   }

   public ProcessingQueryObject getProcessingQueryFromHash(String queryID)
   {
            return ((ProcessingQueryObject) processingQuery.get(queryID));
   }

   public void storeProcessedQuery(String queryID, long respondedTime)
   {
            ProcessingQueryObject p = (ProcessingQueryObject) processingQuery.remove(queryID);

            ProcessedQueryObject pqo = new ProcessedQueryObject(respondedTime, System.currentTimeMillis(),
p.getQuerySourceName(),
p.getQuerySourceType(), p.getQuerySourceLocation());
            processedQuery.put(queryID, pqo);
   }
   // called when the HH did not propagate the query to any other HH
   public void storeInProcessedQuery(String queryID, long respondedTime, QueryQueueObject qqo)
   {
             ProcessedQueryObject pqo = new ProcessedQueryObject(respondedTime, System.currentTimeMillis(),
qqo.getQuerySourceName(),
qqo.getQuerySourceType(), qqo.getQuerySourceLocation());
             processedQuery.put(queryID, pqo);
   }

   public void addPropagatedhhArrayList(String queryID, ArrayList prophhArrayList)
   {
             propagatedhhHashtable.put(queryID, prophhArrayList);
   }

   public ArrayList getPropagatedHHArrayList(String queryID)
   {
             ArrayList propHHArrayList = null;
             if(propagatedhhHashtable.containsKey(queryID))
             {
                propHHArrayList = (ArrayList) propagatedhhHashtable.get(queryID);
             }
             else
             {
                System.out.println("#### ERROR - No entry found in  propagatedhhHashtable for query " + queryID);
             }
             return propHHArrayList;
   }

   public void sendNewFeedback(FeedbackObject feedbackObject)
   {
             feedbackQueue.addFirst(feedbackObject);
   }

   public FeedbackObject getNextFeedback()
   {
             FeedbackObject feedbackObject = null;
             try
             {
                feedbackObject = (FeedbackObject) feedbackQueue.removeLast();
             }
      catch(NoSuchElementException nsee)
      {
         // No more elements in the Queue
      }
             return feedbackObject;
   }

   public ArrayList getSelectedHH(String queryID, ArrayList fromQueryQueue, boolean randomFlag)
   {
             ArrayList propagatedHHList = null;
             if(iqm==null)
             {
         try
         {
           iqm = (IQueryManager) Naming.lookup(qmLocation);
         }
         catch(Exception eee)
         {
           System.out.println(eee);
         }
             }
      if(iqm!=null)
      {
         try
         {
           propagatedHHList = iqm.getPropagatedHHlist(queryID);
         }
         catch(Exception eeee)
```

```
        {
          System.out.println(eeee);
        }
    }

              if(propagatedHHList==null)
              {
                 propagatedHHList = new ArrayList();
              }

              if(fromQueryQueue!=null && fromQueryQueue.size()>0)
              {
                 for(int i=0; i<fromQueryQueue.size(); i++)
                 {
                            String hh = (String) fromQueryQueue.get(i);
                            if(!propagatedHHList.contains(hh))
                            {
                               propagatedHHList.add(hh);
                            }
                 }
              }

              int hhPropagateCount = 3;

              if(randomFlag)
              {
                 return (preferenceType.getHHListToPropagate(hhPropagateCount, queryID, propagatedHHList));
              }
              else
              {
                 return (preferenceType.getHHListToPropagateProfile(hhPropagateCount, queryID, propagatedHHList));
              }

  }

  public boolean checkHHprofileList(String domainName)
  {
              boolean flag = true;
    if(preferenceType==null)
    {
       try
       {
          ArrayList hhList = dsmanager.getHHListForDomain(domainName);
          if(hhList!=null && hhList.size() > 0)
          {
             flag = this.createHHList(hhList);
          }
          else
          {
             return false;
          }
       }
      catch(Exception eee)
      {
          System.out.println(eee);
          return false;
      }
    }
    return flag;
  }

  public void updateProfileInfo(FeedbackObject feedbackObject)
  {
              if(preferenceType==null)
              {
                 // Create Profile list before updating
       try
       {
            ArrayList hhList = dsmanager.getHHListForDomain(domainName);
            if(hhList!=null && hhList.size() > 0)
            {
               this.createHHList(hhList);
            }
            else
            {
               //return false;
            }
        }
        catch(Exception eee)
        {
            System.out.println(eee);
        }

             }

             if(preferenceType!=null)
             {
                // get the Hashtable and pass it to preferenceType
                preferenceType.updateProfileInfo(feedbackObject.getFBprofile(),
this.getPropagatedHHArrayList(feedbackObject.getQueryID()));
      }
             else
             {
                System.out.println("HH: ERROR creating profile list");
             }
   }

   public boolean createHHList(ArrayList hhList)
   {
             if(hhList.size()>0)
             {
                if(hhList.contains(hhLocation))
                {
                           // Remove this HH
                           hhList.remove(hhList.indexOf(hhLocation));
                }
             }

             // Update the HH list
             if(hhList.size()>0)
             {
                if(preferenceType == null)
                {
                           preferenceType = new PreferenceType(hhList);
                }
                else
                {
                           preferenceType.updateHHList(hhList);
                }
             }
             else
             {
                return false;
             }
             return true;
   }

   public void updateHHList(ArrayList hhList)
   {
             if(hhList.size()>0)
             {
                if(hhList.contains(hhLocation))
                {
                           // Remove this HH
                           hhList.remove(hhList.indexOf(hhLocation));
                }
            }

            // Update the HH list
            if(hhList.size()>0)
            {
               if(preferenceType == null)
               {
                           preferenceType = new PreferenceType(hhList);
               }
               else
               {
                           preferenceType.updateHHList(hhList);
               }
            }
}

public float getProfileValue()
{
            return profileValue;
}

/**
 * Sets the results of the query in the queryResults Hashtable
 */
public synchronized void setQueryResults(String queryID, Hashtable ratingResults, int propagatedHHcount)
{
            if(processedQuery.contains(queryID))
            {
            }
            else
            {
               // check whether any other HH have already set the results
               if(queryResults.get(queryID) == null)
               {
                           // Create a new result entry for this query
                           ResultObject resultObject = new ResultObject(ratingResults, propagatedHHcount);
                           resultObject.addNewComponentSet();          // update the propagatedCount as this MR results are updated

                           // set the updated Flag
                           resultObject.setUpdatedFlag();
                           queryResults.put(queryID, resultObject);
               }
               else
               {
                           // One of the Hhs already returned results for this query
                           ResultObject resultObject = (ResultObject) queryResults.remove(queryID);
                           resultObject.addNewComponentSet(ratingResults);

                           // update the propagatedHH count
                           resultObject.setPropagatedCount(propagatedHHcount);
                           queryResults.put(queryID, resultObject);
               }
   }
}

// to be called by other HHs
public synchronized void setQueryResults(String queryID, Hashtable results, String hhName)
{
            // Here results is ratingResults
            if(queryResults.get(queryID) == null)
            {
               if(processingQuery.containsKey(queryID))
               {
                           // Local HH did not enter results in queryResults
                           ResultObject resultObject = new ResultObject(results, 1); // assume propagatedHHcount is 1
                  resultObject.addNewComponentSet();   // update the propagatedCount as this MR results are updated
                           queryResults.put(queryID, resultObject);
               }
            }
            else
            {
               // combine the obtained results with the already available results
               ResultObject resultObject = (ResultObject) queryResults.remove (queryID);
               resultObject.addNewComponentSet(results);
               queryResults.put(queryID, resultObject);
            }
}

// called by other HH when there is no results
public synchronized void setQueryResults(String queryID, String hhName)
{
            if(queryResults.get(queryID) == null)
            {
               // check the processingQuery Hashtable
               if(processingQuery.containsKey(queryID))
               {
                        // Query entry not yet made in the queryResults
                        ResultObject resultObject = new ResultObject();
                        resultObject.addNewComponentSet();
                        queryResults.put(queryID, resultObject);;
               }
            }
            else
            {
               // combine the obtained results with the already available results
               ResultObject resultObject = (ResultObject) queryResults.remove (queryID);
               resultObject.addNewComponentSet();
               queryResults.put(queryID, resultObject);
            }
}

public Hashtable getQueryResults()
{
            return queryResults;
}

public void removeQueryResultEntry(String queryID)
{
   queryResults.remove(queryID);
}

public String getHHname()
{
            return componentTableName;
}


public Hashtable searchForComponent(QueryBean queryBean)
{
   Hashtable resultTable = null;
   long beforeMRaccess = System.currentTimeMillis();
   if(mrHelper == null)
   {
      System.out.println("#### Queue: MR still not initialized");
   }
   else
   {
      try
      {
         resultTable = mrHelper.getSearchResultTable(queryBean);                    // with component confidence
      }
      catch(Exception e)
      {
         resultTable.clear();
      }
   }
```

```
        return resultTable;
    }

    private void createMetaRepository(int num) throws Exception
    {
        mrHelper = new MRHelper(num);
    }

    public static void main(String[] args)
    {
        long mcastTime = 5000;
        String headhunterLocation ="//"+args[0]+":" +args[1]+"/HeadHunter";
        String dsmLocation="//magellan.cs.iupui.edu:"+args[2]+"/DomainSecurityManager";
        int mcastPort = 10000;
        String userName = args[4];
        String password = args[5];
        String domain = args[3];

        try
        {
            System.setSecurityManager(new RMISecurityManager());
            Naming.rebind(headhunterLocation,
                new Headhunter(mcastTime, mcastPort, userName, password, domain, headhunterLocation, dsmLocation));
            System.out.println("HeadHunter is ready");
            System.out.println("-------------------------------------");
        }
        catch (Exception e)
        {
            System.out.println("HeadHunter failed: " + e);
            e.printStackTrace();
        }

    }

    public Headhunter(long mcastTime, int mcastPort, String userName, String password, String domain, String headhunterLocation,
String dsmLocation) throws RemoteException
    {
        AuthenticatedPacket authpacket = null;
        System.out.println("\nHeadHunter activated at " + headhunterLocation);
        componentTableName=userName;
        domainName = domain;
        hhLocation = headhunterLocation;

        try
                {
            System.out.println("Headhunter Contacting DSM for Authentication.");

            dsmanager = (IDomainSecurityManager) Naming.lookup(dsmLocation);
            authpacket = dsmanager.authenticationService( userType, userName, password,
                        headhunterLocation, domain);
                System.out.println("Headhunter Authenticated by DSM");
        }
                catch(Exception e)
                {
                    e.printStackTrace();
                }

        try
        {
            qmLocation = dsmanager.getQMlocation();
        }
        catch(Exception eeee)
        {
            System.out.println(eeee);
        }

        try
                {
```

```
                createMetaRepository(ind);
                System.out.println("HH: MetaRepository Created");
            }
            catch (Exception e)
            {
        System.out.println("#### ERROR in HH (CreateMR) " + e.getMessage());
            }

            // Thread for MulticastSender
            try
            {
                MulticastSender mcastSender = new MulticastSender(mcastTime, mcastPort, authpacket, headhunterLocation);
                Thread senderThread = new Thread(mcastSender);
                senderThread.start();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }

    int threadCount = 1;

    // Thread for QueryQueueThread
    for(int i=0; i<threadCount; i++)
    {
        try
        {
            QueryQueueThread qqThreadObj = new QueryQueueThread(this, componentTableName, headhunterLocation,
qmLocation);
            qqThreadObj.setName(i+1);
            Thread queryQueueThread = new Thread(qqThreadObj);
            queryQueueThread.start();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

            // Thread for CombineResultThread
            try
            {
                CombineResultThread crThreadObj = new CombineResultThread(this);
                Thread combineResultThread = new Thread(crThreadObj);
                combineResultThread.start();
            }
            catch (Exception e)
            {
                e.printStackTrace();
            }

            // Thread for FeedbackThread
            try
            {
                FeedbackThread fbThreadObj = new FeedbackThread(this);
                Thread feedbackThread = new Thread(fbThreadObj);
                feedbackThread.start();
            }
            catch (Exception e)
            {
                System.out.println("#### ERROR in HH (FeedbackThread): " + e.getMessage());
                e.printStackTrace();
            }

  } //end of constructor


  /**
```

```
     * Functions used by the MulticastSenderThread
     *
     */
    private void populateMetaRepository(String regLoc)
    {
       try
       {
          System.setSecurityManager(new RMISecurityManager());
          IActiveRegistry Reg = (IActiveRegistry) Naming.lookup(regLoc.trim());
          Hashtable CompData = (Hashtable) Reg.getComponentData();
          if(CompData!=null)
          {
             // set the component info into the MR
             if(mrHelper!=null)
             {
                mrHelper.addNewComponentSet(CompData);
             }
             else
             {
                System.out.println("#### ERROR in HH - MR access object still NULL");
             }
          }
          else
          {
             System.out.println("HH: No Component found in AR $$$$");
          }
       }
       catch(Exception e)
       {
          System.out.println("#### HH: ERROR in Populate MR");
       }
    }

    public void receiveUnicastCommunication(String regLoc) throws RemoteException
    {
       registryTable.put(regLoc, (new java.util.Date()));
       populateMetaRepository(regLoc);
    }
 }
```

HHProfile.java

```
/*
 * Maintains P, D vectors used for HH profiling
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;

public class HHProfile implements Cloneable, Serializable
{
   public float P;    // Action probability vector
   public float D;    // Reward vector
   public int S;

   public HHProfile(float prob, float temp)   // for testing
   {
              P = prob;
              D = temp;
              S = 50;
   }

   public void updateValueHHadd(float factor)
   {
              this.P = (float) this.P * factor;
   }
   public void updateValueHHremove()
   {
             this.P = (float) (this.P / (1-this.P));
   }

   public float getPvalue()
   {
             return P;
   }

   public float getDvalue()
   {
             return D;
   }
}
```

HHProfileInfo.java

```
/*
 * Maintains profile information to rank the Headhunters
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;
import java.util.*;
import java.lang.*;
import uka.transport.*;

public class HHProfileInfo
{
   private Hashtable profile;

   public HHProfileInfo(ArrayList headhunters)
   {
      profile = new Hashtable();
      initialize(headhunters);
   }

   public void initialize(ArrayList headhunters)
   {
              int ind = 0;
      float probability = (float)1/headhunters.size();
      for(int i=0; i<headhunters.size(); i++)
      {
                String hhName = (String) headhunters.get(i);
                int number = hhName.indexOf(":");
                String port = hhName.substring(number+1,number+5);
         try
         {
            ind = (Integer.valueOf(port.substring(2,4))).intValue();
         }
         catch(Exception ee)
         {
         }

                float d = 0.5f;

         HHProfile hhProfile = new HHProfile(probability, d);
                profile.put(hhName, hhProfile);
             }
   }

   public synchronized void updateProfileValues(Hashtable feedback)
   {
             String pSelect = "P0";
             int dCase = 1; // 1-regular 2-discounting
             float lambda = 0.2f;
             float P0 = 0.0f;
             float P1 = 0.0f;
             Enumeration e;
             HHProfile hhProf = null;

             e = profile.keys();
             while(e.hasMoreElements())
             {
                String hhName = (String) e.nextElement();
                hhProf = (HHProfile) profile.get(hhName);               // remove before update
                if(feedback.containsKey(hhName))
                {
                         FeedbackProfile fbp = (FeedbackProfile) feedback.get(hhName);
                         if(dCase==1)
                         {
                            hhProf.D=( hhProf.S * hhProf.D + fbp.getReward() ) / (hhProf.S + 1) ;
                         }
                         else if (dCase==2)
                         {
                            hhProf.D = hhProf.D * 0.8f + fbp.getReward() * 0.2f;
                         }
                         hhProf.S += 1;
                         profile.remove(hhName);           // remove before update
                         profile.put(hhName, hhProf);      // add back in the Hashtable
                }
                else
                {
                         // update P value - no component returned by this HH
                         // reduce relative to existing P value. Here pFactor is P0 or P1
                         // (1/Po_num) represents the decrement value for each HH
                   // hhProf.P = hhProf.P - lambda * pFactor * pDecrement;

                         if(dCase==1)
                         {
                            hhProf.D = (hhProf.S * hhProf.D) / (hhProf.S + 1) ;
                         }
                         else if(dCase==2)
                         {
                            hhProf.D = hhProf.D * 0.8f;
                   }
                   hhProf.S += 1;
                }
             }
   }

   public Hashtable getHHProfileList(ArrayList propagatedHHList)
   {
             Hashtable tempProfile = new Hashtable();
             Enumeration e = profile.keys();
             while(e.hasMoreElements())
             {
                String hhName = (String) e.nextElement();
                if(propagatedHHList.contains(hhName))
                {
                         // Query already propagated to this HH
                         System.out.println("HHPI: Query already propagated to Headhunter " + hhName);
                }
                else
                {
                         // add in the tempProfile Hashtable
                         HHProfile hhProf = (HHProfile) profile.get(hhName);
                         tempProfile.put(hhName, hhProf);
                }
             }
             return tempProfile;
   }
```

```
}
```

IActiveRegistry.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */

import java.rmi.*;
import java.util.*;

public interface IActiveRegistry extends Remote
{
		public Hashtable getComponentData() throws RemoteException;
}
```

IComponent.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */

import java.rmi.*;

public interface IComponent extends Remote
{
		 public String getUmmSpecURL() throws RemoteException;
}
```

IDomainSecurityManager.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 * @modified: Barun Devaraju, Nov 2004
 */

import java.util.*;
import java.rmi.*;
f
public interface IDomainSecurityManager extends java.rmi.Remote
{
   public void printHashtables() throws RemoteException;
   public void setQMlocation(String qmLoc) throws RemoteException;
   public String getQMlocation() throws RemoteException;

   public java.util.Hashtable getHHListForDomainForProfile(String domainName) throws RemoteException;
   public ArrayList getHHListForDomain(String domainName) throws RemoteException;
   public ArrayList getARListForDomain(String domainName) throws RemoteException;
   public AuthenticatedPacket authenticationService(String userType, String userName, String password, String contactLocation,
String domain) throws RemoteException;
}
```

IHeadhunter.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 * @modified: Barun Devaraju, Nov 2004
 */

import java.rmi.*;
import java.util.*;
import java.security.*;
import java.io.*;

public interface IHeadhunter extends Remote
```

```
{
   public boolean sendNewQuery(QueryQueueObject qqo, String sourceType, String sourceLocation) throws RemoteException;
   public void sendNewFeedback(FeedbackObject fbo) throws RemoteException;
   public void setQueryResults(String queryID, Hashtable results, String hhName) throws RemoteException;
   public void setQueryResults(String queryID, String hhName) throws RemoteException;
   public void updateProfileInfo(FeedbackObject fb) throws RemoteException;
   public String getHHname() throws RemoteException;
   public void receiveUnicastCommunication(String regLoc) throws RemoteException;
}
```

IQueryManager.java

```
/**
 * @author: Nanditha Nayani
 * @date: Aug 2001
 * @modified: Barun Devaraju, Nov 2004
 */

import java.rmi.*;
import java.util.*;

public interface IQueryManager extends Remote
{
   public void getSearchResultTable(QueryBean qBean, long responseTime, String querySourceLocation) throws RemoteException;
   public void setQueryResults(String queryID, Hashtable results, String hhName) throws RemoteException;
   public void setQueryResults(String queryID, String hhName) throws RemoteException;
   public void sendNewFeedback(FeedbackObject feedbackObject) throws RemoteException;
   public void printResponseTimes(int status) throws RemoteException;
   public ArrayList getPropagatedHHlist(String queryID) throws RemoteException;
   public void updatePropagatedHHlistFromHH(String QID, ArrayList propHHlist) throws RemoteException;
}
```

IURDS_proxy.java

```
/**
 * @author: Zhisheng Huang
 * @date: January 2003
 * @modified: Barun Devaraju, Nov 2004
 */

import java.rmi.*;
import java.util.*;

public interface IURDS_Proxy extends Remote
{
   public int obtainResults(String queryID, Hashtable components) throws RemoteException;
}
```

MRHelper.java

```
/**
 * Acts as an MR
 * Stores all component details from AR
 *
 * @author: Barun Devaraju
 * @date: Dec 2004
 */

import java.io.*;
import java.util.*;
import mss.ea.core.RandomUtil;

public class MRHelper
{
   private Hashtable componentList = null;
   private FileWriter out = null;
   private FileReader fin = null;

   public MRHelper(int hhNum)
```

```
{
        String hhNumStr = "";
        componentList = new Hashtable();

        if(hhNum<=9)
        {
          hhNumStr = "0" + String.valueOf(hhNum);
        }
        else
        {
          hhNumStr = String.valueOf(hhNum);
        }

        String fileName = "MR-" + hhNumStr + ".txt";

        loadWithSameComponents(fileName);

        File fileMR = new File(fileName);
        try
        {
          out = new FileWriter(fileMR, false);
          System.out.println("MR: File " + fileName + " created");
        }
        catch(Exception e)
        {
          System.out.println("#### MR: ERROR - File Creation");
        }
        loadMR(hhNumStr);
}


public void loadMR(String hhNumStr)
{
        ArrayList compNames = new ArrayList();
        compNames.add("documentDatabase");
        compNames.add("DocumentServer");
        compNames.add("DocumentTerminal");

        // add random number of components
        RandomUtil rand = new RandomUtil();
        int r=0;

        r = rand.randomInt(1, 100);

        int totalSize = compNames.size();
        int limit = 50;

        for(int i=1; i<=r; i++)
        {
          int cNum = rand.randomInt(1,compNames.size());
          String componentID = hhNumStr + "-" + i;
          String componentName = (String) compNames.get(cNum-1);
          ConcreteComponent cc = new ConcreteComponent();
          cc.setDomainName("Document");
          cc.setSystemName("DocumentManager");
          cc.setComponentName(componentName);

          int qLevel = rand.randomInt(1,10);
          cc.setQualityLevel(String.valueOf(qLevel));

          int cost = rand.randomInt( ((qLevel*5)-5), ((qLevel*5)+5) );
          cc.setCost(String.valueOf(cost*10));

          componentList.put(componentID, cc);
          try
          {
                out.write(componentID + "\t" + componentName + "\t" + String.valueOf(cost*10) + "\t" + qLevel + "\n");
          }
```

```
                catch(Exception fw)
                {
                        System.out.println("#### MR: ERROR - file write " + componentID);
                }
            }
}

public Hashtable getSearchResultTable(QueryBean queryBean) throws java.lang.Exception
{
            Hashtable matchedResults = null;
            if(componentList.size()==0)
            {
               System.out.println("MR: No components available in MR");
            }
   else
   {
      matchedResults = new Hashtable();
      Enumeration e = componentList.keys();
      while(e.hasMoreElements())
      {
         String componentID = (String) e.nextElement();
         ConcreteComponent cc = (ConcreteComponent) componentList.get(componentID);
         boolean matched = false;

                        if(cc.getDomainName().compareTo(queryBean.getDomain())==0)
                        {
                           matched = true;
                           //System.out.println("--- MR: Domain name matched");
                        }
                        else
                        {
                           matched = false;
                           //System.out.println("MR: Domain name did not match");
                        }

                        // check other attributes
                        if(matched)
                        {
                           if( cc.getSystemName().compareTo(queryBean.getAbstractComponent().getSystemName())==0 )
                           {
                                   // SystemName also matched
                           }
                           else
                           {
                                   matched = false;
                           }
                        }

                        if(matched)
                        {
                           if( cc.getComponentName().compareTo(queryBean.getAbstractComponent().getComponentName())==0 )
                           {
                                   // SystemName also matched
                           }
                           else
                           {
                                   matched = false;
                           }
                        }

                        if(matched)
                        {

                           float rating = 0.0f;
                           int queryQualityLevel = Integer.parseInt(queryBean.getAbstractComponent().getQualityLevel());
                           int compQualityLevel = Integer.parseInt(cc.getQualityLevel());
                           int diffQualityLevel = queryQualityLevel - compQualityLevel;
                           if(diffQualityLevel<0)
```

```
                              diffQualityLevel = -diffQualityLevel;

                          // assign component confidence for Quality level
                          rating =  rating + (10-diffQualityLevel) * 0.06f;         // will be between .05 and .5

                        int compCost = Integer.parseInt(cc.getCost());
                        int estimatedCost = compQualityLevel * 50;
                        int diffCost = estimatedCost - compCost;

                          if(diffQualityLevel==0)
                          {
                                  if(diffCost >= 0)
                                     rating = rating +  0.4f ;
                                  else
                                     rating = rating +  0.36f ;
                          }
                          else if(diffQualityLevel <=5)
                          {
                                  if(diffCost > 100 || diffCost <-100)
                                     rating = rating +  0.25f ;
                                  else
                                     rating = rating +  0.3f ;
                          }
                          else
                          {
                                  if(diffCost > 400 || diffCost <-400)
                                     rating = rating +  0.1f ;
                                  else
                                     rating = rating +  0.2f ;
                          }
                ComponentDetailsObject cdo = new ComponentDetailsObject(cc,rating);
                cdo.setTime(System.currentTimeMillis());
                matchedResults.put(componentID, cdo);
              }
            }
          }
          return matchedResults;
        }

        public void loadWithSameComponents(String fileName)
        {
          File ipFile = new File(fileName);
                BufferedReader br = null;
                String line = new String();

          try
          {
            fin = new FileReader(ipFile);
                  br = new BufferedReader(fin);

                  while ((line = br.readLine()) != null)
                  {
                          String compID = "";
                          String componentName = "";
                          String cost = "";
                          String qualityLevel = "";

                ConcreteComponent cc = new ConcreteComponent();
                cc.setDomainName("Document");
                cc.setSystemName("DocumentManager");

                          StringTokenizer parser = new StringTokenizer(line);
                          if(parser.hasMoreTokens())
                          {
                             compID = parser.nextToken();
                          }

                          if(compID!=null && compID.compareTo("")!=0)
```

```
                            {

                            if(parser.hasMoreTokens())
                            {
                               componentName = parser.nextToken();
                            }
                            if(parser.hasMoreTokens())
                            {
                               cost = parser.nextToken();
                            }
                            if(parser.hasMoreTokens())
                            {
                               qualityLevel = parser.nextToken();
                            }

               cc.setComponentName(componentName);
                            cc.setCost(cost);
               cc.setQualityLevel(qualityLevel);

               componentList.put(compID, cc);
                     System.out.println("MR: Added (f) - " + compID + " " + componentName +"\t"+cost +" " + qualityLevel);

                            }
                  }
                  br.close();
                  fin.close();
        }
        catch(Exception fr)
        {
           System.out.println("#### ERROR - Read File pointer error");
        }
     }
  }
```

MulticastReceiver.java

```
/**
 * Receives the Multicast messages from Headhunter
 *
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */

import java.net.*;
import java.io.*;
import java.util.*;
import java.rmi.*;
import java.security.*;

public class MulticastReceiver implements Runnable
{

   private InetAddress groupAddress = null;
   private MulticastSocket multicastSocket = null;
   private int port = 10000;
   private java.lang.String registryLocation;

   public void run()
   {
               try
               {
                  multicastSocket = new MulticastSocket(port);
                  multicastSocket.joinGroup(groupAddress);
                  System.out.println("\n Active Registry joined multicast group at : " + groupAddress);

                  while (true)
                  {
                            byte[] buffer = new byte[1024];
```

```
                        //Receiving data
                        DatagramPacket dataPacket = new DatagramPacket(buffer, buffer.length);
                        multicastSocket.receive(dataPacket);
            //----      Without Encryption ------------
            String dataString = new String(dataPacket.getData());

              System.out.println("Active Registry Received Encrypted Multicasted Headhunter Location : " + dataString);
                        try
                        {
                           System.setSecurityManager(new RMISecurityManager());
                           IHeadhunter hhunter = (IHeadhunter) Naming.lookup(dataString.trim());
                           hhunter.receiveUnicastCommunication(registryLocation);
                           System.out.println( "Active Registry Unicast to Headhunter " +
                                                            dataString + " its Location " + registryLocation);
                        }
                        catch (Exception e)
                        {
                           System.out.println(e);
                        }
                 }
              }
              catch (IOException ioe)
              {
                 System.out.println(ioe);
              }
              finally
              {
                 if (multicastSocket != null)
                 {
                        try
                        {
                           multicastSocket.leaveGroup(groupAddress);
                           multicastSocket.close();
                        }
                        catch (IOException ioe)
                        {
                           System.out.println(ioe);
                        }
                 }
              }
   }

public MulticastReceiver(int mcastPort, AuthenticatedPacket authPacket,
                        String regLocation)
   {
              try
              {
                 groupAddress = InetAddress.getByName(authPacket.getMCAddress());
                 port = mcastPort;
                 registryLocation = regLocation;
                 cryptObject.enCryptObj(registryLocation);
              }
              catch (Exception e)
              {
                 System.out.println(e);
                 System.exit(1);
              }
   }
}
```

MulticastSender.java

```
/**
 * Send multicast messages to Active Registry
 *
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */
```

```
import java.net.*;
import java.security.*;

public class MulticastSender implements Runnable
{
   private InetAddress inetAddress = null;
   private DatagramPacket dataPacket = null;
   private int port = 10000;
   private byte ttl = (byte) 1;
   private String hostLocation = "";
   private byte[] buffer;
   private MulticastSocket multicastSocket = null;
   private long TPmcast;

   public void run()
   {
          Thread CurrentThread = Thread.currentThread();
          try
          {
             while(true)
             {
                   dataPacket = new DatagramPacket(buffer,buffer.length,inetAddress,port);
                   multicastSocket.send(dataPacket,ttl);
                   CurrentThread.sleep(TPmcast);
             }
          }
          catch(InterruptedException ie)
          {
             System.out.println("in InterruptedException");
             System.out.println(ie.getMessage());
             ie.printStackTrace();
          }
          catch(SocketException se)
          {
             System.out.println("in SocketException");
             System.out.println(se);
             se.printStackTrace();
          }
          catch(Exception e)
          {
             System.out.println("in Exception");
             System.out.println(e);
             e.printStackTrace();
          }
   }

   public MulticastSender(long mcastTime, int mcastPort, AuthenticatedPacket authPacket,
          String headHunterLoc)
   {
          try
          {
             TPmcast = mcastTime;
             System.out.println("------ Auth Packet: "+ authPacket.getMCAddress());
             inetAddress = InetAddress.getByName(authPacket.getMCAddress());
             port = mcastPort;
             hostLocation = headHunterLoc;
             buffer = hostLocation.getBytes();
             multicastSocket = new MulticastSocket();
             System.out.println("HH joining Group "+ inetAddress);
             multicastSocket.joinGroup(inetAddress);
             System.out.println("Headhunter joined multicast group:" + inetAddress);
          }
          catch (SocketException se)
          {
             System.out.println(se);
             se.printStackTrace();
          }
```

```
                catch (Exception e)
                {
                    System.out.println(e);
                    e.printStackTrace();
                    System.exit(1);
                }
    }
}
```

PreferenceType.java

```
/**
 * Maintains the preference types of the Query
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.util.*;
import java.lang.*;
import mss.ea.core.RandomUtil;

public class PreferenceType
{
    private String type;
    private HHProfileInfo hhProfileInfo;

    public PreferenceType(ArrayList headhunters)
    {
                hhProfileInfo = new HHProfileInfo(headhunters);
    }

    public void updateHHList(ArrayList hhList)
    {
                hhProfileInfo.updateHHList(hhList);
    }

    public void getProfileStatus()
    {
        hhProfileInfo.printHHprofileValues();
    }

    public void updateProfileInfo(Hashtable feedback, ArrayList propagatedHHArrayList)
    {
                hhProfileInfo.updateProfileValues(feedback);
    }

    public ArrayList getHHListToPropagate(int num, String queryID, ArrayList propagatedHHList)
    {
        int bufferHH = 2;       // number of extra HHs returned
        ArrayList selectedHH = new ArrayList();  // to store the selected Headhunters
        Hashtable hhList = hhProfileInfo.getHHProfileList(propagatedHHList);
        RandomUtil rand = new RandomUtil();
        if(hhList.size() <=0)
        {
            System.out.println("PT: No Headhunters available for query propagation");
        }
        else if(hhList.size() <= num+bufferHH)
        {
            System.out.println("PT: Number of HHs available - " + hhList.size());
            Enumeration e = hhList.keys();
            while(e.hasMoreElements())
            {
                selectedHH.add((String)e.nextElement());
            }
                }
                else
                {
```

```
                ArrayList tempSelectedHH = new ArrayList();
            Enumeration e = hhList.keys();
            while(e.hasMoreElements())
            {
               tempSelectedHH.add((String)e.nextElement());
            }
                 System.out.println("PT: HHs selected for query propagation are ... ");
            for(int i=0; i< num+bufferHH; i++)
            {
               int sel = rand.randomInt(1, tempSelectedHH.size());
                        String hhSel = (String) tempSelectedHH.remove(sel-1);
                        selectedHH.add(hhSel);
                        System.out.print(hhSel + "  ");
            }
                System.out.println();
            }
            return selectedHH;
    }

    public ArrayList getHHListToPropagateProfile(int num, String queryID, ArrayList propagatedHHList)
    {
            int bufferHH = 2;       // number of extra HHs returned
            int randomHHnum = 0;                            // Number of HHs selected randomly
            int dHHnum = num - randomHHnum;          // Number of HH selected using D vector
            ArrayList selectedHH = new ArrayList();  // to store the selected Headhunters
            Hashtable hhList = hhProfileInfo.getHHProfileList(propagatedHHList);

            if(hhList.size() <=0)
            {
                System.out.println("PT: No Headhunters available for query propagation");
            }
            else if(hhList.size() <= num+bufferHH)
            {
                System.out.println("PT: Number of HHs available - " + hhList.size());
                Enumeration e = hhList.keys();
                while(e.hasMoreElements())
                {
                        selectedHH.add((String)e.nextElement());
                }
            }
            else
            {
                System.out.println("PT: Number of HHs available - " + hhList.size());
        String hhName=null;
        float[] pVector = new float[hhList.size()];
        float[] dVector = new float[hhList.size()];
        String[] hhNameList = new String[hhList.size()];    // List of Headhunters
        int[] selectedIndexList = new int[hhList.size()];   // Index to record the selected Headhunters
        int selectedIndex=0;
                int vectorIndex = 0;

        // initialize the vectors
        Enumeration e = hhList.keys();
        while(e.hasMoreElements())
        {
            hhNameList[vectorIndex] = (String) e.nextElement();
            HHProfile hhProf = (HHProfile) hhList.get(hhNameList[vectorIndex]);
            dVector[vectorIndex] = (float) hhProf.D;
            pVector[vectorIndex] = (float) hhProf.P;
            //System.out.println("PT: " + hhNameList[vectorIndex] + ":\t" + pVector[vectorIndex]+ "\t" + dVector[vectorIndex]);
            vectorIndex++;
                }

        // Select best n-2 Headhunters using D vector
        float max;
        int maxIndex;
                 System.out.println("PT: HHs selected thru D vector are ... ");
        for(int i=0 ; i<dHHnum+bufferHH ; i++)
```

```
          {
                        max = 0.0f;
                        maxIndex = -1;
            Random r = new Random();
              for(int j=0; j<hhList.size(); j++)
              {
               if(dVector[j] > max && selectedIndexList[j]!=1)
                {
                        max = dVector[j];
                        maxIndex = j;
                }
               else if(dVector[j] == max && selectedIndexList[j]!=1)
                {
                   float n = r.nextFloat();
                   if(n<=0.9)
                   {
                      // don't change the max
                   }
                   else
                   {
                      max = dVector[j];
                      maxIndex = j;
                   }
                }
             }
                        if(maxIndex>=0)
                        {
                 // insert max D vector in to selected HH list
                 selectedHH.add(hhNameList[maxIndex]);
                 selectedIndexList[maxIndex] = 1;   // Mark that a Headhunter is selected
              }

          } // end FOR loop for D vector
               } // end of else
        return selectedHH;
    }
}
```

ProcessedQueryObject.java

```
/**
 * Data for processed queue
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;

public class ProcessedQueryObject implements Serializable
{
   private long queryRespondedTime;
   private long queryReturnedTime;
   private String querySourceName;
   private String querySourceType;
   private String querySourceLocation;

   public ProcessedQueryObject(long respondedTime, long returnedTime, String sourceName, String sourceType, String
sourceLocation)
   {
              queryRespondedTime = respondedTime;
              queryReturnedTime = returnedTime;
              querySourceName = sourceName;
              querySourceType = sourceType;
              querySourceLocation = sourceLocation;
   }

   public long getRespondedTime()
```

```
    {
              return queryRespondedTime;
    }

    public long getReturnedTime()
    {
              return queryReturnedTime;
    }

    public String getQuerySourceName()
    {
              return querySourceName;
    }

    public String getQuerySourceType()
    {
              return querySourceType;
    }

    public String getQuerySourceLocation()
    {
              return querySourceLocation;
    }
}
```

ProcessingQueryObject.java

```
/**
 * Data for processing query
 *
 * @author: Barun Devaraju
 * @date: Noc 2004
 */

import java.io.*;

public class ProcessingQueryObject implements Serializable
{
    private long queryResponseTime;
    private long queryInsertionTime;
    private String querySourceName;
    private String querySourceType;
    private String querySourceLocation;

    public ProcessingQueryObject(long insertionTime, long responseTime, String sourceName, String sourceType, String
sourceLocation)
    {
              queryInsertionTime = insertionTime;
              queryResponseTime = responseTime;
              querySourceName = sourceName;
              querySourceType = sourceType;
              querySourceLocation = sourceLocation;
    }

    public long getResponseTime()
    {
              return queryResponseTime;
    }

    public long getInsertionTime()
    {
              return queryInsertionTime;
    }

    public String getQuerySourceName()
    {
              return querySourceName;
    }
```

```
   public String getQuerySourceType()
   {
              return querySourceType;
   }

   public String getQuerySourceLocation()
   {
              return querySourceLocation;
   }
}
```

QM_CombineResultThread.java

```
/**
 * This Thread is spawned from Query Manager
 * Combines the results obtained from different HHs and send them to the requestor
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.net.*;
import java.security.*;
import java.util.*;
import java.lang.*;
import java.rmi.*;
import java.rmi.registry.*;

public class QM_CombineResultThread implements Runnable
{
   private IURDS_Proxy iurds = null;
   private QueryManager qm = null;
   private long sleepTime = 1000;
   private long thresholdTime = 1000;

   public void run()
   {
              Thread CurrentThread = Thread.currentThread();
              Hashtable queryResults;
              while(true)
              {
                 try
                 {
                              queryResults = qm.getQueryResults();
                              ProcessingQueryObject processingQuery;
                              boolean enterWhileFlag = false;
              Enumeration e = queryResults.keys();
              while(e.hasMoreElements())
              {
                                 enterWhileFlag = true;
                                 long expiredTime = 0;
                                 long remainingResponseTime = 0;
                 String queryID = (String) e.nextElement();
                 ResultObject resultObject = (ResultObject) queryResults.get(queryID);
                                 processingQuery = qm.getProcessingQueryFromHash(queryID);
                                 if(processingQuery==null)
                                 {
                                              System.out.println("Combine: Query " + queryID + " not being added in the processingQuery
Hashtable");
                                 }
                                 else
                                 {
                                              expiredTime = System.currentTimeMillis() - processingQuery.getInsertionTime();
                                              remainingResponseTime = processingQuery.getResponseTime() - expiredTime;
                                 }

                                 // prints the number HHs to which the query is propgated to
```

```
                        resultObject.checkResultsObtained(queryID);

                        if(remainingResponseTime<thresholdTime)
                        {
                        }

                if(resultObject.isAllResultsObtainedFromQM() || remainingResponseTime < thresholdTime)
                {
                                int finished = -2;
                                String urdsLocation = processingQuery.getQuerySourceLocation();

                                // return results to URDS_Proxy
                                try
                                {
                                   iurds = (IURDS_Proxy) Naming.lookup(urdsLocation);
                                   if(iurds == null)
                                   {
                                          System.out.println("#### Combine: URDS_Proxy object is NULL after lookup in " +
urdsLocation);
                                   }
                                   else
                                }
                                catch(Exception ee)
                                {
                                   System.out.println(ee);
                                }

                                Hashtable matchedComp = null;
                                try
                                {
                                   matchedComp =  resultObject.getMatchedComponents();
                                }
                                catch(Exception me)
                                {
                                   me.printStackTrace();
                                   matchedComp = null;
                                }

                                try
                                {
                                   if(matchedComp!=null && matchedComp.size()>0)
                                   {
                                          // Update the response time with the query results
                                          qm.updateResponseTime(queryID);
                                finished = iurds.obtainResults(queryID, matchedComp);
                                   }
                                   else
                                   {
                                          matchedComp = new Hashtable();
                                finished = iurds.obtainResults(queryID, matchedComp);
                                   }

                                }
                                catch(Exception eeee)
                                {
                                   System.out.println(eeee);
                                   eeee.printStackTrace();
                                }
                                long timeToProcess =  System.currentTimeMillis() - processingQuery.getInsertionTime();

                                // remove the results of this query
                                qm.removeQueryResultEntry(queryID);
                                qm.storeProcessedQuery(queryID, timeToProcess);

                                if(finished==0)
                                {
                                   // Print the response times
                                   qm.printTime(queryID, matchedComp, false);
```

```
                                        }

                                        if(finished==1)
                                        {
                                           qm.printResponseTimes(0);
                                        }
                                        if(finished==2)
                                        {
                                           qm.printTime(queryID, matchedComp, true);
                                           qm.printResponseTimes(1);
                                        }

                        }
                    }
                          CurrentThread.sleep(sleepTime);
                       }
                       catch(Exception e)
                       {
                                  e.printStackTrace();
                       }
                    }
   }

   public QM_CombineResultThread(QueryManager qMan)
   {
              try
              {
                 qm = qMan;
              }
              catch (Exception e)
              {
                 e.printStackTrace();
                 System.exit(1);
              }

   }
}
```

QM_QueryQueueThread.java

```
/**
 * Spawned by the Query Manager
 * This Thread takes each query from the QueryQueue and processes them
 * The QueryQueue has QueryQueue objects, stored as a queue
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.net.*;
import java.security.*;
import java.util.*;
import java.lang.*;
import uka.transport.*;
import java.rmi.*;
import java.rmi.registry.*;

public class QM_QueryQueueThread implements Runnable
{
   private QueryManager qm = null;
   private String qmLocation = null;
   private long profileAccessTime = 50;
   private long combineComponentsTime = 100;
   private long expectedHHcompReturnTime = 500;
   private long sleepTime = 800;
   private ArrayList propagatedHHArrayList = new ArrayList();

   public void run()
```

```
{
        Thread CurrentThread = Thread.currentThread();
        QueryQueueObject queryQueueObject = null;
        QueryQueueObject queryQueueObjectToSend = null;
        boolean propagateFlag = true;

        while(true)
        {
    beginThread = System.currentTimeMillis();
          try
          {
                queryQueueObject = qm.getNextQuery();
                if(queryQueueObject == null)
                {
                  CurrentThread.sleep(sleepTime);
                }
                else
                {
                  String queryID = queryQueueObject.getQueryID();
                  long waitTime = System.currentTimeMillis() - queryQueueObject.getInsertionTime();
                  long remainingResponseTime = queryQueueObject.getResponseTime() - waitTime;
                  remainingResponseTime = remainingResponseTime - profileAccessTime - combineComponentsTime;
                  int propagatedHHcount=0;

                  if(remainingResponseTime > expectedHHcompReturnTime)
                  {
                beginHHselect = System.currentTimeMillis();
                         String hhLocation = null;
                         IHeadhunter ihh;

           // create HH profile List, of not created
           propagateFlag = qm.checkHHprofileList(queryQueueObject.getQueryBean().getDomain());

           if(propagateFlag)
           {
              ArrayList hhList = qm.getSelectedHH(queryQueueObject.getQueryID()
,queryQueueObject.getHHpropagatedList());
                           queryQueueObjectToSend = queryQueueObject.cloneObject(remainingResponseTime,
qmLocation, "QM", "QM-1");

             queryQueueObjectToSend.updateHHpropagateList(new ArrayList());
                        // update in the QM's propagated HH list
                           qm.updatePropagatedHHlistFromQM(queryID, hhList);
                           boolean sentFlag = false;

                  ProcessingQueryObject processingQueryObject = new ProcessingQueryObject(System.currentTimeMillis(),
remainingResponseTime, queryQueueObject.getQuerySourceName(), queryQueueObject.getQuerySourceType(),
queryQueueObject.getQuerySourceLocation());
                  qm.addQueryBeingProcessed(queryQueueObject.getQueryID(), processingQueryObject, hhList.size());

                         propagatedHHArrayList.clear();
                           // Send the query to the selected HHs
                           for(int i=0; i<hhList.size(); i++)
                           {
                             hhLocation = (String) hhList.get(i);
                             try
                             {
                                     ihh = (IHeadhunter) Naming.lookup(hhLocation);
                  sentFlag = ihh .sendNewQuery(queryQueueObjectToSend, "QM", qmLocation);
                                     if(sentFlag)
                                     {
                                       propagatedHHcount++;
                                            propagatedHHArrayList.add(hhLocation);
                                     }
                             }
                             catch(Exception ee)
                             {
                  System.out.println(ee);
```

```
                                                }
                                              }

                                            // update the propagatedHHArrayList in the QueryManager
                                            qm.addPropagatedHHArrayList(queryID, propagatedHHArrayList);
                } // end of flag check
                else
                {
                   System.out.println("No HHs available for Query propagation");
                }
                              }
                              else
                              {
                                         // calculate back the remaining response time
                                         remainingResponseTime = remainingResponseTime + profileAccessTime;
                              }
                   }
                }
                catch(Exception e)
                {
                           e.printStackTrace();
                }
              }
   }

   public QM_QueryQueueThread(QueryManager qMan, String location)
   {
              try
              {
                 qm = qMan;
                 qmLocation = location;
              }
              catch (Exception e)
              {
                 e.printStackTrace();
                 System.exit(1);
              }

   }
}
```

QueryBean.java

```
/**
 * Details for the query
 *
 * @author: Nanditha Nayani, modified by Zhisheng in March 2003
 * @modified: Barun Devaraju, Nov 2004
 */

import java.util.*;
import java.io.*;

public class QueryBean implements Serializable
{
   private AbstractComponent component; //added March 2003
   private int numOffers = 0;
   private int numMetrics = 0;

   public QueryBean(AbstractComponent component)
   {
              this.component = component;
   }

   public String getComponentNameQuery()
   {
              String searchQuery = " Componentname =" + "'"+ component.getComponentName() + "'";
              return searchQuery;
   }

   public String getSystemNameQuery()
   {
              String systemNameQuery = "(" + " UPPER(systemName) = '" + Component.getSystemName() + "' "+ ")";
              return systemNameQuery;
   }

   public String getSubcaseQuery()
   {
              String subcaseQuery =" subcase = '" +component.getSubcase() + "' ";
              return subcaseQuery;
   }

   public String getIDQuery()
   {
              String idQuery = "(" + " UPPER(hostid) = " + component.getID() + "' "+ ")";
              return idQuery;
   }

   public String getVersionQuery()
   {
              String VersionQuery="("+" UPPER(Version) = "+component.getVersion()+"' "+ ")";
              return VersionQuery;
   }

   public String getAuthorQuery()
   {
              String AuthorQuery = "("+" UPPER(Author) = "+component.getAuthor()+"' "+ ")";
              return AuthorQuery;
   }

   public String getDateQuery()
   {
              String DateQuery = "(" + " UPPER(CreatingDate) = " + component.getDate() + "' "+ ")";
              return DateQuery;
   }

   public String getValidityQuery()
   {
              String ValidityQuery =          "(" + " UPPER(Validity) = " + component.getValidity() + "' "+ ")";
              return ValidityQuery;
   }

   public String getAtomicityQuery()
   {
              String AtomicityQuery ="(" +" UPPER(Atomicity) = " + component.getAtomicity() + "' "+ ")";
              return AtomicityQuery;
   }

   public String getRegistrationQuery()
   {
              String RegistrationQuery ="(" +" UPPER(Registration) = " + component.getRegistration() + "' "+")";
              return RegistrationQuery;
   }

   public String getModelQuery()
   {
              String ModelQuery ="(" +" UPPER(Model) = " + component.getModel() + "' "+")";
              return ModelQuery;
   }

   public String getComplexityQuery()
   {
              String ComplexityQuery ="(" +" UPPER(Complexity) = " + component.getComplexity() + "' "+")";
              return ComplexityQuery;
   }
```

```
public String getSecurityQuery()
{
        String SecurityQuery ="(" +" UPPER(Security) = " + component.getSecurity() + "' "+")";
        return SecurityQuery;
}

public String getFaultToleranceQuery()
{
        String FaultToleranceQuery ="(" +" UPPER(FaultTolerance) = " + component.getFaultTolerance() + "' "+")";
        return FaultToleranceQuery;
}

public String getQoSLevelQuery()
{
        String QoSLevelQuery ="(" +" UPPER(QoSLevel) = " + component.getQoSLevel() + "' "+")";
        return QoSLevelQuery;
}

public String getCostQuery()
{
        String CostQuery ="(" +" UPPER(Cost) = " + component.getCost() + "' "+")";
        return CostQuery;
}

public String getQualityLevelQuery()
{
        String QualityLevelQuery ="(" +" UPPER(QualityLevel) = " + component.getQualityLevel() + "' "+")";
        return QualityLevelQuery;
}

public String getDomain()
{
        return component.getDomainName();
}

public AbstractComponent getAbstractComponent()
{
        return component;
}

public String getMobilityQuery()
{
        String mobilityQuery ="(" +" UPPER(MOBILITY) LIKE '%" + component.getMobility().toUpperCase() + "%' "+
        ")";
        return mobilityQuery;
}

public void setNumMetrics(int newNumMetrics)
{
        numMetrics = newNumMetrics;
}

public void setNumOffers(int newNumOffers)
{
        numOffers = newNumOffers;
}

public String[] tokeniseString(String keyWords)
{
        String[] stringTokens = null;
        if (keyWords != null)
        {
                StringTokenizer strTok = new StringTokenizer(keyWords);
                int numTokens = strTok.countTokens();
                stringTokens = new String[numTokens];
                int i = 0;
                while (strTok.hasMoreTokens())
                {
```

```
                                        stringTokens[i] = strTok.nextToken();
                                        i++;
                                }
                        }
                        return stringTokens;
        }

        public String getQuery(String componentTableName)
        {
                        ComponentQoS qoscomponent = component.getComponentQoS();
                        FunctionQoS qosfunction;
                        String query = "";
                        String baseQuery = "SELECT ID FROM " + componentTableName + "UMMSpecification where";

                        if((component.getDomainName() != null) && (!component.getDomainName().equals("")))
                        {
                           query = query + "  Domainname = '" + component.getDomainName() + "' ";
                        }
                        if ((component.getComponentName() !=null) && (!component.getComponentName().equals("")))
                        {
                           query = query + " AND componentname = '" + component.getComponentName() + "' ";
                        }
                        if((component.getSubcase() != null) && (!component.getSubcase().equals("")))
                        {
                           query = query + " AND " + getSubcaseQuery();
                        }
                        if(qoscomponent !=null)
                        {
                           query = query + " Intersect ";
                           Hashtable qoscomponenttable = qoscomponent.getComponentQoS();
                           int size1 = qoscomponenttable.size();
                           int index1 = 0;
                           Enumeration e1 = qoscomponenttable.keys();
                           while (e1.hasMoreElements())
                           {
                                   qosfunction = (FunctionQoS)qoscomponenttable.get((String)e1.nextElement());
                                   if (qosfunction !=null)
                                   {
                                      int index2 =0;
                                      String functionname = qosfunction.getFunctionName();
                                      Hashtable qosfunctiontable =  qosfunction.getFunctionQoS();
                                      int size2 = qosfunctiontable.size();
                                      Enumeration e2 = qosfunctiontable.keys() ;
                                      while(e2.hasMoreElements())
                                      {
                                              try
                                              {
                                                 String QoSParameter = (String) e2.nextElement();
                                                 String value = qosfunction.getFunctionQoS(component.getComponentName(), functionname,
QoSParameter);
                                                 if(QoSParameter.equalsIgnoreCase("throughput"))
                                                 {
                                                         query = query + "Select id from " +componentTableName + "compfuncqos where
functionname = '";
                                                         query = query + functionname + "' and QoSParameter = '" + QoSParameter + "' and value
>=  " + value+" ";
                                                 }
                                                 else if (QoSParameter.equalsIgnoreCase("endToEndDelay"))
                                                 {
                                                         query = query + "Select id from " +componentTableName + "compfuncqos where
functionname = '";
                                                         query = query + functionname + "' and QoSParameter = '" + QoSParameter + "' and value
<= " + value +" ";
                                                 }
                                                 if (index2 < (size2 - 1))
                                                    query = query + " Intersect ";
                                                 index2++;
                                              }
                                        catch(Exception e)
                                        {
                                           System.out.println("Error in updating tables "+e.getMessage());
                                        }
                                   }
                              }
                              if (index1 < (size1 -1))
                                 query = query + "Intersect ";
                              index1++;
                         }
                    }
                    else
                    {
                       System.out.println("ComponentQoS object is null ");
                    }
                    String searchquery = baseQuery + query;
                    System.out.println(" ");
                    System.out.println("  " +searchquery);
                    return searchquery;
   }

   private int hopcount;
   private java.lang.String requestID;

   public int getHopcount()
   {
            return hopcount;
   }

   public java.lang.String getRequestID()
   {
            return requestID;
   }

   public void setHopcount(int newHopcount)
   {
            hopcount = newHopcount;
   }

   public void setRequestID(java.lang.String newRequestID)
   {
            requestID = newRequestID;
   }
}
```

QueryManager.java

```
/**
 * Receives query from URDS_proxy and sends to Headhunter
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;
import java.lang.*;
import java.net.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import mss.ea.core.RandomUtil;

public class QueryManager extends UnicastRemoteObject implements IQueryManager
{
   long QIDCounter=0;
   private long defaultResponseTime = 10000;
   private LinkedList queryQueue = new LinkedList();
   private LinkedList feedbackQueue = new LinkedList();
   private Hashtable queryResults = new Hashtable();
   private Hashtable processingQuery = new Hashtable();
   private Hashtable processedQuery = new Hashtable();
   private PreferenceType preferenceType = null;
   private Hashtable queryResponseTime = new Hashtable();
   private Hashtable propagatedHHlist = new Hashtable();
   private Hashtable propagatedhhHashtable = new Hashtable();
   FileWriter out = null;
   FileWriter out1 = null;
   FileWriter out2 = null;
   FileWriter outHash = null;
   FileWriter outTime = null;
   private long queryBeginTime = 0;
   Hashtable hhConfidenceHash = new Hashtable();
   Hashtable timeGraphHash = new Hashtable();

   private IDomainSecurityManager dsm = null;
   ArrayList QIDList=new ArrayList();
   int hhAvailable = 0;

   public void updatePropagatedHHlistFromQM(String QID, ArrayList propHHlist)
   {
           propagatedHHlist.put(QID, propHHlist);
   }

   public void updatePropagatedHHlistFromHH(String QID, ArrayList propHHlist)
   {
      boolean flag = false;
           ArrayList existingList = (ArrayList) propagatedHHlist.get(QID);

           if(existingList ==  null)
           {
              existingList = new ArrayList();
           }
           for(int i=0; i<propHHlist.size(); i++)
           {
              String hh = (String) propHHlist.get(i);
              if(!existingList.contains(hh))
              {
         flag = true;
                   existingList.add(hh);
              }
      }
           synchronized(propagatedHHlist)
           {
              propagatedHHlist.remove(QID);
              propagatedHHlist.put(QID, existingList);
           }
   }

   public ArrayList getPropagatedHHlist(String queryID)
   {
           ArrayList hhPropList = null;
           if(propagatedHHlist.containsKey(queryID))
           {
              hhPropList = (ArrayList) propagatedHHlist.get(queryID);
           }
           return hhPropList;
   }

   public void sendNewQuery(QueryQueueObject queryQueueObject)
   {
      queryQueue.addFirst(queryQueueObject);
   }

   public QueryQueueObject getNextQuery()
   {
      QueryQueueObject queryQueueObject = null;
      try
      {
         queryQueueObject = (QueryQueueObject) queryQueue.removeLast();
      }
      catch(NoSuchElementException nsee)
      {
         // No more elements in the Queue
      }
      return queryQueueObject;
   }

   public void addQueryBeingProcessed(String queryID, ProcessingQueryObject processingQueryObject, int hhCount)
   {
            processingQuery.put(queryID, processingQueryObject);
            Hashtable results = new Hashtable();
            ResultObject resultObject = new ResultObject(results, hhCount);
            queryResults.put(queryID, resultObject);
   }

   public ProcessingQueryObject getProcessingQueryFromHash(String queryID)
   {
      return ((ProcessingQueryObject) processingQuery.get(queryID));
   }

   public void storeProcessedQuery(String queryID, long respondedTime)
   {
      ProcessingQueryObject p = (ProcessingQueryObject) processingQuery.remove(queryID);
      ProcessedQueryObject pqo = new ProcessedQueryObject(respondedTime, System.currentTimeMillis(),
p.getQuerySourceName(),
p.getQuerySourceType(), p.getQuerySourceLocation());
      processedQuery.put(queryID, pqo);
   }

   public void addPropagatedHHArrayList(String queryID, ArrayList propHHList)
   {
            propagatedhhHashtable.put(queryID, propHHList);
   }

   public ArrayList getPropagatedHHArrayList(String queryID)
   {
            ArrayList propHHArrayList = (ArrayList) propagatedhhHashtable.get(queryID);
            return propHHArrayList;
   }

   public void sendNewFeedback(FeedbackObject feedbackObject)
   {
      feedbackQueue.addFirst(feedbackObject);
   }

   public FeedbackObject getNextFeedback()
   {
      FeedbackObject feedbackObject = null;
      try
      {
         feedbackObject = (FeedbackObject) feedbackQueue.removeLast();
      }
      catch(NoSuchElementException nsee)
      {
         // No more elements in the Queue
      }
      return feedbackObject;
   }

   public ArrayList getSelectedHH(String queryID, ArrayList propagatedHHList)
   {
      int hhPropagateCount = 0;
      return (preferenceType.getHHListToPropagateProfile(hhPropagateCount,queryID,propagatedHHList));
```

```
}

public boolean checkHHprofileList(String domainName)
{
            if(preferenceType==null)
            {
        try
        {
           ArrayList hhList = dsm.getHHListForDomain(domainName);

           if(hhList!=null && hhList.size() > 0)
           {
              this.updateHHList(hhList);
           }
           else
           {
              return false;
           }
        }
       catch(Exception eee)
       {
          System.out.println(eee);
                      return false;
       }
            }
            return true;
}

public void updateProfileInfo(Hashtable feedback)
{
            preferenceType.updateProfileInfo(feedback, new ArrayList());
}

public boolean getFinishedFlag()
{
            return finishedFlag;
}

public void updateHHList(ArrayList hhList)
{
   if(hhList.size()>0)
   {
      if(preferenceType == null)
      {
         preferenceType = new PreferenceType(hhList);
      }
      else
      {
         preferenceType.updateHHList(hhList);
      }
   }
}

public synchronized void setQueryResults(String queryID, Hashtable results, String hhName)
{
   if(queryResults.get(queryID) == null)
   {
   }
   else
   {
      ResultObject resultObject = (ResultObject) queryResults.remove (queryID);
      resultObject.addNewComponentSet(results);
      queryResults.put(queryID, resultObject);
   }
}

public synchronized void setQueryResults(String queryID, String hhName)
{
      if(queryResults.get(queryID) == null)
      {
      }
      else
      {
         ResultObject resultObject = (ResultObject) queryResults.remove (queryID);
         resultObject.addNewComponentSet();
         queryResults.put(queryID, resultObject);
      }
   }

   public Hashtable getQueryResults()
   {
      return queryResults;
   }

   public void removeQueryResultEntry(String queryID)
   {
            queryResults.remove(queryID);
   }


   // Called by the URDS_Proxy
   public void getSearchResultTable(QueryBean querybean, long responseTime, String querySourceLocation) throws
RemoteException
   {
            // Source Name is System Developer
            String sourceType = "SD";
            String QIDString = "";

      // generate query ID for the query
            QIDCounter++;

            if(QIDCounter<=9)
         QIDString = "000";
            else if(QIDCounter<=99)
         QIDString = "00";
            else if(QIDCounter<=999)
         QIDString = "0";
            else
         QIDString = "";
            String QID = QIDString + QIDCounter;

            QIDList.add(QID);
            if(responseTime <= 0)
            {
               responseTime = defaultResponseTime;
            }
            long queryInsertionTime = System.currentTimeMillis();
            QueryQueueObject queryQueueObject = new QueryQueueObject(QID, querybean, queryInsertionTime, responseTime,
querySourceLocation);
            queryBeginTime = System.currentTimeMillis();
            ResponseTimeObject responseTimeObject = new ResponseTimeObject(queryBeginTime);
            queryResponseTime.put(QID, responseTimeObject);
            this.sendNewQuery(queryQueueObject);
   }

   public void updateResponseTime(String queryID)
   {
            synchronized(queryResponseTime)
            {
               ResponseTimeObject responseTimeObject = (ResponseTimeObject) queryResponseTime.remove(queryID);
               responseTimeObject.setQueryEndTime(System.currentTimeMillis());
               queryResponseTime.put(queryID, responseTimeObject);
            }
   }

   public static void main(String[] args)
```

```
    {
      String dsmLocation="//magellan.cs.iupui.edu:"+args[2]+"/DomainSecurityManager";
      String qmLocation ="//"+args[0]+".cs.iupui.edu:"+ args[1] +"/QueryManager";
      try
      {
        System.setSecurityManager(new RMISecurityManager());
        Naming.rebind (qmLocation, new QueryManager(dsmLocation, qmLocation));
        System.out.println ("QueryManager object binded to location " + qmLocation);
        System.out.println ("QueryManager is ready");
      }
      catch (Exception e)
      {
        System.out.println ("QueryManager failed: " + e);
      }
    }

    public QueryManager(String dsmLocation, String qmLocation) throws RemoteException
    {
            try
            {
              System.setSecurityManager(new RMISecurityManager());
              dsm = (IDomainSecurityManager) Naming.lookup(dsmLocation);
              dsm.setQMlocation(qmLocation);
            }
            catch (Exception e)
            {
              System.out.println(e.getMessage());
            }
            loadHashConfidence();

      // Thread for QueryQueueThread
      try
      {
        QM_QueryQueueThread qqThreadObj = new QM_QueryQueueThread(this, qmLocation);
        Thread queryQueueThread = new Thread(qqThreadObj);
        queryQueueThread.start();
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }

      // Thread for CombineResultThread
      try
      {
        QM_CombineResultThread crThreadObj = new QM_CombineResultThread(this);
        Thread combineResultThread = new Thread(crThreadObj);
        combineResultThread.start();
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }

      // Thread for FeedbackThread
      try
      {
        FeedbackThread fbThreadObj = new FeedbackThread(this, dsmLocation);
        Thread feedbackThread = new Thread(fbThreadObj);
        feedbackThread.start();
      }
      catch (Exception e)
      {
        e.printStackTrace();
      }
    }
  }
```

QueryQueueObject.java

```
/**
 * Object to store query details. This is sent between the entities.
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;
import java.util.*;

public class QueryQueueObject implements Serializable
{
   private String queryID;
   private QueryBean queryBean;// Query details
   private long queryResponseTime;
   private long queryInsertionTime;
   private String querySourceName;
   private String querySourceType;              // HH or QM
   private String querySourceLocation;
   private ArrayList propagatedList;
   private int queryLevel;

   public QueryQueueObject(String qID, QueryBean qBean, long insertionTime, long responseTime, String sourceLocation)
   {
             queryID = qID;
             queryBean = qBean;
             queryInsertionTime = insertionTime;
             queryResponseTime = responseTime;
             querySourceLocation = sourceLocation;
             querySourceName = null;
             querySourceType = "SD";
             propagatedList = new ArrayList();
             queryLevel = 0;
   }

   public QueryQueueObject(String qID, QueryBean qBean, long insertionTime, long responseTime, String sourceName, String
sourceType,String sourceLocation)
   {
             queryID = qID;
             queryBean = qBean;
             queryInsertionTime = insertionTime;
             queryResponseTime = responseTime;
             querySourceName = sourceName;
             querySourceType = sourceType;
             querySourceLocation = sourceLocation;
             propagatedList = new ArrayList();
             queryLevel = 0;
   }

   public String getQueryID()
   {
             return queryID;
   }

   public int getQueryLevel()
   {
             return queryLevel;
   }

   public void setQueryLevel(int qLevel)
   {
             queryLevel = qLevel;
   }

   public void incrementQueryLevel()
```

```
    {
            queryLevel++;
    }

    public long getResponseTime()
    {
            return queryResponseTime;
    }

    public long getInsertionTime()
    {
            return queryInsertionTime;
    }

    public QueryBean getQueryBean()
    {
            return queryBean;
    }

    public String getQuerySourceName()
    {
            return querySourceName;
    }

    public String getQuerySourceType()
    {
            return querySourceType;
    }

    public String getQuerySourceLocation()
    {
            return querySourceLocation;
    }

    public ArrayList getHHpropagatedList()
    {
            return propagatedList;
    }

    public QueryQueueObject cloneObject(long remainingResponseTime, String location, String type, String name)
    {
            QueryQueueObject qqo = new QueryQueueObject(queryID, queryBean, queryInsertionTime, remainingResponseTime,
name, type,location);
            qqo.updateHHpropagateList(propagatedList);
            qqo.setQueryLevel(this.queryLevel);
            return qqo;
    }

    public void setNewQuerySource(String location, String type)
    {
            querySourceLocation = location;
            querySourceType = type;
    }

    public void setNewQuerySource(String location)
    {
            querySourceLocation = location;
    }

    public void setNewResponseTime(long newResponseTime)
    {
            queryResponseTime = newResponseTime;
    }

    public void setQueryInsertionTime(long newInsertionTime)
    {
            queryInsertionTime = newInsertionTime;
    }
    public void updateHHpropagateList(String hhName)
    {
                if(!propagatedList.contains(hhName))
                {
                    propagatedList.add(hhName);
                }
                else
                {
                    // hhName already in the propagatedList
                }
    }

    public void updateHHpropagateList(ArrayList newHHList)
    {
                for(int i=0; i<newHHList.size(); i++)
                {
                    String hhName = (String) newHHList.get(i);
                    if(!propagatedList.contains(hhName))
                    {
                        propagatedList.add(hhName);
                    }
                    else
                    {
                                // hhName already in the propagatedList
                    }
                }
    }
}
```

QueryQueueThread.java

```
/**
 * This Thread takes each query from the QueryQueue and processes them
 * The QueryQueue has QueryQueue objects, stored as a queue
 *
 * @author: Barun Devaraju
 * @date: Dec 2004
 */

import java.net.*;
import java.security.*;
import java.util.*;
import java.lang.*;
import uka.transport.*;
import java.rmi.*;
import java.rmi.registry.*;
import mss.ea.core.RandomUtil;

public class QueryQueueThread implements Runnable
{
    private int profileLevel = 1;
    private int threadNumber;
    private IDomainSecurityManager idsm = null;
    private IQueryManager iqm = null;
    private Headhunter hh = null;
    private String componentTableName = null;
    private Hashtable ratingResults = new Hashtable();
    private String hhLocation = null;
    private String qmLocation = null;
    private long projectedMRaccessTime = 50;
    private long profileAccessTime = 50;
    private long combineComponentsTime = 100;
    private long expectedHHcompReturnTime = 500;
    private long sleepTime = 300;
    private ArrayList propagatedhhArrayList = new ArrayList();

    public void run()
```

```
  {
             Thread CurrentThread = Thread.currentThread();
             QueryQueueObject queryQueueObject = null;
             QueryQueueObject queryQueueObjectToSend = null;
             boolean propagateFlag = true;

             while(true)
             {
                try
                {
                         queryQueueObject = hh.getNextQuery();
                         if(queryQueueObject == null)
                         {
                            CurrentThread.sleep(sleepTime);
                         }
                         else
                         {
                            queryQueueObject.incrementQueryLevel();
                            String queryID = queryQueueObject.getQueryID();
                            long waitTime = System.currentTimeMillis() - queryQueueObject.getInsertionTime();
                            long remainingResponseTime = queryQueueObject.getResponseTime() - waitTime;
                            remainingResponseTime = remainingResponseTime - profileAccessTime - combineComponentsTime;
                            int propagatedHHcount=0;
                            if(remainingResponseTime > expectedHHcompReturnTime)
                            {
                                     String locationOfHH = null;
                                     IHeadhunter ihh;
                propagateFlag = hh.checkHHprofileList(queryQueueObject.getQueryBean().getDomain());
                ArrayList propagatedList = null;
                if(propagateFlag)
                {
                                      ArrayList forQueryQueue = queryQueueObject.getHHpropagatedList();
                                      forQueryQueue.add(hhLocation);
                                      boolean randomFlag=true;
                                      int queryLevel = queryQueueObject.getQueryLevel();
                                      if(queryLevel <= profileLevel )
                                      {
                                               System.out.println("QQT: HHs will be selected thru PROFILE Information - level " +
queryLevel);
                                               randomFlag=false;
                                      }
                                      else
                                      {
                                               System.out.println("QQT: HHs will be selected RANDOM..........Level-" + queryLevel );
                                      }


                                     ArrayList hhList = hh.getSelectedHH(queryID, forQueryQueue, randomFlag);

                                     if(hhList!=null)
                                     {
                   queryQueueObjectToSend =
queryQueueObject.cloneObject(remainingResponseTime,hhLocation,"HH",componentTableName);

                                      ArrayList propHHlist = new ArrayList(); // list to store HHs to which query will be propagated
                                      boolean sentFlag = false;
                                      int index = 0;
                                      boolean allSent = false;
                                      propagatedHHcount = 0;
                                      // Randomly decide the number of HHs to propagate the query
                                      RandomUtil rand = new RandomUtil();
                                      int toPropagate = rand.randomInt(1,3);
                                      propagatedhhArrayList.clear();

                                      // Send the query to a few or all of the selected HHs
                                      while(propagatedHHcount<hhList.size() && propagatedHHcount<toPropagate &&
index<hhList.size())
                                      {
                                        sentFlag = false;
                                        locationOfHH = (String) hhList.get(index);
                                        try
                                        {
                                           ihh = (IHeadhunter) Naming.lookup(locationOfHH);
                        sentFlag = ihh.sendNewQuery(queryQueueObjectToSend, "HH" , hhLocation);
                                           if(sentFlag)
                                           {
                                                      propagatedHHcount++;
                                                      if(propagatedHHcount==1)
                                                      {
                                                         // store Info in the processingQuery Hash
                                                         ProcessingQueryObject processingQueryObject = new
ProcessingQueryObject(System.currentTimeMillis(),
remainingResponseTime,queryQueueObject.getQuerySourceName(),
queryQueueObject.getQuerySourceType(),queryQueueObject.getQuerySourceLocation());
                                          hh.addQueryBeingProcessed(queryQueueObject.getQueryID(),
                                        processingQueryObject);
                                                      }
                                        queryQueueObjectToSend.updateHHpropagateList(locationOfHH);
                                                      propagatedhhArrayList.add(locationOfHH);

                                                      // add in the propagated HH list
                                                      propHHlist.add(locationOfHH);
                                     }
                                        }
                                        catch(Exception ee)
                                        {
                        System.out.println(ee);
                                        }
                                        index++;
                                 }
                                 hh.addPropagatedhhArrayList(queryID, propagatedhhArrayList);

                                 // Check for initialization of QM object
                                 if(iqm==null)
                                 {
                                        try
                                        {
                                           iqm = (IQueryManager) Naming.lookup(qmLocation);
                                        }
                                        catch(Exception eee)
                                        {
                                 System.out.println(eee);
                                        }
                                 }
                                 if(iqm!=null)
                                 {
                                        // call QM and update the propagated HH list
                                        try
                                        {
                        iqm.updatePropagatedHHlistFromHH(queryID, propHHlist);
                                        }
                                        catch(Exception eee)
                                        {
                        System.out.println(eee);
                                        }
                                 }
                                }

               } // end of Flag check for profile creation
               else
               {
                  System.out.println("Queue: Profile cannot be created for HHs");
               }
                        }
                        else
                        {
```

```
                        remainingResponseTime = remainingResponseTime + profileAccessTime;
                }

                // Component list returned with component confidence info
                ratingResults = hh.searchForComponent(queryQueueObject.getQueryBean());

                        if(ratingResults.size()>0)
                        {
                          // add hhLocation and time to the component results
          Enumeration e = ratingResults.keys();
          while(e.hasMoreElements())
          {
               String componentID = (String) e.nextElement();
            ComponentDetailsObject cdo = (ComponentDetailsObject) ratingResults.remove(componentID);
                                    cdo.setHeadhunterName(hhLocation);
                          cdo.setTime(System.currentTimeMillis());
            ratingResults.put(componentID, cdo);
          }
                        }

                if(propagatedHHcount>0)
                {
                        // set the results on MR in the Headhunter
                        hh.setQueryResults(queryQueueObject.getQueryID(), ratingResults, propagatedHHcount+1);
                }
                else
                {
                        String sourceLocation = queryQueueObject.getQuerySourceLocation();
                        String hhName = hh.getHHname();
        // Send this result set to the requestor
        if(sourceLocation.indexOf("HeadHunter")>0)
        {
          IHeadhunter ihh = null;
          // requestor is HH - look up for object
          try
          {
            ihh = (IHeadhunter) Naming.lookup(sourceLocation);

            if(ihh == null)
            {
              System.out.println("#### Queue: HH object is NULL after lookup in " + sourceLocation);
            }
            else
            {
                                    if(ratingResults.size()>0)
                                    {
                        ihh.setQueryResults(queryID, ratingResults, hhName);
                                    }
                                    else
                                    {
                                            // return the results that no component is found
                        ihh.setQueryResults(queryID, hhName);
                                    }
            }
          }
          catch(Exception ee)
          {
            System.out.println(ee);
          }
        }
        else
        {
          IQueryManager qm = null;
          // requestor is QM - look up for object
          try
          {
            qm = (IQueryManager) Naming.lookup(sourceLocation);
```

```
                    }
                    catch(Exception eee )
                    {
                       System.out.println(eee);
                    }

                    try
                    {
                       if(ratingResults.size()>0)
                       {
                          qm.setQueryResults(queryID, ratingResults, hhName);
                       }
                       else
                       {
                          qm.setQueryResults(queryID, hhName);
                       }
                    }
                    catch(Exception eee)
                    {
                       System.out.println(eee);
                    }

                 }
                 long timeToProcess =  System.currentTimeMillis() - queryQueueObject.getInsertionTime();
                 // add the query in the processedQuery Hashtable
                 hh.storeInProcessedQuery(queryID, timeToProcess, queryQueueObject);

                          } // end of IF condition for checking the number of query propagated HHs
                  }
                }
                catch(Exception e)
                {
                          e.printStackTrace();
                }
             }
   }

   public void setName(int num)
   {
             threadNumber = num;
   }

   public QueryQueueThread(Headhunter head, String name, String location, String qmLoc)
   {
             try
             {
                hh = head;
                componentTableName = name;
                hhLocation = location;
                qmLocation = qmLoc;
             }
             catch (Exception e)
             {
                e.printStackTrace();
             }
   }
}
```

ResponseTimeObject.java

```
/**
 * Object to store the response time for the queries
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;
import java.util.*;

public class ResponseTimeObject implements Serializable
{
   private long queryBeginTime;
   private long queryEndTime;
   private long responseTime;

   public ResponseTimeObject(long beginTime)
   {
              queryBeginTime = beginTime;
   }

   public long getQueryBeginTime()
   {
              return queryBeginTime;
   }

   public long getQueryEndTime()
   {
              return queryEndTime;
   }

   public long getResponseTime()
   {
              return responseTime;
   }

   public void setQueryEndTime(long endTime)
   {
              queryEndTime = endTime;
              responseTime = queryEndTime - queryBeginTime;

   }
}
```

ResultObject.java

```
/**
 * Object to store the result
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 */

import java.io.*;
import java.util.*;
import uka.transport.*;

public class ResultObject implements Serializable
{
   private int propagatedHHcount;
   private int resultObtainedCount;
   private Hashtable matchedComponents;
   private boolean updatedFlag;

   public ResultObject()
   {
              propagatedHHcount = 0;
              resultObtainedCount = 0;
              matchedComponents = new Hashtable();
              updatedFlag = false;
   }

   public ResultObject(Hashtable components, int hhCount)
   {
              propagatedHHcount = hhCount;
              resultObtainedCount = 0;
              matchedComponents = new Hashtable();
              updatedFlag = false;
              try
              {
                 // Copy the component information obtained into the Result object
           DeepClone dc = new DeepClone();
           matchedComponents = (Hashtable)dc.doDeepClone(components);
              }
              catch(Exception e)
              {
                 System.out.println("#### ResultObject: ERROR in deep clone - " + e);
              }
   }

   public void setPropagatedCount(int newCount)
   {
              propagatedHHcount = newCount;
              updatedFlag = true;
   }

   public void setUpdatedFlag()
   {
              updatedFlag = true;
   }

   public int getUpdatedFlag()
   {
              return propagatedHHcount;
   }

   public int getPropagatedHHcount()
   {
              return propagatedHHcount;
   }

   public int getResultObtainedCount()
   {
              return resultObtainedCount;
   }

   public Hashtable getMatchedComponents()
   {
              return matchedComponents;
   }

   public void checkResultsObtained(String queryID)
   {
              System.out.println("ResultObject: QueryID - " + queryID +"\tFLAG - " + updatedFlag+ "\tPropagatedHHcount - "
+propagatedHHcount+"\tResultObtainedCount - " +resultObtainedCount);
   }

   public boolean isAllResultsObtained()
   {
              if(updatedFlag)
              {
                 if(resultObtainedCount >= propagatedHHcount)
                 {
                            return true;
                 }
                 else
                 {
                            return false;
                 }
              }
              else
              {
                 return false;
              }
```

```
    }

    public boolean isAllResultsObtainedFromQM()
    {
            if(resultObtainedCount >= propagatedHHcount)
            {
                return true;
            }
            else
            {
                return false;
            }
    }

    public void addNewComponentSet()
    {
            // Increase the result Obtained HH count
            resultObtainedCount++;
    }

    public void addNewComponentSet(Hashtable newComponents)
    {
            // Increase the result Obtained HH count
            resultObtainedCount++;

            // Insert the new HHs results
            Enumeration e = newComponents.keys();
            while(e.hasMoreElements())
            {
                String compID = (String) e.nextElement();
                ComponentDetailsObject cdo = (ComponentDetailsObject) newComponents.get(compID);

                if(matchedComponents.containsKey(compID))
                {
                        // components already in the result table
                        System.out.println("ResultObject: Component " + compID + " already in the result table ####");
                }
                else
                {
                        // insert the components in the Hashtable
                        matchedComponents.put(compID, cdo);
                }
            }
    }
}
```

SQLHelper.java

```
/**
 * This is the class which serves as a connection to the oracle database.
 * It establises the database connection and executes queries
 *
 * @author: Nanditha Nayani
 * @date: Aug 2001
 */

import java.util.*;
import java.sql.*;

public class SQLHelper
{
    private java.sql.Connection dbconn = null;
    private java.sql.Statement statement = null;
    String url = "jdbc:oracle:thin:@phoenix.cs.iupui.edu:1521:cs9iorcl";
    String username = "headhunter";
    String password = "newchange4";

    public SQLHelper() throws java.lang.Exception
```

```
  {
          try
          {
            Class.forName("oracle.jdbc.driver.OracleDriver");
          }
          catch (Exception sqlex)
          {
            System.err.println("\n Unable to connect to Oracle Server");
            sqlex.printStackTrace();
          }
  }

  public final void commitTransaction() throws java.lang.Exception
  {
          try
          {
            dbconn = DriverManager.getConnection(url, username, password);
            statement = dbconn.createStatement();
            dbconn.commit();
      statement.close();
      dbconn.close();
          }
          catch (SQLException sqle)
          {
            throw new Exception("\n SQL Exception during commitTransaction with message :" + sqle.getMessage());
          }
  }

public ResultSet executeQuery(String query) throws java.lang.Exception {

          ResultSet resultSet = null;

          try {

        dbconn = DriverManager.getConnection(url, username, password);
        statement = dbconn.createStatement();
                  resultSet = statement.executeQuery(query);
        //statement.close();
        //dbconn.close();
          } catch (SQLException sqle) {
                  throw new Exception(
                          "\n SQL Exception during executeQuery with message:" + sqle.getMessage());
          }

          return resultSet;
  }

  public java.sql.Connection getDbconn()
  {
          return dbconn;
  }

  public java.sql.Statement getStatement()
  {
          return statement;
  }

  public final void initiateTransaction() throws java.lang.Exception
  {
          try
          {
            dbconn.setAutoCommit(false);
          }
          catch (SQLException sqle)
          {
            throw new Exception("\n SQL Exception during initiateTransaction with message :"+ sqle.getMessage());
          }
  }
```

```
    public void rollbackTransaction() throws java.lang.Exception
    {
                try
                {
                   dbconn.rollback();
                }
                catch (SQLException sqle)
                {
                   throw new Exception("\n SQL Exception during rollbackTransaction with message :"+ sqle.getMessage());
                }
    }

    public void setDbconn(java.sql.Connection newDbconn)
    {
                dbconn = newDbconn;
    }

    public void setStatement(java.sql.Statement newStatement)
    {
                statement = newStatement;
    }

    public void shutDown() throws java.lang.Exception
    {
                try
                {
                   if (statement != null)
                               statement.close();
                   if (dbconn != null)
                               dbconn.close();
                }
                catch (Exception e)
                {
                   throw new Exception(e.getMessage());
                }
    }

    public void updateTable(String updateString) throws java.lang.Exception
    {
                try
                {
                   dbconn = DriverManager.getConnection(url, username, password);
                   statement = dbconn.createStatement();
                   dbconn.setAutoCommit(false);
                   statement.executeUpdate(updateString.trim());
                   dbconn.commit();
                   statement.close();
                   dbconn.close();
                }
                catch (SQLException sqle)
                {
                   throw new Exception("\n SQL Exception from function updateTable with message:" + sqle.getMessage());
                }
    }
}
```

URDS_Proxy.java

```
/**
 * This class is used to conduct all the Experiment
 *
 * @author: Barun Devaraju
 * @date: Nov 2004
 *
 */

import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;
import java.util.*;
import mss.ea.core.RandomUtil;

public class URDS_Proxy extends UnicastRemoteObject implements IURDS_Proxy
{
   private boolean timedExperiment = false;
   private boolean randomExp = false;
   private boolean feedbackFlag = true;
   private BufferedReader br = null;
   private IQueryManager queryManager = null;
   AbstractComponent component = new AbstractComponent();
   private QueryBean queryBean = null;
   private long responseTime = 2000000;
   private int totalQueries=1;
   private int queriesSent=0;
   private int resultsReceived=0;

   private int experimentNumber = 400;
   private int experimentDone = 0;
   private int queryLimit = 1;
   private String domainName = "Document";
   private String location = null;

   private Timer timer = null;
   private long interval = 200;
   private ArrayList compNames = new ArrayList();
   private FileReader fin = null;
   private FileWriter out = null;
   private FileWriter outResult = null;

   public URDS_Proxy(String qmPort, String url) throws RemoteException
   {
      compNames.add("DocumentServer");
      compNames.add("DocumentTerminal");

      try
      {
               location = url;
               queryManager =(IQueryManager)Naming.lookup("//magellan.cs.iupui.edu:" + qmPort + "/QueryManager");
      }
      catch(Exception e)
      {
         System.err.println(e);
      }
   }

   class QueryTimer extends TimerTask
   {
            public void run()
            {
               if(queriesSent < totalQueries)
               {
           try
            {
                        queryManager.getSearchResultTable(queryBean, responseTime, location);
                        queriesSent++;
            }
                      catch(Exception e )
                      {
                      }
               }
               else
               {
                      timer.cancel();
               }
            }
    }

    public void searchConcreteComponents() throws RemoteException
    {
            component.setDomainName(domainName);
            component.setSystemName("DocumentManager");
       if(queryManager == null)
       {
          System.out.println("Query Manager is not ready");
       }
            else
            {
               queryBean = new QueryBean(component);
               queriesSent = 0;
               resultsReceived = 0;
               if(timedExperiment)
               {
                        timer = new Timer();
                        timer.schedule(new QueryTimer(), 0, interval);
                        System.out.println("Timer scheduled for interval " + interval);
               }
               else if(randomExp)
               {
            RandomUtil rand = new RandomUtil();
            int r = rand.randomInt(1, compNames.size());
                        String compName = (String) compNames.get(r-1);
                        component.setComponentName(compName);
            int rQualityLevel = rand.randomInt(1, 10);
                        component.setQualityLevel(String.valueOf(rQualityLevel));
                        while(queriesSent < totalQueries)
                        {
              try
               {
                                    queryManager.getSearchResultTable(queryBean, responseTime, location);
                                    queriesSent++;
                                    System.out.println(">>>> " + queriesSent + " at " +System.currentTimeMillis() +
"\t"+component.getComponentName() + " " + component.getQualityLevel());
               }
                           catch(Exception e )
                           {
                                    System.out.println("#### ERROR in contacting Query Manager and getting results ");
                           }
                        }
               }
               else
               {
            String componentName = "";
          String qualityLevel = "";
                        String line = new String();
                        try
                        {
                           if((line = br.readLine()) != null)
               {
                                    componentName = "";
                                    qualityLevel = "";
            StringTokenizer parser = new StringTokenizer(line);
            if(parser.hasMoreTokens())
            {
                           componentName = parser.nextToken();
            }
            if(parser.hasMoreTokens())
            {
                           qualityLevel = parser.nextToken();
            }
                                    component.setComponentName(componentName);
                                    component.setQualityLevel(qualityLevel);
```

```
                                        if(componentName!=null && componentName.compareTo("")!=0)
                                        {
                              try
                              {
                                                  queryManager.getSearchResultTable(queryBean, responseTime, location);
                                                  queriesSent++;
                                                  System.out.println(">>>> " + queriesSent + " at " +System.currentTimeMillis() +
"\t"+component.getComponentName() + " " + component.getQualityLevel());
                              }
                                        catch(Exception e )
                                        {
                                                  System.out.println("#### ERROR in contacting Query Manager and getting results ");
                                        }
                                        }
                              }
                         }
                         catch(Exception ere)
                         {
                            System.out.println("#### File read error");
                         }
                 }
              }
   }

   public int obtainResults(String queryID, Hashtable result)
   {
              resultsReceived++;
              System.out.println("URDS_Proxy: " + result.size() + " components matched for query " + queryID);
      ArrayList resultList = new ArrayList();
      FeedbackObject fbo = new FeedbackObject(queryID,domainName);

      if(result != null)
      {
         Enumeration values = result.keys();
         while(values.hasMoreElements())
         {
                         String componentID = (String)values.nextElement();
                         ComponentDetailsObject cdo = (ComponentDetailsObject) result.get(componentID);
                         if(feedbackFlag)
                         {
               // Create feedback hash
               FeedbackProfile fbp = new FeedbackProfile(cdo.getComponentRating());
               fbo.addNewEntry(cdo.getHeadhunterName(), fbp);
                         }
         }
                if(feedbackFlag)
                {
              try
              {
               System.out.println("FB size - " + fbo.getFBprofile().size());
               queryManager.sendNewFeedback(fbo);
               System.out.println("URDS: Feedback sent to QM -------->>>>>>>>>");
              }
              catch(Exception ee)    {}
         }
   }
              // -1 - don't do anything
              // 0 - one experiment done - for one value of totalQuery
              // 1 - all experiment done - for one value of totalQuery
              // 2 - all experiment done - for all queries
     if(resultsReceived>=totalQueries)
     {
                experimentDone++;
                if(experimentDone<experimentNumber)
                {
              System.out.println();
                    System.out.println("Experiment " + (experimentDone+1) + " outof " + experimentNumber + " for total query "
+totalQueries);
```

```
                        try
                        {
                           this.searchConcreteComponents();
                        }
                        catch(Exception ee)
                        {
                           System.out.println("#### ERROR - calling for next experiment");
                        }
                        return 0;
                }
                else
                {
                        totalQueries+=10;       // increment the totalQueries count
                        experimentDone = 0;  // initialize the number of experiments done
                  if(totalQueries<=queryLimit)
                        {
                           try
                           {
                              this.searchConcreteComponents();
                           }
                           catch(Exception ee)
                           {
                                      System.out.println("#### ERROR - calling for next experiment");
                           }
                           return 1;
                         }
                         else
                         {
                           return 2;
                         }
                 }
     }
             return -1;
  }
  public static void main(String[] args)
  {
    String url = null;
             String qmPort = null;

     if(args.length > 2)
     {
        System.out.println("Usage: java URDS_Proxy server_name responseTime");
     }
     if(args.length == 0)
             {
        url = "//magellan.cs.iupui.edu:4432/URDS_Proxy";
             }
     else
             {
        url = "//magellan.cs.iupui.edu:" + args[0] + "/URDS_Proxy";
             }
     try
     {
        URDS_Proxy urds_Proxy= new URDS_Proxy(qmPort, url);
        Naming.rebind(url, urds_Proxy);
                System.out.println("URDS_Proxy is registered in location " + url);
        urds_Proxy.searchConcreteComponents();
     }
     catch(Exception e)
     {
        System.err.println(e);
        System.exit(1);
     }
  }
}
```