# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TABLE OF CONTENTS</td>
<td>i</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>i</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ii</td>
</tr>
<tr>
<td>LIST OF TERMS</td>
<td>i</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>iii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Problem Definition and Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Objectives: Statement of Goals</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Overall Approach and Contributions</td>
<td>5</td>
</tr>
<tr>
<td>1.5 Organization</td>
<td>7</td>
</tr>
<tr>
<td>2. RELATED WORK</td>
<td>8</td>
</tr>
<tr>
<td>2.1 Resource Discovery</td>
<td>8</td>
</tr>
<tr>
<td>2.1.1 Lookup (Directory) Services</td>
<td>9</td>
</tr>
<tr>
<td>2.1.2 Discovery Services</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Profiling in Discovery Systems</td>
<td>14</td>
</tr>
<tr>
<td>2.2.1 GUARDIANS</td>
<td>15</td>
</tr>
<tr>
<td>2.2.2 Cheshire Project</td>
<td>15</td>
</tr>
<tr>
<td>2.2.3 Ontology-Driven Peer Profiling</td>
<td>16</td>
</tr>
<tr>
<td>2.2.4 Self Profiled Query Processing</td>
<td>16</td>
</tr>
<tr>
<td>2.3 Component-based Approaches in Grid</td>
<td>16</td>
</tr>
<tr>
<td>2.3.1 XCAT (developed by Indiana University, previously CCAT)</td>
<td>16</td>
</tr>
<tr>
<td>2.3.2 CrossGrid</td>
<td>17</td>
</tr>
<tr>
<td>2.3.3 ECSF (Enterprise Computational Services Framework)</td>
<td>17</td>
</tr>
<tr>
<td>3. PREVIOUS WORK</td>
<td>18</td>
</tr>
<tr>
<td>3.1 UniFrame Approach</td>
<td>18</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Implementation</td>
</tr>
<tr>
<td>6.2.4</td>
<td>Validation</td>
</tr>
<tr>
<td>7.1</td>
<td>Metrics</td>
</tr>
<tr>
<td>7.2</td>
<td>Experiments</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Experiments altering the nature of propagation tree</td>
</tr>
<tr>
<td>7.2.2</td>
<td>Experiments using different types of Query Arrival</td>
</tr>
<tr>
<td>7.2.3</td>
<td>Experiments varying the number of entities</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Overall Conclusions</td>
</tr>
<tr>
<td>8.1</td>
<td>Design</td>
</tr>
<tr>
<td>8.2</td>
<td>Specifications</td>
</tr>
<tr>
<td>8.3</td>
<td>Modifications of registration mechanism</td>
</tr>
<tr>
<td>8.4</td>
<td>Background</td>
</tr>
<tr>
<td>8.4.1</td>
<td>Introduction to Grid</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Existing Approaches</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Script based approaches</td>
</tr>
<tr>
<td>8.4.4</td>
<td>Application workflows</td>
</tr>
<tr>
<td>8.4.5</td>
<td>Miscellaneous Approaches</td>
</tr>
<tr>
<td>8.4.6</td>
<td>Observations</td>
</tr>
<tr>
<td>8.5</td>
<td>Case Study</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Using OGSA Grid Services</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Creating a DIFS system with GridFrame</td>
</tr>
<tr>
<td>8.6</td>
<td>Conclusion</td>
</tr>
<tr>
<td>9.1</td>
<td>Conclusions</td>
</tr>
<tr>
<td>9.2</td>
<td>Contributions of thesis</td>
</tr>
<tr>
<td>9.3</td>
<td>Future Work</td>
</tr>
<tr>
<td>9.4</td>
<td>Summary</td>
</tr>
<tr>
<td></td>
<td>LIST OF REFERENCES</td>
</tr>
</tbody>
</table>
1. Introduction

While most traditional computer applications were designed to run on a single computer operating in isolation, more and more applications are being designed to run on networks of cooperating computers. This is because of the recent emergence of the global Internet, new technologies for high bandwidth and wireless data communications, and the availability of ever cheaper and more powerful computers. This has forced the transition of computing paradigms from a centralized approach to a distributed one, leading to the development of distributed computing systems (DCS). The inherent distribution of DCS has meant that components (software and hardware) can be physically as well as logically distributed. This actuality coupled with other requirements of DCS such as the dynamic system configuration and integration makes the software design of a DCS a challenging task.

Component-based systems [KRI04] are increasingly becoming popular in the software design of a DCS. However with the ever-changing requirements and goals of organizations, designing of DCS from scratch is not adequate. As a result, assembling software composed of independently developed, reusable heterogeneous components has emerged for DCS. But though technological alternatives such as J2EE, CORBA, etc., do exist for realizing the software for DCS, most of these choices exist in their own world. Characteristically, these models assume the presence of homogeneous environments i.e., components developed using a particular model, assume the presence of other components, based on the same model. For example, J2EE (Java 2 Enterprise Edition) incorporating its distributed computing model (Java –RMI) explicitly mandates an environment consisting of all Java-based components using method calls for remote communications. Similarly, for CORBA (Common Object Request Broker Architecture) and .NET Framework, although language independence using interface languages is provided, the implicit assumption of homogeneity still holds good.

As local autonomy is inherent in an open DCS [COU01], forcing every component developer to abide by certain rigid rules (common platform, language, etc.) is not a
viable/attractive option. This has led to components adhering to different technological models, thus, making the task of their integration a daunting challenge. Although, different standards (e.g., J2EE – CORBA using IIOP) have been proposed to achieve interoperability between such heterogeneous components, their development and assembling is typically ad-hoc and handcrafted. Such handcrafting is error-prone and thus, automation is needed to reduce the possibility of errors introduced by manual interventions.

An important step in the automated DCS generation is the process of discovering appropriate components. Many applications not only require the software realization of a DCS, but also mandate a high-confidence about it. As the overall confidence about a DCS is a function of the confidences of the individual components and their inter-relations, the task of selecting appropriate components as the building blocks for a DCS is critical. Hence, care must be taken in designing an efficient component discovery system that can interoperate among heterogeneous components. One approach to tackling this problem is to create DCS by integrating geographically scattered heterogeneous software components, using component based models.

UniFrame Resource Discovery Service (URDS) architecture [SIR03] is one such resource discovery service of a framework called UniFrame [RAJ01]. UniFrame provides an approach to achieve a seamless integration of heterogeneous components. A preliminary prototype [SIR02] for URDS was implemented to validate the approach. This thesis deals with three distinct facets of the implemented URDS, namely scalability, search performance and interoperability. Although the URDS approach is comprehensive and proposes a complete solution for resource discovery, the implemented URDS prototype was a basic discovery service encompassing a minimal set of composed entities. As a step towards addressing this shortcoming and for addressing the first aforementioned feature of URDS specifically scalability of the system, new algorithms for query propagation and fault handling have been implemented in this thesis. Consequently, insight and lessons learnt in experimental testing and validation of these algorithms in the course of ascertaining scalability, were employed in enhancing the complexity of the
query propagation algorithms in the form of new learning algorithms for improving the quality of the results with regard to turnaround time, appropriateness to the query, etc. This step addressed the second facet of URDS i.e. search performance. Also, to corroborate these algorithms, a computer simulation was also designed and implemented.

After the first major part was concluded, the next focus of the thesis was on the third aforesaid property of URDS namely interoperability. Interoperability [IEE90] is defined as the ability of two or more systems to exchange information and to use the information that has been exchanged. In the implemented URDS [SIR02] prototype, homogeneity is prevalent with all the constituent entities and the discovered services being based on Java-RMI. But, typically in a real world scenario, various services belonging to different component frameworks exist and operate, discounting the possibility of such homogeneity existing. In this section of thesis, the interoperability of the implemented URDS with services belonging to a framework other than Java-RMI is examined. Here, for reasons such as popularity, Grid Services [OGS02] based on the OGSA framework have been chosen.

1.1 Problem Definition and Motivation

For providing a component model based approach, UniFrame [OLS04] proposes the following concepts: a) the creation of a standards-based meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components, b) an automatic generation of glue and wrappers for achieving interoperability, c) guidelines for specifying and verifying the quality of individual components, d) a mechanism for automatically discovering appropriate components on a network, e) a methodology for the development of distributed component-based systems with service-oriented architectures, and g) mechanisms for evaluating the quality of the resulting component assemblages. To reiterate, the objective of UniFrame is to create a comprehensive framework that unifies the existing and emerging distributed component/service models under a common meta-model that enables the discovery, interoperability, and collaboration of components via generative software techniques. As
noted earlier, URDS [SIR02] is an implementation of the resource discovery mechanism; it provides an automated discovery as well as the selection of components based on their attributes. Since the URDS architecture is designed for a large distributed computing system encompassing many components and service providers, which may in the future extend in size equaling the World Wide Web, the motivation for the first part of the thesis lies in enhancing the existing URDS architecture. This is achieved by conducting empirical studies essential for determining the scalability and fault handling issues and by improving the performance of the approach by incorporating more efficient query propagation algorithms.

In the third part of the thesis, the problem that needs to be solved is to ascertain the interoperability of URDS with Grid Services. As such, in the implemented URDS prototype, though a technology independent model is proposed, all the constituent entities of URDS as well as the discovered components are based on Java-RMI. In order to validate the technology independent feature of the discovery process, interoperability aspects of URDS were evaluated with a popular framework such as Grid. Additionally, at present, there does not exist a library of most widely used components that could be integrated to compose a specific complex application for the Grid infrastructure. As a result, a preliminary component-based framework for Grid known as GridFrame was proposed for composing and integration of Grid services to form a Grid application with validation of their quality requirements.

1.2 Objectives: Statement of Goals

The specific objectives of this thesis are:

1) To empirically evaluate URDS with regard to scalability and fault handling.
2) To enhance URDS by incorporating different query propagation techniques and to design and implement a URDS simulation to empirically validate the enhanced URDS.

3) To explore interoperability aspects of URDS with Grid Services.

1.3. Overall Approach and Contributions

Previous work in the UniFrame project, namely URDS [SIR02] had established a platform independent architectural model wherein an approach for discovering services dynamically, was proposed and implemented. The approach was based on the concept of extending the native registries of different component models to be ‘active’ and ‘introspective’. But the implementation of URDS i.e. the prototype was limited by the fact that though specifications of components to be discovered are sent as queries, no comprehensive query propagation techniques or fault handling methods have been included, for query transmission between constituent entities. Only query transmission for a simplistic scenario, like transmission between individual distinct entities had been incorporated. Also, experiments are necessary to establish the scalability of the approach.

As mentioned earlier, UniFrame proposes guidelines and mechanisms for specifying and verifying the quality of individual components as well for the composed component assemblages. Clearly, success of the system generation process depends on the quality of the obtained components as well as the time taken to discover them. Hence, for URDS which is a pre-cursor to composing and integrating the discovered components, the turnaround time for obtaining the component results and the quality of the results are the foremost requirements. To fulfill these requirements, different reinforcement algorithms have been designed and implemented.

To compare these algorithms, a practical analysis on a large scale is essential. But an analysis involving the prototype on a large scale involving thousands of constituents is
practically impossible, be it in terms of the hardware or the time or the physical effort involved. Though traditionally, the formal modeling of systems has been via a mathematical model, and then attempting to find analytical solutions to problems for enabling the prediction of the behavior of the system from a set of parameters and initial conditions [WIK05], for experimentation in our case, the learning techniques use algorithms for which simple closed form analytic solutions are not possible. As a result, a Computer Simulation mirrored on the actual URDS prototype was designed and implemented.

In the next part of the thesis, experience gained in resource discovery in UniFrame i.e. by using URDS, was utilized in evaluating and exploring interoperability aspects of URDS with Grid Services. As part of the exploration process, the registration mechanism of URDS was modified to interoperate with Grid Services. Additionally, the specifications for Grid Services necessary for interoperation with URDS were explored. Subsequently, the results of the exploration were validated by a basic case study involving a matrix multiplication example. Also, there is no mature application development environment for the Grid - ad hoc approaches that were prevalent in the high-performance computing domain are used to develop Grid applications. Consequently, for addressing these challenges, a preliminary approach, GridFrame – a framework for building quality aware component based Grid systems, along with a simple case study is presented. The broad contributions of this thesis are:

- This thesis proposes and implements enhancements to the URDS architecture, by augmenting it with query propagation algorithms and ascertaining it’s scalability by experimentation.

- This thesis provides learning algorithms for increasing the performance and the quality of the obtained results from URDS. A URDS simulation has also been designed and implemented to validate the algorithms against large set of constituent entities.
- The thesis explores interoperability of URDS with Grid Services with a case study. Additionally, GridFrame – a preliminary component based approach for Grid applications, based on URDS and UniFrame, by which components can be dynamically discovered and composed on the Grid, is proposed.

1.4 Organization

The thesis is organized into eight chapters. This chapter, namely Chapter 1 provided a preliminary introduction to the problem domain of the thesis, detailing the objectives, goals and contributions. Chapter 2 contains the related approaches to each objective of the thesis. Chapter 3 describes the previous work, which is UniFrame approach and URDS principles. Chapter 4 presents the details pertaining to the scalability experiments for the URDS prototype. Chapter 5 details the learning algorithms developed for query propagation. Chapter 6 contains the design and implementation details for the URDS simulation. Chapter 7 contains the experimental results and evaluation of the learning algorithms. Chapter 8 describes the exploration of interoperability aspects of URDS with Grid Services. Chapter 9 contains the conclusions and future work for the thesis.
2. Related Work

This chapter is divided into three segments, each segment corresponding to a thesis objective as specified in Introduction Chapter. The first segment corresponds to establishing the scalability of the URDS and hence, contains related work pertaining to resource discovery. The second segment pertains to related work corresponding to profiling in resource discovery. Since the third objective is applying and evaluating UniFrame concepts in the Grid application domain, the third segment describes the current approaches for constructing component-based Grid systems.

2.1 Resource Discovery

Resource Discovery [RES] is defined as the process of locating, accessing, retrieving and managing pertinent resources from distributed, possibly heterogeneous networks. Since the information is distributed and published across networks, resource discovery involves many technical challenges. According to [RES], these challenges can be grouped into three main categories:

a) Access and Discovery of resources: Intuitive user interfaces are essential to facilitate the management of the discovery process - from the user query specification to the display of the discovered information. User access and resource discovery involve the following questions:
   i) What kind of technology is necessary for users to access resource information?
   ii) How do users specify queries to access resources?

b) Distributed Middle Layer Services: Distributed services are necessary to act as the middleware between the users and information providers, so as to provide a unified view of underlying information repositories. The mechanisms needed for storing, propagating and managing resource descriptions used by users to select and retrieve resources, are the research challenges for designing the middleware.

c) Publishing and Promotion: Resource descriptions should be simple enough for inexperienced users to fathom as well being comprehensive enough for expert users to be able to distinguish between two resources having slightly different abilities.
Since one of the aspects of this thesis deals with Scalability of discovery systems, an additional category of Scalability is added to the above list. Typically, Scalability [WIK05] indicates the capability of a system to increase performance under an increased load when resources (typically hardware) are added. Here, scalability is considered as scaling horizontally i.e. by adding more nodes to the system, such as adding a new computer to a clustered software application, or by increasing the scale of interactions between constituents of the system. In the next few sections, various related approaches are examined and evaluated with respect to the above identified challenges.

2.1.1 Lookup (Directory) Services

Lookup as defined by [MCG00] refers to the “process of locating a specific object or resource either by exact name or address, or by some matching criteria. Lookup is ‘passive’, in that it is initiated by a seeker, and requires the existence of some directory or other agent to answer the request. Lookup may be done in a statically configured environment; the directory need not be writable”. A brief description of a few prominent lookup services is given below.

2.1.1.1 Universal Description, Discovery and Integration (UDDI) Registry

The Universal Description, Discovery and Integration (UDDI) [UDD00] is a set of “specifications, which define a way to publish and discover information about Web services. The term “Web service” describes specific business functionality exposed by a company, usually through an Internet connection, for the purpose of providing a way for another company or software program to use the service.” The UDDI utilizes Web Services Description Language (WSDL), an XML grammar for resource descriptions, i.e., for describing the capabilities and technical details of the Simple Object Access Protocol (SOAP) - based web services. Companies such as Microsoft, IBM, and HP provide public nodes to host UDDI data for providing public services.

The main components in the UDDI architecture are the Web Service Providers, Service Requesters and Service Brokers. Service Providers deploy and publish services by registering them with the Service Broker. Using web-based interfaces, service requestors
find services by searching the Service broker’s registry of published services. After finding the appropriate service, they bind to the corresponding Service Provider and consume the service.

As a part of Web Services, UDDI relies upon open text-based standards (XML) for interoperability, while URDS relies on the automatic generation of glues and wrappers [ZHO04, TUM04] integrating components. Also, URDS uses the notion of “active” discovery wherein the initiative for discovery rests with the discovery service and not components, whereas in the UDDI, the onus for discovery is on the components.

Summing it up, Web Services incorporating UDDI rely on a handcrafted approach for system integration wherein the responsibility of integration lies with the application developer by means of APIs of the Web Service. Contrast this with UniFrame, wherein a comprehensive model-based technique incorporating an architecture-centric, domain-based and a technology-independent approach forms the backbone of the system integration process right from the onset. [NAT03] gives a detailed comparison of the similarities and differences between Web Services and UniFrame in general and the discovery services (UDDI and URDS) in particular.

2.1.1.2 CORBA Trader Service [OMG00]

The CORBA trading object services facilitate the publication and discovery of services of particular types. An object that supports the trading object service in a distributed environment is termed a “trader”. In other words, other objects advertise their capabilities and match their needs against these advertised capabilities through the trader. The discovery of services or matching against needs is called as “import”, while advertisement of capabilities or publicizing a service is called as “export”. Export and import facilitate the dynamic discovery of services followed by binding to the discovered services.

For exporting a service, an object gives the trader a description of the service in Interface Definition Language (IDL) format, and the location of an interface where that service is available. To import a service, the object asks the trader for a service possessing certain characteristics. The trader checks against the service descriptions it holds and sends the location of the selected service’s interface to the importer. Using the location, the
importer then accesses the service. A trader can be linked to other traders to form a federated system. The federation of traders enables the offer spaces of other traders to be implicitly available to its own clients. Although programming language independence using interface languages is provided, an implicit assumption of homogeneity still holds good, unlike URDS which tackles heterogeneity. There is no guarantee that registered objects are available with a Trader Service. The notification and security services are not provided and as a result, CORBA event services and security services need to be augmented with the Trader Service for providing these features.

2.1.1.3 X.500 and Lightweight Directory Access Protocol (LDAP) [WAH97]

The LDAP (Lightweight Directory Access Protocol) was designed to adapt a complex enterprise directory system X.500 to the modern Internet, at the University of Michigan. LDAP is a software protocol for locating resources on a network, be it the Internet or a corporate network. It defines a standard directory protocol that includes a network protocol for accessing information in the directory and an information model for defining the form and character of the information. In the informational model, the LDAP data format is structured as a hierarchy of name spaces, based on entries. An entry is used to store attributes, where attributes can have an associated type with one or more values. Every entry in the namespace has a distinguished name for identification. Data is accessed in LDAP using three fields namely the base DN that defines the position in hierarchy for beginning a search, the filter that indicates the attribute types, values and matching criteria, and finally the scope, which indicates the levels of the directory tree to be searched, relative to the base DN. LDAP is designed for fast ‘reads’ and hence frequent updates as necessary for dynamic resource discovery services result in low performance. Also, LDAP does not have an associated security model, but rather relies on other network services for security.

2.1.1.4 Domain Name System [MOC87]

Domain Name System Service (DNS) is a global network of name servers that translate host names (e.g.: www.yahoo.com) into numerical Internet Protocol (IP) addresses. The DNS protocol provides a hierarchically partitioned static database for name address
mapping. The user sends arguments such as domain names to a local agent, called a resolver, which retrieves information concerned with the domain name. Also, a user can request particular information, say a host address associated with a particular domain name, by passing an appropriate query type to the resolver. The resolver has knowledge of at least one name server. Whenever the resolver receives a user query, it queries a known name server for the information. Depending on the available information, the name server transmits back the requested information or a referral to another name server, which might have the required information.

2.1.2 Discovery Services

The spontaneous process, by which resources or services discover other resources on the network and introduce themselves to the other resources, is termed as “Discovery”. Some of the common protocols employed for service discovery are Service Location Protocol (SLP), JINI and Ninja Project. A brief description of each of them is given below.

2.1.2.1 Service Location Protocol (SLP) [GUT99]

SLP provides a framework for providing hosts with access to information about the existence, location and configuration of networked services. In SLP framework, the client applications are modeled as “User Agents”, services are advertised by “Service Agents” and “Directory Agents” provide caching of service information. The User Agent can issue service requests, on behalf of the client application, specifying the client service requirements. The request can be transmitted to Service Agents or Directory Agents. In case of Service Agents, the request is multicast upon which Service Agents receiving a request for a service unicast a reply containing the service’s location. In case of Directory Agents, the request is unicast upon which the Directory Agents respond with the list of services registered with them. Directory Agents can be discovered in two ways. In the first way, User and Service Agents on startup issue multicast service request for “Directory Agent” service. In the second way, the Directory Agent sends an unsolicited advertisement infrequently, which the User and Service Agents listen for. Some of the disadvantages of SLP are; it doesn’t specify how the services are created, it only provides a simple way of discovering services. Unlike URDS which is the precursor for integration
of the discovered services, SLP does not contain an associated means for utilizing the services and does not specify any procedure for utilizing the services by itself also.

2.1.2.2 JINI [SUN01]

JINI is a Java based framework for spontaneous discovery developed by Sun Microsystems. There are three main components in a JINI system namely Service, Client and a Lookup Service. A logical concept defined and identified by a Java interface is termed as “Service”. Clients download the publicly visible part of the service, termed as “Service Proxy”. The service registers the service object with the Lookup service. Whenever the Client requests the Lookup Service for a service, the Lookup Service returns the service object. The service object is copied to the client’s JVM and the client interacts with the service using this service object. The Lookup Service is a directory, containing a list of registered services under it. When services register themselves or clients request service from a Lookup Service, they receive a “registrar” object. This registrar object acts as proxy to the Lookup Service and runs on the service’s or client’s JVM. The proxy registrar is used in lieu of the Lookup Service for further requests. Unlike URDS containing heterogeneous components and entities, in this framework all the entities including the Client, Server and Lookup Service are developed using Java, which unfortunately discounts the possibility of any heterogeneous issues being addressed. Other drawbacks include a limited filtering mechanism as compared to other services like LDAP, CORBA Trader Service etc. Also, security in JINI is based on Java and hence weak, since cryptography is not mandatory.

2.1.2.3 Ninja Project: Using Secure Service Discovery Service (SSDS) [NIN02]

SSDS is part of the Ninja research project at University of California, Berkeley. The main components of SSDS system are Service Discovery Service (SDS) servers, Clients and Services. SDS servers multicast the multicast address for registration of services at periodic intervals. The Services listen to the SDS Server messages and multicast the description of their services to the multicast address enclosed in the SDS message. These servers cache the service descriptions of the registered services. Clients use Authenticated RMI to connect to the SDS server and submit queries in XML templates. A Certificate Authority is used to sign certificates used by the SDS for authentication of the
components in the SDS system. Again like other research work, heterogeneity is not addressed unlike URDS.

2.1.2.4 Grid Discovery Service (Globus Toolkit MDS) [GRI02]

The Monitoring and Discovery Service (MDS) incorporated in Globus Toolkit consists of two components namely the Grid Information Resource Service (GRIS) and the Grid Index Information Service. The GRIS runs on resources deployed on the Grid and is an information provider framework for specific information sources. At a higher level, the GIIS is a user accessible directory server that accepts information from child GIIS and GRIS instances and aggregates this information into a unified space. MDS also supports searching for resources by characteristics. The discovery service is mainly employed for computational resources deployed on the Grid. Unlike URDS, the performance of a query to the MDS cannot be predicted with a pre-defined formula [MDS03] and is depended on the complexity of the associated hierarchy of GRIS and GIIS.

2.1.2.5 Search Engines

A variety of Search Engines such as Lycos [MAU97], AltaVista [AVS05], Web Crawler [WCB05], A9 [SA905] are available for discovering and collecting resources on the Internet. These search engines employ computer programs such as robots and spiders, which continuously scour the Internet for resources and index them. For example, Lycos uses spiders to index information and matches query with relevant documents using techniques such as regular expression matching, vector space model, etc. Google on the other hand, ranks search results based on a patented PageRank algorithm. Search results are based on sophisticated text-matching techniques, based on the page’s content along with the popularity of the page in terms of the pages linking to it. Also, options for personalized search results using customizable profiles for users are provided.

2.2 Profiling in Discovery Systems

Profiling in general can be divided into two categories; namely, User Profiling and Peer Profiling. User profiling techniques such as GESTALT, GUARDIANS and Cheshire Project concentrate on creating user-centered profiles, based upon which searches are
structured. In user profiling, though the search results are customized based on the preference of the user, there is no profiling activity involved in the resource discovery process. In Peer Profiling, the peer nodes involved in the search mechanism are profiled and coordinate to return matching resources. Ontology-driven peer profiling and self profiled query processing are a few of the research work involved in Peer Profiling. The following sections discuss the aforementioned research work in detail.

2.2.1 GUARDIANS

GUARDIANS [ROU03] is an Information Society Technologies (IST) project aimed at user profiled research discovery. Information about users and content are described in meta-data models. Preferences, activities and preferences of users are captured in profiles, known as Generic User Profile (GUP) based on IMS Learning Information Package (LIP) specification [JON03]. Information in profiles is grouped into sections such as usage history, contact details etc. Users can select service provider, based on the Information Service Provider (ISP) profile. This profile contains an aggregated view of the information stored in the provider’s repository. The profile includes information such as name of service, description, contact details, security information, categories provided by the service provider, etc.

2.2.2 Cheshire project

The Cheshire project [LAR99] is a research work aimed at developing a discovery system capable of searching cross-domain resources hosted in multiple locations. Resources could be available in domains such as textual, numeric, spatial, etc. Here, the format for data delivery is XML while the data manipulation and navigation is through CORBA-enabled Java applets. Also Entry Vocabulary Modules (EVM) are provided that provide a mapping between the natural language of a user and the actual vocabularies used in the description of digital objects and collections.

2.2.3 Ontology-Driven Peer profiling

In this type of approach [PAR03], information and services are hosted on peers. Each peer contains an ontology-based profile containing meta-data representation, which is used as the basis for the semantic search and discovery of hosted information. Depending
on the response to queries, the profiles are updated, ensuring up-to-date knowledge about each peer. Unlike the peer management wherein the publish-and-subscribe method is used, here a query is sent to a peer based on its resources. Possible shortcomings are that if peers are explicitly dissimilar, for example some peers being wireless could affect the discovery process.

2.2.4 Self Profiled Query Processing

This approach [KAT01] proposes a mediator system for developing query compilation and execution techniques that allows for integration of autonomous and distributed participating entities. By measuring parameters such as the system performance and various environmental data, the mediator peers adapt to the changing environment. Minimizing the profiling penalty of profiling, so as to reduce the access time for obtaining profiling information and dynamic control of measured parameters are some of the challenges associated with this approach.

2.3 Component-based Approaches in Grid

In the following sections, a subset of the current approaches by which users can build Grid applications from pre-built applications are discussed in brief. A detailed discussion of these approaches and comparison with the proposed approach is contained in Chapter 8.

2.3.1 XCAT (developed by Indiana University, previously CCAT) [XCA04]

It is one of the popular frameworks that emphasize distributed computing, by providing Grid and Web Services connectivity to CCA [CCA04] based on the Open Grid Specification Architecture (OGSA) [OGS02]. Even though XCAT supports the composition of complex distributed applications on the Grid in a modular fashion, the procedure for the creation and composition of software components results in handcrafted solutions. XCAT deals with stateful components by providing Application Factories which provide pre-packaged applications consisting of a set of distributed components. On a request by authorized users, factory services instantiate all the components that are
part of the application, and connect them together to compose an instance of the distributed application.

2.3.2 CrossGrid [CRO03]
As part of the CrossGrid consortium research framework, Grid applications are composed using a client supplied workflow document containing a list of application elements and element connections with optional dispatch data. Components are built independently and registered with component registries. Flow composer components process the initial workflow document and look-up necessary components in these registries, and build an application corresponding to the document.

2.3.3 ECSF
ECSF (Enterprise Computational Services Framework) [ECS03] provides a distributed computing paradigm suitable for multidisciplinary Grid applications by connecting interdependent stand-alone applications and abstracting the details of distributed resources and underlying Grid infrastructure from the end-user.

In this chapter, related work pertaining to the objectives was discussed in brief. Since the thesis pertains to the experimental evaluation of URDS, the next chapter details the previous work on URDS.
3. Previous Work

This chapter outlines the previous work done as a part of the UniFrame project. It gives an overview of the UniFrame approach and presents an elaborate in-depth view of URDS, since the entire thesis is based on both these aspects.

3.1 UniFrame Approach

Based on Service-Oriented Architecture, the UniFrame research initiative defines a process for constructing a quality-aware distributed computing system from distributed, possibly heterogeneous components. It proposes a semi-automated, unified framework wherein each component offers a particular service and an assembly of components provides the required service-oriented architecture for a specific distributed application. The UniFrame Approach provides the overall process and is based on the Unified Meta-component model (UMM) as noted earlier. A detailed explanation and analysis of the approach can be found in [RAJ00], [RAJ01] and [RAJ01]. In the following sections, an overview of UMM along with the UniFrame process is specified.

3.1.1 Unified Meta Model (UMM)

The core parts of the UMM are components, services and service guarantees and infrastructure. The structure of these parts and their inter-relations form the innovative aspects of UMM. A particular implementation of the Infrastructure – URDS is discussed separately. A brief outline of the other two parts is given below.

3.1.1.1 Component

Components in UMM are autonomous entities whose implementations are non-uniform meaning that each component adheres to a specific distributed-component model and there is no notion of either a centralized controller or a unified implementation framework. Each component has a state, an identity, a behavior, well-defined public interfaces and a private implementation. Additionally, each component in UMM consists of three aspects:

1) Computational aspect: It reflects the task(s) carried out by each component. The tasks in turn may depend upon the objective(s), techniques used to achieve these
objectives, and the precise specification of the functionality offered by the component.

2) Cooperative aspect: It indicates the interaction with other components. This aspect may contain expected collaborators (potential components which may interact with this component), pre-processing collaborators (other components on which this component depends on) and post-processing collaborators (other components which may depend on this component).

3) Auxiliary aspect: It addresses the mobility, security and fault tolerance attributes of the component. For example, if the component is mobile, then the mobility attribute will contain the necessary information, such as its implementation details and required execution environment.

3.1.1.2 Service and Service Guarantees

In UMM, a service can be an intensive computational effort or an access to underlying resources. Since, normally there are numerous components in a DCS offering similar services, a component in addition to indicating its functionality, must be able to specify the cost and quality of service offered. The cost of each service may be dependent on the expected computational effort, resources required, motivation of the owner and the dynamics of supply and demand. The quality of service offered by a component on behalf of its owner is an indication of its confidence to carry out the required services in spite of the constantly changing execution environment and a possibility of partial failures. For specifying UMM, a Quality of Service metrics catalogue has been established for use. To illustrate this, a sample UMM specification of an ATM component is given below. This example is part of a bank account management system with services for deposit, withdraw, check balance etc.

Informal Description: Provide GUI for ATM.

1. Component Name: ATM
2. Component Subcase: ATMCase1
3. System Name: Bank
4. Domain Name: Banking
5. Computational Aspect:
   a) Inherent Attributes:
      Id, Version, Author, Date, Validity, Atomicity, Registration, Model
   b) Functional Attributes:
      Purpose, Algorithm, Complexity,
      Syntactic Contract,
      Technology, Expected Resources, Design Patterns, Known Usages, Aliases
6. Cooperation Aspect:
   PreProcessing Collaborators, PostProcessing Collaborators
7. Auxiliary Aspect:
   Security, Mobility, Fault-tolerance
8. Quality of Service:
   QoS Metrics: Throughput (operations/sec), End-to-End Delay (usec), QoS Level,
   Cost,
   Quality Level

Figure 3.1 UMM specification for ATM component

The UMM specification of a component is composed of four levels: syntactic-level, behavior-level, synchronization-level, and quality-level. This multi-level specification enables a complete description of a component.

3.2 UniFrame process
UniFrame process differs significantly from other approaches [OMG00], [EDW99] and [WEB04] by relying on an expert created generative domain model (KB), based on the Generative Programming [CZY00] paradigm. Experts from the particular domain create the KB containing the details of the distributed application under consideration. The KB [ZHI04] contains details of the software architecture of families of possible systems in terms of the constituent software components, descriptions of the component characteristics and interactions as well as the rules for the composition and decomposition of quality of the constituent components as well as the integrated system.
An illustration of the UniFrame process is given in Fig 3.2. The process can be delineated into two parts:

Component Development and Deployment Process: The process starts with a component developer formulating a UMM requirements specification indicating the functional and non-functional features of the component. Based on theory of Two-Level Grammar (TLG), the specification is refined into a formal XML based specification, similar to [UMM02] indicated earlier. This process also generates interfaces, possibly multi-level for the component. After the generation of interfaces and their implementation by the developer, the component is validated against requirements specifications. If the results are satisfactory, then the component is deployed on the network. If unsatisfactory, the UMM requirement specifications or the implementation is refined and the process repeated.

Automated System Generation: For a specific problem domain, a variety of possibly heterogeneous components will be provided by different developers. When all the components necessary for a specific distributed application are available and a

![Figure 3.2: UniFrame Approach](image-url)
requirement specified, then the components have to be integrated into a solution. The integration process can be decomposed into the following discrete steps:

a) The System Integrator specifies a system query, describing the required characteristics of the desired distributed system, in a structured form of natural language. Using domain knowledge available in the KB of the desired system, the individual components constituting the system are identified. Also, using the Composition/Decomposition model [CHA03], the non-functional requirements, i.e., QoS of individual components are deduced and a UMM description (similar to [UMM02]) of the required components are generated, as a precursor to their actual discovery.

b) Based on the UMM description, the URDS performs a search for components in the domain of the query. Based on theorem proving [TH00] and other matching techniques, existing components’ functional and non-functional features are matched against the query requirements. Matched components are then returned to the system integrator. If no matching components are found, the process is iterated over again with a possibly more relaxed set of constraints for components.

c) A list of available choices of components with their UMM descriptions are displayed to the system integrator. Once a choice of a set of components is made, say on the basis of the QoS, the system is assembled according to the generation rules available in KB. If heterogeneous components have been selected, the composition process is supplemented with appropriate adapter components; with the resultant combination of components and adapters forming a software implementation of the targeted system.

d) Using event traces and a set of test cases, Quality Validation determines whether the composed system meets or does not meet system requirements. Further details about event traces can be found in [ZHO04]. If the system requirements are not met, the steps c) and d) are repeated with a different set of matched components, or the entire process is repeated with the system query being refined to include additional information about the system requirements. The process is iteratively repeated until the functional and non functional requirements of the desired system are satisfied. Once satisfied, the resultant system is ready to be deployed.
In the previous sections, a brief description of the UniFrame approach was specified. Since all the objectives of the thesis pertain only to URDS, a detailed description of UniFrame and all its constituent entities has been avoided to concentrate only on URDS. In the subsequent sections, a detailed description of URDS is specified.

3.3 URDS

URDS is designed to provide the infrastructure necessary for discovering and assembling a collection of components for building a Distributed Computing System (DCS). The URDS architecture provides services for an automated discovery and the selection of components meeting the necessary QoS requirements specified by a System Integrator. In this URDS, new services are dynamically discovered while providing component assemblers with a directory-style access to services.

URDS is organized as a federation of Internet Component Brokers (ICB) and entities such as Headhunters, Active Registries, Services and Adapter Components. The ICBs are linked together by Link Managers (LM) with unidirectional links to form a directed graph. Every ICB has zero or more entities called Headhunters (responsible for discovering components) attached to it. The discovery process in URDS is scoped administratively implying that it locates services within an administratively defined logical domain. Domain is defined as industry specific markets such as Health Care Services, Manufacturing Services, and Financial Services etc. Organizations providing URDS services determine the domains to be supported. Figure 3.3 illustrates this hierarchical organization.

Figure 3.3 gives a broad overview of the URDS Architecture. In a more detailed view, URDS infrastructure (illustrated in Figure 3.4) comprises of the following constituents:

i) Internet Component Broker (ICB) which is a collection of the following services - Query Manager (QM), the Domain Security Manager (DSM), Link Manager (LM) and Adapter Manager (AM).

ii) Headhunters (HHs)

iii) Active-Registries (ARs)
iv) Services (S1..Sn)

v) Adapter Components (AC1.. ACn)

vi) Users (C1..Cn)

A brief look at each of them is provided below:

i) Internet Component Broker (ICB): The ICB acts as an all-pervasive component broker in an interconnected environment. It contains the communication infrastructure necessary to identify and locate services, enforce domain security and handle mediation between heterogeneous components. These services are reachable at well-known addresses. ICBs provide the following functionalities:

1) Authentication of Headhunters and Active Registries and enforcement of access control over domain multicast addresses. (Domain Security Manager)

2) Mediation between Users and Service Providers. (Headhunters and Query Manager).

3) Mediation between components of different component models. (Adaptive Manager)
a) Domain Security Manager (DSM): The DSM serves as an authorized third party that handles the secret key generation and distribution and enforces group memberships and access controls to multicast resources through authentication and use of access control lists (ACL). DSM has an associated repository (database) of valid users, passwords, multicast address resources and domains.

b) Query Manager (QM): The purpose of the QM is to translate a system integrator’s natural language-like query into a structured query language statement and dispatch this query to the Headhunters available in the domain of the query which then return the list of service provider components matching the search criteria expressed in the query. Requests for service components belonging to a specific domain are dispatched to Headhunters belonging only to that domain. The QM, in conjunction with the LM, is also responsible for propagating the queries to other linked ICBs.
c) Link Manager (LM): The LM serves to establish links with other ICBs for the purpose of federation and to propagate queries received from the QM to the linked ICBs. The LM is configured by an ICB administrator with the location information of LMs of other ICBs with which links are to be established.

d) Adapter Manager: The AM serves as a registry/lookup service for clients seeking adapter components. The adapter components register with the AM and while doing so they indicate their specialization, i.e., which component models they can bridge efficiently. Clients contact the AM to search for adapter components matching their needs.

ii) Headhunter (HH): The Headhunters perform the following tasks: a) Service Discovery: detect the presence of service providers (Exporters), b) register the functionality of these service providers, and c) return a list of service providers to the ICB that matches the requirements of the component assemblers/system integrators requests forwarded by the QM. The service discovery process performs the search based on multicasting. The Headhunter stores the UniFrame specification information of exporters adhering to different models in Meta Repositories. These repositories are implemented as standard relational databases.

iii) Active Registry (AR): The native registries/lookup services of various component models (RMI, CORBA, .NET, etc.) are extended to be able to listen and respond to multicast messages from the Headhunters and also have introspection capabilities to discover not only the instances, but also the specifications of the components registered with them.

iv) Services (S1..Sn): Services (Components) are implemented in different component models (RMI, CORBA, etc.,). They are identified by the service type name, component’s informal UniFrame specification and QoS values for that service. The informal specification is a XML specification outlining the computational, functional, co-operational and auxiliary attributes of the component. The interfaces of the component are registered with its local registry. One method in the interface returns the URL of its informal specification. Each component is specific to its domain, i.e., it adheres to the standards of the domain.
v) Adapter components (AC1..ACn): Adapter components serve as bridges between components implemented in diverse models, say for instance RMI and Corba.

vi) Clients (C1..Cn): Clients may be Component Assemblers, System Integrators or System Developers who may be searching for services matching certain functional and non-functional requirements.

In this chapter, a brief explanation of UniFrame was specified in conjunction with a detailed discussion of URDS. Based on these concepts, the various objectives of the thesis are discussed and completed in the succeeding chapters.
4. Enhancements in URDS

This chapter describes the investigations related to the first thesis objective. As indicated earlier, the first objective is explored by enhancing the preliminary version of the URDS by incorporating features related to scalability, fault handling and query propagation.

4.1 Scalability

As indicated in the chapter 1, the preliminary prototype of URDS was limited to a simple configuration containing a small number of entities, e.g., it contained only a single HH, a single AR, and one QM. Furthermore, this prototype did not incorporate fault handling and query propagation techniques. As the preliminary prototype was created to study the feasibility of the URDS approach, such a small size was justifiable. However, as URDS is expected to discover components in a large networked environment, it is necessary to investigate the scalability of the URDS architecture and hence, the preliminary prototype was enhanced (by incorporating fault handling and query propagation techniques) and empirical experiments were carried out to assess its performance in a larger configuration. Scalability [WIK05] is defined as the capability of a system to increase performance under an increased load with the possible addition of resources. As a result, to ascertain the scalability, the performance of the URDS system was experimentally studied by varying the number of entities and by increasing the load i.e. issued queries. The following sections, starting with a brief overview of the sequence of activities in URDS, describe these enhancements and experiments.

4.1.1 Overview of the resource discovery process

In the URDS architecture, when a Principal (Headhunter (HH) or Active Registry (AR)) is created, it contacts the Domain Security Manager (DSM) with it’s authentication credentials in order to obtain the secret key and multicast address for further communication. The DSM authenticates the principal and checks its authorizations against the domain Access Control List (ACL). The DSM returns a secret key and a multicast address mapped to the corresponding domain, if the principal is authentic. In case the principal is a Headhunter, the DSM registers the contact information i.e. the identification
of the Headhunter with itself. The QM to propagate queries uses this information. On being issued a query, the QM generates a structured query language statement. The list of HHs is then obtained from the DSM by the QM and the query is dispatched to the appropriate HHs belonging to the domain of the query. Each HH then returns a list of matching service providers from the entries in its its Meta-repository. The ICBs propagate the search query issued by the Clients (System Integrators) to other ICBs to which they are linked apart from the Headhunters with which they are associated. The LM performs the functions of the ICB associated with establishing links and propagating the queries. Links represent paths for propagation of queries from a source ICB to a target ICB.

After authentication, Headhunters periodically multicast their presence to multicast their addresses, which were prior issued by the DSM. Active Registries, which are listening for the multicast messages from the Headhunters at this group address, respond to a Headhunter’s multicast messages by passing their contact information to the Headhunter. Headhunters query the Active Registries that respond to their announcements, for the UniFrame specification information of all the components registered with them. The Active Registries respond by passing to the Headhunter the list of matching components registered with them along with the detailed UniFrame specification of the components. Each Headhunter stores the component information obtained from the Active Registries onto its Meta-Repository. Headhunters use the Meta-Repositories to retrieve component information on being queried by the QM or other Headhunters. System Integrators contact the QM and specify the functional and non-functional search criteria. If heterogeneous components, out of the matching subset, are selected by the System Integrator to build a system, the AM is contacted to search for adapter components matching the needs. The AM checks against its repository for matches and returns the results, which are used to build the system.

4.1.2 Query Routing

As an initial step (more sophisticated techniques that are incorporated are described in the Chapter 5), a simple query propagation technique, based on a random selection of HHs to
forward the query, is incorporated in the enhanced version of the URDS prototype. This propagation technique is described below.

Figures 4.1 - 4.4 indicate a typical communication pattern that occurs, while processing a query received from the System Integrator (SI), when a propagation technique is incorporated in the URDS process. The SI contacts an authenticated QM (deployed at a known address) and supplies the query. When the QM accepts the query, it contacts the DSM, which provides it with a list of the authenticated HHs from that domain. For discovering components, unlike the preliminary prototype, the QM does not exhaustively
query each HH in the system, instead, choosing to query a single HH termed the Primary Headhunter (PH), which then directs and replicates the query to the remaining HHs in URDS.

In this thesis, this query propagation technique is termed as “Brokering” or “Delegation” At the onset, the QM delegates the responsibility of discovering components to the PH. The PH is responsible for the further transmission of the query and the retrieval of results back to the QM. The PH checks its local Meta-repository and finds available components that match the query. The PH then selects a random (arbitrary) subset of the remaining Headhunters, delegating each Headhunter a list containing a portion of the remaining Headhunters along with the query to be transmitted. The portion allocated is a ratio of the remaining Headhunters to the number of Headhunters in the chosen subset. Each of the subset Headhunters is a PH and is responsible for transmission of the query among the list of Headhunters allocated to it and retrieval of the results back to the Headhunter, that spawned it. The transmission of the query includes selecting a subset of Headhunters and the passing of the remaining Headhunter list and query to the subset. The PH finally combines the results before sending it back to the QM. Figures 4.1- 4.4 illustrate this process with a specific scenario. In Figure 4.1, the QM selects HH3 as the PH from the list of Headhunters. HH3 then selects a random subset, in this case HH1, HH2 and HH4, as the next level of PHs and passes to them the Headhunter subspace in which their transmission of queries should take place (Figure 4.2). Since there are three HHs in the subset and three remaining HHs, each HH in the subset gets 3/3 equals one HH as its Headhunter subspace. As illustrated in Figure 4.3, the Headhunters HH1, HH2 and HH4 transmit the query and retrieve the results from Headhunters HH5, HH6 and HH7 respectively. The complete traversal of the query and retrieval of results among the Headhunter space is shown in Figure 4.4. Thus, the Headhunter search space takes the form of a tree with the PH (randomly selected by the QM) being the root and the subset chosen by it as its children and the subset in turn chosen by each member of the subset as its grandchildren, and so on. Also, when the QM gets a query, the query is assigned a number QueryID that is passed along with the query. This is to avoid repeated processing of the same query by any HH. Each of the HHs maintains a Query List containing the queries already processed by it. So when a HH selects a subset, it checks whether any of
the subset HHs has already processed the query. If it has been processed, then the query is not transmitted to that HH. Whenever the MR of an HH is updated, the entries in the Query List are purged. This ensures that stale results are not retrieved.

The Headhunter search space can be limited by the SI’s requirements on time and number of results requested. If the emphasis is on the best available components, then the Headhunter search space can be extended to all the HHs present in the domain, ensuring that the SI gets a set containing all the available components matching his query. The SI can also specify time limits to curtail the number of HHs involved in the resource discovery process. All the components found before the time limit expires are returned. This is particularly useful in cases, where the SI is interested in finding one or more solutions to the query, but not necessarily the best solution.

4.1.3 Experimental Environment and Results

The enhanced prototype (implemented by adding the propagation technique to the preliminary prototype) was empirically tested. This prototype was implemented using the J2EE version 1.4 software environment. The DSM, QM, HHs and ARs were implemented as Java-RMI services. The repositories used in the URDS, namely DSM’s repository and Headhunter’s Meta repository, were implemented using Oracle version 8.0. The Web tier components used for servicing client interactions were placed in a Tomcat 4.0 Servlet/ JSP container. HHs and ARs communicate with each other using Multicast sockets using UDP/IP, while the unicast communication between HHs is achieved using JRMP (Java Remote Method Protocol). The DSM and HHs establish their Database connections using JDBC (Java Database Connectivity) APIs. The SI interacts through a browser front-end using HTTP protocol. The security infrastructure of URDS is implemented using Java Standard Cryptographic Architecture and Extension frameworks.

Experiments were carried out on this prototype with regard to Scalability. The infrastructure used in this experimentation consisted of Sun SPARC machines connected by an Ethernet 10 MBPS.

4.1.3.1 Client Query Result Retrieval Time (CQRRT)

In the discovery process, components are discovered and returned in finite time based on requirements specified in a query. The finite time associated with a query is a major
characteristic of any changes in environment such as changes in number of HHs. As a result, in the series of experiments, turn-around time known as client query result retrieval time (CQRRT) is used as the metric. CQRRT is the time taken from the point of issue of query by the Client to the point of retrieval of results back to the Client. Since ascertaining the scalability of the system is the objective of this chapter and scalability is the capability of the system to increase performance under increased load, here the number of Headhunters, Active Registries, and Components were varied with and without increased load i.e. issued queries and their effects on CQRRT studied.

To study the effect of increase of these variables on CQRRT, the chain of events resulting from the issue of a query have to be studied and are referenced in the subsequent parts of the chapter. Once the SI issues a query to the QM, the following sequence of actions takes place:

1) The QM determines the domain of the query and contacts and retrieves from the DSM the list of Headhunters pertaining to the domain of the query.

2) The QM selects a PH from the list of Headhunters for further transmission of the query.

3) The PH selects a subset of PHs from the list of Headhunters and allocates a portion of the remaining Headhunters to each.

4) Transmission of query and merging of results take place as explained earlier. The PH then transmits the final results back to the QM.

5) The QM in turn transmits the results to the SI.

4.1.3.1.1 Increasing the number of available components

Consider the situation wherein only a single HH is present. In that case, the chain of events after issue of the query would include the QM obtaining contact information of the HH from the DSM, the QM transmitting the query to the HH, the HH searching its local meta-repository and finally, the transmitting of the results back to the QM and then to the SI.
An increase in the number of available components matching the query would lead to increase in size of the meta repository. Assuming the time taken for the other events to be the same for each query (since the same steps are followed), the only difference to the CQRRT would be searching a larger meta-repository and retrieval of an increased number of results. This search time is omnipresent in all discovery systems employing a directory structure and hence unavoidable. Figure 4.5 shows a graph of component increase versus CQRRT wherein increase in number of components results in proportionate increase in CQRRT.

4.1.3.1.2 Increasing the number of Active Registries

Depending on the multicast addresses assigned to the new ARs, the HHs using the same multicast address will have to spend more time in contacting the new ARs and getting component information from them. On an unrelated note, a new component might want to register with a particular AR with a large number of components rather than a new AR, acting on the presumption that other components would not have registered with the latter AR, if the AR did not offer good visibility for components. An increase in number of ARs will be ideally invariant on CQRRT since the chain of events from the issue of a query, as explained in Section 4.1.3.1.1 does not involve Active Registries. But, when a query is
issued, HHs might be in the midst of contacting ARs for components. Increasing the number of ARs increases the chances of HHs being in this state. As a result, Figure 4.6 presents a graph of CQRRT versus number of ARs, in which it can be observed that with increase in number of ARs, there is a gradual increase in CQRRT.

![CQRRT vs No of Active Registries](image)

Figure 4.6 Increase in number of Active Registries

4.1.3.1.3 Increasing the number of Headhunters

The chain of events as explained in Section 4.3.1 Steps 1 – 5, involves HHs in steps 3 and 4 respectively. The transmission of query and retrieval of results with increase in the number of Headhunters will involve a proportionate increase in CQRRT. This can be attributed to the fact that the HH space for query propagation and retrieval of results increases with increase in number of Headhunters, i.e., the depth of the propagation tree increases. In an experiment, up to 29 Headhunters were run, each on different Sun SPARC machines and their effect on CQRRT studied. Figure 4.7 shows the graph, where it can be observed that there is a linear increase in CQRRT with increase of Headhunters.
4.1.3.1.4 Increasing the number of Query Managers

If multiple SIs issue multiple simultaneous queries to different QMs (with common HH space), there will be a marked increase in traffic among the HHs for the transmission of queries and retrieval of results. Similar to that situation, in this experiment up to 19 Clients issue 10 queries each simultaneously to 10 QMs. At the onset, a single Client issues 10 queries to a single QM and records the average CQRRT for each query. Subsequently, the number of Clients is incremented by one with simultaneous increases in number of QMs and the average CQRRT recorded for a Client. A graph of the average CQRRT for a Client is recorded as the number of Clients is increased from 1-19 and number of queries ranging from 0-190. As the number of simultaneous queries in the system keeps increasing, the load on the QMs and HHs keeps increasing and hence the turnaround-time for retrieval of results, i.e., CQRRT shows a similar trend.

Figure 4.7 Increase in number of headhunters
The effects of increasing the number of available components and HHs in isolation are shown in Figure 4.5 and Figure 4.6 respectively. If both these entities are increased
simultaneously, the issue of whether the increase in CQRRT implies a summation of the individual increases in time is to be determined. An experiment was conducted to address this issue. In this experiment, the system contained 0-20 components, 1-20 HH, 2 AR, 1 QM and 1 SI. The number of Components was varied for various instances of number of Headhunters. To address the above issue between a wide range of HHs, 1HH, 10 HH and 20 HH were chosen as the number of HHs. Figure 4.9 was compared to the summation of the individual increase in HHs and increased components graphs (shown previously in Figure 4.5 and Figure 4.7). This suggests that CQRRT is indeed a summation of the individual increases in time.

4.1.3.1.5 Increasing the number of Active Registries and Components

As seen before, an increase in number of Active Registries brings about only a negligible change in CQRRT, while increase in available components leads to a linear corresponding change in CQRRT. The summation of the effect of these entities can be seen in Figure 4.7. The configuration of the system was 1HH, 1 QM, 1 Client, 0-1 AR, 0-1 Component.

![Increase in Registries and Resources](image)

Figure 4.10 Increase in registries and resources

The initial configuration of the system was 1HH, 1 QM, 1 Client. At point 5 in the graph, 1 Active Registry was added, which as explained before gives a small difference in CQRRT. At point 6, a Component was added into the system, which increases the Headhunter Meta-
Repository search time and thus the CQRRT. Depending on the resources to be added, registries can be introduced into the system. The ratio of registries to resources can be fine tuned, based on experimentation and trials. It can be observed that the graph is a summation of the graphs of individual increases in ARs and Components.

From the series of experiments, it can be observed that there is a proportionate increase in CQRRT with increasing number of entities. For a system that does not scale well under increased load, be it increased number of entities or queries, an exponential increase is expected at the system breakdown point, i.e., there should be exponential increases until a breakdown point at which the system cannot process more queries and breaks down. In the set of experiments, the set of graphs show a increasing trend, but none of them indicate a exponential trend. As a result, it can be concluded that the URDS system is scalable.

4.2. Fault Handling in URDS

Once the scalability of the system had been ascertained, the URDS prototype was enhanced by designing and implementing fault handling techniques. For any large system, fault handling techniques are essential as the probability of failures of entities is very high. Even though comprehensive fault tolerance techniques are ideal, in their absence, some fault handling techniques are essential for the smooth working of the system without interruptions. For each architectural component in URDS, fault-handling techniques have been designed and implemented. The following sections give a description of each of the fault handling techniques employed in URDS.

4.2.1 Domain Security Manager

The fault handling in the case of a DSM is achieved by replication. The functionality of the DSM is achieved by two instances of a DSM – called as the primary and secondary DSM. These two DSMs have different, but well known, addresses. Entities requiring authentication from the DSM contact the primary DSM. Whenever the Primary DSM authenticates an entity, it passes a copy of the information to the Secondary DSM. If the Primary DSM fails at the onset before authentication of any entity, or after authenticating some entities, the new entities contact the Secondary DSM for authentication, after
timing out on trying to contact the Primary DSM. Old entities do not need to re-register
themselves, since the Secondary DSM has a copy of the information of the Primary
DSM. To ensure consistency of information, at periodic intervals both the DSM exchange
information with check pointing information. Thus, the secondary DSM provides a fall
back option to improve the security of the URDS and ensures that the registration
information is not lost.

4.2.2 Active Registry

The DSM regularly pings each of the ARs in its list and removes any unavailable
registries from its list. Also, the DSM informs about this removal to all the HHs on the
same multicast address as that of the failed registry. This step is essential to ensure that
HHs contacting the failed AR can purge the information about these components that
were supplied by that registry. Non-available components are thus removed from the
URDS.

4.2.3 Headhunter

A failure of HHs can occur at two levels:

a) At the Primary Headhunter (PH) level, and

b) At the subset level.

When failure occurs as per the former case, i.e., if the PH selected by QM fails on a
timeout, then the QM catches the exception and then removes the HH from the list and
then again chooses a random PH. The same technique is employed if the second PH also
fails and so on until an active HH is found. If no active HHs are found, the QM returns to
the SI indicating failure of the discovery system.

When failure occurs as per the second case, one of the children of the failed HH takes
over its responsibilities, i.e., transmission of query and retrieval of results. Consider the
same example used in explaining query propagation. In Figure 4.11, HH3 has been
chosen as the Primary Headhunter (PH). HH3 has chosen HH1, HH2 and HH4 as the
next level of PHs. The Headhunter subspace, i.e., children for each of them is just one
Headhunter HH5, HH6 and HH7 respectively. If any Headhunter HH1, HH2 or HH4
fails, then the corresponding child(ren) takes over the role vacated by the failed Headhunter.

Fig 4.11 Fault Handling in case of HHs

If there are no children, the implication being that the Headhunter subspace is empty, then there is no need to take any action. In other words, the failed Headhunter did not have any responsibilities to delegate. If the number of children is more than one, one among them is chosen randomly to take the place of the failed parent HH. This fault handling technique ensures that failure of a HH at any position in the Headhunter search tree does not result in dropping of a query and retrieval in the Headhunter subspace under that failed HH. Another advantage of this technique is that by a combination of the techniques, multiple failures of HH can be taken care of at multiple layers in the Headhunter search space.

4.2.4 Query Manager

Two possible scenarios are tackled in the case of failures related to a QM. The solutions for these scenarios are indicated below:

a) A SI is trying to contact an inactive QM: Since it gets an exception, it gets to know immediately.

b) A SI has issued a query to the QM and then the QM fails: the SI waits for a specific time (TTL –Time to live fixed at its discretion) and then tries again. If the QM fails before the PH has returned the results, the PH retains the results.
4.2.5 Components
Since the services offered by available components changes dynamically, the ARs periodically call on the components to obtain updates on the services offered by them. If a component failure occurs, then the ARs purge the corresponding entries and also inform the communicating HHs about the component failure. These steps ensure that failure of components is detected almost at once and a failed component is never returned as a result for an input query.

4.2.6 System Integrator
If the SI fails after transmission of a query, the QM retains the results of the query for a specific period of time (TTL) (fixed at its discretion or based on the query timestamp). This ensures that stale results are not returned to the SI, if the SI starts running again. Some of the afore-mentioned techniques like timing out, replication, etc., are standard fault handling techniques adopted in distributing computing systems. For the fault handling scenario in HHs, an experiment was carried out to test the functionality of the implemented technique. For this experiment, the same experimental setup as in the previous experiments was used.

4.2.7 Testing of Fault Handling in Headhunters
To test fault handling in Headhunters, an experiment was conducted that involved killing of HHs and reviving them shortly. Thereafter, the values of CQRRT in these stages were noted down. In the graph in Figure 4.12, it can be observed that there is a marked dip in CQRRT at points 7 – 10 respectively. One HH was killed at point 8 and revived at point 11. After point 11, the graph shows a sharp increase and then reverts back to the state before the killing of the HH.
From the graph, it can be concluded that the fault handling functionality in the HHs is properly implemented.

In this chapter, as a prelude to ascertaining the scalability of the URDS, algorithms for propagation of queries were designed and implemented. Consequently, the scalability of the URDS prototype was established by empirical evaluation. Later, fault handling techniques for each entity of URDS were discussed, implemented and validated by experimentation.
5. Profiling Algorithms

In the previous chapter, the scalability of the URDS prototype was empirically demonstrated with a series of experiments. Subsequently, the next endeavor of the thesis was to explore new approaches for improving the performance of the discovery process. The performance of the discovery process could be measured in terms of the turn-around time for a query, the quality of obtained components for a query or the combination of the two. For any significant improvement, it can be argued that any progress in achieving lesser turnaround time is meaningless unless the quality of the obtained components is on par with that of the previously discovered components. The same reasoning holds for the other, i.e. quality of discovered components. Therefore, for any major advancement in the performance, the discovery process has to be improved both in terms of the turn-around time as well as the associated quality of discovered components.

As noted before, the discovery process consists of a series of steps involving the propagation of queries and results to and fro between HHs and QMs. As the scalability experiments indicated, any increase in the number of these entities leads to a larger search space and so increases the turn-around time for a query. Therefore, even though a large number of entities is currently available, a limited subset should be contacted for any improvement in the turn-around time for the discovery process. As noted before, the improvement has to be accomplished without compromising on the quality of obtained results. The determination of the limited subset is the next question to be addressed in this thesis. The other concern is the ascertaining of the entity or entities determining the subset. Since QM and HHs are directly involved in the query propagation process, the propagation process would entail either or both of them determining the subset.

Any determination of entities to be included in the subset would entail learning from the previous experiences. If both QMs and HHs are involved in the determination, both of them have to maintain a ‘profile’ indicating the past experiences dealing with HHs. These profiles would serve as guides for them to choose ‘m’ good HHs from the list of ‘n’ available HHs, with the condition that results obtained from the ‘m’ HHs are comparable with the results obtained from all ‘n’ HHs.

If QMs and HHs do maintain profiles, the next issue to be addressed would be the length of the profiles, i.e., the amount of the past experiences to be stored, the type of
experiences to be stored, i.e., whether direct experiences or indirect experiences or both should be saved. Direct experiences refer to interactions of the entities maintaining the profiles with responders, while indirect experiences refer to indirect interactions between the respondents and the profile maintaining entity. Past experiences could possibly be restricted to short term, i.e., maintaining information with regard to the previous direct experience or long term, i.e. maintaining information with regard to all previous direct experiences or reinforcement i.e. maintaining information with regard to direct as well as indirect experiences. This chapter discusses a short term algorithm, a long term algorithm and a reinforcement algorithm based on [MUK03] and compares them for their effectiveness and performance.

5.1 Background

In the context of the thesis, Profiling [BAR05] is formally defined as a technique that develops a pattern based on the history of events and applies a selection criterion for the future events, thereby improving efficiency. Profiling is a complex task and has to maintain a delicate balance with the amount of information to be stored. The profile information should be generic, maintaining a compromise between storing too little and too much information. In this thesis, profiling is based on an iterative learning process [MUK03] in which an entity facing a problem learns 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. Based on the performed action, a reward or penalty is typically assigned from an outside entity such as an user. Here, the quality of the response, i.e. the rating (described later) of the discovered components determines the corresponding reward or penalty.

One domain where the profiles and learning algorithms are used is the information filtering domain for filtering out unwanted results. At first sight, it might appear that profiles in resource discovery are identical in their usage. In the traditional information filtering domain, documents are classified into different classes and ranked on their relevance to the class domain. Based on the user specifications, the corresponding classes are identified and individual documents ranked on their proximity to the specifications by
means of frequency of words, context, class ranking etc. All these aspects are loosely
applicable to resource discovery of software components too. Except here, as opposed to
static text based documents, software components are dynamically discovered and their
specifications matched against user requirements. Unlike the text matching process
employed in the filtering domain, software resource discovery relies on component
behavior matching both in terms of syntax as well as the associated semantics. In URDS,
components are specified using multi-level contracts, based on the Quality of Service
catalogue [UMM02] as mentioned earlier. An example specification of an
AccountDatabase component is shown below.

I. Syntactic Contract

1. Component Name: AccountDatabase
2. Component Subcase: DatabaseCase1
3. Domain Name: Bank
4. System Name: Basic Bank
5. Informal Description: Provide database service for accounts.
6. Computational Attributes:
   6.1 Inherent Attributes:
      6.1.1 id: N/A
      6.1.2 Version: version 1.0
      6.1.3 Author: N/A
      6.1.4 Date: N/A
      6.1.5 Validity: N/A
      6.1.6 Atomicity: Yes
      6.1.7 Registration: N/A
      6.1.8 Model: N/A

7. Deployment Attributes: N/A

II. Semantic Contract

1. Functional Attributes
   1.1 Function description: Act as database server for accounts in system.
   1.2. Algorithm: N/A
   1.3 Complexity: N/A
1.4 Interface Contract
   1.4.1 Provided Interface: IStore
   1.4.2 Required Interface: None

1.5 Technology: N/A

1.6 Expected Resources: N/A

1.7 Design Patterns: NONE

1.8 Known Usage: NONE

1.9 Alias: NONE

2. Cooperation Attributes:
   2.1 Preprocessing Collaborators: CashierTerminal
   2.2 Postprocessing Collaborators: NONE

3. Auxiliary Attributes:
   3.1 Mobility: No
   3.2 Security: L0
   3.3 Fault tolerance: L0

III. Synchronization Contract
   1. Synchronization policy: Mutual Exclusion

IV. Quality of Service Contract
   1. QoS Metrics: throughput, end-to-end delay
   2. QoS Level: N/A
   3. Cost: N/A
   4. Quality Level: N/A
   5. Effect of Environment: N/A
   6. Effect of Usage Pattern: N/A

Figure 5.1 A UMM specification for AccountDatabase Component

For a query, each contract level of the above specifications is matched sequentially to
determine a match. [Amy] is considered as the basis for matching software specifications.
It relies on formal specifications in terms of pre and post conditions, written as predicates
in first-order logic. Based on these conditions, matches are determined using theorem...
proving. At the first level, namely syntax contract, specifications such as name, domain name, etc, are matched with the query. Likewise, semantic, synchronization and QoS contracts are matched. Further details of the matching process can be found in [ANJ04]. The final outcome of the rating process is a number indicative of the precision of the component’s match with the query. In the following sections, different algorithms incorporating the above concepts are explored and compared.

### 5.2 Algorithms

Before the design of the algorithms, the main issue to be addressed is the necessity of profiles. To address this issue, two algorithms namely random and short term were designed without profiles. In case of the random algorithm, no profiles are maintained by any entity, with requests being forwarded randomly each time. The short term algorithm relies on a query forwarding strategy in which HHs exchange places depending on the rating of their components in relation to the query. To perceive the benefits of profiling, two algorithms, namely long term and reinforcement based on [MUK03], were designed.

<table>
<thead>
<tr>
<th>Concept of Recommended HHs</th>
<th>Random</th>
<th>Short Term</th>
<th>Long Term</th>
<th>Reinforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Profile</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Profile Increment</th>
<th>Random</th>
<th>Short Term</th>
<th>Long Term</th>
<th>Reinforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed quantity</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record Direct Experiences</th>
<th>Random</th>
<th>Short Term</th>
<th>Long Term</th>
<th>Reinforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Record Indirect Experiences</th>
<th>Random</th>
<th>Short Term</th>
<th>Long Term</th>
<th>Reinforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1. Comparison of Algorithms
A brief conceptual comparison of the algorithms is listed in Table 1. The random algorithm does not maintain a profile or the concept of recommended HHs, but relies on randomness as a query forwarding strategy. The short term algorithm does not maintain a profile, but relies on recommended HHs as a query forwarding strategy, while the long term algorithm relies on both profile and recommended HHs. The reinforcement algorithm maintains and updates a profile based on direct as well as indirect experiences. Further details of these algorithms and the non-profiling algorithms are brought forth in the subsequent sections.

5.2.1 Random Algorithm

The first strategy employs a random algorithm, wherein on receipt of a query, a QM contacts the DSM and obtains a list of authenticated HHs in the domain of the query (HHList). For discovering components, the QM does not exhaustively query each HH in the system, instead, among the available HHs, it picks an arbitrary number of random HHs as its acquaintance list [MUK03] (for sake of explanation, in here and other algorithms, three is chosen as the number) and provides each HH with the query. Each of the HHs checks its local Meta-repository and finds available components, matching the query. Matching components are then rated, according to the matching process [ANJ04]. Since better rated components might still be available elsewhere, the HH then contacts three distinct HHs from the HHList to form its acquaintance list. This process continues iteratively until the HHList is empty.

If a HH propagates a query to a HH which has already processed the query, it tries to contact a non-visited HH (with the help of DSM) for obtaining the results. This step is undertaken to ensure that given enough time for a query, all possible headhunters will be contacted (exhaustive search). Each contacted HH is responsible for transmission of the query among the list of HHs in its acquaintance list (unless of-course all the acquaintances had previously processed the query and there is no non-visited HH for the query) and the retrieval of the results back to the HH that spawned it. On the reception of results, a HH caches all these results along with results obtained from its MR and sends them back to the caller HH and so on until the QM acquires the summation of results. In
short, the propagation takes the form of a single parent tree, with the QM at the root and interior nodes (HHs) having three children (acquaintances).

5.2.1.1 Algorithms

In the following sections, algorithms describing the behavior of the random algorithm are specified. Here, only the behavior of the algorithm is specified and algorithms for external behavior of the entities, for example HH initializing, multicasting etc. are specified in Chapter 6.

5.2.1.1.1 Client

The Client is an external agency such as the System Integrator or a System Generator issuing queries.

*Algorithm for displaying obtained results:*

CLIENT_INPUT_RESULT
IN: result
    DISPLAY results
END CLIENT_INPUT_RESULT

5.2.1.1.3 Domain Security Manager

At the onset, the DSM issues random non-contacted HHs to requesting HHs for forming the search space of the query. To avoid looping and closed set of HHs, if a HH cannot contact a child HH in its list for reasons such as query already been processed by the child HH, the DSM if possible issues another HH which was not contacted earlier for that query.

*Algorithm for issuing random HHs:*

DSM_GET_RANDOM_HHS
IN: queryid
OUT: listofhhs

// random_hhList maintains the list of uncontacted HHs for the query
listofhhs = GET three randomly chosen HHs from random_hhList
DELETE hhs in listofhhs from random_hhList
return listofhhs

END DSM_GET_RANDOM_HHS

5.2.1.1.3 Query Manager

On issue of an unprocessed query, the query is stored in a queue and QM obtains a list of random HHs from the DSM for the query. Queries are sent to the HHs in the list and subsequently discovered results are combined and sent to the issuer (Client).

Algorithm for input of query:
QM_INPUT_QUERY

ADD query to query_list

END QM_INPUT_QUERY

Algorithm for processing a query:
// Called when the queue of unprocessed queries is not empty
QM_PROCESS_QUERY

IN: query

hhList = CALL DSM_GET_RANDOM_HHS

END QM_PROCESS_QUERY

Algorithm for propagating a query:
// Called when a unprocessed query is available
QM_PROPAGATE_QUERY

IN: hhList, query

FOR each element hh in hhList

CALL HH_INPUT_QUERY (query)
5.2.1.1.4 Headhunter

When a previously unprocessed query arrives, it is stored in a queue. As noted before, each time a query is issued, the HH contacts the DSM for random HHs. On arrival of results, if the number of contacted HHs is equal to the number of arrived HHs, the results are combined and sent back to the issuer entity, QM or HH.

Algorithm for input of a query: Here, the query propagation is asynchronous. New queries are stacked on to a queue.

HH_INPUT_QUERY

IN: query

ADD query to query_list

END HH_INPUT_QUERY

Algorithm for processing a query:

HH_PROCESS_QUERY

IN: query
hhList = CALL DSM_GET_RANDOM_HHS (query.ID)
CALL HH_PROPAGATE_QUERY (hhList, query)
END HH_PROCESS_QUERY

Algorithm for propagating a query:
HH_PROPAGATE_QUERY
IN: hhList, query
    FOR each element hh in hhList
        CALL HH_INPUT_QUERY (query)
    END FOR
END HH_PROPAGATE_QUERY

Algorithm for storing result:
HH_STORE_RESULT
IN: result
    ADD result to resultList
    // called_list maintains the list of contacted HHs
    // arrived_list maintains the list of arrived HHs
    no_of_called_HHs = GET number of called HHs from called_list
    no_of_arrived_HHs = GET number of arrived HHs from arrived_list
    IF no_of_called_HHs is equal to no_of_returned_HHs
        combined_result = CALL HH_COMBINE_RESULT (query.ID)
        // Get the entity which has transmitted the query
        parent = GET parent from queries_parentList
    END IF
    IF parent is QM
        CALL QM_STORE_RESULT
    END IF
    ELSE
        // Call store_result of parent HH
        CALL HH_STORE_RESULT
END ELSE
END HH_STORE_RESULT

Algorithm for combining results:
HH_COMBINE_RESULT
IN: queryid
OUT: listofresults
    listofresults = GET list of results for queryid
    own_results = GET own results for queryid
    // add own_results to the listofresults
    ADD own_results
    return listofresults
END HH_COMBINE_RESULT

5.2.2 Short Term

The second algorithm is a short term learning-based algorithm. Initially, since a QM does not have any past experiences with any HHs, the previous random strategy is employed with the QM contacting three random HHs and each of these HHs, three other HHs, and so on until each HH in the HHLList has three acquaintances or the HHLList is empty. Once a query is processed by a HH, it sends along with the result the name of a recommended HH (under its propagation tree) which had the best rated component(s). If no HHs were available for contact, the HH recommends itself to its parent HH. The parent HH then updates its acquaintance list by replacing each called HH with its recommended HH (in this case, the same) and then compares the obtained results (if the child HH has returned any results) with its own components. If the obtained results are superior, the parent HH exchanges places with the HH possessing the superior result(s). Corresponding acquaintance lists are also exchanged as part of the change. If obtained results are not superior, status quo is maintained. Also, before the exchange, the parent HH returns results to its own parent HH along with the recommended HH (lets say HH X) which then becomes part of the grandparent acquaintance list. A similar comparison
process takes place with the grandparent HH comparing its components with HH X’s components and other acquaintance’s components. If HH X’s components are the best, the grandparent HH exchanges places with HH X and so. As a result of this process, the best query HHs (in terms of components’ ratings) in the three HH sub trees are pushed to the top of their respective propagation trees. Scrolling from the top, each level of the resultant sub-trees contains HHs in their decreasing order of their components’ ratings, with the lowest distinct node having the lowest ratings. This is essentially a short term algorithm, because the structure of the tree keeps changing with each query, with no long term history being maintained.

5.2.2.1 Algorithms

In the following sections, algorithms mirroring the behavior of the short term algorithm are specified. Here, only the behavior of the algorithm is specified and algorithms for external behavior of the entities, for example HH initializing, multicasting etc. are specified in Chapter 6.

5.2.2.1.1 Headhunter

When a previously unprocessed query arrives, it is stored in a queue. As noted before, initially, the HH contacts the DSM for random HHs. The received HHs are then maintained in a list. For each query, the list is updated on the basis of the above specified behavior.

Algorithm for input of a query: Here, the query propagation is asynchronous. New queries are stacked on to a queue while previously processed queries are dropped

HH_INPUT_QUERY

IN: query

//the query is checked against queries from processed_queriesList
status = check if query has already been processed
IF status is 0

ADD query to query_list

// own_name is current HH name
CALL DSM_SET_BUSY_HH (own_name)
Algorithm for processing a query:
//Called when there is an unprocessed query
HH_PROCESS_QUERY
IN: query
   // first_time indicates whether the HH has previously processed any queries.
   IF first_time is 1
      // query contains a field ID, indicating a unique number for the query
      hhList = CALL DSM_GET_RANDOM_HHS (query.ID)
      first_time = 0
   END IF
   ELSE
      // acquaintances_list maintains the profile of the current three HH acquaintances
      hhList = acquaintances_list
      CALL HH_PROPAGATE_QUERY (hhList, query)
   END ELSE
END HH_PROCESS_QUERY

Algorithm for propagating a query:
HH_PROPAGATE_QUERY
IN: hhList, query
   FOR each element hh in hhList
      response = CALL HH_INPUT_QUERY (query)
/* if response is 0, it indicates that the called HH has already processed the query */

IF response is 0
    // try to obtain another a uncontacted HH for the query
    hh = CALL DSM_GET_FREE_HH
    /* a null value indicates that there are no uncontacted HHs for the query */
END IF

IF hh is not equal to NULL
    CALL HH_INPUT_QUERY (query)
END IF

END FOR

END HH_PROPAGATE_QUERY

Algorithm for storing result:

HH_STORE_RESULT

// Externally called by a contacted HH returning results
IN: result

ADD result to resultList

/* result contains field called_hh indicating the called HH and recomm_hh indicating the best known HH */

IF result.called_hh is equal to result.recomm_hh
    // removing the called HH from the acquaintances_list
    DELETE result.called_hh
    // adding recomm_hh i.e. the best known HH to the acquaintances_list
    ADD result.recomm_hh
END IF

IF no_of_called_HHs is equal to no_of_returned_HHs
    queryid = result.queryid
    combined_result = CALL HH_COMBINE_RESULT (queryid)
    // Get the entity which has transmitted the query
Algorithm for combining results:
HH_COMBINE_RESULT
IN: queryid
OUT: listofresults

\[
\text{listofresults} = \text{GET list of results for } \text{queryid} \\
\text{own_results} = \text{GET own results for } \text{queryid} \\
\text{highest_result} = \text{GET the highest result from the listofresults and own_results} \\
\text{recomm_hh} = \text{highest_result.caller_hh} \\
// add own_results to the listofresults \\
\text{ADD own_results} \\
\text{return listofresults}
\]
END HH_COMBINE_RESULT

5.2.2.1.2 Query Manager

Each QM has an acquaintances_list containing the list of HHs to call for next query. Based on the recommended HHs returned by contacted HHs, the acquaintances_list is updated after each query.
Algorithm for input of query:
QM_INPUT_QUERY
IN: query
status = check if query has already been processed
IF status is 0
   ADD query to query_list
END IF
ELSE
   return 0
END ELSE
END QM_INPUT_QUERY

Algorithm for processing a query:
QM_PROCESS_QUERY
// Called when a unprocessed query is available
IN: query
   // first_time indicates whether the QM has previously processed any queries.
   IF (first_time is 0)
      hhList = CALL DSM_GETRANDOM_HHS
      first_time = 1
   END IF
ELSE
   // acquaintances_list maintains the profile of the current three HH acquaintances
   hhList = acquaintances_list
   CALL QM_PROPAGATE_QUERY (hhList, query)
END ELSE
END QM_PROCESS_QUERY

Algorithm for propagating a query:
QM_PROPAGATE_QUERY
IN: hhList, query
    FOR each element hh in hhList
        CALL HH_INPUT_QUERY (query)
    END FOR
END HH_PROPAGATE_QUERY

Algorithm for storing results:
QM_STORE_RESULT
IN: result
    ADD result to resultList
    IF result.called_hh is equal to result.recomm_hh
        // remove called_hh from acquaintances_list
        DELETE called_hh
        // adds recomm_hh to acquaintances_list
        ADD recomm_hh
    END IF
    IF no_of_called_HHs is not equal to no_of_returned_HHs
        combined_result = GET listofresults from result.queryid
        queryid = result.queryid
        client = GET client from queries_clientList
        CALL CLIENT_INPUT_RESULT
    END IF
END QM_STORE_RESULT

5.2.2.1.3 Domain Security Manager
At the onset, the DSM issues random non-contacted HHs to requesting HHs for forming the search space of the query. To avoid cycles and closed set of HHs, if a HH cannot contact a child HH in its list for reasons such as the query has been already processed by the child HH, the DSM, if possible, issues another HH which was not contacted earlier for that query.
Algorithm for issuing random HHs:

DSM_GET_RANDOM_HHS
IN: queryid
OUT: listofhhs
   // random_hhList maintains the list of uncontacted HHs for the query
   listofhhs = GET three randomly chosen HHs from random_hhList
   DELETE hhs in listofhhs from random_hhList
   return listofhhs
END DSM_GET_RANDOM_HHS

DSM_GET_FREE_HH
IN: queryid
OUT: hh
   // free_hhList contains the list of uncontacted HHs for the query
   hh = GET a randomly chosen HH from free_hhList
   return hh
END DSM_GET_FREE_HH

Algorithm for setting contacted HHs as not free:

DSM_SET_BUSY_HH
IN: queryid, hh
   DELETE hh from free_hhList
END DSM_SET_BUSY_HH

5.2.2.1.4 Client
The Client is an external agency such as the System Integrator or the System Generator.

Algorithm for displaying obtained results:

CLIENT_INPUT_RESULT
IN: result
   DISPLAY results
5.2.3 Long Term

Similar to the Short Term Algorithm, at the onset, the QM has an acquaintance list of three HHs in the query domain. Each HH maintains an acquaintance list of all available HHs with associated rankings, indicating the HH’s past experience with them. At the onset, all the HHs have an equal ranking (zero). On receipt of a query, since the QM does not have any previous experience, it contacts three arbitrary HHs from its acquaintance list for processing the query. Since the contacted HH do not have any previous experience, they too contact three random HHs with the propagation tree similar to the other algorithms. Akin to the previous algorithm, each of the contacted HHs recommends a HH having the best rated component(s) under it in its propagation tree. On receipt of results, a parent HH compares the obtained results of the three contacted HHs. Based on the component ratings, it updates the corresponding HH ranking in its acquaintance list. For instance, if a contacted HH recommended a HH other than itself, the contacted HH’s ranking is downgraded (penalty) by a value ‘X’. If the contacted HH is the same as the recommended HH, then the HHs’ ranking is increased (reward) by 2X. For the experimental analysis, X has been fixed as 0.5, implying that just one bad result would not result in the HH slipping down too many places.

5.2.3.1 Algorithms

In the following sections, algorithms mirroring the behavior of the long term algorithm are specified. As noted before, here only the behavior of the algorithm is specified and algorithms for external behavior of the entities, for example HH initializing, multicasting, etc., are specified in Chapter 6.

5.2.3.1.1 Headhunter

When a previously unprocessed query arrives, it is stored in a queue. As noted before, initially, the HH contacts the DSM for random HHs. The received HHs are then
maintained in a acquaintances list. For each query, the list is updated on the basis of the above specified behavior.

Algorithm for input of a query: Here, the query propagation is asynchronous. New queries are stacked on to a queue while previously processed queries are removed from the queue.

HH_INPUT_QUERY

IN: query
OUT: integer 0 or 1
status = check if query has already been processed
IF status is 0
    ADD query to query_list
    // own_name is current HH name
    CALL DSM_SET_BUSY_HH (own_name)
    return 1
END IF
ELSE
    return 0
END ELSE
END HH_INPUT_QUERY

Algorithm for processing a query:

HH_PROCESS_QUERY

IN: query

// first_time indicates whether the HH has previously processed any queries.
IF first_time is 1
    // query contains a field ID, indicating a unique number for the query
    hhList = CALL DSM_GET_RANDOM_HHS (query.ID)
    first_time = 0
END IF
ELSE
acquaintances_list maintains the profile of the current three HH acquaintances

\[ hhList = acquaintances_list \]

CALL HH_PROPAGATE_QUERY (hhList, query)

END ELSE

END HH_PROCESS_QUERY

Algorithm for propagating a query:
// Called when a unprocessed query is available

HH_PROPAGATE_QUERY

IN: hhList, query

FOR each element \( hh \) in \( hhList \)

\[ response = CALL HH_INPUT_QUERY (query) \]

/* if \( response \) is 0, it indicates that the called HH has already processed the query */

IF \( response \) is 0

// try to obtain another a uncontacted HH for the query

\[ hh = CALL DSM_GET_FREE_HH \]

/* a null value indicates that there are no uncontacted HHs for the query */

END IF

IF \( hh \) is not equal to NULL

//Call HH_INPUT_QUERY of \( hh \)

CALL HH_INPUT_QUERY (query)

END IF

END FOR

END HH_PROPAGATE_QUERY

Algorithm for storing result:
// Called externally by contacted HHs returning results

HH_STORE_RESULT
IN: `result`

ADD `result` to `resultList`

/* `result` contains field `called_hh` indicating the called HH and `recomm_hh` indicating the best known HH */

IF `result.called_hh` is equal to `result.recomm_hh`

// removing the called HH from the acquaintances_list

CALL HH_UPDATE_PROFILE (1, `result.called_hh`)

END IF
ELSE

CALL HH_UPDATE_PROFILE (0, `result.called_hh`)

CALL HH_UPDATE_PROFILE (1, `result.recomm_hh`)

END ELSE

IF `no_of_called_HHs` is equal to `no_of_returned_HHs`

`queryid` = `result.queryid`

`combined_result` = CALL HH_COMBINE_RESULT (`queryid`)

// Get the entity which has transmitted the query

`parent` = GET parent from queries_parentList

END IF

IF `parent` is QM

CALL QM_STORE_RESULT

END IF
ELSE

CALL HH_STORE_RESULT

END ELSE

END HH_STORE_RESULT

Algorithm for combining results:

HH_COMBINE_RESULT

IN: `queryid`

OUT: `listofresults`

`listofresults` = GET list of results for `queryid`
own_results = GET own results for queryid

highest_result = GET the highest result from the listofresults and own_results

recomm_hh = highest_result.caller_hh

// add own_results to the listofresults
ADD own_results

return listofresults

END HH_COMBINE_RESULT

Algorithm for updating profile:

HH_UPDATE_PROFILE

IN: mode, hh

// profile_list maintains the profile of all HHs
profile = GET profile of hh from profile_list

IF (mode is 0)

DECREMENT 0.5 from profile

ENDIF

ELSE

INCREMENT profile by 1.0

END_HH_ADJUST_PROFILE

5.2.3.1.2 Query Manager

Each QM has a acquaintances_list containing the list of HHs to call for next query. Based on the recommended HHs returned by contacted HHs, the acquaintances_list is updated after each query.

Algorithm for input of query:

QM_INPUT_QUERY

IN: query

OUT: integer 0 or 1

status = check if query has already been processed

IF status is 0
ADD query to query_list
  return 1
END IF
ELSE
  return 0
END ELSE
END QM_INPUT_QUERY

Algorithm for processing a query:
QM_PROCESS_QUERY
IN: query
  // first_time indicates whether the QM has previously processed any queries.
  IF (first_time = 1)
    hhList = CALL DSM_GET_RANDOM_HHS
    first_time = 0
  END IF
ELSE
  // acquaintances_list maintains the profile of the current three HH acquaintances
  hhList = acquaintances_list
  CALL QM_PROPAGATE_QUERY (hhList, query)
END ELSE
END QM_PROCESS_QUERY

Algorithm for propagating a query:
QM_PROPAGATE_QUERY
IN: hhList, query
  FOR each element hh in hhList
    CALL HH_INPUT_QUERY (query)
  END FOR
END HH_PROPAGATE_QUERY
Algorithm for storing results:
// Called externally by contacted HHs for returning results
QM_STORE_RESULT
IN: result
ADD result to resultList
IF result.called_hh is not equal to result.recomm_hh
    // remove called_hh from acquaintances_list
    DELETE called_hh
    // adds recomm_hh to acquaintances_list
    ADD recomm_hh
END IF
IF no_of_called_HHs is equal to no_of_returned_HHs
    combined_result = GET listofresults from result.queryid
    queryid = result.queryid
    client = GET client from queries_clientList
    CALL CLIENT_INPUT_RESULT
END IF
END QM_STORE_RESULT

The algorithms for DSM and Client are identical to 5.2.2.2.3 and 5.2.2.2.4, respectively, and hence are omitted to avoid duplication.

5.2.4 Reinforcement Algorithm

This algorithm is loosely based on [MUK03], incorporating profiling concepts. This algorithm differs from the previous algorithms, in that the profiles record both direct and indirect experiences. At the onset, QMs and HHs have an acquaintance list of all HHs in the query domain with rankings indicating their past experience with them. At the beginning, all the HHs have an equal ranking. On receipt of a query, since the QM does not have any previous experience, it contacts three arbitrary HHs from its acquaintance
list for processing the query. As before, since each of the contacted HHs does not have any previous experience, they too contact random HHs within the propagation tree, similar to the other algorithms.

While returning results, a HH includes a list of all contacted HHs and the results returned by them. Using this information, the parent HH then updates its profile of the returned HH, i.e., the child HH as well as the profiles of the HHs under the child HH. The profile update is performed on the basis of the ratings of the component results. A threshold value for a required component’s ratings is fixed. Based on this, the profile of the HH returning the component is either incremented or decremented by the same value dependent on the component’s rating. Based on the threshold value, the profiles of all HHs in the chain of a HH returning a component are updated. For example, say HH11 has a component rated 150 and returns the component to HH10, and HH10 in turn returns the component to HH9, and so on till HH1 obtains the component. Subsequently, based on the threshold value, HH1 updates its profile for all the HHs in the chain by assigning a reward or penalty.

5.2.4.1 Algorithms

In the following sections, algorithms mirroring the behavior of the long term algorithm are specified. As noted before, here only the behavior of the algorithm is specified and algorithms for external behavior of the entities, for example HH initializing, multicasting etc. are specified in Chapter 6.

5.2.4.1.1 Headhunter

When a previously unprocessed query arrives, it is stored in a queue. As noted before, initially, the HH contacts the DSM for random HHs. The received HHs are then maintained in a list. For each query, the list is updated on the basis of the above specified behavior.

*Algorithm for input of a query:* Here, the query propagation is asynchronous. New queries are stacked on to a queue while previously processed queries are dropped.

```
HH_INPUT_QUERY
IN: query
```
OUT: integer 0 or 1

status = check if query has already been processed

IF status is 0

ADD query to query_list

// own_name is current HH name
CALL DSM_SET_BUSY_HH (own_name)

return 1

END IF
ELSE

return 0

END ELSE

END HH_INPUT_QUERY

Algorithm for processing a query:

HH_PROCESS_QUERY

IN: query

// first_time indicates whether the HH has previously processed any queries.

IF first_time is 1

// query contains a field ID, indicating a unique number for the query
hhList = CALL DSM_GET_RANDOM_HHS (query.ID)

first_time = 0

END IF
ELSE

// acquaintances_list maintains the profile of the current three best HH acquaintances
hhList = acquaintances_list

CALL PROPAGATE_QUERY (hhList, query)

END ELSE

END HH_PROCESS_QUERY

Algorithm for propagating a query:
HH_PROPAGATE_QUERY

IN: hhList, query

FOR each element \( hh \) in \( hhList \)

\[ \text{response} = \text{CALL HH_INPUT_QUERY (query)} \]

/* if \( \text{response} \) is 0, it indicates that the called HH has already processed
the query */

IF \( \text{response} \) is 0

// try to obtain another a uncontacted HH for the query

\( hh = \text{CALL DSM_GET_FREE_HH} \)

/* a null value indicates that there are no uncontacted HHs for the
query */

END IF

IF \( hh \) is not equal to NULL

CALL HH_INPUT_QUERY (query)

END IF

END FOR

END HH_PROPAGATE_QUERY


Algorithm for storing result:

// Called externally by contacted HHs for returning results

HH_STORE_RESULT

IN: result

ADD result to resultList

/* result contains field called_hh indicating the called HH and summed_results a
linked list containing details of the contacted HH as well as its children HHs and
their components for the query */

FOR each \( hh \) in result.summed_result

\[ \text{comp_rating} = \text{GET rating of highest component returned by } hh \]

/* Since profile updates are based on comp_ratings, to keep confidence
values low, comp_rating is divided by 1000 */

\[ \text{comp_rating} = \text{comp_rating} / 1000 \]
// threshold indicates the value above which components are permissible
IF comp_rating < threshold
    CALL HH_UPDATE_PROFILE (0 , hh, comp_rating)
END IF
ELSE
    CALL HH_UPDATE_PROFILE (1 , hh, comp_rating)
END ELSE
END FOR
IF no_of_called_HHs is equal to no_of_returned_HHs
    queryid = result.queryid
    combined_result = CALL HH_COMBINE_RESULT (queryid)
    // Get the entity which has transmitted the query
    parent = GET parent from queries_parentList
END IF
IF parent is QM
    CALL QM_STORE_RESULT
END IF
ELSE
    CALL HH_STORE_RESULT
END ELSE
END HH_STORE_RESULT

Algorithm for combining results:
HH_COMBINE_RESULT
IN: queryid
OUT: listofresults
listofresults = GET list of results for queryid
own_results = GET own results for queryid
highest_result = GET the highest result from the listofresults and own_results
recomm_hh = highest_result.caller_hh
// add own_results to the listofresults
ADD own_results
return listofresults
END HH_COMBINE_RESULT

Algorithm for updating profile:
HH_UPDATE_PROFILE
IN: mode, hh, comp_factor
    // profile_list maintains the profile of all HHs
    profile = GET profile of hh from profile_list
    IF (mode is 0)
        DECREMENT comp_factor from profile
    ENDIF
ELSE
    INCREMENT profile by comp_factor
END_HH_UPDATE_PROFILE

5.2.4.1.2 Query Manager

Each QM has an acquaintances list similar to HHs, containing information about all HHs contacted in direct as well as indirect experiences. Based on the components returned by HHs, their values in the acquaintances list are correspondingly updated.

Algorithm for input of query:
QM_INPUT_QUERY
IN: query
    OUT: integer 0 or 1
    status = check if query has already been processed
    IF status is 0
        ADD query to query_list
    END IF
ELSE
    return 0
END ELSE
END QM_INPUT_QUERY

Algorithm for processing a query:
// Called internally when there is an unprocessed query
QM_PROCESS_QUERY
IN: query
   // first_time indicates whether the QM has previously processed any queries.
   IF (first_time)
      hhList = CALL DSM_GET_RANDOM_HHS
      first_time = 0
   END IF
ELSE
   // acquaintances_list maintains the profile of the current three HH acquaintances
   hhList = acquaintances_list
   CALL QM_PROPAGATE_QUERY (hhList, query)
END ELSE
END QM_PROCESS_QUERY

Algorithm for propagating a query:
QM_PROPAGATE_QUERY
IN: hhList, query
   FOR each element hh in hhList
      CALL HH_INPUT_QUERY (query)
   END FOR
END QM_PROPAGATE_QUERY

Algorithm for storing result:
// Called externally by contacted HHs for returning results
QM_STORE_RESULT
IN: result

ADD result to resultList

/* result contains field called_hh indicating the called HH and summed_results a
 linked list containing details of the contacted HH as well as its children HHs and
 their components for the query */
FOR each hh in result.summed_result
    comp_rating = GET rating of highest component returned by hh
    /* Since profile updates are based on comp_ratings, to keep confidence
    values low, comp_rating is divided by 1000 */
    comp_rating = comp_rating / 1000
    // threshold indicates the value above which components are permissible
    IF comp_rating < threshold
        CALL QM_UPDATE_PROFILE (0 , hh, comp_rating)
    END IF
    ELSE
        CALL QM_UPDATE_PROFILE (1 , hh, comp_rating)
    END ELSE
END FOR

IF no_of_called_HHs is equal to no_of_returned_HHs
    queryid = result.queryid
    // Get the entity which has transmitted the query
    client = GET client from queries_clientList
    CALL CLIENT_INPUT_RESULT (resultList)
END IF
END QM_STORE_RESULT

Algorithm for updating profile:
QM_UPDATE_PROFILE

IN: mode, hh, comp_factor

    // profile_list maintains the profile of all HHs
    profile = GET profile of hh from profile_list
IF (mode is 0)
    DECREMENT comp_factor from profile
ENDIF
ELSE
    INCREMENT profile by comp_factor
END QM_UPDATE_PROFILE

The algorithms for DSM and Client are identical to 5.2.2.2.3 and 5.2.2.2.4 respectively and hence are omitted to avoid duplication.

5.3 Conclusion
In this chapter, profiled and non-profiled algorithms were discussed with implementation algorithms. For experimenting with these algorithms, a URDS simulation was designed and implemented. The simulation and the subsequent experimentation with these algorithms are discussed in the forthcoming chapters.
6. Simulation of URDS

In Chapter 4, the scalability of the URDS prototype was established with experimental analyses, using a finite number of elements. The next endeavor of the thesis was to experiment with the different sets of profiling techniques, discussed in Chapter 5. In order to corroborate these experiments and lay claim to a comprehensive set of experimental results, a practical analysis on a large scale is essential. But an analysis involving the prototype on a large scale involving thousands of constituents is practically unrealistic, either in terms of the hardware or the time or the physical effort involved. Though traditionally, the formal modeling of systems has been via a mathematical model, which attempts to find analytical solutions to problems enabling the prediction of the behavior of the system from a set of parameters and initial conditions; for an experimentation in case, the profiling techniques use algorithms for which simple analytic solutions are not possible. As a result, a Computer Simulation mirrored on the actual prototype was designed and implemented. This Chapter discusses the design and the algorithms used for the implementation of the URDS simulation.

6.1 Back-Ground and Requirements

“A model is a simplified representation of a system, and simulation is the process of imitating (appearance, effect, etc.) important aspects of the behavior of the system (or plans or policies) in real time, compressed time, or expanded time by constructing and experimenting with the model of the system” [SIM87]. According to [SIM87], there are two distinct types of systems – continuous systems where variables (attributes of system elements or entities) undergo smooth changes and discrete systems where changes in variables take place in discrete steps. In the case of URDS, the discovery process follows a discrete approach, wherein the changes take in place in discrete steps at separate points in time. For example, the number of processing queries in an HH changes only after a query is processed or a new query is issued. Since this approach is dynamic in nature, the simulation consists of the observation and analyses of the results obtained by generating random events at different points in time in the computer model of the system. Although
simulation models are limited by the fact that they cannot identify optimum solutions and can only compare alternative solutions, in the case of URDS simulation, alternative profiling techniques are being compared and hence simulation is appropriate. Before the design of the simulation, the high level requirements of such a simulation were identified:

- The simulation should closely mirror the actual behavior of the actual system in compressed time [SIM87].
- The behavior of the system should be simplified without sacrificing on accuracy or generality.
- It should be capable of absorbing structural changes in models/strategies without too many changes.

Based on these requirements, in further sections, the process of designing and implementing the simulation is explained.

6.2 Computer Simulation

The simulation was designed and implemented according to Figure 6.1, modified from [SIM87]. According to this process, the first stage involves an analysis of the system to be simulated. This stage involves a classification of the system into subsystems and identification of entities and attributes. The second stage involves adapting the system for the simulation, i.e., replicating the behavior of the system by designing suitable algorithms. Consequently, the algorithms are implemented and verified by means of induction or deduction as defined in [SIM87]. Each of these sections is explained in detail in the following paragraphs.

6.2.1 System Analysis

A system analysis was made easier by the fact that the URDS prototype was already available and a real life process did not have to be modeled. As a result, the identification of subsystems, entities, attributes (parameters and variables), interrelationships and activities was easy, with the boundary and the environment of the simulated system being equal to that of the actual system. The profiling techniques mainly involve the propagation of queries between entities.
Therefore, the design process starts off accordingly, i.e., from the identification of entities directly and indirectly involved in the propagation of queries. In the following section, the entities and their basic, indivisible tasks were identified:

1) DSM:

   i) Basic Responsibility: Authentication.

   ii) Tasks:
a) Authenticates each Principal such as HH, AR, QM on their initialization.
b) Provides a list containing all the active HHs on request.

iii) Query Propagation Role: Indirect.

2) Headhunter:
   i) Basic Responsibility: Propagation of queries and return a list of matched components.
   ii) Tasks:
       a) Multicast its location.
       b) Obtain components from answering ARs.
       c) Process a Query:
           1) Search local meta-repository for any matching components pertaining to a query.
           2) Propagate a query to other HHs.
           3) Send results of a query to the requester HH.
   iii) Query Propagation Role: Direct.

3) Active Registry:
   i) Basic Responsibility: Registration of components.
   ii) Tasks:
       a) Listen for multicast messages.
       b) Provide registered components to requester HHs.
       c) Register interested components.
   iii) Query Propagation Role: Indirect.

4) Query Manager:
   i) Basic Responsibility: Propagation of queries.
   ii) Tasks:
       a) Obtain a list of HHs from DSM to which a query has to be propagated.
       b) Propagate a query to HHs.
       c) Receive results from HHs.
d) Send results of query to requester Clients

5) Component:
   i) Basic Responsibility: Providing services.
   ii) Tasks:
      a) Register with chosen set of ARs.
      ii) Query Propagation Role: Indirect.

6) Client (Simulates the System Generator, in that it takes over the role of issuing queries)
   i) Basic Responsibility: Issue Queries.
   ii) Tasks:
      a) Issue Query.
      iii) Query Propagation Role: Direct.

Since one of the main requirements of the simulated system is compressed time, the process of calculating simulation time is very important. Typically, in a simulation, the simulation time is advanced in two ways, either by using uniform time increments or by a variable time increment method. In uniform time-oriented simulation, time is incremented from time t to (t + dt), where dt is a uniform fixed time increment. In variable time increment method, time is incremented from time t to the next event (most imminent event) time t1 whatever may be the value of t1. The system state changes are made at the event time t and at the next event time t1, with the continuous repetition of this process. The drawbacks of the first approach are obvious; events occurring during the interval (t, t+ dt) are detected only at time (t + dt). In the second approach, the periods between events are treated as inactive or insignificant and therefore no time is consumed even though inter-event activities do consume time in the real system. Since the URDS system follows a discrete approach wherein changes take place in discrete steps, detecting events at fixed intervals will not work. Consequently, the variable time method is a better match for this simulation.
To counter the mentioned drawback of the variable time approach, the concept of Lamport’s logical clock [COU01] is used. “A Lamport logical clock is a monotonically increasing software counter, whose value need bear no particular relationship to any physical clock”. Each process maintains its own logical clock $L_i$, which is used for applying Lamport timestamps to events. For each event issued by a process $p_i$, $L_i$ is updated by 1 (or any positive value). When the process $p_i$ sends a message $m$, the time ($t = L_i$) is also sent with the message. On receiving the message $m$, a process $p_j$ computes $L = \max (L_j, t)$ where $L_j$ is the time indicated by its own logical clock. If $L_j < L$, then the process $p_j$ updates $L_j$ to $L$ for time-stamping the received message as well as its own logical clock. Applying this concept to the URDS, each entity directly involved in the query propagation (identified previously), HH, QM and Client maintains its own logical clock. For each event (propagation of query or transmission of results) issued by the entity, its clock is updated by a corresponding value equivalent to time for the event in the actual system (determined offline). Also, on receiving a query, the clock is updated according to the timestamp of the query, with the query’s timestamp being modified for further transmission of query. After the identification of entities, their interrelations and associated tasks and process for calculation of simulation time, the next stage is the determination of algorithms for adapting the actual system for the computer simulation.

6.2.2 Adapting for Simulation

In the previous sections, the entities of the simulation and their main tasks were identified along with the concept of simulation time. The next few sections outline the algorithms for each of the entities listed, which mirror the behavior of the entities in the actual prototype, respectively. The primary algorithm to be designed is that of the main simulation, indicating the flow of control between intra-entity as well as inter-entities. Since a simulation iteration corresponds to a real time instant, each entity of the system should have a chance to run within a simulation cycle. At any wall clock instant, each entity should be able to execute its tasks. Depending on the state, the task could be starting an activity, executing an activity or being idle. The main simulation algorithm is indicated below:
SIMULATION_MAIN_OUTLINE

// Instantiate entities according to their initial configuration of entities.

hhList = CALL DSM_GETLIST (Headhunter)
arList = CALL DSM_GETLIST (ActiveRegistry)
qmList = CALL DSM_GETLIST (QueryManager)
clientList = CALL DSM_GETLIST (Client)

WHILE TRUE
    // for each element in hhList, HH_DOTASK is called
    FOR each element in hhList CALL HH_DOTASK
    // for each element in arList, AR_DOTASK is called
    FOR each element in arList CALL AR_DOTASK
    // for each element in qmList, QM_DOTASK is called
    FOR each element in qmList CALL QM_DOTASK
    // for each element in clientList, HH_DOTASK is called
    FOR each element in clientList CALL CLIENT_DOTASK
END WHILE
END SIMULATION_MAIN_OUTLINE

In the above algorithm, a single URDS is considered for simplicity. Consequently, since it has an indirect query propagation role, DSM is excluded from the algorithm and is assumed to have the capability to process concurrent requests. In the next few sections, the algorithm for each of the entities are explained briefly.

6.2.2.1 Headhunter

The HH of the simulation mirrors the HH in the actual system. The behavior is replicated with the main tasks of simulation HH akin to that of the actual HH. These tasks are decomposed into subtasks and each of these subtasks is explained in the form of algorithms. These algorithms outline the various activities of a HH.
Algorithm for initialization: On initialization, each HH is assigned a name and a multicast value.

//Called by DSM
HH_INITIALIZATION
` IN: initial_value
  // obtains the name from DSM
  hhName = GET name from DSM
  mcastTime = GET multicast Time from DSM
  // initial_value is generally zero
  clock = initial_value
END HH_INITIALIZATION

Algorithm for selecting main tasks: This algorithm calls the particular method of HH depending upon the option selected (choice).

HH_DOTASK
  choice = GET number from 0-2
  SWITCH (choice)
    CASE 0: CALL HH_MULTICAST
    CASE 1: CALL HH_GET_COMP_FROM_AR
    CASE 2: CALL HH_PROCESS_QUERY
  END SWITCH
END HH_DOTASK

Algorithm for multicasting: This algorithm is responsible for multicasting the HH’s address periodically and update its clock.

HH_MULTICAST
  // time period since last multicast is maintained in a list
  period = GET time period since last multicast
  IF period > mcastTime
    // multicast the address of the HH
    MULTICAST
// updates the clock value of the HH (if necessary)
UPDATE clock
END IF
END HH_MULTICAST

Algorithm for obtaining components from AR: On receiving a response to a multicast, an HH gets a list of components. To reduce complexity, in the simulation, an HH based on the HH to AR ratio contacts a random AR and obtains corresponding components.

HH_GET_COMP_FROM_AR
/// obtains the list of ARs from DSM
arList = GET list of ARs from DSM
/// to ensure proportionality, HHs can interact with only a ratio of ARs
ratio = no_of_HHs / no_of_ARs
FOR each ratio elements in arList
    /// obtain a random distinct AR from the list of ARs
    ar = GET AR from arlist
    /// obtain component List from AR
    compList = GET component list from ar
END FOR
END HH_GET_COMP_FROM_AR

Algorithm for receiving query: When a HH receives a query and if the query has not been processed earlier, it is placed in the WaitingQueriesList of the HH and the clock is updated. requesterEntity is the entity where the HH receives the query from.

HH_RECEIVE_QUERY
IN: requesterEntity (HH or QM), query
/// query is checked against processed_queries, a list containing processed queries.
IF query is not processed
    /// insert the query in HH’s query list
STORE query in WaitingQueriesList
// updates the clock value of HH
UPDATE clock
// records the requesting entity
RECORD requester requesterEntity
END IF
END HH_INPUT_SEARCH

Algorithm for processing query: For processing, an HH obtains a query from the WaitingQueriesList. If the TTL (Time To Live) of the query has not expired, it checks whether it can process the query. If it can process the query, and there are HHs available for propagation, call HH_PROPAGATE_QUERY
HH_PROCESS_QUERY
   // if the WaitingQueriesList is not empty
   IF WaitingQueriesList not NULL
      // obtain query from list
      query = GET query from WaitingQueriesList
      IF TTL of query not expired
         // obtains the HH list from a hashtable
         hhList = GET HH list from query_hhList
         // if hhList is not null
         IF hhList not NULL
            CALL HH_PROPAGATE_QUERY
         END IF
      END IF
   END IF
END IF
END HH_PROCESS_QUERY

Algorithm for propagating query: If the TTL of the query has not expired, then propagate query to child HHs.
HH_PROPAGATE_QUERY
IN: queryId, listofchildHHs

DECREASE TTL of query
IF query TTL has not expired
    PROPAGATE query to listofchildHHs
END IF
END HH_PROPAGATE_QUERY

Algorithm for storing result: When a result is stored, the no of called HHs for the query (corresponding to the result) is compared to the no of returned HHs. If equal, the query is added to the finished_queriesList, implying that the query is finished and more results are not expected.

HH_STORE_RESULT

IN: result, queryId
STORE result in queries_results
no_of_calledHHs = GET HH number from query_HH
no_of_returnedHHs = GET HH number from query_HH_results
IF no_of_calledHHs is equal to no_of_returnedHHs
    ADD query to finished_queriesList
END IF
END HH_STORE_RESULT

Algorithm for checking results: If the finished_queriesList is not empty, a finished query is obtained and corresponding results returned to the issuer of the query.

HH_CHECK_RESULTS

IF finished_queriesList is not empty
    // obtain the query from the processed queries list
    query = GET query from finished_queriesList
    // obtain the result for a particular query
    result = GET result from queries_results for query
    // obtain the issuer of the query
    client = GET issuer of query

END IF
END HH_CHECK_RESULTS
CALL CLIENT_STORE_RESULT (result)
END IF
END HH_CHECK_RESULTS

6.2.2.3 Active Registry
The Active Registry is assigned a name on initialization. When a component registers itself, the component is appended to compList. On request by a HH, the updated compList is provided. choice is the option selected depending upon the task.

Algorithm for initialization:
//Called by DSM
AR_INITIALIZATION
    // name is assigned by the DSM
    arName = GET name from DSM
END AR_INITIALIZATION

Algorithm for main tasks:
AR_DOTASK
    choice = GET a number from 0-1
    SWITCH (choice)
        CASE 0: CALL AR_REGISTER_COMP
        CASE 1: CALL AR_GIVE_COMPLIST
    END SWITCH
END AR_DOTASK

Algorithm for registering components:
AR_REGISTER_COMP
    IN: component
    IF component not registered
        APPEND component to compList
    END IF
Algorithm for returning lists of registered components:

AR_GIVE_COMPLIST

IF compList not empty
    RETURN compList
END IF
END AR_GIVE_COMPLIST

6.2.2.3 Query Manager

The Query Manager on initialization is assigned a name by DSM. When the QM gets a query, its clock is updated and the query is stored in WaitingQueriesList. The identity of the issuer is also recorded for the future transmission of results for that query. Also, on the receipt of a query, the DSM is contacted for a list of HHs, to which the query is propagated. choice is the option selected depending upon the task. On the receipt of results, the QM checks whether all contacted HHs for a query have returned results or not. If they have, the combined results are propagated back to the client, which issued the query.

Algorithm for initialization:

// Called by DSM
QM_INITIALIZATION

    // name is assigned by the DSM
    qmName = GET name from DSM
END QM_INITIALIZATION

Algorithm for main tasks:

QM_DOTASK
    choice = GET number from 0-1
    SWITCH (choice)
CASE 0: CALL QM_GET_SEARCH_TABLE

CASE 1:
   // if the query list is not empty
   IF WaitingQueriesList is not NULL
      CALL QM_PROCESS_QUERY
   ENDIF
   CALL QM_CHECK_RESULTS
END SWITCH

CALL QM_CHECK_RESULTS

END QM_DOTASK

Algorithm for receiving queries:
// Called externally for issuing queries
QM_RECEIVE_QUERY
   IN: requestorEntity (HH or QM), query
   IF query is not processed
      // store the query in the query list
      STORE query In WaitingQueriesList
      // update the clock
      UPDATE clock
      // records the requesting entity
      RECORD requester requesterEntity
   END IF

END QM_INPUT_SEARCH

Algorithm for obtaining search table of HHs for a query:
QM_GET_SEARCH_TABLE
   IN: queryId
   hhList = CALL DSM_GETSOME_HH(queryId)
   GET queryId and hhList from query hhList

END QM_GET_SEARCH_TABLE
Algorithm for processing query:
QM_PROCESS_QUERY

IF query in WaitingQueriesList
  // obtain query from the query list
  query = GET query from list
  IF query TTL not expired
    // obtain propagation HHs list for a particular query
    hhList = GET HH list for a query
    FOR each element in hhList
      CALL QM_PROPAGATE_QUERY
    END FOR
  END IF
END IF
END QM_PROCESS_QUERY

Algorithm for propagating query:
QM_PROPAGATE_QUERY

IN: queryId, listofchildHHs
DECREASE TTL of query
  // if HH’s TTL has not expired
  IF query TTL not expired
    // send the query to other HHs
    PROPAGATE query to listofchildHHs
  END IF
END QM_PROPAGATE_QUERY

Algorithm for storing result:
QM_STORE_RESULT

IN: result, queryId
STORE result in queries_results
  no_of_calledHHs = GET HH number from query_HHs
\[ \text{no\_of\_returnedHHs} = \text{GET HH number from query\_HHs\_results} \]

\[ \text{IF no\_of\_calledHHs} \text{ is equal no\_of\_returnedHHs} \]

\[ \text{ADD query to finished\_queriesList} \]

\[ \text{END IF} \]

\[ \text{END QM\_STORE\_RESULT} \]

\textit{Algorithm for checking results:}

\text{QM\_CHECK\_RESULTS}

\[ \text{// obtain the query from the processed queries list} \]

\[ \text{query} = \text{GET query from finished\_queriesList} \]

\[ \text{// obtain the result for a particular query} \]

\[ \text{result} = \text{GET result from queries\_results} \]

\[ \text{// obtain the issuer of the query} \]

\[ \text{client} = \text{GET issuer of query} \]

\[ \text{CALL CLIENT\_STORE\_RESULT (result)} \]

\[ \text{END QM\_CHECK\_RESULTS} \]

\[ \text{6.2.2.4 DSM} \]

The DSM is responsible for initializing all the entities. Based on the initial configuration supplied (external to the system boundary) by the user, the DSM initializes the entity and stores the initialization information in individual lists. On requests, the DSM provides a copy of these lists. Also, an option for giving a select no of HHs is provided.

\textit{Algorithm for initializing entities:}

\[ \text{// Called by Simulation before the main loop} \]

\text{DSM\_INITIALIZE\_ENTITIES}

\[ \text{IN: no\_of\_entities} \]

\[ \text{FOR each no\_of\_entities} \]
CALL ENTITY_INITIALIZATION
STORE entity in entityList
END FOR
END DSM_INITIALIZE_ENTITIES

Algorithm for returning lists:

DSM_GETLIST
IN: entity
IF entity is HH
    return hhList
ENDIF
IF entity is AR
    return arList
ENDIF
IF entity is QM
    return qmList
ENDIF
IF entity is Client
    return clientList
ENDIF
END DSM_GET_LIST

Algorithm for getting a subset of HHs:
// Called externally when a subset of HHs is required.

DSM_GETSOME_HH
IN: queryId
hhList = GET HH list related to queryId
// obtains the number of branches for a node depending upon the algorithm
somehhList = GET ‘n’ HHs from hhList
DELETE somehhList from hhList
END DSM_GETSOME_HH
6.2.2.5 CLIENT

As noted before, Clients are the entities simulating the behavior of System Generator, which are responsible for issuing queries. Here based on random probabilities, queries are issued, which are then issued to QMs.

*Algorithm for initialization:*
//Called by DSM
CLIENT_INITIALIZATION
   // name is assigned by the DSM
   clientName = GET name from DSM
END_INITIALIZATION

*Algorithm for main tasks:
CLIENT_DOTASK
   IF size of Waiting_queriesList is not equal to zero
      query = first query in Waiting_queriesList
      CALL CLIENT_SEARCH
   END IF
END CLIENT_DOTASK

*Algorithm for initiating a search:*
CLIENT_SEARCH
   // obtain list of QMs from DSM
   qmList = GET list of QMs from DSM
   qm = GET a QM from the qmList
   CALL QM_INPUT_SEARCH
END_CLIENT_SEARCH

6.2.3 Implementation

After the algorithms’ design stage, the main requirement to be considered here is the simplification of the behavior of the system without sacrificing accuracy or generality of
the obtained simulation. In the actual system, all the constituent entities are Java-RMI instances deployable on physically different systems. Since obviously, the simulation cannot contain Java-RMI instances, each real entity is mapped on to simplified simulation entities. Since the actual system was in Java-RMI, the simulation language was chosen as Java to enable an easier correlation and replication of concepts. Consequently, each actual entity is mapped to a Java object with the state of the object being retained in data structures. The random numbers used in the simulation are based on linear congruential formula, as used in the Java programming language.

6.2.4 Validation
To validate typically means to prove that an entity is true or an exact replica of another entity. Here, the simulation is validated against the actual system using the method of inductive reasoning. Inductive reasoning [SIM87] is termed as the method for drawing conclusions by observing, collecting evidence, and detecting patterns and is usually based on the extrapolation of data from known data. In the case of the simulation, the validation is done on the basis of comparison of data with known data as well as extrapolation of data from known data.

Figure 6.2 Experiment using the URDS Prototype
For any comparison between the simulation and the data provided by the graphs of the actual prototype, the data has to be of the same scale. As noted before, the query propagation is the main point of comparison for the profiling algorithms. For correlating time between the two systems, in the actual system, the time period for transactions between each entity involved in the propagation process (previously identified) was recorded for a number of trials. For example, the average of 20 trials for a query to be propagated from a Client to QM was found to be 39.23 ms.
The above values were incorporated into the simulation, by modifying the query’s timestamp and the process for updating clocks. Also, to reproduce network delays occurring in the actual system, an element of randomness based on standard deviation was included. Consequently, when a query arrives from a Client to a QM, the query’s timestamp indicates 39.23 ms + SD, where SD indicates Standard Deviation. In Chapter 4, experiments for scalability have been performed and associated graphs included. Under the same experimental conditions, the simulation was used to reproduce the experiments. The results obtained for the simulation are similar to that of the actual system. For example, Figure 6.2 and Figure 6.3 are the graphs obtained for the experiment involving increase of HHs. Since the results are similar and the trend holds for increased scale of entities as well as interactions, it can be concluded on the basis of induction that the simulation mirrors the actual system in behavior and expected results.

In this chapter, the design and implementation of the URDS simulation was discussed. In the subsequent chapter, an experimental evaluation of the profiling algorithms is performed using the simulation as the experimental setup.
7. Experimental Analyses

Chapter 5 provided details of the different profiling algorithms, while Chapter 6 provided information about URDS simulation. This chapter focuses on the experimental analysis of the profiling algorithms using the experimental setup provided by the simulation.

7.1 Metrics
In Chapter 5, the component discovery process was compared with document classification process from the information domain, with discussion of similarities and differences. As noted, unlike the text matching process employed in filtering domain, software resource discovery relies on matching the component behavior both in terms of syntax as well as the semantics, synchronous and Qos contracts. Although both the processes are inherently different, the metrics for measurement of their performance are similar. Typically, in the information filtering domain, the metrics for measuring performance are Precision [MUK03] and Recall. Precision is defined as the number of relevant documents retrieved divided by the total number of documents retrieved while recall is defined as the number of relevant documents retrieved divided by the total number of relevant documents in the entire search space. Here, the components are measured by their ratings. The algorithms aspire to discover the best components in the shortest possible time. As a result for comparing these algorithms, the metrics have been modified and are termed Precision and Sigma for purposes of this thesis. Here, precision refers to the number of obtained components which are present in the list of top ten components. The list contains the top 10 components pertaining to the query in decreasing order of their ratings. For example, if the highest rated component for a query is obtained, the precision for the query is 50%. If the two highest components are obtained, then precision is equal to $50 + 25 = 75\%$ and so forth. If precision is equal to 100%, all the top ten components for the query have been located. The other metric, Sigma, refers to the sum of discovered component ratings divided by the sum of all component ratings. In both cases, component refers to components pertaining to the query, i.e., if query for component ‘0’ is issued, Sigma for that query refers to the sum of discovered component 0’s ratings divided by the sum of all component 0 ratings.

7.2 Experiments

In the following sections, experiments in various categories were performed using the above indicated metrics.

7.2.1 Experiments altering the nature of the propagation tree
In each of the algorithms, the query propagation takes the shape of a propagation tree with the QM at the root and then the propagated HHs as the branches. In this section, experiments were carried out to find the best shape of the tree, i.e. whether a binary tree, tertiary tree, etc., gives most favorable results. Propagating entities of each the profiling algorithms maintain a list of HHs to which queries are passed on. Depending on the algorithm, the list changes randomly or based on past history. This experiment aims at detecting the optimum number of HHs in that list to which a query should be propagated at a time, i.e. whether QM or HH should propagate to 2, 3, 4 or ‘x’ HHs at a time. The optimal number of HHs is described as the number of HHs for which the turnaround time is as low as possible. This set of experiments was carried out with 1000 HHs, 10 ARs, 10 QMs and 100 Components. Ten distinct queries were issued periodically in a cluster, with the probability of the first query occurring being 50%, the next query 25% and so on. Details of the algorithms were recorded at the 4000th iteration and represent the averages of the performances for queries sent between 3600 – 4000 iterations. Figure 7.1, Figure 7.2 and Figure 7.3 and Figure 7.4 illustrates the results for Random, Short Term, Long Term and Reinforcement algorithms respectively, where the X-axis represents time from the initial issue of queries and Y-axis represents the average precision of the results for the queries. In each of the figures, it can be observed that a 2 children (2c) tree takes more time to reach 100% precision than a 3 children (3c) tree. The better performance of a 3c tree can be ascribed to the fact that having an additional contact means additional knowledge for a HH and faster propagation, since requests are asynchronous and happen concurrently. But as the number of contacts goes up to 4 and 5, though the additional information goes up, the load on each node in the tree goes up too, i.e. time needed to store results, correlate and return results increases. As a result, there is only a small difference in time from 3c and 4c for reaching 100% precision, while 5c takes more time than 4c or 2c, implying that it takes so much more time correlating that the advantage of having additional contacts and parallel propagation is negated. When these experiments were performed, the sigma of the obtained results for queries was also recorded, as in Figures 7.5, 7.6, 7.7 and 7.8. The above observations held for that set of graphs too and the differences between the trees are more pronounced. The reason for the marked
increase is that sigma measures the total number of components while precision measures only the top ten components.

Figure 7.1 Precision for Random Algorithm
Figure 7.2 Precision for Short Term Algorithm

Figure 7.3 Precision for Long Term Algorithm
Figure 7.4 Precision for Reinforcement Algorithm

Figure 7.5 Sigma for Random Algorithm
Figure 7.6 Sigma for Short Term Algorithm

Figure 7.7 Sigma for Long Term Algorithm
A 2c tree takes far more time for achieving 100% precision and sigma as compared to other trees. Generally, 3c and 4c trees are equally matched with 4c a little better in performance. But as the number of children is increased to 5, the performance drops indicating that the cost of correlation and returning results outweighs the advantage of more contacts and knowledge.

7.2.2 Experiments using different types of Query Arrival

In this section, the profiling algorithms were compared with type of query arrivals. Relations between successive queries typify the kind of query arrivals. In random queries, there is no relation between two successive queries or between any queries for the matter. In the second type, queries are aged appropriately at every occurrence, implying that if say three queries x, y and z are scheduled and x occurs, then x is aged appropriately such that x does not occur before y and z have occurred at least once. In clustered queries, the queries are clustered such that if there are ten distinct queries, the probability of the first query occurring would be 50%, the second query 25% and so forth. The motivation for
conducting these experiments is to discover the suitability of the algorithms to the different types of query arrivals. The next few sections detail the experiments conducted with these different kinds of queries.

**7.2.2.1 Experiment using Random Queries**

As noted before, the list of queries (100 in all) is scheduled randomly. The experimental setup consists of 1000 HHs, 10 ARs, 10 QMs and 100 Components. Figure 7.9 illustrates the performance of each of the profiling algorithms in terms of precision, while Figure 7.10 plots them in terms of sigma. As before, the X-axis represents time from the initial issue of queries and Y-axis represents the average precision of the results for the queries. Details of the algorithms were recorded at the 4000th iteration and represent the averages of the performances for queries sent between 3600 – 4000 iterations. As expected, the random algorithm performs badly, since no information about contacted HHs is maintained at any point in the query propagation process. Due to the randomness of the queries, the other three algorithms vary in their performance. Since the reinforcement algorithm maintains information about a larger set of HHs, it obtains the best components faster and hence its performance is better. In the reinforcement algorithm a HH retains information about all the HHs in all the levels under it in the propagation tree, while in the long term algorithm, a HH retains information about only its closest lower level of HHs. As a result, the performance of the long term algorithm is worse than that of the performance of the reinforcement algorithm. The short term algorithm relies on the non-disjointness of components registered with HHs, i.e., it provides good results, if by chance the best HHs containing components for the previous query are also the HHs containing components for the current query. Since the queries are random, there are fluctuations in performance for all the history based profiles. In Figure 7.8 too, the same observations hold good, with the Reinforcement algorithm obtaining better components for the main part.
Figure 7.9 Precision for Random Queries

Figure 7.10 Sigma for Random Queries
7.2.2.2 Experiment using Aged Queries

As explained before, this experiment involved *aging* a set of queries, such that in a round of queries, a query occurs once. When there is more than one query with the same age, a query is randomly selected among them. This implies that the order of queries might not be the same for all the rounds. As before, the experimental setup consists of 1000 HHs, 10 ARs, 10 QMs and 100 Components. The set of queries was restricted to 10. Figure 7.11 illustrates the performance of each of the profiling algorithms in terms of precision, while Figure 7.12 plots in terms of sigma. As before, the X-axis represents time from the initial issue of queries and Y-axis represents the average precision of the results for the queries. Details of the algorithms were recorded at the 4000\textsuperscript{th} iteration and represent the averages of the performances for queries sent between 3600 – 4000 iterations. The reinforcement and long term algorithms are closely matched in their performance. Since only a subset of queries is sent in random order, the long term algorithm adapts to the randomness in their occurrence better than in the previous section when random queries were sent. Short Term performs badly, since it retains information only about the HHs contacted for the previous query. The random algorithm performs the worst of all the algorithms. Figure 7.12 illustrates the performance of the algorithms in terms of sigma. For reasons explained above, the reinforcement algorithm performs the best and the long term next. Short term performs as badly as the random algorithm.
Aged Queries: Algorithms Comparison (Precision)

Figure 7.11 Precision for Aged Queries

Aged Queries: Algorithm Comparison (Sigma)

Figure 7.12 Sigma for Aged Queries
7.2.2.3 Experiment using Cluster Queries

repeated, it learns 4 very rapidly. In precision, Short Term does well as a result of successive queries, but is the worst in sigma. The reinforcement algorithm had learnt 17 previously, but struggles to learn 4. This can be ascribes to the fact that the updating of profiles in the algorithm is based on the component ratings. As a result, if a query has been learnt and repeated successively, the algorithm struggles to learn another query for a period. Since the algorithm retains additional knowledge of HHs at all levels compared to long term, the sigma for learnt queries is better.

Figure 7.13 Precision for Clustered Queries – I group
Figure 7.14. Sigma for Clustered Queries – I Group

Figure 7.15. Precision for Clustered Queries – IV Group
Clustered Queries: Algorithms Comparison (Sigma)

Figure 7.16 Sigma for Clustered Queries – IV Group

Clustered Queries: Algorithms Comparison (Precision)

Figure 7.17 Precision for Clustered Queries – Eighth Group
As expected, the random algorithm performs poorly for all types of queries. Since only the results of the previous query are retained, the short term algorithm performs poorly in precision for random queries, moderately for aged queries (shorter subset of random queries) and much better for clustered queries (successive queries). Also since a profile is not maintained, the nature of the propagation tree is haphazardly changed for every query with the best HHs for the query moving up the tree. As a result, in terms of sigma, the short term performs as badly as the random algorithm. The long term maintains a profile and updates it by fixed increments based on a query. As a result, it is capable of learning new queries quickly. The profile and fast learning enables the algorithm to perform moderately for random queries, competently for aged queries and excellently for clustered queries. In the case of the reinforcement algorithm, the profile updates are performed on the basis of component ratings, which hamper learning quickly. As a result, the algorithm does well in precision for random queries, aged queries and poorly for
cluster queries. On the upside, since the reinforcement algorithms perform updates on profiles relating to all HHs under them, it performs very well in terms of sigma.

7.2.3 Experiments varying the number of entities

The motivation behind conducting this category of experiments is to determine the performance of the algorithms across a wide range of number of active HHs. In this section, different numbers of HHs were started at the onset of experiments and the effects on each of the profiling algorithms were studied. As before, experimental setup consisted of 10 ARs, 10 QMs and 100 Components. As in the past, a set of 10 queries were issued in the form of clusters. As before, the X-axis represents time from the initial issue of queries and Y-axis represents the average precision of the results for the queries. Figures 7.19 – 7.26 illustrate the performance of each algorithm when the number of HHs is 100, 500, 1000 and 2000 respectively. When the number is 100, the search space is small, resulting in faster turn-around time ensuing better precision and sigma. As the number of HHs increases, the search space increasing resulting in lower turn-around times.

![Random Algorithm: Comparison of Varying no. of Queries (Precision)](image)

Figure 7.19 Precision for Random Algorithm
Figure 7.20 Sigma for Random Algorithm

Figure 7.21 Precision for Short Term Algorithm
Figure 7.22 Sigma for Short Term Algorithm

Figure 7.23 Precision for Long Term Algorithm
Long Term Algorithm: Comparison of Varying no. of HHs (Sigma)

Reinforcement Algorithm: Comparison of Varying no. of HHs (Precision)

Figure 7.24 Sigma for Long Term Algorithm

Figure 7.25 Precision for Reinforcement Algorithm
Increasing the number of HHs increases the search space and hence increases the time needed for achieving 100% precision or 100% sigma. Since for any algorithm, the size of the sub trees might be different i.e. unbalanced trees, increase in time might not be directly proportional to the increase in number of HHs.

7.2.4 Overall Conclusions

The performance of each algorithm is based on the number of entities, relation between queries as well as number of permissible branches/children for each node. Experiments were conducted by varying these factors and recording the performance of each algorithm. From the set of experiments and observations from each category of experiments, it can be concluded that each algorithm is suitable for a particular purpose. If a query does not have a time stamp for expiration i.e. infinite time is available, then the random algorithm can be used with minimal cost for exhaustively contacting all the available search space. If the chances of queries reoccurring successively are very high,
and only the best components are needed in a short span of time, than short term is the optimal algorithm for obtaining components without the cost of maintaining profiles. If queries are repeated frequently but not necessarily in a successive manner with chances of random queries being intermittently present, then the long term algorithm is optimal for obtaining good components within moderate time. If queries are purely random and all available components are required within the shortest possible time, then the reinforcement algorithm can be used successfully.
8. Interoperability of URDS with Grid Services

In the previous parts of the thesis, the URDS was experimentally evaluated and enhanced with different profiling techniques. After these evaluations, the next objective of this thesis was to evaluate and examine the interoperability of URDS concepts in a different domain. Although a variety of different domains exist, Grid Computing being an extension of distributed computing was chosen as the testing domain. The Grid contains a wide range of heterogeneous resources and hence serves as a natural fit for testing URDS interoperability. For purposes of this thesis, the terms “Grid” and “Grid Computing” are used in reference to a service oriented Grid [SRI04], which is defined as a set of hardware and software resources that provide seamless, dependable and pervasive access to high-end computational capabilities.

Grid Services [GRI04] are extended Web Services offering stateful and potentially transient services. Like Web Services [WEB04], a Grid Service is a WSDL-defined service that follows a set of conventions regarding its interface definitions and behaviors. But a Grid Service is an instance of an implementation of a type of service, which is a collection of specific interfaces. The instance implements a Grid Services Handle (GSH) which is a sort of Uniform Resource Indicator (URI) for the service instance and is bound to a Grid Service Reference (GSR). The GSR contains the necessary information for binding and using the service. Typically, Grid Services are hosted in OGSA containers like Globus [FOS02], which are akin to Active Registries in URDS. These containers offer services for registration and lookup of the components.

URDS [SIR02] has native registries/lookup services of various component models (RMI, CORBA, .NET) extended to be able to listen and respond to multicast messages from the Headhunters. These registries also have introspection capabilities for discovering not only the instances, but also the specifications of the components registered with them. As noted before, the Grid domain has Grid services registered with OGSA containers such as Globus. For interoperation between URDS and Grid services, a two-step process involving the modification of the registration mechanism in URDS and/or Grid and
modification of the component specifications to incorporate idiosyncrasies associated with the Grid world is essential. In the next few sections, these steps are discussed in detail with regard to an implemented representor Grid service example.

8.1 Design

The representor grid service was created as two separate components, Data Service and Data Service. There is a one-to-one relationship between these components, as can be seen in Figure 8.1.

![Figure 8.1 Representor Service](image)

As represented in Table 1 and Table 2, the representor system contains three main functions, provide() and represent() and getummSpecs(). When the Data Service component is invoked, the provide function is executed. In this method, an input document is randomly selected from a list of documents. After information such as input document location is extracted, they are sent to the remote classify method of Representor. There, the document is converted into structures and then returned. Similar to URDS components, each of these components have a getUmmSpecs() method which returns a UMM specification. Both the components are registered as Grid Services.

<table>
<thead>
<tr>
<th>Data Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actions</strong></td>
</tr>
<tr>
<td>provide()</td>
</tr>
<tr>
<td>getUmmSpecs()</td>
</tr>
</tbody>
</table>

Table 1.
The representing components were implemented as Java-RMI Grid Services using the process mentioned in [GRI04]. Globus toolkit GT3 was installed in a Linux cluster of 3 machines. The toolkit contains an OGSA container known as the Globus container. The code for the implementation can be found in Appendix.

### 8.2 Specifications

In URDS in particular and UniFrame in general, UMM specifications have been defined for describing the characteristics of components or services. Though these specifications are comprehensive and sufficient for software components, they are not adequate for describing Grid services. This inadequacy is based on the fact that Grid services are usually used as part of high performance applications, implying that they can be hardware resources as well. Since there are existing approaches for describing Grid Services, these approaches can be leveraged upon for tackling the inadequacy. After a survey of approaches, [ALI02] was found suitable for incorporation into UMM specifications. As an example, the new specification of a representor component is listed in Figure 8.2. In addition to description of hardware resources, another work [YU05] includes capabilities for indicating economic characteristics of services, which is listed in Figure 8.3. A detailed description of the attributed in the figures is available in the listed references.

```xml
<Service_SLA>
<Service_Information>
<Service_Name> Representor Service </Service_Name>
```
<Service_Description> Provide representation services for documents </Service_Description>
<Service_URL_Address>http://localhost/Services/MyRepresentorService</Service_URL_Address>
<Service_Cost>52</Service_Cost>
<User_Information>
<Service_Requester_Name>Application DffEQ</Service_Requester_Name>
<Service_Req_IP_Address>192.200.168.33</Service_Req_IP_Address>
<QoS_Class>Guaranteed-Service QoS</QoS_Class>
<Temporal_QoS>
<Start_Time>20/6/2002 11:05</Start_Time>
<End_Time>20/6/2002 13:18</End_Time>
</Temporal_QoS>
<Appl_QoS>
<Availability> > 99.5% </Availability>
<Accessibility> > 90% </Accessibility>
<Reliability> High </Reliability>
<Security> Normal </Security>
</Appl_QoS>
<Middleware_QoS>
<Node_IP_Address>135.200.50.101</Node_IP_Address>
<CPU_Count>4</CPU_Count>
<Real_Storage>240 MB</Real_Storage>
<Disk_Storage>40 MB</Disk_Storage>
</Middleware_QoS>
<Network_QoS>
<Source_IP>192.200.168.33</Source_IP>
<Dest_IP>135.200.50.101</Dest_IP>
<Throughput>256 Kbps</Throughput>
<Packet_Loss> 10% </Packet_Loss>
</Network_QoS>
</Service_SLAs>

Figure 8.2 Service Specification

<?xml version="1.0" encoding="UTF-8"?>
<service-details type="…" status="ok">
<service>
  <name>…</name>
  <provider>…</provider>
  <price>
    <hardware>…</hardware>
    <software>…</software>
  </price>
  <address>…</address>
  <description>…</description>
</service>
<service>
  ...
</service>
<service>
  ...
</service>
</service-details>

Figure 8.3 Service specification containing economic characteristics.
8.3 Modification of registration mechanism

As part of an effort to interoperate with Grid Services, both the Globus container and Java-RMI HHs (available as part of the URDS prototype) were extended to enable communication and introspection. The code including modifications can be found in Appendix. The representor Grid Services were registered with the modified Globus Container. Consequently, the implementation setup consisted of the URDS prototype containing the modified HHs and the Globus toolkit incorporating the modified Globus container with registered representor Grid services.

In the case of representor example, a knowledge base is assumed to be available, based on which a user query for representor system is issued. These queries are decomposed and transmitted to URDS by a Client (simulating the System Generator). The modified HHs in the URDS examine the components in their meta-repository. In the meantime, HHs communicates with the Globus Container and receives the specifications of the components. On successful matching, the specifications are returned to the Client, which generates the results by invoking the components.

URDS is a constituent of a component-based framework i.e. UniFrame wherein, required components are discovered by URDS and then integrated into applications. The prior question answered in this chapter was the issue of interoperation of URDS with Grid Services. Since they were found to be interoperable, the next question to be answered is whether the discovered Grid services can be integrated into a Grid application. Consequently, as part of the effort to address this question, component-based approaches for the Grid were evaluated for their feasibility. During the evaluation, it became apparent that for a variety of reasons (explained in section 8.4.4), a comprehensive framework for building quality-aware component based Grid systems was not available. Hence, a preliminary approach for the Grid incorporating UniFrame concepts into it, known as the GridFrame was evaluated by comparing it with a popular component-based Grid approach. This evaluation involved analyzing a simple case study in the domain of information filtering. Subsequently, since the GridFrame approach appeared promising,
the URDS prototype was then experimented using Grid components also known as Grid Services.

8.4 Background

In the initial exploration of the Grid Domain, it was found that majority of the approaches focused on high performance issues and therefore concerned with hardware related resources only. As a result, in the next few sections, the need for a software component framework for Grid is established with the identification of the requirements for such a framework. Subsequently, a few component-based approaches currently available in Grid are compared with respect to these requirements.

8.4.1 Introduction to Grid

In many commercial areas, Software Component Frameworks [SR104] have long been established as the standardized way of building distributed applications from well tested reliable sub-units. With the huge complexity of modern day programming, it is implausible that a single programmer can build a large application from scratch. Modern programmers compose large distributed applications by integrating pre-fabricated components from component libraries or from the open source community.

Traditionally, the predominantly scientific Grid world has been slow in embracing these software engineering concepts, and has focused only on high performance issues. But with the increasing mainstream usage of Grid Computing, software component composition and reuse through service oriented Grids have become an increasing need for today’s Grid mainstream and research programming projects.

For purposes of this thesis, a Grid experience is defined as the Grid utilization process, which runs from the Grid application creation to the final deployment and execution of the application. Most of the existing Grid approaches, such as [CAC00, CON03] are targeted at the latter, i.e., deployment and execution of the application, and tackle challenges such as the requirements analysis, selecting hardware resources and providing
middleware facilities for enabling a user to deploy a Grid application(s). Typically, these approaches assume that the Grid application has already been designed and pre-customized to the Grid deployment phase. Only a few approaches (e.g., [TRI03, FUR01, SRI04, and BUB03]) address the challenges of providing a software component framework for creating component-based Grid applications using pre-built components.

Before elaborating on any of the current Grid software component framework approaches, it is necessary to first identify the requirements and constraints that the Grid places on such a framework. In the current Grid scenario (and for purposes of this thesis), components are services primarily based upon Open Grid Services Architecture (OGSA) [FOS02] framework, which aims at defining a common, standard, and open architecture for grid-based applications. These components are deployed in OGSA containers with associated service data indicating their characteristics. These components are characterized by their dynamic nature, implying that they might be available for varying intervals of time, with frequent changes of their availability status. Also, the components might be heterogeneous and distributed in nature. Hence, a software component framework for Grid must fulfil the following requirements;

- Should be able to tackle heterogeneity in language, model, technology and architecture, etc.
- Allow a way for the dynamic discovery of components.
- Provide a means for ascertaining non-functional attributes such as QoS of individual components as well as the integrated system.
- Provide a user friendly mechanism for the system integration.

In the next section, existing component based approaches for Grid are evaluated with respect to these requirements.

8.4.2 Existing Approaches

In the current Grid scenario, a user developing a Grid application from pre-built components has to either write scripts in a XML representation [FUR01], write scripts in a domain specific language [SRI04, TEL02], or employ application workflow diagrams
and graphical modelers [TRI03, BUB03]. In the following sections, existing work based on these approaches are discussed and evaluated.

8.4.3 Script based approaches

8.4.3.1 ICENI
Conforming to the script-based approach, ICENI [FUR01] provides a component based framework for creating Grid applications from pre-built components, discovered from private as well as public meta-repositories. Whenever a new component is developed, a component specification is created in terms of a CXML (Component – eXtensible Markup Language) document, describing the component’s behaviour and interface. Implementations of the specification are placed in meta-repositories, with meta-data describing their performance characteristics and resource requirements. Based on a problem definition, composition of these implementations to form a Grid application is described in terms of an application description document, which is a CXML specification of the complete component composition. At runtime, the application description document is converted into an active Java representation by utilizing the component specification meta-data within the repository. The run-time representation is used to map the application requirements into available resources, based on requirements and implementations’ meta-data. While this approach does attempt to provide a component-based Grid framework, for satisfying a few of the aforementioned requirements for a Grid framework, it does not succeed on several fronts. Firstly, it does not tackle heterogeneity at the component model level, only at the language level. The component CXML document does not provide a comprehensive enough QoS catalogue for comparing and matching components attributes, or for the prediction of QoS values at the system level. Since CXML does not accord the flexibility to express an application in terms of a hierarchy of possible subsystems, even a small change in the problem definition implies that the application CXML has to rewritten.

8.4.3.2 GRADS Project [GAN04]
This approach aims at providing domain specific high-level programming systems for problem solving environments, by which end users can rapidly develop new applications
using standard notations of their problem domains. Here, the pre-built components are organized into optimized libraries, using a set of library design and specification strategies. Also, the application library is annotated with the following details: a) program transformation specifications detailing how program sequences can be replaced with equivalent, but more efficient sequences, and b) sample calling programs illustrating typical usage patterns. In a separate step, mappings from scripting languages to library implementation language are provided. The scripting languages enable usage of components as primitive objects and define operations on them.

A translator generator processes the enhanced library for hours or days and produces a translator executable. Using any of the allowed scripting languages, the user has to write an application script involving operations, initiation and configurations of the primitive objects to construct a Grid application. The scripts are then translated and compiled using the above domain specific translator. While the approach promises significant improvement in performance issues, it does not provide a software component framework per se. As a result, there are no facilities for discovery of components, predicting QoS of individual components and integrated system. Even though end users can develop new applications using their domain specific notations, there is a significant amount of latency curve associated with learning a new scripting language. The developed applications are invariably individualistic handcrafted solutions with slim possibility of reuse.

**8.4.3.3 XCAT3**

XCAT3 [SRI04] is based on the Common Component Architecture (CCA) model, providing a component based framework by which components (CCA components and/or Grid services based on OGSI) can be instantiated and connected together. Each component contains provides-ports indicating functionality the component provides to other components and uses-ports indicating the needed functionality from other components that the component needs to function. As a result, each component in the XCAT3 framework consists of port interfaces, port implementations and the component implementation. *Builder* service APIs in Java are provided by which instances of components can be created and composed together to form a distributed application. Also, APIs for querying services of components, destroying component instances, and
invoking methods on instantiated components are provided. An example of a XCAT3
script illustrating component instantiation, connection and starting is shown in Figure 8.4.

```python
# create live instances of classifier and representor components
cca.createInstance(classifier)
cca.createInstance(representor)

# connect their ports, so that the #classifierComponent can connect to the
#representorComponent to make remote calls
cca.connectPorts(classifier, "classifierPort", representor, "representorPort")

# start the components
# For goPort
portClassName = "intf.ports.XCATGoPort"
portType = "http://www.cs.iupui.edu/xcat/ports/go"
providesPortName = "providesGoPort"
methodName = "go"

# no parameters
methodParams = zeros(0, Object)
cca.invokeMethodOnComponent(classifier,
    portClassName,
    portType,
    providesPortName,
    methodName,
    methodParams)
```

Figure 8.4 XCAT3 Code fragment for two components `classifier` and `representor`.

XCAT3 suggests a component framework, by which users can use pre-defined APIs for
querying running services for their service data, make a decision about their viability and
use Builder APIs (or Application Factories) for building an application encompassing
these components. While this approach sounds promising, the resulting applications are
basically handcrafted solutions with limited reuse. Users do not have any option for any
preliminary testing of the integrated application, implying that they cannot make any informed decisions about the QoS of the integrated application before the actual deployment. In addition, users have only a manual recourse for discovery of components and analyzing their service data. Also, mainstream Grid users might not have the high level of programming skill as assumed by the approach.

8.4.4 Application workflows

8.4.4.1 Cross-Grid

In this approach, application workflows are relied upon for automating the Grid application creation process. Here, the user supplies an initial application workflow document such as Figure 8.4 detailing the components, their interactions and the workflow. Here, components are CCA based implying provided and uses ports, and are developed independently, and are registered with OGSA registries. A flow composer parses the user workflow diagram, performs component lookups based on Port type or ID attributes and builds alternative final workflow documents with every distinct set of matched components. Finally, the user can choose a final workflow document corresponding to his view of the integrated system.

<?xml version="1.0"?>
<!—Distributed Information Filtering System
Authors: Pradeep, Rajeev
Date: 15 June 2004-->
<workflowDefinition appName="DIFSApp" maxComponents="10"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=" ../schemas/wd.xsd">
  <initComponent id="init1" name="Classifier"/>
  <finalComponent id="final1" name="CentroidGenerator"/>
</workflowDefinition>

Figure 8.5 An initial workflow document example
This approach has intrinsic limitations similar to the script-based approaches in that it places undue importance on the expertise of the user, does not offer any QoS testing and finally results in handcrafted, individualistic solutions with restricted reusability.

8.4.5 Miscellaneous Approaches

In approaches used in Triana [TRI03], graphical modelers and toolkits are employed, which provide the user a higher level of abstraction than application workflows. A user creates an application by dragging, dropping and deleting components and associated relationships in a graphical window. Here, though a greater level of convenience than the application workflow approach is accorded to the user, the other limitations still remain. Other approaches such as ECSF [ECS03] provide a distributed computing paradigm suitable for multidisciplinary Grid applications, but are also limited by the same problems since their underlying principles are not based on the component paradigm. GridLab [ALL04] does not provide a component creation framework, but focuses on providing high level application toolkits that interface between user applications and Grid middleware packages like Globus [FOS02].

8.4.6 Observations

To summarize the related approaches, many challenges such as heterogeneity of components, their resource discovery and QoS prediction etc., associated with creating a component based Grid framework have currently not been addressed satisfactorily. Since the URDS relies on component behavior matching both in terms of syntax as well as the associated semantics and is a precursor to the process of integration of components, a component based Grid framework addressing the above challenges is a necessity for making any evaluation of URDS in Grid. Subsequently, to address these challenges as part of the effort to facilitate the evaluation of URDS, a preliminary approach known as GridFrame [ADCOM] incorporating UniFrame concepts was evaluated for Grid. GridFrame addresses the conception of a semi-automated Component Based Grid System (CBGS) development process involving the dynamic discovery of distributed Grid components, the generation of the composed system and the validation of quality requirements from the UniFrame perspective. In conception, GridFrame is visualized as a
comprehensive framework incorporating Grid services deployment, discovery, and integration into a Grid application and subsequently using existing approaches such as [ALL00, CON03] for a later deployment on dynamically discovered hardware resources. Due to the limited scope of this thesis, only the evaluation of UniFrame concepts in GridFrame is considered here. A case study is included in the next section to illustrate the UniFrame concepts included in GridFrame as well to compare the approach with a currently used Grid component-based approach.

8.5. CASE STUDY

Out of approaches described in 8.4.1, using OGSA services to leverage existing services to form complex distributed solutions is a popular option now. To contrast the OGSA-based approach with the GridFrame approach, a case study from the domain of distributed information filtering is considered. Typically, a distributed information filtering service (DIFS) reduces information overload by supporting personalization of long term information needs of a particular user or group of users with similar needs. Here, a DIFS based on DSIFTER [RAJ97] is considered. Using user profiles and periodic feedback, the DIFS rank-orders documents and performs a mapping from the space of documents to the space of user relevance values. Typically, in a DIFS, more often than not, documents exist at diverse sites and are received by the user through disparate, independent channels. The task of storing such documents, before filtering is handled by a data acquisition service (DAS). A Representation Service (RS) converts these stored documents into structures, which can be efficiently parsed without the loss of vital content. A Classifying Service (CS) classifies these stored structures using clustering algorithms on the basis of user interests specified in a User Profile Service (UPS). The UPS is continuously updated using reinforcement learning algorithms to reflect current user interests. A User Interface Service (UIS) displays the ranked documents and collects user feedback for user profile learning. A Federation Service (FS) enables interconnection of DIFS systems.
8.5.1 Using OGSA Grid Services

Using OGSA Grid Services, a DIFS system can be created in the following manner:

- Assuming a complete DIFS service is not available, a Grid user has to decompose his requirements to form a list of the previously identified Grid services that would aggregate to form a DIFS Grid service. Figure 8.6 illustrates an example of the process by which a user can build a DIFS system using Grid services. It contains the following steps:
  - The user contacts a known registry to identify service providers who can provide the required services and presents a list of requirements including cost and performance.
  - The handles for needed service factories that match user requirements are returned to the user.
  - The user supplies instantiation details such as needed operations, etc., and initial lifetimes for the service instances.
  - If agreeable, the service providers create service instances with user supplied details.
  - Using the service handles, the user writes application programs for aggregating the services to form a DIFS system.

Visions of enterprises using Grid Services approach to dynamically compose new applications such as above to address the specific needs of the business at any point in time have been painted. But there are several limitations with this approach, particularly in regard enterprise applications. The resulting new applications are basically handcrafted solutions with limited reuse. Any slight change in the problem definition, for instance using a .NET display component, if a previously used Java component is not available, will entail a complete rewrite of the previous solution. Also, users do not have options for any preliminary testing of the integrated application, implying that they cannot make any informed decisions about the QoS of the integrated application before the actual deployment. In addition, most of the techniques for discovery of components assume that the components are homogeneous in nature and rely on simple interface matching and
component context dependencies, which are not sufficient enough for a process, which is a precursor for composing high confidence Grid systems. Also, the approach assumes a high level of programming skill of users, which is typically not the case with mainstream Grid users. These are serious drawbacks, particularly considering that mainstream domains such as enterprise applications have stringent requirements about quality and reusability of applications.

8.5.2 Creating a DIFS system with GridFrame

For the sake of brevity, the focus is mainly on the overall outline of carrying out the development of a partial Knowledge Base (KB) as well as the discovery mechanisms of
GridFrame possibly resulting in the omission of in depth details, which can be referred to, using the associated references.

8.5.2.1 KB Process

Due to space constraints, only some of the aspects of KB like feature diagrams and use-cases depicting the configuration knowledge of the integrated system are shown here. In the KB, the components making up the DIFS system are identified along with their functional characteristics such as required interfaces, provided interfaces, etc., and non-functional characteristics such as QoS metrics. In addition to the feature diagram, the KB contains sequence diagrams, which capture the behavioral aspects of the system. Sequence diagrams, such as Figure 8.7 illustrate the interaction of components in the system with each other as well as with users. Figure 8.9 shows a feature diagram illustrating the DIFS family of sub-systems which can be possibly built with the identified components.

![Sequence Diagram for search](image)

Figure 8.7 Sequence Diagram for search

The details about the concept of features and the notation used for describing a feature diagram were proposed in [ZHI03]. The given feature diagram indicates possible architectural alternatives for a DIFS. For example, two possible alternatives for a DIFS
could be: version (a) made up of RM, TM, RP, CL, SP, TI and version (b) made up of WM, RM, TM, RP, CG, CL, CP, DM, SM, EM, GM, ECM, CM. As indicated earlier, depending upon the input query presented by the system integrator, an appropriate alternative will be selected during the system development process. Each node in the feature diagram indicates an abstract component, which will be described by its corresponding UMM specification. The component specification in UMM is a multi-level contract [UMM02] with bookkeeping information such as component id, domain name, and algorithmic, technological information such as function name, algorithm name etc. For example, a partial UMM-specification for a typical classifier could be:

1. Component Name: Classifier
2. Domain Name: Information Filtration
3. System Name: InformationFilter
4. Informal Description: Provide classification service for documents.
5. Computational Attributes:
   5.1 Inherent Attributes:
      5.1.1 id: N/A 5.1.2 Version: version 1.0 5.1.3 Author: N/A 5.1.4 Date: N/A 5.1.5 Validity: N/A 5.1.6 Atomicity: Yes 5.1.7 Registration: N/A 5.1.8 Model: N/A
   5.2 Functional Attributes:
      5.2.1 Function description: Act as classification server for documents in system.
      5.2.2 Algorithm: N/A 5.2.3 Complexity: N/A 5.2.4 Syntactic Contract
         5.2.4.1 Provided Interface: IClassification
         5.2.4.2 Required Interface: NONE
      5.2.5 Technology: N/A 5.2.6 Expected Resources: N/A 5.2.7 Design Patterns: NONE 5.2.8 Known Usage: Classification of documents 5.2.9 Alias: NONE
6. Cooperation Attributes:
   6.1 Preprocessing Collaborators: Representor
6.2 Postprocessing Collaborators: NONE

7. Auxiliary Attributes:
   
   7.1 Mobility: No 7.2 Security: L0 7.3 Fault tolerance: L0

8. Quality of Service Attributes
   
   8.1 QoS Metrics: throughput, end-to-end delay
   8.2 QoS Level: N/A 8.3 Cost: N/A 8.4 Quality Level: N/A
   8.5 Effect of Environment: N/A 8.6 Effect of Usage Pattern: N/A

9. Deployment Attributes: N/A

Figure 8.8 A UMM example for Classifier

Once the KB has been developed, component developers are free to develop and deploy components using their choices of technology, language, etc., according to the specifications in the KB. The developed concrete components have to strictly adhere to the KB abstract specifications, but can be implemented in different technologies, algorithms etc. with corresponding QoS values. For example, one Representor Module (RM) can be implemented in .NET technology using a vector space model [20] with 340 ms turnaround time while another RM can be implemented in Java RMI using a different model with 320 ms turnaround time, with corresponding QoS attributes.

8.2.2.2 Discovery of components and Integration of the system

After the creation of the KB and deployment of components, a system developer can query for an instance of a system using a tabular graphical interface [ZHI03], containing different options for the different possible systems. For example, the options could be a basic DIFS with minimal functionality incorporating instances of RM, TM, RP, CL, SP, TI and GM or an advanced DIFS with increased functionality incorporating instances of WM, RM, TM, RP, CG, CL, CP, DM, SM, ECM, GM. For example, the system developer might query for a simple DIFS with QoS values such as the maximum permissible end-to-end delay and minimum throughput for the system specified as 1800 ms and 400 op/s respectively. Using the decomposition model in [CHA03], the given QoS requirements for the whole system are decomposed into the QoS requirements for
each of the constituent components. By means of the KB and the QoS requirements, for each of the components making up the chosen system, a query is created. These queries are presented to URDS for discovering concrete instances of the components, which can match the requirements. When the URDS receives the requests, a subset of headhunters in the specified domain (in this case, distributed information filtering) is contacted for concrete instances of the components. These headhunters search their local meta-repositories and perform syntax, semantic and QoS matching of the stored specifications with the queries. Each query has an associated timestamp, depending on which the queries can be propagated to other headhunters. As explained in the GridFrame process, the system developer chooses from the listed components on basis of QoS values, (available from the service data), and uses the GridFrame System Integrator to test and build the integrated system.

A brief comparison of the two approaches suggests the following:

Figure 8.9 Feature Diagram of DIFS
• As opposed to handcrafting, the use of a KB in GridFrame enables the creation of standardized solutions by which the reusability of individual components as well as the integrated system is improved.

• Quality of service theme is maintained throughout the GridFrame process, as a result of which predicting and monitoring of component performance at the component level as well as system level is possible.

• GridFrame accommodates heterogeneity by which components can be implemented in different models and technologies.

• By providing a semi-automated framework for composing services, GridFrame ensures that user intervention is minimized, enabling novice end users to integrate systems.

8.5.2.3 Observations on GridFrame

The proposed framework provides a semi automated approach for building Grid systems from pre-built Grid services using concepts of software engineering. Using the framework, it is possible for end users to both predict and reason about the quality of the integrated system as well as the individual services. Although a simple example is provided here, the principles are general enough to be applicable for both mainstream and research Grid projects. The development of Grid systems involves both construction and deployment of the system. Here, only the construction issues of a component based Grid system using a KB were discussed. Utilizing the KB for deployment issues such as assessing hardware resource requirements, selecting ideal resources, etc., is the focus for current research. The GridFrame process has been investigated using the Globus toolkit and the current UniFrame infrastructure by means of trivial examples. While the results are promising and show that such a process is plausible, validation on a realistic, large scale scientific or mainstream Grid application is one of the key research goals for current and future efforts.
8.6 Conclusions

This chapter surveyed the available component-based approaches in Grid and listed the addressed challenges and drawbacks of each approach. A preliminary approach using UniFrame concepts in Grid known as GridFrame was compared to a Grid component-based approach with promising results. Subsequently, it was found that a URDS implementation can be used with Grid Services and OGSA containers. Although only a simple example has been implemented, more complex systems can (in the author’s opinion) be used. This chapter concludes the various objectives of the thesis. In the subsequent chapter, the conclusions and future work of the thesis are presented.
9. Conclusions and Future Work

During the course of the thesis, the different objectives of the thesis were successfully completed. In this chapter, the conclusions of the thesis pertaining to these objectives are presented in Section 9.1. Section 9.2 lists of the contributions of the research while Section 9.3 details the possible future work for the thesis. Finally, the chapter concludes by listing a summary of the thesis in Section 9.4.

9.1 Conclusions

The first objective of the thesis was to empirically evaluate the scalability of the URDS prototype. Although URDS [SIR02] proposed a complete discovery service, only a basic discovery service prototype was implemented encompassing a minimal set of composed entities. Extensions and enhancements for the discovery service in the form of algorithms for query propagation and fault handling were implemented in this thesis. Using a series of experiments, it is inferred that the URDS is scalable. Insight and lessons learnt in the experimental testing and validation of these algorithms were employed in designing and implementing different profiling algorithms. To corroborate these algorithms with an empirical evaluation on a large scale, a computer simulation was also designed and implemented. By experimental validation, it is concluded that the profiling algorithms improve the quality of the discovered components with regard to the shorter turn-around time and appropriateness to the query. Also, it is reasoned that each of the implemented algorithms is suitable for a particular objective, for example, the long term algorithm for clustered queries. Also, an URDS implementation was evaluated in the Grid domain. As a prelude to the evaluation, component-based Grid approaches were evaluated for their feasibility. Since there were not complete solutions available, UniFrame concepts were incorporated in a new component-based Grid approach called GridFrame and compared with a current component-based Grid approach, using a case study. It is concluded that the GridFrame approach is promising and consequently established by another case study that URDS can be used in the Grid domain. Overall, the conclusions can be summarized as below;

- The URDS architecture as proposed in [SIR02] is scalable.
• The profiling algorithms enable the discovery of better rated components in a shorter turn-around time.

• The performance of each algorithm is based on the number of executing entities, relation between queries as well as the number of permissible branches/children for each HH.

• Current component-based Grid approaches are inadequate for building quality aware Grid systems. The component-based Grid approach GridFrame can be used for building quality aware Grid systems.

• The URDS implementation can be used in juxtaposition with Grid services.

9.2 Contributions of thesis

The contributions of this thesis are:
• It provides experimental evaluation, by which the scalability of the URDS was determined.

• It includes profiling algorithms for improving the performance of the discovery system.

• It provides the design and implementation of a URDS simulation for evaluating the profiling algorithms against a large set of constituent entities.

• Using the URDS simulation, the thesis provides an empirical evaluation by which the different types of profiling algorithms were evaluated with respect to their performance and their suitability judged against conditions of the discovery system.

• It proposes a preliminary component-based approach for Grid applications, GridFrame based on URDS and UniFrame, by which components can be dynamically discovered and composed on the Grid.

• It provides a theoretical case study for evaluating the feasibility of GridFrame
9.3 Future Work

One of the objectives of this thesis was to study the different types of query propagation algorithms, such as the short term, long term, etc., and their effect on the performance of the discovery process. Each algorithm was found suitable for particular combinations of the conditions (such as type of queries, no of nodes, etc.). Based on the requirements and hence the chosen type, comprehensive algorithms such as [MUK03] should be implemented. Also, different types of profiling algorithms have been implemented in only QM and HH entities. Profiling for other entities such as AR can be incorporated to improve the performance of the discovery service. Efforts such as [BAR05] are a step in this direction.

GridFrame has been proposed as a two-stage approach for developing and deploying component based Grid systems. In this thesis, only the construction issues of a component based Grid system using a knowledgebase were discussed. The GridFrame process has been investigated using the Globus toolkit and the current UniFrame infrastructure by means of small examples. While the results are promising and show that such a process is plausible, a validation on a realistic, large scale scientific or mainstream Grid application should be one of the key research goals for future efforts. On the other front, deployment of the developed Grid services has not been considered. Utilizing the GDM for deployment issues such as assessing hardware resource requirements, selecting ideal resources, etc., should be investigated. For deployment, works such as [CON03], [ALL00] and [FUR02] can be considered.

9.4 Summary

In this thesis, the URDS architecture was enhanced by augmenting it with query propagation algorithms and ascertaining it’s scalability by experimentation. This thesis also provided profiling algorithms for increasing the performance and the quality of the discovered components from URDS. A URDS simulation was also designed and implemented to validate the algorithms against a large set of constituent entities. As a
prelude to evaluating URDS in Grid, the thesis also proposed a preliminary component
based approach for Grid applications, based on URDS and UniFrame, by which
components can be dynamically discovered and composed on the Grid. Thus the URDS
architecture was validated by ascertaining its scalability, augmented by profiling
algorithms and enhanced by showcasing its versatility in a different domain such as Grid.
LIST OF REFERENCES


[AVS05] – Altavista http://www.altavista.com/about/. Last referenced: April 25, 2005


[TUM04] Tummala K., “Glue Generation Framework In UniFrame for the CORBA-JAVA/RMI INTEROPERABILITY “ M.S. Project TR-CIS-0302-03, Department of Computer & Information Science, Indiana University Purdue University Indianapolis, May, 2004


