Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Lahiru Sandakith Pileththuwasan Gallege

Entitled
Design, Development and Experimentation of a Discovery Service with Multi-level Matching

For the degree of Master of Science

Is approved by the final examining committee:

Prof. Rajeev R. Raje
Chair

Prof. James H. Hill

Prof. Mihran Tuceryan

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Prof. Rajeev R. Raje

Approved by: Prof. Shiaofen Fang (Head of the Graduate Program) 03/01/2013 (Date)

DESIGN, DEVELOPMENT AND EXPERIMENTATION OF A DISCOVERY SERVICE WITH MULTI-LEVEL MATCHING

A Thesis

Submitted to the Faculty

of

Purdue University

by

Lahiru Sandakith Pileththuwasan Gallege

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2013

Purdue University

Indianapolis, Indiana

To *Amma* and *Thaththa*

ACKNOWLEDGMENTS

Being a graduate student at the Department of Computer and Information Science at IUPUI (Indiana University-Purdue University, Indianapolis) has been an immense learning experience for me. The knowledge gained will be valuable to my career, as I step into the computer science research community. I will always cherish my experience and memories of working as a teaching assistant and research assistant as a part of this Institution. I would like to take this opportunity to remember many people who have been very supportive throughout my graduate studies.

First and foremost I would like to thank my advisor, Professor Rajeev R. Raje, for his constant encouragement and guidance through the courses of my graduate studies. He constantly encouraged me to achieve higher goals and help me to realize my goals as a research student. I would also like to thank Prof. Mihran Tuceryan and Prof. James Hill for agreeing to be part of my Thesis Committee and providing their valuable feedback.

I would like to thank my colleagues at our lab (SL 116) for being there to support my research and experimentation. Special thanks goes to my colleague Ketaki for her assistance with the development and testing of the proURDS prototype. I would also like to thank the department staff and IT support staff (especially Nicole, Nancy, Scott and Debby) for their support. I like to thank all the faculty and colleagues at the Department of Computer and Information Science for their cooperation. Also I would like to thank the staff of the Purdue School of Science Graduate Office (especially Debra and Mark) for their help during the thesis formatting reviews.

Finally, I would like to thank my parents, Chandima and Rehan for their unconditional love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

ABBREVIATIONS

DCS	Distributed Computing Systems
CBSD	Component Based Software Development
URDS	UniFrame Resource Discovery Service
MLM	Multi-level Matching
QoS	Quality of Service
UA	UniFrame Approach
AR	Active Registry
HH	Headhunter
SMM	Service Management and Monitoring
DSM	Domain Security Manager
proURDS	Enhanced UniFrame Resource Discovery Service

ABSTRACT

Pileththuwasan Gallege, Lahiru Sandakith. M.S., Purdue University, August 2013. Design, Development and Experimentation of a Discovery Service with Multi-level Matching. Major Professor: Rajeev R. Raje.

Emerging technologies and demanding applications have forced the transition of the computing paradigm from a centralized approach to a distributed approach. This shift leads to the concept of Distributed Computing Systems (DCS). The traditional way of software development lacks the capabilities to address the challenges in software realization of large scale DCS. Out of many methods proposed to develop DCS, one promising approach is the Component Based Software Development (CBSD).

The UniFrame approach, an approach developed at IUPUI, follows the concepts of CBSD and addresses the design and integration complexity of DCS. The UniFrame approach provides a comprehensive framework which enables the discovery, interoperability, and collaboration of components via generative software techniques. It unifies existing and emerging distributed component models to a common meta-model. This framework enables the creation of high-confidence DCS using existing and newly developed distributed heterogeneous components. One essential part of UniFrame is the UniFrame Resource Discovery Service (URDS). URDS is used for the discovery of components that are deployed on the network. Initially, the architecture for URDS was proposed in terms of addressing the objectives of dynamic discovery of heterogeneous software components and selection of components to meet the necessary functional as well as non-functional requirements (Quality of Service - QoS). Many contracts contain information in terms of functional and QoS hence, the dynamic discovery of components which are deployed over the network is a non-trial task. The majority of the components' repositories provide a simple search technique

which is based on string matching of listed attributes. However, the search space of components is large and the information provided by each component is also non-trivial to be represented as attributes. Therefore, a simple attribute-base search is not sufficient to address the requirements of users.

Due to the limitations of the simple attribute based representation of contracts and basic textual matching, the URDS proposes the concepts of Multi-level contract representation and Multi-level Matching (MLM). The URDS contract provides information at many levels including: General, Syntactic, Semantic, Synchronization, and QoS. Matching of component contracts is performed according to the valid matching operations proposed at each of the levels. This narrows down the search space according to the individual requirements at a corresponding level. Hence, based on each operator's capability, related components have a better chance of being included in the result list. However, the validation of a system which integrates URDS and MLM was not present to be experimented. Therefore, as the main contribution of this thesis, the proURDS was developed as a distributed setup by enhancing the URDS architecture which was deployed over the network with real component contracts.

The contribution of this thesis focuses on addressing the challenges of improving and integrating the URDS and MLM concepts. The objective was to find enhancements for both URDS and MLM and address the need of a comprehensive discovery service which goes beyond simple attribute based matching. It presents a detailed discussion on developing an enhanced version of URDS with MLM (proURDS). After implementing proURDS, the thesis includes details of experiments with different deployments of URDS components and different configurations of MLM. The experiments and analysis were carried out using proURDS produced MLM contracts. The proURDS referred to a public dataset called QWS dataset. This dataset includes actual information of software components (i.e., web services), which were harvested from the Internet. The proURDS implements the different matching operations as independent operators at each level of matching (i.e., General, Syntactic, Semantic, Synchronization, and QoS). Finally, a case study was carried out with the deployed

proURDS. The case study addresses real world component discovery requirements from the earth science domain. It uses the contracts collected from public portals which provide geographical and weather related data.

1 INTRODUCTION

The current software systems are inherently complex in nature. With advancement of computing architectures, new demanding applications and technical breakthroughs have forced the transition of computing paradigms from a centralized approach to a distributed approach. This has led to the concepts of Distributed Computing Systems (DCS).

The traditional method of software development lacks the capability to address the challenges (e.g., heterogeneity and scalability) present in Distributed Computing Systems. Out of many proposed approaches for realizing DCS, one promising approach is the Component Based Software Development (CBSD) [1]. One such realization of CBSD is the UniFrame Approach (UA) [2,3]. It provides a comprehensive framework by unifying existing (and emerging) distributed component models to a common meta-model. The UniFrame meta-model enables the discovery, interoperability, and collaboration of components via generative software techniques.

The UniFrame framework enables creation of high-confidence DCS using independently developed and deployed distributed heterogeneous components (or services, i.e., the terms component and services are used interchangeably and refer to the publicly discoverable software entities). Before such systems are created, there is a need to locate appropriate individual components. This task in UniFrame is delegated to a special entity called the UniFrame Resource Discovery Service (URDS) [4,5]. The entity is responsible for the discovery of heterogeneous services that are deployed on the network. The URDS involves matching and selection of software components based on component contracts (i.e., software specifications).

Many component contracts contain information in terms of functional and QoS hence, the dynamic discovery of components which are deployed over the network is a non-trial task. The majority of the components repositories (e.g., UDDI) pro-

vide a simple search technique which is based on string matching of listed attributes. However, the search space of components is large and the information provided by each component can be too non-trivial to be represented as attributes. Therefore, a simple attributed base search is not sufficient to address the requirements of users. Performing simple attribute based matching could either produce a result list which consists of many components or a result list which fails to include a related component. The reasons for the above problems can be: 1) the provided few attributes are common with many components, however most of the components are not related to the search, or 2) the attributes are directly not matching with a component however that component is related to the search. Therefore, based on these complex contracts, the process of matching and selection of the software components presents a challenge.

Due to these limitations of textual matching the concept of the Multi-level Matching (MLM) [6] has been proposed. It is based on the design by contract principles proposed in [7, 8]. To perform MLM, the contract should provide specific details at all levels including general, syntactic, semantic, synchronization and QoS. Once the details are available, the matching of component contracts is done using the appropriate matching operators proposed for all the levels. This narrows down the search space while filtering the existing components according to the requirements at each level. For example, if the result list is large the operators at each level can perform a strict operation or if necessary the operators can relax their matching criterion based on a type hierarchy to include subtypes. The initial experiments of MLM were carried out using a prototype with a database of contracts and database query language implementation of matching operators.

The initial prototype of URDS [9] was developed to experiment on the high-level objectives of discovery of heterogeneous software components from software contracts of components meeting the necessary functional as well as non-functional requirements including QoS. However, the validation of a system which integrates URDS and MLM was not present to be experimented. The initial experiments used a database of

contracts and database query language implementation of matching operators. This included only a proof of concept framework, but not in an actual distributed system setup. The simulations indicated the effectiveness of MLM in locating the most relevant services for a particular query. Also, the experiments did not provide a merger of the discovery and the matching parts of the URDS. These experiments were reported in [4, 5]. Therefore, as the main contribution of this thesis, the proURDS was developed as a distributed setup by enhancing the URDS architecture which was deployed over the network with real component contracts.

The contributions of this report focus on addressing the challenges of integrating the two concepts of distributed URDS and MLM within the context of the UniFrame approach. The resulting setup is called the proURDS. The objective was to come up with enhancements for both URDS and MLM by validating the need for a comprehensive Discovery Service. From now onwards the URDS refers to the initial prototype and proURDS refers enhanced version of URDS (proURDS). The later section of the thesis discusses the challenges in producing proURDS including implementation of the matching operators. The proURDS architecture is validated using software component contracts from QWS Dataset [10]. This dataset contains information from existing services which were harvested from the Internet. The proURDS produced MLM contracts by referring to the QWS dataset. The experiment and result sets were produced by matching contracts at each level. In summary, the goal of the proURDS and its experimental analysis was to indicate the benefits of multi-level matching as opposed to a traditional string matching. Also, another goal was to explore the matching process with a performance evaluation of different queries.

1.1 Objectives

The specific objectives of this thesis are :

- To enhance the existing URDS architecture by incorporating the MLM matching operators.

- To deploy the enhanced URDS (proURDS) in a distributed setup.
- To experimentally validate proURDS by using the QWS dataset [11].
- To provide a case study of the system using components from earth science domain [12].

1.2 Organization

This thesis is organized into eight chapters. Chapter 1 provides introduction and objectives and Chapter 2 presents the related approaches. Chapter 3 describes the summary of previous work as necessary background information for the UniFrame approach. Chapter 4 presents the design, development and integration challenges and proposed solutions (proURDS) pertaining to integration of the URDS with Multi-level matching. Chapter 5 presents the experimentation details with different configurations of proURDS. Chapter 6 consists of experimental results and their detailed analysis. Chapter 7 contains a case study from the domain of Earth Sciences. Chapter 8 presents conclusions and future work. Finally, the supplementary appendix covers some details of source code.

2 RELATED WORK

Efforts of designing discovery systems can be classified according to the semantics of the matching and customization. Most of the current efforts do not go beyond simple text based name-value pair matching. Also, most component (also service, i.e., terms components and services are used interchangeably) selection efforts do not consider the notion of customization with respect to service matching. Based on the matching techniques current discovery systems can be divided into three main categories: simple attribute-based matching, ontology-based matching, and hierarchy-based matching. The notion of discovery is also recently used in Cloud Computing (CC) and hence, a brief survey of Cloud-based efforts are also included in this chapter.

2.1 Simple attribute-based matching

In this category, the attribute-space is flat and matching is done by direct comparison of respective attribute-value pairs. Example discovery systems that use this approach are Jini [13, 14], Universal Plug and Play (UPnP) [15], Service Location Protocol (SLP) [16, 17], UDDI [18], CORBA Trader [19], Monitoring and Discovery Service (MDS Globus) [20], Agora [21], Ninja [22, 23], Web Services Peer-to-Peer Discovery Service (WSPDS) [24].

Jini presents a homogeneous view of services. The services register themselves with the lookup service and thus the matching is performed during the lookup phase based on the simple textual attribute comparisons (e.g., type, name). It supports dynamic downloading of service proxies. UPnP matching mechanism uses vendor specific attributes and syntactical details present in the service descriptions. This also uses a homogeneous approach while matching. The SLP uses special kinds of

service requests, however it also matches the service type against available textual attributes. Other related work such as Ninja and WSPDS, do allow more complex matching techniques which go beyond the basic string matching. However, all of them still follow the concepts of annotated attributes and associated values for the matching. The main drawback in each of these systems is that they fail to provide any customization while performing matching operations.

2.2 Ontology-based matching

In this category, ontology or a similar knowledge representation is created for the attributes of the service. In this context, ontology could be used to represent service related taxonomic hierarchies of service classes, their definitions, and relationships. Then, these service attributes can be matched consulting the ontology. This method provides a more complex type of matching technique than simple attribute matching, so that a particular search for query may return other approximate match results. Example discovery systems that use this approach are DReggie [25] and Ontology-based Interoperability Services [26, 27]. DReggie is based on Jini with Semantic Service Discovery and it attempts to take Jini and similar service discovery systems beyond their simple syntax-based service matching techniques by adding semantic matching capabilities to the service description facilities. DReggie uses DARPA Agent Markup Language (DAML) [28] and intelligent reasoning modules to carry out an ontological matching process. Recent developments around DAML, such as the DAML-S [29] and DAML+OIL [30] go beyond simple matching to more customizable matching. Work done on Ontology-based Interoperability Services improves simple matching and presents an approach to semantic-based web service discovery and a prototypical tool based on syntactic and structural schema matching. The matching is based on an input ontology which describes a service request. The requests are matched with the web services descriptions at the syntactic level through Web Services Description Language (WSDL) or, at the semantic level, through service ontologies.

2.3 Hierarchy-based matching

In this approach, services are arranged in a hierarchy based on their types. This hierarchical structure is similar to the DNS hierarchy structure and types are domain dependent (e.g., weather service, stock service, etc). The attribute matching is done by traversing the hierarchy until a leaf node is encountered and matching the attributes of individual services present. Example discovery systems that use this approach are GloServ [31], Concept-Based Discovery of Mobile Services (CBDMS) [32] and OCTOPOS [33]. CBDMS propose a dynamic overlay network by grouping together semantically related services in a hierarchy. Each such group of services is termed a community and communities are organized in a global taxonomy whose nodes are related contextually. The taxonomy can be seen as an expandable distributed semantic index over the system services, which aims at improving service discovery and matching. GloServ is global service discovery architecture in a flexible hierarchical ordering using the Resource Description Framework (RDF) [34]. GloServ querying can either be done manually or automatically using sensor technology which results in a seamless discovery of services. Recent development of GloServ [35] combines with ontology-based matching to make it a customizable hybrid system. OCTOPOS adopts a dynamic hierarchical tree structure and service aggregation for scalability and availability. It also introduces multiple matching mechanisms which contain an attribute and a semantic matching engines which can be categorized as an effort to provide customization on matching at two levels.

2.4 Cloud-based matching

Although there have been many attempts to design discovery services in the context of service-oriented systems, there are only a few efforts that aim to discover cloud-based services. For the sake of brevity, only the efforts from the domain of Cloud Computing (CC) are discussed in this section. The term Cloud Service Discovery System (CSDS) was introduced in [36]. The CSDS helps the users find the

relevant services of interest and the cloud ontology consists of taxonomy of concepts of different cloud services. The CSDS is realized by building an agent-based discovery system that consults ontology to retrieve information (e.g., similarities of attributes of services) about services. The CSDS consists of a search engine and the three agents: Query Processing Agent (QPA), Filtering Agent (FA) and the Cloud Service Reasoning Agent (CSRA). The QPA is responsible for searching the websites using conventional search engines. The FA filters the many results of the QPA using evidence phrases, frequency analysis of these phrases and the nearness (string similarity, for example, using hamming distance) amongst the keywords. The CSRA performs reasoning to find the similarity between services and rating of the services.

The work proposed by Zeng et al. [37] provides an architecture for the cloud services along with algorithms to measure their performance. The main aim of this work is to perform the service selection with adaptive performances and minimum cost. Their service selection algorithm is based on two-steps. The first step is the selection of the available service (basic keyword search) and the second step is the optimized service selection by using maximized gains and minimized cost of selection.

The work proposed by Sheu et al. [38] applies the semantic computing concepts to CC. They describe a Semantic Search Engine (SSE) that provides users' with a friendly problem-driven interface to search services that would be used to build a solution according to users requirements. The architecture of SSE presents a UI for the user to enter his query in natural language. The Interpreter converts this query to Service Query Description Language (SQDL). SQDL is a machine decodable query language used by SSE to describe the intention of the user. This SQDL is matched against the Service Capability Description Language (SCDL) by a Matcher and the right services are selected. If no single service can fulfill the requirement, the matcher will decompose the SQDL query into several simpler queries, and try to find a series of services that may answer the query. Finally, the service invoker finds the right services. The problem with SSE is that it is biased toward semantics matching, which suppresses the other selection criteria of cloud services.

The work proposed by Raichura et al. [39] highlights the benefits of CC and describes the cloud service discovery as being one of the following: a) keyword search, b) provider search, or c) service interface information. The advanced search options in this proposal include searching by service providers, technology platform and other meta-data information. Also, the Web Service Level Agreement Language (WSLA) and the associated framework proposed by Ludwig et al. [40] are capable of addressing the service selection problem, however, within the WS service interface restrictions.

The work proposed by Patel et al. [41] applies the SLA concept into CC using the WSLA framework developed for SLA monitoring and enforcement in a Service Oriented Architecture. SLA@SOI [12] describes the Open Cloud Computing Interface as an emerging standard that can be used to integrate different SLA management layers to control the life-cycle of the Cloud Services. Services can discover and interoperate by using the Open Cloud Computing Interface API and provide hybrid services. This approach does not include the service semantics and QoS information during the service selection. Although a few of these approaches use limited semantic techniques, others use the conventional approach of attribute-based matching. Such a simplistic view is not adequate to identify the most relevant services for complex CC-based applications.

In summary, the main drawback of all of the above systems is that the matching is done based on simple attributes, where the services are represented using string based attribute-value pairs. By implementing MLM inside proURDS, the work proposed in the following chapters tries to address this challenge. Hence, the next two chapters discuss these challenges in detail and present how proURDS addresses them.

3 UNIFRAME OVERVIEW

The proposed work is closely related to UniFrame approach [2,3], hence, this chapter provides an overview of UniFrame. It will set a proper background to present the proposed proURDS system in the next chapter.

3.1 The UniFrame Approach (UA)

Despite the current improvements in software engineering, the development of scalable distributed systems is still a major challenge. Thus, there is a need for a framework that is flexible and cost effective in developing reliable distributed systems. The UniFrame Approach [2, 3] focuses on exploring innovative approaches to represent knowledge of distributed components and proposing a comprehensive framework, which allows a seamless interoperation of heterogeneous distributed components. The UA creates standards as its meta-model (UniFrame Meta Model - UMM) which can indicate the contracts and the constraints of the components. Having this as part of the framework allows the service assemblers or the component integrators to generate a software solution (for a particular DCS) in a fully or semi automatic way. Thus the knowledge of the UMM can consist of entities such as components, guarantees, and infrastructure related information.

Figure 3.1 presents the UniFrame Approach (UA). UA's main aim is to provide means for an automatic or semi-automatic creation of DCS. The UA provides a framework that helps the component developers to create, test and verify components and DCS from the point of view of functional and QoS. The domain experts create the standards for automatic integration of systems using individually developed components. These standards are categorize according to the domains and provide the starting blueprints for systems. For example, these standards include component in-

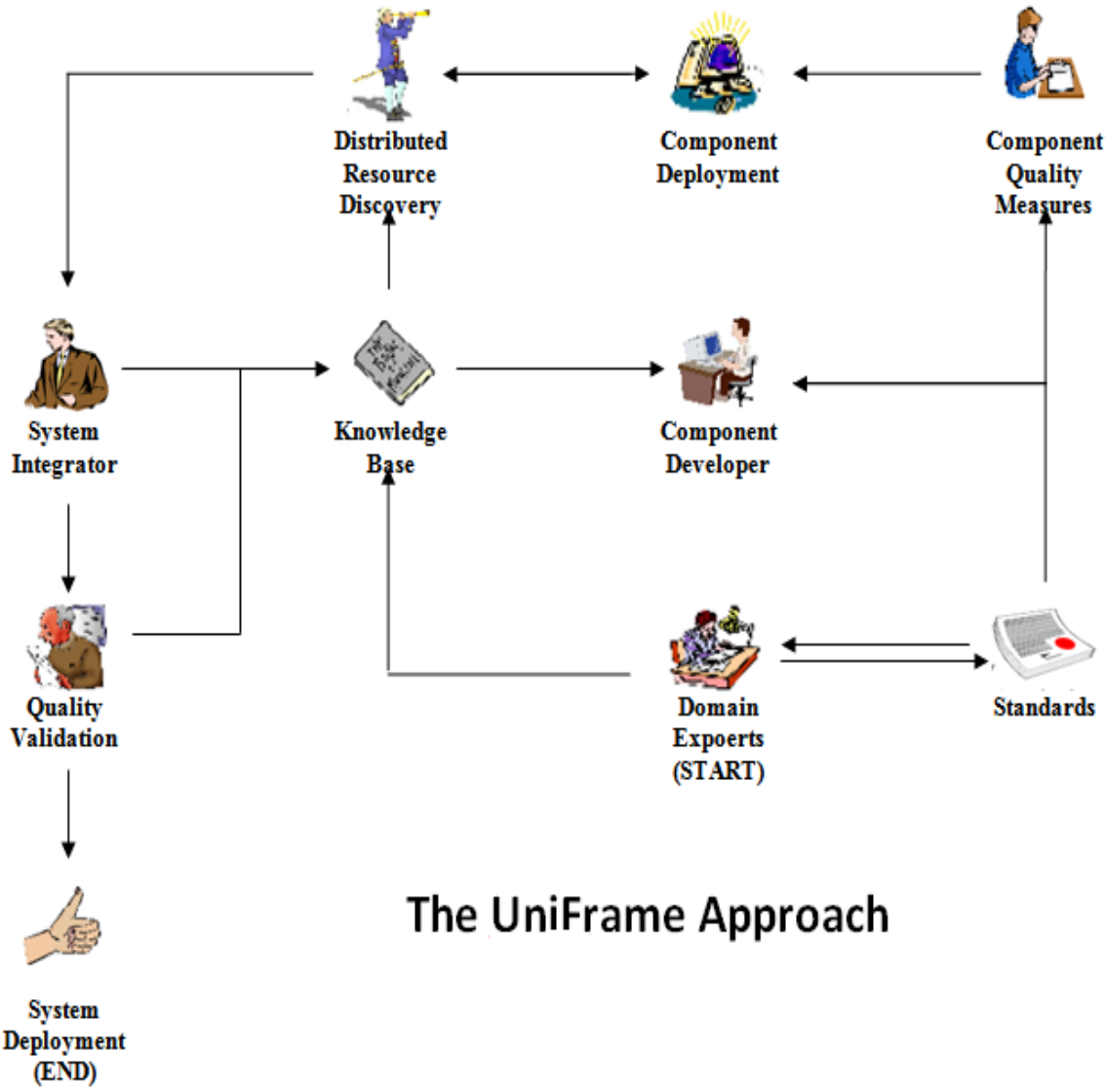


Figure 3.1. UniFrame Approach

terfaces and deployment configurations. These set of standards and expert knowledge are collected into a machine readable format at the Knowledge base (KB). Creating and maintaining this KB is an iterative activity and all the stakeholders of the UA (such as domain experts, component developers, quality measures and integrators) are responsible for updating the KB. Once the standards are in place, the component

developers can browse the standards and KB and decide to start producing individual components of their own. This yields heterogeneous components for the same requirement, which are produced by different developers. After the components pass their quality measures and satisfy the needs of the quality measures then the components are deployed.

After many components become public, the resource discovery service (later the implementation of this service is called as UniFrame Resource Discovery Service (URDS)) starts to aggregate information about available components. The specifications are created to represent each of the components according to their interfaces and other related information. The UA suggests to organize these component specifications into multiple levels (later these specifications are known as Multi-level specifications). The system integrators initiate queries to discover components for their systems. The URDS is responsible to find relevant components and reply back with a list of matching components to the system integrators. During this search the URDS performs Multi-level Matching (MLM), which was defined in [7, 8] and [42]. The MLM produces the result list of matching components for a given input query for the URDS. When all the components are discovered and integrated, the system is validated again as a whole for its quality requirements. The KB is updated with the details of successes and failures and if failed, the UA process starts again iteratively. Finally, if validated, the iterative process of UA ends at the point of the successful deployment of the integrated system.

3.2 UniFrame Resource Discovery Service (URDS)

UniFrame Resource Discovery Service (URDS) [4, 9] is an important part of the UA framework and represents the infrastructural part of the UMM. It provides the functionality of search and selection of software components or services.

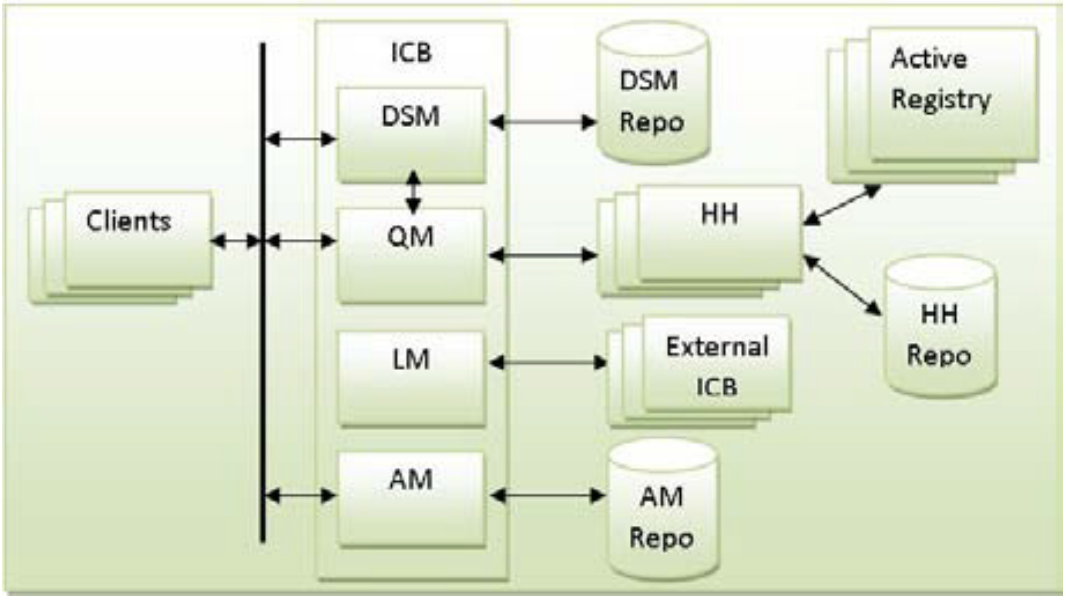


Figure 3.2. URDS Architecture

The architecture of URDS is shown in Figure 3.2. The main components of the URDS are the Internet Component Broker (ICB), Headhunters (HH) and Active Registries (AR). The following subsections describe each component of URDS.

3.2.1 Internet Component Broker (ICB)

The ICB is similar to the Object Broker in CORBA. The ICB handles authentication and authorization, decodes, directs and routes user queries and presents the matching results back to the user. The main four components of the ICB are: Domain Security Manager (DSM), Query Manager (QM), Adapter Manager (AM), and Link Manager (LM). The Domain Security Manager (DSM) is responsible for maintaining the authorization information about all the entities in the system. The Query Manager (QM) is responsible for mapping and routing queries on behalf of the client of the URDS. The Adapter Manager (AM) handles heterogeneity of the system by providing adapter components into the system. The Link Manager's (LM) job is to link different ICBs together. Such a collection of links forms a discovery service fed-

eration, which also includes various mappings of different protocols. Therefore, the ICBs make sure the correct back and forth navigation of queries and the generation of results within the ICB.

3.2.2 Headhunter (HH)

The Headhunter (HH) is the main entity in the URDS. It decodes the propagated query and initiates the discovery process of software specifications and also performs the matching. HHs can be either homogeneous or heterogeneous. A set of homogeneous HHs contain the same matching capabilities and algorithms, while a set of heterogeneous HHs can contain different matching capabilities and matching techniques. HHs can also be either general purpose or serve a special purpose. A general purpose HH accepts specifications from any kind of service, and in contrast, a special purpose HH accepts service specifications belonging to specific types of services or services from a specific domain. Headhunters keep the details of specifications in associated Meta-Repositories. Upon receiving a routed query from the QM, the HHs are actively involved in searching for the most suitable matching components.

3.2.3 Active Registries (AR)

Active Registries (AR) act as the entry points for the new components in the URDS. New components register themselves with respective ARs by presenting their multi-level specifications. Service Exporters register their components and services with ARs by presenting their information in a specification format. New service entry produces a intermediate specification. These specifications are matched against queries generated by the system integrators' needs for components, for their system of interest. This registration process can be active as well as passive. ARs contain heterogeneous details about components, however they can also be rearranged according to specific types and domains. In addition to accepting the registration of services,

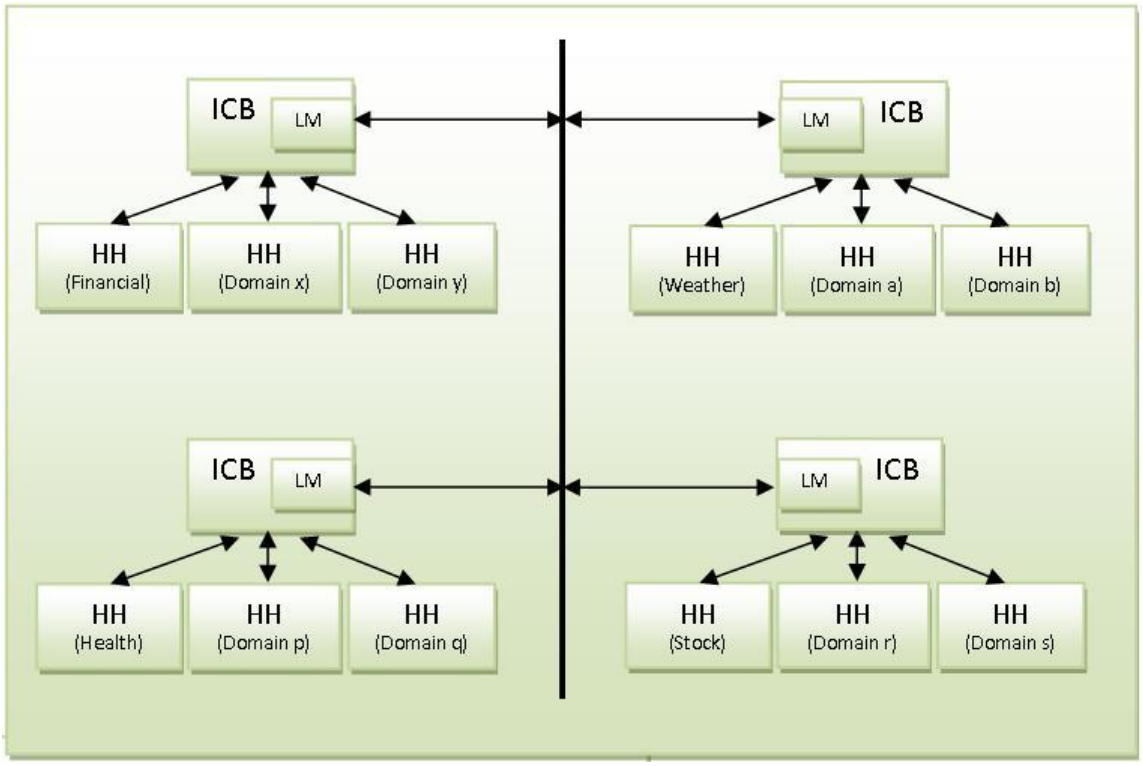


Figure 3.3. Federated ICB hierarchy

ARs communicate with HHs on a routine basis to provide details about the service specifications to the HHs.

The UniFrame Resource Discovery Service (URDS) architecture can be organized as a federated hierarchy in order to achieve scalability. The architecture of federated URDS is shown in Figure 3.3. This shows the hierarchical organization of ICBs. Every ICB has single level hierarchy of zero or more Headhunters attached to it. These ICBs are linked together with unidirectional links to form a directed graph. As mentioned in Section 3.2.1, the LM links different ICBs to form a Discovery Service Federation. Such a federation of multiple URDSes achieves better coverage of a larger service space and thus provides the necessary scalability.

In summary, the URDS is an important entity invoked by the other entities of the UA. The following list is a collection of the drawbacks of the initial URDS prototype had with its operations. As Figure 3.1 indicates, the KB is critical for the UA process

and is being communicated by all other entities. However, the initial prototypes of URDS did not use a KB for its operation. The UA motivates the arrangement of component information into levels inside the specification. Although initial versions had incorporated this using a database, there was no actual service specifications available for the registries. Earlier versions of the URDS was not using all these specification levels at the same time during the matching process and only created simulations using the principles of Multi-level Matching. These experiments showed that the URDS returns more relevant services for a given query compared to the other matching schemes which are based on attributes. Finally, the earlier setup did not deploy entities of URDS over the network as proposed by UA. Therefore, considering scalability of the system, it was not a good approach. However, without the distributed registries and HHs, the management and monitoring did not become an issue. These drawbacks motivated the design and development of the proURDS by incorporating the multi-level matching principles into the URDS architecture.

The next two chapters describe the proURDS within the general domain of service-oriented systems and how it is found to perform better than other approaches, while selecting the relevant services. The proURDS applicability in the context of cloud-based services is described in the subsequent section as a case study from environmental science.

4 PROURDS APPROACH

The proposed proURDS is an extended version of the URDS. Similar to the URDS, the proURDS implements a hierarchical and proactive discovery service. Figure 4.1 presents the architecture of proURDS showing its entities. As seen in Figure 4.1, the Active Registries (ARs) act as the entry point to the services. However, unlike the URDS, they are independent entities distributed over the network. Similar to the URDS, the Headhunters (HHs) in proURDS provide the functionalities of service selection and matching. They proactively collect multi-level specifications of services from different ARs and perform the multi-level matching (described shortly in Subsection 4.3).

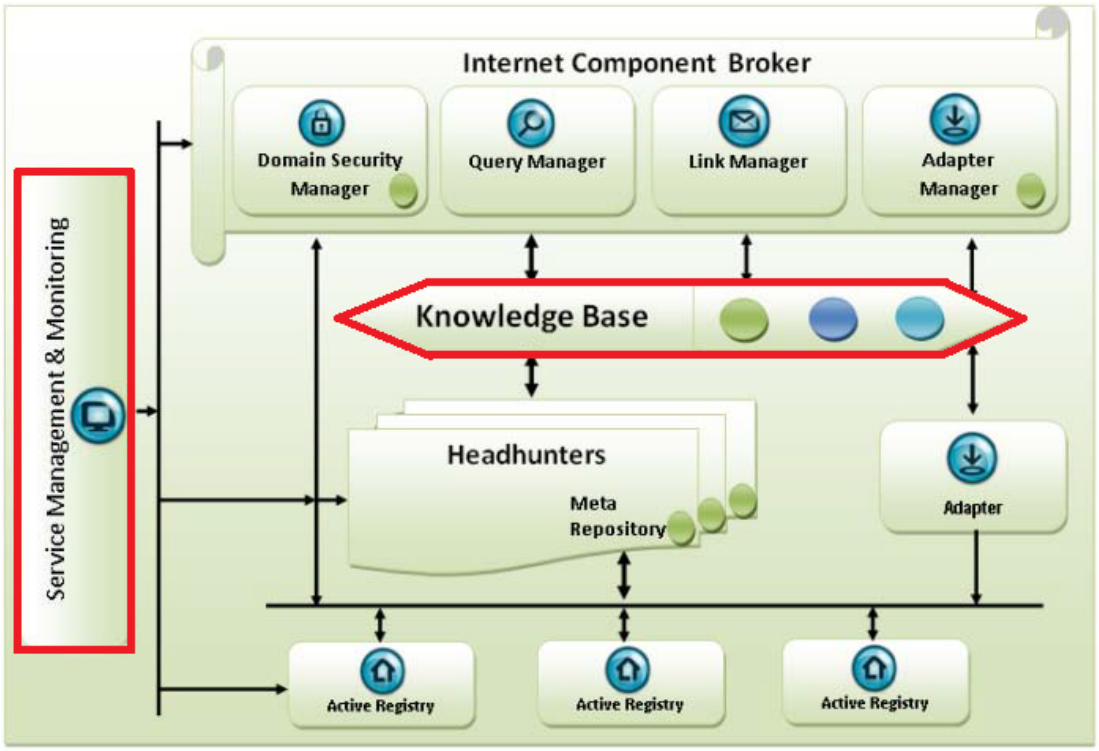


Figure 4.1. proURDS Architecture

The proURDS enhances the URDS by achieving: 1) the incorporation of the necessary contextual knowledge to support multiple matching, and 2) a provision for an effective management and monitoring of the distributed discovery system. Therefore, the proURDS architecture includes two additional modules - the Knowledge base (KB) module and the System Management and Monitoring (SMM) module, which are highlighted in Figure 4.1. The proURDS uses a Knowledge base in its matching operations to improve the process of matching by exacting additional information such as type relations, constraints, and preferences. One other drawback of the URDS is that the experiments of Multi-level Matching (MLM) were not performed in a distributed setup. The proURDS provides the distributed experimental setup wherein the HHs and ARs are distributed over the network. The SMM module is added to provide the management and monitoring of the distributed setup.

Also, other improvements from the URDS to proURDs are that the Multi-lvel Matching features of HHs are enhanced to support different operators with different semantics. The implemented operators are categorized into each level of matching such as at the type level (as described in Section 4.3.2), the proURDS implements type synonyms, type inclusion (i.e., super-type sub-type relations) and type coercion operators. Also, for each matching operator, exact and relaxed types of operational modes are implemented. Finally, the system is deployed in a distributed setup and is experimented with performance and results quality (the experiments and results of the proURDS are presented in Section 5). A discussion of each of these improvements is presented in the following sections of this chapter.

4.1 Knowledge base

The URDS proposed a generalized architecture of the KB which was discussed in details in [43]. This proposed KB design is consistent with the concept of Generative Domain Model [44]. The KB is assumed to be created by the domain experts and contains domain specific information that is updated and maintained periodically.

The KB contains information including the type and configuration to provide solutions for the design of a family of systems. The existing prototypes of the URDS did not incorporate an actual KB.

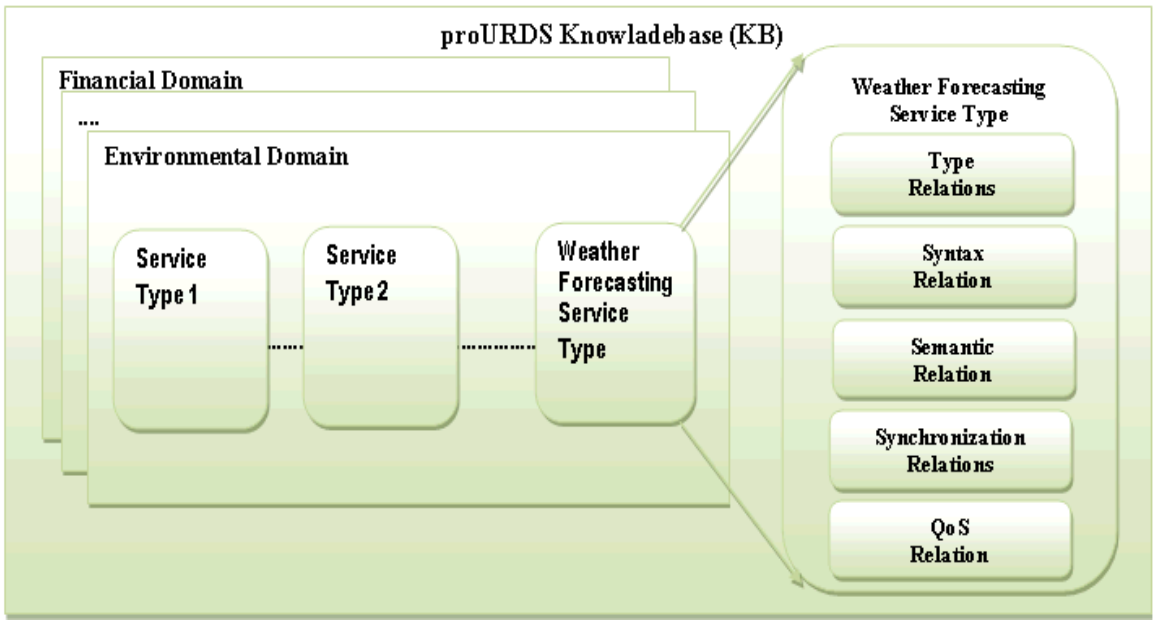


Figure 4.2. Design of the Knowledge base (KB)

In proURDS, the KB contains the necessary information to decode a query and to perform multi-level matching. By using the related information gathered from the KB, the users (i.e., system integrators who are searching for services for their systems) of proURDS construct an XML based query. This query is matched against many instances of its service type using multi-level matching supported by the HHs in the proURDS. Figure 4.2 presents the design and structure of the KB.

The knowledge information is organized according to different service domains such as financial and environmental. Inside each domain, the KB is organized according to valid service categories (i.e., called as service types). Inside each service type, the structure need to match with existing levels of matching. Hence, the KB is also organized into five levels, each corresponding to the level of matching namely:

type, syntax, semantics, synchronization and QoS (described in Section 4.3). For example, for the type levels, the KB contains information about that services' valid types and their synonyms, type hierarchy (if applicable), and information about type compatibility. For the syntax level the KB contains information about the number and order of the arguments, and the return values of the syntactic contract. Similarly, for the semantic level, the KB indicates the key terms and their relations that are used in defining pre-conditions, post-conditions, and invariants for different services. The section in the KB which corresponds to the synchronization level includes information about various synchronization policies. The section in the KB which corresponds to the QoS level includes the appropriate quantification metrics of QoS parameters.

```
<?xml version="1.0" encoding="UTF-8"?>
<proURDSKnowladgeBase domain="Environmental">
   <ServiceType name="WeatherForecasting">
    <TypeRelation>
        <Synonym type="WeatherService">
            <Type>Weather</Type>
            <Type>WeatherForecast</Type>
        </Synonym>
        <inheritance super="MidWestWeatherService" sub="USWeatherService"/>
        <inheritance super="USWeatherService" sub="WeatherService"/>
        <inheritance super="WeatherService" sub="WorldWeatherService"/>
        <coercion super="WeatherServiceWithAllResultsByInteger"
                  sub="WeatherServiceWithAllResultsByFloat" />
    </TypeRelation>
    <SyntaxRelation>
        <Synonym method="GetWeather">
            <Method>GetWeatherData</Method>
        </Synonym>
        <coercion super="int" sub="long" />
        <coercion super="float" sub="double" />
    </SyntaxRelation>
    <SemanticRelation>
        <range super="int" sub="long" />
    </SemanticRelation>
    <SynchoronizationRelation>
        <compatibility super="AnnonymousAccess" sub="AutherizedAccess" />
    </SynchoronizationRelation>
    <QoSRelation>
        <compatibility super="float" sub="double" />
    </QoSRelation>
  </ServiceType>
</proURDSKnowladgeBase>
```

Figure 4.3. Sample of the partial Knowledge base referred by the proURDS matching operators

This KB is internally represented using XML and Figure 4.3 shows a sample partial Knowledge base used by the proURDS. Related to a query, the HHs could refer the KB multiple times while performing the matching process. For example, in Figure 4.3 type relations contain synonyms of the service type of that domain and replaceable service types for a super type using a sub type. The notation super and sub indicates that super type can be replaced by sub type. In syntactic relations, the sample KB contains types which can be coerced from one another. Similarly, other relations contain range and compatibility information such as for this service type anonymous access is compatible with authorized access. Having this KB improved both the querying and matching process.

4.2 Service Management and Monitoring

The Service Management and Monitoring (SMM) module is developed to manage and monitor the distributed setup of the proURDS when it is deployed over the network. It is developed as a Web application using Apache Tomcat servlet container and deployed independently of the other entities of the proURDS. The SMM is the entity with a user interface to control and monitor the system.

Figure 4.4 presents the design of the SMM. The operation of the SMM is based on periodic client server interactions of remote nodes (i.e., physical machines connected over a network) with a monitor node. Periodically the SMM requests information from the nodes about its state and its hosting entities (i.e., HHs and ARs). Hence, using this server, the SMM (which acts as the client) can deploy a given configuration of the proURDS entities (i.e., HHs, ARs etc.) over the network. It can remotely start and terminate proURDS entities and check their availability using frequent heartbeats. There are two options that the proURDS user can take. Either the user can use a configuration file to start entities or alternatively start each entity one by one. This SMM module has other useful capabilities such as the ability to capture a particular

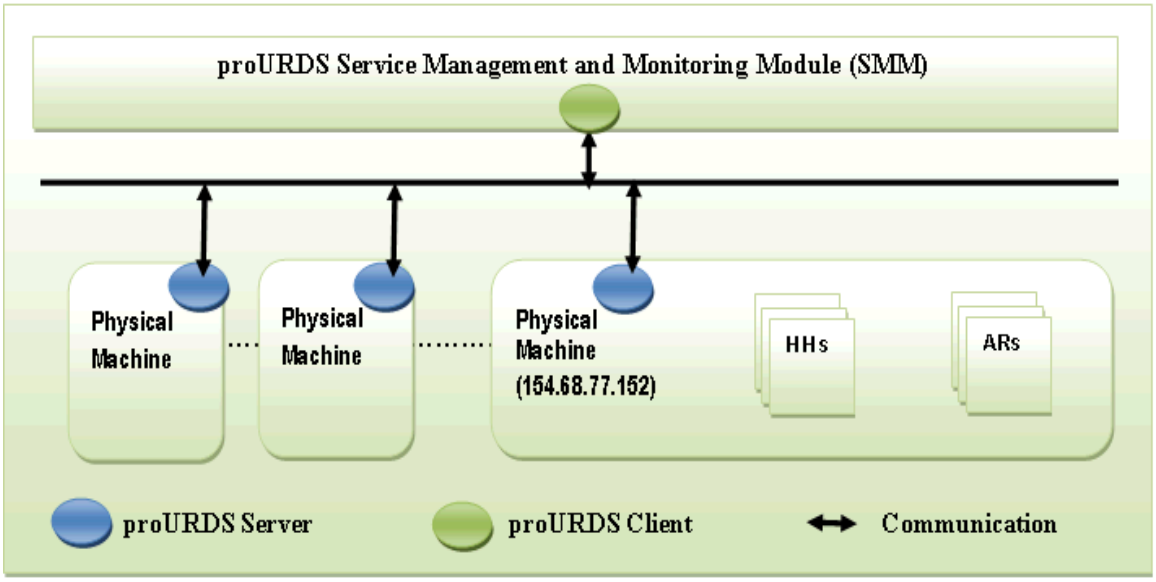


Figure 4.4. Design of the Service Management and Monitoring Module (SMM)

snapshot of the system, to direct and propagate queries to different HHs, and to collect, organize and display the matching service results for different queries.

The advantage of having the SMM module is that it provides the capability of handling a large set of remote entities (such as HHs and ARs) of the proURDS. Also, the other advantage is to monitor the communication happening over the network. The SMM module monitors both unicast communication and multicast communication between HHs and ARs using RMI and Jini Frameworks. The SMM module does not read the content of the communication happen between entities. However it keeps log entries about those communications. It manages the connections to the ARs and HHs internal databases (which keep their own collection of service specifications for fast access) using JDBC APIs, and it also does interactions with the proURDS users (i.e., system integrators) using web based HTTP communication. In the current design of proURDS, the SMM and the Internet Component Broker (ICB) are tightly coupled. The main reason for this was the design choices which are made in favor of rapid implementation of the system.

4.3 Multi-level Matching

In proURDS, a multi-level matching of a service matches different facets of a multi-level specification of a service. The details of Multi-level specifications and matching operators are described in the following two subsections.

4.3.1 Multi-level specification of the proURDS

The proURDS uses the multi-level specifications to represent the services. It is an implementation of the multi-level contracts proposed in the URDS. This specification, in addition to providing a clear separation of multiple facets of a service, helps to perform the operation of multi-level matching. Earlier implemented versions of URDS did not use actual specifications corresponding to existing services.

This multi-level specification contains five different levels. The levels are named: type, syntax, semantics, synchronization and QoS. Hence, each service, in addition to indicating its basic details, may also specify additional details such as the functional details and quality of the service details offered. Initially, the URDS specifications are informally indicated using natural language that includes the computational, co-operative, auxiliary attributes, and QoS metrics of the service. Within proURDS, these specifications are refined into standard XML based specification.

This specification serves two purposes: a) it provides a separation of concerns while designing services, and b) it enables multi-level matching that is more comprehensive than a single dimensional matching based on attributes. An example of a partial multi-level specification (in XML) for a weather service is indicated in Figure 4.5. This partial specification shows four levels: a) Syntactic, b) Semantic, c) Synchronization, and d) QoS. The type level is considered as only the type of the service. In addition, it also indicates other important general features such as deployment and auxiliary attributes.

The functional attributes of a service contain its syntactic interface, along with the necessary pre-conditions and post-conditions, and synchronization schemes employed

```
<?xml version="1.0" encoding="UTF-8"?>
<proURDSContract name="REPO1Pvider1WeatherService" type="WeatherService">
  <ComponentAttributes>
     <property name="DomainName" value="weather"/>
  </ComponentAttributes>
  <ComputationalAttributes>
          <InherentAttributes>
       <property name="license" value="Apache2.0"/>
          </InherentAttributes>
          <FunctionalAttributes>
            <SyntaxAttributes>
               <ContractAttributes>
                     <Contract>
                  <property name="methodName" value="getWeatherData"/>
                  <property name="parameter" value="postalcode" type="string" key="default"/>
                  <property name="returnType" type="string"/>
                     </Contract>
                  </ContractAttributes>
            </SyntaxAttributes>
            <SemanticAttributes>
                     <PreConditon>
                       <property variable="postalcode" type="string" cond="greater" value="0" />
                       <property variable="postalcode" type="string" cond="less"  value="99999"/>
                     </PreConditon>
                     <PostCondition>
                       <property name="ret_weather" value="string" cond="notnull"/>
                     </PostCondition>
            </SemanticAttributes>
            <property name="comlexity" value="O(1)"/>
          </FunctionalAttributes>
  </ComputationalAttributes>
  <QOSAttributes>
          <property name="Reliability" scale="precentage" value="90"/>
          <property name="Security" scale="precentage" value="99"/>
  </QOSAttributes>
  <SynchronizationAttributes>
          <property name="MutualExclusion" value="yes"/>
  </SynchronizationAttributes>
  <DeploymentAttributes>
          <property name="server" value="apache"/>
  </DeploymentAttributes>
  <AuxillaryAttributes>
          <property name="mobility" value="no"/>
  </AuxillaryAttributes>
</proURDSContract>
```

Figure 4.5. Sample partial multi-level specification

(if any). The non-functional (or QoS) attributes represent the QoS parameters supported by the service, along with their values that are guaranteed by its service owner in a specific deployment environment. Services may exhibit special characteristics, such as mobility, security features, and fault-tolerance, which are indicated in their auxiliary attributes. Additionally, the service can include user-defined attributes, for example, the dependencies of the service and its deployment attributes. The entries

in these multi-level specifications have a direct relation to the KB. For example, for the specification entries at the type and the syntax levels, the KB contains information about the structure of types and their synonyms. Therefore, the service provider should refer to the KB while creating the multi-level specifications. If a new service type is created, the service provider updates the KB with possible details at each level.

4.3.2 Matching operators of the proURDS

The challenge with implementing the proURDS matching algorithm is to implement the operators which are needed for each level of matching of a service specification. The proURDS has identified a set of operators to implement at each level of the matching algorithm. The proURS matching operators are implemented to support matching of four out of five different levels (type, syntax, semantics, and QoS). At the type level, the proURDS implements type synonyms, type inclusion (i.e., super-type sub-type relations) and type coercion operators. At the syntactic level, the service specification is matched against three operators, namely, method name, its parameter list and its return parameter. Therefore, in addition to the type operators performed on types of the three syntactic sections, the operators which check for default parameters and order of the parameters are implemented. At the semantics level the proURDS implements an assertion proving mechanism using a theorem prover to check implication, reverse implication and equivalence of assertions. The matching operators of synchronization and Quality of Service levels are implemented to check the compatibility of text list and numeric values (including ranges). Each matching operator has two versions: exact and relaxed. For example, at the type level the relaxed match translates to “is a” relation, i.e., type inheritance.

The technology used while implementing different operators has effects on the operation complexity of the MLM Algorithms. For example, Java Theorem Prover (JTP) [45] is chosen as the main theorem prover to handle the contracts’ seman-

tics matching. Table 5.1 in Section 5.1 displays a summary of these identified and implemented operators at each level of multi-level matching algorithm. HHs in the proURDS can implement any or all of these matching operators, thereby providing the heterogeneity of the matching operations. When performing the multi-level matching for each of the operators, the KB can be invoked to obtain the necessary contextual information. The operator usage of the KB is in relation to the matching level to get appropriate details required for the process (for example, at type level - type hierarchy). New operators can be added at each level by extending the MLM algorithm with corresponding modifications made to the KB. A discussion about the usage of the matching operators and their results is provided in the Section 5 while describing experiments and results.

4.4 The proURDS Implementation

Many efforts of designing discovery systems can be classified according to the usage of semantics matching and ability of customization. Most current efforts do not consider the notion of customization with respect to service matching, because the matching is done based on attributes of a service which were represented using many attribute-value pairs. Based the above argument of categorizing upon the semantics of attribute matching, the current discovery systems can be divided into three main areas: simple attribute-based matching, ontology-based attribute matching and hierarchy-based attribute matching. The design of proURDS could be categorized as a hybrid approach merging related technologies at necessary places.

The proURDS is developed with the Java programming language adhering to Object Oriented (OO) programming systems design and best practices. The technologies involved are Java 1.5, Java RMI, Jini 2.0, MySQL, JTP (A Java based reasoning engine which provides the Theorem Prover [45] capability) and Apache Tomcat 5.0 web and servlet container. The Active Registry (AR) is developed by wrapping Jini Lookup Service [13, 14] which is customized for the proURDS needs. Its plug and

play method of multicasting feature was used for communicating updates of services to Headhunters. Entities such as the Domain Security Manager (DSM), Active Registries (AR) and Headhunters (HH) are Java Remote Method Invocation (RMI) standalone entities. Java RMI is used by HHs in back and forth communication with the service query users and Service Monitoring and Management (SMM) unit. The Management and Monitoring (SMM) system is developed as a web application deployed on servlet container. All the communication and data representation are done using XML based technologies by serializing over the network. MySQL database technologies are used for all the Databases present in the system, in particular SMM's database. JTP is used for semantic level matching as a part of multi-level matching algorithm present in the HHs matching algorithm. The technologies used to create the proURDS and how the entities communicate are shown in Figure 4.6.

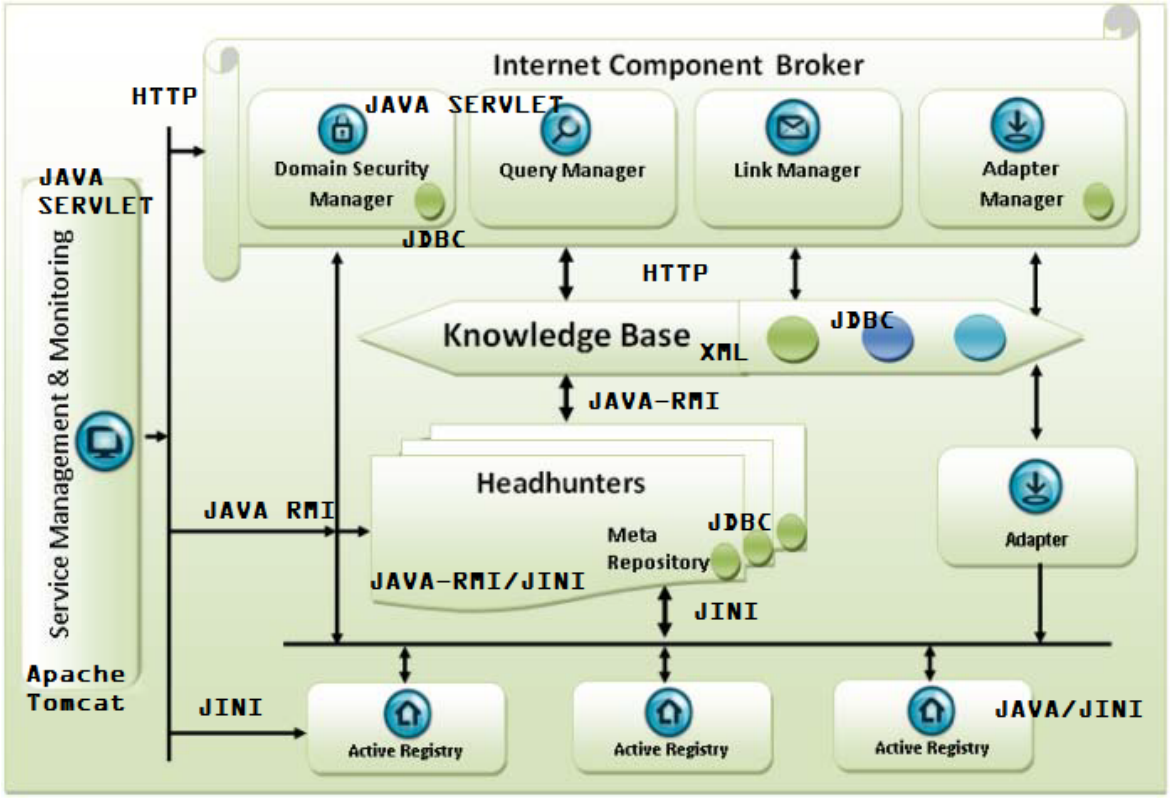


Figure 4.6. Communication protocols used in different messages of the proURDS

The main challenge with the implementation of the proURDS was to integrate the independently developed entities of proURDS (i.e., ARs and HHs) with Multi-level Matching (MLM). Design choices were made to develop MLM matching algorithms as plugins, so that they can independently perform selected operations at each matching level. This gives the flexibility to extend the MLM algorithm for future needs of different matching techniques. The new DS was developed to address the need of reducing the complexity in integrating, controlling and monitoring of all the components of the enhanced discovery service.

The integration phase should consider the effective ways of communicating and getting the information to flow. For example, the MLM operators and HH are simultaneously invoking KB for additional information. Thus it is important to find a balance in communicating with the KB, such that the discovery process works without delays. Each partition of the KB accessible to entities without waiting for others and this improves the proURDS performance. However, the updates to the KB had to be done in a non-blocking fashion. It is known that updates to the KB mainly happen offline, hence this was not considered as an issue. The SMM follow similar guidelines to collect, display and control proURDS entity related information. For example, configuration updates such as starting and terminating entities like HHs are handled after all the queries served by that entity. Many challenges with proURDS and MLM integration are handled by following the best practices such as following design patterns and implementing mutual exclusions.

Although some of the components in the current design of proURDS are tightly coupled, the attempt was made to design SMM to move closer to real world independent entity interactions. For example, the SMM supports the construction and destruction of a proURDS instance using user provided configurations. It also acts as the container for central system control and monitoring of the proURDS. A system developer can specify a configuration by indicating the number of entries (HHs and ARs) and capabilities of each entity. For example, a user can indicate the access privileges of a HH, and degrees of matching (e.g., how many levels) that the HH

provides. The MMM also has other useful features such as providing a snapshot of the proURDS and recording the history of its execution. The MMM can also display the query execution results graphically.

4.5 The proURDS validation with the URDS

The initial proposal of the URDS indicates many goals during a design of a discovery service. The main goals were to handle service heterogeneity, different communication patterns, complexity of the distributed setup, failures of entities, and experiments with a data set. The following discussion provides details about validation of the proURDS according to the Uniframe proposal goals of a the discovery service and how the proURDS was designed and developed to address them.

The proURDS handles the service heterogeneity at the AR level by abstracting them into multi-level service specifications. The heterogeneous services could register their specifications with any registry, which service can access. At the HH level it is done by abstracting different capabilities to each HH. For example, different HHs provide the same interface for different clients who are searching for the suitable service which matches with their requirements.

Handling the challenge related to different communication patterns in the system is achieved by using different methods of communication (e.g., HTTP, Java-RMI, Jini Multicast) for different layers (such as, HHs and ARs) of the proURDS architecture (Figure 4.6). Unicast communication is used for the communication between client (here the SMM) and different HHs. This communication is based on Java-RMI. To provide a seamless integration within discovery service, multicast communication is used between HHs and ARs. This communication is achieved through Multicast Sockets based on UDP/IP provided by the Jini Framework. The connections to the HH's internal databases AR database and the SMM database are established using JDBC APIs at the levels of both HHs and ARs. Interactions between the clients

(users) and the Service Management and Monitoring (SMM) component are based on HTTP protocol.

The complexity of the proURDS setup is addressed by giving the SMM the ability to deploy different configurations over the network. Having a Java-RMI server node is the only requirement for a node (in this case a physical machine) to be a part of the proURDS deployed configuration. The SMM acts as Java-RMI clients to these server nodes, which are recognized as one per node. After that, all nodes are checked for activeness by a configurable heartbeat which is a randomly communicated signal which can be initiated either by the SMM or the server nodes. If this signal was designed to be a synchronous activity, then the network would have been flooded. Therefore, the ability of each entity to configure and initiate communication at its own time and speed asynchronously makes the proURDS more scalable. The Service Monitoring and Management (SMM) system was designed to handle this complexity, as the deployed proURDS could handle any number of server nodes. In addition the resource consumption of each entity (such as HHs and ARs) is within the reachable limits (around 1 megabyte of physical memory). For example, with a node with 1GB physical memory it was tested that more than 500 entities could be started and communicated as active. This is well over the required limit, since distributed systems presume the entities are distributed. Communication delays are noticed when the overall entity (i.e., HHs and ARs) limit exceeds around 215 with the total proURDS. This could be due to the synchronized functional methods which are present in HHs, when HHs communicate with each other.

The proURDS is designed to handle failures through periodic announcements such as heartbeat probes and information caching (at the levels of ARs and HHs). Lack of communication from the entities of proURDS (i.e., HHs and ARs) beyond a threshold time (which can be set at the SMM) is considered as a failure of that entity, and the state of the system is accordingly reset. The caches of the Headhunter and Link Manager are updated based on the responses received from Active Registries and Link Managers in other ICBs, respectively, or purged based on their availability.

However, one of the issues related to proURDS validation is the availability of a good dataset with valid services. To address multi-level matching at each level of the specification, the proURDS proposes the operators (discussed in Subsection 4.3.2) which themselves should be flexible enough to handle different datasets. The main dataset of services chosen was the Quality of Web Services (QWS) Dataset [10] from the University of Guelph, Canada. This dataset has collected over 5,000 web services and performed various measurements on Quality of Service (QoS) of each individual service. The detailed discussion of the dataset is included in the experiments Section 5.

The next section on experimentation (Section 5) describes the experimentation setup, the results and analysis.

5 EXPERIMENTATION, RESULTS AND ANALYSIS

5.1 Experimentation

This chapter describes the various experiments carried out using the proURDS prototype to assess the benefits of multi-level matching and associated tradeoffs between the performance and quality of the results. Two important parts of these experiments are the dataset and the experimental setup of the proURDS. The next two subsections describe these two parts in details.

5.1.1 The proURDS dataset

The main dataset of services chosen for the empirical validation is the Quality of Web Services (QWS) Dataset [10] from the University of Guelph, Canada. This dataset contains 5,000 web services with their Quality of Service (QoS) parameters. The services in the QWS Dataset were collected using the Web Service Crawler Engine (WSCE) from public sources on the web including Universal Description, Discovery, and Integration (UDDI) registries, search engines, and service portals. The measurements of each service in this dataset consist of nine entries with QoS attributes and other general service details, such as Response Time, Availability, Throughput, Successability, Reliability, Compliance Best Practices, Latency, Documentation, WsRF, Service Classification, Service Name, and WSDL Address. This dataset is used during the proURDS experimentation with modifications. Since the time from the dataset published, some of the services have relocated to different web addresses. Therefore, after verifying the WSDL cached on web search engines at the old location with the new location, the dataset was updated with the new locations of the services.

The other update was done related to the "Service Classification" parameter of the dataset. The experiments replaced this parameter with the service type.

As the services in the QWS dataset did not contain multi-level specifications, the first step is to create such multi-level specifications for a subset of services from the QWS dataset. The synchronization level is not used in the proURDS matching process as the synchronization contracts for these services could not be created due to the unavailability of their source code. Many instances of these services are created and deployed, along with their specifications, in the experiments that are carried out with the proURDS. These services are distributed randomly into the active registries.

5.1.2 The proURDS experimental setup and operation

The experimental setup had ten Dell PCs connected through the Local Area Network (LAN) and running windows XP. The impact of network topology and geographical separation to the service discovery time is not considered to be a part of the current set of experiments and is a part of the future work. The experiments with the proURDS are initiated by starting the proURDS remote server endpoints of each node (i.e., physical machine) and starting the SMM on any of the nodes. Later the entities (i.e., the HHs the ARs) can be started according to a selected configuration using the SMM. On startup, the proURDS Active Registries (which are extended native registries developed by wrapping Jini Lookup Service) refresh themselves with the currently available services list and then obtain a multicast group address from the DSM and listen for multicast messages from the Headhunters on these multicast groups. Once deployed in the proURDS environment, the Headhunters periodically communicate to their multicast group. These multicast groups are listed by both the Headhunters and Active Registries. They are actively involved in locating accessible ARs according to the policies obtained from the DSM. When Active Registries receive a multicast message from a particular Headhunter with its location they respond to the message by unicasting their location information to that Headhunter.

The Headhunters maintain a cache of pairs (registry address, Last updated time-stamp) to validate the liveness of the meta-data of the services that they stores in their meta-repositories. The Headhunter uses these registry locations to query the list of accessible Active Registries to get the meta data of the interested services. During the registration, the Headhunter stores all the details of the services of interest into the its meta-repository, including the multi level specifications. This stored information is used during the multi level matching process by the Headhunters where it tries to find services that satisfy the computational, cooperation, auxiliary attributes and QoS metrics specified in the search query. A particular service may be registered with multiple Headhunters when the system progresses according to the DSM policies. The services are identified by their service offers comprising of service type name, the proURDS specification which includes all the multi-level details of the service for example, zero or more syntactic contracts and QoS values for that service. The specification is stored as an XML file at the AR level and the details are stored in a private database of each HH. As mentioned in previous Section 4.3.1, these specifications are the XML based Multi Level Specifications (as presented in Figure 4.5) and the information is ordered in multi-level

Many instances of the services (from the QWS Dataset) are deployed, along with their specifications, in the experiments that are carried out with the proURDS. These services are distributed randomly into the active registries and queries are manually written and validated with the knowledge of existing services. At each AR, the specifications are ordered for specific domains such as Financial Services, Health Care Services, and Stock Services. This defines the AR's specialty of the domains of services. When the system integrator identifies the needed components for its DCS construction, queries for each service are passed to the Query Manager(QM) in the SMM which in turn produce a multi-level query to select a subset of accessible HHs in the proURDS. An example of a partial multi-level query is indicated in Figure 5.1. The queries used in these experiments are a subset of service specifications and are expressed in XML.

The proURDS produced query (which is shown in Figure 5.1) describes different facets of the desired service together with the query configuration. The query configuration part contains matching related settings, such as which version (from exact or relaxed) and semantics of the associated operators at each level to use during the matching process. The "query config" element contains the configuration for matching. Inside this element of the query, the details of the semantics of the associated operators are provided. For example, this sample query contains type level relaxed in operators for type synonyms, inheritance and coercion. The elements following the "query config" contain multi-level query information expressed as service attributes at each level such as type (component), syntax, sementics, and QoS. For example, "QoS Attributes" element contains the query information related to the reliability and the response time of the service that this query is searching for.

When a HH receives the Multi Level Query, it queries the KB to obtain necessary domain information which is essential in decoding the query and also in performing the MLM. For example, the Semantic level matching needs the service of a TheoremProver to perform matching of assertions such as the equivalence and the implication. The proURDS invokes the Java Theorem Prover [45] which is an object-oriented modular reasoning system based on a simple and general reasoning architecture. When the MLM completes the matching at a level the results are injected to the next level for further processing. The list of results is routed back to the end user via the HHs and QM of the proURDS.

Multiple experiments are carried out to test different levels of multi-level matching, query evaluation and performance evaluations of the proURDS prototype. The following section discusses the details of the results and their analysis.

5.2 Results and Analysis

The results which are described in this section are related to a particular multi-level query which is submitted to the proURDS. In each experiment, a query is

```
<MLQueryConfig>
   <MLSQuery id="32">
      <QueryConfig  level="all">
         <TypeRelaxed synonyms="true" inheritance="true" coercion="true">true</TypeRelaxed>
         <SyntaxRelaxed synonyms="true" inheritance="true" coercion="false"
                        default="false" order="false">true</SyntaxRelaxed>
         <SemanticsRelaxed implication="true" revImpil="false" eq="false">true</SemanticsRelaxed>
         <QoSRelaxed compatibility="true">true<\ QoSRelaxed >
      </QueryConfig>
      <ComponentType>WeatherService</ComponentType>
      <SyntaxAttributes>
         <ContractAttributes>
                  <Contract>
                           <property name="name" value="GetWeather"/>
                           <property name="parameter" value="postalcode" type="string" key="default"/>
                           <property name="return"   type="XML"/>
                  </Contract>
         </ContractAttributes>
      </SyntaxAttributes>
      <SemanticAttributes>
         <PreConditon>
                  <property variable="postalcode" type="string" cond="greater" value="46202" />
                  <property variable="postalcode" type="string" cond="less" value="46254"/>
         </PreConditon>
         <PostCondition>
                  <property name="return" cond="notnull"/>
         </PostCondition>
      </SemanticAttributes>
      <QoSAttributes>
                  <property name="reliability" cond="percentage" value="90"/>
                  <property name="responce" cond="ms" value="200"/>
      </QoSAttributes>
   </MLSQuery>
</MLQueryConfig>
```

Figure 5.1. Sample proURDS multi-level query

displayed with the results and special attention is needed to differentiate the different facets of the query from the query configuration.

5.2.1 UDDI vs proURDS Evaluation

The first experiment compared the proURDS with a publicly available prototype of UDDI, the jUDDI [46]. jUDDI supports simple attribute-level matching, which is a subset of the matching supported by the proURDS. Figure 5.2 shows the outcome of this experiment. The $y-axis$ of Figure 5.2 represents the average response time of

the discovery service (i.e., in this case both jUDDI and proURDS) and the $x - axis$ denotes different numbered Multi-level queries. In this experiment, randomly generated queries are sent to the jUDDI and two versions of the proURDS. One version of proURDS supports only the exact type matching and the other version supports relaxed matching at four levels (i.e., Type, Sysntax, Semantics and QoS). Because synchronization details are not present in these services, they are not included. The response time (T_q) for each query is measured by repeating the same query one hundred times and taking the average of the response times obtained in each of these iterations. As seen from Figure 5.2, the jUDDI and the proURDS containing only the type matching require comparable times to service these queries and both of these systems yielded identical services for each of these queries. The MLM for different queries, as shown in Figure 5.2 resulted in a higher response time, as expected. This increase is due to the cost of implementing the additional matching operations.

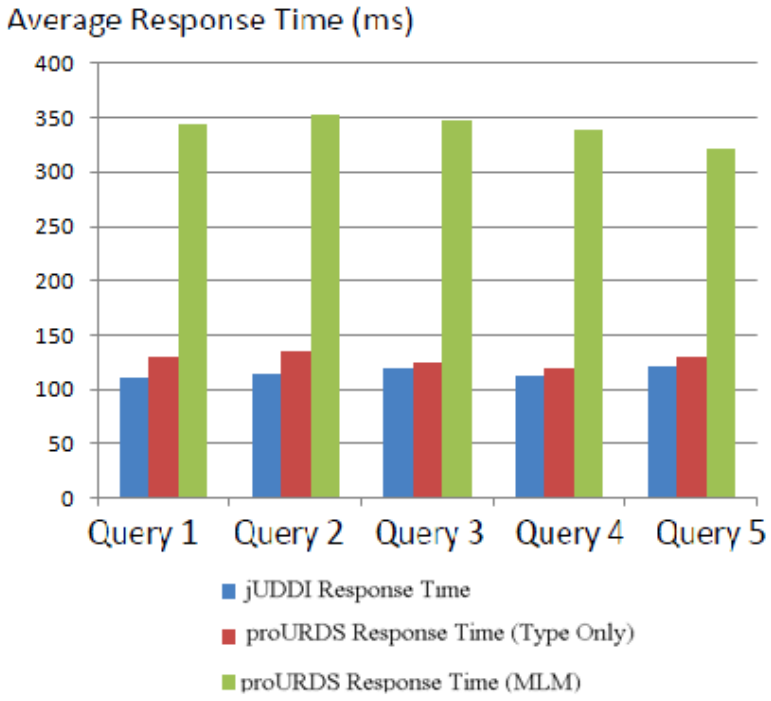


Figure 5.2. Response Time Comparisons

5.2.2 Quality Evaluation

The next set of experiments compared the quality of the results returned by the jUDDI and proURDS prototypes. These results (i.e., number of services returned after the matching process) were manually inspected for their quality (i.e., their relevance for a particular query).

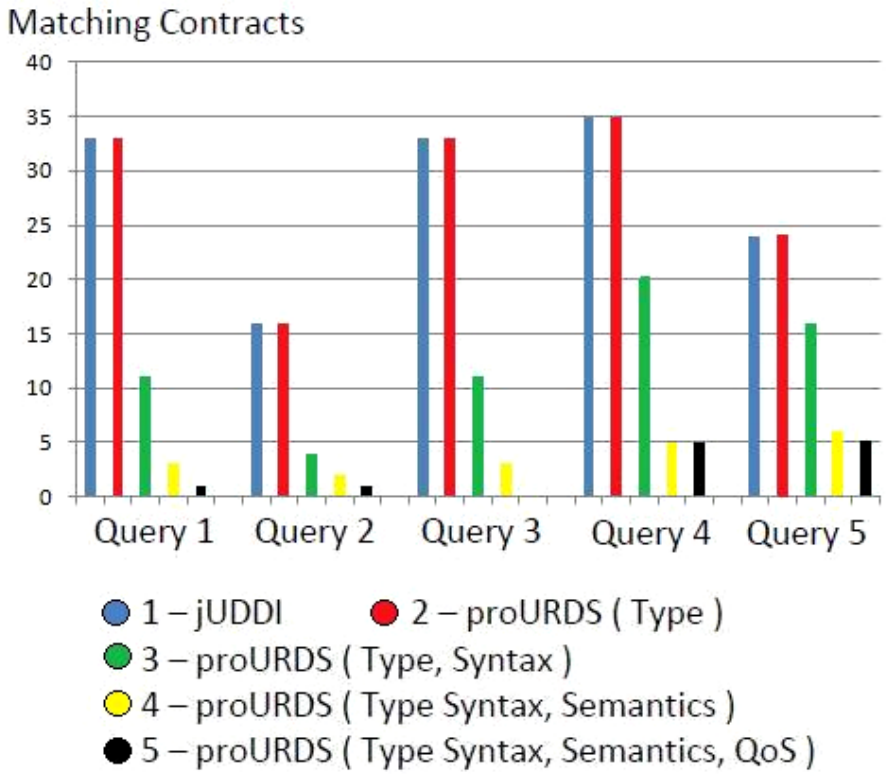


Figure 5.3. Comparison of the Quality of Result (Exact Matching)

Figure 5.3 shows the outcome of this experiment. The $y - axis$ of Figure 5.3 represents the number of matching service contracts resulting from the discovery service (i.e., in this case both jUDDI and proURDS) and the $x - axis$ denotes different numbered Multi-level queries where exact matching is enabled. As the jUDDI supports only the type level matching, it returned the same number of relevant services as those returned by the proURDS with type matching semantics. The number of services that returned with all the four levels of matching, for a particular service, is

typically much smaller than those returned at the end of only type matching as seen from Figure 5.3. However, these services are more relevant to the query than the ones obtained at the end of only the first level of matching. Hence, the quality of results (i.e., degree of relevance) increases as the levels of matching increases at the cost of higher response time (shown in Figure 5.3).

Table 5.1
MLM Levels and Operators

Level	**Operator**
Type	Synonym (Exact)
	Inheritance (Relaxed)
	Coercion (Relaxed)
Syntax	Synonym (Exact)
	Inheritance (Relaxed)
	Coercion (Relaxed)
	Default Parameters (Relaxed)
	Parameter Order (Relaxed)
Semantics	Equivalence (Exact)
	Implication (Relaxed)
	Reverse Implication (Relaxed)
Synchronization	Compatibility
QoS	Comparability

In the second part of the query evaluation experiment, the precision and recall of the results, returned by the proURDS, were computed. The precision is defined as the number of relevant services retrieved by a query divided by the total number of services retrieved by that query, and recall is defined as the number of relevant services retrieved by a query divided by the total number of existing relevant services (which should have been retrieved). In these experiments, exact matching operators,

listed in Table 5.1, were used. Since the response time is not a consideration in these experiments, only the results obtained by the use of exact matching were considered. These results are listed in Table 5.2.

Table 5.2
Exact Matching Results

Query No	**Total No of Relevant Services**	**No of Returned Services**	**Resulted No of Relevant Services**	**Precision %**	**Recall %**
Query 1	2	1	1	100	50
Query 2	1	1	1	100	100
Query 3	1	0	0	0	0
Query 4	7	5	4	80	57
Query 5	6	5	5	100	83

It can be seen from Table 5.2, the higher the precision (optimal 100%), the higher the quality of the results returned. Also, the higher the recall (Optimal 100%), the better the quality of the results. According to Table 5.2, only query 2 is able to achieve optimal results, but all the other queries yielded acceptable results approaching the optimal, except query 3. It failed to produce any results - the reason as seen from Figure 5.3, is that at the QoS level no service contracts were able to fulfill the requirement of the query.

The same experiment was repeated with relaxed matching semantics for all the operators. Figure 5.4 shows the outcome of this experiment. The $y-axis$ of Figure 5.4 represents the number of matching service contracts resulting from the discovery service (i.e., in this case both jUDDI and proURDS) and the $x-axis$ denotes different numbered Multi-level queries where relaxed matching is enabled. Figure 5.4 shows an increase in the results returned at each level when compared with the Figure 5.3.

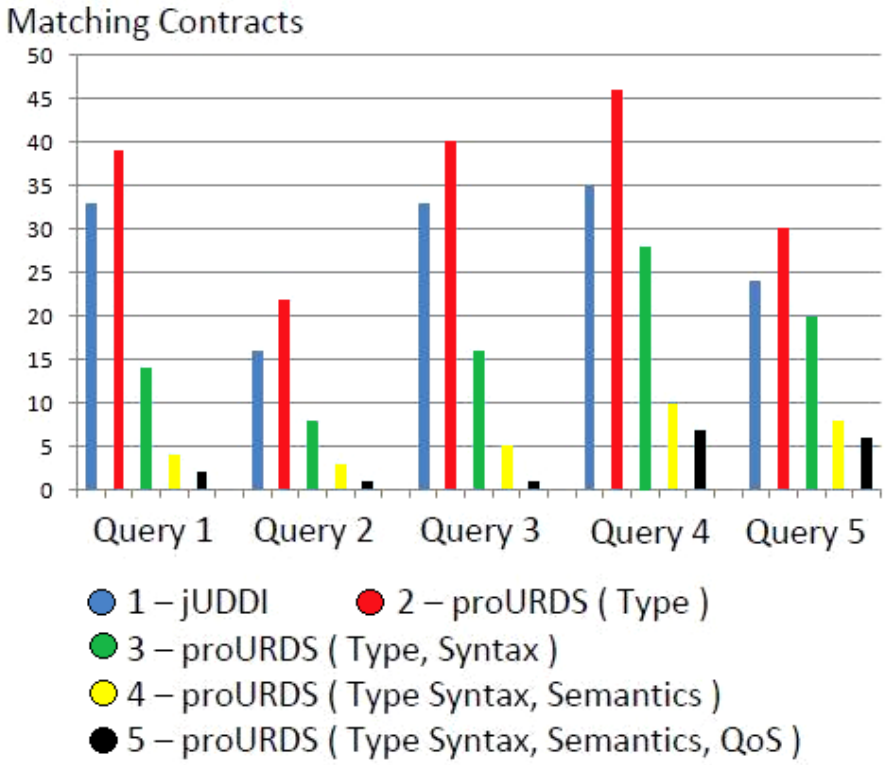


Figure 5.4. Comparison of the Quality of Result (Relaxed Matching)

This is as expected, as relaxed matching, due to its inherent nature, will return more services. The precision and recall evaluation for the experiment are indicated in Table 5.3.

When the results in Table 5.3 are compared with the results in Table 5.2, most of the queries were able to reach the optimal results. This is, again, as expected due to the relaxed nature of the operators.

5.2.3 Performance Evaluation

Additional experiments were conducted to test the performance of the proURDS prototype. In addition to T_q, the Matching Time (T_m) is used as a metric in these experiments. T_m is defined as the time taken by a HH to perform the MLM depending

Table 5.3
Relaxed Matching Results

Query No	**Total No of Relevant Services**	**No of Returned Services**	**Resulted No of Relevant Services**	**Precision %**	**Recall %**
Query 1	2	2	2	100	100
Query 2	1	1	1	100	100
Query 3	1	1	1	100	100
Query 4	7	7	6	85	85
Query 5	6	6	6	100	100

on its capabilities. If a HH performs matching at all the five levels then T_m is the sum of matching times observed at each level. T_q is summation of T_m and the time required for propagating a query to a particular HH and bringing the results back from that HH.

Figure 5.5 shows the matching times required at each level for five random queries. The $y - axis$ of Figure 5.5 represents T_q as the response time taken by the proURDS to perform matching and the $x - axis$ denotes different numbered level of matching (from level 1-4) as Type, Syntax Semantics and Qos. As expected, each level of matching increases the response time. However, the increase in the time required for the semantic matching is substantially more than the other levels, as it involves the use of a theorem prover to establish the equivalence relation between the query and the set of available services.

Figure 5.6 shows the increase in T_q as a function of number of services. Again, as expected, with the increase in service space, the response time increases proportionately for a set of queries.

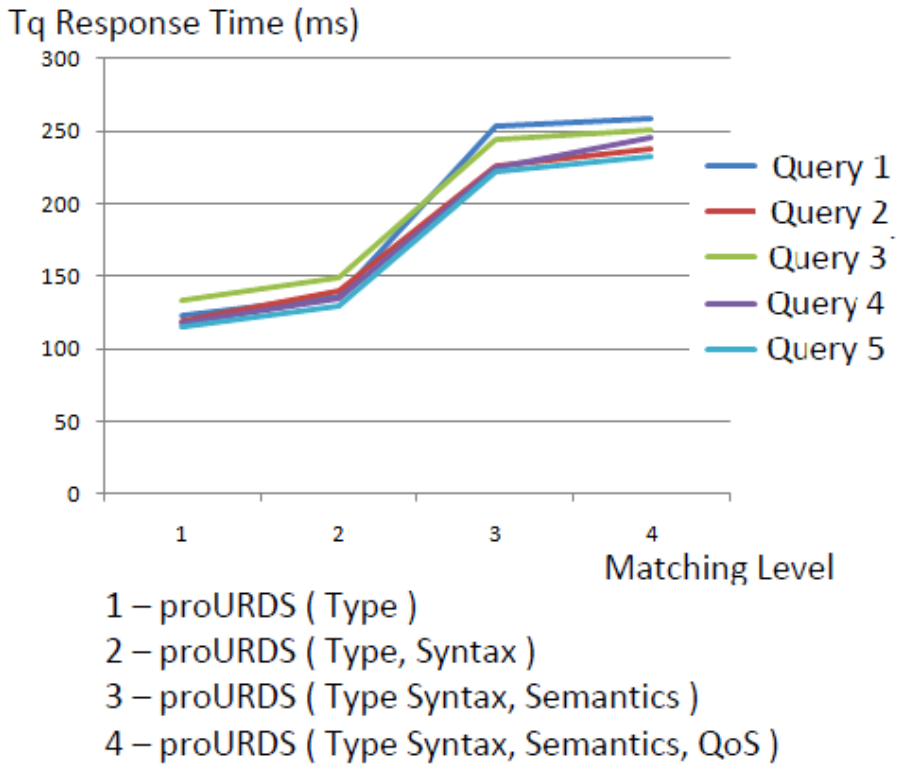


Figure 5.5. Individual Matching Times

5.2.4 Matching with Timing Constraints

As seen from Figures 5.2 and 5.4, there is a tradeoff associated with the response time and the quality of the results returned for a particular query - typically, the higher the quality (i.e., number of relevant services returned), the higher the response time, and matching at more levels is needed to achieve the high quality. Hence, the final experiment was carried out to study this tradeoff. In this case, an upper limit, arbitrarily chosen, for the T_q was set and the services returned were inspected for their quality when this limit expired. Figure 5.7 shows the outcome of this experiment.

The first bar indicates the results returned when the upper limit was reached, while the second bar indicates the results when there was no upper limit. For all five queries, the quality of the results when the limit was reached was lower than the case with no limit, again, as expected. The degree of loss in the quality will depend upon many factors such as the value of the upper limit, the nature of the query, and

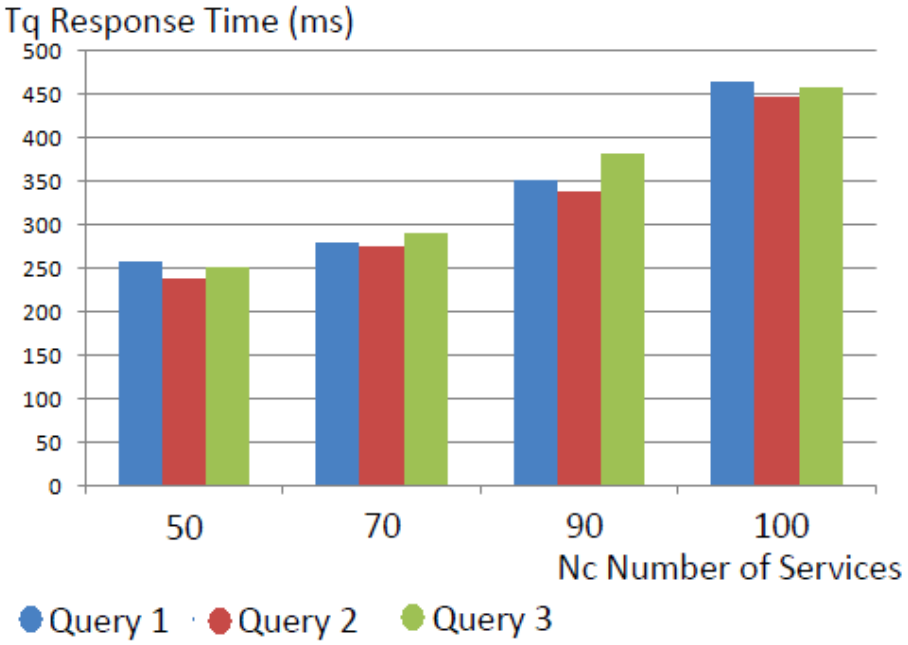


Figure 5.6. T_q as a Function of Size of Service Space

the number of services matching the query. Hence, if an application is time sensitive, then it can probably accept not the most relevant service but a "close-enough" service. On the other hand, if the service selection is an off-line process (i.e., carried out as a separate phase from the service compositional phase) then a higher response time can be tolerated to obtain the most relevant service(s) for a particular query.

In summary, the results described in the above set of experiments indicate that the proURDS is able to find relevant services with better quality. The quality improvements of the services returned by the proURDS is verified in terms of precision and recall. This improvement is possible as a result of the multi-level matching semantics the proURDS. However, the increased result quality is achieved at the cost of increased response time. When compared to the other alternatives such as UDDI, the average response time of the proURDS is high, however this was expected due to the additional work performed by the multi-level matching. Hence, a need arises to test the proURDS with real world requirements, and therefore the next section

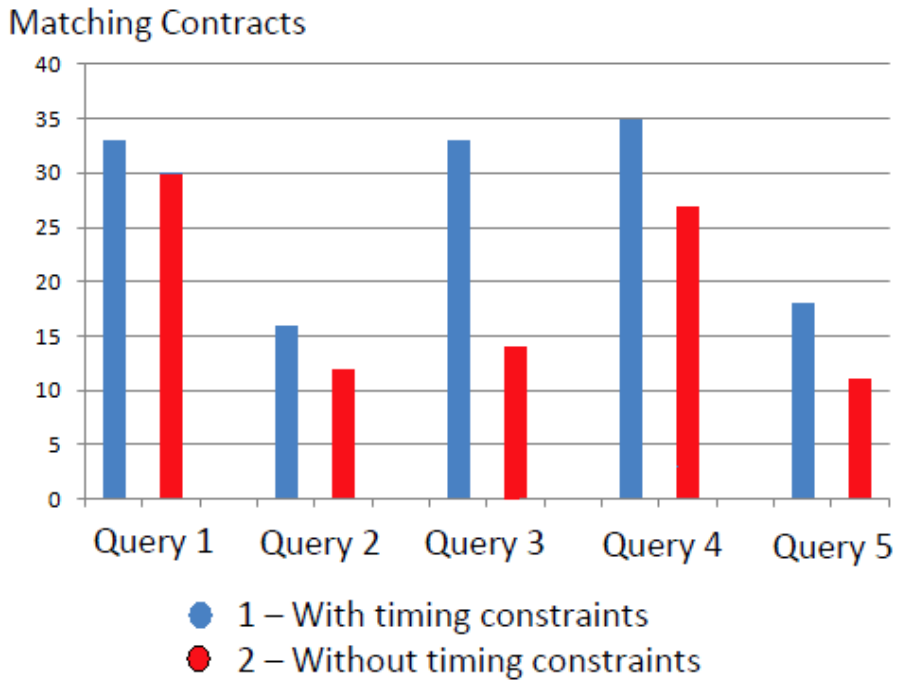


Figure 5.7. Matching with Time Constraints

provides a case study to explain how proURDS can be used with the services from environment sciences domain.

5.3 Case Study : Cloud Service Selection

As a consequence of the results section (i.e., Section 5.2), the need appears to test the proURDS with a real world service requirement. Therefore, the proURDS is tested with the service requirements for environmental science domain as a case study in the following sections. This case study describes the background and the application of the proURDS to the domain of cloud-based services from the Environment Sciences domain. Also, it presents the experiments and results of the proURDS behavior in this context.

5.3.1 Cloud Service Selection

Cloud Computing (CC) promises to deliver computing as a utility. Users can request on-demand access to software services hosted inside clouds. For a given application, many similar services, that are developed independently, could be hosted in a cloud. Hence, automatically selecting an appropriate service from these available choices to fulfill a particular requirement is a challenge. CC provides flexible ways for hosting, consuming, and delivering Internet-based services. Mainly due to the reasons of economy, ease of creation and use, flexibility, and scalability, software realizations of CC-based applications would be achieved as coalitions of independently created services that are deployed in clouds, public and/or private. Selecting appropriate cloud-based services is a critical step in composing CC-based applications. Consider a typical environmental monitoring system which can be created as an ensemble of many independently developed services. For example, such a system can be used to monitor the effects of a contaminant spill in a large body of water. To create this system, scientists from the Environmental Sciences domain will need to integrate data-set monitoring services with different environmental simulation (e.g., watershed models, climate models, and ecological models) services. When a team of Earth Scientists searches for Precipitation, Land Cover, Water Flow, Water Quality, and Weather Forecast services, multiple instances for each type of these services (e.g., deployed by USGS [47], USDA [48], NASA [49], and NOAA [50]) may be available that the team can choose from. These instances might be hosted in public or private clouds (as shown in Figure 5.8) along with the necessary datasets. This selection function could be made available as a feature of a cloud-based middleware. For example, in the Environmental Science domain, there is a frequent need to select appropriate data services from the available choices and integrate them to create an environmental monitoring and decision support system. The prevalent cloud related service selection methods employ simple attribute-based matching which may not yield the most relevant alternatives for such an application from Environmental Sciences.

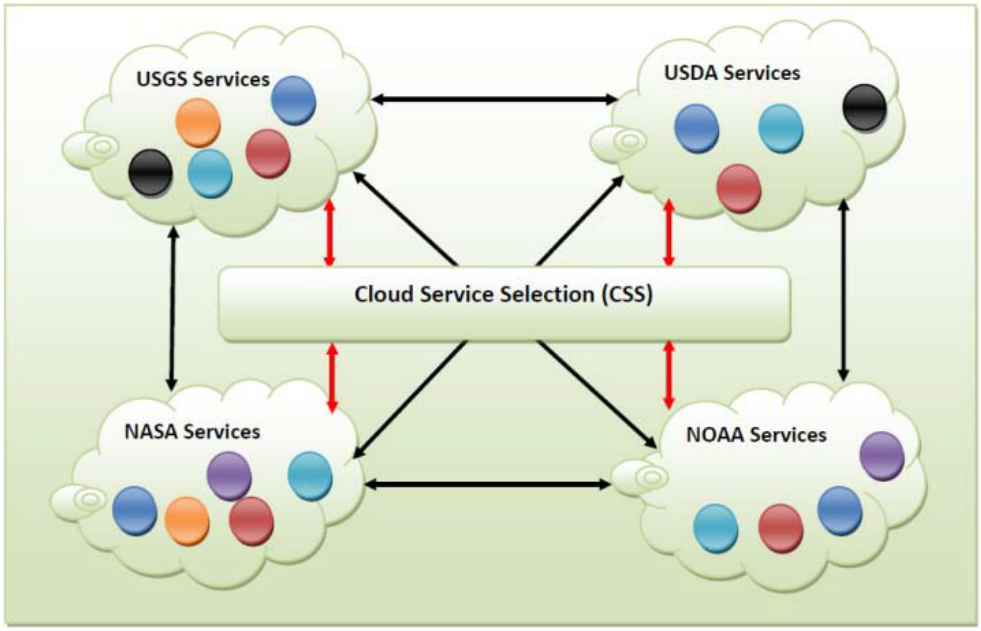


Figure 5.8. Environmental Science Service Clouds and CSS

Due to the large number of such available individual services, their possible permutations, and the associated inherent complexity, this task of discovery and selection of relevant service instances is highly time consuming and error prone, especially if the team has a specific bias or is not very familiar with a particular model of service. Also, a specific service may not be able to easily couple with another particular service, if these services operate at different time and spatial scales. In addition, these datasets and modeling services will usually have different formatting requirements, different software designs and technologies, different storage requirements, different computational requirements (especially syntactic and semantics) and different operational and concurrency semantics. And finally, some of these services might be available freely from national agencies (e.g., NASA and NOAA) and could be hosted in a public cloud, while others may be hosted in private clouds and their owners might charge for these services. All these factors will further increase the complexity of the selection process. Hence, the discovery and selection of appropriate services from the available ones in clouds need to consider many dimensions such as the underlying al-

gorithmic and technological techniques used, the nature and types of the inputs, the Quality of Service (QoS) associated with the results, the ability to handle concurrent requests, the cost of using the services, etc. The prevalent CC-based service selection methods use simplistic matching semantics that use a limited set of attributes. Such an approach is not suitable in many complex applications from a variety of scientific domains including Environmental Sciences.

The task of applying the principles of the proURDS to this case study of selecting earth science services is far from trivial due to: a) the inherent complexity (e.g., the number of available services and their peculiarities) of the Environmental Sciences domain, b) the unavailability of multi-level specifications for these services, and c) the continuous need for the involvement of an expert from that domain to decide the matching semantics and to assess the quality of the results (i.e., number of services) returned.

5.3.2 Multi-level Specification (of a Cloud Service)

Multi-level specifications (or contracts) and associated multi-level matching for software services is presented in Subsection 4.3.1 and Subsection 4.3.2. An example of a partial multi-level specification (in XML) for a Land Cover Data Service (from the domain of Environmental Sciences) is indicated in Figure 5.9. This partial specification shows six levels: a) General b) Syntactic, c) Semantic, d) Synchronization e) QoS and f) Auxiliary. How these different attribute levels of the specifications are matched by the Headhunters of the proURDS is also presented in Subsection 5.2 and the Table 5.1.

5.3.3 Scenario Motivation

The domain of Environmental Sciences frequently involves handling of the environmental preservation activities. In such situations, teams of Earth Scientists need to perform the cause-effect analyses to conclude about the health of certain ecological

```
<?xml version="1.0" encoding="UTF-8"?>
<CSSContract name="USDALandCoverDataService100" type="LandCoverDataService">
  <ComponentAttributes>
    <property name="DomainName" value="EarthScience"/>
      <property name="location"  value="http://www.ers.usda.gov/Data/MajorLandUses/"/>
  </ComponentAttributes>
  <ComputationalAttributes>
      <InherentAttributes>
      <property name="license" value="gov"/>
      </InherentAttributes>
      <FunctionalAttributes>
        <SyntaxAttributes>
          <ContractAttributes>
              <Contract>
               <property name="methodName" value="getLangUsageData" / >
                <property name="param1"    name="location"  type="state"/ >
                <property name="param2"    name="resolution"  type="all"/ >
                  <property name="returnType"  type="file" format="exel"/>
              </Contract>
            </ContractAttributes>
        </SyntaxAttributes>
        <SemanticAttributes>
          <PreConditon>
              <property variable="validLocation(location)" type="bool" value="true" />
           </PreConditon>
            <PostCondition>
              <property name="standardResultsFormat" value="bool" value="true"/>
          </PostCondition>
          <Invariant>
            <property name="uniformDataTimeValue" value="bool" value="true"/>
          </Invariant>
        </SemanticAttributes>
        <property name="complexity" value="O(n)"/>
      </FunctionalAttributes>
  </ComputationalAttributes>
  <QOSAttributes>
      <property name="timeFrame" scale="year(2002-2010)" value="2010"/>
      <property name="sorted" scale="bool" value="true"/>
      <property name="cost"  value="free"/>
  </QOSAttributes>
  <SynchronizationAttributes>
      <property name="MutualExclusion" implementation="websession"/>
  </SynchronizationAttributes>
  <AuxillaryAttributes>
      <property name="mobility" value="no"/>
  </AuxillaryAttributes>
</CSSContract >
```

Figure 5.9. Multi-level specification of a Land Cover Data Service

systems. These analyses are achieved by the creation of distributed software systems that are composed from a variety of individual services. At present, such research teams mostly depend on human intervention to make ad-hoc choices about relevant services. For example, an ecological monitoring system called as Emergent Environment Effects Forecasting System (EEEFS) that monitors the effects of an oil spill

on a body of water may consist of different types of environmental services that are hosted in public and/or private clouds along with the necessary data sets. Figure 5.10 shows the types of the services needed for composing the EEEFS.

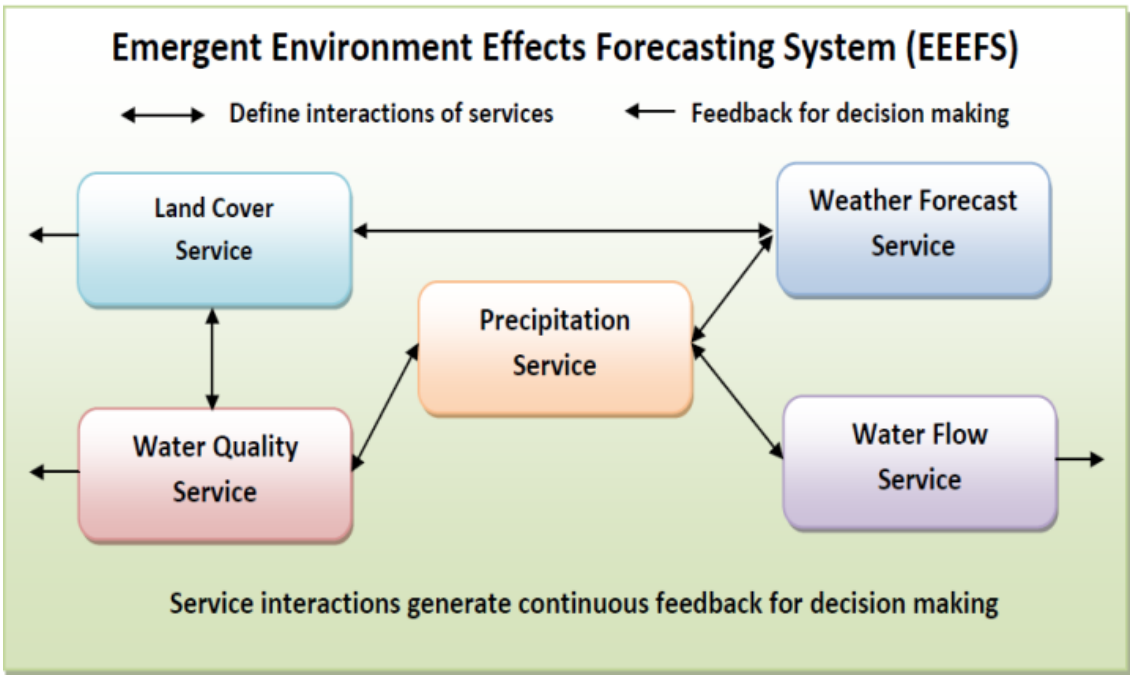


Figure 5.10. Architecture of the EEEFS

Selecting a proper instance of each of these types of services is based on a specific criterion that depends on the inherent nature of each type of service and also compatibility between various instances of different types. For example, the selection of an appropriate instance of the Weather Service may include considering the input/output parameter syntax details, associated semantics, and the QoS values. Due to the inherent complexity and various permutations between different instances, the EEEFS is an ideal choice to act as a case study to assess the applicability of the proURDS principles in the context of a cloud-based discovery.

5.3.4 Service Selection for EEEFS

To study the applicability of the proURDS and associated multi-matching principles in the context of EEEFS, the experimental infrastructure that simulated services from the categories of watershed modeling (water-flow and water quality) and spatial data modeling (land, soil, and elevation) and forecasting (weather forecasting) is created. Publically available services such as USGS [47], USDA [48], NLCD [51], SSURGO [52], and STATSGO [53] are used in the experiments. These existing services did not contain multi-level contracts and hence, their multi-level specifications are created. The main challenge in this step is to identify different instances of services (which required domain knowledge) and extract the details for each level of the multi-level specification of these services. Instances of these services specifications are deployed in the experimental setup. These services are distributed randomly into the active registries of the proURDS and queries were manually written and validated against the experts domain knowledge of existing services. Also, a sample KB for this domain is created in consultation with the domain expert. Figure 5.11 shows a part of this KB. As indicated earlier, the KB is consulted during the query process.

5.3.5 Results and Performance Evaluation

The experiment setup is made up of ten Dell machines running XP. Around 100-120 services are created for each category of services for EEEFS to test suitable services. All the levels of matching are performed except the synchronization level, because the synchronization contracts for the existing Environmental Science services could not be extracted due to the unavailability of their source code, and because most of the services use the default Web session synchronization technique. Also, the exact and relaxed matching semantics at each of the four levels are included in the experiments. Multi-Level Queries (MLQ) are issued to find the most appropriate services out of these instances. The MLQs used in the following experiments are a subset of Multi-level service specifications and are expressed in XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<CSSKnowladgeBase>
    <TypeRelation>
        <Synonym type="LandCoverService">
            <Type>LandCover</Type>
            <Type>LandUsage</Type>
            <Type>LandUsageService</Type>
              ...
        </Synonym>
        <inheritance super="ForestCoverService" sub="LandCoverService"/>
        <inheritance super="MidWestLandCoverService" sub="USLandCoverService"/>
        <inheritance super="USLandCoverService" sub="WorldLandCoverService"/>
            ...
        <coercion super="LandCoverServiceWithAllResultsByInteger"
                  sub="LandCoverServiceWithAllResultsByFloat" />
            ...
    </TypeRelation>

        .....
</CSSKnowladgeBase>
```

Figure 5.11. Partial Knowledge base

The first set of experiments compares the quality of the results returned by the proURDS prototype. The quality is measured as the number of relevant services returned for a particular query along with the usual metrics of precision and recall. These results of this experiment are manually inspected for their relevance. An example of such a MLQ for a Land Cover Service is shown in Figure 5.12.

```
<CSSQuery>
  <Query id="11">
    <QueryConfig level="0">
      <TypeRelaxed synonyms="true" inheritance="true" coercion="true">
        True
      </TypeRelaxed>
    </QueryConfig>
    <ComponentType>LandCoverService</ComponentType>
  </Query>
</CSSQuery>
```

Figure 5.12. Sample Query for Type exact matching

```
<CSSQuery>
  <Query id="11">
    <QueryConfig level="0">
      <TypeRelaxed synonyms="true" inheritance="true" coercion="true">
        True
      </TypeRelaxed>
    </QueryConfig>
    <ComponentType>LandCoverService</ComponentType>
  </Query>
</CSSQuery>
```

Figure 5.13. Sample Query for Type relaxed matching

As seen from Figure 5.12, each query is associated with a unique ID. The query configuration level indicates how many levels of the multi-level specification should be used in the process of matching. For example, in Figure 5.12 this attribute is 0, indicating that the matching should only take place at level 0, i.e., only at the type level. Also, in this query, relaxed matching semantics is not required. This is achieved by setting that specific attribute to false. This query resulted in 51 relevant services.

The sample query 2, shown in Figure 5.13, is used to retrieve Land Cover Services with a relaxed matching semantics only at the level of type.

The relaxed matching will not only retrieve services of type Land Cover, but also return services of type Forest Cover, as these two types are related by inheritance. Hence, more services (in this case 65) are returned for this query.

Table 5.4
Land Cover Service Query Results Comparison

Query Level	**Type**	**Syntax**	**Semantics**	**QoS**
Exact Matching	51	22	6	0
Relaxed Matching	65	25	8	2

```
<MCSSQuery id="21">
  <QueryConfig level="3">
    <TypeRelaxed synonyms="true" inheritance="true" coercion="true">
        True
    </TypeRelaxed>
    <SyntaxRelaxed synonyms="true" inheritance="true" coercion="true" default="true"
                   order="true">true</SyntaxRelaxed>
    <SemanticsRelaxed impl="true" revImpl="false" eq="false">true</SemanticsRelaxed>
    <QoSRelaxed compatibility="true" >true</ QoSRelaxed>
  </QueryConfig>
  <ComponentType>LandCoverData</ComponentType>
  <SyntaxAttributes>
     <ContractAttributes>
        <Contract>
           <property name="methodName" value="GetLandCoverData"/>
           <property name="parameter" value="zipcode" type="string" key="default"/>
           <property name="returnType"  type="file" format="exel" />
        </Contract>
     </ContractAttributes>
  </SyntaxAttributes>
  <SemanticAttributes>
        <PreConditon>
           <property variable="postalcode" type="string" cond="greater" value="0" />
        </PreConditon>
        <PostConditon>
           <property name="standardResultsFormat" value="bool" value="true"/>
        </PostConditon>
        <Invariant>
           <property name="uniformDataTimeValue" value="bool" value="true"/>
        </ Invariant>
  </SemanticAttributes>
  <QoSAttributes>
        < QoSAttribute>
           <property variable="timeFrame" type="year" value="2010" />
           <property variable="sorted" type="bool" value="true" />
           <property name="cost"  value="free"/>
        </ QoSAttribute>
  </QoSAttributes>
</ MCSSQuery>
```

Figure 5.14. Sample Query for All-level relaxed matching

Figure 5.14 indicates another query, for the Land Cover Service, which uses the relaxed semantic for all the levels of the specification. Hence, it consists of the details for all the levels of the multi-level specification and is more comprehensive than the first two queries.

Table 5.4 indicates the comparison of results for the query presented in Figure 5.14. As seen from Table 5.4, the exact matching at all levels does not yield any results

Table 5.5
EEEFS Relaxed Matching Criteria

Weather	**Possible Relaxed Matching Criterion**
Precipitation	(1) Minimize the distance from desired latitude-longitude or any other location indicator, (2) minimize cost
Water Flow	(1) Minimize the time duration overlap, (2) minimize the cost
Water	(1) Minimize the distance from desired latitude-longitude or any other location indicator, (2) maximize the overlapping time duration with respect to the desired time duration, (3) minimize cost
Quality	(1) Maximize the overlapping water quality parameters with respect to the desired water quality variables
Land Cover	(1) Maximize the overlap time of the map published, (2) minimize the distance between the grid size and the desirable grid size

for the Land Cover Service. However, relaxing the semantics of the operators at each level resulted in addtional matching instances for the Land Cover Service. Table 5.5 indicates the relaxed selection criteria, specified by the domain expert, used in this experiment.

Many more queries are executed in the given experimental setup with both the exact and relaxed matching semantics. Figure 5.15 shows the results of a few of these experiments. Here, for various types of queries the number of matching services returned after each level of matching and with exact and relaxed semantics is shown. As seen from the Figure 5.15, there is an increase in the number of matching services

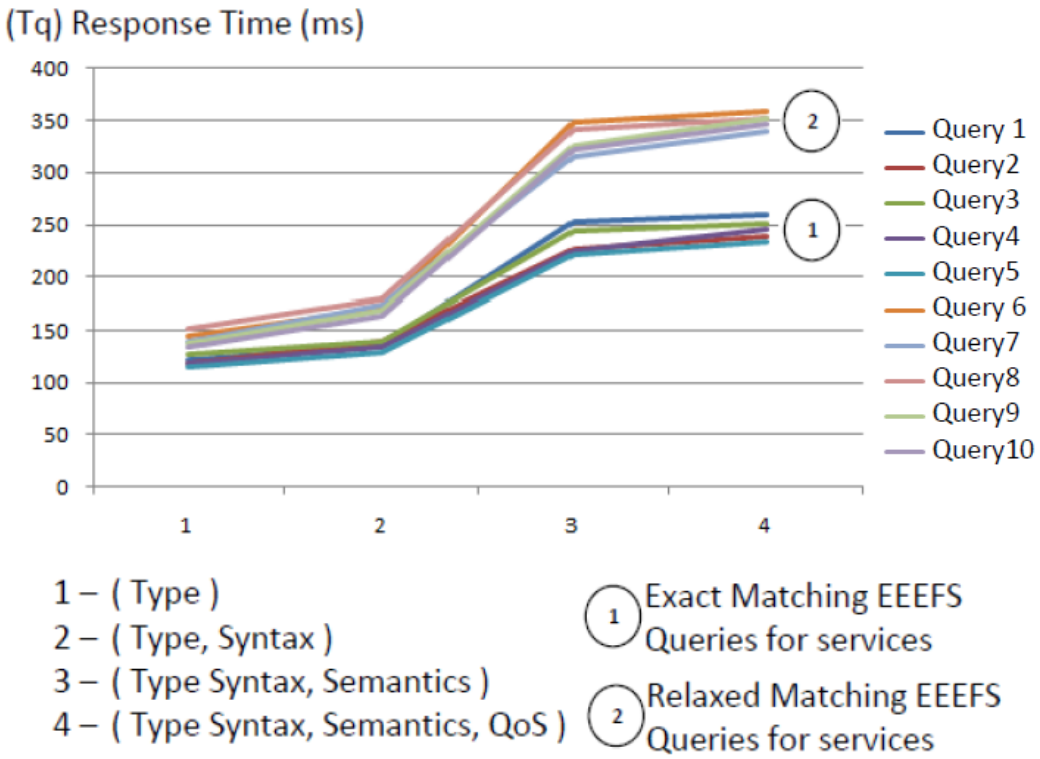


Figure 5.15. Comparison of the Quality of Result (Exact and Relaxed Matching)

in the case of the relaxed semantics as opposed to the exact matching semantics. This is expected due to the inherent nature of the relaxed matching operators.

Tables 5.6 and 5.7 indicate the precision and recall values for these queries using the exact and relaxed matching semantics respectively. As seen from these tables, it is evident that relaxed matching results in better precision and recall values.

Additional experiments are conducted to test the performance, as indicated by the time required to carry out the matching operations of the proURDS prototype. The Matching Time (T_m) is used as a metric in this set of experiments. T_m is defined as the time taken by the proURDS Headhunter (HH) to perform the MLM depending on its capabilities. If a HH performs matching at all the five levels then T_m is the sum of matching times observed at each level. T_q is the summation of T_m and the time

Table 5.6
Exact Matching Results for each type of Query

Query No	**Total No of Relevant Services**	**No of Returned Services**	**Resulted No of Relevant Services**	**Precision %**	**Recall %**
1.Land Cover Query	2	0	0	0	0
2.Weather Query	6	5	4	80	66
3.Precipi -tation Query	5	5	3	60	60
4.Water Quality Query	3	3	1	50	33
5.Water Flow 2Query	4	3	2	66	50

required for propagating a query to a particular HH and bringing the results back, and thus, indicates the end-to-end response time for a query.

Figure 5.16 shows the matching times required at each level for the both the semantics (exact/relaxed) of five types of EEEFS queries. As expected, each level of matching increases the response time. It is evident, from Figure 5.16, that the increase in the time required for the semantic matching is substantially more than the other levels, as it involves the use of a predicate proving with theorem prover [45] to

Table 5.7
Relaxed Matching Results for each type of Query

Query No	**Total No of Relevant Services**	**No of Returned Services**	**Resulted No of Relevant Services**	**Precision %**	**Recall %**
1.Land Cover Query	2	2	2	100	100
2.Weather Query	6	6	6	100	100
3.Precipi -tation Query	5	5	5	100	100
4.Water Quality Query	3	3	3	100	100
5.Water Flow 2Query	4	4	3	75	75

establish necessary relation between the semantic part of the query and the semantic specifications of the available Environmental Sciences services. Also it can be seen that among the two groups of queries, there is a tendency to increase response time in relaxed matching due to weaker matching semantics and associated KB inferences. Similarly, it is evident that T_m increases as a function of number of services.

In summary, selecting appropriate services from a set of available ones deployed in a cloud is a crucial, laborious, and possibly error-prone step. This case study has

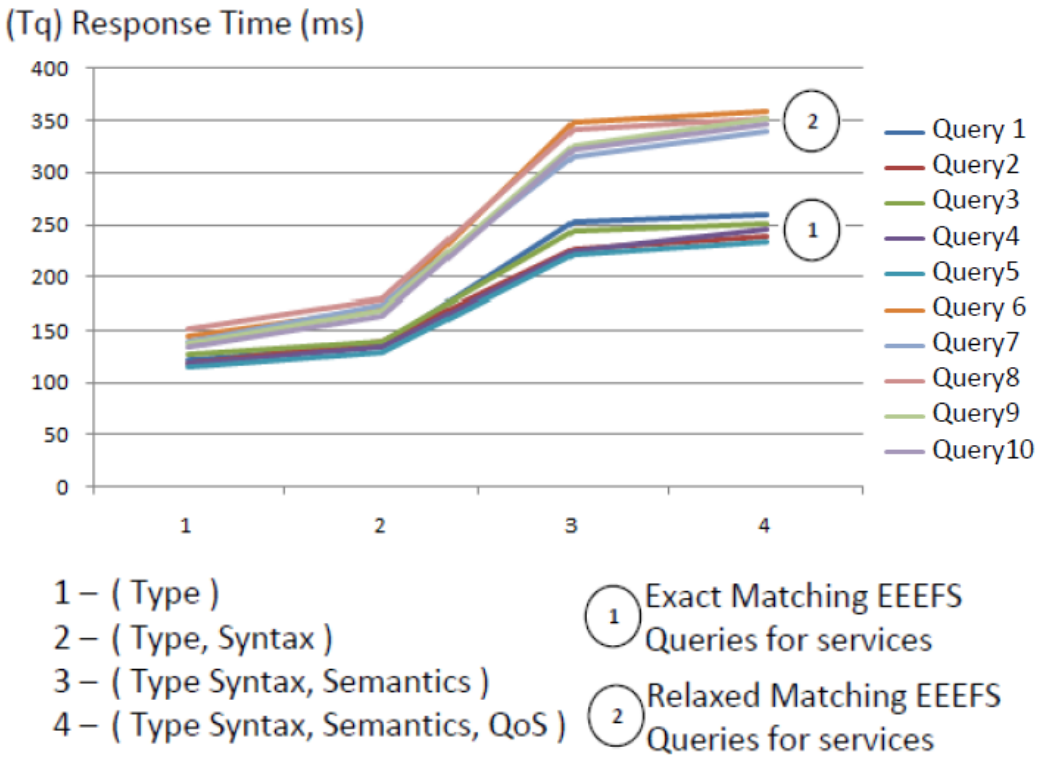


Figure 5.16. Individual Matching Times

empirically validated the applicability of the proURDS in this context. The results indicate that proURDS returns relevant cloud services as a result of the multi-level matching semantics.

6 CONCLUSION AND FUTURE WORK

Selecting services from a set of available ones over a network is a crucial step in developing distributed systems that are composed of individual services. Various techniques, ranging from simplistic attribute comparisons to multi-level matching, can be used for matching a query against a set of service specifications. Selecting appropriate services from a set of available ones deployed in the cloud is also a crucial, laborious, and possibly error-prone step. However, this step is essential in developing distributed applications that are composed of individual services which are deployed in the clouds. Hence, an automated and more extensive approach (than the prevalent ones) is needed to discover and select such relevant services. The URDS is one such hierarchical discovery system that uses the principles of multi-level specifications and associated matching.

The work presented (proURDS) in this thesis indicates an improved architecture from the previous version of URDS. The proURDS enhances the URDS by conducting the service discovery experiments in a distributed setup using actual services from a public dataset. The addition of the two new modules namely: the Knowledge base (KB) module and the Service Management and Monitoring (SMM) module enhance the URDS architecture by providing necessary domain knowledge and control, management and monitoring capabilities. Also the case study has presented an empirical validation of the proURDS using environmental science services. It also compared the performance of the proURDS with jUDDI, a publicly available discovery service implementation.

The results described in this thesis indicate that the proURDS returns relevant services (i.e., services with better quality) as a result of the multi-level matching semantics at the cost of increased response time. The quality of the services returned by the proURDS is measured in terms of precision and recall. Although the average

response time of the proURDS is high, this was expected due to the extra work performed by the multi-level matching.

Future work will include, in addition to more comprehensive experimentation, the investigation of multi-level matching in the context of uncertainty and incomplete service specifications. Other directions include further experimentation with the proURDS to investigate the effects of cloud service distribution topology on the matching process, creation and experimentation of additional levels of specification and matching, for example, trust contracts, economics contracts, and legal contracts.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Carney D. Foreman J. Haines, G. Component-based software development and cots integration. http://www.sei.cmu.edu/str/descriptions/cbsd.html, 2007.

[2] Olson, A., Raje, R., Bryant, B., Auguston, M., Burt, B. UniFrame - A Unified Framework for Developing Service-oriented, Component-based Distributed Software Systems. In Stojanovic, Z. and Dahanayake, A, editor, *Service-Oriented Software System Engineering: Challenges and Practices*, pages 68–87, IGI Publishing Hershey, PA, USA, 2005.

[3] Olson, A., Raje, R., Bryant, B., Auguston, M., Burt, B. UniFrame - Automating the Construction of Large-Scale Distributed Systems. In Dai, Y., Pan, Y., and Raje, R, editor, *Advanced Parallel and Distributed Computing: Evaluation, Improvement, and Practice*, New York, 2006.

[4] Siram, N. An Architecture for the UniFrame Resource Discovery Service. Master's thesis, Indiana University Purdue University Indianapolis, 2002. Department of Computer and Information Science.

[5] Devaraju, B. Enhancement of the UniFrame Resource Discovery Service. Master's thesis, Indiana University Purdue University Indianapolis, 2005. Department of Computer and Information Science.

[6] Katuri, P. Experimenting with Multilevel Matching Concepts for Software Components. Master's thesis, Indiana University Purdue University Indianapolis, 2006. Department of Computer and Information Science.

[7] Zaremski, A., and Wing, J. Specification matching of software components. In *Proceedings of SIGSOFT'95 Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 6–17, October 1995.

[8] Zaremski, A. and Wing, J. Specification Matching of Software Components. *ACM Transactions on Software Engineering*, 6(4):333–369, 1995.

[9] Siram, N., Raje, R., Bryant, B., Olson, A., Auguston, M., and Burt, C. An Architecture for the UniFrame Resource Discovery Service. In *Proceedings of SEM 2002, the 3rd International Workshop on Software Engineering and Middleware, Springer-Verlag Lecture Notes in Computer Science, Vol. 2596*, pages 20–35, 2003.

[10] *The QWS Dataset,*
URL: http://www.uoguelph.ca/ qmahmoud/qws/index.html, 2000.

[11] Lahiru S Gallege, Ketaki P Pradhan, and Rajeev R Raje. Experiments with a multi-level discovery system. In *Proceedings of the series International Conference in Computing (ICC 2010)*, New Delhi, India, 2010.

[12] Lahiru S Gallege, Aboli Phadke, Meghna Babbar-Sebens, and Rajeev R Raje. Cloud service selection for earth science domain. In *the 2nd International Conference on Recent Trends in Information Technology and Computer Science (ICRTITCS 2012)*, International Journal of Computer Applications (IJCA), 2012.

[13] Sun Microsystems. *Jini Specifications V2.0.*

[14] J Newmarch. *A Programmer's Guide to Jini Technology.* Apress, 2000. ISBN 1-893115-80-1.

[15] UPnP Organization. *UPnP Home Page*
URL: http://www.upnp.org, 2005.

[16] J Kemp. *Service Location Protocol for Enterprise Networks.* Wiley and Son Inc. ISBN 0-47-3158-7.

[17] OpenSLP Organization. *OpenSLP Home Page*
URL: http://www.openslp.org, 2005.

[18] *UDDI Technical White Paper,*
URL: http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf, 2000.

[19] *Trading Object Service Specification,*
URL: ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf, 2000.

[20] *Globus Toolkit,*
URL: http://www.globus.org/toolkit/, 2007.

[21] Seacord, R., Hissam, A., Wallnau, K. AGORA: A Search Engine for Software Components. *IEEE Internet Computing*, 1998.

[22] *Ninja Project,*
URL: http://ninja.cs.berkeley.edu, 2005.

[23] von Behren, J., Brewer, E., Borisov, N., Chen, M., Welsh, M., MacDonald, J., Lau, J., Culler, D. Ninja: A Framework for Network Services. In *Proceedings of USENIX Annual Technical Conference*, 2002.

[24] Banaei-Kashani, F., Chen, C., Shahabi, C. WSPDS: Web Services Peer-to-Peer Discovery Service. In *Proceedings of International Conference on Internet Computing*, 2004.

[25] Chakraborty, D., Perich, F., Avancha, S. and Joshi, A. DReggie: A Smart Service Discovery Technique for E-Commerce Applications. In *Proceedings, 20th Symposium on Reliable Distributed Systems*, October 2001.

[26] Di Martino, B. Semantic web services discovery based on structural ontology matching. In *Proceedings of IJWGS*, 2009.

[27] Lin, C., Wu, Z., Deng, S., Kuang, L. Automatic Service Matching and Service Discovery Based on Ontology. In *GCC Workshops*, 2004.

[28] DARPA. *The DARPA Agent Markup Language,*
URL: http://www.daml.org/, 2006.

[29] Ankolenkar, A., Burstein, M., Hobbs, J., Lassila, O., Martin, D., McDermott, D., McIlraith, S., Narayanan, S., Paolucci, M., Payne, T., Sycara, K. DAML-S: Web Service Description for the Semantic Web. In *Proceedings of First International Semantic Web Conference*, 2002.

[30] Horrocks, I. DAML+OIL: a reason-able web ontology language. In *Proceedings of Extending Database Technology*, 2002.

[31] Arabshian, K. and Schulzrinne, H. GloServ: global service discovery architecture. *Mobile and Ubiquitous Systems: Networking and Services*, pages 319–325, 2004.

[32] Chara Skouteli, George Samaras, and Evaggelia Pitoura. Concept-based discovery of mobile services. In *MDM '05: Proceedings of the 6th international conference on Mobile data management*, pages 257–261, New York, NY, USA, 2005. ACM.

[33] Gu, T., Qian, H., Yao, J., Pung, H. An Architecture for Flexible Service Discovery in OCTOPUS. In *Proceedings of 12 th ICCCN*, 2003.

[34] W3C. *Resource Description Framework, URL: http://www.w3.org/RDF/*, 2004.

[35] Arabshian, K., Dickmann, C., Schulzrinne, H. Ontology-Based Service Discovery Front-End Interface for GloServ. In *Proceedings ESWC* , 2009.

[36] Taekgyeong Han and Kwang Mong Sim. An ontology-enhanced cloud service discovery system. *Computer*, I:644–649, 2010.

[37] Wenying Zeng, Yuelong Zhao, and Junwei Zeng. Cloud service and service selection algorithm research. *Proceedings of the first ACMSIGEVO Summit on Genetic and Evolutionary Computation GEC 09*, page 1045, 2009.

[38] Phillip C-y Sheu, S H U Wang, Q I Wang, K E Hao, and R A Y Paul. Semantic computing, cloud computing, and semantic search engine. *2009 IEEE International Conference on Semantic Computing*, 1:654–657, 2009.

[39] Raichura Bhavin and Agarwal Ashutosh. Infosys cloud computing white paper. http://www.infosys.com/cloud-computing/white-papers/Documents/service-exchange-cloud.pdf, 2009.

[40] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P King, and Richard Franck. Web service level agreement (wsla) language specification. *Language*, pages 1–110, 2003.

[41] Pankesh Patel, Ajith Ranabahu, and Amit Sheth. Service level agreement in cloud computing. *Cloud Workshops at OOPSLA09*, pages 1–10, 2009.

[42] R. Raje, P. Katuri, A. Kumari, and O. Tilak. Experiments with a multi-level discovery system. In *Proceedings of the International Conference on Computer Communication and Instrumentation*, Mumbai, India, 2009.

[43] Jejurikar, A. Knowledgebase Architecture for Distributed Computing Systems. Master's thesis, Indiana University Purdue University Indianapolis, 2007. Department of Computer and Information Science.

[44] Eisenecker U. Czarnecki, C. *Generative Programming.* Addison-Wesley, 2000.

[45] *The Java Theorem Prover, An object-oriented modularreasoning system, URL: http://www-ksl.stanford.edu/software/jtp/*, 2000.

[46] *UDDI Reference Implementation for Java (Apache jUDDI), URL: http://ws.apache.org/juddi/index.html*, 2000.

[47] *United Stated Geological Survey (USGS), URL: http://eros.usgs.gov*, 2012.

[48] *United Stated Department of Agriculture (USDA), URL: http://www.ers.usda.gov/Data/MajorLandUses/*, 2012.

[49] *National Aeronautics and Space Administration (NASA) , URL:* , 2012.

[50] *National Oceanic and Atmospheric Administration (NOAA), URL: http://www.ngs.noaa.gov/productsservices.html*, 2000.

[51] *National Land Cover Data (NLCD), URL: http://www.usgsquads.com/prodNLCD.htm*, 2012.

[52] *Soil Survey Geographic Data (SSURGO), URL: http://soils.usda.gov/survey/geography/ssurgo/*, 2012.

[53] *United Stated General Soil Map (STATSGO), URL: http://soils.usda.gov/survey/geography/statsgo/*, 2012.

APPENDICES

APPENDIX A THE PROURDS USER GUIDE

This appendix presents a user (i.e., service integrators point of view) guide for the proURDS. The guide is explained with respect to a sample scenario associated with the discovery service operation. The experimental setup of this guide contains three machines as experimental nodes. It presents a series of screen captures which illustrate how to create the sample setup of proURDS and how to query and obtain results.

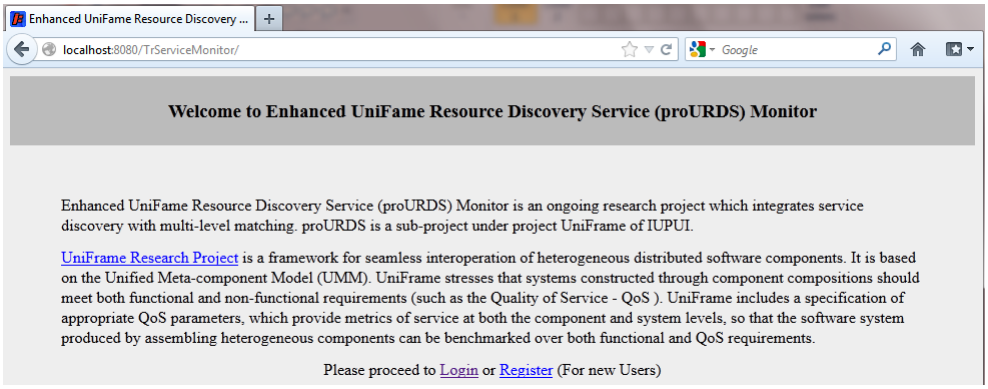


Figure A.1. SMM Startup Screen of the proURDS

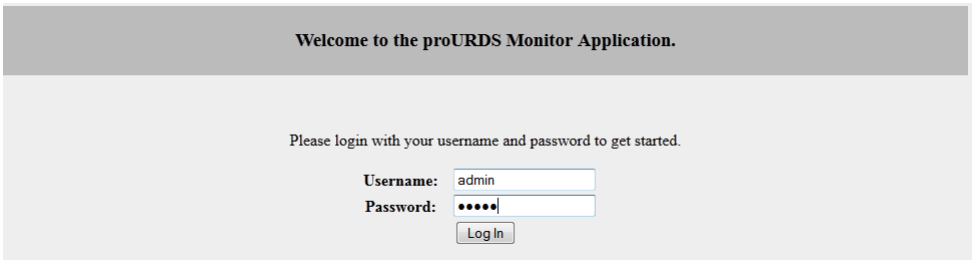


Figure A.2. Login screen of the proURDS

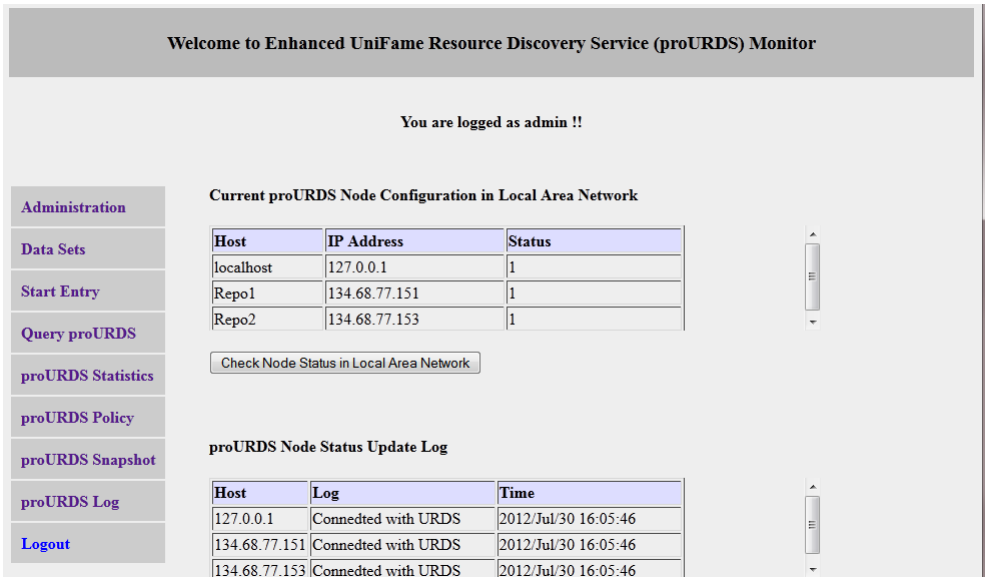


Figure A.3. Administration screen of the proURDS which allows to monitor the distributed setup

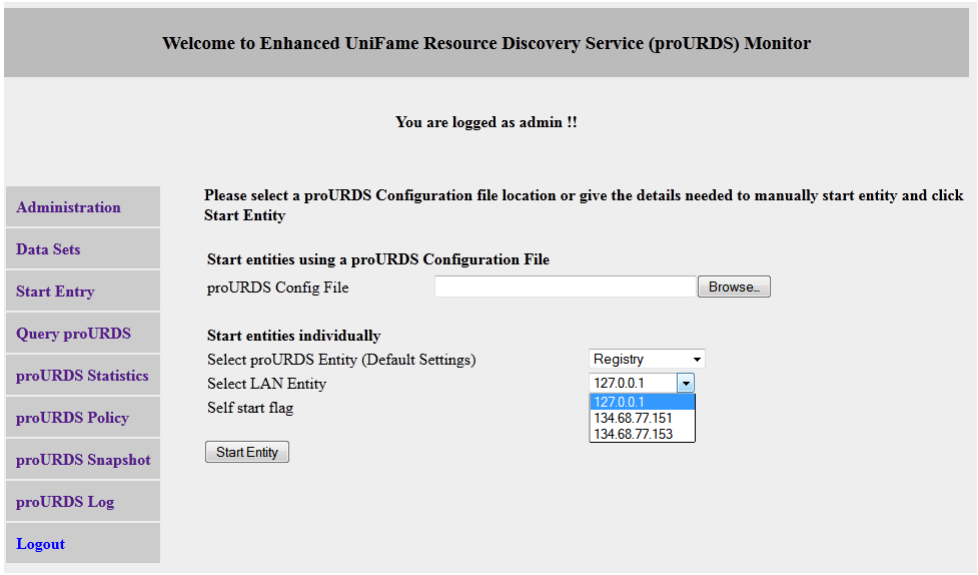


Figure A.4. Configuration page of the proURDS which allows to configure (i.e., deploy a configuration or manually start and stop entities) the setup

```
<?xml version="1.0" encoding="UTF-8"?>
<proURDSEntityConfig>
    <proURDSEntity>
        <machine>in-caci-rrpc01</machine>
        <ip>10.234.140.27</ip>
        <entity>Registry</entity>
        <quantity>1</quantity>
        <selfstart>true</selfstart>
        <repositorylocation>REPO1</repositorylocation>
    </proURDSEntity>
    <proURDSEntity>
        <machine>in-caci-rrpc02</machine>
        <ip>10.234.140.28</ip>
        <entity>Registry</entity>
        <quantity>1</quantity>
        <selfstart>true</selfstart>
        <repositorylocation>REPO2</repositorylocation>
    </proURDSEntity>
    <proURDSEntity>
        <machine>in-caci-rrpc03</machine>
        <ip>10.234.140.29</ip>
        <entity>Registry</entity>
        <quantity>1</quantity>
        <selfstart>true</selfstart>
        <repositorylocation>REPO3</repositorylocation>
    </proURDSEntity>
    <proURDSEntity>
        <machine>in-caci-rrpc04</machine>
        <ip>10.234.140.31</ip>
        <entity>Registry</entity>
        <quantity>1</quantity>
        <selfstart>true</selfstart>
        <repositorylocation>REPO1</repositorylocation>
    </proURDSEntity>
    <proURDSEntity>
        <machine>in-caci-rrpc05</machine>
        <ip>10.234.140.32</ip>
        <entity>Registry</entity>
        <quantity>1</quantity>
        <selfstart>true</selfstart>
        <repositorylocation>REPO2</repositorylocation>
    </proURDSEntity>
    <proURDSEntity>
        <machine>in-caci-rrpc06</machine>
        <ip>10.234.140.33</ip>
        <entity>Registry</entity>
        <quantity>1</quantity>
```

Figure A.5. Sample configuration file of the proURDS which is used to start entities

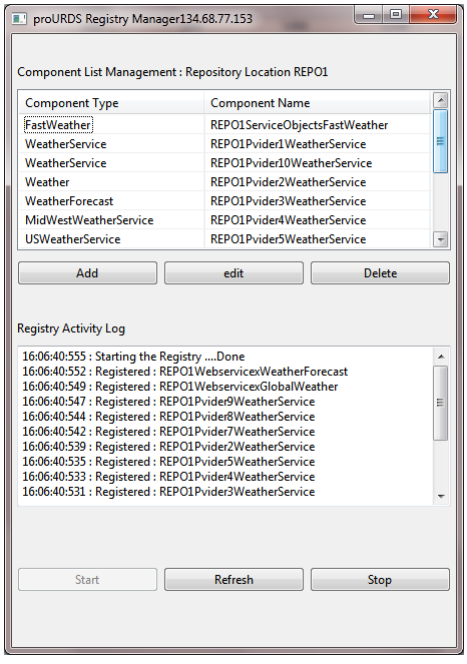


Figure A.6. Started proURDS Registry Manager UI which displays a list of available contracts and the event log

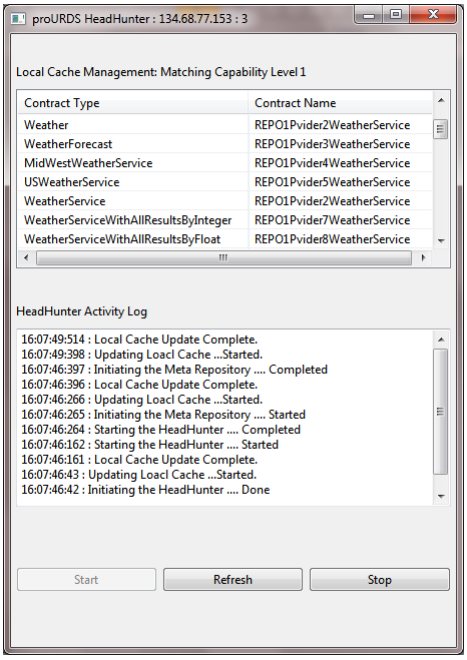


Figure A.7. Started proURDS Headhunter UI which displays a list of currently acquired contracts (from various registries) and its event log.

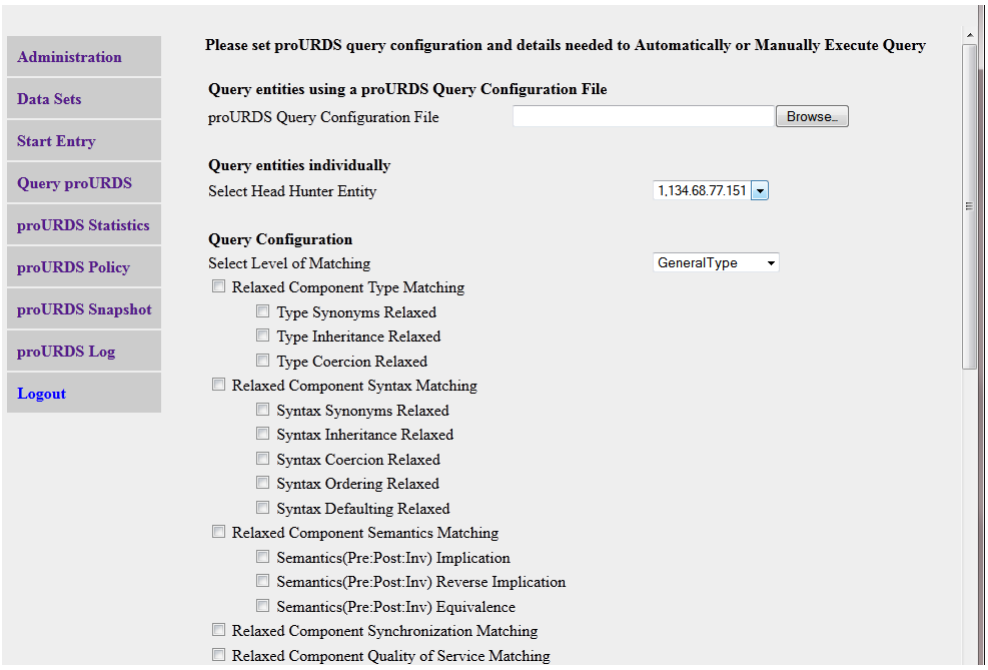


Figure A.8. Query interface for the user provided by the SMM which allows either use of a query configuration file or use of the user interface controls

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

<proURDSQueryConfig>

    <proURDSQuery id="1">
        <QueryConfig level="0">
            <TypeRelaxed synonyms="false" inheritance="false" coercion="false">false</TypeRelaxed>
        </QueryConfig>
        <ComponentType>Weather</ComponentType>
    </proURDSQuery>

</proURDSQueryConfig>
```

Figure A.9. A partial query configuration part of a query (type configuration)

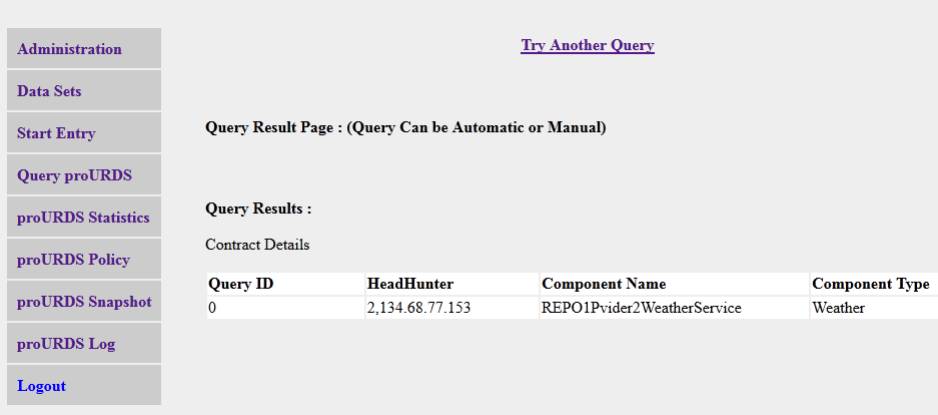


Figure A.10. Results obtained from only one HH for a sample query (with no relaxed operations)

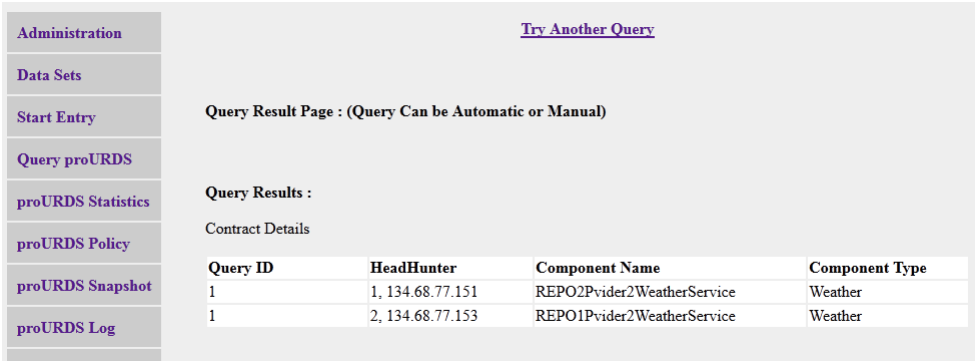


Figure A.11. Results obtained from two HHs for the same query (with no relaxed operations)

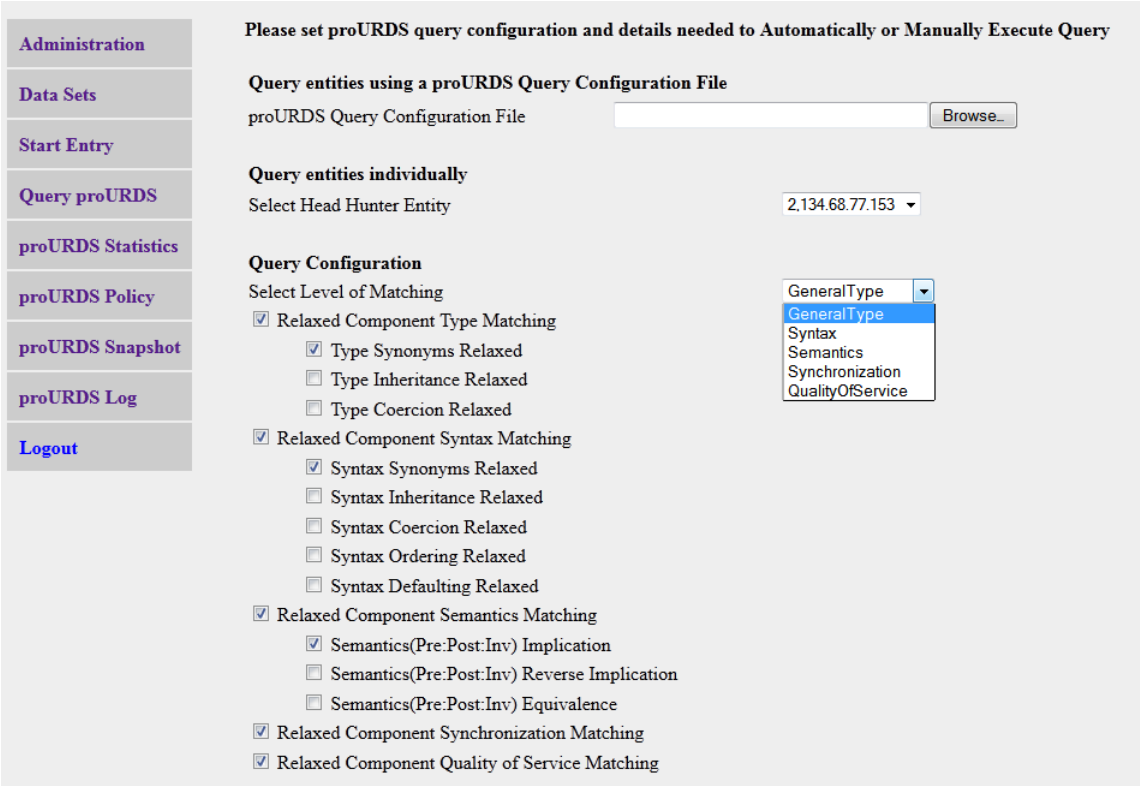


Figure A.12. Different levels of query configuration provided by the user interface

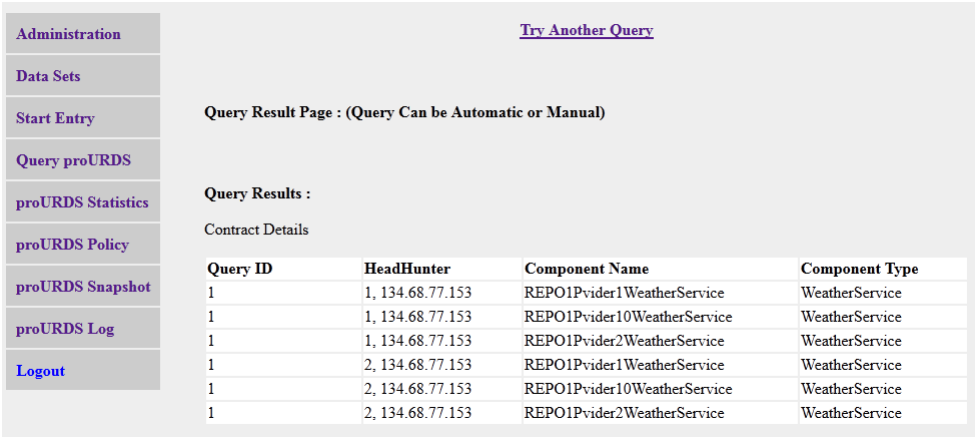


Figure A.13. Sample results for the initial query with relaxed matching enabled and the maximum level of matching set to syntax

```
<proURDSQuery id="2">
    <QueryConfig level="2">
        <TypeRelaxed synonyms="true" inheritance="true" coercion="true">true</TypeRelaxed>
        <SyntaxRelaxed synonyms="true" inheritance="false" coercion="false" default="false" order="false">true</SyntaxRelaxed>
        <SemanticsRelaxed impl="false" revImpl="false" eq="false">false</SemanticsRelaxed>
    </QueryConfig>
    <ComponentType>WeatherService</ComponentType>
    <SyntaxAttributes>
        <ContractAttributes>
            <Contract>
                <property name="methodName" value="GetWeather"/>
                <property name="parameter" value="postalcode" type="string" key="default"/>
                <property name="returnType"  type="string"/>
            </Contract>
        </ContractAttributes>
    </SyntaxAttributes>
    <SemanticAttributes>
        <PreConditon>
            <property variable="postalcode" type="string" cond="greater" value="0" />
            <property variable="postalcode" type="string" cond="less" value="25"/>
        </PreConditon>
        <PostCondition>
            <property name="ret_weather" value="string" cond="notnull"/>
        </PostCondition>
    </SemanticAttributes>
</proURDSQuery>
```

Figure A.14. Sample multi-level query configuration file which allows setting of different operators at different levels including details for exact and relaxed matching

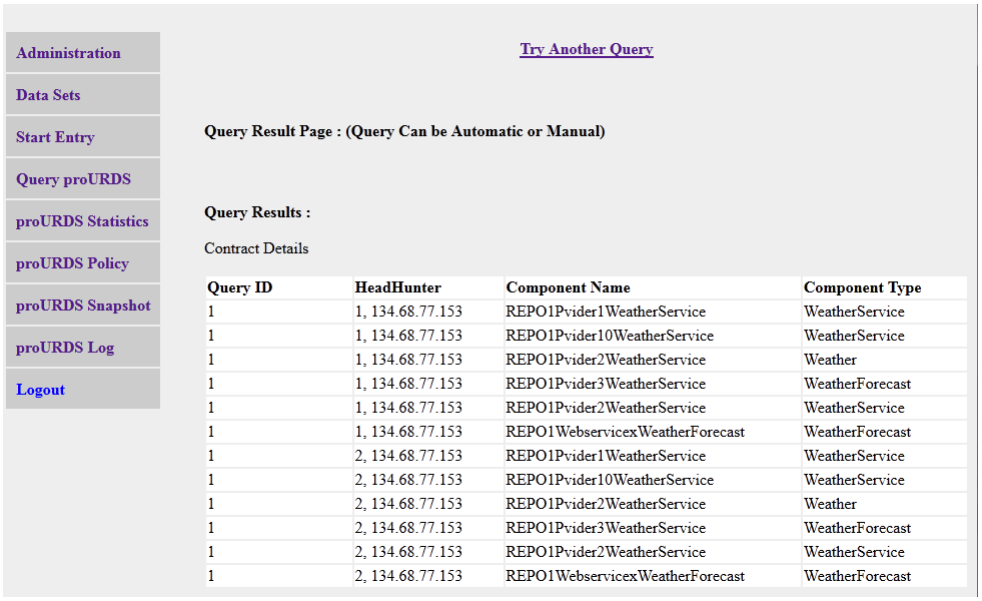


Figure A.15. Sample results for a query with relaxed matching enabled and the maximum level of matching set to QoS

```xml
<?xml version="1.0" encoding="UTF-8"?>
<proURDSKnowladgeBase>

    <TypeRelation>
        <Synonym type="WeatherService">
            <Type>Weather</Type>
            <Type>WeatherForecast</Type>
        </Synonym>
        <inheritance super="MidWestWeatherService" sub="USWeatherService"/>
        <inheritance super="USWeatherService" sub="WeatherService"/>
        <inheritance super="WeatherService" sub="WorldWeatherService"/>
        <coercion super="WeatherServiceWithAllResultsByInteger" sub="WeatherServiceWithAllResultsByFloat" />
    </TypeRelation>

    <SyntaxRelation>
        <Synonym method="GetWeather">
            <Method>GetWeatherData</Method>
        </Synonym>
        <coercion super="int" sub="long" />
        <coercion super="float" sub="double" />
    </SyntaxRelation>

</proURDSKnowladgeBase>
```

Figure A.16. Sample of the partial Knowledge base which referred by the matching operators of the proURDS

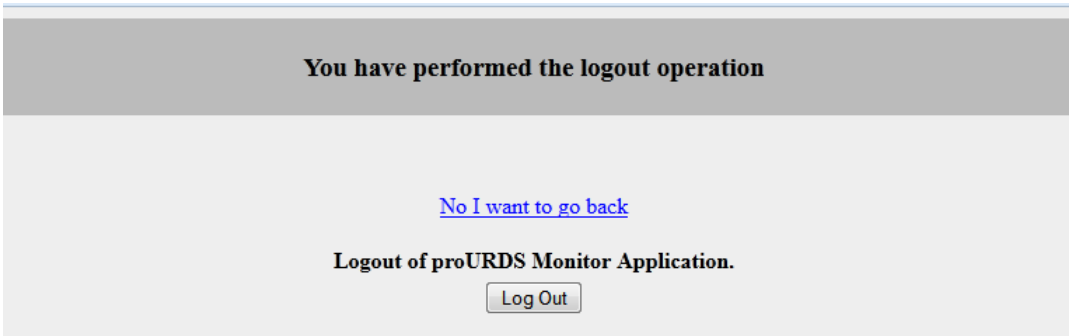


Figure A.17. Logoff screen of the proURDS which allows to go back or to terminate the setup

APPENDIX B THE DESIGN DIAGRAMS

This supplement appendix displays partial class diagrams of some selected packages of the proURDS source code.

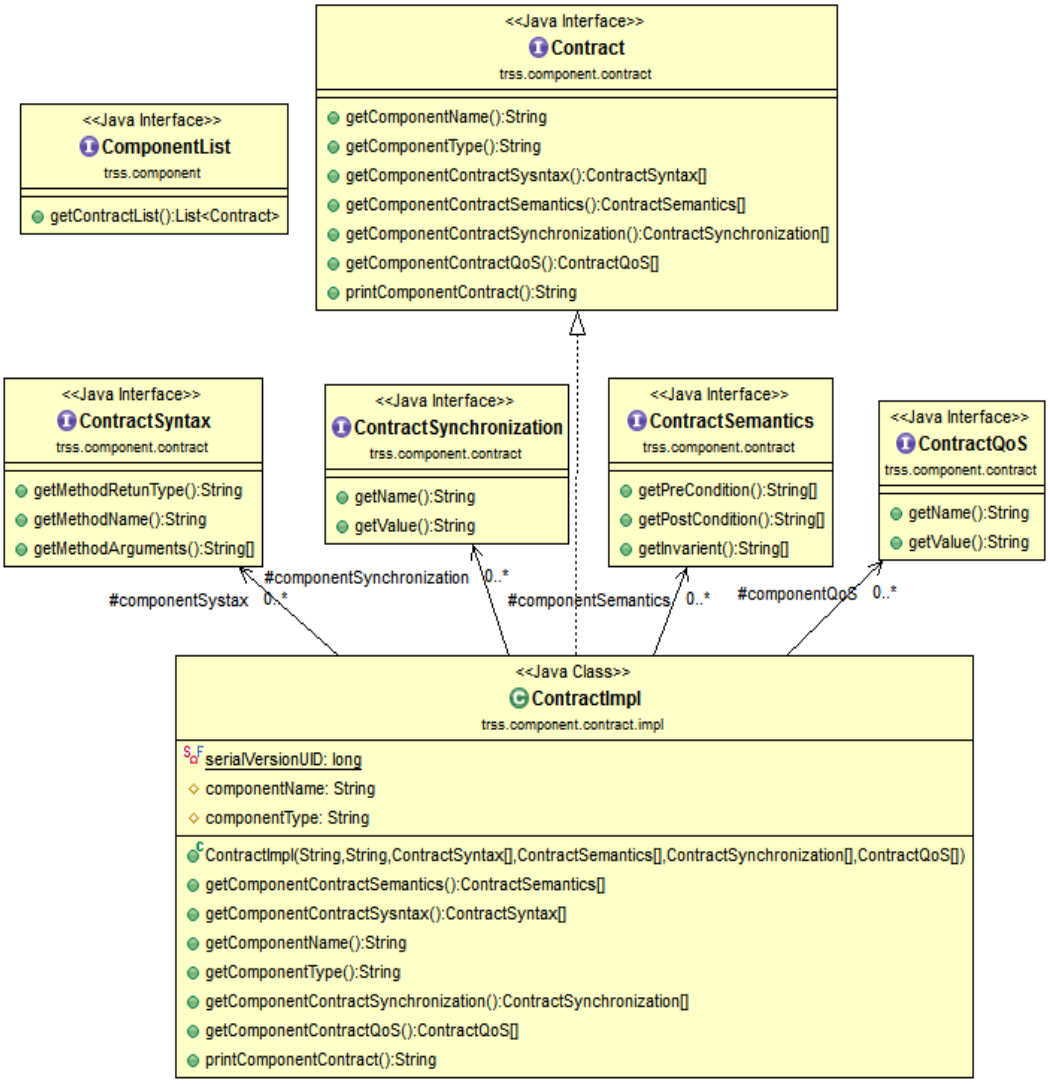


Figure B.1. Partial Class Diagram of the Contract interfaces and implementation classes

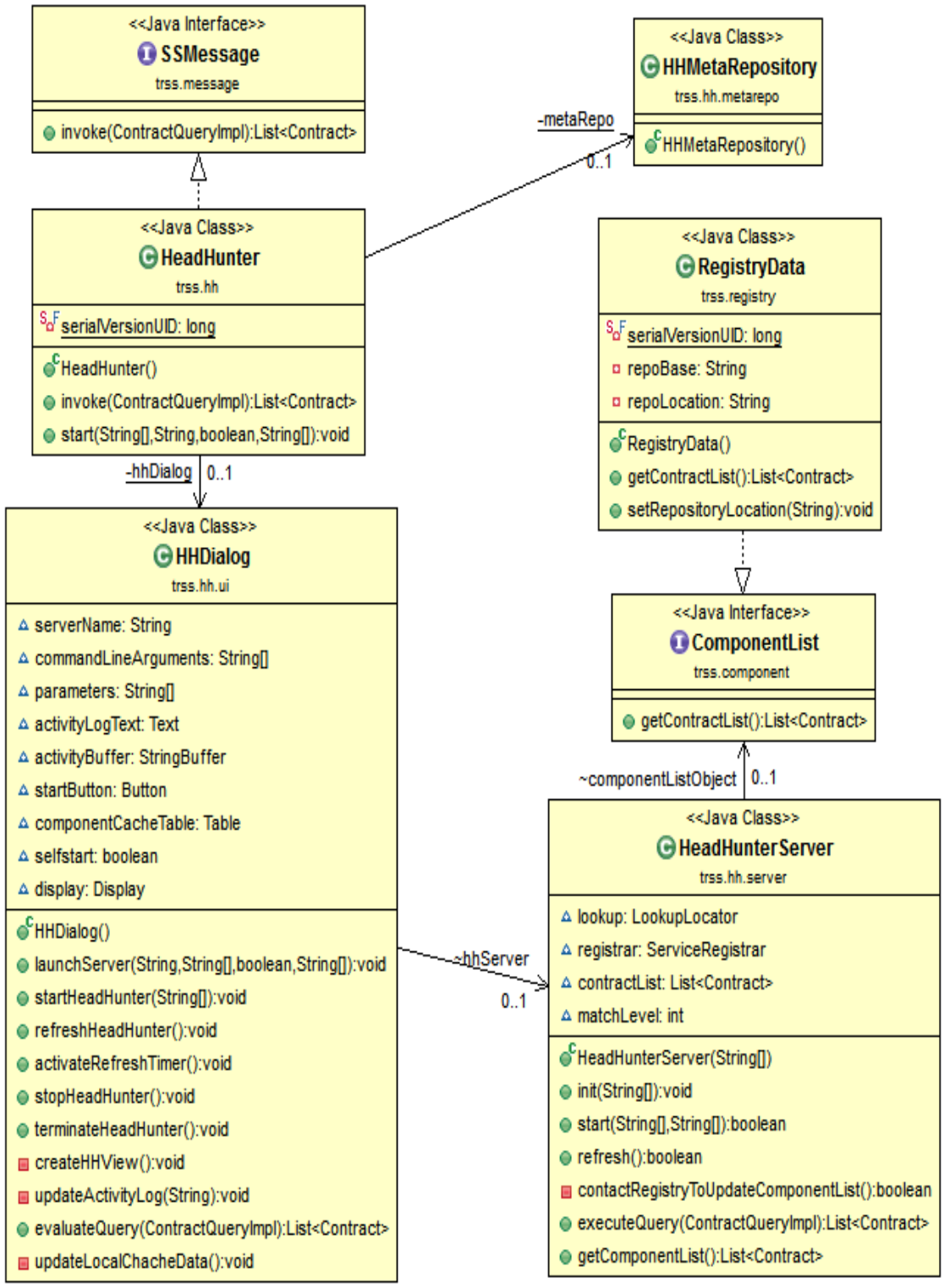


Figure B.2. Partial Class Diagram of the Headhunter (HH) interfaces and implementation classes

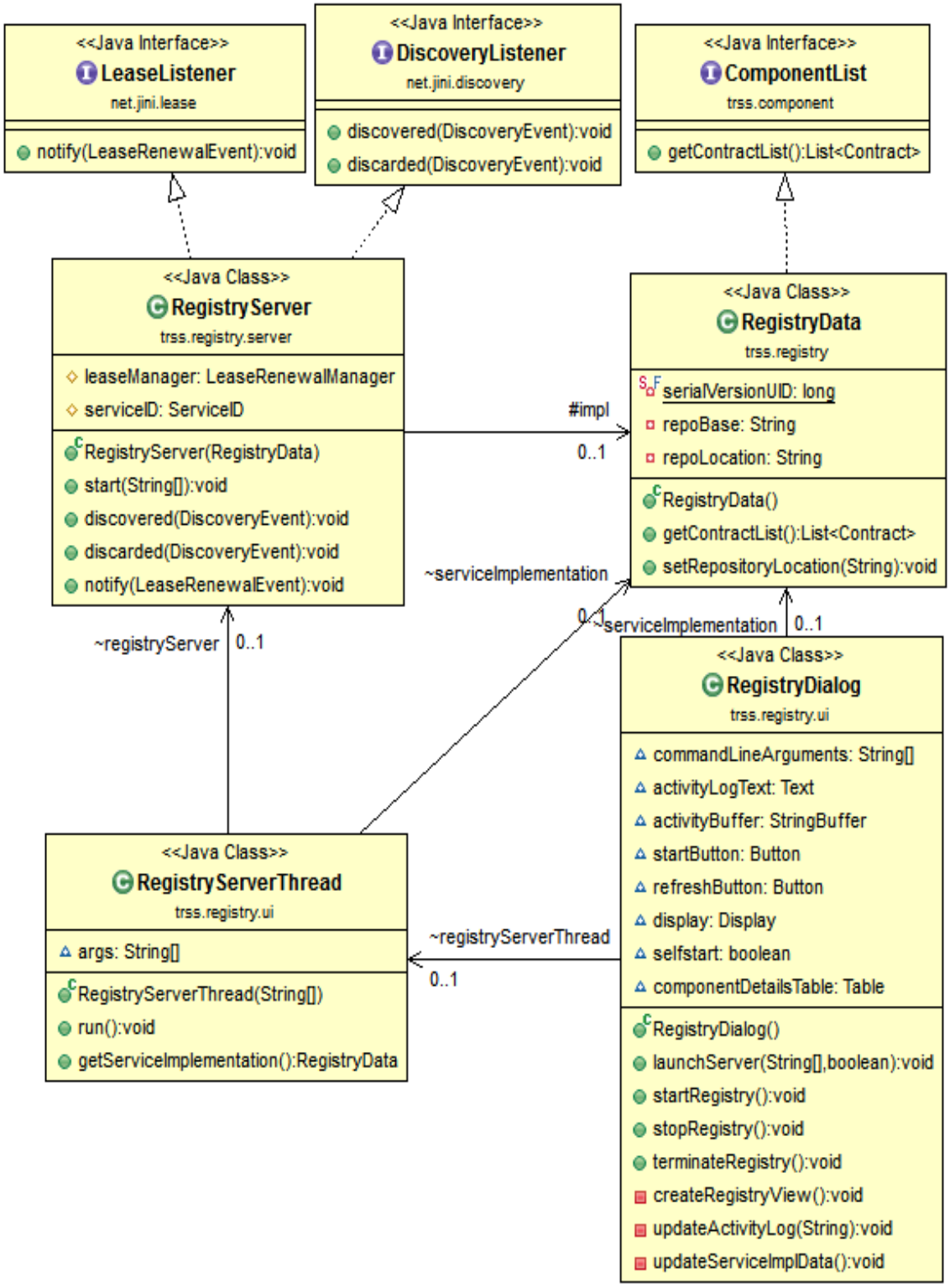


Figure B.3. Partial Class Diagram of the Active Registry (AR) interfaces and implementation classes

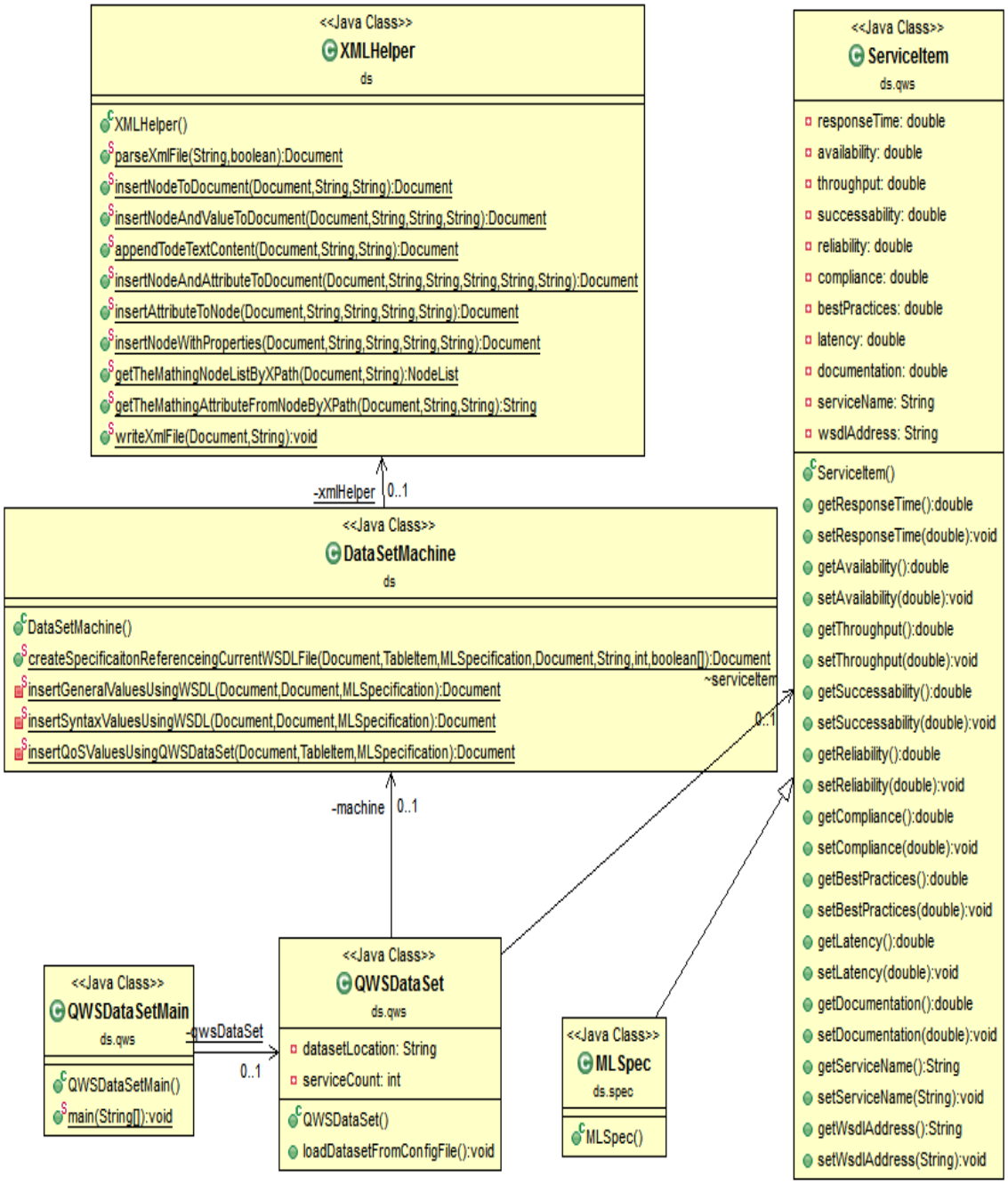


Figure B.4. Partial Class Diagram of the Dataset implementation classes

APPENDIX C THE SOURCE CODE

This supplement appendix displays a list of sample source code snippets taken of the proURDS. The list includes the contract interface, the message interface, the control interface, the Headhunter (HH) and the Active Registry (AR) threads, the matching algorithm (which performs multi-level matching), a sample web application user interface page (Java Server Page) which displays results, the web application configuration (web.xml) file, a sample Maven build script and part of the database setup script. For more details of the proURDS source code, please refer to the UniFrame project website or email the project team (unframe@cs.iupui.edu).

```
1  package trss.component.contract;
2
3  import trss.component.contract.ContractSemantics;
5
6  public interface Contract {
7
8      // Component Name
9      public String getComponentName();
10
11     // Component Type
12     public String getComponentType();
13
14     // Component Syntax
15     public ContractSyntax[] getComponentContractSysntax();
16
17     // Component Semantics
18     public ContractSemantics[] getComponentContractSemantics();
19
20     // Component Semantics
21     public ContractSynchronization [] getComponentContractSynchronization();
22
23     // Component Semantics
24     public ContractQoS[] getComponentContractQoS();
25
26     public String printComponentContract();
27
28 }
```

Figure C.1. Contract interface of proURDS code

```
1  package trss.message;
2
3  import java.rmi.Remote;
9
10 public interface SSMessage extends Remote {
11     List<Contract> invoke(ContractQueryImpl message) throws RemoteException;
12 }
```

Figure C.2. Serializeable message interface of the proURDS

```
1  package trss.server;
2
3  import java.rmi.Remote;
5
6  public interface SSControl extends Remote {
7
8      boolean isURDSServerAlive() throws RemoteException;
9
10     boolean startQueryManager(String[] args) throws RemoteException;
11
12     boolean stopQueryManager(String[] args) throws RemoteException;
13
14     boolean startHeadHunter(String[] args, boolean selfStartFlag) throws RemoteException;
15
16     boolean stopHeadHunter(String[] args) throws RemoteException;
17
18     boolean startRegistry(String[] args, boolean selfStartFlag) throws RemoteException;
19
20     boolean stopRegistry(String[] args) throws RemoteException;
21
22 }
```

Figure C.3. Control interface of the proURDS distributed setup

```java
package trss.server;

import java.rmi.Naming;

public class SSServer extends UnicastRemoteObject implements SSControl{

    private static final long serialVersionUID = -4401845276729445341L;

    protected SSServer(String[] args) throws RemoteException {
        super();
    }

    public boolean isURDSServerAlive() throws RemoteException {
        return true;
    }

    public boolean startHeadHunter(String[] args, boolean selfStartFlag) throws RemoteException {
        HeadHunterThread headHunterThread = new HeadHunterThread();
        headHunterThread.setSelfStartFlag(selfStartFlag);
        headHunterThread.setHeadHunterName(args[0]);
        headHunterThread.setPriority(java.lang.Thread.MIN_PRIORITY);
        if (args.length >1 && args[1] != null) {
            headHunterThread.setParams(new String[] { args[1] });
        }
        headHunterThread.start();
        return true;
    }
```

Figure C.4. Part of the proURDS server code base which implements the control interface

```java
package trss.server;

import trss.hh.HeadHunter;

public class HeadHunterThread extends Thread {

    private String arg1 = "-Djava.security.policy=policy.all";
    private String arg2 = "-Djava.rmi.server.codebase=file:bin";
    private String hhName;
    private boolean selfStartFlag = false;
    private String[] params = {"1"}; //default matching level

    public void run() {
        HeadHunter headHunter;
        try {
            headHunter = new HeadHunter();
            headHunter.start(new String[]{arg1,arg2}, hhName, selfStartFlag,params);
        } catch (Exception e) {
            //TODO Assume mostly self terminated
            e.printStackTrace();
            //Send the update details to the URDSServer and terminate
        }
    }

    public synchronized void setHeadHunterName(String name){
        this.hhName = name;
    }
```

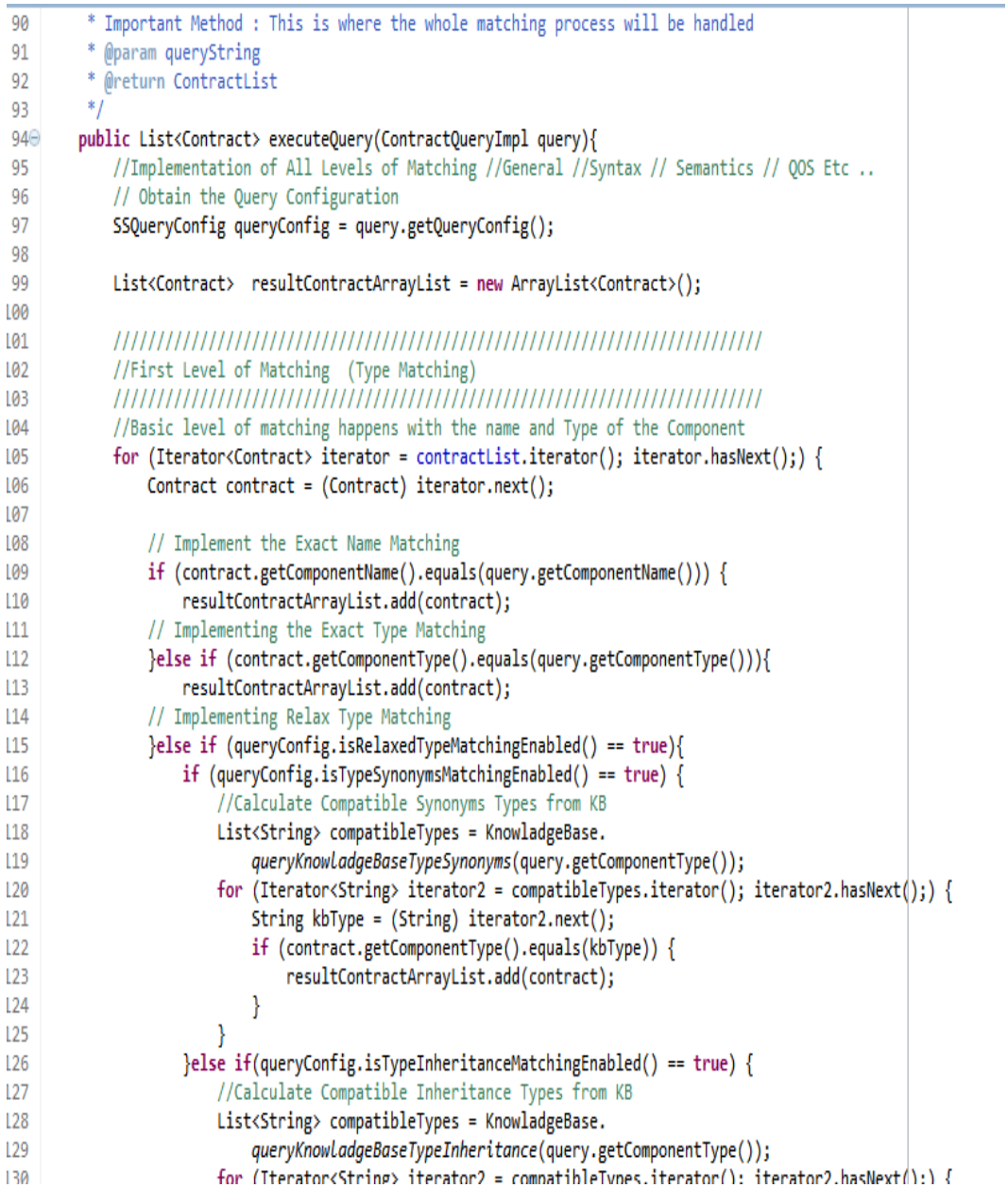
Figure C.5. Part of the source code of the Headhunter (HH) thread

```java
package trss.registry.server;

import java.io.DataInputStream;

public class RegistryServer implements DiscoveryListener, LeaseListener {

    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();
    protected ServiceID serviceID = null;
    protected   RegistryData impl;

    public RegistryServer(RegistryData impl){
        // Update the service implementation data
        this.impl = impl;
    }

    public void start(String[] argv) {

        // Try to load the service ID from file. It isn't an error if we can't load it,
        // because maybe this is the first time this service has run
        DataInputStream din = null;
        LookupDiscovery discover = null;
        try {
            din = new DataInputStream(new FileInputStream("RegistryData.id"));
            serviceID = new ServiceID(din);
            System.setSecurityManager(new RMISecurityManager());
            discover = new LookupDiscovery(LookupDiscoveryManager.ALL_GROUPS);
        } catch(Exception e) {
            System.err.println("Discovery failed " + e.toString());
            //Do nothing
        }

        discover.addDiscoveryListener(this);
```

Figure C.6. Part of the source code of the Active Registry (AR) thread

```
drop table APP.USERS;
drop table APP.HOSTS;
drop table APP.Registry;
drop table APP.HeadHunter;
drop table APP.QueryManager;
drop table APP.Policy;
drop table APP.Results;
drop table APP.Stats;
drop table APP.UpdateLog;
drop table APP.ActivityLog;

-- \************* App: Stats ****************************\

create table APP.Stats
   (
      query varchar(40) not null,
      headhunter varchar(40) not null,
      result varchar(40) not null,
      time int not null
   );

alter table app.Stats add constraint qm_query unique (query);

-- \************* App: Results ****************************\

create table APP.Results
   (
      query int not null,
      headhunter varchar(40) not null,
      componentName varchar(40) not null,
      ComponentType varchar(40) not null
   );

-- alter table app.Results add constraint qm_name_type unique (componentName);

-- \************* App: Policy ****************************\

create table APP.Policy
   (
      entity varchar(40) not null,
      machine varchar(40) not null,
      rule varchar(40) not null
   );
```

Figure C.7. Part of database setup script of the proURDS



```
90      * Important Method : This is where the whole matching process will be handled
91      * @param queryString
92      * @return ContractList
93      */
94     public List<Contract> executeQuery(ContractQueryImpl query){
95         //Implementation of All Levels of Matching //General //Syntax // Semantics // QOS Etc ..
96         // Obtain the Query Configuration
97         SSQueryConfig queryConfig = query.getQueryConfig();
98
99         List<Contract>  resultContractArrayList = new ArrayList<Contract>();
100
101        ////////////////////////////////////////////////////////////////////////
102        //First Level of Matching  (Type Matching)
103        ////////////////////////////////////////////////////////////////////////
104        //Basic level of matching happens with the name and Type of the Component
105        for (Iterator<Contract> iterator = contractList.iterator(); iterator.hasNext();) {
106            Contract contract = (Contract) iterator.next();
107
108            // Implement the Exact Name Matching
109            if (contract.getComponentName().equals(query.getComponentName())) {
110                resultContractArrayList.add(contract);
111            // Implementing the Exact Type Matching
112            }else if (contract.getComponentType().equals(query.getComponentType())){
113                resultContractArrayList.add(contract);
114            // Implementing Relax Type Matching
115            }else if (queryConfig.isRelaxedTypeMatchingEnabled() == true){
116                if (queryConfig.isTypeSynonymsMatchingEnabled() == true) {
117                    //Calculate Compatible Synonyms Types from KB
118                    List<String> compatibleTypes = KnowladgeBase.
119                        queryKnowladgeBaseTypeSynonyms(query.getComponentType());
120                    for (Iterator<String> iterator2 = compatibleTypes.iterator(); iterator2.hasNext();) {
121                        String kbType = (String) iterator2.next();
122                        if (contract.getComponentType().equals(kbType)) {
123                            resultContractArrayList.add(contract);
124                        }
125                    }
126                }else if(queryConfig.isTypeInheritanceMatchingEnabled() == true) {
127                    //Calculate Compatible Inheritance Types from KB
128                    List<String> compatibleTypes = KnowladgeBase.
129                        queryKnowladgeBaseTypeInheritance(query.getComponentType());
130                    for (Iterator<String> iterator2 = compatibleTypes.iterator(); iterator2.hasNext();) {
```

Figure C.8. Partial code of the matching algorithm which is performed by the HHs

```
<%@page import="trss.monitor.SSConstants"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql"%>

<HTML>
    <HEAD>
        <TITLE><%=SSConstants.WebAppName%> Monitor</TITLE>
        <style type="text/css" media="all">@import url("./css/monitor.css");</style>
    </HEAD>
    <BODY>
        <p align="center"><a href="query.jsp"><H4>Try Another Query</H4></a></p>
        <br/>
        <H4>Query Result Page : (Query Can be Automatic or Manual)</H4>
        <br/>
        <c:if test="${queryResult != null}">
            <br/>
            <strong>Query Results :</strong>
            <br/>
            <p>Contract Details</p>
                <table width="100%" border="0" bordercolor="#FFFFFF" class="body">
                    <tr>
                       <td width="12%" bgcolor="#FFFFFF"><b>Query ID</b></td>
                        <td width="12%" bgcolor="#FFFFFF"><b>HeadHunter</b></td>
                        <td width="12%" bgcolor="#FFFFFF"><b>Component Name</b></td>
                        <td width="12%" bgcolor="#FFFFFF"><b>Component Type</b></td>
                    </tr>
                        <sql:query var="components">
                            select query,headhunter,componentName,componentType from APP.RESULTS
                        </sql:query>
                        <c:forEach var="component" items="${components.rows}">
                            <tr>
                                <td width="12%" bgcolor="#FFFFFF">${component.query}</td>
                                <td width="12%" bgcolor="#FFFFFF">${component.headhunter}</td>
                                <td width="12%" bgcolor="#FFFFFF">${component.componentName}</td>
                                <td width="12%" bgcolor="#FFFFFF">${component.componentType}</td>
                            </tr>
                        </c:forEach>
                </table>
        </c:if>

    </BODY>
</HTML>
```

Figure C.9. Part of the code base of a jsp page which displays the matching results for different queries

```xml
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_ID" version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/
    <display-name>proURDSMonitor</display-name>

    <servlet>
        <description></description>
        <display-name>LoginServlet</display-name>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>trss.monitor.LoginServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>/LoginServlet</url-pattern>
    </servlet-mapping>
    <servlet>
        <description></description>
        <display-name>SSServerServlet</display-name>
        <servlet-name>SSServerServlet</servlet-name>
        <servlet-class>trss.monitor.SSServerServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SSServerServlet</servlet-name>
        <url-pattern>/SSServerServlet</url-pattern>
    </servlet-mapping>
    <servlet>
        <description></description>
        <display-name>SSQueryServlet</display-name>
        <servlet-name>SSQueryServlet</servlet-name>
        <servlet-class>trss.monitor.SSQueryServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>SSQueryServlet</servlet-name>
        <url-pattern>/SSQueryServlet</url-pattern>
    </servlet-mapping>
    <servlet>
        <description></description>
        <display-name>CheckLANServlet</display-name>
        <servlet-name>CheckLANServlet</servlet-name>
        <servlet-class>trss.monitor.CheckLANServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>CheckLANServlet</servlet-name>
        <url-pattern>/CheckLANServlet</url-pattern>
    </servlet-mapping>
    <servlet>
        <description></description>
        <display-name>SSStatisticsServlet</display-name>
        <servlet-name>SSStatisticsServlet</servlet-name>
        <servlet-class>trss.monitor.SSStatisticsServlet</servlet-class>
    </servlet>
```

Figure C.10. Part of the deployment script (web.xml) of the servlet container

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>seds</groupId>
    <artifactId>proURDSServiceSelection</artifactId>
    <packaging>jar</packaging>
    <version>0.1-SNAPSHOT</version>
    <name>proURDS Service Selection</name>
    <url>http://www.cs.iupui.edu/uniFrame</url>

    <repositories>
        <repository>
            <id>global</id>
            <url>http://repo1.maven.org/maven2</url>
        </repository>
        <repository>
            <id>swt-repo</id>
            <url>https://swt-repo.googlecode.com/svn/repo/</url>
        </repository>
    </repositories>

    <build>
        <directory>target</directory>
        <outputDirectory>target/classes</outputDirectory>
        <finalName>${project.artifactId}-${project.version}</finalName
        <testOutputDirectory>target/test-classes</testOutputDirectory>
        <sourceDirectory>src</sourceDirectory>
        <scriptSourceDirectory>scripts</scriptSourceDirectory>
        <testSourceDirectory>test</testSourceDirectory>
        <resources>
            <resource>
                <directory>resources</directory>
            </resource>
        </resources>
        <testResources>
            <testResource>
                <directory>/resources</directory>
            </testResource>
        </testResources>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <source>1.7</source>
                        <target>1.7</target>
                    </configuration>
                </plugin>
            </plugins>
```

Figure C.11. Part of Maven 2 build script of the proURDS project