1-1-2010

A User-Centric QoS-Based Web Service Selection Framework

Delnavaz Mobedpour
Ryerson University

A USER-CENTRIC QOS-BASED WEB SERVICE SELECTION FRAMEWORK

by

Delnavaz Mobedpour

B.E. in Software Engineering, Shahid Beheshti University, Iran, 2001

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Computer Science

In the program of

Computer Science

Toronto, Ontario, Canada, 2010

AUTHOR'S DECLARATION

I hereby declare that I am the sole author of this thesis.
I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

DELNAVAZ MOBEDPOUR

Signature Date

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

DELNAVAZ MOBEDPOUR

Signature Date

A USER-CENTRIC QOS-BASED WEB SERVICE SELECTION FRAMEWORK

Delnavaz Mobedpour
Master of Science, Computer Science, 2010
Ryerson University

ABSTRACT

With the proliferation of web services, the selection process, especially the one based on the non-functional properties (e.g. Quality of Service – QoS attributes) has become a more and more important step to help requestors locate a desired service. There have been many research works proposing various QoS description languages and selection models. However, the end user is not generally the focal point of their designs and the user support is either missing or lacking in these systems. The QoS language sometimes is not flexible enough to accommodate users' various requirements and is too complex so that it puts extra burden on users. In order to solve this problem, in this thesis we design a more expressive and flexible QoS query language (QQL) targeted for non-expert users, together with the user support on formulating queries and understanding services in the registry. An enhanced selection model based on Mixed Integer Programming (MIP) is also proposed to handle the QQL queries. We performed experiments with a real QoS dataset to show the effectiveness of our framework.

ACKNOWLEDGEMENTS

Pursuing my Master degree was an interesting experiment that was not possible without the guidance and support from many people. First of all I want to thank my supervisor Dr. Cherie Ding, who supported and guided me through all the obstacles I had. Her constant support motivated me beyond imagination and her precious guidance and suggestions have greatly and majorly contributed to this research.

Furthermore, I would like to acknowledge the support of the Computer Science Department of Ryerson University. Especially I want to pay my respect and appreciation to my committee members, Dr. Abhari, Dr. Soutchanski and Dr. Woungang for their valuable guidance. Also I want to thank my friends Elmira, Kian, and Shermineh, who helped and encouraged me a lot.

Finally, I would like to thank my husband, Rouzbeh Alibeik, for the invaluable love and support that he has unconditionally given to me during these last two years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

AHP: Analytic Hierarchy Process

CP: Constraint Programming

CSP: Constraint Satisfaction problem

DL: Description Logic

FMCDM: Fuzzy Multiple Criteria Decision Making

IR engine: Information Retrieval engine

MCDM: Multi-Criteria Decision Making

MIP: Mixed Integer Programming

MIPP: Mixed Integer Programming Problem

Ont: Ontology

QoS: Quality of web Services

QQL: QoS query language

OWL: Ontology Web Language

SLA: Service Level Agreement

UI: User Interface

UDDI: Universal Description, Discovery and Integration

URI: Uniform Resource Identifier

URL: Uniform Resource Locator

W3C: World Wide Web Consortium

WSDL: Web Service Description Language

XML: Extensible Markup Language

CHAPTER 1

INTRODUCTION

1.1. General Settings

Based on the W3C (World Wide Web Consortium) definition, a web service is "a software application identified by a URI, whose interface and bindings are capable of being defined, described, and discovered as XML artifacts". "A Web Service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols." [1] This description emphasizes that a web service needs to be defined by its provider, then advertised by its provider or a third party, and afterwards discovered by clients, in addition a web service has the capability to be combined and interacted with other web services to function as a new composite web service, in an Internet standard environment.

Each provider describes its web service functionality in a WSDL (Web Service Description Language) standard file and publishes it on UDDI (Universal Description, Discovery and Integration) repository. A UDDI registry hosted on the web allows various web services to be stored on it and be discovered on the Internet. After deploying the web services on the Internet the major step is web service discovery. The discovery process happens when a user sends a request, through an IR (Information Retrieval) engine, to find services according to his/her functional requirements. The mentioned process will search among the *.wsdl* files in the available UDDI repositories to find the web services that satisfy user's query. However, the discovery process, based on functional requirements, won't be completely accurate without non-functional features (such as availability, response time, etc). Thus there should be QoS (Quality of Service) or non-functional matching in addition to the functional matching.

With more and more web services published online, the selection process is becoming more and more important, particularly the one based on the non-functional or QoS attributes, in regard to assist requestors to find a desired service. In general, there are two types of service requestors – the human users who could be the direct consumers of services or developers who want to use them in more complex application development, or programs (e.g. a service composition engine) which automatically send requests and select services for further tasks. There have been many research works proposing various QoS description languages and selection models. However, the end user is usually not the focal point of their designs and the user support is either missing or lacking in these systems. Without the proper user support, the accuracy of the QoS requests cannot be guaranteed, and without accurate QoS requests, even the best selection model cannot satisfy users' requirements. Therefore, we believe that a service selection system should be user-centric, which is especially crucial for the human-involved service selection.

In our proposed framework we assume that there is a certain way to collect the QoS data and we just simply use the data for further processing. Moreover, we assume that the data we get is reliable and trustworthy. In our experiment, we will use a real QoS dataset, and again we assume that it is trustable and accurate.

1.2. Main Issues in QoS Languages and Selection Algorithms

There are a few major issues in current selection approaches concerning the user support. First of all, many systems assume that users are capable of formulating requests which precisely reflect their QoS requirements. This assumption may not be true due to many reasons. For instance, a user may not have the knowledge about what the realistic QoS values are. With a

request on reliability greater than 98%, there could be zero matching service. But 98% may be a randomly picked number inferring high reliability. Lowering this number by a few percent, we may find some matching services. Because of this kind of difficulty of choosing a right number, it is not reliable for a selection system to assume the accuracy of the QoS requests from users and build its success on top of this assumption. It is very desirable if the selection system can guide users to choose the right QoS values.

Secondly, in many current systems, the user interface design is not a big concern. Different selection models are proposed and then it is assumed that users would have the ability to submit a proper query which works with the model, no matter how complex the query might be. The user may need to have the knowledge on ontology, utility functions, fuzzy membership functions, etc. In reality, many of the users don't have this kind of knowledge. So we should have a not-so-complex query language which is targeted for non-expert users, and a carefully designed interface to help users formulate the query. The interface should provide a lot of assistance tools to guide users making sure they will not feel overwhelmed.

Thirdly, the expressiveness of the QoS query language could be enhanced to allow requestors to define their requirements in a more precise and comprehensive way. For instance, in many papers, the QoS requirement is represented as either a number (e.g. reliability: 98%), or a fuzzy description (e.g. reliability: very good). However, it is very possible that users may have a mixed request –numeric values on some attributes and fuzzy expressions on others. Therefore, the QoS language as well as the selection model should have the ability to support this kind of request.

Another issue we want to address for the query language and the selection model is the lack of support for defining relaxation policies, e.g. which quality attribute should be relaxed first if

services cannot satisfy all of the soft constraints. Usually the order and degree of relaxation is decided by users' preferences on those attributes, which might not be true all the time. For instance, a user cares about price more than reliability if both requirements can be satisfied. However, if none of the services can satisfy both of those, when relaxing the constraints, because it is a critical task, the reliability just cannot be sacrificed or only to a small extent, then the price constraint should be relaxed first and more. From this example we could see that it is necessary to define a separate preference order and relaxation order, which is lacking in many current works.

The final issue we want to address is the missing time dimension in many QoS languages, e.g. time to invoke the service, and for how long the service will be used. If a service is supposed to be used in future, then the selection should be based on the predicted future QoS values. Or if a service is to be used for a long period of time, then the history record of this service, and its consistency, stability or reputation would be more important factors to be considered. If the time dimension could be added to queries, the selection model could be more properly designed.

1.3. Main Contributions

The main purpose of this thesis is to solve the above mentioned problems. We want to propose a more expressive and powerful QoS query language targeted for non-expert users, together with the user support on formulating queries and understanding services in the registry. Our goal is to achieve a lower cognitive overload on users and in the mean time more options for users to express their QoS requirements accurately, so that the selection process afterwards could be more accurate and the result will be more satisfying.

Furthermore, we propose an enhanced selection model based on Mixed Integer Programming (MIP) [2]. The reason for choosing MIP model is because of the existing problems in the ontology and constraint programming approaches. The former has performance problems and the latter has accuracy problems, in addition to not handling the over-constrained queries. Moreover, in MIP, we can manage continuous, discrete and enumerated variables. In this thesis, we build our selection model based on the original MIP algorithm [2], and we modify it in a way that can handle relaxation policies per variables based on user defined relaxation orders, and provide informative results in two categories of super-exact matches and partial matches. Our selection model could also handle the fuzzy requirements using a clustering-based approach.

Our main contributions are three-fold: we define a more consumer (i.e. non-expert user) oriented QoS query language with the support on various useful features such as a separate relaxation order, fuzzy requirements and time dimension, we design a selection system interface with the guided user support, and finally we propose an enhanced MIP selection algorithm supportive of our language's new features.

1.4. Outline of Thesis

The rest of the thesis is organized as follows.

Chapter 2 gives a review on the related works, including various QoS languages both semantic and syntactic based, then we go through major QoS-based service selection models such as Description Logic based reasoning, Constraint Programming , Mixed Integer Programming, Multi-Criteria Decision Making (MCDM), etc.

We explain QQL – our QoS query language in terms of elements and operators, and user interface design for the guided query formulation process in Chapter 3. Moreover, we use a

query example to explain the validation process. After that, we present the architecture of our user-centric selection system, and we also discuss how to process the newly proposed query features such as fuzzy values and separate relaxation and preference orders. Next, we explain our enhanced selection algorithm in terms of matching, relaxing and ranking methods. At the end of the chapter, we illustrate our algorithm with a step-by-step case study.

In Chapter 4, we explain our actual system implementation and how we handle those new features. In the experiment part, we provide some sample queries and their results. Then we present the evaluations of our proposed selection algorithm compared with the simple MIP selection method in terms of their efficiencies and accuracies.

Finally, Chapter 5 concludes the thesis and presents future works for further exploration.

CHAPTER 2
RELATED WORKS

Since there is no well accepted standard yet, in different research works, various types of QoS languages have been defined. In general, there are two streams of approaches for QoS specification and QoS-based service selection – one is built on semantic web technologies, and the other is non-semantic based. In this chapter, we will first review QoS description languages, and then provide an overview of the selection models under both categories.

2.1. Review on Web Service QoS Languages

Ontology is a description of existing concepts and entities and their relationship in a particular domain, with its specific rules. It is usually closely connected with the semantic web technology. The first category of QoS language is semantic-based and it is often called QoS ontology, such as DAML-QoS [3], QoSOnt [4], WS-QoS [5], OWL-Q [6], onQoS [7], WS-QoSOnto [8], QoS ontology defined in [9] [10], etc. A few common QoS properties include name, category, data type, unit, scale, tendency, relationship, metric, priority, etc. Some ontology supports even more, for instance, the dynamic attributes which are not fixed values and are context-related [10], or the composite QoS attributes in which a few QoS attributes can be combined with a given function [12]. As for the data types, the work in [8] supports a fairly complete list consisting of single value types (Boolean, string, numeric, and enumeration) and multiple value types (range, set, list, and vector). It also defines comparison rules for each type. When defining the constraints on these QoS attributes, the users could specify them as either

hard (compulsory) or soft (optional) constraints, different operators can be used on different data types, and both linear and non-linear constraints can be specified [6] [10] [8].

In DAML-QoS ontology [3], three layers exist, the QoS profile layer includes matchmaking process, the QoS property definition layer contains domain and constraint definitions, and the metrics layer has QoS metrics specifications. In this ontology, the user can define his/her request of QoS constraints in the QoS profile layer. Despite supporting quality levels and roles for requesters and providers, the DAML-QoS ontology could not support QoS tendency, preferences and differentiation between hard or soft constraints. Also, the QoS metric model is not very strong.

In [13], the proposed ontology concentrates on a fairly comprehensive list of terms for defining QoS features of a web service. However, the support for metrics and value types definition is not very strong, and the mentioned conversion method is working only between units that are not metrics.

The onQoS ontology [14] supports different aspect of QoS features such as metrics, metric conversion, value types, and QoS, except that it does not support QoS features such as tendency, unit, dynamic discovery, QoS relationship, QoS preferences, QoS mandatory, and QoS quality levels.

QoSOnt [4] has been proposed for the purpose of developing service centric systems. This ontology has three main levels: unit layer, attribute layer and domain layer. The SQRM is a graphical tool in QoSOnt [4] for stating QoS queries which is hardly user supportive. In this proposal the upper ontology contains QoS vocabulary and concepts; the middle ontology includes definition of QoS aspects about distributed systems. This ontology has similar point of view as that discussed in [14], except it is not supporting relationships between QoS attributes.

Moreover, the conversion method has usage only for QoS units, not QoS metrics or mapping QoS properties.

The QoS-MO ontology [15] supports multiple quality levels, interdependent QoS requirements between providers and requesters, though gives a weak support for QoS units, value types, metrics, and preferences.

The OWL-Q ontology in [6] is an upper ontology that extends OWL-S by using the semantic QoS metric matching, resulting in more relevant offers by just applying syntactic matching. OWL-Q has a fairly complete list for QoS features, except tendency and the QoS properties' usage support.

The designed ontology in [8] supports defining great details for QoS features at various levels. It is by far the most comprehensive one according to our knowledge.

The proposed Policy Centered Meta-model (PCM) for QoS features of web services in [10] is based on the clear differences between requestors' and providers' NFP (Non-Functional Properties), in which policies expressing NFP specifications should be aggregated to one entity under an applicable criteria, and a list of NFP constraints operators. This model is described using a BNF (Backus Normal Form) syntax in which semantics comes from an ontology based on OWL-DL and WSML (Web Service Modeling Language).

The second category of QoS languages is syntactic based. They are usually XML based and include both hard and soft quality constraints.

QRL [19] supports the temporal-aware requirement, e.g. availability > 98% during working hours, > 90% otherwise. This language is based on constraint programming in which each constraint is checked by a constraint solver. Thus, there is no need to write a separate method to match the temporal awareness of the request and offers. This is one of the few QoS languages

which consider time factors. However, it is different from our time dimension which considers when to invoke the service, and for how long the service will be used. Moreover, our time feature will be used in selecting offers based on predicted QoS attributes.

The QoS model proposed in [17] is more from a consumer's perspective, in which QoS is measured by the difference between the perceived and expected quality. The model also introduces a compensation factor for unsatisfied quality requirements and briefly mentions about the importance of the temporal dimension.

The QoS model used in [18] is based on the UML QoS Framework, with an extension on defining priorities between QoS attributes and dimensions. Q-WSDL [16] is a QoS extension to WSDL and is considered as a meta-model.

onQoS-QL [7] is a QoS query language defined based on the QoS ontology onQoS. Users can express a subjective, personalized and contextualized way to evaluate a service on selected QoS parameters and get an aggregated QoS overall value. A QoS query is essentially a few predicates combined with aggregation functions. Defining such a query is not easy for non-expert users. However, the system didn't provide a graphical interface for users to specify queries.

A few of these papers also reported their user interface designs. The work in [5] implemented a WS-QoS editor, which can be used by both service providers and requestors to specify their QoS requirements or offers without knowing the underlying XML schema. This schema includes three types of elements: the *SQoSRequirementDefinition* element representing user's requirements; the *WSQoSOfferDefinition* element that shows QoSattributes of the offered web services; and finally the *WSQoSOntology* element that contains user (a client or service provider) defined QoS parameters and necessary protocol references.

The work in [4] also provides a graphical tool to help users edit their QoS requirements which are visualized as a tree, with QoS attributes as leaves and comparison operators as nodes. Accordingly the matchmaking algorithm traverses the requirement tree from bottom leaves to upper ones till reaches the root. The evaluated final value of the whole tree would be a Boolean value, either true, which represents matching of the query and offer, or false, that shows otherwise. However, the user support in both systems is very limited.

One problem with these QoS languages is that they don't have enough support for helping the users define their QoS requirements accurately. They usually assume that users would like to spend time on learning their QoS languages and also assume that the QoS specifications fed into their selection systems are accurate. It might be true for providers, but most likely not true for consumers.

Most of these works focus on defining the QoS description languages, which could be used by both providers and consumers. Few works look into the query language itself, which is used by only consumers and should be closely related with the user interface support.

2.2. Review on QoS-based Web Service Selection Methods

The web service matchmaking or selection process checks offered web services to find the ones that satisfy all the user's requirements. There are two types of matchmaking: functional and non-functional. Here, we will mainly review the QoS-based selection methods. One of the important issues in matchmaking is how to find web services that users would like to choose despite their differences from the query. Moreover, the matchmaker algorithm should provide a ranking order to suggest well informed options to the requester.

QoS-based web service selection is usually considered as an optimization problem. Different approaches have been proposed, such as Description Logic (DL) based reasoning, Constraint Programming (CP), Mixed Integer Programming (MIP), Multi-Criteria Decision Making (MCDM), etc.

In [12], the discovery process has three steps: using DL reasoning to guarantee the semantic compatibility, translating QoS conditions into constraints and using CP to find satisfying values, and finally selecting services by optimizing the global utility function.

In [6] a semantic QoS metric matching algorithm is proposed that can match offers and request even if they use similar concepts but different instances from the OWL-Q ontology. This algorithm runs through three steps, firstly it produces CSP (Constraint Satisfaction problem) for the request and each offer, secondly it solves all the produced CSPs with an existing CSP solver, and thirdly it finds only the common metrics in the solution space of the request and offer, afterwards the algorithm needs to find out if every solution of the offer exists in the solution space of the request and returns the matched ones.

In [2], the semantic QoS description is transformed into MIP problems, and then a MIP engine is exploited for matchmaking. MIP is a method of mathematical optimization in which the problem's specifications are coded with some variables, and constraints and an objective function are to be minimized or maximized. MIP approach is proved to perform better than CP according to their experiment.

Sometimes QoS is represented as a vector for each service, and then the matching and ranking are based on distance or similarity measurement, or a weighted sum of all attributes [11] [20]. In [20], the matching algorithm compares the lower and upper bound values of each QoS attributes of numeric data types one by one, if all of the offered web service's QoS attributes

satisfy the requirement , that offer will be selected. In case of boolean type, it simply checks if they are equal or not. This paper did not consider any QoS attributes with fuzzy values. The ranking part models QoS data in a vector and sorts the services based on aggregation of consumer's given weight to each attribute. The selection algorithm in [11] chooses the offers according to user's constraints and arranges them in a matrix where each row shows a service and each column represents a constraint. Based on this matrix, all the offers will be ranked after a procedure of normalization and distance calculation for each service.

Outranking algorithm (a type of MCDM techniques) is applied to evaluate and trade-off between alternative services based on their QoS priorities in [18]. Outranking method introduces a global priority constraint in the selection algorithm that can be used like an ordinary constraint. With this global priority constraint the priority of QoS features could be defined relatively. The outranking methods evaluate each offer according to a list of conditions which can be decision maker's priorities or other problem specifications. The PROMETHEE [23] class that has been used in the outranking methods compares each two alternative offers at a time which results in the larger deviation offers as the higher preferred ones.

The selection model in [8] is based on AHP (Analytic Hierarchy Process) – another type of MCDM methods. AHP has three connected steps: decomposing the problem, comparative decisions and synthesizing priorities. The generated hierarchy includes the ultimate goal, conditions and their sub-conditions, and different alternative solutions. The QoS ranking problem can be expressed as a MCDM problem , since there is a process of ranking different web services (alternative solutions in AHP) based on their QoS attributes, in compare with the user's query (ultimate goal in AHP) which contains various QoS constraints (conditions in AHP). The flow of the selection algorithm in [8] is as the following: In the first step, a hierarchy

is designed by the decision makers for the problem at hand, and the offers are ranked based on their QoS values. In the second step, in order to find the relative priorities, the elements of each pair of two conditions are compared. Then for each two conditions, their solutions are compared to identify their relative local rank. In the third step, to calculate the whole rank of each solution, the relative local ranks of all conditions are aggregated.

Another category of the QoS-based service selection methods [24] is based on fuzzy theory. Usually in these approaches, QoS criteria can be categorized into a few groups such as "very poor, medium poor, poor, medium, good, medium good, very good" [26]. Then, a group of evaluators should assess web services based on those QoS criteria. By using different fuzzy set models, the final fuzzy ranking values can be calculated. In [26], selecting web services has modeled as FMCDM (Fuzzy Multiple Criteria Decision Making) with three types of weights: objective (*reliability of evaluation*) and subjective (users' preferences) weights and a new synthetic weight that combines and balances them together. Fuzzy numbers are used to represent subjective weights and entropy values that show the average amount of information quantity of each QoS attribute, or objective weights, which are used to develop the consistency of decision making. A fuzzy decision making model is used in [25] to express the users' inaccurate preferences, then calculate the weight of each QoS condition based on LEM (Linguistic Entropy Method), and finally select the relevant offers according to the query. LEM is a new method based on traditional entropy weighting technique that, by using fuzzy logic, prioritized each QoS attribute according to user's preferences and confidence level. The major issues with these fuzzy ranking models include a high requirement on user evaluation efforts, the subjective nature and possible untruthful evaluations, a requirement for users to define fuzzy numbers or membership functions, and an ignorance of crisp form data types.

2.3. Summary

In this chapter, we reviewed QoS description languages both semantic and syntactic based. Semantic description languages are commonly known as ontology based descriptions and are proposed through various research works such as DAML-QoS, QoSOnt, OWL-Q, etc. Syntactic description languages are usually XML-based and require less processing time.

Furthermore, we reviewed different selection models for both semantic and syntactic QoS languages. QoS-based web service selection is usually considered as an optimization problem and different optimization approaches have been used such as CP [6], MIP [2], MCDM [12], etc.

Based on these reviews we could identify a few issues which should be addressed, such as users' incapability of formulating requests which precisely reflect their QoS requirements, the insufficiency of the user interface design, lack of the expressiveness of the QoS query language in case of a mixed request on numeric values and fuzzy expressions, lack of support for defining independent relaxation policies from preference orders, and finally the missing time dimension in many QoS languages. Therefore in our work we tried to address all these issues by the following solutions: we define a more consumer oriented QoS query language with the support on various useful features such as a separate relaxation order, fuzzy requirements and time dimension; we design a selection system interface with the guided user support; and at last we propose an enhanced MIP selection algorithm supportive of our language's new features.

CHAPTER 3

QOS-BASED WEB SERVICE SELECTION FRAMEWORK

3.1. QQL – Our User-Centric QoS Query Language

3.1.1. The Language Definition

Our main interest of defining a QoS language is to add a few properties which we feel are necessary to express the real user requirement in a more accurate and comprehensive way and are missing in the current languages. These properties can be added to an existing QoS language or included in a new language. Here, we take the second approach – design a new language, in which we also include many of the commonly supported properties. When we compare the semantic and syntactic approaches of defining QoS languages, semantic languages are usually complex and time consuming and the subsequent selection models may suffer from the performance problems, whereas the syntactic models may have a low accuracy due to the mismatching vocabulary or metrics [6]. In the current stage, we choose the syntactic approach. Nonetheless, the same properties can be added into QoS ontology in a similar way.

In our proposed language, a QoS query can be represented as a 6-tuple:

QQL Query : <qID, uID, sbTime, timeConstraints, qosConstraints, dataSource>

The first three components represent the query ID, user ID and the submission time respectively. These components are mainly used for the logging purpose so that it is possible to run a query-log mining process in future.

Time constraint itself includes four elements as defined below:

timeConstraints : < *Invocation start date, invocation end date, duration of usage, frequency of usage* >

Invocation start date is defined as the date when the service is first invoked, *invocation end date* is defined as the date when the service is last invoked, *duration of usage* is defined as how long the service is used in each invocation (e.g. a stock price service could be used for 8 hours per invocation, whereas a currency exchange service may be used only once), and *frequency of usage* is defined as how often the service is invoked during the period from start date to end date (e.g. 5 times per week). It should be noticed that these are only estimated values based on user's expectation during the query time, and the actual values signed in the final contract may not be the same. By providing a time dimension in the query language, users are capable of defining a possible usage pattern, which could give more information for a more accurate selection in the later stage. All these time constraints are optional. If a user doesn't have such a requirement, or enough knowledge of specifying it accurately, it could be left blank and the selection would be based on the current QoS values. If a user doesn't specify all of the time constraints, then the prediction would be based on only the available ones.

dataSource defines which data source the selection process relies on. There are three common sources to get the service QoS data – descriptions published by providers, SLAs (Service Level Agreement) signed between requestors and providers, and the actual monitored data from each invocation instance. The published QoS description is usually available and thus is the default data source. The other two, the SLAs or monitored data, may not be available, and they often indicate a higher level of complexity of the selection algorithm and a longer processing time. We keep this property for future usage, and for now we just take the default value.

QoS constraints include constraints on all *N* QoS attributes the system supports, which can be represented as an *N*-tuple:

qosConstraints : <$qosConstraint_1$, $qosConstraint_2$, …, $qosConstraint_N$>

If a user doesn't have a concern on a QoS attribute, the corresponding constraint will be empty. For any non-empty QoS constraint, it can be further described by an 8-tuple as:

$qosConstraint_i$: <*name, type, unit, tendency, preference order, relaxation order, weight, values*>

Name defines the name of the attribute, e.g. reliability, response time. *Type* refers to the data type of the attribute, and currently we support Boolean, string, enumeration, numeric, and fuzzy types. *Unit* defines the measurement unit of the attribute, e.g. millisecond for response time. The conversion will be done automatically between the supported units. *Tendency* represents user's expectation on the attribute values, positive tendency means a higher value is preferred, and negative tendency means a lower value is preferred. It is usually a predefined value depending on the data type, e.g. tendency for response time is negative, and tendency for reliability is positive.

Preference order defines the order of user preference on each attribute. If there are *N* attributes, the value range of the preference would be between 0 to *N*, with 1 referring to the most preferred attribute and a bigger value referring to a less preferred attribute. It is possible that a user might assign a same preference value to different attributes. If the preference value on an attribute is zero, it means that user doesn't have a concern on this attribute and it should not be checked during the selection process. *Relaxation order* defines the order of relaxation when there has to be a trade-off among different quality attributes. Its value range is also between 0 and *N*, with 1 referring to the attribute whose value should be first relaxed, and a bigger value referring to an attribute which should be relaxed later. There could be multiple attributes having

a same relaxation order value, and zero means relaxation cannot be done on this attribute (i.e. a hard constraint). As we mentioned before, relaxation order could be unrelated to the preference order. However, if this value is not specified in a query, by default, it is opposite to the preference order, e.g. the most preferred attribute will be relaxed the last. By defining the relaxation order, we can differentiate the soft and hard constraints, and furthermore define how we want to deal with soft constraints. *Weight* measures the preference level of each attribute. It is not defined by the user. It is automatically converted from the preference order by normalizing its value to (0, 1) range.

Values define the user requirement on the attribute values. Our query language supports three types of value representation. Boolean, string and enumeration types are represented as a single value. Numeric type is always represented as a range. The reason is that most of the constraints on numeric values could be defined as a range, e.g. "response time < 2 second" could be represented as (0, 2), "reliability > 95%" could be rewritten as (95%, 100%), etc. For fuzzy type, depending on the granularity level we want to achieve, we could define different linguistic expressions. One example could be "good", "medium", and "poor". It also supports this particular expression – "best available", because we believe it is a common requirement in many users' minds.

Definitely, this property list can be further expanded with good features from those existing languages, e.g. the validity period [19], the composite attribute [12], etc.

Regarding the QoS attributes, there have been a few research efforts such as [27] of defining a quite complete list of them and grouping them into different categories. Our language could support all of them. However, in the later examples, we will only use a few. Figure 3.1 below shows the hierarchical relationship between different language elements.

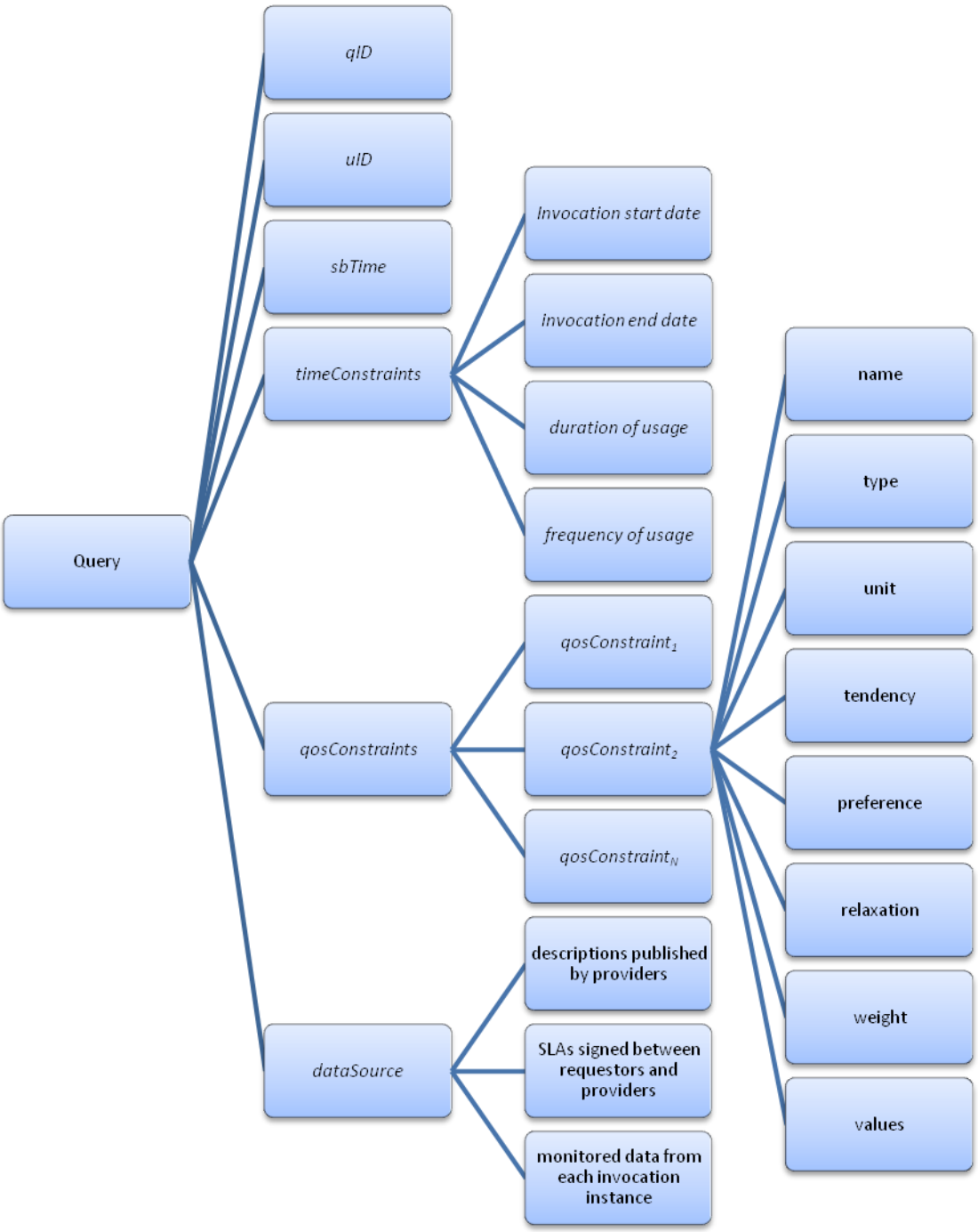

Figure 3.1- The hierarchy of the QQL language elements

Figure 3.2 shows a sample query in accordance with our proposed query language. In the proposed framework an XML document is generated for each query. The root element of the

document is a <*QoSQuery*> that includes at least one <*qID*>, <*uID*>, <*sbTime*>, <*dataSource*> element, and may have zero or more <*timeConstraints*> or < *qosConstraints*> elements.

```
<?xml version="1.0" encoding="utf-8"?>
<QoSQuery>
  <qID>1</qID>
  <uID>10</uID>
  <sbTime>01/10/2010 16:40:40 ET</sbTime>
  <timeConstraints>
     <startDate>02/17/2010</startDate>
     <endDate>07/17/2010</endDate>
     <frequencyOfUsage>
       <value>3</value>
       <unit>week</unit>
     </frequencyOfUsage>
     <durationOfUsage>
       <value>2</value>
       <unit>hours</unit>
     </durationOfUsage>
  </timeConstraints>

<QoSConstraints>
   <QoSConstraint>
    <name>price</name>
    <type>numeric</type>
     <unit>US dollar</unit>
     <tendency>negative</tendency>
    <preference>4</preference>
    < relaxationOrder>0</relaxationOrder>
     <weight>0.3</weight>
     <values type="range" >
      <from>0</from>
      <to>150</to>
     </values>
   </QoSConstraint>
……
   <QoSConstraint>
    <name>reliability</name>
    <type>numeric</type>
     <unit>%</unit>
    <tendency>positive</tendency>
    <preference>3</preference>
    < relaxation>1</relaxation>
     <weight>0.1</weight>
     <values type="fuzzy" >
      <value>good</value>
     </values>
   </QoSConstraint>
  </QoSConstraints>

<dataSource>provider</ dataSource >
</QoSQuery>
```

Figure 3.2- A segment of a sample QQL query

3.1.2. Validation

In our proposed language, the validation process for the original query and offers constraint specifications is using the XML-schema [28] as shown in Figure 3.3. In this schema we describe each element, together with its type and usage in terms of being mandatory or optional.

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="Query_Schema" targetNamespace="http://tempuri.org/Query_Schema.xsd"
elementFormDefault="qualified"
    xmlns="http://tempuri.org/Query_Schema.xsd"
xmlns:mstns="http://tempuri.org/Query_Schema.xsd"
xmlns:xs="http://www.w3.org/2001/XMLSchema" >

  <xs:element name="Query">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="constraints">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="QoSAttribute">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="name" type="xs:positiveInteger" use="required"/>
                      <xs:attribute name="unit" type="xs:string" use="required"/>
                      <xs:attribute name="Type" type="xs:string" use="required"/>
                      <xs:attribute name="tendency" type="xs:string" use="required"/>
                      <xs:attribute name="weight" type="xs:string" />
                      <xs:attribute name="preference" type="xs:int" use="optional"/>
                      <xs:attribute name="relaxation" type="xs:int" use="optional"/>
                      <xs:attribute name="value" type="xs:string" use="optional"/>
                      <xs:complexType>
                        <xs:simpleContent>
                          <xs:attribute name="from" type="xs:positiveInteger" use="optional"/>
                          <xs:attribute name="to" type="xs:positiveInteger" use="optional"/>
                        </xs:simpleContent>
                      </xs:complexType>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>

            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <!-- end of constraints-->
```

Figure 3.3- QQL schema

3.1.3. Operators

In our language, we have two types of operators, between constraints and inside the constraint; the former includes Boolean operators such as *"AND"* and *"OR"*, e.g.

(q.availability> 85% AND q.responsetime < 5 msec); and the latter contains operators like "> *",* *"<"* and "= *",* e.g. (q. availability > 85%)

The Boolean operator *"AND"* is explicitly used between constraints of a query. However, if any QoS attributes have relaxation order greater than zero, the *"OR"* operator will be used implicitly between constraints. For instance, if the query is:

(q. availability > 85% , q.avaialability.relaxation:1;

q.responsetime < 5 msec , q.responsetime.relaxation:2;)

The matchmaking process with these two constraints is described as the following: in the first step, without any relaxation, we gather all the offers that satisfy the *availability* constraint *"AND" response time* constraint; then in the second step, with availability relaxed, we get all the offers that satisfy the *availability* constraint *"OR"* the *response time* constraint; and in the third step, with both availability and reliability relaxed, we collect all the offers either satisfy the *availability* constraint *"OR"* the *response time* constraint *"OR"* none of them.

3.2. The Guided Query Formulation Process

For a user-centric service selection system, the user interface (UI) design is really crucial. On one hand, it should be compatible with our expressive query language, and on the other hand, it shouldn't put too much burden on users to define a complex QoS query. We believe that a guided process is necessary for query formulation and the UI design should facilitate this process.

The following figures show the key interfaces during the query formulation stage. Figure 3.4 illustrates the first step – selecting the QoS attributes and the keyword. The keyword is an item that should be selected in this page to lead the framework to produce XML-based

documents, compatible with our system, for each offer. By defining the keyword, e.g. flight, the user specifies his/her query to be applied to offers that have this particular functionality.

Then there is a list of all supported QoS attributes in the left side of the page. When a user selects an attribute from this list, it will be added to the list on the right side. After the user picks up all the concerned attributes, he/she could move the mouse over each item of the right list to get more information (e.g. definition of the attribute, or a sample request on this attribute). Adding or removing any attributes will be done by pressing the buttons between the lists. By restricting the attribute names to the system supported list, it could solve the vocabulary mismatch problem to a certain extent.

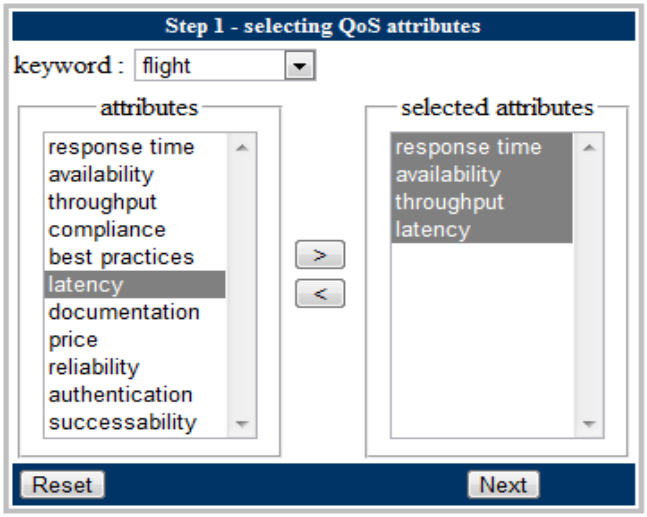

Figure 3.4- Step 1 of query formulation: selecting QoS attributes

The second step – defining the QoS requirement, is illustrated in Figure 3.5. As we discussed before, sometimes users may not have any knowledge about the realistic value ranges on those QoS attributes. In this case, the users could click for example the “Reliability Browse” link, and then a new window will be popped up to show the distribution patterns of the values of this selected QoS attribute, as shown in Figure 3.6. The main purpose of this browsing function is to let users know what QoS values the current services have and what patterns could be

revealed from service clusters, so that users could put down more reasonable values in their queries. The detailed discussion on this function can be found in [29]. In the browsing page the user could find three groups of services clustered based on their values on the selected QoS attribute. By clicking on any of these groups, the actual QoS values (the interval data) of services in that cluster are shown in details in the middle textbox. The summary information for each cluster is shown in the bottom textbox, including the centroid of the cluster, the size, and the closest three services to the center point.

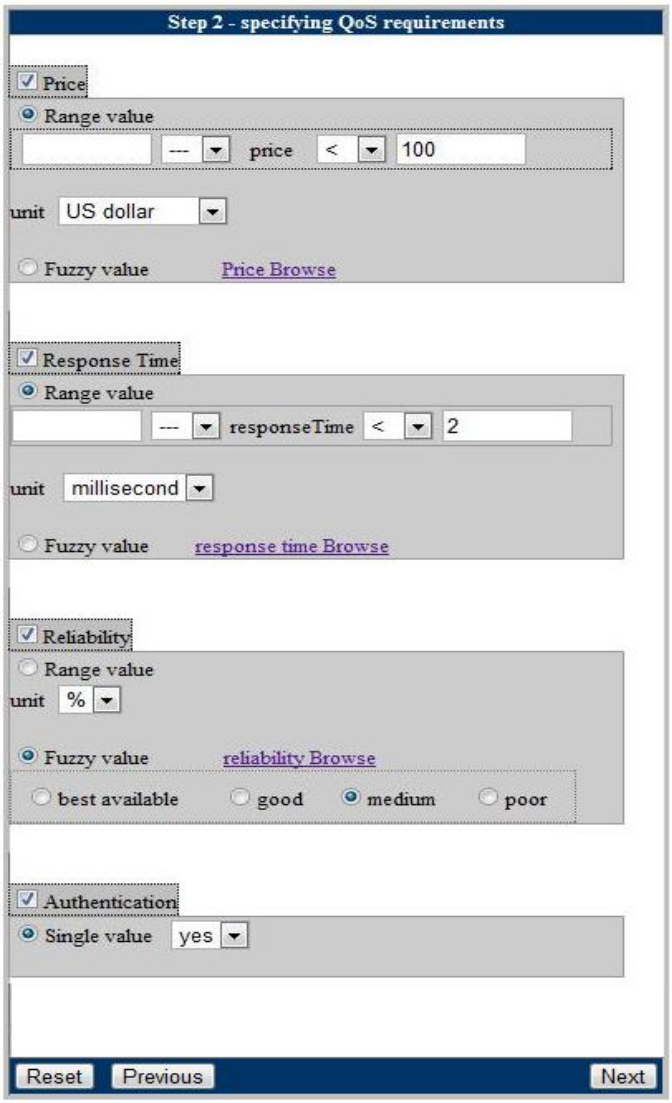

Figure 3.5- Step 2 of query formulation, specifying the QoS requirements

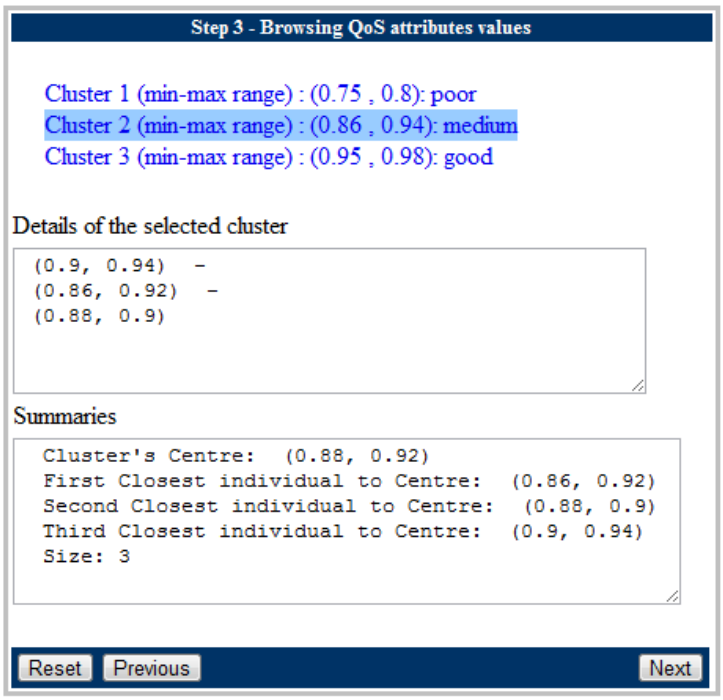

Figure 3.6- Step 3 of query formulation: browsing QoS attribute's values (e.g. reliability)

After users have gained some knowledge on the value ranges and distribution patterns of the selected attributes, for each attribute, they could choose to define their QoS requirement either as a single/range value or as a fuzzy value. Depending on the attribute type, the system decides whether it should be a single or range value. For the range value, if the first textbox is empty, it means the requirement is less than (or equal to) the value specified in the second textbox, and if the second textbox is empty, then it is greater than (or equal to) the value specified in the first textbox. For the single value, again, depending on the type, users could either choose from the valid values supported by the system or enter the free text. If users choose fuzzy values, currently supported values include "best available", "good", "medium" and "poor".

Figure 3.7 shows the fourth step – defining the order of preference and relaxation. For the selected attributes in the first step, users are able to define which attributes they are more concerned about and which attributes should be relaxed first in the service selection stage. For

the unselected attributes, their preference values are set to zero. For the selected attributes, if users leave their preference value fields empty, the values would be set to 1 plus the biggest preference value specified by users. For instance, based on the example input in Figure 3.7, preference for response time is 2, for price 4, for reliability 3, and for authentication 1. For the relaxation order, if an attribute is selected in step 1, but its relaxation order is undefined, it means it is a hard constraint and shouldn't be relaxed. As shown in Figure 3.7, price and response time are hard constraints. Relaxation order is only meaningful when the attribute is selected in step 1.

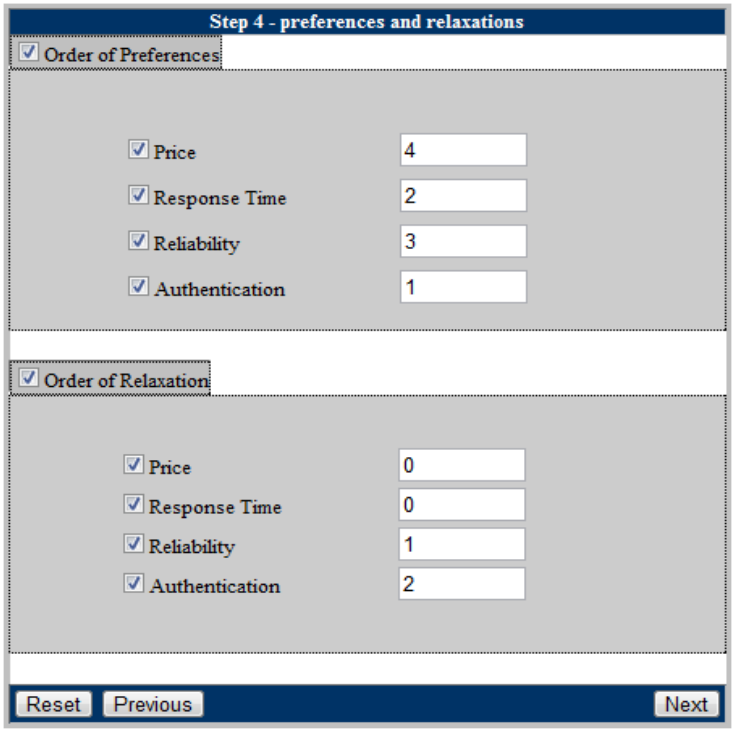

Figure 3.7- Step 4 of query formulation: defining QoS attributes' preferences and relaxations

Figure 3.8 shows the last step – defining the time constraints. When a user performs a search, he/she may want to use this service right away, or sometimes in future. If the service is supposed to be invoked in a future period of time, it would be more accurate if the system could predict the service quality level at the specified future time and make the selection decision

based on the predicted QoS values. The expected usage patterns would also affect how we want to do the selection. This interface is quite straightforward. Users just need to input different time values as requested. If the invocation is actually now, then the box beside Time shouldn't be checked, and the prediction step will be skipped. Or if a user is not sure about the exact invocation frequency or usage duration, he/she could either input an estimated value or leave those fields blank. Prediction is only based on the given inputs.

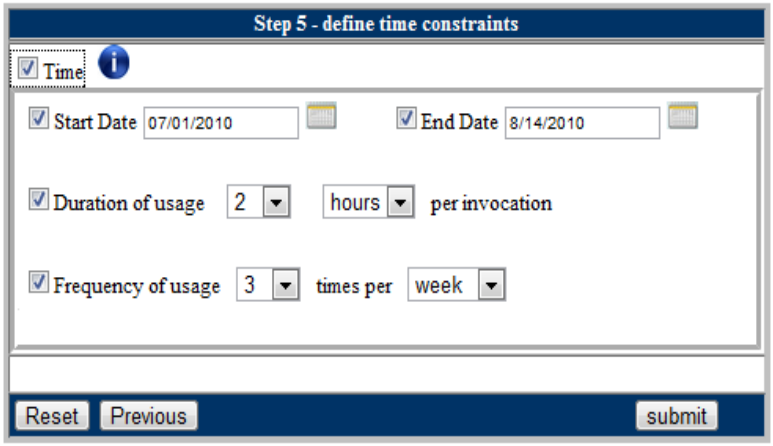

Figure 3.8-Step 5 of query formulation, definition of time constraints

The sample query shown in Figure 3.2 is formulated based on the user input illustrated by Figures 3.4 to 3.8. By introducing this kind of UI design, we could make sure users could get full support when formulating their QoS queries and they also have the freedom of skipping many of the input boxes and take the default settings. It reduces users' cognitive overload and in the mean time provides the capability of submitting an expressive QoS query.

After formulating the query, user will be led to the result page, as shown in Figure 3.9. The first grid shows the query, the second one shows the offers that match the query or are better than the requirements, and the last grid shows offers that partially matches the query. All the results are ranked based on their matching scores.

request and results

Request

Constraint_Id	Attribute Name	Unit	type	tendency	Values	relaxation	preference
0	response time	millisecond	range	negative	from :0 to: 100	2	4
1	availability	%	range	positive	from :75 to: 100	1	2
2	throughput	bite-rate	fuzzy	positive	from :8 to: 50	3	1
3	latency	millisecond	fuzzy	negative	from :0 to: 20	4	3

Results

Super-Exact Matches

rank order	Sname	responsetime	availability	latency	throughput
1	S2	60.05	77	4.1	9.1
2	S1	59.58	77	3.95	8.8
3	S16	61.42	79	5.37	8.9

Partial Matches

rank order	Sname	responsetime	availability	latency	throughput
1	S8	62.05	72	6.05	10.2
2	S12	71.21	71	6	10.7
3	S9	60.84	70	5.95	10.5
4	S7	57.42	66	5.31	12.5
5	S11	64.75	86	7.95	7.8
6	S17	121.15	88	8.2	6.4
7	S4	229.79	79	41.32	7.4
8	S15	58.21	62	4.68	13.3
9	S3	65.4	92	8.25	6.8
10	S14	226.58	90	37.37	7.2
11	S13	243	89	49.11	7
12	S6	247	89	55.1	6.6
13	S10	263.89	87	70.52	7
14	S5	305.37	87	90.95	8

Home

Figure 3.9- The final result page

3.3. The Architecture Model

Having explained our QoS query language and the UI design, in this section we will illustrate how we can use them in a user-centric service selection system. Figure 3.10 shows the

system architecture. We could see that the service requestor is interacting with the system through the user interface, unaware of the complex query language and underlying selection models. When a user chooses to browse the QoS data in a service repository, the browsing component will rely on the clustering component to show the distribution pattern of the QoS values. Browsing step could give a user more confidence on choosing the right values for the query. Afterwards, the user follows the four steps as explained before and enters all the required data, and then the input data will be fed into the QoS query formulating component. The generated XML-based query is then sent to the selection and ranking component. If the user has entered the time constraints, the prediction component will be triggered so that the selection is based on the predicted data instead of the current data. Finally the matching services will be presented to the user as a ranked list. QoS prediction is not studied and implemented in this thesis. However, the prediction algorithms from previous research works [30] can be added into our framework.

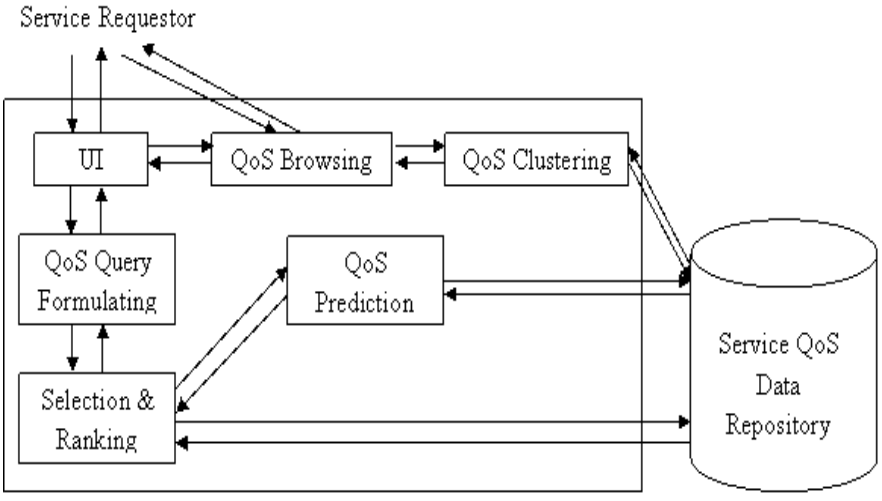

Figure 3.10- An architecture model illustration

3.4. Handling Fuzzy Values

Before we move on to explain our selection model, we first discuss how we process the mixed fuzzy and range requirements. Usually the fuzzy requirement is handled by the fuzzy set theory, and users are required to input their evaluations on the services. We tackle this issue from a different perspective. When a user enters a fuzzy requirement, e.g. reliability: good, it basically means the user doesn't have a specific value to define, as long as the quality level on this attribute is on high end, he/she will be happy. Instead of asking users to evaluate which service has a good or poor reliability, we cluster all services into a few groups on this particular attribute. If we define three quality levels: good, medium and poor, then we cluster the services into three groups. All services fall into one of the three categories. When we define more quality levels, we will have more clusters in the result. Take the most popular clustering algorithm – k-means algorithm [31] as an example, the number of quality levels decides the number of clusters – the value *k*.

Again, we will use an example to illustrate the selection process for mixed type queries. Suppose the QoS query is: (price < \$100, reliability: good, response time: medium). In order to do matchmaking, first, we cluster services into 3 groups (S_{rel}^1, S_{rel}^2, S_{rel}^3) based on reliability, and take the cluster with high end values – S_{rel}^3. Then services are clustered into 3 groups (S_{res}^1, S_{res}^2, S_{res}^3) based on response time, and we take the clusters with high end and medium range values – S_{res}^2, S_{res}^3. The reason we choose 2 clusters is that users always prefer a better quality service, and if users specify the fuzzy requirement as medium, it means both medium and good are acceptable. The next step is to get all services satisfying the requirement on price – S_{price}. The final matching services will be the intersection of the three sets:

$S = S^3_{rel} \cap (S^2_{res} \cup S^3_{res}) \cap S_{price}$. Afterwards, the preference-aware selection models can be used to rank these services.

3.4.1. Clustering

K-means clustering algorithm categorizes the QoS data into *k* groups or clusters. Since in our system for the QoS attributes with fixed values we support single and range values, and single values can be easily converted to range values (e.g. 1 converted to [1, 1]), the data objects in the clustering algorithm are interval vectors representing values of different QoS attributes of web services. It chooses *k* initial centers, and then places each interval data in the group with the closest distance. The distance is calculated by the Euclidean function as shown below:

$$D_E(a, b) = \sum_{i=1}^{n}((a_{il} - b_{il})^2 + (a_{iu} - b_{iu})^2) \qquad (3\text{-}1)$$

Where *a*, *b* represent two QoS vectors $[(a_{1l}, a_{1u}), (a_{2l}, a_{2u}), \ldots, (a_{nl}, a_{nu})]$ and $[(b_{1l}, b_{1u}), (b_{2l}, b_{2u}), \ldots, (b_{nl}, b_{nu})]$, a_{il} is to represent the lower bound of the interval and a_{iu} is to represent the upper bound of the interval, and *n* is the number of attributes the system supports.

After going through all the interval data, the algorithm calculates the new centers for each cluster and re-assigns vectors to clusters again. These steps will be repeated until an adequacy criterion converges to an optimal value. Below is the formula for the adequacy function,

$$E = \sum_{i=1}^{k} \sum_{q \in C_i} {D_E}^2(q, G_i) \qquad (3\text{-}2)$$

Where *k* is the number of clusters, vectors $(G_1, G_2, \ldots, G_k)$ are the centroids of the partition $(C_1, C_2, \ldots, C_k)$, and *q* is a random vector from C_i.

3.5. Selection Algorithm

Ideally, a matchmaking algorithm would result in offers that match both functional and QoS requirements of the user's query. However, in reality returning faultless offers is impossible. One of the problems is how to select the offers that despite their differences from the query, the user might want to choose. Another problem is the ranking part; the matchmaking algorithm should calculate the distance between query and offer in a way that the user can decide clearly which offer to choose. The goal of solving the first problem leads us to design a selection algorithm based on user's defined relaxation and preference orders to find offers that user might choose even though they might not be exact matches. In our matchmaking algorithm, we design two result lists: super-exact matches and partial matches. The former includes offers that satisfy all the user's requirements or have better QoS features than the request. And the latter contains offers that have at least one worse QoS features than the request.

Our proposed algorithm has 3 parts – selection based on functional requirements, selection based on QoS requirements, and ranking part which is the answer to the second issue mentioned earlier. Figure 3.11 shows the detailed architecture inside the QoS selection component.

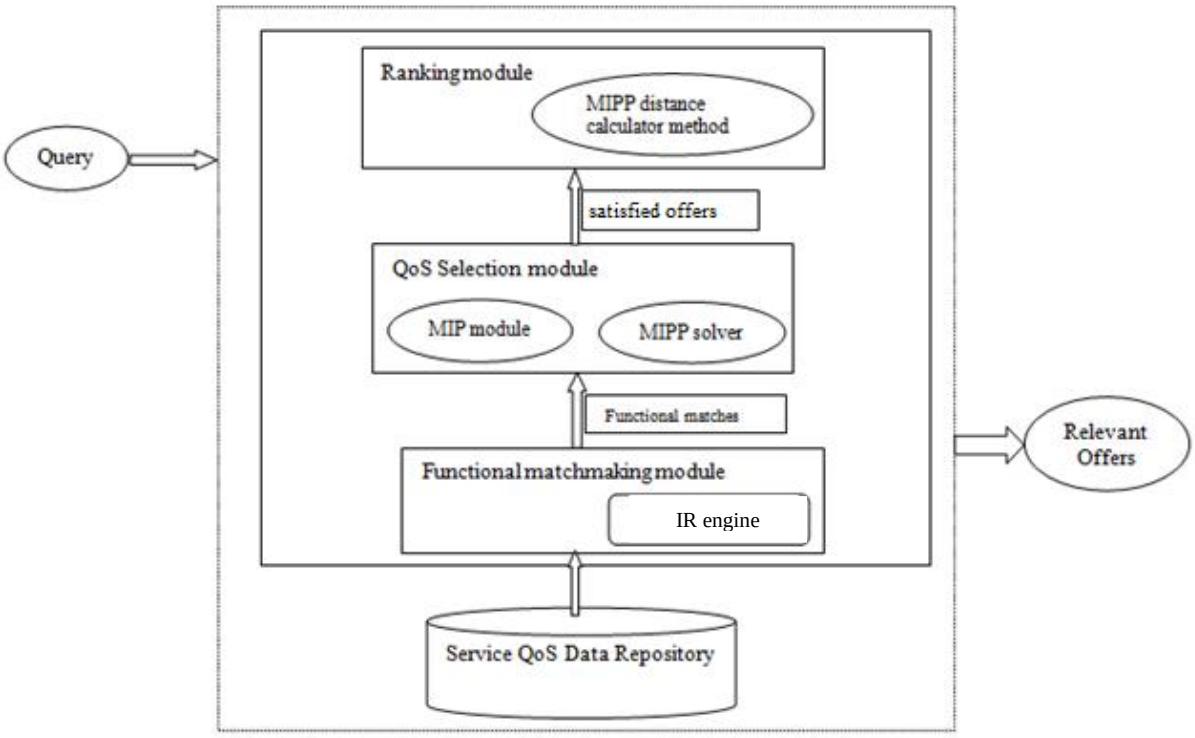

Figure 3.11- Detailed architecture of selection and ranking component

Figure 3.12 shows the flowchart of the selection algorithm:

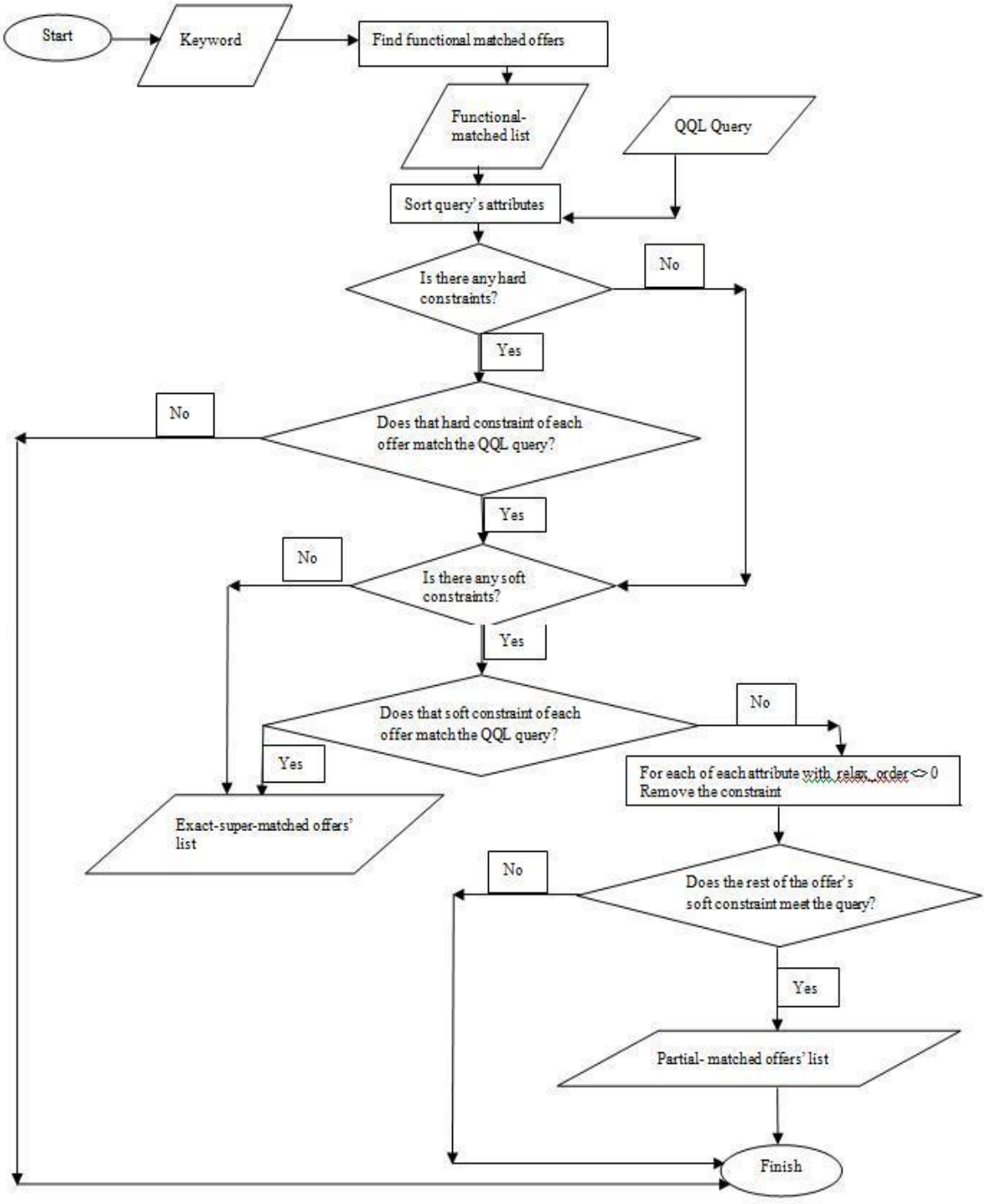

Figure 3.12 - The flowchart of the selection algorithm

3.5.1. Selection based on functional requirements

In order to find functionally matched web services, an IR (Information Retrieval) system has been used to find the offered web services according to user's chosen keywords. The IR system can index all the keywords related to web services in the repository and those keywords can be found from the WSDL files or other sources. Then the system provides an interface to users, in which users can type in a keyword, and then the matching services will be discovered and retrieved. For instance, if we want to find services related to "protein", we just need to type in this term, and then the result is a list of web services that has something to do with protein.

3.5.2. Selection based on QoS requirements

The next step in our algorithm is finding web services that satisfy the QoS requirements. This process will take place on the result list returned from the previous step. In order to explain, the query and offers are represented as:

$$Q = (q_1, q_2, \ldots, q_n) \quad , \quad O = (qs_1, qs_2, \ldots, qs_n) \qquad , i=1,2,\ldots,n$$

where n is the number of quality attributes. For instance if our QQL query is Q = (0.95, 2, 100, true), n would be 4 and q_1 represents the required value on reliability, q_2 refers to the value of response time, q_3 is the value of price and q_4 is the value of authentication. Respectively qs_1, qs_2, qs_3, qs_4 show the values of these QoS attributes from the offers. The below representations show offers in the same format as the query:

$$O_1 = (0.90, 1, 90, \text{true}) \quad , \quad O_2 = (0.90, 1, 90, \text{true})$$

In QoS-based selection part, there are 3 steps: sorting, handling hard constraints, and handling soft constraints.

In sorting step, we sort each attribute q_i of the query Q for the purpose of efficiency. The desired order for different types of attributes is *single*, *range* and *fuzzy* attributes. Since matching

the *single* attribute is the simplest one of the matching process, we sort the attributes in a way to consider the *single* attributes first. The *single* matching process checks the constraints of the offer and query to see if they are equal or not. For example q_4 of the Q is a single attribute which has only two values, true or false; the *single* matching process compares $Q.q_4$ and $O_1.q_4$, since they have the same value the result will be true for this particular attribute of O_1 on q_4. Then range matching process is considered and the unsatisfying services are ruled out so as to leave fewer offers for applying clustering in the *fuzzy* attributes situations.

In the next step, we check the query's hard constraints by going through the sorted list of the functionally matched web services, to make sure each offer will satisfy them. The constraints with $q_i.relax = 0$ are identified as hard constraints and are not allowed to be relaxed. Thus the selection algorithm will go through each offer and select the ones that satisfy all the hard constraints.

In the next step, the algorithm will handle the soft constraints. The soft constraints are identified by non zero relaxation order or $q_i.relax <> 0$. Basically in this step all the offers from the previous step (offers satisfying the hard constraints) are considered for soft constraints matching. Offers satisfying all the soft constraints are added to the super-exact matches list. Then the algorithm goes through the relaxation routine. In this routine the respective constraints based on the relaxation order are removed one by one from the smallest value to the greatest. And all other remaining constraints are checked to find if they satisfy the query requirement. If any offer matching at least one soft constraint is found, it will be added to the partial matched list.

The routine we use to find if each attribute of an offer satisfies the constraint of the query is MIP [2]. We will produce a MIP problem for each attribute of offers and the request. The MIP problem is an optimization problem depending on $q_i.tendency$ in terms of maximization or

minimization. Based on the tendency of an attribute, we can calculate utility function for each q_i and its corresponding preference-aware weight. The following formula shows the utility function of Q_i:

$$uq_i = \frac{Q_i - min}{max - min} \qquad (3\text{-}3)$$

where *min* and *max* in formula (3-3) represents the smallest and greatest value of q_i in the whole functionally matched list of offers. Also Q_i represents a variable that can be either q_i of the query or qs_i of an offer.

The weight parameter for each attribute q_i is calculated as formula (3-4):

$$\omega_i = (max\_pref - qi.pref + 0.1)/(max\_pref - min\_pref) \quad (3\text{-}4)$$

Formula (3-4) basically indicates that the weight parameter is based on preference order of each attribute. max_pref and min_pref variables are the maximum and minimum value of the preferences in the QQL query. Since in some cases max_pref and $qi.pref$ are equal and consequently ω_i is zero, we include a small random value 0.1 in formula to avoid having zero weights. We will continue with producing MIP problems (MIPP) as shown below:

If (q_i.tendency = positive) then

$$MIPP_{q_i} = min\ \omega_i \times uq_i \quad (3\text{-}5) \qquad MIPP_{j_{qs_i}} = \min_j\ \omega_i \times uq_i \quad (3\text{-}6)$$

If (q_i.tendency = negative) then

$$MIPP_{q_i} = max\ \omega_i \times uq_i \quad (3\text{-}7) \qquad MIPP_{j_{qs_i}} = max_j\ \omega_i \times uq_i \quad (3\text{-}8)$$

In these formulas, *j* indicates the total number of offers, and formula (3-6) and (3-8) imply MIPP for all offers and formula (3-5) and (3-7) show MIPP for the QQL query. Depending on

the tendency, MIPP for offers is to find either the maximum or minimum value of the weighted utility functions among all offers, MIPP for query in our case is just one single value.

The MIP problem will be solved by using *lp_solve*. lp_solve is a free open source application [35] for solving linear integer programming optimization problems based on branch and bound techniques. With branch and bound method, the solver can manage integer variables (or single variables), semi-continuous (or range variables) variables and Special Ordered Sets (or enumeration and fuzzy variables). After solving each MIPP, the algorithm decides if each offer satisfies the QQL query according to (3-9) and (3-10). These two formulas show if the MIPPs of the offer and query are met we will have a matching offer to put into the result lists, either in super-exact matches list or in partial matches list. If the matching routine was called for an offer after matching all the hard and soft constraints, the offer will be placed in the super-exact matched list. But if the matching routine was called during the relaxation stage, the offer will be categorized in the partial matched list.

If (q_i.tendency = positive) then *(e.g. reliability)*

If ($MIPP_{j_{qs_i}} >= MIPP_{q_i}$) *then*

Matched-list.add (S_i) (3-9)

If (q_i.tendency = negative) then *(e.g. response time)*

If ($MIPP_{j_{qs_i}} <= MIPP_{q_i}$) *then*

Matched-list.add (S_i) (3-10)

3.5.3. Ranking

After matching and selection we will have two lists including super-exact matched and partial matched offers. The services satisfying all the constraints are ranked together and will be placed in the former list, and then other services that satisfy all the hard constraints and some of the soft constraints will be ranked in the latter list. In order to rank the selected services we continue with already produced MIP problems. First we will calculate *Q.MIPP* for the query and S_j*.MIPP* for each service as an aggregation of all the MIPPs' attributes as shown by formula (3-11). As explained earlier, *j* corresponds to the total number of offers, and *i* represents the number of QoS attributes. Thus in (3-11) S_j*.MIPP* represents MIPP for offers and *Q.MIPP* shows MIPP for the QQL query.

$$S_j.MIPP = \sum MIPP_{j_{qs_i}} \ , \ Q.MIPP = \sum MIPP_{q_i} \quad (3\text{-}11)$$

Then, the difference between S_j*.MIPP* of each offer and *Q.MIPP* of the request will be calculated according to (3-12) and taken as the ranking score for that service. After calculating the scores we can sort the services, from the smallest ranking value to the greatest, and present them to the requester.

$$S_i.\ ranking\_score\ = |\ S_j.MIPP - Q.\ MIPP\ |\quad (3\text{-}12)$$

where S_i*. ranking_score* is the absolute value of the difference between S_j*.MIPP* of an offer and *Q.MIPP*.

3.6. Case Studies of Processing QQL Queries

We will use one example to show the role of the relaxation order in the selection process. Suppose a user selects four concerned QoS attributes in step 1: reliability, response time, price and authentication. Table 3.1 shows a sample QoS requirement (Req.) and 6 QoS offers (s_1 to s_6), alongside with user defined relaxation (Relax.) and preference order (Pref).

Table 3.1- The case study with a sample QoS query and six QoS offers

	q1:Reliability	q2:Response Time	q3:Price	q4:Authentication
Pref.	3	2	1	3
Relax.	0	1	2	3
Q	> 95%	< 2s	< $100	True
S_1	> 95%	< 3s	$95	True
S_2	> 98%	< 1s	$110	True
S_3	> 98%	< 1s	$90	False
S_4	> 98%	< 4s	$80	True
S_5	> 90%	< 1s	$90	True
S_6	> 95%	< 2s	$99	True

According to MIP formula described before we have utility function for each attribute according to (3-3) as below.

$$uq_1 = \frac{Q_1 - 0.9}{0.98 - 0.9} \qquad uq_2 = \frac{Q_2 - 1}{4 - 1} \qquad uq_3 = \frac{Q_3 - 80}{110 - 80} \qquad uq_4 = \frac{Q_4 - 0}{1 - 0}$$

Moreover we have weight parameter for each attribute based on (3-4):

$\omega_1 = \frac{(3-q_1.pref+0.1)}{3-1} = 0.05$ $\omega_2 = \frac{(3-q_2.pref+0.1)}{3-1} = 0.55$

$\omega_3 = \frac{(3-q_3.pref+0.1)}{3-1} = 1.05$ $\omega_4 = \frac{(3-q_4.pref+0.1)}{3-1} = 0.05$

We first calculate MIPP for reliability. According to (3-5) we will have:

$min: 0.05 \times (12.5 \times q1 - 11.25) = 0.625 \times q1 - 0.5625$

q_1 >= 0.95; → Q.MIPP.q_1 = 0.03125

According to (3-6) we will have:

$Si: min: 0.05 \times (12.5 \times Si.qs1 - 11.25) = 0.625 \times Si.qs1 - 0.5625$

S_1.qs1 >= 0.95 → S_1.MIPP.qs_1 = 0.03125

Since reliability has positive tendency, according to (3-9), we can decide that S_1 satisfies the request on this attribute. We can continue to check other services.

Similarly we can calculate MIPP for the other three attributes. Then, according to (3-11) and (3-12) we will have the ranking scores of each offer:

Q.MIPP = Q.MIPP.q_1 + Q.MIPP.q_2 + Q.MIPP.q_3 + Q.MIPP.q_4 = 0.61975

MIPP_S_1 = S_1 .MIPP.qs_1 + S_1 .MIPP.qs_2 + S_1 .MIPP.qs_3 + S_1 .MIPP.qs_4 = 0.64375

→ *S_1. ranking_score* = 0.024

MIPP_S_2 = 5 → *S_2. ranking_score* = 0.248

MIPP _S3 = 2 → *S_3. ranking_score* = 0.528

MIPP _S4 = 3.98 → *S_4. ranking_score* = 0.152

MIPP _S5 = 2 → *S_5. ranking_score* = 0.528

MIPP _S6 = 2 → *S_6. ranking_score* = 0.036

Table 3.2 below shows the results of the case study. In the table, we highlight the service attribute values which could satisfy the constraints.

Table 3.2-The results of the case study

Q	MIPP.q_1	MIPP.q_2	MIPP.q_3	MIPP.q_4	Q.MIPP
	0.03125	0.1815	0.357	0.05	0.61975

	MIPP.qs_1	MIPP.qs_2	MIPP.qs_3	MIPP.qs_4	S.MIPP	Diff	Rank
S_1	***0.03125***	0.363	***0.1995***	***0.05***	0.64375	0.024	1
S_2	***0.05***	***0***	0.672	***0.05***	0.772	0.248	3
S_3	***0.05***	***0***	***0.042***	0	0.092	0.528	4
S_4	***0.05***	0.5445	***-0.273***	***0.05***	0.3715	0.152	2
S_5	0	***0***	***0.042***	***0.05***	0.092	0.528	-1
S_6	***0.03125***	***0.1815***	***0.3255***	***0.05***	0.58825	0.036	0

Based on what we explained before, the preference order of the 4 attributes is (3, 2, 1, 3), and the relaxation order is (0, 1, 2, 3). When the relaxation order is zero, it means the corresponding constraint is a hard constraint. Among the six services in our case study S_6 will be placed at the top of the result list since it satisfies all the constraints. Besides, when calculating S_6.*MIPP*, its value is greater than the value for Q.*MIPP*, which shows that S_6 is slightly better than what the requester is looking for so it will be placed in the super-exact matched list. The offer S_5 will be filtered out since it does not meet the hard constraint on reliability. All the other four offers are qualified. Since none of them satisfy all the requirements, we need to look at the relaxation order to decide how we can trade-off among different offers. According to their MIPP distance, the final ranking for partially matched offers could be (S_1, S_4, S_2, S_3).

From this example, we could see that when the trade-off decision needs to be made, the relaxation order should be considered first, and then the preference order. Most of the current

selection models would not consider them differently, which we feel couldn't fully capture the real user requirements.

3.7. Summary

In this chapter we defined our QoS language features, in terms of variables' different properties and operators, which we feel are necessary to express the real user requirement in a more accurate and comprehensive way and are missing in the current languages. Our proposed XML-based QoS language has the property of portability, extensibility, modularity, verification and validation.

By portability we mean it could work well alongside other applications which is provided by XML features, e.g. to encapsulate information in order to be used by other systems. With XML infrastructure as the underlying structure for our language, extensibility will be provided. Moreover, by using XML we will have a modular and structured document (query) and using XML and XML-schema allows our language to verify and validate user's queries easily and clearly. Later in the chapter we introduced our user-centric selection framework's UI which could guide users through different steps of formulating their queries.

Moreover, in this chapter we explained the user-centric selection system architecture, and we also discussed how to process the newly proposed features in our query language, such as fuzzy values and separate relaxation and preference orders. Handling fuzzy values is implemented by an interval-based clustering algorithm. Then we presented our selection algorithm using a flowchart and then moved on to explain it in terms of matching, relaxing and ranking methods. At the end of the chapter we further illustrated our selection and ranking algorithm with a case study step by step.

CHAPTER 4
IMPLEMENTATION AND EVALUATION

4.1. Implementation

As discussed earlier, our contributions are proposing a new QoS query language which leads us to design a user-centric interface to help users formulate queries and a selection method that supports fuzzy values and separate relaxation and preference orders. In this chapter first we are going through various steps of implementing the QoS-based selection framework, and then we present some examples and finally discuss the evaluation of our selection algorithm.

The dataset we used in our experiment is a web service dataset called QWS [32], [33] which includes information of 2507 web services. For each service, it contains real data for various QoS attributes including response time, availability, throughput, successability, reliability, compliance, best practices, latency and documentation, as well as the service name and its WSDL address.

In the functional matching layer, we implement an application using *Lucene* [34], to discover the functional matched offers from this QWS dataset. In order to get relevant keywords for each service, we extract terms from "service name" field, and we download the WSDL file if it is still available and extract terms from it. Then we use Lucene to index all the keywords to form our search database. Lucene is one of the Apache projects. This open-source and free search engine could be used as an API in any application that needs text searching and indexing functionalities [34]. Lucene architecture is mainly based on documents including text fields. This structure helps Lucene API to work with any file format as long as it could extract the data of the fields, so naturally files like PDFs, XML, HTML, word documents and etc. are usable in this

model. By using Lucene's search package, we wrote a Java program to search through the service name and WSDL file of each web service based on an input keyword. If our defined keyword was found, that web service would be written in a text file as one of the functional matches according to that keyword. The output file of this program will be used as the input for the next layer - non-functional matching layer, of our framework. We design our experiments using some most frequently occurring keywords such as development, management, google, amazon, commerce, etc. As the result of this layer we have text files containing matched offers for each keyword.

In the non-functional matching layer, we used C# ASP.Net to implement our whole framework. Our program is a web application that runs on a web server. Thus it can be accessed by users easily without installing or configuring anything on their local machines, just by typing the URL of the application like any other web pages. According to the prototype design as explained earlier, we have six steps in our selection framework. In the first step, user is guided to choose a keyword and some QoS attributes to formulate his/her query. By choosing a keyword, the text file of the previous step is found and all the offers in it will be transformed to XML files based on their QoS values. Each offer's XML document has a structure like our language schema that was introduced in Figure 3.3. As a result we will have an XML repository, on the web server, that has XML files of functionally matched offers. In the second step, user will have to put together all the QoS requirements simply by typing each value in its textbox. To get more information on the value distribution of each attribute, he/she can use browse link on that particular attribute. This link leads user to a pop up web page, implemented as step 3 of the framework, which shows data clusters based on k-means clustering algorithm. The result of step 2 is an XML document representing user's query. The fourth step is about specifying preference

and relaxation order per attribute. Preference order helps calculating the weight to be used in the ranking process, and relaxation order helps in relaxation process to be used in selection procedure. Since we already have the query and all the offers' XML documents, we just need to update them in this step with the new information of relaxation and preference orders. The fifth step is about time constraints, despite designing this web page the framework won't do anything based on this page, it is just extra information right now, though it can be used in prediction procedure in future works. The sixth step is to do the matching and ranking for the query and generate the results page. This page will show the request, super-exact matches and partial matches separately. In order to generate this page, two main tasks are done: selection and ranking. Before explaining how they are implemented, we first clarify the matching method, which is a step in the selection process.

4.1.1. The Matching Method

In the *matching method,* for each attribute of each offer and query, we produce a mixed integer programming problem. This problem is an optimization problem, maximization or minimization, based on attribute's tendency. Figure 4.1 shows an example MIPP for the availability attribute of an offer. In cases of single type values and fuzzy values, first they are converted to range values and then the MIP problems are produced. The single type values are converted to value 1 or 0 depending on whether it is *true* or *false*. The fuzzy type values are converted to range values based on the range of the chosen cluster. For example, if user chooses *medium* cluster and the min-max range was (0.8, 0.92), the new range value would be this range instead of the *medium* term that represents a fuzzy value.

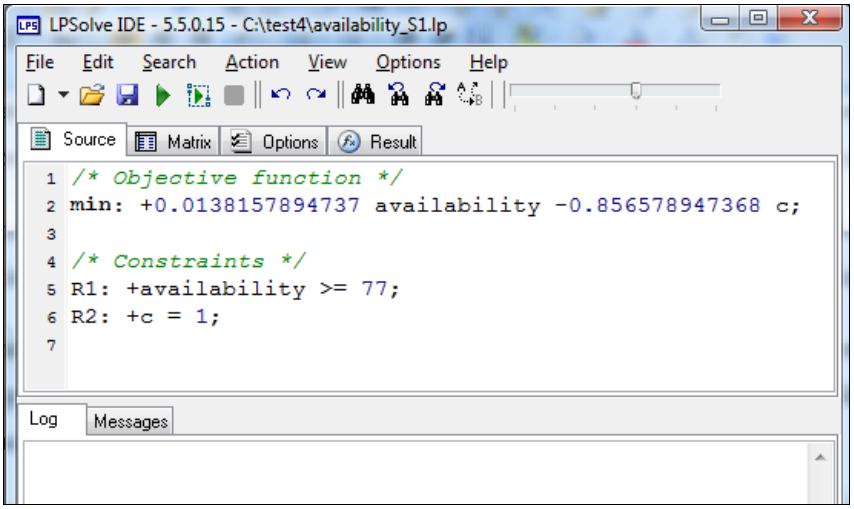

Figure 4.1- An instance of an offer's MIPP

Then the program solves each pair of two problems, the query and an offer's MIPP. Solving the MIPP is implemented by using a free API, called *lp-solve.5.5.0.15* [35]. The result of solving the query and an offer's MIPP is compared with each other, as shown in Figure 4.2:

```
if ((reqMIPP <= offerMIPP) && (attr_tend == "positive"))
            ret = true;
if ((reqMIPP >= offerMIPP) && (attr_tend == "negative"))
            ret = true;
```

Figure 4.2- Query and offer comparison

For instance, if we have the availability requirement as *"Query.Availability > 75"* and an offer is *"offer.Availability = 77"*, the MIPP of the offer is shown in Figure 4.1 and the MIPP of the query is in Figure 4.3.

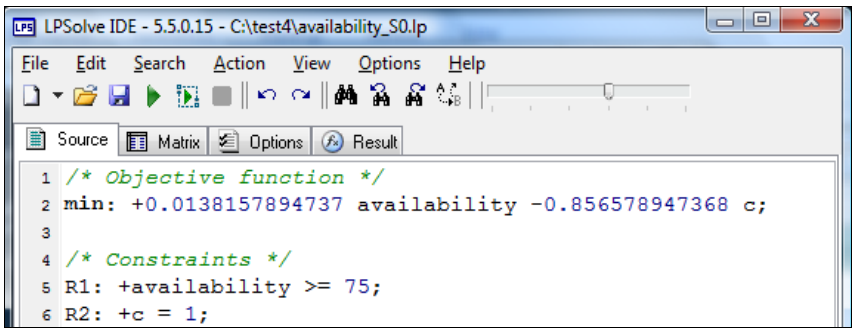

Figure 4.3- The availability MIPP of the query

After solving each problem, the objective function's result of the query and offer is shown in Figure 4.4, 4.5.

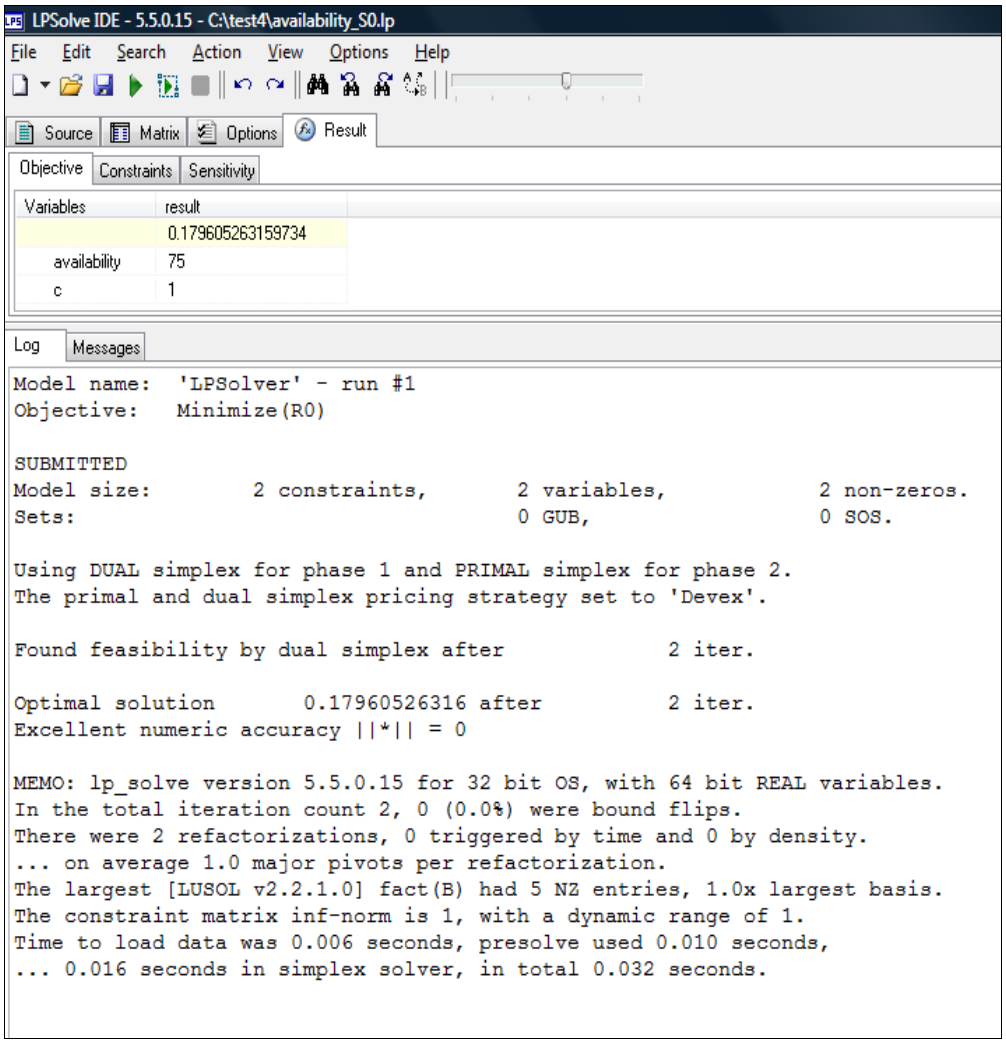

Figure 4.4-The result of objective function of the query

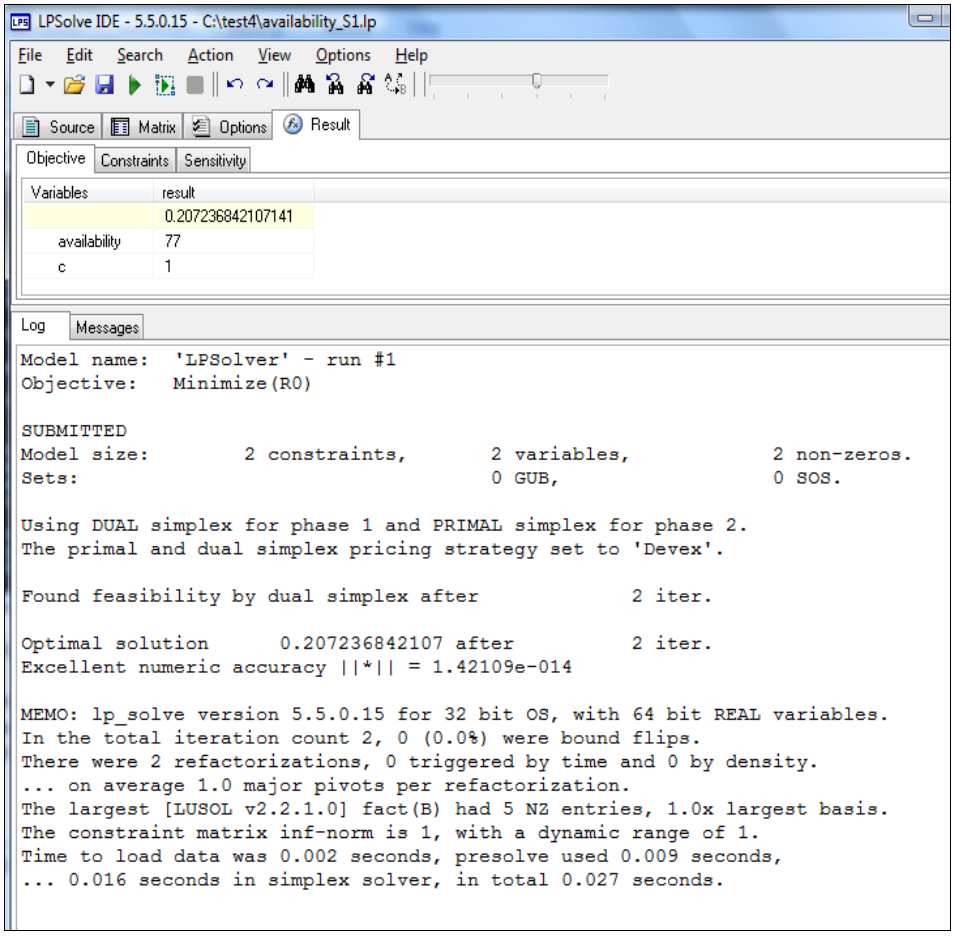

Figure 4.5-The result of objective function of the offer

Since the availability's tendency is positive, based on the conditions in Figure 4.2, the result of the comparison is true, the query's MIPP value 0.17 is less than the offer's value 0.2, meaning that the offer satisfies the request on this particular attribute. Thus for all the attributes of the offer and the query, this process will be applied, and if all the offer's attributes meet the requirements, that offer will be a match for the query.

In order to create MIPP model in lp-solve, we follow these steps:

- Make MIPP model with make_lp(0, 2) method with 2 columns, this method creates a blank MIPP model that will be configured with our desired variables during the following steps.

- Set the columns' names, one as our attribute, the other as a constant.

  We have to include this constant variable, because the API solver needs the model to contain at least two variables.

- Set the max or min of the problem based on the attribute's tendency.

```
if (str_tendency == "negative")
       lpsolve55.lpsolve.set_maxim(lp);

if (str_tendency == "positive")
       lpsolve55.lpsolve.set minim(lp);
```

- Calculate the weight based on the preference order of the QoS attribute.

```
double w = (find_max_pref() – qi.pref + 0.1)/ (find_max_pref() - find_min_pref());
```

- Then normalize the attribute's value and with the calculated weight from the previous step, produce the utility function.

```
max = find_max(attr_name);
min = find_min(attr_name);
Uq = w*(q-min)/(max-min)
```

- Set the objective function based on the utility function.

```
lpsolve55.lpsolve.str_set_obj_fn(lp, uq.ToString());
```

- Add the constraints based on tendency.

```
if (str_tendency == "positive")
    lpsolve55.lpsolve.str_add_constraint(lp, "1 0",
              lpsolve.lpsolve_constr_types.GE, Convert.ToDouble(lowerVal));
if (str_tendency == "negative")
    lpsolve55.lpsolve.str_add_constraint(lp, "1 0",
               lpsolve.lpsolve_constr_types.LE, Convert.ToDouble(upperVal));
```

- Add the constraint for the constant.

```
lpsolve55.lpsolve.str_add_constraint(lp, "0 1", lpsolve.lpsolve_constr_types.EQ, 1);
```

- Solve the problem and return the value as MIPP.

```
lpsolve55.lpsolve.solve(lp);
```

- Then delete the problem, to free the used space.

```
lpsolve55.lpsolve.delete_lp(lp);
```

4.1.2. The Selection Method

First we need to find exact or super matched offers for the requested query. In this method we find all the offers that satisfy all the constraints of the query, and then insert them into a table for later presentation as the super-exact matches in the final results. The *matching method* is as explained earlier.

Then we consider the remaining offers, and find the ones that satisfy all the hard constraints (attributes with the relaxation order of zero). We apply the relaxation method on these offers. Relaxation method goes through all the attributes with the relaxation order other than zero, from smallest to the greatest. For each offer, it will remove the corresponding constraint and apply the matching method to see if it will satisfy other constraints in the request. In this way we will find partial matches for the request, and then insert them into another table.

4.1.3. The Ranking Method

In this method, we just solve all MIP problems of each offer and the query. Then we calculate the distance between each offer and query, the result would be the ranking score.

4.2. Experiments

We design our experiments based on queries of different keywords. We rank the keywords for those services based on their frequencies and choose the 9 most frequent terms including amazon, business, flight, commerce, protein, development, google, management, and Net. For each one we design one or more query with different QoS requirements.

4.2.1. Sample Queries and Results

In the first sample query, the keyword is "amazon" and the QQL query is $Q_1 = (280, medium, 9, 8)$, where q_1 is response time, q_2 is availability, q_3 is latency and q_4 is throughput. The relaxation and preference order of the query are defined as follows:

Relaxation: (1,2,3,4) *Preference: (2,1,2,1)*

The returning result from our framework is shown in Table 4.1. In this table, the first 13 rows presented in darker background show the offers that satisfy all the constraints. The rest of the rows show offers that partially match the query's constraints in the ascending order of their ranking scores, from the smallest value to the largest. As presented in Table 4.1, the framework applied the clustering method on the fuzzy value of q_2 and returns the constraint as $q_2>65$ which contains the "*medium*" and "*good*" value for the availability. Since there are not any hard constraints in this sample query, the partially matched offers are the ones that at least satisfy one of the soft constraints in the order of their relaxation order values as q_1 with relaxation order of 1, q_2 with relaxation order of 2, q_3 with relaxation order of 3, and q_4 with relaxation order of 4.

Table 4.1- Sample Result for a Query with “Amazon” as the Keyword

amazon	rank		ranking score	response time	availability	latency	throughput
		Q1		<280	>65	<9	>8
	1	S18	0.008550896	57.42	66	5.31	12.5
	2	S22	0.024599329	60.84	70	5.95	10.5
	3	S25	0.033349621	71.21	71	6	10.7
	4	S19	0.036351968	62.05	72	6.05	10.2
	5	S1	0.060302372	59.58	77	3.95	8.8
	6	S5	0.062099425	60.05	77	4.1	9.1
	7	S31	0.06952583	60.47	78	4.47	9.2
	8	S39	0.07506712	61.42	79	5.37	8.9
	9	S34	0.07979269	61.42	80	4.95	8.6
	10	S35	0.083303475	64.28	80	5.78	9.1
	11	S8	0.144199525	55.5	81	4.75	19.1
	12	S14	0.193058355	60.18	93	8.93	13.1
	13	s46	0.208196477	56.17	97	7.17	11.3
	14	S45	0.01377315	58.21	62	4.68	13.3
	15	S4	0.016057652	47.27	61	2	20.3
	16	S15	0.019428451	79.65	61	2.75	13.3
	17	S28	0.071988848	527.5	72	21	7.2
	18	S36	0.089496311	105.43	87	5.57	1.2
	19	S20	0.108595933	107.57	89	5.71	2.2
	20	S26	0.109686376	159.5	78	84.75	6.8
	21	S24	0.117063182	64.75	86	7.95	7.8
	22	S11	0.118105114	123.45	86	5.8	7.2
	23	S40	0.128006082	121.15	88	8.2	6.4
	24	S6	0.151481954	65.4	92	8.25	6.8
	25	S7	0.168704826	229.79	89	41.32	7.4
	26	S43	0.171742184	226.58	90	37.37	7.2
	27	S30	0.172012811	243	89	49.11	7
	28	S23	0.172510097	263.89	87	70.52	7
	29	S12	0.172809361	227.53	90	40.16	7.1
	30	S37	0.17287087	217.05	91	28	7.3
	31	S16	0.173462995	247.05	89	55.1	6.6
	32	S32	0.17473687	248.32	89	50.11	7.3
	33	S9	0.193240098	305.37	87	90.95	8
	34	S33	0.220863002	64.64	32	30	9.2
	35	S27	0.310104096	328.67	21	10.43	3.7
	36	S10	0.336343448	221	18	11.35	4.3
	37	S44	0.338165815	70.23	17	5.58	8.3
	38	S41	0.339074645	143.65	19	9.69	4.1
	39	S17	0.339757767	75.13	20	5.17	4.4
	40	S13	0.344245792	63.83	19	3.92	5.1
	41	S3	0.346567029	68.91	19	5.91	4.4
	42	S29	0.352055077	71.54	18	8.54	4.3
	43	S2	0.354542435	64.96	18	5.15	4.3
	44	S42	0.35465465	70.62	18	3.97	4.3
	45	S21	0.354778446	71.62	17	5.5	5.3
	46	S38	0.360462028	68.88	17	5.76	4.3
		relaxation		1	2	3	4
		preference		2	1	2	1

In the second sample query, the keyword is "protein". The QQL query is: $Q_2 = (250, 84, 30, 9)$, where q_1 is response time, q_2 is availability, q_3 is latency and q_4 is throughput. The relaxation and preference order of the query are as follows:

Relaxation: (0,0,1,2) *Preference: (1,2,4,3)*

The returning result in Table 4.2 shows 62 offers, the first 16 rows in the darker background are the offers that satisfy all the constraints. The rest of the table shows offers that partially match the query's constraints in the order of their ranking score, from the smallest value to the largest. Since in this query we have hard constraints on q_1 and q_2, the partially matched offers are the ones that satisfy these two constraints but may not satisfy others. Based on the query and the relaxation and preference order, the partially matched offers should have q_1 less than 250 and q_2 greater than 84 but if other constraints are not met, they will be all right as well.

4.3. Evaluation of QoS Selection Algorithm

Since our algorithm adds extra parts such as relaxation and clustering components on top of the original MIP algorithm, we would like to check the efficiency of our algorithm. We conducted a comparison of the execution time between our selection method and the original (plain) MIP selection algorithm. The data for this analysis is shown in Table 4.3 which includes five columns, the first one is the keywords, the second one is the average time of running our algorithm in milliseconds, and the third column shows the average execution time of the plain MIP algorithm in milliseconds. The fourth column is the difference of the second and third columns and the last one is percentage of the difference. As illustrated in Figure 4.6, the time difference between our algorithm and plain MIP is not greater than 34%, which shows although we added the clustering and relaxation modules to the plain MIP algorithm, the processing time of the new algorithm is not increased that much.

Table 4.2- Sample Result for a Query with “Protein” as the Keyword

protein	rank		ranking score	response time	availability	latency	throughput
		Q1		<250	>84	<30	>9
	1	S130	0.003382145	222.6	85	6	15.6
	2	S7	0.003419488	77.9	91	20.57	17.1
	3	S95	0.042889808	243	88	10.2	13.8
	4	S28	0.055394938	220.8	90	8.4	14.6
	5	S22	0.070671616	227	91	10.6	14.2
	6	S58	0.080741478	223.6	92	8.8	14.6
	7	S44	0.10971979	233	94	11.4	14.5
	8	S104	0.11696956	224.2	95	10.4	14.3
	9	S51	0.143971738	230.6	97	9.6	14.2
	10	S29	0.15299771	224.8	98	9.2	14.6
	11	S93	0.166954317	229.4	99	8.6	13.7
	12	S19	0.169859133	211.4	100	7.8	14.7
	13	S118	0.175898414	223.4	100	9.2	13.7
	14	S13	0.177057122	225.2	100	10.4	14
	15	S111	0.178707136	227.8	100	11.6	14.6
	16	S89	0.17998213	230.4	100	11.4	14.5
	17	S69	0.003710691	243.6	85	7	1.9
	18	S78	0.005810916	201.4	87	8	0.7
	19	S9	0.007085123	222.6	85	8.4	1
	20	S63	0.007149852	244.2	85	30.4	2.2
	21	S25	0.011796631	213	85	8.2	1.8
	22	S15	0.013776192	193	88	8.6	1.6
	23	S126	0.01539573	220.2	87	7.2	1
	24	S134	0.017592433	202.4	85	7.4	0.8
	25	S102	0.021611383	231.8	87	9.2	1.1
	26	S96	0.021637581	208.6	88	8.6	1.2
	27	S74	0.026900021	242	87	6.2	2.8
	28	S100	0.028520051	246.4	87	5.6	0.8
	29	S92	0.040992052	223.2	89	6.2	2.2
	30	S105	0.043018122	227.4	89	7.6	1.1
	31	S20	0.047476865	212.2	90	8.8	1.6
	32	S77	0.049916035	217.2	90	9.6	0.8
	33	S8	0.052421736	245.8	89	7.4	1.2
	34	S21	0.053834362	201.6	91	7.8	1.5
	35	S112	0.058756764	243.07	89	59.53	4.6
	36	S49	0.058860589	248.6	89	45.4	1.4
	37	S117	0.064965407	200.2	92	8.2	1
	38	S30	0.069978065	233.4	91	7.2	1.4
	39	S37	0.07013982	243.02	90	56.36	4.2
	40	S120	0.076374251	222	92	9	1.6
	41	S10	0.079140744	203.8	93	9.4	1.9
	42	S11	0.085885386	216	93	1.5	7.5
	43	S27	0.085961944	218.6	93	5.4	1.1
	44	S65	0.086297602	218.2	93	6.8	2.4
	45	S17	0.089785588	201.8	94	8	1.6
	46	S48	0.093540367	247.4	92	42.6	0.9
	47	S64	0.094313789	249.2	92	40	1.6
	48	S79	0.098591388	219.2	94	8.4	1.1
	49	S5	0.098783932	197	95	6	0.8
	50	S90	0.107317335	236.6	94	4.6	2.3
	51	S106	0.116917185	231.2	95	10.2	1.2
	52	S81	0.120493459	238.4	95	9.2	1.3
	53	S45	0.121860368	234.8	95	33.8	1.3
	54	S40	0.122938661	220.6	96	6.2	1.3
	55	S12	0.148331203	199.4	99	7.6	2.2
	56	S87	0.156898027	234.4	98	26.6	2.7
	57	S86	0.164386002	231.4	99	6.8	1.5
	58	S80	0.169358995	241.2	99	7.8	0.9
	59	S66	0.173027176	225	100	5.8	1.9
	60	S26	0.176101235	245.6	99	40.8	1.6
	61	S39	0.181189411	241.2	100	6.6	1.1
	62	S35	0.188785049	246.4	100	42.8	1.9
		relaxation		0	0	1	2
		preference		1	2	4	3

Table 4.3- Comparison of average execution time of plain MIP and our algorithm

Keyword	Avg. Exec. Time of our algorithm (ms)	Avg. Exec.Time of Plain MIP(ms)	Difference	% of Difference
amazon	64.3745	58.832	5.5425	8.6
commerce	11.376	7.818	3.558	31.2
development	37.025	26.128	10.897	29.4
flight	3.376	3.049	0.327	10.7
google	66.1335	55.18	10.9535	10.7
management	38.602	32.913	5.689	17.28
.Net	25.06533	16.46867	8.59667	34.2
protein	456.741	366.021	90.72	19.8
business	31.152	21.271	9.881	31.7

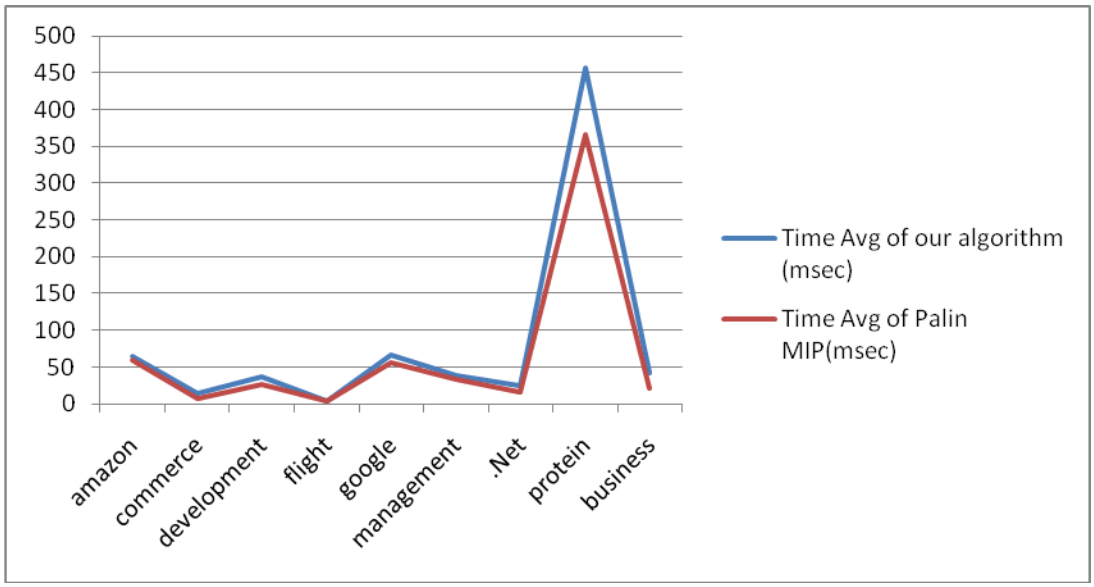

Figure 4.6- Average execution time of plain MIP and our algorithm

In the next evaluation, we conduct an experiment to measure the precision of our selection algorithm. In this experiment we measure the precision of the plain MIP which is the base of our

algorithm; then we try to measure the precision of our proposed algorithm and finally compare them with each other.

By finding the relevant offers in our algorithm result and plain MIP result and use the (4-1) formula the precision is measured. Since for each keyword, we have at least two queries, thus we calculate the average value of precision for each keyword based on its total number of queries.

$$Precision\ percentage = \frac{number\ of\ retrieved\ relevant\ offers}{number\ of\ retrieved\ offers} \times 100 \quad (4\text{-}1)$$

According to our implementation, based on the 9 chosen keywords, the average precision of the selection algorithm of simple MIP and our algorithm is shown in Table 4.4.

Table 4.4- Precision of plain MIP and our algorithm

keyword	Avg. Precision of our algorithm (%)	Avg. Precision of Plain MIP (%)
amazon	97.82	28.26
commerce	94.44	17.65
development	95.83	16.66
flight	80	57
google	94.44	64.71
management	100	50
.Net	90.91	40
protein	98.38	26.23
business	91.23	4.5

As discussed earlier the matching method checks if each of the QoS attributes of an offer will satisfy the request based on formula 3-4 and 3-5. Thus, as we expected, the result of the plain MIP is only the offers satisfying all the constraints and offers with better QoS values without considering the relaxation process. However our new algorithm returns partial matches

for the request as well as super and exact matches. If we consider just the precision of the matched offers the precision for both algorithm are the same, the difference is the precision of the partial matched offers. Figure 4.7 below shows the same result.

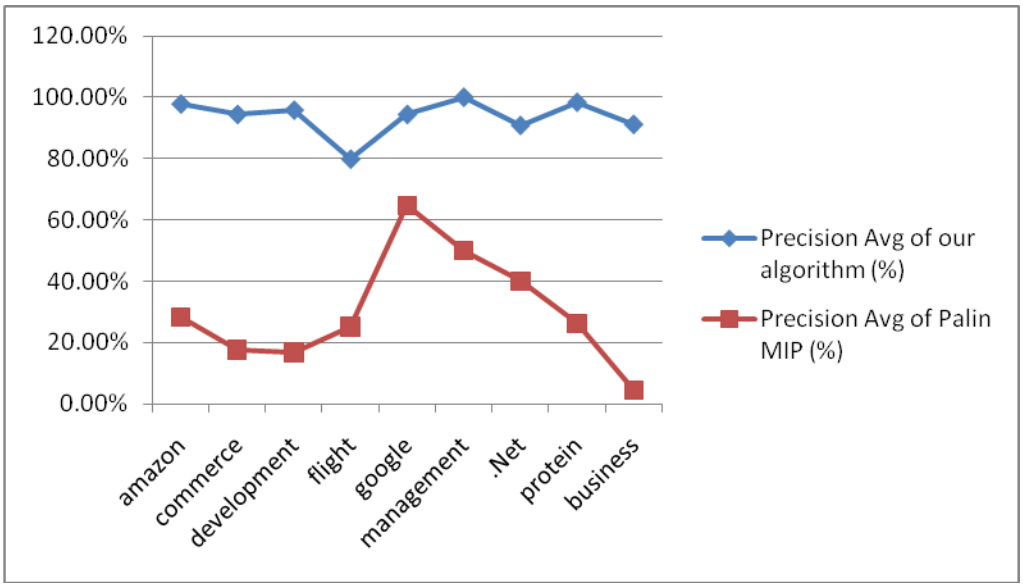

Figure 4.7- Precision of plain MIP and our algorithm

Another evaluation is done regarding the ranking results of our selection algorithm. In this experiment we measure the ranking difference between our algorithm's results and the manual ranking result. We also compare the ranking results of the plain MIP algorithm with the manual result. Taking the manual result as the baseline, we compare the ranking effectiveness of our algorithm and the plain MIP. The ranking difference is calculated by Kendall tau method [36] as a correlation coefficient between two ranked lists. In order to estimate the Kendall tau value, we use a free application called "Past" [37] that implements this method and only needs inputs in the form of spread sheets. The result is shown in Table 4.5 for each keyword.

According to [38], the value of calculated correlation between [0.1, 0.3] or [-0.3,-0.1] shows a small correlation, values between [0.3, 0.5] or [-0.5,-0.3] represent a medium correlation and between [0.5, 1.0] or [-1.0,-0.5] demonstrate a large correlation. The large value for correlation coefficient means that the two compared ranking sets have high level of similarity. According to Table 4.5, we can see that among the 9 keywords, the results for 4 of them from our approach are very close to the manual ranking, 2 are close, and 3 are not close. And the ranking from our approach is more similar to the manual ranking than the plain MIP approach. One of the reasons for the low Kendall Tau values for some queries is that Kendall Tau test is to measure the exact match between two lists, which might be hard to achieve in many cases. Since the size of our test set is quite limited, more experiments are required to reach a conclusion.

Table 4.5 – Comparison of Kendall Tau correlation coefficient between our algorithm and plain MIP

keyword	Kendall Tau Of our algorithm	Kendall Tau Of plain MIP
amazon	0.25	0.21
commerce	0.75	0.67
development	-0.6	-0.1
flight	0.33	0.33
google	0.1	0.1
management	0.21	-0.14
.Net	-0.67	-0.67
protein	0.18	0.18
business	0.1	0.099

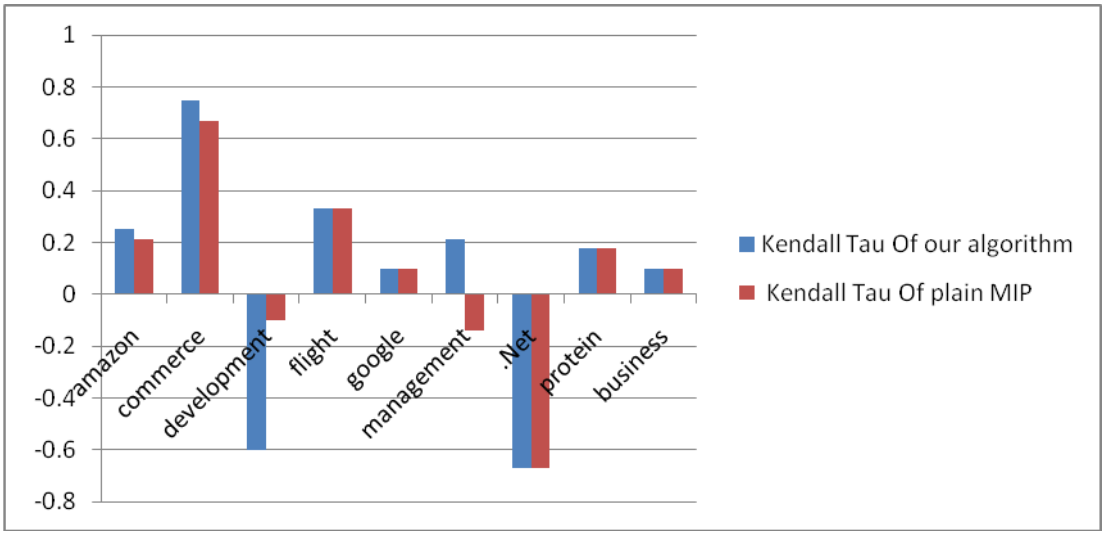

Figure 4.8- Correlation of plain MIP and our algorithm

4.4. Summary

In this chapter, we explained our framework's implementation including different layers such as functional matching layer and QoS matching layer. In the functional matching layer, we implemented an application using *Lucene*, to discover the functionally matched offers from a test dataset. In the QoS matching layer, we could help users formulate their QoS queries step by step in a guided way. Furthermore we discussed the implementation of our selection algorithms, ranking method and matching method. And then we presented two sample queries with different values, in terms of fuzzy and range types. Finally we analyzed our selection algorithms by measuring and comparing the average running time and precision between the plain MIP and our enhanced selection algorithm. We also measured the accuracy of our selection algorithm by checking the difference of the ranking results from the plain MIP ranked results.

CHAPTER 5
CONCLUSIONS

5.1. Conclusion

We started our thesis by reviewing various researches on QoS query language descriptions and selection models. According to these reviews, we found some issues that have not been solved yet. Thus we proposed our new QoS query language and selection model to support the solution for these problems, and we specially emphasized on the support for the user-centric selection process. Our language has a few new features such as the separate preference and relaxation order, the mixed fuzzy and value requests, which could give users more flexibility and power to express their actual requirements in a more accurate way. We understand that a new query language always poses some cognitive overload on users, and thus we design our system in a guided way to make sure it is easy for users to formulate their queries. To support all these new features, our selection model adds extra handlers on top of the MIP algorithm. From our experiment, we could see that although our algorithm takes longer time to generate the result compared with the original MIP algorithm, the precision of the ranking result is largely improved.

5.2. Main Contributions

There are three main contributions of this thesis:

- We proposed a more expressive and powerful QoS query language suitable for unprofessional users, together with the user support on formulating queries and understanding services in the registry.
- We provided new features such as time dimension, separate relaxation and preference policies and browsing various QoS attributes for our new language.
- We proposed an enhanced selection model based on MIP which is capable of handling relaxation policies per variables based on user defined relaxation orders, processing fuzzy requirements, and providing ranked results in two categories of super-exact matches and partial matches.

5.3. Future Works

There are a few directions we can work on in the future. One possibility for improving our framework is implementing the prediction part according to the time dimension of the proposed language. Another option to improve our algorithm efficiency is using other distance functions in clustering component, such as Hausdorff [39] , or City Block [29] distance functions. Another alternative for expanding our research would be to convert our syntactic language to semantic language. And finally we can work on further improving our ranking algorithm.

APPENDIX A -Selection method

```
void selection()
    {
        request = S_assign_Xml[0];
        for (int p = 0; p < offers_cnt; p++)
        {
            Range_result[p] = selection_Hard(S_assign_Xml[p]);
        }
        AllHard_result = Range_result;
        for (int h = 1; h < offers_cnt; h++)
            insert(AllHard_result[h], h, "tbl_Hard_selected");

        for (int j = 0; j < GridView2.Rows.Count; j++)
        {
            Label lblSID = (Label)GridView2.Rows[j].FindControl("lblSID");
            TextBox txtRank = (TextBox)GridView2.Rows[j].FindControl("txtRank");
            updateRank(lblSID.Text, Convert.ToDouble(txtRank.Text), "tbl_Hard_selected");
        }


       for (int i = 0; i < offers_cnt; i++)
       {
            Range_result[i] = selection_Range(S_assign_Xml[i]);
       }

        whole_result = Range_result;

        for (int h = 1; h < offers_cnt; h++)
            insert(whole_result[h], h, "tbl_Selected");


        for (int j = 0; j < GridView2.Rows.Count; j++)
        {
            Label lblSID = (Label)GridView2.Rows[j].FindControl("lblSID");
            TextBox txtRank = (TextBox)GridView2.Rows[j].FindControl("txtRank");
            updateRank(lblSID.Text, Convert.ToDouble(txtRank.Text), "tbl_Selected");
        }

        int empty_whole = 0;
        for (int p = 1; p < offers_cnt; p++)
        {
            if (whole_result[p].isEmpty(whole_result[p]) == false)
            {
                empty_whole++;
            }
        }
        //startdate
        for (int r = 1; r < attr_cnt; r++)
        {
            for (int j = 1; j < offers_cnt; j++)
            {
                Range_result[j] = selection_Range_Soft(whole_result[j],r);
            }
        }

        whole_result_soft = Range_result;

        int empty_whole2 = 0;
        for (int p = 1; p < offers_cnt; p++)
        {
            if (whole_result_soft[p].isEmpty(whole_result_soft[p]) == false)
            {
                ++empty_whole2;
            }
        }
```

```
        if (empty_whole2 == 0) // it means the whole intersection of soft constraints is
empty
        {
            for (int r = 1; r <= attr_cnt; r++)
            {
                for (int g = 1; g < offers_cnt; g++)
                {
                    if (r == whole_result[g].reliability.relax_order)
                    {
                        for (int j = 1; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                    if (r == whole_result[g].responsetime.relax_order)
                    {
                        for (int j = 1; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                    if (r == whole_result[g].price.relax_order)
                    {
                        for (int j = 1; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                    if (r == whole_result[g].authentication.relax_order)
                    {
                        for (int j = 0; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                    if (r == whole_result[g].availability.relax_order)
                    {
                        for (int j = 1; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                    if (r == whole_result[g].latency.relax_order)
                    {
                        for (int j = 1; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                    if (r == whole_result[g].throughput.relax_order)
                    {
                        for (int j = 1; j < offers_cnt; j++)
                        {
                            Range_result[j] = selection_Range_Soft(whole_result[j], r);
                        }
                    }
                }
            }
            whole_result_soft = Range_result;

            for (int h = 1; h < offers_cnt; h++)
                insert(whole_result_soft[h], h, "tbl_Selected");

            delete("tbl_Selected", "tbl_Hard_selected");


            for (int j = 0; j < GridView2.Rows.Count; j++)
            {
                Label lblSID = (Label)GridView2.Rows[j].FindControl("lblSID");
                TextBox txtRank = (TextBox)GridView2.Rows[j].FindControl("txtRank");
                updateRank(lblSID.Text, Convert.ToDouble(txtRank.Text), "tbl_Selected");
            }
        }
        //stopdate
        GridView4.DataBind();
        GridView5.DataBind();
        System.GC.Collect();
    }
```

APPENDIX B - Matching method

```
   bool range(string a_name, string attr_tend, int pref, object req_up_val,
               object req_low_val, object S_up_val, object S_low_val)
   {
        bool ret = false;
        double reqMIPP = 0.0, offerMIPP = 0.0;
        reqMIPP = lp_solve_matching(a_name, attr_tend, pref,
                                    Convert.ToDouble(req_low_val), Convert.ToDouble(req_up_val));
        offerMIPP = lp_solve_matching(a_name, attr_tend, pref,
                                     Convert.ToDouble(S_low_val) , Convert.ToDouble(S_up_val));
        if ((reqMIPP <= offerMIPP) && (attr_tend == "positive"))
            ret = true;
        if ((reqMIPP >= offerMIPP) && (attr_tend == "negative"))
            ret = true;
        return ret;
    }
```

```
    double lp_solve_matching(string attr_name, string str_tendency, int prefOrder,
                          double lowerVal , double upperVal)
    {
        string uq = "";
        double min = 0.0, max = 0.0;
        double MIP_pref = 0.0;
        if (attr_name != "")
        {
            int lp;
            lp = lpsolve55.lpsolve.make_lp(0, 2);
            if (attr_name == "response time")
                lpsolve55.lpsolve.set_col_name(lp, 1, "responsetime");
            else
                lpsolve55.lpsolve.set_col_name(lp, 1, attr_name);
            lpsolve55.lpsolve.set_col_name(lp, 2, "c");
            if (str_tendency == "negative")
                lpsolve55.lpsolve.set_maxim(lp);
            if (str_tendency == "positive")
                lpsolve55.lpsolve.set_minim(lp);
            if (str_tendency == "neutral")
                lpsolve55.lpsolve.set_maxim(lp);
            max = find_max(attr_name);
            min = find_min(attr_name);
            double w = find_weight(attr_name, prefOrder);
            writeXML_weights(attr_name, w);
            uq = (w / (max - min)).ToString();
            uq += " " + (-w * min / (max - min)).ToString();
            lpsolve55.lpsolve.str_set_obj_fn(lp, uq.ToString());
            if (str_tendency == "positive")
                lpsolve55.lpsolve.str_add_constraint(lp, "1 0", lpsolve.lpsolve_constr_types.GE,
                      Convert.ToDouble(lowerVal));
            if (str_tendency == "negative")
                lpsolve55.lpsolve.str_add_constraint(lp, "1 0", lpsolve.lpsolve_constr_types.LE,
                      Convert.ToDouble(upperVal));
                     lpsolve55.lpsolve.str_add_constraint(lp, "0 1",
lpsolve.lpsolve_constr_types.EQ, 1);

            lpsolve55.lpsolve.set_outputfile(lp, "C://test6//" + attr_name + "_S" +
"_result.lp");
            lpsolve55.lpsolve.write_lp(lp, "C://test6//" + attr_name + "_S" + ".lp");
            lpsolve55.lpsolve.solve(lp);
            lpsolve55.lpsolve.print_objective(lp);
            lpsolve55.lpsolve.print_solution(lp, 2);
            lpsolve55.lpsolve.print_constraints(lp, 2);

            MIP_pref = lpsolve55.lpsolve.get_objective(lp);

            lpsolve55.lpsolve.delete_lp(lp);
        }
        return MIP_pref;
    }
```

APPENDIX C - Ranking method

```
double ranking(string SID)
 {
     double s_r = 0.0, req_r = 0.0;
     double diff = 0.0;
     req_r = lp_solve(S_assign_Xml[0], 0);
     int ind = Convert.ToInt32(SID.Substring(SID.IndexOf("S")+1));
     s_r = lp_solve(S_assign_Xml[ind], ind);
     diff = Math.Abs(req_r - s_r);
     return diff;
 }
```

```
double lp_solve(S_Obj S_beRanked, int counter)
{
     string attr_name = "";
     string[] uq = new string[1000];
     double rank_whole = 0.0;
     double min = 0.0, max = 0.0;
     string str_tendency = "neutral";
     for (int k = 0; k < attr_Order.Count(); k++)
     {
         attr_name = attr_Order[k].ToString();
         if (attr_name != "")
         {
             str_tendency = S_beRanked.find_tendency(attr_name).ToString();
             int lp;
             lp = lpsolve55.lpsolve.make_lp(0, 2);
             if (attr_name == "response time")
                 lpsolve55.lpsolve.set_col_name(lp, 1, "responsetime");
             else
                 lpsolve55.lpsolve.set_col_name(lp, 1, attr_name);
             lpsolve55.lpsolve.set_col_name(lp, 2, "c");
             if (str_tendency == "negative")
                 lpsolve55.lpsolve.set_maxim(lp);
             if (str_tendency == "positive")
                 lpsolve55.lpsolve.set_minim(lp);
             if (str_tendency == "neutral")
                 lpsolve55.lpsolve.set_maxim(lp);

             max = find_max(attr_name);
             min = find_min(attr_name);
             double w = find_weight(attr_name, S_beRanked.find_pref_order(attr_name));
             writeXML_weights(attr_name, w);
             uq[k] = (w / (max - min)).ToString();
             uq[k] += " " + (-w * min / (max - min)).ToString();

             lpsolve55.lpsolve.str_set_obj_fn(lp, uq[k].ToString());

             if (str_tendency == "positive")
                 lpsolve55.lpsolve.str_add_constraint(lp, "1 0",
     lpsolve.lpsolve_constr_types.GE, Convert.ToDouble(S_beRanked.find_lower_val(attr_name)));
             if (str_tendency == "negative")
                 lpsolve55.lpsolve.str_add_constraint(lp, "1 0",
     lpsolve.lpsolve_constr_types.LE, Convert.ToDouble(S_beRanked.find_upper_val(attr_name)));
             if (str_tendency == "neutral")
             {
                 if (S_beRanked.find_val(attr_name).Equals(true))
                     lpsolve55.lpsolve.str_add_constraint(lp, "1 0",
                                     lpsolve.lpsolve_constr_types.EQ, 1.0);
                 else
                     lpsolve55.lpsolve.str_add_constraint(lp, "1 0",
                           lpsolve.lpsolve_constr_types.EQ, 0.0);
             }
```

```
                lpsolve55.lpsolve.str_add_constraint(lp, "0 1", lpsolve.lpsolve_constr_types.EQ, 1);

                lpsolve55.lpsolve.set_outputfile(lp, "C://test4//" + attr_name + "_S" +
counter.ToString() + "_result.lp");
                lpsolve55.lpsolve.write_lp(lp, "C://test4//" + attr_name + "_S" + counter.ToString()
+ ".lp");

                  lpsolve55.lpsolve.solve(lp);
                  lpsolve55.lpsolve.print_objective(lp);
                  lpsolve55.lpsolve.print_solution(lp, 2);
                  lpsolve55.lpsolve.print_constraints(lp, 2);
                  double rank = lpsolve55.lpsolve.get_objective(lp);
                  rank_whole += rank;
                  lpsolve55.lpsolve.delete_lp(lp);
              }
          }
          return rank_whole;
      }
```

APPENDIX D - Clustering method

```
void Browse(int cond, string attribute)
    {
            int minInd = 0, maxInd = 0, medInd = 0;
            DateTime startTotalTime = DateTime.Now;
            csCluster selectedCluster = new csCluster();
            csKMeans kMeans = new csKMeans();
            string f_attr = "C:\\Users\\delnavaz\\Documents\\Visual Studio
2008\\WebSites\\myQL\\App_Data\\Cluster_data_" +
                            attribute + ".txt";
            initialCluster = kMeans.RetrieveData(f_attr);
            DateTime startRoutineTime = DateTime.Now;
            selectedCluster = (csCluster)(kMeans.CloneObject((csCluster)(kMeans.kMeans(initialCluster,
3))));
            DateTime endRoutineTime = DateTime.Now;
            TimeSpan routineDuration = endRoutineTime - startRoutineTime;

            double min = 0, max = 0, med = 0;
            selectedCluster =
(csCluster)(kMeans.CloneObject((csCluster)(GetClustersSummery(selectedCluster))));

            if (selectedCluster.Count > 0)
            {

                min = selectedCluster[0].MinLowerBound;
                max = selectedCluster[0].MaxUpperBound;
                med = selectedCluster[0].MinLowerBound;

                for (int i = 0; i < selectedCluster.Count; i++)
                {
                    if (min > selectedCluster[i].MinLowerBound)
                    {
                        minInd = i;
                        min = selectedCluster[i].MinLowerBound;
                    }

                    if (max < selectedCluster[i].MaxUpperBound)
                    {
                        maxInd = i;
                        max = selectedCluster[i].MaxUpperBound;
                    }

                }

                for (int e = 0; e < 3; e++)
                {
                    if ((e != maxInd) && (e != minInd))
                    {
                        medInd = e;
                        med = (selectedCluster[e].MinLowerBound + selectedCluster[e].MaxUpperBound) / 2;
                    }
                }

                double[] rangemin = new double[2];
                double[] rangemed = new double[2];
                double[] rangemed_child = new double[2];
                double[] rangemax = new double[2];

                rangemin = selectedCluster[minInd].Max();
                rangemed = selectedCluster[medInd].First();
                rangemax = selectedCluster[maxInd].Max();
```

```
            rangemed_child = selectedCluster[medInd][0];
             double min_med, max_med;
             min_med = rangemed_child[0];
             max_med = rangemed_child[1];
             for (int p = 1; p < selectedCluster[medInd].Count(); p++)
             {
                 if (selectedCluster[medInd][p].Min() < min_med)
                     min_med = selectedCluster[medInd][p].Min();
                 if (selectedCluster[medInd][p].Max() > max_med)
                     max_med = selectedCluster[medInd][p].Max();
             }
             if (cond == 2)
             {
                 result[0] = rangemin[0];
                 result[1] = rangemin[1];
             }
             if (cond == 1)
             {
                 result[0] = min_med;
                 result[1] = max_med;
             }
             if (cond == 0)
             {
                 result[0] = rangemax[0];
                 result[1] = rangemax[1];
             }
             cluster = (csCluster)(kMeans.CloneObject((csCluster)(selectedCluster)));

        }
```

REFERENCES

[1] D. Austin, A. Barbir, C. Ferris, and S. Garg, ”Web services architecture requirements”, *W3C Working Group Note*, W3C, 2002. Available at http://www.w3.org/TR/wsa-reqs. Last retrieved at July 2010.

[2] K. Kritikos, and D. Plexousakis*,”* Mixed-Integer Programming for QoS-Based Web Service Matchmaking*”, IEEE Transaction on Services Computing*, Vol. 2, Issue 2, pp.122-139, 2009.

[3] C. Zhou, L.T. Chia, and B.S. Lee, “Web Services Discovery with DAML-QoS Ontology”, *International Journal of Web Services Research*, Vol. 2, Issue 2, pp. 43-66, 2005.

[4] G. Dobson, R. Lock, and I. Sommerville, “QoSOnt: a QoS Ontology for Service-Centric Systems”, in *Proceedings of the 31st ERUOMICRO Conference on Software Engineering and Advanced Applications*, pp. 80-87, 2005.

[5] M. Tian, A. Gramm, H. Ritter, and J. Schiller, “Efficient Selection and Monitoring of QoS-aware Web Services with the WS-QoS Framework”, in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence*, pp. 152-158, 2004.

[6] K. Kritikos, and D. Plexousakis, “Semantic QoS Metric Matching”, in *Proceedings of the European Conference on Web Services*, pp. 265-274, 2006.

[7] G. Damiano, E. Giallonardo, and E. Zimeo, “onQoS-QL: A Query Language for QoS-based Service Selection and Ranking”, in *Proceedings of the International Conference on Service Oriented Computing – Workshops*, pp. 115-127, 2007.

[8] V.X. Tran, H. Tsuji, and R. Masuda, "A New QoS Ontology and its QoS-based Ranking Algorithm for Web Services", *Simulation Modeling Practice and Theory*, Vol 17, Issue 8, pp. 1378-1398, 2009.

[9] E.M. Maximilien, and M.P. Singh, "A Framework and Ontology for Dynamic Web Service Selection", *IEEE Internet Computing*, Vol 8, Issue 5, pp. 84-93, 2004.

[10] F.D. Paoli, M. Palmonari, M. Comerio, and A. Maurino, "A Meta-Model for Non-Functional Property Descriptions of Web Services", in *Proceedings of the IEEE International Conference on Web Services*, pp. 393-400, 2008.

[11] Y.T. Liu, A.H.H. Ngu, L.Z. Zeng, "QoS Computation and Policing in Dynamic Web Service Selection", in *Proceedings of the International Conference on World Wide Web*, pp. 66-73, 2004.

[12] Q. Ma, H. Wang, Y. Li, G. Xie, and F. Liu, "A Semantic QoS-aware Discovery Framework for Web Services", in *Proceedings of the IEEE International Conference on Web Services*, pp.129-136, 2008.

[13] I.V. Papaioannou, D. T. Tsesmetzis, I. G. Roussaki, M. E. Anagnostou., "A QoS ontology language for web services", *in Proceedings of the 20th International Conference on Advanced Information Networking and Applications, IEEE Computer Society*, pp. 18–25, 2006.

[14] E. Giallonardo, and E. Zimeo, "More semantics in QoS matching", *in Proceedings of the IEEE International Conference on Service Oriented Computing and Applications, IEEE Computer Society*, pp. 163–171, 2007.

[15] G.F. Tondello, and F. Siqueira, "QoS-MO ontology for semantic QoS modeling", *in Proceedings of ACM Symposium on Applied Computing, ACM Press*, pp. 2336–2340 , 2008.

[16] A. D'Ambrogio, "A Model-driven WSDL Extension for Describing the QoS of Web Services", in *Proceedings of the IEEE International Conference on Web Services*, pp.789-796, 2006.

[17] Q.X. Du, C.H. Chi, S. Chen, and J.M. Deng, "Modeling Service Quality for Dynamic QoS Publishing", in *Proceedings of the IEEE International Conference on Services Computing*, pp. 307-314, 2008.

[18] C. Herssens, I.J. Jureta, and S. Faulkner, "Dealing with Quality Tradeoffs during Service Selection", in *Proceedings of the International Conference on Autonomic Computing*, pp. 77-86, 2008.

[19] O. Martín-Díaz, A. Ruiz-Cortés, A. Durán, and C. Müller, "An Approach to Temporal-Aware Procurement of Web Services", in *Proceedings of the International Conference on Service Oriented Computing*, pp. 170-184, 2005.

[20] J. Yan, and J. Piao, "Towards QoS-based Web Service Discovery", in *Proceedings of the International Conference on Service Oriented Computing*, pp. 200-210, 2008.

[21] A. Zisman, J. Dooley, and G. Spanoudakis, "Proactive Runtime Service Discovery", in *Proceedings of the IEEE International Conference on Services Computing*, pp. 237-245, 2008.

[22] A. Ruiz-Cortés, O. Martín-Díaz, A.D. Toro, and M. Toro, "Improving the Automatic Procurement of Web Services Using Constraint Programming", *International Journal on Cooperative Information Systems*, Vol 14, Issue 4, pp. 439-468, 2005.

[23] J. Brans, and P. Vincke. "A preference ranking organization method*", Management Science*, Vol 31, Issue 6, pp. 647–656, 1985.

[24] V.X. Tran, and H. Tsuji, "QoS based Ranking for Web Services: Fuzzy Approaches", in *Proceedings of the 4th International Conference on Next Generation Web Services Practices*, pp. 77-82, 2008.

[25] P. Wang, K. Chao, C. Lo, C. Huang, and Y. Li, "A Fuzzy Model for Selection of QoS-Aware Web Services", in *Proceedings of the IEEE International Conference on E-Business Engineering*, pp. 585-593, 2006.

[26] P. Xiong, and Y. Fan, "QoS-Aware Web Service Selection by a Synthetic Weight", in *Proceedings of the 4th International Conference on Fuzzy Systems and Knowledge Discovery*, pp. 632-637, 2007.

[27] S. Ran, "A Model for Web Services Discovery with QoS", *ACM SIGecom Exchanges*, Vol 4, Issue 1, pp. 1-10, 2003.

[28] Cover pages: XML Schemas, Core Standards, http://xml.coverpages.org/schemas.html, Last retrieved at July 2010.

[29] C. Ding, P. Sambamoorthy, and Y. Tan, "QoS Browsing for Web Service Selection", in *Proceedings of the International Conference on Service Oriented Computing*, pp. 285-300, 2009.

[30] M. Li, J.P. Huai, and H.P. Guo, "An Adaptive Web Services Selection Method Based on the QoS Prediction Mechanism", in *Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence – Workshops*, pp. 395-402, 2009.

[31] A.K. Jain, M.N. Murty, and P.J. Flynn, "Data Clustering: A Review," *ACM Computing Surveys*, Vol 31, Issue 3, pp.316-323, 1999.

[32] E. Al-Masri, Q. H. Mahmoud, "Discovering the best web service", (poster) In Proceeding of: 16th International Conference on World Wide Web, pp. 1257 – 1258, 2007.

[33] E. Al-Masri, Q. H. Mahmoud, "QoS-based Discovery and Ranking of Web Services", In Proceeding of: IEEE 16th International Conference on Computer Communications and Networks, pp. 529 – 534, 2007.

[34] Lucene, Apache project, http://lucene.apache.org/, Last retrieved at July 2010.

[35] lp_solve reference guide, http://lpsolve.sourceforge.net/5.5/, Last retrieved at July 2010.

[36] M. Farah, and D. Vanderpooten, *"*An outranking approach for rank aggregation in information retrieval*", In Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 591 – 598,2007.

[37] PAST, PAlaeontological, http://folk.uio.no/ohammer/past/, Last retrieved at July 2010.

[38] J. Cohen, "*Statistical power analysis for the behavioral sciences*", 2nd Edition, publisher: Taylor & Francis, Inc., ISBN: 0805802835, 1988.

[39] M. Chavent, F.A.T. De Carvalho, Y. Lechevallier, R. Verde, "New Clustering Methods for Interval Data". Computational Statistics, Vol 21, Issue 2, pp. 211–229, 2006.