Selecting the Best Web Service

Julian Day
Department of Computer Science
University of Saskatchewan
julian.day@usask.ca

Ralph Deters
Department of Computer Science
University of Saskatchewan
deters@cs.usask.ca

Abstract

Web services are applications that communicate over open protocols such as HTTP using structured forms of XML such as the Simple Object Access Protocol (SOAP [18]) or Remote Procedure Calls for XML (XML-RPC [20]). The success of web services is largely based on the continuous development of standards that ensure interoperability. Among the many standards developed and widely accepted are: the Web Service Description Language (WSDL [5]), used for describing web services' syntax; and the Universal Description, Discovery and Integration protocol (UDDI [1]), often used as a discovery mechanism for dynamically finding new services. However, there have been fewer efforts to describe the interactions between clients and services. This paper focuses on augmenting web service clients as a means for determining optimal service providers. A system is discussed and analyzed for using the Resource Description Framework (RDF [14]), the Java Expert Systems Shell (JESS), WEKA [21], and the Web Ontology Language (OWL [16]) to augment web service clients. The clients can collect, report, and analyze data about their experiences with the quality of service (QoS) of web services, as well as their own system context information. The clients are able to parse and use the reported information to dynamically select the best service for their needs, to re-configure themselves to use the new service, and continue operation transparently.

Copyright © 2004 Julian Day and Ralph Deters. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

1 Introduction

XML Web services were introduced with the specification of XML-RPC by UserLand software in 1998 [20]. Web services use XML-RPC or SOAP [18] for encoding data over a transport layer, usually HTTP. Popularized by their inclusion in software packages such as Microsoft Visual Studio and Sun's Web Services Developer Pack, web services have recently seen an explosion in popularity. Their design intention is to increase interoperability between applications whose interfaces are publicly available on the World Wide Web (web) through the use of open standards. SOAP is a W3C recommendation, and XML-RPC is essentially halted: development of the specification stopped in 1999, except for a small update in 2003 to allow for Unicode strings.

The semantic web, as envisioned by Tim Berners-Lee, et al., is the markup of web data so that its semantics are machine-understandable [2]. It was conceived after the realization that despite the vast amounts of data stored online, computers could understand the semantics of virtually none of this information. One of the efforts produced by researchers in the semantic web community has been the creation of semantic markup languages [10]. The purpose of these languages is to give semantic structure to data, allowing systems to "understand" information in ways similar to how humans would. A variety of languages have been produced. Among the best-known are RDF [14], RDF Schema [3], DAML+OIL [12], OWL [16], and OWL-S [6].

The system described in this paper provides an automated web service client augmented with semantic models based on RDF and OWL.

These semantic models represent the client's system context information at the moment of interacting with a web service, as well as the results of that interaction. The clients also contain a number of reasoning engines. There have been two implementations written: the first, using JESS, is a rule-based expert system based on the RETE [9] algorithm. The second, using the WEKA [21] library, uses naive Bayesian learning to classify web services based on the reported experiences and system contexts. The semantic models and reasoners, combined with publicly accessible forums to which the clients report their experiences, allow clients to reason about which of a number of syntactically identical web services will provide them with the best service. Once the client has determined this, it can transparently switch its calls over to that service.

This paper is structured as follows: Section 2 gives a description of the problem of web service selection, and the two approaches taken in this project. Section 3 describes the system in detail. Experimentation and testing of the two reasoners is presented in Section 4, and conclusions in Section 5. Future possibilities for research are presented in Section 6.

2 Problem Definition

Selecting a web service for a client is typically a task performed by the designer of the client. The problem of web service selection by designers has been discussed by de Moor and van den Heuvel [7]. They describe the ways in which web service selection can be tied into the process of information system development. The designer would know about one or more services, and select services based on "a syntactic discovery process, a semantic matching process, as well as a pragmatic interpretation process of web service functionalities." However, the selection of a web service can also be dynamic, occuring at runtime. In this paper, a solution is proposed whereby the designer is freed at least somewhat from the selection process, allowing both the user and the system to work together to find and use the best service available. The problem is then this: given a number of user-specified, syntactically identical web services that purport to provide the same services to the client, which should the client select? An example of this can be seen in Figure 1.

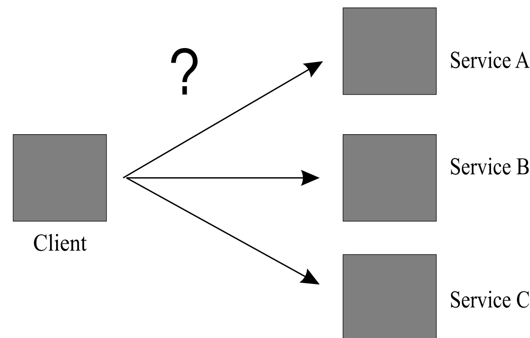

Figure 1: An illustration of the selection problem

There are a number of possible approaches to this problem. One approach could be through negotiation. If each web service could be seen as having a cost associated with it, is the cheapest service sufficient? Not necessarily. And neither is the most expensive service necessarily the best. Another possibility is that of semantic suitability. Perhaps two web services have identical syntax, but one is meant for purpose A, and the other for purpose B. If a client knew that its purpose were B, it should then select service B. A third possibility involves the past experience of others with the services. The QoS observed by past users could be examined and reasoned over, with the selected service being that which provided the best QoS in the past.

Because cost is not a certain indicator of QoS, we leave it, and instead tackle the problem of dynamic selection by examining the past experience of other clients. Semantic suitability is important, but the most suitable service using only that as a guide might not provide good QoS. While it should be pointed out that past experience is no certain indicator of present QoS, it at least provides knowledge about past QoS, while cost and semantic suitability talk only of what the service costs, or what it provides if a client can connect to it, respectively.

Liu, Ngu, and Zeng [13] approach the selection problem in this way, detailing a dynamic selection process for web services based on QoS

computation and policing. Their system uses an "open, fair, and dynamic QoS computation model for web services selection" by means of a central QoS registry. They describe an "extensible QoS model", arguing that web services are so diverse that a single, static model cannot capture all of the relevant QoS parameters, and that domain-specific parameters for one service may be completely inapplicable to others. They define a number of generic quality criteria, including execution price, execution duration, and reputation. Execution price is the monetary cost the service requestor must pay the service provider to use the service. Execution duration is simply the time it takes, in seconds, to call the service and get the result back. Reputation is a parameter that can be specified by each user for any particular web service he or she uses.

Their QoS registry is similar to the approach taken in this paper, explained in detail in Section 3. Their registry takes in data collected from the clients, stores it in a matrix of web service data in which each row represents a web service and each column a QoS parameter, and then performs a number of computations on the data, such as normalization. Clients can then access the registry, getting rankings of web services based on their individual preferences.

The method used to evaluate services in this paper is to build an evaluation function, f, which guides the client as to which service to select. f takes in the raw data about a number of web services, processes it, and then returns the best web service for that particular client. f's definition varies by implementation within the current system. In the rule-based expert system, f is currently fairly straightforward, applying a set of weights to the mean experience availability, reliability, and execution time reported by the clients. These weights have default values, but can easily be modified by the user to reflect his or her preferences. This implementation of the reasoner could, through the use of more JESS rules, be extended to reason over such factors as IP addresses of the reporting clients, allowing for trust-based weightings, or weightings based on geographic location to the web service. The use of JESS for this reasoner makes the system open and scalable to more rules than the simplistic set used.

For the naive Bayes implementation, f takes more information into account. In addition to the QoS parameters mentioned above, the reasoner also is given information about the client's system context: the processor load; percentage of memory in use; amount of bandwidth in use; and the number of running processes, as well as their names, and how much memory and processor time they were using at the time of the call.

Two approaches were considered for getting the information required for sharing experiences: server-side or client-side augmentation. A server-side augmentation would allow the web service and its designers to talk about the guarantees that could be received from it. By contrast, a client-side approach treats the web service as a kind of black box. The client knows about the operations available on the web service (as a programmer has coded a client for their definition based on a WSDL definition), but otherwise, the web service's semantics are ignored. Instead, the system captures the automated calls made by the client, and notes a number of things: whether or not the calls got through, whether or not the expected operation return type was returned, and how long, in seconds, the call took to execute.

The approach taken is to augment the client, rather than the service, and to treat the web service like a black box to which we make requests and receive responses, but know otherwise little or nothing about. This approach was selected for a number of reasons. First, there have been some efforts to use server-side information to make selection decisions ([15], [17]), but these operate on the principle of objective publishing. If a service is not entirely forthright with the accuracy of its semantic information or service guarantees, then a client could be fooled into selecting an inferior service. Second, the approach allows the clients to use any number of web services; the clients would not be limited to services marked up in this manner. Third, reasoners can be swapped in and out, so long as they use a standard interface. In the current implementation, the only difference between the system using the rule-based reasoner and the naive Bayesian-based one is a command line switch. This allows the client more flexibility than if the reason-

ing components were contained, say, as a web service. The clients act as autonomous entities. As execution continues, they report or do not report their experiences to a central web service designed to accomodate this data. It can be thought of as a kind of QoS forum, to which the clients report their experiences and look up the experiences of others. The decision of whether to report or not is made according to the specific policy the client follows. In the current implementation, clients gather information and reason over it when they start up, and then use this information to pick the web service which they want to continue with. This is illustrated in Figure 2.

It is for those reason that the clients is augmented: assuming no tampering or database flooding, correct data should be compiled and made available to the clients, who may then act on it as they wish. The no-tampering assumption may be a strong one, however, and it is discussed further in Section 6.

The four questions to be answered are as follows:

1. How can a web service be evaluated?
2. Which semantic markup language (RDF, DAML+OIL, OWL, OWL-S, etc) should be used for the representation?
3. What is needed to describe a web service QoS ontology?
4. Can such a system be built?

3 Approach

As described in Section 2, our approach to the selection problem is to build on to an existing web service client an augmentation that allows for reporting of, and reasoning on, the user's experience in using particular web services. These web services currently must be syntactically identical.

The two main parts of the system are the augmented client and the QoS forums. The web services are not modified at all, and can be seen to be separate from the rest of the system. The representation of QoS data, the semantic models, bear detailed explanation.

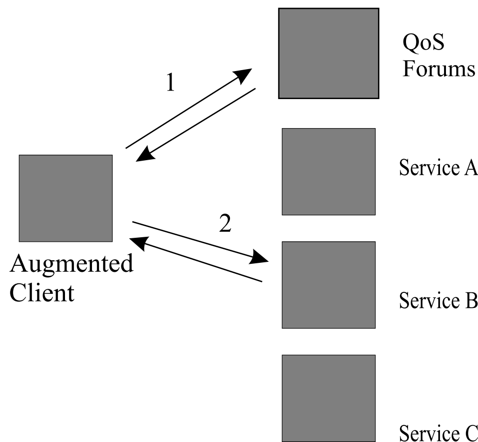

Figure 2: Selecting the Best Web Service through Reasoning and the QoS Forum

3.1 The Semantic Models

The semantic models represent interactions between the client and a web service. Each client may have zero or more models; in fact, if the client updates far less frequently than it interacts, then it may at a particular time contain a number of semantic models.

Each semantic model contains a number of properties. For the purposes of the current implementation of the system, generic QoS parameters were chosen so that they can be applied to any web service. These are:

- *Availability* measures whether or not the client can connect to the web service. It takes a value of 0 (cannot connect) or 1 (able to connect).
- *Reliability* refers to whether the operation the client wishes to perform can be performed, and whether or not it returns the type it was specified to. It takes a value of 0 (unable to perform the operation, or received a bad return type) or 1 (able to perform the operation and got the specified type in return). If a service is not reachable, the reliability is assumed to be 0 for that interaction.
- *Execution Time* is the time, in seconds, that it takes to access the service, perform the requested operation, and have a value returned.

These are parameters often discussed in the literature. Musa, Iannino, and Okumoto write that availability is "the ratio of up time to the sum of up time plus down time, as the time interval over which the measurement is made approaches infinity", and that reliability is, "the probability of failure-free operation of a computer program for a specified time in a specified environment" [19]. They define failure simply as, "the program in its functionality has not met user requirements in some way."

For availability, the time interval in our system is not a regular interval, but rather occurs with each interaction with the service. These interactions have no specified regularity. The ratio, then, can be taken by looking at a number of interactions, observing the number of times the service was available, and dividing that by the total number of interactions.

For reliability, the probability can be found by taking the ratio of the number of reliable interactions to the number of total interactions. Reliability is determined by whether the returned value of an interaction is of the type specified in the WSDL file used to initially configure the client. It is certainly possible that operations on the web service could change signatures, or even be removed, after the client has been configured, and it is upon this that we define reliability.

The three parameters were chosen because they were generic enough that they could be applied to any web service. Because they are so generic, future versions of this system could use the same basic model, but extend it to add extra generic parameters, or include domain-specific ones. Because the clients simply ignore any parameters that they do not understand, adding this extra information would not affect the reasoners of the earlier versions.

The ontology of QoS parameters was formalized into a hierarchy. Parameters are either service-specific or call-specific. Call-specific parameters are those dealing solely with the context of the call to the service itself. The reason that this category is included is because the QoS that the user experiences is not dependent on the web service alone: parts of the call to the service, from creating the SOAP or XML-RPC request to decoding the response, happen on the client side. Because we treat web services like a black box, and because we do not have access to the details with regards to creation, processing, and transmission of requests, it is difficult to decouple the time spent creating the request and sending it to the service with the time the service spends executing. Due to this, it was decided that a separate major category dealing with web service QoS would be created, and the call-specific category was born.

Service-specific parameters are those that deal with the service itself. The most generic ones of these include availability, and reliability, discussed above. However, service-specific parameters could also include things such as cost and semantic suitability. Domain-specific parameters can be added to categories that are subclasses of the service-specific category. For example, if we wanted to extend the ontology to also deal with car dealerships, parameters such as "mean cost", "newest vehicle", and so on could be added to a category perhaps called "car dealership parameters." The QoS hierarchy ontology is visualized in Figure 3.

From the beginning, our goal was to use semantic markup languages for the semantic models. The basic premise of our work, web service selection by reporting QoS parameters, has been explored at least once before (though in a different way), by Liu et al. [13]. They do not, however, make use of semantic markup, nor of client-side reasoning, which is discussed in Sections 3.3 and 3.4. Because web services are web-based – that is, they communicate over HTTP using structured XML – and because much work has already been done in developing languages to describe web-based resources, the choice was made to represent the models with one of the popular semantic markup languages.

RDF [14] was chosen to act as the underlying representation for the data, with an ontology describing the data written in the OWL Lite sublanguage of OWL [16]. Because all that was needed was a simple classification hierarchy and simple datatype restrictions, and because the external reasoning was done with the JESS engine, OWL Lite seemed more appropriate than the bulkier OWL DL or OWL Full. RDF Schema [3] was a possibility for writing the ontology. However, it does not allow for cardinality constraints on properties, which is

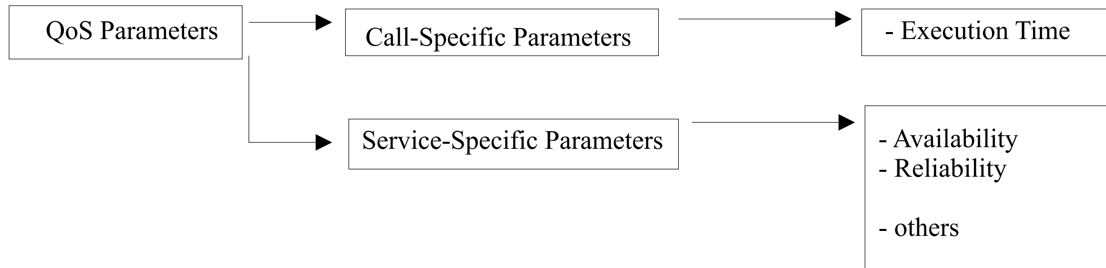

Figure 3: A hierarchy representing the ontology of web service QoS parameters

required when talking about certain parameters. At the moment, these parameters are only boolean and real numbers, which have specified mappings within XML Schema. However, for complex datatypes, this is not the case, and it would be useful to specify constraints if future versions of the system were to progress in that way. The clients could talk about various parameters and not have to worry about checking to see if the value of certain parameters falls within the required range. As such, OWL was chosen over RDF Schema as the ontology language for the semantic models.

3.2 The Augmented Client

The augmentation currently consists of three parts:

1. A *policy manager* acts as a coordinator of the client's calls with updates to the QoS system.

2. A number of stored *semantic models* each capture one interaction with the client's web service.

3. A *reasoning engine* is capable of taking in a list of semantic models, extracting information from them, and based on its analysis, selecting the service best for the user. The reasoning engine is generic, and new ones can be easily swapped in. Currently, there are reasoners based on rule-based expert systems, and naive Bayesian classifiers.

The policy manager controls such aspects as how often the client makes calls to the web service, and how often updates are sent to the central QoS forums. In a system where a user would be using the client (rather than our current automated client), the policy manager would only control how often updates were sent to the QoS forums, leaving the use of the web service to the user.

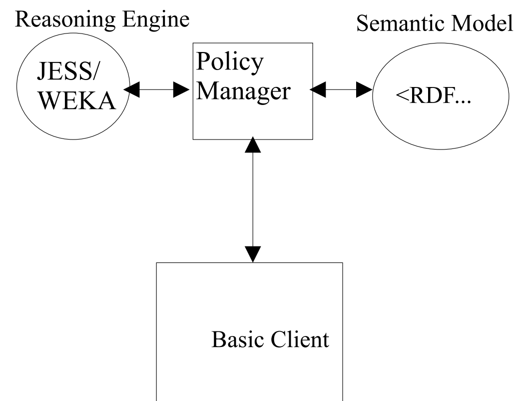

Figure 4: The Automated Client with Augmentations

When the client is first started, the policy manager connects to the QoS forum, and queries it for all the information stored about a number of user-supplied web service locations. The forum returns a vector of semantic models, each of which represents one interaction with a web service by a particular client.

This information is then passed to the reasoning engine. The engine has two versions: an expert system written in JESS, the Java Expert Systems Shell; and a naive Bayes version

```
(defrule updateBestWS
   (and (ws ?x ?y) (best-t ?z)) =>
   (try-update ?x ?y ?z))

(deffunction try-update (?x ?y ?z)
 (if (> ?y ?z) then
  (and (retract (fact-id ?bt-id))
  (retract (fact-id ?bs-id))
  (bind ?bs-id (assert (best-s ?x)))
  (bind ?bt-id (assert  (best-t ?y)))
)))
```

Figure 5: JESS code to determine the best web service

written in WEKA.

3.3 The Rule-Based Reasoner

JESS was chosen because rule-based expert systems are a well-known and well-studied area of AI, and are suitable for reasoning over a number of rules and facts. As well, the RETE algorithm has been shown to be fast and efficient for large data sets [9]. Scalability of rule- and fact-sets was an important consideration in choosing a reasoning method. While the current rule- and fact-sets are relatively small, the ability to scale up to dozens or hundreds of rules without a significant increase in performance cost was an important consideration.

The first thing the reasoning engine does is take in the information about the web services' availability, reliability, and execution time. Execution time is only factored in if the service is available. The reasoning engine then processes these into separate data structures, and then based on user-specified weights, creates a weighted sum for each:

$$w_x \frac{\sum_{i=1}^{n} x_i}{n} \tag{1}$$

In this equation, w_x is the weight for QoS parameter x. x_i is the ith reported parameter x for the web service whose data we are currently processing. From there, we have a simple evaluation: take the highest weighted sum as best. The JESS rules that perform this can be seen in Figure 5.

```
(assert (ws http://wslocation 3.78))
```

Figure 6: JESS triples that tell the expert system about the web services

```
(bind ?bs-id (assert (best-s ...)))
(bind ?bt-id (assert (best-t 0)))
```

Figure 7: Keeping track of the best service in the expert system

In these rules, slightly edited to allow for formatting, "best-s" and "best-t" represent the current best service and its associated weighted sum of its mean availability, reliability, and execution time QoS parameters. Higher values indicate that greater quality of service has been experienced in the past. However, because we are only looking at past information, and because we do so before the service is actually called, it must be pointed out that past success does not guarantee future results.

The facts for the JESS reasoning engine are generated dynamically at runtime. Because the augmented client does not know beforehand what services it may select from, the reasoning engine generates the facts based on the information provided to it from the QoS forums and the services specified by the user. The basic forms of the facts can be seen in Figure 6.

The first part of the triple is the indication that this fact is about a web service. The second part is the location of the web service. The third is the weighted sum of the mean QoS parameters as computed by the reasoning engine earlier.

The system also keeps track of the best service, as seen in Figure 7.

These allow us to keep track of the best service (with bs-id) and the best weighted sum (bt-id). The clients have built into them an initial web service location, which is initially taken to be the best service. If our information shows that another service is better than the default weighted sum (taken to be 0 in the current implementation), then the RETE algorithm will match the facts against the rules, and update the best-service and best-time tu-

ples. Once JESS has been allowed to run on these facts and rules, the reasoner extracts best service location from the best-service tuple. This is the service that the reasoner has computed to be best for the client. This value is returned to the client, and if it is different from the current web service, the client is switched over to the new service.

From there, execution continues as normal on the best web service. Semantic models are stored each time an interaction is made, and when the policy manager indicates that the system is ready to update, it contacts the QoS forums to upload the semantic models. This process is more fully described in Section 3.5.

3.4 The Naive Bayes Reasoner

The second reasoner in our system uses a naive Bayes reasoner. The reason for this was that all the system context information gathered would add much complexity to a rule-based expert system. We decided to try a naive Bayes classifier. The attributes of each instance would be the QoS parameters and the system context information, with the class of the instances to be predicted by the classifier.

Naive (or "Simple") Bayes is a method of machine learning in which attributes of each instance are assumed to be conditionally independent of one another, given the class. If an instance has n attributes $< a_1, a_2, \ldots, a_n >$, then that instance is classified to a possible class c of a set of classes C by the equation:

$$c = \text{argmax}(\forall c_j \in C, P(c_j) \prod_{i=1}^{n} P(a_i|c_j)) \quad (2)$$

The assumption of conditional independence is a strong one, and often violated in practice. However, the algorithm is optimal when the assumption holds. But as Domingos and Pazzani point out, naive Bayes performs very well in practice even when strong attribute independencies are present [8]. Because of this, naive Bayes was selected as an alternative reasoner to complement the rule-based one.

Each web service interaction is an instance given to the naive Bayes classifier. The attributes of each instance are not conditionally independent given the class. There are a number of attributes used: available (true, false); reliable (true, false); execution time (real); processor load (real); total memory used (real); percentage of memory used (real); bytes sent/received/total per second (real/real/real); and the number of processes (numeric). The classifier attempts to classify the web services into the categories <terrible, poor, acceptable, good, excellent>. Web services are selected from the best available class, and if there are multiple services in a class, then the reasoner assumes that they are equally good, and selects one arbitrarily.

The naive Bayes reasoner follows the same procedure as the rule-based one, receiving data about a number of user-specified services upon startup. And as with the rule-based reasoner, semantic models representing the interaction are sent to the forum system after each call. The difference between the two systems is that in the naive Bayes implementation, system context information is also included in the model. Interactions from both types of clients can be used by either, as each simply ignores any information it is not looking for. In the naive Bayes implementation, this extra information is used when attempting to classify new instances.

The training data is built from the past experiences. Most of the attributes are known, and the classes are determined dynamically. The system has a function which takes as its input all the attributes of a web service instance, and produces as an output the class it belongs to. Currently, the QoS parameters have the most weight in this function, though context information is also taken into account.

The data is not complete. Because there were two iterations of the system – the first used the rule-based expert system, and the second the naive Bayesian classifier – and because system context information was only gathered during the second iteration, there are gaps in the data sets. Not all instances will have system context information, instead missing those values. The naive Bayes reasoner is able to handle this.

3.5 The QoS Forums

The QoS forums are a web service designed to store information about clients' experience

with particular web services. A web service is wrapped around a database which stores semantic models reported to the service.

The semantic models, as explained in Section 3.1, are a combination of RDF and OWL. Besides these, the forums also keep track of the date, the sender's IP address, and the web service's location. The first two are tracked because if the system is expanded in the future, they could provide useful information in addition to what is already in the semantic model. The reasoner could find out, for example, all clients reported that a web service x was not available on a certain day, and perhaps conclude that this is an exception rather than the rule. Alternatively, if one IP range keeps reporting experiences that are wildly different from every other client, and at a much higher rate, it might bias its calculations against that IP range, concluding that it is flooding the forums. At the moment, however, the reasoners are not capable of making realizations such as this. The last feature kept, the web service's location, is just so that the database can more easily return ranges of semantic models, rather than all of them.

4 Experimentation

To show that this system could be more than just a simple proof-of-concept, we needed to test it on a real system. I-Help is a system designed by the ARIES laboratory (http://www.cs.usask.ca/research/ research_groups/aries/) at the University of Saskatchewan. Its purpose is to allow students to find help for problems relating to their computer science courses [11]. Recently, it has been extended into web services, allowing for posting, reading, and other operations without having to use the traditional webpage-based interface . There are two questions to answer: first, can the system, given a number of syntactically identical web services, select the best from among them. And second, what sort of performance costs are associated with the system, what is its overhead, and how scalable might it be.

In addition to the main I-Help web service, two replicates of the service were created.

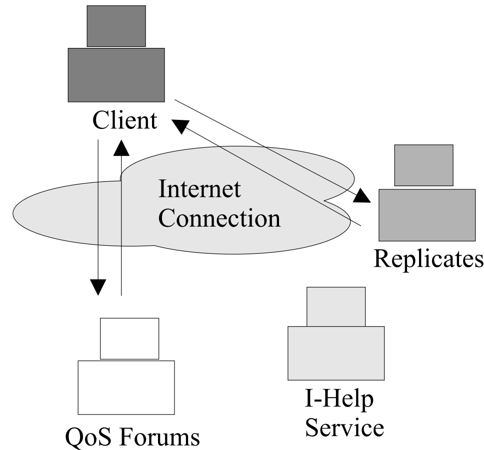

Figure 8: Interacting with the QoS forums and the I-Help web services

These reside at different locations on the web, but have identical functionality. The replicates vary from the main service in the amount of time they take to process requests. Clients were generated to use each service, and 500 semantic models for each service were submitted to the QoS forums through the clients' interactions with the I-Help services.

An illustration of this process is shown in Figure 8.

4.1 Dynamic Service Selection using the Rule-Based Reasoner

To answer the questions above, we first tested the system using the rule-based reasoner.

The statistics for the main service and its two replicates are found in Table 1. $\bar{a}$, $\bar{r}$, and $\bar{e}$ refer to the mean availability, reliability, and execution time, respectively. σ_a, σ_r, and σ_e refer to the standard deviation of the availability, reliability, and execution time.

Each client gathered data over the course of about six hours. As can be seen from the data, no service had 100% uptime, with Copy B having the highest uptime at just under 99%. There were a number of crashes as the clients made requests quickly of the service and its replicates. Whenever this happened, the services were promptly rebooted. Thus, the imperfect availability and reliability.

	Main Service	Copy A	Copy B
$\bar{\mathbf{a}}$	.814	.960	.986
$\bar{\mathbf{r}}$	.814	.960	.986
$\bar{\mathbf{e}}$	5.70	6.09	9.61
σ_a	.389	.195	.118
σ_r	.389	.195	.118
σ_e	1.25	1.38	1.55

Table 1: Data Gathered About The Web Services

Class	Prior Probability
terrible	0.08
poor	0.08
acceptable	0.34
good	0.35
excellent	0.16

Table 2: Learned prior probabilities for each class

Code was used on the client side to simulate network delays and server-side slowness, as well as connection problems. The first service was the quickest, followed by the first and second replicates. However, the first service crashed the most, which is reflected in its availability and reliability.

When the default weights were defined, execution time was deemed to be most important; however, availability and reliability were not to be sacrified either. The default weight for both availability and reliability is 10; the default weight for execution time is -2. These values may all be changed by the user.

With these weights defined, the service the client should have reasoned to be best, given the reasoning rules from Section 3, is Copy A, as the weighted sums (taken by applying equation 1 to the data in Table 1) for the original service and its replicates are 4.875, 7.021, and 0.495, respectively. When the client ran, gathered the data, and ran it through the expert system reasoner, it found that Copy A provided the best service, and transparently switched its operation to that replicate. The client had originally been configured to run on the main service.

4.2 Web Service Classification and Selection with the Naive Bayes Reasoner

The second iteration of the system used naive Bayes as its core reasoner. Because classification is the purpose of naive Bayes, the experiment was to see if, given a set C of classes = <terrible, poor, acceptable, good, excellent>, whether the reasoner could classify web service interactions, and use these as the basis for selecting the best service. The selection process was to take a web service from the highest available class. So if there were a good and an excellent service, the reasoner would tell the client to pick the excellent one; and if there were two excellent services, the client would select a service from the two excellent ones, and tell the client to use that.

The same test data was used for both reasoners. The data from the I-Help web services and its replicates is gathered on startup, fed to the naive Bayes reasoner, which then converts it to ARFF format [21], and reads it back in. Each class of web service is represented in the data set, though not equally. Table 2 gives the distribution of all the classes within the data set.

To classify the data, the process of 10-fold stratified cross-validation is used. For a discussion of cross-validation and stratification, see Witten and Frank [21]. Because of the inherent randomness in determining the training and testing sets, this process is repeated ten times, with the mean taken as the result. The results can be found in Table 3.

As can be seen from Table 3, the results are generally very good. Even with gaps in the data set, the rate of correct classification is still above 90%, and the error rates are quite low, too.

The kappa statistic, described by Carletta [4], is a measure of class accuracy within a classifier. It corrects for expected chance agreements between classes:

$$\kappa = \frac{P(A) - P(E)}{1 - P(E)} \quad (3)$$

In the above equation by Carletta, P(A) is

Statistic	Values	
Correctly classified instances	1387	92.4667%
Incorrectly classified instances	113	7.5333%
Kappa statistic	0.8971	
Mean absolute error	0.0526	
Root mean squared error	0.1508	
Relative absolute error	17.7259%	
Root relative squared error	39.5026%	
Total number of instances	1500	

Table 3: Results of 10-fold stratified cross-validation, run 10 times

A	B	C	D	E	← Classified As
118	0	1	1	0	A = terrible
1	108	7	0	0	B = poor
0	26	440	38	0	C = acceptable
0	0	14	493	13	D = good
0	0	0	12	228	E = excellent

Table 4: Confusion matrix for the naive Bayes classifier on the web services data set

the proportion of times that the classes agree, and P(E) the proportion of times that the classes would be expected to agree by chance. Carletta, quoting Krippendorf, writes that, "researchers generally think of $\kappa > .8$ as good reliability, with $.67 < \kappa < .8$ allowing tentative conclusions to be drawn." With this in mind, the kappa statistic for the naive Bayes web service classifier is 0.8971, well above Krippendorf's 0.8. So even with corrected chance agreements between classes, the accuracy is close to the accuracy of correctly classified instances.

The confusion matrix of Table 4 is a breakdown of all the classifications. So as an example, 228 of the 240 "excellent" instances were correctly classified as "excellent", while 12 were misclassified as "good". No class had a perfect classification rate; the best was the "terrible" class, while the class with the highest error rate was the "acceptable" class. 64 of 504 total "acceptable" instances were misclassified, an error rate of nearly 13%. Still, the rest of the classes fared better, with the next highest misclassification rate being just over 5% (the "good" class).

Witten and Frank write that the mean absolute error is an alternative to mean-squared error and root mean-squared error. The latter two tend to "exaggerate the effect of outliers – instances whose prediction error is larger than the others – while absolute error does not have this effect: all sizes of error are treated evenly according to their magnitude" [21]. The classifier's mean absolute error is quite low. However, there is currently nothing to compare these to, since only one type of classifier has been implemented. Further research could include other classification methods, along with a comparison of error rates; this is more fully discussed in Section 6.

4.3 Scalability of the Clients

As seen in the previous section, the proof-of-concept works as intended. The next question, though, is to determine how well these clients could scale up in usage.

To judge the scalability of the clients, there are four identified parameters to be studied in terms of resource consumption. The first is the *query cost*. The query cost is the cost incurred when the system queries the QoS forums for information on a particular web service. The

second is the *analysis cost*, which is the cost of taking the data provided by a query, parsing it in such a way that the reasoner can make sense of it, and once this is done, running the reasoner and then asking it for the best service. The third is the *monitoring cost*, which describes the cost of monitoring calls to the web service, and preparing a semantic model based on the findings. The fourth is the *reporting cost*, which specifies what is required to report the monitored data to the QoS forums.

To judge these parameters, two factors were considered: first, how often they are used by the client; and second, where the main bottleneck lies. The result can be seen in Table 5.

The number of times these parameters are used is due to the design of the system. Currently, selecting the best web service is a one-time event done just after the client starts up, and the analysis, done after the data arrives, also occurs only once. However, the bandwidth and memory requirements can be rather steep. In our test cases, 500 semantic models for each service were stored. The bare XML representation of the models (not the objects in the system, which would have a greater size) had a mean size of 21.7 KB. 500 such objects would then take at least 10.86 MB if only their XML structure were stored. But when each is wrapped in a Java object, and all of those objects are stored within a vector, the memory costs suddenly jump considerably, in the order of tens of megabytes. This was not a huge amount on the main development machine; however, transferring that amount of data over the network is not insignificant. As the amount of interactions increases, the amount of data sent over the network will only increase, as will the memory consumption by the client. This is a problem, and an issue that will need to be addressed in future versions of the system.

Monitoring and reporting occur variably. Monitoring is handled by the policy manager, and occurs each time the client attempts to make a call on the target web service. Despite this, though, they tend to require fewer resources than do query and analysis. Monitoring simply creates a new semantic model after a call is made, and queues it in the unsent list of semantic models. The only varying part of the models is all the information about all the processes. Thus, the size (and creation cost) of the models is linear by the the number of processes on the client machine at call-time. Because all the other information stored (the number of QoS parameters, the amount of system context information not dealing with processes) is constant, the creation of models takes a fairly constant amount of time and memory, except for dealing with process information.

Reporting occurs whenever the policy manager says to (at the moment, this occurs at the same time interval as calling the service). It takes no more memory than does monitoring, but bandwidth is a potential bottleneck, especially if large numbers of semantic models have been stored but not sent.

While the concept implemented in the system could be scalable, there are a number of issues that need to be addressed before the implementation would be. There is a large memory and bandwidth overhead when reasoning about the best service, and that will only grow with increased numbers of interaction snapshots within the QoS forums. However, the rest of the interactions should go smoothly, as the costs of monitoring and reporting can be fairly small, especially if the default policy of "report as soon as the interaction is observed" is followed.

5 Conclusions

The process of selecting a web service does not have to be a static, design-time decision. Instead, it can occur dynamically, with web service clients deciding which service to use. In this implementation, web service clients reasoned over the experiences of others clients as to which service to select. The experiences concerned generic QoS parameters, including availability, reliability, and execution time; and also system context information gathered at the time of the call, such as processor load, memory usage, and bandwidth information. The advantage of this approach over unaltered clients is that the augmented clients can avoid potentially bad situations, and if a service is unreachable, attempt to find a better one.

To represent a model of its knowledge, the

Cost	Times Used	Bottleneck
Query	constant	memory, band.
Analys	constant	memory
Monitor	num. calls	memory
Report	by policy	bandwidth

Table 5: Bottlenecks of Querying, Analysis, Monitoring, and Reporting

clients in the implementation used a combination of RDF and OWL. These experiences were stored in a central forum system available as a web service to any interested party. To reason about the best service, a JESS-based expert system, and a WEKA-based naive Bayesian classifier, were used to analyze the data received from the QoS forums, and make a recommendation to the client. When this was done, the client was automatically re-configured to use the new web service.

This approach, while generally good, only suffered from a few issues relating to scalability. The clients operated on an "all knowledge is potentially useful" assumption, which meant that they requested all of the interaction snapshots from the QoS forums. If the forums had hundreds or more interactions stored about a particular web service, the analysis process would take a noticeable amount of time as the data from the semantic models had to be first parsed by whichever reasoner the client was currently using, and then reasoned over. However, building up the reasoner is a one-time cost incurred when the client starts up, and the variable-time occurances of both monitoring and reporting had constant costs associated with them.

The use of an expert system, as described earlier, was good, but suffered from a few drawbacks. The user was able to specify the weights for the various parameters, but the idea of changing numbers to select a different web service is perhaps unintuitive unless one understands the process of the rule-based reasoner intimately.

The naive Bayes classifier performed well, with a classification accuracy rate of around 92.5%. The expansion of the system to add the client's system-specific context information at the time of the call added a number of new attributes for the reasoner to consider, such as the client's CPU load and memory usage. The classifier was able to deal with this data and deal with gaps in the old data. The use of machine learning algorithms for service selection seems worthwhile, with potential for comparisons between different methods.

Because the clients have a say in which service they select, there is a sort of self-organization that occurs. While this is interesting, there are potential drawbacks: the expert system always picks the highest weighted sum, so all clients using the expert system will always pick that service, possibly leading to a lowered QoS due to the inherent determinism in that reasoner. This drawback is less likely to happen within the naive Bayes reasoner, however. Because the classifier classifies services into a small number of discrete classes, a number of services can share the same class. The classifier does not care which service within the best possible category is chosen, as it picks at random, and there is therefore a lower risk of such behaviour.

Overall, the approach of reasoning over experiences seems to be useful for selecting among web services. The approach of a client-side reasoning engine appears solid, with both implementations, using only the most generic QoS parameters, able to pick among the best services, leaving the rest alone. The QoS ontology, written in RDF and OWL Lite, had two broad categories into which all the parameters fell, and could be easily extended to add new domain-specific parameters.

6 Future Work

In this paper, the web service selection problem was tackled purely from a past-experience standpoint. There were other approaches that,

for the purposes of time, were neglected: for instance, negotiation, or semantic suitability. In the future, the system could be extended to have full OWL-S descriptions for the web services to fully talk about what they offer, and have the clients take that into consideration, too. In the case where there is a cost associated with a web service, the clients could perhaps also apply negotiation strategies. It would be interesting to consider a hybrid approach of all three strategies, with clients attempting to select a service with good previous QoS experience, semantic suitability, and a low cost.

At the moment, the QoS forums are currently a stand-alone web service. There are currently no mechanisms in place to prevent flooding of the database with bogus semantic models, as there are in Liu, et al [13]. One possibility would be to set up a peer-to-peer-like forum system in which each client contains its own database and stores its own experiences, but no-one else's. Clients could then operate like peers on a P2P network, sharing requests for information on various web services. As well, the semantic models contained within could be updated to contain more generic, or domain-specific, QoS parameters.

There is the matter of bandwidth. XML-based languages are an excellent basis for description, but suffer from all the extra space that the tags take up. There are a number of possible extentions that could be done to speed up the process. First, instead of receiving all data from the QoS forums and storing none of it, the client could instead store the data it receives, and request deltas. In this way, all results newer than a certain time and date could be spent, potentially saving much data from being sent over the wire. Another possible extension to the forum system could be sending the data in a compressed format, if it were found that it would be faster to compress the data, send it, and decompress it, rather than sending it uncompressed.

The two iterations of the system were as follows: first was a rule-based expert system, and then a naive Bayes classifier. Being able to describe the attributes of a problem, and then encode instances and provide those to a classifier, allowed a much greater degree of flexibility than by just using an expert system. The expert system was fast, but there was not a lot of expert knowledge about web services encoded. The use of classifiers allowed us to identify classes and attributes, collect the data, and have the classifier learn the relationships between the classes and attributes. A third iteration, in which we might try different learning schemes (such as decision trees using C4.5, etc.) and compare the results, could be interesting.

Acknowledgements

We would like to acknowledge Chris Brooks, who set up, and providing the interface to, the I-Help web services. We would also like to thank the following people, whose discussions with us have been extremely valuable: Mike Horsch, Chris Brooks, Kamal Elbashir, Tay Hock Keong, Mike Winter, and Chris Worman.

About the Authors

Julian Day is a graduate student in the MADMUC laboratory at the University of Saskatchewan. His e-mail address is julian.day@usask.ca.

Ralph Deters can be reached at deters@cs.usask.ca.

References

[1] Tom Bellwood, Luc Clement, and Editors Claus von Riegen. Uddi version 3.0.1. http://uddi.org/pubs/uddi_v3.htm.

[2] Tim Berners-Lee. Semantic web road map. http://www.w3.org/DesignIssues/Semantic.html, September 1998.

[3] Dan Brickley and R.V. Guha. Rdf vocabulary language description 1.0: Rdf schema. http://www.w3.org/TR/rdf-schema/.

[4] Jean Carletta. Assessing agreement on classification tasks: The kappa statistic. *Computational Linguistics*, 22(2):249–254, 1996.

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana.

Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

[6] The OWL Services Coalition. Owl-s: Semantic markup for web services. http://www.daml.org/services/owl-s/1.0/owl-s.pdf.

[7] Aldo de Moor and Willem-Jan van den Heuvel. Web service selection in virtual communities. In *37th Hawaii International Conference on System Sciences*, 2004.

[8] Pedro Domingos and Michael J. Pazzani. Beyond independence: Conditions for the optimality of the simple bayesian classifier. In *International Conference on Machine Learning*, pages 105–112, 1996.

[9] Charles Forgy. Rete: A fast algorithm for the many patterns/many objects match problem. *Artificial Intelligence*, 19(1):17 – 37, 1982.

[10] Yolanda Gil and Varun Ratnakar. A comparison of (semantic) markup languages. In *Proceedings of the* 15th *International FLAIRS Conference*, 2002.

[11] Jim Greer, Gord McCalla, Julita Vassileva, Ralph Deters, Susan Bull, and Lori Kettel. Lessons learned in deploying a multi-agent learning support system: The i-help experience. In *Proceedings of AIED*, pages 410 – 421, 2001.

[12] Ian Horrocks, Frank van Harmelen, and editors Peter Patel-Schneider. Daml+oil. http://www.daml.org/language/.

[13] Yutu Liu, Anne Ngu, and Liangzhao Zheng. Qos computation and policing in dynamic web service selection (to appear). In *Proceedings of the WWW 2004*, May 2004.

[14] Frank Manola and Eric Miller. Rdf primer. http://www.w3.org/TR/rdf-primer.

[15] E. Michael Maximilien and Munindar P. Singh. Agent-based architecture for autonomic web service selection. In *Workshop on Web Services and Agent-based Engineering at Autonomous Agents and Multi-Agent Systems*, 2003.

[16] Deborah L. McGuinness and Frank van Harmelen. Owl web ontology language overview. http://www.w3.org/TR/owl-features/.

[17] Sheila McIlraith, Tran Cao Song, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems*, 16(2):46 – 53, March/April 2001.

[18] Nilo Mitra. Soap version 1.2 part 0: Primer. http://www.w3.org/TR/soap12-part0/.

[19] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software Reliability, Measurement, Prediction, Application*. McGraw-Hill Book Company, 1987.

[20] Dave Winer. Xml-rpc specification. http://www.xmlrpc.com/spec.

[21] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann Publishers, 2000.