A Framework for Autonomic Web Service Selection

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Departmentof Computer Science
University of Saskatchewan
Saskatoon

By
Julian Clark Day

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building
110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan
Canada
S7N 5C9

Abstract

Web services are a form of distributed computing. As applications accessible over standard internet protocols, web services allow access to disparate computational resources. Recently, with an increased commoditization of web services, there has been a greater interest in the problem of selection. If a web service client can be configured to use one of a number of different web services, which should it select? In this thesis, an approach based on examining the past quality of service (QoS) parameters of similar clients is presented. Standard web service clients are augmented to report their experiences, and can reason over both these and the experiences of others using a number of formal techniques, thereby arriving at an informed decision.

Acknowledgements

I would like to thank my supervisor, Dr. Ralph Deters, for his supervision and ideas through the course of my M.Sc. I would also like to thank my thesis committee for their suggestions: Dr. Ralph Deters, Dr. John Cooke, Dr. Michael Horsch, and Dr. Christopher Zhang. Finally, I would like to acknowledge Christopher Brooks, who set up and maintained the I-Help web services, which were used in earlier versions of this research.

To all my family.

Contents

List of Tables

List of Figures

List of Abbreviations

CLIPS	C Language Integrated Production System
DAML	DARPA Agent Markup Language
DAML+OIL	DAML with the Ontology Inference Layer
DARPA	Defense Advanced Research Projects Agency
FIPA	Foundation for Intelligent Physical Agents
FIPA ACL	FIPA Agent Communication Language
GOLOG	alGOl in LOGic
HTTP	Hypertext Transfer Protocol
JESS	Java Expert Systems Shell
KQML	Knowledge Query Manipulation Language
KIF	Knowledge Interchange Format
OWL	Web Ontology Language
OWL-S	Web Ontology Language for Services
P2P	Peer-to-Peer
QoS	Quality of Service
RDF	Resource Description Framework
RDFS	Resource Description Framework Schema
RETE	Latin for "Net"
SOAP	Simple Object Access Protocol
TTL	Time to Live
UDDI	Universal Description, Discovery, and Integration Protocol
UML	Unified Modeling Language
W3C	Word Wide Web Consortium
WSDL	Web Service Description Language
XMI	XML Metadata Interface
XML	eXtensible Markup Language
XML-RPC	XML-based Remote Procedure Calls

Chapter 1

An Introduction to Web Services and the Semantic Web

Web services are applications whose interfaces are exposed over open protocols such as the Hypertext Transfer Protocol (HTTP [1]) and the Extensible Markup Language (XML [2]). Clients and services communicate through the use of XML-based protocols: the Simple Object Access Protocol (SOAP [3]) and XML-based remote procedure calls (XML-RPC [4]) are used for encoding the requests to and responses from the service, while the Web Service Definition Language (WSDL [5]) is often used to provide a syntactic description of the service itself. The Universal Description, Discovery, and Integration (UDDI [6]) protocol allows for dynamic discovery of web services. The intention of web services is that, through open standards such as the ones described above, there will be increased interoperability between applications developed with heterogeneous languages.

Web services are a useful computational abstraction. Because the interfaces to the services are well-specified by standard languages, web services and their clients can be written in different languages. This language-independence provides greater freedom for both the service owner, as well as developers of clients. An example of web services are the Google Web APIs, found at http://www.google.com/apis/. Whatever language the services are written in is irrelevant to the designer of a program that uses these services; all the designer has to do is make calls to the web service's search or spell-checking functions (formally called "operations" in WSDL), and all the details are handled automatically.

One of the shortcomings of web services is that WSDL, SOAP, and so on, only describe a service's syntax - they cannot describe the semantics. To deal with this, researchers have begun to integrate concepts such as machine-understandable semantics from the semantic web [7, 8, 9]. The semantic web is an extension to the current web in which data is given well-defined meaning so that it can be automatically parsed and understood by machines. Tim Berners-Lee writes that "The Web was designed as an information space, with the goal that it should be useful not only for human-human communication, but also that machines would be able to participate and help" [10]. The problem, he writes, is that most of the data on the web is geared towards people, not programs, and the structure of the data is not evident to automatic processes.

The main innovation from semantic web research that has wide applicability to web services is the group of so-called *semantic markup languages*. These languages allow one to create ontologies of specific domains, and give well-defined meaning to data. Computer programs can then use the ontology to not only parse, but understand the data in terms of these ontologies. The most basic of the semantic markup languages is RDF, the Resource Description Framework [11]. Built on this are RDF-Schema [12], which allows for simple vocabularies to be built in RDF; OWL [13], the Web Ontology Language, which allows for more expressive vocabularies than does RDF-Schema; and OWL-S, the Web Ontology Language for Services [8], which is an ontology language for the domain of web services.

Semantic markup languages were first used in web services to define the semantics of the service, as well as for automated service composition. McIlraith and Son adapt the GOLOG [14] logic language for use with the semantic web, integrating it into the DAML [15] markup language to allow for agent-based service composition [16]. They then extend their work to a full, working system based around constraints and queries [7].

Semantic markup languages offer researchers possibilities to extend and enhance web services, allowing for greater possibilities than with just WSDL and SOAP alone. Clients, by using ontologies, are able to use a common set of terms and relationships to make meaning of large amounts of data.

1.1 Web Service Selection

Web services are a form of distributed computing that allow access to disparate computational resources using standard protocols such as HTTP. Part of the promise of web services is interoperability between languages. Through the use of standard web service protocols such as WSDL [5], SOAP [3], and XML-RPC [4], the hope is that a programmer can easily harness the computational resources of services, whether written in Java, C#, Python, or any other language supporting these standard protocols.

The first of these protocols is the Web Service Description Language (WSDL), a syntactic description language for web services based upon XML. WSDL allows a service designer to describe exactly what his or her service offers in terms of inputs, outputs, and operations. WSDL enables applications in different languages to communicate using the common language of XML.

An example of WSDL can be seen in Figure 1.1. It specifies a simple "echo" web service, which has a single operation, shown in bold text as the "<**operation**>" tags. This operation takes in a string, and because it is an "echo" service, it presumably returns the string it was given (both strings are in boldface, and the messages in which they are wrapped are italicized for emphasis). We cannot, however, know this for certain by just looking at the WSDL definition. WSDL specifies only syntax, and not semantics.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="SimpleService"
   xmlns="http://schemas.xmlsoap.org/wsdl/"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:ssns="http://www.ecerami.com/wsdl/SimpleService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">

< message name="echoRequest">
     <part name="firstName" type=" xsd:string0"/>
< /message>
< message name="echoResponse">
      <part name="greeting" type=" xsd:string"/>
< /message>
<portType name="Simple_PortType">
      < operation name="echo">
         < input message="ssns:echoRequest" />
         < output message="ssns:echoResponse" />
      < /operation>
</portType>
<binding name="Simple_Binding" type="ssns:Simple_PortType">
      <soap:binding style="rpc"
         transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="echo">
          <soap:operation soapAction="echo"/>
          <input>
             <soap:body
                encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                namespace="urn:examples:simpleservice"
                use="encoded"/>
          </input>
          <output>
             <soap:body
               encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               namespace="urn:examples:simpleservice"
               use="encoded"/>
          </output>
      </operation>
</binding>
<service name="Simple_Service">
      <documentation>This is a simple WSDL file.</documentation>
      <port binding="ssns:Simple_Binding" name="Simple_Port">
         <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
      </port>
</service>
</definitions>
```

Figure 1.1: A WSDL Definition for a Simple Web Service

```
<operation name="foo1" parameterOrder="String\_1">
  <input message="ns:sampleIF\_foo1" />
  <output message="ns:sampleIF\_foo1Response" />
</operation>
<operation name="foo2" parameterOrder="">
  <input message="ns:SampleService\_foo2" />
  <output message="ns:SampleService\_foo2Response" />
</operation>
```

Figure 1.2: An Excerpt of Two Operations from a WSDL File

The second and third protocols mentioned above, SOAP and XML-RPC, are used to encode messages. The Simple Object Access Protocol (SOAP) and XML-based remote procedure calls (XML-RPC) encode the request/response pairs sent to the web service by the client. Each offers facilities for specifying operations, parameters, and data types.

The recent increase of interest in web services has led to the development of many similar services by people all over the world. This increase of the commoditization of services leads to an interesting question: can a client be written to select autonomously from a number of similar web services, switching to the best one each time?

At the present time, the selection of a web service for a particular client is usually a design-time decision made by the service's implementors, as evidenced by the major tools available at the time of writing: most view selection as static, generating stubs for the programmer for a particular web service. There has also been research in web service selection as part of the software development process [17]. However, there is no reason that selection must be static; a dynamic approach could be used, with clients connecting to new services dynamically at runtime.

Consider the list of publically available services at XMethods (http://www.xmethods.net/), a site that hosts listings of web services by third-party developers. As an example, there are a number of public services that provide weather forecasts, given a ZIP code. Is there any reason to be tied to a particular service, since a number of these provide identical functionality? The quality of the service, not the location, should be the most important factor for a designer of a client. If all these services provide the same functionality, giving the web service clients the ability to switch to the best service increases the program's reliability, while still performing the task for which it was originally intended.

Given the dynamic nature of web services, that they can come and go; given that they can interact both with clients and other web services; and given that the experiences of two different clients with a particular service might be extremely different due to geographical distance, system issues, or a number of other factors; how can a web service client seek out and use a service which is likely to give it the best Quality of Service (QoS) out of a number of similar services? This problem, client-centric web service selection, is at the heart of this thesis, and is illustrated in Figure 1.3.

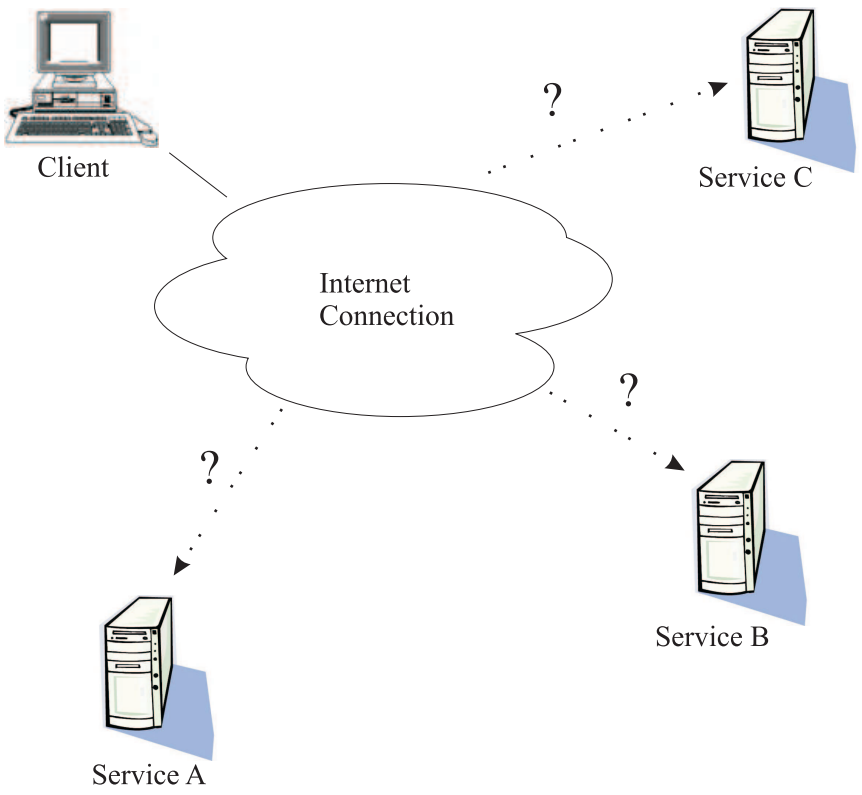

Figure 1.3: A Client About To Select Between Similar Web Services

Chapter 2

Literature Review

When designing a framework to select from a number of similar services dynamically (at runtime), there are several areas of computer science and computing that offer a number of useful tools and concepts. The base for the research, in terms of communication and operation, is handled by web and web services research technologies. In particular, standard technologies such as XML, SOAP, WSDL, RDF, and OWL are employed to create the web service, and facilitate communication between the service and clients. The selection mechanisms of the clients draw from research in the area of web service selection, with the specifics of the mechanisms coming from well-studied areas of Artificial Intelligence: rule-based expert systems and naive Bayesian classifiers. Finally, autonomic computing provides the ideas of component self-optimization and self-healing, as well as policies, to help provide structure and manage the system as a whole.

2.1 Web Services

Web services were first discussed by the W3C (http://www.w3c.org) in 2000. While XML-RPC had been finalized and a W3C mailing list (xml-dist-app@w3.org) created for XML protocol discussion in 1999, it was not until 2000 that the W3C started to create plans for XML protocols. In February of 2000 they created an interim plan for XML protocols, and in September of that year, they began investigating using XML for a protocol to facilitate application-to-application messaging. A year and a half later, in January of 2002, they extended this by launching the Web Services Activity. Its aim was to extend the scope of the ongoing XML investigation into all aspects of web services. The goal of this research, as they put it, has been "to design a set of technologies fitting in the Web architecture in order to lead Web services to their full potential" [18].

These activities have produced a number of results. One of the most important was a common protocol on which most web services are based: SOAP. The first version of SOAP, 1.1, was detailed in a W3C note from May 8, 2000 [19]. The specification was authored by a number of people from a number of large companies such as IBM and Microsoft. The most recent version of SOAP, 1.2, is a W3C recommendation as of June 24, 2003.

Also part of the W3C are the Web Services Description Working Group, and the Web Services

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
            xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#">
  <contact:Person rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:fullName>Eric Miller</contact:fullName>
    <contact:mailbox rdf:resource="mailto:em@w3.org"/>
    <contact:personalTitle>Dr.</contact:personalTitle>
  </contact:Person>
</rdf:RDF>
```

Figure 2.1: RDF Code Describing the Creator of a Web Resource

Choreography Working Group. The former group is largely responsible for WSDL, while the latter seeks to create a language to describe the relationship between web services.

Much of the material drawn upon by the industry for web services has been standardized by the W3C, including most of the core underlying technologies: XML, upon which the encoding mechanisms of SOAP and XMLRPC are based; SOAP, and WSDL. However, one area that the W3C has been pursuing since the late 1990s has been the semantic web, which holds much promise for the future of web services.

2.1.1 The Semantic Web and Semantic Web Services

The W3C published its semantic web roadmap [10] in 1998. The promise of the semantic web is an approach that "instead develops languages for expressing information in a machine processable form," rather than the current web, which is geared more to human-to-human interaction.

Machine understandable information, according to the roadmap, is presented in a basic assertion model: the Resource Description Format, or RDF. The basic model contains assertions (Subject-Verb-Object statements), and quotations, which are assertions about assertions [10]. An example mentioned earlier is the fragment of RDF code about the creator of a resource. In that example, one such S-V-O statement would be "Eric Smith is-full-name-of-creator-of resource x". Because of this very basic model, the language is limited in what it can do. The authors point out that it does not contain any concept of negation or implication; these and other limitations are addressed in the languages that have been built on top of RDF: DAML+OIL [15], OWL [13], and OWL-S [8]. RDF is built on top of XML. An example of RDF markup can be found in Figure 2.1.

RDF Schema is a language for describing RDF vocabularies, and it provides a way to talk about RDF resources in known terms [12]. Though it provides a mechanism for writing ontologies, it is rather minimal. It allows for classes, subclasses, and inheritance, but does not allow for element cardinality (necessary if talking about booleans, real numbers, and so on).

DAML, the DARPA Agent Markup Language, is a language built on top of RDF. In addition to the advantages provided by RDF, DAML+OIL (the DAML language plus the Ontology Inference Layer) provides a number of useful constructs. In terms of datatypes and values, provides constructs

such as bounded lists, basic datatypes (through the use of XMLSchema), and enumeration of data values. For logic and set theory operations, it provides negation (through the use of the <daml:ComplementOf> tag), disjunctive and conjunctive classes, as well as necessary and sufficient conditions for membership, and inverse and transitive properties. RDF and RDF Schema have none of these things. RDF and DAML are well-supported in packages such as HP Labs' JENA library (http://jena.sourceforge.net/), which provides classes for RDF, DAML, and other languages. A good analysis of the features of DAML+OIL, as well as how it compares to other semantic markup languages, has been done by Gil and Ratnakar [20].

OWL-S, a language that has been getting much attention lately, is the Ontology Web Language for Services, and at the time of writing is at version 1.0. For all previous versions, it was known as DAML-S. The aim of OWL-S is to define an ontology for web service discovery, composition (using multiple services together in such a way that it appears as only one to the user) and interoperation, invokation, and execution monitoring [8]. The authors admit that no work has yet been made on the last point, but that it should be included anyway because they felt it was important. Their structuring of the ontology is motivated by the need to provide what the service requires of the user, and what it provides for them; how the service works; and how it is used [8].

The service *profile* describes what the service does. This is interesting because it provides the sort of information needed by, as the authors put it, "a service-seeking agent" [8]. Traditionally, this sort of information would be semantic-less and stored somewhere in a UDDI registry: perhaps a fragment of text like, "a dictionary service for Irish Gaelic." OWL-S provides authors of web services a way of describing their services semantically, so that search-agents or the like do not have to guess by looking for keywords. This knowledge would allow, for example, a service-seeking agent to understand that a service such as that of Figure 1.1 provides a service for echoing back messages (which, while not terrifically helpful in most circumstances, at least gives details about what the service provides).

The service *model* describes what happens when the service is carried out, detailing the content of requests and responses. As the authors point out, it could potentially be used by service-seeking agents for a second tier of decision making [8]. Should a service's profile match that which the service-seeking agent is looking for, the agent could examine several candidates' models as a consideration for which one to choose.

Finally, the service *grounding* specifies exactly how a web service may be accessed. A grounding will typically specify "a communication protocol, message formats, and other service-specific details such as port numbers used in contacting the service" [8]. This seems at first glance to be similar to WSDL, and the authors later confirm that "a OWL-S/WSDL grounding uses OWL classes as the abstract types of message parts declared in WSDL, and then relies on WSDL binding constructs to specify the formatting of the messages." The OWL-S service grounding is not meant to replace

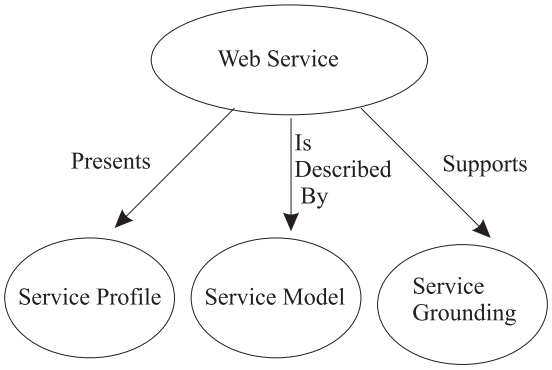

Figure 2.2: The Service Profile, Model, and Grounding of OWL-S

WSDL, but rather to complement it, as "the two languages do not cover the same conceptual space" [8]. Both OWL-S and WSDL are languages built on top of XML, so it is easy to build some OWL-S constructs on top of WSDL. WSDL documents, such as that of Figure 1.1, serve as the basis on which service groundings are to be built.

The three core features of OWL-S, the service profile, model, and grounding, are shown in Figure 2.2. The diagram is similar to one found in "OWL-S: Semantic Markup for Web Services" [8].

OWL, the Web Ontology Language provides three sublanguages. The first, *OWL Full*, provides no guarantees as to the computability of any conclusions within the language. The second, *OWL DL*, provides less expressiveness than OWL Full, but is guaranteed to be decidable. The third, *OWL Lite*, is a minimal sublanguage designed for, as McGuinness and van Harmelen put it, "those users primarily needing a classification hierarchy and simple constraints" [13]. An example of an OWL Lite-conformant ontology is the web service QoS ontology used in this thesis; it is contained in Appendix A.

That there are several sublanguages to select from is indicative of the fact that different needs are served by different languages. This is not only true of the OWL sublanguages, but also of the semantic markup languages in general. The choice of which language to use is an important application-specific decision to be made by the application designer.

McIlraith et al. address this issue in "Semantic Web Services" [7]. In it, the authors describe their system, which uses semantic markup in DAML+OIL to mark up a number of existing commercial web services: Yahoo's driving direction information service and United Airlines' flight booking service. Their markup provides declarative advertisements of the service, like the service profile of OWL-S; declarative APIs, built on WSDL, to allow for automatic service execution; and finally, specifications of the pre-requisites and consequences of individual service use necessary for automatic discovery, composition, and interoperation [7].

The authors make a strong case for the use of DAML+OIL for web services. RDF, they argue, is insufficient as a general semantic markup language due to its lack of expressive power and underspecified semantics [7]. Since the paper was written in 2001, however, RDF's semantics have been undergoing constant extension and revision. The current W3C Recommendation on RDF Semantics dates from February 10, 2004. Prior to that revision, the previous recommendation dated from December 15, 2003. RDF is one of the central building blocks of the semantic web, and the W3C remains committed to making it better. That several semantic markup languages use it as its base is testament to the strength and elegance of RDF's basic representation model, and of its extensibility. That said, it is probably not suitable for all applications. OWL-S and DAML+OIL are built on top of RDF for a reason. They offer more functionality than RDF. However, there is an increased overhead, and in some cases, no guarantees as to computability. The choice of which markup language to use should be made on a case-by-case basis, taking into account the needs of the system.

Semantic web languages and ideas can be used to tackle the selection problem, as mentioned earlier. Maximilien and Singh report their system, whereby agents serve as proxies for web services [21]. The agents select services based on reputation, making their choices by talking with other agents about reputations of services, with unknown services being selected if they are recommended by trusted third-party agents. Discovery of these services occurs through UDDI registries that have been augmented to allow ratings based on QoS attributes.

The languages of the semantic web provide the ability to both reason about marked-up data, as well as simply to classify, allowing for multiple implementations to talk about the same qualities through use of data ontologies. They build on each other. XML is the foundation, with RDF built on top of that. RDF Schema is built on RDF. OWL and DAML build on RDF and RDF Schema, and OWL-S builds on OWL. The use of these languages can be of great benefit, allowing heterogenous systems to talk about resources using the same set of terms and facilitating intercommunication.

2.2 Web Service Selection

The purpose of web service selection is to select an optimal web service for the current task, however one wishes to define "optimal". In defining this term, there are a number of key questions that are useful to consider:

1. At what point should a selection mechanism be used?
2. What kind of information is needed in the selection process?
3. How can this information be obtained?
4. How will this information be used in the selection process?

Researchers have approached the first question from a number of different angles. As an example, de Moor and van den Heuvel look at the role of web service selection during the software development cycle [17]. In virtual communities, they argue, web service selection should not be left to the software developer alone. Instead, it is important to involve members of the community, to understand their requirements, and involve them in the process of selection. This involvement answers the second and third questions: information is needed from the users and members of the community, and is obtained by involving them in the development. There is no explicit answer as to how to use this information in the selection process (question four). Rather, they write, in community-centered development, "it should be a continuous process of refinement and extension, instead of a one-time waterfall-type development project."

Research has also focused on selecting web services dynamically at run-time. Several of the solutions proposed by researchers are based on the idea of personal agents: autonomous pieces of software performing a task on behalf of the user. Maximilien and Singh describe a system in which proxy agents gather information on services, and also interact with other proxy agents to maximize their information [21]; the conceptual model they use to interact with the services is detailed elsewhere [22]. The proxy agents lie between the service consumer and the service providers. The agents contact a service broker, which contains information about all known services, as well as ratings about its observed QoS. From there, the information is combined with its own historical usage, and the combined knowledge is used to select a service, though the authors do not detail how.

Liu, Ngu, and Zeng [23] also approach the first question dynamically; the selection mechanism within their system works at run-time. As for the type of information needed in the selection process, they detail a dynamic selection mechanism 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. These QoS parameters can be determined either by getting the information from the service provider, or by execution monitoring on the client side.

To perform the selection, the QoS registry in their system 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, and are given a service based on the parameters that the client prefers.

Another method for dynamic service selection is specified by Balke and Wagner [24]. As the number of services grows, they write, so too will the need for personalizing the service based on the user's preferences. The authors describe a system whereby a user's desires influence the selection of the service. A SQL-like query language is used to narrow down the potential service(s). Any services that do not allow querying with the user's terms are discarded; this is what the authors describe as "hard contraints." If the user has any preferences, these are extracted from his or her profile, and used to even further narrow down the number of services; these are "implicit soft contraints" [24]. From there, the services are ordered by their utility: that is, the number of soft constraints that the service satisfies. The service with the highest utility is returned.

Day et al. detail a dynamic method of selection that is also based on QoS information [9]. The approach employs a centralized forum system to keep track of all the data, and mark up interactions between client and web service, thereby allowing clients to reason over and select from a number of potential services [9]. The system captures interactions between clients and web services, where clients can reason over these interactions to make informed decisions about which service is likely to provide good QoS. The interactions themselves are described in the RDF and OWL semantic web languages. As clients make calls to services, they mark up information about these calls, and report them to the central forum system. Clients can access information on various services, and reason over this data using either a rule-based expert system or a naive Bayesian classifier. Using the former, each service is given a ranking based on mean observed values for particular QoS parameters and user-defined weights for those parameters. In the latter, the classifier attempts to classify the service into one of five categories ("excellent", "good", "adequate", "poor", and "terrible") based on a number of observed QoS parameters.

These client-side reasoning abilities give more autonomy to the clients, as they do not specify *what* information should be used in the decision making process; that is up to the clients. As such, the information contained in, and used from, the interactions can be heterogenous, varying from client to client. One client might use a certain subset of the provided information; another client might use another subset; a third client might simply ignore the data it has collected, and pick randomly (though this might not be a useful approach).

The interactions describe two things: QoS information gathered by observing an interaction attempt, and system context information gathered at the time of the call. The QoS information covers the observed availability, reliability, and execution time of the service. The system context information includes, but is not limited to, CPU and memory usage, bandwidth used, and number of running processes. This information is used to give the client more information to determine the probable quality of a service. The idea is to allow a client to reason about the QoS information

	Selection	Information	Method
de Moor et al.	Design-time	User, community requirements	Involving users in development cycle
Maximilien et al.	Run-time	Service locations, QoS ratings	Proxy agents
Liu et al.	Run-time	QoS data	Client feedback
Day et al.	Run-time	client-specific data	Client feedback

Table 2.1: Current Approaches to Web Service Selection

separate from the context of the call; by providing all this additional information, it is hoped that clients will be able to tell when a service is performing poorly versus when the client's environment is not conducive to quick responses.

A summary of these approaches can be found in Table 2.1.

2.3 Reasoning with Expert Systems

An expert system, according to Peter Jackson, is "a computer program that represents and reasons with knowledge of some specialist subject with a view to solving problems or solving advice" [25]. An expert system "*simulates human reasoning* about a problem domain, rather than simulating the domain itself," and solves problems by "*heuristic or approximate methods* which, unlike algorithmic solutions, are not guaranteed to succeed" [25].

A popular algorithm for expert systems is the RETE algorithm [26] ("Rete" is Latin for "Net"), which compiles facts and rules into networks. This algorithm is efficient for pattern-matching due to its tree-structured sorting network that reduces the number of iterations over productions. The algorithm works as follows: patterns on the left-hand side of the production are compiled into the network. Then, the match algorithm computes a *conflict set* (a set of elements that, given the current conditions, cannot be all working correctly) for the current cycle by processing the network. The iteration on the working memory between cycles is eliminated due to the processing of a set of tokens which indicate which patterns match various elements of the working memory. This set of tokens is updated when the working memory changes [26].

A number of tools used by researchers in the semantic web and web technologies are heavily influenced by expert systems. Protégé, an ontology editor created by Stanford Medical Informatics, has available extensions to allow for compiling knowledge bases into JESS facts (JESS uses a language based on the CLIPS expert system language). DAMLJessKB and its successor, OWLJessKB, are description logic reasoners for the semantic web languages DAML and OWL, respectively. These tools allow for reasoning over ontologies using JESS for the semantics of the language.

2.4 Classification Using Naive Bayes

Naive Bayes is an assumption of conditional independence in a domain, which in turn allows for an easy machine learning algorithm. "Machine Learning" usually refers to a set of techniques whereby a system, given certain inputs, changes its behaviour in such a way that it performs better in the future [27]. This definition is perhaps vague, but Witten and Frank argue that it is necessary to have such a definition because it ties the learning to performance, rather than knowledge. When machine learning is tied to performance, they argue, it becomes more objective, and we can more easily measure things such as accuracy and misclassifications.

Naive Bayes, also known as "Simple Bayes" or even "Idiot Bayes" in the literature, is based around classification of instances of data into a set of classes [28]. Each attribute in an instance is assumed to be conditionally independent of every other attribute in the instance, given the class. Formally, if each instance has a set of n attributes $a_1, a_2, \ldots, a_n$, and if there are k classes $C = c_1, c_2, \ldots, c_k$, then the instance is classified to a particular class c_a by the equation:

$$c_a = \underset{c_j}{\operatorname{argmax}}(P(c_j) \prod_{i=1}^{k} P(a_i|c_j)) \tag{2.1}$$

As an example, consider Figure 2.3. Say that we have a very limited amount of student data, just each student's name, age, and sex. We wish to see if, from this data, we can classify students as undergraduate, Master's, or Ph.D. students (the "student type" node at the top). In the Naive, or Simple, Bayes model, the three attributes are considered independent of each other given the type of student we are observing. This may not be the case. But as Domingos and Pazzani point out, Naive Bayes can often perform well even when there are strong interdependencies between the variables [28]. The result is that the algorithm remains useful for for a large variety of problems.

Still, the assumption of conditional independence is powerful. Hand, Mannila, and Smyth write that "the conditional independence model is linear in the number of variables p rather than being exponential" [29]. In the general case, they write, classification is done simply by estimating that an object from each class will fall in each value of a discrete variable, using Bayes' rule to get the classification. And as they point out, in the general case, classification in this way is difficult because of the number of probabilities that must be estimated — for p k-valued variables, $O(k^p)$. Reduction from exponential to linear time may come at a cost of accuracy, but the speedup from exponential time to linear is significant, as is the reduced amount of training data. When learning, the amount of training data needed decreases when using Naive Bayes.

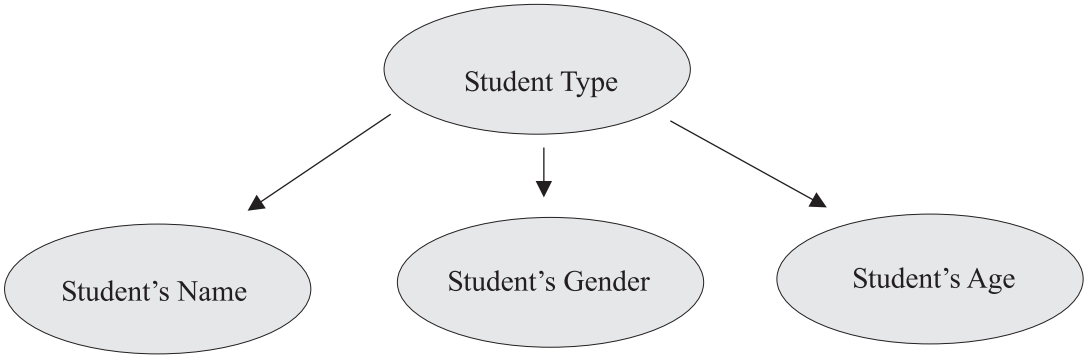

Figure 2.3: A Naive Bayesian Classifier for the Student Modelling Problem

2.5 Autonomic Computing

Autonomic computing is an attempt to deal with the management of large systems-of-systems that have become more common with the advent of greater computational resources. This management comes from a self-organization at the component level of the system. Waldrop, quoting IBM Vice-President Alan Ganek, writes that even only a decade ago, an example of a large computer system would be the ATM network for a large city [30]. Now, Ganek argues, some clients of IBM have that many servers within an organization. Systems are getting larger, and new techniques are needed to deal with this complexity.

The metaphor of autonomic computing comes from that of the human nervous system. The human body is incredibly complex, with all sorts of interactions between the various components. But we as people do not need to think about these interactions: we only need to know it works, and that the combination of all the interactions allows us to breathe, move, etc. While there is disagreement on exactly what techniques should be used on components in an autonomic system, Waldrop identifies four that he states are commonly agreed upon:

First, components should be *self-configuring*. The software equivalent of the "plug-and-play" idea in computer hardware, self-configuring components would seamlessly be able to fit within an existing infrastructure, and configure themselves to work within it without intervention. One such system is Glean, a system developed by Emre Kiciman and Yi-Min Wang [31]. Based on automatic monitoring for invalid settings, Glean uses constraints to differentiate between valid and invalid settings, gleaning these constraints from samples of known good configurations.

The components should be *self-optimizing*, able to find the most efficient way to distribute and perform tasks within the system. In theory, this is quite difficult, as scheduling is known to be NP-complete in the general case. However, an approximation of the optimal distribution might be possible in a reasonable frame of time. An example of a system with self-optimizing components

is that of Day et al. [9]. In the system, clients are augmented to reason over past experience data, and attempt to infer the best service for their needs.

Components should also be *self-protecting*, able to detect faults instead of blindly running into them. Chen et al. describe a system that uses decision trees to detect failures in large internet sites [32]. Trained on eBay data, their system has a great deal of success in identifying the causes of failures.

And finally, the components should practice some form of *self-healing*, which would allow fault recovery in the case of errors or failure. Though less-researched than the other techniques (in particular self-optimization), self-healing has become more popular recently. Dudley et al. present an architecture and implementation of a self-healing system, along with case studies to review the efficiency of their approach [33].

Autonomic computing has number of ideas that apply well to service selection. Perhaps the most important is self-optimization, which in the context of service selection could be taken to mean selecting the best service. But there are other good ideas: a self-protecting framework could have mechanisms in place to detect whether services are available or not, and notify components of this so that they would not attempt to use unavailable services. A self-healing framework could have ways of re-starting failed services, or replicating a service in case of failure.

Another core idea of autonomic computing is that of *policies*. It has been proposed that autonomic systems should manage their behaviour based on policies set by human administrators [34]. Kephart and Walsh define a policy as "any type of formal behavioral guide" [35]. Formal policies can allow a large degree of flexibility in the management of an autonomic system.

Kephart and Walsh define three types of policies. The first, *action policies*, or conditional policies, define how the system is to act under a certain set of circumstances. The second, *goal policies*, specify a desired goal, rather than a particular action. These allow greater flexibility, but more uncertainty, as the system is then expected to compute an action or set of actions that lead to this goal state. The problem with this is that planning is a well-known NP-complete problem, and thus can be very computationally expensive. The third type of policies are *utility function policies*. As Kephart and Walsh point out, this type of policy is a generalization of goal policies. Rather than classifying states into desirable vs. undesirable, a real value is given to each particular state, with the system attempting to maximize its utility somehow. Utility function policies have the advantage of being able to resolve conflicts between conflicting goals. But a disadvantage to them, as the authors point out, is that "utility functions policies can require policy authors to specify a multi-dimensional set of preferences" [35]. This can pose a number of problems: the exact difference between preferences, especially in high dimensions, can be difficult; as well, testing these specifications for correctness can be extremely complex.

The use of policies is well-suited to web service systems. Interactions between clients and

services, instead of being ad-hoc, could follow pre-defined policies. This allows a greater degree of flexibility and control within the system.

2.6 Conclusions

Each of the fields above provides much towards the problem of dynamic and autonomic web service selection. The base is provided by the research done in web services: the basis of WSDL and SOAP, along with research done in selection, provide a starting point. The idea of policies from autonomic computing, as well as ideas such as self-optimization, self-protection, and self-healing, bring powerful mechanisms for managing large amounts of components and data. Techniques from Artificial Intelligence, including rule-based expert systems and naive Bayesian classifiers, allow for the discovery of patterns and groups within data, and provide a useful means of parsing the collected data. But before parsing can be done, meaning must be derived from the data. To do this, semantic markup languages such as RDF and OWL can provide meaning and structure to data, allowing it to be not only parsed, but relationships to be drawn using the ontology.

Together, these fields and tools and concepts allow for greater strength and flexibility in tackling the problem of service selection, permitting a greater understanding of the problem and the techniques that can be used in solving it. Web service technology, including XML, SOAP, and WSDL, allow for the creation of the services themselves. Clients are augmented with reasoning capabilities taken from artificial intelligence, both rule-based expert systems and naive Bayesian classifiers. But these reasoners need information to reason about, and that is provided by the interaction models marked up with RDF and OWL. Finally, to help manage the system as a whole, there is research on self-optimization, self-healing, and policies from autonomic computing.

Chapter 3

A Framework for Autonomic Web Service Selection

The selection framework is one of client-side augmentations. A basic web service client is augmented with the ability to gather QoS data, reason over that data, and report its own interactions with web services. There are a number of advantages to this approach. First, the information gathered is complete and accurate. By treating the web service itself as a black box, the client is in complete control of the information gathered. Second, by keeping the selection ability client-side, the fault-tolerance of the system is increased. Were this ability to be intermediate, in the form of a web service, a proxy, or something similar, failure of the intermediary would mean failure of the selection mechanism, which could prove disastrous.

These augmentations, described in detail below, allow the client to monitor and report QoS interactions with a number of syntactically identical web services, as well as dynamically transfer calls from one service to another should the augmentations recommend this action. The assumption of similarity through identical WSDL definitions is used to reduce the scope of the problem.

There are a number of advantages to this approach over the possibilities mentioned in Section 2.2 and summarized in Table 2.1. First, a purely price-based negotiation is not sufficient alone for selection, as price tells us nothing about what the service actually provides. Second, while a process based on semantic suitability is useful, it tells us nothing about the sort of QoS that a web service provides. If we assume that the services we can select from are syntactically and semantically equivalent (which does not seem to be an unreasonable assumption), the process of selecting based on prior QoS experiences seems to be worthwhile.

The difference between the system described in this thesis over that of of Liu et al. [23] (discussed earlier, in Section 2.2) is that the selection mechanism described is not dependent on any external factors. In their system, if the QoS registry goes down, the entire selection mechanism is lost with it. An approach based on augmenting the clients themselves does not suffer from that problem.

To evaluate services, clients have an abstract evaluation function:

$$f(s_1, s_2, \ldots, s_n) = s_k \tag{3.1}$$

This function takes as input the QoS information about n different web services, and provides a recommendation as to which service it thinks will provide the best QoS based on the past data (s_k). A policy manager, described in detail in Section 3.4 controls the workings of the augmentations. It is important to note that this evaluation function can be defined differently for different clients, and this is why raw QoS information, rather than pre-processed data, is sent to the clients.

But how should this function be implemented? One could define an evaluation function which selects services randomly, but this will generally be worse than an evaluation function that bases its decisions on the QoS information provided. With that in mind, there are a few terms that are useful to define in talking about "better" and "best" services, which are otherwise subjective terms.

Let S be the set of web services considered. For each service $s \in S$, let the set of observed QoS parameters $P = p_1, \ldots, p_n$ be parameters which an evaluation function considers, and let $u(s_i, p_i)$ be the utility given by the function to the value of parameter p_i of web service s_i. These parameters may vary from service to service (for example, a car rental service may define a "financing" parameter). Now, consider two services, $s_1, s_2 \in S$. s_1 is *strictly better* than s_2 if, for all $p_i \in P$, $u(s_1, p_i) > u(s_2, p_i)$. s_1 is said to be *weakly better* than s_2 if there exists a parameter $p_i \in P$ such that $u_(s_1, p_i) \geq u(s_2, p_i)$. That is, a service is weakly better than another if there is a parameter for that service that the evaluation function considers to be better than that parameter for the other service. Depending on how an evaluation function considers parameters, it might be perfectly rational to select a service that is better than another in only one QoS parameter: that parameter might be of the utmost importance.

An evaluation function is optimal if, among services $s_1, s_2, \ldots, s_n$, it always selects a service that is strictly better than the others, if such a service exists. If no such service exists, the function is optimal if it selects a service weakly better than the rest.

An illustration of the client as a whole can be seen in Figure 3.1. The QoS interactions will be discussed in Sections 3.1 and 3.2. The reasoners will be discussed in Section 3.3. Finally, the policy manager will be discussed in Section 3.4.

3.1 Capturing Interactions

Interactions are at the heart of the architecture. Each interaction captures one usage of a web service by a client. The interactions, termed "interaction models" or "QoS interactions", are stored in an RDF/OWL format by the clients, and contain information about both QoS parameters and system context information.

The QoS parameters stored are as follows:

- *Availability* is whether or not the client can connect to the web service, and takes the boolean values of 0 (cannot connect) or 1 (able to connect).

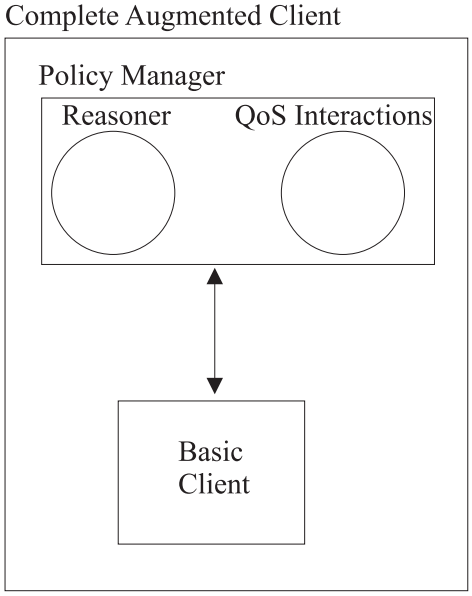

Figure 3.1: A View of the Augmented Web Services Clients

- *Reliability* refers to whether the service has kept to the WSDL definition used to generate the client's call stubs – to whether the syntax remains unchanged. If a call succeeds and does not return an error relating to bad return type or method signature, the reliability for that interaction takes the boolean value 1, and 0 otherwise.

- *Execution Time* is the time, in seconds, that it takes to call a service and get a valid result returned.

In addition to QoS information, later versions of the framework have included system context information. Calls to web services are not divorced from the system from which the call is made; the computer might be loaded down with dozens and dozens of processes, and may have few cycles to spare. Because of this, clients might misclassify a service's QoS based on local conditions. A web service might have massive resources at its disposal, but if a client has 95% CPU usage and low bandwidth, it might think the QoS poor. The client's system context is provided with each interaction to attempt to avoid this sort of situation.

The following system context parameters are recorded when marking up an interaction: process load, total memory used, percentage of memory used, bytes sent/received/total per second over all network devices, and number of processes.

The QoS interactions are extensible. Clients can mark up whatever other parameters they wish, and if another client picks up that interaction, any information that it is not familiar with is simply ignored. This allows for a great deal of openness within the framework, and means that heterogenous clients, designed for different services, are able to easily communicate with one

another, provided they understand some common terms.

3.1.1 A Generic QoS Ontology

For clients to understand each other, two things are needed: a common language, and a common set of terms. For the former, the web service clients use the RDF and OWL languages. For the latter, it is important to determine which parameters should be used by all clients.

The approach used is to create a basic and extensible QoS ontology. There is a basic division into service-specific and call-specific parameters. Service-specific parameters are those specific to one or more services; thus, even parameters common to all services are put into this category in the ontology. Call-specific parameters relate to the call itself. The execution time parameter discussed in Section 3.1 is a combination of both the execution time on the server itself with the time it takes to create the SOAP or XML-RPC request, the time to send it across the network, and the time to send the response back. Because of this, the ontology has support for parameters specific as much to the call as the service. Were it possible to separate the time spent creating the request, sending it, having the service process it, and sending the response back, a "service execution time", more representative of QoS, could be included in the service-specific category. But until then, the imperfect execution time parameter will have to do.

As an example of service-specific parameters, consider availability and reliability as defined in Section 3.1. At a basic level, regardless of what a service is about, a service is accessed by clients, and these parameters describe that access: whether an access succeeds, and whether the execution is error-free.

While these parameters are used by all clients, there is nothing to say that other parameters could not be used by a subset of the clients. Due to the openness and extensibility of the framework, any number of other parameters could be included. In the context of an e-learning system, for example, a client accessing a video-based learning object might include information about whether the user found it useful, or the length of the video.

The QoS ontology itself can be found in Appendix A. While perhaps a bit obtuse, that is more a function of the wordiness of XML-based languages than of the ontology itself. The ontology is visualized in Figure 3.2. In this Figure, the square boxes represent classes, while a directed arc from A to B indicates that B is a subclass of A. As can be seen, it is a basic taxonomy of terms that can be easily extended to handle more generic or domain-specific QoS paramters. This would be done by subclassing either "Call-Specific Parameters" or "Service-Specific Parameters" and adding appropriate fields and values.

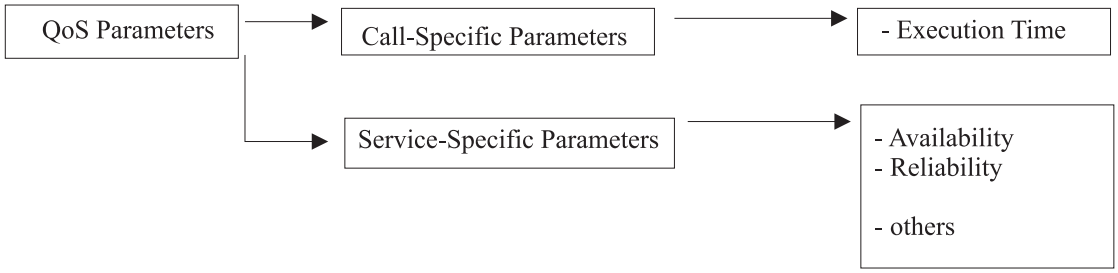

Figure 3.2: Visualization of a Generic Ontology for QoS Interactions

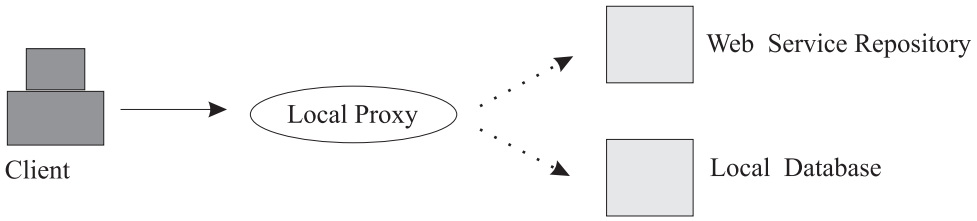

Figure 3.3: Updating Knowledge by Sending Interactions Through a Local Proxy

3.2 Storing Interactions

When storing QoS interactions, one of two approaches are used. In the centralized approach, interactions are sent to a central database wrapped in a web service. In the decentralized approach, interactions are stored in one or more locations. Clients store, at a minimum, their own interactions with web services. Sending these interactions takes place by sending them to a local proxy, which is part of the framework. The proxy then either stores the interactions in a local database, or contacts a specified web service and sends the interactions there. This process is shown in Figure 3.3.

3.2.1 A Centralized Approach: QoS Forums

In the centralized approach, QoS interactions are stored openly, in a SQL database with a web service wrapper. Clients can send requests to the service for either all information on a particular service, a certain number of interactions for a particular service, or all information on a particular service after a certain date.

The advantage to this approach is that clients have the ability to have perfect information: all instances reported will be available, since all clients report to the same repository. However, this common access point has the disadvantage that if it were to go down, all the clients using the service would not be able to obtain new information on various web services.

The single point of failure has been alleviated somewhat by having the clients save the state of their reasoners. When the client gathers data by querying, it automatically writes the state of the

reasoner to disk after the reasoner has been built. This allows the client to load the state of the reasoner at startup, which in turn gives the client a base of knowledge to work with should it be unable to connect to its prescribed web service.

3.2.2 Peer-to-Peer Decentralization

To improve the robustness of the framework, decentralization of the single QoS repository described in Section 3.2.1 is implemented at the client level. The idea is a fairly straightforward one: each client stores its own interactions, and supplies them to other clients when requested. Each client maintains a local database of its interactions, keeping the basic underlying storage mechanism from the QoS forum: data can be requested in whole or in part. Clients can then talk to each other, and use the QoS information gained to make an informed selection decision.

There are a number of advantages to this approach. The most important is the increased fault-tolerance. Increasing the number of clients, assuming a common communication mechanism, and knowledge of other peers, increases the stability of the knowledge component of the system.

A second advantage is that, even if the client does not know about any other clients, and cannot request information from them, it has its own knowledge as a base to work from. As well, accessing the data in isolation is very fast, as no network requests need to be made, and after initially loading the state file on startup, all reasoning is done in-memory.

The approach is not without disadvantages, however. Clients will generally have limited knowledge, as ensuring complete knowledge would require knowing about every peer in the network. For large networks, querying could be computationally prohibitive.

In the peer-to-peer configuration, clients will know directly about one or more peers; otherwise, they will have to work in isolation. To get information about the web services of interest, the client initiates a two-part process. First, a list of peers is built by the client by asking its known peers for their known peers. A low time-to-live (TTL) value is sent as well, so that as these requests are propagated, there is an upper bound to the number of messages that will be sent. These TTL values decrease by 1 each time they are transmitted, and thus set a bound for the number of peers that may be contacted in total for any given query. The default TTL value used in the framework is 2, so that a client can still discover a number of peers, while keeping the time required for each query to be low. This number was chosen so that a number of peers could be added to a client's master peer list, while limiting the amount of traffic sent over the network.

Once the request has been sent, the peers aggregate the responses they get, and send them back to the requestor. When the client finally receives this list, it updates its list of peers to include these new peers, if they are not already present. This can be seen in Figure 3.4. In this figure, a directed arc from a node A to another node B means that A knows about B – that is, that A has B in its peer list.

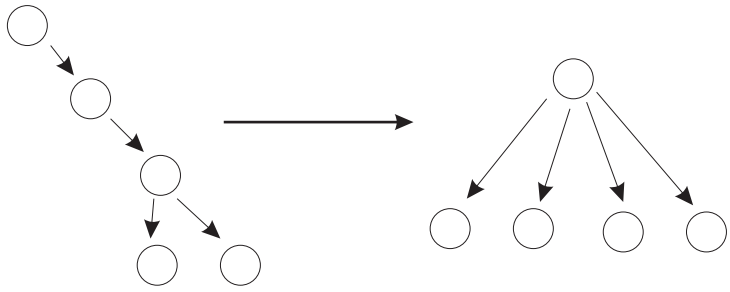

Figure 3.4: Building the Master Peer List

Once the client has built a master list of the peers in its neighbourhood, it requests a number of interactions from each peer. For each service, the client divides the number of total interactions it wants by the number of peers in its master list. The client sends a request to each peer, and includes with it a filter. The filter contains things such as the web service of interest, and the number of interactions needed.

Because the client does not know about how much information the other peers has, it may receive less data than requested. If this is the case, it divides the number of interactions still required between the number of peers in its master list, and begins the process again. This continues until the client has the total number of interactions desired. If the client ever gets a total of 0 interactions from all of its peers, it stops trying to query them, to avoid a possible infinite loop.

3.3 Using the Interactions

In the decentralized approach, clients may either request the raw data, or a recommendation. While gathering raw data allows for much more versatility, doing so takes much more time and memory than if a client simply collects a number of recommendations.

If a client decides to use raw data instead of getting recommendations, the interactions are handled by the reasoning component of the client. Once a number of interactions are received, they are passed to the reasoning component, which then provides a recommendation: change to a new web service s, or remain with the current one.

The reasoner is abstract: a reasoner should be able to reason over data and provide a recommendation, but not necessarily be tied to any particular architecture. Because of this, clients are able to use one of two types of reasoners: a rule-based expert system, and a Naive Bayesian classifier. These are the only types of reasoner implemented in the framework, but the framework is general enough that other algorithms could be easily implemented. So long as a reasoner implements the `ReasoningEngine` class (takes as input a vector of QoS interactions, and returns a service recommendation as a result), it can be implemented in the framework.

```
(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 3.5: JESS Code for Web Service Selection

3.3.1 Rule-Based Expert System

The rule-based expert system is implemented in JESS, the Java Expert Systems Shell. Expert systems are designed for reasoning over a number of rules and facts, and the RETE algorithm on which many are based has been shown to be fast and efficient for large data sets [26]. While we use only a few rules within our system, scalability of the rules and fact-sets was judged to be important for reasons of extensibility.

The expert system takes in the basic QoS data: availability, reliability, and execution time, though the latter is only used should the service be both available and reliable for a particular interaction. This is so that small execution times resulting from unreachable services are not factored into the calculations. Once the reasoner has this information, it creates a weighted sum for each web service ws_j:

$$ws_j = \sum_{p_x} (w_x \frac{\sum_{i=1}^{n} x_i}{n}) \tag{3.2}$$

w_x is the weight for QoS parameter p_x. These weights are user-specified for each of the QoS parameters. The default values are 10 for both availability and reliability, and -1 for execution time. These weights are arbitrary – one of the problems with the expert system is determining which weight values to use. x_i is the i^{th} reported instance of parameter p_x for the web service we are currently examining.

Once the reasoner has a weighted sum for each of the services it is considering, it uses a simple heuristic: take the service with the highest-weighted sum to be best, and provide that as the recommendation. The JESS rules that perform this can be found in Figure 3.5. These rules simply compare each service against the known best service. If the weighted sum is higher, then that service becomes the new best service. Initially, the best service is set to be null, with a value of -1000, so that any of the services compared will be considered better.

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

Figure 3.6: JESS Triples Representing Web Service Information

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

Figure 3.7: Representing the Best Service So Far in the Expert System Reasoner

The facts about the web service may be determined dynamically at runtime by the reasoner. If a state file does not exist, the reasoner creates a new fact base based on the interaction snapshots provided. The basic form of these facts can be seen in Figure 3.6. The first element of the triple tells the reasoner that this fact is information about a web service. The second and third elements represent the location of the web service and the weighted sum (computed by the reasoner using Equation 3.2). The reasoner also keeps track of the best service to date, along with its weighted sum. This is done with facts of the form displayed in Figure 3.7. "best-s" represents the location of the best service so far, while "best-t" represents the weighted sum for that service. Higher values indicate that clients have experienced better QoS from that service in the past. While this says nothing about the *current* QoS, it represents a testable heuristic.

The rules and facts have been edited slightly to meet the margin requrements of this document – otherwise, they would extend beyond them.

3.3.2 Naive Bayesian Classification

The second reasoner in the system is based on Naive Bayesian classification. The decision to add this type of reasoner was made for two reasons. First, the rule-based expert system's heuristic takes the highest-weighted sum. While this is a useful approach in selecting a good web service, problems may arise when many clients, reasoning over the same data, all select the same service. Each will transfer execution to this new service, and as more and more clients hit the service, will experience increasingly poor QoS. That will cause them to share more and more information that the current system is poor, and during the next inference cycle, will cause them all to jump to another service. This hopping effect can be avoided if clients select among a number of services with similar QoS.

The second reason why the Naive Bayes algorithm is implemented is because of the complexity of dealing with all the system context parameters discussed in Section 3.1. The time required to explore all the interactions between them and formulate JESS rules was deemed to be too prohibitive. By contrast, a Naive Bayesian classifier could take this information, along with the QoS information, and attempt to find relationships between these and the class of the service.

The Naive Bayes reasoner takes in interactions in the same way as does the rule-based expert

system, except unlike the latter, the Naive Bayes classifier uses all the information it receives. The classifier attempts to classify each service s into one of the categories <terrible, poor, acceptable, good, excellent>. To do so, it considers the following attributes:

- Availability: {true, false}
- Reliability: {true, false}
- Execution Time: real
- CPU Usage: real
- Memory Used, MB: real
- Memory Used, %: real
- Avg. Bytes Sent/Sec: real
- Avg. Bytes Received/Sec: real
- Avg. Total Bytes: real
- Number of Processes: numeric

When making a recommendation back to the client, a web service is selected from the highest-available class. If there are multiple services in the highest classification category, then one is selected with a uniform probability distribution.

3.3.3 Dealing with Recommendations: Peer Models and Trust

When a client collects recommendations, there are a number of immediate advantages. First, no data has to be parsed, so if recommendations are taken as-is, much processing time can be avoided. Second, the amount of time needed to receive recommendations is less than the time to receive even one interaction snapshot, due to the time required to send a simple URL versus the time needed to send an XML document containing information about the service, its location, and observed QoS parameters.

But if a client takes a recommendation as-is, there are potential problems and risks. For instance, what if recommendations are not actually based on the experiences of the client? One could easily imagine a client that when given a list of services, always returns a particular service. This could be done maliciously (to defame a particular service), selfishly (talking up a particular service), or for a number of other reasons. Regardless, a truthful recommendation is the most useful, as it is based off the actual past experiences of one or more clients. The problem is how to judge the truthfulness of a recommendation.

The problem of decision making based on recommendations is very similar to the Byzantine Generals problem [36]. The basic problem is that components of a system may send conflicting information to the other components within a system. The components, or "Generals" in the analogy, communicate only indirectly, via message passing. In the context of a web service framework, the clients might send different recommendations to different clients. In this context, the problem is much weaker than the original formulation. Because we always know, through direct socket communication, from whom we are receiving a recommendation, the identity of a particular client cannot be faked without much difficulty. With these known identities, and with the mechanisms to gather data from peers described in Section 3.2.2, there is a solution to determine which peers give good advice, and thus should be listened to; and which are giving bad advice, and which can safely be ignored.

To do so, a trust mechanism is used, in which a client creates models of its peers. These models are then used to add weights to the given recommendations: the client's trust towards a particular peer p_i for a particular service s is expressed as $T(s, p_i)$, and is a value between 0 and 1, inclusive. If a service is recommended by a peer ($R(s, p_i)$), it is considered to have a value of 1. Likewise, if it is not recommended by a peer, its weight is 0. The client selects the service with the highest value, S_b:

$$S_b = \underset{s \in S}{\operatorname{argmax}}(\sum_{i=1}^{n} T(s, p_i) * R(s, p_i)) \tag{3.3}$$

In this way, completely untrustworthy peers (those with a trust value of 0) are unable to impact the decision making of other clients. The trust between two clients for a particular service is the level of prior agreement between the two clients on that particular service. Thus, even if a number of peers attempt to skew the decision making process towards a particular service, this can be averted. The trust a client has towards another peer for a particular service is computed as follows:

1. Each peer begins with a trust of 0.5 for each service: no peer has any bias, initally. A base trust score of 0.5 allows an increase towards 1 (increased trust), and towards 0 (decreased trust).

2. Contact the peer, and request a specified number of interaction snapshots for the specified service.

3. Get an equivalent number of interactions from our own database, if possible. If this is not possible, get the maximum possible from our own database, and reduce the number of interactions given by the peer to this number.

4. Sort each list of interactions chronologically.

5. If possible, "line up" the lists so that, pairwise, each element of each list is chronologically close to each other, and ensure that each list is of identical length. Chronologically close is taken to mean "within half an hour". If this is not possible, exit, and give a trust value of 0.5.

6. For each pair of elements P_i and C_i, $1 \leq i \leq n$, where P_i is the i^{th} element of the peer's list, and C_i is the i^{th} element of the client's list:

   (a) If all QoS parameters are close, then increment the trust by a normalized increment, α, where $\alpha = 0.5/\text{size}(C)$.

   (b) If the QoS parameters are not close, then decrement the trust by α.

Closeness is determined on a per-parameter basis. For availability and reliability, the values are required to be identical to be close, since these parameters take on binary values. Execution time is more fuzzy: a value of four seconds was used to determine closeness, as some arbitrary value was needed. If additional parameters are to be considered in the calculation of trust, clients will need some way to compare these parameters.

These trust values are computed for each service the client is interested in. Peers in complete agreement can be seen to have a value of 1 for each service considered. Perfectly untrustworthy peers have a value of 0 for each. Peers for which there is no agreeing data (the data is too far away chronologically, there is no data gathered on that particular service, etc) have the default value of 0.5. This is to represent the idea that the peer is neither reputable nor disreputable with regards to that service.

3.4 Action Policies: Controlling the Process Within the System

Querying, reasoning, switching, and reporting are controlled by the policy manager. These actions are controlled by policies which have the form of the action policies described by Kephart and Walsh [35]. They describe action policies as how the system is to act under a particular set of circumstances.

The policy manager uses three types of circumstances for its action policies. These are:

- *One-shot* circumstances: policies with a "one-shot" condition are fired whenever the policy manager is ready. In practice, this occurs immediately.

- *Reactive* circumstances: if a policy has a reactive circumstance, it is fired after a particular circumstance (such as a successful call to a web service, querying, reasoning, etc.) occurs.

- *Recurrent* circumstances: policies with a recurrent circumstance are fired at a particular interval specified externally. This could be every five seconds, ten minutes, four days, or any number of other possibilities.

The action policies are described externally, in a language specified by the grammar shown in Figure 3.8. An example of a language generated by that grammar can be seen in Figure 3.4.2.

3.4.1 Actions Controllable by Policies

The specifics of policies are specified by the grammar. First, there are the actions for which policies may be specified. Currently, the grammar allows for querying, reasoning, switching, and reporting.

Querying deals with getting either data or recommendations from the central QoS repository or a client's peers. If data is given, then a reasoner is built with each query, and trained with that data. If recommendations are given, the recommendation is stored for future use.

Reasoning is the process of asking whatever reasoner has been built for the best service, given the training data. Whether a naive Bayes classifier, or a rule-based expert system, the reasoners have a common interface, and return the URL of the service which the reasoner has determined is best.

Switching is simply changing the location of the service which the client is calling. Because the services are assumed to have a common interface, switching is just a matter of changing a URL within the client.

Reporting occurs when clients send their stored QoS interactions to either the centralized QoS repository or to their own internal database.

These actions are performed when certain circumstances occur. As described in Section 3.4, recurrent events occur at specific time intervals, and one-shot events occur only once. Reactive events can occur whenever one of the actions above has been performed, but also after a call to a web service has succeeded or failed; this is represented by the "<Action>" non-terminal rule in Figure 3.8.

3.4.2 Implementation of the Policy Manager

The policy manager is implemented as a rule-based expert system written in the CLIPS language, but run within the JESS environment. While the policy-controlling mechanisms of the expert system could have been written in pure Java, JESS was selected for two reasons. First, it allowed the focus to be on the problem of controlling policies, rather than implementation details. Second, using the RETE algorithm (which JESS uses) is more efficient and time-saving than attempting to write a similar algorithm – a well-tested and efficient algorithm is already provided. The policy manager is a separate thread of execution in the framework. The different types of conditions (one-

shot, recurrent, and reactive) are represented as rules, with the circumstances to execute them written as facts. As the expert system runs, the combination of facts and rules produces actions which are to be executed externally within the framework.

Each action (querying, reasoning, switching, reporting) has a list of possible conditions. Each condition is represented as a fact in the system, and has the form shown in Figure 3.4.2. In that Figure, we see that querying is both one-shot and recurrent, and that reasoning is reactive, occuring after querying. In that example, the reason that we bind the one-shot event is so that after running one iteration of the expert system, the fact can be retracted, to ensure that it runs only once. By contrast, we assume that the reactive and recurrent conditions hold for as long as the framework is running, so we don't ever need to retract those facts.

To keep track of time within the expert system for the policies with recurrent conditions, a separate thread is created by the policy manager. This thread creates a list of conditions it needs to track. Keeping track of the time, this thread checks the list every second to see if any of the conditions have become true. If they have, the thread contacts the policy manager, and asserts a fact of the form "(currently <TimeType> <TimeUnit>)". Another iteration of the expert system is then run, to update the list of actions that need to be performed.

Whether a policy's conditions are recurrent, reactive, or one-shot (or some combination of the three), the end result is to call a "ready" function, which tells the expert system that we should execute that action within the system. The specifics of each condition rule can be seen in Figure 3.4.2, and the "ready" function can be seen in Figure 3.4.2. As can be seen by the body of the function, calling the "ready" function causes a fact such as "doAction reporting" to be asserted. The policy manager, with each iteration of the expert system, checks to see if any facts of this form have been asserted. If they have, it creates a queue of actions to be performed, and from there executes those actions in order.

```
<QOS> ::= <QOSParameters> |  ϵ

<QOSParameters> ::= <QOSParameter> | <QOSParameter> <QOSParameters>

<QOSParameter> ::= policy <Parameter> <ConditionList> end

<ConditionList> ::= <Condition> | <Condition> <ConditionList>

<Condition> ::= one-shot | after <Action> | every <TimeUnits> <TimeType>

<Action> ::= success | failure | <Parameter>

<TimeUnits> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 | <TimeUnits><TimeUnits>

<TimeType> ::= seconds | minutes | hours | days

<Parameter> ::= querying | reporting | switching | reasoning
```

Figure 3.8: The BNF of the CFG for Controlling Action Policies Within the Framework

```
policy querying
    after failure
end

policy reasoning
    after querying
end

policy switching
    after reasoning
end

policy reporting
    after failure
    after success
end
```

Figure 3.9: An Example of an Action Policy Specification

```
(bind ?factid-oneshot-querying (assert (oneshot querying)))

(assert (isrecurrent querying))
(assert (every 10 minutes querying))

(assert (isreactive reasoning))
(assert (after querying reasoning))
```

Figure 3.10: Examples of Condition Facts in the Policy Manager

```
(deffunction ready (?x) (bind ?doX (assert (doAction ?x) ) ) )
```

Figure 3.11: The "Ready" Function, Called to Indicate An Action Should be Performed

```
(defrule once (oneshot ?policy) => (ready ?policy))

(defrule reactive
   (and (completed ?priorPolicy)
        (isreactive ?policy)
        (after ?priorPolicy ?policy)
    ) => (ready ?policy))

(defrule recurrent
    (and (isrecurrent ?policy)
         (every ?unit ?num ?policy)
         (currently ?unit ?num)
    ) => (ready ?policy))
```

Figure 3.12: Rules for One-Shot, Reactive, and Recurrent Policies

Chapter 4

Experiment 1: Overhead of the Framework

While the purpose of the framework is useful – few would argue against improving quality of service – each additional component added on to a basic client comes with a cost. The purpose of the first experiment is to determine the overhead of the framework, both at the system and network level. The overhead of both the framework, as well as a basic web service client, are examined.

At the system level, there are a number of areas with which to be concerned. The memory consumption and CPU usage of the framework are important, as these help determine how fast the framework appears to the user, and whether or not it is a burden on the user's computational resources. As well, as the length of the call is important. If there is a significant delay in the time required to package the SOAP request, send it, and decode the response, then the utility of the framework could certainly be called into question, as the overall QoS experienced is supposed to improve, not decrease.

At the network level, the first experiment monitors the amount of bytes sent and received by the client, in both augmented and unaugmented settings. But because the augmented client can be running with a number of different policies (see Section 3.4.1), the framework is examined in a number of these different configurations. These configurations are discussed in Section 4.1, with the exact policies specified in Appendix C.

4.1 Policy Details

To test the framework evenly, the experiment runs the clients in a number of different policy configurations for querying, reasoning, and switching. The exact syntax of these configurations is found in Appendix C.

The first configuration is termed *conservative*: data is gathered, and the reasoner invoked, on startup only, though interactions with whatever service is selected are reported after each call.

The second configuration is *reactive*. In it, decisions about which service to use are made after each failure, with reporting of QoS data done after each call.

The third policy configuration is *recurrent*. Querying is recurrent, and occurs after a certain period of time has elapsed. In this experiment, that span is fixed at ten minutes. Reasoning in this

configuration is reactive, occuring after querying has been completed.

Because the framework has two major architectures for storing and retrieving QoS data (centralized and decentralized), the experiment runs each of the above configurations with both centralized and decentralized architectures.

4.2 Experimental Setup

Three separate machines are set up as clients. The first is a P4 running at 3 GHz, with 2 GB of RAM. The second uses an AMD Athlon CPU at 807 MHz, with 640 MB of RAM. The third has a P3 processor with 512 MB of RAM. The second and third clients are quite comparable in terms of their processing capabilities. The first client, more powerful than the rest, is a more recent computer. It was chosen to allow a heterogeneous configuration, so that the performance of the framework could be viewed on a wider range of systems. The reason three computers were designated for clients was because it would help tell if certain trends in the data were actually trends. As more computers are added, it becomes easier and easier to see trends for the different policy configurations. But running large numbers of clients on large numbers of machines becomes increasingly complex to set up, and to obtain data. It was felt that three computers was a good tradeoff between complexity and data gathering.

Besides the clients, there is a computer running the central QoS repository, and three running web services. The repository and one of the three web services each run on an 807 MHz, 512 MB Athlon machine. The other two web services run on P3s at 800 MHz; one has 384 MB RAM, the other 256. The network used for experimentation is 100Mb Ethernet.

Each of the services is fixed to have a certain QoS. The first service runs as normal, and processes requests as fast as it can. The second accepts requests as normal, but adds an extra five to eight seconds to the length of the call. The third service has poor QoS, returning an error at least 10% of the time, and adding a significant delay of 20 to 30 seconds to the length of the call. The QoS of these three could loosely be characterized as “excellent”, “adequate”, and “poor”, respectively.

To ensure an even distribution among the services, each client makes 250 sequential calls to each service, for 750 total calls each. The three clients are configured to call different services each time, and each service is guaranteed to be called once for each set of 250 calls. So for the first set of 250 calls, each of the three services will be called; for the next 250 calls, each of the three services will be called, but by different clients; and so on for the next 250 calls.

Attached to each client is a data logger, which both monitors the time taken for each call, and also logs system information at a five-second interval. This system information is the basis for the comparison of the framework’s overhead against that of a standard, unaugmented web service client.

Each of the three services are Fibonacci calculators. The services are all locally administered; while it would be possible to use third-party services for testing the framework, the lack of control could prove troublesome should a service crash, and need to be recovered quickly. For this reason, it was decided to keep control of the services.

4.3 CPU Overhead

One useful method of measuring overhead is to look at CPU usage. The basic web service client written for experimentation purposes makes calls to a particular web service, but ignores the results of the calls. The augmented client will not only make calls to the service, but also keeps track of the results, and stores the results either locally or in a central, shared location. This information will then be used to make any decisions needed about switching services due to poor QoS. The CPU usage examined will be the per-process CPU usage, which is the percentage of elapsed time that the threads of the process use the CPU to execute instructions.

Using the method described above, in Section 4.2, the augmented and unaugmented clients make a total of 750 calls each. The CPU usage of the clients is shown in Figure 4.1, and the statistics are summarized in Table 4.1. As can be seen in the Figure, the CPU usage is very low, except for a few spikes. These can be explained by the basic client being dormant most of the time; apart from making calls to the web service, the client remains idle. The spikes are likely caused by some part of the calling process; the reason there are relatively few spikes is because the data sampling occurs only every five seconds. Therefore, the only time spikes are recorded is when the data sampling and web service calls overlap. The sampling interval of five seconds was chosen so that the sampling of data would not become CPU-intensive and thus impact the measurement of CPU data.

It can be seen that the unaugmented client has very little overhead as far as CPU usage is concerned. The mean CPU usage for the basic client is less than 1% on each machine, with relatively few large spikes.

As an example of the framework, consider the augmented client using a conservative policy configuration, shown in Figure 4.2. As can be seen, the CPU usage is much greater than that of the unaugmented client. Certain components of the framework, such as the policy manager, are constantly running and gathering information, and this sort of activity will increase the CPU load greatly. The basic client, in contrast, makes calls at a regular interval, but otherwise remains idle, and this is reflected in the CPU usage.

But are these differences significant, or are the samples taken from the same statistical distribution? First, an assumption. Assume that the CPU data, along with all the other overhead data collected, is normally distributed. The raw data certainly suggests this. As an example, most of

	Mean	Standard Deviation
Client 1, Unaug	0.356	2.650
Client 1, Aug	30.461	26.361
Client 2, Unaug	0.408	1.876
Client 2, Aug	68.420	6.871
Client 3, Unaug	0.453	1.778
Client 3, Aug	63.381	5.291

Table 4.1: CPU Load Statistics for the Unaugmented and Augmented Clients (N=1006 and 1449 samples, respectively)

the CPU measurements for the augmented clients are clustered around 16% for Client 1, and 77% for Clients 2 and 3. With this assumption, we can use statistical hypothesis testing to determine whether the augmented and unaugmented data differ significantly, which will in turn, along with other overhead measures, will illuminate whether or not the framework adds a significant amount of overhead over the basic client.

Let H_0, the null hypothesis, be that the true mean CPU load for the unaugmented client, μ_1, is equal to that of the augmented client (in which we will use the centralized conservative policy configuration), μ_2 (that is, that $\mu_1 - \mu_2 = 0$). We will test this versus the alternate hypothesis, H_A, that $\mu_1 - \mu_2 \neq 0$.

The test statistic, Z, for a large-sample hypothesis test is given by $Z = \frac{m_1 - m_2}{\sqrt{\frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}}}$, where n_1 and n_2 are the samples sizes of the data for the unaugmented and augmented clients, respectively; where m_1 and m_2 are the observed means, and σ_1^2 and σ_2^2 are the observed variances [37]. Because this is a two-tailed test (that is, the alternate hypothesis is that $\mu_1 \neq \mu_2$ rather than $\mu_1 > \mu_2$ or $\mu_1 < \mu_2$), the rejection region for H_0 for $\alpha = .05$ is $|z| > z_{\alpha/2} = z_{0.025} = 1.96$.

Using the data from Table 4.1, we can see that for the first client, $Z = -151.26$, the absolute value of which is well over $z_{0.025} = 1.96$. Z for the second client is even higher, at an absolute value of $Z = 358.057$. For the third client, the absolute value of Z is 419.870. Because all three of these fall within the rejection region, the null hypothesis is rejected, and the alternate hypothesis is accepted. There is a statistically significant difference between the true mean CPU times of the augmented and unaugmented clients, as could be expected by looking at these values.

Complete CPU data for each client in each policy configuration can be found in Appendix B.

4.4 Memory Usage

Another important consideration when examining overhead is how many bytes a particular program requires. A program with a large memory footprint can be detrimental to a computer's overall performance, as it may have to switch to virtual memory, which is computationally expensive to

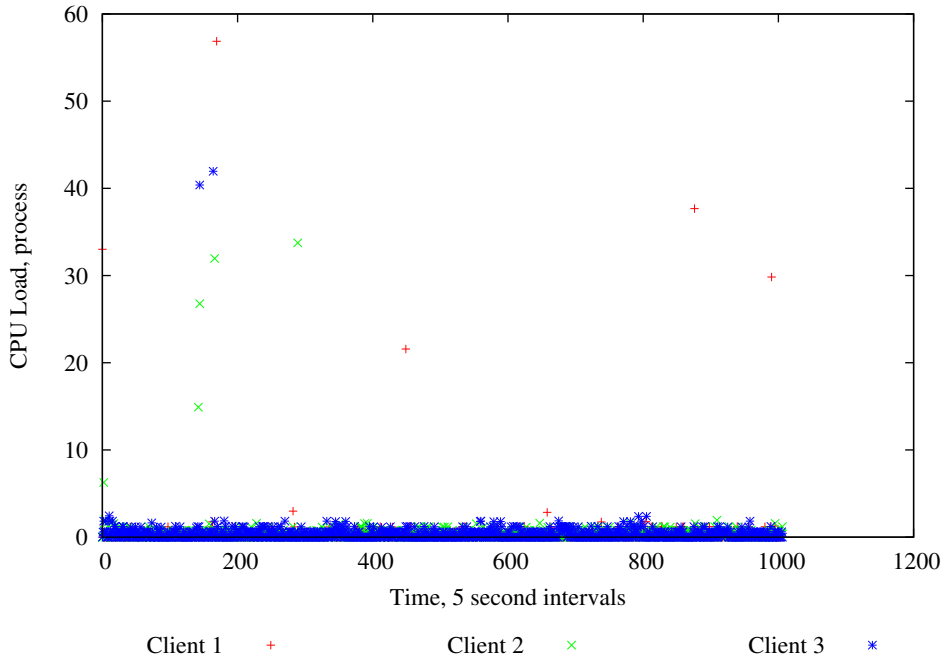

Figure 4.1: CPU Load for the Unaugmented Clients

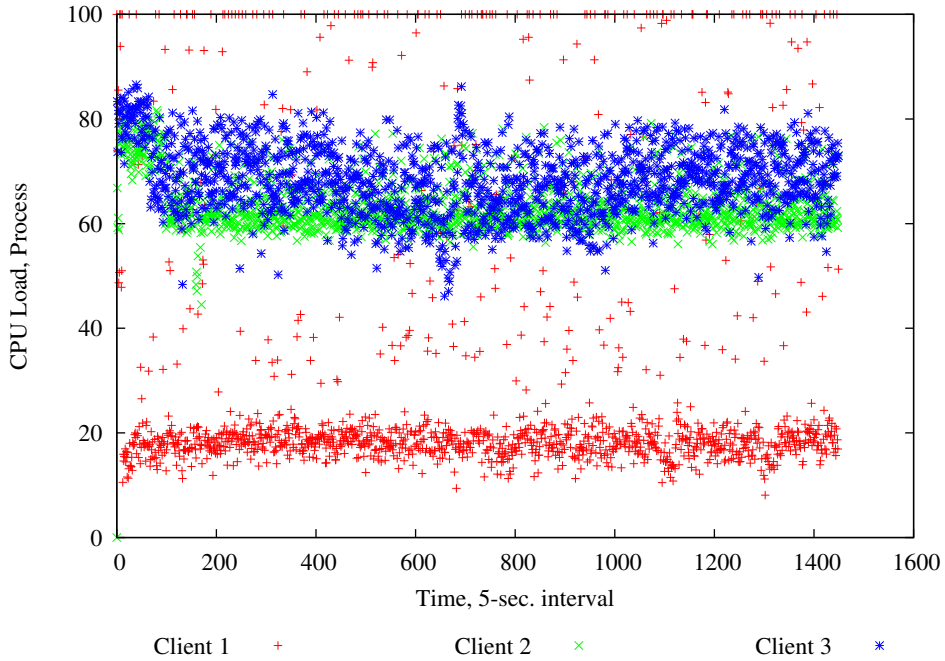

Figure 4.2: CPU Load for the Augmented Clients in a Conservative Policy Configuration with Centralized Storage

	Minimum	Maximum	Mean	Standard Deviation
Client 1	2.473E7	2.722E7	2.519E7	1.77E5
Client 2	2.417E7	2.668E7	2.469E7	1.476E5
Client 3	2.344E7	2.582E7	2.450E7	1.602E5

Table 4.2: Memory Consumption in Bytes for the Unaugmented Clients, N=1006 samples

	Minimum	Maximum	Mean	Standard Deviation
Client 1	3.071E7	3.497E7	3.419E7	3.349E5
Client 2	2.490E7	4.566E7	3.874E7	4.301E6
Client 3	2.662E7	3.385E7	3.302E7	5.236E5

Table 4.3: Memory Consumption in Bytes for the Augmented Clients, Centralized, Conservative, N=1449 samples

access.

By examining the amount of memory that both the framework and the basic client consume over time, the overall memory consumption can be determined. Of importance are the minimum, maximum, mean, and standard deviation. These give upper and lower bounds for consumption, as well as the expected amount of memory needed. For comparison, the centralized conservative policy configuration will be used, as it was for examining the CPU load.

The difference in mean memory requirements between the augmented and unaugmented clients (shown in Tables 4.2 and 4.3) can be seen to be about 10 MB. While this is certainly a large difference, and to be expected given the complexity of the framework, it is difficult to tell whether this is significant to the user – this amount is less than the base amount required to run even the simplest client, and the amount of base memory found in modern computers is typically much greater.

Given the huge amount of bytes involved, it is interesting to ask whether this difference is just a slight variation from a common mean. Let the null hypothesis, H_0, be that the mean memory consumption of the augmented and unaugmented clients are identical, and the alternate hypothesis, H_A, be that the mean memory consumptions are not identical.

Using the data found in Tables 4.2 and 4.3, Z values for each client are calculated. These values are included in Table 4.4.

As can be seen, all of the absolute Z values fall in the rejection region of $Z > 1.96$. Thus the null hypothesis is rejected, and it is shown that there is a significant statistical difference in the mean memory consumption between the unaugmented and augmented clients.

	Client 1	Client 2	Client 3
Z	864.81	124.26	581.65

Table 4.4: Absolute Z Values for Memory Consumption for Each Client

4.5 Call Times

While examining the consumption of system resources such as processor cycles and bytes of memory is important, perhaps the most important thing to examine is the length of a call, from packaging the SOAP request, to decoding the response given by the server. The framework will consume more resources than a stand-alone web services client, but if the call times are identical (or nearly so), then the QoS experienced by the user has not degraded as a result of the framework itself.

To measure call times, a timer is set up client side. It starts timing just before the client packages the SOAP request, and ends timing after the server response has been decoded. This length is the call time, which includes packaging, sending the request, having the server process the request, and decoding the response sent by the server (if any). It encompasses several different operations by both the client and the service; thus, it may be influenced by a number of different factors. It remains useful for judging the overhead of the system, as the user will notice a significantly degraded call time. But the cause of the degradation, should it occur, is crucial. If the cause is at the network level, then the framework should not be blamed.

Examining Tables 4.5 and 4.6, we can see that there is a difference in the mean times. There are two things to consider: whether the difference is statistically significant, and whether the difference would be noticeable by the user.

To determine statistical significance, we will, for each client and service, set H_0 to be that the mean observed call times for the unaugmented and augmented clients are equal. The alternate hypothesis, H_A, is that they are not equal. The test statistic, Z, is the same as in Sections 4.3 and 4.4.

The Z value for each client and service can be found in Table 4.7. As before, H_0 is rejected for $|Z| > 1.96$. It can be seen that in some cases, the null hypothesis is rejected, but not always. All three clients have no significant difference in the call times for Service 2. As well, for Service 3, Clients 2 and 3 have no significant difference in the call times. But what is evident is that for the excellent service, Service 1, the low mean coupled with a small standard deviation means that the additional features of the augmented client causes a significant statistical difference in the mean call time. But what is also interesting is whether this statistical significance is actually significant to the end user. As an example, the difference between 0.018 and 0.0129 (the mean call times for Service 1 by Client 1) is 0.0051 seconds.

The significance of the difference in call times is dependent on whether the user of a web service

	Service 1	Service 2	Service 3
Client 1, Unaugmented	0.0129	5.460	30.012
Client 1, Augmented	0.018	5.38	26.882
Client 2, Unaugmented	0.0133	5.392	28.837
Client 2, Augmented	0.097	5.767	28.526
Client 3, Unaugmented	0.0180	5.734	29.095
Client 3, Augmented	0.100	5.876	27.589

Table 4.5: Mean Call Times, in Seconds, N=250 Calls to Each Service

	Service 1	Service 2	Service 3
Client 1, Unaugmented	0.017	3.501	9.256
Client 1, Augmented	0.021	3.355	12.473
Client 2, Unaugmented	0.013	3.486	9.968
Client 2, Augmented	0.173	3.316	12.056
Client 3, Unaugmented	0.032	3.362	10.241
Client 3, Augmented	0.595	3.387	12.144

Table 4.6: Standard Deviation, in Seconds, for Calls from All Clients, N=750 Calls

client is human, or another automatic process. Simple reaction time is defined as "the interval between the onset of the stimulus and the response under the condition that the subject has been instructed to respond as quickly as possible" [38]. Mean reaction time varies depending on age, but for college-aged individuals, the mean visual reaction time is 190 milliseconds [39]. So if the difference between two call times is less than this value, it is fair to assume that the user would not notice the difference.

However, web services are not used solely by humans. Automated processes, such as computer programs, can make use of services automatically. The different response times makes more of a difference when the user of a service is a computer – unlike people, who need a certain amount of time to process information, a computer can respond instantaneously. The difference in call times is then far more important when the user of a service is automated.

	Service 1	Service 2	Service 3
Client 1	2.985	0.261	3.186
Client 2	7.628	1.232	0.314
Client 3	2.176	0.470	1.499

Table 4.7: Absolute Z Values for Call Times for Each Client and Service

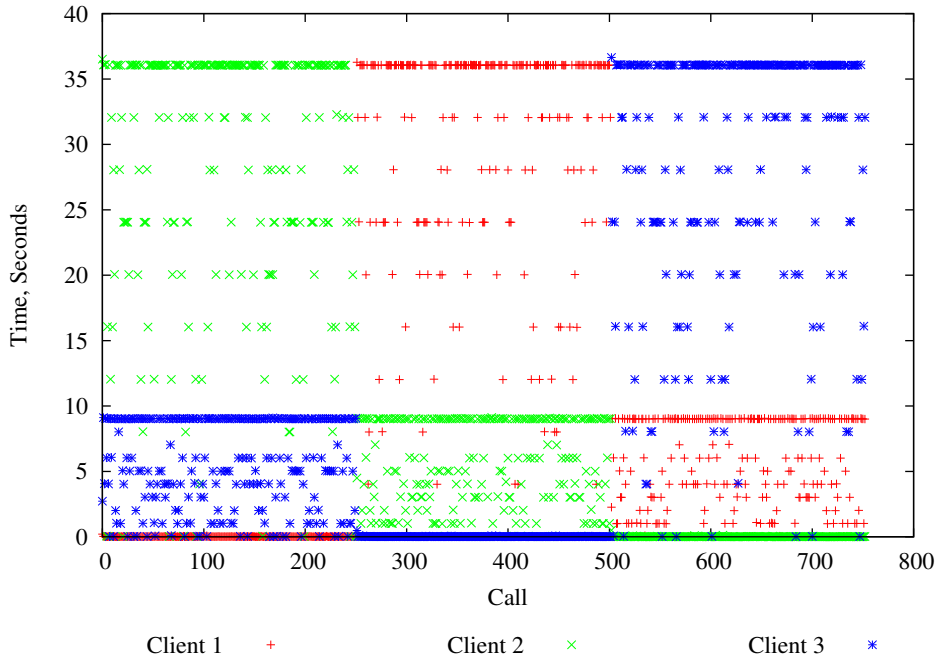

Figure 4.3: Visualized Call Times for Each Client

4.6 Bandwidth Usage

To test bandwidth usage, the network protocol analyzer Ethereal (http://www.ethereal.com/) was set up on each of the client machines, so that the packets sent to and from these machines to the web services, QoS repository, and each other could be captured and counted.

To ensure that only packets from the framework were analyzed, a filter was created to include only the machines specifically delegated for the experiment. For the centralized clients, the filter was extended to include the restriction of HTTP-only traffic, as clients using the centralized selection framework use HTTP requests both to call the web services, and to get or post information to the QoS repository.

When the clients were in a decentralized configuration, the filter was updated so that only HTTP packets to and from the services, and packets sent to port 8082 on the clients, are displayed. The packets to port 8082 are requests for QoS information from peers; storing QoS interactions is done locally, by inserting them into a database, and thus does not actually consume any network bandwidth.

The results of both an unaugmented client, and augmented clients in each policy configuration, are shown in Figure 4.8. Besides logging just the number of bytes sent and received by the program, a number of other factors are considered. The number of successful calls gives an indication of the

	Success. Calls	Total Calls	Switches	Bytes	Avg. Bytes/Sec
Unaug. Client	574	750	0	1797206	99.334
Client 1					
Cons., Cen.	635	750	2	8680925	1142.263
Cons., Decen.	658	750	0	2173617	148.176
Reac., Cen.	747	750	3	11512126	1167.917
Reac., Decen.	747	750	3	2966122	341.433
Recur., Cen.	709	750	12	9228768	2699.604
Recur., Decen.	674	750	11	9230228	2700.031
Client 2					
Cons., Cen.	505	750	3	12177728	804.350
Cons., Decen.	668	750	0	2142640	134.295
Reac., Cen.	747	750	3	12453653	1557.420
Reac., Decen.	749	750	1	2872761	305.270
Recur., Cen.	708	750	13	17903121	2088.493
Recur., Decen.	715	750	13	9440969	983.628
Client 3					
Cons., Cen.	750	750	3	12921628	1325.191
Cons., Decen.	662	750	0	2188269	138.348
Reac., Cen.	747	750	3	12453638	1191.785
Reac., Decen.	749	750	1	2647828	249.304
Recur., Cen.	723	750	14	18533033	1871.491
Recur., Decen.	727	750	14	9471073	859.552

Table 4.8: Network Usage Statistics for Each Client

success of the policy with respect to the baseline of the unaugmented client. The number of times the client switches services is also a useful measurement, and the number of bytes per second measures the number of bytes sent and received divided by the time, in seconds, from first to last filtered packet.

The lesser network usage of the decentralized clients can be explained by the fact that the decentralized clients store all QoS interactions locally, rather than sending them either one-by-one or batched to a central repository. Due to the size of these interactions, local storage can be seen to save a great deal of bandwidth.

It is also evident that the recurrent configuration has at least as much network overhead, and often more, than the other policies. Given that under that configuration, the client queries and reasons every ten minutes, this shouldn't be too surprising. And as can be seen, the reactive configuration usually has far more successful calls that not, and all while having a low to moderate network footprint.

4.7 Conclusions

The results of mean CPU load, memory consumption, network usage, and call times all show that the framework does add a significant amount of overhead over a basic web services client. But what remains uncertain is the practical effect of this overhead. The time required to create

a SOAP call and parse the result increases, but often by such a small timespan that the result might not be evident to the user. The memory usage is the most concerning, with the augmented client requiring around 10 MB more than the basic web services client. This could partially be solved by optimization of the framework's code, but the overhead remains concerning. Much of the CPU overhead is the result of the time monitoring mechanism of the policy manager, which could potentially be improved in a later version of the framework – there are inefficiencies in the code.

While there is a cost associated with the framework, its impact on day-to-day usage of the client is less certain, and optimizations could certainly be made. If the user of the framework is a human, then the overhead is less costly, since the difference in call times could, depending on the services, not even be noticed. But if the user of a service is an automatic process, then the differences between the basic clients and the augmented ones become more prominent, as the differences that might have been undetectable by a person are now apparent each and every time the client makes a call.

Chapter 5

Experiment 2: Performance of the Framework

In Experiment 1, it was shown that there is a certain overhead associated with using the framework. CPU and memory usage for the augmented and unaugmented clients were found to differ significantly, and the mean call times to a particular service were also found to be higher.

In the previous experiment, the call times measured were to a fixed service. That is, 250 calls were made to a particular service, and the times to do so recorded for statistical testing. This was solely for the purpose of measuring overhead, for the entire purpose of the framework is to enable the client to switch to a different service should another service present better QoS.

This experiment concerns the overall speed of the framework, and consists of two parts. In the first, the clients will run in both centralized and decentralized architectures, with each architecture testing the three policy configurations identified in Section 4.1. The fastest policy (that is, the policy which has the lowest mean call time) will be identified for the experiment setup. In the second, the speed of the centralized and decentralized retrieval methods will be tested, with each approach retrieving a varied number of QoS interactions. This will help illuminate the precise speed differences between the two storage/retrieval architectures.

5.1 Experimental Setup

The hardware configuration for this experiment is identical to that of Experiment 1, detailed in Section 4.2. The web services' quality is also the same, with one excellent, one adequate, and one poor service made available to the clients. It is important to note that this choice in the QoS distribution may lead to certain policy configurations being superior to others for minimizing mean call time, when in another setup (say, three services of identical QoS), this might not be the case.

To test the mean call time, 750 total calls, divided up into three parts, are made to the web services. A client is configured to call a certain web service at the beginning of each 250-call part, but this can change depending on whether the client has determined that another service is better than that which is currently used.

Because the clients in this experiment use QoS data to make decisions about the best possible service, a decision had to be made as to what sort of QoS information would be initially available.

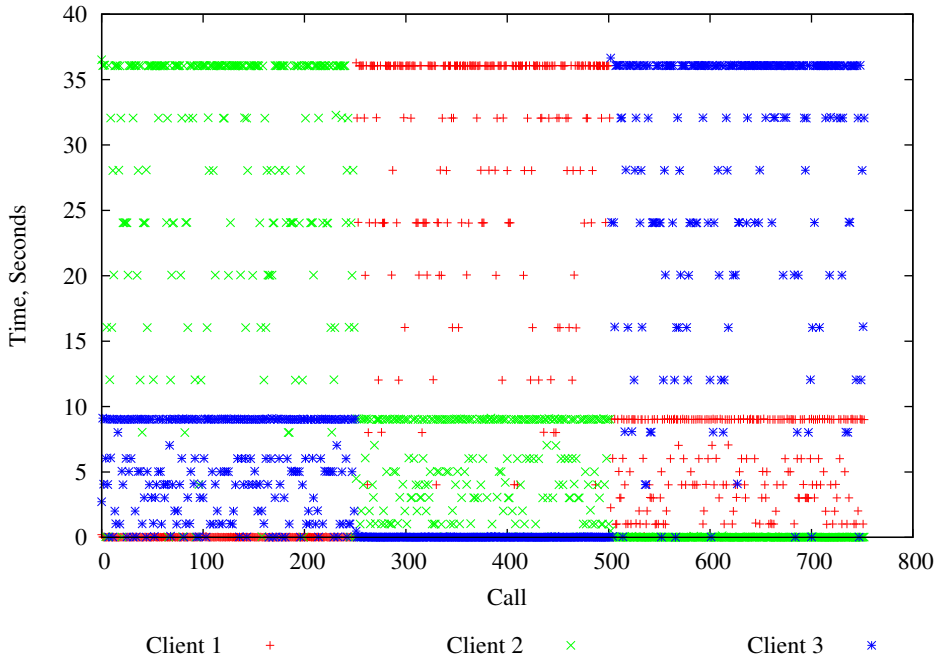

Figure 5.1: Visualized Call Times for the Unaugmented Clients

In this experiment, clients work with initially no knowledge about any of the services. That is, the only knowledge they use is the information generated by themselves.

5.2 Mean Call Times

To measure the mean call times, unaugmented clients running on three separate machines are compared to augmented clients. For completeness with regards to the augmented clients, data is collected for each policy configuration mentioned in Section 4.1. To keep the number of runs reasonable, the number of QoS interactions retrieved is fixed at 10.

The unaugmented clients call only the web service originally specified. The results of this can be seen in Figure 5.1. One service has very high call times, one moderate, and one very low. The clients then switch between these services, which results in the pattern shown in the graph. The call times graphed are exactly those used in Section 4.5. The mean and standard deviation for calls to the individual services can be found in Tables 4.5 and 4.6, respectively, while the overall means and standard deviations can be found in Table 5.1. These mean call times represent the mean amount of time for all 750 calls. For the unaugmented clients, all three services are represented in the calls, including the terrible service, which is why the mean time is so high.

The mean and standard deviation of the length of call times for each client in each of the three major policy configurations are in Table 5.2. The table shows how long each configuration takes,

	Mean	Standard Deviation
Client 1	11.828	14.244
Client 2	11.413	13.920
Client 3	11.615	14.033

Table 5.1: Overall Mean and Standard Deviation of Call Times, in Seconds, for the Unaugmented Clients

on average, to call the services 750 times (250 calls, 3 times). To test the goodness of these policy configurations versus the unaugmented clients, a statistical test similar to those of Experiment 1 is performed. Let H_0, the null hypothesis, be that the mean call time for each client is identical to the mean for a particular policy configuration for that client. Let H_A, the alternate hypothesis, be that the two are not equal. As with the previous experiment, data is assumed to be normally distributed. The test is performed for each policy configuration for each client, in both centralized and decentralized configurations.

The absolute Z values are shown in Table 5.3. These Z values are calculated from the values in Tables 5.2, 4.5, and 4.6. All of the policies are at least as good as using an unaugmented client. The decentralized, conservative policy for Clients 1 and 2 are no better or worse than the unaugmented equivalents, having absolute Z-values less than 1.96. Every other policy configuration is better, having Z-values greater than 1.96. Looking at the table, the best policy configuration for the experimental setup is the reactive policy, with the recurrent policy also a strong choice. The mean call time using the reactive policy is never more than 1.070 seconds, with the recurrent being no more than 2.436 seconds. Both policies allow the clients to find a good service quickly, and then stay with it. As an illustration of this, consider Figures 5.1. and 5.2. In the first, an inability to switch services means that when a client encounters poor QoS, it is unable to deal with it. By contrast, the latter figure shows that poor QoS is quickly countered by switching to a service which is more likely to provide better QoS.

5.3 Speed of the Centralized and Decentralized Retrieval Architectures

Originally, the framework used only the centralized retrieval method described in Section 3.2.1 [9]. But after a crash in the QoS repository, it was realized that the centralized architecture led to a single point of failure. This quickly led to the development of a P2P decentralization approach, with the aim being to increase the robustness of the framework.

As described in Section 3.2.2, the P2P decentralization is much more robust than using the centralized approach. Even if a client knows about no other peers, it can at least draw on its own

	Mean	Standard Deviation
Client 1		
Conservative, Centralized	9.312	14.851
Conservative, Decentralized	11.244	14.228
Reactive, Centralized	0.355	3.305
Reactive, Decentralized	0.087	1.335
Recurrent, Centralized	0.988	4.858
Recurrent, Decentralized	0.968	4.556
Client 2		
Conservative, Centralized	2.192	3.558
Conservative, Decentralized	11.759	14.295
Reactive, Centralized	0.334	1.843
Reactive, Decentralized	1.070	3.054
Recurrent, Centralized	2.436	5.403
Recurrent, Decentralized	1.023	4.816
Client 3		
Conservative, Centralized	2.061	3.379
Conservative, Decentralized	3.834	4.636
Reactive, Centralized	0.603	5.638
Reactive, Decentralized	0.947	2.769
Recurrent, Centralized	1.555	4.755
Recurrent, Decentralized	1.501	4.635

Table 5.2: Call Time Statistics, in Seconds, for Each Client and Policy Configuration

	Z
Client 1	
Conservative, Centralized	3.34
Conservative, Decentralized	0.79
Reactive, Centralized	21.487
Reactive, Decentralized	22.475
Recurrent, Centralized	12.006
Recurrent, Decentralized	19.887
Client 2	
Conservative, Centralized	12.373
Conservative, Decentralized	0.475
Reactive, Centralized	21.608
Reactive, Decentralized	19.876
Recurrent, Centralized	16.465
Recurrent, Decentralized	19.318
Client 3	
Conservative, Centralized	18.127
Conservative, Decentralized	14.419
Reactive, Centralized	19.941
Reactive, Decentralized	20.425
Recurrent, Centralized	18.594
Recurrent, Decentralized	18.742

Table 5.3: Absolute Z-values for Comparison Between Augmented and Unaugmented Means

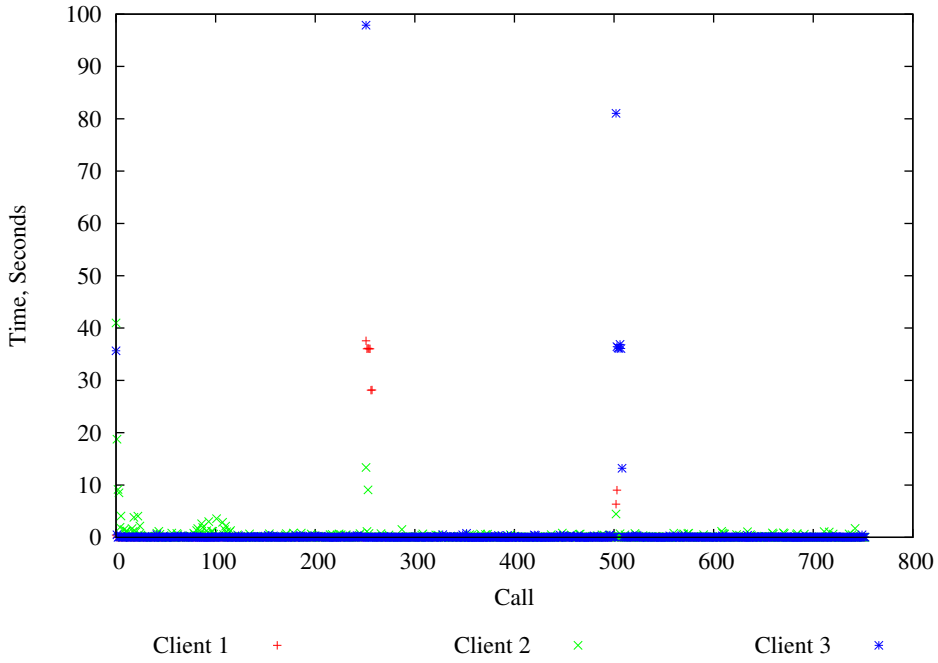

Figure 5.2: Call Times for the Centralized Reactive Policy Configuration

interactions with the web services to train its reasoning engine. But at what cost?

Using both retrieval architectures, a client was set to retrieve 10, 25, 50, 100, 200, 300, and 400 interactions. As can be seen in Figure 5.3, the retrieval times vary from approximately one second for 10 interactions, to 15-30 seconds for 400 interactions, depending on the architecture used. Of interest is the fact that the time to retrieve a number of models seems to be approximately linear to the number of models retrieved, regardless of the architecture selected. The decentralized architecture takes roughly the same amount of time to retrieve small amounts of interactions as the centralized, but becomes slower and slower as the number of interactions increases. This is likely due to receiving the interactions from multiple sources, peers, rather than receiving them all in a batch from the central QoS repository.

5.4 Conclusions

Most of the policy configurations described in Section 4.1 provide a significant increase in the mean call time, given the experimental setup. Though the centralized retrieval architecture is faster than the decentralized, both are roughly linear to the number of interactions retrieved. The choice of architecture did not seem important towards the mean call time: only five of the nine policy configurations for the three clients were faster under the centralized architecture. However, part of this may stem from the fact that the number of models was fixed at a relatively low value of

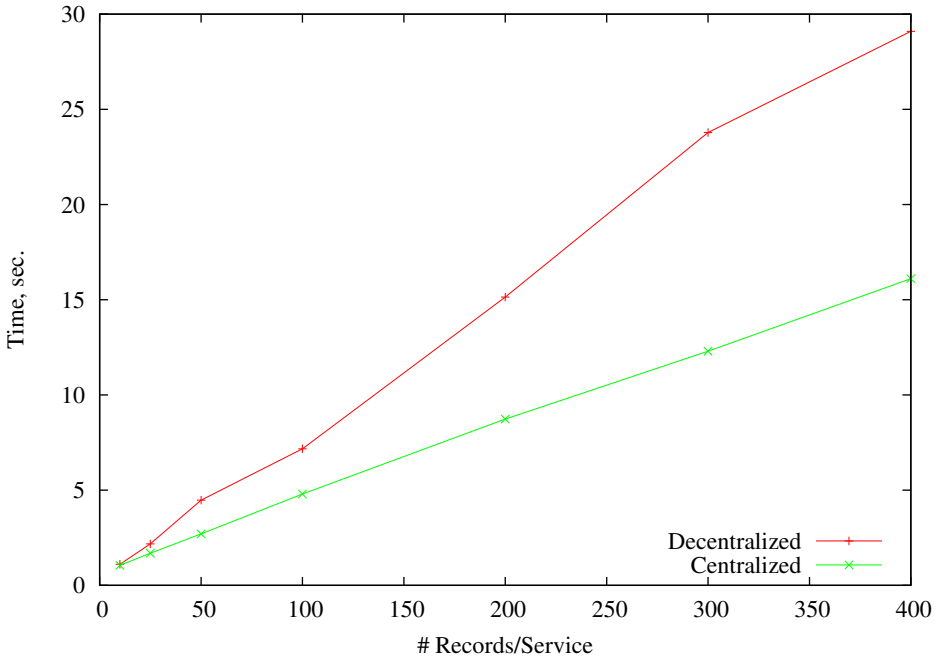

Figure 5.3: Retrieval Times for the Centralized and Decentralized Retrieval Architectures

10. Future work could include varying the number of interactions fetched at each query, and then seeing how this impacts the mean call time.

There is clearly a large benefit, reflected in the mean call time, to using the framework. Provided that a best service exists, the reactive policy configuration is an excellent way to control the client's execution. And, as demonstrated in Experiment 1, it provides both an excellent success rate, as well has having a bandwidth usage that is either no worse, or better, than the other configurations.

Chapter 6

Experiment 3: Accuracy of the Framework

In Experiments 1 and 2, the overhead of the framework, and the speed of both calls and retrieval were examined, respectively. In this experiment, the subject is the heart of the framework: the reasoning engine, which allows selection between services. There is a time cost to gathering the data, as was shown in Experiment 2. But the reasoners must also be built, and in the case of the naive Bayes classifier, trained with training and test data.

This experiment measures three things. First, the time required to built up the two different types of reasoners described in Section 3.3 will be measured. Second, the accuracy of the classifications of the Naive Bayes classifier will be studied, for varying amounts of interactions. Both standard accuracy and the kappa statistic will be examined. Finally, this experiment will also examine how well clients running in the framework can deal with recommendations given by other peers in their neighbourhood.

6.1 Experimental Setup

To test the reasoners, training data in the form of QoS interactions is needed. A three-client, three-service setup like the ones used in the previous experiment are used. The clients are set to use the centralized storage/retrieval architecture, so that all the interactions are stored in the QoS repository. As the focus is solely on the reasoning component, the storage/retrieval architecture is largely irrelevant for this experiment.

Once the clients are so configured, timing mechanisms are added to the code, so that the time taken to retrieve data and build a reasoner based on it can be precisely measured. A client is then used to build up both rule-based and naive Bayes reasoners, and the times measured. This is done for 10, 25, 50, 100, 200, 300, and 400 QoS interactions.

All timing measurements are done on the main client machine, a 3GHz P4 machine with 2 GB RAM.

6.2 Accuracy of Service Classification

Three questions must be answered about the classification accuracy: first, can a high accuracy be achieved? Second, how many interactions are required to train the reasoner to a certain accuracy? And finally, what are the time requirements to do so?

To test the accuracy of the naive Bayes classifier, three web services were set up. Each of the three clients interacts with each service, and each provides 150 interaction snapshots for each service to the central QoS repository.

To test the rule-based expert system, the clients ran with 10, 25, 50, 100, 200, 300, and 400 interactions retrieved per service. Each time, the reasoner was able to determine that the excellent service provided the best QoS: it had the highest weighted sum each time, for the default weights (described in Section 3.3.1).

The same test was applied to the naive Bayes reasoner: it was set to retrieve a fixed number of models equal to the numbers retrieved for the expert system. Each time it was able to recognize that the excellent service was the best. However, its accuracy on the data set created from the received interactions is initially low (for ten interactions), but improves dramatically as the number of interactions increases as well. Thus, as the number of interactions increases, so does the trust that the recommendation is based on a high degree of internal accuracy. The accuracy of the naive Bayes classifier is shown in Figure 6.1.

This accuracy is tested using 10-fold cross validation. Unlike the traditional holdout method, whereby data is divided into training and test sets, and a function learns using the training set only, n-fold cross validation divides the data set into n subsets, and the holdout method is repeated n times, with the average error computed across all the repetitions. One of the problems with the holdout method is that the learning function can be very different depending on what data is put into the training set. n-fold cross validation eliminates that problem by ensuring that each point in the data set will be included in the training data $n - 1$ times, and the test data exactly once.

The main disadvantage to the approach is the computation time. As described above, the algorithm takes n times longer than the traditional holdout method, as the algorithm is repeated n times. However, this disadvantage is not a problem in this experiment, as evaluation by cross-validation is generally not performed in the framework. While building up the reasoning engine is a task performed with some regularity, evaluation of the reasoner falls outside day-to-day use.

While accuracy is very important in looking at the utility of classifiers, there are other statistics that can be considered. One such statistic is the kappa statistic. As described by Carletta [40], the kappa statistic is a measure of class accuracy within a classifier. It corrects for chance agreements between classes, and is described by Equation 6.1. The literature [40, 41] refers to "coders"; in machine learning, one can think of two "coders" being a classifier trained on the dataset, and the

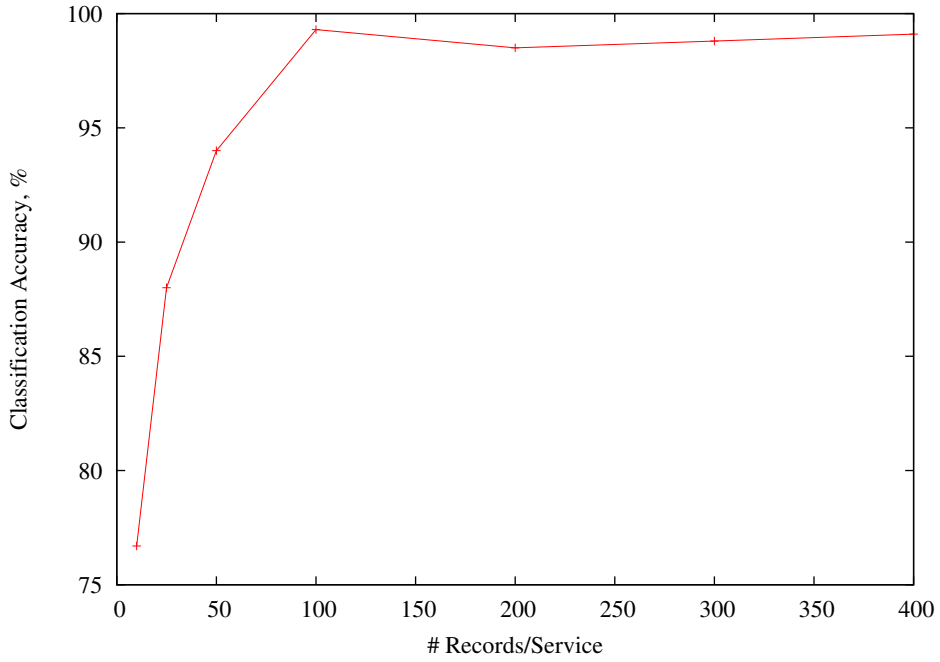

Figure 6.1: Accuracy for the Naive Bayes Classifier

true values (which could be thought of as a "perfect classifier").

$$\kappa = \frac{P(A) - P(E)}{1 - P(E)} \tag{6.1}$$

In Equation 6.1, $P(A)$ is the proportion of times that the classes agree, and $P(E)$ the proportion of times that the classes are expected to agree by chance.

The "chance agreement between classes", $P(E)$, has been the source of much discussion. Di Eugenio and Glass point out that there are two methods of computing this term in the computational linguistics literature [41], and that for each approach, estimation is done of this probability distribution. They write that the first approach, described by Cohen [42], involves each coder having a personal distribution. The latter approach, described by Siegel and Castellan [43], among others, has a single distribution for all coders, derived from "the total proportions of categories assigned by all coders" [41]. The details of the computation of $P(E)$ for both approaches is beyond the scope of this thesis, but both are shown, side-by-side, by di Eugenio and Glass [41], and may be of interest.

Quoting Krippendorf, Carletta 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 graph of the κ values, shown by Figure 6.2, proves useful.

For 10 interactions retrieved/service, the κ value is 0.58 – not enough for even tentative conclusions. But by the time 25 interactions per service are retrieved (for a data set of size 75), κ becomes 0.82, just above the threshold required for good reliability. As the size of the data set grows, so too

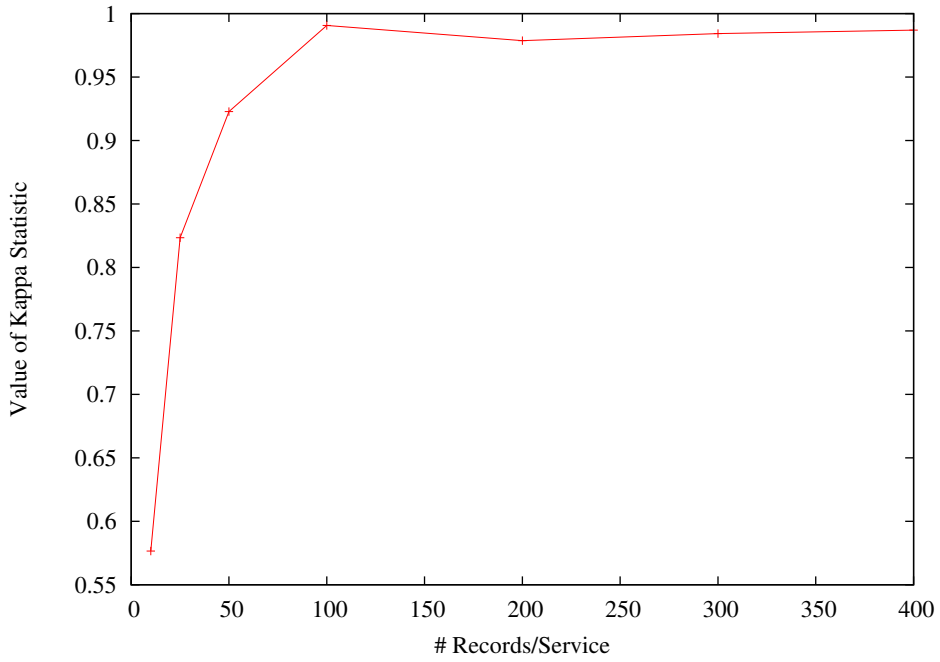

Figure 6.2: Kappa Statistic for Increasing Numbers of Records

does κ, so that by the time the data set reaches 150 (50 interactions per service), κ is above 0.9.

It must be pointed out that these thresholds are arbitrary; Krippendorf meant for these to be guidelines, rather than absolute truths. Still, if one treats these as guidelines, then tentatively, 25 interactions allows the conclusion that the accuracy within the classifier is quite good, relative to the actual data.

From Figures 6.1 and 6.2, it can be seen that a large data set greatly increases the accuracy of the classifier. However, there are costs associated with this. Each of the interaction snapshots contains a great deal of information, and sending a large number of these over a network can take a great deal of time, never mind the cost of building up the reasoners themselves.

As can be seen in Figure 6.3, the expert system-based client takes less time in each case to query the QoS repository and process the results. Part of this, however, comes from the fact that the Naive Bayes reasoner parses all the system context information in the interaction snapshots, which contains information not only about the system's memory and CPU usage, but also information about all the processes currently running.

Clearly there is an opportunity cost for accuracy: increased accuracy requires an increase in time. It is an open question as to how much time a user could tolerate. How important this cost is depends on the system configuration. Because the option exists for cacheing of reasoning engines, this could be a one-time cost, with the results stored to disk and loaded each time thereafter. But without cacheing, the client might have to deal with these costs each time it gathers data and builds its reasoner. Depending on the policy configuration, this could be costly indeed.

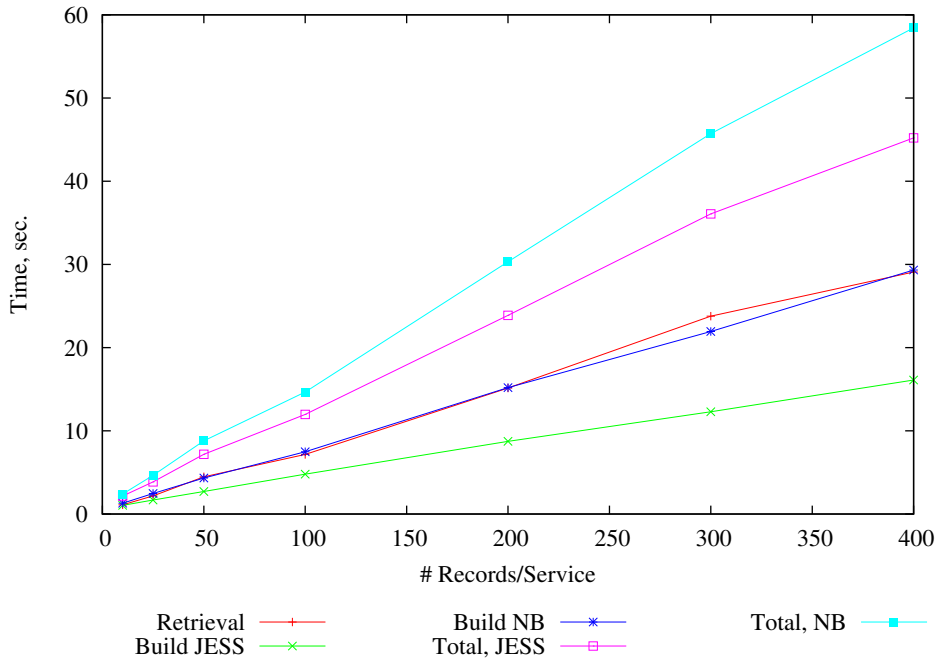

Figure 6.3: Elapsed Time for Retrieving Records and Building Reasoners, Centralized Retrieval

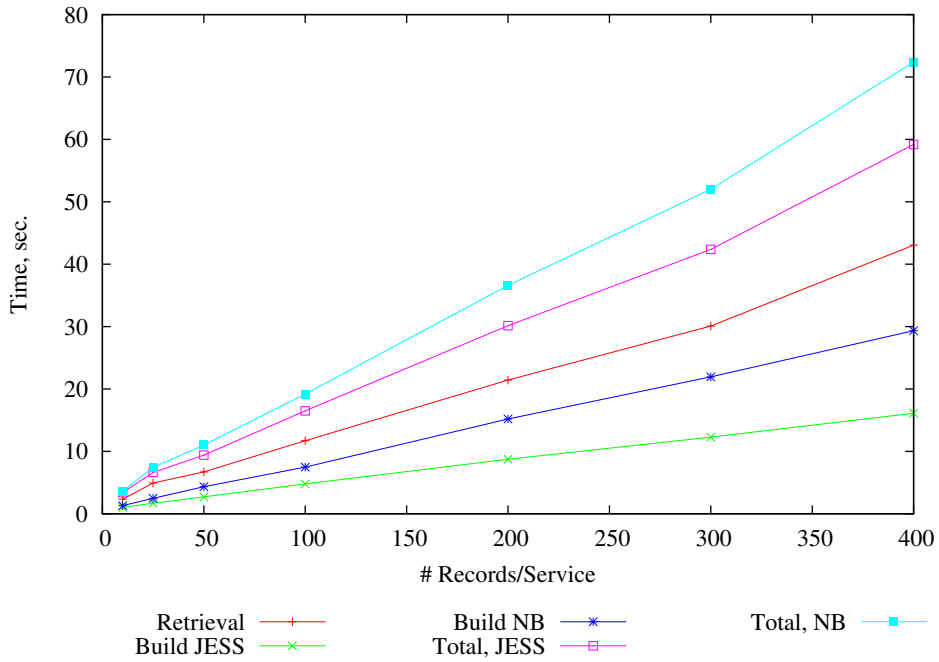

Figure 6.4: Elapsed Time for Retrieving Records and Building Reasoners, Decentralized Retrieval

6.3 Recommendations and Peer Modelling

Besides processing raw data received from peers or a QoS repository, the reasoner also has the task of dealing with recommendations. As detailed in Section 3.3.3, recommendations have the advantage of being much faster than dealing with raw data. However, there is also the problem that a number of peers might not be entirely truthful about their recommendations. This might be due to maliciousness, or it might just be due to the fact that the reasoning engine was built on information that has since become outdated.

To deal with this, a basic peer modelling approach, also detailed in Section 3.3.3, was developed. But without proper testing, it would be nothing more than a theoretical construct.

To test the peer models, a client (designated the "main client") interacts with five other peers in its network. Two are trustworthy – their reasoning engines are up to date, and they give recommendations based on the service recommended to them by their own reasoning engine. The remaining three clients are malicious and untrustworthy. Their data is completely contrary to the main client's for each service, and is created by querying the main client for its data. However, instead of using it to assist in building their reasoning engine, the malicious clients instead flip the availability and reliability data. Given that the experiment involves the same QoS distribution of the web services as in the previous experiments, this flipping of data will cause the two malicious clients to recommend services that have poor QoS.

The main client knows about one of the peers, which knows about all the remaining peers. Thus, the main client builds a master peer list of all the peers used for this experiment.

The three non-malicious clients gather data about the services by collecting 250 interactions about each. This data is stored in the local databases of each machine. The three malicious clients keep querying the main client, and adding false data from its interactions, as described above, so that their recommendation will be Service C.

The clients recommend services as in Table 6.1. The service is selected by applying Equation 3.2 to the recommendations as follows:

$$S_A = 0.5 * 1 + 0.5 * 1 + 0.0 * 0 + 0.0 * 0 + 0.0 * 0 = 1.0 \tag{6.2}$$

$$S_B = 0.5 * 0 + 0.5 * 0 + 0.0 * 0 + 0.0 * 0 + 0.0 * 0 = 0.0 \tag{6.3}$$

$$S_C = 0.5 * 0 + 0.5 * 0 + 0.0 * 1 + 0.0 * 1 + 0.0 * 1 = 0.0 \tag{6.4}$$

The other clients have trust scores of only 0.5 because they gathered their data about the services at different times than the main client; because the main client is thus unsure as to whether their data is useful or not, the rating of 0.5 is given, as justified in Section 3.3.3.

	Trust, S_A	**Trust, S_B**	**Trust, S_C**	**Vote**
Client 2	0.5	0.5	0.5	Service A
Client 3	0.5	0.5	0.5	Service A
Malicious Client 1	0.0	0.0	0.0	Service C
Malicious Client 2	0.0	0.0	0.0	Service C
Malicious Client 3	0.0	0.0	0.0	Service C

Table 6.1: The Main Client's Trust Values for Each Peer for Each Service Considered

The main client computes the sums for the services as in Equation 6.2, and selects the highest-weighted service, Service A. Were there no peer models, a simple majority voting algorithm would have selected Service C, which provides very poor QoS. The peer models can thus be seen to be a useful first step in countering malicious peers.

6.4 Conclusions

The naive Bayes-based reasoners can acheive a high degree of accuracy from a relatively small data set. The accuracy and kappa statistic increase at roughly the same rate; if Krippendorf is to be believed, and if a kappa value of > 0.8 is considered good reliability, then somewhere in the range of 10-25 models should be sufficient for a general purpose classifier, given the high degree of accuracy acheived in that range as well. This is dependent on a sufficient number of instances of each category being present in the data.

However, the information contained within the QoS interactions, combined with a verbose RDF/OWL-based implementation, means that there can be a significant delay in retrieving the interactions. As can be seen in Figures 6.3 and 6.4, the time to retrieve 10-25 models and train the reasoner can take between around 2 and 10 seconds, depending on the retrieval architecture used. While this is alleviated somewhat by having the reasoning engine running in a separate thread of execution from the basic web services client, this cost can become more or less serious depending on policy configuration in which the framework is currently running, and whether or not basic cacheing is used. As well, the time to retrieve the models and build the reasoner will certainly vary depending on the network connection, and the hardware configuration on the client machine.

Because a client cannot be sure of anything other than its own measurements, the peer modelling approach outlined in Section 3.3.3 has proven quite useful in identifying malicious clients. However, it is still limited: it relies on obviously contrary data from a point in time close to when the client gathered its own data. While still useful in its current state, the peer modelling algorithm could certainly be modified to be more robust and general.

Chapter 7

Summary & Conclusions

The QoS-based framework includes ideas from a number of areas: policies from autonomic computing, reasoning techniques from artificial intelligence, and ontology languages from the semantic web community, to name a few. These three ideas played a key role in the framework, regulating behaviour, guiding the clients to better services, and reporting interactions needed for the decision making process.

7.1 Action Policies: A Powerful Approach to Behaviour Specification

Walsh and Kephart [35], in their paper on policies in autonomic computing, write that policies act as "any type of formal behavioral guide." These guides allow for flexible external specification of behaviours with a system. In the framework described in this thesis, action policies are used to describe how the framework should act in certain circumstances. In the initial phases, querying and reasoning were specified programatically to occur at start-up [9]. With the addition of a policy manager to the framework, and the specification of a basic policy language, querying, reasoning, switching, and reporting could all be controlled externally.

This, in turn, allowed for testing of what was termed policy configurations. In Section 3.4.1, conservative, reactive, and recurrent policy configurations were specified in general terms – the precise specification for each may be found in Appendix C. These policies allowed for different sets of behaviours for the framework. The different behaviours then had an impact on the client itself: for example, an augmented client using a reactive policy configuration would query and reason over the query data with each failed call. Initially, it was thought that this would be an expensive policy, both computationally, and in terms of network resources. But combined with the findings of Experiment 3, in which it was learned that a naive Bayes-based reasoner needs relatively little data to acheive decent levels of accuracy, this turned out to be an extremely effective configuration.

Indeed, the initial policy configuration used in the framework, conservative, was shown in Experiment 2 to be the least effective in terms of reducing the mean call time, and had a high standard deviation of call times. This is likely due to the fact that querying and reasoning were each invoked

only once per run of 250 calls. If the clients had no information about the services, they would often stay with whatever service they were currently using. If this was a poor service, then the mean call time for the run of 250 calls would be high, and would contribute to a high standard deviation once the client switched to the best service with far lower call times.

The recurrent policy configuration acted as a kind of a halfway-point between the other two configurations. The mean call times were better than the conservative policy, because querying, reasoning, and switching were done at predefined time intervals, rather than only once. However, the configuration was not nearly as effective as the reactive one, because the latter would query, reason, and potentially switch after each failed call. By contrast, the former policy would sometimes see a number of failed calls before the process was invoked. The recurrent configuration also had a major disadvantage of sending and receiving substantially more data, due to the use of time intervals. As can be seen in Table 4.8, the two slower clients (clients 2 and 3) had substantially more bytes sent and received, because they took longer to run for the predefined 250 calls, and thus invoked the process more often.

If the recurrent configuration is to be useful, it would likely be for processes that run and stay active for long periods. An example of such a process would be a program that uses a number of different web services to continually monitor a number of sensors. If such a program ran for days or weeks at a time, a recurrent configuration that queried, reasoned, and potentially switched services once or twice per day would not present a significant overhead, and would still be more useful than a conservative configuration. Indeed, the latter seems most useful for minimizing the resource consumption of the framework, rather than improving QoS the most.

7.2 Artificial Intelligence Techniques for Reasoning

The reasoner component of the framework had two implementations: a naive Bayesian classifier, which attempted to classify services into QoS categories; and a rule-based expert system, which created an ordered ranking of the services for the client.

Each implementation had its strengths and weaknesses. The rule-based expert system, implemented in JESS, was shown in Experiment 2 to be the quickest to train on a set of data. As well, due to its process of ranking using weighted sums, there is almost always a deterministic "best" service, given a particular data set. However, the weighted sum is something of an artificial construct, and the weights given by the user are arbitrary – it is hard to determine whether a weight of 8 or 8.5 should be given, for example. Evaluation of the expert system poses a problem as well. While evaluating the naive Bayes classifier can be done using the holdout method, or cross-validation, or a number of other techniques, there is no surefire way of evaluating the expert system other than perhaps comparing it to a previously defined ranking of the services.

The naive Bayesian classifier, whose classification algorithms are based on conditional probabilities, is a neater approach to reasoning. Instead of creating an arbitrary ranking, the classifier instead attempts to place the service into a given QoS category, based on the training data it has been given. Evaluation of the classifier can be done, as it was in Experiment 3, using n-fold cross validation. Alternately, the traditional holdout method could be used. These methods are well-studied, and known to be excellent for evaluating classification algorithms in general; they are not specific to naive Bayes. Another advantage to naive Bayes is that the classifier is able to handle incomplete data. Instances can be missing one or more attributes, and the classifier will be able to handle this gracefully. Finally, new attributes can be easily defined, and added to the data set: the classifier does the work of making sense of the relationships between the class and its attributes. By contrast, the expert system needs an expert in the field to define the relationships in the data set, which can be both time-consuming and error-prone.

The disadvantage to the classifier is that it takes longer to build than the expert system, given an identical set of training data, though the query times are both miniscule once they have each been built. There is a tradeoff between the two: increased accuracy and better evaluation methods for naive Bayes, and better speed for the expert system. The choice of reasoner, then, should be dependent on how essential speed is, as well as how often the reasoner is used. In cases where it is infrequently used, or where speed is not the overriding factor, Naive Bayes is the better choice of the two.

7.3 Ontologies and XML-Based Ontology Representation Languages

Knowledge in the framework is specified by means of an OWL ontology. Because the basic ontology currently only uses classification, rather than defining inter-relationships between terms, the ontology specification falls under the OWL Lite sublanguage.

The use of the ontology meant that the definition of the QoS parameters used in the decision making process could be specified externally, rather than in the code of the framework itself. Take the "reliability" parameter, for instance. Without a common ontology, one developer might interpret it as a "yes" or "no" string, while another might code it as a boolean value. Ontologies, used effectively, can reduce these sorts of misunderstandings and miscommunications.

The disadvantage to using an ontology language such as OWL over hardcoding specific knowledge deals with an increase in overhead. OWL is based on RDF, which is in turn based on XML. These languages are all very descriptive, and easily parsed by computers. However, the descriptiveness is also a disadvantage, in that it adds much in the way of size. Because the framework creates a QoS interaction for each call, this overhead adds up quickly. As can be seen in Table 4.8, both

centralized and decentralized storage and retrieval architectures add quite a bit of overhead over a basic client. While the centralized adds the most, due to having to both send and retrieve models over a network (rather than just retrieve, as is the case with the decentralized model), both will typically add much network overhead. Much of this is due to the representation in OWL: while it allows precise specification of the ontology, and terms in an interaction, it also increases the overhead greatly.

Ultimately, ontologies seem a useful tool for specification and standardization of knowledge. However, the current favourites for marking up data in web-based systems (RDF, OWL, OWL-S) are verbose, and can add significant network overhead. The use of these ontology languages for QoS-based information on web services should be considered in the context of whether it is likely that developers will create a number of different clients, and if so, whether facilitating standards between the developers is more important than the increased network traffic.

7.4 Centralization vs. Decentralization

Originally, there was only one option for storing and retrieving QoS information in the framework. Clients would contact the central QoS repository, and either post or request interactions. This approach worked well, and had the advantage of complete information: if a client had need for it, all data reported by all clients on a particular service could be requested. But it suffered from a single point of failure: if the repository went down, the framework was no better than an unaugmented client.

To increase the fault-tolerance in the framework, a decentralization option was added, whereby clients would store their data locally, but share information within their peer neighbourhood. While this approach is slightly slower than the centralized one, as master peer lists need to be created, it allowed for greater reliability of the framework, since a client could at least rely on its own information.

In terms of affecting the mean call time, the two architectures tended to vary. At times, the centralized approach was faster, and vice versa, without a clear indication as to which was fastest. However, in terms of bandwidth, there was a clear distinction between the two. The decentralized approach, storing all interactions locally, needed only to fetch interactions over the network. The centralized approach, however, had to both fetch *and* store the interactions, leading to a very significant difference in terms of the number of bytes sent over a particular run. This is evident in Table 4.8: in all but one case (the recurrent configuration for Client 1) the decentralized architecture sent significantly fewer bytes, thus reducing network traffic compared to its centralized equivalent. The decentralized approach also, due to the local database, allows for quicker storage than the centralized approach.

The major area in which the centralized approach performs better comes in the time needed to retrieve a set number of interactions. In Experiment 3, this time was measured, and as can be seen in Figures 6.3 and 6.4, the centralized architecture consistently allows faster data retrieval.

Because of the lack of major differences in call times, and because the bandwidth usage is in almost all cases significantly less, the decentralized approach is useful in a variety of situations. If one seeks to minimize the time to query and reason, the centralized architecture seems the better choice; but in all other aspects, the decentralized approach seems the better option.

7.5 The Utility of QoS-Based Selection

Using past QoS information is just one approach to the problem of web service selection. It is, however, a useful one. The use of only three generic QoS parameters allowed a great deal of speed-up; the use of other QoS parameters, in particular domain-dependent ones, might allow even greater increases in performance. As shown by the experimentation, a framework using QoS-based selection can achieve much greater QoS than a standard web services client. This increase is partially dependent on the policy configuration of the client. Reactive policies give the greatest increase in mean call time, though it must be pointed out that the experimentation involved at least one excellent service to which the clients could switch.

QoS-based selection is just one possible approach to the problem. Researchers in web services and the semantic web have explored a number of different options: selecting a service based on semantic constraints [7], QoS information reported to a central registry [23], or even as part of the software development lifecycle [17]. Each of these approaches is useful, and could be combined with the QoS approach described in this thesis. As each approach gives new techniques for evaluating services, the service selected for the client becomes less and less of an approximation of the best available service.

7.6 Future Work

This thesis describes a web service selection framework, where decisions are made based on QoS information, and where the QoS information is described using an OWL ontology. One of the discoveries during the experimentation was that OWL can add a significant overhead to the system, in terms of the amount of information spent. Future work could include investigation of other languages for knowledge representation, such as KQML/KIF [44], and the FIPA ACL [45], to see which would be most suitable for use in providing an open and extensible language for QoS information. As well, the Unified Modeling Language (UML [46]) is widely used for specifying how a particular piece of software works. It contains facilities for specifying class relationships – these could potentially be used to represent the QoS ontology. Since the XML Metadata Interface (XMI

[47]) allows for UML models to be transferred between applications, UML could also potentially be used as a knowledge representation language.

The QoS parameters studied were based on an ontology that described a number of generic terms. In the future, it would be worthwhile to investigate the use of domain-specific QoS parameters, to see whether these could improve the performance of the framework.

The reasoning engine has two different implementations: Naive Bayes, and a rule-based expert system. An interesting extension would be to test naive Bayes against other machine learning algorithms. The WEKA toolkit used for the implementation of naive Bayes has interfaces that allow for easily switching the machine learning algorithms.

Currently, the Naive Bayes classifier in the framework gets the true classifications of instances from the framework itself, using the values of a number of QoS parameters to determine an instance's classification. This could be improved in a number of ways. The framework could be extended to allow the system's administrator to specify classifications for both a service, as well as its operations, as certain operations could be known to take longer and generally have different QoS requirements than others. As well, an unsupervised learning algorithm could be used to find clusters, which could then be used as the QoS classifications. Finally, to improve the performance of the framework, an updatable Naive Bayes algorithm could be used whenever the Naive Bayes classifier is selected. Whenever the framework gathers data, it builds a new instance of the reasoner. This is inefficient, and sometimes unnecessary. Using an updatable Naive Bayes algorithm could improve performance by allowing updates rather than complete rebuilds.

While past QoS has been shown to be useful in selecting web services, it is just one possible way to select between services. One future direction could be to integrate the current framework with a number of the other approaches discussed in Chapter 3. There has been an increased amount of interest in languages such as OWL-S. OWL-S allows the semantic specification of a service, which could be very useful towards automating service discovery and selection. Currently, the framework relies on the administrator or user to specify the services from which to select. OWL-S could allow for automated service discovery, freeing the user or administrator from having to update the list of services.

Appendix A

A Basic Web Service QoS Ontology in OWL

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rss="http://purl.org/rss/1.0/"
    xmlns="http://a.com/ontology#"
    xmlns:jms="http://jena.hpl.hp.com/2003/08/jms#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns:vcard="http://www.w3.org/2001/vcard-rdf/3.0#"
    xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xmlns:dc="http://purl.org/dc/elements/1.1/"
  xml:base="http://a.com/ontology">
  <owl:Ontology rdf:about=""/>
  <owl:Class rdf:ID="Service-SpecificQOS">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="QoSHierarchy"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:ID="Call-SpecificQOS">
    <rdfs:subClassOf rdf:resource="#QoSHierarchy"/>
  </owl:Class>
  <owl:DatatypeProperty rdf:ID="Execution_Time">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="QOSReliability">
    <rdfs:domain rdf:resource="#Service-SpecificQOS"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="ExecutionTime">
    <rdfs:domain rdf:resource="#Call-SpecificQOS"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#float"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="Reliability">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
```

```
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="Availability">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
    <rdfs:domain rdf:resource="#Service-SpecificQOS"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="Accessibility">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#boolean"/>
  </owl:DatatypeProperty>
  <owl:FunctionalProperty rdf:ID="Location">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#DatatypeProperty"/>
  </owl:FunctionalProperty>
</rdf:RDF>
```

Appendix B

System Overhead Data

B.1 CPU Process Information

The per-process CPU data allows the examination of the time the process spends using the CPU. While looking at CPU usage at a whole is useful, there can often be a number of complications: other processes might be using the CPU intensively, so increases in CPU usage are not necessarily attributable to the web services framework.

The data measured in this section allows the reader to see the percentage of elapsed time that the threads in the process use the processor to execute instructions. Included are both the unaugmented client (Figure B.1) and the client with the web service selection framework, in different configurations.

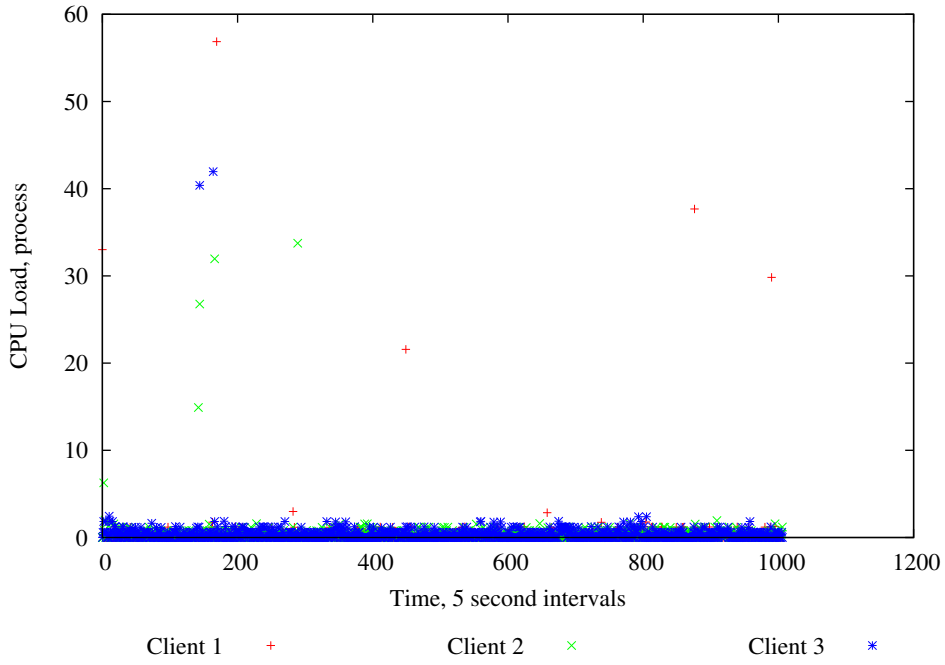

Figure B.1: Process CPU Usage, No Augmentations

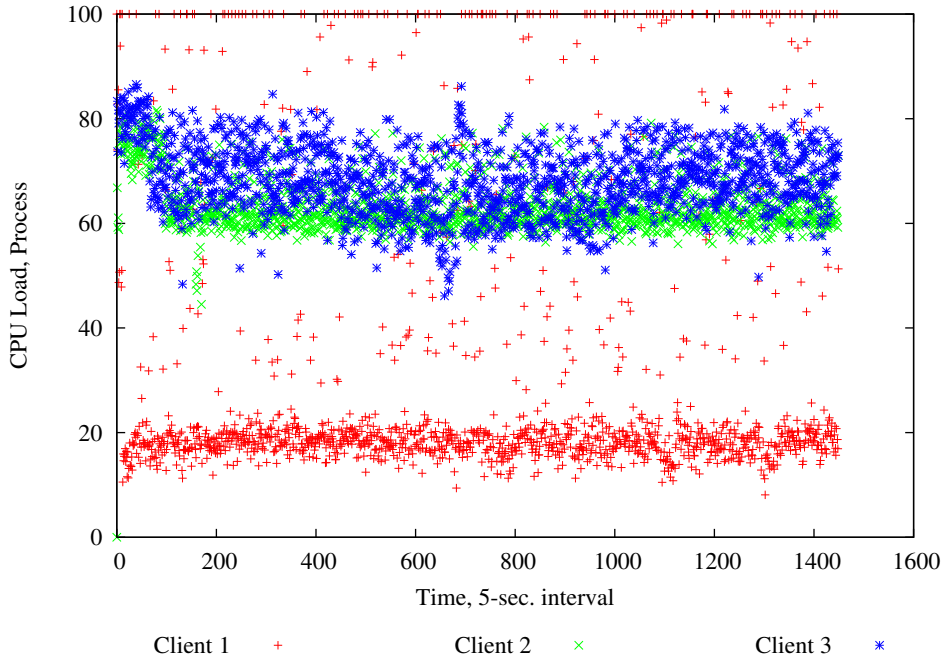

Figure B.2: Process CPU Usage, Augmented: Conservative Policy, Centralized

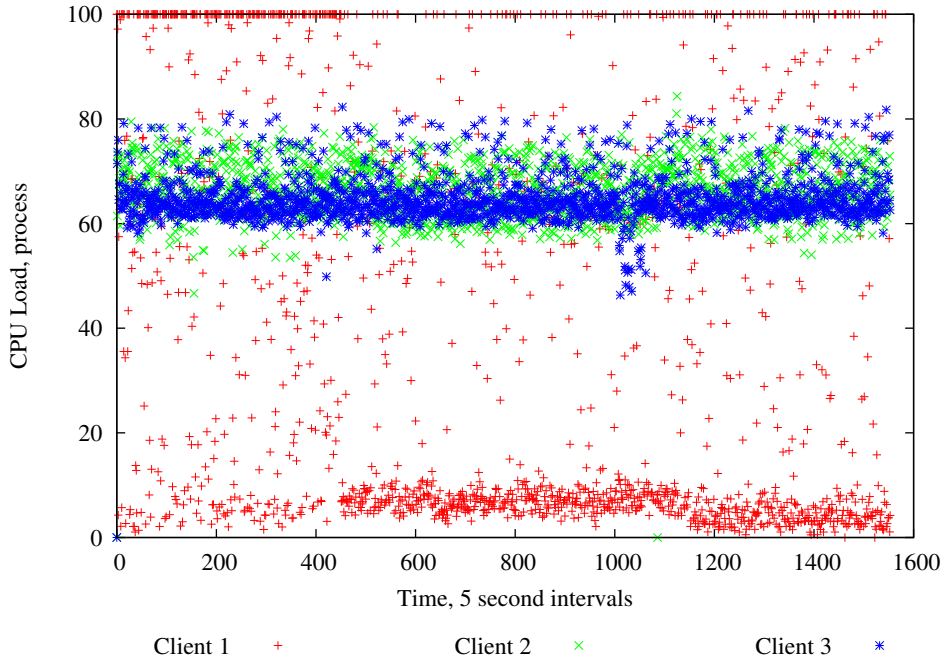

Figure B.3: Process CPU Usage, Augmented: Conservative Policy, Decentralized

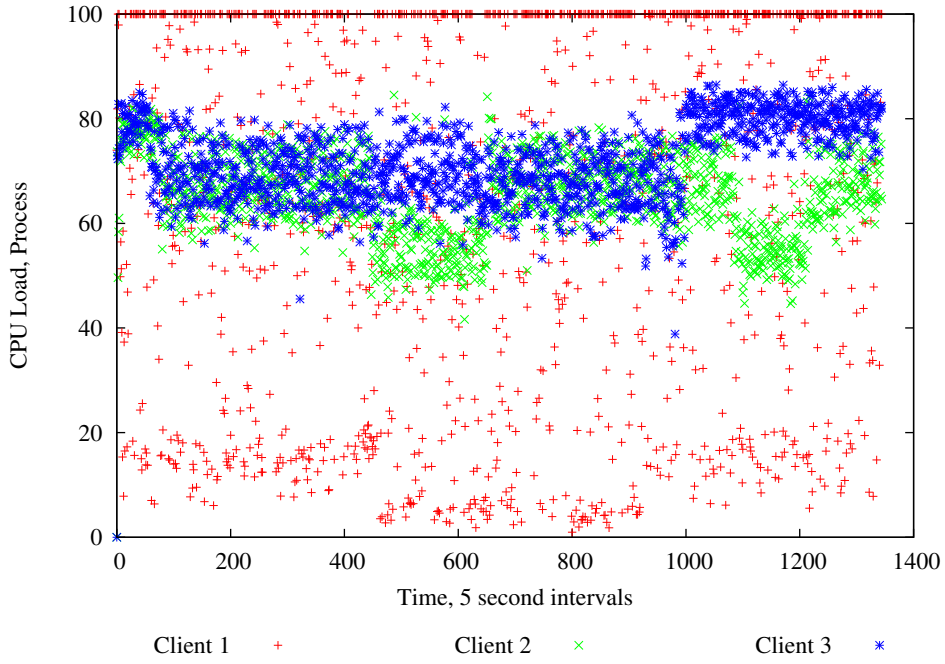

Figure B.4: Process CPU Usage, Augmented: Reactive Policy, Centralized

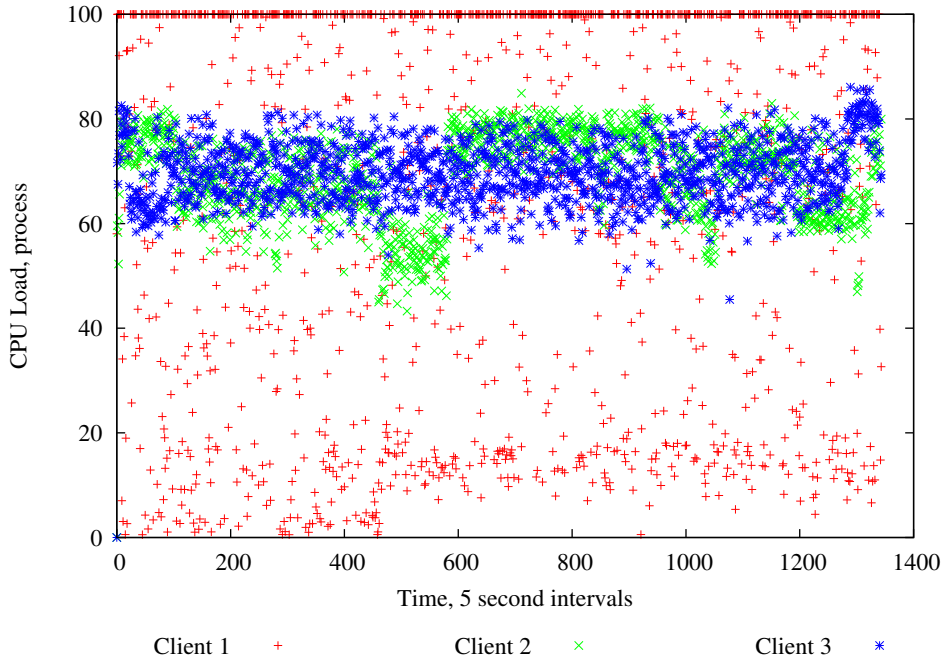

Figure B.5: Process CPU Usage, Augmented: Reactive Policy, Decentralized

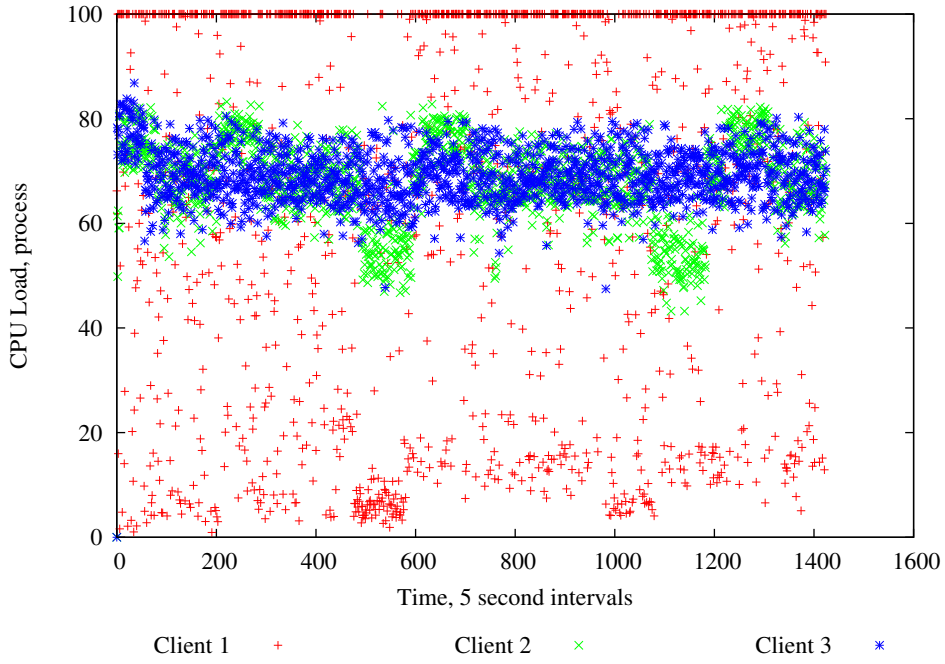

Figure B.6: Process CPU Usage, Augmented: Recurrent Policy, Centralized

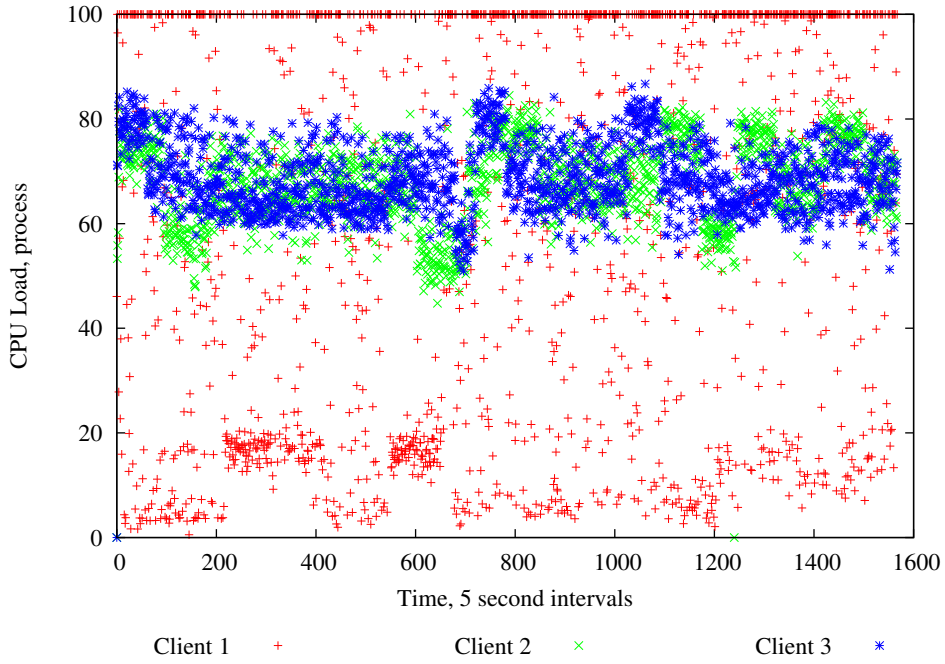

Figure B.7: Process CPU Usage, Augmented: Recurrent Policy, Decentralized

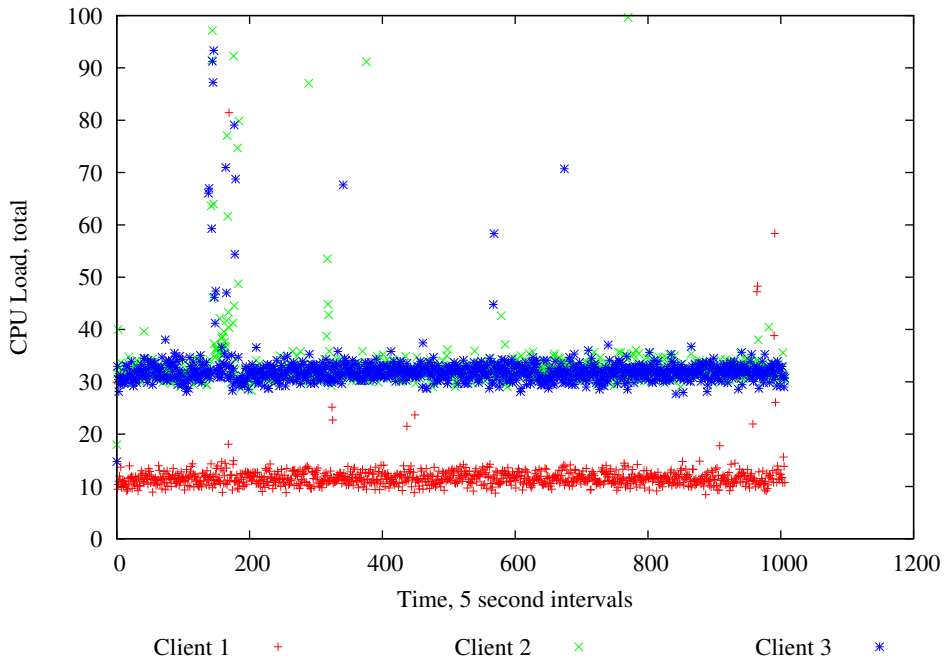

Figure B.8: Total CPU Data, No Augmentations

B.2 Total CPU Usage

This section shows measurements of the overall CPU usage of the clients. Because the clients are diverse in their hardware configuration, it can be seen that the overhead varies considerably.

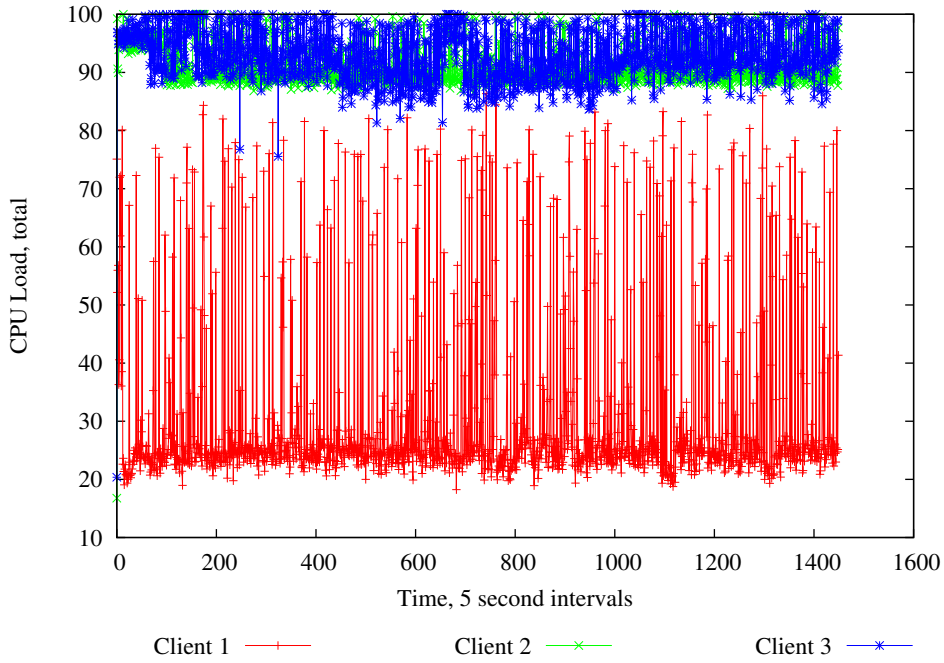

Figure B.9: Total CPU Data, Augmented: Conservative Policy, Centralized

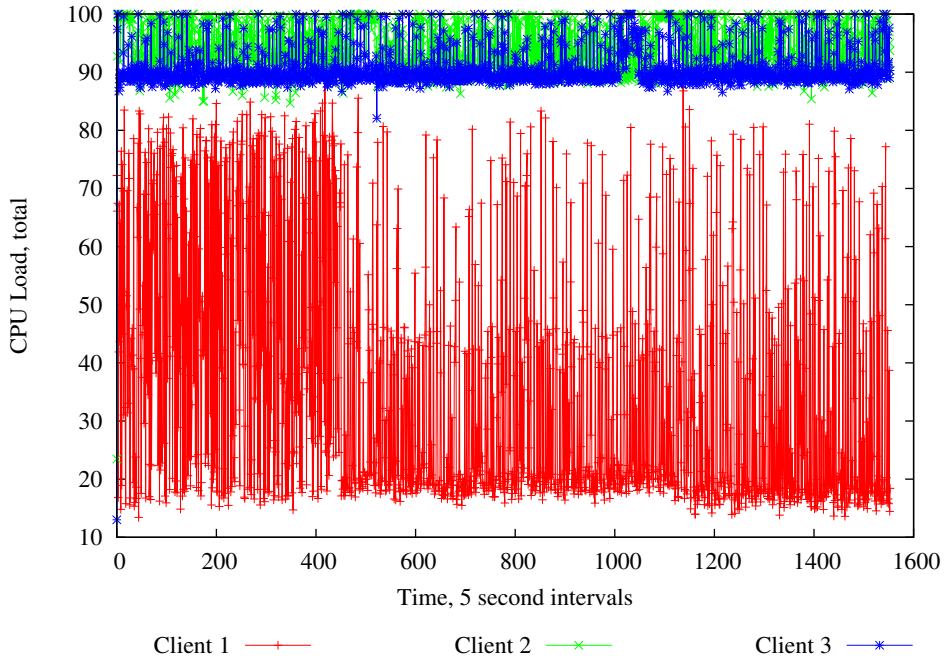

Figure B.10: Total CPU Data, Augmented: Conservative Policy, Decentralized

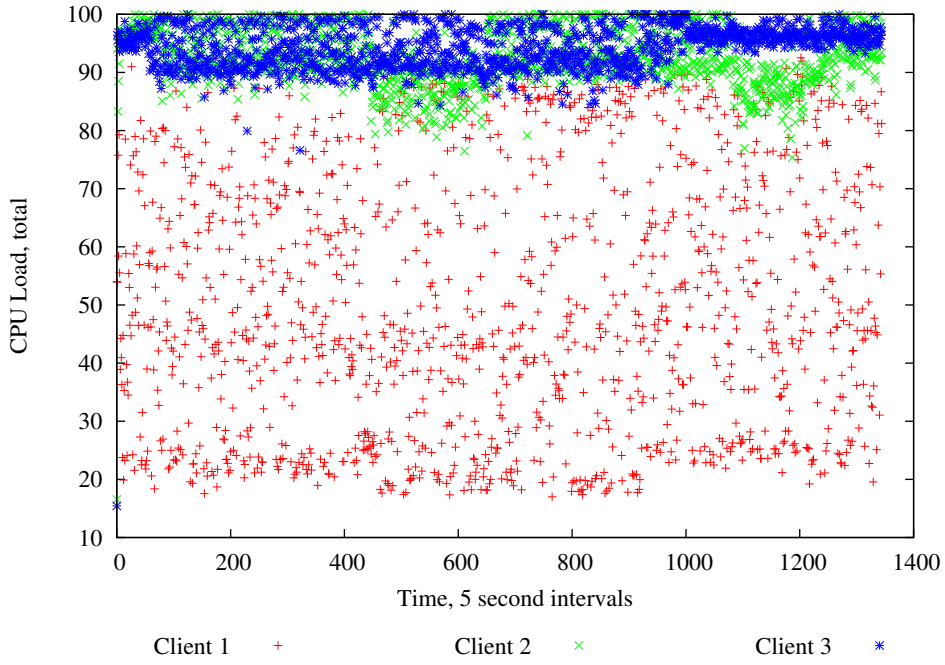

Figure B.11: Total CPU Data, Augmented: Reactive Policy, Centralized

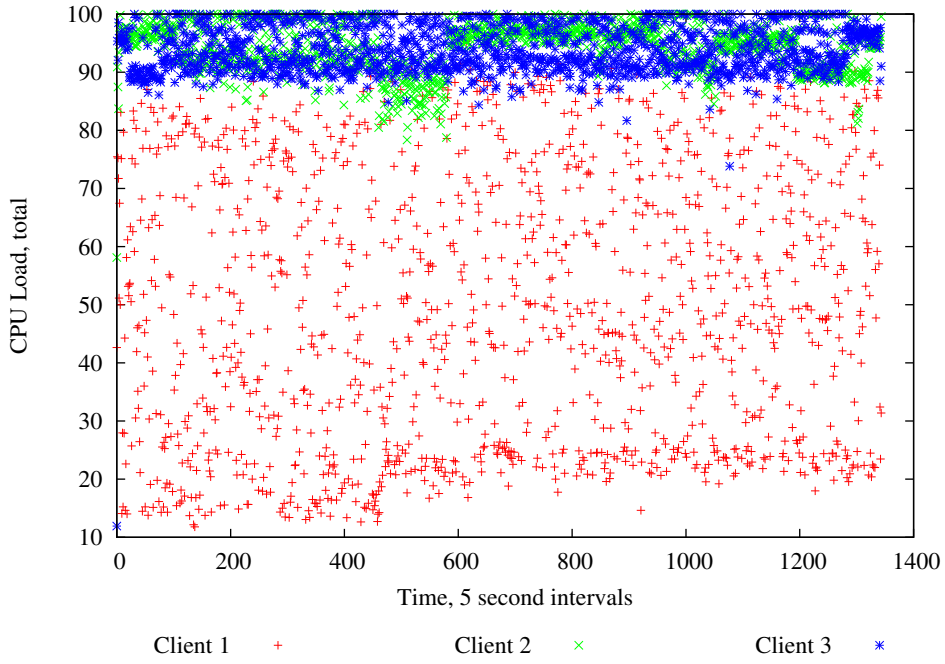

Figure B.12: Total CPU Data, Augmented: Reactive Policy, Decentralized

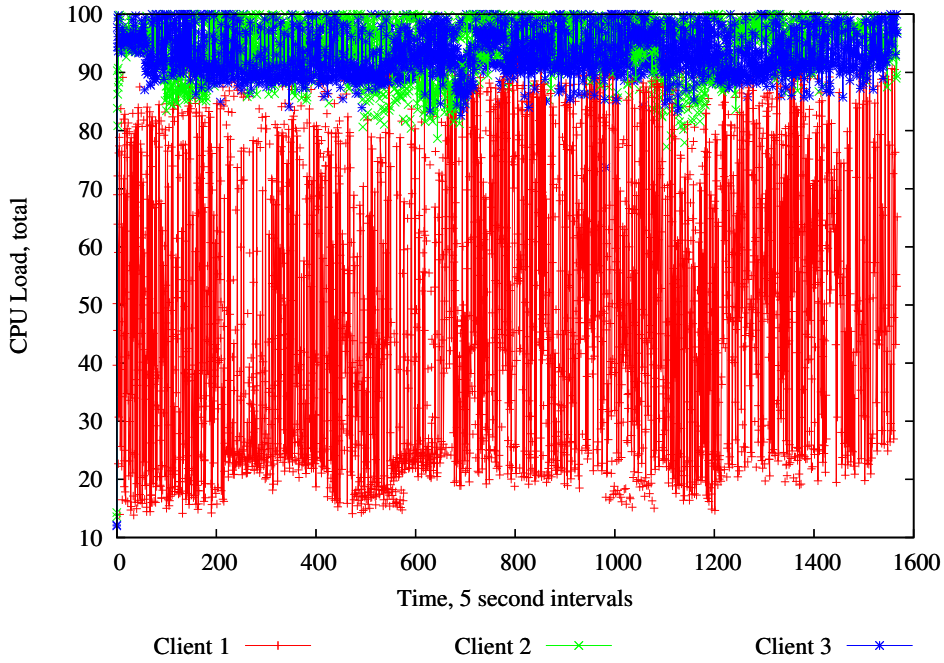

Figure B.13: Total CPU Data, Augmented: Recurrent Policy, Centralized

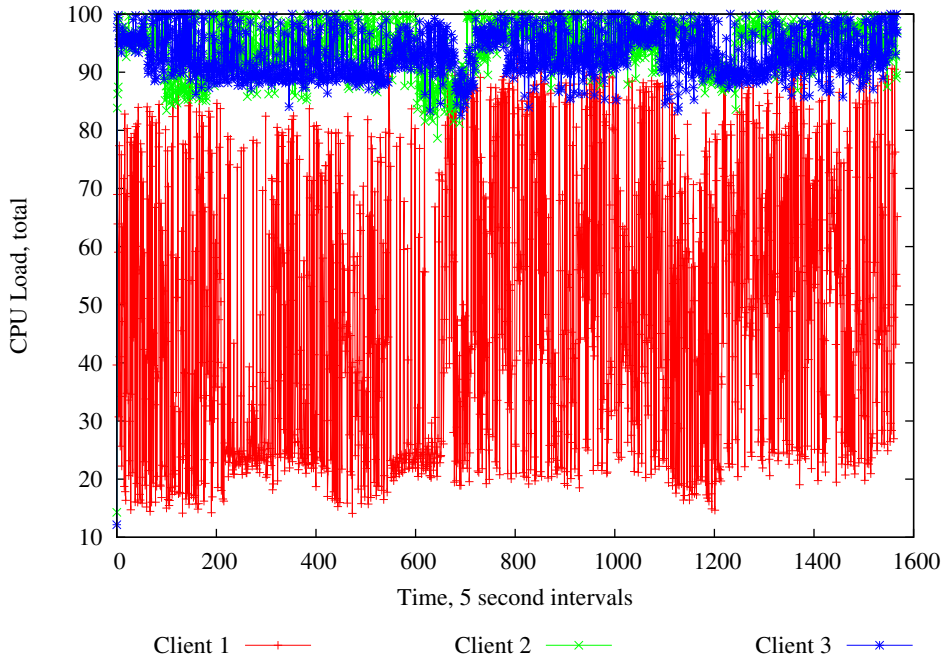

Figure B.14: Total CPU Data, Augmented: Recurrent Policy, Decentralized

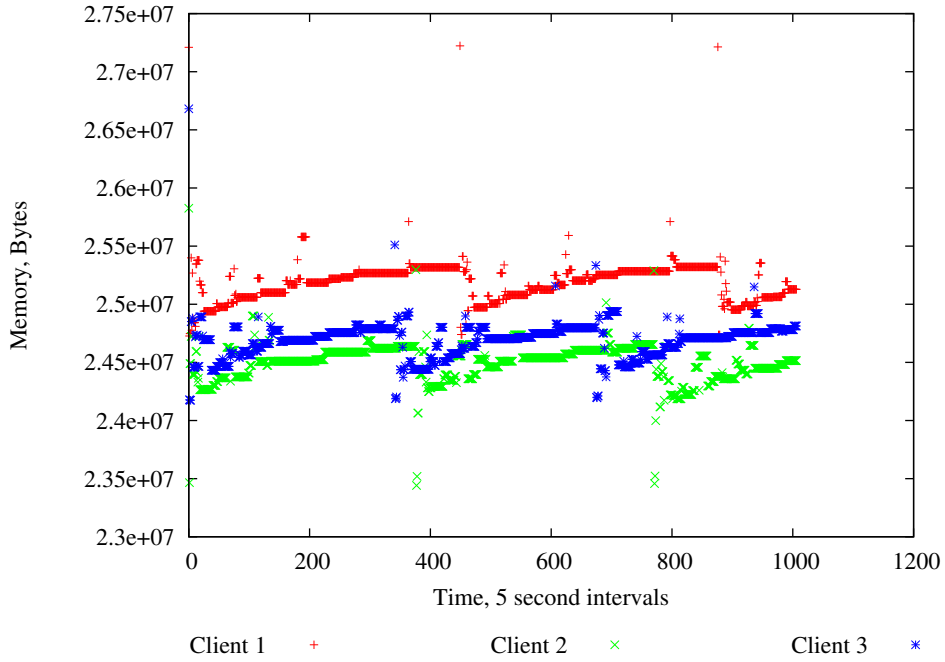

Figure B.15: Total Bytes, No Augmentations

B.3 Bytes

Another useful indication of overhead is the amount of bytes used by the process. As one indication of the overhead of the framework, one can compare the overhead of the different configurations of the augmented client to that of the basic, unaugmented client.

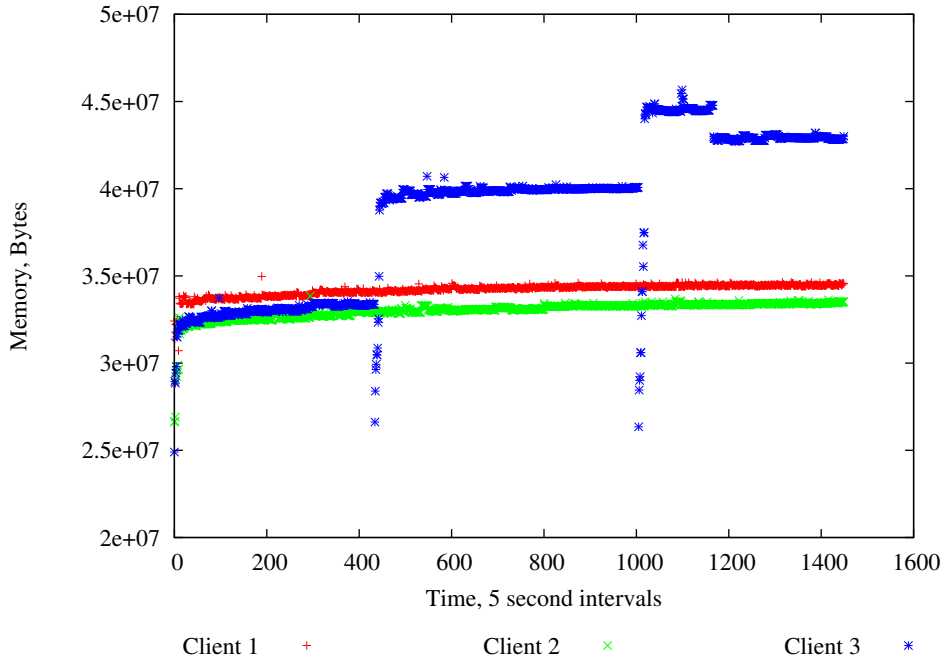

Figure B.16: Total Bytes, Augmented: Conservative Policy, Centralized

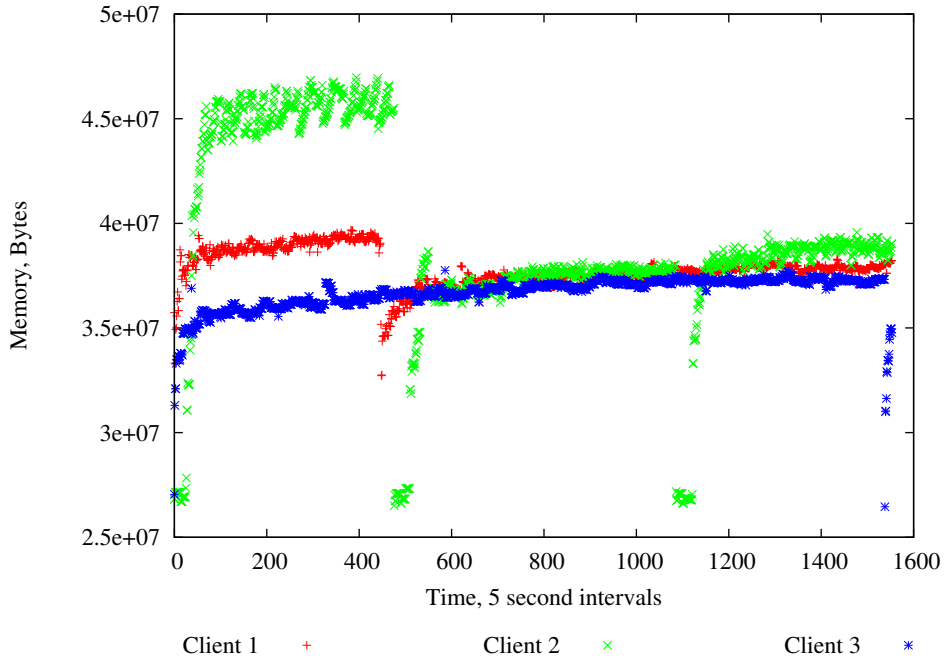

Figure B.17: Total Bytes, Augmented: Conservative Policy, Decentralized

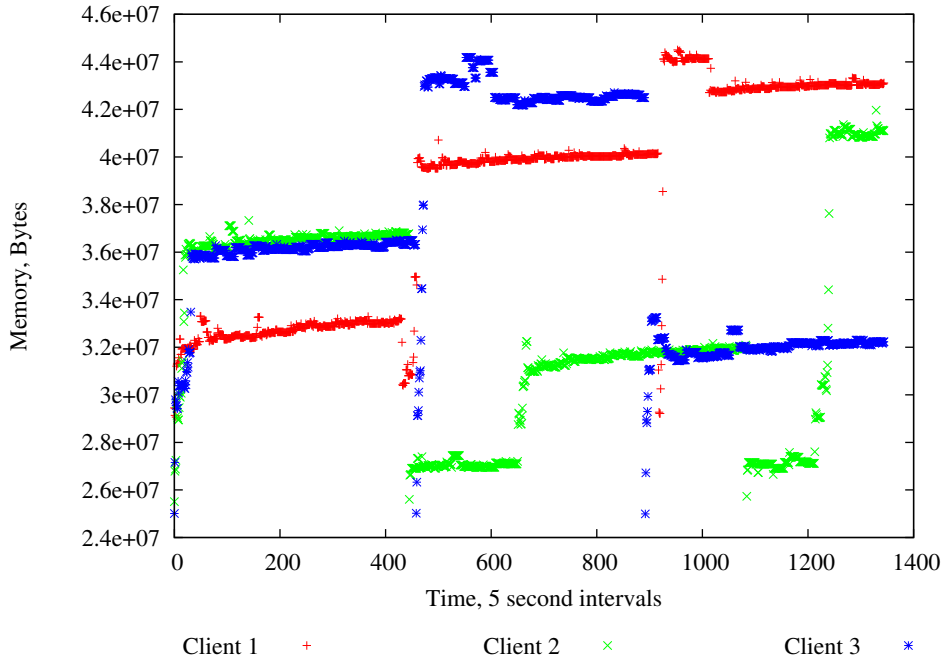

Figure B.18: Total Bytes, Augmented: Reactive Policy, Centralized

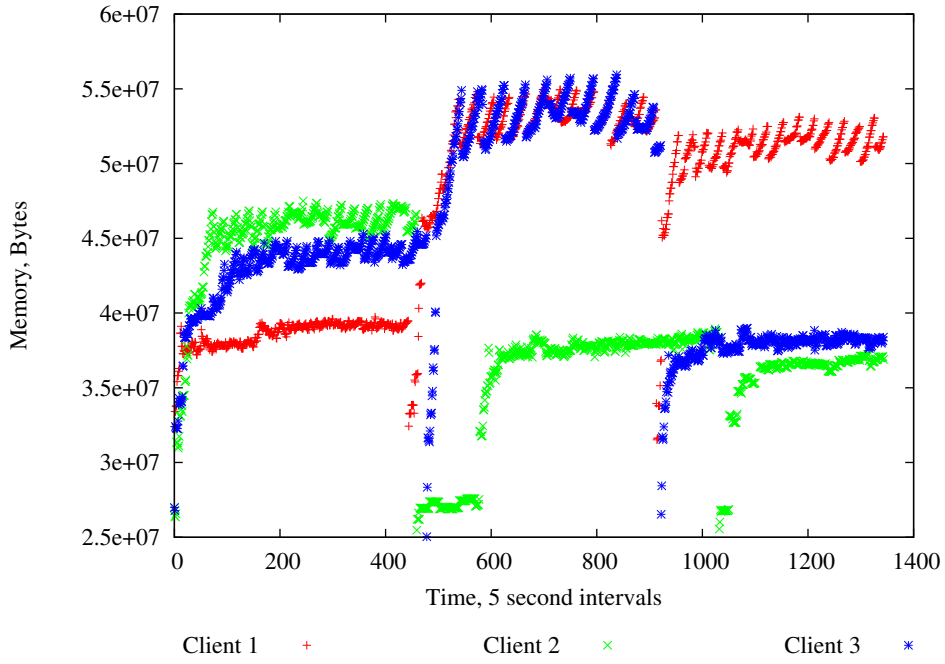

Figure B.19: Total Bytes, Augmented: Reactive Policy, Decentralized

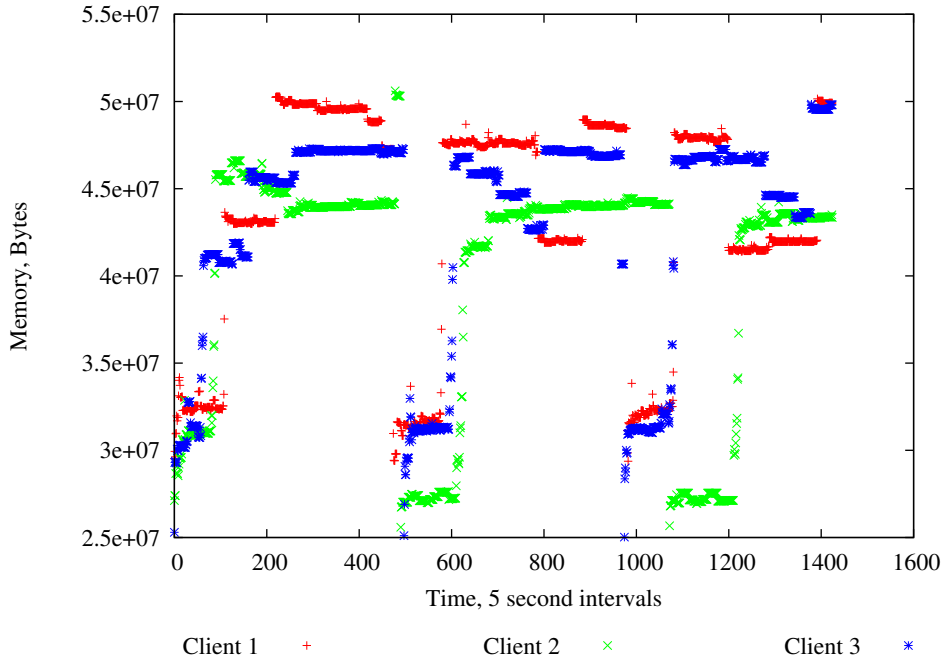

Figure B.20: Total Bytes, Augmented: Recurrent Policy, Centralized

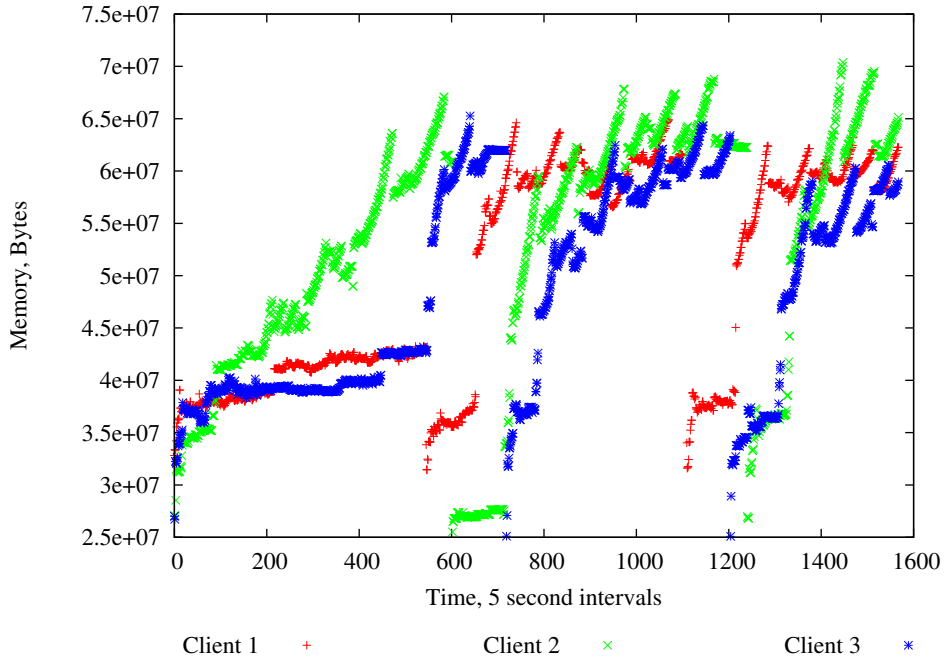

Figure B.21: Total Bytes, Augmented: Recurrent Policy, Decentralized

Appendix C

Complete Policy Configurations

C.1 Conservative Policy Configuration

```
policy querying
        one-shot
end

policy reasoning
        after querying
end

policy switching
        after reasoning
end

policy reporting
        after failure
        after success
        end
```

C.2 Reactive Policy Configuration

```
policy querying
        after failure
end

policy reasoning
        after querying
end
```

```
policy switching
        after reasoning
end

policy reporting
        after failure
        after success
end
```

C.3 Recurrent Policy Configuration

```
policy querying
        every 10 minutes
end

policy reasoning
        after querying
end

policy switching
        after reasoning
end

policy reporting
        after failure
        after success
end
```

References

[1] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[2] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, and Francois Yergeau. Extensible markup language (xml) 1.0 (third edition). http://www.w3.org/Protocols/rfc2616/rfc2616.html.

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

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

[5] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. Web services description language (wsdl) 1.1. http://www.w3.org/TR/wsdl.

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

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

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

[9] Julian Day and Ralph Deters. Selecting the best web service. In *Proceedings of the IBM Centers for Advanced Study Conference (CASCON '04)*, pages 293–307, 2004.

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

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

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

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

[14] Hector J. Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31:59 – 84, 1997.

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

[16] Sheila McIlraith and Tran Cao Son. Adapting golog for programming in the semantic web. In *Fifth International Symposium on Logical Formalizations of Commonsense Reasoning*, pages 195 – 202, 2001.

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

[18] Activity Lead Hugo Haas. Web services activity statement. http://www.w3.org/2002/ws/Activity.

[19] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

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

[21] 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.

[22] E. Maximilien and M. Singh. Conceptual model of web service reputation, 2002.

[23] 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.

[24] Wolf-Tilo Balke and Matthias Wagner. Towards personalized selection of web services. In *Proceedings of the 12th International World Wide Web Conference (WWW 2003)*, Budapest, Hungary, 2003. ACM.

[25] Peter Jackson. *Introduction to Expert Systems, Second Edition*. Addison-Wesley Publishing Company, 1990.

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

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

[28] 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.

[29] David Hand, Heikki Mannila, and Padhraic Smyth. *Principles of Data Mining*. The MIT Press, 2001.

[30] M. Mitchell Waldrop. Autonomic computing: The technology of self-management. Foresight and Governance Project, Woodrow Wilson International Center for Scholars, 2003.

[31] Emre Kiciman and Yi-Min Wang. Discovering correctness constraints for self-management of system configuration. In *Proceedings of the International Conference on Autonomic Computing*, pages 28 – 35. 2004.

[32] Mike Chen, Alice X. Zheng, Jim Lloyd, Michael I. Jordan, and Eric Brewer. Failure diagnosis using decision trees. In *Proceedings of the International Conference on Autonomic Computing*, pages 36 – 43. 2004.

[33] Gary Dudley, Neeraj Joshi, David M. Ogle, Balan Subramanian, and Brad B. Topol. Autonomic self-healing systems in a cross-product it environment. In *Proceedings of the International Conference on Autonomic Computing*, pages 312 – 313. 2004.

[34] J. O. Kephart and D. M. Chess. The vision of autonomic computing. In *Computer*, pages 41 – 52, 2003.

[35] Jeffrey O. Kephart and William E. Walsh. An artifical intelligence perspective on autonomic computing policies. In *Fifth IEEE International Workshop on Policies for Distributed Systems and Networks*, pages 3 – 12, 2004.

[36] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 4, pages 382–401, New York, NY, USA, 1982. ACM Press.

[37] Dennis D. Wackerly, William Mendenhall III, and Richard L. Scheaffer. *Mathematical Statistics with Applications*. Duxbury Press, 1996.

[38] W.H. Teichner. Recent studies of simple reaction time. *Psychological Bulletin*, 51:128–149, 1954.

[39] J.T. Brebner and A.T. Welford. Introduction: An historical background sketch. In A.T. Welford, editor, *Reaction Times*, pages 309–320. Academic Press, New York, 1980.

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

[41] Barbara Di Eugenio and Michael Glass. The kappa statistic: A second look. *Computational Linguistics*, 30(1):95–101, 2004.

[42] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46, 1960.

[43] Sidney Siegel and Jr. N. John Castellan. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill, Boston.

[44] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an Agent Communication Language. In N. Adam, B. Bhargava, and Y. Yesha, editors, *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM'94)*, pages 456–463, Gaithersburg, MD, USA, 1994. ACM Press.

[45] Foundation for Intelligent Physical Agents. Fipa communicative act library specification, 2001.

[46] Object Management Group. Unified modeling language 1.5. http://www.omg.org/technology/documents/formal/uml.htm.

[47] Object Management Group. Corba, xml, and xmi resource page. http://www.omg.org/technology/xml/.