Toward Autonomic Web Services Trust and Selection

E. Michael Maximilien[*]
IBM and NCSU
5506 Six Forks Road
Raleigh, NC 27609
maxim@us.ibm.com

Munindar P. Singh
North Carolina State University
Department of Computer Science
Raleigh, NC 27695
singh@ncsu.edu

ABSTRACT

Emerging Web services standards enable the development of large-scale applications in open environments. In particular, they enable services to be dynamically bound. However, current techniques fail to address the critical problem of selecting the right service instances. Service selection should be determined based on user preferences and business policies, and consider the trustworthiness of service instances.

We propose a multiagent approach that naturally provides a solution to the selection problem. This approach is based on an architecture and programming model in which agents represent applications and services. The agents support considerations of semantics and quality of service (QoS). They interact and share information, in essence creating an ecosystem of collaborative service providers and consumers. Consequently, our approach enables applications to be dynamically configured at runtime in a manner that continually adapts to the preferences of the participants. Our agents are designed using decision theory and use ontologies. We evaluate our approach through simulation experiments.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multiagent systems*; D.1.0 [**Software Engineering**]: Programming Techniques—*general*; D.2.8 [**Software Engineering**]: Metrics—*process metrics, performance measures*

General Terms

Algorithms, Reliability, Experimentation

Keywords

Service selection, Service binding, Quality of Service (QoS), Software Agents, Autonomic Computing, Trust

[*]IBM Software Architect and NCSU Ph.D. candidate.

1. INTRODUCTION

The Web services architecture, standards, and technologies support describing, finding, and binding to services. The overarching vision is that services would be dynamically created and administered, and would be incorporated into software systems in execution without the need for frequent human intervention. Although the current Web services standards and technologies are necessary to realize this vision, they are far from sufficient.

In simple terms, between finding and binding lies another crucial step, which the current approaches ignore. This is the step of *selection* wherein a specific service instance is chosen by a prospective consumer. Conceptually, service selection is difficult because it faces the main challenge of an open environment: you cannot easily predict the quality of service (QoS) that a given service instance will deliver. The challenge arises partly because you may not be able to trust the other party, and partly because you lack knowledge of the environment within which it is executing.

To explain our approach, we distinguish three phases of interactions between consumers and services.

Service discovery. When a consumer finds a desired service interface. This search is typically based on common service repositories such as Universal Description Discovery and Integration (UDDI).

Service selection. When a consumer selects a service instance (implementing a discovered interface). Selection is based on nonfunctional attributes such as QoS and trust. A service instance may be replaced by another at runtime if it doesn't meet the customer's needs, becomes untrustworthy, or a better instance is found.

Service binding. When a consumer begins using a selected instance. Binding typically occurs at time of first need.

Our contribution is a comprehensive agent-based trust framework for service selection in open environments. We introduce a policy language to capture service consumer's and provider's profiles, algorithms to select services based on those policies, and representations to dynamically capture data about service performance with respect to various (customizable) QoS dimensions. As a result, service-based applications are dynamically configured at runtime to choose the "best" services with respect to each participant's preferences. We demonstrate the effectiveness of this approach in selecting good services via simulation experiments.

1.1 Organization

The remainder of this paper is organized as follows. Section 2 motivates the dynamic service selection and binding problems. Section 3 gives a general overview of our framework, including the QoS ontology, QoS policy, and matching algorithm. Section 4 presents empirical results showing the emergence of trust and other autonomic characteristics. Section 5 compares related literature on QoS in software engineering and Web services to our approach. Section 6 discusses key directions for future work.

2. MOTIVATIONS

We address service selection and subsequent dynamic binding via an open multiagent system that facilitates these for service consumers.

2.1 Scenarios: Use Cases

Consider the following two use cases to motivate our solution.

1. B2C search service. A user is searching the Web for goods or services. The application employs Web search engines services to execute the user's query and chooses one of them based on the preferences of the user along with the search words. The search tool uses a service agent for each search service and at runtime based on the input the agent picks the "best" search service to execute the query.

2. B2B loan service. A car dealership provides financing to its potential buyers. Because the dealership wants to offer various financial options, it does not have any fixed relationship with any banks. Instead, it selects loan services based on the reputations of the services aggregated from previous user episodes. Briefly, the dealership financing software uses a service agent, which it instantiates for each user episode. Using the business policies and user preferences, the agent selects the most appropriate services and presents the top financial alternatives to the user.

A comprehensive scenario can be found in [15].

2.2 QoS Motivated

Representationally, the key idea behind our work is to adorn service descriptions with QoS annotations. These augmented descriptions, however, would not be produced by the service providers but would, in essence, be developed by the service consumers. Instead of the service consumers working individually, which would limit their effectiveness in judging the service instances with which they have directly interacted, it is helpful for them to share knowledge about the service instances. However, QoS knowledge cannot be easily shared because it is based on the judgments of the various participants, which can be subjective. A simple, practical way to reduce the effect of the potential arbitrariness of individual consumers, is to aggregate the consumers judgments into general opinions. The twin design goals of sharing and aggregation can be satisfied through the mechanism of an *agency*.

Agency. A rendezvous node in the system where agents are able to share information about services.

To facilitate representing QoS knowledge, we formulate an ontology for QoS that includes a categorization of the various attributes that may apply to a given service.

2.3 Trust and Trustworthiness

As discussed in Section 1, the selection step of the consumer involves making a rational selection decision between many providers and service instances. If a service provider is already determined to be *trustworthy*, then the selection step becomes one of selecting the services from providers of the highest level of *trust*. Naturally, determining the level of trust to assign to a provider is nontrivial, especially in open environments. We are specifically talking about environments where service consumers and providers are autonomous, that is, free to behave as they wish; and where there are no central trusted authorities that can act as a brokers of trust.

Our proposed framework provides a solution to this trust problem by having agents collect and share information on their interactions with the services that they selected on behalf of service consumers. The shared information provides a basis for establishing trust.

Service quality reputation. The aggregation of collected service quality data for a given quality.

Since the quality of service will change over time, it is also necessary that the quality reputation calculation take these changes into account. Following Zacharia and Maes [26] we dampen the quality reputation so that recent data matters more in determining reputation. We resolve the damping factors for a quality from the shared QoS ontology.

Combining this calculated quality reputation with the consumer's QoS policy and a provider's QoS advertisement, our service agent is able to intelligently assign a trust level to the current set of available services.

2.4 Autonomic Systems

An important characteristic for trust in open environments is that it should be *self-adjusting*. That is, service providers who behave badly are purged from the system by virtue of not being selected. Poor service providers accumulate a low reputation. Conversely, when a bad service starts to behave correctly, we would like the agents to increasingly consider the service. This dynamic and self-adjusting consideration of trust for selection falls under the goals of autonomic computing [11].

Self-adjusting trust. The autonomic characteristic of a multiagent system whereby the levels of trust between the interacting parties are dynamically established and adjusted to reflect recent interactions.

Handling service selection and binding in open environments, presupposes self-adjusting trust.

3. FRAMEWORK

Our solution involves attaching a software agent, autonomously-configured software component, to each Web service. This agent proxies the service for its consumer, exposing the same interface as the service but additionally enabling other useful functionality. Instead of communicating directly with the service, the service consumers now communicate with the agent. In this manner the agent can add value to the consumer-service interaction.

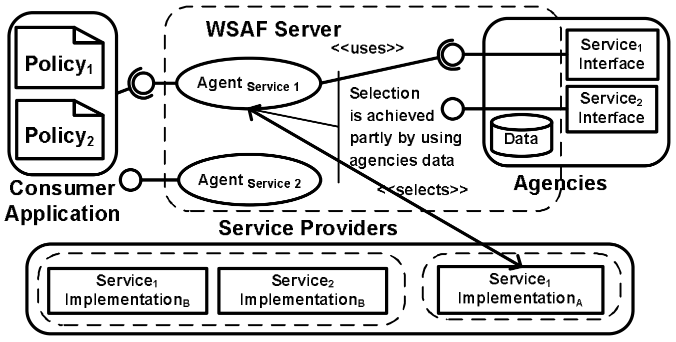

Figure 1: Architecture overview.

3.1 Architecture

Our architecture uses service agents that conceptually reside between the consumer of a service and the service itself. The consumer makes use of one agent per Web service interface, hence the moniker: *Web service agent*. The architecture essentially expands on the classic service-oriented architecture (SOA) diagram by augmenting the service consumer's role with software agents that automate some of that role's tasks [6]. We extend the service broker role with agencies that facilitate the sharing of data among the participating agents.

Figure 1 gives an overview of the architecture showing the static components and their relationships. In our current implementation, the agents and agencies reside in a server that the consumers know about *a priori*. However, this topology can be changed to co-locate the consumers and agents. By having the agents on a server, the computational burden of the agent is decoupled from the consumer and also, importantly, the agent can expose a Web service interface to the client and thus allow cross-platform consumer-to-agent interactions. For instance, the consumer can be a .NET or Python application while the agent is implemented in Java. To better understand the overall architecture and responsibilities of the various components, we discuss each of the four main components that constitute the Web Service Agent Framework (WSAF).

1. *Service providers.* We assume that providers describe their services using the WSDL standard. WSDL contains both the interface description for the service and the binding information for the implementation. In Figure 1 we show two service providers A and B implementing service $Interface_1$ and $Interface_2$. Notice that $Interface_1$ is implemented by both providers.

2. *Service brokers.* The SOA architecture calls for broker registries where providers register their services along with classification attribute data to enable service consumers to find services. We extend the registry concept with the notion of agency where our service agents can collaborate and share data.

3. *WSAF server.* The WSAF server is where the service agents reside and are managed. QoS ontologies and configuration files are also located here.

4. *Consumer Applications.* These are the consumers of services using the service agents. Typical applications contain various business objects along with proxy objects that act as local surrogates for the services being consumed via the service agents.

3.2 QoS Ontology

Our framework takes a different approach to QoS specification and monitoring than explicit semi-contractual documents which are used in current QoS specifications like Web service-level agreement (WSLA). For one, our QoS specification is an ontology that allows us to match services semantically and dynamically. The semantic matching allows the service agent to match consumers to services using the provider's advertised QoS policy for the services and the consumers' QoS preferences. The provider policy and consumer preferences are expressed using the concepts in the ontology. Further, by using the same ontology, the service agent is able to automatically configure itself with the correct behaviors (attached to the ontology) so that interactions between the consumer and the service are monitored and recorded in the appropriate agencies.

Figure 2 is an overview of the key aspects of our QoS upper ontology. Some of its main concepts are:

- *Quality.* Represents a measurable nonfunctional aspect of the service within a certain domain. Quality instances have attributes, are measured by agents, and have relationships with each other.

- *QAttribute.* The value of a Quality concept is determined by the type of QAttributes that constitute it. For instance, a Quality that has a monotonic float attribute will necessarily have values that are float and which are monotonic; that is, where larger values reflect improving qualities.

- *QMeasurement.* How a Quality is measured. This measurement can be objective or subjective. Objective measurements are made automatically via a software agent, whereas subjective measurements are via some human agent. A measurement has a validity period and can be certified.

- *QRelationship.* Qualities are typically related in a manner that shows through their values. These relationships are manifestations of the tradeoffs that providers make in their service implementations. We distinguish the following relationships:
  - *Independent.* The qualities are completely independent of each other. In other words, a change in one quality value has no effect on the other.
  - *Related.* We distinguish the following:
    - *ValueImpact.* The values affect each other. The impact is measured as a strength which we divide into: *Weak*, *Mild* and *Strong*.
    - *ValueDirection.* The values of the related attributes exhibit a general directional relationship such as: *Opposite* and *Parallel*.

- *AggregateQuality.* A Quality that is composed from other qualities. For instance, the PricePerformance ratio combines Price and Performance.

Further details on our QoS ontology, including the middle ontology, can be found in [15].

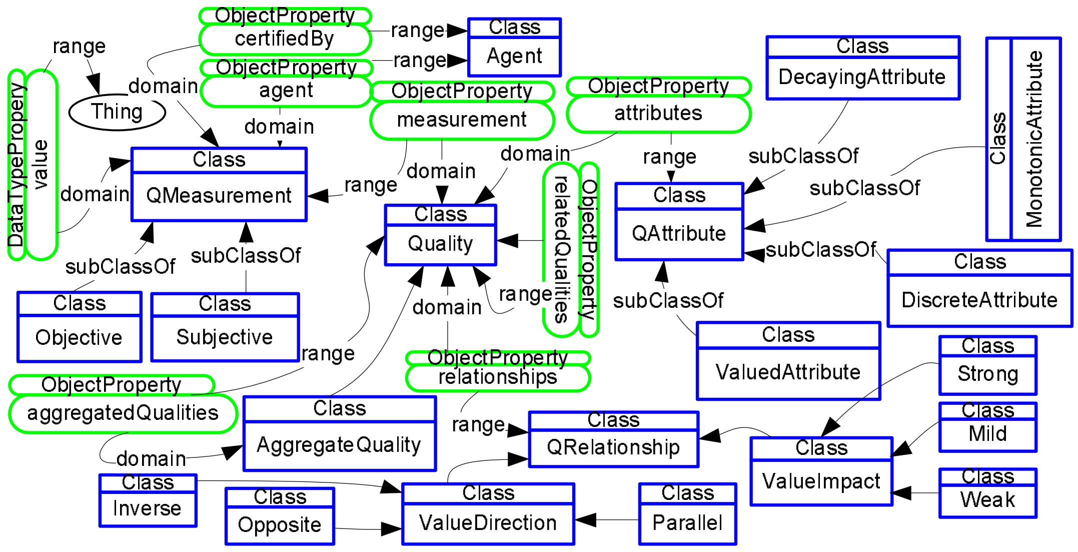

Figure 2: QoS upper ontology.

3.3 QoS Policy

The consumers of a particular service have a wide range of quality expectations for the services they use. For instance, a stock quote service available for quick viewing on a financial page does not have to have the same response time or availability needs as in a brokerage application.

Similarly, service providers have different policies for each service implementation that they publish. The binding of consumers to providers needs to take into account these policies. Since we have a shared conceptual notion of qualities in the QoS ontology, the policy language needs to use the ontology concept as its atoms.

3.3.1 *Provider Policies*

Consumer and provider policies are defined in XML with a specified XSD schema. Listing 1 shows a simplified example of the provider policy.

WsPolicy The root element. It has a required *name* attribute which is used to uniquely identify it. The required *type* attribute must equal *provider* indicating that what follows is a provider policy.

Services A sequence of Service elements containing the details for each service that this policy applies to.

Ontologies This element, starting in line 7, is a collection of Ontology elements each referring to an ontology.

QoSPolicy Captures provider's advertised QoS policy for a service or set of services. For each quality specified, the *promise* attribute indicates the level of commitment of the provider to the advertised policy. This promise can be one of: *bestEffort*, *guaranteed*, *notSpecified*, or *noGuarantee*.

qValue This element gives the policy details for the values of a quality. For instance, line 15 specifies the value details for the ResponseTime quality. They are: *typical*, *min*, *max*, and *unit* for the specified quality.

Listing 1: Provider policy example

```
   <WsPolicy ... name='Provider1' type='provider'>
2    <Services>
       <Service name='Service1'
4               interface='http://.../s1?wsdl'/>
       <Service name='Service2' interface='...'/>
6    </Services>
     <Ontologies>
8      <Ontology name='QoSOnt'
                 uri='http://.../owl/qos.owl'/>
10   </Ontologies>
     <QoSPolicy ontology='QoSOnt' methods='*'
12              services='Service1, Service2'>
       <QoS name='#ResponseTime'
14          promise='bestEffort'>
       <qValue>
16       <typical>60</typical>
         <min>50</min>
18       <max>100</max>
         <unit>ms</unit>
20     </qValue>
     </QoS></QoSPolicy>
22 </WsPolicy>
```

Similar to a provider advertised policy, the customer preference policy specifies a set of services and ontologies and a set of QoS policies. However, some aspects of the specification differs. Specifically, there is a *preferred* element to the qValue element indicating the favored value for the quality in question. Additionally, the consumer policy contains a BindingPolicy element, indicating the consumer's required service binding policy.

BindingPolicy Like the QoSPolicy element, it contains the *ontology* which refers to the ontology used. The *services* attribute is a comma separated list of services names. Additionally it contains is a sequence of Bind elements.

Bind This element specifies the binding condition for a specific time in the service life cycle. The *when* attribute can have the values:

- *onConnect*. When the agent first connect to a service.
- *onFailure*. A failure occurs invoking a service.

- *onQoSValueViolation*. A monitored QoS value is observed to be outside the range specified in the policy.
- *onRebind*. The agent is performing a rebind to potentially a new service implementation.

Different kinds of bindings can be specified. This is indicated with the *type* attribute whose possible values are: *bestMatch*, *firstMatch*, or *anyMatch*.

Constraints Constraints can be specified for each Bind element. For instance, a retry constraint can be specified when a failure occurs.

3.4 Matching Algorithm

We now give an overview of the matching algorithm used to match consumer policies to advertised provider service policies. The algorithm is divided into Listings 2, 3, and 4 and specified in Python.

Listing 2 shows the high level part of the algorithm. Two methods serve as entry points to the algorithm:

findBestService This essentially returns the "best" matching service found. It returns the first service in the sorted list returned by *findBestServices*. If no matching service is found then it throws an exception (line 5).

findBestServices The first step in finding all of the matching services is to only consider services matching the *interface*. This is done in line 10 with a call to the *iMatch* method (starting in line 22). Next it performs a policy match on the list returned in line 11. Next it reduces the returned list with a semantic match in line 12. If the resulting list is not empty it is sorted in line 15 which is then subsequently returned. However, if the policy match is empty, it performs a semantic match on the services matching the interface in line 17 and then performs a sort.

The sorting of services (line 30) is done according to the *dMatch* attribute of a *service* object (set in the code of the matching algorithms).

Listing 2: Matching algorithm.

```
   #@return the 'best' matching service
 2 def findBestService(intf, policy, services):
     matches = findBestServices(intf, policy,
         services)
 4   if len(matches) == 0:
       raise NoServiceMatchException()
 6   return matches[ 0 ]

 8 #@return a sorted list of matching services
   def findBestServices(i, p, s):
10   iMatches = iMatch(s, i)
     pMatches = pMatch(iMatches, p)
12   spMatches = sMatch(pMatches, p)
     matches = []
14   if len(spMatches) != 0:
       matches = sortMatches(spMatches)
16   else:
       sMatches = sMatch(iMatches, p)
18     matches = sortMatches(sMatches)
     return matches
20
   #@return a list of services matching interface
22 def iMatch(services, interface):
     list = []
24   for s in services:
       if s.interface == interface:
26       list.append(s)
     return list
28
   #@return reverse list of degree matches
30 def sortMatches(services):
     return services.sort(lambda s1, s2: s2.dMatch
         -s1.dMatch)
```

The policy matching algorithm is illustrated in Listing 3. The methods are:

pMatch A policy match is performed by matching the advertised policy for each service with the required policy. As such, this method iterates (line 35) over all services and performs a *pMatchAdvert* on each. If the *dMatch* of a service is positive (thereby indicating some match) then the service's *dMatch* is set in line 38 and added to the list of *matches* to be returned.

pMatchAdvert The policy-to-advertisement match is performed by iterating over all QoS for the *policy* (line 45) and finding a matching QoS in the *advertisement* (inner for-loop in line 46). If such a match is found then in line 47, a quality match is performed which results in the increase of the *dMatch*. Note that the best possible match occurs when all QoS in the policy perfectly match the advertisement QoS, thereby resulting in a maximum value for the *dMatch* of the *advertisement* and *policy* pair.

qualityMatch A quality match occurs when the quality *type* and *unit* are the same (lines 52 to 55). We additionally make sure, starting in line 57, that if the qualities are monotonically increasing then the first quality's (or *q1*) *preferred* value is within the range of the second (or *q2*) and also that *q1*'s range is a subset of *q2*'s. Similarly for monotonically decreasing quality types. When the conditions are satisfied the *degree* is calculated by *calculateDegree* (starting in line 74). When the conditions are not satisfied, the degree is forced to be negative since even though there is a match of quality, the advertised policy values do not match.

calculateDegree This is achieved by first adjusting the *dataSet* by adding the reputation for the quality [14]. Then we calculate the second moment of the *q1* about the *q2.preferred* value and then normalizing the result in equation 5.

$$\begin{aligned}
\vec{x} &= \langle x_1, x_2, \ldots, x_n \rangle && (1)\\
Moment(\vec{x}, a) &= \frac{\sum_{i=1}^{n} (a - x_i)^2}{n-1} && (2)\\
\vec{q} &= \langle q_1.min, q_1.max, q_1.typical, && (3)\\
&\quad q_2.min, q_2.max, q_2.preferred \rangle && (4)\\
degree(\vec{q}) &= norm(Moment(\vec{q}, q_2.preferred)) && (5)
\end{aligned}$$

Listing 3: Policy matching algorithm.

```
32 #@return list of services matching provider
       advertised policy
```

```
     def pMatch(services , policy):
34     matches = []
       for s in services:
36       dMatch = pMatchAdvert(s.advertisement ,
             policy)
         if dMatch > 0:
38         s.dMatch += dMatch
           matches.append( s )
40     return matches

42   #@return the degree of policy match
     def pMatchAdvert(advertisement , policy):
44     dMatch = 0
       for qos in policy.qosList:
46       for aQoS in advertisement.qosList:
           dMatch += qualityMatch(aQoS.quality , qos.
               quality):
48     return dMatch

50   #@return degree of match between q1 and q2
     def qualityMatch(q1, q2):
52     if type(q1) != type(q2):
         return 0
54     degree = 0
       if q1.unit != q2.unit:
56       convertUnit( q1, q2 )
       if checkMonoIncreasingQs( q1, q2 ) or
           checkMonoDecreasingQs( q1, q2 ):
58       dataSet = [q1.min, q1.max, q1.typical , q2.
             min, q2.max, q2.preferred]
         adjustWithReputation(q2, dataSet)
60       degree=calculateDegree(preferred , dataSet)
       else:
62       degree = -100
       return degree
64
     #@return true if the two monoIncreasing quality
         qValue are compatible
66   def checkMonoIncreasingQs( q1, q2 ):
       return q2.type.monoIncreasing and q1.type.
           monoIncreasing and q2.preferred >= q1.min
            and q2.preferred <= q1.max and q1.min
            >= q2.min and q1.max <= q2.max
68
     #@return true if the two monoDecreasing quality
         qValue are compatible
70   def checkMonoDecreasingQs( q1, q2 ):
       return q2.type.monoDecreasing and q1.type.
           monoDecreasing and q2.preferred <= q1.min
            and q2.preferred >= q1.max and q1.min
            <= q2.min and q1.max >= q2.max
72
     #@return the normalized degree
74   def calculateDegree(preferred , dataSet):
       degree = 0
76     m = stats.moment(2, dataSet , preferred)
       if m < 1:
78       degree = 100 - m
       else:
80       degree = 100 / m
       return degree
```

Listing 4 shows the details of the semantic matching algorithm. It comprises the following methods:

sMatch Similar to *pMatch* listed starting in line 33, *sMatch* iterates over all services to determine the semantic *dMatch* of each.

sMatchAdvert Also, similar to the policy advertisement match, a semantic advertisement match figures out the total *dMatch* for the *policy* and the *adverstisement* by doing a *semanticQualityMatch* on each pair of matching qualities.

semanticQualityMatch We semantically match two qualities by iterating over their relationships (line 85) to find if they are related. We then determine the degree of the relationship in line 87. The total degree of match is the sum of the degree of all of the matching relationships.

degreeOfRelationship The degree of a relationship is calculated by determining if the *relationship isSubclassOf* of *ValueImpact* (line 93) or of *ValueDirection* (line 100) and assigning increasing degree values for the actual type of relationship (lines 94 to 98 and lines 101 to 105). The actual constant value affecting the degree calculation defaults to the value listed but can be refined for a specific domain by attaching it to the ontology.

Listing 4: Semantic matching algorithm.

```
82   #@return semantic match for q1 and q2
     def semanticQualityMatch(q1, q2):
84     dMatch = 0
       for r in q1.relationships:
86       if r.relatedQs.contains(q2):
           dMatch += degreeOfRelationship(r)
88    return dMatch

90   #@return the degree of the relationship
     def degreeOfRelationship(relationship):
92     degree = 0
       if issubclass(type(relationship), ValueImpact
           .type):
94       if type(relationship)==Weak.type:
           degree = 10
96       elif type(relationship)==Mild.type:
           degree = 20
98       elif type(relationship)==Strong.type:
           degree = 30
100    elif issubclass(type(relationship),
           ValueDirection.type):
         if type(relationship)==Parallel.type:
102        degree = 10
         elif type(relationship)==Opposite.type:
104        degree = -10
         elif type(relationship)==Inverse.type:
106        degree = -20
       return degree
```

4. EMPIRICAL EVALUATION

The primary goal of our evaluation is to show empirically that the system enables a certain level of trust between consumers and providers while respecting autonomy. The experiment is made up of a series of simulations of consumers that use a set of integer sorting services. We first give an overview of the environment for the simulations and then discuss the expected and actual results.

4.1 How to Evaluate?

Along with the framework, we constructed a simulation component that allows the creation of different types of consumers in a simple Java scripting language: Jython [8]. Each consumer can have their own preference or policy. Consumers can be part of a group where they share the same script and policy. Each consumer script terminates by collecting data on its agent choices and the selected service. The following simulation parameters can also be specified:

1. *Service dopings.* These are runtime behavioral modifications to a service, specified to affect some service quality. The dopings can be parameterized to affect the level of each doping, e.g., a *FaultDoping* has options to specify the period of the faults and whether the faults should be random.

2. *Consumer groups.* Various groups of consumer can be specified for a simulation along with the group's size.

3. *Execution strategy.* This specifies the execution strategy for the various consumers. We have created two main strategies. First, a random sequential strategy where all consumers (for all groups) are executed in sequence but the selection as to which consumer to execute is random; and second, a random parallel strategy where n consumers execute in parallel but their selections from the consumer groups is random.

4.2 Setup Summary

We conducted five simulations for this experiment. Each simulation consists of three groups of five identical providers and five identical consumer instances.

Table 1 shows the secondary parameters for each simulation. Each consumer executes a *random-sort.py* script which calls the sorting service with a set of random integers. The set size is also random but bounded to 1000. Each simulation is set up to iterate 10 times.

Table 1: Simulations secondary parameters.

Simulation	Doping	Agency data?
0	N/A	No
1	Full doping	No
2	Full doping	Yes
3	Delayed full doping	Yes
4	Delayed partial doping	Yes

We implement each simulation using service providers of predictable behaviors. The primary service domain is *Math*. The domain and service interface are kept simple to facilitate measurement and fine-tuning of system parameters. The following artifacts are also used in the experiment.

- *Sorting service consumers.* We deploy three pools of consumers, each with its own QoS policy (part of which is summarized in Figure 3).
    1. *Careful.* As the name implies, this consumer's primary concern is *safety*.
    2. *Mellow.* This consumer's policy is to primarily find any service that is reliable and performs relatively well compared to the others.
    3. *Rushed.* This consumer is primarily concerned with execution speed.
- *Integer sorting interface.* This is the interface used by all sorting services. Essentially, it provides a method to sort an array of integers.
- *Integer sorting services.* Three implementations of the sorting service interface are deployed: *BubbleSort*, *MergeSort* and *QuickSort*.

  We create a pool of five identical provider instances for each implementations. Table 2 shows the identifier number assigned to each instance in the pool along with the set of instances that is doped and the set that remains clean.

Table 2: Providers number assignment (doped and clean).

	Size	Doped	Clean
BubbleSort pool	5	$\{0, 1, 2, 3\}$	$\{4\}$
MergeSort pool	5	$\{5, 6, 7, 8\}$	$\{9\}$
QuickSort pool	5	$\{10, 11, 12, 13\}$	$\{14\}$

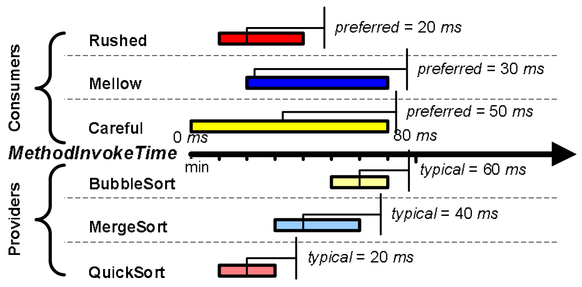

Figure 3: *MethodInvokeTime* policies.

- *Provider doping.* A provider's service is doped by artificially decreasing or increasing a particular quality. For this experiment we introduce three types of service doping:
    1. *DelayDoping.* Introduces a delay in a service method invocation.
    2. *FaultDoping.* Increases a service method fault rate by introducing artificial faults.
    3. *AvailDoping.* Decreases the availability of a service.
- *Sorting service agent.* This is the service agent for the sort service.

4.3 Results

Now we discuss the results obtained for all five simulations. For each simulation we show the obtained results as graphs illustrating the service selection choice for each pool of consumers. For each graph, the y-axis denotes the service number of the selection. Services are numbered according to Table 2. The x-axis shows the normalized execution sequence for the consumers of a pool. Since the execution strategy for all of the simulations is set to random sequential, the actual execution sequence is non overlapping for each pool of consumers. By normalizing the execution sequences, we effectively show the order of execution for each consumer pool. For instance, point 10 on the x-axis for any consumer pool represents the 11th consumer execution for that particular pool.

Finally, all simulations start with a clean database for all agencies. That is, the agencies are set up so that collected data do not persist across simulation runs. Thus, in the initial stages of any simulation run there will be no quality data to bias the agents toward any particular service for any pool.

4.3.1 Simulation 0 and 1: Base line service selection

In these first two simulations we try to establish a base line for the service selections of our pool of consumers. We

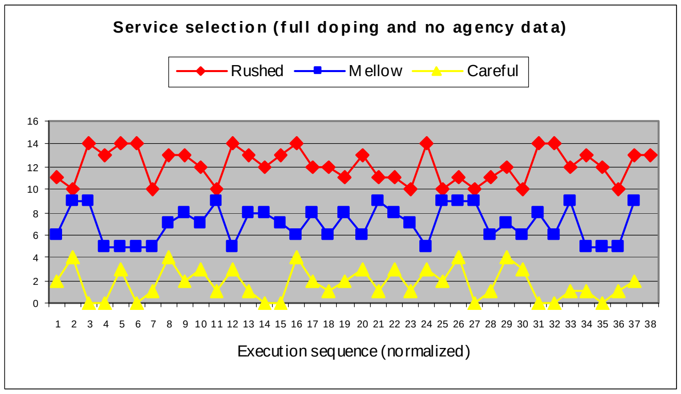

Figure 4: With doping but no agency data (Simulation 0).

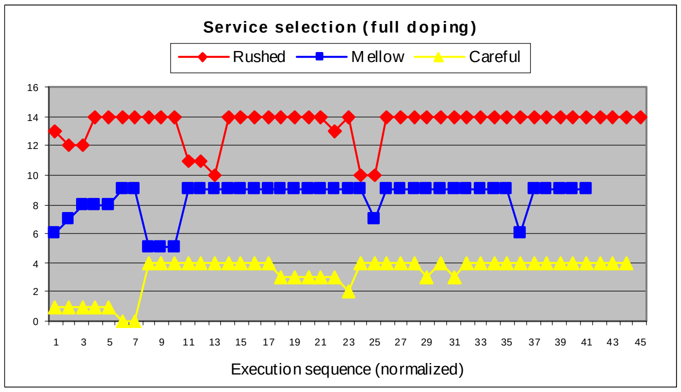

Figure 5: Full doping (Simulation 2).

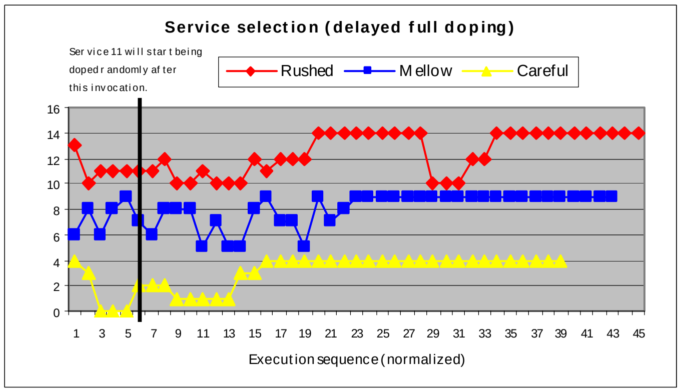

Figure 6: Delayed full doping (Simulation 3).

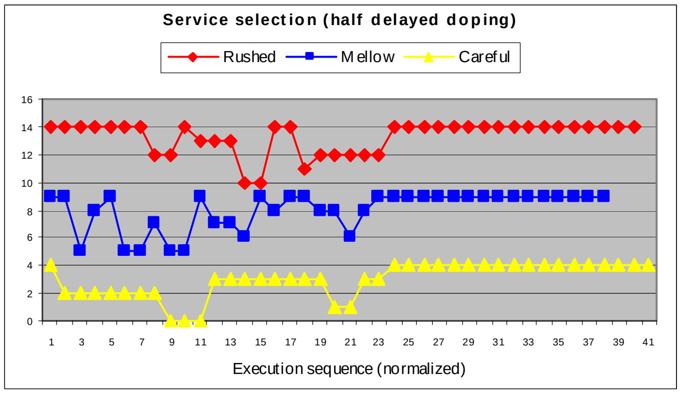

Figure 7: Delayed half-service doping (Simulation 4).

run the simulations without any agency data for the entire duration of the simulation. That is, the service agents do not consider the collected quality data in their selection decisions. The first simulation runs with doping turned off and the second with doping turned on.

As expected, the results show that service agents for a particular pool of consumer randomly selects between members of the service pool for which their policy biases them.

Figure 4 is similar to the graph obtained when doping is turned on and no agency data is used by the agents. This is the case because the agents are not considering the agency data in their decisions; although the services are doped, which increases their *MethodInvokeTime*, *FaultRate* and decreases *PercentAvailability*, the agents are blinded to that fact and thus cannot improve their decisions.

The remaining simulations show what happens when the agents start accounting for the agency data into their selection decisions.

4.3.2 *Simulation 2: Full service doping*

In this simulation all services but the last numbered service of each pool are doped. Since the last service of each pool is clean, we would expect that the agent would, in time, find that service and increasingly select it. Figure 5 shows that for all three pools of consumers, we obtain, as expected, convergence of all consumers for each pool to the lone clean service instance of each service pool. Notice also that since the doping occurs from the start of the simulation the convergence is gradual and start early on the simulation.

4.3.3 *Simulation 3: Delayed full doping*

In this simulation we introduce a delay for all doping. Essentially, all service doping will only start occurring after the 4th invocation for a particular service. For instance, Figure 6 shows that service 11 will not experience any doping until the 7th invocation or so—since all doping are random.

As expected, the delay essentially shifts the convergence to the clean service to the right of the graph. Figure 6 shows clearly that that although all services are doped as in Simulation 2, the delay in the doping causes the agent to converge more slowly to the clean service.

4.3.4 *Simulation 4: Delayed half doping*

This simulation introduces delayed doping of services but in a more incremental manner. Figure 7 shows the results for this simulation. As can be noticed from the graph, there is a slight push for convergence to the right part of the graph when compared to the full delayed case. Further, the convergence seems to occur in two steps at around execution sequence 11. This agrees with what we except since with half of the services doped, the agent should find the group of clean services early in the graph and then when doping starts taking effect for the other half, the agent gradually begins converging to the sole clean service.

5. RELATED WORK

Quality of service has been extensively studied in the context of computer networks and specifically the Internet. However, QoS in the context of software engineering and

Web services has seen a flurry of recent research activity. We discuss relevant literature on QoS requirements, QoS models and metrics, QoS middleware and frameworks and QoS-driven service discovery and selection.

5.1 QoS Requirements

Early articles [13, 16] highlighted the need for QoS in Web services as well as the main Web services standards themselves, e.g., UDDI.

The Web Services Level Agreement (WSLA) [7] addresses dynamic Web services procurement and qualities. WSLA emphasizes contracts agreed on by the providers and consumers for some usage. While WSLA solves some of the problems to guarantee quality levels, it lacks support for dynamism as required for a truly open environment.

Inspired by previous work [19, 7, 23], the W3C [24] summarizes the key requirements of QoS for Web services. Ludwig summarizes the current efforts in the field, especially by contrasting the work on service level agreement with other QoS brokering approaches [12].

5.2 QoS Models and Metrics

Although many works on QoS mention similar metrics such as reliability, availability, and security, most fail short of giving a full ontology of QoS in the context of Web services. Sabata et al. [20] sketched a QoS taxonomy, mostly in the context of Web applications. Recent work [23, 19, 12] gives some level of QoS modeling and hints at a QoS taxonomy without expanding the details. Our work, addresses this shortcoming by providing an initial QoS ontology.

Interestingly, the work by [23] also considers higher level of qualities such as Quality of Experience (QoE), the subjective quality perception of the end user; and Quality of Business (QoBiz), the economic characteristic of the service to the service provider. This work also stresses the importance of relationships among qualities. We also model relationships between qualities and use such relationships to improve service selection by better computing the preferences of the consumers.

The Web Services Policy Framework (WS-Policy) [4] gives a flexible open language for expressing policy constraints for services. Wohlstadter et al. [25] extend WS-Policy to express QoS of services. Our own policy language can be incorporated into WS-Policy in a straightforward manner.

UML profiles to model QoS are emerging [1, 5]. Aagedal and Ecklund [2] propose extensions focused on reliability. Our models are similar; however, we focus on Web services and model semantic quality relationships, which the UML profiles do not address.

5.3 QoS Middleware and Frameworks

Sheth et al. [21] highlight the need for QoS in composed services for dynamic selection of their parts. They point out that QoS concepts have dependent parameters. For instance, the request-response time is dependent on the input and output size of the service invocation. We adopt this notion of explicit QoS parameterization and model all QoS concepts to allow parameters.

Wohlstadter et al. [25] propose a middleware to pair clients and servers using QoS policies. They describe a policy language for expressing client and server policy requirements. They also describe a protocol executed by the middleware to find matches. Our work differs in the following main aspects. First, we use a decentralized multiagent architecture but their middleware is a centralized matchmaker. Second, we infer QoS relationships and interactions at runtime; they assume design time setup.

Menascé et al. [17] describes a framework and architecture to create software components that are QoS aware; that is, these components expose a service interface and allow negotiation of QoS requirements. Our work differs primarily from this one in that we use a multiagent systems architecture whereas they use a centralized middleware approach. They consider QoS negotiation as an explicit step whereas we do not; and instead, our agents determine the best service providers automatically without requiring negotiation.

5.4 QoS-Driven Discovery and Selection

Brokers can enable dynamic selection of services using QoS [19, 22]. The brokers use third party certifiers to collect QoS data on the services. The main difference with our work is that we do not use a centralized broker and certifier to determine a service's QoS characteristics.

Al-Ali et al. [3] introduce the concept of application QoS (AQoS) and describe a framework for adding QoS considerations in Grid Services for selection and management of individual services within the Grid. They differ primarily from our approach in that our aim is to select services in open environments.

Kalepu et al. [9] propose a new QoS metric to help select Web services. Building on previous work [14, 7, 19], they introduce the notion of *Verity*, which measures the consistency of compliance over time.

Zeng et al. [27] discuss a global planning approach for selecting composed services. They propose a simple QoS model using the examples of price, availability, reliability, and reputation. They apply linear programming for solving the optimization QoS matrix formed by all of the possible execution plans that result in the plan with the maximum QoS values. This work is similar to ours in some of the QoS modeling (which we extend) and applying statistical variance to deal with the nondeterministic characteristics of QoS values. The major differences with our work is that we extend reputation to encompass all qualities and our model for reputation has a dampening temporal characteristics.

Poladian et al. [18] present a mathematical model of the problem of configuring user tasks. They assume that each configuration is based on selecting services from providers of differing QoS. They give an optimization algorithm, which is known to be NP-complete but show a heuristic that simplifies it. Our work too models selection decision as an optimization problem.

6. CONCLUSIONS

We have described a framework to achieve service selection in a manner that considers the preferences of service consumers and the trustworthiness of providers. We evaluated our approach with simulation experiments and showed that a level of self-adjusting trust emerges from the system. We are exploring the idea of adding *explorer agents* to our framework to achieve better self-adjusting trust so that as bad services correctly behave they are reconsidered. In future work, we will expand our selection algorithm to take into account multiple quality objectives [10] and additional structural properties of QoS, e.g., statistical distribution of values and their correlation.

7. ACKNOWLEDGMENT

We thank Nirmit Desai of NCSU and the anonymous reviewers for useful comments.

8. REFERENCES

[1] J. O. Aagedal, M. A. de Miguel, E. Fafournoux, M. S. Lund, and K. Stolen. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms. Technical Report 2004-06-01, Object Management Group, June 2004.

[2] J. O. Aagedal and E. F. E. Jr. Modelling QoS: Towards a UML Profile. In *Proc. of* $\ll$ *UML* $\gg$ *2002*, pp. 275–289, Dresden, Germany, Oct. 2002. Springer LNCS.

[3] R. J. Al-Ali, O. F. Rana, D. W. Walker, S. Jha, and S. Sohail. G-QoSM: Grid Service Discovery Using QoS Properties. *Computing and Informatics Journal*, 21(4):363–382, 2002.

[4] D. Box et al. Web Services Policy Framework (WSPolicy) Specification Version 1.01. www-106.ibm.com/developerworks/library/ws-polfram/, June 2003.

[5] V. Cortellessa and A. Pompei. Towards a UML profile for QoS: a contribution in the reliability domain. In *Proc. of the fourth international workshop on Software and performance*, pp. 197–206. ACM Press, 2004.

[6] IBM Corporation. Web Services Conceptual Architecture (WSCA 1.0). www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf, 2001.

[7] IBM Corporation. Web Services Level Agreements. www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf, 2003.

[8] Jython. Jython 2.1. www.jython.org, 2001.

[9] S. Kalepu, S. Krishnaswamy, and S. W. Loke. Verity: A QoS Metric for Selecting Web Services and Providers. In *Proc. of the First Web Services Quality Workshop*, Rome, Italy, Dec. 2003. IEEE Computer Society.

[10] R. L. Keeney and H. Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. John Wiley & Sons, Hoboken, NJ, 1976.

[11] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, Jan. 2003.

[12] H. Ludwig. Web Services QoS: External SLAs and Internal Policies Or: How do we deliver what we promise? In *Proc. of the First Web Services Quality Workshop*, Rome, Italy, Dec. 2003. IEEE Computer Society.

[13] A. Mani and A. Nagarajan. Understanding Quality of Service for Web Services. www-106.ibm.com/developerworks/webservices/library/ws-quality.html, Jan. 2002. IBM DeveloperWorks.

[14] E. M. Maximilien and M. P. Singh. Conceptual Model of Web Service Reputation. *SIGMOD Record*, 31(4):36–41, Dec. 2002.

[15] E. M. Maximilien and M. P. Singh. A Framework and Ontology for Dynamic Web Services Selection. *IEEE Internet Computing*, 8(5):84–93, Sept. 2004.

[16] D. A. Menascé. QoS Issues in Web Services. *IEEE Internet Computing*, 6(6):72–75, Nov. 2002.

[17] D. A. Menascé, H. Ruan, and H. Gomaa. A Framework for QoS-Aware Software Components. In *Proc. of the Fourth International Workshop on Software and Performance*, pp. 186–196. ACM Press, 2004.

[18] V. Poladian, D. Garlan, M. Shaw, and J. P. Sousa. Dynamic Configuration of Resource-Aware Services. In *Proc. 26th International Conference on Software Engineering (ICSE 2004)*, pp. 604–613, Edinburgh, Scotland, May 2004. IEEE Computer Society.

[19] S. Ran. A Framework for Discovering Web Services with Desired Quality of Service Attributes. In L.-J. Zhang, editor, *Proc. of the International Conference on Web Services*, pp. 208–213, Las Vegas, NV, June 2003. IEEE Computer Society.

[20] B. Sabata, S. Chatterjee, M. Davis, J. J. Sydir, and T. F. Lawrence. Taxonomy for QoS Specifications. In *Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, Newport Beach, CA, Feb. 1997. IEEE Computer Society.

[21] A. Sheth, J. Cardoso, J. Miller, and K. Kochut. QoS for Service-Oriented Middleware. In *Proc. of the 6th World Multiconference on Sytemics, Cybernetics and Informatics (SCI02)*, volume 8, pp. 528–534, Orlando, FL, July 2002.

[22] M. Tian, A. Gramm, T. Naumowicz, H. Ritter, and J. Schiller. A Concept for QoS Integration in Web Services. In *Proc. of the First Web Services Quality Workshop*, Rome, Italy, Dec. 2003. IEEE Computer Society.

[23] A. van Moorsel. Metrics for the Internet Age: Quality of Experience and Quality of Business. Technical Report HPL-2001-179, Hewlett-Packard, Erlangen, Germany, July 2001.

[24] W3C. QoS for Web Services: Requirements and Possible Approaches. www.w3c.or.kr/kr-office/TR/2003/ws-qos/, Nov. 2003. Note.

[25] E. Wohlstadter, S. Tai, T. Mikalsen, I. Rouvellou, and P. Devanbu. GlueQoS: Middleware to Sweeten Quality-of-Service Policy Interactions. In *Proc. of 26th International Conference on Software Engineering (ICSE 2004)*, pp. 189–199, Edinburgh, Scotland, May 2004. IEEE Computer Society.

[26] G. Zacharia and P. Maes. Trust Management Through Reputation Mechanisms. *Applied Artificial Intelligence*, 14:881–907, 2000.

[27] L. Zeng, B. Benatallah, A. H. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. QoS-Aware Middleware for Web Services Composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.