Bootstrapping Performance and Dependability Attributes of Web Services

Florian Rosenberg, Christian Platzer, Schahram Dustdar
VitaLab, Distributed Systems Group, Information Systems Institute
Vienna University of Technology
1040 Vienna, Argentinierstrasse 8/184-1, Austria
{florian, christian, dustdar}@infosys.tuwien.ac.at

Abstract

Recently, Web services gain momentum for developing flexible service-oriented architectures. Quality of service (QoS) issues are currently not part of the Web service standard stack, although non-functional attributes like performance, dependability or cost and payment play an important role for service discovery, selection, and composition. A lot of research is dedicated to different QoS models, at the same time omitting a way to specify how QoS parameters (esp. the performance related aspects) are assessed, evaluated and constantly monitored. Our contribution in this paper comprises a) an evaluation approach for QoS attributes of Web services, which works completely service- and provider independent, b) a method to analyze Web service interactions by using our evaluation tool and extract important QoS information without any knowledge about the service implementation. Furthermore, our implementation allows assessing performance specific values (such as latency or service processing time) that usually require access to the server which hosts the service. The result of the evaluation process can be used to enrich existing Web service descriptions with a set of up-to-date QoS attributes, therefore, making it a valuable instrument for Web service selection.

1. Introduction

We consider the availability of a QoS description for a set of services as an *enabler* for solving many problems currently heavily investigated by different research groups. Such problems include composition, especially dynamic composition [2, 3, 23, 18, 6] as well as service discovery, search and selection of Web services [14, 9, 11].

A major drawback of Web services in the current version is the lack of Quality of Service (QoS) attributes [12, 10] for service descriptions. Such attributes define non-functional attributes of a service. These include availability, latency, response time, authentication, authorization, cost, etc. Our investigations showed, that primarily performance-related aspects of Web services are required by various researchers. For the time being, those very specific measurements are not available a priori. The only thing that can be taken for granted is the availability of a service description itself.

In this paper, we introduce a framework, which provides the possibility to assess QoS attributes for a given set of Web services. The main contribution lies in our automatic approach for bootstrapping and constantly monitoring QoS parameters for existing services that are currently lacking such valuable descriptions. We aim to achieve both, a maximized dynamic sampling and a broad spectrum of usable Web services. As a result, we provide a tool that allows to categorize their service repository according to the most important characteristics. Sometimes, a service is very important and needs a high availability rating, while in other cases the response time is much more important. Therefore, QoS-driven development finally becomes possible.

Here, we mainly deal with performance and availability related QoS attributes by using a flexible Web service invocation mechanism combined with aspect-oriented programming which allows us to weave performance measurement aspects directly into the byte-code of the Web service stubs [8]. We omit business related values, such as cost, payment, etc, which cannot be determined automatically. These attributes are rather defined by the business and cost model implemented by the service provider.

The result of our work is a basic set of QoS attributes for a given service. To further use them, we abstract from the concrete QoS attribute attachment mechanism, thus, we support two possibilities, (a) publish the QoS attributes together with the service description in UDDI (proposed by [19] or [15]) or (b) add them to the WSDL file by using WS-Policy [20].

The remainder of this paper is organized as follows: Section 3 describes our QoS model by categorizing them into different groups and give a definition for each attribute. Section 4 describes our main approach for bootstrapping and

evaluating QoS attributes for Web services. In Section 5, we evaluate our approach by applying it to some example Web services. Section 2 briefly discusses existing approaches related to our work. Finally, Section 6 concludes this work and highlights some future work.

2. Related Work

Quality of service research encompasses different research domains such as the network, software engineering and more recently service-oriented computing community. A lot of research is dedicated to the area of QoS modeling and enriching Web services with QoS semantics. To the best of our knowledge, most of the existing work leaves open the way how QoS parameters are bootstrapped or evaluated.

In [21], Wickramage and Weerawarana define 15 distinguishable periods in time, a SOAP request goes through before completing a round trip. This value, which is also referred to as response time, can be split up into different components where it is essential to bootstrap as much of them as possible. Our approach does not use all 15 periods identified in this work, because not all periods are interesting to consumers of a service and can be determined from the client.

Suzumura et al. [17] did some work on performance optimizations of Web services. Their approach is to minimize XML processing time (which we call wrapping time) by using differential deserialization. The idea is to deserialze only these parts which have not been processed in the past. In contrast to our approach, we do not try optimize the performance, we try to measure the different attributes, without dealing with the wrapping time itself.

A QoS model and a UDDI extension for associated QoS to a specific Web service is proposed in [15]. The QoS model proposed in this paper is very similar to our model, hence, the author does not specify how these values are actually assessed and monitored. It is assumed the QoS attributes of a service are specified by the service provider in UDDI. In [19], another approach for integrating QoS with Web services is presented. The authors implemented a tool-suite for associating, querying and monitoring QoS of a Web service. In contrast to our work, it is not specified how QoS attributes are actually measured and evaluated to associate them to certain Web services.

In [16], the Song and Lee propose a simulation based Web service performance analysis tool called sPAC which allows to analyze the performance of Web process (i.e., a composition) by using simulation code. Their approach is to call the Web service once under low load conditions and then transform these testing results into a simulation model. Our work focuses also on the performance aspects of Web services whereas we do not deal with Web processes. Furthermore, we do not use simulation code, we perform our evaluation on real Web services with the constraint that we do not have access to the Web service implementation.

QoS attributes in Web service composition also raise a lot of interest due to the fact that they can be used by the compositor to dynamically choose an suitable service for the composition regarding the performance, price or other attributes. A simple but illustrating example of such a composition is presented in [13]. In [22] and [23], the authors propose a QoS model and a middleware approach for dynamic QoS-driven service composition. They investigate a global planning approach to determine optimal service execution plans for composite service based on QoS criteria. In contrast to our work, the authors do not specify how QoS attributes for atomic services are measured and assessed. It is assumed that the atomic services already have QoS reasonable attributes.

3. QoS Model

This section describes our basic QoS model which is used for expressing QoS attributes for Web services. The most important point to realize here is that most of the evaluated attributes are dynamic and site-dependent. The service response time, for example, will experience a significant variation, depending on the type of connection used to evaluate it (e.g., Modem, DSL, T1 etc.). As a result, the produced values cannot be seen as global attributes, but as site-specific statistics with a strong local context. This is an intended behavior, because the parameters, influenced by the local conditions, increase the significance of the whole value. We assume two Web services, for example, named A and B with the same implementation and the same hardware. The Web service consumer is located at a remote place where the routing of the actual IP packets is the only difference between the two services. Therefore, A may respond faster than B in this case while B could be the faster service when queried from another place. With our framework, a developer can choose the currently best suitable service depending on the provided QoS attributes at runtime.

In the next section we will break down the QoS attributes in different categories and describe how they are evaluated in a real-world scenario.

3.1. Categorization

We categorize our model into several groups with each group containing related QoS attributes. We identified four main QoS groups, namely *Performance*, *Dependability*, *Security* and *Cost and Payment*. As already mentioned in the introduction, we focus on bootstrapping, evaluating and constantly monitoring the QoS attributes of the first two groups. The other ones are out of scope of this paper.

3.1.1 Performance

Processing time: Given a service S and an operation o, the processing time $t_p(S, o)$ defines the time needed to actually carry out the operation for a specific request R. The processing of the operation o does not include any network communication time and is, therefore, an atomic attribute with the smallest granularity. Its value is determined by the implementation of the service and the corresponding operation. To take Google as an example, t_p entitles the actual search time that is also displayed for search-requests sent by the Web interface.

Wrapping Time: The wrapping time $t_w(S, o)$ is a measure for the time that is needed to unwrap the XML structure of a received request or wrap a request and send to the destination. The actual value is heavily influenced by the used Web service framework and even the operating system itself. In [21], the authors even split this time into three sub-values where receiving, (re-)construction and sending of a message are distinguished. For our purpose it does not matter if the delay is caused by the XML-Parser or maybe the implementation of the socket connection, because it is constant for the same request.

Execution Time: The execution time $t_e(S, o)$ simply is the sum of two wrapping times and the processing time: $t_e = t_p + 2 * t_w$. It represents the time that the provider needs to finish processing the request. It starts with unwrapping the XML structure, processing the result and wrapping the answer into a SOAP envelope that can be sent back to the requester.

Latency: The time that the SOAP message needs to reach its destination is depicted as latency or network latency time $t_l(S)$. It is influenced by the type of the network connection the request is sent over. Furthermore, routing, network utilization and request-size play a significant role for the latency.

Response Time: The response time of a service S is the time needed for sending a message M from a given client to S until the response R for message M returns back to the client. The response time is provider specific, therefore, it is not possible to specify a globally valid value for each client. The response time $t_r(S, o)$ is calculated by the following formula: $t_r(S, o) = t_e(S, o) + 2 * t_l(S)$.

Round Trip Time: The last time-related attribute is the round trip time t_{rt}. It gives the overall time that is consumed from the moment a request is issued to the moment the answer is received and successfully processed. It comprises all values on both, requester and consumer side. Considering the formula above it can be calculated as:

$$t_{rt} = (t_w)_{con.} + t_l + (t_p + 2 * t_w)_{provider} + t_l + (t_w)_{con.}$$

See Figure 1 for a graphical representation of all involved time frames.

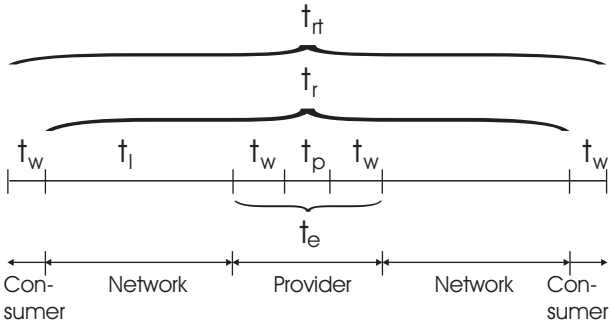

Figure 1. Service Invocation Time Frames

Throughput: The number of Web service requests R for an operation o that can be processed by a service S within a given period of time is referred to as throughput $tp(S, o)$. It can be calculated by the following formula:

$$tp(S, o) = \frac{\#R}{time\ period\ (in\ sec)}$$

This parameter depends mainly on the hardware power of the service provider and is measured by sending many requests in parallel for a given period of time (e.g., one minute) and count how many request come back to the requester.

Scalability: A Web service that is scalable, has the ability to not get overloaded by a massive number of parallel request. A high scalability value states the probability for the requester of receiving the response in the evaluated response time t_r.

$$sc(S) = \frac{t_{rt}}{t_{rt(Throughput)}},$$

where $t_{rt(Throughput)}$ is the round trip time which is evaluated during the throughput test.

3.1.2 Dependability

Availability: The probability that a service S is up and running. The availability can be calculated the following way:

$$av(S) = 1 - \frac{downtime}{uptime}$$

The downtime and uptime are measured in minutes.

Accuracy: The accuracy $ac(S)$ of a service S is defined as the success rate produced by S. It can calculated by evaluating all invocations starting from a given point in time and examining their results. The following formula expresses this relationship:

$$ac(S) = 1 - \frac{\#failed\ requests}{\#total\ requests}$$

Robustness: It is the probability that a system can react properly to invalid, incomplete or conflicting input messages. It can be measured by tracking all the incorrect input messages and put it in relation with all valid responses from a given point in time:

$$ro(S) = \frac{\sum_{i}^{n} f(resp_i(req_i(S)))}{\#total\ requests}$$

The part $resp_i(req_i(S))$ represents the i^{th} response to the i^{th} request to the service S, where n is the number of total requests to S. The utility function f is calculated as:

$$f = \begin{cases} 1, & isValid(resp_i) \\ 0, & \neg isValid(resp_i) \end{cases}$$

and is used to evaluate, if the response was correct for a given input.

4. Bootstrapping, Evaluation and Monitoring Approach

A number of possibilities exist to design a QoS-based evaluation and monitoring approach for Web services but all approaches share the same premises concerning the knowledge and access about the Web service under evaluation. We mainly consider two different premises:

Access to the Web service implementation: The approach seems to be the easiest one, because full access to the service implementation is available. This implies that performance measurements, such as latency or the service processing time of Web service requests can be exactly determined by combining the evaluation software running on the client-side with information on the server side (e.g., by implementing special handlers or loggers as it is possible with the Apache Axis Web service environment [1]. In practice, this approach is not feasible because the implementation is considered private property of the service provider.

No access to the Web service implementation: This approach is a client-side technique, where only the WSDL file of the service description is available. Therefore, alternative techniques for measuring the *processing time* and *latency* has to be implemented.

4.1. Overview

Our bootstrapping and evaluation approach for the different QoS parameters from Section 3 is such a *client-side technique* which works completely Web service and provider independent. Many different steps are necessary to successfully bootstrap and evaluate QoS attributes for arbitrary Web services. An overview of the main blocks of our system architecture and the three different phases of our evaluation process are depicted in Figure 2.

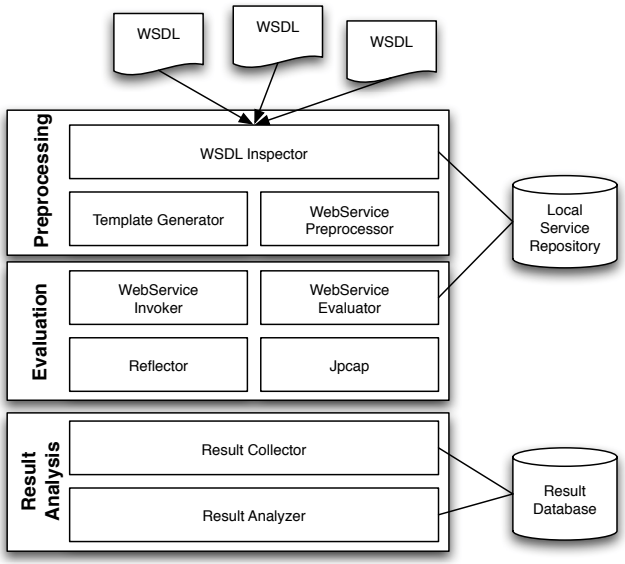

Figure 2. System Architecture

Preprocessing Phase: In this initial phase, the `WSDL Inspector` takes the URL of one or more WSDL files as an input and fetches it into the *local service repository*. Then, the WSDL file is parsed and analyzed to determine the SOAP binding name (we only support SOAP bindings, HTTP GET or POST is not supported). From the binding name, we can retrieve the corresponding `portType` element, thus, getting all operations which we have to evaluate. Furthermore, we have to parse all XSD data types defined within the `types` tag to know all the available types needed for invoking the service operations. The information gathered by analyzing the WSDL is used in the evaluation phase to dynamically invoke different operations of a service. As a next step, the Web service stubs are generated as Java files by using the WSDL2Java tool from Axis [1]. The performance measurement code itself is implemented by using aspect-oriented programming (AOP), thus, we defined an aspect which captures the evaluation information where it occurs. We discuss the aspect in detail in Section 4.3. The Java source files together with the aspect are compiled with the AspectJ compiler to generate the Java classes. All the aforementioned steps are fully automated by the `WebServicePreprocessor` component and do not need to be executed every time a specific service has to be evaluated. It has to be done only once, then the generated code is stored in the local service repository and can

be reused for further re-executions of the evaluation process itself.

Evaluation Phase: During the evaluation phase, the information from the WSDL analysis in the preprocessing phase is used to map the XSD complex types to Java classes created by the WSDL2Java tool. Furthermore, we heavily use Java Reflection, encapsulated in the `Reflector` component, to dynamically instantiate these complex helper classes and Web service stubs. The `WebServiceInvoker` component tries to invoke a service operation just by "probing" arbitrary values for the input parameters for an operation. If this is not possible, e.g., because an authentication key is required, we support a template based mechanism that allows us to specify certain parameters or define ranges or collections to be used for different parameters of a service operation. Such a template is also generated by the `TemplateGenerator` component during the preprocessing phase based on the `portType` element information in the WSDL file. The main part of the evaluation is handled by the `WebServiceEvaluator` and the `EvaluationAspect`. The aspect defines a pointcuts for measuring the performance related QoS attributes from Section 3. For example, the response time t_r is measured by defining a pointcut which timestamps before and after the `invoke(..)` method of the Axis `Call` class. The `Call` class handles the actual invocation to the Web service within the previously generated stub code. The response time itself is then calculated by subtracting the timestamp after the `invoke(..)` call from the timestamp before the call. Due to our client-side mechanism, the QoS parameter such as the latency t_l cannot be measured as comfortable as t_r. Therefore, we additionally use the packet capturing library Jpcap [5] within the `EvaluationAspect` to measure the latency and the processing time on the server by using information from the captured TCP packets. Details are discussed in the next sections.

Result Analysis Phase: In this phase, the result generated from the `WebServiceEvaluator` are collected by using the `ResultCollector` which represents a singleton instance. It collects all results generated by the `WebServiceEvaluator` and the `EvaluationAspect` and stores it in a database. Afterwards, the `ResultAnalyzer` iterates over these collected results and generates the necessary statistics and QoS attributes. Moreover, these resulting QoS attributes can be attached directly to the evaluated service as mentioned in Section 1. It is relatively straightforward to do so, and therefore not explained in detail in this paper.

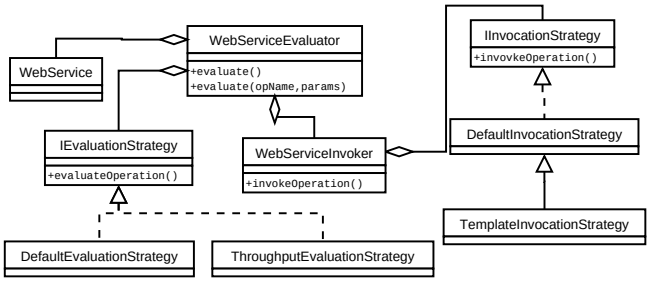

Figure 3. Architectural Approach

4.2. Architectural Approach

The architecture for evaluating and thus invoking arbitrary Web services is quite flexible because we applied many design patterns from [7]. In Figure 3, some parts of our architecture are depicted as an UML class diagram. The core class is the `WebServiceEvaluator` which encapsulates all the evaluation specific parts. A `WebService` class encapsulates all information about a Web service (endpoint, reference to WSDL, location in repository, port type, binding information, etc). An evaluation is either performed at the operation level or the service level. The operation level denotes that only one given operation of a service is evaluated by using the `evaluate(String operationName, Object[] parameters)` method. The service level means that all operations of a service are evaluate by using the `evaluate()` methods, which implicitly invokes the aforementioned method for every operation. Furthermore, we have different invocation mechanisms for a Web service with respect to the way we generate reasonable input parameters for the different operations.

Our architecture encapsulates the algorithms how we actually evaluate a service or invoke certain operations of a service by using the strategy pattern [7]. We have two strategies how we evaluate an operation. The `DefaultEvaluationStrategy` simply calls a service operation once with a given implementation of the `IInvocationStrategy` interface. The QoS attributes are encapsulated in the `EvaluationAspect`, which is woven into the byte code of our application and the stub code of the service. By contrast `ThroughputEvaluationStrategy` allows us to measure the throughput of a Web service by sending multiple requests in concurrent threads to the service, according to the formula given in Section 3.

Service Invocation Strategies. The invocation strategy defines how we invoke a service operation. Again, two different choices are available. The

`DefaultInvocationStrategy` implements a default behavior by iterating over all parts of the input messages of a service and instantiating the corresponding input type as generated by the WSDL2Java tool. The instantiation of complex types (even nested ones) is handled by the `Reflector` component. The `TemplateInvocationStrategy` uses the invocation template generated during the service preprocessing to invoke the service operation. The template can be edited by the user to add various pre-defined values for the different input parameters. In Listing 1, the main algorithm for invoking a service operation with a previously generated template is depicted. The algorithm uses the stubs for the Web service which are generated during the preprocessing phase and tries to find a value for each parameter in the XML template file. If no parameters can be found in the template or the template is not available, the `Reflector` tries to instantiate the required parameter type with a default value (handled by the `initalizeParameter()` method).

The main advantage of this flexible architecture is the possibility to add new evaluation and invocation mechanisms or even selecting them at runtime without changing the structure of the system and the aspect.

```
public Object invokeOperation(String operationName,
                              Object[] paramValues) {
  Operation op = service.getOperation(operationName);
  Message inputMsg = op.getInput().getMessage();
  Class[] parameters = new Class[parts.size()];
  Object[] paramInstances = new Object[parts.size()];

  // go through each part and try to find Java class
  // or simple type and initialize it
  for(Part p : inputMsg.getParts()) {
    QName type = p.getTypeName();
    Class param = convertXSDTypeToJavaType(type);
    if (param == null) { // it is not a simple type
      String javaName = convertQNameToPackageName(type);
      param = Class.forName(javaName, false, classloader);
    }
    parameters[i] = param;
    paramInstances[i] = initalizeParameter(operationName,
                            p.getName(), param);
  }
  // use reflection to invoke the operation
  Method m = service.getStubClass().getMethod(
                        operationName, parameters);
  return m.invoke(service.getStub(), paramInstances);
}

public Object initalizeParameter(String operationName,
                    String paramName, Class paramType) {
  // try to find param value from template
  String value = findParamValue(operationName, paramName);
  if (value != null) {
    return convertToObject(paramType, value);
  }
  return Reflector.instantiate(paramType);
}
```

Listing 1. invokeOperation Algorithm

4.3. Evaluating QoS Attributes using AOP

Our approach measures the performance related QoS values which is achieved by using aspect-oriented programming (AOP). It is an ideal technique for modeling cross-cutting concerns. Our evaluation part is such a cross-cutting concern since it spans over each service we have to invoke during the evaluation. The basic idea of our approach is described in Figure 4. During the preprocessing phase, we generate the stubs for the service which should be evaluated by using the WSDL2Java tool. For the Google Web service, as one example, the main stub class that is generated is called `GoogleSearchBindingStub`. Each stub method looks similar, first is the wrapping phase, where the input parameters are encoded in XML. Secondly, the actual invocation is carried out by using the `invoke(..)` method of the `Call` class from the Axis distribution. At last, the response from the service is unwrapped and encoded as Java arguments and returned to the caller.

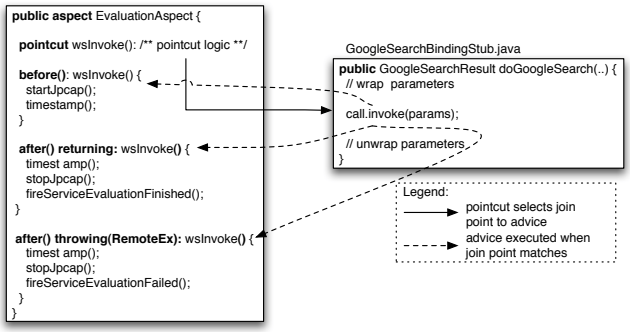

Figure 4. Aspect for Service Invocations (simplified)

Therefore, the `EvaluationAspect` defines the following pointcut to measure the response time t_r:

```
pointcut wsInvoke(): target(org.apache.axis.client.Call)
                  && ( call( Object invoke(..)) ||
                       call( void invokeOneWay(..)));
```

Whenever a service operation is invoked by using the `WebServiceEvaluator`, the `wsInvoke()` pointcut defined for this join point is matched. Before the actual service invocation the `before` advise is executed. This is where the actual evaluation has to be carried out. It mainly consists of a timestamp and the generation of an `EvaluationResult`, as well as starting the packet-sniffer JPCAP to actually trace the TCP traffic caused by this request.

After the `wsInvoke()` pointcut the execution of the corresponding `after` advise is triggered, depending whether the service invocation was successful or not. At

this point, packet capturing can be stopped and the collected data can be extracted. The timestamps taken before and after the invocation can directly be used to calculate the response time. To distinguish between latency and execution time, we have to move one level deeper to the TCP level. One Web service invocation actually consists of at least three sub-messages. The first and the last are always handshake messages, with no payload attached. Those messages are perfectly adequate, because the time difference between them is only caused by latency. The other TCP packets include the actual payload. It can either be transferred in a single packet or as a multipart message. From the moment, the last packet was transferred until the first packet that includes the response, the Web service provider processes the request. Therefore the execution time can be extracted by subtracting the average latency from the time difference where the execution time is included.

All values are stored in the MySQL-database because the more results are available, the more accurate the overall evaluation is. See Section 5 for a sample of 100 request/response cycles.

4.4. Implementation Details

The system is implemented with the Java 5 platform and AspectJ 1.5 [4] for implementing the evaluation part. For parsing and analyzing the WSDL files we use the WSDL4J library from SourceForge[1]. The transformation from WSDL to Java classes is handled by the Axis WSDL2Java tool [1]. The calculation of the latency and the execution time is done by using Jpcap [5], a Java wrapper for libpcap, which allows to switch to the promiscuous mode to receive all network packets from the NIC not only these addressed to our MAC address. These raw evaluation results are collected and stored in a MySQL database.

5. Evaluation

In this section we finally provide a sample of three Web services we analyzed with our tool. The following Web services were queried:

- Google: The well-known Google Web service for searching Web sites.
- CaribbeanT: A Web service to search for historical tourism data in the caribbean.
- Zip2Geo: A Web service to convert a zip code to longitude and latitude values.

To provide a representative overview on the QoS values, we monitored the mentioned services for 24 hours while

[1] `http://sourceforge.net/projects/wsdl4j`

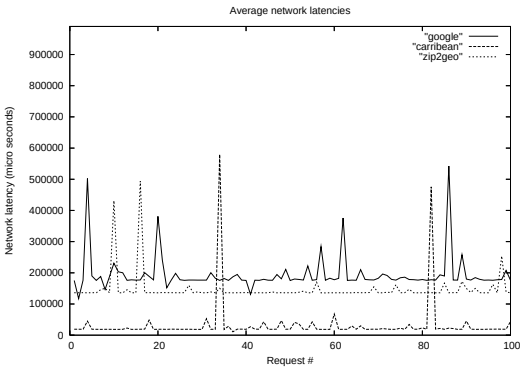

Figure 5. Network Latency

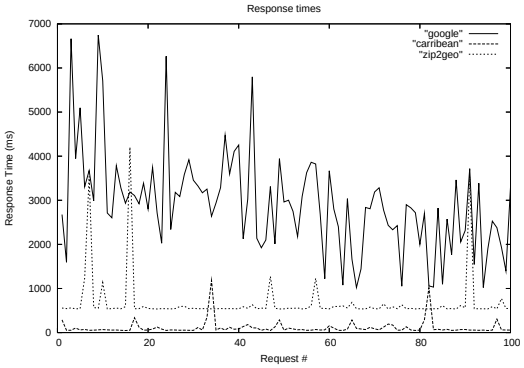

Figure 6. Response time

constantly evaluating all parameters. The data was bootstrapped on a 2,4 GHz Intel Pentium IV with 1GB of main memory, running Fedora Core 4. Figure 6 shows a graphical representation of the overall response time for all three services. It shows that the caribbeanT Web service responds very efficiently to a request and has quite a small variation with about 1000ms as the maximum response time in our time window. Figure 5 shows the network latencies on TCP level. Finally, we evaluated the most important statistics for all three services. See Table 5 for the values, according to the time frames explained in Figure 1.

The displayed values are an average of approximately 400 different samples, recorded during the above mentioned time period of 24 hours. We elided the values for throughput

	Google	CarribeanT	Zip2Geo
Latency	98ms	25,9ms	143ms
Response Time	2668ms	94,4ms	635ms
Execution Time	2,4ms	7,5ms	17,6ms
Availability	100%	100%	100%
Accuracy	99,2%	100%	80,9%

Table 1. Statistics

and scalability on purpose because we run our evaluation tool at the university with a 100Mbit Internet connection, where a throughput test with maximum load would simply result in a denial of service and could therefore be rated as an attack. Furthermore, we assume that the Google Web service scales quite well, so we would need at least 1000 requests per minute to produce significant results.

Summing up, caribbeanT produced the best overall results but the reader has to keep in mind that Google is the only service with a high utilization but still produces very good results and a low failure rate.

6. Conclusions

In this paper we have introduced an approach for bootstrapping, evaluating and monitoring performance related QoS attributes for Web services. Furthermore, our approach works completely service independent, allowing invocation of arbitrary services (at least semi-automatically). Our implementation uses a combination of object-oriented and aspect-oriented programming techniques to gain maximum flexibility. Our evaluation results prove the usability of our approach for QoS based service selection.

Nevertheless, there is a lot of additional research needed in this field. Our future work will improve our current prototype by using additional QoS categories which are not only related to performance. Furthermore, we want to encapsulate our implementation as a Web service allowing easier integration with other applications, thus, making it usable for some of our previous work for searching and selection of Web services [14], as well as a novel way to optimize service compositions based on these values.

References

[1] Apache Software Foundation – Apache Axis. `http://ws.apache.org/axis`, 2005.

[2] F. Casati and M.-C. Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.

[3] S. Dustdar and W. Schreiner. A Survey on Web services Composition. *International Journal of Web and Grid Services*, 1, 2005.

[4] Eclipse Foundation, Inc. Eclipse AspectJ. `http://www.eclipse.org/aspectj/`, 2005.

[5] K. Fujii. Jpcap – Java package for packet capture. `http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html`, 2005.

[6] K. Fujii and T. Suda. Dynamic service composition using santic information. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 39–48, New York, NY, USA, 2004. ACM Press.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[9] Y. Liu, A. H. Ngu, and L. Z. Zeng. QoS computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW'04)*, pages 66–73. ACM Press, 2004.

[10] A. Mani and A. Nagarajan. Understanding quality of service for web services. `http://www-128.ibm.com/developerworks/library/ws-quality.html`, 2002.

[11] E. M. Maximilien and M. P. Singh. Toward autonomic web services trust and selection. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC'04)*, pages 212–221, New York, NY, USA, 2004. ACM Press.

[12] D. A. Menasce. QoS issues in Web services. *IEEE Internet Computing*, 6(6):72–75, November/December 2002.

[13] D. A. Menasce. Composing Web Services: A QoS View. *IEEE Internet Computing*, 8(6):88–90, November/December 2004.

[14] C. Platzer and S. Dustdar. A Vector Space Search Engine for Web Services. In *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)*, 2005.

[15] S. Ran. A model for web services discovery with QoS. *SIGecom Exchanges*, 4(1):1–10, 2003.

[16] H. G. Song and K. Lee. sPAC (Web Services Performance Analysis Center): Performance Analysis and Estimation Tool of Web Services. In *Proceedings of the 3rd International Conference on Business Process Management (BPM'05)*, pages 109–119, 2005.

[17] T. Suzumura, T. Takase, and M. Tatsubori. Optimizing Web services performance by differential deserialization. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 185–192, 2005.

[18] S. Tai, R. Khalaf, and T. Mikalsen. Composition of coordinated web services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 294–310, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[19] M. Tian, A. Gramm, H. Ritter, and J. Schiller. Efficient Selection and Monitoring of QoS-aware Web services with the WS-QoS Framework. In *Proceedings of the International Conference on Web Intelligence (WI'04), Beijing, China*, 2004.

[20] Web Services Policy Framework. `http://www-128.ibm.com/developerworks/library/specification/ws-polfram/`, 2004.

[21] N. Wickramage and S. Weerawarana. A benchmark for web service frameworks. In *Proceedings of the IEEE International Conference on Service Computing (SCC'05)*, 2005.

[22] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Z. Sheng. Quality driven web services composition. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03)*, pages 411–421, New York, NY, USA, 2003. ACM Press.

[23] L. Zeng, B. Benatallah, A. 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.