Experiments Results and Large Scale Measurement Data for Web Services Performance Assessment

Riadh Ben Halima [(1,2)], Emna Fki [(2)], Khalil Drira [(1)] and Mohamed Jmaiel [(2)]
[(1)] LAAS-CNRS, University of Toulouse, 7 avenue de Colonel Roche, 31077 Toulouse, France
{rbenhali,khalil}@laas.fr
[(2)] University of Sfax, National School of Engineers, B.P.W, 3038 Sfax, Tunisia
fkiemna@yahoo.fr, Mohamed.Jmaiel@enis.rnu.tn

Abstract

Service provisioning is a challenging research area for the design and implementation of autonomic service-oriented software systems. It includes automated QoS management for such systems and their applications. Monitoring and Measurement are two key features of QoS management. They are addressed in this paper as elements of a main step in provisioning of self-healing web services. In a previous work [1], we defined and implemented a generic architecture applicable for different services within different business activities. Our approach is based on meta-level communications defined as extensions of the SOAP envelope of the exchanged messages, and implemented within handlers provided by existing web service containers. Using the web services technology, we implemented a complete prototype of a service-oriented Conference Management System (CMS). We experienced our monitoring and measurement architecture using the implemented application and assessed successfully the scalability of our approach under the French grid5000. In this paper, experimental results are analyzed and concluding remarks are given.

1 Introduction

Internet progress has enabled data exchange between remote collaborators and the multiplication of on-line services. Several platforms of services are available today to support the design, the deployment and the implementation of these services in reduced scales of time. However, users require more exigencies. They dislike to encounter problems while using a service. Such as, waiting for a long time in order to book a room, checking whether the crashed rent car service is restored, searching for services with improved QoS. Therefore, we have to deal with these problems at the design time, and provide a strategy for recovery in order to satisfy users requirements. Such systems, have to inspect their behavior and change it when the evaluation indicates that the intended QoS is not achieved, or when a better functionality or performance is required. This implies the need of deploying entities for supervising traffic between web service providers and requesters in order to act for healing or preventing [3].

Three main steps are distinguished in the self-healing process [1]: *Monitoring* to extract information about the system health (using knowledge about the system configuration), *Diagnosis & Planning* to detect degradations and generate repair plans, and *Repair* to enforce reconfiguration actions in order to heal the system. In this paper, we focus on QoS monitoring of a web service-based application. We are achieving experiments on the cooperative reviewing system which is developed in the framework of the WS-DIAMOND European project. We carry out experiment on the "*conference search*" web service. The monitoring is carrying out while using monitors-based approach. We present the performance measurement values and analyze experiment results. The measurement is fulfilled under the grid5000 which is an experimental grid platform gathering 3000 nodes over 8 geographically distributed sites in France.

This paper is organized as follows. Section 2 describes our self-healing architecture. Section 3 presents the monitoring framework. Section 4 details experiments and gives concluding remarks. Section 5 presents related works. The last section concludes the paper.

2 Self-Healing Architecture

Figure 1 shows the different self-healing modules interaction within the dynamically reconfigurable web services bus. The bus offers a flexible framework for invoking services and managing QoS.

In the following, we describe the self-healing compo-

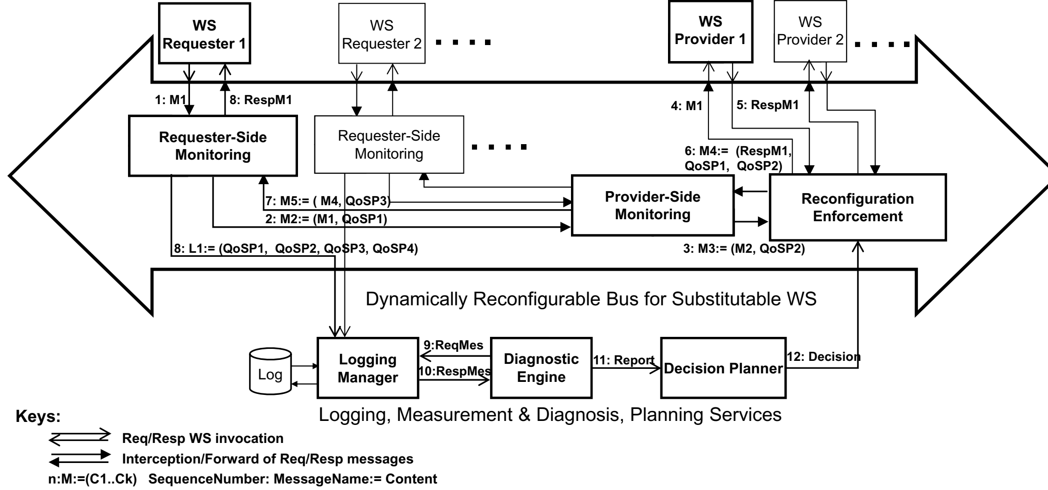

Figure 1. QoS-Oriented, Monitoring-based self-healing architecture

nents [1]:

- *Monitoring & Measurement:* It observes and logs relevant QoS parameters values. It is composed of *Requester-Side Monitoring*(*RSM*), *Provider-Side Monitoring* (*RSM*),and *Logging Manager*.

- *Diagnostic Engine & Decision Planner:* It detects degradation and defines repair plans.

- *Reconfiguration Enforcement:* It enforces repair plans by rerouting requests to the suitable web service provider.

In the sequel, we present the interaction messages exchanged between the web services and the self-healing components. The *WS Requester* sends a request message *M1*. This message is intercepted by the *RSM*. Message *M1* is then extended by the first QoS parameter value (*QoSP1*) in the output message *M2*. For example, *QoSP1* may represent the invocation time of the service by the requester. Message *M2* is intercepted by the *PSM* for a second time. *M2* is extended by the QoS parameter value (*QoSP2*) in the output message *M3*. To illustrate this, *QoSP2* may represent the communication time spent by the message to reach the provider-side network. The current bound *WS Provider* executes the request. The message response *M4* is intercepted by the *PSM* for a third extension by the QoS parameter value (*QoSP3*) in the output message *M5*. For example, *QoSP3* may represent the execution time associated with the request. Message *M5* is intercepted by the *RSM*. It is then extended by the fourth QoS parameter value (*QoSP4*). For example, *QoSP4* may represent the time spent by the response to reach the requester-side. The QoS data is extracted at this connector-level and sent to the *Logging Manager* which is a web service responsible of saving data in a log. The *Diagnostic Engine* questions periodically the *Logging Manager*, analyzes statistically QoS values (Messages *ReqMes* and *RespMes*), and sends alarms and diagnostic reports (Message *Report*) to the *Reconfiguration Decision Planner*. When a QoS degradation is detected, the *Reconfiguration Decision Planner* plans a reconfiguration and solicits *REC* for enforcement (Message *Decision*). For example, the *Reconfiguration Decision Planner* can ask for leaving *WS Provider 1* and binding requesters to *WS Provider 2*. Consequently, requests will be routed to *WS Provider 2* instead of *WS Provider 1*.

3 QoS Monitoring Framework

3.1 Monitoring Approach

Our Monitoring approach is based on *Monitor* which is a software entity used to intercept and to enrich SOAP messages with QoS information. A SOAP message is encapsulated in a SOAP Envelop which is divided into two parts: The Header and the Body. The SOAP Body element provides a mechanism for exchanging mandatory information like method name, parameters, and invocation result while the SOAP Header allows extension of SOAP messages [2]. The *Monitor* intercepts a SOAP message and enriches its Header with QoS information.

In the following:

- *t1* represents the time at which the request has been issued by the service requester. It denotes the value of *QoSP1*,

- *t2* represents the time at which the request has been received by the service provider. It denotes the value

of *QoSP2*,

- *t3* represents the time at which the response has been issued by the service provider. It denotes the value of *QoSP3*, and

- *t4* represents the time at which the response has been received by the service requester. It denotes the value of *QoSP4*.

We interrogate the log and measure QoS values, namely: *Execution Time*, *Communication Time*, *Response Time*, *Throughput*, *Availability* and *Scalability*, according to formulas shown below:

Execution Time: The time spent by the service to execute a request; $T_{exec} = t3 - t2$

Response Time: The time from sending a request until receiving a response; $T_{resp} = t4 - t1$

Communication Time: The transport time of the request and the response; $T_{comm} = T_{resp} - T_{exec}$

Throughput: The number of requests served in a given period [4] which is calculated through this formulas;
Throughput = Number of requests/period of time

Availability: This parameter is related to the number of failures of a service in a time interval [4]. It is calculated through this formulas:
Availability = Number of successful responses/Total number of requests)

Scalability: A web service that is scalable, has the ability to not get overloaded by a massive number of parallel requests [6] (see table 3).

3.2 Conference Management System

The Cooperative Management System is concerned with a multi-services application involving massively cooperating web services. Its architecture aims to ensure data exchange flexibility between system components. It includes three tiers composed of the following components:

Requesters: They are composed of system actors namely: administrators, authors, reviewers and chairmen.

The self-healing components: They manage QoS degradation between each pair of requester/provider.

Web services: They include requester-side and provider-side web services. The requester-side web services are used by requesters to dynamically explore and invoke specific services. Provider-side web services offer functionalities related to cooperative reviewing.

The cooperative reviewing process starts by searching a suitable conference for authors. So, they send requests to the *ConfSearch* web service across the bus looking for appropriate conferences (topics, publisher, deadline, etc.). In this paper, we monitor QoS of the web service based Conference Management System. We interest mainly in the measurement and evaluation of QoS presented in section 3.1 for the *ConfSearch* web service.

4 Experimental Environment

To perform QoS tests of the *ConfSearch* web service, we used the French grid5000. The deployment architecture is shown in figure 2. The grid5000 is composed of several nodes operating with fedora linux (see table 1).

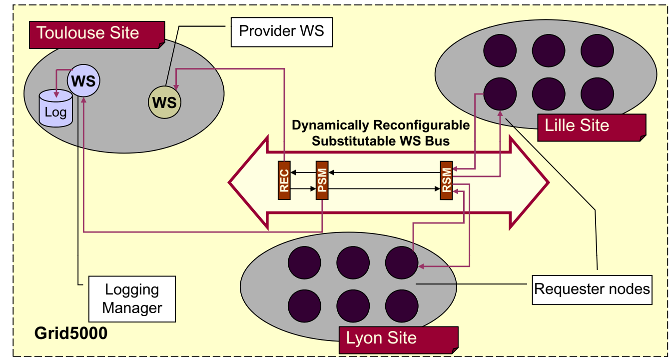

Figure 2. Infrastructure of experiments

The network configuration for the experimental environment is illustrated in table 1. We reserved nodes located in different sites. We run two nodes for servers. The first for CMS web services and the second for the Logging Manager web service. We reserve and run respectively 1, 3, 5, 10, 25, 50, 75, 100, 200, 350, and 500 requesters[1]. We use Apache Tomcat5.5 as a web server, Axis1.4 as a SOAP engine, Java1.5 as a programming language, and MySQL5 as a database management system.

Using unix shell scripts, we can build and run multiple requesters at the same time. Each requester sends SOAP requests to the *ConfSearch* web service. The requester continuously invokes the services during 10 minutes. The experiments are carried out more than 10 times. Each request will be intercepted four times in order to enrich it with QoS values. Before the response reaches the requester, the *RSM* runs a thread which logs QoS values. Based on formulas shown in section 4, we interrogate log and compute the response and execution times, as well as availability and throughput.

Table 2 shows experiment results of the *ConfSearch* web service. The first line shows that 1 requester invokes the

[1] Experiment data are available at
http://www.laas.fr/~khalil/TOOLS/QoS-4-SHWS/index.html

Site Name	Toulouse	Sophia		Bordeaux	Lyon
		Azur (cluster1)	Helios (cluster2)		
Model	Sun Fire V20z	IBM eServer 325	Sun Fire X4100	IBM eServer 325	IBM eServer 325
CPU	AMD Opteron 248 2.2 GHz (dual core)	AMD Opteron 246 2.0 GHz (dual core)	AMD Opteron 275 2.2 GHz (dual core)	AMD Opteron 248 2,2 GHz (dual core)	AMD Opteron 246 2.0 GHz (dual core)
Memory	2 GB	2 GB	4 GB	2 GB	2 GB
Network speed	Gigabit Ethernet	2 x Gigabit Ethernet	4 x Gigabit Ethernet	Gigabit Ethernet	Gigabit Ethernet

Table 1. Grid5000 nodes configuration.

Requesters Number	Request Number	Succeeded Requests	Failed Requests	Experiment duration	Execution Time (ms)			Communication Time (ms)		
					Min	Max	Avg	Min	Max	Avg
1	6464	6193	271	10Min	10	222	18,163	40	1253	59,767
3	16285	15368	917	10Min	10	654	22,795	35	2447	71,895
5	18218	16903	1315	10Min	9	638	28,496	35	155917	99,683
10	29783	25528	4255	10Min	9	964	55,076	27	5454	128,297
25	35304	26337	8967	10Min	9	2989	79,059	33	5700	310,039
50	39087	25563	13524	10Min	9	4903	87,033	35	21554	737,783
75	42227	24554	17673	10Min	9	6052	95,330	31	185258	1352,367
100	43118	24380	18738	10Min	9	6021	97,900	39	211162	1780,646
200	44072	24084	19988	10Min	9	5626	116,434	34	245921	1653,905
350	44243	24869	19374	10Min	9	6271	116,985	24	489053	1735,029
500	47981	25736	22245	10Min	9	5919	117,739	24	217436	1660,850

Table 2. Performance.

ConfSearch web service for 6193 times in 10 minutes. At 500 concurrent requesters, about 50% of requests failed. We remark that the growth of requesters number leads to overload the server and to turn down the performance. However, The execution time value is monotonically increasing while the communication time is varying due to the traffic injected by other users of grid5000.

Figure 3.a displays the communication time variation according to the requester number growth. The curve keeps growing until the level of 100 requesters. After that level, it stills around 1,7 second. The communication time varies between about 100ms for 10 requesters to about 2000ms for 100 requesters. It increases highly with the number of requesters showing the importance of this parameter in response time observed by the requesters. Such information is being analyzed and modelled to support a correct monitoring and diagnosis for this application.

Figure 3.b shows the evolution of the execution time while increasing the requester number. It increases continuously from about 20ms at 1 requester to 120ms at 500 concurrent requesters. The growth of requesters number overloads the service and turns down the performance.

Figure 3.c presents the throughput variation from 1 requester to 500 requesters. It allows us to conclude that the web service can respond at the maximum of about 40 requests per second. This threshold is reached with 25 concurrent requesters and remains stable while the requesters number increases.

We have registered the number of triggered exceptions and erroneous service response. After that, we have drawn up the service availability (see figure 3.d). We point out that the 1 requester continuous invocation during 10 minutes may trigger 271 exceptions. The service responds to less than 80% of requests if the simultaneous requesters number exceeds 100. We notice that most error responses were related to ”connection refused” exceptions, which means that the application server capacity is exceeded in term of scheduling.

We classified the response time in table 3 into different intervals (in seconds): [0, 1] contains requests that took less than 1s, [1, 2] includes requests that took between 1 and 2 s, etc. We divided the number of requests of each interval by the total number of succeeded requests to get the result of the percentage in different time slots. Those percentages depict the scalability of the *ConfSearch* web service. The first three lines of table 3 show the execution result of 1, 5 and 10 concurrent clients. Approximately, 100% of requests are served in a response time less than 1 second. When we exceed 50 clients, the web service suffers from the big number of concurrent requests and slows down its response time. For instance, about 80% of requests are completed in a response time more than 1 second at 100 and 200 concurrent requesters. We notice that the performance degrades when the number of requesters growth and the availability turns down to about 20% (see figure 3.d).

In figure 4, we drew up two curves of the response time. In the first curve, the measurement is achieved with monitors. In the second one, the measurement is done in the client code and without using monitors. Less than 50 concurrent clients, both curves are similars and the load of con-

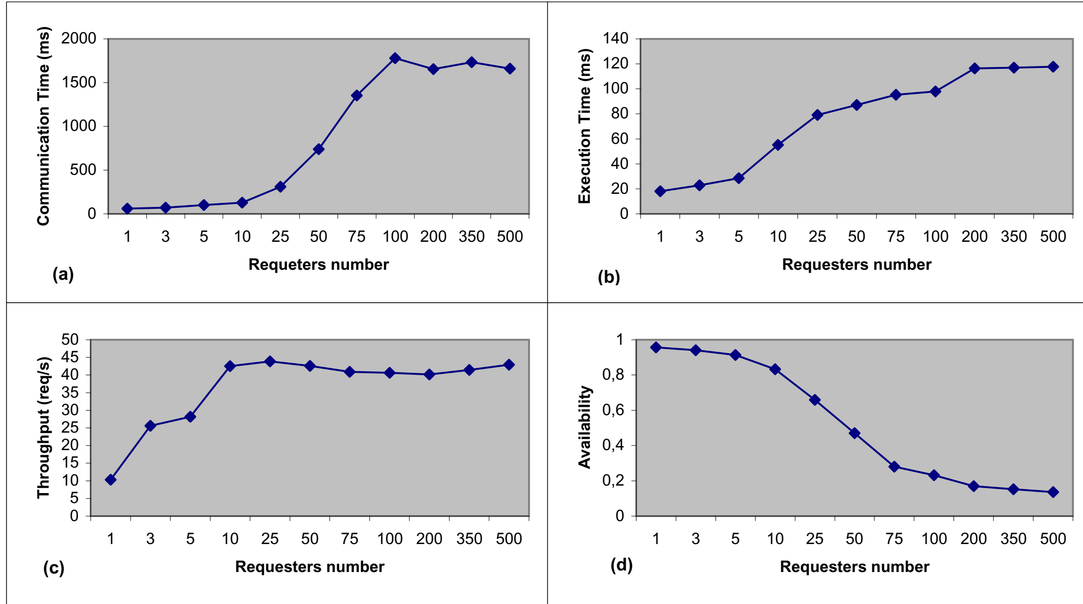

Figure 3. QoS parameters variation

Load level Requesters number	<1sec	<2sec	<3sec	<4sec	<5sec	<6sec	<7sec	<8sec	<9sec	<10sec	≥10sec
1	99,97%	0,03%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
5	99,96%	0,02%	0,02%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
10	99,24%	0,63%	0,13%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
25	93,94%	3,84%	2,07%	0,15%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%	0,00%
50	54,27%	35,90%	6,82%	2,30%	0,55%	0,15%	0,01%	0,00%	0,00%	0,00%	0,00%
75	26,98%	57,63%	9,38%	4,44%	0,81%	0,29%	0,36%	0,09%	0,02%	0,00%	0,00%
100	18,37%	50,80%	15,99%	9,25%	4,01%	1,24%	0,24%	0,06%	0,03%	0,01%	0,00%
200	18,22%	51,86%	11,52%	8,55%	6,32%	0,86%	0,56%	0,68%	0,44%	0,30%	0,69%
350	11,98%	50,60%	22,48%	7,97%	2,72%	1,02%	0,54%	0,60%	0,41%	0,53%	1,15%
500	11,06%	45,43%	24,93%	6,01%	2,76%	1,46%	0,34%	0,25%	0,08%	0,09%	7,59%

Table 3. Scalability of the *ConfSearch* web service.

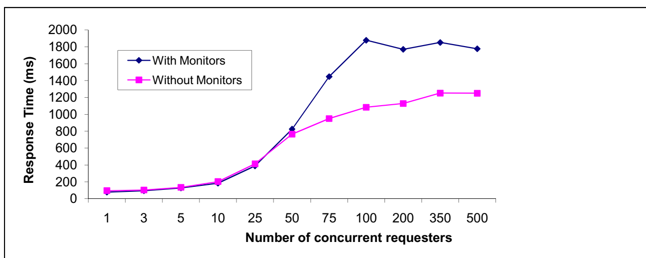

Figure 4. Load of connectors

nectors is unimportant and about zero. The delay comes to half second when we exceed 50 concurrent clients. This implies that our architecture is very suitable for a service invoked by less than 50 concurrent requesters at the same time and we have to take into account the added load when we exceed 50 concurrent requesters.

5 Related Work

El Saddik [7] addresses the measurement of the scalability and the performance of a web services based e-learning system. He carries out experiments on web service based e-learning system under LAN and DSL environment. He uses multi-clients system simulator which runs concurrent threads requesters. El Saddik interprets collected monitoring data. As a conclusion, he suggests a proxy-based approach for scheduling a massive flow of concurrent requests. But, this delays the problem from the server level to the proxy level.

		[8]	[7]	[5]	[6]	Our Work
Measured QoS	Response Time	x	x	x	x	x
	Throughput	x		x		x
	Scalability		x			x
	Availability				x	x
Technics		SOAP Engine Library modification	Use TestMaker tool to run concurrent clients	Analyse IP/TCP & HTTP protocols	Aspect Oriented Programming	SOAP Message Interception
web service		(i) Google web API, (ii) Amazon Box, (iii) webserviceX.net	Own developed service : e-learning services	Own developed service: not provided	(i) Google web API, (ii) CaribbeanT, (iii) Zip2Geo	Own developed service: Reviewing process services
Deployment		Cable connection + Internet ADSL 512Kb/s	LAN 100 Mb/s + Internet ADSL 960Kb/s	LAN 100 Mb/s		Grid5000
Points in favour		Automatic measurement	Use of two connection methods	No modification of client and service code	Separate Measurement from client code	Large-scale experiment
Points against		Implementation dependency	The clients are run from only one machine	Great CPU load	Implementation dependency	Grid is closed, no connection to google, etc.

Table 4. Synthesis of the related work

The work proposed in [5] presents an approach for monitoring performance across network layers as HTTP, TCP, and IP. It aims to detect faults early and reconfigures the system at real time while minimizing the substitution cost. But, the parsing of many layers takes enough time and consumes resources which will affect the performance. In addition, the experiment is fulfilled only under two nodes which will not reflect the behavior of such system in a large scale use.

Authors of [8] propose a framework for QoS measurement. They extend SOAP implementation API in order to measure and log QoS parameters values. The API modification have to be done in both sides: client and provider sides. This automates the performance measurement values. Also, it allows continuously updating information about QoS of services. An experiment is achieved with available services under the net. They run about 200 requests per day during 6 days and measure only the response time. However, this approach is dependent on the SOAP implementation. The extension have to be set up on the provider SOAP implementation which is difficult.

In [6] the authors propose both an evaluation approach for QoS attributes of web service, which is service and provider independent, and a method to analyze web service interactions and extract important QoS information without any knowledge about the service implementation. They implement their monitors using the Aspect Oriented Programming (*AOP*). They alter the behavior of the code base by applying additional behavior at various join points in the program. However, the aspect language is dependent to the used programming language.

6 Conclusion

In this paper, we presented our experimental results of a large-scale web service-based application implementing the conference management and the cooperative reviewing process. This experiment aims to validate the self-healing architecture that we developed in the context of the IST WS-DIAMOND project. The performance measures how QoS vary under many load conditions, and how to prevent service from QoS degradation.

The monitors load come to 0,5 second when we exceed 50 requesters. Also, when we exceed 50 requesters, the service availability turns down to less than 50%. Consequently, we have to limit the number of concurrent requests addressed to a service to less than 50 requesters.

References

[1] R. BenHalima, M. Jmaiel, and K. Drira. A qos-oriented reconfigurable middleware for self-healing web services. In *IEEE International Conference on Web Services (ICWS 2008)*, pages 104–111, Beijing (Chine), 2008. IEEE CS.

[2] M. Gudgin, M. Hadley, J.-J. Moreau, and H. F. Nielsen. *Simple Object Access Protocol (SOAP)*. W3C, http://www.w3.org/TR/2001/WD-soap12-20010709/, 2001.

[3] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[4] C. Patel, K. Supekar, and Y. Lee. Provisioning resilient, adaptive web services-based workflow: A semantic modeling approach. In *Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, page 480. IEEE CS, 2004.

[5] N. Repp, R. Berbner, O. Heckmann, and R. Steinmetz. A cross-layer approach to performance monitoring of web services. In *Proceedings of the Workshop on Emerging Web Services Technology*. CEUR-WS, Dec 2006.

[6] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping performance and dependability attributes ofweb services. In *Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, pages 205–212. IEEE CS, 2006.

[7] A. E. Saddik. Performance measurements of web services-based applications. *IEEE Transactions on Instrumentation and Measurement*, 55(5):1599–1605, October 2006.

[8] N. Thio and S. Karunasekera. Automatic measurement of a qos metric for web service recommendation. In *Proceedings of the Australian conference on Software Engineering (ASWEC'05)*, pages 202–211. IEEE CS, 2005.