Non-intrusive QoS Monitoring and Analysis for Self-Healing Web Services

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

Abstract

Monitoring and analysis of QoS are crucial steps for the provisioning of self-healing web services and for managing web service-based distributed interactive applications. Dealing with these issues becomes even more challenging when applications are dynamically built by composition of distributed services involving different service providers. In this case, assuming access to the internal logic and its implementation within the composed web services is not realistic. In this paper, we propose an architectural framework for monitoring and analysis of QoS driven by models for QoS analysis. This framework has been implemented and experimented for the web service technology within the European WS-DIAMOND [1] project. We consider the general context where only SOAP messages between web services are monitored. The main novelty of our approach is, on the one hand, to provide a generic application-independent framework. On the other hand, we provide models allowing QoS deficiencies to be detected and considered as an indicator of the health degradation of the monitored web services.

1 Introduction

Building distributed applications by dynamically selecting and connecting web services constitutes a powerful adaptation and repair mechanism. This also leads to complex composite systems where design-time requirement analysis solutions are no more sufficient. Additional management is needed not only for deciding about the capability of a given service to satisfy a given set of requirements, but also for assessing continuously this capability during the exploitation of the services. This relies on observing and analyzing the behavior of the service. It is useful for providing self-healing and self-optimizing web services and associated applications as addressed by autonomic computing and communication. Both functional and non functional properties may be analyzed through observation of exchanged messages between the interconnected web services. For QoS-like properties, analysis may rely on a standard SLA verification. Observing mismatches is targeted, and exploited for a number of purposes such as billing negotiation, or future selection of services.

Assuming the existence of a pre-defined SLA may not apply in practice for all situations, such as free or informal cooperation between web services. SLA may not be known or defined in many situations. In these situations observing QoS parameters such as response and execution times may be analyzed by run-time comparison with similar values. When SLA is not predefined, analysis may be conducted following a collaborative technique by comparing, during its exploitation, the QoS parameters of a given service to those of services of the same class, obtained from past or current observations. This is the approach we adopted in the WS-DIAMOND project for providing self-healing solution for web service-based distributed applications. In this project, both functional and QoS properties are managed. The functional-related analysis is implemented for monitoring a given process execution, providing the so-called instance-level self-healing. The QoS-related analysis is implemented for monitoring a multiple process executions, providing the so-called class-level self-healing. QoS parameters values are analyzed in their trends as indicators of a predictable degradation of the service health. For this purpose, we implemented a monitoring framework and defined a measurement approach for QoS parameters analysis. Our approach is application-independent and is applicable for any deployment context, both for requester and provider sides.

No assumption is required on the internal logic and the implementation details of web services. We rely on SOAP-level interceptors that may be deployed on requester-side only when access policies restrict access to the provider-side.

[1] Web Services - DIAgnosability, Monitoring and Diagnosis

The analysis and diagnosis accuracy may require information about the global architecture when dealing with orchestration or choreography between several services cooperating to provide a common global function. This is reasonable since it may be deduced by analyzing the business process of the implemented application.

In this paper, we present a self-healing framework able to manage web service-based distributed interactive applications. Our framework focuses on QoS monitoring and uses models for QoS analysis. It considers the communication level monitoring while intercepting exchanged SOAP messages and extending them with QoS parameter values. It is achieved using dynamically deployed handlers. The analysis of logged QoS parameter values allows the detection of QoS deficiencies and the identification of the deficient source. This is achieved based on statistical functions and time-related constraints which represent and indicator of the health degradation of the monitored web services.

This paper is organized as follows. Section 2 presents the elaborated models for monitoring and QoS degradation detecting. Section 3 focuses on the analysis of the degradation source. Section 4 details the proposed architectural framework within the FoodShop application. Section 5 discusses related work. The last section concludes the paper.

2 Models for Monitoring and Detection

Monitoring software applications aims to observe their constituting components to estimate the current health level. We introduce, in this section, a QoS-driven approach to achieve this task for web service-oriented systems. It separates clearly the business logic of a web service from its monitoring functionality. In addition, we believe that laying on QoS characteristics observations is an efficient way to predict and prevent service breakdown. Indeed, a continuous increasing of the response time or a continuous decreasing of admission rate is a significant indicator to an imminent service deny. Hence, we monitor the evolution of a given QoS characteristic more than its absolute values. QoS values such as response time may differ from one context to another (e.g. deployment machine, available network bandwidth, etc.) while the evolution of these values within the same context is the real indicator of the service health. The general approach involves two complementary aspects. On the one hand, we use statistical functions (mean, max, min and standard deviation[2]) to have reference values related to the normal functioning of the system. And on the other hand, we use patterns, which are sets of events interlinked by time constraints, to monitor the evolution of these QoS values.

[2]The standard deviation of a set of values distribution is a measure of the spread of its values.

Several QoS parameters may be considered, such as response time [8, 9], throughput [6, 10] and availability [6, 8]. These QoS parameters are measured while considering the following four time values; $t1$: the time at which the request has been issued by the service requester, $t2$: the time at which the request has been received by the service provider, $t3$: the time at which the response has been issued by the service provider, and $t4$: the time at which the response has been received by the service requester.

The considered QoS parameters are: (1) The *Response Time*: defined as the time elapsed between sending a request and receiving its response; $Tresp = t4 - t1$, (2) The *Execution Time*: defined as the time elapsed for processing a request; $Texec = t3 - t2$, (3) The *Communication Time*: defined as the round trip time of a request and its response; $Tcomm = Tresp - Texec$, (4) The *Throughput*: defined as the amount of requests that can be processed in a specified period of time; $Throughput = Number\ of\ requests/period\ of\ time$, and (5) The *Availability*: defined as the probability that the service is accessible; $Availability = Number\ of\ successful\ executions/Total\ number\ of\ invocations$.

Our approach is proactive. It is based on observing the evolution of runtime computed QoS values to detect QoS degradation considered as the symptom of an imminent deficiency. To take into account a reference behavior, we use pre-computed and/or on-the-fly-computed statistical indicators. Considered events may correspond for instance to detect or not that a measured QoS value exceeds a given threshold. Time constraints correspond mainly to the occurrence or not of a given event within a given time lapse.

Monitoring QoS aims to evaluating the health of a given service and not only a specific interaction within a given conversation. The degradation detection scope includes interactions between all requesters and providers. In our point of view, observing *N* response time increases, when dealing with requests from *M* distinct requesters, is considered as a QoS degradation in the same way as for *N* response time increases when dealing with *M* requests from the same requester. *N* is related to the tolerated threshold of acceptable degradation of each QoS parameter. For instance, the threshold = the QoS parameter average (*Avg*) + a tolerated delay (*TD*) and *N=3* in order to raise degradation alarms related to *Tresp*.

3 Models for Analysis

The analysis task lays on the monitoring and the detection of deficiency patterns. Indeed, QoS degradation may have several sources and may be triggered by one or many services of the application. For implementing effective self-healing, we need to **locate** the deficient services and **reason** about the degradation source. For instance, combining dif-

ferent QoS parameter values such as response time and execution time allows discriminating network and processing deficiencies while reasoning about architectural dependencies allows eliminating QoS degradation propagation that are irrelevant for repair.

Let's consider a pair $\langle$ provider, requester $\rangle$ for which alarms revealing QoS degradation on $Tresp$ are raised, as shown in Table 1 at line 3. This algorithm discriminates between network (*Communication*) and processing (*Execution*) deficiencies. Three cases are distinguished:

```
1  LocateDegradation(Texec,AvgTexec,TDTexec,Tresp,AvgTresp,TDTresp)
2  begin
3      if (Tresp > AvgTresp + TDTresp) then
4          if (Texec ≤ AvgTexec + TDTexec) then
5              Degradation_levels="Communication";
6          else  Delay_Texec = Texec – (AvgTexec+ TDTexec);
7                if (Tresp – Delay_Texec ≤ AvgTresp + TDTresp) then
8                    Degradation_levels="Execution";
9                else
10                   Degradation_levels="Execution&Communication";
11               endif
12         endif
13     enif
14 end
```

Table 1. The discrimination algorithm

In the first case (lines 4,5), the $Texec$ value does not exceed the max acceptable value (*Average + Tolerated Delay*). Since the response time is composed of execution and communication times, we deduce that the degradation is located at the network level. In the second case (lines 6,7,8), only the $Texec$ exceeds the max value and its delay ($Delay_{Texec}$) is the origin of the $Tresp$ degradation raise. The degradation comes from the processing level. In the third case (lines 9,10), both communication and execution times exceed the max acceptable values and the degradation is at both levels: processing and network.

After locating the degradation level (for instance in execution resource level), we start the reasoning about its source. For this purpose, let's consider the scenario of interlocked communication illustrated in Figure 1. WS2 execution time exceeds the max acceptable value. It is deficient. Monitoring and detection mechanisms related to service WS1 also detects execution time degradation due to the propagation phenomena. Not detecting that the second deficiency is a simple propagation of the first one may lead to useless reconfiguration actions.

As illustrated in Figure 1, the requester's message is processed by WS1, which calls WS2 to achieve a part of the required task. $Texec_{WS1}$ represents the execution time of the first pair Requester/WS1 and $Texec_{WS2}$ represents the execution time of the second pair WS1 (as requester)/WS2.

WS2 generates an important delay ($Delay_{WS2}$) during

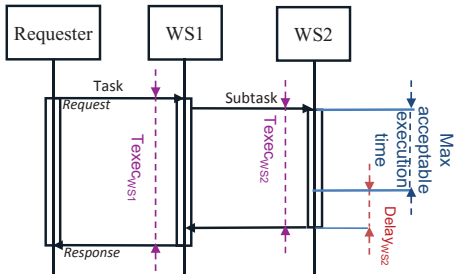

Figure 1. QoS degradation propagation

the processing of each request leading to a high overhead for both $Texec_{WS1}$ and $Texec_{WS2}$ values. The two detection processes, related to $Texec_{WS1}$ and $Texec_{WS2}$ trigger alarms.

In the case of naive analysis, two independent analysis processes are considered. The first is related to WS2 web service. It compares the response time and the communication time with the max acceptable values. It deduces that the problem comes from the processing level. It decides, for instance, to substitute WS2 by an equivalent web service which provides the same WSDL interface. Similarly, the local analysis related to WS1 also detects a QoS degradation. It decides also to substitute WS1 by an equivalent web service.

When the analysis is made locally and the degradation of WS1 and WS2 is considered separately, each analysis leads to the verdict of a local deficiency:

$Local\_Analysis(WS1, degradation) \Rightarrow WS1\ deficiency$

and

$Local\_Analysis(WS2, degradation) \Rightarrow WS2\ deficiency.$

When this analysis verdict is handled by repair functionalities, reconfiguration actions would substitute each one of the two services:

$Substitute(WS1, WS1')$, where WS1' is equivalent to WS1.

and

$Substitute(WS2, WS2')$, where WS2' is equivalent to WS2.

When considering global architectural dependencies, analysis is more accurate. It makes possible to identify that only WS2 is the source of degradation. Detected degradation of WS1 is correctly analyzed as a propagation manifestation.

$$Global\_Analysis(WS1, WS2, degradation, degradation) \equiv$$
$$Local\_Analysis(WS1, degradation) \wedge$$
$$Local\_Analysis(WS2, degradation) \wedge$$
$$(Texec_{WS1} - Delay_{WS2} \leq AvgTexec_{WS1} + TDTexec_{WS1})$$
$$\Rightarrow WS2\ deficiency \wedge WS1\ uses\ a\ deficient\ WS$$

With WS1 degradation is due to degradation propagation.

The corresponding reconfiguration sequence is more efficient and requires only substituting WS2:

$Substitute(WS2, WS2')$ where WS2' is equivalent to WS2.

Different analysis situations may be distinguished as follows:

1- First case: WS2 responses come with delay and WS1 responses come with delay after eliminating the WS2's delay propagation.
Both services are deficient. In this case, the global analysis is equivalent to the local analysis of WS1 and WS2. Both services have to be substituted.

$$Global\_Analysis(WS1, WS2, degradation, degradation) \equiv$$
$$Local\_Analysis(WS1, degradation) \wedge$$
$$Local\_Analysis(WS2, degradation) \wedge$$
$$(Texec_{WS1} - Delay_{WS2} \geq AvgTexec_{WS1} + TDTexec_{WS1})$$
$$\Rightarrow WS1\ deficiency \wedge WS2\ deficiency$$

2- Second case: WS2 responses come with delay and if we eliminate the WS2's propagated delay, WS1 responses would not come with delay.
Both services seem to be deficient, but the WS2 is the source of degradation, and the delay engendered by this degradation ($Delay_{WS2}$) propagates and affects the WS1. The global analysis identifies the degradation source, and requests for WS2 substitution.

$$Global\_Analysis(WS1, WS2, degradation, degradation) \equiv$$
$$Local\_Analysis(WS1, degradation) \wedge$$
$$Local\_Analysis(WS2, degradation) \wedge$$
$$(Texec_{WS1} - Delay_{WS2} \leq AvgTexec_{WS1} + TDTexec_{WS1})$$
$$\Rightarrow WS2\ deficiency \wedge WS1\ uses\ a\ deficient\ WS$$

3- Third case: WS2 responses come with delay and not WS1 responses.
Only the WS2 web service seems to be deficient, and the high speed of WS1 execution absorbs the WS2's Delay ($Delay_{WS2}$).

$$Global\_Analysis(WS1, WS2, \neg deficient, deficient) \equiv$$
$$Local\_Analysis(WS1, \neg degradation) \wedge$$
$$Local\_Analysis(WS2, degradation)$$
$$\Rightarrow WS2\ deficiency$$

4- Fourth case: WS1 responses come with delay and not WS2 responses.
Only the WS1 web service is degraded.

$$Global\_Analysis(WS1, WS2, deficient, \neg deficient) \equiv$$
$$Local\_Analysis(WS1, degradation) \wedge$$
$$Local\_Analysis(WS2, \neg degradation)$$
$$\Rightarrow WS1\ deficiency$$

4 Implementation

4.1 Architectural Framework

We implemented a self-healing architecture (QoS-SHA) in the context of the WS-DIAMOND project. Four main components compose this architecture [3]:

-*The Monitoring component* : It includes observing and storing relevant QoS parameter values entities using two monitors (Requester-Side Monitor: in short *ReqSideMon*, and Provider-Side Monitor: in short *ProvSideMon*) and a *Logging Manager*.

-*The Detection component* : It inspects the service behavior and detects QoS degradation.

-*The Analysis component* : It reasons about degradation based on QoS values stored by the monitoring and identifies the deficiency source.

-*The Repair component*: It switches requesters to substitutable providers using a dynamic binding connector.

The *Monitoring* component intercepts request/response messages and extends them with metadata describing the involved QoS parameters and the related values obtained at runtime. These parameters may need to be processed on the provider side (as execution time), or on the requester side (as response time), or on both sides (as communication time).

In a local context, we deal separately with each provider and its requesters. For each web service provider, we deploy a monitor and for each requester, we deploy a monitor per web service. Additional components for logging, detecting and analyzing QoS degradation are implemented as web services. We deploy an instance of these components for each web service provider and its requesters. In a global context, we need to connect the components implementing the analysis in order to exchange the pertinent information. This allows the identification of the source of QoS degradation through the whole set of involved web services, on the basis of the received alarms from the distributed detection components.

4.2 Application

We consider, here, the instantiation of the QoS-SHA to the FoodShop scenario of the WS-DIAMOND project.

The FoodShop example is concerned with a web service-based company that sells and delivers food. The company has an online *Shop*, several warehouses (*WH1, ..., WHn*) responsible for stocking imperishable goods. Customers (*C1, ..., Ck*) interact with the Shop in order to make their orders, pay the bills and receive their goods. In case of perishable items, that cannot be stocked, or in case of out-of-stock items, the warehouses must interact with several suppliers (*SUP1, ..., SUPm*).

Figure 2 shows the QoS-SHA deployment details between each pair of requester/provider from the three considered actors: the FoodShop, *Shop*, the warehouse, *WH1*, and the supplier, *SUP1*. Grouping the analysis WSs as illustrated by the dashed box of Figure 2, allows comparing

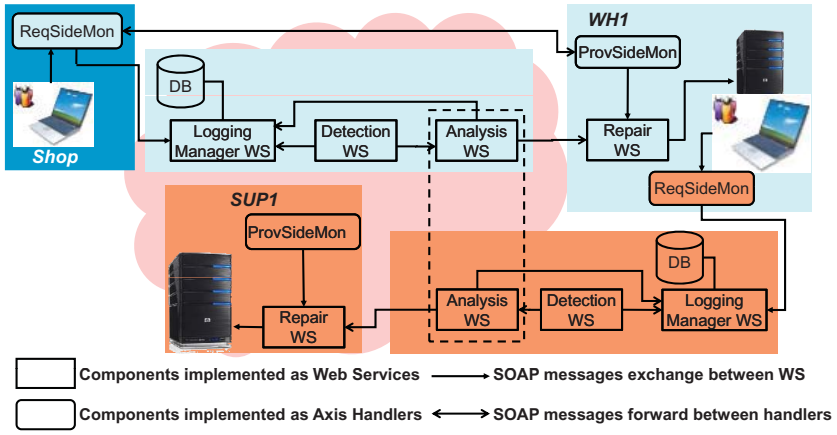

Figure 2. Details of QoS-SHA applied to the FoodShop scenario

execution time of the two involved web services, *WH1* and *SUP1*. This makes possible to diagnosis correctly the QoS degradation and to decide, for instance, substituting only *SUP1*, considered as the only deficient web service.

More precisely, we consider the global QoS-related dependencies between these three services showing the importance of the global monitoring and analysis for efficient reconfiguration actions. In the first phase, these dependencies are given as assumptions for the *Analysis* web services. This requires the knowledge of the all possible interactions between web services which are not easy to fix. In the second phase, we used WS-Addressing in order to associate together requests of both synchronous and asynchronous invocations (using the message headers *MessageId* and *RelatesTo*) and to handle dependent services (using the message header *Source*). Doing so, we deduced automatically the structural dependencies from the monitoring data.

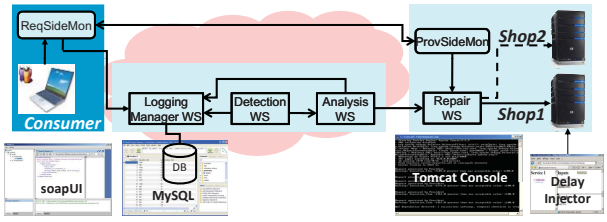

Figure 3. The FoodShop prototype

Figure 3 details the developed prototype[3] of the FoodShop application. We consider, on the left, the consumer acting as a requester of the *Shop1* web service which is on the right side. Between the requester and the provider, we deploy the QoS-SHA. We use a *Delay Injector* to simulate QoS degradation. As a result, the *Detection* and *Analysis* web services detect and identify the degradation. The *Repair* web service asked for recovery, substitutes *Shop1* by *Shop2* in a seamless way to the requester and the provider. We use Apache Tomcat5.9 as web server, Axis1.4 as web service container, ActiveBPEL2.1 as BPEL engine, soapUI1.5 as a client, MySQL5 as database management system, and Java as programming language.

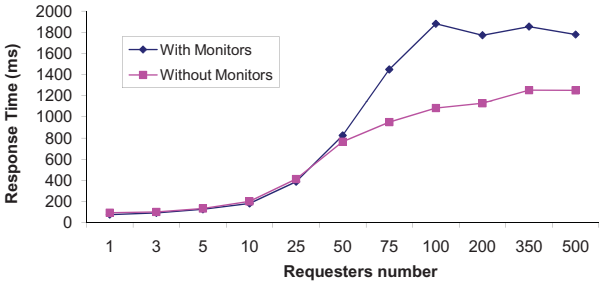

Figure 4. Overload of monitoring

In order to estimate the monitor overload, we conduct a large scale experiments under the gird5000 to measure the response time of web services while varying the requesters number from 1 to 500. We obtained the two curves shown in Figure 4. In the first, the monitoring is achieved using the *ReqSideMon* and the *ProvSideMon* (monitoring components). In the second, the measurement is done in the client code and without using monitors. We can see for instance that for less than 50 concurrent clients, both curves are similar and the overload of monitors is negligible. For the largest requesters number (500), the overload is smaller than 0.5s.

[3]Demonstration is available at http://www.laas.fr/ khalil/Video.html

5 Related Work

In this section, we first present different QoS monitoring approaches compared to our work. However, the monitoring may be performed differently at three levels, namely: service level [5, 8], communication level [3, 10, 7] (as our approach) and orchestration level [2, 4].

The service level monitoring considers the basic monitoring approach. It inserts the monitoring code within the web service requesters/providers code. Such monitoring may be achieved while inserting directly, for instance, a timer within the client code [5], or encapsulating it in an aspect, which is merged into the functional code thanks to the Aspect Oriented Programming (AOP) [8].

The communication level monitoring intercepts exchanged messages between web service providers and requesters. Such approach targets only the interactions where a given service is involved and do not need access to its internal state which is generally hidden due to security reasons. This approach may be applied at the SOAP level while using standard XML parsing libraries [3, 10], or at the HTTP level while using proxies [7].

The orchestration level monitoring supervises orchestrated services as BPEL using handlers provided by the orchestration engine such as Active BPEL. Such approach may observe its behavior by intercepting the input/output messages that are received/sent by the processes [2]. The approach presented in [4] monitors behaviors of orchestrated web services with respect to an already expected behavior which is specified formally with algebraic notations and saved in a registry. The monitoring is achieved using the AOP inside the BPEL engine.

The analysis of a web service behavior may be performed at instance level that deals with the execution of a single instance [1] from a specific requester, either at class level, that considers all instances like our work. The work in [2] deals with boolean and time related properties at instance and class levels.

In [1], authors propose an instance level and hierarchical analysis for complex services. They deploy local analyzer on each basic service, and a global one associated with the workflow that collects local analysis and reasons about faults in current running activity. However, web services are usually multi-providers, which do not allow modifying their services in order to interact with an external component, like the local analyzer. But, in our approach, no assumptions are required.

6 Conclusion

In this paper, we presented an application-independent approach to monitor, detect and analyze QoS degradation for web services. This constitutes one of the two parts of the self-healing framework developed in the context of the IST WS-DIAMOND project. The presented approach has been implemented and experimented under Axis and Apache WS container. It acts at the communication level, with no intrusion in the application code, and achieves detection and analysis based-on statistical functions and time-related constraints.

Large scale experiments have been performed to measure the QoS monitoring overload. We are now working on improving the repair enactment in order to reduce the unavailability of the service during reconfiguration.

References

[1] L. Ardissono, L. Console, A. Goy, G. Petrone, C. Picardi, M. Segnan, and D. T. Dupre. Enhancing web services with diagnostic capabilities. In *ECOWS '05: Proceedings of the Third European Conference on Web Services*, page 182, Washington, DC, USA, 2005. IEEE Computer Society.

[2] F. Barbon, P. Traverso, M. Pistore, and M. Trainotti. Run-time monitoring of instances and classes of web service compositions. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services*, pages 63–71, Washington, DC, USA, 2006. IEEE Computer Society.

[3] R. BenHalima, M. Jmaiel, and K. Drira. A qos-driven reconfiguration management system extending web services with self-healing properties. In *16th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises WETICE*, pages 339–344, Paris, France, 18-20 June 2007. IEEE Computer Society.

[4] D. Bianculli and C. Ghezzi. Monitoring conversational web services. In *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*, pages 15–21, New York, NY, USA, 2007. ACM.

[5] A. Mani and A. Nagarajan. Understanding quality of service for web services. Technical report, IBM DeveloperWorks, www-106.ibm.com/developerworks/webservices /library/ws-quality.html, Jan 2002.

[6] D. A. Menascé. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, 2002.

[7] 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.

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

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

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