Providing Predictive Self-Healing for Web Services: A QoS Monitoring and Analysis-based Approach
Riadh Ben Halima, Mohammed Karim Guennoun, Khalil Drira, Mohamed Jmaiel

To cite this version:

HAL Id: halshs-00367395
https://halshs.archives-ouvertes.fr/halshs-00367395
Submitted on 11 Mar 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 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.

Keywords: QoS, Self-healing, Web service, Monitoring, Analysis.

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. It relies on observing and analyzing the behavior of the service. This 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 these 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 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 the requester and the 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 the requester-side only when access policies restrict access to the provider-side. 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 an 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 for QoS degradation detection. Section 3 presents the analysis algorithm 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 the admission rate is a significant indicator to a future denial of service. 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) to have reference values related to the normal functioning of the system. And on the other hand, we use temporal chronicles to monitor the evolution of QoS values in order to avoid considering transient QoS degradation.

A temporal chronicle or pattern is a set of on-off events, interlinked by time constraints, and whose occurrence depends on the context [5]. 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.

Several QoS parameters may be considered, such as response time [11, 12], throughput [7, 13] and accuracy [7, 11]. These QoS parameters are measured while considering the following four time values as shown in Figure 1; \( t_1 \): the time at which the request has been issued by the service requestor, \( t_2 \): the time at which the response has been received by the service provider, \( t_3 \): the time at which the response has been issued by the service provider, and \( t_4 \): the time at which the response has been received by the service requestor.

2 The standard deviation of a set of values distribution is a measure of the spread of its values.
(Avg) and the Tolerated Delay (TD) which is proportional to the standard deviation. It can be pre-computed, on the basis of already achieved experiments, or computed on-the-fly from current measured values. N is equal to the number of successive violations leading to the service invocation failure. In order to determine a relevant value of N, we use the following expression, which corresponds to the probability of N successive TrespV less than the mean probability of the failed request measured experimentally (1-Accuracy):

\[ P[N \text{ successive } \text{Tresp}V] \leq (1 - \text{Accuracy}) \]

In addition, the probability that "the next state is TrespV", is equal to the probability of being in state TrespOk multiplied by the probability of going from this state to the state TrespV added to the probability of being in state TrespV multiplying the probability of staying in this state for the next invocation:

\[ P[\text{any state } \rightarrow \text{TrespV}] = P[\text{TrespOK} \times P[\text{TrespOK } \rightarrow \text{TrespV}]] + P[\text{TrespV}\times P[\text{TrespV } \rightarrow \text{TrespV}]] \]

The probability of obtaining N successive violations is equal to the probability to reach a TrespV state, after leaving any state, multiplied by the probability of staying in TrespV state (N-1) times:

\[ P[N \text{ successive } \text{Tresp}V] = P[\text{any state } \rightarrow \text{TrespV}] \times P[\text{TrespV } \rightarrow \text{TrespV}]^{N-1} \]

When replacing the probability \( P[N \text{ successive } \text{Tresp}V] \) in the first inequality, we obtain the following expression providing the lower bound for N:

\[ P[\text{any state } \rightarrow \text{TrespV}] \times P[\text{TrespV } \rightarrow \text{TrespV}]^{N-1} \leq (1 - \text{Accuracy}) \]

\[ \iff N \geq \lambda, \text{ where:} \]

\[ \lambda = 1 + \frac{\ln(1 - \text{Accuracy})}{\ln(P[\text{any state } \rightarrow \text{Violation}]/P[\text{Violation } \rightarrow \text{Violation}])} \]

When we choose an N's value greater than the smallest one (more than \( \lambda + 1 \)), we accurate the degradation detection rate.

Figure 2 presents examples of temporal chronicles involving the response time QoS characteristic. The chronicles show two ways to compute the threshold of the maximal acceptable value using either pre-computed or on-the-fly computed values. For these chronicles, considered events relate to the occurrence or not of TrespOK and TrespV events at a given moment or a given interval. The first chronicle is triggered when three occurrences of event TrespV (t1, t2 and t3 in figure 2) are observed while no occurrence of event TrespOK is observed. The second chronicle is detected when three occurrences of event TrespV (t1', t2' and t3' in figure 2) are observed in a time interval I1. The third chronicle is triggered when three successive increases of the response time (t1", t2" and t3" in figure 2) are observed.

The chronicle based on an on-the-fly computed threshold may not be significant when increasing progressively the Tresp, because it increases the acceptable value interval and affects the service level agreement. Also, the chronicle based on a pre-computed threshold does not take into account the context of the running application and the Tresp measurement may be unsatisfactory. However, combining the two chronicles is more related to the context thanks to the on-runtime measured values, and more robust while avoiding the case of progressive and continuously increasing thanks to large scale values already measured. More complex chronicles may be designed for robust reasoning about QoS degradation.

We have developed algorithms for degradation detection, which trigger alarms when N successive violations happen (represented by Max_Nbr_Succ_Violation in Table 1 and Table 2).

As illustrated in Table 1, on each measured QoS parameter value (line 6), the algorithm increases the number of received violations (Nb Received Violation, in line 8) if the new value exceeds the max acceptable value (line 7). In case of non-successive violations (line 9), the execution will jump to the initialization at line 4. Otherwise, it increments the...
number of obtained violation until triggering alarms (line 12). In this algorithm, the Average and the Tolerated Delay, are pre-computed and deduced from a large scale experiment (lines 2,3).

1 begin
2 Avg= Compute_Avg() // Constant
3 TD= Compute_Tolerated_Delay()
4 Idle: Nb_Received_Violation=0
5 loop
6 On_New_Measured_Tresp()
7 if (Tresp>Avg+TD) then
8 Nb_Received_Violation++;
9 else goto: Idle
10 endif
11 if (Nb_Received_Violation ≥ Max_Nbr_Succ_Violation) then
12 Notify_Degradation_To_Analysis_Service()
13 endif
14 endloop
end

Table 1. Degradation detection Algorithm on response time (Pre-computed average)

In the algorithm presented in Table 2, the Average and the Tolerated Delay, are computed and deduced on-the-fly (lines 5,6), after each new measured QoS parameter value (line 4). Non-successive violations re-initialize the algorithm (line 9). In case of degradation, alarms are sent to the analysis service (line 12).

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 to reason about the degradation source. For instance, combining different 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 of provider/requester for which alarms revealing QoS degradation on Tresp are raised, as shown in Table 3 at line 3. This algorithm discriminates between network (Communication) and processing (Execution) deficiencies. Three cases are distinguished.

In the first case (lines 3,4), the Texec value does not exceed the maximal acceptable value (Average + Tolerated Delay). Since the response time is composed of execution time and communication time, we deduce that the degradation is located at the network level. In the second case (lines 5,6,7), only the Texec exceeds the maximal value and its delay (DelayTexec) is the origin of the Tresp degradation raise. The degradation comes from the processing level. In the third case (lines 8,9), both communication time and execution time exceed the maximal acceptable values and the degradation is at both levels: processing and network.

1 begin
2 if (Tresp > AvgTresp + TD) then
3 if (Texec ≤ AvgTexec + TD) then
4 Degradation_Levels="Communication"
5 else DelayTexec= Texec – (AvgTexec + TD)
6 if (Tresp – DelayTexec ≤ AvgTresp + TD) then
7 Degradation_Levels="Execution"
8 else Degradation_Levels="Execution&Communication"
9 endif
10 endif
11 endif
12 endif
13 end

Table 2. Degradation detection Algorithm on response time (On-the-fly computed average).

Let’s consider the scenario of interlocked communication depicted in Figure 3. WS2 execution time exceeds the maximal acceptable value. It is deficient. Monitoring and detection mechanisms related to service WS1 also detects execution time degradation due to delay propagation.

For considering structural dependencies, the monitoring is targeting global QoS parameters belonging to several pairs of providers and requesters. Typically, we consider execution time related to a request sent to two different providers involved in the same orchestration or in the same choreography.

Let’s consider the scenario of interlocked communication depicted in Figure 3. WS2 execution time exceeds the maximal 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 3, the requester’s message is processed by WS1, which calls WS2 to achieve a part of the required task. TexecWS1 represents the execution time of the first pair Requester/WS1 and TexecWS2 represents the execution time of the second pair WS1 (as a requester)/WS2. WS2 generates an important delay (DelayWS2) during the processing of each request leading to a high overhead for both TexecWS1 and TexecWS2 values. The two detection processes, related to TexecWS1 and TexecWS2 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 maximal acceptable values. It deduces that the problem comes from the processing level. It decides, for instance, to substitute WS2 by an equivalent web service. Similarly, the local analysis related to WS1 also
detects a QoS degradation. It decides also to substitute WS1 by an equivalent web service.

<table>
<thead>
<tr>
<th>Requester</th>
<th>WS1</th>
<th>WS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Request</td>
<td>TexecWS1</td>
<td></td>
</tr>
<tr>
<td>Subtask</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Response</td>
<td>TexecWS2</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3. QoS degradation propagation**

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

\[
\text{Local\_Analysis}(WS1, \text{degradation}) \Rightarrow WS1 \text{ deficiency}
\]

and

\[
\text{Local\_Analysis}(WS2, \text{degradation}) \Rightarrow WS2 \text{ deficiency}
\]

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

\[
\text{Substitute}(WS1, WS1')
\]

and

\[
\text{Substitute}(WS2, WS2')
\]

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.

\[
\text{Global\_Analysis}(WS1, WS2, \text{degradation}, \text{degradation}) \equiv \text{Local\_Analysis}(WS1, \text{degradation}) \land \\
\text{Local\_Analysis}(WS2, \text{degradation}) \land \\
(Texec_{WS1} - Delay_{WS2} \leq \text{AvgTexec}_{WS1} + \text{TDTexec}_{WS1})
\]

\[
\Rightarrow WS2 \text{ deficiency} \land WS1 \text{ uses a deficient WS}
\]

With WS1 degradation is due to a degradation propagation. The corresponding reconfiguration sequence is more efficient and requires only substituting WS2:

\[
\text{Substitute}(WS2, WS2')
\]

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.

\[
\text{Global\_Analysis}(WS1, WS2, \text{degradation}, \text{degradation}) \equiv \text{Local\_Analysis}(WS1, \text{degradation}) \land \\
\text{Local\_Analysis}(WS2, \text{degradation}) \land \\
(Texec_{WS1} - Delay_{WS2} \geq \text{AvgTexec}_{WS1} + \text{TDTexec}_{WS1})
\]

\[
\Rightarrow WS1 \text{ deficiency} \land WS2 \text{ deficiency}
\]

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 (\(\text{Delay}_{WS2}\)) propagates and affects the WS1. The global analysis identifies the degradation source, and requests for WS2 substitution.

\[
\text{Global\_Analysis}(WS1, WS2, \text{degradation}, \text{degradation}) \equiv \\
\text{Local\_Analysis}(WS1, \text{degradation}) \land \\
\text{Local\_Analysis}(WS2, \text{degradation}) \land \\
(Texec_{WS1} - Delay_{WS2} \leq \text{AvgTexec}_{WS1} + \text{TDTexec}_{WS1})
\]

\[
\Rightarrow WS2 \text{ deficiency} \land WS1 \text{ uses a deficient WS}
\]

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 (\(\text{Delay}_{WS2}\)).

\[
\text{Global\_Analysis}(WS1, WS2, \neg\text{degradation}, \text{degradation}) \equiv \\
\text{Local\_Analysis}(WS1, \neg\text{degradation}) \land \\
\text{Local\_Analysis}(WS2, \text{degradation})
\]

\[
\Rightarrow WS2 \text{ deficiency}
\]

4- Fourth case: WS1 responses come with delay and not WS2 responses.

Only the WS1 web service is degraded.

\[
\text{Global\_Analysis}(WS1, WS2, \text{degradation}, \neg\text{degradation}) \equiv \\
\text{Local\_Analysis}(WS1, \text{degradation}) \land \\
\text{Local\_Analysis}(WS2, \neg\text{degradation})
\]

\[
\Rightarrow WS1 \text{ deficiency}
\]

4. Implementation

4.1 Architectural Framework

We implemented a QoS manager providing a self-healing architecture in the context of the WS-DIAMOND project. It includes four main components [3]:

- The **Monitoring component**: It includes observing and storing relevant QoS parameter values entities. It is composed of:
  - The **Requester-Side Monitor** which implements the QoS monitoring at the requester side (in short: **ReqSideMon**), and
  - The **Provider-Side Monitor** which implements the QoS monitoring at the provider side (in short: **ProvSideMon**)
  - The **Logging Manager** which manages QoS monitoring data.

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

- The **Analysis component**: It reasons about the degradation by exploiting the QoS values stored by the monitoring and identifies the deficiency source. Then, it sends the analysis to the **Repair component** for possible reconfiguration.

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

This architecture is deployed between each provider and its requesters. The **Monitoring component** is composed of several monitors; **ReqSideMon**, deployed one on each requester side and a unique monitor, **ProvSideMon**, deployed on the provider side.

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 the **execution time**), or on the requester side (as the **response time**), or on both sides (as the **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 the QoS degradation through the whole set of involved web services, on the basis of the received alarms from the distributed detection components.

Two different techniques are used to implement monitors. In the first, we used a SOAP handler within the web service container, which intercepts SOAP envelop of each request and each response [3]. In the second, we used a HTTP proxy, which intercepts HTTP messages of the exchanged data between providers and requesters [8].

### 4.2 Application

We consider, here, the instantiation of the QoS manager 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 perishable 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).

In this application, the shop acts as a requester who, for example, orders cereal and milk from the warehouse. The cereals are an imperishable product available at the warehouse, the request is served locally. The milk is a perishable product, not available locally and will be orders by the warehouse from the supplier. Hence the warehouse acts simultaneously as a provider and a requester.

Figure 4 shows the QoS manager 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 4, allows comparing 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 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 [14] 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 5 shows the QoS degradation propagation.

For instance, we consider the case of repeated violations between "ordering milk and cereals" and "receiving these products", detected in WH1 and SUP1 web services. In this situation, the first QoS manager which monitors interactions between the Shop and the WH1 and the second QoS manager which monitors between the WH1 and the SUP1 web services, detect QoS degradation (using algorithm showed in Table 2). Then, it locates the deficient level, and identifies the degradation source (see Figure 3 for more details). TexecWH1 and TexecSUP1 exceed the maximal acceptable values and the detection services trigger alarms. The QoS manager analysis services identify that the degradation is coming from TexecSUP1, and that TexecWH1 deficiency is a degradation propagation. SUP1 is substituted.

Figure 5 details the developed prototype of the FoodShop.
application. The consumer acts as a requester of the Shop web service which starts a new process. Between each requester and each provider, we deploy the QoS manager. We use a Delay Injector to simulate QoS degradations. As a result, the Detection and Analysis web services detect and identify the degradation. The Repair web service asked for recovery, substitutes SUP1 by SUP2 in a seamless way to both the requester and the provider. We use Apache Tomcat 5.9 as a web server, Axis 1.4 as a web service container, ActiveBPEL 2.1 as a BPEL engine, soapUI 1.5 as a client, MySQL 5 as a database management system, and Java as a programming language.

We computed the $N$ value (see section 2), in order to estimate after how much successive violations we trigger a substitution operation. The used values and probabilities are deduced from past execution experiments under the French Grid5000:\footnote{Demonstrator is available at http://www.laas.fr/~khalil/TOOLS/QoS-4-SHWS/Video.html}

\[P[\text{any state} \rightarrow \text{TrespV}] = 0.78 \times 0.13 + 0.23 \times 0.09 = 0.1221\]
\[P[\text{TrespOK} \rightarrow \text{TrespV}] = 0.23, \quad P[\text{TrespOK} \rightarrow \text{TrespOK}] = 0.13, \quad \text{and} \quad P[\text{TrespV} \rightarrow \text{TrespV}] = 0.09.\]
\[\Leftrightarrow 0.1221 \times 0.09 N - 1 \leq 0.04, \quad \text{With Accuracy} = 0.96.\]
\[\Leftrightarrow N \geq 2.115\]
That means, $N$ must at least be equal to 3.

**Figure 6.** Runtime recovering of degradation

Figure 6 illustrates the whole self-healing cycle applied to the Foodshop application. The QoS manager monitors response time at runtime of the SUP1. It detects degradation according the first chronicle showed in Figure 2, with $N=3$. The green curve represents the normal execution of SUP1. At the level of the point $A$, we inject degradation and we follow the behavior of QoS manager. After three successive violations, the QoS manager detects a degradation (red curve in figure 6), and reacts by substituting the SUP1 web service at point $B$. After rerouting requesters to the new supplier (SUP2), the response time regains the normal behavior (blue curve in figure 6).

In order to estimate the monitor overload, we conduct a large scale experiments under the grid5000 to measure the response time of web services while varying the requester’s number from 1 to 500. We obtained the two curves shown in Figure 7. 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 requester’s number (500), the overload is smaller than 0.5s.

**Figure 7.** Overload of monitoring

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 \[6, 11\], communication level \[3, 13, 9\] (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 \[6\], or encapsulating it in an aspect, which is merged into the functional code using the Aspect Oriented Programming (AOP) \[11\].

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 \[3, 13\], or at the HTTP level while using proxies \[9\].

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. Robinson [10] presents a goal-driven methodology for monitoring requirements using temporal logic and KAOS. It uses agents with specific patterns in order to inspect the observable system events. Contrarily to our approach, this approach is not generic for web service-based applications. It supposes that it can access to provider sites in order to deploy monitors to track web services. But, this is not realistic when involving different service providers.

6. Conclusion
In this paper, we presented an application-independent framework to monitor, detect and analyze QoS degradation for web services. Our framework relies on logging and appropriately analyzing web service conversations using SOAP messages header information. It applies to both situations where a reference model of QoS values is available and situations where such a model has to be built dynamically on the fly during the execution of the application.

Acting at the communication level by intercepting conversations allowed us to handle web services as black boxes. Two prototypes have been implemented at the SOAP and HTTP levels using service container handlers (Axis) and HTTP proxies, respectively. Our implementation exploits header information to manage both synchronous and asynchronous interactions for which QoS values are computed differently. Two application scenarios have been successfully experimented. The first scenario implements a collaborative review process using massively interacting services [3]. The second, discussed in the paper, implements a supplier chain-like scenario where several web services acts simultaneously as providers and requesters of each others with highly interlocked conversations.

Our future work includes extending and improving our framework by implementing more advanced detection and analysis algorithms that can learn from previous execution traces to choose the best decision. Our approach was successfully experimented with an orchestrated web service-based distributed application using a centralized QoS manager. Managing choreographed web services may also be implemented by a distributed deployment of replicated copies of the QoS manager. Our interceptor-based framework has also been used to experiment dynamic routing between a requester and a collection of equivalent web services providers. Managing communications at the HTTP level showed more powerfulness for handling state-sensitive cooperation scenarios. Such situations may be encountered when handling choreographed between two or more orchestrated collections of web services. In such hybrid scenarios state preservation is necessary when a new provider is used to substitute a degraded provider.

References
Author Biographies

Riadh BEN HALIMA is a PhD thesis student (software engineering) at Université Paul Sabatier (UPS) of Toulouse and Université de Sfax. The current working title of the thesis is the QoS provisioning within the cooperative application based on the service oriented architecture. He received his diploma of engineer in Computer Science in 2002 and his M.S. degree (DEA) in 2004 in Computer Science from the Université de Sfax.

Karim GUENOUN received the M.S. degree (DEA) in Computer Science from Paul Sabatier University (UPS) of Toulouse in 2002, and the Ph.D. degree in Computer Science from UPS, in 2006. Since 2008, he is assistant professor in the Hassania Engineering School of Casablanca. He is also associate Researcher at the Engineering Sciences Laboratory of Casablanca. His main fields of interest include QoS provisioning and dynamic software architecture formal description, management and verification.

Khalil DRIRA received the Engineering and M.S. (DEA) degrees in Computer Science from ENSEEIHT (INP Toulouse), in June and September 1988 respectively. He obtained the Ph.D. and HDR degrees in Computer Science from UPS , University Paul Sabatier Toulouse, in October 1992, and January 2005 respectively. He is since 1992, Chargé de Recherche, a full-time research position at the French National Center for Scientific Research (CNRS). Khalil DRIRA's research interests include formal design, implementation, testing and provisioning of distributed communicating systems and cooperative networked services. His research activity addressed and addresses different topics in this field focusing on model-based analysis and design of correctness properties including testability, robustness, adaptability and reconfiguration. He is or has been involved in several national and international projects in the field of distributed and concurrent communicating systems. He is author of more than 150 regular and invited papers in international conferences and journals. He is or has been initiator of different national and international projects and collaborations in the field of networked services and distributed and communicating systems. Khalil DRIRA has been editor of a number of proceedings, books and journal issues in these fields. More details are available on his wiki: http://www.laas.fr/~khalil/wiki.

Mohamed JMAIEL obtained his diploma of engineer in Computer Science from Kiel (Germany) University in 1992 and his Ph.D. from the Technical University of Berlin in 1996. He joined the National School of Engineers in Sfax as Assistant Professor of Computer Science in 1995. He became an Associate Professor in 1997 and full Professor in October 2003. He participated to the initiation of many graduate courses at the University of Sfax. His current research areas include software engineering of distributed systems, formal methods in model-driven architecture, component oriented development, self-adaptable and pervasive systems, autonomic middleware. He published regular and invited papers in international conferences and journals, and has co-edited four conferences proceedings and three journals special issues on these subjects.