A QoS-Oriented Reconfigurable Middleware For Self-Healing Web Services

Riadh Ben Halima [(1,2)], Khalil Drira [(1)] and Mohamed Jmaiel [(2)]
[(1)] LAAS-CNRS, Université de Toulouse, 7 avenue de Colonel Roche, 31077 Toulouse, France
[(2)] University of Sfax, National School of Engineers, B.P.W, 3038 Sfax, Tunisia

Abstract

Maintaining the Quality of Service (QoS) is important for self-healing web service-based distributed interactive applications. It requires the ability to deal with permanently changing constraints both at the communication and the execution levels. Preventing or repairing QoS degradation also requires the capacity of identifying its possible or actual sources and the capacity of reconfiguration decision and enforcement. Dealing with these issues is especially challenging for web services since the self-healing solution has to preserve the dynamic composition property and to be seamless for the service requesters, while being always usable under the different deployment constraints. In this paper, we present a self-healing middleware framework able to provide the self-healing properties for QoS management in web service-based distributed interactive applications. The framework implementation has been achieved in the context of the WS-DIAMOND project. It covers the whole cycle of adaptation management including monitoring and analysis of QoS values, and substitution-based reconfiguration.

1 Introduction

Dealing with QoS properties is important for implementing self-healing web services. The QoS parameters constitute good indicators for assessing the health state of a given application and its composing web services. Maintaining the QoS may be conducted following a reactive or a predictive self-healing policy. For both policies, this requires the ability to deal with permanently changing constraints both at the communication and the execution levels. Preventing or repairing the QoS degradation also requires the capacity of identifying its possible or actual sources and the capacity of reconfiguration decision and enforcement.

Moreover, a self-healing solution for web service-based applications has to preserve the dynamic composition property and to be seamless for the service requesters, while being always applicable under the different deployment constraints. Dealing with web services is even more complex since both synchronous and asynchronous interactions have to be handled on the requester side and on the provider side. Moreover different situations have to be distinguished for reconfiguration when the monitoring predicts or detects a QoS degradation. Applicable and efficient solutions have to consider single and composite substitutions with and without overlap of interfaces.

For implementing QoS-oriented self-healing, we developed a non-intrusive solution for observing exchanged messages between the distributed web services that compose a given application. The observed messages are extended with QoS parameters defined as metadata extending the SOAP messages headers both from the requester and the provider sides. When the accessibility constraints prevent from deploying monitors on the provider side, our solution is still applicable. In such situations, we may use the requester side or even a third party side to deploy the different monitoring components. The same property applies to the different components of our architecture. This makes it applicable to a large family of web service-based applications under different deployment constraints. Such a property has been tested within the context of the WS-DIAMOND project. Our prototype has been successfully applied to two different web service-based applications: the collaborative conference management system [2], and the FoodShop system implementing on-line management of remote interactions involving multiple requesters and multiple providers. Our reconfiguration algorithm relies on dynamic binding and provides different substitution policies.

This paper is organized as follows. In section 2, we discuss the properties of the reactive and the predictive self-healing policies. In section 3, we present a self-healing middleware framework able to provide the self-healing properties for QoS management in web service-based distributed interactive applications. The self-healing middleware covers the whole cycle of adaptation management including monitoring and analysis of QoS values, and substitution-based reconfiguration. In section 4, we describe the experience of deploying and integrating our prototype with the FoodShop application implementing a scenario of the sup-

plier chain automation category. In section 5, we discuss the related work. In section 6, we provide our concluding remarks and the future work directions.

2 The self-healing WS policies

Four main steps are distinguished in the self-healing process [2]: *Monitoring* to extract information about the system health, *Analysis* to detect possible degradations, *Diagnosis & Decision* to identify the degradation source and to plan for repair actions, and *Repair* to execute these actions.

Following a reactive self-healing policy, the monitoring services cooperate with the diagnosis services to detect service degradation and to react appropriately by repair plans.

Following a predictive self-healing policy, the monitoring services cooperate with the prognosis services to predict service degradation and to act appropriately by reconfiguration plans.

Both repair and reconfiguration plans may include adapting service composition by switching locally between different web service instances. They may also include switching remotely between distributed web services implementing the same interface. Such plans may also include activating, deactivating and dynamically deploying new instances of web services. These actions may be achieved to react to service degradation or for preventing such degradation from happening or worsening.

To handle a predicted or a detected degradation associated with a given web service, repair and reconfiguration plans may act by preventing future requesters to use the current service involved in the degradation. In some cases, decision has to deal with situations where several requesters are using the same web service or are using different web services that share communication or computation resources. Reactive repair policies and predictive reconfiguration policies may rely on dynamically binding some of the requesters to a new instance of the web service. Choosing the requesters to be bound to new requesters may rely on a classification of requesters according to different subscription contracts. The group of requesters subscribed with a low priority service class will be penalized. The group of requesters subscribed with a high priority service class will be privileged. Depending on the source of degradation, routing a requester towards a new web service instance may be more appropriate than maintaining its association with the current instance. For applications that have no hierarchies of requester classes, the decision may rely on standard load balancing algorithms. In some cases, the analysis of logged monitored values may help improving such a decision.

More complex adaptation actions may be planned, when it is not possible or not sufficient to switch between instances of the same web service or between web services implementing the same WSDL (Web Services Description Language) interface. The repair and reconfiguration plans may act by rerouting requests to one or more different web services implementing partially or totally the given WSDL interface. The requests may be routed to a new web service whose interface offers more operations that the WSDL interface of the deficient web service. This is called substitution. Following this principle, substitution may also replace a given web services by two web services implementing, each a part of the WSDL interface.

3 The QoS-oriented self-healing middleware framework

This section presents the QoS-Oriented Self-Healing middleware (QOSH) which implements monitoring and reconfiguration functionalities, as shown in Figure 1.

3.1 The Monitoring & Analysis

The *Monitoring* module includes observing and storing relevant QoS parameters values. It acts on the communication level; it intercepts exchanged SOAP messages and extends them with QoS metadata within their correspondent parameters values. It is implemented by dynamically deployed handlers for the *Requester-Side Monitor* (*RSM*) and the *Provider-Side Monitor* (*PSM*). The *Logging Manager* saves the data in the log database.

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       SOAPEnvelop.release();
        end
    4   OnResponse( SOAPEnvelop)
        begin
    5       SOAPEnvelop. AddInHeader(QoSMetadata, QoSParamValues)
    6       SOAPEnvelop.release()
        end
endloop
```

Table 1. The Provider-Side Monitor behavior for synchronous communication

As illustrated in Table 1, the *PSM* intercepts the incoming SOAP message (line1). It adds QoS metadata (e.g. issuing request time "t2", line2) and forwards the request to be performed (line3). The response message is intercepted (line4), extended by QoS parameters values (e.g. issuing execution time "t3", line5) and released to the requester.

The *RSM* intercepts the outgoing SOAP message (line1), as described in Table 2. It adds QoS metadata (e.g. issuing request time "t1", line2) and forwards the request to the *PSM* (line3). The response message is intercepted (line4), extended by QoS parameters values (e.g. issuing response

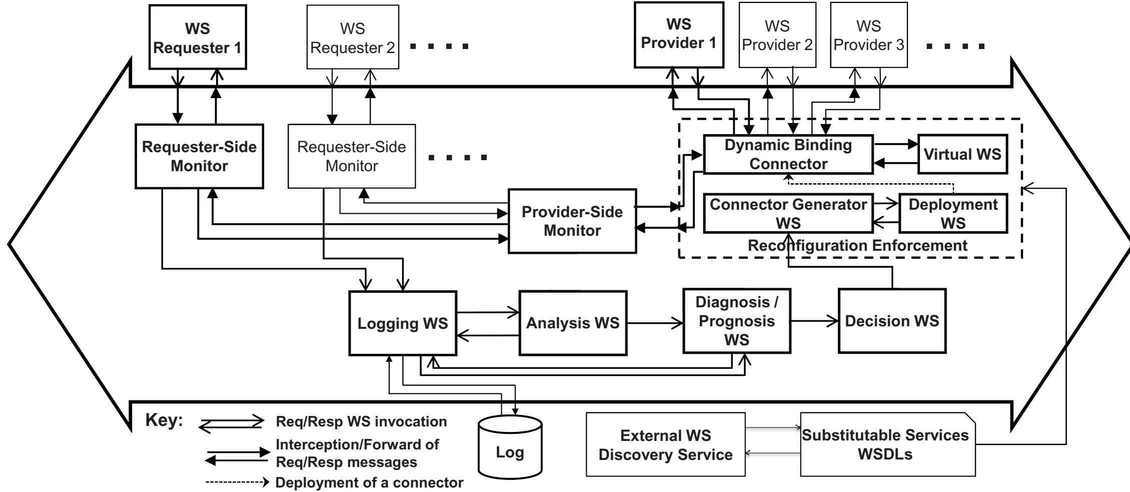

Figure 1. QoS-Oriented Reconfigurable Middleware for Substitutable WS

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       SOAPEnvelop.release();
        end
    4   OnResponse( SOAPEnvelop)
        begin
    5       SOAPEnvelop. AddInHeader (QoSMetadata, QoSParamValues)
    6       <Li>=ExtractQoSParamValues(SOAPEnvelop)
    7       SOAPEnvelop.release()
    8       <QoSPi>=ComputeQoSValues(<Li>)
    9       Log(<QoSPi>)
        end
endloop
```

Table 2. The Requester-Side Monitor behavior for synchronous communication

time ”t4”, line5) and released (line7) after the extraction of the QoS parameters list (e.g. ”t1, t2, t3 and t4”, line6). After that, it computes the QoS parameters values (namely for, execution time: ”t3 - t2” and response time: ”t4 - t1”, line8) and logs them.

The monitoring is more complex with the asynchronous communication (or ”*one way message*”). We deal only with requests, and responses received as requests. To discriminate between them, we use the *WS-Addressing* specification [13] in order to identify and to relate requests for asynchronous invocations thanks to message headers ”*MessageId*” and ”*RelatesTo*”.

In Table 3, the *RSM* intercepts the outgoing one way SOAP message (line1), adds QoS metadata (e.g. issuing request time ”t1”, line2) and forwards the request to the *PSM* (line3).

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       SOAPEnvelop.release()
        end
endloop
```

Table 3. The Requester-Side Monitor behavior for asynchronous communication

```
loop
    1   OnRequest( SOAPEnvelop)
        begin
    2       SOAPEnvelop.AddInHeader(QoSMetadata, QoSParamValues)
    3       <Li>=ExtractQoSParamValues(SOAPEnvelop)
    4       <WS-Adrs>= ExtractWSAddressingData(SOAPEnvelop)
    5       SOAPEnvelop.release()
    6       if ("RelatesTo" ∉ <WS-Adrs>)   /*Handle as a request*/
    7           Local_Log (<Li>, "MessageId")
    8       else  /*Handle as a response*/
    9           <Li.new>= Find_in_Local_Log_MessageId_EqualTo("RelatesTo")
   10           <QoSPi>=ComputeQoSValues(<Li>,<Li.new>)
   11           Log(<QoSPi>)
   12       Endif
        End
endloop
```

Table 4. The Provider-Side Monitor behavior for asynchronous communication

As illustrated in Table 4, the *PSM* intercepts the incoming one way SOAP message (line1). It adds QoS metadata (e.g. issuing request time ”t2”, line2) and releases the request (line5). After the extraction of the QoS parameters list (e.g. ”t1 and t2”, line3), it extracts the WS-Addressing data (line4). If this data does not contain the

/*Considered WSDL interface of the deficient WS to substitute*/
$WSDL_Interface(S_{deficient}) = \{Op_1, \dots, Op_N\}$
/*Browse the available services*/
Analyze_Available_WS_Interfaces();
/*Considered web services for the substitution*/
$S \leftarrow \{S_1, \dots, S_M\}$
(1) *Single_Substitution* $\leftarrow \exists\, S_t$ *such that* $WSDL_Interface(S_t) \supseteq WSDL_Interface(S_{deficient})$
if *(Single_Substitution)*
  then **for each** *(Op_i)-request*
    Reroute_request_to S_t;
  endfor
endif
(2) *Composite_Substitution_NoOverlap* $\leftarrow \exists\, \{S_h, \dots, S_k\}$ *such that:*

$$\left(\left(\bigcup_{i=h}^{k} WSDL_{Interface(S_i)}\right) \supseteq WSDL_{Interface(S_{deficient})}\right)$$
$$\wedge$$
$$\left(\left(\forall_{x,y}\left(WSDL_Interface(S_x) \cap WSDL_Interface(S_y)\right) \cap WSDL_Interface(S_{deficient})\right) = \emptyset\right)$$

if *(Composite_Substitution_NoOverlap)*
  then **for each** *(Op_i)-request*
    Reroute_request_to S_j, *such that* $Op_i \in WSDL_Interface(S_j)$
  endfor
endif
(3) *Composite_Substitution_Overlap* $\leftarrow \exists\, \{S_h, \dots, S_k\}$ *such that:*

$$\left(\left(\bigcup_{i=h}^{k} WSDL_{Interface(S_i)}\right) \supseteq WSDL_{Interface(S_{deficient})}\right)$$
$$\wedge$$
$$\left(\exists Op_i \in WSDL_Interface(S_{deficient}), \exists (x,y) \text{ such that } Op_i \in \left(WSDL_Interface(S_x) \cap WSDL_Interface(S_y)\right)\right)$$

if *(Composite_Substitution_Overlap)*
  then **for each** *(Op_i)-request*
    Reroute_request_to S_j, *such that* $Op_i \in WSDL_Interface(S_j),\ \nexists k: Op_i \in WSDL_Interface(S_k)$
  and
/*Dealing with Overlapped Operations: First policy: Following the service availability*/
  if *(Service_Availability_Policy)*
    for each *(Op_i)-request, such that* $Op_i \in \{Sj_1, \dots, Sj_a\}$
      Reroute_request_to Sj_k, *such that* $Sj_k \leftarrow$ *High_ Availability*($\{Sj_1, \dots, Sj_a\}, Op_i$)
    endfor
/* Dealing with Overlapped Operations: Second policy: Switching fairly between services*/
  else
    for each *(Op_i)-request, such that* $Op_i \in \{Sj_1, \dots, Sj_a\}$
      Reroute_request_to Sj_k
      j_{k++}
    endfor
  endif
endif

Table 5. The reconfiguration algorithm

message header "*RelatesTo*" (line6), we deal here with a request. The QoS parameters values are saved in a local log (line7). Unless, the request is handled as a response (e.g. "t1" represents "t3", and "t2" represents "t4"). It looks for the related QoS parameters values in the local log where the "*RelatesTo*" represents a "*MessageId*" of a previous request (line9). Then, it computes the QoS values (namely for, execution time: "t3 - t2" and response time: "t4 - t1", line10) and logs them (line11).

The *Analysis* module targets evaluating the health of a given service and not a specific interaction within a given conversation. Hence, the degradation detection considers interactions with a provider and its requesters. Our approach acts proactively by observing the evolution of QoS values computed during runtime. It aims to detect QoS degradation which is considered as the symptom of a future or an imminent deficiency. Indeed, a continuous increasing of the response time or a continuous decreasing of admission rate is a significant indicator to an imminent service deny. In our point of view, having a response time increase when dealing with requests from *N* different requesters is considered as QoS degradation in the same way as for response time increase involving handling *N* requests from the same requester.

3.2 The Diagnosis/Prognosis & Decision

Depending on the adopted policy according to the application context, two models may be used: *Diagnosis* and *Prognosis*. Proactively or reactively on receiving alarms, the modules implementing these models, inspect the service behavior on the basis of the logged QoS parameter values. This allows the identification/prediction of the past/eminent deficiency. The *Decision* is based on reconfiguration actions

Self-Healing components & services	Deployment Side			
	Constraints on the Provider Side	Constraints on the Requester Side	Constraints on the Requester and the Provider Sides	Free deployment
PSM	Third party Side	Provider Side	Third Party Side	Provider Side
RSM	Requester Side	Third party Side	Third Party Side	Requester Side
Logging WS	Third Party Side	Third Party Side	Third Party Side	Third Party Side
Log	Third Party Side	Third Party Side	Third Party Side	Third Party Side
Analysis	Third Party Side	Third Party Side	Third Party Side	Third Party Side
Diagnosis/Prognosis	Third Party Side	Third Party Side	Third Party Side	Third Party Side
Decision	Third Party Side	Third Party Side	Third Party Side	Third Party Side
Reconfiguration Enforcement services	Third party Side	Provider Side	Third Party Side	Provider Side

Key: Third Party Side   Requester Side   Provider Side

Table 6. The possible configuration under the different constraints

for prognosis and repair actions for diagnosis.

The diagnosis is more sophisticated in the case of global management. In such case, it is not limited to the analysis of interaction between a single pair of requester/provider, but it reasons about the interaction of the web service with multiple other web services of the global application. This global view of the system gives us the possibility to identify the source of the degradation, and to optimize the repair effort by avoiding over-reactions and useless reconfiguration actions. Such a situation occurs for QoS degradation due to delay propagation.

3.3 Reconfiguration Enforcement

The proposed approach is based on the architectural reconfiguration. It substitutes the deficient service by another equivalent from the selected *Substitutable Services WSDLs* or by a composition of several services (see algorithm described in Table 5). The reconfiguration execution is performed thanks to the *Dynamic Binding Connector* which unbinds the current connection, and reroutes requests to the new selected services in a seamless way for remote requesters. The *Dynamic Binding Connector* is automatically generated and deployed using runtime compilation and reflective programming.

The substitution of a deficient web service may be done through one or many other services. We distinguish three different cases as shown in the algorithm of Table 5.

The first case addresses the single substitution. All requests are routed to a new web service, which offers the same operations as the deficient WS. In such case, $S_{deficient}$ is entirely replaced by S_t.

The second case considers the composite substitution by two or more services, where their union covers the operations offered by the deficient one. No overlapping is detected in the offered operations. In such case, $S_{deficient}$ is replaced by a set of services.

The third case is similar to the second, but some provided operations appear many times in the considered web services for the substitution. Two policies are used. In the first, the substitution services may be used by other requesters which bypass the QOSH middleware. In this case, we follow the high availability to reroute requests. In the second, we deal with services accessible only through the QOSH middleware. In this case, switching requests between these services seem to be the more realistic while no other applications use them.

The single substitution may be also viewed as a total substitution of all the service operations (all operations are deficient) and the composite substitution as a partial substitution of only the deficient service operations.

3.4 Self-healing service deployment assumptions

The deployment of the self-healing services and components is distributed on three sides, as described in Table 6. The third party is a trusted party between the provider and the requester. The best deployment configuration is the *Free deployment* one. *RSM* should be close to the requester-side. Similarly, *PSM* should be close to the provider-side. This makes more accurate the QoS measurement. Many self-healing components are deployed in the third party-side in order to non overload the provider and requester sides.

Deployment constraints rise due to many reasons as security issues, and access rights. For example, applications orchestrated with BPEL engine limit the access to the requester-side in order to deploy *RSM*. This may be resolved while using the BPEL-provided monitor. Also, usually providers limit the access to their web services for security reasons and they prevent deploying monitors or other self-healing components. In this case, we can deploy *PSM* and the *Reconfiguration Enforcement services* within the third party-side.

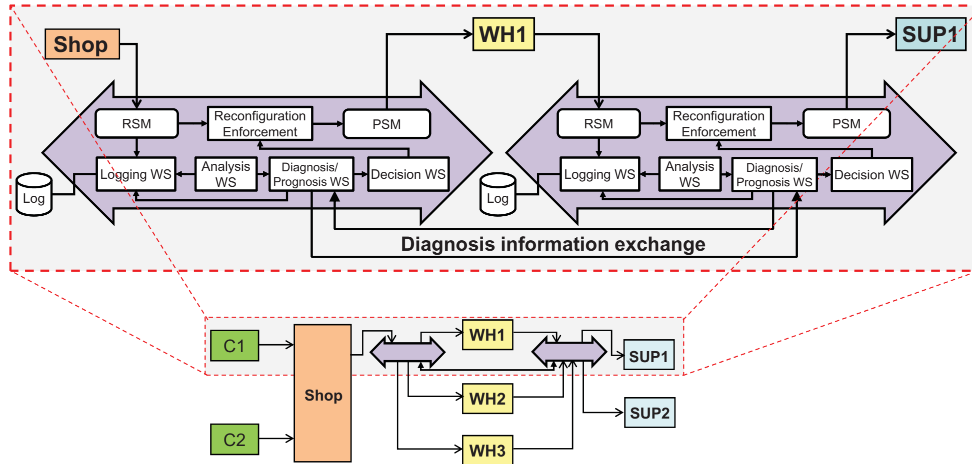

Figure 2. Details of global distributed self-healing architecture applied to the FoodShop case study

4 Illustration: The example of the supplier chain automation category

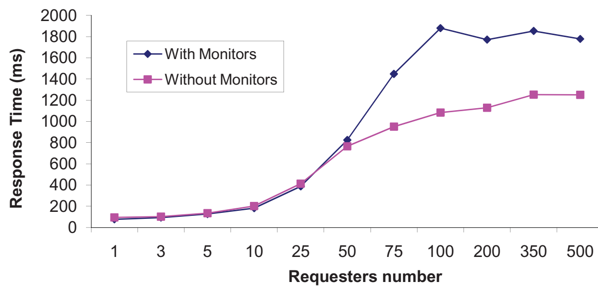

Figure 3. Load of monitors

The FoodShop[5] application is studied in the framework of the WS-DIAMOND project (see Figure 2). It is concerned with a web service-based application representing a company that sells and delivers food. The company has an online Shop and several warehouses (*WH1*, ..., *WHn*) located in different areas that are responsible for stocking imperishable goods and physically delivering items to customers. In case of perishable items, that cannot be stocked, or in case of out-of-stock items, the warehouse must interact with several suppliers (*SUP1*, ..., *SUPm*).

The FoodShop is a composition of interactive web services. It is implemented using BPEL (*ActiveBPEL* version 2.1). Applying the QOSH middleware, we face the deployment of handler issues. The requester is assumed to be the BPEL engine, which constraints the deployment of the monitor on the requester-side (see section3.4). Two solutions are possible: The first relies on extending the ActiveBPEL provided handlers on the requester side, and the second deploys monitor on the third party-side.

The QOSH middleware is deployed within the FoodShop application between each pair of provider/requester as shown in Figure 2. The *Diagnosis* modules exchange information, in order to coordinate the healing actions. For instance, for the two interlocked services *WH1* and *SUP1*, the QoS degradation of *SUP1* may propagate to *WH1* from the *Shop* point of view. This triggers a healing process within the two QOSH middleware instances. If not in coordination, each QOSH middleware substitutes its provider. However, the global reasoning about the degradation deduces that the *WH1* QoS degradation is due to the propagation and only *SUP1* has to be substituted.

This implementation shows the feasibility of the QOSH middleware[1]. We inject degradation in the FoodShop application, and we follow the QOSH middleware behavior. The *Analysis* service detects the degradation and sends alarms to the *Diagnosis* service which identify the degradation. The *Decision* service asks the *Reconfiguration Enforcement* service for generating a new connector to repair. Then, next requests are executed by a new provider.

We have experimented the monitoring cost with a large scale experiment under the gird5000 [1] to measure the response time of web services while varying the requesters number from 1 to 500.

Figure 3 shows how the response time varies with the requester number. In the first drawing, the monitoring is achieved using the *RSM* and the *PSM* (With Monitors). In the second, the measurement is achieved in the client code and without using monitors (Without Monitors). We conclude that with less than 50 concurrent clients, both drawing are similar and the load of monitors is negligible (about

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

zero). From these experiment results, we find that the in-code monitoring is better than the interceptor monitoring while we exceed 50 concurrent clients. But remember that the in-code is dependent to the client application.

5 Related Work

In this section, we first present related work on QoS-aware middleware compared to our work. Most works are centered on the network and transport levels [6, 5]. Less effort in QoS-aware middleware has been done at the application and business process levels [14] and few approaches address both levels [7].

Authors of [6] present a middleware to manage QoS inside a cluster of application servers. This approach proposes three modules, implemented as component, to ensure QoS adaptation. The first is the *Monitoring* module which is in charge of observing and detection QoS degradation. The second is the *Configuration* module which is responsible for ensuring service availability while rebuilding the cluster in order to meet the SLA specification. The third is the Load Balancing module which acts as a HTTP proxy dispatching client requests to services according to their availability.

Work in [5] considers the management of network resources and services. It presents a layered framework architecture. The first layer includes the Reconfiguration Executors which adapt network components. The second layer is responsible of monitoring and diagnosis of QoS degradation. The last layer is a supervision graphical interface for the system administrator. The abstraction of the exchanged data between layers is based on the capabilities of the XML.

Work in [3] presents a monitoring system composed of three parts. The first addresses monitoring report generation. The second gathers, centralizes and analyzes monitoring data. The third part in charge of presenting graphically monitored data to user. It improves the proposed architecture while integrating automatically any applications within the monitoring system.

Authors of [14] address the QoS management on web service composition. They deal with this issue at two levels: The first considers the selection of a web service which satisfies user requirements specified in a QoS model, and the second handles the adaptation of the running process when a QoS degradation occurs while substituting the deficient web service. The monitoring and the diagnosis steps are omitted in this approach.

Work in [7] deals with QoS during three phases: 1) QoS specification at development phase, 2) QoS setup at discovery phase, and 3) QoS adaptation at runtime phase. The adaptation operates on the level of the resource management to enhance the QoS. This approach has a centralized Decision module which handles the content adaptation, the resource allocation or the resource adaptation.

Other works address the monitoring and evaluation QoS parameters of web services. El Saddik [9] addresses the measurement of the scalability and the performance of a web service 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 moves the problem from the server level to the proxy level.

The work proposed in [8] 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 [10] propose a framework for QoS measurement. They extend SOAP implementation API in order to measure and log QoS parameters values. The API modification has to be achieved at both requester and provider sides. This automates the performance measurement values. Also, it allows continuously updating information about the 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 has to be set up on the provider SOAP implementation which is not possible in all cases.

Authors of [12] present an online monitoring and analysis approach. They observe and measure QoS parameters and resources specified as a quality model and send data to a central analyzer which checks for degradation. A visualization component is used to give a view of the web services state to the administrator. The paper discusses the selection of a suitable monitoring mechanism for each system.

Tosic et al. [11] present WSOL (Web Service Offerings Language) which is an XML-based language allowing the QoS management of web services. Each Service Offering describes a set of QoS, functional constraints and access rights. Service Offerings are dynamic and can be manipulated at runtime. Providers can offer more than one Service Offerings for each web service. Requesters have to choose an appropriate one and can switch dynamically between them. Similarly to our work, WSOL allows the monitoring and the evaluation of QoS between web service providers and requesters. We note that WSOL does not support a management action when degradation happens; it only describes monetary penalties to be made.

IBM [4] proposes WSLA (Web Service Level Agree-

ments) as a specific SLA for web services. As WSOL, it is an XML-based specification. An SLA defines the QoS that a web service delivers to service requesters. SLA defines QoS parameters and expresses how to measure them. Similarly to our work, SLA allows the monitoring of QoS parameters and specifies the management action to take when detecting degradation. The monitoring is ensured by a trusted third party, and generally the management is reduced to a notification of concerned peers.

6 conclusion

In this paper, we presented a QoS-oriented self-healing middleware for web service-based applications. This middleware enables the monitoring of both synchronous and asynchronous communications. It supports predictive and reactive repair policies. The repair enactment is based on the architectural reconfiguration providing single and composed substitutions for the web services at the origin of the QoS degradation. An implementation has been developed to assess the applicability of the monitoring and the repair within the designed middleware.

In the presented work, we have provided solutions for managing self-healing at the communication level with a minimum access to the managed web service implementation and with reduced assumptions on the deployment capabilities of the middleware components. This allows a large applicability according to a black box approach that needs no knowledge nor control on the web services and any possible associated orchestration process. Relaxing such assumptions allows the unavailability of the web services to be reduced during the reconfiguration step. On the other hand this also allows handling self-healing for both stateful and stateless web services. This constitutes our current work. We are developing a new prototype which acts at the HTTP level and which addresses the global monitoring and diagnosis of QoS degradation. Our future wok also includes the automatic generation of the monitor behaviours from WSDL descriptions extended, using annotations, with additional QoS-related information.

References

[1] F. Cappello and al. Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform. In *6th IEEE/ACM International Workshop on Grid Computing*, 2005.

[2] R. B. Halima, K. Drira, and M. Jmaiel. A qos-driven reconfiguration management system extending web services with self-healing properties. In *Proceedings of the 16th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 339–344, Washington, DC, USA, 2007. IEEE Computer Society.

[3] K. Kang, J. Song, J. Kim, H. Park, and W.-D. Cho. Uss monitor: A monitoring system for collaborative ubiquitous computing environment. *IEEE Transactions on Consumer Electronics*, 53(3):911–916, 2007.

[4] A. Keller and H. Ludwig. Defining and monitoring service-level agreements for dynamic e-business. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 189–204, Berkeley, CA, USA, 2002. USENIX Association.

[5] P. S. L. Dimopoulou, E. Nikolouzou and I.Venieris. Qm-tool: An xml-based management platform for qos aware ip networks. *IEEE Network*, 17(3):8–14, 2003.

[6] G. Lodi, F. Panzieri, D. Rossi, and E. Turrini. Sla-driven clustering of qos-aware application servers. *IEEE Trans. Softw. Eng.*, 33(3):186–197, 2007.

[7] K. Nahrstedt, D. Xu, D. Wichadakul, and B. Li. Qos-aware middleware for ubiquitous and heterogeneous environments. *IEEE Communications Magazine*, 39(11):140–148, 2001.

[8] 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 (in conjunction with IEEE ECOWS 2006)*. CEUR-WS, Dec 2006.

[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 2005 Australian conference on Software Engineering*, pages 202–211, Washington, DC, USA, 2005. IEEE Computer Society.

[11] V. Tosic, B. Pagurek, K. Patel, B. Esfandiari, and W. Ma. Management applications of the web service offerings language (wsol). In *Advanced Information Systems Engineering, 15th International Conference, CAiSE 2003, Klagenfurt, Austria, June 16-18, 2003, Proceedings*, volume 2681 of *LNCS*, pages 468–484. Springer, 2003.

[12] Q. Wang, Y. Liu, M. Li, and H. Mei. An online monitoring approach for web services. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 1-*, pages 335–342, Washington, DC, USA, 2007. IEEE Computer Society.

[13] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[14] L. Zeng, B. Benatallah, A. H. Ngu, M. Dumas, J. Kalagnanam, and H. Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, 2004.