REPLICATION OF WEB SERVICES FOR QOS GUARANTEES IN WEB SERVICE COMPOSITION

Dirk Thissen and Thomas Brambring
Department of Computer Science, RWTH Aachen University, Ahornstr. 55, 52074 Aachen, Germany

Keywords: Web services, replication, quality of service, proxy.

Abstract: The concept of *web services* defines a middleware for implementing distributed applications independent of used platforms and programming languages. When developing new software systems, re-use of functionnality of existing services can be done to reduce development time and costs. This process of re-use is called *web service composition*. But, current web service standards are not equipped to consider non-functional requirements, i.e. *quality of service* (*QoS) aspects* of a user to a composed service. Thus, *capabilities* of composed services cannot be guaranteed. This paper presents an approach to integrate QoS aspects into the composition of web services by using *service replication*. At composition time, service instances are chosen depending on the QoS requirements of a user to the whole service, and it is decided which services in the composition have to be replicated and which replication strategy to use. Replication ensures that the QoS requirements are not only considered at service selection time, but also can be granted at service runtime.

1 INTRODUCTION

A *web service* is some software which is seen as a service it offers, aiming at automatic machine-to-machine communication independent of the implementation of the software. A software architecture is defined to give standards for service definition (WSDL) and interaction (SOAP). Web services are loosely coupled; the search for services to communicate with is done dynamically at runtime using a service registry (UDDI). The interaction across platforms and programming languages enables easy and fast deployment of new software: complex software systems can be plugged together from existing services to a collaborating group. This is called *web service composition*. One vision is to achieve an automatic composition and interaction of services. But, the standards are not equipped to consider *non-functional* demands to a composition, i.e., QoS requirements like performance, reliability, or cost.

For acceptance by customers, a business should provide good quality of its composed services. This paper focuses on handling QoS aspects in web service composition. The service registry UDDI was enhanced to select web services for a composition with respect to the QoS requested by a user. A *QoS broker* was added to UDDI to manage the services' QoS information and to calculate the best combination of services in the composition due to a user's requirements. But, QoS aspects can be dynamic, so a QoS-oriented selection is only considering a kind of system snapshot. Depending on the time between service selection and the execution of a selected service in the composition (basing on the composition pattern and the interaction between the services), QoS information used at selection time can be outdated at service usage time. Thus, the architecture is enhanced by *service replication* to guarantee the QoS from selection time also at runtime. A *flexible replication framework* was developed to allow for as well performance-related as fault tolerance- and availability-related replication of services.

The paper is structured as follows. Chapter 2 presents a short overview about related work in the area of QoS and replication in web service composition. In chapter 3, the principle of the QoS broker is explained. Chapter 4 describes the replication architecture and gives an overview about the current implementation status. Finally, chapter 5 concludes the paper and gives an outlook on the ongoing work.

2 WEB SERVICES AND QOS

Several approaches for the composition of web services exist. A prominent example is the *Business*

Thissen D. and Brambring T. (2008).
REPLICATION OF WEB SERVICES FOR QOS GUARANTEES IN WEB SERVICE COMPOSITION.
In *Proceedings of the International Conference on e-Business*, pages 155-160
DOI: 10.5220/0001906301550160

Process Execution Language (BPEL) for Web Services (OASIS, 2007). But these approaches are not considering quality of service in the composition process. Though there is a lot of research in web services and composition, not much is related to QoS. The existing work mostly refers only to a part of the whole problem. (Liu et al, 2004) for example present a framework to *publish up-to-date QoS information* for web services, but the success of this mechanism depends on feedback of users about the quality of the services they consume. (Zeng et al, 2004) present a method to select services that fit to a user's interest (expressed as QoS parameters). *Local optimization* and *global planning* are combined to find the best set of services for a composition. But, in case of highly dynamic QoS parameters, the global planning approach might take more time for re-calculation than the execution of the service would need. Thus the approach itself can violate the QoS. (Jaeger et al, 2004) propose a mechanism which could be more efficient by using an *aggregation scheme for QoS* aspects. The approach sounds well but was not implemented nor tested by the authors. We used this approach as basis for the implementation of an own solution to consider QoS aspects in composition (Thißen and Wesnarat, 2006) which is explained more detailed in chapter 3.

All these approaches have the same weakness: services for the composition are chosen some time before execution. If QoS parameters change, during service execution the QoS demands of a user nevertheless can be violated. *Replication* is a possible solution to deal with dynamic QoS parameters on performance, high availability, and fault tolerance/ reliability. Instead of running a single instance of a service, several copies are used. Replication defines methods for keeping consistent all copies (called *replicas*). The complexity is hidden from the user of a service by a frontend which acts as the service from the user's view. A lot of replication algorithms are given. In *active replication*, all replicas act in the same way. The frontend uses group communication to distribute a request to all replicas. It decides how to deal with responses of the replicas, depending on the QoS aspect which should be considered. To ensure service available or to decrease the response time of a service (performance), the frontend returns the first response to the user. To improve fault tolerance, it compares and combines all responses. A different approach is *passive replication*. One replica is a primary, and the frontend only communicates with this replica. The primary forwards the requests to all other replicas (backups) to keep them consistent. If the primary fails, a backup can take over its role, which improves fault tolerance and availability.

There are lot of other replication algorithms, and also approaches exist to implement them within a web service architecture. E.g., (Ye and Shen, 2005) discuss active replication for web services. But, the focus is only on reliability of web services, and only active replication is implemented. The same holds for (Chan et al, 2007): it is focussed on reliability. WS-Replication (Salas et al, 2006) also uses active replication to achieve high availability, and WS-multicast is used for communication between the replicas. WS-multicast is SOAP-based and maybe causes a high overhead. (Osrael et al, 2007) is a more flexible approach, implementing passive replication and designing an open system for later addition of other replication strategies. Consistency can be weakened in this approach to reduce the performance overhead caused by update propagation. But, till now only a variant of passive replication is realized, and the focus is on fault tolerance.

Concluding, the replication approaches either focus on only one replication strategy, use multicast on SOAP level which decreases performance, or only consider a certain QoS aspect, e.g. availability. Thus we designed an own replication framework for integration with composition, which offers more flexibility, see chapter 4.

3 QOS IN SERVICE SELECTION

For composing web services under QoS constraints, we followed the approach presented in (Jaeger et al, 2004): a workflow pattern is given, showing the relations between services. Aggregation rules are used to combine quality measures assigned with single services to come to an overall rating of sets of services. We identified *relevant QoS information* and basic *composition patterns* (SEQUENCE, AND, OR, XOR, and LOOP) from which the whole workflow pattern can be formed. Next, we defined corresponding *aggregation rules* and a *selection mechanism* to choose the best service candidates. Given the workflow pattern for a composed service, aggregation of QoS parameters is done by collapsing the whole composition graph step-wisely into a single node, starting with the innermost composition pattern. By aggregating the properties recursively, only one node is left in the final state. A set of formulas was defined to model the aggregation of the QoS parameters performance, cost, reliability, and availability. This mechanism enables us to check the resulting QoS of a set of services. Because

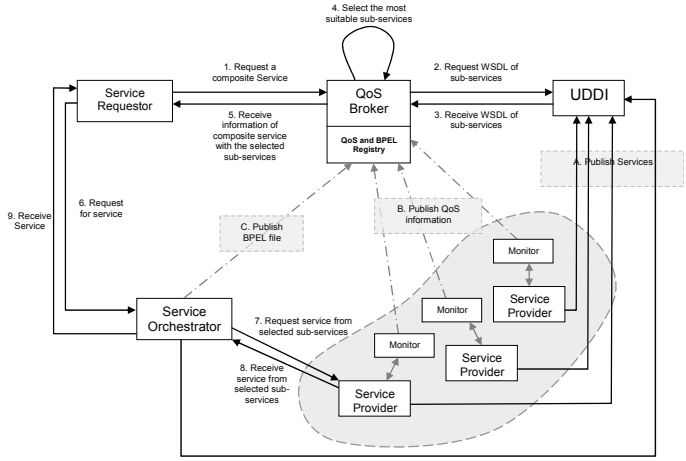

Figure 1: Enhancement of the web service architecture.

we need to find a set of services to be executed, each possible combination of service candidates for the current composition pattern is evaluated, and the best ones regarding the user's demands are selected. Multiple criteria decision making and weighting are used to combine a service set's aggregations for different QoS parameters into one value, a *quality scores*. For a composition pattern, the set of service candidates with highest quality score is selected, and it is done aggregation of the next innermost composition pattern till a single node (the composed service) remains with assigned QoS values.

For implementation of this approach we have designed a prototype which enhances the general web services architecture. *Apache tomcat* was used as web container for the provided web services, *Apache Axis* services as SOAP implementation. *jUDDI* was chosen as UDDI registry, for executing composed services the *Oracle BPEL Process Manager* was used. It provides a service orchestrator which can be assigned a workflow pattern and a set of basic services; it then manages the execution of the services due to the pattern. For integrating QoS consideration as described before, we have implemented some more components, see figure 1.

The central component is the *QoS broker* which implements the QoS aggregation rules. It involves a *BPEL registry* and a *QoS registry*. Service providers as usual register their services with UDDI (step A in figure 1). To publish QoS information, a *monitor* is assigned each service, registering with the QoS registry when a service is put into UDDI (step B). When a composed service is deployed, the workflow pattern is stored in the BPEL registry (step C).

When a service requestor searches for a web service, it contacts the QoS broker (step 1). It does not need to know if a service is a composed one or not; the broker uses the BPEL registry to search for a composition pattern. If one is found, the broker asks UDDI for available candidates to all services in the pattern (step 2). Having retrieved a list of all available candidates (step 3), the broker connects to the QoS registry to get the QoS values for these services. By stepwise aggregation of the values according to the pattern from the BPEL registry and by selection of the best fitting candidates, a set of basic services is chosen (step 4). The requestor gets back a reference to an orchestrator for using the service (step 6/9). The orchestrator manages the service execution (step 7/8). After execution, it gives feedback to the broker. In other requests to the same composed service, the broker can make use of it.

Not included in figure 1 is the use of the QoS monitors. Getting the QoS information for aspects like cost is no problem: the values are constant for a longer period of time and can be filled in by the service provider at service setup. But most aspects, are dynamic, e.g. like performance. Thus a monitor is assigned each service to record its behaviour, to compute floating averages, and to forward this information to the QoS registry. To avoid that service providers have to modify their services, the monitors are independent components. They get the needed information from so called valves placed on Tomcat engine level. Here, e.g. timestamps can be used to get statistics about the queuing time of a request.

Nevertheless, the QoS broker cannot guarantee the QoS from selection time to be constant at runtime, thus we had to enhance this architecture by a mechanism which allows for some control at execution time of the services.

4 REPLICATION FOR QOS GUARANTEES AT RUNTIME

The architecture presented in chapter 3 is only able to consider user demands at selection time. Our next step was to enhance the architecture by capabilities of replication, to control the selected QoS at runtime. Because of the disadvantages of existing approaches, we designed an own replication architecture considering the following goals:

- *Allow for flexible choice of replication algorithm at runtime*. We want to use replication for guarantees on several QoS aspects, thus we need different replication strategies supporting performance, availability, and fault tolerance in one approach.
- *Open architecture which can easily be enhanced with new replication algorithms*. For the beginning, we only considered the most prominent algorithms: active and passive replication.
- *Decide at composition time which services have to be replicated to fulfil a requestor's demands*. E.g., the use of several replicas may improve the reliability, but may contradict the cost of service usage if one has to pay for each extra replica. Thus, in the selection process a tradeoff is necessary between gain and costs of using replication, including the number of replicas to use.
- *Transparently use group communication and avoid communication overhead by using SOAP*. Otherwise, replication could contradict the QoS.
- *Automatically generate request and result classes for web services from WSDL files*. Reduce the costs and time for integrating the mechanisms into each application newly.

We designed our replication architecture oriented at these goals and allowing for easy integration with our QoS-based selection of service candidates in a composed service (chapter 3). In the following, the components of the architecture and their interaction are described in more detail.

The *QoS broker* remains the central component of the architecture. It is enhanced by enabling the selection of a replication strategy as well as a set of suitable replicas for a service. For simplicity, we started with the consideration of simple services within the replication process, but oriented at the composition architecture for easy integration.

The interaction of the QoS broker with service requestor and the replicas of a single service is shown in figure 2. The replicas are all registering with UDDI as usual (step (a) in figure 2). The replicas additionally register with the QoS broker resp. the assigned QoS registry via their monitors (b) as described in chapter 3. The services' monitors do not need to know if they are belonging to a replicated service or to a simple one, they have to submit the same information as before. The monitors regularly measure the QoS values of their replicas, calculate advanced information like floating averages, and deliver the resulting values to the QoS broker.

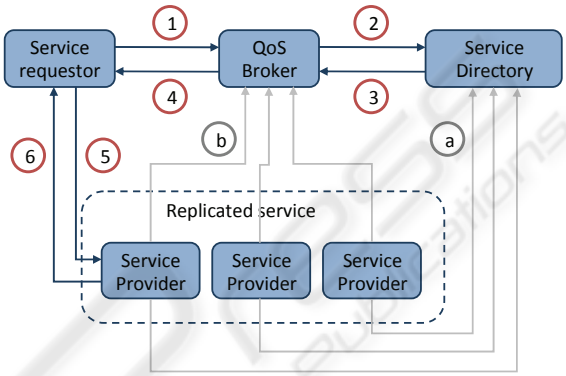

Figure 2: Replication enhancement.

If a service requestor contacts the QoS broker to ask for a service (1), the broker interacts with UDDI to find all replicas to the requested service (2 + 3). Based on the requested QoS, the broker now can select a subset of fitting replicas which seems to be sufficient to fulfil the requestor's demands. Simultaneously, it can decide on the best replication strategy regarding the requested QoS. If e.g. high performance is needed primarily, active replication is chosen to reduce response times. If availability has priority, passive replication is more appropriate to reduce the communication overhead. Based on the known availability probability of the service, also the number of replicas could be determined. Currently, active and passive replication are implemented in our prototype, and only a few rules are implemented, which strategy to use in which cases.

The service requestor gets back a reference for its service (4) and can use it (5 + 6). The detailed information transmitted in these steps depend on the replication strategy chosen by the QoS broker since the service requestor maybe has to contact a single service or maybe a service group.

If the broker chooses *passive replication*, the requestor only communicates with a single replica. In contrast to common passive replication there is no fixed primary replica which all the time is contacted. Instead, the QoS broker chooses the actually best replica due to the requestor's demands and returns a reference to this replica to the requestor. The other

replicas only serve as backups and are invisible to the requestor. The dynamic primary selection allows for a better average QoS level in terms of performance because it enables a kind of load balancing between all available replicas. But, one has to keep in mind that using different primaries for different requests can cause consistency violations. Thus it depends on the service itself if the weakening of the consistency is useful.

On the other hand, in *active replication* the requestor has to communicate with all replicas simultaneously. Thus he has to communicate with a group of services instead with a single service.

To hide the different usage for the replication schemas, *proxies* are used. They encapsulate the functionality of communication with replicated services. Only a single interface is offered to the requestor. Independent if passive or active replication is used, the requestor gets back a reference to the used proxy instead of a reference to a concrete service (in step 4 of figure 2) – the proxy itself seems to be the service for the requestor. This schema intentionally is designed similar to the usage of composed services via an orchestrator as descrybed in chapter 3, to *merge* the functionalities of orchestrator and proxy. The only difference for the requestor is that the QoS broker not only sends back a reference to a service (the proxy), but also some additional configuration parameters the requestor has to use in its request to enforce a certain replication process (which was chosen by the broker).

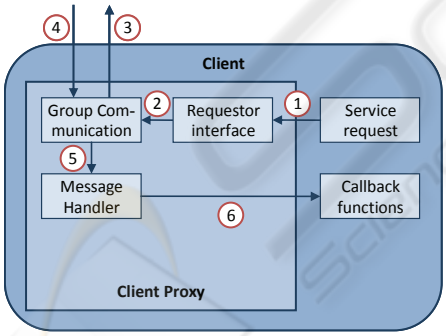

Figure 3: Client proxy.

Figure 3 shows the structure of such a proxy. The client only holds a reference to its proxy, which is capable of performing all replication strategies for any kind of request. The proxy is informed by the QoS broker about the set of replicas to use for a request. To inform the proxy how to handle a certain request, the requestor now has to include the configuration parameters chosen by the QoS broker in its request (step 1 in figure 3). Such a request may look as follows:

```
Proxy.requestActively(request,
      READ_ONLY, GET_FIRST, 2000);
```

The original request of the user is passed to the proxy only as one parameter `request`. The proxy is able to process this request by using the correspondingly assigned replicas. In which way to use the replicas, is defined by the other parameters of the user's call. The proxy implements functions `requestActively` and `requestPassively`. Depending on which replication mechanism is chosen the client has to call the corresponding function. The client gets this information from the QoS broker as part of the configuration information. The second parameter of the request tells the proxy if the request is read-only or not. In case of read-only, consistency is relaxed, which can improve the performance of a request. This parameter is followed by an information if the first response has to be forwarded to the client (e.g. for performance or availability aspects), or if the proxy has to wait for all responses and to combines them in some way to achieve fault tolerance. The last parameter is a timeout. It defines how long the proxy has to wait for responses before combining the received results (or before sending an error message back to the requestor).

The proxy now can inform the group communication component about the needed communication mechanism (2) and the request correspondingly is passed only to a single service or to a group of services (3). The results which are coming back from the replicas (4) are passed on to a message handler (5) which can treat the responses in different ways as described above. If passive replication was used, the proxy immediately uses a callback function to deliver the result to the requestor (6). In case of active replication, it can forward the first response to the client, or collect all requests coming in before a timeout and form a consensus out of them before passing only a single response to the requestor.

Also on server side a proxy is needed to coordinate all replicas corresponding to the chosen replication strategy, see figure 4. The request comes in over the group communication mechanism (step 1 in figure 4) and is forwarded to the message handler (2). The message handler in the background interacts with the QoS monitor (a) to allow for statistics about the number of requests per second, response times, etc which is part of the QoS parameters collected by the monitor. Because in active replication consistency requires a sorted execution of requests

from different clients on all replicas, the holdback queue (3) delays all requests till they can be executed without violating consistency to other replicas. To do so, requests have to be sorted the same way for all replicas. For this purpose Lamport Timestamps are used. The requests are sorted into a delivery queue (4) which simply implements a FIFO strategy and executes one request to the service after the other. The requests can be passed on to the web service by using a callback function (5 + 6). The delivery queue also gets back the response (7 + 8) and initiates the transmission of this response back to the requestor (9). Again, the group communication mechanism takes over the transmission of the result to the requestor (and to the backups, in case of passive replication).

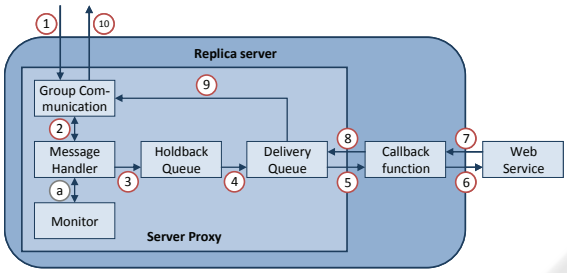

Figure 4: Server proxy.

Using client and server proxy, the whole replication is transparent for users and services. When new replication strategies are implemented, only the proxies have to be enhanced.

5 CONCLUSIONS

Currently, the architecture as described in chapter 4 is finished, and experiments are performed to evaluate the behaviour of the replication framework. On one hand the experiments should validate the correctness of the implementation. On the other hand (and more important for the ongoing work) the evaluations also should help in comparing gains and costs of the replication strategies. These comparisons are necessary for fine tuning of the decision rules inside the QoS broker: for which combination of requested parameters which strategy should by used, and with how many replicas. In parallel, implementation has started to integrate the replication enhancement into composed services. This task is easy because the architecture of the replication system was oriented at the existing composition architecture (integration of orchestrator with client proxy, implementation of server proxy as valves like the monitors). Afterwards, the gain of using replication in the composition again has to be evaluated by a number of experiments.

Replication only is one way to improve the quality of a service. After finishing our current work, beside integrating more replication strategies we want to examine if instead executing the same service several times, also equivalent services of different providers could be used. Also, we plan to enhance the functionality of the proxies by other strategies, e.g. load balancing as a mechanism with weaker guarantees as replication, but on the other hand cheaper if services – and quality guarantees – have to be paid for.

REFERENCES

Chan, P.P.W., Lyu, M.R., Malek, M., 2007. *Reliable Web Services: Methodology, Experiment and Modeling*. Proc. IEEE International Conference on Web Services (ICWS 2007), Salt Lake City, USA.

Jaeger, M.C., Rojec-Goldmann, G., Mühl, G., 2004. *QoS Aggregation for Web Service Composition using Workflow Patterns*. Proc. 8th IEEE International Enterprise Distributed Object Computing Conference (EDOC'04), Monterey, USA.

Liu, Y., Ngu, A.H.H., Zeng, L, 2004. *QoS Computation and Policing in Dynamic Web Service Selection*. Proc. 13th International World Wide Web Conference, New York City, USA.

OASIS, 2007. *Web Services Business Process Execution Language Version 2.0*. OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf

Osrael, J., Froihofer, L., Weghofer, M., Goeschka, K.M., 2007. *Axis2-based Replication Middleware for Web Services*. Proc. IEEE International Conference on Web Services (ICWS 2007), Salt Lake City, USA.

Salas, J., Pérez-Sorrosal, F., Patino-Martínez, M., Jiménez-Peris, R., 2006. *WS-Replication: A Framework for Highly Available Web Services*. Proc. 15th International World Wide Web Conference (WWW 2006), Edinburgh, Scotland.

Thißen, D., Wesnarat, P., 2006. *Considering QoS Aspects in Web Service Composition*. Proc. 11th IEEE Symposium on Computers and Communications (ISCC'06), Cagliari, Sardinia, Italy.

Ye, X., Shen, Y., 2005. *A Middleware for Replicated Web Services*. Proc. IEEE International Conference on Web Services (ICWS'05), Orlando, USA.

Zeng, L., Benatallah, B., Ngu, A.H.H., Dumas, M., Kalagnanam, J., Chang, H, 2004. *QoS-Aware Middleware for Web Services Composition*. In: IEEE Trans. Software Eng., Vol.30, No.5.