A Middleware for Reflective Web Service Choreographies on the Cloud

Thiago Furtado*, Emilio Francesquini†, Nelson Lago*, Fabio Kon*

*Department of Computer Science, University of São Paulo, Brazil
{tfurtado, lago, kon}@ime.usp.br
†Institute of Computing, University of Campinas, Brazil
francesquini@ic.unicamp.br

ABSTRACT
Web service composition is a commonly used solution to build distributed systems on the cloud. Choreographies are one specific kind of service composition in which the responsibilities for the execution of the system are shared by its service components without a central point of coordination. Due to the distributed nature of these systems, a manual approach to resource usage monitoring and allocation to maintain the expected Quality of Service (QoS) is not only inefficient but also does not scale. In this paper, we present an open source choreography enactment middleware that is capable of automatically deploying and executing a composition. Additionally, it also monitors the composition execution to perform automatic resource provisioning and dynamic service reconfiguration based on pre-defined Service Level Agreement (SLA) constraints. To achieve that, it keeps a meta-level representation of the compositions, which contains their specifications, deployment statuses, and QoS attributes. Application developers can write specific rules that take into account these meta-data to reason about the performance of the composition and change its behavior. Our middleware was evaluated on Amazon EC2 and our results demonstrate that, with little effort from the choreography developer or deployer, the middleware is able to maintain the established SLA using both horizontal and vertical scaling when faced with varying levels of load. Additionally, it also reduces operational costs by using as little computational resources as possible.

Categories and Subject Descriptors
D.1.3 [Programming Techniques]: Distributed programming

General Terms
Reliability, Performance

Keywords
Middleware, QoS, Reflection, SOA

1. INTRODUCTION
A computational system is said to be scalable if it is able to perform its duties in an acceptable manner (as defined by the user) even in the event of wide variations of load. Cloud environments present these systems with the possibility of scaling up or down in a very straightforward manner. Therefore, a well designed middleware for these environments should be able to react to the infrastructure and system conditions. Reactions might include, for example, automatic usage of additional computing resources.

A common approach to build a distributed system is to use a Service-Oriented Architecture (SOA) [10, 16]. This architecture has enjoyed widespread adoption mainly because it facilitates the interoperability and reuse of legacy components. The basic component of a SOA-based system is a service. Services are independent entities that provide specific functionalities to their clients. If the desired functionality requires the use or cooperation of several services, they can be organized into compositions. Compositions with a centralized control structure are known as orchestrations whereas compositions without a centralized control structure are known as choreographies [8, 1].

As each service provides different functionalities, they are bound to different usage patterns. Not only the intrinsic functionality of each service might influence the load, but also they are subject to fluctuations due to the time of the day or the day of the week. For instance, a shopping application might experience a high load during the evening but face very low demands during the morning. These variations in the load are reflected on the amount of CPU and I/O resources required to keep the service working within the acceptable QoS level. In this context, it is desirable that the middleware responsible for the execution of a choreography on the cloud also becomes responsible for the allocation and release of resources in accordance with the load.

To maintain the expected QoS levels, resources allocated to each service must be tuned continuously, thus adapting to the variation of load. This adaptability requires a fundamental change in the way services are deployed and managed. A choreography that is capable of "reasoning" about its current state, with the help of the underlying middleware, and also to dynamically reconfigure its own internal components (services) and connections (service bindings) becomes, therefore, reflective. In this sense, reflective means that the system is aware of its own structure and is capable of dynamically adapting this structure according to runtime needs [9].

Our reflective choreography management middleware focuses on the deployment and QoS management of choreographies in order to serve as a Platform as a Service (PaaS) system for scalable choreographies. It represents each chore-
Choreographies as an abstract specification. This representation includes the component services and the relationship these services have between themselves as well as to other services. By keeping that information at the meta-data level, the choreography developer or deployer can use the middleware to support rules that reason about the choreography state to redesign its own deployment specifications to comply to the established SLAs. After these specifications are recalculated, the middleware can then automatically perform the necessary deployment actions.

As a significant improvement to the previous version of our work [6], in this paper we present a middleware to support reflective choreographies that are capable of dynamic reasoning and adaptation to the changing environmental parameters to keep the QoS within the specified SLAs. To evaluate the effectiveness of our approach, we performed an experimental evaluation in which we analyze SLA violations when we apply vertical and horizontal scaling approaches to a choreography deployment. Our results show that the resulting system is able to automatically and dynamically maintain the QoS within acceptable levels even under large load variations. This is achieved by using additional resources when the load increases and releasing underused resources when the load decreases.

The management of a composition QoS demands, in addition to the QoS model, dynamic adaptation mechanisms. To that aim, in this work we employ Complex Event Processing (CEP) and reflection. CEP [5] is an event-based approach used to measure system properties through the analysis of events and their comparison to the related SLAs. Complex events can be also employed to trigger actions indicating which reconfiguration should be done to keep the expected QoS levels. A complex event can be defined as the correlation between a set of other events (simple or complex). A complex event could be generated, for instance, when a higher than normal operation response time event occurs concomitantly with a CPU and I/O usage at the server side. CEP is generally regarded as being a practical and efficient choice for monitoring and executing reactive tasks in distributed systems [5].

Additionally, the reconfiguration of choreographies can be further improved by the knowledge of the facts related to their deployment. In this text, an event is a representation of a functional characteristic of the composition as, for example, the response time. A fact is a representation of a structural state, for example, the IP address of a service instance or the list of cloud nodes on which some service is currently deployed. When the monitoring framework indicates that some correlation of events should trigger a reconfiguration, the middleware reflectively can reason using information contained in the meta-data layer to perform changes and dynamically adapt the deployment of the compositions on the cloud environment ensuring the expected QoS is met.

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response Time</td>
<td>The amount of time elapsed between sending a request and receiving the response.</td>
</tr>
<tr>
<td>Throughput</td>
<td>The amount of requests a service can process in a given time unit, normally measured in requests/second.</td>
</tr>
<tr>
<td>Availability</td>
<td>Often measured in percentages, is defined as uptime / (uptime + downtime) where uptime represents the time a service is running and answering requests and downtime, the time the service is not answering requests.</td>
</tr>
<tr>
<td>Financial Cost</td>
<td>The monetary value owed to the infrastructure provider for the use of resources.</td>
</tr>
</tbody>
</table>

Table 1: QoS Properties

means of user-defined rules.
Some virtualization technologies allow users to modify VM resources such as memory or number of processing units on the fly. However, currently, no major cloud provider offers this possibility.

1 Some virtualization technologies allow users to modify VM resources such as memory or number of processing units on the fly. However, currently, no major cloud provider offers this possibility.

4. EXPERIMENTAL EVALUATION

The evaluation of the proposed middleware was done experimentally using a simple benchmark choreography. Both the benchmark and our middleware are open source and licensed under the Mozilla Public License V2. Their code is available at: https://github.com/chores/ enactment_engine/releases/tag/v2014-09.

4.1 Experimental Setup

The benchmark choreography consists of two services: Service A, which exposes an operation that runs a CPU-intensive task (a recursive naïve version of the Fibonacci algorithm) and Service B which performs several requests to Service A. The load brought on by these calls is evenly distributed among various replicas of Service A.

Our evaluation is based on the observation of the middleware reactions to the variations on the services response time. This variation is induced by a test client application that exercises distinct access patterns, with different number of requests per minute. To simulate situations that would be likely faced on a real production environment, we used real access patterns from Wikipedia. For practical purposes, the original 24-hour trace was linearly mapped to two hours of simulated accesses.

Due to the nature of the benchmark, our evaluations employ a simplifying assumption that only the response time is enough to evaluate the QoS of the composition. However, other factors such as CPU use might be included as part of the QoS evaluation. For an I/O-bound application, similar rules based on throughput levels instead of response times could also be used to make better scaling decisions.

In our tests, the middleware was configured to avoid response times above one second. The actual rule, written in Drools’ Rule Language, is depicted in Figure 2. Lines 02-09 initialize variables with response time events, SLA information and the configured scalability policy. Next, Lines 11-14 count the number of response time events collected during the last 30 seconds. Then, Lines 15-19 evaluate if at least 95% of the requests observed the specified SLA. Lines 20-21 avoid concurrent reconfigurations and Lines 22-23 account for the delay between a previous reconfiguration and actual response time stabilization. Finally, if all the conditions previously evaluated were fulfilled, Lines 24-29 generate a high response time complex event. The generated event will be evaluated by another rule that will take appropriate action to ensure QoS levels are in accordance to the specified SLA.

Evaluations of the horizontal and vertical scaling were performed independently. Depending on the chosen policy, the middleware creates and deploys additional service replicas on new nodes (horizontal) or migrates the service to more powerful nodes (vertical). Similarly, when the load on the system is reduced, service replicas are deactivated or migrated to smaller machines.

4.2 Experimental Platform

For our experiments, we employed two private servers in our university, in addition to Amazon EC2 instances. The DM and the RMA were executed on private servers, an Intel Core 2 Duo P8700 @ 2.53 GHz with 4 GB of RAM and an Intel Xeon E7-3870 @ 2.40 GHz with 16 GB of RAM, respectively. The service composition was executed on three different Amazon instance types: small (1 virtual core, Xeon E5-2650 @ 2.26 GHz, 1.7 GB of RAM), medium (1 virtual core, Xeon E5-2650 @ 2.26 GHz, 3.75 GB of RAM), and large (2 virtual cores, Xeon E5645 @ 2.0 GHz, 7.5GB of RAM). All machines were running GNU/Linux 3.2.0 and Open JDK 6.

For the horizontal scaling experiment, we used VMs of small instance type whereas for the vertical scaling experiment, we employed small, medium, and large instances.

4.3 Experimental Results

Figure 3 and Figure 4 show the experimental results including monitoring and reconfiguration data. These figures separate the results into two graphs to facilitate the understanding of the relationship between response time and workload of the machines. The graph on the top shows, on the left axis, the response time experienced by Service B clients during execution. On the right axis, we show the number of requests per second made to the service. The graphs on the bottom part of the figures track the workload on the virtual machines. The blue boxes highlight the time intervals comprising the moment the system detects the need for a reconfiguration and the end (deployment/reification) of the reconfiguration task.

As expected, the workload on the nodes that host Service A is tied to the response time: when the workload increases, the response time increases as well. This behavior prompts a reaction from the RMA during the evaluation of the event correlation rules. This reaction triggers a reconfiguration intended to maintain the QoS level established by the pre-configured SLA. When we analyze the service response times, we realize that the adaptation mechanisms of the middleware are capable of maintaining the expected QoS for most of the time. There are some occasional spikes in the response time, which can be attributed to the delay between the increase of load and the reaction of the middleware, due to the use of a 30-second sampling window by the rule engine.
4.3.1 Horizontal Scaling

The simulation begins with only a small instance. Since the number of requests is high, SLA violations are detected right at the beginning of the execution (first ∼100s). These violations trigger the reconfiguration mechanisms of the middleware. Since the middleware was configured to perform horizontal scaling, it decides to add a new replica. As the number of requests decreases from ∼400s to ∼2000s of execution time, the low CPU usage triggers a new reconfiguration releasing the recently created instance. Once again, as the load on the system increases, a new reconfiguration is performed at ∼4980s to increase the number of simultaneous instances to two.

Right at the end of each reconfiguration, we can observe a brief increase in the average response time. This higher than normal response time is due to the hand-over procedure performed between the previous and new instances where the service is deployed. Additionally, at about 3100s of execution, there is a peak in response time that we could not explain. However, we suspect this was related to an overload on the shared physical machine hosting our VM.

Removal of a service instance is simpler than addition and, therefore, this kind of reconfiguration is faster. On average, our experiments took ∼55s to perform reconfigurations that only removed an instance and ∼335s to carry out those that involved the creation of a new cloud instance.

4.3.2 Vertical Scaling

In contrast to the horizontal scaling approach, vertical scaling performs mainly service migrations across different VM instance types to maintain QoS; it is only when the usage of the most powerful instance is close to its maximum that the middleware decides to use more replicas of the services. Using the same access patterns of the previous simulation, we can compare their results. The first noticeable difference is that, in this approach, the middleware never had the need to use more than one replica. Amazon’s medium and large instances were enough to fulfill the response time requirements, even when the number of requests greatly increased after 3500 seconds of simulation.

Similarly to the horizontal scaling experiment, in the beginning of the execution, the number of requests is too high to be fulfilled by only a small instance. SLA violations are detected and a reconfiguration is performed, as expected, almost at the same time as the previous experiment (around 100s of execution). As the number of requests decreases from ∼600s to ∼2000s of execution time, the low level of CPU usage triggers a new reconfiguration releasing the medium instance in favor of a small one. Once again, as the load on the system increases, a new reconfiguration is performed at ∼3100s to use a medium instance until ∼6540s, when a new increase from a medium to a large instance is performed. On average, each reconfiguration took ∼298s. Since vertical scaling always involves the creation of a new instance, differently from horizontal scaling, the duration of every reconfigurations is virtually the same.

Throughout the experiments, for most of the time, the specified SLA was observed. The middleware only violated the SLA in 7% and 1.5% of the requests for the horizontal and vertical policies, respectively. These percentages do not include the violations that occurred during the first reconfiguration, at the beginning of the execution. If we consider these violations, the percentages rise to 22% and 3.7% respectively. This difference highlights the impact a poor initial deployment has on QoS. However, it also demonstrates our middleware’s capability to dynamically adapt to QoS degradation. Nonetheless, even when the SLA was being respected, downsampling reconfigurations intended to reduce the financial costs of operation were performed, for example, at around 1800s and 2000s of execution time for the horizontal and vertical policies respectively.

5. RELATED WORKS

Research on choreography QoS maintenance has been gaining importance due to the ever increasing use of service compositions. Current research on choreography QoS maintenance revolves around two main approaches. The first approach employs service selection based on service degradation factors [14, 7, 17, 20]. In this case, when a service degradation is detected, a search is performed and an equivalent non-degraded service is chosen to fulfill the requests in lieu of the degraded one. The second approach is based on service reconfiguration taking into account resource utilization and the topology of the service [21, 18]. This approach uses both service migration as well as horizontal scaling to maintain an acceptable QoS.
Differently from related works [19], we use a reflective approach to service composition reconfiguration [9]. Most related works directly use monitoring analysis results to perform deployment reconfigurations. However, in our case, monitoring analysis is performed using CEP to update queryable representations of the deployment, statuses, and specifications of the system. Then, using a rule-based approach based on these meta-data, we delineate the most appropriate reconfiguration strategy in an incremental fashion.

6. CONCLUSION

In this paper, we presented a middleware to support the easy enactment of reflective web service choreographies on the cloud. Our middleware maintains runtime information about the choreography specifications, deployments status, and QoS attributes. An event-based monitoring approach based on CEP analyzes the system performance and is able to reflectively reason and modify its own behavior. The rules governing this self adaptation are based on QoS properties, such as response time, and are defined by the application. Experimental results show that our system is able to deploy an experimental composition and adapt it to cope with load variations, limiting resource usage while allowing expected QoS to be violated for only brief amounts of time.

The combination of reflection and CEP provides great flexibility in handling different situations. Indeed, reflection enables the middleware to gather and maintain knowledge about all relevant aspects of the choreography, feeding data to the CEP module. Concomitantly, CEP rules may be arbitrarily complex and deal with any number of detected events. Our middleware focuses on performance aspects, but the approach can be applied to other needs as well.

A limitation of the current prototype is the manual creation of the reconfiguration rules based on the developer’s knowledge of the services. We plan to use the Scalability Explorer tool [13] to reflectively create rules for service compositions. The choice of a rule-based CEP engine will allow us to easily adopt a vast number of different reconfiguration strategies to our prototype.

Acknowledgments

This work was funded by the European Commission’s FP7-ICT-2009-5 under grant agreement number #257178 (project CHOReOS – Large Scale Choreographies for the Future Internet – www.choreos.eu), by FAPESP, CNPq, and CAPES.

7. REFERENCES