Towards an Enactment Engine for Dynamically Reconfigurable and Scalable Choreographies

Thiago Furtado, Emilio Francesquini, Nelson Lago, Fabio Kon
Institute of Mathematics and Statistics
University of São Paulo
São Paulo, Brazil
{tfurtado,emilio,lago,kon}@ime.usp.br

Abstract—Service compositions have recently been in the spotlight. Although they are not something new, as the complexity of service based systems grows, we observe an ever increasing interest in these approaches. 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 any central point of coordination. Choreography clients expect a minimum level of Quality of Service (QoS); however, due to the distributed nature of these systems, a manual approach to resource usage monitoring and allocation is not only inefficient but also does not scale. In this paper we present an open source choreography enactment engine that is capable of automatically deploying and executing a given composition. Additionally, it also monitors a composition execution to perform automatic resource provisioning and dynamic service reconfiguration based on pre-defined Service Level Agreement (SLA) constraints. We evaluated our system on Amazon EC2 and preliminary results demonstrate that it is able maintain the QoS of a composition, even when faced with varying levels of load, while at the same time reducing costs by using as little computational resources as possible.

Keywords—QoS; reconfiguration; monitoring; SOA

I. INTRODUCTION

The use of service compositions is increasingly seen as an interesting approach to build complex networked applications comprised of multiple independent entities [1], [2]. Orchestration is centralized approaches to service composition coordination while Choreographies [3] offer an approach that aims at keeping the business logic and coordination distributed among the involved parties as a means to improve the scalability of the composition as a whole. A service composition is typically opaque to the client, which sees it as a single service and not as an intricate aggregation of distinct services.

In these compositions, some of the component services naturally process more requests than others. Additionally, their behavior is not necessarily the same, since they have different levels of resource utilization such as I/O and CPU. The arbitrary composition of these services and their distinct resource utilization levels make the beforehand knowledge of the amount of computational resources needed for an efficient execution very difficult predict.

A composition client expects correct results from a service operation with adequate Quality of Service (QoS). QoS may be quantified by associating, to each relevant quality-related property, a metric and an acceptable range of values for that metric. For instance, throughput (property), measured in requests/second (metric), should be above 100 (range). Normally, these user QoS expectations are formalized by means of an Service Level Agreement (SLA), which defines a set of non-functional properties that each service composition needs to comply with. SLAs, therefore, can be seen as a formalized set of QoS constraints that define the desired non-functional behavior of a service.

To guarantee that the expected QoS is maintained throughout the composition life-cycle, enough resources must be allocated to the execution of each component service. Therefore, any allocation of resources that cannot adapt itself to the changing environment, such as number of simultaneous connections, is bound to impact QoS attributes of the service composition due to workload fluctuations.

Dynamic reconfiguration [4] gives systems the ability to adapt to changing conditions on their environment in order to keep functioning properly. Cloud computing environments provide their users with a simple means to dynamically change the amount of available resources. The current increase in the adoption of these environments highlights the relevance of dynamic reconfiguration as a means to balance QoS and financial cost. Automatic dynamic reconfiguration schemes normally take into account information gathered during run-time by some form of monitoring system and, based on this information, modify some properties of the system. Oftentimes a knowledge base is used in tandem with the monitoring results to improve the decision process, either as a means to improve the precision of a QoS prediction or as a repository of proven solutions to a recurrent situation [5].

In this paper, we describe our work for the development of an enactment engine [6] for service compositions. This engine is capable of automatically selecting (creating if needed) and configuring virtual machines in a cloud environment to deploy and execute a given service composition. Additionally, it can monitor the composition’s resource usage levels as well as its QoS attributes and employ automatic dynamic reconfiguration...
based on horizontal scaling techniques to ensure that the required SLAs are met. To demonstrate the effectiveness of our approach, our prototype was evaluated on Amazon EC2. We experimentally show that we can effectively maintain QoS attributes both when the load on the system increases as well as when it decreases while at the same time reducing operation costs by disabling underused machines.

The remaining of this document is organized as follows. Section II describes how service composition and monitoring relates to service quality. Section III, outlines our proposal for an enactment engine with support for dynamically reconfigurable and scalable choreographies. We show the experimental evaluation of our prototype in Section IV, compare our proposal to related works in Section V and finally conclude in Section VI.

II. BACKGROUND

We now present the basic concepts necessary for the development of our work. We start by presenting service compositions and QoS. Next, we show how the QoS of a composition can be monitored and regulated through a set of rules based on events to perform dynamic reconfigurations.

Service Composition: A service composition is a distributed system composed of a set of services that might themselves be atomic or composite. An atomic service is a service that is able to perform all the operations it exposes without relying on external services. A common way to provide scalability for these atomic services consists of instantiating several replicas of these services at runtime. These replicas are, by definition, independent and can be used interchangeably. Conversely, composite services are those that, to process a request, need to access external services to perform parts of the requested operation. There are two main service composition approaches: orchestration and choreography. Orchestration-based compositions have a central point of control whereas in choreography-based compositions the work-flow control responsibilities are shared among the component services.

Quality of Service: Composition clients expect a certain level of QoS. The QoS of a composition is indirectly defined by the QoS of its component services. QoS management of these highly distributed service compositions requires the analysis of several distinct impacting properties such as response time and availability. A QoS model is a formal description of a set of properties and the metrics used to assess them. Based on service compositions main characteristics, we can define a general set of properties which is described in Table I. There are clearly many more possible properties of interest that might better describe the behavior of a specific composition, however, here we present a representative subset of common properties that can be measured for most service compositions.

Complex Event Processing: To control the QoS of a composition, one needs, in addition to the QoS model, some kind of monitoring and dynamic adaptation strategy. Complex Event Processing (CEP) [7] can be used as a monitoring and reconfiguration approach to guarantee acceptable levels of QoS. In an event-driven architecture, an event (typically generated by some monitoring infrastructure) can influence the status of the whole system. The interaction and correlation between events and their effects can be defined, for instance, by a set of rules evaluated by a rule engine. An event is said to be complex when it is defined as a correlation between two or more simple events. As an example of a simple event, we can cite a higher than normal operation response time and, as a complex event, we can correlate high CPU and I/O usages to timeouts on the client side.

III. ENACTMENT ENGINE

In this section we present our proposal for a service composition enactment engine. This middleware uses CEP to evaluate QoS-related events from the deployed compositions and to provide reaction strategies that maintain the QoS of the composition at acceptable levels. A dynamic reconfiguration strategy is also offered in such a way that the number of resources is minimized, which is essential to reduce operational costs. First, we present our prototype’s general architectural characteristics and then we discuss some of its implementation details.

A. Architecture

The dynamic reconfiguration process is traditionally seen as being composed of four sequential steps (monitoring, analysis, planning, and execution) that are repeatedly performed in an endless loop. These steps are known as MAPE loop [5]. The monitoring phase gathers relevant information about resource usage and QoS levels. Based on this information,
the analysis phase tries to correlate the distinct monitoring events, occasionally identifying a new complex event. When the analysis is concluded, the planning phase defines the most suitable reconfiguration strategy. Next, the execution phase prepares the middleware and performs the reconfiguration. Our enactment engine has an event-based architecture. It uses CEP-based monitoring and a set of rules evaluated by a rule engine to devise and deploy a reactive reconfiguration plan. Figure 1 depicts its simplified architectural diagram.

Figure 1. Enactment Engine architectural diagram.

Our architecture is composed of two main modules. The Deployment Manager (DM) and the Resource Manager Aggregator (RMA). The DM is responsible for deploying and reconfiguring service compositions. It takes as input an XML specification of a choreography and allocates as many resources as needed; for that, it employs computation nodes running on a cloud provider such as Amazon EC2. Additionally, it also manages the deployment of service artifacts and its dependencies. When the service composition is already running, event data are collected by the Monitoring Probe present at each cloud node (Monitoring phase). The probe feeds these data into the RMA. The RMA uses a rule engine to perform CEP (Analysis phase). Events are analyzed and their correlation to known system behaviors is assessed. A single event or a set of correlated events might indicate, for instance, that the performance of some service is degraded or that more resources than necessary were allocated. In any case, the rule engine is employed to determine which is the best reactive course of action. This reaction plan is then sent to the DM that performs the necessary reconfigurations.

Execution: When some action is deemed necessary by the RMA, it notifies the DM with the new deployment plan. This plan is compared to the status quo (managed by the Deployment Monitor component of the DM) and only the necessary modifications are performed by the Choreography Deployer. The choreography deployer might perform horizontal upscaling (or downscaling) as well as service composition code updates.

B. Implementation

Our enactment engine focuses on the deployment and QoS management of service choreographies. Its purpose is to serve as a middleware to provide a Platform as a Service (PaaS) system for scalable service compositions.

In cloud environments, the resources dedicated to a task can be easily increased or reduced [8]. When dealing with IaaS (Infrastructure as a Service) cloud providers, these resources are simply virtual machine instances where the service can be deployed and run. If at any point in time the load on the computational nodes is too high and additional resources are needed, supplementary nodes can be requested from the cloud provider. Average load on each node can thus be reduced as the deployed services are replicated into these new nodes. Conversely, if the average load is below a minimum threshold and therefore, some resources are no longer needed, part of the nodes can be disabled. This simple node-load monitoring concept can be extended to services. By monitoring both the services and the nodes which host them, one can capture important information such as service response time. These data allow the system to pinpoint specific services or cloud nodes with poor performance. Monitoring and adaptation are therefore done from the
composition provider perspective. Thus, domain specific QoS attributes will not be covered in this work since the purpose of the engine is to maintain compositions in general running without perceptible QoS degradation for their clients.

Adaptation mechanisms can take into account previous knowledge about the behavior of the services that compose a choreography. This previous knowledge allows the system to make better informed decisions regarding, for example, the type of cloud node most suitable to perform the initial service deployment. It is also possible to employ this knowledge to predict the possible impact a reconfiguration might have.

The Scalability Explorer tool, currently under active development [9], can be used to automate experiments to assess the impact several distinct reconfigurations have on a specific service. Possible parameterizations include different amounts of load, resources and node characteristics. The results obtained from these experiments can be used to build a knowledge base employed by the adaptation mechanisms and improve the likelihood that the proposed reconfiguration will be enough to maintain the expected levels of QoS.

Unfortunately, not all scaling problems can be handled by simple approaches. Some services might not be easily parallelizable, for example. In these cases, more robust solutions should be adopted. One such example is the migration of services to more powerful (in terms of memory, CPU, clock, etc) virtual nodes. Therefore a monitoring system cannot be limited to the detection of situations that demand reconfiguration actions, it also needs to be able to make decisions about the most suitable approach to handle the problem at hand. The expected requirement for an efficient reconfiguration system based on cloud infrastructures is to be able to decide whether to scale (up or down) horizontally (by adding or removing nodes), vertically (by migrating services to nodes with more adequate characteristics), or some combination of both.

**Monitoring:** As we previously discussed, dynamic reconfiguration usually depends on some form of monitoring. Monitoring is done by the monitoring probe present in each node. The probes monitor metrics related to both resource usage and QoS. Platform metrics (memory and CPU usage, disk and network I/O, etc) are gathered from the virtual nodes using Ganglia [10]. Ganglia is a lightweight monitoring tool that provides several distinct resource utilization measurements using default Linux APIs. Whenever a measurement is considered as being possibly anomalous, it is forwarded to the RMA. Additionally, regular events regarding QoS status and violations are continuously fed into RMA. This is made possible because, in deployment time, services are configured to run on a instance of Tomcat which is instrumented with Tomcat Valves\(^2\) that enables monitoring the response time and throughput of services independently.

**Analysis and Reconfiguration Planning:** The RMA component of our enactment engine uses Glimpse [11] to perform CEP. Glimpse is a tool that supports flexible and generic event monitoring. It provides a network interface based on a publish/subscribe protocol and creates queues that are filled by the monitoring probe and consumed by the Status Manager. These events are processed by Glimpse using a rule engine. Glimpse is built on top of Drools [12], a rule engine implementation for event monitoring and execution of rule-based reaction tasks. This combination of frameworks allows us to perform CEP using temporal reasoning. CEP is generally regarded as being a practical and efficient choice for monitoring and executing reactive tasks in distributed systems [7]. In an event-based architecture, it is important to detect events that have a relationship among them. In this context, CEP provides a way to, from simple events, generate complex events to capture this by means of a set of logical rules.

In this setting, Drools’ main role is to detect complex events, correlating virtual machine resources status with QoS measurements. Such complex events, in turn, may trigger the adequate reconfiguration rules for each different situation. There are some rules exclusive to perform event analysis and some others that instruct the DM to take some action. In the first case, when events happen, event correlation rules are triggered. If the rules evaluation (most of them based on thresholds) indicates a problem with a service, a new complex event is generated. In the second case, in which complex events are processed, rules that determine DM actions are evaluated. Then, if necessary, the RMA reacts by notifying the DM the necessary steps to correct the problem. The use of CEP greatly simplifies the process of creating the reconfiguration plans, since the triggering of a simple rule allows the whole system to send a reactive reconfiguration plan to the DM. The reconfiguration mechanisms were implemented on top of the platform provided by the CHOReOS project [13].

**Execution:** The DM continuously receives information from the RMA, which informs it about necessary service reconfigurations due to, for example, QoS degradation. The DM stores meta-data reflecting the deployed service compositions. Based on the reconfiguration plan received from the RMA, the DM performs the necessary actions and updates these meta-data accordingly. To minimize the impact of these actions, the DM evaluates the reconfiguration plan sent by the RMA and executes only the necessary changes to conform to the new configuration. For that, it calculates the difference between what is deployed to what (according to the new configuration) should be deployed. The ensemble of these steps are called choreography update. Choreography updates not only update the deployment configuration but also they can perform choreography code updates.

---

of this prototype. We outline the cloud environment and the results regarding QoS maintenance for a choreography benchmark.

IV. EXPERIMENTAL EVALUATION

For the evaluation of our prototype we devised a simple, yet instructive, benchmark composition. Both the prototype as well as the benchmark are open source (licensed under the Mozilla Public License version 2.0) and are available for download at https://github.com/choreos/enactment_engine/releases/tag/v2014-04.

The benchmark service composition has two services. Service A receives a request to an operation that, internally, increases the use of CPU by running a naïve recursive Fibonacci algorithm. Service B is the interface endpoint that calls that operation. For this test we evaluated the impact on the average response time as a function of the number of requests. During the evaluation, we vary the number of requests per second. The intention is to simulate a scenario in which a varying number of client requests are made. When the Service A replicas become overloaded, the monitoring system correlates the high load of the cloud node to a possible QoS degradation and requests the creation and deployment of an additional replica of the service on a different machine (horizontal scaling).

The SLA settings used for the experiment take into account the service response time. We establish that the response time should never exceed one second. Therefore we have configured the enactment engine’s RMA component with the following two main parameters. First, we set the threshold for a possible QoS degradation to 900 milliseconds. Then, we set the maximum ratio of the number of requests above 900 milliseconds to 15%. If the ratio exceeds the predefined value, we try to, before QoS degradation happens, detect if a new replica is needed. The beforehand determination of the amount of resources a service will consume can be quite intricate. For this reason in this simulation we employ a simplifying assumption. We assume that the CPU usage has a direct impact on the response times. This does not reduce the generality of our approach since our engine can be configured to use any metric measurable by the monitoring probe. In this simulation we set two thresholds: if the average CPU usage is below 25%, this indicates that a replica can be destroyed and the resource can be released; in contrast, when the average CPU usage is above 90%, a new replica should be created. These values were chosen arbitrarily but they can be empirically tuned with the help of the Scalability Explorer tool.

A. Experimental Platforms

For our experiments we employed two private servers in addition to some Amazon EC2 instances. The full description of these platforms is presented in Table II. The enactment engine components were installed on the private servers.

The DM was executed on Private Server 1 whereas the distributed RMA was executed on Private Server 2. The service composition was executed on Amazon instances (small size), which contained all service instance artifacts.

In spite of the CPU working frequency of Amazon 1 being higher than that of Amazon 2, its processor is from a much older generation. In our benchmarks, we verified that Amazon 1 instances performance is roughly half that of Amazon 2. Unfortunately, Amazon does not allow us to choose the configuration in advance, as both are classified as small instances.

B. Experimental Results

Our experiments were devised to evaluate the capacity of the enactment engine to maintain the QoS of the service composition by applying horizontal scaling. We have monitored both the response time from Service B and the CPU usage of the virtual machines hosting the different replicas of Service A. Figure 2 shows the experimental results including monitoring and reconfiguration data. The figure separates 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 at the same time the number of replicas and the number of requests per second made to the service. The graph on the bottom tracks the workload on the virtual machines. The blue boxes highlight the time intervals comprising the intervals between the moment the system detects the need for reconfiguration and the end (deployment) of the reconfiguration task.

As expected, as the workload increases on the nodes that host the replicas, the response time also increases. This 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.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>CPU</th>
<th>Memory</th>
<th>Software Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private Server 1</td>
<td>2 cores (4 threads) 4GB at 2.53GHz Core 2 Duo P8700</td>
<td>Linux Kernel 3.2.0</td>
<td>Java OpenJDK 6</td>
</tr>
<tr>
<td>Private Server 2</td>
<td>1 Virtual Core at 2.6GHz i7 Nehalem 4GB</td>
<td>Linux Kernel 3.2.0</td>
<td>Java OpenJDK 6</td>
</tr>
<tr>
<td>Amazon 1</td>
<td>1 Virtual Core at 2.6GHz Xeon E5430 1.7GB</td>
<td>Kernel 3.2.0</td>
<td>OpenJDK 6.0 Tomcat 6</td>
</tr>
<tr>
<td>Amazon 2</td>
<td>1 Virtual Core at 2.26GHz Xeon E5-2650 1.7GB</td>
<td>Kernel 3.2.0</td>
<td>OpenJDK 6.0 Tomcat 6</td>
</tr>
</tbody>
</table>
Even if our previous empirical evidence suggested that just one replica should be sufficient for a small number of requests, at the start of the experiment, the response time was higher than expected (red line). When we investigated this issue we realized, as mentioned before, that there are at least two different kinds of small instances provided by Amazon EC2, even if they advertise them as being the same. The performance ratio between these two kinds of instances is \( \sim 2 \). This difference in performance causes an important impact in the service composition QoS, forcing a reconfiguration right at the beginning of the simulation at \( \sim 133 \) seconds of execution.

**Replica 1** is running on an Amazon 1 instance, the slower machine. When **Replica 2** starts the execution we realize that it only consumes about half the CPU of **Replica 1**, although it is subject to exactly the same number of requests. This happens because it was allocated on a Amazon 2 machine. A similar behavior can be observed when we compare **Replica 3** (Amazon 1) to **Replica 2** between \( \sim 1600 \) and \( \sim 2800 \) seconds and **Replica 3** and **Replica 4** (Amazon 2) between \( \sim 3800 \) and \( \sim 4650 \) seconds of execution.

When we increase the number of requests per second (green line) up to 3.5 we notice an abrupt increase in response time. This triggers a “high response time” event that causes the enactment engine to create a new replica (**Replica 1**). The effective release happens at \( \sim 2800 \) of execution.

When the second replica is released, the only replica that is maintained is hosted by an Amazon 1 instance. Unfortunately, our prototype is not prepared to deal with different kinds of machines and this release causes the response time to increase considerably. The increase in the response time and the high levels of CPU usage promptly causes the creation of a new instance (at \( \sim 3560 \) seconds of execution).

Additionally, our prototype has a limitation that creates a short unavailability period when a reconfiguration is performed. These unavailabilities can be seen as troughs in CPU usage in every reconfiguration. When the reconfiguration takes longer, these downtimes can be prohibitive for a real application. The final version of the prototype will not have this limitation.

When we analyze the service response times we realize that the adaptation mechanisms of the enactment engine are capable of maintaining the expected QoS most of the time. There are some occasional spikes in the response time which can be attributed to two main reasons. The first is the already mentioned difference in the performance of the virtual nodes. The second is the delay between the increase of load and the reaction of the enactment engine. In this execution the engine uses a floating window of two minutes to evaluate the performance of the system which causes, on average, a reaction delayed by one minute.

### V. RELATED WORKS

Due to the ever increasing interest in services composition, the maintenance of adequate QoS levels has been a very active field of research. We can roughly classify the research in this field in two major categories: service selection and QoS-driven reconfiguration. In the first category are the works related to the discovery of the best service composition to optimize the QoS using resources that are already deployed. The second category, which includes our own research, include works that try to predict the behavioral changes a composition undergoes after a reconfiguration event as a means of improving the effectiveness of the reconfiguration process. We now discuss some of these works in more detail.

**Service Selection:** Works in this category often present pragmatic approaches to compositions reconfigurations by service selection. Yin and Li [14] show how non-functional reconfiguration by replacement of components that appear defective for already existing components can be achieved. QoS management approaches based on machine learning and genetic algorithms were also proposed [15], [16].

Wang et al. [17] use proactive and passive approaches. The application produces QoS measurements and sends them to the monitoring module which also actively polls for QoS information. This module then tries to find the reasons of the eventual anomalies and defines action plans depending on their type. Yet, neither the gathered parameters nor the
way in which they are processed into QoS metrics are clearly stated. In a similar way, Halima et al. [18] use a proactive monitoring approach to QoS. If composition response time increases beyond a pre-defined limit, the execution of an algorithm is triggered to search for QoS degradation sources. However, even if the reason for the degradation is found, it does not include any means of resolution.

Yau et al. [19] present a way to develop adaptive software by considering activities, status, and events using an intricate QoS model. The model tries to maintain QoS constraints of timeliness, throughput, accuracy, and security. However, this work also does not propose any means of resource adaptation.

**QoS-driven Reconfiguration:** Contrary to the works already presented, research in this category provides insights into how QoS-driven reconfiguration of a composition can be achieved. During reconfiguration, some services might become unavailable or even present severe QoS degradations. This is exactly the problem tackled by Li [20], in which a framework that maintains QoS requirements in reconfiguration time is presented.

Calinescu et al. [21] provide an overview of QoS in SOA architectures and argue that self-adaptation is not only a problem of service selection or workflow restructuring, but also a matter of resource allocation and parametrization for services over which the user has some form of control. They propose a framework that unifies several aspects of adaptation, with algorithms for probabilistic assessment of services, a formal model of QoS requirements specification, and a method for deriving global QoS from what is known of each individual service, effectively reusing a few tools previously developed. This work is, therefore, mainly an integration effort.

In contrast to the related works, in our approach we are interested in adjusting the performance of individual services, so that, as a whole, global QoS constraints can be maintained. But, in order to do that and, at the same time, devise good reconfiguration strategies, it is essential to have a comprehensive knowledge of how reconfigurations affect the behavior of the services. In some cases, it might be possible to know beforehand the expected behavior of an application and use this information to assist the reconfiguration process. Part of our approach involves using the Scalability Explorer [9] tool, which allows us to run a set of scalability experiments and collect QoS influencing metrics such as resource usage and response time. The result of this analysis makes it possible to do better reconfiguration planning.

**VI. CONCLUSION AND PERSPECTIVES**

As we have shown, our system was able to deploy an experimental composition and adapt it to cope with load variations, limiting resources usage while allowing expected QoS to be violated for only brief amounts of time. Larger compositions should be used to assess the system. In order to do that, we have started the development of a mechanism to solve that problem. There is, however, room for improvement in some areas which we discuss below.

The system presented here is still a proof-of-concept and deals with single choreographies only; it should be able to reconfigure several running choreographies. It would also benefit from taking a more holistic approach instead of focusing on the QoS status of each single service: only the relevant composition endpoints should be analyzed, and reconfiguration should be undertaken only when these endpoints present QoS degradation.

The status of the whole choreography is not being taken into account to perform reconfigurations with more precision. For example, the system could avoid removing instances that appear to be deployed in one of the better cloud nodes. In a scenario with a large number of services, it is desirable to be able to deploy several services on each available server. This would allow us to maintain two or more lightweight services in only one machine. Data collected during run-time (QoS and the corresponding resource usage predictions) might also be used to fine-tune the services which need dedicated cloud nodes. Currently, we perform only horizontal scaling. Further research on how to choose the best scaling approach (horizontal, vertical, or a combination of both) for each distinct scenario is clearly required.

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 to automatically 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.

**ACKNOWLEDGMENT**

This work was partially supported by the European Commission’s Seventh Framework Programme FP7-ICT-2009-5 under grant agreement number #257178 (project CHOREoS – Large Scale Choreographies for the Future Internet – www.choreos.eu).

**REFERENCES**


