Monitoring Service Choreographies from Multiple Sources

Amira Ben Hamida[2], Antonia Bertolino[1], Antonello Calabrò[1], Guglielmo De Angelis[1], Nelson Lago[3], and Julien Lesbegueries[2]

[1] CNR–ISTI, Italy
{antonia.bertolino,antonello.calabro,guglielmo.deangelis}@isti.cnr.it
[2] Linagora R&D Toulouse, France
{amira.benhamida,julien.lesbegueries}@linagora.com
[3] University of São Paulo, Brazil
lago@ime.usp.br

Abstract. Modern software applications are more and more conceived as distributed service compositions deployed over Grid and Cloud technologies. Choreographies provide abstract specifications of such compositions, by modeling message-based multi-party interactions without assuming any central coordination. To enable the management and dynamic adaptation of choreographies, it is essential to keep track of events and exchanged messages and to monitor the status of the underlying platform, and combine these different levels of information into complex events meaningful at the application level. Towards this goal, we propose a Multi-source Monitoring Framework that we are developing within the EU Project CHOReOS, which can correlate the messages passed at business-service level with observations relative to the infrastructure resources. We present the monitor architecture and illustrate it on a use-case excerpted from the CHOReOS project.

Keywords: Monitoring, Choreographies, Complex Event Processing, SOA, SLA, QoS.

1 Introduction

The Future Internet (FI) context envisions a global environment that expands itself along two key dimensions, the Internet of Services and the Internet of Things. Among the others, a key feature offered by these dimensions concerns the availability of loosely-coupled methods for the management of remote resources allowing the execution of distributed and composite service-based applications.

In this vision, black-box entities (i.e., either the services, or the things) are discovered, chosen and bound at run-time. Specifically, this run-time binding can take place based on the functional interface each entity exports, or on the Quality of Service (QoS) levels they manifest. Both negotiations and run-time bindings leave space to unexpected events or scenarios that were not considered when designing either the single services or the whole composition.

P. Avgeriou (Ed.): SERENE 2012, LNCS 7527, pp. 134–149, 2012.

Because of this inherent nondeterminism and dynamism, system *adaptation* is the key feature to pursue: paraphrasing Darwin, the applications that are going to survive in the fight for survival among the plethora of arising services, will be the ones that are the most adaptable to change. At the basis of adaption is awareness: systems must be enhanced with the capability to monitor the behaviour resulting from the composition of services and things and trigger adaption as failures occur.

We investigate such concern in the context of service choreography. Choreographies are conceived as abstract specifications, typically defined and managed by third party organizations, aimed at modeling dynamic and flexible composition of services into complex business workflows [1]. Specifically, a service choreography is a description of the *peer-to-peer* externally observable interactions that cooperating services should put in place. Such multi-party collaboration model focuses on message exchange, and no central coordination can be assumed.

The EU Project CHOReOS[1] addresses the challenges posed by the development, management and assessment of choreography-centric distributed applications, advocating a highly dynamic and user-centric model-driven approach. The project is developing an Integrated Development and Runtime Environment (IDRE), as well as an associated development process. Part of the IDRE is a framework for testing and monitoring the developed and enacted choreographies. Here, comprehensive testing techniques such as [2,3,4], have to be necessarily complemented with run-time monitoring approaches that are essential to support adaptation as well as informed run-time management activities [5].

Monitoring consists in collecting data from a running application so that they can be analyzed to either notify, or predict run-time anomalies. Several works in the literature deal with run-time software monitoring [6,7,8,9,10]. Nevertheless, the emphasis of the service-oriented paradigm natively drives the building of software systems as multi-layered [11]. Consequently, monitoring is often dealing with layer-specific events, and addressing layer-specific issues.

The main contribution of this paper is to propose a Multi-source Monitoring Framework that we are investigating within the CHOReOS project, and that can correlate the messages monitored at business-service level, with the observations captured by the infrastructure monitoring the low level resources.

The rest of the paper is organized as follows: Section 2 presents the overall architecture of our multi-source monitoring approach; then the next three sections detail the features of each specific source. Section 5 illustrates a case study while in Section 6 we contrast our results against related work. Finally, Section 7 closes the paper drawing conclusions and future work.

2 Proposed Approach

Figure 1 depicts the high-level architecture of our Multi-source Monitoring Framework. Specifically, the architecture relies on the Distributed Service Bus

[1] `http://www.choreos.eu`

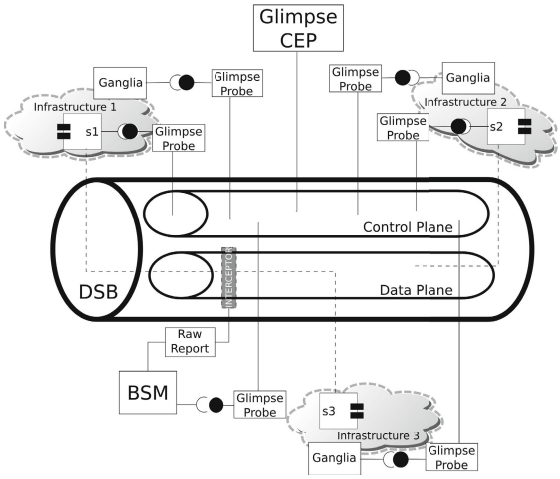

Fig. 1. Multi-source Monitoring

(DSB) component, a shared and distributed communication channel for the monitored events. The DSB distinguishes between a set of channels dedicated to the monitoring activities (i.e., `Control Plane`), and other channels where both coordination and application messages can flow (i.e. `Data Plane`). The data passing through the `Control Plane` can be correlated and analyzed by means of a Complex Event Processor (CEP).

The Multi-source Monitoring Framework integrates three different solutions by means of the DSB. Each solution provides monitoring facilities for a specific kind of source, specifically:

Infrastructure Monitoring: The monitoring elements belonging to this kind of source are focused on the knowledge of the status of the environment where both services and things are running. In this sense, these sources provide support for the monitoring of resources, both in terms of utilization and health status. As detailed in Section 3, among others approaches on resource monitoring, we will mainly implement such sources by means of Ganglia [12].

Business Service Oriented Monitoring: This kind of source is responsible for monitoring messages exchanged among services cooperating within either a workflow or a choreography by means of the DSB. Specifically, distributed interceptors are deployed in the DSB in order to capture the messages that services exchange. As reported in Section 3.1, the goal of this source is to analyze the temporal sequence of messages passing through the bus and look for violations of the choreography specification with respect to functional, QoS, or Service Level Agreement (SLA) violations.

Event Monitoring: This kind of source belongs to a generic event-based monitoring infrastructure able to bridge the notifications coming from the other two

sources. As detailed in Section 4, such sources are based on Glimpse[2], which can include a coherent set of domain-specific languages, expressed as meta-models. In this way, we can exploit the support for automation offered by model-driven engineering techniques [13].

3 Infrastructure-Oriented Monitoring

Any large scale cloud-based system needs to support the monitoring of resources, both in terms of utilization and health status. This is what allows the system to perform corrective actions in order to maintain optimal resource usage (avoiding both overloading and wasting resources) and to handle failures such as crashed or overloaded nodes. A lot of sophisticated monitoring systems dedicated to resource monitoring in large-scale computing environments such as grids already exist (see Section 6); in our work, we leverage previous works by borrowing heavily from Ganglia [12]. Ganglia is one of the most successful grid monitoring systems, offering good performance, low overhead, and flexibility, which explains why it is used in several high-performance computing clusters.

The resource monitor subsystem has two main components:

1. A set of data collectors that gather local information such as load average, I/O rates, and network utilization. These collectors run on every active node of the cloud. Data for each node is both made available on demand over TCP/IP and, at the same time, periodically pushed over UDP to be replicated in nearby nodes.
2. A notification mechanism that detects potentially relevant events, such as exceptional load average or too little available disk space, and generates a corresponding notification that is forwarded to the event monitoring subsystem so that this will trigger some corrective action. This might be replicating an overloaded service, migrating services that communicate a lot to be closer together etc.

For data collection, we simply reuse the *gmond* component of Ganglia, since it offers rich functionality, low overhead, and depends on almost no configuration to work. However, *gmond* depends on static pre-configuration to identify peers for data replication, which is not adequate for a highly dynamic cloud environment. In order to tackle this configuration problem, the notification daemon running on each machine also continuously refines the list of replication peers and modifies the configuration of *gmond* accordingly.

The notification mechanism simply polls *gmond* periodically and identifies eventual relevant events according to some hard-coded rules, such as "load average above 3" or "free disk space less than 10%". Such events are forwarded to the higher-level event monitor system described in Section 4 for further analysis. It should be noted that those constitute *potentially* relevant events; it is up to the event monitor to make more refined decisions regarding what is and is not

[2] See at `http://labse.isti.cnr.it/tools/glimpse`

significant and what party to notify, according to the parameters defined by the clients of the monitoring system. Therefore, these local rules do not need to be reconfigured at run-time, which alleviates the need for communication from the CEP (to deploy configuration rules on each node); instead, they just provide "hints" about suspected problems. Accordingly, the provided information may be either used or ignored according to the more dynamic monitoring rules that are active in the CEP at each moment.

Beyond detecting problems, this monitoring subsystem may also be used to detect under-used virtual nodes, which may in turn guide the deployment of new services to reuse such nodes or trigger their removal altogether. Some of this may be accomplished by rules that provide notifications if some resource utilization is *below* some predefined rule. However, inspecting the recent history of each node's resource utilization is much more useful in this regard. We are currently working on a layer dedicated to the collection and usage of such data by making use of Ganglia's *gmetad*, but that is not yet integrated with the rest of the system in the current implementation.

3.1 Business Service Monitoring

The Business Service Monitoring (BSM) [14] is responsible for providing the monitoring functionality that relates to business services and choreographies. It ensures a multilevel service supervision, from the QoS of a service to the global business workflow control, realizing an incremental supervision from a finer to a coarse-grained level.

Basically, we take benefit from the Enterprise Service Bus (ESB) technology to build the BSM architecture on a distributed topology of bus nodes. A BSM node is considered as a bus node, onto which a particular "profile" is added in order to fulfill the requirements of the monitoring of Service Oriented Architecture (SOA). This profile provides an additional administration service, and deploys monitoring components specific for QoS and SLA management as well as for Choreographies.

Figure 2 depicts the overall architecture of the BSM. Specifically, it is composed of a Service Level Monitoring, a Choreography Level Monitoring and a Data Collector components. The BSM is exclusively based on the WS-Notifications standard that brings loosely coupled and event-driven capabilities. We assume that business services are exposed thanks to a middleware, for our case, we rely on the Petals Distributed Service Bus (DSB). The BSM can also be applied to other kinds of middleware, provided that they expose a WS-Notification producer interface.

More precisely, we implement specific interceptors that are able to send reports summarizing the message exchanges between services. We design and implement a report model for formalizing the exchange information. These reports are named Raw Reports, we detail them in Section 3.2.

Furthermore, we rely on the Data Collector that is in charge of the subscription to the middleware. For instance, when a connection to this node is requested,

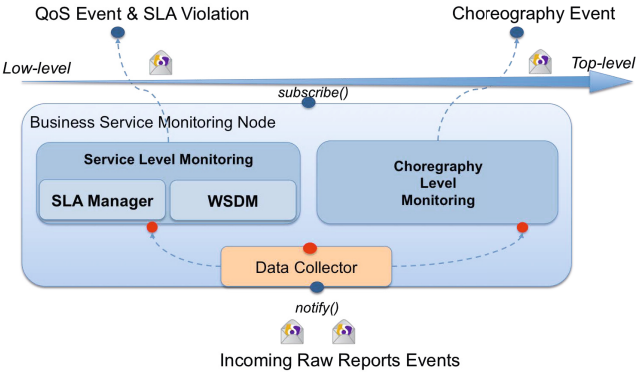

Fig. 2. Business Service Monitoring Architecture

it acts as a WS-Notifications broker. Then, Components that are interested in specific topics subscribe on its events.

In the following, the internal mechanisms of the BSM are detailed. Specifically, Section 3.2 details the interception mechanisms enabling the monitoring when deployed on the middleware. Then, Section 3.3 presents the Runtime Quality Assessment. Finally, Section 3.4 describes the choreography monitoring.

3.2 Interception Mechanisms

Targeting ultra-large scaled environments, the monitoring mechanisms needs to be as transparent and non intrusive as possible. Adopting lightweight and decoupled architectures enhances the ability of the system to be maintained in a flexible way. To that purpose, we adopt an interception mechanism that we deploy in the middleware automatically for enabling services monitoring when activated. Each time an endpoint to a service is created into the middleware, we assign to it an interceptor that will listen to the calls from and to the service.

In each exchange a set of basic information can be extracted. In order to formalize this information, we create a dedicated model that we call Raw Report. Raw Reports contain useful data such as the identifiers for the consumer and the provider, the monitored exchange id, the called operation, a date, the size of the message, and informs if the response is a fault or not. It is important to notice each Raw Report contains two reports, corresponding to the different places the interception is made. Indeed, the interception mechanism allows to perform interceptions at 4 different time stamps (see Figure 3), at the following steps: (i) T1, before the request goes from the client to the provider, (ii) T2, after the request reaches the provider, (iii) T3, before the response goes from the provider to the client, and (iv) at T4, after the response reaches the consumer.

We exploit these Raw Reports in order to assess the services are behaving as contracted in the SLA. In the following section, we detail the operated QoS run-time assessment.

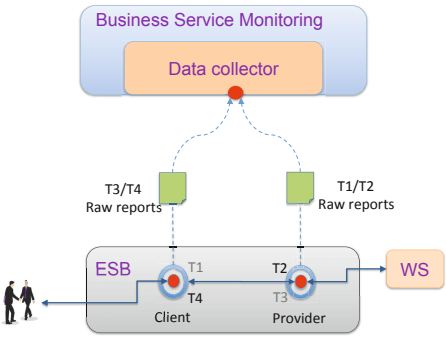

Fig. 3. Raw Reports time stamps illustration

3.3 Runtime Quality Assessment

We rely on a standard based mechanism for ensuring the services run-time quality monitoring and assessment. The use of standards increases the compliance with a wider range of services. We implement standards coming from the Web Services domain, namely, the Web Services Distributed Management (WS-DM[3]) and the Web Services Agreement[4].

We adopt the following process. At the beginning, services contract service level agreements that describe their performances (time, security, etc.). Once deployed, services face the run-time conditions and their real performances may be different from the ones stated in the SLA. More precisely and referring to the BSM architecture in Figure 2, the WSDM Manager receives the Raw Reports from the DataCollector and computes QoS metrics for each service. A prior subscription-based mechanism (with *CreationResource* topic) triggers the creation of a non-functional endpoint in the BSM, for each functional endpoint deployed, then a notification is sent to the BSM when a connected DSB declares a new endpoint. This non-functional endpoint stores current metrics, computed thanks to the gathered Raw Reports. These metrics are updated each time the monitored service is invoked.

The second component dedicated to the Service Level Monitoring is the SLA Manager which is in charge to check if the SLA metrics are being violated. To fulfill this, the SLA component receives the Raw Reports from the DataCollector and checks if a particular exchange is violating an agreement (Service Level Agreement). When an agreement is loaded in the SLA Manager, we define a consumer. Then, an SLA alert can potentially be sent as an upper level monitoring notification. We apply the Common Alerting Protocol (CAP[5]) to formalize the alert.

[3] `http://www.oasis-open.org/committees/wsdm`
[4] `http://www.ogf.org/documents/GFD.107.pdf`
[5] `http://docs.oasis-open.org/emergency/cap/v1.2/CAP-v1.2-os.html`

In addition to the performances and QoS verification, the BSM is also able to detect if a service deployed on a remote machine is answering or not thanks to the interceptions that we realize at 2 times. Indeed, once the T1/T2 report is received by the SLA Manager, a timer is started (provided the agreement is about latency) and if the time waiting the T3/T4 report is higher than the latency defined in the SLA, a first potential alert is sent. If the T3/T4 report is finally received and the effective latency is really violating the SLA, then a confirmation alert is sent. The alert can also be invalidated and the confirmation alert is canceled. The latency computed is the difference between T3 and T2 dates values. This calculus represents the time taken by the service request and response outside the DSB. The times taken between T1 and T2, and T3 and T4 are considered as negligible. However, if a problem occurring in the middleware provokes the violation of the SLA, the service is not considered as the origin of the violation. The collected events are sent to the CEP that identifies the nature of the violation and forwards an alert to a governing decision entity (e.g. a Choreography Governing Board) that would trigger when needed the reconfiguration of the choreography by the replacement of the failed services. Meanwhile, detailing the applied decision mechanism is beyond of the scope of this paper.

3.4 Choreography Level Monitoring

In addition to the Runtime Quality Assessment, the BSM dedicates a component for the monitoring of the choreographies, the Choreography Level Monitoring Manager. It is responsible for the communication level and gathers the messages exchanged within services collaborations and interactions. The main functionality of this component is to ensure that the choreographies are behaving according to the specification policies.

The Choreography Level Monitoring Manager takes as input the choreography specification written in BPMN 2.0 and put interceptors on the services involved in the choreography and exposed thanks to the middleware. The services are event producers that trigger Raw Report as described in Section 3.2.

We realize a correspondence between the choreography model and the monitoring activities generated from the model. A specific structure called MEMB (for Message Exchange Monitoring Behavior) is implemented for this sake. Each MEMB subscribes to Raw Reports, related to the choreography coordination logic. Then, it waits for the expected notifications and checks the several timestamps validity. In case of timeout or not acceptable timestamps, alerts are sent to the CEP 4.

4 Event-Oriented Monitoring

In [13], the authors proposed a distributed event-based monitoring infrastructure called Glimpse, developed with the goal of decoupling the event specification from the analysis mechanism. We reuse Glimpse as a component of the Multi-source Monitoring framework.

Glimpse (see Figure 4) collects a representative set of raw observation data, and then needs to interpret such raw information in order to recognize composite events that may be relevant at higher abstraction levels. The proper combination and correlation of such raw events make it possible either to timely detect unexpected behaviors of the systems, or to predict failures for enhancing system resilience.

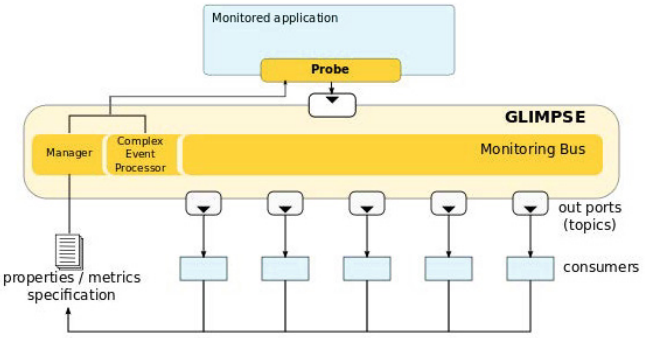

Fig. 4. Glimpse: high-level architecture

Probes are in charge to collect and/or send raw data deriving from the execution of a process within a choreography. Any instance of a Glimpse probe can implement a component in execution at a given observation layer of the software, for example at the infrastructure level (i.e. interacting with Ganglia – see Section 3), or at business level (i.e. interacting with the BSM – see Section 3.1). These two kind of probes can provide data that Glimpse can use to infer complex pattern of events. A detailed description about the communication interfaces defined by a Glimpse probe can be found in [15].

Glimpse implements the data transmission layer by means a publish-subscribe bus, that constitutes the communication backbone conveying all information (events, requests, notifications) flowing among all components. With respect to our Multi-source Monitoring architecture depicted in Figure 1, this bus is implemented by the DSB, as anticipated in Section 2.

The core of the event-oriented monitoring system offered by Glimpse is the CEP. Specifically, the CEP is a rule engine which analyzes the raw events, generated from the probes belonging to all sources (i.e. infrastructure, business service message), and infers complex events matching a set of rules that can be dynamically loaded. In this sense the resilience of the monitored system is enhanced by dynamically adapting to the evolving criticalities of the system. As detailed in [13], the current implementation of the CEP is based on the Drools Fusion rule language[6].

[6] See `http://www.jboss.org/drools/drools-fusion.html`

Finally, the Manager component is the orchestrator of the Glimpse architecture. It manages all communications between the various instances of the Glimpse components. Specifically, the Manager fetches requests received from consumer, analyzes them and instructs the CEP. It then creates a dedicated channel on which it will provide the results produced by the CEP.

5 Case Study

We present one of the use-cases developed within the CHOReOS project. Specifically, it is based on the "Passenger-friendly Airport" [16] scenario, modeling the interactions that take place among different actors (i.e. both services and things) in an airport by means of a set of choreographies. In particular, this case study refers to the monitoring of non-functional properties during the interactions between a Weather Forecast Service (i.e. WF) and a smart-device referred to as Mobile Internet Device (i.e. MID).

In the following we describe how an SLA violation can be due to two possible issues that our Multi-source Monitoring framework contributes to analyze and discover. Specifically, we assume that monitoring the interactions between the WF and the MID as described in Section 3.1, an SLA violation is revealed. Such violation could be due to either the current status of the infrastructure hosting WF, or the implementation of the service WF.

In the former case, the monitoring system reveals that the SLA is going to be violated due either an overload, or a crash of the machine hosting the service. Thus, the rule can suggest a migration of the service on another (more powerful, more reliable) machine in the infrastructure. While in the latter, the monitoring system reveals that the SLA is going to be violated even though the machine is available and is not overloaded. Here, the CEP can notify the redeployment of an updated version of the service WF.

Configuration of the Infrastructure Monitoring: On each created node, we set up the DSB server, deploy the Ganglia *gmond* daemon and the notification mechanism, and inject the address for the Glimpse CEP onto its configuration file. Finally, we deploy the actual services we are interested in. The notification mechanism periodically sends an "alive" message to the Glimpse CEP; this allows Glimpse to detect node failures. Other than that, we set it to notify the CEP whenever the load average of any node goes above 3[7]. While the typical load average on a production system may vary a lot depending on the application, such a value is a common indicator that the machine is overloaded either because of too much I/O or too much CPU utilization. More sophisticated information might be used, but we leave these out of scope for this example.

Configuration of the BSM Mechanism: First, the BSM receives raw reports giving information about each service exchange, in particular involving the WF

[7] On Unix-like systems, the load average is the average number of processes in the queue waiting for processor time in the last minute.

service. This is done thanks to a prior subscription to the ESB middleware on *Raw Report* topic. In addition, a SLA between the WF and the MID services is loaded. This action launches a routine that uses raw reports notifications to check for possible violations. When a violation occurs, the BSM sends a SLA alert to the CEP.

Presentation of the Rules for the CEP: The CEP must be instructed with a set of rules matching the event set that satisfies the monitoring request.

```
<ComplexEventRuleActionList ... >
   <Insert RuleType="drools">
   <RuleName>Machine Overloaded</RuleName>
   <RuleBody>
   ...
    rule "Check for overloads"
   ...
    when
     $aEvent : GlimpseBaseEventImpl(this.serviceID == "WF", this.serviceInstanceID == "
     WF1234", this.getConsumed == false, this.isException == false);
     $bEvent : GlimpseBaseEventImpl(this.machineID == "hubble.eclipse.ime.usp.br", this
     .data == "Machine overload", this.isException == false, this.getConsumed == false,
      this after $aEvent);
    then
     $aEvent.setConsumed(true);
     update( $aEvent );
     retract( $aEvent );
     ResponseDispatcher.NotifyMeValue(... ...);
    end
   </RuleBody>
  </Insert>
</ComplexEventRuleActionList>
```

Listing 1. Check for machine overload rule

```
<ComplexEventRuleActionList ... >
  <Insert RuleType="drools">
     <RuleName>Machine Overloaded No Ganglia Notification</RuleName>
       <RuleBody>
   ...
      rule "Machine Overloaded No Ganglia Notification"
   ...
      when
       $aEvent : GlimpseBaseEventImpl(this.serviceID == "WF", this.serviceInstanceID ==
              "WF1234", this.getConsumed == false, this.isException == false)
                not (GlimpseBaseEventImpl(this.machineID == "hubble.eclipse.ime.usp.br"
                     , this.isException == false, this.getConsumed == false, this after
                     [0s, 30s] $aEvent))
      then
       $aEvent.setConsumed(true);
       update( $aEvent );
       retract( $aEvent );
       ResponseDispatcher.NotifyMeValue( ... ... );
      end
   </RuleBody>
  </Insert>
</ComplexEventRuleActionList>
```

Listing 2. SLA Violation and machine is not answering

```
1  <ComplexEventRuleActionList ... >
2  <Insert RuleType="drools">
3    <RuleName>SLAViolation</RuleName>
4      <RuleBody>
5      ...
6        rule "SLAViolation"
7      ...
8        when
9         $aEvent : GlimpseBaseEventImpl(this.serviceID == "WF", this.serviceInstanceID == "
         WF1234", this.getConsumed == false, this.isException == false);
10        $bEvent : GlimpseBaseEventImpl(this.machineID == "hubble.eclipse.ime.usp.br", this
          after $aEvent, this.data == "ALIVE");
11        $cEvent : GlimpseBaseEventImpl(this.machineID == "hubble.eclipse.ime.usp.br", this
          after $bEvent, this.data == "ALIVE");
12       then
13        $aEvent.setConsumed(true);
14        update( $aEvent );
15        retract( $aEvent );
16        $bEvent.setConsumed(true);
17        update( $bEvent );
18        retract( $bEvent );
19        $cEvent.setConsumed(true);
20        update( $cEvent );
21        retract( $cEvent );
22        ResponseDispatcher.NotifyMeValue(... ...);
23     end
24   </RuleBody>
25 </Insert>
26 </ComplexEventRuleActionList>
```

Listing 3. SLA Violation due to uncorrect service behaviour rule

According to this case study, three rules have been implemented in order to cover the possible behaviors of the services involved. Such implementations correlate the events notified from the BSM related to a violation of an SLA, with the status/overload events coming from Ganglia.

Listing 1 reports a first rule that reveals if any of the machines used in the infrastructure is overloaded. Specifically, in this case the rule matches if two events are triggered to the CEP : the notification of an SLA violation form the BSM about service WF (see at line 9); the notification that the specific machine hosting the service is overloaded (see at line 10). In this case, the body of the rule specifies to consume the event about the SLA notification (see lines 12-14) , and it notifies possibly to an entity (e.g. a Choreography Governing Board) responsible for the correct operativeness of the choreography (see line 15). Similarly, the rule reported at Listing 2, matches if any SLA violation has been notified by the BSM at the service-level, and if one of the machine at the infrastructure level failed to send the "alive-notification" to the CEP. This rule highlights the case that either a machine crash occurred, or anyhow that it is not reachable anymore. Finally, the last rule (see Listing 3) correlates a notification from the BSM that the service WF is violating the SLA (see line 9), with the monitoring of the infrastructure revealing that the machine hosting the service is not overloaded. Specifically, in this case study a machine is not overloaded when the CEP receives at least two "alive" messages after the notification of the SLA violatoin (see lines 10-11). As introduced above, in this case the rule infers that the violation might be due to the incorrect (non-functional) behavior of WF. Note that, in these examples

we used a hard-coded hostname here, but more sophisticated means of defining this or querying a remote list of machines might be feasible. For example, within the scope of the CHOReOS project we will rely on the specific API that the CHOReOS IDRE will provide.

6 Related Work

As we are arguing in this work, also [11] discusses how monitoring in SOA cannot separately address layer-specific issues. In fact, problems that from one layer affect the others cannot be captured and understood. Nevertheless, the work in [11] focused on the monitoring and adaptation of service orchestrations (i.e. BPEL processes) that are deployed onto a dynamic infrastructure. Thus, the solution does not directly refer to distributed choreographies where service aggregation is coordinated in a decentralised way. In this sense, our approach cannot rely on any entity that is specifically responsible of enacting the choreography, or of restructuring and adapting any running instance.

Also in [17], a multi-layered service-oriented monitoring framework that focuses on both the platform and the infrastructure layers is presented. The primary goal of that approach is to collect and aggregate monitored information with regard to specific performance constraints. Two are the main differences with our framework. First, our architecture is based on a distributed service bus implementing publish/subscribe communication mechanisms, while the core of the monitoring framework in [17] is implemented as a centralized orchestrator (i.e. a Globus Service) monitoring all the applications on the connected virtual environments. Thus, our architecture appears more scalable, since the publish/-subscribe paradigm natively allows to adapt the number of component instances by replicating them over the bus. Note that also the DSB can be implemented by federating several distributed instances of a bus. As a second difference between the two solutions is that our Multi-source Monitoring framework explicitly supports a correlation technique that makes use of complex event processing, while the approach in [17] mainly focuses in collecting, and storing the monitored indexes in suitable repositories.

In [18] the authors proposed a monitoring approach that enables autonomous service provisioning in federated clouds. Among the others, the framework was mainly conceived to support either the deployment, or the decommission of the requested services as virtual machines on a specific IaaS. Thus the main difference with our approach is that such solution conceives the sources of the monitoring mainly as layers of the same kind. In our approach we combine events that happen onto different abstraction layers.

Another family of works related to this paper concerns those monitoring solutions that are tailored to capture events at a specific abstraction layer (i.e. the infrastructure level). Accordingly, basic monitoring of hardware system resources is an essential component of virtually any production environment.

There are many reasonably similar monitoring systems available today that focus on hardware resources[8].

Furthermore, a lot of sophisticated monitoring systems that deal with large-scale computing environments such as grids also exist [12,19,20,21]. Some of them have been designed with a specific environment in mind [22,23,24] and, therefore, are either tied to characteristics of these environments or serve some specific purpose within them. Most, however, try to be generic. Accordingly, a proposal for the general characteristics expected of a monitoring system has been prepared [25].

7 Conclusions and Future Work

The FI world challenges the SOA by raising scalability, distribution and heterogeneity issues. In this vision, either the services, or the things are discovered, chosen and bound at run-time. In this context, usually cooperations are regulated by means of choreographies, modeling dynamic and flexible composition of services/things. Nevertheless, as choreographies are abstract specifications, they may include interaction schemas that can evolve after the design phase, so that unexpected events or scenarios may actually take place at run-time.

In this work we presented a Multi-source Monitoring Framework, through which the non-functional properties of choreographies can be kept under observation. Specifically, it supports the observation at run-time of anomalies that are due to phenomena originated from sources operating at different abstraction layers.

Both the overall architecture of the framework, and the application case study are developed within the context of the CHOReOS project. So, as future work we will keep working of the refinement of the implementation we already have, and we are planning to extensively validate our framework by applying it on final version of one the use-cases of ultra-large scale service choreographies that will be released by the CHOReOS project.

Acknowledgements. This paper is supported by the EU FP7 Projects: CHOReOS (IP 257178), NESSoS (NoE 256980), and CONNECT (FET IP 231167).

References

1. Barker, A., Walton, C.D., Robertson, D.: Choreographing web services. IEEE T. Services Computing 2(2), 152–166 (2009)
2. Besson, F.M., Leal, P.M., Kon, F., Goldman, A., Milojicic, D.: Towards automated testing of web service choreographies. In: Proc. of AST, pp. 109–110. ACM, Waikiki (2011)

[8] Popular examples are Nagios (`www.nagios.org`), Big Brother and Xymon (`www.bb4.org`, `xymon.sourceforge.net`), cacti (`www.cacti.net`), and zabbix (`www.zabbix.com`).

3. Bertolino, A., De Angelis, G., Kellomäki, S., Polini, A.: Enhancing service federation trustworthiness through online testing. IEEE Computer 45(1), 66–72 (2012)
4. De Angelis, F., De Angelis, G., Polini, A.: A counter-example testing approach for orchestrated services. In: Proc. of ICST, pp. 373–382. IEEE CS, Paris (2010)
5. Bertolino, A., De Angelis, G., Polini, A.: Validation and verification policies for governance of service choreographies. In: Proc. of WEBIST. SciTePress (April 2012)
6. Bianculli, D., Ghezzi, C.: Monitoring conversational web services. In: Di Nitto, E., et al. (eds.) IW-SOSWE, pp. 15–21. ACM (2007)
7. Campos, J.: Survey paper: Development in the application of ict in condition monitoring and maintenance. Comput. Ind. 60(1) (2009)
8. Hofmann, R., Klar, R., Mohr, B., Quick, A., Siegle, M.: Distributed performance monitoring: Methods, tools, and applications. IEEE Trans. Parallel Distrib. Syst. 5(6), 585–598 (1994)
9. Maia, J.L., Zorzo, S.D.: Socket-Masking and SNMP: A Hybrid Approach for QoS Monitoring in Mobile Computing Environments. In: Proc. of JCC, p. 106. IEEE CS, Washington, DC (2002)
10. Wang, C., Xu, L., Peng, W.: Conceptual design of remote monitoring and fault diagnosis systems. Inf. Syst. 32(7), 996–1004 (2007)
11. Guinea, S., Kecskemeti, G., Marconi, A., Wetzstein, B.: Multi-layered Monitoring and Adaptation. In: Kappel, G., Maamar, Z., Motahari-Nezhad, H.R. (eds.) ICSOC 2011. LNCS, vol. 7084, pp. 359–373. Springer, Heidelberg (2011)
12. Sacerdoti, F.D., Katz, M.J., Massie, M.L., Culler, D.E.: Wide area cluster monitoring with ganglia. In: Proc. of CLUSTER (2003)
13. Bertolino, A., Calabrò, A., Lonetti, F., Di Marco, A., Sabetta, A.: Towards a Model-Driven Infrastructure for Runtime Monitoring. In: Troubitsyna, E.A. (ed.) SERENE 2011. LNCS, vol. 6968, pp. 130–144. Springer, Heidelberg (2011)
14. Lesbegueries, J., Ben Hamida, A., Salatgè, N., Zribi, S., Lorrè, J.: Experience report: Multilevel event-based monitoring framework for the petals enterprise service bus. In: Proc. of DEBS. ACM (to appear, 2012)
15. Bertolino, A., De Angelis, G., Polini, A. (eds.): V&V tools and infrastructure – strategies, architecture and implementation. Number Del. D4.2.1. The CHOReOS Consortium (2012)
16. Chatel, P., Leger, A., Lockerbie, J. (eds.): "Passenger-friendly airport" scenarios specification and requirements analysis. Number Del. D6.1. The CHOReOS Consortium (2011)
17. Katsaros, G., Kousiouris, G., Gogouvitis, S.V., Kyriazis, D., Menychtas, A., Varvarigou, T.: A self-adaptive hierarchical monitoring mechanism for clouds. JSS 85(5), 1029–1041 (2012)
18. Kertész, A., Kecskemeti, G., Marosi, C.A., Oriol, M., Franch, X., Marco, J.: Integrated monitoring approach for seamless service provisioning in federated clouds. In: Stotzka, R., Schiffers, M., Cotronis, Y. (eds.) PDP, pp. 567–574. IEEE (2012)
19. Newman, H.B., Legrand, I.C., Galvez, P., Voicu, R., Cirstoiu, C.: Monalisa: A distributed monitoring service architecture. In: Talk from the Computing in High Energy and Nuclear Physics (2003)
20. Truong, H.-L., Fahringer, T.: SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid. In: Dikaiakos, M.D. (ed.) AxGrids 2004. LNCS, vol. 3165, pp. 202–211. Springer, Heidelberg (2004)
21. Andreozzi, S., De Bortoli, N., Fantinel, S., Ghiselli, A., Rubini, G.L., Tortone, G., Vistoli, M.C.: GridICE: a monitoring service for grid systems. Future Generation Computer Systems 21(4) (April 2005)

22. Boulon, J., Konwinski, A., Qi, R., Rabkin, A., Yang, E., Yang, M.: Chukwa, a large-scale monitoring system. In: Proc. of CCA (2008)
23. Park, K.S., Pai, V.S.: CoMon: a mostly-scalable monitoring system for PlanetLab. OSR 40(1), 65–74 (2006)
24. Wolski, R., Spring, N.T., Hayes, J.: The network weather service: a distributed resource performance forecasting service for metacomputing. Future Generation Computer Systems 15(5-6) (October 1999)
25. Tierney, B., Aydt, R., Gunter, D., Smith, W., Swany, M., Taylor, V., Wolski, R.: A grid monitoring architecture. Memo GFD-I.7. Global Grid Forum (2002)