

Monitoring as-a-service to drive more efficient future system design

Frédéric Lemoine, Tatiana Aubonnet, Ludovic Henrio, Soumia Kessal, Eric Madelaine, Noëmie Simoni

▶ **To cite this version:**

Frédéric Lemoine, Tatiana Aubonnet, Ludovic Henrio, Soumia Kessal, Eric Madelaine, et al.. Monitoring as-a-service to drive more efficient future system design. EAI Endorsed Transactions on Cloud Systems, 2017, 3 (9), pp.1 - 15. 10.4108/eai.28-6-2017.152754 . hal-01582593

HAL Id: hal-01582593
https://hal.science/hal-01582593

Submitted on 6 Sep 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

EAI Endorsed Transactions
on Cloud Systems
Research Article EAI.EU

Monitoring as-a-service to drive more efficient future system design

Frédéric Lemoine[1,*], Tatiana Aubonnet[1,4], Ludovic Henrio[2], Soumia Kessal[4], Eric Madelaine[3], Noëmie Simoni[4]

[1]CEDRIC, Conservatoire National des Arts et Métiers, 292 rue Saint-Martin, 75003 Paris, France
[2]CNRS, University of Nice Sophia-Antipolis, Sophia-Antipolis, France
[3]INRIA, Sophia-Antipolis, France
[4]Télécom ParisTech, 46 rue Barrault, 75013 Paris, France

Abstract

In the services world, the expected benefits are the fastest time to market, lower costs, greater consistency in the application, and increased agility. The reuse and sharing properties of software components are useful to address these challenges. However, to achieve this, it is necessary to be able to observe each service and to control the service composition. This article proposes to rethink the company's organisational process of application development and use the power of monitoring to help the application design. The proposed Monitoring as-a-service (MaaS), whose properties are detailed, will be used for the computation of the offered Quality of Service (QoS), for the services calibration during the service creation phase and to inform the QoS Controller during the operational phase. For effective design, the architect will place MaaS at crucial points of its architecture according to its decision-making process. Finally, we present experimental results and a conclusion ends the paper.

Received on 13 June 2016; accepted on 17 November 2016; published on 28 June 2017

Keywords: Quality of service, Monitoring, As a Service, Service component, Self-control, Service composition

Copyright © 2017 Frédéric Lemoine *et al.*, licensed to EAI. This is an open access article distributed under the terms of the Creative Commons Attribution license (http://creativecommons.org/licenses/by/3.0/), which permits unlimited use, distribution and reproduction in any medium so long as the original work is properly cited.

doi:10.4108/eai.28-6-2017.152754

1. Introduction

Cloud computing and Future Internet promise a new ecosystem where everything is "as a service". Architects mutate to the service-oriented architecture (SOA). The reusability and loose coupling properties facilitate the implementation of applications. Indeed, applications are built through the composition of services that exist today in the enterprise or can be provided by Cloud providers.

No doubt, we are in the era of the services and the service is at the heart of the architecture.

Monitoring is needed to perform business analytics for improving the operation of systems and applications [1] or for verifying compliance with an service level agreement (SLA) contract. There are different types of layers to be monitored: Application, middleware, OS, networks, hardware [2][3][4]. These layers can be seen as where to put the probes of the monitoring system. In fact, the layer at which the probes are located has direct consequences on the phenomena that can be monitored and observed:

- Application, middleware, and OS: bugs, malfunctions, vulnerabilities, etc.
- Network: bandwidth, throughput, etc.
- Hardware: CPU, memory, temperature, voltage, etc.

The measured value in the upper layers (e.g. the performance of the application) may or may not include the values of the lower layers (e.g. the transfer rates on the network). The processing time for a task (top layer) depends on the hardware (lower layer) on which it runs and the load of the virtualised environment.

Each component of service has to be defined, controlled and managed. However, to manage, it is necessary to know the values and metrics of the service:

*Corresponding author. Email: frederic.lemoine@cnam.fr

- Values allow checking service status, triggering an alert and sending a notification related to an abnormal behaviour (out contract), which implies immediate action. This is the supervision and control responsibility.
- Metrics allow logging and observing each measurement point. This is the metrology responsibility.

It is essential that this monitoring, which regroup these two concepts of supervision and metrology, be placed at each service and composition level. Software components provide means to structure service composition and ensure better re-usability, adaptability, and scaleability of services. In our preceding works [5], we introduced this vision of the monitor, by proposing a Self Controlled service Component (SCC).

But, the problems of heterogeneous services, their SLA compliance, and service composition automated management are raised.

For improving the system design and make it more efficient, we need to adapt existing composition models in order to make this design and this automated management converge. For this, we need to answer the following questions:

- What are the properties needed for monitoring services to adapt to heterogeneous environments?
- Where the measurement points have to be placed to have the right information for fast reactions?
- How to know the values and metrics of service in general and of "re-used" service in particular?
- Can we take these problems into account during the design phase?

We show in this paper how the adoption of a component oriented structure helps the service composition to provide a guaranteed quality of service.

Our main contributions are the following:

- We design a generic monitoring component template that can be placed in each hierarchical level.
- We define a calibration technique to compute the nominal/offered quality of service (QoS) and to help their composition.
- We provide a method for the design architect to structure his application (service composition) by respecting SLA compliance.

This paper is organised as follows: The related works of the properties of monitoring systems and their analyses are described in Section 2. Section 3 presents the SCC proposed in the OpenCloudware project [6], but also extended SOA properties and autonomic capabilities of these SCC components. Section 4 is devoted to our propositions for efficient driving, i.e. the advantages of Monitoring as-a-service (MaaS) within SCC architecture, method for design architect, monitoring as-a-service for calibration and design. Section 5 proposes the beginnings of solution for assuring autonomic management of the global service (service composition). A prototype implementation of a single SCC, of an SCC components composition, and their calibrations are proposed in Section 6. Finally, in Section 7, we highlight the advantages of our approach to drive future system design.

2. Related Works

We present here our analysis of the properties related to systems monitoring. These properties must be the same as those of the monitored system (*Scaleability*, *Elasticity*, *Adaptability*, and *Autonomicity*) or system component (*Availability* and *Resilience*). Its integration must be done at a lower cost (*Intrusiveness*, *Comprehensiveness*). The *Timeliness* property is needed for agility and quick decision-making at runtime. We analyse their issues and discuss how they have been addressed in literature.

Timeliness. A monitoring system is timely if detected events are available in time for their intended use [7].

The difficulties are:

- The time between the occurrence of an event and its treatment can vary depending on the measurement, analysis, and the communication delay.
- To obtain up-to-date information, a trade-off between accuracy and sampling frequency is necessary because the shorter the sampling interval, the smaller the delay between the time a monitored condition happens and is captured.
- Analysis is problematic because it can be complex and require computing time to be relevant.
- Communication delay can be a problem if it is necessary to aggregate multiple data sources in order to process them.

Adaptability. Monitoring requires computing and communication resources that can be costly. Adaptability should be used to find the right compromise between accuracy and invasiveness (environmental disruption).

Autonomicity. A monitoring system is autonomic if it is able to self-manage its distributed resources by automatically reacting to unpredictable changes, i.e., if it is able to react to detected changes, failures, performance degradation without manual intervention

[8].
The difficulties are:

- The control loop receives data from a large number of sensors and propagates the action to a large number of actuators, which leads to coordination and scaling difficulties.
- The analytical capacity must be adapted to the complexity of the infrastructure (Layers)
- It is difficult to implement steering policies that respond adequately to events detected by the monitoring system.

Elasticity. Elasticity consists of coping with dynamic changes of monitored entities (created or destroyed by expansion and contraction) [9].
Types of changes are:

- New assignment of resources for the user.
- Change in the monitoring needs for the user.
- Change of the number of users.

Intrusiveness and Comprehensiveness. A monitoring system is intrusive if its adoption requires significant modifications of the monitored system [10].

A monitoring system is comprehensive if it supports different types of resources (physical and virtualised) and is multiple tenants [11]. The latter requires:

- To adopt a single monitoring API regardless of the measure that is currently used.
- To deploy and maintain a single monitoring infrastructure.

Having a low Intrusiveness minimises cost instrumentation.
The difficulties are:

- Comprehensiveness requires supporting different underlying architectures, technology, resources, and multi-tenancy.
- The heterogeneity of resources and settings of the different layers.

Resilience and Availability. A monitoring system is resilient if it can support a number of faulty components while continuing to operate normally.

It is available if it provides services according to the system design whenever users request them [12].

A system must be resilient and available at least for reasons of payment, SLA compliance, and resource management.
The difficulties are:

- Services can be migrated from a physical computer to another, striking down classical monitoring logic and affecting the reliability of the monitoring system.
- Because of the complexity of tracking and managing heterogeneous monitored and monitoring resources, we should take into account possible faults of the monitoring system itself.

Table 1. Platforms comparative

Platform	Properties	Multi-Layers
AzureWatch [13]	Scaleability, Adaptability, Autonomicity	Yes
Boundary [14]	Timeliness, Resilience, Availability	Yes
CloudClimate [15]	Timeliness, Resilience, Availability	No
CloudCruiser [16]	Timeliness, Resilience, Availability	No
Cloudfloor [17]	Timeliness, Resilience, Availability	No
CloudHarmony [18]	Timeliness, Comprehensiveness	No
CloudSleutch [19]	Timeliness	No
CloudStack ZenPack [20]	Timeliness	No
CloudWatch [21]	Elasticity, Timeliness	Yes
Cloudyn [22]	Timeliness, Resilience, Availability	No
Consul [23]	Availability, Scaleability	No
Dargos [24]	Adaptability, Intrusiveness	No
New Relic [25]	Timeliness, Resilience, Availability	No
Sensu [26]	Availability, Scaleability, Comprehensiveness	No
Up.time [27]	Timeliness, Resilience, Availability	Yes
VR. Hyperic [28]	Timeliness	No

Scaleability. The aim of a scalable monitoring system is to manage a large number of probes [9]. A system is scalable if it is able to efficiently collect, transfer, and analyse large amounts of data without affecting the functional part.
The difficulties are the large number of parameters to be monitored and the large amount of data from multiple distributed locations to aggregate and filter.

The table 1 show a comparative of different platforms according to their properties.

In the following, we present different works, described in literature, aimed to satisfy or improve preceding properties.

To improve **Timeliness**, [29] proposes a behavioural model to predict the best measurement time interval. [7] reduces the time of analysis and communication by assembling and processing information of near nodes and by adapting the analysis and communication topology.

Concerning **Adaptability**, [7, 10, 30–32] propose to fine-tune the amount of monitored resources and the monitoring frequency. [30] proposes to predict the resource consumption for adapting the time interval to push monitoring information Monalytics [32] configures its agents in real time depending on the monitoring topology (collect, process, and transmit)

by providing new analysis and monitoring codes or by changing the methods being used.

For **Autonomicity**, focusing on bottlenecks, [33] proposes two methods to detect and resolve them as well as the identification and reduction of resources if too many have been provisioned. These methods require a maximum response time and are useful for the SLA compliance. [34] proposes a monitoring system based on agents having the ability to continuously check the status of virtual machines (VM) and to restore them in case of malfunction. [35] allocates computing resources to services and deploys them on virtualised infrastructures. [35] detects violations of SLAs and offers automatic dynamic reactions combining low-level resource metrics with service level objectives (SLO) and a knowledge base for the analysis of monitoring information.

Concerning **Elasticity**, most of the tools were designed for slow changes of the physical infrastructure (Ganglia [36], Nagios [37]) and do not support rapid and dynamic changes. They use a push strategy (the physical host notifies the tool on the status and the presence of the running VMs) [38] or publish-subscribe to decouple communications ends and thus to support dynamism. An hypervisor controller checks the list of virtual execution environment (VEE) and add or remove a monitor according to the detected number [9]. An extension of Nagios [38] allows the use of active verification method (pulling) by remote code execution. An extension of Nagios [10] offers a push-pull model. The monitoring information is sent by agents to a Manager (push) and information consumers can obtain data from it (pull). Monalytics [32] was designed for scaleability and efficiency in highly dynamic scenarios: discovery at runtime of resources to monitor and configuration at runtime of monitoring agents. Brokers at different hierarchical levels, collect process and transmit the monitoring information.

To improve **Intrusiveness** and **Comprehensiveness**, [11] proposes an architecture based on agents that monitor directly the flow of information through the same workflow system. They are connected with adapters, which abstract from data of a specific technology. [39] monitors events at the VM level. Sensu [26] provides built-in metric translation that allows you to collect metrics in various formats from disparate data sources, and mutate them into a proprietary intermediate format that has been optimized for portability.

In the literature, several works search for the reasons impacting **Resilience**: Resource Volatility [30, 40], virtualisation technology [34]. To improve the **Availability**, [41] provides a publish-subscribe paradigm for communication and a set of redundant brokers for events management while providing tolerance to attacks and malfunctions. Consul [23] is a distributed, highly available system. The agents talk to one or more Consul servers. The Consul servers are where data is stored and replicated to avoid failure scenarios leading to data loss.

To ensure **Scaleability**, two methods are commonly used to reduce the amount of data collected by the controller:

- Data aggregation consists of combining multiple metrics into a single one,
- Filtering avoid spreading unnecessary data to the Controller.

Most of the proposed architectures use a subsystem to propagate event announcements [7, 10, 11, 42] or agents for collecting, filtering, and aggregate data [10, 11, 23, 26, 31]. Sensu [26]'s use of the publish/subscribe pattern of communication allows for automated registration and de-registration of ephemeral systems, allowing you to dynamically scale the infrastructure up and down. By capturing changes only, Consul [23] reduces the amount of networking and compute resources used by the health checks, allowing the system to be much more scalable.

Although each property has been addressed in various studies presented above, no platform includes them all to the best of our knowledge. We think that a monitoring and an analysis placed close to each functional component would have many advantages:

- The volume of data exchanged and thus the communication resources would be extremely low since the analysis would be done on site. Only its result would be sent.
- The code would be simplified and hence require less computing resources (Adaptability).
- The analysis would be faster, more relevant, and reaction times would be minimised (Timeliness).
- At each addition / removal of a functional component, a monitoring and controlling component would be therefore added / removed (Scaleability, Elasticity).
- Monitoring and controlling component would be located at any hierarchical level: In the same place as any functional component.

We would not be intrusive if monitoring and analysis were external to the functional component. (Intrusiveness). A generic monitoring and analysis independent of the functional component would be comprehensive (Comprehensiveness) and might be present at all levels of architecture.

We will show how such a system would also be a valuable aid to the application design for the architect. This one could experience during the design and before

being put into production if its composition is properly sized i.e. whether resources will be sufficient to operate and meet the requested QoS.

Our motivation is thus double:

- Show that our MaaS, by its design, responds to most of the preceding properties.
- Show that it can also be used to help the architect to choose the best component when designing his application.

3. Background

In the service era, the service is the centre of architecture, to enjoy all the benefits expected from this concept, we have proposed in [43] a component called SCC, which we recall the description (Section 3.1) with SoA extended properties (Section 3.2) and autonomic capabilities (Section 3.3).

3.1. Self-controlled service Component

To describe the behaviour of our components and permit homogeneous QoS management, we define a generic QoS model [44]. Four criteria are proposed to describe the QoS: availability, integrity, time, and capacity.

- Availability represents the accessibility rate of the service component (for example : accessibility rate).
- Integrity represents the capacity to run without alteration of information (for example : error rate).
- Time represents the time required for request processing (for example : response time).
- Capacity represents the maximum load the service component can handle (for example : processing capacity).

This revealed to be useful and sufficient in all the practical cases we studied.

To increase the structural decomposition and the reuse of non-functional QoS components, we have separated its internal functions and proposed an architecture that separates the monitoring and QoS functions of the remaining functions called "control". We have specified this model in the OpenCloudware project [6] to address the behavioural aspects through QoS.

The membrane of our SCC includes (Figure 1):

- Input monitoring (InMonitor) and output monitoring (OutMonitor) components. They play an interceptor role. Incoming service requests are intercepted and transmitted (unchanged) to the functional component via the corresponding internal interfaces. The OutMonitor intercepts outgoing service requests. They provide measurement information on the flow they intercept.
- A QoS component (QoSControl), associated with the business component.
- A non-functional interface (client) for QoS control (IQoSStatus), by which it will send the information of violation of QoS contracts, i.e. "InContract" notifications when the behaviour is compliant with the contract or "OutContract" otherwise.
- A non-functional interface (server) of configuration (IConfigQoS, IConfigMonitor), whose role is to receive component configuration commands.

The QoSControl component checks the current behaviour of the resource and its conformity with the contract. For this, it triggers a timer and regularly requests to the monitors (InMonitor and OutMonitor) the parameter values (getValues method) of the IControlMonitor interface (Figure 2). It compares each current value to the corresponding threshold value not to exceed. It sends an OutContract notification if the current value is less (or more) than the threshold value; in this case the dynamic management consists of replacing on the fly the failing component by a ubiquitous service fulfilling the requirements. Otherwise, it sends an InContract notification. We define two types of QoS:

1. The requested QoS: client side, SLO.
2. The offered QoS also called nominal QoS is computed under resource conditions of the underlying level: provider side, SCC components based.

The QoS requested by the customer is provided by catalogue components with an offered QoS and/or components with adaptation mechanisms (SCC+). SCC+ component is indeed necessarily a composition. The provider responds to the client's request (requested QoS) by establishing a user session based entirely on SCC and SCC+ components.

We obtain a SCC component, self-monitoring and self-controlling Component. The sub-components of the membrane (monitors and QoS) are activated in order to perform monitoring of the quality of service and to notify its degradation.

3.2. Extended SOA properties

We based on the recommended service SOA with the properties of description, invocation, autonomy, reuse, and loose coupling. In [5], we have added the

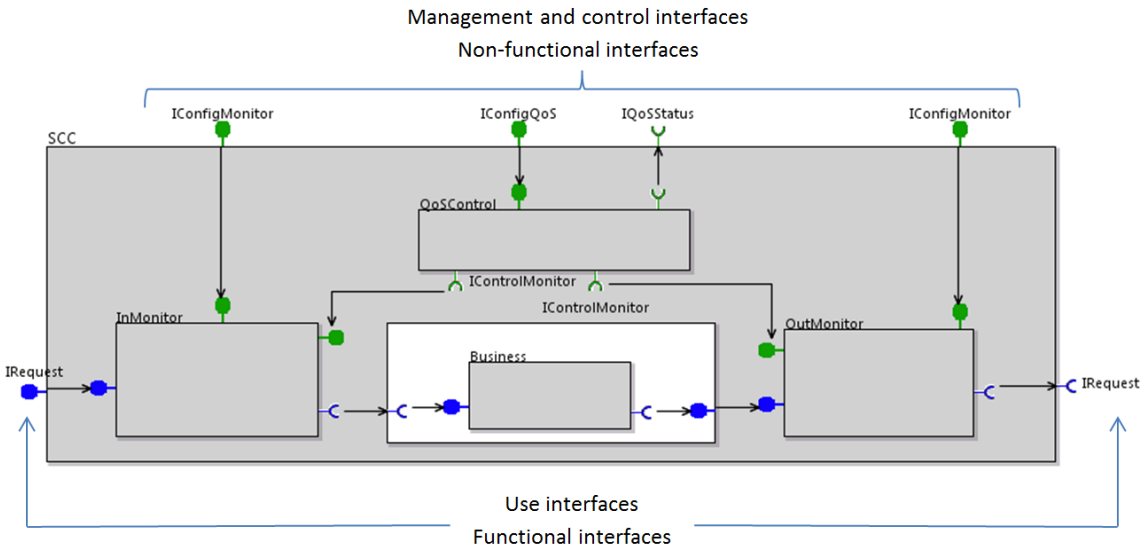


Figure 1. Self-Controlled service Component (SCC)

following properties: stateless, mutualisation, ubiquity, and exposability. These properties, named SOA+, allow exposing components in a library (catalogue), sharing components for use in different applications, and assembling them in a personalised session.

In this article we focus on the properties that the architect/developer must particularly take into account:

- Autonomy, which will be presented in Section 3.3.
- Reusability: A service has an agnostic logic and thanks to this can be positioned as a reusable resource.
- Composability: A service has to be designed so that they can be used in a service composition. This property is used via system information blocks (SIB) in Intelligent Network of Telecommunication services ([45]).

3.3. Autonomic capabilities

GCM/ProActive [46] is the component platform we used for out experimentation. What motivates this choice is the design of the component model imposing a strong encapsulation between components. In GCM/ProActive each component is seen as an autonomous entity in a much service-oriented manner. GCM/proactive enforces a strong separation of concerns, well separating the component management from the functional behaviour of the components [46]. It also revealed efficient for implementing autonomic services.

In the Grid Component Model (GCM) [47], a structure is defined for the membrane elements: the non-functional part of the component can thus be defined as an assembly of components. These components can then be connected with other components within the same membrane or with non-functional interfaces of other components. This structure has been precisely and formally specified [48, 49].

4. Towards an efficient driving

In cloud computing, services platforms, and the Internet of Things (IoT), the component is the cornerstone. Each component is responsible for its action. It can belong to several providers. It's chosen according to his contract. Each application (service composition) responds to a client request based on the resources and possibilities of its environment. So, the questions are: How to help the service provider to calibrate their service? How to help the application architect designer?

In the next sections, we will present the advantages of our monitoring service and our SCC architecture (Section 4.1) and we will show how a calibration technique based on the SCC (Section 4.3) can help the service provider to create his catalogue (Section 4.4) and the design architect to compose his application (Section 4.2 and 4.5).

4.1. The advantages of MaaS within the SCC architecture

As presented on the section 3.1, an SCC component includes a monitoring and an analysis for each functional component. Furthermore, the monitors and

the QoScontrol surround it and are close to it. The SCC architecture covers most of the properties related to monitoring systems defined in section 2. Indeed it has many advantages:

1. Our SCC component allows self control inside by signalling malfunction (out contract) and automatic reacting outside (*Autonomicity*).
2. The code is simplified and hence requires less computing resources (*Adaptability*).
3. The analysis is faster, more relevant, and reaction times are minimised (*Timeliness*) because we are as closely as possible to the functional component.
4. Monitoring and controlling components are:
    - Generic so they are independent of the functional component (*Comprehensiveness*) and may be present at all levels of architecture.
    - Not intrusive because they are external to the functional component because they are inside the service component membrane and operate in parallel with the functional component. They have no effects on the second (*Intrusiveness*).
5. At each addition/removal of a functional component, a monitoring and controlling component is therefore added/removed (*Scaleability*, *Elasticity*).
6. The volume of data exchanged and thus the communication resources are extremely low since the analysis would be done on site. Only its result would be sent.
7. We measure a QoS (Section 3.1) of each component (hardware of software) allowing better diagnostic of various malfunctions whereas most existing tools monitor network traffic or CPU usage when they should monitor the functional component performance.

In the following section we propose a method compatible with the objectives of self-control, i.e., dynamic reaction as well as the management of service composition.

4.2. System design: Method for design architect

By principle, component oriented programming requires the programmer to think about re-use and sharing properties of software components at the time of their creation. Here, we push this methodology further and require the application provider to also consider QoS and monitoring purposes at design time, which modifies the company's organisational process. So, we propose a new method based on SCC components to help the design architect when choosing the best component and designing his application.

This method has four steps:

1. We begin with the calibration of SCC components. The technique, using self-tests, described in Section 4.3, consists, for a SCC component, to evaluate his nominal/offered QoS and threshold value under resources conditions of the underlying level.
2. Secondly, the service provider creates his catalogue by putting into the preceding calibrated SCC components (Section 4.4).
3. Thirdly, to design an application or service, the architect chooses multi-tenant SCC components in provider's catalogue, based on the specified nominal/offered QoS and thresholds value. He calibrates the composition (SCC+) with the same technique described in Section 4.3 to also obtain the nominal QoS and threshold value of the full composition. If the composition is entirely SCC composed then it can be put in a catalogue too.
4. Finally, in Section 4.5, we propose SLA management actions to ensure the adequacy of his nominal QoS to the requested QoS (SLO).

4.3. Monitoring as-a-service for calibration

As mentioned in Section 4.2, the calibration concerns a single SCC that is intended to be placed in the catalogue's provider or a composition of SCC components. The calibration consists to compute their offered/nominal QoS and their associated threshold value.

First, we focus on the calibration of a single SCC. Our SCC component includes 4 membrane localised non-functional subcomponents: InMonitor, OutMonitor, QoSControl, and Self-test (Figure 2). The two monitors surround the business.

The offered/nominal QoS is obtained by a self-test procedure triggered by the QoSControl. The self-test procedure is performed in a closed loop (in-situ). The InMonitor no longer receive requests from outside during the self-test phase but N intelligently chosen queries (for example for their complexity, computing time or resource consumption) are generated to compute the values of QoS. We are no longer dependent on the outside for obtaining the QoS and the measurements obtained are more reliable.

In a normal utilisation, each external request is intercepted by the inMonitor, which records it whereas in self-test situation, each request is generated by the Self-test component, which is recorded too by the InMonitor. The request is then processed by the

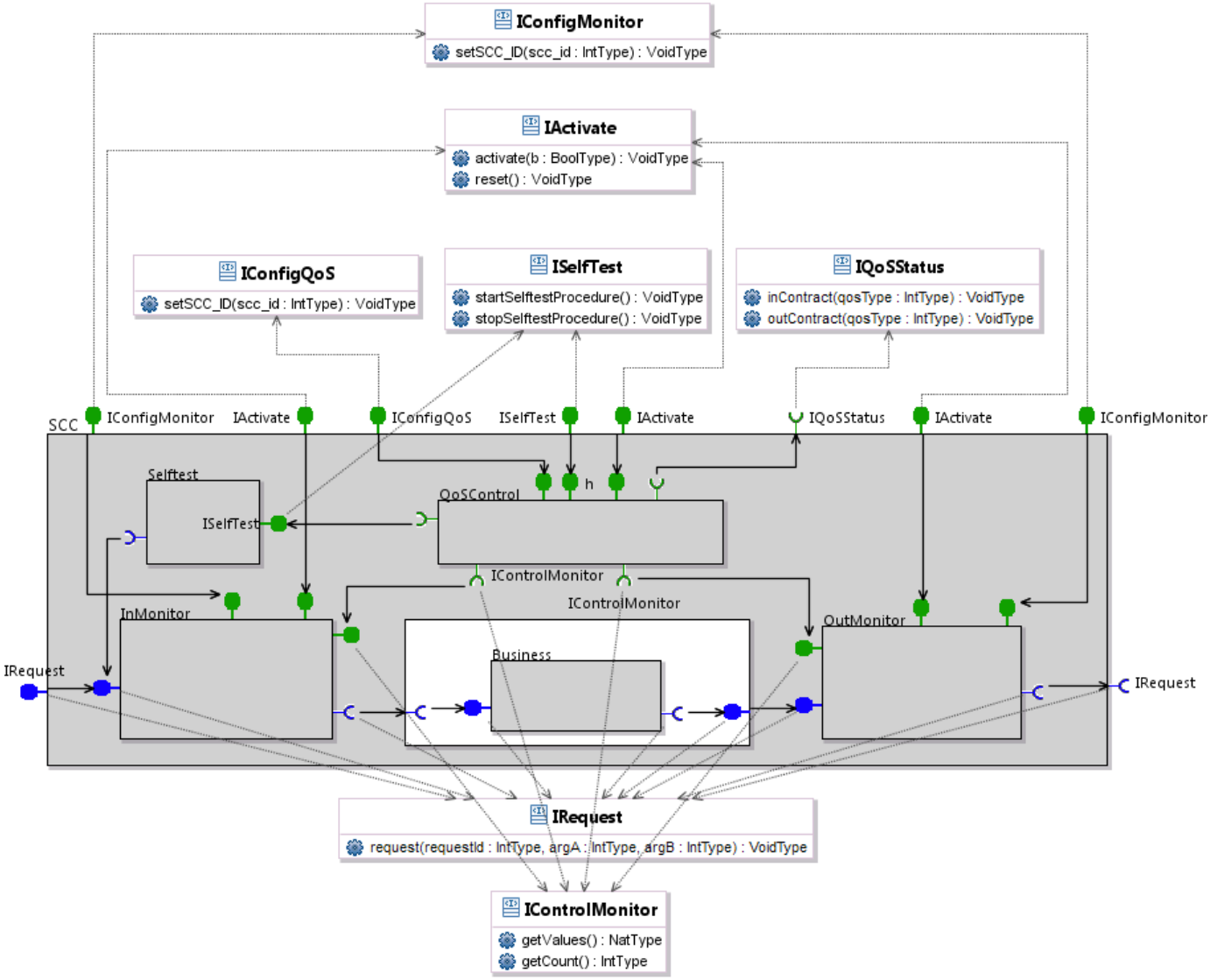


Figure 2. Self-test SCC component.

business code and the result is intercepted by the OutMonitor, which records it.

We measure a QoS (Section 3.1) from raw data. The number of incoming or outgoing requests (queries, packets, primitive) and their timestamp are recorded by the two monitors. The QoSControl periodically ask the monitor for their records. It can detect if a request has been processed by the business component or not and can compute the number of processed/unprocessed requests and the processing time by subtracting the out and in time stamps. The QoSControl can compute other metrics like for example, the availability of the component or the number of processed request by minutes and consider richer model like moving averages. In a normal situation, it checks compliance with the SLA by comparing the result with a reference threshold and send an in or out contract. In the self-test procedure, it's used to compute the offered/nominal QoS and the threshold values from which the business component stops responding by gradually increasing the numbers of requests. The obtained QoS are given on resources conditions because they depend on their environment. Reference tests can also be processed by modifying the resources to highlight the effect of the environment on the measures.

Second, we focus on the calibration of a composition (SCC+). The same procedure can be used for a composition of SCC components. A composition includes two surroundings monitors and QoSControl, the self-test procedure computes the nominal QoS and the threshold value of the composition with the same method as for a single SCC. We determine the

threshold value from which the business component stops responding by also gradually increasing the numbers of requests.

4.4. Catalogue

The catalogue is a showcase for reusable components. The architect chooses them according to their QoS. But as we mentioned, for reusing, it is better to know the offered QoS and the needed resources to provide this QoS. Indeed, for the same functionality, different algorithms and treatments may be used and therefore different QoS are provided. The consumed resources are not the same.

That is why, the provider's catalogue is filled with SCC calibrated components. If a composition is entirely SCC composed then it can be put in a catalogue too. Each component is given with his offered/nominal QoS and the associated resources conditions. Each component, located at the layer N, depends on the QoS of the layer N-1. A component located at the lower layer depends on hardware resources (CPU, RAM).

4.5. Monitoring as-a-service for design

Based on SLA, with SCC reusable components selected from the catalogue, the architect and/or developer build the desired application by composing services. In a normal utilisation, each external request (user transaction) is intercepted by the inMonitor/outMonitor of highest level, which records it. The QoSControl checks compliance with the SLA by comparing the result with a reference threshold and send an in or out contract. But between the behaviour of the composition from end to end (application) and that of each SCC component, there are several subsets, which are the responsibility of the architect.

The recommended method, as the decision process progresses, is to progressively build SCC composites with a new membrane containing the InMonitor, OutMonitor, and QoSControl (Figure 5). So the MaS will be the cornerstone of the design of the application structure. Thanks to our MaaS and the architecture chosen by the architect, self-control helps to locate the root cause of malfunction but how to ensure that the reactions remain QoS compliant? How to guarantee service continuity and have an efficient driving? We now have to look to self-adaptation.

5. Autonomic management for QoS compliance

Autonomic adaptation after the detection of an OutContract event is out of scope of this paper but we wish to expose some directions to explore. In the next sections, we will present some reference works (Section 5.1) and the advantages offered by our solution for a self-adaptation as-a-service (Section 5.2).

5.1. Autonomic adaptation

Generally, the adaptation procedure and management decisions can be structured as a Monitor-Analyse-Planning-Execute-Knowledge (MAPE-K) loop for autonomic computing [50–52].

In the literature several solutions have been proposed to perform autonomic adaptation and make suitable adaptation decisions. In [53, 54], Garlan et al proposed the Rainbow framework which provides general, supporting mechanisms for self-adaptation which can be customised for different systems. Rainbow is based on one large control loop that is in charge of all activities related to the self-adaptability issue for the whole system. [55] propose a framework for Self-Adaptation of distributed services, enables the dynamic evolution of service-based architectures by providing all the functionalities of the MAPE model.

5.2. Self-adaptation as-a-service

Having an efficient driving is not easy because efficiency is architect and user dependent. It has to be personalised. Management has to be considered as-a-service. We want to highlight that with our architecture and according to the choices of the architect, a QoS based MAPE loop can be put in the membrane of each component or similarly, at the top of any composition. We have the capability to monitor a composition from end to end and thus the usability perceived by the user. The architect puts a QoS based MAPE loop at any location it considers appropriate and he wants to manage. They are locations where he wants and can make decisions. Root cause analysis is then simplified because we can be as close as possible to a business component if we wish to.

As Services can be geographically distributed, the cause of a faulty composition may be their internal nodes or links. In case of a composition, QoSControl and Monitors may be also geographically distributed. Note that communications between Monitors and QoSControl use another route than the business services. This way a network communication problem in the first has no incidence on the second. Similarly, note that history of the metrics being collected can be retained, but in that case, because of the stateless property of the SCC component, it will be stored by an external dedicated service. OutContract event gives us the faulty component so constitution and record of a history is optional because it has no influence on the decision-making.

Concerning autonomic adaptation, the analysis of the composite is complex and is still an open issue, however, some cases are simplified. Namely, if the OutContract come from a primitive component, it can be replaced automatically [44, 56–58]. If we have only

InContracts from primitive components and one OutContract from the final composition, then the composition is faulty (links). The location of the faulty component is simplified. When some corrective action is required within a self-controlled application, the Analysis component of its MAPE loop receives the notification (OutContract) from some QoSControl components, and sends the diagnostic to the Planning component. This one will request either a replacement SCC component [58], or some additional service instances, to the external management environment. According to the number of requests, the system is elastic and can add or remove service instances (Consequently, communication, network, CPU, memory resources). When receiving back the requested resources, it will build a reconfiguration script, and pass it to Execute. GCM/ProActive [46] provides a framework for structuring the elements of the MAPE loop (Monitoring, Analysis, Planning, Execution) as components embedded in the component membrane and easing the programming of autonomic adaptation procedures. These MAPE loops can act at the any level of the composition, but also interact through the hierarchical nature of GCM applications.

In conclusion of this section, we show that we know to locate the problem accurately and timely (Timeliness properties) and to send the notification, generating decision-making, to the right place.

6. Implementation

In this section, we bring our SCC component on the Proactive Platform. GCM/ProActive is a Java library that includes a component model and has a strong support for large-scale distributed execution of programs. It relies on an active-object pattern for the interaction between the different entities (i.e. components). According to what has been described in the method (Section 2), we present the experimentation of two calibrations: Firstly for a single SCC (Section 6.1) and secondly for SCC composition (Section 6.2).

6.1. From design to configuration (experiments for calibration)

As a reminder, the offered/nominal QoS is obtained by a self-test procedure triggered by the QoSControl (Figure 2). The self-test procedure is performed in a closed loop (in-situ). The InMonitor no longer receive requests from outside during the self-test phase but N intelligently chosen queries are generated to compute the values of QoS.

For the specification, verification and validation of the architecture of applications built from SCC components, we use the VerCors platform from INRIA [59]. Components can be connected with other components within the same membrane or with non-functional interfaces of other components. Having a tool-supported methodology is important for the design phase, when the designer builds his application, using functional components as basic bricks, and assembling them into compositions. The VerCors Component Editor (VCE) (Figure 3) helps the user to specify the architecture of an application, the interfaces, and the behaviour of assembled components. Furthermore, the tool can generate executable code containing the whole architecture description and the skeleton of the final application. Several validations are performed like structural coherency aspects of the application model for ensuring that the code generation will terminate correctly, and that the code will not fail during deployment of the application components. A library of components integrating the non-functional aspects (Monitors and QoS-Control) is provided. These components are instantiated by the application developer. A set of files is generated allowing the deployment of the application like Architecture Description Language (ADL) for the Architecture Description. These files are then used to build an executable application that can be executed within the GCM/ProActive [46] execution environment.

Each implementation includes five steps:

- Diagram design on VCE with classes and interfaces
- Checking of the validity of the diagram
- Generation of the ADL file and code template of classes and interfaces.
- Creation of a proactive project with enriched code
- Execution of the application

We are going to create a composition based on two SCC. The first service component performs authentication of users based on Digest Access Authentication (challenge-response codes) [60]. The second is responsible for checking the right a user has of doing some actions. The user should provide the good "response" code in his request to prove it is authenticated otherwise the first SCC sends 401 Unauthorized asking him to authenticate first.

First, we focus on the calibration of only one SCC (Figure 2). As mentioned earlier, for this implementation, the business role consists here to perform a Digest Access Authentication. Note that service tasks could be of any nature and can cover a lot of domains as computer vision systems and image processing, signal processing, web services, internet of things services, etc. Furthermore, VCE has already been used to build a real application called Springoo. It's a web application that conforms to

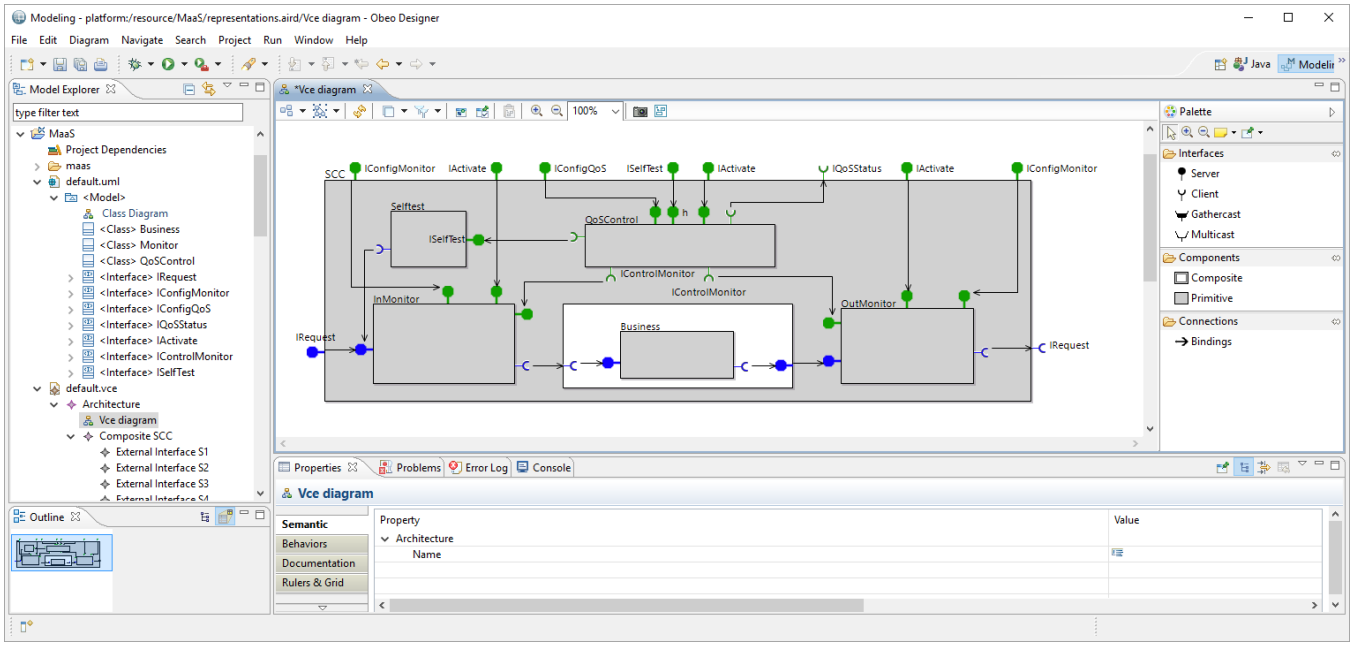


Figure 3. VerCors Component Editor.

the three-tier Java Enterprise Edition (JEE) platform architecture, providing typical commercial web services through an Apache/Jonas/MySQL architecture. This application is one of the end-to-end case-studies of the OpenCloudware project[5].

In self-test situation, each request is generated by the Self-test component, which is recorded by the InMonitor. The record consists of the request number and a time stamp. Note that several different metrics could be taken into account by the same monitor for a more complex QoS processing. The request is then processed by the business code and the result is intercepted by the OutMonitor, which records it. The request is based on a generic Interface: IRequest, which includes his number (requestId) and a list of functional parameters for the business code. The InMonitor, OutMonitor, and QoSControl can be (un)activated via the IActivate interface (activate(b: boolType) method). Their records can be erased via the reset() method. The self-test procedure is triggered/stopped by the call of the startSelfProcedure()/stopSelfProcedure() method of the ISelf-test interface. Each non-functional subcomponent receives the number of his community (setSCC_ID() method) via their IConfigQoS interface.

The QoSControl is a thread, which periodically ask the monitor for their records. It can detect if a request has been processed by the business component and can compute the processing time by subtracting the out and in time stamps. In the self-test procedure, QoSControl computes the offered/nominal QoS and the threshold values from which the business component stops responding.

Then, we explain the logs of the self-test experiment. As already mentioned, we consider firstly a single SCC component. The business role consists to process a mathematical task. The logs highlight the following events in order:

- The thread of the QoSControl starts.
- The Self-test component sends five requests to the business component via the InMonitor.
- The business receives five requests having the identifier 0,1,2,3, and 4 to process.
- The OutMonitor see five results coming from the business component.
- The QoSControl ask the two monitors their tables of recordings.
- It hears that five requests have been recorded. They have successfully been processed by the business.
- The QoSControl get the records from the InMonitor and OutMonitor (time stamp and requestId).
- The QoSControl compute the processing time.

Note that ProActive supports multi-active objects that allow a single active object to execute several requests in parallel if they do not conflict. This is particularly useful here to ensure that the main flow of requests is handled efficiently while not conflicting with the rest of the behaviour of the monitor.

By repeating the operation and by increasing at each time the number of requests, we can compute the average processing time for a given physical resources level (Figure 4, see curve Single SCC).

Sample:

- Number of requests: 220
- Total processing time: 26814 ms
- Average processing time per request: 121.8 ms

Given physical resources:

- Memory: 681616 bytes
- CPU: Intel Core i5 3.6 GHz

By still increasing the number of requests we determine the threshold value from which the business component stops responding. Here for 250 requests. The service provider choose a nominal value, which may be defined, for example, at 70 % of the threshold value: 71.2 ms for 175 requests. This experiment show how to use the self-test procedure to compute the offered QoS and the threshold value from which the business component stops responding. This ends the first step of the method for design architect described in Section 4.2. After the calibration of the component is complete, it could be put in the service providers' catalogues (second step).

6.2. Towards the desired architecture (experiments for control)

Second, we focus on the calibration of a composition (SCC+) including an authentication and authorisation SCC. As mentioned in the third step of the method (Section 4.2), to design an application or service, the architect chooses multi-tenant SCC components in providers' catalogues, based on the specified nominal/offered QoS and thresholds value. He calibrates the composition (SCC+) with the same technique described in Section 4.3 to also obtain the nominal QoS and threshold value of the full composition.

The composition is given at the figure 5. Two chained SCC components are included in an SCC component called "Composition". This composition includes 6 monitors and 3 QoSControl. Thanks to the two surroundings monitors and QoSControl, the self-test procedure computes the nominal QoS and the threshold value of the composition (Figure 4, see curve Composition).

We determine the threshold value from which the business component stops responding. Here for 250 requests. The service provider choose a nominal value, which may be defined, for example, at 70 % of the threshold value: 146.6 ms for 175 requests. This experiment shows that the self-test procedure is useful too to compute the nominal QoS and the threshold value for a SCC component composition. As mentioned above, if the composition is entirely SCC composed (as is the case here) then it can be put in a catalogue too.

7. Conclusion

In this article, we stood from the point of view of an architect or developer in the new ecosystems that refer to paradigms of Cloud, SOA or IoT and are based on the properties of the "service". We started from the reuse property by advocating the SCC component. Thus, during the design of an application or a composite service, the architect and/or the developer will be able to select from a catalogue, for example that of the cloud supplier, the desired(s) service(s) according to the exposed features and associated QoS. The SCC component integrates, during the operation, the control of the contract compliance. Furthermore, the most significant proposed help is the use of MaaS to drive more efficiently the design process. Our MaaS as specified allows designers to: (i) assess the offered QoS during the service creation, (ii) test the offered QoS through the catalogue in its deployment environment, (iii) structure, in terms of decisional process, the composite services by placing MaaS at the crucial points of the architecture of the application and allow the control of the SLA contract. All these elements are integrated within a new method for the design architect.

8. List of abbreviations

- ADL: Architecture description language (Section 6.1)
- GCM: Grid component model (Section 3.3)
- IoT: Internet of Things (Section 4)
- MaaS: Monitoring as-a-service (Section 1)
- MAPE: Monitor-analyse-planning-execute (Section 5.1)
- QoS: Quality of service (Section 1)
- SCC: Self Controlled service Component (Section 1)
- SIB: System information blocks (Section 3.2)
- SLA: Service level agreement (Section 1)
- SLO: Service level objectives (Section 2)
- SOA: Service oriented architecture (Section 1)
- VCE: VerCors Component Editor (Section 6.1)
- VM: Virtual machine (Section 2)

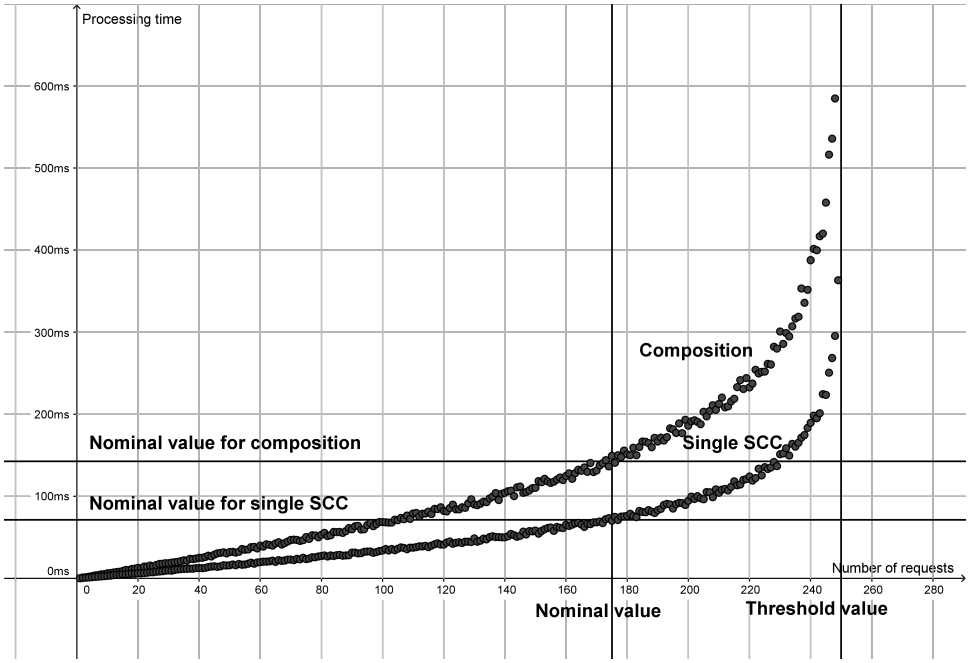


Figure 4. Number of requests and processing time for the composition.

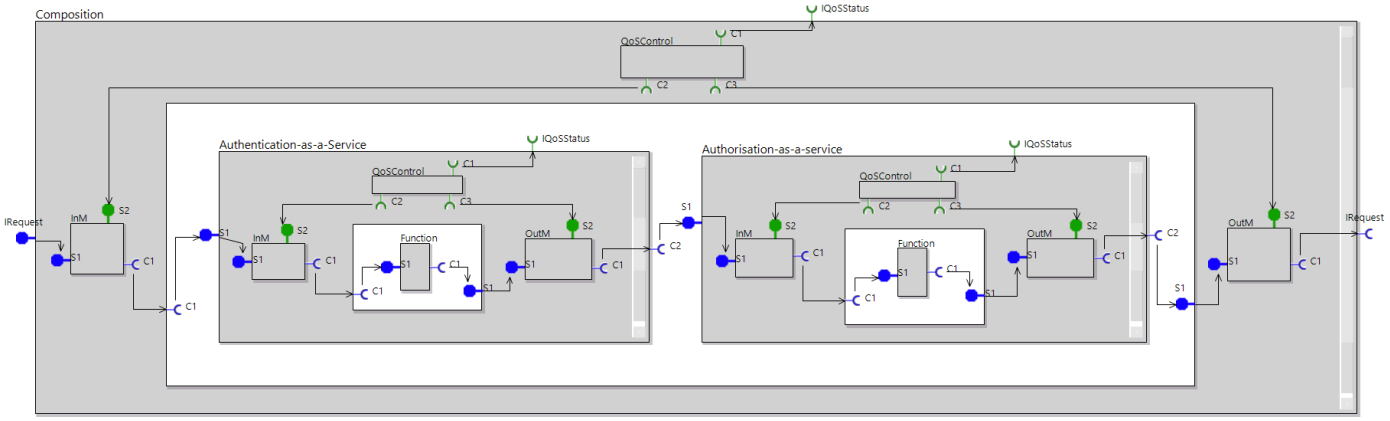


Figure 5. Example of composition (SCC+).

Acknowledgement. The authors would like to thank for their help and contribution:

- Oleksandra Kulankhina: INRIA, Sophia-Antipolis, France
- Cristian Ruz: Pontificia Universidad Católica de Chile, Santiago de Chili, Chile

This work is supported by the OpenCloudware project [6]. OpenCloudware is funded by the French FSN (Fond national pour la Société Numérique) and is supported by Pôles Minalogic, Systematic, and SCS.

References

[1] Kumar, V., Cai, Z., Cooper, B.F., Eisenhauer, G., Schwan, K., Mansour, M., Seshasayee, B. *et al.* (2006) Implementing diverse messaging models with self-managing properties using iflow. In *Autonomic Computing, 2006. ICAC'06. IEEE International Conference on* (IEEE): 243–252.

[2] Brunette, G. and Mogull, R. (2009) Security Guidance for critical areas of focus in Cloud Computing V2.1. *CSA (Cloud Security Alliance), USA. Online: http://www.cloudsecurityalliance.org/guidance/csaguide.v2* **1**.

[3] Spring, J. (2011) Monitoring cloud computing by layer, part 1. *Security Privacy, IEEE* **9**(2): 66–68. doi:10.1109/MSP.2011.33.

[4] Spring, J. (2011) Monitoring cloud computing by layer, part 2. *Security Privacy, IEEE* **9**(3): 52–55. doi:10.1109/MSP.2011.57.

[5] Aubonnet, T., Henrio, L., Kessal, S., Kulankhina, O., Lemoine, F., Madelaine, E., Ruz, C. *et al.* (2015)

Management of service composition based on self-controlled components. *Journal of Internet Services and Applications* **6**(15): 17. doi:10.1186/s13174-015-0031-7, URL https://hal.inria.fr/hal-01180627.

[6] The opencloudware project. URL http://www.opencloudware.org/.

[7] Wang, C., Schwan, K., Talwar, V., Eisenhauer, G., Hu, L. and Wolf, M. (2011) A flexible architecture integrating monitoring and analytics for managing large-scale data centers. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11 (New York, NY, USA: ACM): 141–150. doi:10.1145/1998582.1998605, URL http://doi.acm.org/10.1145/1998582.1998605.

[8] Mian, R., Martin, P. and Vazquez-Poletti, J.L. (2013) Provisioning data analytic workloads in a cloud. *Future Gener. Comput. Syst.* **29**(6): 1452–1458. doi:10.1016/j.future.2012.01.008, URL http://dx.doi.org/10.1016/j.future.2012.01.008.

[9] Clayman, S., Galis, A. and Mamatas, L. (2010) Monitoring virtual networks with lattice. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*: 239–246. doi:10.1109/NOMSW.2010.5486569.

[10] Katsaros, G., Kübert, R. and Gallizo, G. (2011) Building a service-oriented monitoring framework with REST and nagios. In *IEEE International Conference on Services Computing, SCC 2011, Washington, DC, USA, 4-9 July, 2011*: 426–431. doi:10.1109/SCC.2011.53, URL http://dx.doi.org/10.1109/SCC.2011.53.

[11] Hasselmeyer, P. and d'Heureuse, N. (2010) Towards holistic multi-tenant monitoring for virtual data centers. In *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*: 350–356. doi:10.1109/NOMSW.2010.5486528.

[12] Shirey, R. (2007), Internet Security Glossary, Version 2, RFC 4949 (Informational). URL http://www.ietf.org/rfc/rfc4949.txt.

[13] Azurewatch. URL http://www.paraleap.com/azurewatch.

[14] Boundary. URL http://www.bmc.com/truesightpulse/.

[15] Cloudclimate. URL http://www.cloudclimate.com.

[16] Cloudfloor. URL http://cloudfloor.com/.

[17] Cloudcruiser. URL http://cloudcruiser.com/.

[18] Cloudharmony. URL http://cloudharmony.com/.

[19] Cloudsleuth. URL http://www.dynatrace.com.

[20] Cloudstack. URL https://cloudstack.apache.org/.

[21] Cloudwatch. URL https://aws.amazon.com.

[22] Cloudyn. URL http://www.cloudyn.com/.

[23] Consul. URL https://www.consul.io/.

[24] Corradi, A., Foschini, L., Povedano-Molina, J. and López-Soler, J.M. (2012) Dds-enabled cloud management support for fast task offloading. In *2012 IEEE Symposium on Computers and Communications, ISCC 2012, Cappadocia, Turkey, July 1-4, 2012*: 67–74. doi:10.1109/ISCC.2012.6249270, URL http://dx.doi.org/10.1109/ISCC.2012.6249270.

[25] Newrelic. URL http://newrelic.com.

[26] Sensu. URL https://sensuapp.org/.

[27] Uptimesoftware. URL http://www.uptimesoftware.com.

[28] Vrealize hyperic. URL http://www.vmware.com.

[29] Zissis, D. and Lekkas, D. (2012) Addressing cloud computing security issues. *Future Gener. Comput. Syst.* **28**(3): 583–592. doi:10.1016/j.future.2010.12.006, URL http://dx.doi.org/10.1016/j.future.2010.12.006.

[30] Park, J., Yu, H., Chung, K. and Lee, E. (2011) Markov chain based monitoring service for fault tolerance in mobile cloud computing. In *Advanced Information Networking and Applications (WAINA), 2011 IEEE Workshops of International Conference on*: 520–525. doi:10.1109/WAINA.2011.10.

[31] Clayman, S., Clegg, R., Mamatas, L., Pavlou, G. and Galis, A. (2011) Monitoring, aggregation and filtering for efficient management of virtual networks. In *Proceedings of the 7th International Conference on Network and Services Management*, CNSM '11 (Laxenburg, Austria, Austria: International Federation for Information Processing): 234–240. URL http://dl.acm.org/citation.cfm?id=2147671.2147708.

[32] Kutare, M., Eisenhauer, G., Wang, C., Schwan, K., Talwar, V. and Wolf, M. (2010) Monalytics: Online monitoring and analytics for managing large scale data centers. In *Proceedings of the 7th International Conference on Autonomic Computing*, ICAC '10 (New York, NY, USA: ACM): 141–150. doi:10.1145/1809049.1809073, URL http://doi.acm.org/10.1145/1809049.1809073.

[33] Iqbal, W., Dailey, M.N., Carrera, D. and Janecek, P. (2011) Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Gener. Comput. Syst.* **27**(6): 871–879. doi:10.1016/j.future.2010.10.016, URL http://dx.doi.org/10.1016/j.future.2010.10.016.

[34] Ayad, A. and Dippel, U. (2010) Agent-based monitoring of virtual machines. In *Information Technology (ITSim), 2010 International Symposium in*, **1**: 1–6. doi:10.1109/ITSIM.2010.5561375.

[35] Emeakaroha, V.C., Netto, M.A.S., Calheiros, R.N., Brandic, I., Buyya, R. and De Rose, C.A.F. (2012) Towards autonomic detection of sla violations in cloud infrastructures. *Future Gener. Comput. Syst.* **28**(7): 1017–1029. doi:10.1016/j.future.2011.08.018, URL http://dx.doi.org/10.1016/j.future.2011.08.018.

[36] Massie, M.L., Chun, B.N. and Culler, D.E. (2004) The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing* **30**(7): 817 – 840. doi:http://dx.doi.org/10.1016/j.parco.2004.04.001, URL http://www.sciencedirect.com/science/article/pii/S0167819104000535.

[37] Nagios. URL http://WWW.nagios.org.

[38] de Carvalho, M. and Granville, L. (2011) Incorporating virtualization awareness in service monitoring systems. In *Integrated Network Management (IM), 2011 IFIP/IEEE International Symposium on*: 297–304. doi:10.1109/INM.2011.5990704.

[39] Xiang, G., Jin, H., Zou, D., Zhang, X., Wen, S. and Zhao, F. (2010) Vmdriver: A driver-based monitoring mechanism for virtualization. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*: 72–81.

doi:10.1109/SRDS.2010.38.

[40] Hoang, D.T., Lee, C., Niyato, D. and Wang, P. (2013) A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless Communications and Mobile Computing* **13**(18): 1587–1611. URL http://dblp.uni-trier.de/db/journals/wicomm/wicomm13.html.

[41] Padhy, S., Kreutz, D., Casimiro, A. and Pasin, M. (2011) Trustworthy and resilient monitoring system for cloud infrastructures. In *Proceedings of the Workshop on Posters and Demos Track*, PDT '11 (New York, NY, USA: ACM): 3:1–3:2. doi:10.1145/2088960.2088963, URL http://doi.acm.org/10.1145/2088960.2088963.

[42] Azmandian, F., Moffie, M., Dy, J.G., Aslam, J.A. and Kaeli, D.R. (2011) Workload characterization at the virtualization layer. In *MASCOTS* (IEEE Computer Society): 63–72. URL http://dblp.uni-trier.de/db/conf/mascots/mascots2011.html.

[43] Aubonnet, T. and Simoni, N. (2014) Self-control cloud services. In *2014 IEEE 13th International Symposium on Network Computing and Applications, NCA 2014, Cambridge, MA, USA, 21-23 August, 2014*: 282–286. doi:10.1109/NCA.2014.48.

[44] Tatiana Aubonnet and Noëmie Simoni (2013) Service creation and self-management mechanisms for mobile cloud computing. In *Wired/Wireless Internet Communication - 11th International Conference, WWIC 2013, St. Petersburg, Russia. Proceedings*: 43–55. doi:10.1007/978-3-642-38401-1_4.

[45] (1997) *Service plane for Intelligent Network, Capability Set2*. Tech. rep., ITU-T Recommendation Q.1222. URL https://www.itu.int/rec/T-REC-Q.1222.

[46] Baude, F., Henrio, L. and Ruz, C. (2014) Programming distributed and adaptable autonomous componentsâĂŤthe gcm/proactive framework. *Software: Practice and Experience* : n/adoi:10.1002/spe.2270, URL http://dx.doi.org/10.1002/spe.2270.

[47] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L. and PÃŁrez, C. (2008) GCM: A Grid Extension to Fractal for Autonomous Distributed Components. *annals of telecommunications - annales des tÃŕlÃŕcommunications* URL https://hal.inria.fr/inria-00323919.

[48] Baude, F., Henrio, L. and Naoumenko, P. (2009) Structural reconfiguration: An autonomic strategy for gcm components. In *Proceedings of the 2009 Fifth International Conference on Autonomic and Autonomous Systems* (Washington, DC, USA: IEEE Computer Society): 123–128. doi:10.1109/ICAS.2009.28.

[49] Henrio, L., Kulankhina, O., Liu, D. and Madelaine, E. (2014) Verifying the correct composition of distributed components: Formalisation and Tool. In *FOCLASA* (Rome, Italy). URL https://hal.inria.fr/hal-01055370.

[50] Horn, P. (2001) Autonomic Computing: IBM's Perspective on the State of Information Technology .

[51] Computing, A. and others (2006) An architectural blueprint for autonomic computing. *IBM White Paper* .

[52] Kephart, J.O. and Chess, D.M. (2003) The vision of autonomic computing. *Computer* **36**(1): 41–50. doi:10.1109/MC.2003.1160055.

[53] Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B. and Steenkiste, P. (2004) Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer* **37**(10): 46–54. doi:10.1109/MC.2004.175.

[54] Garlan, D., Schmerl, B. and Cheng, S.W. (2009) *Software Architecture-Based Self-Adaptation* (Boston, MA: Springer US), 31–55. doi:10.1007/978-0-387-89828-5_2, URL http://dx.doi.org/10.1007/978-0-387-89828-5_2.

[55] Gauvrit, G., Daubert, E. and Andre, F. (2010) SAFDIS: A Framework to Bring Self-Adaptability to Service-Based Distributed Applications (IEEE): 211–218. doi:10.1109/SEAA.2010.25, URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5598099.

[56] Noëmie Simoni and Xiaofei Xiong and Chunyang Yin (2009) Virtual community for the dynamic management of NGN mobility. In *Fifth International Conference on Autonomic and Autonomous Systems, ICAS 2009, Valencia, Spain, 20-25 April 2009*: 82–87. doi:10.1109/ICAS.2009.33, URL http://doi.ieeecomputersociety.org/10.1109/ICAS.2009.33.

[57] Houda Alaoui Soulimani and Philippe Coude and Noëmie Simoni (2011) User-centric and qos-based service session. In *2011 IEEE Asia-Pacific Services Computing Conference, APSCC 2011, Jeju, Korea (South), December 12-15, 2011*: 267–274. doi:10.1109/APSCC.2011.64, URL http://dx.doi.org/10.1109/APSCC.2011.64.

[58] Nassar, R. and Simoni, N. (2013) Semantic handover among distributed coverage zones for an ambient continuous service session. *IJHCR* **4**(1): 37–58. doi:10.4018/jhcr.2013010103, URL http://dx.doi.org/10.4018/jhcr.2013010103.

[59] Cansado, A. and Madelaine, E. (2008) Specification and Verification for Grid Component-Based Applications: From Models to Tools. In Boer, F.S.d., Bonsangue, M.M. and Madelaine, E. [eds.] *Formal Methods for Components and Objects*, no. 5751 in Lecture Notes in Computer Science (Springer Berlin Heidelberg), 180–203. URL http://link.springer.com/chapter/10.1007/978-3-642-04167-9_10. DOI: 10.1007/978-3-642-04167-9_10.

[60] Rfc 2617 basic and digest access authentication. URL http://www.rfc-base.org/rfc-2617.html.