A Formal and Tooled Framework for Managing Everything as a Service

www.occiware.org

Deliverable D2.2.3

QoS Model for OCCI

OCIIware project is supported by Fonds national pour la Société Numérique (FSN) and the following competitive clusters Systematic, Minalogic, PICOM, Images & Réseaux and Solutions Communicantes Sécurisées.
Abstract
The deliverable describes QoS techniques that are designed to be used in the OCCIware project and proposes an OCCI extension that manages these techniques. The work described in this deliverable is a research work, mostly conducted by PhD students at UGA. The proposed techniques rely on the so-called concept of "performance debugging", a research field that aims at profiling and understanding the behaviour of distributed systems in order to detect performance bottlenecks.

Tags
OCCIware, Open Cloud Computing Interface (OCCI), Quality of Service (QoS), Service Level Agreement (SLA)
Document versioning

<table>
<thead>
<tr>
<th>Version</th>
<th>Changes</th>
<th>Author(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>Initial version</td>
<td>Amadou Diarra</td>
</tr>
<tr>
<td>0.2</td>
<td>Revised version for better description of the integration with the OCCI specifications</td>
<td>Amadou Diarra</td>
</tr>
<tr>
<td>1.0</td>
<td>Review corrections</td>
<td>Philippe Merle</td>
</tr>
</tbody>
</table>

Document reviews

<table>
<thead>
<tr>
<th>Review</th>
<th>Date</th>
<th>Ver.</th>
<th>Reviewer(s)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plan</td>
<td>...</td>
<td>0.2</td>
<td>Un réviseur</td>
<td>Comment...</td>
</tr>
<tr>
<td>Brouillon</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Validation</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>BAT</td>
<td>December 12, 2016</td>
<td>1.0</td>
<td>Philippe Merle</td>
<td></td>
</tr>
</tbody>
</table>
Deliverable | D2.2.3 | Status | Final
Expected | November 01, 2016 | Dissemination | public
Delivered | December 12, 2016 | Version | 1.0
Contents

1 Introduction ........................................ 7

2 Integration of QoS techniques in OCCIware ............ 9

3 A performance model to manage QoS in the OCCIware platform .................................................. 11
   3.1 Overview ........................................ 11
   3.2 Constructing component profiles .................... 12
   3.3 Computing placement strategies ................... 12
   3.4 Modeling performance ................................ 13
   3.5 Predicting the throughput of a placement strategy .......................................................... 14
   3.6 Illustration on a concrete use case .................. 14

4 Conclusion ............................................. 17
List of Figures

2.1 OCCI-SLA Extension (according to OGF) ........................................... 9
2.2 Overview of QoS for OCCI ................................................................. 10
3.1 Performance Modeling ................................................................. 11
3.2 Pivot Tracing Overview ................................................................. 12
List of Tables

1.1 A survey on QoS approaches in cloud computing ........................................... 8

3.1 Strategy 1: Resources consumption modeling, $x$ is an input workload(req/s) .... 14
3.2 Strategy 2: Resources consumption modeling, $x$ is an input workload(req/s) .... 15
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Status</th>
<th>Dissemination</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2.2.3</td>
<td>Final</td>
<td>public</td>
<td>1.0</td>
</tr>
<tr>
<td>Expected</td>
<td>November 01, 2016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivered</td>
<td>December 12, 2016</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The deliverable describes QoS techniques that are designed to be used in the OCCIware project and proposes an OCCI extension that manages these techniques. The work described in this deliverable is a research work, mostly conducted by PhD students at UGA. In this first chapter, we briefly introduce the notion of "Quality of Service" and describe a set of approaches that have been proposed in the literature.

Cloud computing is becoming essential in many fields of today technologies. It provides a set of resources (that can be either software or hardware) that are remotely accessible as a service. Cloud systems involve two kinds of actors: consumers and providers. The relationship between these actors is based on the notion of "Quality of Service" (QoS). QoS allows defining the quality that is expected from a provided service. It has thus the goal of guaranteeing some level of performance in a cloud computing service. In order to precisely define the expected QoS level, the two actors can rely on contracts called "Service Level Agreement (SLA)".

In the last ten years, several approaches have been proposed to handle QoS in cloud systems. Many of these approaches cover the same three fundamental aspects of QoS that are "availability", "reliability" and "performance". Nevertheless, each approach handles QoS in a different way. Among the most recent approaches, let us cite the work described in [1] that provides automated QoS management methods, the work described in [2] that focuses on systems of systems, or the work described in [3] that focuses on the case when multiple independent Cloud providers seamlessly cooperate to provide QoS guarantees. Most existing approaches are described in an extensive survey [4] that is summarized in Table [1.1].

Our starting point for handling with QoS in the OCCIware project is the following: none of the existing systems rely on a principled approach to understanding performance bottlenecks in distributed systems. Indeed, existing works focus on rather simple performance metrics that are considered individually for each component of the system. Rather, we propose to focus on the design of techniques that allow understanding the performance of systems as a whole. To be more precise, the aim of performance debugging techniques is to precisely identify which component (software or hardware) of a system is a bottleneck, based on the observation (i.e., profiling) of the system as a whole and based on performance models that allow defining and reasoning on the causality relationships between the various components. The reason to focus on performance debugging techniques is the following: in many practical systems, the bottleneck component is not a component that is used at 100% of its capability, but an idle component (this has been illustrated by research work conducted at UGA this year [5]). Catching over-used components is trivial and can easily be achieved without any sophisticated performance model. Understanding that performance are limited by the idleness of some components and pinpointing the root causes of idleness is a complex research problem that is currently being tackled by major companies in the world, including Facebook [6], Microsoft [7], Google [8].
The remainder of this document is organised as follows: in Chapter 2, we explain how QoS techniques will be integrated within the OCCIware framework. In Chapter 3, we describe our new performance model for OCCIWare, and in Chapter 4, we conclude this deliverable.
Chapter 2

Integration of QoS techniques in OCCIware

As explained in the introduction, the QoS techniques that UGA proposes for the OCCIware project are based on "Performance debugging" techniques. These techniques are currently being designed by PhD students at UGA. They will aim at profiling and modeling a system in order to understand its performance and identify the performance bottlenecks. The goal of this chapter is to explain how these techniques will be integrated within the OCCIware framework.

The OGF has defined an OCCI-SLA extension that allows describing a set of performance criteria. Within the context of OCCIware, this extension has been integrated using the cloud-designer (studio) that is one of contributions of the project (sub-project 3). As depicted on Figure 2.1 this extension defines two kinds: on one side, Agreement resources manage SLA established between consumer and provider, and on the other side, AgreementLink links associate agreement instances with other resource instances. As we can see on Figure 2.1, the Agreement kind inherits from Resource and AgreementLink kind inherits from Link.

![Figure 2.1: OCCI-SLA Extension (according to OGF)](image)

To integrate the work on QoS via performance debugging done by UGA, we will add a mixin to the Agreement kind described above (the overall picture is depicted in Figure 2.2). This mixin will be in charge of implementing the QoS techniques that are described in the next chapter. The outcome of this mixin will be twofolds: performance modeling and bottleneck identification.
Figure 2.2: Overview of QoS for OCCI
Chapter 3

A performance model to manage QoS in the OCCIware platform

This chapter presents a first analytical performance model that we will integrate inside the QoS component of the OCCIware platform. This first model allows replying to the following question. Let us assume that we have an application $A$ composed of $n$ components $C_1, C_2, ..., C_n$ running on a distributed infrastructure $D$ with $m$ machines $M_1, M_2, ..., M_n$ (machines are potentially heterogeneous). What is the most efficient deployment for $A$ on $D$? In the third year of the project, we will extend this model to cover other questions related to performance.

3.1 Overview

The overall picture of the QoS system we propose for OCCIware is depicted in Figure 3.1.

To build our model we first need to identify a set of relevant metrics (CPU, memory, disk, etc.). The idea is first to compute component profiles and inter-component relationships using the set of identified metrics. These profiles are being constructed using the current placement strategy. These profiles essentially contain information about request flows between components (and associated CPU consumption) and are thus independent of the placement strategy used to construct them. Once the component profiles are done, a performance model allows predicting...
the performance of any possible placement strategy. In the current form of our QoS component, causality relationships are not taken into account and we need to gather metrics for the different placement strategies under study. We are currently working on integrating causality relationships information to allow reasoning on strategies for which no profiling has been done.

3.2 Constructing component profiles

There exists several tools to profile components of a distributed application. In the context of the work done by UGA, we use the two following tools to construct component profiles and inter-component relationships:

1. **SAR (System Activity Report)**: this tool allows collecting information related to CPU, memory, I/O usage in Unix-like operating system.

2. **Pivot Tracing**: this very recent tool [9] allows gathering causality relationships between the requests of a distributed systems (and thus the components involved in processing these requests). As illustrated on Figure 3.2, Pivot Tracing (PT) works by dynamically instrumenting the system (using so-called agents). The instrumented code uses a notion of tracepoints and baggages to dynamically construct the causality relationships between requests. Pivot Tracing can be queried using LINQ, a relational query language, similar to SQL.

![Figure 3.2: Pivot Tracing Overview](image)

3.3 Computing placement strategies

An application with several components running on a distributed system can be deployed using different strategies. We consider the example of the application \( A = \{C_1, C_2, \ldots, C_n\} \) a set of \( n \) components described above. Let \( d_n \) the number of deployment strategies, \( d_n \) is the number of partition of set \( A \). This number is given by the Bell number\(^2\) in combinatorial mathematics:

\[
d_0 = d_1 = 1 \quad \text{for} \quad n \leq 1 \quad \text{and} \quad d_{n+1} = \sum_{k=0}^{n} C_n^k d_k, \quad \text{where} \quad C_n^k \quad \text{is the number of combination of} \quad k \quad \text{elements among} \quad n.
\]

For example, if \( A \) is a 3-tiers application composed of a web server \( WS \), an application server \( AS \) and database server \( DS \), \( d_3 = 5 \). Indeed, we have the following strategies:

\(^1\)https://en.wikipedia.org/wiki/Partition_of_a_set
\(^2\)https://en.wikipedia.org/wiki/Bell_number
• \{\{WS\},\{AS\},\{DS\}\} (one component per machine)
• \{\{WS,AS\},\{DS\}\} (WS and AS are colocated on the same machine)
• \{\{WS,DS\},\{AS\}\} (WS and DS are colocated on the same machine)
• \{\{WS\},\{AS,DS\}\} (AS and DS are colocated on the same machine)
• \{\{WS,AS,DS\}\} (all components run on the same machine)

3.4 Modeling performance

Using values of metrics gathered from component profiles, we create a set of linear equations to predict a system performance. Let consider a distributed system running an application with an input workload \(\lambda_i\) during a time interval \(t\), we denote by \(u_i\) the corresponding resource (CPU, network, disk I/O,...) usage. For each component we can estimate \(u_i\) as a linear combination of input workload:

\[
\forall i \in \{1, ..., n\}, u_i = \beta_1 + \beta_2 \lambda_i + \epsilon_i
\]  

(3.1)

where \(n\) is the total number of time intervals, \(\beta_1\) and \(\beta_2\) are unknown parameters of the model, \(\epsilon_i\) is the estimation error. So, using vectorial notation, Equation (3.1) can be written:

\[
U = \beta_1 I + \beta_2 \Lambda + \epsilon
\]  

(3.2)

where:

• \(U = [u_1, ..., u_n]^T\) a random vector of dimension \(n\)
• \(I = [1, ..., 1]^T\) a vector of dimension \(n\) where each component is 1
• \(\Lambda = [\lambda_1, ..., \lambda_n]^T\) a given vector of dimension \(n\)
• \(\epsilon = [\epsilon_1, ..., \epsilon_n]^T\) a random vector of dimension \(n\)

To complete our model construction, we use ordinary least squares regression to estimate the parameters \(\beta_1\) and \(\beta_2\). This technique allows to obtain values that minimize the quantity \(\sum_{i=1}^{n} (u_i - \beta_1 - \beta_2 \lambda_i)^2\). Then if \(\hat{\beta}_1\) and \(\hat{\beta}_2\) denote these values, the predicted value of the total usage of a resource \(r\) is given by:

\[
U_r = \hat{\beta}_1 + \hat{\beta}_2 \lambda
\]  

(3.3)

where:

• \(\hat{\beta}_1 = \bar{u} - \hat{\beta}_2 \bar{\lambda}\), with \(\bar{u} = \frac{1}{n} \sum_{i=1}^{n} u_i\) and \(\bar{\lambda} = \frac{1}{n} \sum_{i=1}^{n} \lambda_i\)
• \(\hat{\beta}_2 = \frac{\sum_{i=1}^{n} (\lambda_i - \bar{\lambda}) u_i}{\sum_{i=1}^{n} (\lambda_i - \bar{\lambda})^2}\)
• \(\lambda\) is the input workload.
3.5 Predicting the throughput of a placement strategy

The throughput prediction is based on a method described in [10]. Indeed, let consider the following resources: CPU, network, disk. Using Equation 3.3, we obtain the following set of equations for each component of an application running on a distributed system with \( m \) machines:

\[
\begin{align*}
U_c &= \hat{\beta}_{1c} + \hat{\beta}_{2c} \lambda_c \\
U_b &= \hat{\beta}_{1n} + \hat{\beta}_{2n} \lambda_n \\
U_d &= \hat{\beta}_{1d} + \hat{\beta}_{2d} \lambda_d 
\end{align*}
\]  

(3.4)

where \( U_c, U_n \) and \( U_d \) denote respectively the usage of CPU, network and disk; \( \hat{\beta}_{1c}, \hat{\beta}_{2c}, \hat{\beta}_{1n}, \hat{\beta}_{2n}, \hat{\beta}_{1d} \) and \( \hat{\beta}_{2d} \) are estimated parameters. Given a component placement strategy, we can compute the maximum input request rate that saturates each of these resources on each machine using Equation 3.4. Let take the example of a component \( c \) running on a machine \( M \) of a distributed system, the input request rate \( \lambda_{cpu}(M) \) that saturates CPU resource is given by \( \hat{\beta}_{1c} + \hat{\beta}_{2c} \lambda_{cpu}(M) \leq 100 \). If \( \lambda_{disk}(M) \) and \( \lambda_{network}(M) \) denote respectively such saturation rate of disk and network resources at machine \( M \), the system throughput \( T \) can be estimated as the lowest saturation for all resource types at all machines. It is given by the formula:

\[
T = \min_{\forall M \in D} \{ \lambda_{cpu}(M), \lambda_{disk}(M), \lambda_{network}(M) \}
\]  

(3.5)

where \( D \) is a distributed system.

3.6 Illustration on a concrete use case

To validate our model we used one use case: the Rubis application\(^3\). We will validate the model in the third year on a second use-case: the Roboconf-Linshare system. Rubis is a 3-tiers application that emulates an auction website. It is composed of a web server (Apache), an application server (JBoss) hosting EJB (Entreprise Java Bean) entities and a database server (MySQL).

We deployed Rubis on a cluster of three machines A, B, C connected to each other using a Gibabyte ethernet network. We used SAR to profile components and identified 5 deployment strategies, for a question of relevance, we present the results of 2 strategies:

- Strategy 1: one component by machine (Apache on A, JBoss on B and MySQL on C)
- Strategy 2: Apache on A and (JBoss+MySQL) on B

According to Strategy 1, we obtain in Table 3.1 linear modeling of resources (CPU, network) consumption in function of input workload in each machine. The results of Strategy 2 are in Table 3.2.

<table>
<thead>
<tr>
<th>Machine</th>
<th>CPU</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3, 3.10^{-3}x + 23, 6</td>
<td>4, 8.10^{-3}x + 32, 88</td>
</tr>
<tr>
<td>B</td>
<td>1, 7.10^{-3}x + 10, 54</td>
<td>3.10^{-3}x + 2, 12</td>
</tr>
<tr>
<td>C</td>
<td>9, 9.10^{-3}x + 51, 59</td>
<td>3.10^{-5}x + 6.10^{-2}</td>
</tr>
</tbody>
</table>

Table 3.1: Strategy 1: Resources consumption modeling, \( x \) is an input workload(req/s)

\(^3\)http://rubis.ow2.org/
Table 3.2: Strategy 2: Resources consumption modeling, $x$ is an input workload(req/s)

Using the performance model, we predict the system throughput in each strategy, to determine the best strategy. So in first strategy we obtained:

- on machine A: let $x_{cpu}(A)$ and $x_{net}(A)$ the workloads that stature the CPU and network resources, these values are given respectively by $3, 3.10^{-3}x_{cpu}(A) + 23, 6 \leq 100$ and $4, 8.10^{-3}x_{net}(A) + 32, 88$, so $x_{cpu}(A) = 2, 3.10^4$req/s and $x_{net}(A) = 1, 4.10^4$req/s
- on machine B: $x_{cpu}(B) = 5, 2.10^4$req/s and $x_{net}(B) = 3, 26.10^5$req/s
- on machine C: $x_{cpu}(C) = 4, 8.10^3$req/s and $x_{net}(C) = 3, 3.10^6$req/s

The system throughput in Strategy 1 is

$$Thr_1 = \min\{x_{cpu}(A), x_{net}(A), x_{cpu}(B), x_{net}(B), x_{cpu}(C), x_{net}(C)\} = 4, 8.10^3$$(3.6)

As in first strategy, in second strategy we have:

- on machine A: $x_{cpu}(A) = 4, 4.10^6$req/s and $x_{net}(A) = 4, 3.10^5$req/s
- on machine B: $x_{cpu}(B) = 2, 1.10^4$req/s and $x_{net}(B) = 1, 6.10^7$req/s

In this strategy the system throughput is

$$Thr_2 = \min\{x_{cpu}(A), x_{net}(A), x_{cpu}(B), x_{net}(B)\} = 2, 1.10^4$$ (3.7)

To conclude, in this use-case the performance model indicates that the best deployment is obtained using Strategy 2, which is confirmed by practical experiments.
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>D2.2.3</th>
<th>Status</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expected</td>
<td>November 01, 2016</td>
<td>Dissemination</td>
<td>public</td>
</tr>
<tr>
<td>Delivered</td>
<td>December 12, 2016</td>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Deliverable</td>
<td>D2.2.3</td>
<td>Status</td>
<td>Final</td>
</tr>
<tr>
<td>------------</td>
<td>--------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>Expected</td>
<td>November 01, 2016</td>
<td>Dissemination</td>
<td>public</td>
</tr>
<tr>
<td>Delivered</td>
<td>December 12, 2016</td>
<td>Version</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Chapter 4

Conclusion

This deliverable has presented the first QoS model proposed by UGA and its integration within the OCCLware platform. We have tested this model on a first component-based application (Rubis) to assess its effectiveness. This work is nevertheless still an undergoing research work and we plan to pursue our efforts in the following two directions: (1) integrate parts of this work in the OCCLware platform, (2) extend the performance model with causality relationships to allow reasoning on placement strategies without having to profile them.
<table>
<thead>
<tr>
<th>Deliverable</th>
<th>Status</th>
<th>Dissemination</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>D2.2.3</td>
<td>Final</td>
<td>public</td>
<td>1.0</td>
</tr>
<tr>
<td>Expected</td>
<td>November 01, 2016</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivered</td>
<td>December 12, 2016</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


