Extending Cloud Management Tools at the IaaS and PaaS Layers for Cloud Interoperability

Student: Oriol Collell Martin
Director: Ana Maria Juan Ferrer
Company: AtoS
Supervisor: Javier Franch Gutiérrez
Department: ESSI
College: Facultat d’Informàtica de Barcelona (FIB)
University: Universitat Politècnica de Catalunya (UPC) BarcelonaTech
Degree: Enginyeria Informàtica
Mode: B
Number of Credits: 37.5

February 20, 2013
Contents

Acknowledgements xii
Preface xiii

I Foundations and Project Plan 1

1 Background 2
  1.1 Cloud Computing ..................................................... 2
  1.1.1 Infrastructure as a Service .................................. 4
  1.1.2 Platform as a Service ......................................... 6
  1.2 Cloud Computing Interoperability ................................. 8

2 Goals 10

3 Project Plan 11
  3.1 Initial plan ............................................................ 11
  3.1.1 Phases ............................................................. 11
  3.1.2 Working hours estimation ....................................... 13
  3.2 Deviation from the initial plan ................................... 13

4 Cost analysis 15
  4.1 Human Resources ................................................... 15
  4.2 Material Resources .................................................. 15

II Exploring interoperability at the PaaS layer 17

5 Introduction 18

6 Cloud4SOA 20
  6.1 Matchmaking ......................................................... 20
  6.2 Deployment .......................................................... 21
  6.3 Migration .............................................................. 21
  6.4 Monitoring ............................................................ 21
11.2.4 Domain Model ........................................ 77
11.2.5 Client library conceptual model .................... 78
11.2.6 Behavioral Model .................................. 79

11.3 Implementation ........................................ 99
11.3.1 Technologies involved ............................... 100
11.3.2 Local Adapter ........................................ 101
11.3.3 Remote Adapter ...................................... 102
11.3.4 Integration with Cloud4SOA ......................... 103

11.4 Testing ................................................ 103
11.4.1 JUnit Overview ..................................... 103
11.4.2 Local Part Tests ..................................... 105
11.4.3 Remote Part Tests ................................... 107
11.4.4 Discussion on NFRs .................................. 108

12 PaaS Hybrid Cloud Scenarios ............................ 109
12.1 Hybrid Cloud ........................................... 109
12.2 Scenarios Description .................................. 110
12.2.1 Hybrid Deployment Scenario ....................... 110
12.2.2 Bursting on Demand Scenario ...................... 110
12.3 Context-Aware Application Description ............... 112
12.3.1 Application deployment map ....................... 113
12.4 Cloud4SOA Evaluation ................................ 114
12.4.1 Deployment Process ................................ 114
12.4.2 Governance Process ................................. 121
12.4.3 Monitoring Process ................................. 124
12.4.4 Migration Process ................................... 125
12.5 Summary ............................................... 127

13 Conclusions .............................................. 129

III Exploring interoperability at the IaaS layer ............... 131

14 Introduction .............................................. 132

15 OPTIMIS ................................................ 135

16 Interoperability at the IaaS layer ......................... 137
16.1 Interoperability problems at the IaaS level ............ 137
16.2 Standards for IaaS interoperability .................... 138
16.2.1 Open Cloud Computing Interface (Open Grid Forum - OGF) ............. 138

17 Requirements Overview ................................... 140
17.1 Methodology ............................................ 140
17.2 Functional Requirements ............................... 141
17.2.1 Service Management ............................... 143
17.2.2 Data Management ................................. 144
17.2.3 Monitoring ........................................ 145
17.2.4 Non-Functional Requirements ................. 146

18 Service Design ............................................. 148
  18.1 Data Model ............................................. 148
  18.1.1 Definitions ........................................ 150
  18.1.2 Classification System ......................... 150
  18.1.3 Entities Representation System .............. 152
  18.2 Behavioral Model .................................... 157
  18.2.1 Authentication ................................... 160
  18.2.2 Requests and Responses syntax ................ 160
  18.2.3 Create Virtual Machine ....................... 160
  18.2.4 Delete Virtual Machine ....................... 161
  18.2.5 Update Virtual Machine ....................... 161
  18.2.6 Execute Action .................................. 162
  18.2.7 Get Virtual Machine ............................ 162
  18.2.8 Get Service Virtual Machines ................. 163
  18.2.9 Get all Virtual Machines ...................... 164
  18.2.10 Terminate Service .............................. 164

19 Proxy Design .............................................. 166
  19.1 OPTIMIS reference architecture overview ........ 166
  19.1.1 Service Deployment ............................. 168
  19.1.2 Service Operation .............................. 169
  19.2 External Providers Integration ................... 170
  19.2.1 CloudQoS Integration ......................... 170
  19.2.2 Data Manager Integration ..................... 171
  19.2.3 Monitoring Integration ....................... 171
  19.3 CloudQoS Proxy design ............................ 172
  19.3.1 Interface Layer Conceptual Model ............. 173
  19.3.2 Operations Layer Conceptual Model .......... 175
  19.3.3 Interface Layer Behavioral Model ............ 176
  19.3.4 Operations Layer Behavioral Model .......... 179
  19.4 Implementation .................................... 181
  19.4.1 Technologies involved ......................... 182
  19.4.2 Example .......................................... 182
  19.5 Testing .............................................. 183
  19.5.1 Proxy Tests ..................................... 183

20 Integration with Amazon EC2 .......................... 185
  20.1 Integration Analysis ................................ 185
  20.1.1 CloudQoS Integration ......................... 185
  20.1.2 Data Manager Integration ..................... 186
  20.1.3 Monitoring Infrastructure Integration ....... 188
  20.2 Design .............................................. 192
  20.2.1 CQoS Proxy ..................................... 192

iv
20.2.2 Amazon S3 Wrapper ................................................. 198
20.2.3 AWS Collector .................................................. 205
20.3 Implementation ..................................................... 213
  20.3.1 Technologies Involved ....................................... 213
  20.3.2 S3 Wrapper implementation ................................. 215
  20.3.3 Collector implementation ................................. 216
20.4 Testing ............................................................. 216
  20.4.1 S3 Wrapper Tests ............................................. 216
  20.4.2 Collector Tests ............................................... 219
20.5 OPTIMIS Integration ............................................... 219

21 Conclusions .............................................................. 222
Conclusions & Lessons Learned ........................................... 224

A JSON OCCI Messages Specification ....................................... 226
  A.1 Resource Instance ............................................. 226
  A.2 Link .......................................................... 227
  A.3 Action ......................................................... 228
  A.4 Attributes ..................................................... 228
  A.5 Example ....................................................... 229

B HTTP Response Codes .................................................. 231

C Produced Code .......................................................... 233

Bibliography ................................................................. 234
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>High-level view of the Cloud stack.</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>High-level view of a virtualization environment.</td>
<td>5</td>
</tr>
<tr>
<td>3.1</td>
<td>Gantt diagram describing the initial project planification</td>
<td>12</td>
</tr>
<tr>
<td>9.1</td>
<td>CloudFoundry architecture</td>
<td>39</td>
</tr>
<tr>
<td>9.2</td>
<td>Interaction between users and a CloudFoundry platform.</td>
<td>39</td>
</tr>
<tr>
<td>9.3</td>
<td>High-level view of the authorization workflow.</td>
<td>40</td>
</tr>
<tr>
<td>9.4</td>
<td>Deployment view of the VMs that will host CloudFoundry.</td>
<td>43</td>
</tr>
<tr>
<td>9.5</td>
<td>High-level BOSH architecture.</td>
<td>46</td>
</tr>
<tr>
<td>9.6</td>
<td>Results from the “vmc info” and “vmc services” commands.</td>
<td>49</td>
</tr>
<tr>
<td>9.7</td>
<td>Results from the “vmc frameworks” and “vmc runtimes” commands</td>
<td>50</td>
</tr>
<tr>
<td>9.8</td>
<td>Result of the “vmc push” command.</td>
<td>50</td>
</tr>
<tr>
<td>9.9</td>
<td>Pushed application shown in a browser.</td>
<td>51</td>
</tr>
<tr>
<td>11.1</td>
<td>Requirements gathering process</td>
<td>66</td>
</tr>
<tr>
<td>11.2</td>
<td>Cloud4SOA 3-layer architecture</td>
<td>73</td>
</tr>
<tr>
<td>11.3</td>
<td>Components that form the governance layer</td>
<td>73</td>
</tr>
<tr>
<td>11.4</td>
<td>PaaS Offering semantic model</td>
<td>74</td>
</tr>
<tr>
<td>11.5</td>
<td>Cloud4SOA communication pattern with the adapters</td>
<td>75</td>
</tr>
<tr>
<td>11.6</td>
<td>Invocation patterns between Cloud4SOA, the adapter and CloudFoundry</td>
<td>76</td>
</tr>
<tr>
<td>11.7</td>
<td>Cloud Foundry semantic profile</td>
<td>104</td>
</tr>
<tr>
<td>11.8</td>
<td>Result of the local part tests execution</td>
<td>107</td>
</tr>
<tr>
<td>11.9</td>
<td>Results of the remote part tests execution</td>
<td>108</td>
</tr>
<tr>
<td>12.1</td>
<td>Hybrid Cloud</td>
<td>110</td>
</tr>
<tr>
<td>12.2</td>
<td>Hybrid Cloud Scenario</td>
<td>111</td>
</tr>
<tr>
<td>12.3</td>
<td>Bursting on Demand Scenario</td>
<td>112</td>
</tr>
<tr>
<td>12.4</td>
<td>PTIN Context-aware Multimedia Framework</td>
<td>113</td>
</tr>
<tr>
<td>12.5</td>
<td>Application Deployment Map</td>
<td>114</td>
</tr>
<tr>
<td>12.6</td>
<td>Context Enabler Application Profile</td>
<td>115</td>
</tr>
<tr>
<td>12.7</td>
<td>Matching PaaS Offers for Context Enabler</td>
<td>116</td>
</tr>
<tr>
<td>12.8</td>
<td>Deploying Context Enabler</td>
<td>116</td>
</tr>
<tr>
<td>12.9</td>
<td>CloudFoundry credentials</td>
<td>117</td>
</tr>
<tr>
<td>12.10</td>
<td>Deploying Context Enabler web archive</td>
<td>117</td>
</tr>
<tr>
<td>12.11</td>
<td>Context Enabler running in CloudFoundry</td>
<td>118</td>
</tr>
<tr>
<td>12.12</td>
<td>Group Manager Enabler Database Credentials</td>
<td>119</td>
</tr>
<tr>
<td>12.13</td>
<td>Group Manager Enabler Database Created</td>
<td>120</td>
</tr>
</tbody>
</table>
# List of Tables

1.1 A classification for PaaS platforms ........................................ 8  
3.1 Working hours estimation .................................................. 13  
8.1 Comparative of the features offered by each of the analyzed PaaS providers ... 35  
9.1 Description of the different installation options ......................... 42  
9.2 List of the components that each VM will host .......................... 44  
10.1 Integration complexity of the methods in the harmonized API ........ 64  
11.1 Cloud4SOA Functional requirements .................................... 68  
11.2 Requirements template .................................................. 68  
11.3 Cloud4SOA Non-Functional requirements ............................... 71  
11.4 Specialized Cloud4SOA non-functional requirements .................. 72  
11.5 Description of the local part unit tests .................................. 106  
11.6 Description of the remote part unit tests ............................... 107  
12.1 Cloud4SOA vs CloudFoundry Provider API deployment time (in seconds) ... 118  
12.2 Cloud4SOA vs CloudBees creating database (in seconds) ............... 119  
12.3 Cloud4SOA vs CloudBees deployment time (in seconds) ................. 120  
12.4 Cloud4SOA vs CloudFoundry Provider API stopping application (in seconds) ... 122  
12.5 Cloud4SOA vs CloudFoundry Provider API undeployment time (in seconds) ... 123  
17.1 OPTIMIS Functional requirements ...................................... 142  
17.2 OPTIMIS Non-Functional requirements ................................. 146  
17.3 Specialized OPTIMIS non-functional requirements ..................... 147  
18.1 Integrity Restrictions of the OCCI Data Model ........................ 150  
18.2 Changes of state performed by Compute actions ........................ 154  
18.3 Changes of state performed by Network actions ....................... 155  
18.4 IPNetworkingMixin attributes ....................................... 155  
18.5 Changes of state performed by Storage actions ....................... 156  
18.6 Monitoring mixin attributes ....................................... 157  
18.7 Methods exposed by the service ..................................... 159  
19.1 Methods exposed by the CQoS interface ................................ 170  
19.2 Methods exposed by the Data Manager interface ....................... 171  
19.3 Description of the OCCI CQoS proxy unit tests ....................... 184
20.1 S3 Wrapper requirements . . . . . . . . . . . . . . . . . . . . . . . . . . 187
20.2 Metrics measured by OPTIMIS and their support in AWS . . . . . . . . . . . 190
20.3 Attributes of an OPTIMIS measurement . . . . . . . . . . . . . . . . . . 191
20.4 Description of the process to get each OPTIMIS metric . . . . . . . . . . . . 210
20.5 Description of the AWS S3 Wrapper tests . . . . . . . . . . . . . . . . . . . 217
20.6 Description of the AWS Collector tests . . . . . . . . . . . . . . . . . . . . 220

21.1 Some of the differences between an OCCI-compliant service and AWS . . . . 222

A.1 Resource instance structure object members description . . . . . . . . . . . . 227
A.2 Link structure object members description . . . . . . . . . . . . . . . . . . . 228
A.3 Action structure object members description . . . . . . . . . . . . . . . . . . 229

B.1 HTTP Return codes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 232
# List of Diagrams

11.1 CloudFoundry domain model .................................................. 77
11.2 Components model of the client library and its use by the Adapter. ........ 80
11.3 Authentication sequence diagram ............................................. 81
11.4 Create application sequence diagram ........................................ 82
11.5 Get space sequence diagram ................................................... 83
11.6 Delete application sequence diagram ........................................ 84
11.7 Update application sequence diagram ....................................... 85
11.8 Deploy sequence diagram ....................................................... 86
11.9 Deploy to environment sequence diagram .................................... 87
11.10 Upload and deploy to environment sequence diagram ....................... 88
11.11 Get app status sequence diagram ............................................ 89
11.12 Get running status sequence diagram ....................................... 89
11.13 Get application details sequence diagram .................................. 90
11.14 Start/stop application sequence diagram .................................... 91
11.15 Check app availability sequence diagram ................................... 92
11.16 List applications sequence diagram ........................................ 93
11.17 Create environment sequence diagram ..................................... 94
11.18 Delete environment sequence diagram ...................................... 94
11.19 Update environment sequence diagram ..................................... 95
11.20 Create database sequence diagram .......................................... 96
11.21 Get database list sequence diagram ........................................ 97
11.22 Delete database sequence diagram .......................................... 98
11.23 Export application sequence diagram ....................................... 98
11.24 Download/restore database sequence diagram .............................. 99

18.1 OCCI Data Model ................................................................. 149
18.2 OCCI Kind instances example ................................................ 151
18.3 OCCI Mixin instances example ................................................ 152
18.4 OCCI Actions defined in the standard ...................................... 153

19.1 Component-level view of the OPTIMIS architecture ....................... 167
19.2 High-level service deployment sequence diagram .......................... 169
19.3 Interface layer conceptual model ............................................. 174
19.4 Operations layer conceptual model .......................................... 175
19.5 Create Agreement Sequence Diagram ...................................... 177
19.6 Negotiate Sequence Diagram .................................................. 178
19.7 Terminate Agreement Sequence Diagram .................................... 179
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>19.8</td>
<td>Deploy Service Sequence Diagram</td>
<td>180</td>
</tr>
<tr>
<td>19.9</td>
<td>Query Service Sequence Diagram</td>
<td>181</td>
</tr>
<tr>
<td>19.10</td>
<td>Terminate Service Sequence Diagram</td>
<td>181</td>
</tr>
<tr>
<td>20.1</td>
<td>AWS CQoS operations layer conceptual model</td>
<td>193</td>
</tr>
<tr>
<td>20.2</td>
<td>AWS CQoS Deploy Service Sequence Diagram</td>
<td>195</td>
</tr>
<tr>
<td>20.3</td>
<td>AWS CQoS Get Amazon EC2 Client Sequence Diagram</td>
<td>196</td>
</tr>
<tr>
<td>20.4</td>
<td>AWS CQoS Query Service Sequence Diagram</td>
<td>197</td>
</tr>
<tr>
<td>20.5</td>
<td>AWS CQoS Terminate Service Sequence Diagram</td>
<td>198</td>
</tr>
<tr>
<td>20.6</td>
<td>AWS S3 Wrapper conceptual model</td>
<td>199</td>
</tr>
<tr>
<td>20.7</td>
<td>AWS S3 Wrapper Constructor Sequence Diagram</td>
<td>200</td>
</tr>
<tr>
<td>20.8</td>
<td>AWS S3 Wrapper Upload Sequence Diagram</td>
<td>201</td>
</tr>
<tr>
<td>20.9</td>
<td>AWS S3 Wrapper Progress Changed Sequence Diagram</td>
<td>202</td>
</tr>
<tr>
<td>20.10</td>
<td>AWS S3 Wrapper Download Sequence Diagram</td>
<td>203</td>
</tr>
<tr>
<td>20.11</td>
<td>AWS S3 Wrapper List Images Sequence Diagram</td>
<td>204</td>
</tr>
<tr>
<td>20.12</td>
<td>AWS S3 Wrapper Delete Image Sequence Diagram</td>
<td>205</td>
</tr>
<tr>
<td>20.13</td>
<td>AWS Collector conceptual model</td>
<td>207</td>
</tr>
<tr>
<td>20.14</td>
<td>AWS Collector Main Method Sequence Diagram</td>
<td>209</td>
</tr>
<tr>
<td>20.15</td>
<td>AWS Collector Get CloudWatch Measurement Sequence Diagram</td>
<td>211</td>
</tr>
<tr>
<td>20.16</td>
<td>AWS Collector Get OS Release Sequence Diagram</td>
<td>212</td>
</tr>
<tr>
<td>20.17</td>
<td>AWS Collector Get Disk Size Sequence Diagram</td>
<td>213</td>
</tr>
</tbody>
</table>
Acknowledgements

I want to dedicate this space to say a big and warm "Thank you!" to all the people that have made this project possible and that have walked by me during the last 5 years and a half. This goes to all of you.

To ARI for embracing me in this project, especially Ana Maria Juan and Francesco D’Andria for having made a bit of room in their busy schedule to attend and help me during the project and having reviewed the project report during the last weeks. This project would not have been possible without your help.

To all my colleges at ARI that have helped me a lot in getting used to the company and the work environment and in solving numerous issues with the projects and the servers, in especial to Sergio García that has helped me multiple times with all sorts of Linux and \LaTeX{} hacks.

To David Cunha from PTIN (Portugal Telecom Inovação) for helping me during the implementation of the adapters for Cloud4SOA and the evaluation of the hybrid cloud scenarios.

To Xavier Franch, in the first place for helping me in getting an internship at Atos, and in second place for supervising the project and providing me with helpful advice about the project, the documentation and the presentation. Thanks also to him and all the GESSI staff for embracing me in the GESSI group during the last 3 years and providing me with a LOT of experience that will certainly help me during my career.

To the people that have walked by me during the course of the degree, in especial to Pere Joan García, Oscar Mercè and Marc Rodríguez with whom I have shared numerous classes and successful projects; and with whom I have spent so many good and fun moments.

And last but not least to Cristina Palomares for supporting and encouraging me, always being by my side, and helping me with helpful feedback about the project report during the last weeks of the project.

To all of you, thank you very much!
Preface

The term “Cloud Computing” has recently emerged as one of the buzzwords in the IT industry. A lot of diverse IT companies seem to be using the term as their elevator pitch to attract more users. As it simplest, the “cloud” is the evolution of traditional data centers into virtualized environments which expose resources (computation, storage and applications) that can be used on-demand and that follow a “utility” pricing model where customers are charged based on their utilization of resources. The “cloud” presents itself into numerous forms, depending on the type of services it offers. The most notable forms are: Infrastructure-as-a-Service (IaaS) which offers on-demand computational resources that can be acquired in a self-service way; Platform-as-a-Service (PaaS) which offers a development and hosting platform in which all sorts of applications can be deployed; and Software-as-a-Service (SaaS) which offers complex applications to end-uses, who can use them remotely without the need to install or configure any software [NIS11].

Due to the novelty of this IT paradigm, there are still numerous issues and challenges to address in order to consolidate it. One of these issues is cloud interoperability. The high competitiveness in the cloud market prevents the major cloud providers (Amazon, Google, Microsoft, etc.) from agreeing into a set of standards but, instead, each of them presents its own proprietary interfaces and models. This puts off a lot of SMEs (Small and Medium Enterprises) that feel that will become “locked” into a cloud once they adopt it. The market could greatly benefit from having an interoperable environment in which applications and data could be more easily migrated between providers in the event of peaks of demand or Service-Level Agreement (SLA) violations.

This project will explore the problems of interoperability in the cloud, more concretely at the IaaS and PaaS layers, by contributing to two ongoing European research projects: Cloud4SOA\(^1\) and OPTIMIS\(^2\) at Atos.

Cloud4SOA aims at providing a broker-like multi-cloud platform that connects heterogeneous PaaS providers into a single location. Developers can use the platform to choose the provider that best matches the requirements of an application and manage its full lifecycle (deployment, governance, monitoring and undeployment) through it. Cloud4SOA is also able to monitor the SLA created between a developer and a provider in order to detect violations, in which case it may suggest the developer to migrate the application, action that can be taken through the same Cloud4SOA platform. This project will contribute to the platform by connecting it to an additional PaaS, which will be selected after analyzing different alternatives and analyzing the effort required to integrate the new provider with each of the Cloud4SOA use cases.

\(^1\)The project identifier is: CLOUD4SOA FP7-ICT-257953  
\(^2\)The project identifier is: OPTIMIS Project FP7-ICT-257115
OPTIMIS aims at optimizing IaaS cloud services by providing a set of inter-related tools for infrastructure providers (IPs), service providers (SPs) and service developers. The optimization covers the full service lifecycle (construction, deployment and operation). The tools for service providers and developers allow them to easily implement, package, deploy and operate services, run legacy applications on the cloud and make informed decisions based on TREC (Trust, Risk, Eco-efficiency and Cost) measurements. The tools for infrastructure providers, on the other hand, allow them to manage the infrastructure (VMs, servers, data storage, etc.) used to operate services. By monitoring the services running on the IP’s infrastructure, OPTIMIS can detect problematic issues and take the necessary actions to overcome them. One of such actions may be to burst a service or some of the resources that it is using to another OPTIMIS-enabled IP from the network of available providers. This project will contribute to this network in two ways: It will provide a new standardized communication channel with any provider that adopts the OCCI\(^3\) standard and it will add a new provider to the network.

With these contributions we expect to gain a deeper understanding of the current interoperability issues at the PaaS and IaaS layers, their impact in the development of multi-cloud tools and how these issues can be solved.

**Organization of the document**

This document is structured in 3 parts:

I Foundations and Project Plan

II Exploring interoperability at the PaaS layer

III Exploring interoperability at the IaaS layer

Part 1 – “Foundations and Project Plan” – presents some fundamental concepts of cloud computing, with special attention to the IaaS and PaaS layers, and introduces the current situation of the cloud with respect to interoperability. This part also describes the work plan of the project, commenting on the deviations that it suffered during its course; the goals of the projects and an analysis of its costs.

Part 2 – “Exploring interoperability at the PaaS layer” – gives more detail about the current interoperability issues at the PaaS layer, makes a comparative analysis of some of the currently most important PaaS providers with the intention of selecting one, and describes in detail the contribution that was made to Cloud4SOA by integrating the selected provider with the platform.

Part 3 – “Exploring interoperability at the IaaS layer” – gives more detail about the current interoperability issues at the IaaS layer and describes in detail the contributions made to OPTIMIS, namely, the standardization of a communication channel that can potentially be used to integrate any OCCI-compliant IP with OPTIMIS and the addition of a new IP to the network of OPTIMIS-enabled clouds.

The document finishes by giving some overall conclusions and describing the lessons learned about interoperability during the realization of the project.

---

\(^3\)OCCI is one of the standards for IaaS providers that is gaining more relevance. Details about it are given later in the document.
Part I

Foundations and Project Plan
Chapter 1

Background

1.1 Cloud Computing

When we plug an electric appliance to an outlet, we don’t care about how electric power is
generated or how it is possible that it gets to that outlet. That is possible because electricity is
virtualized, that is, it is readily available from a wall socket that hides the complex details about
how this electricity is generated, managed and transported. When extended to information
technology, this means delivering computational resources and functions without knowing the
internals of how these resources are obtained or managed [BBG11].

Cloud Computing aims at delivering large amounts of computational resources in a fully
virtualized manner, by aggregating large pools of physical machines (commonly known as
data centers) in a single system view. This huge aggregation of computational resources is
used to deliver on-demand portions of computational power, usually in the form of Virtual
Machines. An important characteristic of cloud computing is that computing power is delivered
as a utility, that is, consumers pay providers based on usage (“pay-as-you-go”) just as we do
with traditional public services such as water, electricity and gas.

There have been numerous attempts, both from researchers and practitioners, to describe
what exactly “cloud computing” is and what unique characteristics it presents. One such defi-
nition is provided by the NIST that says: “Cloud computing is a model for enabling ubiquitous,
convenient, on-demand network access to a shared pool of configurable computing resources
(e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned
and released with minimal management effort or service provider interaction.” [NIS11] De-
spite all the different definitions of cloud computing, the majority agrees on a set of features
that are inherent on the term [BBG11] [NIS11]:

Self-Service  Clouds must allow a self-service access to their services so that customers can
request, use, customize and pay for resources without needing any human intervention.

Per-Usage Metering and Billing  Clouds must allow consumers to request and use only the
necessary amount of resources without any up-front commitments. Moreover, resources
must be charged on short-term basis (usually by the hour) giving consumers the flexibility
to use resources only the time they need. This includes providing metering capabilities
for different type of resources (computing, storage, memory, bandwidth, etc.) so that
usage can be properly reported.
**Elasticity** One of the key strengths of cloud computing is the ability to quickly and dynamically change the amount of resources that are dedicated to a particular service. A cloud gives customers the illusion of infinite computing resources and, thus, they expect it to be able to increase the amount of resources in the event of high workloads and reduce them when the workload decreases.

**Customization** A cloud must be prepared to deliver resources to multiple customers with disparate requirements; therefore, delivered resources must be customizable. This includes being able to choose the amount of resources to rent (cpu power, memory, storage capacity), the OS to use, the application services to install, etc.

**Broad network access** The services offered by a cloud must be available over the network and accessible through different mechanisms: through a web interface, through a web service, etc.

**Resource pooling** Clouds should offer the possibility to specify the location of the rented resources independently of internal redistributions of resources due to optimization or clean-up tasks. This can be necessary for consumers that need to optimize the latency of their services or that need to deal with data confidentially issues.

Clouds are usually characterized across two independent features: their service model and their deployment model.

The deployment model describes where the computational resources offered by a cloud live and who can access them and can be classified in a public, private or hybrid deployment:

**Public deployment** denotes a cloud that is available to the general public, usually in a pay-as-you-go basis. This is the most typical deployment model and the one used by the majority of commercial providers such as Amazon or Rackspace.

**Private deployment** denotes a cloud that a company or organization internally deploys, usually in its own data center, and is used exclusively by the company or organization members. In most cases, creating a private cloud means restructuring the infrastructure of the organization by adding virtualization and cloud-like capabilities.

**Hybrid deployment** combines both of the above to create a mixed cloud that minimizes their disadvantages and combines their advantages. Usually such a deployment is created by supplementing a private cloud with capacity from a public cloud, so that the public infrastructure can be used in the event of load spikes or infrastructure failure. The process of temporarily renting capacity from a public cloud is known as cloud bursting.

The service model (the “as-a-Service” part) describes the type of services offered. Clouds are classified in mainly three different service models: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS) (see figure 1.1).

The IaaS and PaaS models are described in detail in the following subsections as they are the models directly related to this project. A major difference between these two and a SaaS is that the former are addressed to companies or ISV (Independent Software Vendors) while the latter is addressed to end-users. A SaaS is an application that resides on top the cloud stack and can be accessed by end users through web portals, avoiding the need to install any software locally. Traditional desktop applications such as text processing tools or spreadsheets are now
being ported to the cloud (Google Docs is an example of these tools). Also, all sorts of more complex applications are also starting to be cloudified, perhaps one of the most famous is the CRM offered by Salesforce.com.

![Image of Cloud stack and users](image)

**Figure 1.1:** High-level view of the Cloud stack and the users directly related to each layer.

### 1.1.1 Infrastructure as a Service

Infrastructure as a Service (IaaS) was the first layer of Cloud Computing that provided the opportunity to rent on-demand computational and storage resources in a per-hour basis. The IaaS paradigm popularity burst with the foundation of Amazon Web Services\(^1\) and its famous EC2 instances, which provided consumers with elastic computational resources that could be very easily managed in a self-service way. Nowadays, we can find lots of IaaS providers in the market such as Rackspace Cloud\(^2\), Windows Azure\(^3\), Terremark\(^4\), Google Compute Engine\(^5\) and Joyent\(^6\). The NIST defines Infrastructure as a Service as: “The capability provided to the consumer to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls).” \[^{NIS11}\] In

---

\(^{1}\)http://aws.amazon.com  
\(^{2}\)http://www.rackspace.com  
\(^{3}\)http://www.windowsazure.com  
\(^{4}\)http://www.terremark.com  
\(^{5}\)https://cloud.google.com/products/compute-engine  
\(^{6}\)http://www.joyent.com
practice, an IaaS provider makes use of hardware virtualization technologies to offer Virtual Machines to consumers. Consumers acquire complete control over these machines and can customize them, start or stop them, attach virtual disks to them, etc.

The roots of the IaaS paradigm can be found on grid computing and on the advances in hardware virtualization [BBG11]. Grid computing can be defined as the federation of computer resources, distributed across a number of locations, to reach a common goal. Commonly, grids are used for a variety of purposes rather than being tied to a single application. Grids, however, have been found to have some limitations such as difficulty in ensuring QoS, lack of performance isolation and lack of availability of resources with diverse software configurations. In this direction, advancements in hardware virtualization technologies have presented an opportunity to overcome these limitations, by allowing to split a data center or computer cluster into a number of individual and isolated machines that can be used for any purpose. Hardware virtualization allows running multiple operating systems and software stacks on a single physical platform. As seen in figure 1.2, a software layer called hypervisor mediates access to the physical hardware presenting to each guest operating system a virtual machine (VM), which has access to a limited set of the hardware resources. Researchers and practitioners have identified three main benefits that virtualization provides to workload management [BDF+03]:

**Workload isolation** is achieved since a program execution is fully confined inside a VM. This leads to improvements in security, reliability and performance control.

**Consolidation** of several individual and heterogeneous workloads onto a single physical platform can be achieved, which leads to better resources utilization.

**Application mobility** can be easily accomplished by encapsulating a guest OS state within its VM, allowing it to be suspended, serialized, migrated to a different platform and resumed immediately.

![Figure 1.2: High-level view of a virtualization environment.](image)

Apart from the common desirable features to all cloud platforms, an IaaS should be based in the following features and capabilities [BBG11]:

5
**Geographic presence** IaaS providers typically build several data centers distributed around the world. For some consumers this can improve availability and responsiveness, as well as deal with law issues with regard to data confidentiality.

**User interface and access to servers** An IaaS provider should offer as many ways as possible to interact with its cloud. Usually, providers offer a combination of Web interfaces, Command-Line interfaces (CLI) and Web services APIs.

**Advance reservation of capacity** An IaaS provider should allow the option to reserve resources for a specific time frame in the future, thus ensuring that cloud resources will be available at that time. Consumers use this feature when they expect to have a high peak of demand, for instance, an online shop could reserve an exceptionally large amount of resources during the days before Christmas.

**Automatic scaling and load balancing** Automatic scaling gains special importance in the IaaS layer. An IaaS should allow consumers to set conditions for when they want their resources to scale up and down, based on a variety of infrastructure and application-specific metrics. Moreover, when the number of virtual machines is increased, incoming traffic must be automatically balanced among the available machines.

**Service-Level Agreement** Service-Level Agreements (SLAs) are offered by IaaS providers to express their commitment to deliver a certain Quality of Service (QoS). To customers, this serves as a warranty. Such an SLA usually contains availability and performance guarantees, with availability being clearly the focus point for most IaaS.

### 1.1.2 Platform as a Service

In the last years, a new paradigm has emerged between the bottom most layer of the cloud (IaaS) and the up most (SaaS) with the name of Platform-as-a-Service (PaaS). The NIST defines Platform-as-a-Service as: “The capability provided to the consumer to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages, APIs, services, and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment.”[NIS11]

The roots for the emergence of this new middle layer can be found on Application Service Provision (ASP) concepts, where services such as Salesforce.com were offering a customization layer of some sort for their platforms. Soon, driven by the existence of these services and the eruption of this market with the entrance of Google’s App Engine, the middle layer between IaaS and SaaS became clear.

The main function of a PaaS is to provide a layer of abstraction on top of an IaaS, in which applications can be developed, deployed, tested and governed without worrying about the underlying physical structure. Furthermore, multiple programming models and specialized services (databases, authentication, payment gateways, etc.) are offered as building blocks for applications [App09].

Apart from the common desirable features to all cloud platforms, a PaaS should be able to support the following set of features and capabilities [BBG11]:

---

**NIS11**: National Institute of Standards and Technology

**App09**: Application Provisioning

**BBG11**: Benefits of Building Blocks
Programming models, languages and frameworks A PaaS should support the development and efficient execution of applications that require specialized programming models. In the cloud computing domain, the most common activities that require specialized models are: processing of large data sets in clusters of computers (using the MapReduce model), development of request-based Web services, definition and orchestration of business processes in the form of workflows (Workflow model), and high-performance distributed execution of diverse computational tasks. Also, a PaaS should support a set of programming languages and frameworks. Currently, the most commonly supported languages are Java and .NET, along with the Spring, Java EE and diverse .NET frameworks.

Persistence Options A PaaS should provision applications with access to a persistence layer. The ability to save and store data in a structured and efficient way has become essential in all domains of software development, with relational databases being the most popular persistence method. In the domain of cloud computing, however, this technology is becoming short as it lacks scalability to handle vast amounts of data. In order to tackle this, distributed storage technologies have emerged which seek to be robust and highly scalable at the expense of relational structure and convenient query languages. In the same direction, innovative technologies such as NoSQL are also seeking to solve the problem of the management of very large amounts of data (of the order of petabytes).

Developer Tools A PaaS is expected to offer a set of tools to both let developers interact with the cloud (self-service) and to ease the development of complex applications, especially those requiring access to services such as databases. PaaS providers usually offer a CLI (Command Line Interface) to perform all operations (create applications, deploy, monitor, etc.) and a graphical application (usually a plug-in for Eclipse or similar IDEs) to perform a subset of these operations in a more intuitive and usable way. Additionally, it is common to offer a web service to allow the execution of these operations programmatically.

Automatic Scaling As discussed earlier in this work, one of the beauties of the cloud is its ability to scale resources up and down according to workload changes. With respect to a PaaS, it is desirable to offer a way to automatically scale applications in terms of assigned memory, CPU power, number of running instances, etc. or at least offer a manual way of doing so.

The literature has elaborated all sorts of PaaS categorizations. Red Hat, for instance, classifies PaaS according to their purpose, ranging from a SaaS with extensions to a General-purpose PaaS [Red10] (see table 1.1). While the more specialized PaaS deliver successful results in tackling specific problems, the more general ones are addressed to organizations that need to use a variety of technologies to deploy both new and existing applications.

Another relevant distinction between PaaS is the type of underlying infrastructure, i. e. the IaaS, in which they can be deployed: own infrastructure, tied to a single IaaS or multiple IaaS. Providers such as Google App Engine, Windows Azure or AWS Benstalk are examples of the first category, with the main benefit being that the developer does not need to worry about the infrastructure since it will be provided by the platform. The second category includes Heroku and Aneka, both tied to Amazon EC2 servers. Finally, the “write once and deploy anywhere” [Red10] category is clearly the most flexible and is becoming the market trend as well as the elevator pitch of a lot of companies, such as Cloud Foundry, Stackato and RedHat Open Shift.
### Type of PaaS | Description
--- | ---
**SaaS with extensions** | Customize and extend the capabilities of a SaaS application. Salesforce.com (the SaaS), for instance, offers a customization layer through force.com (the PaaS).

**Purpose-built PaaS** | A PaaS that simplifies the development of a specific type of application, with a limited set of capabilities.

**PaaS tied to a single application paradigm** | Provides general capabilities, but supports only one programming model or development/deployment environment.

**PaaS tied to a single cloud** | May provide general capabilities and programming models, but is tied to only one type of private or public infrastructure (IaaS).

**Middleware hosted in the cloud** | Eases distribution of middleware across the organization, but adds no other value.

**General-purpose PaaS** | Comprehensive, open, and flexible solution that simplifies the process of developing, deploying, integrating and managing applications in public and private clouds. The majority of well-known providers fit in this category.

---

**Table 1.1: A classification for PaaS platforms**

### 1.2 Cloud Computing Interoperability

Interoperability has been widely identified as one of the highest risks and challenges of Cloud Computing [BBG11, CH09]. Current cloud computing solutions have not been built with interoperability as a primary concern; rather, they “lock” customers into their own infrastructure, processes and interfaces, preventing the portability of data and applications. Probably the reasons that have caused this situation are both the rapid growth of the cloud market and the increasing competition between the leading vendors (Amazon, Google, Microsoft, etc.), each of them promoting their own, incompatible formats [MHS09].

It is clear that an interoperable cloud environment would benefit customers, as they would be able to migrate their data and applications between heterogeneous cloud providers without compromising the integrity of the data or having to reconfigure the applications. Moreover, they would be able to interact with any cloud provider in the same way and compare different providers across a set of common characteristics such as resources, pricing and Quality of Service (QoS). This would further attract more SMEs (Small and Medium Enterprises) into the cloud since they won’t feel the fear of getting locked-in into one single provider but, rather, would have the option to change provider when it stops fulfilling their requirements.

The solution to this situation is the creation and adoption of standards. In this direction there have been numerous efforts to create open standards for cloud computing. The Cloud
Computing Interoperability Forum (CCIF), for instance, was formed by a set of organizations such as Intel, Sun and Cisco in order to “enable a global cloud computing ecosystem whereby organizations are able to seamlessly work together for the purposes for wider industry adoption of cloud computing technology”. One of the outcomes of this body has been the Unified Cloud Interface (UCI) that aims at creating a standard point of access to an entire cloud infrastructure. In a similar direction, but related to virtualization, the Open Virtual Format (OVF) aims at facilitating the packing and distribution of software to be run on VMs so that virtual appliances can be made portable between heterogeneous hypervisors.
Chapter 2

Goals

This project will contribute to two ongoing European research projects (within the FP7 framework), namely, Cloud4SOA and OPTIMIS, by providing mechanisms to enhance the interoperability between their tools and the cloud platforms with which they interact. On one hand, the contribution to Cloud4SOA, which corresponds to the work developed within the “Exploring interoperability at the PaaS layer” part, has the following goals:

- Gain a fairly deep understanding of existing private PaaS offerings by studying and comparing them.
- Learn and experiment the typical workflow that developers undertake with a PaaS cloud.
- Select the most appropriate private PaaS platform and deploy it into the company’s data center.
- Integrate Cloud4SOA with the deployed instance and test its proper functioning.
- Evaluate if Cloud4SOA is able to properly handle an application deployed into the private cloud and migrate it to a public cloud or vice-versa.

On the other hand, the contribution to OPTIMIS, which corresponds to the work developed within the “Exploring interoperability at the IaaS layer” part, has the following goals:

- Study the current main standards for IaaS management and operation.
- Understand the architecture of the components that have to interact with the non-OPTIMIS OCCI clouds.
- Study the OCCI standard in depth and design the OCCI-compliant interface that the OPTIMIS proxy exposes.
- Design, implement and test the component that will communicate to the non-OPTIMIS cloud proxy, in an OCCI-compliant way.
- Evaluate if OPTIMIS is able to properly handle a bursting scenario with the implemented component.
- Extend the OPTIMIS eco-system by adding support for Amazon EC2. This includes the design and implementation of a proxy able to use Amazon EC2 resources during a cloud bursting operation.
Chapter 3

Project Plan

3.1 Initial plan

The project is expected to last for 5 months and a half approximately working full-time, starting on the 17th of September. The Gantt diagram of the different phases and tasks of the project can be seen in figure 3.1. We are conscious that is very likely that the actual work deviates from the plan at some point due to unexpected circumstances or mistaken predictions. For this reason, we will keep track of how the project actually goes in order to compare it with the plan at the end of the project.

3.1.1 Phases

Learning & Research

The first phase of the project will run up to the 28th of September and consists on getting an overview of Cloud Computing in general and gain a more in-depth insight on concepts closely related to the project, such as standards for the PaaS and IaaS layers and the Cloud4SOA and OPTIMIS projects. This task will also involve comparing several private PaaS vendors and selecting the most appropriate for deployment into the company’s data center. This last task will provide further insight on concepts related to cloud computing and will be an input for the “Exploring interoperability at the PaaS layer” part.

PaaS Part

During the next month and a half approximately (up to the 5th of November) the tasks inside the PaaS part phase will be performed. These tasks consist on installing and configuring the private PaaS instance on one of the servers of the company, integrate it with Cloud4SOA and evaluate the result of the integration.

IaaS Part

Finally, during the rest of the duration of the project, the tasks inside the IaaS part phase will be performed. These tasks basically consist on the analysis, design and implementation of the OCCI communication link between OPTIMIS and any OCCI-based provider and on the integration of AWS with OPTIMIS.
Figure 3.1: Gantt diagram describing the initial project planification
3.1.2 Working hours estimation

With the different tasks decided and planned and considering that we will work full-time during the duration of the project, except for the company holydays, we estimate that the project will take 792 hours to complete distributed as shown in table 3.1.

<table>
<thead>
<tr>
<th>Task</th>
<th>Hours</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Document Writing</td>
<td>90</td>
</tr>
<tr>
<td>Project Document Review</td>
<td>20</td>
</tr>
<tr>
<td><strong>Learning &amp; Research</strong></td>
<td></td>
</tr>
<tr>
<td>PaaS comparative &amp; selection</td>
<td>35</td>
</tr>
<tr>
<td>Books and papers reading</td>
<td>25</td>
</tr>
<tr>
<td>Cloud4SOA testbed set-up</td>
<td>20</td>
</tr>
<tr>
<td><strong>PaaS Part</strong></td>
<td></td>
</tr>
<tr>
<td>PaaS installation &amp; testing</td>
<td>35</td>
</tr>
<tr>
<td>Cloud4SOA integration</td>
<td>75</td>
</tr>
<tr>
<td>Evaluation</td>
<td>60</td>
</tr>
<tr>
<td><strong>IaaS Part</strong></td>
<td></td>
</tr>
<tr>
<td>Standards study</td>
<td>13</td>
</tr>
<tr>
<td>OCCI Integration Analysis</td>
<td>24</td>
</tr>
<tr>
<td>Arsys Service Design</td>
<td>30</td>
</tr>
<tr>
<td>Proxy Design</td>
<td>65</td>
</tr>
<tr>
<td>Proxy Implementation</td>
<td>110</td>
</tr>
<tr>
<td>Integration with AWS</td>
<td>190</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>792</td>
</tr>
</tbody>
</table>

Table 3.1: Working hours estimation

According to the rules established by the Barcelona School of Informatics for the realization of the PFC, which determine that the work load by the student is of about 20 hours per credit, the total number of hours estimated for the realization of the project should be 750, value that closely adjusts to the number of hours estimated for this project in particular.

3.2 Deviation from the initial plan

As initially expected, there were some deviations from the initial planning during the course of the project, mostly due to the lack of knowledge about Cloud Computing which has required researching and learning more about it; and the costly internal processes required to acquire resources inherent on any big company. Despite that, almost all the goals have been fully completed, only the integration with Arsys and the integration with AWS have been partially completed leaving some work out of the project due to a lack of time.

Within the PaaS part, the installation and integration of CloudFoundry took more time than expected due to the fact that CloudFoundry is still a young Open Source project and there is a lack of documentation about the platform. This has required taking extra time diving into the CloudFoundry code and getting involved in the community in order to understand the platform in detail and to clarify all questions. In addition to that, the actual installation of CloudFoundry...
was delayed to the second half of November because the necessary resources in the data center were not provided until then. This situation complicated the schedule of the project since during this waiting time tasks from the IaaS part were intertwined with tasks from the PaaS part. Another consequence of this delay was that the evaluation task had to be also delayed and could not be performed as exhaustively as it was intended.

Something similar happened in the IaaS part with the implementation and testing of the OCCI proxy since it was dependent on the implementation of the service by part of Arsys and they didn’t have it ready until half February. We were able to implement the proxy but were not able to fully test it until the service was ready and running. This, again, complicated the schedule by having to intertwine the “Integration with AWS” tasks with this.

Due to these changes of schedule and the fact that some tasks took more time than initially expected, the goals related to the last tasks, namely the integration of OPTIMIS with OCCI and AWS, were not completely fulfilled. Specifically, the new OCCI proxy could not be used to evaluate the bursting scenario and the integration of AWS was not fully performed, leaving out the work required to connect the implemented components to OPTIMIS.
Chapter 4

Cost analysis

After having prepared the schedule of the project we can make an economic assessment about it. To do so, we have to take into account both human and material resources.

4.1 Human Resources

The higher cost of the project is in human resources. In European Projects, this cost is calculated using the “person-month” concept which is an abstraction of the required work effort to complete a task. The tasks performed in part II map almost directly to a task within Work Package 8 (WP8) of Cloud4SOA. This task, as stated by the Description of Work (DoW), has a required effort of 8 Person-Months (PMs). With respect to part III, the tasks performed are not reflected in the OPTIMIS DoW since they have been extra tasks not initially planned. However, comparing the effort required to complete these tasks with the effort required to complete part II, we can assign an effort of 5 PMs to them.

Taking the average monthly rate that Atos calculates for PMs, the total cost of the 13 PMs is about 78,000 €. However, this cost is a simulation considering a team of people from the different partner organizations. Being this a final degree project, there has only been one person working on it with the supervision of the director and some support from one of the partners during the Cloud4SOA evaluation task. Taking into account that the project has lasted for 6 months, the actual cost for the student has been of 6 PMs which equal to 4,000 € approximately.

4.2 Material Resources

In order to develop the project the following material goods have been required:

- A laptop with all the necessary software installed, valued in 1000 €.
- A negligible cost of less than 10 € from Amazon usage fees, both from EC2 and from S3.
- Books and electronic resources valued in 125 € approximately, which includes some paperback books and a subscription to an electronic library.
We have also used the infrastructure in the company data center to host the CloudFoundry installation. However, given that the server in which it has been deployed is shared with other applications, it is difficult to estimate its actual cost.
Part II

Exploring interoperability at the PaaS layer
Chapter 5

Introduction

This part of the project focuses on interoperability at the Platform-as-a-Service (PaaS) layer. Platform-as-a-Service is a fairly new concept that has emerged from the evolution of the services offered by some of the major Cloud providers. It comprises a middle layer between the IaaS and the SaaS aimed at providing a scalable development and hosting environment for all sorts of applications. The main goal of a PaaS is to “make the developer’s life easier” by providing a set of tools and resources to help in the development and deployment of scalable applications, be them simple or complex.

Since the emergence of this concept it has gained a fair amount of popularity and, today, we can find lots of PaaS providers in the market: Amazon Elastic Beanstalk, Google App Engine, Heroku, OpenShift, etc. In such a competitive field, where each provider imposes its own “lock” in an effort to become the de facto platform, interoperability and application portability gets highly damaged. This heterogeneity prevents the establishment of an agreement on a widely accepted, standardized way to input/output Cloud details and specifications [MHS09]. However, cloud developers could highly benefit from an interoperable cloud environment, where applications can be migrated from one vendor to another with minimal hassles and risks. This would let them compare Cloud offerings across a set of characteristics such as resources, pricing, available application services, Quality of Service (QoS), etc. and choose the most cost-effective provider. Furthermore, this would let them easily migrate an application in case that the chosen provider stops delivering the expected results.

The Cloud4SOA European research project pretends to make a step forward in this direction by providing a broker-like multi-cloud platform that connects multiple heterogeneous PaaS providers into a single location. The platform allows developers to search among a variety of PaaS for the one that best fits their requirements and to seamlessly deploy, govern and monitor application on it. Furthermore, Cloud4SOA also allows migrating applications from one PaaS to another when an unexpected condition happens, such as an SLA violation.

Currently, Cloud4SOA only manages public clouds (Amazon Benstalk, CloudBees and Heroku, among others). The goal of this part of the project is to explore the possibility of making hybrid deployments through the platform. To this end, a private PaaS platform will be selected and installed on-premise in the Atos infrastructure, and Cloud4SOA will be extended to support this platform. Once this test-bed is set up, several hybrid deployment scenarios will be used to evaluate the usability and performance of Cloud4SOA when facing such scenarios.

The rest of this part is structured as follows. Chapters 6 and 7 provide some background that will help the reader understand the rest of the document. Next, chapter 8 evaluates and com-
pares a set of private PaaS platforms and explains which one has been selected. Then, chapter 9 provides an overview of the selected platform along with an explanation of the installation environment and process. Following, chapter 10 analyzes how the selected platform can be integrated with Cloud4SOA. Chapter 11 shows the design and implementation of the adapter (the extension to the Cloud4SOA platform). After that, chapter 12 shows the evaluation results of the different hybrid deployment scenarios. Finally, chapter 13 concludes the part.
Chapter 6

Cloud4SOA

Cloud4SOA is a European research project started in 2010 and with 3 years of duration. The goal of the project is to provide a platform to reduce the semantic interoperability barriers between different cloud vendors in order to move a step forward in the realization of a global Cloud market [DBG+12]. Cloud4SOA introduces a broker-based architecture whose main goal is to address and tackle semantic interoperability challenges at the PaaS layer.

The Cloud4SOA tool aims at facilitating Cloud-based application developers in searching for, deploying and governing their business applications in the PaaS that best fits their needs. Additionally it provides support for migrating an application from one PaaS to another without compromising the integrity of the application or the data. This functionality is achieved by semantically interconnecting heterogeneous PaaS offerings through a set of adapters that transform Cloud4SOA-based requests into platform-specific requests for each different PaaS.

In order to handle the interoperability issues, Cloud4SOA introduces a three-dimensional semantic interoperability framework [LKT11] that aims to capture any sort of semantic interoperability conflict at the PaaS layer, while also mapping it to the appropriate PaaS entity and type of semantics.

Cloud4SOA focuses its functionality in four use cases: Matchmaking, Deployment, Migration and Monitoring.

6.1 Matchmaking

This use case allows to search among the existing PaaS offerings for those that best match the user’s requirements. The matchmaking mechanism uses semantic technologies to align the user requirements with the PaaS offering features, even if they are expressed in different forms.

The matchmaking algorithm capitalizes on the PaaS and Application semantic models, and aligns them in order to be able to compare them. The PaaS model captures technical information and characteristics about the platform such as supported Programming Languages and pricing model. The Application model represents an application created by a cloud-based application developer that needs to be hosted by a PaaS offering and it captures information such as the programming language, required services, required QoS (Quality of Service), etc. Using these models, “PaaS offering requests” are matched with the available “PaaS offerings” in order to provide the set of offerings that best match the requests.

Moreover, the matchmaking algorithm is able to identify equivalent concepts between diverse PaaS offerings and “harmonize” them in order to be able to match them with the user’s
requirements. For example, Heroku uses *dynos* as the basic unit of computation while other PaaS providers use the number of CPUs.

### 6.2 Deployment

After the developer decides, driven by the recommendations of the matchmaking, on the best PaaS to deploy, she can easily deploy an application to it by providing a set of required details, such as his credentials on the platform. The deployment performs an analysis of the application’s requirements and builds a specific application deployment descriptor. This descriptor is compliant with the format defined by the selected PaaS offering. The deployment is made through the harmonized Cloud4SOA API, which is implemented by all the platform-specific adapters.

### 6.3 Migration

After an application has been running for some time on a PaaS, the developer may realize that the platform is not delivering good enough results and may decide to change to another provider. In this direction, migration offers the functionality to achieve this. First, it retrieves the deployed application archive and translates the application requirements into a platform-specific deployment understandable by the new PaaS provider.

### 6.4 Monitoring

Once the developer has deployed an application into a PaaS provider, she can get real-time performance statistics such as response time and application health. The Monitoring module “harmonizes” metrics offered by different PaaS providers in a similar way to the matchmaking algorithm.

One of the key aspects of Cloud4SOA with respect to monitoring is the management of Service Level Agreements (SLAs). SLAs describe the service that is delivered, the functional and non-functional properties of the resources, and the duties of the parties involved. Additionally, SLAs define guarantees for the functional and non-functional resources properties. These guarantees define service level objectives that must be met in order to fulfill the agreement along with the compensation action to take in case that the guarantee is not fulfilled.

Cloud4SOA provides a RESTful implementation of the WS-Agreement standard in order to implement the SLA control and enforcement. Additionally it provides a negotiator component to perform automatic negotiations on behalf of PaaS providers, based on the semantic description of offerings and the QoS requirements specified by an application developer.
Chapter 7

Interoperability at the PaaS layer

This section dives a little deeper into the current interoperability issues at the PaaS layer and briefly describes the most important standardization initiatives that have appeared with the purpose of solving these issues.

7.1 Interoperability problems at the PaaS level

Since the emergence of the PaaS concept a lot of providers have appeared, each one showing their particular characteristics and exposing unique and proprietary ways of interacting with them. The major interoperability problems between different PaaS providers appear in their offered development tools which usually use their proprietary runtime frameworks, programming languages and APIs. These differences difficult the uniform interoperation of PaaS providers and the consistent migration of application code between them.

PaaS runtime frameworks often vary significantly. On one hand there are frameworks based on traditional application runtimes and visual programming concepts. On the other hand, there are frameworks with pluggable support for multiple application runtimes [Cha10]. Furthermore, PaaS also widely differ in supported programming languages. Google App Engine, for instance, supports Java and Python while Force.com supports a proprietary programming language called Apex. The lack of standards is also present in SDKs which tend to be vendor-specific, locking-in some parts of applications which should be redesigned in case of migration.

Data types and storing methods also differ widely. Some of them use existing services such as MySQL while others use proprietary stores and access mechanisms. For instance, Microsoft Azure uses the SQL Azure Database and Google App Engine uses an on-premise DataStore which is accessed using the GQL language, a restricted language similar to SQL.

Conflicts also arise in the back-end functionalities of applications like accounting, billing, metering or advertising since each PaaS provider introduces its own standards.

In summary, the current PaaS scene has the following major interoperability problems:

- Lack of common/standardized Cloud PaaS APIs.
- A great diversity in frameworks, languages, toolsets and SDKs.
- Heterogeneous data types and storing methods.
- Non-interoperable accounting, billing, metering and advertising services.


7.2 Standards for PaaS Interoperability

With the intention of overcoming the aforementioned issues and create a network of cooperating clouds, some standardization bodies have proposed standards that can be adopted by PaaS providers to make a step forward in the achievement of such a network. However, probably due to the novelty of the PaaS, the efforts spent on this direction are far lesser than the efforts spent in standardizing the IaaS cloud. In contrast to standards for the IaaS, it is hard to find standard addressed specifically to the PaaS, most of the standards that address this layer, address other layers too. Some of the standards that address the PaaS, as well as other layers, are: some standards from the Distributed Management Task Force (DMTF)\(^1\) that address service-level agreements (SLAs), quality of service (QoS), workload portability, automated provisioning, and accounting and billing [DMT09], the Unified Cloud Computing (UCC) from the Cloud Computing Interoperability Forum (CCIF)\(^2\) and the some standards released by the Global Inter-Cloud Technology Forum (GICTF)\(^3\).

One standard that addresses only interoperability problems at the PaaS layer is the Cloud Application Management for Platforms (CAMP)\(^4\) from the Organization for the Advancement of Structured Information Standards (OASIS). The CAMP standard aims at leveraging the inherent similarities in any PaaS, such as the applications lifecycle management or monitoring processes, to produce a generic application and platform management API that is language, framework and platform neutral.

\(^1\)http://www.dmtf.org/
\(^2\)http://www.Cloudforum.org/
\(^3\)http://www.gictf.jp
\(^4\)http://www.cloudspecs.org/paas/
Chapter 8

Private PaaS Comparison and Selection

We have evaluated and compared a set of private PaaS offerings in order to select the one that will be used in Cloud4SOA to evaluate the hybrid cloud deployment scenarios. The selected PaaS will be deployed on-premise in the Atos data center and will be integrated into Cloud4SOA through the implementation of an adapter.

In the next sections a comparative analysis of several PaaS offerings is made, along with a selection of the most appropriate one.

8.1 Private PaaS comparative analysis

We have compared 5 of the currently most important private PaaS platforms:

- CloudFoundry
- Aneka
- Stackato
- Apprenda
- CumuLogic

For each of them we give an overview and evaluate their capabilities with respect to the main features of a PaaS. Specifically we evaluate the following areas:

- Tools and functionalities offered to support development activities.
- Tools and functionalities offered to support the deployment of applications.
- Monitoring and Governance functionalities offered.
- Data services offered.
- Scaling capabilities offered (elasticity).

Note that this document compares PaaS capabilities from the developer perspective; it does not include an evaluation of the PaaS management capabilities.
8.1.1 CloudFoundry


Overview

CloudFoundry is a PaaS platform developed by VMWare that can be found in basically two shapes: A public cloud ran by VMWare and the CloudFoundry engine itself released as an Open Source project, allowing anyone to build a private PaaS and to contribute to the engine by extending it.

One of the strengths of CloudFoundry is that it encompasses a wide range of programming languages, frameworks and application services which in the open version of the engine is even wider as anyone can contribute by extending it. Similarly, applications created using CloudFoundry can be deployed to any IaaS.

A weakness with respect to the rest of the offerings is that it does not provide a web Interface at the moment, Management of the cloud has to be done through a CLI (Command Line Interface).

Products

Cloud Foundry offers three different forms of a PaaS:

CloudFoundry.com Public instance of the open CloudFoundry PaaS operated by VMWare. CloudFoundry.com is now in beta and can be used for free.

Micro Cloud Foundry Complete CloudFoundry instance contained within a virtual machine. This product is intended to be installed directly in a developer’s computer in order to simulate the interaction with a real cloud foundry-based private or public cloud with the same environment, ensuring that applications that run locally will run in the cloud too.

Cloudfoundry.org Open Source project hosting the Cloud Foundry technology. With the tools within this project a private PaaS can be built and deployed on top of any IaaS. Different configurations can be built, achieving different supported languages, frameworks and application services.

Installing and running

In order to install Cloud Foundry the following minimum requirements have to be met:

- VM with an Ubuntu 10.04.4 server image
- At least 1GB of memory in the VM

Development

CloudFoundry does not offer the possibility to run development operations directly into the cloud (such as create the skeleton code of certain types of applications), however, it can be integrated with development tools (such as Eclipse) in order to ease development. Micro CloudFoundry can be used as a way to palliate the inconvenience of having to build a whole devel-
Development environment locally, by allowing to deploy to a local mini-version of CloudFoundry where all services are already available.

Debugging support can be turned on in the private cloud to enable debugging operations directly into the cloud from development tools (such as Eclipse).

The platform offers the possibility to open tunnels to services to interact with application services provisioned in the cloud. This way, it is possible to interact with any service as if we were accessing it locally.

CloudFoundry uses an auto-reconfiguration mechanism by which applications that follow a set of standards can be bound automatically to application services, without having to lock-in the code to the specific details of the connection.

It offers the following tools:

- CLI client.
- Eclipse-based SDK for Java-based apps.

The following programming languages and frameworks are supported:

- Java (Spring framework)
- Ruby (Ruby on Rails and Ruby and Sinatra frameworks)
- JavaScript (Node.js framework)
- Groovy (Grails framework)
- PHP
- Python (Django framework)

Deployment

When developing JVM based (Spring, Grails, etc.) applications using the Eclipse plug-in, deployment can be easily made through the same plug-in interface. However, applications in other languages (PHP, Python, etc.) have to be deployed using the CLI. During the deployment process, the app can be easily bound to different application services (such as a database) and it can be set how much memory and how many instances assign to it.

CloudFoundry has an intelligent mechanism for uploading application code that only uploads new and changed resources. Files that have been already uploaded are skipped. This becomes especially useful when working with libraries, since they will be uploaded only once.

CloudFoundry also allows to update deployments without users experiencing any downtime by using a mechanisms that creates a new applications, maps it to the same deployment URL, then dissociates the mapping of the old app to the URL and finally delete the old application.

Monitoring and Governance

CloudFoundry does not offer practically any native monitoring capabilities, except for Java applications, for which the Insight component can be used to get runtime information. Despite that, it offers a free New Relic service which includes a wide range of monitoring tools.

Basic governance operations (start, stop, etc.) are supported by all clients (VMC and Eclipse plug-in).
Data

The following persistence options are included:

- Relational Databases (MySQL, PostgreSQL, ...)
- NoSQL Databases (CouchDB, MongoDB, ...)

Elasticity

Cloud Foundry does not support auto-scaling. Applications can be manually scaled by adding/removing instances and memory.

8.1.2 Aneka

Site: http://www.manjrasoft.com/products.html

Overview

Aneka is a commercial platform that can enable a private PaaS. Aneka is more focused on High-Performance Computing (HPC) applications rather than Web applications [BBG11]. For this reason, applications that run in Aneka are usually standalone applications, which differ from Web applications in that they are not continuously running but instead run just the time to complete the necessary jobs.

Aneka can integrate with a variety of IaaS clouds, such as private ones based on VMWare and Citrix Zen and public ones (Windows Azure, Amazon EC2 and GoGrid). Moreover, it offers a fairly transparent view of computing resources (underlying IaaS infrastructure), thus allowing its low level management. It also includes automatic failover capabilities that can increase availability based on SLAs.

One of the main strengths of Aneka is its resource provisioning service that is able to automatically scale an application based on its requirements.

Installing and running

In order to install Aneka the following minimum requirements have to be met:

- At least 1 machine with 3 Gb of RAM and 2 cores.
- One of the following OS:
  - Linux if the Mono Runtime is available
- .NET Framework 2.0 or Mono Runtime 2.6 or higher.
- One of the following DB engines:
  - SQL Server 2005, SQL Server 2008 or MySQL 5.1+
Development

Aneka supports the Task, Thread and MapReduce programming models. Using a set of APIs, developers can express distributed applications using the aforementioned programming models, thus taking the maximum profit from a distributed execution environment, but at the same time making applications highly locked in. It supports any programming language that can run on the .NET Runtime Environment.

The only tool it offers is a standalone SDK that can be used to manage the infrastructure and to create and run applications.

Deployment

As stated in the previous sections, Aneka differs from others PaaS in that in usually runs standalone applications rather than Web ones. This means that an application is not really deployed to the cloud, but instead it just runs on the cloud the time necessary to complete a set of tasks.

Applications that are to be run in an Aneka cloud make heavy use of specific Aneka constructs and expressions in order to maximize efficiency.

Monitoring and Governance

Aneka offers limited support for monitoring since it is not strictly necessary on a cloud of this sort.

Data

The following persistence options are included:

- Relational databases
- Hadoop Distributed Files System (HDFS)

Elasticity

One of the strengths of Aneka is its horizontal auto-scaling features. Automatic scaling is done through the Aneka Resource Provisioning service that scales an application based on its desired QoS. Since Aneka can have a heterogeneous set of underlying IaaS, it can also automatically burst to a public cloud [BBG11].

8.1.3 Stackato

Site: http://www.activestate.com/stackato

Overview

Stackato is a commercial private PaaS platform based on a set of components, the main one being CloudFoundry. Its goal is to provide uniform access to a wide variety of programming languages, frameworks and underlying IaaS infrastructures (any IaaS is supported).
Stackato permits the deployment of applications with and without a web front-end, so it can be used for a variety of purposes. Also, it provides tools to ease the migration of data by dumping the data in a DB to a file and then importing it to another DB.

**Products**

Stackato offers two different forms of a PaaS:

- **Stackato Micro Cloud**  A PaaS with the same characteristics as the Micro CloudFoundry.
- **Stackato Enterprise**  Commercial private PaaS version.

**Installing and running**

In order to install Stackato the following minimum requirements have to be met:

- x86 processor with support for Virtualization.
- 3GB+ memory
- 3GB+ disk space

**Development**

Stackato does not offer the possibility to run development operations directly into the cloud (such as create the skeleton of applications), but it integrates with development tools (such as Eclipse) in order to ease development.

It also has an auto-configuration mechanism like CloudFoundry. Moreover, it can recognize an application’s middleware dependencies and automatically install and configure all required software. The following programming languages and frameworks are supported:

- Java (Spring, JBoss, Grails, Lift and Buildpack - Java)
- Python (Django)
- Perl
- .NET (C#, ASP.NET, etc.)
- PHP
- Ruby (Ruby on Rails, Ruby and Sinatra)
- Erlang
- Scala
- Clojure
- Javascript (Node.js)

It offers the following tools:
- CLI client
- Eclipse plug-in
- Komodo IDE based tool
- Web interface

The usage of the tools is very similar to CloudFoundry.

Deployment

Deployment operations in Stackato are done in a similar way to CloudFoundry, including the selection of allocated memory and instances, the intelligent upload of resources when deploying an application and the ability to redeploy an application without experiencing any downtimes.

Stackato includes a Health Monitor aimed at maximizing availability. The monitor continuously reports on the health of the different nodes and balances the load between application instances. Moreover, if an instance goes down another one gets the load assigned.

It allows to define hooks to the deployment process. A hook is a process that is executed at a certain point in the deployment (such as pre-staging, post-staging, pre-running, etc.) and that can be used to perform configuration operations or to take any other sort of action.

Monitoring and Governance

Stackato offers an API to get basic data about how computing resources are being used. It relies on a third party tool (*New Relic*) to perform extensive monitoring.

It allows the same basic governance operations that CloudFoundry (start, stop, etc.).

Data

The following persistence options are included:

- Relational DB (MySQL, PostgreSQL, SQL Server, . . .)
- NoSQL (MongoDB, CouchDB, etc.)

Elasticity

Stackato can automatically scale applications based on a set of rules when the number of requests increases or decreases, as long as it maintains its resources into the established limits.

8.1.4 Apprenda

Site: http://www.apprenda.com
Overview

Apprenda offers a commercial private PaaS platform, along with a free public PaaS whose intention is to let developers try the software out without the hassles of installing and running a private instance. Unlike other offerings, such as CloudFoundry, they do not intend to get into the public PaaS market with it, it is offered just as a test environment.

Apprenda offers support for .NET technologies running on Windows environments and claim a “deploy-anywhere PaaS platform”. In case of having a Windows server private cloud and an Azure public cloud account, hybrid cloud deployments are enabled in an easy way by allowing to spread applications’ components between different servers (both private and public).

One of the strengths of Apprenda is that it provides support for creating multi-tenant application, including billing and usage metering, thus enabling the creation of SaaS applications. Apprenda makes it easy to create an application in which different plans can be configured, each plan corresponding to a limited set of features and with a possible pricing associated. Then, user subscriptions can be managed to let users make use of an application with one of these plans.

In general, Apprenda offers a larger set of features for development and deployment than other platforms, but lacks the ability to handle multiple languages or DBMSs.

Products

Apprenda offers the following products:

**ApprendaCloud.com** Public PaaS, owned by Apprenda, running their engine. The main goal of this public instance is to let developers try the software out without having to download, install and run a private instance.

**Apprenda** This is the main product offered by Apprenda and is the platform itself.

Installing and running

In order to install Apprenda the following minimum requirements have to be met:

- Windows Server 2003+
- SQL Server 2005 SE+
- IIS 7+
- .NET 4 Runtime Environment
- 2-Core processor
- 2 GB RAM
- 40 GB disk
Development

Development for Apprenda is done entirely in .NET based technologies. Typically, apps developed with Apprenda follow a SOA architecture, containing a set of UIs, a set of web services (Windows Communication Foundation services) and a set of DBs. It offers the following tools:

- CLI client
- Visual Studio based SDK
- Web interface

The three tools allow creating bare applications from templates configured to run on Apprenda, package an application to be deployed on Apprenda and mimic a live Apprenda environment locally. Apprenda exposes a set of APIs (services) to be used by applications, such as the ability to create inflated log messages with contextual information, user and user roles management and a usage metering and billing system.

Deployment

Deployment on Apprenda is usually done through the web interface by uploading a Deployment Archive containing the .NET application. Several versions of the same application can be deployed at the same time, allowing to have testing, development and production environments simultaneously. Nevertheless, deployment can also be done by using the other two tools: the Visual Studio SDK and the CLI.

Apprenda provides a set of tools to help ensure a high availability rate, such as support for database failover and support for starting new instances of an application when it fails, along with exhaustive logging functions.

When deploying multi-tenant applications, the type of data model to use can be chosen (it could be chosen to have, for instance, one isolated DB for each end user) along with the UI model (for instance, a separate UI for each end user). Furthermore, resource policies are assigned to the different components of an application. A resource policy defines the maximum resources (CPU, memory) to use for that component.

Monitoring and Governance

Apprenda does not offer extensive monitoring tools natively. However, it does provide the capability to monitor usage of End Users in SaaS applications in order to get accurate billing information.

It can integrate with Microsoft’s System Center Operations Manager (SCOM), providing access to Platform infrastructure details and application Workloads details. This allows the more extensive monitoring of Platform health and provides diagnostics, recoveries and tasks to resolve common issues.

Data

Apprenda only offers SQL Server for persistence storage.
Elasticity

Apprenda does not offer auto-scaling capabilities. However, it provides an API to get workload information, thus enabling developers to create responses to changing conditions. Manual scaling can be done by selecting how many instances will run a particular component of an app and where these instances will be physically allocated.

8.1.5 CumuLogic

Site: http://www.cumulologic.com/

Overview

CumuLogic is a commercial private Java PaaS platform that can be deployed on either private, public or hybrid IaaS clouds. Currently it can integrate with VMWare vSphere, Citrix CloudPlatform, Eucalyptus and OpenStack for private clouds and Amazon EC2 and HP Cloud Services for public clouds.

One of the strengths of CumuLogic is its multi-tenant platform architecture. As they state: “The metering feature allows you to track application usage so you can properly charge your organizational departments. Unlike any other cloud platforms, CumuLogic PaaS calculates the resource utilization by each application and provides an application cost metric making it easy to understand the actual cost of running an application”. Also they allow to set different levels of isolation for applications allowing, for instance, to share database resources among several applications while maintaining application cores isolated.

CumuLogic offers a set of tools to ensure high availability rates by continuously monitoring health and availability of all infrastructure services and recovering an app when it crashes by assigning other healthy instances to it or recovering data from a back-up.

Products

CumuLogic offers the following products:

Public Cloud Offered jointly with HP Cloud Services, a public instance of the CumuLogic PaaS is offered with a price of $200/developer/month.

Private platform The private PaaS platform itself.

Installing and running

In order to install CumuLogic the following minimum requirements have to be met:

- CentOS, Fedora or Red Hat Linux OS
- 4-core processor
- 50GB+ free disk
- JDK 1.6+
Development

CumuLogic supports only Java (along with the Spring framework) and PHP. Additionally, within these languages, plug-ins can be implemented to support any framework.

CumuLogic offers the following tools:

- CLI client
- Eclipse plug-in
- Web interface

Both the CLI and the Eclipse plug-in offer support for adding new applications to the Application Repository, however they don’t provide support for creating the applications themselves. CumuLogic has a very limited offer on application services. It currently only offers basic database services, without support for backups.

Deployment

CumuLogic allows deploying several version of the same application in order to allow simultaneously production, testing and development environments.

When deploying an app, the deployment environment has to be selected or, alternatively, create a new one. An Environment consists primarily on a Resource Pool which provision Resources (server instances) to applications, and of a set of Application Services (databases).

Monitoring and Governance

Various performance metrics for provisioned (running) environments can be inspected. Metrics include: Availability, CPU Usage, Average Load, Used Memory and I/O network connections. Charts are generated for each metric, and can be further filtered and customized.

Usage reports can also be generated for applications, which show total running time, IaaS charges, total cost and status (Active/Inactive). Reports can be further decomposed into the concrete instances in which an application is running.

Basic governance of applications can be made using any of the provided tools.

Data

The following persistence options are included:

- Relational databases (MySQL, DB2)
- NoSQL (MongoDB)

One particularity of this provider with respect to the others is that databases can be shared across applications (of different cloud users).

Elasticity

CumuLogic offers an auto-scaling features for web applications in all of its tiers (front-end, application and database) using policies. The engine allows users to set policies on different IaaS clouds (public or private) in a generic way, thus making them platform-independent. Furthermore, notifications are sent to application owners when an app gets auto-scaled.
<table>
<thead>
<tr>
<th>PaaS</th>
<th>Target use</th>
<th>Programming Languages, Frameworks</th>
<th>Tools</th>
<th>Persistence Options</th>
<th>Autoscale Underlying Private IaaS</th>
<th>Monitoring Support</th>
<th>Auto Recovery</th>
<th>License</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Foundry</td>
<td>Web apps.</td>
<td>Java (Spring), Ruby (RoR and Sinatra), Node.js, Groovy (Grails), PHP, Python (Django)</td>
<td>CLI, Eclipse plug-in</td>
<td>RDBMS, NoSQL</td>
<td>No</td>
<td>Any</td>
<td>Using 3rd party service</td>
<td>No</td>
</tr>
<tr>
<td>Aneka</td>
<td>Standalone apps., HPC.</td>
<td>.NET</td>
<td>Standalone SDK</td>
<td>RDBMS</td>
<td>Yes</td>
<td>VMWare, Citrix Zen</td>
<td>Limited</td>
<td>Yes</td>
</tr>
<tr>
<td>Stackato</td>
<td>Web apps., standalone apps.</td>
<td>Java (Spring, JBoss, Grails, Lift), Python (Django), Perl, .NET, PHP, Ruby, Erlang, Scala, Clojure, Node.js</td>
<td>CLI, Eclipse plug-in, Komodo plug-in, Web interface</td>
<td>RDBMS, NoSQL</td>
<td>Yes</td>
<td>Any</td>
<td>Using 3rd party service</td>
<td>Yes</td>
</tr>
<tr>
<td>Apprenda</td>
<td>Web apps.</td>
<td>.NET</td>
<td>CLI, Visual Studio SDK, Web interface</td>
<td>RDBMS (SQL Server)</td>
<td>No</td>
<td>Any</td>
<td>Infrastructure, apps usage</td>
<td>Yes</td>
</tr>
<tr>
<td>CumuLogic</td>
<td>Web apps.</td>
<td>Java, PHP</td>
<td>CLI, Eclipse plug-in, Web interface</td>
<td>RDBMS, NoSQL</td>
<td>Yes</td>
<td>VMWare vSphere, Citrix Cloud-Platform, Eucalyptus, OpenStack</td>
<td>Infrastructure, apps usage</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 8.1: Comparative of the features offered by each of the analyzed PaaS providers
8.2 Private PaaS selection

The function of the private PaaS in this project is to provide a sort of test bed in which several hybrid deployment scenarios can be executed in order to test Cloud4SOA. The overall performance of the private cloud has to be acceptable and resemble, up to a certain point, a real production environment scenario. This way, its performance can be more accurately compared to the performance when using a public cloud.

The cloud will not be used in a real production environment and, therefore, it does not need to offer a full spectrum of capabilities (advanced user management, support for multi-tenant applications, auto scaling, etc.). It just needs to include enough capabilities to be able to integrate with Cloud4SOA. This includes the creation and deployment of small applications, basic governance operations, migration functions and monitoring functions. For this same reason, it is desirable that the platform has a monetary cost as low as possible. It needs to support at least Java applications and MySQL databases, since the test application that will be used for the hybrid deployment uses these technologies. The cloud, also, has to be able to run in the environment reserved for it which is formed by a set of machines with a UNIX based OS.

It is also desirable that the cloud is as scalable as possible, so that it could be extended at some point to fit the requirements of the project or to improve its overall performance. In this direction, it is preferred that the cloud has a modular structure that permits the distribution and replication of components.

In summary, the selected platform should fulfill the following requirements:

- Support for all the main Cloud4SOA use cases:
  - Match-making
  - Deployment
  - Governance
  - Monitoring
  - Migration

- Support for Java and MySQL.

- Low monetary cost, optimally free.

- It has to be able to run in a UNIX based environment.

- Acceptable performance levels.

- Platform scalability, permitting its extension.

Taking into account these requirements and the evaluated PaaS (see table 8.1), CloudFoundry is the platform that fits better: It offers support for practically all use cases of Cloud4SOA, including support for Java applications and MySQL databases, it is the only free platform, it runs in a Linux environment and it can achieve good performance and scalability levels depending on how it is deployed. Refer to section 9 for more details about the performance and scalability of Cloud Foundry.
Aneka has been discarded because it only supports .NET applications. Moreover, it focuses on HPC applications, a field which is out of the scope of Cloud4SOA. In a similar way, Apprenda has been discarded for only supporting .NET.

Stackato has been discarded because, in the context of this project, it does not provide any additional benefit from CloudFoundry, but it has a remarkably large monetary cost.

Finally, CumuLogic has been discarded because its use is restricted to a set of IaaS, and the availability of these cannot be ensured in the reserved environment, and also because it has a commercial license.
Chapter 9

CloudFoundry Installation

We have chosen the Open Source project from CloudFoundry (cloudfoundry.org) to install the platform on the Atos infrastructure, since this project offers a complete private CloudFoundry deployment. This section gives an overview of the CloudFoundry architecture and details the installation environment and procedure.

9.1 CloudFoundry architecture overview

Since the CloudFoundry project was released as an Open Source project, it has undertaken major changes in its architecture driven by the community supporting it, both in the addition of features (such as support for the Python and PHP programming languages) and in the refactoring and improvement of the architecture. As one of the CloudFoundry engineers explains in a blog post\(^1\), efforts have been focused on development Below the Water Line, meaning that they are not visible as features, but rather improve the inner architecture of the platform.

The components within the CloudFoundry architecture and their interactions are depicted in figure 9.1 and 9.2, showing the different components that form it and how they are accessed from the outside. The architecture has been designed so that different components can be deployed on different physical nodes by using a decoupled message bus.

The main component, and the main entry point to the platform, is the Cloud Controller. This component performs all required actions to handle incoming requests and orchestrates the other components in the architecture. This component is responsible for all state changes in the system by ensuring all dependencies are available and by binding applications to services and, in general, is responsible of all actions related to applications and services, such as create and scale an application or bind services to an application.

In the initial architecture, functionality regarding user management was embedded in the Cloud Controller; however, as a result of the refactoring process mentioned earlier, the User Account and Authentication component has been created, and has been assigned all responsibilities regarding user management. This component basically implements the OAuth2\(^2\) protocol to authenticate users. The protocol has been designed to provide a mechanism to authorize Clients (for example, a web application) to use resources in a Resource Server (for example,

\(^1\)The blog entry can be read at: http://blog.cloudfoundry.org/2012/04/27/cloud-foundry-roadmap-below-the-water-line/

the Cloud Controller, the UAA component itself or any other back-end application), on behalf of End-Users, by providing access tokens to Clients. A high-level work-flow of the authorization process is depicted in figure 9.3; in the Cloud Foundry case the UAA component acts
primarily as the *Authorization Server*. The protocol allows for a Single Sign On (SSO) mechanism by which *End-Users* only need to log-in one time in applications on the Cloud Foundry platform. Moreover, it provides centralized identity management which provides a unique safe place in which to authenticate and avoids malicious applications to gather user credentials. It also provides other features such as user account management (for instance, to get information about a user) and client application registration.

![Diagram of the authorization workflow](image-url)

**Figure 9.3: 3 High-level view of the authorization workflow.**

The *Stager* is another of the new components that has been detached from the Cloud Controller as a result of the refactoring. It is basically responsible for staging jobs (applications) for execution. This process includes creating a *droplet* (a package with all the application code plus its dependencies) for an application and storing it for later retrieval by de DEA. To create a droplet, the Stager gets the objects that comprise an application plus its dependencies, refactors its configuration files to point to the appropriate services and then packages all together.

The *Droplet Execution Agent (DEA)* is the component responsible for actually executing applications at the VM level. When the DEA receives a droplet to execute, it first checks that it is able to run it in its VM and then starts the droplet or rejects the petition if its configuration does not match the configuration required by the droplet. The DEA also monitors applications (CPU usage, memory usage, etc.) and reports changes in state of applications to the Health Manager.

The *Health Manager* monitors applications activity across all DEA containers and notifies the Cloud Controller on actions to take. To check for inconsistencies, the Health Manager periodically scans the Cloud Controller database and compares it with the actual state of the world. This way it can detect, for instance, the loss of an app or a whole DEA container.

The *Router* not only routes incoming requests to either the Cloud Controller, the UAA or a user app, but also listens for notifications of new or deleted apps from the DEA. The router also acts as a load balancer by evenly distributing requests when a component is replicated.

The platform also contains *Services* (data, messaging, caching, etc.). The Cloud Controller has a uniform API to provision services, so that an app can be bound to any service with the same commands. Moreover, services can be shared across multiple applications.
Finally, to connect all the pieces together, Cloud Foundry uses a **Message Bus (NATS Server)**. When a component starts, it subscribes to the “topics” of his interest on the message bus and publishes messages to it. This way, each component receives the messages that it needs to process.

As a side note, it is curious to mention that the entire CloudFoundry platform has been coded in Ruby, using frameworks and libraries such as Rails and EventMachines.

### 9.2 Installation environment description

#### 9.2.1 Installation environment resources

A CloudFoundry instance will be deployed in the data center of the Atos offices in Barcelona, with the purpose of integrating it with Cloud4SOA. The platform will be hosted in the *Albíroix* server, which has the following characteristics:

- **CPU**:
  - **Model**: Intel Xeon CPU 32 bits
  - **Speed**: 3.00 GHz
  - **Cores**: 1

- **RAM**: 8 GB

- **HDD capacity**: 542 GB

- **OS**: CentOS 6.3 (Linux based)

The Atos infrastructure department can provision a maximum of 4 Virtual Machines dedicated to the platform from this server due to the fact that this same server is shared with other projects. Each one of the provisioned VMs will have the following characteristics:

- **RAM**: 1 GB

- **HDD capacity**: 20 GB

- **OS**: Ubuntu 10.04.4 (Linux based)

#### 9.2.2 Cloud Foundry Deployment options

Cloud Foundry can be deployed in 3 different ways as shown in table 9.1

The first two types are done using a script that semi-automatically installs all the necessary prerequisites, clones the GIT repositories that contain the source code and installs and configures the components. The third type uses a completely different approach to deploy. It uses BOSH (see the BOSH section below), a system independent of CloudFoundry, to deploy the CloudFoundry instance according to a set of specifications and configurations.

The **Level 1** deployment mode consists on running all the CloudFoundry components in one single VM. While this is the most easy and quick way to get a running CloudFoundry instance it presents numerous drawbacks such as scalability and performance limitations and, thus, is
only adequate for testing CloudFoundry or for using it as the development environment. This approach only requires running the deployment script without any modification.

The next level of deployment consists on distributing the components between more than one VM. However, some of the components are required to run in the same host due to the limitations of the installation approach, specifically the Cloud Controller, Health Manager and the Service gateways must run in the same host. This type of deployment still has some limitations with respect to the third but they are milder than in the Level 1 approach. Scalability is improved with respect to it since new nodes can be added at any time without much effort and performance is also improved as the workload can be distributed across several nodes, and even several nodes of the same type. Therefore this type of deployment is addressed to small scale production environments, with at least 4 VMs dedicated to it.

Finally the Level 3 deployment is the appropriate approach to take in a large scale production environment, in which high levels of performance and availability are required. This is also the hardest and most complex way to deploy CloudFoundry since it requires first to deploy a distributed version of BOSH, which is not trivial, and then configuring and deploying the CloudFoundry instance through it. This method of deployment does not impose any limitations or constraints, any desired topology can be achieved. Apart from the clear benefits that this method has over the other two, it also allows for the easy maintenance of the cloud as explained in the BOSH section. To be able to properly run an installation of this sort, a considerably big pool of VMs is required, since some need to be dedicated to run BOSH and some others to run CloudFoundry itself; as a reference, an example scenario\(^3\) from the Cloud Foundry documentation recommends to have at least 6 VMs dedicated to BOSH and at least 40 VMs dedicated to CloudFoundry. Furthermore, a hypervisor (currently only vSphere) is required in the physical computer where BOSH runs.

One early conclusion to take from this is that, in any case, CloudFoundry is effectively platform agnostic, since it does not depend on any IaaS (vSphere is only required for BOSH, not for CloudFoundry), it just needs a number of VMs running.

We have chosen to perform a Level 2 deployment for this project. On one hand the Level 1 deployment is too simplistic and neither reflects a real scenario nor supposes any challenges. On the other hand, we consider it to be a better option than a Level 3 deployment for the following reasons:

\(^3\)https://github.com/cloudfoundry/oss-docs/blob/master/bosh/tutorial/end_to_end_on_vsphere.md

<table>
<thead>
<tr>
<th>Level</th>
<th>Deployment type</th>
<th>Purpose</th>
<th>Brief description</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>Single Host</td>
<td>Development</td>
<td>All CF components run in the same VM</td>
</tr>
<tr>
<td>L2</td>
<td>Multi-Host small scale</td>
<td>Small-scale production</td>
<td>CF components can be distributed and replicated across several VMs with some restrictions.</td>
</tr>
<tr>
<td>L3</td>
<td>Multi-Host large scale</td>
<td>Large-scale production</td>
<td>CF components can be freely distributed and replicated across any number of VMs.</td>
</tr>
</tbody>
</table>

Table 9.1: Description of the different installation options
• A Level 3 deployment is intended for remarkably large scale production environments (46+ VMs and at least 10GB+ of RAM only for BOSH), which is not required for the goals of the project, that is, to reproduce different scenarios with a hybrid cloud setup. Additionally, we don’t have a pool of resources big enough allocated to this project to properly reproduce such a deployment.

• This deployment approach would limit the number of VMs that are actually available for CloudFoundry, since a reasonable number of them would be occupied by BOSH. Given the amount of resources allocated for this project, this would deliver less performance than in a Level 2 deployment where all available VMs run CloudFoundry.

• BOSH requires the VMWare vSphere hypervisor, however, this is a very expensive commercial product intended for large-scale production environments and acquiring it would be out of scope of this project.

• A Level 2 deployment still fulfills the desired requirement for the private cloud, such as scalability, cost and good performance.

At the end of this section a description of BOSH is given for the sake of completeness

9.2.3 Deployment Design

As said in the previous section, we will deploy an instance of CloudFoundry following a Level 2 approach distributed across 4 VMs in a single physical server. For performance reasons, we will only install the MySQL service, since we are not going to deploy any application that will require any other service, furthermore additional services could be easily installed in the future. Figure 9.4 shows a high level view of the different VMs and how they are connected.

Figure 9.4: Deployment view of the VMs that will host CloudFoundry.

The content of each VM is depicted in table 9.2.

CloudFoundry architecture has been designed so that components are as decoupled as possible from each other, in fact, each component only depend on the NATS Server in order to
interact with other components. Therefore, as it can be seen in figure 9.4, VMs only depend on
the Director VM. While this design principle has some positive effects it also has some draw-
bcks. On one hand its advantages include increased robustness since if a component (other
than the NATS server) fails the platform can keep running, increased scalability since adding
any component only requires it pointing to the NATS server and increased modifiability of the
structure. On the other hand, performance is harmed because all actions must first pass through
the message bus and because the Cloud Controller can become a bottle neck, also the reliability
of the whole platform depends on the NATS server, if it fails, the whole platform fails. Among
all the VMs, only the Router VM requires to be publicly accessible, since it will be the entry
point of all incoming requests (API requests and application requests).

Note that services are split between two different components, its gateway and its node.
The gateway is just the agent that interacts with the service, while the node is the component
that stores the data (the DB in this case). By splitting the node into a separate VM we get the
benefits from the “Distributed Persistence” design pattern [CFB+02].

We have decided to split the DEA into its own VM because it is the component with the
biggest workload and, thus, it deserves its own space. Similarly, the router is the entry-point for
all requests and separating it into its own VM reduces its risk to becoming a bottleneck. Note
that there could be other deployment structures possible, for instance, it could make sense to
replicate the DEA node to increase performance, especially when there are several applications
running; or to replicate the NATS server in order to increase reliability.

### 9.2.4 BOSH

BOSH (BOSH Outer Shell)\(^4\) is a tool chain for release engineering, deployment and lifecycle
management of large scale distributed services. BOSH was originally developed in the context
of CloudFoundry, but the framework is general purpose and can be used to deploy other dis-
tributed services on top of one of the supported IaaS. In practice, BOSH can be used to deploy
a fully distributed instance of CloudFoundry that can be used in large-scale production envi-
ronments. Not only that, but BOSH can easily be used to maintain this instance by repairing it,
updating its components, etc.

\(^4\)https://github.com/cloudfoundry/bosh
Currently BOSH supports AWS and VMWare vSphere based IaaS natively, and a plugin for OpenStack is being currently developed by external contributors.

BOSH can be deployed both in its micro form and in its distributed form. Micro BOSH is a single VM that contains all the BOSH components and is only apt to be used in development environments. A distributed version of BOSH has to be used in order to manage distributed systems in a production environment. Actually, the distributed version of BOSH is deployed using a Micro BOSH instance.

Figure 9.5 depicts the high level architecture of BOSH. The core of the system is the BOSH Director who orchestrates all the components and controls the creation of VMs, deployment and other life cycle events of software and services. Each VM started by BOSH contains an Agent that is given a role within CloudFoundry and a set of jobs, and also listens for instructions from the BOSH director. The Director transfers required files (to run jobs) to Agents through the Blobstore and also stores some meta-data about each VM in a database. Users interact with BOSH through a Command Line Interface that sends commands to the Director.

The Director communicates with the underlying hypervisor using a vendor-specific CPI (Cloud Provider Interface) that conforms to a common interface defined in BOSH. This interface includes methods to manage steamcells (VM template with an embedded BOSH agent), to manage VMs, to configure networks, etc.

9.3 Installation and evaluation

9.3.1 Installation

As said in the previous section, a Level 2 deployment across 4 VMs will be made. In order to perform the installation, CloudFoundry provides a shell installation script that performs the following operations:

- Updates system packages to the last version.
- Installs some required tools: `wget`, `git`, `ruby`, `rubygems`, `chef`, `rake` and others.
- Clones the GIT repositories containing the whole CloudFoundry source code.
- Launches the `chef` tool which installs the necessary components as specified in the deployment configuration file.

Chef\(^5\) is an Open Source tool that can be used to configure servers by using a set of recipes. A recipe is basically a ruby program that defines how the server should be configured: IP, network configuration, etc. Having the CloudFoundry source code cloned from the GIT repositories, Chef sets up the server in order to make the components ready to use.

If the system needs to be updated in the future, it can be done by updating the GIT repositories and re-running the chef scripts to configure the components appropriately.

This installation script has to be executed in each of the 4 VMs, with different deployment configuration files. A configuration file defines which components need to be installed and where are located the components that they require, which in this case is only the message bus (NATS server). The following snippets show the content of the 4 configuration files. Note that

\(^5\)https://github.com/opscode/chef
the only installed service will be MySQL since this will be the only application service needed at this moment. More services could be easily installed later if needed.

**Director VM**

```yaml
deployment:
  name: "director"
jobs:
  install:
    - nats_server
    - cloud_controller:
      builtin_services:
        - mysql
```
Router VM

deployment:
   name: "router"
jobs:
   install:
      - router
   installed:
      - nats_server:
         host: "[Director VM IP]"
         port: "[port]"
         user: "[user]"
         password: "[password]"

MySQL VM

deployment:
   name: "mysql"
jobs:
   install:
      - mysql_node:
         Index: "0"
   installed:
      - nats_server:
         host: "[Director VM IP]"
         port: "[port]"
         user: "[user]"
         password: "[password]"

DEA VM

deployment:
   name: "dea"

jobs:
   install:
      - dea
   installed:
- nats_server:
  host: "[Director VM IP]"
  port: "[port]"
  user: "[user]"
  password: "[password]"

9.3.2 Installation issues

The installation process of CloudFoundry in the Atos server ran smoothly and completed successfully without issues. However, some problems arose when running and operating the platform. The platform can be accessed through the cf.cloud4soa.eu domain.

The first was a problem with the DNS configuration. It was not enough to point the cf.cloud4soa.eu domain to the router VM, we had to add an additional wildcard rule redirecting *.cf.cloud4soa.eu to this same VM in order to be able to access the API (api.cf.cloud4soa.eu), the UAA (uaa.cf.cloud4soa.eu) and the deployed apps (appname.cf.cloud4soa.eu).

Then, there was also an issue with the UAA configuration because its URL was not being registered properly. To solve this, we had to explicitly define the UAA URL and port (8080) in the IP of its host (the director VM) in its configuration file. Specifically, we had to add the following to the configuration file:

```yaml
uaa:
  uris:
    - uaa.cf.cloud4soa.eu
    - login.cf.cloud4soa.eu
  host: 192.168.252.6
  port: 8080
```

Finally, the runtimes configuration file was incorrectly pointing to executables in the Director VM, so we had to change all paths to point to the DEA VM instead, since this component is the responsible for running applications.

9.3.3 Configuration

After having the platform properly running, we changed some of its configuration in order to enhance its security and to adapt it to the Atos and Cloud4SOA needs. We performed the following tweaks:

- Create an admin user and add it in the list of admins in the Cloud Controller configuration file.
- Change the token of the MySQL service both in the Cloud Controller and in the MySQL gateway in order to make the communication between both securer.
- Change the default description of the platform in the Cloud Controller configuration file and the welcome message of the API by a custom message.
9.3.4 Validation

To validate that the platform has been properly installed, some actions have been executed to check that everything is working correctly. Figure 9.6a, 9.6b, 9.7a and 9.7b shows the output of the info, services, runtimes and frameworks commands respectively, which show some basic information about the installed platform. The info command, among other information, shows an introductory message (“Atos PaaS Cloud Platform”), the URL of the entry point to the platform (the Target attribute), the User currently logged in and the Usage that the user is making of the resources with respect to her limits. The services command shows that there is only one application service available (MySQL) and that there is database created with this service named “mysql-env”. The runtimes command shows all the programming languages available and their version and the frameworks command shows all the frameworks available for these languages.

![Figure 9.6: Results from the “vmc info” and “vmc services” commands](image)

Another check has been to upload a simple application. Figure 9.8 shows the output of the push command which deploys the application. The output of the command shows, after the answer to all questions, that all operations required to deploy and start the application have completed successfully (they are OK). We can see that a war archive has been deployed with the standard configuration (512Mb of RAM, 1 instance and no services) and is accessible in the sample_app.cf.cloud4soa.eu URL.

Finally, we checked that the application was really running and accessible. As seen in figure 9.9, accessing the URL of the application shows that the application is properly running.
Figure 9.7: Results from the “vmc frameworks” and “vmc runtimes” commands.

Figure 9.8: Result of the “vmc push” command.
Figure 9.9: Pushed application shown in a browser.
Chapter 10

Cloud4SOA Integration Analysis

Cloud4SOA defines a common interface (known as the harmonized API) to connect to PaaS platforms. The interface is the result of a detailed study of several PaaS capabilities and contains methods to perform the most common operations in a PaaS. Due to the fact that the interface has to embrace different PaaS, some of the method may not be applicable in some cases.

This section describes the methods in the harmonized API and how they can be mapped to the CloudFoundry API in order to integrate it with Cloud4SOA.

10.1 API Overview and integration Analysis

The CloudFoundry API is exposed as a RESTful web service. It can be accessed directly through HTTP requests or through a set of client libraries contributed by the community that encapsulate the necessary business logic to perform the most common operations (create an application, start, etc.). Libraries are written in several languages. This project is interested in the Java one (see Section 10.3 for further details).

The API provides two REST endpoints, the V1 endpoint and the V2 endpoint. The V1 endpoint corresponds to the first version of CloudFoundry that was released and is accessed through the URLs indicated below in the Integration Steps of each method, and the V2 corresponds to a new version that was released after making major changes to the platform (see section 9.1 for more details) and is usually accessed by appending “/v2” to it. Since Cloud4SOA may be used to manage either one, we have to consider both and dynamically determine the version of the underlying platform at the time of using its API. The Java client library includes a mechanism to make this distinction.

The Environment concept in the harmonized API maps to the Space concept of CloudFoundry, which is a new concept in the V2 architecture. However, this concept is still very new and it lacks a lot of functionality and support; currently a space is a simple concept defined as an isolated set of applications and services.

The following sections describe each method in the harmonized API and how it can be mapped to the CloudFoundry API.

10.1.1 Authentication

In order to use most of the methods, the client must authenticate first. Authentication with CloudFoundry is done by issuing a token to the client. However, the way tokens are issued
differs depending on whether there is a UAA component or not.

**Old mechanism (without UAA)**

In this case the Cloud Controller itself exposes a basic authentication mechanism that consists on issuing tokens pregenerated at the time of registering a new user.

The pregenerated token for a user can be obtained by sending a POST request to /users/{email}/tokens, specifying the email of the user and the password on the request body.

**New mechanism (with UAA)**

In the new architecture the OAuth2 protocol is used, through the UAA component, to authenticate clients acting on behalf of users (see section 9.1 for details), by obtaining an access token. Normally this would be a 2-step process: first the user grants access to the client and then the client requests the access token. However, the UAA also provides a method to perform both steps in the same request.

Once the access token has been issued, the client can access the API by placing the token in the Authorization HTTP header. Access can be granted and the token obtained by sending a POST request to /oauth/authorize in the authorization endpoint (the UAA component).

**10.1.2 Monitoring**

Monitoring currently is done through the doMonitoring and doDetailedMonitoring methods. The Monitoring component in the Governance layer polls this methods at regular intervals and stores the results in the database. The description of these methods can be found in the following sections.

**10.1.3 Development Life-cycle supportive methods**

**Create an application**

**Signature** createApplication(in name, in description, in environment) :: URI

**Description** Creates an empty application with the given name and description and within the specified environment (in case of V2). If the application cannot be created then it returns NULL, otherwise it returns the new application URI.

**Integration** The CloudFoundry API can create an empty application within a given environment (space) with a set of URIs assigned and a set of bound services. When creating an application, it can be specified how many instances to run, how much memory to assign, etc.

**Integration Complexity** Low

**Integration** POST to /apps with app information, or use createApplication method of Java client library.
Delete an application

**Signature**  deleteApplication(in name) :: Boolean

**Description**  Attempts to delete an application. It returns false if the application does not exist or it cannot be deleted, and true otherwise.

**Integration**  CloudFoundry can delete an application given its name or its id. It returns an error if the application cannot be found or cannot be deleted. After deleting an application, its URL is discarded, but it can be reused later by another application.

**Integration Complexity**  Low

**Integration**  DELETE to /apps/{appName}, or use deleteApplication method of Java client library.

Update an application

**Signature**  updateApplication(in name, in description, in environment) :: URI

**Description**  Updates the elements of an application. If the given application cannot be updated or is not found it returns NULL, otherwise it returns the application URI. This method updates elements that are not related to the application code, such as its name, bound services, etc. The attributes that can be updated depend on each PaaS provider.

**Integration**  CloudFoundry can update information about an application. Among others, its name, URIs, instances, memory and services can be updated. The method returns an error if part of the configuration is invalid.

**Integration Complexity**  Low

**Integration**  PUT /apps/{appName} with app information, or use updateApplication* methods (updateApplicationMemory, updateApplicationInstances, etc.) of Java client library.

Create an application version

**Signature**  createAppVersion(in appVersionInfo, in appName) :: String

**Description**  The documentation on this method is confusing and unclear. Moreover, the method has been deprecated in the last versions of the API. Therefore, it will not be implemented in the CloudFoundry Adapter.

Delete an application version

**Signature**  deleteAppVersion(in appName, in version) :: Boolean
**Description**  Deletes the given version of the application. If the application or the version cannot be found then false is returned, otherwise true is returned.

This method is not applicable to CloudFoundry because it does not include the Version concept.

**Deploy**

**Signature**  
```
deploy(in name, in version) :: String
```

**Description**  Deploys the code of an application version previously created. If the application or version cannot be found, then it returns NULL, otherwise its URL is returned. Platforms not having a version concept ignore this parameter.

**Integration**  The API exposes a method to upload an application binary or archive. If a prior version of the application already exists it will be replaced by this one.

**Integration Complexity**  Medium

**Integration**  
```
PUT /apps/{appName}/application with the archive or use uploadApplication from the Java client library.
```

**Undeploy**

**Signature**  
```
undeploy(in appName, in version)
```

**Description**  Undeploys the given application version. In providers not supporting VCS (Version Control System), the application version will not be applicable, but the application will be undeployed anyway.

CloudFoundry does not support the undeploy operation.

**Deploy an application to an environment**

**Signature**  
```
deployToEnv(in name, in version, in env) :: URL
```

**Description**  Deploy the given application version into the given environment. If neither object can be located then return NULL, otherwise return the application version URL. If the provider does not support multiple environments then this operation is not applicable. In providers not supporting VCS, the application version will be ignored.

An Environment is a container in which all required software and resources for an application to run are available, this includes all application services that the application use, environment variables, files, etc.

**Integration**  CloudFoundry does not support this operation natively, it only support deploying an app in the space to which it pertains. The method can be implemented by first updating the application with the new space and then deploying through the deploy method.

Note that this operation is only available in V2.
Integration Complexity   High

Integration the update can be done by sending PUT to v2/apps/{appId}, and deployed through the deploy method.

Upload an application and deploy it to an environment

Signature   uploadAndDeployToEnv(in name, in version, in env) :: URL

Description Creates a new application (if needed), then a new application version, uploads the code and deploys it to the given environment. The method executes the following method chain: createApp -> createEnvironment -> attachEnvironment -> deployToEnv. However, this chain can vary depending on the provider and whether it supports versions and environments. The method returns NULL if something goes wrong and the URL of the deployed application otherwise.

Integration In CloudFoundry this method translates to the following chain: createApplication (if it does not exist) or updateApplication to update its space (in case of V2), and then deploy.

Integration Complexity   High

Integration see createApplication, updateApplication and deploy.

10.1.4 Monitoring methods

Get the status of an application

Signature   getAppStatus(in name, in version) :: AppStatus

Description Get the status of the given application version. If the application or the version cannot be found then it returns NULL, otherwise it returns one of the following values: {CREATED, DEPLOYED}. In providers not supporting VCS, the application version will be ignored.

Integration CloudFoundry completely decouples an application from its code, in the sense that either can be freely operated without considering the other. This permits, for instance, to start or scale an application that has not been deployed yet. This decoupling makes it fairly hard to get the status of an application, as required by this operation, since this piece of information cannot be directly obtained from the application attributes.

A possible workaround is to try to fetch the files of an application and check whether they can be retrieved properly (application has been deployed) or an error is thrown (application has not been deployed). This way we can distinguish between “CREATED” applications and “DEPLOYED” ones. It is important to note that, on some cases, this workaround may not work properly, since the API method exposed to get files only returns a generic error message, we have to assume that this error will always be caused by the inexistence of the resources.
Integration Complexity  High

Integration  /apps/{name}/instances/0/files/app and determine whether the call is successful or not.

Get the running status of an application

Signature  getRunningStatus(in appName, in version) :: InstanceStatus

Description  Get the status of the running instance that corresponds to the deployed application version. If the application or the version cannot be found then it returns NULL, otherwise it returns one of {UNRESPONSIVE, RUNNING, STOPPING, STOPPED}

Integration  The CloudFoundry API exposes a method to get info about an application which contains, among other things, its status. However it only distinguishes between STOPPED (which maps to STOPPING and STOPPED) and STARTED (which maps to RUNNING).

Integration Complexity  Low

Integration  GET /apps/{appName} and look for state attribute or use the getApplication method from the Java client library.

Get summary statistics about an application

Signature  doDetailedMonitoring()

Description  Get statistics from all applications. Current metrics are: latency (response time) and application status.

Integration  This method is, actually, a dummy method that simply returns a success message; it does not need to communicate to the platform. Calculations about latency and status are made at the Governance layer.

Integration Complexity  Low

Integration  None

10.1.5  Instance Handling Methods

Get versions information

Signature  getAppVersions(in name) :: List of VersionedApplication
**Description**  Returns a list of application versions found in the platform for a specific application. In providers that do not support VCS this method is not applicable. If the given application does not exist then it returns a NULL object, otherwise a list of applications is returned.

This method is not applicable to CloudFoundry because it does not support Versions.

**Get environments information**

**Signature**  getAppEnvironments(in name) :: list of Environment

**Description**  Get all environment information about all versions of the given application. If the given application does not exist, then it returns a NULL object, otherwise it returns a list of Environments. This method is not applicable in providers that do not support multiple versions.

This method is not applicable to CloudFoundry because it does not support Versions.

**Get application details**

**Signature**  getApplicationDetails(in name, in env) :: Application

**Description**  Get the details of an application. If the given application cannot be found, then it returns NULL. Otherwise it returns all data about the application.

**Integration**  The API has methods to return information about an application

**Integration Complexity**  Low

**Integration**  GET to /apps/{name} returns info about an application. The getApplication method from the Java library can also be used.

**Start or Stop**

**Signature**  startOrStop(in name, in version, in op)

**Description**  Starts or stops an instance of the specified application version. In platforms not supporting VCS, the application version is ignored.

**Integration**  The CloudFoundry API exposes start and stop methods that do exactly that.

**Integration Complexity**  Low

**Integration**  PUT to /apps/{appName} with a state variable set to STARTED/STOPPED in the request body or use the startApplication or stopApplication method in the Java client library.
10.1.6 Resource Handling Methods

Check Availability of an application

**Signature**  
checkAppAvailability(in name) :: Boolean

**Description**  
Checks whether the specified application name is available. This method is especially critical for providers that share a common namespace across users such as CloudFoundry does.

**Integration**  
The API does not directly expose a method to check the availability of a name. However, it can be tried to get an application by the given name and see whether it properly retrieves it or it does not find it.

**Integration Complexity**  
Medium

**Integration**  
GET /apps/{name} with the name of the app as a parameter or use the getAppId method of the Java client library.

List all applications

**Signature**  
listApplications() :: list of Application

**Description**  
Returns a list of all applications registered for a specific user. Information returned may vary between PaaS providers.

**Integration**  
The API exposes a method to get information about all applications, so a simple call to this method is required to get the result.

**Integration Complexity**  
Low

**Integration**  
GET /apps or the getApplications method of the Java Client library

Create an environment

**Signature**  
createEnvironment(in env)

**Description**  
Creates a new environment with the information provided. Each platform models environments in a different way; therefore the internals of an Environment depend on the underlying platform.

**Integration**  
The V1 endpoint does not support the environment concept, so the operation is not applicable in this case.

The V2 endpoint exposes a method to create a space (environment), although the Java client library does not have any method to do so.
**Integration Complexity**  Medium

**Integration**  POST to /v2/spaces with the space information

**Delete an environment**

**Signature**  deleteEnvironment(in env) :: boolean

**Description**  Deletes the environment that corresponds to the given name. If no environment can be found, or the environment cannot be deleted, then it returns false. Otherwise, it returns true.

**Integration**  As with the `createEnvironment` operation, this method is not applicable in the V1 endpoint while the V2 endpoint exposes a method to delete a space.

**Integration Complexity**  Medium

**Integration**  DELETE to /v2/spaces/{spaceId} with the id of the space.

**Attach environment to application**

**Signature**  attachEnvironment(in appName, in version, in envName) :: boolean

**Description**  Associates the given application version with the given environment. If the application, version or environment cannot be found or the association fails, then it returns false. Otherwise, it returns true. In platforms not supporting multiple environments this operation is not applicable and in platforms not supporting multiple versions the version argument is ignored.

**Integration**  The API does not explicitly expose a method to perform this operation, however it can be achieved by updating an application with the new space.

**Integration Complexity**  Medium

**Integration**  See updateApplication for details

**Update an environment**

**Signature**  updateEnvironment(in env)

**Description**  Updates the details of an environment.

**Integration**  As with the rest of the methods to manage environments, this operation can only be implemented through the V2 endpoint.
Integration Complexity  Medium

Integration  PUT to /v2/spaces/{spaceId} with the id of the space and the new information in the request body.

Create a new Database

Signature  createDB(in database, in appName) :: Database

Description  Creates an empty database bound to the given application. If the given database cannot be created, then it returns NULL. Otherwise, the created database details are returned.

Integration  The API exposes a method to create a service and another to bind a service to an application.

Integration Complexity  High

Integration  POST to /services with the service information (in V1) or POST to /v2/service_instances (in V2), and PUT to /apps/{appNam} to bind the service (in V1) or POST to /v2/service_bindings (in V2). Alternatively use the createService and updateApplicationServices methods from the Java client library.

Get all databases

Signature  getDBList() :: list of Database

Description  Returns information about all databases of a user. Information returned this way may vary depending on the underlying platform.

Integration  The API exposes a method to get all application services of a user. The operation can be implemented calling this method and selecting only database services.

Integration Complexity  Medium

Integration  GET /services (for V1) or GET /v2/service_instances (for V2). Alternatively, call the getServices method of the Java client library.

Delete a Database

Signature  deleteDB(in dbName)

Description  Deletes a database.

Integration  The API exposes a method to delete an application service.
Integration Complexity  Low

Integration  DELETE /services/{service} (for V1) or DELETE /v2/service_instances/{service} (for V2)

10.1.7 Migration Handling Methods

Export a version of an application

Signature  exportAppVersion(in appName, in version) :: ExportObject

Description  Exports an application plus any associated external data, as defined in its associated environment. Possible data export types include database dumps, file system copies or raw data exports. For instance, in Java EE applications, the object being returned is an .ear or .war file along with a potential escorting database dump. In platforms not supporting versions the version attribute is ignored.

Integration  The API provides a method to download an application. The type of downloaded archive will depend on the type of application (.zip, .war, etc.)

Integration Complexity  Medium

Integration  GET /apps/{appName}/application will download the application archive.

Download a Database

Signature  downloadDB (in database, in path)

Description  Saves the database data and schema to a specified path.

Integration  According to the design principles of CloudFoundry, the platform does not make database services accessible through the internet, mainly due to security concerns. In order to be able to access these services remotely, CloudFoundry has designed a mechanism to create a tunnel to one of these services using an application called Caldecott. This application has first to be deployed and started in the CloudFoundry instance and then a tunnel to the server containing the CloudFoundry instance can be created. Then, while the tunnel remains open, the service can be accessed through localhost. This way, clients such as mysql or mysqldump can be ran locally to communicate to the remote database.

Integration Complexity  High
**Integration**  Operations from the Caldecott Java client provided by Cloud Foundry can be used. Alternatively the methods from the API can be used directly. The following steps are required:

1. **Check existence of Caldecott app:** Use getTunnelAppInfo from the library or the same API methods as in getApplicationInfo.

2. **Deploy Caldecott app if non-existent:** Use deployTunnelApp from the library or the same methods as in: createApplication, uploadApplication and startApplication.

3. **Bind DB service to Caldecott app:** Use bindServiceToTunnelApp from the library or the same method as in createDB.

4. **Get tunnel info:** Use getTunnelServiceInfo from library or GET to \(\text{caldecottURL}/\text{services}\).

5. **Open Tunnel:** Use openTunnel from library or POST to \(\text{caldecottURL}/\text{tunnels}\).

6. **Execute commands locally to dump database to a file.**

7. **Close Tunnel:** Use closeTunnel from library or DELETE to \(\text{caldecottURL}/\text{tunnels/\{path\}}\)

---

**Restore a Database**

**Signature**  restoreDB (in database, in source)

**Description**  Restores data and schema to a database from a specified source.

**Integration**  The process to follow is the same as in the previous operation. The only difference is in the commands that are executed locally, which will run a SQL script to generate and populate the database.
10.2 Integration Complexity Summary

Table 10.1 shows a summary of the integration complexity of each CloudFoundry method in the harmonized API.

<table>
<thead>
<tr>
<th>Method</th>
<th>Integration Complexity</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Development Life-Cycle Supportive Methods</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>createApplication</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>deleteApplication</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>updateApplication</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>createAppVersion</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>deleteAppVersion</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>deploy</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>undeploy</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>deployToEnv</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>uploadAndDeployToEnv</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td><strong>Monitoring Methods</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAppStatus</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>getAppStatistics</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>getRunningStatus</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>getSummaryStatistics</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td><strong>Instance Handling Methods</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>getAppVersions</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>getAppEnvironments</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>getApplicationDetails</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>startOrStop</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Integration Complexity</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resource Handling Methods</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>checkAppAvailability</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>listApplications</td>
<td>Low</td>
<td></td>
</tr>
<tr>
<td>createEnvironment</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>deleteEnvironment</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>attachEnvironment</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>updateEnvironment</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>createDB</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>getDBList</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>deleteDB</td>
<td>Low</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Integration Complexity</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Migration Handling Methods</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>exportAppVersion</td>
<td>Medium</td>
<td></td>
</tr>
<tr>
<td>downloadDB</td>
<td>High</td>
<td></td>
</tr>
<tr>
<td>restoreDB</td>
<td>High</td>
<td></td>
</tr>
</tbody>
</table>

Table 10.1: Integration complexity of the methods in the harmonized API

10.3 CloudFoundry Java library

After the release of Cloud Foundry as an OSS project, the community has contributed to it by developing a set of client libraries that can be used to access the Cloud Foundry API through its REST interface. The libraries can be used to more easily perform the most common operations (such as create application, start and deploy) by encapsulating all the business logic necessary to process required data and make the appropriate requests.

We have carefully examined the Java library and determined that it fits well in the project since most of the methods in the harmonized API can be implemented through the library. Using the library as a base, moreover, provides a reliable and tested component that has been developed, reviewed and improved by a whole community of developers, including developers from CloudFoundry itself.

Taking the library as a starting point we will extend and modify it according to our needs, so that all methods from the harmonized API can be properly implemented. We even consider the option of contributing with these changes (or a subset of them) to the OS project, so that
they can help to any developer interested in working with CloudFoundry. This idea will further enforce us to write code that complies to a set of rules and best practices, specifically we will try to:

- Minimize changes.
- Ensure that most of the changes are general enough and relevant to the community.
- Ensure that the code is at the level of quality of the rest of the library.
- Ensure that the changes comply with the library’s coding standards.
Chapter 11

Adapter Design and Implementation

11.1 Requirements Overview

The first phase of the Cloud4SOA project consisted on the gathering and elicitation of the platform requirements, both functional and non-functional. The following is an overview of the methodology that was followed along with the subset of requirements that concern the design of the adapters and a specialization of this high-level list of requirements into a new list of finer-grained requirements specific to this project.

The intention of this section is just to serve as an overview of all the work done in this matter. The full documentation of the work can be found in the project site1 as it has been made publicly available.

11.1.1 Methodology

Figure 11.1 shows the process that was followed in order to gather and elicit the requirements. The first step was to, in parallel, review the current State of the Art and determine the primary stakeholders’ needs. From this analysis, a comprehensive list of requirements was obtained and later prioritized in order to obtain the final list of requirements to cover.

![Method followed during the requirements elicitation phase.](image)

The goal of the State of the Art was to identify gaps, deficiencies, needs and problems in the research fields most closely related to the Cloud4SOA scope such as: cloud computing architectures, focusing on semantic interoperability issues; and resource and data models.

---

1[www.cloud4soa.eu](http://www.cloud4soa.eu)
The review started by searching for relevant papers in the major research databases of computer science (ACM Digital Library, IEEE Xplore, SpringerLink, etc.), by using a set of keywords representing the covered research fields. From the papers obtained, references were checked and additional papers were found from sources not covered by the initial search, such as blogs or electronic journals. Finally, initiatives coming from standardization bodies, leading vendors and funded projects were also included in the study. From all the papers found, a number of them were selected as the most relevant and were analyzed in depth in order to obtain a list of requirements derived from gaps, needs and common features found on the selected works.

In parallel to this activity, some of the consortium commercial partners acted as stakeholders in the development of two usage scenarios. The scenarios focused on the functionalities that the Cloud4SOA platform should offer to address specific requirements of these partners, as well as on the expected benefits of the Cloud4SOA solution.

Once these two parallel steps were completed, a set of functional and non-functional requirements were identified for each of the activities. Finally, these requirements were prioritized according to the following criteria:

- Top priority was given to requirements coming from both activities
- Medium priority was given to requirements coming only from the usage scenarios
- Low priority was given to requirements coming only from the State of the Art

Later it was decided to address only the top and medium priority requirements.

### 11.1.2 Functional Requirements

Two primary stakeholders were identified:

**Cloud-based application developer** This may be an enterprise or an ISV (Independent Software Vendor) that wants to develop an application and host it on the cloud.

**Cloud PaaS provider** This is any PaaS offering that wants to be as available as possible to customers.

Table 11.1 shows a list of the requirements relevant to the design of the adapter, classified by stakeholder and priority. Low priority requirements are omitted since it was decided not to address them. As the table clearly depicts, the adapters are primarily address to developers, since the use cases related to them are the ones that need to communicate to the different underlying platforms.

Taking this list as a starting point, we have refined the requirements into a new list of finer-grained requirements specific to this project.

The new requirements have been elicited using a template inspired in the requirement templates of the Volere methodology [RR12] but with a simplified set of fields adjusted to the needs of this project. Table 11.2 describes this template.
<table>
<thead>
<tr>
<th>Stakholder</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud-based application developer</td>
<td>F1 Monitor, manage and configure a Cloud-based application’s lifecycle (deployment, federation and migration)</td>
<td>Top</td>
</tr>
<tr>
<td></td>
<td>F2 Be able to migrate an application from one Cloud4SOA-enabled platform to another</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F3 Receive notifications when a SLA violation is raised</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F4 Establish agreements with Cloud4SOA enabled providers (when registered)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F5 Deploy (part of) an application on a Cloud4SOA enabled platform</td>
<td></td>
</tr>
<tr>
<td></td>
<td>F6 Acquire additional resources upon request</td>
<td></td>
</tr>
<tr>
<td>Cloud PaaS provider</td>
<td>F7 Map their platform’s API to the Cloud4SOA API</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 11.1: Cloud4SOA Functional requirements

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>Requirement identifier</th>
<th>Originator</th>
<th>High-level requirement that originates this requirement</th>
<th>Priority</th>
<th>Requirement priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fit Criterion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11.2: Requirements template inspired in the Volere template.
The following subsections describe the new requirements for each stakeholder.

**Developer**

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>CD1</th>
<th>Originator</th>
<th>F1</th>
<th>Priority</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The system must enable the deployment of applications to CloudFoundry by following the same process than with any other PaaS.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td>To be able to add CloudFoundry to the repository of available PaaS.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FitCriterion</td>
<td>The system must be capable of deploying a complex Java Web application (using a database) packaged as a war file to CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>CD2</th>
<th>Originator</th>
<th>F1</th>
<th>Priority</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The system must enable the undeployment of applications running on CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td>To be able to provide the required functionalities to fully integrate CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FitCriterion</td>
<td>The system must be capable of undeploying any application previously deployed to CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>CD3</th>
<th>Originator</th>
<th>F1</th>
<th>Priority</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The system must enable the governance (start/stop) of applications running on CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td>To be able to provide the required functionalities to fully integrate CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FitCriterion</td>
<td>The system must be capable of starting and stopping any application previously deployed to CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Req. Id</td>
<td>CD4</td>
<td>Originator</td>
<td>F1</td>
<td>Priority</td>
<td>Top</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------------</td>
<td>-----</td>
<td>----------</td>
<td>-----</td>
</tr>
<tr>
<td>Description</td>
<td>The system must enable the monitoring of applications running on CloudFoundry. The monitoring must be able to collect the same metrics collected for other PaaS.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td>To be able to provide the required functionalities to fully integrate CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fit Criterion</td>
<td>The system must be capable of showing monitoring data for any application deployed to CloudFoundry.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>CD5</th>
<th>Originator</th>
<th>F2</th>
<th>Priority</th>
<th>Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The system must enable the migration of an application from CloudFoundry to any compatible PaaS and vice-versa. The application and its environment must not be altered or damaged during this migration.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td>To be able to integrate Cloud Foundry in the migration feature of Cloud4SOA which is one of the key features of the platform.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fit Criterion</td>
<td>The system must be capable of migrating a complex Java Web application (using a database) packaged as a war file to any compatible PaaS.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>CD6</th>
<th>Originator</th>
<th>F4</th>
<th>Priority</th>
<th>Medium</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>The system must automatically create an agreement between the developer and CloudFoundry when an application is deployed to it. The details of the agreement can be defined by the developer at the time of creating the application.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rationale</td>
<td>To be able to have an agreement between developers and the PaaS to which they deploy applications, thus enabling Cloud4SOA to have a control over the performance of the cloud and to warn the developer about any unexpected situations.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fit Criterion</td>
<td>The system must create an implicit agreement with CloudFoundry when deploying any application to it. The agreement should contain the terms specified by the developer when creating the application.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PaaS Provider
The interface of CloudFoundry must be mapped to the harmonized Cloud4SOA API. The system must implement as much methods as possible from this interface and state clearly the methods that are not applicable.

**Rationale**

To take the most profit of the CloudFoundry API and implement as much operations as possible.

**Fit Criterion**

All methods from the harmonized Cloud4SOA API must be mapped to a method or set of methods of the Cloud Foundry API, or stated that is not applicable to CloudFoundry.

### 11.1.3 Non-functional requirements

Non-functional requirements were identified and specified using the ISO-9126 standard [ISO01]. Table 11.3 shows the final list of non-functional requirements relevant to the design of the adapter.

<table>
<thead>
<tr>
<th>Requirement type</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Security)</td>
<td>NF1 Use authentication and single sign-on, authorization and role-based access control</td>
<td>Top</td>
</tr>
<tr>
<td></td>
<td>NF2 Licensing and security issues to span different Cloud platforms</td>
<td>Medium</td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Interoperability)</td>
<td>NF3 Use common service descriptions (common terminology e.g. semantics, abstracting provider’s heterogeneity)</td>
<td>Top</td>
</tr>
<tr>
<td></td>
<td>NF4 Supporting commonly used standards, standards syntax, open APIs, widely available tools, technologies, methodologies and best practices</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NF5 Use a common API (common set of tools) that supports provisioning/deployment/configuration and control across different Cloud resources</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NF6 Semantic interoperability between Cloud4SOA-enabled platforms</td>
<td>Medium</td>
</tr>
</tbody>
</table>

Table 11.3: Cloud4SOA Non-Functional requirements

As with functional requirements, we have specialized this high-level list into a new list of finer-grained requirements specific to this project, as depicted in table 11.4.

Not surprisingly, the majority of relevant non-functional requirements are categorized as interoperability requirements. The design of the adapter will have to ensure the fulfillment of all the stated requirements.
Table 11.4: Specialized Cloud4SOA non-functional requirements for this project

<table>
<thead>
<tr>
<th>Requirement type</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality (Security)</td>
<td>NF1.1 Use the authentication mechanisms provided by Cloud Foundry on any request made to it.</td>
<td>Top</td>
</tr>
<tr>
<td>Functionality (Interoperability)</td>
<td>NF3.1 Use an abstraction of the concepts managed by Cloud Foundry that is compliant with the generic terms used in Cloud4SOA.</td>
<td>Top</td>
</tr>
<tr>
<td></td>
<td>NF5.1 Use the common API generic to all PaaS providers to interact with Cloud Foundry</td>
<td></td>
</tr>
</tbody>
</table>

11.2 Adapter Design

11.2.1 Cloud4SOA Reference Architecture Overview

In order to understand where adapters are placed within the Cloud4SOA architecture, it is worth giving an overview of it. The architecture aims at resolving semantic incompatibilities that are raised between heterogeneous PaaS platforms. To this end, it utilizes the knowledge gathered during the State of the Art activity to capitalize on existing trends. The architecture is comprised by 3 horizontal layers (following a 3-layer architecture) and 2 transversal vertical layers (see figure 11.2):

- The **Front-end layer** supports the user-centric focus of Cloud4SOA and the easy access of Cloud developers to the Cloud4SOA functionalities.

- The **SOA layer** implements the core functionality offered by the Cloud4SOA system, such as service discovery, match-making and deployment.

- The **Distributed Repository** puts in place the technical infrastructure, e.g., distributed registries on top of which the Cloud4SOA system is built.

- The **Semantic layer** puts in place the semantic models and tools offered by Cloud4SOA

- The **Governance layer** offers a toolkit to monitor and manage Cloud4SOA services and applications.

The main layer of interest in this project is the Governance one since, among other things, it contains the endpoint that communicates directly to the adapters through the harmonized API, both for application life-cycle management and for monitoring purposes. A component-level view of the layer is depicted in figure 11.3.

The two components of interest within this layer are the **Monitoring Module** and the **Execution Management Service Module** (EMS). The Monitoring module provides the means to create and run monitoring jobs in order to collect monitoring data of the applications managed by Cloud4SOA and return the results, additionally it gathers monitoring data that the SLA module needs to check the current status of the platform and evaluate SLA contracts. The Execution Management Service Module is the responsible for governing the applications that run on the cloud. This component takes care of the initialization of applications, deployment, service (like...
Figure 11.2: Cloud4SOA 3-layer architecture

Figure 11.3: Components that form the governance layer

databases) life cycle, migration operations, etc. Moreover, this component may receive messages from the SLA Module regarding failures to meet its SLA in order to execute recovery actions.

Another layer of interest in the project is the Semantic one. This layer implements an interoperability framework formed by a User model, an Application model and a PaaS Offering model that is aimed at modeling these entities and at capturing and resolving differences between instances of them. The PaaS Offering Model (see figure 11.4) is used by Cloud PaaS providers to describe their semantic profile in a uniform way. Specifically, the model captures the available platform functionalities as well as technical- and business-driven characteristics.

This model permits describing the functionalities of the offering through the Operation class and the ways to execute these operations (Web Interface, CLI, API, etc.). It also allows describing its characteristics in terms of available Programming Languages, Software Components (such as database services) and Hardware Components (such as storage capacity).
The model is the result of a “meet-in-the-middle” approach consisting of both a top-down and a bottom-up approach. On one hand, the top-down approach consisted on the study of existing ontologies related to Cloud, SOA and application development and the specialization of these into more concrete terms relevant to Cloud4SOA. On the other hand, the bottom-up approach consisted on the extensive study of existing PaaS platforms in order to find a generalization encompassing all of their concepts.

### 11.2.2 Platform Adapters

Platform Adapters are external to the Cloud4SOA central system and communicate with it by means of the harmonized API (see figure 11.5).

A Platform Adapter can be seen as a proxy between the Cloud4SOA platform and the various PaaS platforms. It receives requests through an interface compliant to the harmonized API and invokes the appropriate platform-dependent operations directly on the PaaS. This wiring,
However, sometimes can be direct (just invoking an operation on the PaaS with the parameters received) and sometimes can require some additional work (see Section 10 for details).

As it can be seen in Figure 11.5, platform adapters do not run inside the Cloud4SOA platform but remotely in the PaaS platform. This decision has been motivated by the fact that there are some operations that can only be performed from inside the cloud, like getting some monitoring metrics, and also by the will of making the adapters more extensible and flexible. With this structure, adapters can be completely decoupled from the centralized platform, so that additional adapters can be easily incorporated in the future in order to widen the range of available PaaS platforms. Nevertheless, the adapters also have a local part that performs operations that cannot be performed by the remote adapter, the most obvious being the deployment of an application (along with the deployment its corresponding adapter).

It was decided to have one separate adapter for each application in order to reduce the risk of the adapter becoming a bottleneck and to achieve a certain degree of isolation for each group. When an application is deployed the first time, its corresponding adapter is deployed too and, similarly, when the application is deleted or un-deployed, the same is done with the adapter.

Adapters are implemented as RESTful Web Services exposing a set of methods compliant with the harmonized API. Each method is characterized as a route and a HTTP verb, always trying to follow the REST conventions as much as possible, some method also expect a set of parameters either in the query part of the URL or in the body of the request. Adapters are
simple entities that only contain business logic, they do not need a UI or a database since they rely on the PaaS platform API to complete the requested operations.

11.2.3 CloudFoundry Adapter Design

The CloudFoundry adapter, as the rest of adapters, is deployed as a RESTful service that is invoked by the Governance layer in the Cloud4SOA platform. The Adapter receives requests, processes them and ultimately invokes the RESTful API exposed by the Cloud Controller to perform the necessary operations on CloudFoundry (see figure 11.6).

![Figure 11.6: Invocation patterns between Cloud4SOA, the adapter and CloudFoundry](image)

As said in Section 10.3 we will use the Java client library as a starting point to design and build the adapter, and make quality contributions to it in order to fulfill the needs of this project. Although it would not be strictly necessary, the library has a data model representing the structure of CloudFoundry that uses to model the parameters of requests before sending them to the API. Having this intermediate model representation between the parameters received from Cloud4SOA and the parameters sent to CloudFoundry provides the following benefits to the architecture:

- Increased **changeability** since structural changes will only need to be introduced in the model, not in every affected operation.
- Increased **clarity** and readability, since the structure can be more easily examined.
- Increased **modularity** since the structure is decoupled from the service skeleton and, thus, can be altered without affecting the operations that use it.

Additionally, this decision helps to fulfill the NF4 requirement (“Supporting commonly used standards, standards syntax, open APIs, widely available tools, technologies, methodologies and best practices”), since it follows well-known best practices and methodologies from Software Engineering.

In the following sections we will present the CloudFoundry domain model, the high-level component structure of the client library and the adapter; and the Sequence Diagrams of each of the operations in the harmonized API.
11.2.4 Domain Model

Diagram 11.1 shows the CloudFoundry domain model. This model has been manually built by examining the source code of the CloudFoundry Java library, and is presented here to depict all the CloudFoundry concepts and their relationships as this can help to understand the library. Note that although the model tries to be as common as possible for both V1 and V2 there are some differences between them.

In the following sections there is a description of the classes.

Diagram 11.1: CloudFoundry domain model

Entity

This abstract class has metadata properties common to some of the entities in CloudFoundry. The GUID is used to uniquely identify each entity.

Application

An application created on the CloudFoundry platform. The num_instances and memory properties are used to set how many resources to assign to this application in terms of number of
concurrent agents executing it and available memory. These parameters can be tweaked to manually scale an application. Note also that an application can be bound to more than one URI.

**Space and Organization**

A space is basically an environment in which applications and services can live. Typical spaces are Testing, Development and Production. A space pertains to an organization, so that several organizations could coexist together in a more or less isolated way.

**Application Service**

This encompasses any external service that can be used by applications, the most common being databases. The way to model services is slightly different in V1 and V2, however in both versions services are created from available Service Configurations (for example, “MySQL database”).

In V1 a Service Configuration offers a set of Tiers, representing different usage plans (for instance, “free plan”, “light plan” and “pro plan”). Services are created from one of this configurations and plans. Note that the attributes in the Application Service and Service V1 are replicated; this is because a service, actually, has no direct access to its configuration.

In V2 a Service Configuration is associated to an Offering (such as “MySQL database”), which at the same time has a set of Plans (analog to Tiers). Note that since both Offering and Plan specialize from Entity their basic information, like the name, is represented through the entity.

**Stats**

Stats for an application are calculated through the stats of each of its running instances. CloudFoundry only provides information about basic metrics: CPU usage, memory usage, disk occupation and uptime.

**Info**

This encompasses a set of general information about the CloudFoundry installed platform. Since CloudFoundry allows placing the authorization component (UAA) anywhere, or even having a different authorization server based on OAuth2, the authorization_endpoint property tells how to access this authorization server. The info also contains information about available runtimes (such as “Java 1.6”) and frameworks (such as “Spring”).

**11.2.5 Client library conceptual model**

The client library will be used as a black-box whenever possible following the rule of minimum modifications; however, it is useful to see its structure and the exposed methods since it can help to understand how to use the library and how to extend it when needed.

Diagram 11.2 shows a class diagram depicting the different components that form the library and how the adapter will use them. It has been simplified to show only the relevant parts.
for the project. The library is divided into two separate components, the library used to execute REST operations on the Cloud Controller (from now on, REST library) which includes most of the operations needed by the harmonized API, and the library used to open tunnels to application services through Caldecott (from now on, Caldecott library).

The adapter uses the REST library to perform most of the operations. The library’s point of contact with the adapter is the CloudFoundry Client. This class is basically a proxy for the Controller Clients. The CF Client first uses the Factory to create a Controller Client and from that point, it interact with the created Client. The Factory either creates a V1 or V2 client depending on the version of the Cloud Controller. The Controller Client interface shows the methods that both versions of the client implement. The clients may also optionally have an OAuth2 Client, in case the Cloud Controller is using the UAA-based authentication mechanism.

The adapter uses a custom client to access the Caldecott library. The client first creates a Cloud Foundry Client which uses to deploy, govern and bind the service to the Caldecott app through the Tunnel Helper. Then it creates a local Tunnel Server that is the responsible for opening the actual tunnel. The server then uses a set of auxiliary classes to open the tunnel at the CloudFoundry side by sending a request through HTTP and to connect to it through a Socket. Once the socket is bound to the service, it is accessible through the “localhost” address.

As it can be seen from the diagram, the library makes a strong use of the Factory design pattern\(^2\) and an extensive use of interfaces. This further contributes to make the adapter more scalable and extensible.

There is actually much more going on in the internals of both libraries. The intention of these descriptions is not to give a full and detailed description of these components but rather to give an overview that helps to understand them.

### 11.2.6 Behavioral Model

This section shows the design sequence diagram of each of the operations in the harmonized API. The internal business logic of the client libraries is not shown, however the REST calls made from the library to the Cloud Controller are still shown so that it can be seen how finally a request reaches the CloudFoundry instance. It is assumed that the library communicates with a V1 Cloud Controller, except for the operations that are not available in this version.

As said in the previous sections, there is both a remote and a local part of the adapter, therefore, it is specified which one contains each operation. The diagrams also show the concrete signature of each method.

#### Authentication

Authentication in CloudFoundry is done using access tokens issued by an OAuth2 compliant authorization server, so that when the adapter wants to act on behalf of a certain user, it first has to get an access token by using the e-mail and password of the user. Once the access token has been issued, the adapter can freely use it to access the Cloud Foundry API until it expires.

---

\(^2\)The Factory design pattern was introduced in [GHJV95] and consists on having a factory class whose responsibility is to create and configure objects of a particular type. This encapsulates the creation and configuration of complex objects into a single place, increasing the cohesion and maintainability of the system. For example, the Controller Client Factory is a factory object that has a createClient method, which receives a set of configuration parameters and returns an Abstract Controller Client sub-type object, properly initialized.
In order to take profit of this SSO (Single Sing On) process, it would be desirable to store the access token of each user so that it can be reused as long as it does not expire. This would remove the burden of having to unnecessarily grant access every time the adapter has to be used. Unfortunately, Cloud4SOA is a web application and, as such, it is stateless. It could still use some form of sessions to store these access tokens; however storing them as cookies would comprise the security of the user and persisting them in a database would produce an additional burden. The final decision is to get the token every time from the authorization server.

In order to perform this authentication process the constructor function of the REST library automatically gets the access token by using the login method, so that it can freely interact with...
the Cloud Foundry API. The login method either uses an OAuth2 authentication server or the legacy mechanism from the Cloud Controller. Diagram 11.6 shows the process that is executed at the beginning of each operation. This part is not shown in the rest of the diagrams in order to avoid redundancy.

![Diagram 11.3: Authentication sequence diagram](image)

**Create application**

Diagram 11.4 shows the sequence diagram corresponding to the create application method. When creating an application in CloudFoundry, two parameters regarding its runtime environment can be specified: the programming language and the framework. However, Cloud4SOA only considers the first, so the framework is left undefined. Moreover, adapters in Cloud4SOA are created to manage only a single programming language which in this project will be Java in order to be compliant with the rest of the adapters. Additional parameters defined when creating the application are: the memory assigned which is set to a default value, its URI, its application services which is left undefined and a boolean indicating if the existence of an app with the same name has to be checked which is set to true in order to avoid duplicate applications.

Note that the description parameter is not used, since CloudFoundry does not have such a concept for applications. The diagram for the getSpace method is shown in diagram 11.5. This method also creates a space if it does not find it. The library currently does not have a method to create a space, so it will need to be implemented.
Diagram 11.4: Create application sequence diagram
Diagram 11.5: Get space sequence diagram
Delete an application

Diagram 11.6 shows the sequence diagram of the delete application method.

```
adapter : RemoteAdapter
```

```
DELETE /ems/application/{app_name}
```

```
HTTP Response
```

```
lib : REST library
deleteApplication(app_name)
```

```
Cloud Controller
```

```
DELETE /apps/{app_name}
```

```
(credentials, host)
```

```
request header parameters:
- email
- password
```

Diagram 11.6: Delete application sequence diagram

Update an application

Diagram 11.7 shows the sequence diagram of the update application method. Of all the parameters that Cloud4SOA allows to update, CloudFoundry only allows to update the environment (space).
Diagram 11.7: Update application sequence diagram

**Deploy**

Diagram 11.8 shows the sequence diagram of the deploy method. In order to deploy an application, first it has to be checked whether it exists or not. If it does not exist, then a NULL object is returned.
Deploy to environment

Diagram 11.9 shows the sequence diagram of the deploy to environment method. This method is basically a composition of the update application and deploy methods. First, the environment is updated with the update application method and then it is deployed in the same way as in the deploy method.
Diagram 11.9: Deploy to environment sequence diagram

Upload and deploy to environment

Diagram 11.10 shows the sequence diagram of the upload and deploy to environment method. This method is a composition of the following methods: create application, update application and deploy. The main interest of this operation is that it covers the full development life-cycle in one single method by using other API method as a support.
Get application status

Diagram 11.11 shows the sequence diagram of the get application status method. This method is actually more complex than it looks, since CloudFoundry is not prepared to get the status of an application in the form defined by Cloud4SOA. First, it has to check whether the application exists or not, and if it exists it will try to fetch the application code and use the result of the trial to determine whether the application is actually deployed, or created but not deployed yet.
Diagram 11.11: Get app status sequence diagram

Get running status

Diagram 11.12 shows the sequence diagram of the get running status method. It simply makes a transformation from the status value returned by the library to the values defined in Cloud4SOA.

Diagram 11.12: Get running status sequence diagram
Get application details

Diagram 11.13 shows the sequence diagram of the get application details method. It basically maps an application in CloudFoundry form into an application in Cloud4SOA form.

Start/Stop application

Diagram 11.14 shows the sequence diagram of the start or stop method. The method first retrieves information about the space associated with the received environment and creates a client object bound to this space (note that this is only done when interacting with a V2 Cloud Controller), and then invokes the start or stop method of the client depending on the operation requested (op parameter).
Diagram 11.14: Start/stop application sequence diagram

Check application name availability

Diagram 11.15 shows the sequence diagram of the check app availability method. It simply tries to get an application with the given name in order to determine if it already exists or not.
Diagram 11.15: Check app availability sequence diagram

**List applications**

Diagram 11.16 shows the sequence diagram of the list applications method. The method first retrieves information about all available applications for the user with the given credentials and then creates a Cloud4SOA application object containing a subset of the application’s information, for each of the applications retrieved.
Diagram 11.16: List applications sequence diagram

**Create environment**
Diagram 11.17 shows the sequence diagram of the create environment method. The implementation of this functionality requires implementing a new `createSpace` method in the client library, which receives the name of the space and the organization that it pertains to (in this case the organization is a default value since Cloud4SOA does not have this concept). Note, however, that this functionality is only applicable to V2 Cloud Controllers.
Delete environment

Diagram 11.18 shows the sequence diagram of the delete environment method. The delete space method has to be implemented and added to the library.

Update environment

Diagram 11.19 shows the sequence diagram of the updated environment method. The update space method has to be implemented and added to the library.
Create database

Diagram 11.20 shows the sequence diagram of the create database method. This operation has to be completed in two steps in CloudFoundry, given that it decouples applications from services. Services are created independently from applications and, then, application can be bound to these services. This binding process, additionally, automatically reconfigures the application in order to provide it with the appropriate credentials to access the database. CloudFoundry has a powerful auto-reconfiguration mechanism that is able to deal with almost any type of application. It is important to remark that the database credentials provided as parameters are not used, because CloudFoundry does not allow binding an application to a service with custom credentials, it needs to use the username and password automatically generated by him.

CloudFoundry bases the creation of services on Service Configurations, therefore, the method has to first retrieve an appropriate configuration for the given database type and then create the service based on it.
Diagram 11.20: Create database sequence diagram

Get databases list

Diagram 11.21 shows the sequence diagram of the get Databases list method. CloudFoundry does not have a clear distinction of application services types, it encompasses all of them (databases, messaging, backup, etc.) in the same “Service” concept. Therefore, all services needs to be retrieved and then select the database ones. Prior to that, however, the space to which the application pertains has to be set in the library.
Diagram 11.21: Get database list sequence diagram

Delete database

Diagram 11.22 shows the sequence diagram of the delete Database method. In Cloud4SOA, DBs are always bound to applications and, thus, the path of this method is a specialization of the path to access an application. The method first retrieves information about the application in order to get its space and then deletes the given database from that space.
Diagram 11.22: Delete database sequence diagram

Export application

Diagram 11.23 shows the sequence diagram of the export app version method. It requires the implementation of the new `downloadApp` method in the client library which receives the name of the application and returns the archive containing it.

Download/restore database

Diagram 11.24 shows the sequence diagram of the download and restore database methods. Both method share the same structure and differ only in the command to execute while the
tunnel is opened.

Diagram 11.24: Download/restore database sequence diagram

11.3 Implementation

The implementation of the adapter has been split in two parts: the local part and the remote part. The local part has been integrated in a currently existing project in the Cloud4SOA ecosystem while the remote adapter has been implemented as a brand new RESTful service decoupled from the rest of the components.

The Cloud4SOA ecosystem is formed by a set of projects, each one implementing one component. Moreover, each component exposes its interface in a common project (the API project), so that other components that need to use it only have to set the API project as a dependency. This creates a decoupled infrastructure, where the connection point is this API project. Remote adapters follow a similar approach, by pointing to a common project that
defines the exposed API, so that clients using the remote adapters only need this common project as a dependency.

11.3.1 Technologies involved

Both adapters have been implemented in Java, since this has been the programming language adopted for the Cloud4SOA project. Other technologies involved include Maven and JAX-RS

Maven

Maven\(^3\) is a project management and building tool for Java projects. This tool is used consistently in all projects within the Cloud4SOA ecosystem in order to manage dependencies and to manage the project’s building process, although Maven can be used for a much wider range of functionalities.

Maven manages a project through an XML file (the Project Object Model or POM) that contains a description about its components, dependencies, build actions, etc. in a similar fashion to GNU make or Ant. When building a project, all specified dependencies are automatically downloaded from Maven repositories and included in the project classpath.

Maven also allows creating hierarchies of POMs, so that all actions and dependencies can be inherited. This mechanism is used within Cloud4SOA by specifying a “Master POM” from which the rest of POMs inherit.

JAX-RS

JAX-RS\(^4\), part of JAX-WS, is a Java library that eases the creation of RESTful Web Services. In Cloud4SOA, this library is used to create the RESTful services for the remote part of the adapters.

JAX-RS allows transforming an entity class with a set of methods into a RESTful Web Service by annotating its methods with the path, the parameters and the HTTP verb. An example of such a method is shown in the following code snippet:

```java
@GET
@Path("/ems/application/{appid}")
@Produces({"application/json", "application/xml", "text/plain"})
public ListApplicationResponse getApplicationDetails(@PathParam("appid") String appid) {
    ListApplicationResponse response = new ListApplicationResponse();
    [...] return response;
}
```

With these annotations, JAX-RS configures the service appropriately, deals with HTTP headers, incoming requests, outgoing responses, etc. JAX-RS provides a wide set of tools to perform advanced operations such as filters, message interceptors, HTTP headers operations, etc. However, Cloud4SOA only uses the core functions of the library.

\(^3\)http://maven.apache.org
\(^4\)http://jax-rs-spec.java.net/
11.3.2 Local Adapter

The implementation of the local part of the adapter is integrated within one of the currently existing projects which contains the business logic for the local implementation of all the adapters. An example of a method in the CloudFoundry local adapter can be seen in the following code snippet:

```java
public static String createAndDeployCF(String email, String password, String appName, String binary) throws Cloud4SoaException {
    CloudFoundryClient client = null;
    try {
        client = init(email, password);
        // check whether an application with appName name
        // already exists
        try {
            client.getApplication(appName);
        } catch (CloudFoundryException ce) {
            // create it if it doesn’t exist, otherwise propagate
            // the exception
            if (ce.getStatusCode() == HttpStatus.NOT_FOUND) {
                AuxAdapter.createApplicationCFbase(email, password, appName, client);
            } else {
                throw ce;
            }
        }
        File f = new File(binary);
        client.stopApplication(appName);
        // stop application before uploading the code to
        // avoid conflicts
        client.uploadApplication(appName, f);
        // after uploading the code, start the application again
        client.startApplication(appName);
        CloudApplication app = client.getApplication(appName);
        return app.getUris().get(0);
    } catch (CloudFoundryException e) {
        // In case of error, check it and propagate the
        // appropriate exception
        if (e.getStatusCode().equals(HttpStatus.NOT_FOUND)) {
            throw new Cloud4SoaException("The application cannot be found");
        }
        throw new Cloud4SoaException(e.getMessage());
    } catch (IOException e) {
        throw new Cloud4SoaException("An error occurred uploading");
    } finally {
        if (client != null) finalize(client);
    }
}
```
The first part of the method initializes a new CloudFoundryClient by creating a new instance of the object and logging in with the provided credentials. Then, the method checks whether an application with the same name has already been created and creates it otherwise. Finally, the application is stopped, the new code uploaded and the application started again so that the Cloud Controller can stage the application with the new code.

The following snippet shows the code of the init method:

```java
public static CloudFoundryClient init(String email, String password) throws Cloud4SoaException {
    try {
        // create client object with the given credentials and URL
        CloudFoundryClient client = new CloudFoundryClient(
            new CloudCredentials(email, password), new URL(CC_URL));
        client.login();
        return client;
    } catch (ResourceAccessException e) {
        throw new Cloud4SoaException(
            "Connection with the PaaS was refused.");
    } catch (CloudFoundryException e) {
        // check if the error is due to incorrect credentials
        if (e.getStatusCode().equals(HttpStatus.FORBIDDEN)) {
            throw new Cloud4SoaException("Authentication error.");
        }
    }
    throw new Cloud4SoaException(e.getMessage());
}
```

The method basically creates a new client object with the given credentials and URL and authenticates the user. The method also checks the exceptions returned and tries to determine the cause of the error in order to propagate the appropriate exception (refused connection, bad credentials or other).

11.3.3 Remote Adapter

The remote adapter has been implemented as a brand new project that basically contains a RESTful web service implementing the Harmonized API, along with the CloudFoundry Java library. We have used JAX-RS in order to implement the external interface of the web service, by annotating each of the methods with its path and its HTTP verb.

As said in the introduction of this section, the remote adapter complies with a common interface defined in an independent project. This project defines the routes and its associated methods along with their Request and Response objects which group the parameters of the requests and responses. The method shown in the JAX-RS section above, for instance, returns a ListApplicationResponse object.
11.3.4 Integration with Cloud4SOA

A crucial part of the implementation of the adapters is to fully integrate them with the centralized part of Cloud4SOA.

Cloud4SOA has two points of contact with the adapters: the Execution Management Service Module (EMS) and the Monitoring Module, both in the Governance layer. Due to the decoupled nature of the remote part of the adapters, the integration with it has been straightforward and has not required any changes on these modules. When an application is deployed through Cloud4SOA, the URL of the remote part of the adapter is stored in the database and it is then used to execute operations on it. This way, the behavior in the governance layer is the same regardless of the actual adapter that is being invoked. Integration with the local part, in contrast, has required including some additional logic into the EMS in order to distinguish the case of CloudFoundry and to invoke the adapter methods accordingly.

Another crucial part of the integration with Cloud4SOA is to create the semantic profile for the platform based on the PaaS Offering semantic model and include it into the PaaS repository. The goal is to ultimately allow users themselves to add new platforms to the repository through the Cloud4SOA dashboard, however at the time of implementing the CloudFoundry adapter this feature is still not functional, instead the following steps have to be followed:

1. Create the semantic profile using an ontology editor, based on the PaaS Offering model
2. Use the Semantic Initializer component to update the database with the new semantic profile.

Figure 11.7 shows a part of the CloudFoundry instantiation of the model.

11.4 Testing

Testing is an important phase when developing complex systems with numerous components that have to work together. In such a scenario, the developer of each component must ensure that it is as reliable as possible since a bug in it can cause the whole system to crash.

In this matter, automatic testing techniques can be used to provide a set of tests for each component to be run each time the code is changed. In the case of Cloud4SOA, each component has its own test suite which is run each time the component is built (using Maven). Moreover, there is a parent project with references to every other project. By building the parent project, all the referenced projects are built (and their tests executed). This way, the parent project can be built to ensure that the whole Cloud4SOA ecosystem remains stable.

Cloud4SOA uses JUnit to implement these tests, as it’s the de facto automatic testing tool for Java. Using this technology, we have implemented two test suites in the CloudFoundry adapter: one for the local part and another for the remote part. Below, an overview of JUnit is given along with the description of the designed tests.

11.4.1 JUnit Overview

Automatic testing refers to the control of tests execution, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions and other test control and test reporting functions [KH07]. This technique is aimed at partially replacing the classical manual testing
The main benefit is that tests only need to be written once, and then they can be quickly run each time a program changes, making the testing process quick, but reliable at the same time. Automatic testing techniques have been being used for years; however they are now gaining more popularity with the popularization of Agile Software Development methodologies (such as eXtreme Programming).

JUnit\(^5\) is an automatic testing framework for Java focused on Unit testing. Unit testing refers to the testing of the smallest testable part of a program, and is usually the starting point for other more complex types of tests. What the smallest testable part of a program is depends on the actual type of application. In functional programming, it is usually a function while in object-oriented programming it is a class.

JUnit is mainly used through annotations. The main annotation is @Test, which indicates that a method is actually a test that has to be run. Other annotations include @Before and @After which indicate actions that must be performed before and after each test. Inside tests, JUnit provides a set of Assert methods that can be used to compare actual outcomes with expected outcomes. An example of a JUnit test can be seen in the following snippet:

\(^5\)www.junit.org
@Test
public void testDeleteApplication() throws Exception {
    System.out.println("Testing Delete Application");
    System.out.println("--------------------------------------");
    System.out.println("Delete existing application");
    Adapter.createApplication("CloudFoundry", USER, PASS, "", APP_NAME, null);
    Assert.notNull(AuxAdapter.getApplicationCF(USER, PASS, APP_NAME));
    Adapter.deleteApplication("CloudFoundry", "", USER, PASS, "", APP_NAME, null, null, null, null, null, null, null);
    try {
        AuxAdapter.getApplicationCF(USER, PASS, APP_NAME);
        fail("No Exception!");
    } catch (Cloud4SoaException e) {
        assertTrue(e.getMessage().contains("cannot be found"));
    }
    System.out.println("Delete non-existent application");
    Adapter.deleteApplication("CloudFoundry", "", USER, PASS, "", "fakename", null, null, null, null, null, null, null);
    System.out.println("Bad name parameter");
    Adapter.deleteApplication("CloudFoundry", "", USER, PASS, "", "", null, null, null, null, null, null, null);
}

This @Test checks that an application recently created can be retrieved properly (by asserting that the getApplication method neither returns a not-null object nor throws an exception). Then deletes it and checks that it cannot be retrieved by checking that the proper exception is thrown; the fail methods makes the test fail immediately and is used to indicate that an expected exception was not raised. The two last calls to deleteApplication check that in case that an application is not found, the method does not return any exception and returns properly.

11.4.2 Local Part Tests

A test has been designed for each of the main implemented methods, along with a test for the authentication mechanism. Each test brings the method under certain situations and checks whether the outcomes coincide with the expected output (see table 11.5). Tests are independent from each other, since no order can be guaranteed when executing unit tests.

The only two tests that have some preconditions are the “Download database” and “Restore database”. The first requires that a specific database with a specific table exists, while the second only requires the existence of the database. The rest of the tests prepare the environment themselves, for instance, the delete application test first creates the application. This approach reduces the effectiveness of tests since, for instance, the delete application method may be working but the test could still fail because of the create application method; however it makes the tests independent from each other, in the sense that they can be executed in any order.

Figure 11.8 shows the result of the tests execution.
<table>
<thead>
<tr>
<th>Use Case</th>
<th>Situation</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication</td>
<td>Authenticate with correct credentials</td>
<td>No Exception</td>
</tr>
<tr>
<td></td>
<td>Authenticate with incorrect credentials</td>
<td>Unauthorized exception</td>
</tr>
<tr>
<td>Create application</td>
<td>Create application with correct data</td>
<td>The application is created and can be retrieved</td>
</tr>
<tr>
<td></td>
<td>Create application with empty name</td>
<td>Exception saying that the name cannot be empty</td>
</tr>
<tr>
<td>Delete application</td>
<td>Delete existing application</td>
<td>The application is deleted and an exception is raised when trying to get it</td>
</tr>
<tr>
<td></td>
<td>Delete non-existent application</td>
<td>No exception</td>
</tr>
<tr>
<td>Deploy application</td>
<td>Deploy existent application</td>
<td>The application can be accessed through its URL, and it returns the correct HTML page</td>
</tr>
<tr>
<td></td>
<td>Deploy non-existent application</td>
<td>Not Found Exception</td>
</tr>
<tr>
<td></td>
<td>Deploy non-existent binary file</td>
<td>IO Exception</td>
</tr>
<tr>
<td>Get application status</td>
<td>Create application and get status</td>
<td>Status is CREATED</td>
</tr>
<tr>
<td></td>
<td>Deploy application and get status</td>
<td>Status is DEPLOYED</td>
</tr>
<tr>
<td></td>
<td>Get status from non-existent application</td>
<td>Not Found Exception</td>
</tr>
<tr>
<td>Get running status</td>
<td>Create application and get status</td>
<td>Status is STOPPED</td>
</tr>
<tr>
<td></td>
<td>Deploy application and get status</td>
<td>Status is RUNNING</td>
</tr>
<tr>
<td></td>
<td>Stop application and get status</td>
<td>Status is STOPPED</td>
</tr>
<tr>
<td></td>
<td>Get status from non-existent application</td>
<td>Not Found Exception</td>
</tr>
<tr>
<td>Check application availability</td>
<td>Create application and check availability</td>
<td>False is returned</td>
</tr>
<tr>
<td></td>
<td>Check availability of non-existent application</td>
<td>True is returned</td>
</tr>
<tr>
<td>Download database</td>
<td>Download an existing database</td>
<td>The dumped files contains the correct SQL statements</td>
</tr>
<tr>
<td></td>
<td>Download a non-existent database</td>
<td>Not Found Exception</td>
</tr>
<tr>
<td>Restore database</td>
<td>Restore an existing database</td>
<td>The database is restored properly</td>
</tr>
<tr>
<td></td>
<td>Restore a non-existent database</td>
<td>Not Found Exception</td>
</tr>
</tbody>
</table>

Table 11.5: Description of the local part unit tests
11.4.3 Remote Part Tests

To implement the unit tests for the remote part of the adapter we have followed the same approach. Table 11.6 shows the different tests and the tested situations.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Situation</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication</td>
<td>Authenticate with correct credentials</td>
<td>No Exception</td>
</tr>
<tr>
<td></td>
<td>Authenticate with incorrect credentials</td>
<td>Unauthorized exception</td>
</tr>
<tr>
<td>Get application info</td>
<td>Get information of existent application</td>
<td>The application is retrieved properly</td>
</tr>
<tr>
<td></td>
<td>Get information of non-existent application</td>
<td>Not found exception</td>
</tr>
<tr>
<td>List applications</td>
<td>Get the list of applications</td>
<td>The list of applications is properly retrieved</td>
</tr>
<tr>
<td>Start/stop application</td>
<td>Start/Stop existing application</td>
<td>The status of the application is updated accordingly</td>
</tr>
<tr>
<td></td>
<td>Start/Stop non-existent application</td>
<td>Not Found Exception</td>
</tr>
<tr>
<td>Create database</td>
<td>Create database with correct data</td>
<td>The database is created and can be retrieved</td>
</tr>
<tr>
<td></td>
<td>Create database with empty name</td>
<td>Exception is given saying that the name cannot be empty</td>
</tr>
<tr>
<td>Get DB information</td>
<td>Get information of existing database</td>
<td>The information can be retrieved properly</td>
</tr>
<tr>
<td></td>
<td>Get information of non-existent database</td>
<td>Not Found Exception</td>
</tr>
<tr>
<td>List databases</td>
<td>Get the list of databases</td>
<td>The list of databases is properly retrieved</td>
</tr>
<tr>
<td>Delete database</td>
<td>Delete exiting database</td>
<td>The database is deleted and an exception is raised when trying to get it</td>
</tr>
<tr>
<td></td>
<td>Delete non-existent database</td>
<td>No Exception</td>
</tr>
</tbody>
</table>

Table 11.6: Description of the remote part unit tests
Unlike tests in the local adapter, these tests have more preconditions since operations such as create application are only implemented in the local part of the adapter. This way, for instance, the delete method requires that an application with a specific name has been already created.

Figure 11.9 shows the results of the test execution.

![Test Results]

Figure 11.9: Results of the remote part tests execution

11.4.4 Discussion on NFRs

The definition of NFRs has been out of the scope of this project, since it has been already done in the first phases of the Cloud4SOA project. The focus of the team on the elicitation of requirements has been on functional requirements rather than on non-functional ones. As explained in section 11.1, a list of non-functional requirements has been gathered; however the goal of these requirements is to provide developers with a set of principles to follow rather than to provide a strict set of rules to be validated. For this reason, non-functional requirements are described in a high-level and non-measurable way.

The implementation of the adapter has taken into account these NFRs by making use of the CloudFoundry authentication mechanisms (NF1.1) and by complying with the common interfaces of Cloud4SOA, specifically the harmonized API and the common REST interface (NF3.1 and NF5.1). Furthermore, standard technologies used through all the Cloud4SOA project, such as Maven, JAX-RS and JUnit, have been adopted in the adapter (NF4).
Chapter 12

PaaS Hybrid Cloud Scenarios

We have evaluated the Cloud4SOA platform by walking through two hybrid cloud scenarios involving a context-aware applications developed by one of the Cloud4SOA partners (PTIN).

In the next sections, these two scenarios will be described, as well as the benefits of a hybrid approach, and the Cloud4SOA system will be evaluated using these scenarios as a basis.

12.1 Hybrid Cloud

Initially Cloud4SOA supported diverse public Platform-as-a-Service offerings, such as, CloudBees, Amazon Beanstalk and Heroku. With the addition of a private cloud we have opened the door to a hybrid deployment approach can bring the benefits of enabling the deployment of applications via a paradigm that has gained a lot of importance in the cloud community during the last years.

The hybrid cloud seeks to alleviate the inherent limitations of purely public and private approaches by combining the two of them into a heterogeneous cloud that leverages their strengths. Hybrid clouds have a dynamic nature; they change their composition and topology over time in order to adapt to varying conditions such as peaks in demand or SLA constraints. This type of cloud allows organizations to mix their own cloud infrastructure with public services in order to achieve the highest levels of performance, maximum utilization and compliance with laws and regulations, maximizing, this way, the ROI (Return-on-Investment). A hybrid deployment approach fits especially well in situations where an application is composed of heterogeneous pieces of software, flexible enough to be distributed among distinct clouds. For instance, let's say that an application has a component that deals with sensitive customer data which needs to be stored under a set of strict regulations and another component which runs a complex algorithm that needs high computational power. In such a scenario, a hybrid deployment would fit perfectly since the first component could be deployed into a private cloud while the second could be hosted in a public one. Another suitable scenario for a hybrid deployment is when an application is expected to have punctual peaks in demand. In such situation, resources can be migrated from a private cloud to a public one in a process known as cloud bursting. For some community members [ZCB10], the hybrid paradigm is considered ideal for any user wishing to adopt cloud computing while maintaining the flexibility and security that is required.

Figure 12.1 depicts an example of how a hybrid cloud approach highlights the association of public cloud services with private cloud services.
In summary, these are specific situations where a hybrid approach is an added-value for the organization adopting cloud computing:

- Legacy applications which can be perfectly hosted in an on-premises environment.
- The need for bursting to the cloud in event of traffic peaks.
- The need to scale web applications through a public cloud.
- To avoid vendor lock-in.
- The costs related to backups systems are unaffordable.

### 12.2 Scenarios Description

#### 12.2.1 Hybrid Deployment Scenario

The hybrid deployment scenario was designed to highlight some of the benefits of this type of cloud. In this scenario we use a private instance of CloudFoundry together with the public cloud CloudBees to distribute an application formed by several components. In the scenario, Cloud4SOA will be used, through its web interface, to create, deploy and monitor two components of the same major application across the two selected platforms.

Figure 12.2 shows the high level picture of the hybrid scenario.

#### 12.2.2 Bursting on Demand Scenario

The bursting on demand scenario showcases the transference of resources between the private and public environment in the event of violated SLA terms (uptime, response time, etc.). In
the scenario, Cloud4SOA informs the developer when any threshold is exceeded, producing a violation in a SLA term, and then the developer decides to migrate his affected application’s resources, re-deploying them into another PaaS through the Cloud4SOA interface.

Figure 12.3 illustrates the interaction between the Cloud4SOA system and the developer in this scenario.
12.3 Context-Aware Application Description

One of the partners of the Cloud4SOA consortium, Portugal Telecom Inovação (PTIN), has developed a framework that includes a set of pervasive services which can intelligently take proactive actions to select, rate and deliver suitable content (e.g. photos, videos, etc.) based on user’s context information. With this framework, public spaces can become smart by offering a variety of mobile services that react to the changing client environmental conditions.

The framework is composed by several modules that capture and process information from the consumer’s mobile terminal. A central and fundamental component of the framework is an Extensible Messaging and Presence Protocol (XMPP) server called Context Broker. This server has an active role in the actions performed by the 3 Java service enablers: the Context Enabler, the Group Manager Enabler and the Content Selection Enabler:

The Context Enabler is a simple SOAP web service that interacts directly with the Context Broker subscribing and receiving context information about a specific user. The existence of this enabler allows the loose coupling and the reutilization of services.

The Group Manager Enabler is a rule engine responsible for identifying, creating and managing groups of consumers based on their context information. This service subscribes to the Broker in order to receive context information and establishes several groups based on pre-conditions set by the administrator of the enabler: location, social preferences, sensors information, presence, gender, etc.). The Group Manager Enabler uses a MySQL database.

1XMPP Standards Foundation: http://xmpp.org/
The Content Selection Enabler is also a rule engine that aims at rating and selecting content (e.g. photos, videos, etc.) for users based on their group(s). This module is as an essential element in the decision and delivery of content based on context information. For the Content Selection Enabler to be able to select content from the Content Provider, it must be previously configured with administrator rules. The configuration is nothing more than a set of rules that relate the content meta-data to the high-level context information. The Content Selection Enabler uses a MySQL database for keeping the matching rules.

Since the framework is based on several autonomous services, it is possible to exploit the distribution of its components across several platforms and test their behavior in each environment through the defined hybrid cloud scenarios. Figure 12.4 depicts the framework that will support the defined hybrid scenarios.

In order to produce a comprehensive design of the framework architecture, PTIN has adopted a widely accepted model that provides multiple points of view of a system’s architecture. This model is widely known as the “4+1” View Model of Software Architecture introduced in 1995 by Philippe Kruchten [Kru95]. This document describes the physical view of the system. Other views can be found in other deliverables of the Cloud4SOA project.

In the two hybrid cloud scenarios we will deploy two services from the context-aware multimedia framework: Group Manager Enabler and Context Enabler.
The Group Manager Enabler will be firstly deployed in a unique instance of a simple servlet container, such as Tomcat, from CloudBees and bound to a MySQL database provided by CloudBees. On the other hand, the Context Enabler SOAP service will be deployed in the Atos CloudFoundry instance. CloudFoundry offers Tomcat containers where the application archive can be deployed and scaled if necessary. Finally, the XMPP server (Context Broker) will be running on-premises within PTIN and receiving the users’ sessions from their mobile devices. Figure 12.5 illustrates the physical view and deployment diagram planned for the two hybrid cloud scenarios.

![Application Deployment Map](image)

**Figure 12.5: Application Deployment Map**

### 12.4 Cloud4SOA Evaluation

The Cloud4SOA evaluation through the proposed scenarios reveals the behaviour of the main supported features and provides some insights on the aspects that should be improved. We have divided the evaluation into a set of essential operations derived from the scenarios, such as, deployment, governance and monitoring. For each of these operations we show how it can be performed through the Cloud4SOA interface and discuss some relevant aspects about it. It is important to remark that this is an evaluation from the user point of view, revealing some important issues about the interface and evaluating aspects such as the usefulness of the platform or the overhead in time imposed by the platform when performing operations on the cloud. To evaluate the latter we have tested the operations across two strategies: By using Cloud4SOA and by directly using the CloudFoundry or CloudBees API, and compared the average time required for each by repeating the operation 10 times.

#### 12.4.1 Deployment Process

After the straightforward registration process via the Cloud4SOA GUI, we can create an application profile using the provided editor. This way, the Context Enabler’s profile is constructed
by inserting its name, version, language and some QoS metrics values, such as the uptime and maximum latency. These QoS metrics will be used later by Cloud4SOA to check whether they are respected by each PaaS provider.

We think that the concept of application profile can be very useful for the user since it lets him describe a type of application in abstract terms and use this definition multiple times to deploy multiple applications in a quicker way. However, we have found that the interface should describe the concept of application profile clearer and make more explicit the need to create an application profile before deploying an application.

Figure 12.6 shows the creation process of an application profile for the Context Enabler.

The “Search” button enables to retrieve which providers better match the Context Enabler’s application profile. From the returned list, depicted in Figure 12.7, we choose the Cloud-Foundry platform which is running as a private instance at Atos infrastructure.

The list of matched PaaS platforms is ordered according to the matching score, that is, platforms that better match the requirements defined in the application profile appear first in the results. This provides a great benefit to developers since it is often difficult to decide which PaaS to use among the wide range of platforms in the market. We think, however, that the
search screen could be further enhanced by providing some insight about the score given to each platform. This could help the developer even more to decide which provider to use.

![Figure 12.7: Matching PaaS Offers for Context Enabler](image)

In the left-hand screen there is the “Deploy” button that we can use to start deploying the application to CloudFoundry, as stated in Figure 12.8.

![Figure 12.8: Deploying Context Enabler](image)

Finally, we need to insert our CloudFoundry account credentials and select the Context Enabler web archive, which has a size of 4MB, before proceeding to the actual deployment process. Figure 12.9 and Figure 12.10 shows these steps.
Figure 12.9: CloudFoundry credentials

Figure 12.10: Deploying Context Enabler web archive
We have found that the credentials screen is too generic and should adapt better to each PaaS. We see that this screen offers three fields to enter the credentials: “Public key”, “Private key” and “Account name”, however, in the case of CloudFoundry the credentials are expressed in form of email and password. This can be confusing to developers which may have trouble discovering how to enter their credentials.

One critical issue that we have found when performing the last steps of the deployment is a lack of details in error messages. Actually, Cloud4SOA only returns a generic error message when the deployment fails regardless of the cause: invalid credentials, service unavailable, incorrect parameters, etc.

After entering the required information about it, the application is deployed. Depending on the web archive size and the latency to the Cloud4SOA server and the Cloud Foundry API endpoint, the deployment time can vary. Once the process is finished, the application is deployed and ready to be monitored as depicted in Figure 12.11.

It is important to evaluate whether the overhead in time added by the Cloud4SOA system is significant or not for the deployment process, since this can be a factor of relevance when deciding to use Cloud4SOA. The results from Table 12.1 exhibit a slight difference between the two strategies. However we can consider this difference acceptable considering the added value that Cloud4SOA provides to the integral application management process.

![Figure 12.11: Context Enabler running in CloudFoundry](image)

<table>
<thead>
<tr>
<th>Cloud4SOA Platform</th>
<th>CloudFoundry Provider API</th>
</tr>
</thead>
<tbody>
<tr>
<td>aprox. 42 sec.</td>
<td>aprox. 28 sec.</td>
</tr>
</tbody>
</table>

Table 12.1: Cloud4SOA vs CloudFoundry Provider API deployment time (in seconds)
At this point, the Context Enabler is running in the private platform CloudFoundry. Next, we can replicate the same process for the other web service (the Group Manager Enabler). The Group Manager Enabler requires MySQL for persistence which must be explicitly specified in the application profile when creating it.

During the deployment process, the database can be created through the same Cloud4SOA platform, as depicted in Figure 12.12.

We think that making the creation of the database an integral part of the deployment process is very convenient since it avoids having to create the database in alternative ways, thus speeding up the setting up of the application.

When creating the database we also evaluated the time spent when using the two possible strategies (through Cloud4SOA or directly through the CloudBees API). The results (see table 12.2) show a meaningful difference, however, it can be considered acceptable taking into account the convenience of this feature. Nevertheless, we think that decreasing this overhead could enhance the user experience when deploying an application with a database which, actually, is the most typical case.

After this process, the database gets created and Cloud4SOA shows its details such as the database URL for using in import/export data processes. Figure 12.13 details the returned information.

<table>
<thead>
<tr>
<th>Cloud4SOA Platform</th>
<th>CloudBees API</th>
</tr>
</thead>
<tbody>
<tr>
<td>aprox. 22 sec.</td>
<td>aprox. 5 sec.</td>
</tr>
</tbody>
</table>

Table 12.2: Cloud4SOA vs CloudBees creating database (in seconds)
During the deployment of the Context Enabler we have again compared the time using the two strategies. In general, the deployment process proves to be slower to CloudBees than to CloudFoundry. The reason is, on one hand, that the Group Manager Enabler is a bigger application (10MB), and on the other hand, that the CloudBees API endpoint takes more time to process the request.

The results from Table 12.3 show an overhead of about 25 seconds which, again, is acceptable.

Figure 12.14 shows that at the moment the 2 services are deployed and running simultaneously, conforming to the hybrid cloud scenario.

<table>
<thead>
<tr>
<th>Cloud4SOA Platform</th>
<th>CloudBees API</th>
</tr>
</thead>
<tbody>
<tr>
<td>approx. 155 sec.</td>
<td>approx. 139 sec.</td>
</tr>
</tbody>
</table>

Table 12.3: Cloud4SOA vs CloudBees deployment time (in seconds)
The Group Manager Enabler administration interface (see figure 12.15) running on CloudBees, details the pre-configured groups and respective mobile users’ that are assigned to them. In this case, one user is online and belongs to “People in Aveiro” group based on his current position.

![Group Manager Enabler Administration Interface](image)

Figure 12.15: Group Manager Enabler Administration Interface

### 12.4.2 Governance Process

Cloud4SOA enables basic governance operations such as start, stop and undeploy an application. To stop the Context Enabler running in CloudFoundry, for instance, a developer can click on the “Stop” button as illustrated in Figure 12.16 and Figure 12.17.

![Start/Stop Application](image)

Figure 12.16: Start/Stop Application
After a confirmation dialog, the application is stopped which is indicated by the status attribute. If we try to access the application we get a “Not Found” error showing that, effectively, the application has been stopped (see Figure 12.18).

As previously done with the deployment and the creation of the database, we have compared the time spent by following each of the two strategies to stop the application, (see Table 12.4). This time, we have found that the overhead in time introduced by the use of Cloud4SOA is meaningful and should be reduced in order to enhance the user experience.

<table>
<thead>
<tr>
<th>Cloud4SOA Platform</th>
<th>CloudFoundry Provider API</th>
</tr>
</thead>
<tbody>
<tr>
<td>aprox. 11 sec.</td>
<td>aprox. 5 sec.</td>
</tr>
</tbody>
</table>

Table 12.4: Cloud4SOA vs CloudFoundry Provider API stopping application (in seconds)

The undeploy feature enables a quick deletion of any created application. By selecting the application and then clicking the “Undeploy” button we can easily remove the Context Enabler. After a confirmation dialog the application is undeployed and removed from the list (see Figure 12.20). One critical issue that we have found when performing this operation (as well as the stop operation) is that, as it happens when deploying an application, there is a lack of detail in error messages which makes it very difficult to debug a problem.

Table 12.5 shows the overhead added by Cloud4SOA in the undeployment operation. Again, the difference is significant and should be reduced in order to improve the user experience.
Figure 12.19: Undeploy Application

<table>
<thead>
<tr>
<th>Cloud4SOA Platform</th>
<th>CloudFoundry Provider API</th>
</tr>
</thead>
<tbody>
<tr>
<td>aprox. 15 sec.</td>
<td>aprox. 6 sec.</td>
</tr>
</tbody>
</table>

Table 12.5: Cloud4SOA vs CloudFoundry Provider API undeployment time (in seconds)

Figure 12.20: Context Enabler undeployed
12.4.3 Monitoring Process

The monitoring process is crucial for checking the behaviour of an application and detecting problems even before they occur. Cloud4SOA supports the monitoring of two metrics: response time and application health. These metrics can be shown selecting the application and then clicking on the "See Monitoring Details" button. Figure 12.21 illustrates this step.

Cloud4SOA returns a unified view of the metrics independently of the provider where the application is hosted. This makes it easy to compare applications running on different providers. Figure 12.22 shows the response time for the Context Enabler application.

The application health is abstracted as the response code returned by HTTP calls made to the application and is represented by a pie chart. Figure 12.23 shows the monitoring for the Group Manager Enabler which is working correctly (it is responding HTTP 200 status code).
The ability to check the status and performance of an application and compare it to other applications in an abstract way can provide a great benefit to developers by providing data that can be used to take decisions such as the migration of an application. We think, however, that the list of metrics should be expanded and include other meaningful metrics such as CPU, and memory usage or availability. With a larger and more comprehensive set of metrics the developer can feel more confident when using these data to take decisions.

12.4.4 Migration Process

Currently, Cloud4SOA doesn’t trigger notifications when an SLA term is violated. However, if this process were available, a developer would be informed about such violations and could decide to migrate his resources in order to ensure better performance results.

The migration process can be done through the “Migration” screen of the Cloud4SOA interface. After selecting the application to migrate, we can choose another provider that matches the application profile in the same way as it is done when deploying an application for the first time. In this scenario we want to migrate the Context Enabler, which is actually running in CloudFoundry, thus, we select CloudBees as the new platform to host the service. This process is illustrated in Figure 12.24.
Cloud4SOA asks once again for the CloudBees account credentials (see Figure 12.25).

Finally, we have to select the application’s web archive before proceeding to the migration process, as depicted in Figure 12.26.

The migration process is almost identical to deploying an application for the first time, but it adds the removal of the application that was deployed in the previous PaaS provider to the overall process.

Figure 12.27 and Figure 12.28 show that the Context Enabler is now deployed in CloudBees and the migration process has finished successfully.
12.5 Summary

Through the detailed scenarios we have evaluated the main features provided by Cloud4SOA and uncovered some possible improvements that could be made to enhance the user experience. With the help of Cloud4SOA, we deployed 2 applications from the PTIN multimedia context-aware framework: the Context Enabler and the Group Manager Enabler, across the CloudBees and CloudFoundry platforms. In addition, we compared the overhead in time imposed by Cloud4SOA system against the strategy of using directly the aforementioned platforms’ APIs.

Firstly, the deployment process involves the definition of an application profile that describes synthetically a service (programming language, QoS requirements, etc.). This profile is then used to retrieve the platforms that better match the defined requirements. However, the returned matching score is still ambiguous, since in the performed tests, the returned platforms matched the same score and there is no explanation about why a platform is given a particular score. Some specific changes could be performed in order to help developers even more when choosing a platform provider. Before deploying an application, Cloud4SOA shows a dialog which requires to introduce the developer’s credentials on the selected platform through a simple form with 3 text fields: “Public key”, “Private key” and “Account name”. We think that this is a solution that tries to unify the different approaches used by all the supported providers,
however it would be beneficial for the user to have customized fields depending on the selected platform. When the application requires a database-as-a-service, Cloud4SOA requests the filling of another form with the database credentials and initialization script. Once this is done, a web archive can be selected and the application deployed. After the deployment, the governance process enables basic operations such as start, stop and undeploy, and, in turn, the monitoring process helps developers to detect and check eventual problems through 2 basic metrics: response time and application health (HTTP status code). We think that new metrics should be supported in order to expand this set with a larger and more comprehensive set of cross-layer metrics. Finally, the migration feature of Cloud4SOA reveals to be a key point of the overall architecture enabling a straightforward bursting of resources between public clouds, or in this case, between a hybrid cloud environment.

As aforementioned, we performed some performance tests and collected several samples which allowed us to evaluate the overhead in time added by Cloud4SOA in the several supported processes. We found that this metric depends on various factors such as the network activity, geo-location and on the specific internals of each PaaS provider. From the achieved results, Cloud4SOA demonstrates an acceptable overhead for the deployment process considering the added value that this feature provides for developers. We found for CloudFoundry provider an overhead of 14 seconds which represents 50%, and for CloudBees, an overhead of 25 seconds which represents almost 20%. These values can seem high but since Cloud4SOA is not a time-critical system, the user experience will not be deteriorated. In the remaining operations the overhead is much higher exceeding 100% and in some cases 200%, such as, the database creation process. For these cases, the developer will notice such discrepancy.

In sum, the hybrid scenario was considered successful since Cloud4SOA allowed us to distribute our context-aware applications across heterogeneous PaaS, one public and another private. The hybrid cloud advantages were very present through the experiments, especially in the monitoring and migration (bursting) processes.
Chapter 13

Conclusions

This part of the project has explored the possibility to manage a hybrid cloud set-up using Cloud4SOA. To this end a private PaaS, namely CloudFoundry, has been installed in the Atos infrastructure and it has been integrated with Cloud4SOA through the implementation of an adapter. Ultimately, the vision of Cloud4SOA is to allow platform owners themselves to integrate their platform in an easy way, thus creating a user-driven expansion of the Cloud4SOA ecosystem. In this direction, the realization of this work has been very useful to evaluate the experience of performing such an integration process. We have found that the efforts being spent within the project are well-directed to achieve this goal; however, the experience is still not very positive and requires a fair amount of intervention. It’s worth noting that the experience has been much more positive with respect to the remote part of adapter than with the local one. As discussed in section 11.3.4, the remote part of the adapter has been implemented as a brand new independent component and has not required any additional wiring code in the Governance layer. Moreover, Cloud4SOA provides a Generic Remote Adapter template that greatly helps in implementing the adapter. With respect to the local part, some of the key insights that we have gained are listed below:

- The implementation of the local adapter requires digging into the core code of Cloud4SOA. This requires a well understanding of the Cloud4SOA architecture and code, especially on the Governance layer. Moreover it supposes a threat for the integrity of the platform since a change in the code of this layer may affect the functioning of the whole platform.

- Currently, in order to fully integrate the adapter with Cloud4SOA, some intervention of the Cloud4SOA project team is required to create the semantic model and include it into the database through the Semantic Initializer. This, however, will be mitigated by providing a way for users to create PaaS instances thorough the Cloud4SOA dashboard.

- The code in the governance layer requires some refactoring in order to clean it up and to make it more understandable. Currently there are some methods of direct interest to adapter implementators which are obsolete or with parameters that are not used.

The work has also been useful to evaluate the installation of CloudFoundry. We found that the installation process is fairly easy; however there is little documentation which makes resolving issues more difficult. In general we think that all the CloudFoundry documentation could be greatly improved, specially the API documentation which currently is very incomplete. We have needed to dig directly into the CloudFoundry source code in order to get the necessary
knowledge to perform the integration and to get involved in the community forums to solve some problems we encountered.
Part III

Exploring interoperability at the IaaS layer
Chapter 14

Introduction

This part of the project focuses on interoperability at the Infrastructure-as-a-Service (IaaS) layer. Infrastructure-as-a-Service is the main enabler component of Cloud Computing. It was the first form of Cloud to appear in the market with the release and spread in popularity of Amazon EC2 and, since then, it has been thoroughly used to enable other sorts of higher level Cloud paradigms [JTR+12]. Today, we can find a large and heterogeneous set of IaaS providers both public such as Amazon, Rackspace, GoGrid, Joyent or Bluelock; and private such as OpenShift, OpenNebula, Microsoft VMM SSP 2 or VMWare vCloud.

The IaaS paradigm is an evolution of Grid Computing, in which big data centers were used to offer utility computing resources to customers at a reasonable cost. Grid computing, however, suffered from several problems such as security issues and the lack of isolation between different customers [Jin05]. The key enabler of the transition to Cloud Computing has been the advances in hardware virtualization technologies. By making use of virtualization, a data center comprised by hundreds or thousands of physical servers can be abstracted into a single enormous pool of computing resources on top of which virtual machines can be started and delivered to consumers. This model permits a finer-grained personalization of the resources delivered to customers (virtual machines can be started in a variety of forms and configurations) and provides isolation for different customers.

The OPTIMIS European research project seeks to optimize IaaS cloud services by providing an architectural framework and a development toolkit. The optimization covers the full cloud service lifecycle, from construction to deployment and operation. OPTIMIS gives service providers the capability to easily build and orchestrate services, run legacy applications on the cloud and make informed deployment decisions based on their preference regarding Trust, Risk, Eco-efficiency and Cost (TREC). The OPTIMIS toolkit is formed by several tools grouped within three categories:

**OPTIMIS Base Toolkit**  A set of tools that provides functionalities common to all components that help make optimal service deployment and operation decisions, and also provides fundamental services such as monitoring and security

**OPTIMIS Service Provider (SP) Tools**  A set of tools that enable service providers to implement, package, deploy and operate services with assessed and guaranteed TREC levels.

**OPTIMIS Infrastructure Provider (IP) Tools**  A set of tools that enable infrastructure providers to effectively and efficiently manage the infrastructure (VMs, servers, data storage, etc.) required to operate services.
IaaS providers can make use of the IaaS self-management tools provided by OPTIMIS by installing them in a data center. The tools can be used to monitor and optimize the services running on the infrastructure, where a service is some sort of application running on a set of virtual machines. By monitoring the physical infrastructure on top of which these VMs run, OPTIMIS can detect problematic issues such as a too large error-rate, too much energy consumption or excessive cost and take the necessary actions to resolve them. One of the key points to resolve these issues is the formation of a network of interconnected OPTIMIS-enabled providers in which OPTIMIS can rely to delegate the execution of services.

Infrastructure and Service Providers can adopt OPTIMIS in several ways, depending on their needs. OPTIMIS currently contemplates 3 adoption models:

**Full OPTIMIS** In this type of cloud, OPTIMIS manages the virtual resources and provides monitored resource data, historical performance data and cost parameters to external parties.

**OPTIMIS Compliant (non-OPTIMIS)** These are non-OPTIMIS providers that don’t incorporate OPTIMIS tools but allow external OPTIMIS providers to rent resource from their platform in a standard way using a standard API.

**OPTIMIS Enhanced** Anything in between the above types is possible. A provider can incorporate only the OPTIMIS tools that it needs. For instance, it could only advertise the single TREC parameter which is currently capable of measuring.

![Figure 14.1: The OPTIMIS testbed is formed by 6 OPTIMIS-enabled clouds across 4 different IPs](image)

Currently the OPTIMIS eco-system is formed by 6 heterogeneous OPTIMIS enabled clouds (see figure 14.1) using different adoption models. The “Exploring interoperability at the IaaS layer” part of this project will extend and enhance this network in two ways. On one hand, the API provided by Arsys will be replaced by a standard interface based on the Open Cloud
Computing Interface (OCCI) and the components used to interact with it will be refactored to be able to consume this interface. This will leave the door open to the possibility of easily adding other providers that adopt this standard. On the other hand, Amazon Web Services (AWS) will be added as a new non-OPTIMIS provider that will be used in a similar way as Arsys. Having a public commercial cloud in the network supposes both a challenge, because its system is closed and its usage has to comply with their rules and standards, and a benefit because it is a highly reliable provider that can be used to host services with the assurance that it will deliver the expected performance.

The rest of this part is structured as follows. Chapters 15 and 16 provide some background that will help the reader understand the rest of the document. Next, chapter 17 provides an overview of the OPTIMIS requirements. Following, in chapter 18, the design of the new Arsys service is described. Chapter 19 describes the design and implementation of the OCCI compliant proxy. After that, chapter 20 describes the integration with AWS. Finally, chapter 21 concludes the part.
Chapter 15

OPTIMIS

OPTIMIS is a European research project started in 2010 and with 3 years duration. The goal of the project is to provide a framework that Infrastructure Providers (IPs) – an alternative name for IaaS providers - can use to enhance and optimize their physical infrastructure making it more reliable and trustworthy and, thus, more attractive to Service Providers (SPs) that intend to use the infrastructure provided by the IP.

OPTIMIS provides an architectural framework and development toolkit that covers the full service lifecycle: service construction, cloud deployment and operation. OPTIMIS is addressed to three main stakeholders: Infrastructure Providers (IPs), Service Providers (SPs) and Service Developers. It lets organizations easily provision resources on multi-cloud and federated cloud infrastructures and allows them to optimize the use of resources from multiple providers in a transparent, interoperable and architecture-independent fashion [ZJB+12]. In order to perform infrastructure optimization tasks, OPTIMIS introduces the TREC framework which stands for Trust, Risk, Eco-efficiency and Cost and describes the areas in which OPTIMIS actuates.

Each of the main stakeholders can use the OPTIMIS framework as follows:

**Infrastructure Providers** can manage services in compliance with SLAs made with SPs, while still using the infrastructure according to the IP’s own policies regarding trust, risk, eco-efficiency and cost. The IP may use only its own infrastructure or subcontract resources from other IPs in federation.

**Service Providers** can deploy and manage services on their own or external infrastructures provided by one or more IPs, possibly via a broker. By using OPTIMIS, they can compare and select the provider that best fits its requirements based on TREC parameters.

**Developers** can easily orchestrate cloud services from scratch and run legacy applications on the cloud.

The project provides a software toolkit that IPs can deploy in their data centers. The toolkit stands on top of a Virtual Infrastructure Manager (VIM) and along orchestration and application lifecycle management platforms. The toolkit can be broken down into three main components: The **OPTIMIS Base toolkit** with functionalities common to all components, the **OPTIMIS SP Tools** that enable service providers to implement, package, deploy and operate services, and finally the **OPTIMIS IP Tools** with functionalities to manage the infrastructure (VMs, servers, storage devices, etc.) required to operate services.

The **OPTIMIS Base toolkit** provides the following services:
• The **Monitoring** service gathers information about the physical hardware, virtualized hardware, energy consumption, performance and reliability. This information is then analyzed by the TREC framework.

• The **Trust** service allows IPs and SPs to determine the reputation of each other.

• The **Risk** service enables IPs and SPs to define, assess and manage risk such as the likelihood of a failure and its impact on services operation.

• The **Eco-efficiency** service assesses and predicts metrics such as energy consumption and carbon emission levels in order to ensure compliance with environmental laws.

• The **Cost** service allows IPs and SPs to assess and predict costs for provisioning a service, in order to evaluate alternative options and configurations.

The **OPTIMIS SP Tools** contains these tools:

• The **Construction Tools**, like the Programming Model and IDE, allow the implementation of service and service compositions and the generation of service manifests in order to deploy these services.

• The **Deployment Tools**, like the Service Deployer and Service Optimizer, decide upon the best IPs in which to deploy based on TREC values and manage the deployment of a service, be it in a private or public cloud.

• The **Operation Tools**, like the Service Manager, keep track of all service runtime data and make it simple for the SP to implement service-specific management functions.

The **OPTIMIS IP Tools** contains these tools:

• The **Deployment tools**, like the Cloud Optimized and the SLA Manager, combine the monitoring and assessment tools in the Base Toolkit with other components in order to create a self-managed cloud driven by the IP’s requirements and ensure that deployed services meet their service-level objectives.

• The **Operation Tools**, like the Data Manager, the Elasticity Manager and the VM Manager, operate the infrastructure by starting/stopping virtual machines, triggering elasticity rules, etc.
Chapter 16

Interoperability at the IaaS layer

This chapter explains in more detail the current interoperability issues at the IaaS layer and briefly describes the most important standardization initiatives that have appeared with the purpose of solving these issues. Moreover, the OCCI standard is explained in further details since it is used extensively on this part of the project.

16.1 Interoperability problems at the IaaS level

Interoperability issues at the IaaS layer vary depending on the type of service consumed. Although there are some common issues, others vary depending on whether we are requesting computational power, storage, etc. The major interoperability problems at this layer appear both at their management tools, which expose proprietary interfaces, and at the Virtual Machines (VMs) format which is currently dependent on the hypervisor.

Currently, each provider exposes its own management tools and interfaces. Usually, providers expose a Web Service that can be accessed programmatically, but there exist numerous differences between these services, from the type of web service (RESTful or SOAP-based) to the concepts that model the interface. These interfaces are usually tight to the specific concepts managed by each provider, thus preventing them to be easily standardized and requiring a learning effort from the user before starting to use them.

At the virtualization level, IaaS providers usually offer hypervisor based virtual machines. Software and VM metadata is bundled together so that it can be easily deployed [CH09]. However, the way in which this bundle is created and its format depends on the hypervisor, thus preventing VM images in one particular format to be used on multiple providers. Storage services offered by IaaS providers also differ in nature, from simple key/value pairs data stores to policy enhanced file based stores [CH09]. Moreover, the set of features offered by each provider can differ significantly as well as the storage semantics. In summary, the IaaS cloud layer presents the following interoperability issues:

- Lack of common and standardized Cloud IaaS APIs.
- Lack of a common and standardized format to bundle Virtual Machines.
- Diversity in the nature of storage services offered, their features and their semantics.
16.2 Standards for IaaS interoperability

One of the main issues with Cloud Computing is interoperability between providers’ interfaces [DWC10]. A lot of players fight in the Cloud market to become the de facto standard by providing proprietary interfaces and APIs. This scenario clearly damages cloud interoperability and application migration.

In an effort to overcome this situation, several organizations have proposed standards which can be used by IaaS providers to implement standardized interfaces. However, still today, the major providers seem to be reluctant to adopt any standard and keep offering their proprietary interfaces and models. Some of the standards that have been developed for IaaS management are: The CIMI (Cloud Infrastructure Management Interface) standard developed by DMTF\(^1\), some standards developed by the OCC (Open Cloud Consortium)\(^2\), the Unified Cloud Computing developed by CCIF\(^3\), the Intercloud Interface developed by GICTF (Global Inter-Cloud Technology Forum)\(^4\) and the OCCI (Open Cloud Computing Interface) standard developed by OGF\(^5\).

Among all of these, one of the standards that is getting more attention is the OCCI. The following section gives an overview of it since it will be used extensively during this part of the project.

16.2.1 Open Cloud Computing Interface (Open Grid Forum - OGF)

OCCI was initially envisioned as a standard remote API for the management of IaaS based services, such as Virtual Infrastructure Managers, but recently has evolved into a much more flexible standard that covers the full Cloud spectrum, from IaaS to SaaS. The goal of OCCI is to enable the development of interoperable tools by exposing a standard interface allowing tools on an upper layer to be infrastructure-agnostic. The standard focuses on infrastructure resources management operations, such as CRUD (Create-Read-Update-Delete) operations on virtual machines.

The OCCI standard consist on a RESTful API that stands at the same level of the native APIs provided by an Infrastructure Provider, and gives access to the functionalities in the internals of it (see figure 16.1). It is important to remark that the OCCI protocol talks directly to the internals of an IP, not to its API, so in some sense, is a full replacement for the native API.

The OCCI specification consists on a Core Model, defining the core management functionality, a set of extensions to this core model and a set of “renderings” to interact with the core model and its extensions.

Core Model

The OCCI core model [OW11a] defines a representation system for resources which can be manipulated through a rendering. A fundamental feature of the model is that extensions to the core are visible to clients at run-time, so that a client can discover and understand the various Resources and Link-types supported by the implementation.

\(^1\)http://dmtf.org/standards/cloud  
\(^2\)http://openCloudconsortium.org/  
\(^3\)http://www.Cloudforum.org/  
\(^4\)http://www.gictf.jp/index_e.html  
\(^5\)http://occi-wg.org/
The main concept in the model is the Resource concept, used to represent, for instance, a virtual machine, a job in a job submission queue, etc. Resources at the same time have a Category and they can have Actions which represents invocable operations on an instance.

Extensions

Currently, the only available extension is the IaaS extension [OW11b], which offers a description on how to build an IaaS API following the standard. This extension introduces a set of new resources related to infrastructure concepts: compute (virtual machine), network and storage. The API allows managing these resources in a CRUD style, invoke actions on them, query the provider in order to find out available resources, etc.

Renderings

The concept of Rendering is used in the OCCI to describe a way in which an OCCI-compliant interface can be used. Currently, the only available rendering uses the RESTful protocol [OW11c]. This rendering simply consists on a scheme to define URLs associated to HTTP verbs to access the different actions offered by the API and to discover resources through Queries.
Chapter 17

Requirements Overview

The OPTIMS [ZJB+12] consortium has given a lot of importance to requirements from the start of the project. As they state: “The quality of requirements has a direct impact on the chances of success of a project”. In the first year of the project, an initial list of requirements was developed and was later reviewed and extended in both year 2 and year 3. This section gives an overview of the requirements gathering process and shows the subset of requirements that affect this project. Moreover, these requirements are specialized into a set of new finer-grained requirements that this project will address. The full documentation can be found in the OPTIMIS project website\(^1\).

17.1 Methodology

Several activities were conducted in order to gather both functional and non-functional requirements for the project. The three main activities that drove the process were:

- The elaboration of a full project glossary capturing the domain vocabulary.
- The use of a scenario-based approach to discover requirements of various stakeholders.
- The use of a goal-oriented methodology to structure those scenarios into goal trees.

Three different scenarios were identified and studied, as they cover all areas in which OPTIMIS results are applicable:

**Cloud Programming Model** This scenario demonstrates the OPTIMIS toolkit applicability for Service Providers, both for the construction of new services but also to enable existing legacy applications to take advantage of their deployment in the cloud.

**Cloud Bursting** This scenario demonstrates the OPTIMIS toolkit applicability for final users that have deployed services in the cloud and need to have them scale up and down automatically.

**Cloud Brokerage** This use case demonstrates the OPTIMIS toolkit applicability for Cloud Brokers or Aggregators and how OPTIMIS can integrate multiple clouds.

\(^1\)http://www.optimis-project.eu/sites/default/files/content-files/document/optimis-public-deliverable-d1113-requirements-analysis.pdf
These scenarios produced a set of functional requirements which were combined with a second list of requirements that was obtained by using a goal oriented methodology. This methodology is based on the progressive refinement of abstract and general objectives into more concrete goals and requirements. This process led to the identification of the main entities with their relationships and the main agents responsible for the satisfaction of the requirements.

The requirements gathered during the first phase of the project were reviewed and extended during the following years by gathering new requirements from OPTIMIS stakeholders and business partners, with special attention to the plans for new use cases and scenarios identified. Moreover, all identified requirements were put in SMART (Specific, Measurable, Attainable, Relevant and Time bound) format and were elicited by using structured templates in which all relevant information of a requirement could be clearly stated (use case mapping, rationale, status, dependencies, etc.).

17.2 Functional Requirements

The following main stakeholders were identified:

**End User / Consumer** Entity (person or process) who is actually using the service provided by the Service Provider (SP).

**Customer** Organization who makes the contract with the SP, in order to get a service (or application).

**Service Provider (SP)** The provider of a service/application to the Customer.

**Infrastructure Provider (IP)** The provider of the infrastructure resources, also known as an IaaS provider. It provides easy access to computing, storage and networking resources, according to the requests from the SP to deploy and execute his application.

**Application/Service developer** Developer of an application/service which will be deployed and executed within the IPs resources. This stakeholder is responsible to make the best use of the OPTIMIS tools, in order to take the maximum advantage of the OPTIMIS-enabled environment.

**Broker** Sits between the IP and the SP. In addition to providing the abstraction layer of the individual IPs and their heterogeneous interfaces to the SPs, it provides service governance, matching engine, aggregation and intermediation.

The requirements have been grouped by functional area in the following categories:

**Service Management** Provides the capability to deploy and govern services across the IPs in the OPTIMIS network. It also handles bursting operations when they are needed.

**Data Management** Provides the capability to transfer service images to and between IPs.

**Monitoring** Provides the capability to monitor services running on IPs and to detect SLA violations.
<table>
<thead>
<tr>
<th>Component</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service Management</strong></td>
<td>F1 Be able to deploy an application according to the related service manifest, and be able to unde‌ploy it.</td>
</tr>
<tr>
<td></td>
<td>F2 To be able to handle a federated scenario by allowing a service to be deployed across multiple IPs. The service should be constructed in a manner that it can be split across multiple targets, deployment should be configured to handle this functionality and runtime services should ensure consistency.</td>
</tr>
<tr>
<td></td>
<td>F3 Be able to govern services deployed on OPTIMIS-enabled IP.</td>
</tr>
<tr>
<td><strong>Data Management</strong></td>
<td>F4 Be able to automatically deploy and utilize resources on federated providers. The deployment will be based on pre-existing storage VM templates on the providers. The new VMs will be dynamically deployed and contextualized in order to join the existing service</td>
</tr>
<tr>
<td></td>
<td>F5 Possibility to migrate VMs from one node to another, in order to redistribute the resources at runtime in function of the changes in the workload patterns.</td>
</tr>
<tr>
<td><strong>Monitoring</strong></td>
<td>F6 Guarantee interoperability of the Monitoring Infrastructure. Collect VM-related monitoring data from cloud middleware recommended by the interoperability task force.</td>
</tr>
</tbody>
</table>

Table 17.1: OPTIMIS Functional requirements
Table 17.1 shows the list of functional requirements that affect this project the most. Taking this list as a starting point, we have refined the requirements into a new list of finer-grained requirements specific to this project. Moreover, we have prioritized these requirements on a scale of 3 degrees (Low, Medium and High) according to the priority of these requirements as stated by the project consortium.

We have used the template described in section 11.1 of the “Exploring Interoperability at the PaaS layer” part of the project to describe the requirements. The following subsections describe the new requirements for each functional area.

17.2.1 Service Management

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>Originator</th>
<th>Description</th>
<th>Rationale</th>
<th>Fit Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>SM1</td>
<td>F1</td>
<td>The system must enable the deployment of services (from a service manifest) to any OCCI-compliant IP.</td>
<td>To be able to open the door to integrate OPTIMIS with any IP adopting the OCCI standard.</td>
<td>The system must be capable of deploying a standard service to Arsys through an OCCI-compliant communication channel.</td>
</tr>
<tr>
<td>SM2</td>
<td>F1</td>
<td>The system must enable the undeployment of services from any OCCI-compliant IP in which it has running services.</td>
<td>To be able to provide the necessary operations to manage services in OCCI-compliant IPs.</td>
<td>The system must be capable of undeploying any previously deployed service from Arsys using an OCCI-compliant communication channel.</td>
</tr>
<tr>
<td>SM3</td>
<td>F3</td>
<td>The system must enable the query and governance (start/stop) of services from any OCCI-compliant IP in which it has running services.</td>
<td>To be able to provide the necessary operations to manage services in OCCI-compliant IPs.</td>
<td>The system must be capable of querying, starting and stopping any deployed service on Arsys using an OCCI-compliant communication channel.</td>
</tr>
</tbody>
</table>
### Requirement SM4

**Description**
The system must enable the deployment of services (from a service manifest) to Amazon Web Services.

**Rationale**
To be able to add a robust, reliable and trustworthy IP to the OPTIMIS eco-system.

**Fit Criterion**
The system must be capable of deploying a standard service to AWS.

---

### Requirement SM5

**Description**
The system must enable the undeployment of services from Amazon Web Services.

**Rationale**
To be able to provide the necessary operations to manage services running on AWS.

**Fit Criterion**
The system must be capable of undeploying any service running on AWS.

---

### Requirement SM6

**Description**
The system must enable the query and governance (start/stop) of services from AWS.

**Rationale**
To be able to provide the necessary operations to manage services running on AWS.

**Fit Criterion**
The system must be capable of querying, starting and stopping any service running on AWS.

---

### 17.2.2 Data Management

---

### Requirement DM1

**Description**
The system must enable the upload of service images to AWS, in a format compliant with it.

**Rationale**
To have a tighter integration with AWS that permits a more extensive usage of it.

**Fit Criterion**
The system must be capable of uploading a typical service image, of a size of 2GB approximately. A service deployed to AWS, then, must be able to use this image.
<table>
<thead>
<tr>
<th>Req. Id</th>
<th>Originator</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM2</td>
<td>F4</td>
<td>Medium</td>
<td>The system must enable to query and delete images previously uploaded to AWS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Rationale</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To have a tighter integration with AWS that permits a more extensive usage of it.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Fit Criterion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The system must be capable of querying and deleting any service image that has been previously uploaded to AWS.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>Originator</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DM3</td>
<td>F5</td>
<td>Low</td>
<td>The system must enable the automatic transfer of a service image to AWS as a result of a bursting operation. This transfer should not require any human intervention and should produce the same results as the manual upload.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Rationale</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To be able to use AWS to burst highly demanding or critical services.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Fit Criterion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The system must be able to choose AWS as the destination IP when performing a bursting operation and must be able to automatically transfer a typical service image, of a size of 2GB approximately, to it.</td>
</tr>
</tbody>
</table>

17.2.3 Monitoring

<table>
<thead>
<tr>
<th>Req. Id</th>
<th>Originator</th>
<th>Priority</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>F6</td>
<td>Medium</td>
<td>The system must enable the collection of the set of monitoring metrics defined by OPTIMIS from any OCCI-compliant service IP in which it has running services.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Rationale</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>To be able to provide the necessary operations to monitor services running on OCCI-compliant IPs and provide data from them to decide if it is necessary to burst.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Fit Criterion</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>The system must be able to collect at least the following metrics from Arsys in an OCCI-compliant way:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• CPU usage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Memory usage</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Incoming network traffic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Outgoing network traffic</td>
</tr>
</tbody>
</table>
The system must enable the collection of the set of metrics defined by OPTIMS from services running on AWS.

**Rationale**
To be able to provide the necessary operations to monitor services running on AWS and provide data from them to decide if it is necessary to burst.

**Fit Criterion**
The system must be able to collect at least the following metrics from AWS:

- CPU usage
- Memory usage
- Incoming network traffic
- Outgoing network traffic

### 17.2.4 Non-Functional Requirements

Non-functional requirements were identified and elicited using the ISO-9126 standard [ISO01]. Table 17.2 shows the final list of non-functional requirements relevant to the design of the adapter. In contrast with functional requirements, non-functional ones were prioritized by the consortium within 3 categories:

**High** A requirement that must be implemented in OPTIMIS.

**Medium** A requirement that should be implemented in OPTIMIS.

**Low** A requirement that could be implemented in OPTIMIS, but won’t be in its first version.

Requirements with Low priority are omitted since it was decided not to include them for the moment.

<table>
<thead>
<tr>
<th>Requirement type</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality (Security)</td>
<td>NF1 Communication between OPTIMIS components should be secured in terms of authentication, confidentiality, and integrity due to the distributed nature of these components.</td>
<td>Medium</td>
</tr>
<tr>
<td>Functionality (Interoperability)</td>
<td>NF2 Use of standards to integrate with IPs.</td>
<td>Top</td>
</tr>
<tr>
<td></td>
<td>NF3 Use a common API that supports provisioning/deployment/configuration and control across different Cloud resources.</td>
<td></td>
</tr>
</tbody>
</table>

Table 17.2: OPTIMIS Non-Functional requirements
As with functional requirements, we have specialized this high-level list into a new list of finer-grained requirements specific to this project, as depicted in table 17.3.

<table>
<thead>
<tr>
<th>Requirement type</th>
<th>Requirement</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Security)</td>
<td>NF1.1 Communication between OPTIMIS and the new integrated IPs (AWS and, in general, any OCCI-compliant IP) should use a strong authentication mechanism.</td>
<td>Medium</td>
</tr>
<tr>
<td></td>
<td>NF1.2 When uploading images to AWS, its integrity should be checked at the end of the transfer.</td>
<td></td>
</tr>
<tr>
<td><strong>Functionality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Interoperability)</td>
<td>NF2.1 The components that interact with OCCI-compliant IPs should ensure the complete adherence to the standard.</td>
<td>Top</td>
</tr>
<tr>
<td></td>
<td>NF3.1 The interface to access the Service Management capabilities should be common when managing services on AWS and in OCCI-compliant IPs.</td>
<td></td>
</tr>
</tbody>
</table>

Table 17.3: Specialized OPTIMIS non-functional requirements for this project
Chapter 18

Service Design

Currently, Arsys provides a set of 3 non-standard Web Services that act as an OPTIMIS compliant proxy of the underlying infrastructure. These services are implemented using the SOAP protocol, but one of them, the Virtual Machines Manager will be replaced by a RESTful service compliant with OCCI [OW11c].

In this project, the service interface will be designed along with the data model used to represent the communication parameters. However, the actual Web Service will be implemented in Arsys since it will form part of their proprietary cloud platform, and it falls out of the scope of this project.

The interface will expose all the necessary methods needed by OPTIMIS in an OCCI compliant way, which will be consumed by a proxy client from OPTIMIS. This chapter focuses on the design of the interface, while the next will show the design of the client.

18.1 Data Model

The Data Model is formed by two differentiated parts: the classification and discovery system and the entities representation system.

The classification system, formed by the Category, Kind and Mixin types is used to identify resources (virtual machines, storage devices, etc.) and to define which capabilities they have in terms of attributes and actions. The Kind type is only used to identify the type of resources in a self-discoverable way. For instance, a Compute resource (like a Virtual Machine) would be associated with the Compute Kind, which would have a set of attributes (memory, cpu, etc.) and a set of actions (start, stop, suspend, etc.). In some sense, a Kind instance characterizes a type of resources. By using this structure, all resources of a given type can be easily discovered. By querying the Compute Kind instance, for example, we can “discover” all available Virtual Machines (Compute Resources). The Mixin type complements the Kind type allowing to “extend” resources capabilities by adding new attributes and actions. For instance, we could have an Operating System mixin that adds an attribute to specify the OS running on a VM and an action to reboot the OS. Finally, the Category type simply groups these two classes and is also used to classify Actions.

The entities representation system, formed by the Entity, Resource, Link and its subtypes is used to represent manipulable resources such as Virtual Machines. Each Resource type is associated with a unique Kind instance, for example, the Compute resources is associated with
the *Compute Kind*, the *Link resource with the Link Kind*, etc. A more detailed explanation of all types is given in the following sections.

Diagram 18.1 shows the data model that will be used to represent the interchanged data between OPTIMIS and the service, along with table 18.1 that defines its integrity restrictions. In order to better understand the model and restrictions, some definitions are given in the Definitions section.

Diagram 18.1: OCCI Data model extended with the custom OptimisCompute type. The green classes form the classification and discovery system while the blue classes pertain to the entities representation system.

This model is a union of the OCCI Core model [OW11a] and the infrastructure extension model [OW11b], along with the custom *OptimisCompute* type.
An Action cannot be associated with a Kind and with a Mixin at the same time.

There cannot be two instances of Kind with the same entity_type value.

There cannot be two instances of Category with the same value obtained from the concatenation of the scheme value and the term value.

There cannot be two Entities with the same id value.

A resource instance has to be associated with the Kind instance whose entity_type value is its class.

Given that a Kind instance (K1) has E1 as its entity type value and another Kind instance (K2) has E2 as its entity type value, K1 can only be related to K2 if E2 is the parent class of E1.

The attributes attribute of Kind must contain the names of all the attributes of its entity_type entity.

Table 18.1: Integrity Restrictions of the OCCI Data Model

### 18.1.1 Definitions

In order to better understand the model and its restrictions it is important to give some definitions:

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>resource instance</td>
<td>An instance on any Entity subclass: a Resource, a Link, a subclass of Resource or a subclass of Link</td>
</tr>
<tr>
<td>OCCI base type</td>
<td>The OCCI base types are Entity, Resource, Link and Action.</td>
</tr>
</tbody>
</table>

### 18.1.2 Classification System

The classification system is mainly formed by Kind and Mixin types and it has two main features:

**Discoverability** Through the navigation of Kind and Mixin instances, all available resources and actions can be discovered at run-time.

**Extensibility** The model can be easily extended in such a way that the extensions will be discoverable at run-time. Extensions can be done at design time by sub-typing Resource or Link or at run-time by creating new instances of Mixin.

Each OCCI base type, including sub-types of them, is assigned a unique Kind instance whose entity type value is its class (see IR5); moreover, this Kind instance will be related to the Kind instance of the parent class, if any. This structure permits the resource hierarchy to be discovered at run-time by navigating associations between Kind instances.
An example of this structure is given in diagram 18.2. In the figure, the initial Kind objects are shown in an implementation-like way, where the \textit{related} association between two Kinds and the \textit{actions} association have been materialized as attributes of Kind. The arrows between kind instances are merely informative and point always to the related kind. Note that only specialized resources (compute, network, etc.) and links have \textit{Related} and \textit{Actions}. An important thing to remark in the diagram is that the structure of \textit{related} kinds resembles the entity type hierarchy.

Additionally, any number of Mixin instances can be associated to a resource, providing it with additional capabilities in terms of attributes and actions. Mixins can be also used to \textit{tag} resources, by assigning a mixin with no extra capabilities.

Diagram 18.3 shows an example of a \textit{Compute} object associated with a Mixin object. The mixin object provides two additional attributes to the compute instance: \textit{os\_name} and \textit{os\_version}. The compute instance could be associated with other mixins as well.

The following sections describe each type of the three classes in the classification system in detail.

Diagram 18.2: Example of the initial object structure of kinds at run-time. Note that the structure of related kinds resembles the entities hierarchy.
Diagram 18.3: Example of an object of Mixin at run-time. The compute object c1 is associated with this mixin in order to be augmented with its attributes

**Category**

The Category type is used to classify entities and actions. The entities classifications is done through the Kind and Mixin sub-types, while actions classification is done by using instances of the Category type itself. A Category is identified by concatenating its scheme URI with its term e.g. http://example.com/category/scheme#term. This is done to enable discovery of Category definitions in text-based renderings such as HTTP. The attributes attribute defines the set of attributes that a resource of this category has.

When identifying Actions, title refers to the name of the Action and attributes to its parameters.

**Kind**

The Kind type identifies all Entity types present in the model, through the entity_type attribute. When a new resource is instantiated it must be associated with its type identifier to make itself discoverable. A Kind instance exposes the attributes and actions of the identified entity type through the attributes attribute.

Diagram 18.2 shows an object diagram representing the initial objects implicitly defined by the model. These objects have to be always present in the service.

**Mixin**

The Mixin type is used for two purposes: to tag resources and to add extra capabilities to a resource in form of attributes and actions. A tag is simply a Mixin with no attributes or actions and can be useful as an alternative form of classification for resources to indicate that a resource requires some special treatment, for instance a compute resource could be tagged as small to indicate that it should be configured with the minimum amount of resources. An example of a mixin that augments capabilities could be the OS Template mixin which adds an attribute to indicate the name of the OS.

Mixin instances can be related. In such a case, if mixin M1 is related to mixin M2, any resource instance associated with M1 will receive the capabilities (attributes and actions) defined both by M1 and M2.

**18.1.3 Entities Representation System**

The core of the entities representation system is the Resource type and includes also the Entity, Link and Action types. A Resource is anything that can be inspected and manipulated in the
underlying infrastructure of the IP such as a virtual machine (compute) or a disk (storage). In particular, there are four resource types: Compute, OPTIMIS Compute, Network and Storage. Resources can be connected through the Link type.

Resources and Links can have Actions through their Kind. Actions are basically operations which can be invoked on resources. The Action type itself is an abstract type and it must be sub-typed for any specific Action that a resource can have. The parameters of an Action are exposed through its attributes and, consequently, through the attributes attribute of the identifying Category. Figure 18.4 shows all the actions defined in the standard. More actions could be added to perform additional operations, for instance, it may be reasonable to add “resize” operations to other resources different from storage. A resize operation in the compute type could change its memory or CPU speed in order to make VMs elastic and to achieve better scalability.

The following sections describe each type of resource in detail and their actions.

Diagram 18.4: Actions defined in the OCCI standard. More actions can be added by providers to represent provider-specific operations

**Compute (Virtual Machine)**

This type represents an information processing resource, which in this project will only be a Virtual Machine. There are four operations which can be invoked on compute instances, all of them are used to change the state of the machine (see Figure 18.1) as well as to perform any operations derived from this change in state. Table 18.2 shows the resulting state for each operation.

In addition to the changes of state shown, a Virtual Machine can be deleted at any time, in which case its resources would be freed completely.
Figure 18.1: Compute type state diagram, where state changes correspond to actions

### OPTIMIS Compute

This is a custom type that augments the capabilities of the compute type in order to be compliant with the OPTIMIS requirements. Specifically it adds to attributes:

- **service_id**: Defines the service running on a VM.
- **image**: Defines the image of the disk that will be attached to the VM.

When OPTIMIS deploys a new VM it specifies which service (job, task, program, etc.) will run on it. There can be several VMs running the same service. Moreover, in some cases, OPTIMIS needs to attach disks to these VMs.

### Network

This type represents a low level networking resource, such as a virtual switch. There are two operations which can be invoked on network instances which are used to perform state changes (see figure 18.2). Table 18.3 shows the resulting state for each operation.

Additionally, the IPNetworking mixin is included in order to extend network resources with IP related capabilities. The IPNetworking mixin is assigned the scheme of “http://schemas.ogf.org/occi/infrastructure/network#” and the term “ipnetwork”. Its attributes are described in table 18.4.

With the combination of the Network resource and the IPNetworking mixin, the location of a VM within a network can be fully represented.
Figure 18.2: Network type state diagram, where state changes correspond to actions

<table>
<thead>
<tr>
<th>Action</th>
<th>Target State</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>active</td>
</tr>
<tr>
<td>stop</td>
<td>inactive</td>
</tr>
</tbody>
</table>

Table 18.3: Changes of state performed by Network actions

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>address</td>
<td>IPv4 or IPv6 address range</td>
<td>IP Network Address (e.g. 192.168.0.1/24)</td>
</tr>
<tr>
<td>gateway</td>
<td>IPv4 or IPv6 address</td>
<td>IP Network Address gateway (e.g. 192.168.0.1)</td>
</tr>
<tr>
<td>allocation</td>
<td>dynamic, static</td>
<td>Address allocation mechanism: dynamic uses DHCP while static uses supplied static network configurations.</td>
</tr>
</tbody>
</table>

Table 18.4: IPNetworkingMixin attributes

**Storage**

This type represents a resource that records information to a data storage device. There are six operations invocable on storage resources which are used to perform state changes (see figure 18.3). Table 18.5 shows the resulting state for each operation.
Network Interface

This link sub-type represents a network client device, such as a network adapter, that is, it’s a link between a compute resource and a network resource. It has the same operations and states that the Network resource.

In a similar way to the Network resource, this link type is extended through the IPNetwork-Interface mixin which provides IP capabilities to a network interface. It has the same attributes as the IPNetwork mixin.

Storage Link

This link sub-type represents a connection between a Resource (usually a Compute resource) and a Storage device. Figure 18.4 illustrates the state diagram of a StorageLink instance. It has the same operations and states than the Network resource.

Monitoring Mixin

Currently, the OCCI-WG is working on a draft\(^1\) for a Monitoring extension for OCCI. Despite that the extension is still very incomplete it already provides some guidelines on how such an

---

extension should be done.

The extension defines a generic *metric* mixin object with the attributes shown in table 18.6.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>Metric</td>
</tr>
<tr>
<td>scheme</td>
<td><a href="http://schemas.ogf.org/occi/monitoring#">http://schemas.ogf.org/occi/monitoring#</a></td>
</tr>
<tr>
<td>title</td>
<td>A metric mixin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>String</td>
<td>1</td>
<td>Value of the metric being monitored</td>
</tr>
<tr>
<td>timestamp</td>
<td>String</td>
<td>1</td>
<td>Time of the last calculation on the metric</td>
</tr>
<tr>
<td>samplerate</td>
<td>Float</td>
<td>0..1</td>
<td>Rate at which the metric is calculated</td>
</tr>
<tr>
<td>resolution</td>
<td>String</td>
<td>0..1</td>
<td>Indicates the multiple or fraction of the unit²</td>
</tr>
</tbody>
</table>

Table 18.6: Monitoring mixin attributes

Having this mixin as a base, any number of OPTIMIS–specific metrics can be created in form of mixins related to it. This way, two mixins are defined:

- **used_cpu_metric**:
  - **Title**: cpu.used
  - **Term**: used
  - **Scheme**: [service_scheme_url]/metric/compute/cpu#
  - **Related**: metric mixin

- **used_mem_metric**:
  - **Title**: mem.used
  - **Term**: used
  - **Scheme**: [service_scheme_url]/metric/compute/mem#
  - **Related**: metric mixin

This document presents the design of how monitoring could be implemented following the OCCI standard. However, since the monitoring process in Arsys deals with some legacy applications and methods, it would be very costly to re-implement. Therefore, for the moment, the monitoring function will be achieved using the legacy web service.

### 18.2 Behavioral Model

The Arsys RESTful web service exposes a set of operations accessible through the GET, POST, PUT and DELETE HTTP verbs. The OCCI model defines a fairly big amount of operations,
since it is designed to fit in any platform, but the Arsys web service will only implement the operations required by OPTIMIS, leaving the door open for future extensions derived from changes in the OPTIMIS requirements. In particular, OCCI includes a set of query and discovery methods that can be used to discover and navigate through the providers’ resources and get information about the features and capabilities of the provider. These operations are not implemented since the only client accessing the service is OPTIMIS.

As said in the previous section, monitoring in the Arsys service will be done using the legacy service. However, this document still shows the design of an OCCI compliant implementation of the monitoring methods in order to have a clearer picture of how a complete OCCI compliant solution could be implemented.

The main resource that is handled by the web service is the OPTIMIS Compute resource which represents a Virtual Machine used by OPTIMIS. Virtual Machines are organized in a hierarchical way, where running services are the parents and VMs the children, and are accessible through the /{service_id}/{vm_name} URL (see figure 18.5 for an example).

![Figure 18.5: Example of the URL name-space hierarchy](image)

Table 18.7 shows the operations that will be implemented on the service. Below there is a description of each of the methods exposed.
<table>
<thead>
<tr>
<th>Name</th>
<th>Verb</th>
<th>Path</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create VM</td>
<td>POST</td>
<td>/optimis_compute/</td>
<td>Creates a new Virtual Machine according to the information provided in the request body.</td>
</tr>
<tr>
<td>Delete VM</td>
<td>DELETE</td>
<td>/vms/{service_id}/{vm_name}</td>
<td>Deletes the vm_name Virtual Machine running the service_id service.</td>
</tr>
<tr>
<td>Update VM</td>
<td>PUT/POST</td>
<td>/vms/{service_id}/{vm_name}</td>
<td>Updates the vm_name Virtual Machine’s attributes (memory, cpu speed, . . . ) running the service_id service with the information provided in the request body</td>
</tr>
<tr>
<td>Execute action</td>
<td>POST</td>
<td>/vms/{service_id}/{vm_name}-/?action={action_name}</td>
<td>Performs action on the vm_name Virtual Machine running the service_id service.</td>
</tr>
<tr>
<td>Get VM</td>
<td>GET</td>
<td>/vms/{service_id}/{vm_name}-/?monitoring=[0/1]</td>
<td>Get information about the vm_name Virtual Machine running the service_id service. Also retrieves its monitoring data if the monitoring parameter is set to 1.</td>
</tr>
<tr>
<td>Get Service VMs</td>
<td>GET</td>
<td>/vms/{service_id}-/?monitoring=[0/1]</td>
<td>Returns information (status, IP address) about the VMs running the service_id service. The monitoring parameter specifies whether monitoring data should also be retrieved.</td>
</tr>
<tr>
<td>Get all VMs</td>
<td>GET</td>
<td>/vms-/?monitoring=[0/1]</td>
<td>Returns information (status, IP address) about all OPTIMIS VMs. The monitoring parameter specifies whether monitoring data should also be retrieved.</td>
</tr>
<tr>
<td>Terminate Service</td>
<td>DELETE</td>
<td>/vms/{service_id}</td>
<td>Removes all VMs running the service_id service and frees every IP assigned to the VMs.</td>
</tr>
</tbody>
</table>

Table 18.7: Methods exposed by the service
18.2.1 Authentication

Authentication with the server will be done through the pass of credentials (username and password). Every call made to the web service will contain the username and the password encoded in the header of the request, specifically it will be passed in the standard header field “Authorization”\(^3\). The value passed in this field will be the string “[username]:[password]”, encoded as a base64 string in order to protect its confidentiality.

The following example illustrates the headers of a request made by the username: optimis with the password secret:

> Content-type: text/plain
> Accept: text/plain
> Authorization: bXl1c2VyOnNlY3J1dA==
> [...] 

18.2.2 Requests and Responses syntax

OCCI currently only supports the transfer of messages in plain text, using the text/plain and text/occi formats which define a language used to build and parse messages. However, the OCCI Working Group is also working on a specification of the messages in JSON\(^4\), using the application/occi+json format. We believe that sending the messages in a structured text format provides a more reliable and accurate communication. Moreover, it eases the parsing of incoming messages. Therefore, we will adopt this specification to format messages transferred between this service and the OPTIMIS client.

The structure of the messages is detailed in Appendix A. Another important consideration is the HTTP return codes that the service should return, see Appendix B for details.

18.2.3 Create Virtual Machine

This method starts a new Virtual Machine to be used by OPTIMIS, started from the image specified in the image parameter. Additionally, the VM starts running the service identified by service_id. Note that the upload of the image is out of scope of this method and has to be done by other means; this method assumes the existence and integrity of the image. This method is also responsible for properly allocating the VM in the network and providing it with an IP address.

A request to create a new Virtual Machine contains the rendering of an OPTIMIS compute instance. If the request completes successfully, the service will return the URL of the newly created VM in the Location header of the response, along with the 201 return code. Otherwise it will return the appropriate return code.

The following example illustrates the data involved in a request and a response when invoking this method:

> POST /optimis_compute/ HTTP/1.1
> Content-type: text/plain
> [...] 

\(^3\)http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html
\(^4\)http://www.ogf.org/pipermail/occi-wg/attachments/20120512/17a14882/attachment-0001.pdf
18.2.4 Delete Virtual Machine

This method deletes a virtual machine. As a side effect, the IP assigned to the VM is released. No data is needed on the response or the request. The response returns a 200 code if the request can be completed successfully or the appropriate return code otherwise.

18.2.5 Update Virtual Machine

This method updates the attributes of a virtual machine, such as its memory or its cpu speed, and its associated mixins. It’s important to remark that the actual modifications that can be done depend on the infrastructure, for instance it’s possible that the infrastructure does not allow to change the memory assigned to a Virtual Machine. The update can be done through two methods: partial update and full update. The partial update operation is invoked by using the POST verb. In this case, the request data that must be passed is similar to the create VM
method, but only specifying those mixins that need to be added or the attributes that need to be updated.

The full update operation is invoked using the PUT verb. In this case, the request data must contain the full representation of the instance, as in the create VM method, since the existing VM instance will be replace by the new one.

In both cases, the response returns a 200 return code if the update completes successfully, with no data in the body, or the appropriate response code otherwise.

### 18.2.6 Execute Action

This method is used to execute an action on a virtual machine, such as stop or suspend.

A request to execute an action contains the rendering of an `Attributes` object with values assigned to the action parameters.

The response returns a 200 return code if the action is executed properly or the appropriate return code otherwise. In particular, it will return 400 (Bad Request) code if the action is not available or does not exist.

The following example illustrates the form of a request and response of this type:

```plaintext
> POST /vms/s1/vm1?action=stop HTTP/1.1
> Content-type: text/plain
> [...] { "method": "poweroff" }
< HTTP/1.1 200 OK
< [...] 
```

### 18.2.7 Get Virtual Machine

This method is used to get information about a Virtual Machine. The `monitoring` parameter in the query part of the URL indicates if monitoring statistics should also be returned or not.

The information returned consists on the value of the attributes of the Virtual Machine, its IP address and the value of the monitored metrics, if applicable. Specifically, the response contains a rendering of an `OPTIMIS Compute` instance.

The response returns a 200 return code is the retrieval can be done properly or the appropriate code otherwise.

The following example illustrates the use of this method:

```plaintext
> GET /vms/s1/vm1 HTTP/1.1
> Content-type: text/plain
> [...] 
< HTTP/1.1 200 OK
< [...] 
```
18.2.8 Get Service Virtual Machines

This method is used to get information about all VMs that are running a particular service. This method returns a list of OPTIMIS Compute resource objects.

The response returns a 200 return code is the retrieval can be done properly or the appropriate code otherwise.

The following example illustrates the use of this method:

> GET /vms/s1 HTTP/1.1
> Content-type: text/plain
> [...] 

< HTTP/1.1 200 OK
< [...] 
{
  "resources": [ 

    
}
18.2.9 Get all Virtual Machines

In a similar way as the last two methods, this method returns information about all OPTIMIS-related VMs deployed in the IP.

This method uses a special message structure specific to OPTIMIS that eases the parsing of messages returned by it:

```
{
  "services": [  
    {
      "id": String,
      "resources": resources array
    },
    {
      "id": String,
      "resources": resources array
    }
  ]
}
```

18.2.10 Terminate Service

This method is used to delete all VMs that are running a particular service. As a side effect, all IP addresses associated to the VMs will be released.
No data is needed on the response of the request. The response returns a 200 code if the request can be completed successfully or the appropriate return code otherwise.
Chapter 19

Proxy Design

19.1 OPTIMIS reference architecture overview

In order to understand where the proxy is placed within the OPTIMIS platform, it is worth giving an overview of its architecture.

The architecture is formed by 3 toolsets: the basic toolkit, the IP tools and the SP tools (see diagram 19.1, where the components that will play a role in the project have been highlighted). Altogether, these sets of tools enable SPs and IPs to manage services (and infrastructure) throughout the whole service lifecycle (service construction, deployment and operation). The architecture has been designed following the modularity principle, where each component is as decoupled as possible from the others, thus allowing providers to only adopt a part of the OPTIMIS toolkit. Most of the core components are implemented as web services in order to enable a distributed platform deployment between SPs and IPs.

The Basic Toolkit is a set of fundamental tools for monitoring and assessing clouds services and infrastructures, as well as interconnecting these securely.

- The Monitoring infrastructure allows the runtime state of physical infrastructure and virtual infrastructure to be captured, stored, and analyzed.
- The TREC-factors provide a framework for SPs and IPs to take management decisions based on the following factors:
  - The Trust tools enable SPs and IPs to assess each other’s reputation prior to engaging in a business relationship.
  - The Risk management tools allow SPs and IPs to reason about certain aspects of service deployment and operation, the risk factors associated with these, and estimate the potential consequences.
  - The Eco-efficiency tools enable an IP to assess power consumption, carbon emissions, etc. in order to achieve certain ecological goals.
  - The Cost tools can be used to assess and predict the service operation costs, both from an SP and IP perspective.
- The Security Framework provides a set of access and credential management capabilities that facilitates the interconnection of multiple clouds, as well as services that operate across these.
The **Infrastructure Providers** tools allow an OPTIMIS IP to optimize provisioning of services according to its requirements.

- The **Admission Control** component is used to determine whether or not to accept a new service request, and thus handle the tradeoff between increased revenue and increased workload (with potential SLA violations as a consequence).

- The **CloudQoS** (CQoS) component is used to model, negotiate, and monitor SLAs between the IP and SPs whose services the IP runs. It is also responsible for interacting with the necessary components to initiate the deployment of a service.

- The **Data Manager** provides mechanisms to transfer VM images between SPs and IPs.

- The **VM Manager** handles the VM lifecycle (launches VMs, monitors VMs, shuts VMs down) as well as performs VM placement in order to maximize the utility of the IPs infrastructure.

- The **Elasticity Engine** has the purpose of adjusting the capacity (in terms of VMs) allocated to a service in order to meet SLAs upon rapid changes in workload of the VMs that constitute the services.

### Diagram 19.1: Component-level view of the OPTIMIS architecture
• The **Fault Tolerance Engine** provides a VM re-starting service, thus contributing to a self-healing cloud infrastructure.

• The **Cloud Optimizer** (CO) orchestrates the monitoring system, the IP-level TREC assessment tools, and the above listed management engines to create a self-managed cloud infrastructure driven by infrastructure requirements.

The **Service Providers** tools allow OPTIMIS SPs to create, deploy, and run services.

• The **OPTIMIS Programming Model** simplifies the construction of SOA applications, either single web services or orchestration of services.

• The **Image Creation Service** allows the construction of VM images that embed the applications developed with the programming model.

• The **Integrated Development Environment** provides a user-friendly way to implement and prepare services for deployment, using the programming model and image creation service.

• The **License Management** tools allow SPs to incorporate license-protected software in services, and also provide the runtime support needed to manage software licenses for services deployed in remote IPs.

• The **Service Deployment Optimizer** (SDO) coordinates the service deployment process; ranks and selects suitable IPs, negotiates SLAs for the service, prepares the VM images and transfers these, and ensures that the service is launched properly.

• The **VM Contextualization** tools provide a convenient mechanism to prepare VM images with information needed for services to self-contextualize once launched. This can be any data: propagation of network parameters, security credentials, etc.

• The **Service Manager** monitors the runtime state of deployed services, allowing the SP to keep track of its services and manage these, e.g. by migrating a service upon unacceptable behavior.

The components highlighted in diagram 19.1 (Monitoring Infrastructure, Data Manager and CloudQoS) form the point of contact with external IPs, along with the SDO that orchestrates the deployment of a service to an external IP, during service deployment and operation.

### 19.1.1 Service Deployment

In order to deploy a service, first the SDO parses the service manifest (a document describing the requirements of the service) and gathers information about all available IPs through the TREC tools. Based on the manifest and the TREC properties, the SDO ranks the found IPs and negotiates the deployment of the service through the CQoS component of the better ranked IPs. In the IP side, the Admission Control decides whether or not to give a deployment offer to the SP, a decision based on current workload and TREC factors. After an IP finally accepts an offer, the VM images of the service are contextualized using the VM Contextualization tool and uploaded using the Data Manager. On the IP side, the CQoS receives the agreement to create
and initiates the deployment of the service with the help of the CO that initiates its deployment and monitoring. Additionally, the CO initializes the Fault Tolerance Engine and the Elasticity Engine to perform self-management tasks. Diagram 19.2 shows a high-level sequence diagram of this process.

Diagram 19.2: High-level service deployment sequence diagram

19.1.2 Service Operation

Service Operation mainly occurs in the IP side, in which almost all its components are involved. During Service Operation, the monitoring manager continuously collects runtime data about the state of physical servers, VMs and relevant service-level metrics. This data is then feed into a set of assessment engines such as the TREC, the elasticity and the fault tolerance engines.
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAgreementTemplates()</td>
<td>Returns the supported agreement templates.</td>
</tr>
<tr>
<td>createAgreement(offer)</td>
<td>Creates a new agreement based on an offer. As part of the agreement creation, the service deployment is initiated.</td>
</tr>
<tr>
<td>initiateNegotiation(context)</td>
<td>Creates a new negotiation process that is used by the SP to exchange offers and counter offers with the IP.</td>
</tr>
<tr>
<td>negotiate(offers)</td>
<td>Method used to send a set of notification offers to the IP and receive a set of counteroffers.</td>
</tr>
<tr>
<td>terminate(reason)</td>
<td>Terminates a negotiation process with a given reason.</td>
</tr>
</tbody>
</table>

Table 19.1: Methods exposed by the CQoS interface

The latter two may decide to request new VMs to be started due to a failure or an increase in workload. This decision is delegated to the CO which requests additional VMs to the VM Manager. All this structural changes are always notified to the Service Manager in the SP side so that it can keep track of the status of the service.

19.2 External Providers Integration

One of the key points of OPTIMIS is that it can interact with multiple clouds in order to create different types of cloud networks: a federated cloud, a multi-cloud structure through a broker, a private cloud that rents resources from other clouds, etc. OPTIMIS can dynamically and at run-time make use of a particular external provider in order to automatically rent resources (VMs) from it. For this reason, the components that interact with external IPs to perform these operations need to be able to talk to a variety of clouds. This section describes how this integration is handled in each of the three OPTIMIS contact points: CQoS, Data Manager and Monitoring Infrastructure, and how this project will contribute to it.

19.2.1 CloudQoS Integration

The CloudQoS component is used for all operations regarding SLA negotiation between an SP and an IP which basically consists on the negotiation and creation of agreements. The CQoS component uses the WS-Agreement standard [OGF07] of the Open Grid Forum (OGF) since it is a well-accepted standard for creating and monitoring SLAs in distributed environments. The implementation of the WS-Agreement standard is done through the WSA4J framework which is a comprehensive set of tools that can be used to manage SLA templates, negotiate and create SLAs, monitor SLAs, etc.

In order to enable SPs to dynamically negotiate and create SLAs with IPs, the CQoS exposes a standard WS-Agreement based interface, as seen in table 19.1. With this structure, the SDO can be completely IP-agnostic and just invoke the methods defined in the standard interface regardless of the actual IP that receives it. When integrating a new non-OPTIMIS IP, a custom CQoS Proxy exposing this standard interface has to be implemented, encapsulating the necessary wiring to integrate with such provider.

1http://packcs-e0.scai.fraunhofer.de/wsag4j/
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>createUserRepository(provider, service)</td>
<td>Creates a new user repository where images can be stored for the given service.</td>
</tr>
<tr>
<td>uploadVMImage(provider, service, image)</td>
<td>Uploads a new image to the repository of the given service.</td>
</tr>
<tr>
<td>getVMImages(provider, service)</td>
<td>Returns the list of images stored in the given service repository.</td>
</tr>
<tr>
<td>downloadVMImage(provider, service, image)</td>
<td>Downloads a particular image from the given service repository.</td>
</tr>
<tr>
<td>deleteAccount(provider, service)</td>
<td>Deletes a service repository.</td>
</tr>
</tbody>
</table>

Table 19.2: Methods exposed by the Data Manager interface

From all the methods exposed by this interface, the createAgreement and terminate methods are the responsible for initiating and terminating service deployment, respectively, and are the ones that need to be re-implemented in order to refactor the current integration with the Arsys services into an OCCI-compliant implementation.

### 19.2.2 Data Manager Integration

The Data Manager is a very complex component that, among other things, is responsible for storing VM images used to run services. In a similar fashion to the CQoS, the Data Manager exposes a standard interface, so that the SDO can invoke it in an IP-agnostic way. When interacting with external providers, such as the case of Arsys, a proxy is implemented that exposes this standard interface, encapsulating the wiring with the provider. Table 19.2 shows the part of the interface that is related to the upload and download of images.

At the moment, this component will not be refactored since the types of operations that it performs, such as image upload, are out of the scope of the OCCI standard. Moreover, refactoring the component would require further changes in the structure of the Arsys cloud which is not seen feasible by this partner.

### 19.2.3 Monitoring Integration

The monitoring infrastructure performs tasks such as:

- Collection of monitoring data from different OPTIMIS sources: energy sensor, physical hosts, virtual machines and services running on VMs.
- Collection of monitoring data from non-OPTIMIS sources: physical hosts and virtual machines running on non-OPTIMIS clouds (such as Arsys).
- Delivery of data via a web service to other components (TREC, Elasticity Engine, etc.)

The monitoring process follows a complex procedure in which a set of collectors gather metrics measurements from a set of sources (sensors, measurements from non-OPTIMIS IPs, etc.). The collectors then feed these data into an aggregator which updates the monitoring DB accordingly.
One of these collectors is designed specifically to gather monitoring data from Arsys. However this component will not be refactored at the moment for 3 reasons:

- Monitoring operations are not officially supported by the OCCI standard.
- Given that the monitoring process follows a rather complex process, it would require a lot of effort to refactor it.
- Refactoring the process would require work from the Arsys part, which does not intend to do.

### 19.3 CloudQoS Proxy design

As said in the previous section, the only component that will be refactored into an OCCI-compliant component is the CQoS Proxy. The Proxy is, actually, a substitution for the CQoS component, with the difference that it runs in the SP side instead of the IP (see figure 19.1).

The proxy is structured in two layers:

- The **Interface Layer** provides the standard CQoS interface and its implementation. Methods from this layer invoke the necessary methods in the Operations layer to perform actions on the IP.
- The **Operations Layer** invokes the necessary methods from the web services of an IP to perform operations such as deploy service and terminate service in an OCCI compliant way.

![Diagram of the CQoS Proxy](image.png)

**Figure 19.1**: High-level view of the proxy CQoS proxy architecture and its interaction with OPTIMIS and with an IP.
19.3.1 Interface Layer Conceptual Model

The Interface Layer provides an abstract WSA4J based interface compliant with the CQoS component. As seen in diagram 19.3, the interface is divided in several components where each component handles an action. Below there is a brief description of each of the components. Further details can be found in the WSA4J documentation.

TemplateAction

This component exposes a method to retrieve the SLA template that the IP offers. The template defines a set of parameters and conditions that have to be met, such as the maximum number of CPUs dedicated to the service, maximum amount of memory, availability rate, pricing, etc. In the case of OPTIMIS, the template also contains an initial service manifest with the defined parameters filled out.

NegotiationAction

This component exposes a method to negotiate an SLA agreement with the IP. This method retrieves the service manifest parameter from the agreement and validates it to check that it fulfills all conditions. In case the manifest violates some conditions, the method returns a new offer where all incorrect parameters are set to values within the accepted limits.

CreateAgreementAction

This component exposes a method to materialize an agreement with the IP. The method first validates that the manifest in the agreement is valid (it fulfills all conditions) and then materializes it by deploying the service as specified in the manifest. In order to deploy the service, this component communicates with the Operations layer.

ManifestValidator

This component contains a method to validate that a manifest fulfills all conditions imposed by the IP, that is, it checks that all parameter values are within the required limits.

VMAgreementType

This component represents an Agreement between the SP and the IP. It is the type of object created by the createAgreement method. This class contains all information about the agreement, the most important being the service manifest. It also contains a method to terminate the agreement, which communicates with the Operations layer in order to undeploy the service.
19.3.2 Operations Layer Conceptual Model

The Operations layer provides a framework that enables the implementation of specific clients for non-OPTIMIS providers, such as the OCCI-compliant client that is used to operate the Arsys cloud. Diagram 19.4 shows the conceptual model of this layer and below there is a description of each component.

Diagram 19.4: Operations layer conceptual model

**VMMManagementClient**

This interface is an abstraction of a VM Management system and acts as the point of contact between the Interface layer and the Operations layer. It contains the set of operations that OPTIMIS needs to operate a non-OPTIMIS cloud, such as deploy a service or terminate a service. The interface is the core of the designed extension mechanism, as it allows having custom implementations for each specific non-OPTIMIS provider willing to join the OPTIMIS ecosystem.

**ServiceComponent, VMProperties, Service and Action**

These classes are used to model the data that is shared between the Interface layer and the Operations layer. **ServiceComponent** typically represents a VM, **VMProperties** represents a set of properties about a VM, **Action** represents an operation that has to be executed on a VM, such as reboot or stop and **Service** represents an aggregation of ServiceComponents.
The `VMProperties` type is a sub-type of `Map` which allows including any additional provider-specific property in key-value form.

**VMManagementClientFactory**

The factory component is used to create and set up provider specific client objects. The `createClient` method takes the type and URL of the client and creates and configures the appropriate object.

The factory has a `default_client` attribute to define the type of client that should be created if none is specified.

**OCCIClient and OCCI_RESTClient**

These components are used to operate non-OPTIMIS clouds that expose an OCCI compliant interface, such as Arsys.

The implementation of the interface methods use the `OCCI_RESTClient` component to perform the calls to the web service. This component contains a set of auxiliary methods to render the objects received in the parameters as OCCI-specific text (using the JSON notation) and to transform the received text into objects. More information about the methods can be found on the previous chapter.

**19.3.3 Interface Layer Behavioral Model**

This section shows the behavioral model of the methods in the main types of the Interface layer.

**Create Agreement**

Diagram 19.5 shows the sequence diagram of the `createAgreement` method. The method first creates an OPTIMIS service manifest with the agreement offer information (number of VMs, memory, cpu speed, etc.) and validates it to make sure that all the values are below the threshold imposed by the system. The method then creates a client of the appropriate type ("OCCI" in the case of this project) and deploys the service according to the VMs information in the manifest. Finally a new agreement is initiated and returned.
Diagram 19.5: Create Agreement Sequence Diagram

Diagram 19.6 shows the sequence diagram of the *negotiate* method. The method basically checks the current offer and rejects or accepts it depending on whether all values are below the threshold imposed by the system. In case the offer is rejected, the counter offer is updated with a different set of values below the thresholds.
Diagram 19.6: Negotiate Sequence Diagram

**Terminate**

Diagram 19.7 shows the sequence diagram of the *terminate* method, which basically creates a client of the appropriate type, terminates the service through it and finally checks that it has been properly terminated.
19.3.4 Operations Layer Behavioral Model

This section shows the behavioral model of the methods in the OCCIClient type, which is the main type in this layer. The diagrams also show the REST calls made to the web service in order to have the full picture.

Authentication

As explained in section 18.2.1, authentication is made by encoding the username and password and passing the encoded string in the Authorization header of the request. The string will be a base64 codification of the "[username]:[password]" string.

Deploy service

Diagram 19.8 shows the sequence diagram of the deployService method.

The renderCompute method converts a ServiceCompute object into an OCCI-compliant JSON representation of an OPTIMIS Compute object. The resulting JSON object looks like:

```json
{
   "kind": "[service_schemas_url]#optimis_compute",
   "attributes": {
      "occi": {
         "compute": {
            "speed": 2,
            "memory": 4,
            "cores": 2
         }
      }
   }
}
```

Diagram 19.8: Deploy Service Sequence Diagram

```
"optimis": {
  "occi": {
    "optimis_compute": {
      "service_id": "s1",
      "image": "fedora8image.vmdk"
    }
  }
},

"actions": [
  ...
],

"id": "vm1",
"title": "Optimis Compute resource"
```

**Query Service**

Diagram 19.9 shows the sequence diagram of the `queryServiceProperties` method.

The `extractVMPropeties` method takes a JSON object returned by the REST call and parses it in order to create a VMProperties object. This parsing uses the structures presented in Appendix A as a base to parse the JSON object.
19.4 Implementation

The implementation of the proxy has consisted on refactoring the existing proxy to transform it into an OCCI compliant implementation. The interface of the proxy has been respected in order to be compatible with the other OPTIMIS components that use it, namely the SDO, while the internals of the communication with IPs has been completely refactored.
The OPTIMIS eco-system follows a similar structure as the Cloud4SOA eco-system. Each component is implemented as an independent Maven Java project with dependencies on the other components that needs to use. By keeping each component in an independent project, the whole structure is easier to maintain and change. Moreover, this makes easier distributing the effort among the different partners.

19.4.1 Technologies involved

The proxy has been implemented in Java, since this has been the programming language adopted for the OPTIMIS project. Other technologies involved include Maven, Jersey and Jettison. The following sections describe these technologies, except Maven which has already been described in section 11.3.1

Jersey

Jersey\(^2\) is a Java library based on JAX-RS (see Section 11.3.1) that can be used to build both RESTful web services and clients for a RESTful service. In this project the library has been used for the latter purpose, to build a client for an OCCI RESTful service.

Jersey can be used to perform **POST**, **GET**, **PUT** and **DELETE** actions on a WebResource, where a WebResource represents an entity, such as a virtual machine or a storage device, accessible through a URL. The library allows handling both the outgoing and incoming headers and the sent and received content in an easy way.

Jettison

Jettison\(^3\) is a Java library that supports the creation and parsing of JSON. In this project the library is used both to generate the messages sent to the OCCI service and to parse the messages received from it. Jettison manages JSON objects with the JSONObject class which is basically a map with string keys and any object such as a string, a number or another JSONObject as values.

19.4.2 Example

The following code snippet shows the implementation of the createVM method:

```java
public String createVM(String service_id, ServiceComponent sc, int index) throws ServiceInstantiationException {
    // Get an OPTIMIS Compute Resource
    WebResource r = client.resource(base_url + "/optimis_compute");
    try {
        // POST the JSON rendering of the Service Component
        // to the resource and capture the response
        ClientResponse res = r.type(MediaType.APPLICATION_JSON_TYPE)
            .post(ClientResponse.class, renderCompute(service_id, sc, index));
```

\(^2\)http://jersey.java.net/

\(^3\)http://jettison.codehaus.org/
The method first gets a pointer to the “/optimis_compute” web resource and posts to it a JSON object representing the ServiceComponent (the Virtual Machine). The method, then, determines whether the call has completed successfully, in which case it returns the value of the Location header, or not, in which case it throws an Exception.

19.5 Testing

As discussed in Section 11.4, testing is an important part of a complete software development process. In this part of the project we have followed an approach based on automatic unit testing as we did for the Cloud4SOA adapter. The OPTIMIS eco-system has a testing methodology similar to Cloud4SOA, in which a set of unit tests are automatically run each time a component is built. This ensures that changes made to a component do not compromise the integrity of the system.

19.5.1 Proxy Tests

A test has been designed for each of the main use cases. Each test brings the use case under certain situations and checks whether the outcomes coincide with the expected output (see table 19.3). Tests are independent from each other, since no order can be guaranteed when executing unit tests.

All the tests prepare the environment themselves, so they do not have any preconditions. For instance, the “Service Termination” test first deploys the service to ensure that it exists. Moreover, after each test all services that it has deployed are terminated to ensure that the following test will start with a clean environment.

Figure 19.2 shows the results of the tests execution.
<table>
<thead>
<tr>
<th>Use Case</th>
<th>Situation</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Service Deployment</strong></td>
<td>Deploy service with incorrect image id parameter.</td>
<td>Exception saying that the deployment has failed because the image does not exist.</td>
</tr>
<tr>
<td></td>
<td>Deploy service with correct parameters.</td>
<td>The service is deployed and all its instances are running.</td>
</tr>
<tr>
<td></td>
<td>Deploy already deployed service.</td>
<td>Exception saying that the service has already been deployed.</td>
</tr>
<tr>
<td></td>
<td>Deploy service with too much instances.</td>
<td>Exception saying that the number of instances to deploy exceeds the maximum.</td>
</tr>
<tr>
<td><strong>Service Query</strong></td>
<td>Query a non-existent service.</td>
<td>Exception saying that the service has not been deployed.</td>
</tr>
<tr>
<td></td>
<td>Query an existent service.</td>
<td>The service can be properly retrieved.</td>
</tr>
<tr>
<td><strong>Service Termination</strong></td>
<td>Terminate an existing service.</td>
<td>The service is terminated and none of its instances are running.</td>
</tr>
<tr>
<td></td>
<td>Terminate already terminated service.</td>
<td>Exception saying that all the instances of the service have already been terminated.</td>
</tr>
<tr>
<td></td>
<td>Terminate non-existent service.</td>
<td>Exception saying that the service does not exist.</td>
</tr>
</tbody>
</table>

Table 19.3: Description of the OCCI CQoS proxy unit tests

Figure 19.2: OCCI CQoS unit testing results
Chapter 20
Integration with Amazon EC2

As part of the work of extending OPTIMIS with support for more Infrastructure Providers, support for Amazon Web Services (AWS) has been added. The benefit of adding this IP to the OPTIMIS eco-system is twofold: on one hand it increases the pool of available IPs and on the other hand it provides the eco-system with a very robust and reliable public cloud.

Given that the Amazon cloud is public and using it has a cost that can be potentially high, its role will be to act as a last-option IP when there is no other IP available or that matches a service’s requirements. Since the Amazon cloud is a non-OPTIMIS IP, it can only be used for bursting purposes, that is, to temporarily rent additional resources when there is a need for elasticity or to replace crashed machines.

20.1 Integration Analysis

As seen in chapter 19, there are three components responsible for communicating with IPs: The CloudQoS, the Data Manager and the Monitoring Infrastructure. Each of these components has its own extension mechanisms to support additional IPs.

The following sections describe how each component is extended with support for Amazon EC2 and analyzes the Amazon API to see how it can be used to achieve this integration. Amazon provides two basic mechanisms for programmatically interacting with it: a REST API and a Java SDK which encapsulates the calls to the API in a reliable and easy-to-use way.

20.1.1 CloudQoS Integration

In order to integrate Amazon with the CQoS, a provider-specific proxy has to be created as it was done to integrate OCCI. In particular, as described in section 19.2, a component has to be created that implements the VMManagementClient interface. The following sections describe how each method in the interface can be integrated with the Amazon EC2 API.

Deploy Service

The EC2 API provides a method to run a new EC2 instance of a certain type and from a certain image. The type of the instance determines its hardware (CPU speed, RAM, network I/O, Disk space, etc.) and capabilities, from very small general-purpose instances with 615 MB of RAM and 1 virtual core at low speed to cluster instances with 60 GB of RAM and 16 high
performance virtual cores. The image is a snapshot of the software running in the instance, namely its operating system, applications and services.

Amazon provides two ways of launching an EC2 instance: from an Amazon Machine Image (AMI) created or selected from the Amazon catalog and from a previously imported VM from a Citrix Xen, Microsoft Hyper-V or VMware vSphere virtualization platform. The latter allows importing custom images (not tied to Amazon) and using them to run EC2 instances. To this end Amazon provides a set of tools that can convert these images to AMIs.

This method will select the type of instance to run from the attributes of the service to deploy. It will select the smallest type that fulfills all resource requirements in the agreement. Then it can use the `runInstances` method from the SDK with the selected type and the given image id.

Amazon also allows tagging most if its resources with custom key-value pairs, enabling to filter them based on these tags. After creating the instance, this method can use the `createTags` method from the SDK to add a `service_id` tag with the id of the service to deploy.

**Query Service**

The EC2 API provides a method to query (or describe, in Amazon terminology) the properties of an instance, which includes the following values, among others: instance id, IP address, status and hostname. The `describeInstances` method from the SDK can be used to obtain the properties of all instances of a given service by applying a tag filter on the `service_id` tag.

**Query All Services**

In a similar way, the properties of all instances from all services can be obtained by using the `describeInstances` method with no filters.

**Terminate Service**

The EC2 API provides a method to terminate an instance given its id. This method will first use the query service method (described above) to get the ids of all instances of the given service and then use the `terminateInstances` method from the SDK to shut down all the instances of the service.

### 20.1.2 Data Manager Integration

As part of the work of integrating AWS with OPTIMIS a fully functional Data Manager will be installed on the AWS environment, converting it into a somehow OPTIMIS-compliant IP. This will prove the feasibility of deploying functional OPTIMIS components into this platform and will open the door to the possibility of using the AWS IP to deploy services regularly rather than limiting its use to service bursting.

During service deployment, the DM will be used by the DM client to upload the service (and data) images needed by it. The DM, then, should store the images into an Amazon S3\(^1\) bucket to make them available to the EC2 instances running the service. This process, however, would

\(^1\)Amazon Simple Storage Service (S3) is an AWS service that can be used to store any amount of data of any sort. Data is stored in buckets. A bucket is a logical resource owned by an AWS account that can store an infinite amount of objects.
produce an unnecessary overhead since the same work would be done twice (first uploading the image to the DM and then sending it to S3). To overcome this we will develop, as part of this project, a component that will perform the upload directly to S3 from the DM client thus saving one step (see figure 20.1). This component will provide the basic operations for uploading, downloading and managing image files in S3. In particular it should fulfill the requirements described in table 20.1.

![Figure 20.1: High-level view of the interaction between the Data Manager and Amazon S3](image)

<table>
<thead>
<tr>
<th># Req</th>
<th>Requirement</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>R1</strong></td>
<td>Provide an upload method to upload image files for a given service. This method should be prepared to upload big files (of the order of GB). This method has to return the URL of the uploaded object.</td>
</tr>
<tr>
<td><strong>R2</strong></td>
<td>Provide a download method to download images files for a given service.</td>
</tr>
<tr>
<td><strong>R3</strong></td>
<td>Provide a method to list all uploaded images for a given service.</td>
</tr>
<tr>
<td><strong>R4</strong></td>
<td>Provide a method to delete an image file for a given service.</td>
</tr>
<tr>
<td><strong>R5</strong></td>
<td>Provide a mechanism to obtain the progress of the transfer operation (upload or download) in a percentage scale.</td>
</tr>
</tbody>
</table>

Table 20.1: S3 Wrapper requirements

The following sections describe the methods that will be provided and how the Amazon S3 API will be used.

**Upload Image**

The S3 API provides two methods to upload files to a bucket. The simplest method is intended for small files (less than 5GB) and consists on a synchronous regular file upload. The more complex method can be used to upload a file of any size (up to 5TB), although Amazon recommends using this method for all files larger than 100MB, and consists on a multi-part asynchronous upload where a file is divided into a set of smaller parts which are uploaded in
parallel and then joined again once all of them are ready. The second method, moreover, provides a mechanism to get the progress of the upload. In order to fulfill requirements R1 and R5 the second method will be used.

Within the S3 service, files are stored into buckets where a bucket is nothing more than a logical resource used to organize objects. Buckets are shared across all AWS users and, thus, it must be ensured that names used to create new buckets are unique. This method will store images into a bucket per service where the name of the bucket will be a concatenation of a certain string identifying OPTIMIS and the service id.

This method can use the `upload` method from the `TransferManager` class in the SDK to perform the multi-part upload and attach a `ProgressListener` to keep track of the progress of the upload. Then it can use the `generatePresignedURL` to generate a public URL that expires after a certain amount of time.

**Download Image**

The S3 API provides a method to download files from a bucket. The method is similar to the multi-part upload; it downloads a file asynchronously and provides a mechanism to track the progress of the download.

This method can use the `download` method from the `TransferManager` class in the SDK in a similar way to the upload image method.

**List Images**

The S3 API provides a method to list the properties (name, size, type of file, etc.) of all objects in a bucket. The `listObjects` method from the SDK accomplishes this goal.

**Delete Image**

The S3 API provides a method to delete an object from a bucket. The `deleteObject` method can be used for this.

20.1.3 Monitoring Infrastructure Integration

As described in chapter 19, the monitoring infrastructure has a fairly complex architecture formed by a set of heterogeneous components: Sensors, collector scripts, an Aggregator, a Message queue, a Database, etc. At a high level, the monitoring process consists on the collector scripts periodically collecting monitoring data from heterogeneous sources, such as sensors and non-OPTIMIS IPs APIs, and feeding it into the aggregator which then updates the monitoring DB accordingly. One of these collectors is designed specifically to gather monitoring data from Arsys, and another one will be designed to gather data from Amazon (see figure 20.2).

The AWS Collector will gather monitoring data from EC2 instances running OPTIMIS services through the provided API, and will generate and send structured messages containing these data. AWS offers the CloudWatch service to provide monitoring metrics and statistics about Amazon resources, such as EC2 instances.

The main concept in Amazon CloudWatch is the metric. A metric is characterized by a `name`, a `namespace` and a set of `dimensions`, and contains a time-oriented set of `data points`, where each data point has a value, a unit and a timestamp. A namespace is simply a conceptual
Figure 20.2: High-level view of the interaction between the monitoring components and AWS container for metrics. Metrics in different namespaces are isolated from each other. A dimension is similar to a tag in a resource, that is, a key/value pair that can be used both to identify metrics and to aggregate them.

Resources living in the Amazon eco-system are responsible for publishing metric data points periodically to the CloudWatch service. CloudWatch then can aggregate these data points and provide statistics such as averages, minimums and maximums across custom periods of time.

There are two types of metrics:

**AWS provided metrics** For each resource, Amazon automatically publishes a set of metrics at a 5-minutes interval. In the case of EC2 instances these metrics include: CPU utilization, disk read operations, network bytes transferred, etc. Moreover, Amazon automatically adds an “instance-id” dimension to these metrics in order to be able to easily retrieve statistics for a particular instance.

**Custom metrics** Any resource can publish custom metrics through the CloudWatch API. The resource is responsible for sending the data points periodically (at the desired rate).

A particular case in which custom metrics have to be used is to make memory-related measurements. Since the AWS provided metrics have to be OS-agnostic (they have to work regardless of the AMI), AWS cannot provide metrics such as memory use. In this particular case, given that monitoring memory consumption is critical for any serious monitoring application, Amazon provides a script that can be executed in a Linux or Windows EC2 instance to publish these data to custom memory-related metrics.

Table 20.2 shows the list of metrics that OPTIMIS monitors and whether they can be obtained from EC2 instances and how. Some metrics will be obtained from CloudWatch, either through AWS provided or custom metrics while others can be obtained directly from instances’ or AMIs’ attributes. Some of the metrics, such as “Total memory” and “CPU speed”, are apparently static in the sense that they do not change over time. These metrics, however, can actually
change if an instance is scaled. Also, OPTIMIS uses these metrics to compare the resources that were requested for a service to the actual resources provided by the IP.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Unit</th>
<th>Supported</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total memory</td>
<td>MB</td>
<td>Yes</td>
<td>Type of EC2 instance</td>
</tr>
<tr>
<td>CPU speed</td>
<td>MHz</td>
<td>Yes</td>
<td>Type of EC2 instance</td>
</tr>
<tr>
<td>OS release</td>
<td>-</td>
<td>Yes/No</td>
<td>Instance’s AMI</td>
</tr>
<tr>
<td>Machine type</td>
<td>-</td>
<td>Yes</td>
<td>Instance’s attributes</td>
</tr>
<tr>
<td>Disk total</td>
<td>GB</td>
<td>Yes</td>
<td>Instance’s attributes</td>
</tr>
<tr>
<td>CPU virtual cores</td>
<td>-</td>
<td>Yes</td>
<td>Type of EC2 instance</td>
</tr>
<tr>
<td>Memory used</td>
<td>%</td>
<td>Yes</td>
<td>Custom CloudWatch metric</td>
</tr>
<tr>
<td>CPU used</td>
<td>%</td>
<td>Yes</td>
<td>AWS provided CloudWatch metric</td>
</tr>
<tr>
<td>VM state</td>
<td>-</td>
<td>Yes</td>
<td>Instance’s attributes</td>
</tr>
<tr>
<td>Network bytes out</td>
<td>Bytes</td>
<td>Yes</td>
<td>AWS provided CloudWatch metric</td>
</tr>
<tr>
<td>Network bytes in</td>
<td>Bytes</td>
<td>Yes</td>
<td>AWS provided CloudWatch metric</td>
</tr>
</tbody>
</table>

Table 20.2: Metrics measured by OPTIMIS and their support in AWS

The “OS release” metric cannot be directly obtained from an AMI’s attributes since it only distinguishes between Windows and non-Windows platforms. This metric’s value, however, could be obtained by parsing the name of the AMI, given that AWS and community provided AMIs usually contain the name of the OS and its release in some part of the AMI name.

In order to be able to get the “memory used” metric, the custom script will have to be configured to execute periodically in each instance launched by OPTIMIS. This could be done in two ways:

- By setting it up in the service image, so that no additional tasks are required after launching a new instance.
- By manually setting it up after launching a new instance. In theory it could be done automatically by programmatically connecting to an instance and configuring it appropriately. However, this is a complex and error-prone process. Moreover, this option would not be feasible on Windows AMI instances, and, for Linux AMIs, the configuration steps would be different depending on the Linux distribution.

The collector has to generate and send an XML file containing measurements for the supported metrics for all instances running an OPTIMIS service. Table 20.3 describes the attributes of a measurement. The collector script will be called periodically, at a 2-minute interval, by the monitoring infrastructure.

The following sections describe how metrics from the different sources can be obtained.
**Attributes Description**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Physical resource ID</strong></td>
<td>An ID assigned to physical machines on top of which service VMs are executed. Since AWS does not provide information about its underlying infrastructure, all instances will be treated as pertaining to the same physical resource.</td>
</tr>
<tr>
<td><strong>Metric name</strong></td>
<td>The name of the metric being measured.</td>
</tr>
<tr>
<td><strong>Metric value</strong></td>
<td>The value of the measurement.</td>
</tr>
<tr>
<td><strong>Metric unit</strong></td>
<td>The unit used to interpret the measurement.</td>
</tr>
<tr>
<td><strong>Timestamp</strong></td>
<td>The time when the measurement was taken.</td>
</tr>
<tr>
<td><strong>Virtual resource ID</strong></td>
<td>The ID of the virtual machine being measured. In the AWS case, this is the instance id.</td>
</tr>
<tr>
<td><strong>Service ID</strong></td>
<td>The ID of the service that this instance is running.</td>
</tr>
<tr>
<td><strong>Resource type</strong></td>
<td>The type of resource being measured. At the moment only virtual resources are measured.</td>
</tr>
<tr>
<td><strong>Collector ID</strong></td>
<td>A custom ID assigned to the collector component.</td>
</tr>
</tbody>
</table>

Table 20.3: Attributes of an OPTIMIS measurement

**Metrics obtained from the type of EC2 instance**

The memory, CPU speed and number of virtual cores can be deduced from the type of instance. Amazon calculates CPU speed in terms of EC2 Compute Units (ECU), where 1 ECU is equivalent to 1.1 GHz approximately. For example, according to the AWS documentation, an instance of type `small` has 1 ECU (equivalent to 1.1 GHz), 1.7 GB of memory and 1 virtual core.

**Metrics obtained from the instance’s AMI**

The EC2 API provides a method to `describe` the attributes of an AMI, such as its platform (Windows or non-Windows), name, description etc. The `describeImages` method from the SDK can be used for this.

**Metrics obtained from the instance’s attributes**

These metrics can be obtained in a similar way to the `query service` method in the CQoS proxy.

**Metrics obtained from CloudWatch**

The CloudWatch API provides a method to get statistics for a metric across a given period. The period will be between the current time and 2 minutes ago. The `getMetricStatistics` method from the SDK can be used for this.
20.2 Design

This section describes the design of the components involved in the AWS integration with OPTIMIS.

20.2.1 CQoS Proxy

The design of the proxy for the CQoS follows the same principles as the Arsys CQoS proxy. It is structured in two layers (see section 19.3 for further details):

- The **interface layer** which provides a standard WSA4J interface. This layer follows the same structure as the corresponding layer in the Arsys proxy, with only small differences to adapt it to the new IP.

- The **operations layer** which talks to AWS in order to launch, query or terminate services.

**Operations Layer Conceptual Model**

This layer follows the extensibility mechanism provided by the CQoS by implementing a class that wraps the AWS SDK in order to provide VM management operations (launch, query and terminate instances). Diagram 20.1 shows the conceptual model of this layer. We have included a representation of the relevant part of the Amazon EC2 SDK.

The main class of this layer is the `AmazonClient` class which implements the `VMManagementClient` interface with the functionality necessary to interact with AWS. This class in turn uses the AWS SDK to perform the actual interaction with the AWS EC2 service. The `access_key` and `secret_key` attributes are set through the `setAuth` method and are used to authenticate the user in every request made to Amazon.

**EC2 SDK** The main entry point to the SDK is the `AmazonEC2` class which provides all the necessary methods to perform any operation exposed by the EC2 API. Each method follows the same pattern: it has a single `request` parameter whose type is a class that contains all the required and optional parameters used to invoke that method in the API and an optional analogous `result` parameter.

All the types presented in the diagram are self-explanatory, more information can be found in the Amazon EC2 API documentation\(^2\).

\(^2\)http://docs.aws.amazon.com/AWSEC2/latest/APIReference/Welcome.html
Diagram 20.1: AWS CQoS operations layer conceptual model
Operations Layer Behavioral Model

This section shows the behavioral model of the methods in the AmazonClient type, which is the main class in this layer.

Deploy Service  Diagram 20.2 shows the sequence diagram of the deployService method.

The method iterates over all the ServiceComponents that compose this service and creates a RunInstancesRequest for each of them that will start as many instances as specified by the component, of the selected type and with the specified image id. After the instances are created the method iterates over them and attaches a service_id tag to each one so that they can be identified later on.

The diagram of the getAmazonEC2Client is shown in diagram 20.3. It basically creates a credentials object and initiates a new EC2 client. The selectInstanceType method compares the attributes of a service component with the attributes of the different InstanceTypes provided by Amazon and chooses the smaller type that covers all of the component’s attributes.
deployService(id: String, comps: List(ServiceComponent))

loop

[ for sc in comps ]

sc : ServiceComponent

getNumInstances() : n_instances

getImageId() : image_id

selectInstanceType(sc) :

type

(image_id, n_instances, n_instances)

req : RunInstancesRequest

setInstanceType(type)

setDoMonitoring(true)

addSecurityGroup("optimis-security-group")

runInstances(req) : res

c2 : AmazonEC2

getInstances() : instances

instance : Instances

getId() : instance_id

(resources, tags)

createTags(req2)

tag : Tag

("service_id", id)

tags := {tag}

(resources, tags)

req2 : CreateTagsRequest

res : RunInstancesResult

Diagram 20.2: AWS CQoS Deploy Service Sequence Diagram
Diagram 20.3: AWS CQoS Get Amazon EC2 Client Sequence Diagram
Query Service  Diagram 20.4 shows the sequence diagram of the queryService method.

This method basically sets a filter to tag resources by the service_id tag and then creates a DescribeInstances request. Finally it iterates over the returned objects and fills out a set of ServiceProperties objects representing the properties of the different components of the service.

![Diagram 20.4: AWS CQoS Query Service Sequence Diagram](image)

Terminate Service  Diagram 20.5 shows the sequence diagram of the TerminateService method.

The method first queries for all the instances running the service and then creates a TerminateInstances request with all the returned instance ids.
20.2.2 Amazon S3 Wrapper

The S3 Wrapper is the component that will complement the Data Manager Client to upload image files directly to S3 without using the Data Manager as an intermediary.

S3 Wrapper Conceptual Model

Diagram 20.6 shows the conceptual model of the wrapper. We have included a representation of the relevant part of the Amazon S3 SDK.

The main class is the S3Wrapper class which provides a set of image management methods (upload, download, delete and list). This class in turn uses the AWS SDK to perform the actual interaction with the AWS S3 service. The acces_key and secret_key attributes are set through the constructor and are used to authenticate the user in every request made to Amazon. The progress attribute’s value is updated periodically to reflect the current progress of a transfer operation (either upload or download).

This class shares a TransferManager object across all methods, as recommended by the AWS developer guidelines. It also shares a Transfer object between the download/upload methods and the progress listeners.

---

3 "TransferManager is responsible for managing resources such as connections and threads; share a single instance of TransferManager whenever possible." – from the SDK API reference
S3 SDK  This part of the AWS SDK follows the same structure as the EC2 part. The main entry point is the `AmazonS3` class which exposes a set of methods that follow the request–result object pattern, apart from offering some other utility methods.

The `TransferManager` class is used to initiate upload and download operations. Both operations return a `Transfer` object that can be queried to determine the status of the transfer (progress, transfer status, etc.). It is important to remark that the upload and download methods run asynchronously, that is, the methods return immediately after invoking them and run the transfer operations in a separate thread (actually, it may dedicate more than one thread to it if it decides to perform a multi-part transfer).

More information about the classes presented in the diagram can be found in the S3 API documentation⁴.

⁴http://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html
S3 Wrapper Behavioral Model

This section shows the behavioral model of the methods in the S3Wrapper type.

**Constructor**  
Diagram 20.7 shows the sequence diagram of the class constructor method.

```
wrapper : S3Wrapper
(public_key, secret_key)
credentials : BasicAWSCredentials
(access_key, secret_key)
txmanager : TransferManager
(credentials)

this
.access_key := access_key
this
.secret_key := secret_key
```

Diagram 20.7: AWS S3 Wrapper Constructor Sequence Diagram

**Upload**  
Diagram 20.8 shows the sequence diagram of the upload method.

The method first checks whether the bucket associated to the service is already created or not, and creates it if necessary. It then creates a PutObject request and associates the file and a new progress listener to it, invokes the upload method and stores the returned transfer object so that the progress listener can query it periodically to update the progress attribute. Finally the method generates a temporary URL that will expire after 1 day and can be used to download the file within that time.

Bucket names for services are built by concatenating the string “optimis-“ with the id of the service.

Diagram 20.9 shows the sequence diagram of the progressChanged method of the TransferProgressListener class. The method updates the progress attribute of the S3Wrapper class and checks whether the transfer has completed or failed. When the transfer fails the progress attribute is set to -1 to indicate the failure.
Diagram 20.8: AWS S3 Wrapper Upload Sequence Diagram
Diagram 20.9: AWS S3 Wrapper Progress Changed Sequence Diagram
**Download**  Figure 20.10 shows sequence diagram of the download method.

The method performs a sequence of actions similar to the upload method, it first checks the existence of the bucket associated to the service and then invokes the *download* method to initiate the download.

![Diagram 20.10: AWS S3 Wrapper Download Sequence Diagram](image)

**List images**  Figure 20.11 shows the sequence diagram of the list images method.

The method first checks whether the service has a bucket associated and then creates a *ListObjects* request with the bucket name to get the key (name) of all the files in that bucket.
Diagram 20.11: AWS S3 Wrapper List Images Sequence Diagram
Delete image  Figure 20.12 shows the sequence diagram of the delete image method.

The method first checks the existence of a bucket associated to the service and then creates a `DeleteObject` request with the bucket name and the name of the image file to delete.

![Diagram 20.12: AWS S3 Wrapper Delete Image Sequence Diagram](image)

20.2.3 AWS Collector

The Collector is a stand-alone executable script that performs the following sequence of actions:

1. Get all EC2 instances running OPTIMIS services.
2. For each instance, get metric measurements from several AWS sources.
3. Create an XML file containing these measurements.
4. Send the XML file to the Aggregator.

This script is executed on the monitoring infrastructure at a 2-minute interval through the use of a task scheduler such as `cron`.

Collector Conceptual Model

Diagram 20.13 shows the conceptual model of the collector. We have included a representation of the relevant part of the Amazon CloudWatch and EC2 SDK.

The main class is the Collector which provides a set of methods to get different kind of measurements (obtained from the instance type, from the instance itself and from CloudWatch). This class in turn uses the EC2 SDK to get information about instances and the CloudWatch
SDK to get metric statistics. The script receives the access_key and secret_key attributes as parameters and uses them to authenticate the user in every request made to Amazon.

The Measurement class is used to model a measurement obtained from Amazon. Due to some Amazon limitations, the physical_resource_id and the resource_type attributes have static values. The former has a value formed by the concatenation of a name and a number (for instance, “optimis001”) while the latter’s value is “virtual”. The collector_id attribute will be received as an input parameter to the script.

The collected measurements are used to generate an XML file with the following structure:

```xml
<MonitoringResources>
  <monitoring_resource>
    <physical_resource_id>optimis001</physical_resource_id>
    <metric_name>cpu_speed</metric_name>
    <metric_value>1995</metric_value>
    <metric_unit>MHz</metric_unit>
    <metric_timestamp>1358331248</metric_timestamp>
    <service_resource_id>s1</service_resource_id>
    <virtual_resource_id>i-1ea16b8c4</virtual_resource_id>
    <resource_type>virtual</resource_type>
    <monitoring_information_collector_id>00007</monitoring_information_collector_id>
  </monitoring_resource>
  [...]
</MonitoringResources>
```

This part of the AWS SDK follows the same structure as the others. The main entry point is the AmazonCloudWatch class which exposes a set of methods that follow the request object – result object pattern. More information about the classes presented in the diagram can be found in the CloudWatch API documentation⁵.

---
⁵http://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/Welcome.html
Collector Behavioral Model

This section shows the behavioral model of the main methods in the Collector class.

Main Method   The main method is the method executed when running the script. Diagram 20.14 shows its sequence diagram.

The first thing that the method does is to get all EC2 instances that are running an OPTIMIS service by querying all instances that have a tag whose name is “service_id” regardless of its value. Then, for each retrieved instance, a set of attributes and metrics are obtained. We do not show how all the metrics are get in the diagram to keep it simpler, see table 20.4 for a description on how each metric is obtained.

Finally an XML document is generated with data about all the measurements and it is sent to the Aggregator for its processing.
Diagram 20.14: AWS Collector Main Method Sequence Diagram
<table>
<thead>
<tr>
<th>Metric</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total memory</td>
<td>Call to <code>getTotalMemory(type, instance_id, service_id)</code>. This method determines the amount of memory from the instance type and creates a new Measurement by calling to <code>Measurement(&quot;mem_total&quot;, value, &quot;MB&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>CPU speed</td>
<td>Call to <code>getCPUSpeed(type, instance_id, service_id)</code>. This method determines the cpu speed from the instance type and creates a new Measurement by calling to <code>Measurement(&quot;cpu_speed&quot;, value, &quot;MHz&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>OS release</td>
<td>Call to <code>getOSRelease(ami_id, instance_id, service_id)</code>. This method determines the OS Release from the image of the instance and creates a new Measurement by calling to <code>Measurement(&quot;os_release&quot;, value, &quot;,&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>Machine type</td>
<td>Create a new measurement by calling to <code>Measurement(&quot;machine_type&quot;, architecture, &quot;,&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>Disk total</td>
<td>Call to <code>getTotalDiskSize(volume_ids, instance_id, service_id)</code>. This method determines the total size of the disks attached to the instance and creates a new Measurement by calling to <code>Measurement(&quot;disk_total&quot;, value, &quot;GB&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>CPU virtual cores</td>
<td>Call to <code>getNumVCores(type, instance_id, service_id)</code>. This method determines the number of virtual cores from the instance type and creates a new Measurement by calling to <code>Measurement(&quot;cpu_vnum&quot;, value, &quot;,&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>Memory used</td>
<td>Call to <code>getCWMMeasurements(&quot;MemoryUsed&quot;, true, instance_id, service_id)</code>. This method uses the CloudWatch SDK to get the memory used metric and creates a new Measurement by calling to <code>Measurement(&quot;mem_used&quot;, value, &quot;,&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>CPU used</td>
<td>Call to <code>getCWMMeasurements(&quot;CPUUtilization&quot;, false, instance_id, service_id)</code>. This method uses the CloudWatch SDK to get the cpu utilization metric and creates a new Measurement by calling to <code>Measurement(&quot;cpu_used&quot;, value, &quot;,&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>VM state</td>
<td>Create a new measurement by calling to <code>Measurement(&quot;vm_status&quot;, state, &quot;,&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>Network bytes out</td>
<td>Call to <code>getCWMMeasurements(&quot;NetworkOut&quot;, false, instance_id, service_id)</code>. This method uses the CloudWatch SDK to get the bytes that have been sent out through the network and creates a new Measurement by calling to <code>Measurement(&quot;network_out&quot;, value, &quot;Bytes&quot;, now(), instance_id, service_id)</code></td>
</tr>
<tr>
<td>Network bytes in</td>
<td>Call to <code>getCWMMeasurements(&quot;NetworkIn&quot;, false, instance_id, service_id)</code>. This method uses the CloudWatch SDK to get the bytes that have been received through the network and creates a new Measurement by calling to <code>Measurement(&quot;network_in&quot;, value, &quot;Bytes&quot;, now(), instance_id, service_id)</code></td>
</tr>
</tbody>
</table>

Table 20.4: Description of the process to get each OPTIMIS metric
Get CloudWatch Measurement  Diagram 20.15 shows the sequence diagram of the \textit{getCWMeasurement} method. It first determines the namespace to use depending on whether the metric is custom or provided by AWS. In the case of metrics provided by AWS the namespace is always “AWS” while the custom metrics have a custom namespace such as “optimis”. It then performs the request to CloudWatch with the desired metric name, a dimension specifying the instance id, an interval of 2 minutes and stating that only the “average” statistic is needed. By specifying a period that is equal to the difference between the start time and end time we ensure that the request will only return one data point and, thus, we can obtain the average measurement directly from it.

Diagram 20.15: AWS Collector Get CloudWatch Measurement Sequence Diagram

Get OS Release  Diagram 20.16 shows the sequence diagram of the \textit{getOSRelease} method. The method simply makes a request to the EC2 API to describe the image with the given id and
then obtains the *platform* value from it. As discussed earlier in this section, the *platform* value only discriminates between Windows and non-Windows platforms. A more complex deduction of the OS could be done by parsing the *name* or *description* of the image.

### Diagram 20.16: AWS Collector Get OS Release Sequence Diagram

**Get Disk Size**  Figure 20.17 shows the sequence diagram of the *getTotalDiskSize* method. The method basically makes a request to the EC2 API to describe all the volumes attached to the instance and then calculates the sum of the size of all of them.
Diagram 20.17: AWS Collector Get Disk Size Sequence Diagram

20.3 Implementation

We have implemented 3 independent self-contained projects, one for each of the designed components. These components will be plugged into the OPTIMIS eco-system by the responsible of each major component (Data Manager, Monitoring Infrastructure and Cloud QoS) at the time of performing the full integration with Amazon.

The following sections describe the technologies involved and the implementation details of the S3 Wrapper and the Collector. The CQoS proxy follows the same principles as the OCCI CQoS proxy.

20.3.1 Technologies Involved

The three components have been implemented in Java in order to be compliant with the components with which they will be integrated. Other technologies include Maven, Commons CLI, Jersey and JAXB. The following sections describe these technologies, except Maven and Jersey that have already been described in sections 11.3 and 19.4 respectively.

Commons CLI

Commons CLI is a library from Apache designed to provide a mechanism to parse the arguments passed to a script. Arguments are defined by a tag, and can be of different types such as boolean arguments where only the presence of the tag is checked, or valued arguments where
the tag has to be followed by a value. For example we could define an argument “-c <value>” to specify the path to a configuration file.

The library automatically parses all the arguments passed to a script and provides an interface to query for their presence and value. The library also provides a mechanism to print help or usage messages in a structured format.

JAXB

JAXB is a library that works in conjunction with Jersey to provide support for XML in REST calls by annotating resources with XML-related annotations. These annotations are used by the JAXB engine to generate an XML document with the attributes’ values of a resource.

For example, the following annotated resource:

```java
@XmlElement (name="monitoring_resource")
public class Measurement {
    @XmlElement
    private String physical_resource_id;

    @XmlElement
    private String metric_name;

    @XmlElement
    private String metric_value;

    @XmlElement
    private String metric_unit;

    @XmlElement (name="metric_timestamp")
    private Date timestamp;

    @XmlElement
    private String service_resource_id;

    @XmlElement
    private String virtual_resource_id;

    @XmlElement
    private String resource_type;

    @XmlElement
    private String collector_id;
}
```

Would generate pieces of XML such as:

```xml
<monitoring_resource>
  <physical_resource_id>optimisaws001</physical_resource_id>
  <metric_name>machine_type</metric_name>
</monitoring_resource>
```
20.3.2 S3 Wrapper implementation

The implementation of the S3 wrapper consists on a class that contains the methods designed in the previous section. As a matter of example, the following code snippet shows the implementation of the `uploadImage` method.

```java
public URL uploadImage(String serviceId, String image) throws Exception {
    AmazonS3 s3 = getAmazonS3Client();
    String bucketName = getBucketName(serviceId);
    String key = "";
    try {
        if (!s3.doesBucketExist(bucketName)) {
            System.out.println("Creating bucket " + bucketName + "...");
            s3.createBucket(bucketName, Region.EU_Ireland);
        }

        File fileToUpload = new File(image);
        if (!fileToUpload.exists()) {
            throw new Exception("The specified file does not exist");
        }

        System.out.println("Uploading file...");
        key = fileToUpload.getName();
        // Create an upload request and attach a progress listener to it
        PutObjectRequest req = new PutObjectRequest(bucketName, key, fileToUpload)
            .withProgressListener(new TransferProgressListener());
        // begin the upload and store the transfer object
        transfer = tx.upload(req);
    } catch (AmazonServiceException se) {
        printServiceException(se);
    } catch (AmazonClientException ce) {
        printClientException(ce);
    } catch (SecurityException e) {
        System.out.println("[ERROR] Exception when trying to read the file: " + e.getMessage());
    }
}
```
20.3.3 Collector implementation

The implementation of the collector consists on a script that receives two parameters:

- “-i <collector id>” can be passed to specify the collector id value to use.
- “-c <path>” can be passed to specify the configuration file to use.

The script uses a configuration file to set the AWS credentials and the URL of the aggregator. Having an external configuration file makes it easier and less error-prone to configure the collector for a specific usage.

The script first parses the input parameters and reads the configuration file, then gathers all the measurements as designed in the previous section and finally sends them to the aggregator using Jersey to perform an HTTP POST call to it.

20.4 Testing

We have tested each of the components in isolation to ensure that they will work as expected when integrated with the Cloud4SOA eco-system. We have followed an approach based on unit testing for the CQoS proxy in the same way that we did with the OCCI CQoS proxy. However, due to the complexity of unit testing the other two components, we have opted for a manual approach in which we have executed each method in a variety of conditions and checked that they work as expected.

The following sections describe the tests that we have performed both for the S3 Wrapper and the collector.

20.4.1 S3 Wrapper Tests

We have created a simple GUI to test the wrapper component in an easier way. When executing the GUI application, a "-c <path>" parameter can be passed to indicate the path of a properties file containing the user’s AWS access and secret keys. The application cannot be executed without providing this properties file, since it needs these values to communicate with Amazon S3. Through the interface we have tested that each of the functionalities provided by the wrapper works properly in a variety of conditions. Table 20.5 describes the different tests performed.

As a matter of example, we show below the basic functioning of the wrapper.

Figure 20.3 shows the simple GUI that we have implemented. At the top there is a button to choose and upload a file, a progress bar that indicates the progress of an upload or download and a box that displays the URL of the uploaded image. At the left there is a list that displays all the uploaded images and a couple of buttons to download or delete an image.
<table>
<thead>
<tr>
<th>Use Case</th>
<th>Situation</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upload Image</td>
<td>Upload a non-existent image.</td>
<td>Exception saying that the file cannot be found.</td>
</tr>
<tr>
<td></td>
<td>Upload image for a service for the first time.</td>
<td>A bucket is created and the image is uploaded.</td>
</tr>
<tr>
<td></td>
<td>Upload image for a service that already has some images.</td>
<td>The image is uploaded to the bucket related to the service.</td>
</tr>
<tr>
<td></td>
<td>Upload small file image (of the order of MB).</td>
<td>The image is uploaded properly.</td>
</tr>
<tr>
<td></td>
<td>Upload large file image (of the order of GB).</td>
<td>The image is uploaded properly.</td>
</tr>
<tr>
<td>Download Image</td>
<td>Download a non-existent image.</td>
<td>Exception saying that the image cannot be found.</td>
</tr>
<tr>
<td></td>
<td>Download image from non-existent service.</td>
<td>Exception saying that the image cannot be found.</td>
</tr>
<tr>
<td></td>
<td>Download small existent image.</td>
<td>The image is downloaded properly.</td>
</tr>
<tr>
<td></td>
<td>Download large existent image.</td>
<td>The image is downloaded properly.</td>
</tr>
<tr>
<td>List Images</td>
<td>List images for non-existent service.</td>
<td>No images are listed.</td>
</tr>
<tr>
<td></td>
<td>List images for existent service.</td>
<td>All images names are listed.</td>
</tr>
<tr>
<td>Delete image</td>
<td>Delete a non-existent image.</td>
<td>Exception saying that the image cannot be found.</td>
</tr>
<tr>
<td></td>
<td>Delete image from non-existent service.</td>
<td>Exception saying that the image cannot be found.</td>
</tr>
<tr>
<td></td>
<td>Delete an existent image</td>
<td>The image is deleted properly.</td>
</tr>
</tbody>
</table>

Table 20.5: Description of the AWS S3 Wrapper tests

![Figure 20.3: AWS Wrapper Test Interface](image)

At the beginning there are no images uploaded, so we upload a small test file (see figure 20.4). The progress bar shows the progress of the update, and the text box at the left shows the
URL where the file can be accessed once the upload has finished.

![AWS Wrapper Test Interface while performing an Upload operation](image1)

Figure 20.4: AWS Wrapper Test Interface while performing an Upload operation

Figure 20.5 shows that there is a bucket created in S3 with the uploaded image file. The name of the bucket is a combination of “optimis-” and the service id.

![AWS S3 interface showing the bucket associated to the service after uploading the image](image2)

Figure 20.5: AWS S3 interface showing the bucket associated to the service after uploading the image

Figure 20.6 shows the interface after having uploaded the image file. Now the list shows the uploaded file.

![AWS Wrapper Test Interface after the Upload operation](image3)

Figure 20.6: AWS Wrapper Test Interface after the Upload operation

If we download the image, it will ask for a path and then download the image into the specified path. Again, the progress bar will show the progress of the download (see figure 20.7).
Finally, figure 20.8 shows the contents of the bucket after deleting the image.

20.4.2 Collector Tests

The testing of the collector script has consisted on executing it in a variety of conditions and checking that it generates a correct XML file. Moreover, we have implemented a dummy web service to simulate the Aggregator. The web service just receives the file, displays it on the standard output and returns it back to the caller. Table 20.6 describes the conditions within which the script has been tested.

20.5 OPTIMIS Integration

In addition to integrate each of the implemented components with the OPTIMIS eco-system, the Amazon IP needs to be added to the IP registry in order to be able to fully use the system to burst services to Amazon. The IP registry is an XML file that describes the characteristics of each available IP, such as their name and their endpoints.

Once OPTIMIS is aware of the new IP, it can burst services to it with the help of the implemented components by following the following sequence of actions:
<table>
<thead>
<tr>
<th>Situation</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute the script with invalid parameters.</td>
<td>The script exits with an error and shows the usage help message.</td>
</tr>
<tr>
<td>Execute the script with syntactically valid parameters but with an invalid configuration file path.</td>
<td>The script exits with an error saying that the configuration path cannot be found.</td>
</tr>
<tr>
<td>Execute the script with no parameters.</td>
<td>The default parameters are used and the script runs normally.</td>
</tr>
<tr>
<td>Execute the script with correct and valid parameters.</td>
<td>The values passed are used and the script runs normally.</td>
</tr>
<tr>
<td>Execute the script with invalid AWS credentials in the configuration file.</td>
<td>The script exits with an “Unauthorized access” error from AWS.</td>
</tr>
<tr>
<td>Execute the script with an invalid Aggregator URL in the configuration file.</td>
<td>The script exits with a “404 Not Found” error from Jersey.</td>
</tr>
<tr>
<td>Execute the script having a normally deployed OPTIMIS service</td>
<td>The script generates and sends an XML with all the metrics except the “memory used”.</td>
</tr>
<tr>
<td>Execute the script having a service whose instances are running the memory monitoring script provided by AWS</td>
<td>The script generates and sends an XML file containing all the metrics, including the “memory used”.</td>
</tr>
</tbody>
</table>

Table 20.6: Description of the AWS Collector tests

1. A service is first deployed manually in a full-OPTIMIS IP. Its manifest contains a rule that states that bursting should be performed when a certain condition happens, such as that the percentage of CPU usage reaches a given threshold.

2. At some point the monitoring infrastructure warns the Elasticity Engine (EE) that the threshold has been reached.

3. The EE asks the Cloud Optimizer to take the appropriate action to resolve the issue, such as start a new VM in another IP.

4. The Cloud Optimizer determines whether it is necessary or not to burst the service and, if positive, tells the Service Deployment Optimizer to perform the bursting.

5. The SDO decides which IP to burst to, in this case it decides Amazon, and contacts its CQoS (the implemented CQoS proxy) to negotiate the deployment of the service.

6. Once an agreement is found between the two, the SDO creates the agreement through the CQoS who, in turn, deploys the service.

7. The SDO then, asks the Data Manager to upload the image of the service in case it has not been uploaded. The Data Manager, in turn, uses the S3 Wrapper to upload the image to Amazon S3.
8. From that point on, the Monitoring Infrastructure starts monitoring the deployed service and may decide to take appropriate actions based on the gathered data.
Chapter 21

Conclusions

In this part of the project we have contributed to the OPTIMIS European project by providing two new communication channels: one to any OCCI-based IP and another to Amazon Web Services. In the first case we have designed and implemented a Cloud QoS proxy able to communicate with OCCI-based RESTful services in order to perform operations such as start and stop Virtual Machines hosting OPTIMIS services; moreover, we have connected this proxy to the Arsys service and tested the proper functioning of the communication channel. In the second case we have designed and implemented a set of components that can be used to enable AWS support in OPTIMIS, specifically we have designed integration components for the Cloud QoS, the Data Manager and the Monitoring Infrastructure.

We think that the Cloud market could really benefit from the adoption of a standard way to define interfaces and processes as this would greatly enhance the interoperability between different providers. However, in today’s market, the big players (like Amazon) seem reluctant to adopt these standards. This produces a heterogeneous field in which each provider exposes a unique way of communicating with it. If we compare the OCCI CQoS proxy with the AWS one, for instance, we see that both use a remarkably different communication mechanism (see table 21.1 for some examples). By introducing the adoption of a standard, platforms such as OPTIMIS could interact with any IP by just providing a single communication channel, creating a tighter and more interoperable cloud eco-system.

<table>
<thead>
<tr>
<th>OCCI</th>
<th>Amazon Web Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Uses a RESTful interface</td>
<td>• Uses a SOAP interface</td>
</tr>
<tr>
<td>• Provides a discovery interface for resources</td>
<td>• Uses proprietary concepts, such as EBS Volume or Amazon Machine Image</td>
</tr>
<tr>
<td>• Does not enforce any particular authentication mechanism.</td>
<td>• Uses a proprietary authentication mechanism.</td>
</tr>
<tr>
<td>• …</td>
<td>• …</td>
</tr>
</tbody>
</table>

Table 21.1: Some of the differences between an OCCI-compliant service and AWS
Introducing a commercial IP into the OPTIMIS eco-system has happened to have a set of implications that need to be addressed. Probably the most relevant is the usage fees associated to a commercial provider, which introduce a new variable that needs to be taken into account when considering bursting to it. “Does the cost impact on the rating of this provider?” or “Does the cost impact on the amount of resources that are burst to it?” are examples of questions that need to be answered. Moreover, a stronger contractual relation has to be established between Service Providers and OPTIMIS in order to allow the latter to incur an economic cost on behalf of the first. Other issues that have to be dealt with are the impossibility of installing custom OPTIMIS components on their infrastructure; the set of rules and regulations that AWS imposes on its usage, such as limits on the network traffic or on the number of concurrent EC2 instances, and the format and content of VM images which has to comply to the Amazon specifications. Despite all this, integrating AWS in the OPTIMIS eco-system provides it with a strong, reliable and trustworthy IP that can be used for highly demanding services when no other IP can deliver the necessary levels of performance or for critical applications that require a highly reliable infrastructure.
Conclusions and Lessons Learned

In this project we have contributed to two on-going European research projects (Cloud4SOA and OPTIMIS) with the goals of extending the tools they provide and understanding in detail the current issues with cloud interoperability, especially at the IaaS and PaaS layers of the cloud stack. In both contributions we have extended the tools that the projects provide by connecting them to additional cloud providers of different natures: commercial, Open Source, public and private. This has permitted us to observe the differences between distinct kinds of cloud providers and the difficulties associated with interacting them.

We have found several difficulties during the course of the project that have slowed it down and, as a consequence, some of the goals in the “Exploring interoperability at the IaaS layer” part have not been fully fulfilled. One of such difficulties has been the lack of CloudFoundry documentation which has required to look directly into the code and to get involved in the community of developers to resolve all doubts, both to design the adapter and to install CloudFoundry. In another direction, connecting the OCCI Cloud QoS (CQoS) proxy with Arsys has required a slow and iterative process in order to tweak both the adapter and the Arsys service to conform to the needs of both parts. For example, initially the adapter was designed and implemented to send and receive messages in plain text but since Arsys required messages to be sent in an structured format (such as JSON or XML) we had to change the adapter to use the OCCI JSON rendering. Another difficulty that we have found across the entire project has been having to understand in detail the architecture and the code of both Cloud4SOA and OPTIMIS, due in part to the complexity of the tools and platforms.

Despite all the difficulties, we can consider that the project has been successful since we have provided an added-value contribution for the two European projects and we have learned more about the interoperability in the cloud. We think that the cloud community should make a big effort to extend the adoption of standards, especially across the bigger players such as Amazon, Google or Microsoft. This would progressively move the currently disperse cloud environment into a global network of interconnected cooperating clouds, usually referred in the literature as the InterCloud [Hof10], and would enable the quicker development of multi-cloud tools that leverage this network to provide users with a single-point access to a multitude of providers. We think that the current solution based on developing independent plug-ins for each provider is not feasible in the long run as the cloud market grows since it requires a considerable effort and constant update to adapt to emerging providers. Taking OPTIMIS, for instance, we developed a proxy based on the OCCI standard that can be used to interact with any provider that adopts this standard and a proxy for AWS that can be used only to interact with AWS. In an ideal situation in which a single standard were adopted by all cloud providers, OPTIMIS and Cloud4SOA would not need any proxies or adapters since they could directly talk to any cloud by following this standard. This would reduce the effort and the complexity of the tools and would provide users with the opportunity to choose any cloud in the market.
We have also seen that it is not easy to deal with clouds of different natures. When we interact with a private cloud, for instance, we have to be aware that it has a limited set of resources which may limit the number of applications or VMs that we can deploy on it, or that the reliability is lower than in a high-scale public cloud. Similarly, commercial providers impose their own regulations and limitations; AWS, for instance, limits the maximum number of VMs running at the same time to 20. Also, public clouds often have a usage cost, while private clouds usually not. All these distinctions require a different treatment of each provider; in particular, SLAs have to be created and managed differently to comply with the particular characteristics of each of provider.

At a personal level, this project has been a great experience. It has given me the opportunity to work with one of the largest and most relevant companies in the IT sector in Europe and, although it has been tough at times due to the nature of such a huge company, it has been useful to learn how a big enterprise like Atos works from the inside. I have also experienced how it is like to work in a European research project. It is certainly not easy to work with a large team of people in which each individual has its own peculiarities, and to have to learn all the ins and outs of a complex project such as Cloud4SOA and OPTIMIS. One experience that has been particularly useful has been the design of the Arsys service (see chapter 18) since it has resembled, up to a certain level, a situation in which a design is made by one party and is implemented by another independent party. During this decoupled process, we have needed to clarify things about the documentation multiple times and adapt some parts to their needs. Moreover, when the implementation was ready we observed that some parts of the design where not being followed and we had to interact with them to progressively achieve a full and compliant implementation. Finally, at an academic level, the project has required me to remember knowledge from subjects such as “Computer Networks”, “Software Engineering”, “Computers Architecture” and “Software Design based on the Web” and has given me further abilities and knowledge, especially with respect to cloud computing and virtual infrastructure management, that can greatly help me in my career.
Appendix A

JSON OCCI Messages Specification

All data transferred using the JSON Content-Type is structured using the below specifications. The following structures are used to build and parse messages:

- Resource Instance
- Link
- Attributes
- Action

Note: This document only describes the structures necessary for this service, which reduces to the structures to represent resource instances along with their links, attributes and actions. OCCI also specifies a set of structures to represent Categories (Kind and Mixin) and Actions along with the definition of their attributes. These structures are only used to answer to queries made through the discovery interface which, as discussed in chapter 18, is not implemented in this service.

A.1 Resource Instance

A resource instance, such as an instance of OptimisCompute, Network, or Storage, is represented as a Resource object with the following structure:

```json
{
    "kind": String,
    "mixins": [String, String, ...],
    "attributes": Attributes,
    "actions": [Action, Action, ...],
    "id": String,
    "title": String,
    "summary": String,
    "links": [Link, Link, ...]
}
```

An array of resource instances is represented also as a JSON object with the following structure:
Table A.1 shows the definition of the object members.

<table>
<thead>
<tr>
<th>Object member</th>
<th>JSON type</th>
<th>Description</th>
<th>Necessity</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>Kind identifier</td>
<td>Mandatory</td>
</tr>
<tr>
<td>mixins</td>
<td>array of strings</td>
<td>List of mixin identifiers</td>
<td>Mandatory if resource has mixins</td>
</tr>
<tr>
<td>attributes</td>
<td>object</td>
<td>Instance attributes</td>
<td>Mandatory if resource has attributes</td>
</tr>
<tr>
<td>actions</td>
<td>array of objects</td>
<td>Actions applicable to the resource instance (see the Action structure)</td>
<td>Mandatory if resource has applicable actions</td>
</tr>
<tr>
<td>id</td>
<td>string</td>
<td>Resource identifier</td>
<td>Mandatory</td>
</tr>
<tr>
<td>title</td>
<td>string</td>
<td>Title of the resource</td>
<td>Optional</td>
</tr>
<tr>
<td>summary</td>
<td>string</td>
<td>Summary describing the resource</td>
<td>Optional</td>
</tr>
<tr>
<td>links</td>
<td>array of objects</td>
<td>Associated OCCI Links (see the Link structure)</td>
<td>Mandatory if resource has outgoing links</td>
</tr>
</tbody>
</table>

Table A.1: Resource instance structure object members description

A.2 Link

Links can be represented both within the “links” member of a Resource object or within a standalone “links” array of Link objects. Both representations use the same structure, with the only difference that the “source” is omitted in the first case since it is implicitly defined.

A link is represented using a Link object with the following structure:

```json
{
    "kind": String,
    "mixins": [String, String, ...],
    "attributes": Attributes,
    "actions": [Action, Action, ...],
    "id": String,
    "title": String,
    "target": String,
    "source": String
}
```

An array of links is represented also as a JSON object with the following structure:
Table A.2 shows the definition of the object members.

<table>
<thead>
<tr>
<th>Object member</th>
<th>JSON type</th>
<th>Description</th>
<th>Necessity</th>
</tr>
</thead>
<tbody>
<tr>
<td>kind</td>
<td>string</td>
<td>Kind identifier</td>
<td>Mandatory</td>
</tr>
<tr>
<td>mixins</td>
<td>array of strings</td>
<td>List of mixin identifiers</td>
<td>Mandatory if link has mixins</td>
</tr>
<tr>
<td>attributes</td>
<td>object</td>
<td>Instance attributes</td>
<td>Mandatory if link has attributes</td>
</tr>
<tr>
<td>actions</td>
<td>array of objects</td>
<td>Actions applicable to the link (see the Action structure)</td>
<td>Mandatory if link has applicable actions</td>
</tr>
<tr>
<td>id</td>
<td>string</td>
<td>Link identifier</td>
<td>Mandatory</td>
</tr>
<tr>
<td>title</td>
<td>string</td>
<td>Title of the link</td>
<td>Optional</td>
</tr>
<tr>
<td>target</td>
<td>string</td>
<td>Absolute location of target resource</td>
<td>Mandatory</td>
</tr>
<tr>
<td>source</td>
<td>string</td>
<td>Absolute location of source resource</td>
<td>Mandatory unless rendered within the source resource</td>
</tr>
</tbody>
</table>

Table A.2: Link structure object members description

A.3 Action

An action is represented as an Action object with the following structure:

```json
{
    "title": String,
    "href": String,
    "category": String
}
```

Table A.3 shows the definition of the object members.

A.4 Attributes

Attributes defined by Entities and Actions are rendered using the Attributes object. The Attributes object organizes attributes in a hierarchical object structure, according to hierarchy defined in an attribute’s name. A hierarchy is defined in an attribute’s name by separating
<table>
<thead>
<tr>
<th>Object member</th>
<th>JSON type</th>
<th>Description</th>
<th>Necessity</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>string</td>
<td>Title of the action</td>
<td>Optional</td>
</tr>
<tr>
<td>href</td>
<td>string</td>
<td>The relative location of the path to execute the action</td>
<td>Mandatory</td>
</tr>
<tr>
<td>category</td>
<td>string</td>
<td>The absolute location of the Action definition.</td>
<td>Mandatory</td>
</tr>
</tbody>
</table>

Table A.3: Action structure object members description

the different parts with dots. For example, the attributes “org.occi.optimis.compute.speed”, “org.occi.optimis.compute.memory” and “org.occi.optimis.compute.cores” are rendered as:

```json
"org": {
  "occi": {
    "optimis": {
      "compute": {
        "speed": value,
        "memory": value,
        "cores": value
      }
    }
  }
}
```

A.5 Example

This section shows an example representation of an optimis_compute resource instance.

```json
{
  "kind": 
    "[service_schemas_url]#optimis_compute",
  "attributes": {
    "occi": {
      "compute": {
        "speed": 2,
        "memory": 4,
        "cores": 2
      }
    }
  },
  "optimis": {
    "occi": {
      "optimis_compute": {
        "service_id": "1234",
        "image": "fedora8image.vmdk"
      }
    }
  }
}
```
"actions": [
{
   "title": "Start VM",
   "href": "/optimis_compute/1234/vm1?action=start",
   "category": "http://schem.ogf.org/oCCI/compute/action#start"
}
],
"id": "vm1",
"title": "Optimis Compute resource"
Appendix B

HTTP Response Codes

Table B.1 shows the possible return codes that the service can return as a result of any operation.
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
<td>Indicates that the request was successful.</td>
</tr>
<tr>
<td>201</td>
<td>OK</td>
<td>Indicates that the request was successful. The response must contain a HTTP Location header to the newly created resource instance.</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>Used to signal parsing errors or missing information (e.g. an attribute that is required is not supplied in the request).</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td>The client does not have the required permissions/credentials.</td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
<td>Used to signal that a particular Mixin cannot be applied to a resource instance of a particular Kind. Used to signal that an attempt was made to modify an attribute that was marked as immutable.</td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
<td>Used to signal that the request had information (e.g. a kind, mixin, action, attribute, location) that was unknown to the service and so not found.</td>
</tr>
<tr>
<td>405</td>
<td>Method Not Allowed</td>
<td>The service does not allow the client to issue the HTTP method against the requested path/location.</td>
</tr>
<tr>
<td>409</td>
<td>Conflict</td>
<td>A request contains content (e.g. mixin, kind, action) that results in an internal service, non-unique result (e.g. two types of start actions are found for Compute). The client must resolve the conflict by re-trying with specific Category information in the request.</td>
</tr>
<tr>
<td>410</td>
<td>Gone</td>
<td>A client attempts to retrieve a resource instance that no longer exists (i.e. it was deleted).</td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
<td>The state before the request should be maintained in such an error condition. The implementation MUST roll-back any partial changes made during the erroneous execution.</td>
</tr>
<tr>
<td>501</td>
<td>Not Implemented</td>
<td>If an implementation chooses not to implement a particular OCCI feature, it must signal the lack of that feature with this code. This implicitly points to a non-compliant OCCI implementation.</td>
</tr>
<tr>
<td>503</td>
<td>Service Unavailable</td>
<td>If the OCCI service is taken down for maintenance, this error code should be reported from the root of the name-space the provider uses.</td>
</tr>
</tbody>
</table>

Table B.1: Possible HTTP Status codes that the service can return, and their meaning
Appendix C

Produced Code

The code produced as an outcome of this project has been uploaded to public GitHub repositories so that it can be accessed freely by any party interested. We have decided to use this dissemination mechanism rather than providing a traditional disk because it does not require providing any physical resource and because it allows an easier update and inspection of the code. We have made available 5 public repositories, one for each of the developed pieces of software:

**Cloud4SOA CloudFoundry Adapter** This repository can be cloned from https://github.com/oriolbcn/c4s-cloudfoundry-adapter.git and contains the source code of the remote part of the Cloud4SOA CloudFoundry adapter packed as a Maven Java project. The local part of the adapter is not included because it is coupled with the base Cloud4SOA source code. Refer to the README file in the repository for more details.

**OPTIMIS OCCI CQoS Proxy** This repository can be cloned from https://github.com/oriolbcn/optimis-occi-cqos.git and contains the source code of the OCCI CQoS proxy packed as a Maven Java project. Refer to the README file in the repository for more details.

**OPTIMIS AWS CQoS Proxy** This repository can be cloned from https://github.com/oriolbcn/optimis-aws-cqos.git and contains the source code of the AWS CQoS proxy packed as a Maven Java project. Refer to the README file in the repository for more details.

**OPTIMIS AWS S3 Wrapper** This repository can be cloned from https://github.com/oriolbcn/optimis-aws-s3wrapper.git and contains the source code of the AWS S3 Wrapper packed as a Maven Java project. Refer to the README file in the repository for more details and for a description on how the wrapper interface can be executed for testing purposes.

**OPTIMIS AWS Collector** This repository can be cloned from https://github.com/oriolbcn/optimis-aws-collector.git and contains the source code of the AWS Collector packed as a Maven Java project. Refer to the README file in the repository for more details and for a description on how the collector can be executed.
Bibliography


