Project no. FP7-216556

Instrument: Integrated Project (IP)

Objective ICT-2007.1.2: Service and Software Architectures, Infrastructures and Engineering

**Deliverable D.A4.a SLA-aware Infrastructure Management - evaluated framework**

**Keywords:**
Web Site, Service Level Agreement, Service-Oriented Infrastructure

**Due date of deliverable:** 31st July 2011

**Actual submission to EC date:** 29th July 2011

**Start date of project:** 1st June 2008

**Duration:** 38 months

**Lead contractor for this deliverable:** Intel

<table>
<thead>
<tr>
<th>Dissemination level</th>
</tr>
</thead>
<tbody>
<tr>
<td>PU</td>
</tr>
<tr>
<td>Public</td>
</tr>
<tr>
<td>Yes</td>
</tr>
</tbody>
</table>
Document Status

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Juan Lambea (Telefonica)</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Miha Stopar (XLAB)</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Ramin Yahyapour</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>18 July 2011</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>20 July 2011</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>22 July 2011</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>27 July 2011</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>28 July 2011</td>
</tr>
</tbody>
</table>

Contributors

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>John Kennedy, Andy Edmonds, Pat Cheevers</td>
</tr>
<tr>
<td>UDO</td>
<td>Kuan Lu, Florian Feldhaus, Peter Chronz</td>
</tr>
<tr>
<td>XLAB</td>
<td>Miha Stopar, Damjan Murn, Jaka Močnik</td>
</tr>
</tbody>
</table>

Notices

The information in this document is provided "as is", and no guarantee or warranty is given that the information is fit for any particular purpose. The above referenced consortium members shall have no liability for damages of any kind including without limitation direct, special, indirect, or consequential damages that may result from the use of these materials subject to any liability which is mandatory due to applicable law. Copyright 2011 by the SLA@SOI consortium.

* Other names and brands may be claimed as the property of others.

This work is licensed under a Creative Commons Attribution 3.0 License.
# Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>17 June 2011</td>
<td>John Kennedy</td>
<td>Initial draft based on M26.</td>
</tr>
<tr>
<td>0.2</td>
<td>8 July 2011</td>
<td>John Kennedy</td>
<td>Integrated contributions from XLAB, Intel, UDO.</td>
</tr>
<tr>
<td>0.3</td>
<td>18 July 2011</td>
<td>John Kennedy</td>
<td>Consolidated for internal review.</td>
</tr>
<tr>
<td>0.4</td>
<td>25 July 2011</td>
<td>John Kennedy</td>
<td>Incorporating feedback from internal reviewers plus additional contributions.</td>
</tr>
<tr>
<td>1.0</td>
<td>28 July 2011</td>
<td>John Kennedy</td>
<td>Final tweaks based on PMT review.</td>
</tr>
</tbody>
</table>
Executive Summary

The SLA@SOI project is committed to research, engineer and demonstrate technologies that can embed SLA-aware infrastructures into the service economy. These technologies span the business, service and infrastructure layers a service economy depends upon. In SLA@SOI the Infrastructure Management work package is responsible for the infrastructure layer.

This document details the outputs of the Infrastructure Management work package in SLA@SOI. It builds upon the achievements of the first two years of the project documented in the Year 1 and Year 2 deliverables [1], [2] and thus focuses on the advancements and effort in this, the final year of the project.

Achievements include the substantial maturing of the reference implementation of an SLA-enabled infrastructure layer, with significant functionality added at all levels of the stack. The Open Grid Forums’s ground-breaking Open Cloud Computing Interface was enhanced and developed into version 1.1, and published. The internet-accessible testbed hosted by Intel was expanded, and complemented by an XLAB-hosted testbed facility. Key opportunities to integrate SLA-aware technologies into the community were also seized, with collaborations with Reservoir (OpenNebula), Apache Tashi, JClouds and Contrail all reinforced or initiated.

Interestingly, with SLAs at its core, and with a perspective not aligned with any specific cloud provider, SLA@SOI Infrastructure Management has maintained a focus unique and, we believe, valuable to the industry. As more and more businesses move to the cloud, and as more and more customers become aware of their SLAs when issues arise, the pull for machine-readable SLAs at the infrastructure layer will become significantly stronger. The adoption, integration and continuing enhancement of the work done by SLA@SOI will realise the project’s vision: of SLAs empowering a dependable service economy.
# Table of Contents

1  Introduction ................................................................................................................. 10
  1.1  Context and Scope ................................................................................................. 10
  1.2  Document Overview ............................................................................................... 14

2  Contribution Overview ................................................................................................. 16
  2.1  Key Innovations ...................................................................................................... 16
  2.2  Framework Contributions ....................................................................................... 17
  2.3  Additional outputs ................................................................................................. 17
  2.4  Task-level activities in 3rd project year ............................................................... 18

3  Architecture .................................................................................................................. 20
  3.1  Infrastructure Requirements ................................................................................. 20
  3.2  Monitoring Requirements ...................................................................................... 22
  3.3  State of the Art ...................................................................................................... 24
  3.4  Overall Architecture ............................................................................................. 29
  3.5  Infrastructure SLA Manager .................................................................................. 30
  3.6  Infrastructure Service Manager ............................................................................. 31
  3.7  Infrastructure Monitoring ..................................................................................... 32
  3.8  Deployment Scenarios ........................................................................................... 32

4  Infrastructure SLAs ........................................................................................................ 35
  4.1  Infrastructure Vocabulary ..................................................................................... 35
  4.2  Example Infrastructure SLAs ................................................................................ 35

5  Infrastructure Interface ................................................................................................. 41
  5.1  Introduction ............................................................................................................ 41
  5.2  Open Cloud Computing Interface (OCCI) ........................................................... 41

6  Implementation .............................................................................................................. 45
  6.1  Infrastructure SLA Manager ................................................................................... 45
  6.2  Infrastructure Planning and Optimization Component ........................................... 45
  6.3  Infrastructure Provisioning and Adjustment ........................................................ 53
  6.4  I-SM Proxy ............................................................................................................ 54
  6.5  JClouds OCCI API ................................................................................................ 60
  6.6  OCCI I-SM for Apache Tashi ................................................................................ 62
  6.7  Provisioning System – Apache Tashi ...................................................................... 64
  6.8  Infrastructure Monitoring ....................................................................................... 72
  6.9  OCCI I-SM for OpenNebula .................................................................................. 95

7  Deployment .................................................................................................................... 97
  7.1  External Testbed – Intel ........................................................................................ 97
  7.2  External Testbed – XLAB ..................................................................................... 99
  7.3  Internal Testbed ..................................................................................................... 101

8  Conclusions .................................................................................................................. 102
  8.1  Contributions and Achievements .......................................................................... 102
  8.2  Lessons learned ..................................................................................................... 102
  8.3  Outlook .................................................................................................................. 103

9  References .................................................................................................................... 104

Appendix A: Glossary ....................................................................................................... 107
Appendix B: Abbreviations ............................................................................................. 110
Appendix C: Multiple VM SLA Model ............................................................................ 111
Appendix D: Example RegisterService request ................................................................ 115
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Conceptual Architecture of SLA@SOI</td>
<td>11</td>
</tr>
<tr>
<td>Figure 2</td>
<td>High level Cloud Computing Taxonomy (Intel)</td>
<td>12</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Cloud Computing Focus Areas (Intel)</td>
<td>12</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Top level Architecture</td>
<td>29</td>
</tr>
<tr>
<td>Figure 5</td>
<td>G-SLAM architecture</td>
<td>30</td>
</tr>
<tr>
<td>Figure 6</td>
<td>I-SLAM Architecture</td>
<td>31</td>
</tr>
<tr>
<td>Figure 7</td>
<td>High-level deployment scenarios supported by the architecture</td>
<td>33</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Implementation approaches supported by the architecture</td>
<td>34</td>
</tr>
<tr>
<td>Figure 9</td>
<td>OCCI at the boundary of a service provider</td>
<td>42</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Computational geometry representation</td>
<td>47</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Computational geometry representation of different plans</td>
<td>48</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Class diagram for computational geometry implementation</td>
<td>49</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Finding alternative solutions by moving the request point</td>
<td>49</td>
</tr>
<tr>
<td>Figure 14</td>
<td>Fragment statistics</td>
<td>51</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Resources utilization of all servers</td>
<td>52</td>
</tr>
<tr>
<td>Figure 16</td>
<td>PAC class diagram</td>
<td>53</td>
</tr>
<tr>
<td>Figure 17</td>
<td>Relationship between OSGI, I-SLAM, I-SM Proxy, JClouds and OCCI I-SM for Apache Tashi</td>
<td>55</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Example provision request type</td>
<td>56</td>
</tr>
<tr>
<td>Figure 19</td>
<td>Internals of the OCCI I-SM for Apache Tashi</td>
<td>63</td>
</tr>
<tr>
<td>Figure 20</td>
<td>Apache Tashi Scheduler flow</td>
<td>65</td>
</tr>
<tr>
<td>Figure 21</td>
<td>Ideal vs. Real CPU Share</td>
<td>70</td>
</tr>
<tr>
<td>Figure 22</td>
<td>Infrastructure monitoring architecture</td>
<td>73</td>
</tr>
<tr>
<td>Figure 23</td>
<td>Infrastructure Monitoring Agent database schema</td>
<td>78</td>
</tr>
<tr>
<td>Figure 24</td>
<td>Architecture of OCCI I-SM for OpenNebula</td>
<td>95</td>
</tr>
<tr>
<td>Figure 25</td>
<td>Intel external testbed</td>
<td>97</td>
</tr>
<tr>
<td>Figure 26</td>
<td>XLAB external testbed</td>
<td>99</td>
</tr>
<tr>
<td>Figure 27</td>
<td>Intel’s internal testbed, hosting developers, VMs and Intel’s CEO Paul Otellini</td>
<td>101</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Document changes against previous version ........................................ 15
Table 2: Infrastructure Features from SLA@SOI Use Cases .......................... 22
Table 3: Infrastructure Requirements from SLA@SOI Use Cases .................. 23
Table 4: ORC Gold SLA ............................................................................. 36
Table 5: Agreement Term for VM_0 .......................................................... 38
Table 6: Server capacity ............................................................................ 50
Table 7: Satisfaction ratio of requests with different shifting steps .............. 50
Table 8: JClouds Methods ......................................................................... 60
Table 9: Intel external testbed physical node attributes .............................. 98
Table 10: Supplementary Intel external testbed physical node attributes ....... 98
1 Introduction

1.1 Context and Scope

The SLA@SOI project has been committed to research, engineer and demonstrate technologies that can embed SLA-aware infrastructures into the service economy.

The main project objectives [3] include

1. To design and implement a consistent SLA-management framework that relates perspectives of relevant stakeholders (software/service/infrastructure provider and customer) in a consistent and transparent way
2. To design and implement foundations of an adaptive SLA-aware infrastructure that allows for harmonized access to different virtualization technologies.
3. To advance standards on interfaces and protocols for SLA specification and management and on service-oriented infrastructures.

In this context, several key actors, functional components, and the information flows between them were identified, as illustrated in the conceptual architecture in Figure 1.

The base layer of this conceptual architecture is infrastructure: the layer of hardware (both real and virtual) and software on which Services are deployed and executed as per their SLAs. The realisation of this layer was assigned to a dedicated Work Package (WP) within the project, WP A4 Infrastructure Management.

Formally, the Infrastructure Management work package had the following objectives:

- to define an abstracted infrastructure based on fine grained resource virtualisation.
- to build the foundations for network based resource sharing.
- to build support for SLA-aware service provisioning and SLA enforcement.
- to provide infrastructural support for orchestrated, heterogeneous and dynamic resources.
- to enable network aware resource virtualisation and harmonisation.

Thus, this work package focused on the SLA-aware management of infrastructural resources such as computers, networks and storage, be they real or virtual. It has sought to define and demonstrate an abstract, service-oriented framework for SLA-aware dynamic resource provisioning and configuration in support of a Service Oriented Architecture (SOA). This framework manages the resource issues associated with the specification, negotiation, fulfilment and enforcement of SLAs.
Infrastructure management, and indeed much else of the SLA@SOI project, is directly concerned with and relevant to cloud computing. There are many definitions of cloud computing - Intel, for example, define cloud computing as a computing paradigm where services and data reside in shared resources in scalable data centres, and those services and data are accessible by any authenticated device over the Internet [4]. In Intel’s Cloud computing taxonomy, SLA@SOI Infrastructure management work package primarily maps to Infrastructure as a Service concerns.

**Figure 1: Conceptual Architecture of SLA@SOI**
Some of the issues addressed by Infrastructure Management over the three years of the project have included:

- Framework definition – identifying the infrastructure resource features that need to be exposed at an abstract level.
- Abstract infrastructure virtualisation – focusing on abstracted hardware environments with harmonised, implementation technology neutral interfaces.
- Management roles, entities and interfaces – enabling automated policy enactment and its integration with operator UI-based decision making.
- Resource specification – enabling the determination of virtual resource requirements in support of SLA negotiation.
• Configuration, scheduling and provisioning – enabling the SLA-aware management of virtual and physical resources.
• Dynamic re-provisioning – enabling an infrastructure to be responsive to changing SLA requirements.
• Demand anticipation – enabling automatic and autonomic control of an infrastructure.
• SLA negotiation and support – helping automate the initial settling on an SLA, and the on-going monitoring of them
• Alerts, monitoring and logging – enabling an infrastructure to cope with failure and to meet SLA requirements.

The work package has also provided test bed resources together with workload synthesis tools for this and other relevant work packages, supporting in particular the SLA@SOI Adhoc Demonstrator and the relevant SLA@SOI Industrial Use Cases.

**Relationship with other Work Packages**

WP A4 has worked closely with many of the other work packages in the project. In particular:

- WP A4 has worked with **WP A1 – Architecture & Integration** to help define the overall architecture, and then to implement the infrastructure specific components it describes. A4 has also contributed advice and effort into the overall integration of the software developed by the project. WP A1 work is documented in Deliverable D.A1a [5].

- WP A4 has worked closely with **WP A5 – SLA Management and Foundations** to document a suitably detailed Infrastructure SLA, and build rich Infrastructure-specific implementations of the Planning and Optimisation, and Provisioning and Adjustment components (IPOC and IPAC respectively). WP A5 work is documented in Deliverable D.A5a [8].

- WP A4 has also engaged with **A2 – Business Management** [6], **A3 – Service Management** [7] and **A6 Predictable Systems Engineering** [9], especially when components related to infrastructure needed to be considered or developed.

- The infrastructure layer and testbed provided by WP A4 have been exercised by several work packages, including in particular **WP B2 – Open Reference Case** [10], **WP B4 – Use Case: Enterprise IT** [11], and **WP B5 – Use Case: Service Aggregator** [12].

**Main achievement**

The main achievement of WP A4 Infrastructure Management could be considered to be the delivery of a reference implementation of an SLA-enabled infrastructure layer.

This software illustrates the implementation of the consistent SLA-management framework defined in SLA@SOI (objective 1), and showcases the foundations of an adaptive SLA-aware infrastructure that allows for harmonized access to different virtualization technologies (objective 2) through ground-breaking standards like Open Grid Forum’s Open Cloud Computing Interface, a standard co-chaired by WP A4 Infrastructure Management.

The implementation of WP A4 was achieved in collaboration with numerous external parties including Reservoir (OpenNebula), Apache Tashi, and JClouds,
broadening the reach of the project in the community. On-going engagements with projects such as Contrail, Timbus, and FI-WARE provide for the maximum exploitation of the work of WP A4.

1.2 Document Overview

This document describes the overall output of WP A4 - Infrastructure Management, focusing on the efforts and progress in year 3 of the project.

Chapter 2 describes the key contributions of the work at a high level. Key innovations, framework contributions, additional outputs and task-level activities are all described.

Chapter 3 addresses the architecture of the Infrastructure Management layer. Overall requirements are summarised, a summary of the current state-of-the-art is presented, the evolution of the architecture is described, and the current architecture itself is outlined. The deployment options that this architecture supports are also illustrated.

Chapter 4 explains how the SLA Model has been adopted by the Infrastructure Layer. Example SLAs are shown, illustrating how single VM and multiple VM infrastructure SLAs can be described with a variety of functional and non-functional parameters.

Chapter 5 focuses on Infrastructure Interfaces. In particular, this chapter describes the OCCI interface standard, now at version 1.1, which WP A4 Infrastructure Management has not just contributed to but also co-chaired.

Chapter 6 describes the reference implementation of all components in the infrastructure stack, from the Infrastructure SLA Manager through JClouds OCCI API to the Infrastructure Service Manager (both Apache Tashi and OpenNebula based) and Low Level Monitoring System.

Chapter 7 addresses the deployment of the Infrastructure Manager layer. This chapter documents the internal and external testbeds.

Finally, Chapter 8, Conclusions, summarises the key outcomes and learning’s from this work package, and outlines some of the expected avenues of exploitation.

The following table lists all the changes and new elements of this deliverable compared to the previous deliverable version.
<table>
<thead>
<tr>
<th>Section</th>
<th>Change overview</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Refreshed, reflecting activities in Year 3. Now includes links to overall project objectives and other work packages.</td>
</tr>
<tr>
<td>2</td>
<td>Refreshed, reflecting activities in Year 3. Software contributions made into non-SLA@SOI open-source projects have been summarised.</td>
</tr>
<tr>
<td>3</td>
<td>Refreshed. OpenStack has been added to the state of the art analysis (Section 3.3.3). The validation against Nagios has been referenced (Section 3.3.4). The overall architecture, description of sub-components, and deployments scenarios have been updated (Sections 3.4 through 3.8)</td>
</tr>
<tr>
<td>4</td>
<td>Minor edits. Previous content still relevant. Included for completeness.</td>
</tr>
<tr>
<td>5</td>
<td>Comprehensive rewriting of the entire chapter.</td>
</tr>
<tr>
<td>6</td>
<td>Comprehensive rewriting of the entire chapter. All previous sub-sections completely rewritten and updated. New sub-sections added on JClouds OCCI API and OpenNebula based Infrastructure Service Manager.</td>
</tr>
<tr>
<td>7</td>
<td>Refreshed, reflecting activities in Year 3. Description of new XLAB-hosted testbed added.</td>
</tr>
<tr>
<td>9</td>
<td>Refreshed, reflecting activities in Year 3. Some lessons learned and an outlook on future activities has been added.</td>
</tr>
</tbody>
</table>
2 Contribution Overview

This chapter summarizes the main contributions of the Infrastructure Management work package. These contributions are in the form of key innovations, contributions to the overall SLA@SOI software framework, additional contributions outside of the SLA@SOI framework, and task-level activities.

2.1 Key Innovations

- The primary innovation from the Infrastructure Management work package is the **SLA-enabling of Infrastructure Providers**. As explained in the consortium’s State of the Art Analysis [13], at this the close of the project there are still no infrastructure providers in the industry providing holistic, automated SLA management of their infrastructure. From a service consumer point of view it is not possible to browse supported SLAs in a machine readable fashion, nor is automatic negotiation of personalised SLAs supported. From a service provider point of view, a framework for automatically managing a landscape of provisioned infrastructure based on the individual SLAs with customers is not yet available. These areas are all addressed by SLA@SOI Infrastructure Management. Following the architecture driven by WP A1 Architecture and Integration, and with key contributions from WP A5 SLA Management and Foundations, WP A4 Infrastructure Management has delivered a reference SLA-enabled Infrastructure layer. With the initial implementation built on top of Apache Tashi, a second infrastructure provisioning system, Open Nebula, is also being SLA-enabled.

- A separate and significant innovation from this work package has been the **harmonized interface to access heterogeneous infrastructure resources**. WP A4 has played a leadership role in the creation of OGF’s Open Cloud Computing Interface standard [13], with the April 2011 publication of the three documents describing Version 1.1 of this ground-breaking standard. The extensive analysis of third party Infrastructure Interfaces and third-party Infrastructure Models documented in the Year 1 deliverable [1], together with the models and interfaces developed inside the work package, have been contributed to the standards effort, with SLA@SOI pleased to participate as co-chair of the standards group.

- Another key innovation delivered to date has been a **multi-layer, self-provisioned distributed infrastructure monitoring system** integrated with Ganglia, a widely used open-source infrastructure monitoring framework. As described in the State of the Art Analysis [13] existing open infrastructure monitoring frameworks typically support a centralised approach to managing infrastructure. This innovation sees low level monitoring components automatically provisioned, depending on the needs of the consumers of the monitoring information. These distributed components are also managed particularly efficiently, with management commands distributed from a node on one layer to all associated nodes on the layer below. This decentralised management helps ensure the solution scales to very large infrastructure deployments.
2.2 Framework Contributions

- The contributions of the infrastructure management work package to the overall SLA@SOI Framework are
  - the infrastructure specific components of the reference implementation of the Infrastructure SLA Manager,
  - the reference Infrastructure Service Manager built of top of Apache Tashi and RESERVOIR’s OpenNebula
  - Low Level Monitoring System optimized for large scale Infrastructure management.

- The reference implementation of the Infrastructure SLA Manager includes an infrastructure-specific Planning and Optimisation component (IPOC), and an Infrastructure Provisioning and Adjustment component (IPAC). These have been developed in close partnership with work package A5 – SLA Management and Foundations.

- The reference Infrastructure Service Manager implementation includes a client-side proxy for the Infrastructure SLA Manager, an SLA@SOI-developed JClouds OCCI API, an OCCI-compatible interface, and Infrastructure Service Managers built on top of both Apache Tashi and OpenNebula.

- The Infrastructure Monitoring System is a monitoring layer built on top of arbitrary instrumentation systems - such as Ganglia or Nagios - that allows for automated configuration, deployment and management of monitoring components to meet the client’s needs. It uses an open messaging framework - currently XMPP – to avoid the bottlenecks possible with synchronous communications.

2.3 Additional outputs

WP A4 Infrastructure Management has also contributed significantly to additional channels, including open standards initiatives and complementary open source projects.

- WP A4 Infrastructure Management has put significant effort and resources into the development, managing and publication of the OGF’s Open Cloud Computing Interface standard, now at Version 1.1. Contributions from our state of the art analysis, review of Cloud APIs and models, and experiences gained from our implementations within SLA@SOI all helped feed and influence the current version of this open standard.

- Several modules have been contributed to the Apache Tashi [15] project. Improvements offered have included self-registration of Tashi nodes, improved schedulers and reprovisioning functionality. Some of these contributions have already been integrated into the mainline.

- JClouds [16] provides a consistent library for Java developers to manipulate arbitrary cloud providers and provisioning systems. SLA@SOI has developed a JClouds OCCI API to allow users of JClouds to manipulate OCCI-compatible provisioning systems.

- SLA@SOI has also developed and contributed an OCCI interface to OpenNebula [17], helping this provisioning system enhanced by the RESERVOIR FP7 project [18] to be SLA-enabled by the Infrastructure SLA Manager developed in SLA@SOI.
2.4 Task-level activities in 3rd project year

The Infrastructure Management work package is organized into seven tasks whose contributions, activities and progress are summarized in the following sub-sections.

A4.1 Abstract Infrastructure Framework Definition

In Year 1 this task has resulted in two models being defined: an Infrastructural model detailing infrastructure concerns from an external point of view, and a Landscape model for internal management purposes. The Year 1 prototype incorporated implementations of both these models.

In Year 2, these models have been submitted to the OGF OCCI standards working group, and have directly influenced the OGF OCCI standard as described in Chapter 5, Infrastructure Interface, of this document.

In Year 3 these models have matured, in parallel with the OGF's OCCI, and the implementations have been updated. In particular, an ANTLR [19] grammar was defined allowing for automatic generation of OCCI compatible models in multiple programming languages.

A4.2 Harmonized Interface for Heterogeneous Virtualised Infrastructure Services

In Year 1 this task resulted in a generic interface on top of heterogeneous virtualized infrastructure components being defined and demonstrated in the Year 1 prototype. This work and our experiences were also contributed to OGF OCCI standards group in Year 2 of the project. This work has been enhanced and matured in Year 3 of the project, resulting in the publication of V1.1 of the OCCI in April 2011, as described in Chapter 5, and in the implementation of the OCCI compatible Infrastructure Service Managers for both Apache Tashi and OpenNebula as described in Chapter 6.

A4.3 Infrastructure Foundations, Resource Specification and Management

In Year 1 this task involved creating a bespoke manager of arbitrary hypervisors – with both KVM and Xen hypervisors being integrated successfully. In Year 2 efforts moved to integrating a higher level third party cluster manager, Apache Tashi, and implementing various enhancements to allow it deliver SLA-enabled infrastructure. In Year 3 two significant enhancements were developed: a new flexible scheduler and additional re-provisioning support. These enhancements were submitted to Apache Tashi and some have already been integrated into the project codebase. The enhancements to Apache Tashi are described in Chapter 6.

A4.4 Monitoring, Logging and Alerting Services

In Year 1 a low level, scalable, infrastructure monitoring framework was developed. In Year 2 this framework was adapted and evolved into a Low Level Monitoring System (LLMS) compatible with and complementing the evolved architecture. In Year 3 the LLMS was expanded with a historical database. This allowed archival of historical information, and provided multiple interfaces for
interested clients to access and review historical data. Dynamic monitoring configuration, custom Tashi sensors, and abstractions allowing support for Nagios and OpenStack have also been implemented. The LLMS design and implementation are described in Chapters 3 and 6 respectively.

**A4.5 Dynamic Provisioning & Re-provisioning**

The Year 1 prototype demonstrated basic dynamic provisioning and reprovisioning. In Year 2 this functionality was ported to the evolving architecture, and logic implemented in the Infrastructure PAC, Infrastructure POC, and Infrastructure Service Manager. In Year 3 the IPAC, IPOC and I-SM were all extensively updated and enhanced, as described in Chapters 3 and 6.

**A4.6 Test Bed Implementation**

Year 1 saw a basic test bed being deployed and made accessible to the SLA@SOI consortium at large via the internet. In Year 2 this tested was expanded significantly, with a total of 64 cores and 140 GB of memory being made available. In Year 3 this testbed was expanded further, and complemented by a second testbed hosted by XLAB. This infrastructure has been used by several use cases and is described in Chapter 7.

**A4.7 SLA Service Provisioning**

This task effort began as scheduled in Year 2, and has delivered a complete reference implementation of an Infrastructure SLA Manager, including Infrastructure POC and Infrastructure POC, as described in Chapter 6, in association with work package 5 SLA Management and Foundations. In Year 3 progress included extensive upgrading of algorithms within both components. For example, computational geometry has been applied to identify optimal deployment strategies. This task has also continued to enhance the SLA Model to accommodate infrastructure concerns, helping to realize the SLA templates and models described in Chapter 4.
3 Architecture

This chapter describes the architecture of the Infrastructure Management Layer at a high level. It begins with an analysis of the requirements for infrastructure, is followed by a summary of the state-of-the-art in infrastructure management, summarises the evolution of the architecture of the Infrastructure Management components to their current state, and describes some of the deployment scenarios that this architecture supports.

3.1 Infrastructure Requirements

Both implicit and explicit requirements have been documented for Infrastructure Management in SLA@SOI. Implicit requirements have been derived from the formal description of the Infrastructure Management work package in the Description of Work [3], the deployment scenario for the SLA@SOI Adhoc Demonstrator, and the capabilities of existing Infrastructure Management offerings analysed during the state of the art review [13]. These implicit requirements have been the main driver for the architecture, implementation and reference implementation of the infrastructure components.

In parallel, explicit requirements have been documented by the various SLA@SOI Use Cases.

3.1.1 High Level Requirements

Functionally, at a high level SLA@SOI Infrastructure Management must:

- Describe the offered infrastructure to potential customers.
- Provision infrastructure according to negotiated SLAs.
- Monitor provisioned infrastructure for SLA violations, and disposition accordingly.
- Allow existing provisioned infrastructure be re-provisioned dynamically.
- Allow provisioned infrastructure to be managed by customers.

There are also important non-functional requirements for SLA@SOI Infrastructure Management:

- The SLA@SOI software must be open source, and its license and the license of third party dependencies must be legally compatible.
- Existing or maturing standards should be used where possible, with extensions proposed to the relevant standards organisations if appropriate
- The resulting framework must be scalable, potentially supporting tens of thousands of nodes.
- The resulting framework must not mandate the use of particular operating systems, middleware, hypervisors or external infrastructure providers. Generic interfaces and abstractions should be implemented where possible.
- The system must not introduce unnecessary complexity, keeping models and data structures as generic and extensible as possible. For example the model used to describe infrastructure requests should be related to the model describing the infrastructure landscape.
3.1.2 Infrastructure SLA Requirements

SLAs are at the core of SLA@SOI and Infrastructure SLAs must:

- Be compatible with the overall SLA@SOI SLA architecture.
- Completely describe infrastructure requests, including both functional and non-functional properties.
- Allow multiple VMs to be requested at once.
- Allow the precise details of individual VMs to be defined.
- Allow the relationship between VMs to be defined (e.g. if they need to be booted in a particular order).
- Allow non-functional properties to be assigned both to individual resources and to the overall provisioning request.
- Allow resources other than VMs to be described (e.g. physical machines, routers, load balancers etc.).
- Allow ranges of values as well as explicit values to be defined.
- Allow arbitrary parameters to be supported without code having to be modified.

Infrastructure SLAs should ultimately allow:

- The initial configuration of VMs to be defined.
- Monitoring and logging details to be declared.

3.1.3 Interface and Management Requirements

Infrastructure Management must expose the following functionality to consumers:

- Preregistering of software images for potential deployment in the future. These images include operating system, middleware, end-user services and configuration scripts.
- Negotiation of provisioned infrastructure – including the ability to reserve infrastructure for provisioning in the future.
- Management of instantiated infrastructure including listing, starting, stopping and adjusting the customers virtual machines. This should support dynamic re-provisioning and potentially live-migration where appropriate.
- Support a hierarchy of deployed infrastructure, e.g. clustering of resources and the federation of these clusters.

Infrastructure Management should also:

- Allow for submission of infrastructure requests in a variety of formats (e.g. XML, JSON, CIM, OVF) and via a variety of protocols (e.g. RPC, WSDL etc.).
- Provide a self-describing REST-style interface that allows the discovery and manipulation of appropriate resources.
- Limit functional leakage in the interface. For example, the management of hypervisors should be independent of the management of resources hosted by hypervisors.
- Internally, Infrastructure Management must support the definition of, storage of, and on-going compliance with overall infrastructure operating policies.
3.2 Monitoring Requirements

Infrastructure Monitoring in SLA@SOI must:

- Be compatible with the overall Monitoring architecture.
- Be able to monitor low-level infrastructure metrics including server uptime as well as cpu, memory, storage and network utilisation and performance.
- Be able to notify relevant components of appropriate events, e.g. using a pub-sub mechanism.
- Be able to warn of potential SLA violations.
- Detect SLA Violations.
- Store historical monitoring data.

Ideally, monitoring should be:

- Self-configuring. The explicit configuration of monitoring nodes will be difficult and time consuming in large-scale deployments without some degree of autonomous configuration.

3.2.1 Use Case Requirements

In terms of explicit requirements, the various SLA@SOI Use Cases have registered the infrastructure features and requirements as listed in Table 2 and Table 3.

Table 2: Infrastructure Features from SLA@SOI Use Cases

<table>
<thead>
<tr>
<th>WP</th>
<th>Feature</th>
<th>Use Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>B3</td>
</tr>
<tr>
<td>Service Provisioning</td>
<td>Virtual hardware infrastructure provisioning</td>
<td>X</td>
</tr>
<tr>
<td>SLA enforcement</td>
<td>Virtual hardware infrastructure adjustment</td>
<td>X</td>
</tr>
<tr>
<td>Service Monitoring</td>
<td>Virtual hardware infrastructure monitoring rules</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>extraction from SLA</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Virtual hardware infrastructure observation and</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>violation detection</td>
<td></td>
</tr>
<tr>
<td>Service Reporting</td>
<td>Virtual hardware infrastructure reporting</td>
<td>X</td>
</tr>
<tr>
<td>Area</td>
<td>Y3 RID</td>
<td>Summary</td>
</tr>
<tr>
<td>------------------------------------------------</td>
<td>--------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>Virtual hardware infrastructure provisioning</td>
<td>6.4.1</td>
<td>Provisioning of storage and network</td>
</tr>
<tr>
<td></td>
<td>6.4.2</td>
<td>VM persistence and backup as specified in SLA</td>
</tr>
<tr>
<td>Virtual hardware infrastructure adjustment</td>
<td>7.2.1</td>
<td>Scale virtual infrastructure up and down</td>
</tr>
<tr>
<td>Virtual hardware infrastructure monitoring rules extraction</td>
<td>9.2.1</td>
<td>Virtual hardware infrastructure monitoring rules/constraints should be derived for agreed SLAs and applied to monitoring configuration of infrastructure service</td>
</tr>
<tr>
<td>Virtual hardware infrastructure reporting</td>
<td>10.4.1</td>
<td>Virtual hardware infrastructure reporting GUI including historical graphs</td>
</tr>
</tbody>
</table>

The delivery of these requirements has been tracked throughout the lifecycle of the project. For more context and details please refer to the SLA@SOI Scientific Evaluation Report [20].
3.3 State of the Art

An extensive state of the art analysis in infrastructure management was undertaken at the start of this project. This analysis has been updated as the project has matured to reflect more recent developments in this area. As Service Oriented Infrastructures (SOIs) can be considered synonymous with Cloud Computing platforms, this analysis includes frequent references to current Cloud Computing offerings and research initiatives.

To help structure this analysis, this section is organised into the following key sub-sections:

- Infrastructure Models
- Infrastructure Virtualization
- Infrastructure Platforms
- Infrastructure Monitoring
- Infrastructure Messaging

To avoid repetition, this chapter does not go into detail on any particular technology, platform or product. For detailed reviews please refer to the overall SLA@SOI State of the Art Technical Analysis report [13].

3.3.1 Infrastructure Models

Essentially concerned with the data model by which all aspects of Infrastructure Management are described, this state of the art analysis has reviewed both draft and published infrastructure data-models, as well as the implementation-specific data-models employed in commercial and open-source Infrastructure as a Service offerings.

The key infrastructure data models reviewed included the Common Information Model (CIM) [21], issued by the Distributed Management Task Force (DMTF) [22], Open Virtual Machine Format (OVF) [23], and Open Grid Forum’s (OGF) [24] GLUE v. 2.0 [25].

CIM was observed to be an extremely comprehensive and detailed industry standard, however the size and scope of it was seen to make it too low-level and detailed for our purposes. The DMTF that manages the standard is itself currently exploring lighter-weight models to address the needs of virtualisation in particular.

OVF is one such standard, originating from a proposal from Dell, HP, IBM, Microsoft, VMware, and XenSource submitted to the DMTF. It has a direct relationship with CIM, using CIM schemas and vocabularies to help describe its schema. Although addressing functional aspects of virtual infrastructures, it does not address non-functional parameters. A more in-depth review of OVF can be seen in the Month 12 deliverable [1].

GLUE v. 2.0 is a specification, published in March 2009, with its roots in the Open Grid Forum community. Whilst GLUE could be used to describe services (atomic or composed) and the infrastructure on which they are deployed in SLA@SOI, it cannot be used immediately as drafted as it cannot accommodate SLA@SOI requirements such as capturing the geographical location of infrastructure nodes.
Regarding the data models employed by existing implementations of Infrastructure as a Service offerings, details of both the compute resource request and compute resource representation have been reviewed for Amazon EC2, Sun Cloud API, Flexiscale, ElasticHosts, GoGrid, Enomalism, OpenNebula, Slicehost, Eucalyptus, Globus Nimbus, AppNexus, and Apache Tashi. Details are provided in the Month 12 deliverable. General requirements for the SLA@SOI data models that were revealed by this analysis are documented in the previous section.

In summary, it was observed that no current or draft model currently exists that met all the needs of SLA@SOI. OVF was considered to be the most relevant model, and indeed SLA@SOI has had the potential to propose extensions to OVF to enable support for non-functional parameters that are at the core of SLA-enabled infrastructures. From a design and implementation point of view, it was foreseen that SLA@SOI should be engineered to accommodate requests for infrastructure in arbitrary formats – including at least OVF and potentially CIM and GLUE also, the latter depending on how the standard matures and is embraced by the community.

The SLA@SOI architecture was designed to support the recommendations of this analysis. Whilst potential contributions to OVF from SLA@SOI were considered, our engagement with OCCI was prioritised in order to make it as useful a standard as possible. It was designed in such a way as to support OVF descriptors. Interestingly, there have recently been some cross-standards meetings at which opportunities for integrating OCCI with other standards have been proposed. These discussions have resulted in the publication of a joint article on how Cloud standards could be integrated [26].

### 3.3.2 Infrastructure Virtualization

Infrastructure virtualisation can be considered to refer to the abstraction of physical compute resources into virtual (software based) equivalents – virtual machines. Fundamentally, virtual machines allow particular hardware to be emulated, allowing for example several types of virtual servers to be hosted by one physical machine. Various levels of infrastructure virtualization are possible.

The term full virtualization describes when a complete layer of hardware is simulated, allowing an unaltered operating system to run in complete isolation of the physical hardware on which it is hosted.

In para-virtualization, the hosted operating system is specially modified to redirect appropriate low-level operations (e.g. network operations) to the hypervisor on the host physical machine. This allows potentially compute-expensive operations to be processed in the most efficient manner.

In operating system virtualization, the virtualization is done within the host operating system, providing isolated virtualised operating system environments (named containers in Linux) to the applications it is hosting.

Infrastructure virtualisation remains a very active topic in the industry. It spans both software and hardware technologies, with open-source and proprietary approaches that can be hosted privately by an organisation, or remotely by a
third party. To help abstract the virtual machines in which host operating systems and applications can be run, various standards and frameworks are being developed.

Sponsored by Redhat, Libvirt [25] is an open-source API that provides a generic way to interact with different types of open source virtualization technologies (including Xen [28] and KVM [29]). It allows the complete life cycle of these VMs to be managed independently of the underlying virtualisation technology, allowing a project like SLA@SOI to focus on higher level concerns.

OpenNebula [17] is, as its website describes, an open-source distributed VM manager that enables the dynamic placement of VMs on a pool of physical resources. With this level of abstraction, even the physical location of the server hosting the VMs is abstracted. Whilst not relevant for all envisioned use-cases of SLA@SOI, there are numerous aspects of infrastructure virtualisation management such as federation and scheduling that SLA@SOI could learn from and thus integration with OpenNebula at an appropriate level should also be considered.

NASA and Rackspace have combined forces to produce a family of open-source cloud computing enablers called OpenStack [30]. Currently three projects, OpenStack includes OpenStack Compute for provisioning required virtual machines, Object Storage which provides for decentralised, redundant storage of binary objects, and Image Service, which allows a catalogue of virtual machine images to be managed. OpenStack is receiving significant attention in industry and is an increasingly active project. Although SLAs are not explicitly comprehended in the current version, future plans include the possibility of adding SLA support.

The SLA@SOI architecture has been designed to support arbitrary virtualisation technologies, with the OCCI interface specifically conceived to provide a generic interface to heterogeneous virtualisation providers. As part of our collaboration with RESERVOIR we developed explicit plans for a proof of concept to demonstrate the SLA@SOI framework SLA-enabling OpenNebula. This work is presented in Section 6.9.

### 3.3.3 Infrastructure Platforms

Whilst locally-hosted service oriented infrastructures are relevant and necessary for some organisations, many have generic hosting requirements that could be accommodated by external providers over the internet. An analysis of the functionality they provide is thus very relevant for SLA@SOI, and a review of the APIs of the major infrastructure providers has been undertaken and documented in year 1 of the project. Amazon EC2, Sun Cloud API, Flexiscale, ElasticHosts, GoGrid, Enomaly – Enomalism, OpenNebula, Slicehost, Globus Numbus, Eucalyptus AppNexus, F5.com, Apache Tashi and CohesiveFt were all reviewed.

This review of the IaaS platform providers has helped identify key generic requirements of the API that SLA@SOI Infrastructure Management has since architected for and implemented, and has also delivered an understanding of
some of the explicit third-party implementations that SLA@SOI should be able to integrate with. After deciding to adopt a third party infrastructure management framework for our year 2 implementation, this analysis helped us select Apache Tashi as the first infrastructure platform to integrate. This was primarily due to the favourable licensing, and the opportunity and willingness of our contacts in Apache Tashi to accept any appropriate software enhancements that we may be in a position to contribute into their codebase. Subsequently OpenNebula was also integrated into the software stack.

At the conclusion of this project, it is interesting to note that it remains the case that none of the platforms reviewed offer holistic SLA-awareness. Continuing to be a topical subject, several conversations are ongoing with third-party providers such as SensibleCloud [31], and have started with research projects such as FWARE [32] and Contrail [33], to help integrate the results of SLA@SOI into their platforms.

### 3.3.4 Infrastructure Monitoring

A key part of any SLA-aware system is the monitoring of the performance of the system – without it there is no awareness. The state-of-the-art analysis for monitoring technologies involved a review of several key frameworks and technologies in this area including EVEREST, Ganglia, Nagios, Groundwork, MonALISA and Zabbix. Although Nagios, Groudwork, MonALISA and Zabbix all have interesting aspects, from a low-level infrastructure point of view the most relevant framework continues to be Ganglia [34].

Infrastructure monitoring brings its own unique challenges which Ganglia has been optimised to address: a plethora of metrics that may need to be monitored, a huge variety of hardware, operating systems and middleware for which metric drivers may be required, as well as the potentially massive distribution and scale of modern cloud computing infrastructures. Ganglia is a scalable distributed monitoring system designed for high-performance computing systems. It is built upon open and widely used technologies, and it uses carefully engineered data structures and algorithms to achieve very low per-node overheads and high concurrency. An open-source and extensible monitoring framework, Ganglia is being continuously improved and remains the industry-standard in this field.

Ganglia has been adopted as the default lowest level monitoring framework on which our SLA low-level monitoring system has been built. System administrators will be familiar with the capabilities it offers, and the breadth of monitoring data that it can expose for SLA Monitoring purposes.

However, Ganglia is not the only infrastructure monitoring system supported by SLA@SOI. The project has implemented a generic interface into the instrumentation layer, and the SLA@SOI-developed infrastructure monitoring system has also been demonstrated to work on top of Nagios [35].

### 3.3.5 Infrastructure Messaging

To deliver a truly scalable SLA-aware infrastructure layer it was clear that a traditional RPC-style interface would soon run into issues and so the potential of various messaging protocols was examined as part of the state-of-the-art review.
XMPP, the Extensible Messaging and Presence Protocol [36], is probably best known as the standard enabling the Jabber chat platform. However, XMPP provides a mature, distributed, comprehensive and highly extensible messaging protocol with many features relevant to SLA@SOI infrastructure management. Particularly interesting extensions include Adhoc-Command and IO-Data XEPs. The powerful pub/sub pattern scales well and Multi-User Chatroom functionality could allow for powerful debugging via command-line interaction.

JMS, the Java Message Service [37], is a Java Message Oriented Middleware (MOM) API for sending messages between two or more clients. JMS is a part of the Java Platform, Enterprise Edition, and is defined by a specification developed under the Java Community Process as JSR 914. JMS also provides a pub/sub model, but is targeted at applications developed using the Java programming language so less generic than a protocol like XMPP.

AMQP, the Advanced Message Queuing Protocol [38], is an open-standard wire-level messaging protocol for message oriented middleware. It was initially designed to support enterprise messaging requirements and addresses message orientation, queuing, routing (including publish-subscribe), reliability and security. Version 1.0 publication is imminent and AMQP is gathering significant interest in industry.

Based on this analysis it was determined that XMPP offered the most appropriate messaging solution for initial implementations. However, with the maturing of AMQP it was observed that the messaging layer in SLA@SOI should be implemented in such a fashion that the messaging protocol can be easily replaced. The current SLA@SOI infrastructure implementation has adopted this layered approach.
3.4 Overall Architecture

The architecture for Infrastructure Management evolved as requirements emerged and the overall architecture of the SLA@SOI framework matured, but by the end of the second year of the project had stabilised as documented in the year two version of deliverable D.A4a [2]. The final top-level architecture is illustrated in Figure 4 and detailed in Deliverable D.A1a Framework Architecture [5].

![Figure 4: Top level Architecture](image)

From an infrastructure point of view the key components of this top level architecture are the Infrastructure SLA Manager, the Infrastructure Service Manager, and the deployed infrastructure services.

Year 2 saw the transformation of the initial Infrastructure architecture into the refined top-level architecture of Figure 4. This process has concluded in Year 3, with enhanced functionality and additional layers of abstraction added at all levels in the stack.

The individual architectures of these components are introduced in the following sub-sections.
3.5 Infrastructure SLA Manager

The **Infrastructure SLA Manager (I-SLAM)** is an instantiation of the **Generic SLA Manager (G-SLAM)** as described in Deliverable D.A5a [8], tailored to infrastructure services. The I-SLAM does not try to address each exotic feature of each possible infrastructure service, but rather focuses on the most important features typically offered by Infrastructure as a Service (IaaS).

The G-SLAM includes all those features necessary for the full lifecycle management of SLAs. It provides:

- Negotiation mechanics via an extensible Protocol Engine and interoperable Syntax Converters;
- Persistency for SLAs and SLA Templates via the two respective registries;
- A publish/subscribe system for advertising SLA templates, thus enabling service discovery based on both functional and non-functional properties; and
- A flexible definition of per-SLA monitoring frameworks, through a generic Monitoring Manager.

The G-SLAM also includes two components which are considered to be domain/use-case specific; the **Planning/Optimization Component (POC)** and the **Provisioning/Adjustment Component (PAC)**. These two components are expected to be implemented by interested parties, and to replace the default placeholders. The I-SLAM is, ultimately, a G-SLAM with a custom POC & PAC, targeting the least common denominator of IaaS. Figure 6 illustrates a high-level overview of the I-SLAM architecture, as it extends a G-SLAM (more information about the latter can be found in Deliverable D.A5a).

---

**Figure 5: G-SLAM architecture**

As Figure 5 illustrates, the G-SLAM includes all those features necessary for the full lifecycle management of SLAs. It provides:

- Negotiation mechanics via an extensible Protocol Engine and interoperable Syntax Converters;
- Persistency for SLAs and SLA Templates via the two respective registries;
- A publish/subscribe system for advertising SLA templates, thus enabling service discovery based on both functional and non-functional properties; and
- A flexible definition of per-SLA monitoring frameworks, through a generic Monitoring Manager.

The G-SLAM also includes two components which are considered to be domain/use-case specific; the **Planning/Optimization Component (POC)** and the **Provisioning/Adjustment Component (PAC)**. These two components are expected to be implemented by interested parties, and to replace the default placeholders. The I-SLAM is, ultimately, a G-SLAM with a custom POC & PAC, targeting the least common denominator of IaaS. Figure 6 illustrates a high-level overview of the I-SLAM architecture, as it extends a G-SLAM (more information about the latter can be found in Deliverable D.A5a).
Figure 6: I-SLAM Architecture

As it can be seen from this Figure, the I-SLAM interacts externally with:

- The Business (SLA) Manager from WP A2 (described in detail in Deliverable D.A2a [6]), from which it receives policies and business-specific customizations related to negotiation, while it also provides runtime status information used to make business decisions; and
- The Infrastructure Service Manager, which provides resource information, and controls resources in general (reservations, initialization, runtime management).

In general terms, POC is responsible for assessing and customizing the infrastructure SLA from the customer. It evaluates the feasibility of provisioning the requested service by considering the availability of the service as well as other Quality of Service (QoS) terms, and plans the optimal service provisioning and monitoring strategies.

The PAC has, as its name suggests, a double-faced mission. On one hand, it plays a crucial role at provisioning time, effectively instructing the Service Manager to provision the optimized provisioning plans provided by the POC. On the other hand, the SLA enforcement ensures continual identification, monitoring and reviewing of the optimally agreed levels of services as required by the business, and ensures that given SLAs are fulfilled such that previous estimates on the performance should be equal to the final performance of the running services. To this end, the PAC dynamically readjusts the service in cases where the required quality levels are not being met.

3.6 Infrastructure Service Manager

The Infrastructure Service Manager (I-SM) is the component responsible for exposing the SLA@SOI Service Manager Interface onto the Infrastructure Provisioning System. It does not need to concern itself with SLA concerns. Fundamentally, it understands the native interface of the Provisioning System, and transforms calls from the Infrastructure SLA Manager into appropriate calls for the specific provisioning system being manipulated. The I-SM is also responsible for implementing any logic necessary to SLA-enable the Provisioning System which is not already included in the Provisioning System.
In the SLA@SOI Infrastructure architecture, two reference I-SM’s have been implemented. An I-SM Proxy has been implemented as an OSGI bundle to provide for seamless interoperability and connectivity with the I-SLAM. This I-SM Proxy then communicates with the I-SM proper, which is standalone and directly connected to the provisioning system.

To demonstrate the practicalities and usefulness of an abstracted infrastructure interface, the communication between I-SM Proxy and I-SM proper is through OCCI, a REST-ful interface.

To make the OCCI calls, the I-SM Proxy invokes a JClouds [40] OCCI API developed specifically by SLA@SOI for this purpose. This will allow Java developers to easily communicate with OCCI providers.

The implementation of the I-SM proper is dependent on the implementation of the provisioning system being manipulated. In a reference implementation on top of Apache Tashi, the I-SM is a Grails application written in Groovy that communicates with Tashi through an XMPP messaging bus. As Apache Tashi does not support reservations by default, that logic is built into the I-SM. To minimize duplication and provide for system consistency, the Apache Tashi and the Apache Tashi I-SM share the same back end database. See Section 6 for more information.

### 3.7 Infrastructure Monitoring

The scope and position of Monitoring in the general architecture is described in D.A1a Framework Architecture [5]. The infrastructure monitoring implementation employs the generic Monitoring Manager within the I-SLAM, with the lower layers customized as per the architecture. The monitoring calls to and from the IPAC and IPOC are passed inside OCCI invocations, through the I-SM and into the Infrastructure Monitoring Agent.

The Infrastructure Monitoring Agent includes several components: the Service Registration Manager, Metrics Gatherer, Data Store, Monitoring Data Provider, Infrastructure Reporting and finally the QoS Computation and SLA Compliance Evaluation component.

The Infrastructure Monitoring Agent manipulates the infrastructure sensors through an abstracted layer. This allows arbitrary instrumentation systems to be employed, and both Ganglia and Nagios have been successfully integrated.

Besides SLA monitoring related to the services provisioned, infrastructure monitoring also addresses the monitoring of available resources. This information is used for infrastructure optimization purposes.

### 3.8 Deployment Scenarios

The current architecture of the infrastructure management layer is flexible enough to support many deployment scenarios. The actual scenario adopted will depend on how many provisioning systems are in scope, at what levels SLAs are relevant, and the domain or domains in which the provisioning systems are deployed.

Some supported high level deployment scenarios are illustrated in Figure 7. Scenario (a) illustrates a typical deployment – one Provisioning System is managed by one Infrastructure Service Manager which is managed by one Infrastructure SLA Manager. Should two Provisioning Systems need to be managed simultaneously, the deployment in scenario (b) can be considered. If multiple provisioning systems are accessible, but not necessarily within the same
domain, the deployment in scenario (c) is supported. Should the local provisioning system not be able to meet the needs of a provisioning request, the Infrastructure SLA Manager has the ability to negotiate with a remote SLA Manager to try to delegate some or all of the request.

**Figure 7: High-level deployment scenarios supported by the architecture**

The Infrastructure Management work package prototype implements scenario (a). Thanks to the addition of the OCCI abstraction layer, both Apache Tashi and OpenNebula provisioning systems can be SLA-enabled by the same I-SLAM.

The architecture is also flexible enough to allow various implementation approaches to be adopted as illustrated in Figure 8. Either generic or provisioning system specific components can be implemented depending on whether the priorities are on minimal development effort or maximum performance and optimisation.
The Infrastructure Management prototype implements approach (b), and two OCCI compatible I-SM’s have been implemented as initially presented in a joint technical report authored by SLA@SOI and RESERVOIR partners [39]. The I-SMs add provisioning system specific reservation management and scheduling functionality, features required for SLA-enabling.

Indeed, OCCI opens up the possibility of interesting brokerage scenarios for the Infrastructure SLA Manager should it be aware of multiple OCCI-compliant infrastructure services.

Figure 8: Implementation approaches supported by the architecture
4  Infrastructure SLAs

At the core of SLA@SOI are machine readable SLAs. A comprehensive, detailed and extensible SLA model has been developed and is documented in Deliverable D.A5a [8].

This model has been influenced by contributions from the Infrastructure Management work package, enabling realistic Infrastructure SLAs to be represented. This chapter describes only the infrastructure-specific vocabulary additions, and presents some reference SLAs for both single and multiple virtual machine requests. This chapter displays SLAs in BNF form. An XML version is available in Appendix C.

4.1  Infrastructure Vocabulary

Several terms have been specified as part of the SLA Vocabulary that are specifically related to infrastructure. These include vm_cores, cpu_speed, memory, persistence and vm_image.

The namespace for these terms is http://www.slaatsoi.org/resources# and the formal definitions of these terms are reproduced below.

- **infra:vm_cores**
  - expression: vm_cores(r : RESOURCE_TYPE_SERVICE): COUNT
  - definition: denotes the number of cores of r

- **infra:cpu_speed**
  - expression: cpu_speed(r : RESOURCE_TYPE_SERVICE): FREQUENCY
  - definition: denotes the processing speed of r

- **infra:memory**
  - expression: memory(r : RESOURCE_TYPE_SERVICE): DATA_SIZE
  - definition: denotes the amount of memory of r

- **infra:persistence**
  - expression: persistence(r : RESOURCE_TYPE_SERVICE): BOOLEAN
  - definition: denotes whether the virtual machine image of r is persisted between instantiations

- **infra:vm_image**
  - expression: vm_image(r : RESOURCE_TYPE_SERVICE): UUID
  - definition: a reference to the virtual image with which r should be instantiated

4.2  Example Infrastructure SLAs

4.2.1  ORC Gold SLA: Single VM Provisioning

The following example illustrates a basic SLA that specifies the provisioning of a single virtual machine with guaranteed states for both Functional Properties (FPs) and Non Functional Properties (NFPs). The SLA contains two different types of Agreement Terms. Reliability is an NFP, whilst Performance of the VM is a FP.
Table 4: ORC Gold SLA

sla{
    agreedAt = Fri Jan 01 12:00:00 GMT 2010
    effectiveFrom = Wed May 05 12:00:00 BST 2010
    effectiveUntil = Sat May 05 12:00:00 BST 2012
    templateId = ORC_InfrastructureSLATGold
    uuid = ORC_InfrastructureSLAGold
    sla_model_version = sla_at_soi_sla_model_v1.0
}

/* ---- PARTY DESCRIPTIONS -------------------------------------------- */

party{
    id = SLASOIProvider
    role = provider
}

party{
    id = BOB
    role = customer
}

/* ---- INTERFACE DECLARATIONS---------------------------------------- */

interface_declr{
    id = VM_Access_Point
    provider_ref = http://www.slaatsoi.org/slamodel#provider
    interface_resource_type{
        name = VM
    }
}

/* ---- VARIABLE DECLARATIONS----------------------------------------- */

/* ---- AGREEMENT TERMS----------------------------------------------- */

agreement_term{
    id = Reliability
    guaranteed_state{
        id = MTTRState
        priority = ""
        mttr(VM_Access_Point) < "3" hrs
    }
    guaranteed_state{
        id = MTTFState
        priority = ""
        mttf(VM_Access_Point) > "455000" hrs
    }
}

agreement_term{
    id = Performance
    VM_IMAGE_VAR is member_of [http://www.intel.ie/ORC_images/ORC.xml]
    guaranteed_state{
        id = VM_CORES
        priority = ""
        vm_cores(VM Access_Point) = "4"
    }
}
In this particular SLA, the Agreement Term **Reliability** refers to NFPs that need to be taken into account during the lifecycle of the SLA. Specifically these properties define the mode of operation or a quality of the resources during runtime.

### NFP Agreement Term: Reliability

<table>
<thead>
<tr>
<th>Guaranteed State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean time to recovery (MTTRState)</td>
<td>3 hours</td>
</tr>
<tr>
<td>Mean time to failure (MTTRState)</td>
<td>455000 hours</td>
</tr>
</tbody>
</table>

The functional properties (FPs) of the Agreement Term **Performance** are as follows.

### FP Agreement Term: Performance

<table>
<thead>
<tr>
<th>Guaranteed State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image (vm_image)</td>
<td><a href="http://www.intel.ie/ORC_images/ORC.xml">http://www.intel.ie/ORC_images/ORC.xml</a></td>
</tr>
<tr>
<td>Speed (cpu_speed)</td>
<td>2.5 Ghz</td>
</tr>
<tr>
<td>Memory (memory)</td>
<td>4096 Mb</td>
</tr>
<tr>
<td>Cores (vm_cores)</td>
<td>4</td>
</tr>
</tbody>
</table>

The SLA also details several metadata fields such as identifiers, time periods, and provider/customer details.

### 4.2.2 Multiple VM (Dynamic Provisioning)

The previous example of the ORC GOLD SLA has illustrated how a single VM SLA aware provisioning can be defined in terms of an SLA. The following example
illustrates a more complex scenario with an SLA that allows to dynamically configure (at negotiation time) different types of VM configurations from a predetermined range of VMs and the amount (count) of VMs that will be provisioned.

The Agreement Term defined in Table 5 defines an agreement terms for a VM type VM_0.

Table 5: Agreement Term for VM_0

```
agreement_term{
    id = autogen
    VM_0 is subset_of(VM_Access_Point)
    START_TIME is "effectiveFrom" = "2002-05-30T09:00:00" xsd:dateTime
    END_TIME is "effectiveUntil" = "2002-05-30T09:00:00" xsd:dateTime
    VM_QUANTITY_VAR is =("10" xsd:integer)
    VM_ISOLATION_VAR is =("false" xsd:boolean)
    VM_LOCATION_VAR is =("IE" xsd:string)
    VM_CORES_VAR is =("8" xsd:integer)
    VM_CPU_SPEED_VAR is =("2.6" GHz)
    VM_MEMORY_SIZE_VAR is =("1024" Mb)
    VM_PERSISTENCE_VAR is =("true" xsd:boolean)
    VM_IMAGE_VAR is =("ubuntu-9.10-32" xsd:anyURI)
    guaranteed_state{
        id = VM_COUNT
        count(VM_0) = VM_QUANTITY_VAR
    }
    guaranteed_state{
        id = VM_ISOLATION
        isolation(VM_0) = VM_ISOLATION_VAR
    }
    guaranteed_state{
        id = VM_LOCATION
        location(VM_0) = VM_LOCATION_VAR
    }
    guaranteed_state{
        id = VM_CORES
        vm_cores(VM_0) = VM_CORES_VAR
    }
    guaranteed_state{
        id = VM_CPU_SPEED
    }
}
```
cpu_speed(VM_0) = VM_CPU_SPEED_VAR

} 

guaranteed_state{
    id = VM_MEMORY_SIZE
   memory(VM_0) = VM_MEMORY_SIZE_VAR

}

guaranteed_state{
    id = VM_PERSISTENCE
   persistence(VM_0) = VM_PERSISTENCE_VAR

}

guaranteed_state{
    id = VM_IMAGE
   vm_image(VM_0) = VM_IMAGE_VAR

} 

}
The FPs of the Agreement Term for VM_0 are:

**FPs Agreement Term: VM_0 (Virtual MachConfiguration 0)**

<table>
<thead>
<tr>
<th>Guaranteed State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Image vm_image(VM_0)</td>
<td>ubuntu-9.10-32-(32 bit)</td>
</tr>
<tr>
<td>Speed cpu_speed(VM_0)</td>
<td>2.6 Ghz</td>
</tr>
<tr>
<td>Memory (memory)</td>
<td>1024 Mb</td>
</tr>
<tr>
<td>Cores vm_cores(VM_0)</td>
<td>8</td>
</tr>
<tr>
<td>Count count(VM_0)</td>
<td>10</td>
</tr>
</tbody>
</table>

Count refers to the number of VMs of this type that we would like to provision.

The VM_0 Agreement Term also specifies a set of NFPs:

**NFPs Agreement Term: VM_0 (Virtual MachConfiguration 0)**

<table>
<thead>
<tr>
<th>Guaranteed State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Isolation isolation(VM_0)</td>
<td>False</td>
</tr>
<tr>
<td>Location location(VM_0)</td>
<td>Ireland</td>
</tr>
<tr>
<td>Persistency persistence(VM_0)</td>
<td>True</td>
</tr>
<tr>
<td>Availability (availability)</td>
<td>&gt; 95 %¹</td>
</tr>
</tbody>
</table>

VM_1 can be defined in a similar way except that the values of the guaranteed states are different. In our example VM_1 could specify 8 VMs, with 4 cores, 2.6Ghz speed, 512 Mb of ram, with a Ubuntu 64 bit image. The full SLA specification can be found in Appendix C.

¹ Specified as a global guarantee term
5 Infrastructure Interface

5.1 Introduction

In order to operate with heterogeneous management systems and their related and sometimes very different resource offerings, it was considered essential to offer an abstract interface and related data model as the primary interface to the I-SM. To this end, the appropriate infrastructure resource types and features were identified, modeled and exposed at an abstract level. This allowed for a harmonised, implementation neutral interface to the infrastructure to be defined. The Infrastructure Management's abstract interface and model is embodied in OCCI. More dynamic aspects are being addressed in ongoing work on monitoring and SLA specific extensions to the OCCI standard.

5.2 Open Cloud Computing Interface (OCCI)

OCCI is a RESTful protocol and API for the management of cloud service resources. It comprises a set of open community-lead specifications delivered through the Open Grid Forum. OCCI was originally initiated to create a remote management API for IaaS model based services. It has since evolved into a flexible API with a strong focus on integration, portability, interoperability and innovation while still offering a high degree of extensibility.

OCCI has, since almost its inception, been driven and contributed to by SLA@SOI. Most noticeably, the A4 year two deliverable [2] contributed heavily to what is now OCCI version 1.1. This version of the specification is split into three documents, two of which were published in April 2011, and the remaining one in June.

OCCI aims to leverage existing open standard specifications and integrate those that already comprehensively address a required feature of OCCI. For example, when CDMI and OVF are combined with OCCI, it provides a comprehensive profile for open and interoperable infrastructural cloud services [26].

The main design foci of OCCI have been:

- **Flexibility**: enabling a dynamic, adaptable model.
- **Simplicity**: avoiding mandating a large number of requirements for compliance with the specification. Focusing on providing the lowest common denominator in terms of features and then allowing providers to supply their own differentiating features.
- **Extensibility**: enabling providers to specify and expose their own individual service features that are discoverable and commonly understood (via the core model).

The specification itself is currently comprised of 3 modular parts:
Core [41]: This specifies the basic types in the standard, and presents them through a meta-model. It is this specification that dictates the common functionality and behaviour that all specialisations of it must respect. It specifies how extensions may be defined.

Infrastructure [42]: This specification is an extension of Core, and provides a practical example of how other parties can create extensions. It defines the types necessary to provide basic infrastructure as a service offerings. Of specific note with regard to SLA@SOI involvement, the majority of this specification was authored by the Infrastructural Management work package.

HTTP Rendering [43]: this document specifies how the OCCI model is communicated both semantically and syntactically using the REST-ful architectural-style.

From an architectural point of view OCCI sits on the boundary of a service provider as illustrated in Figure 9. It does not seek to replace the proprietary protocols or APIs that a service provider may have already provided.

The main capabilities of OCCI are:

- Definitions (attributes, actions, relationships) of basic types:
  - Compute: defines an entity that processes data, typically implemented as a virtual machine.
  - Storage: defines an entity that stores information and data, typically block-level devices, implemented with technologies like iSCSI and AoE.
  - Network: defines both client (network interface) and service (OSI Layer 2 and 3 switches) networking entities, typically implemented with software defined networking frameworks.

- Discovery system enables discovery of types and their instances' URL schema. Providers can dictate their own schema. Extensions are also
discoverable through this system. In OCCI this is known as the query interface. Complete details can be found in the Core specification.

- **Extension Mechanism** allows service providers to expose their differentiating features. Clients learn of these extending features through the discovery system. The Core specification details how extensions are made. The OCCI Infrastructure specification is itself an OCCI Core extension.

- **Resource (REST) handling (CRUD)** of individual and groups of resource instances. This is detailed in the HTTP Rendering specification.

- **Tagging & Grouping of Resources**. This allows both providers and, optionally, clients to organise their resource instances into groups. An instance can be associated with one or more tags. These tags are implemented using the OCCI Mixin mechanism described in the Core specification.

- **Dynamic Composition** allows for the runtime addition of new attributes and functional capabilities. This is also implemented using the OCCI Mixin mechanism. Networking in the infrastructure specification is implemented using Dynamic Composition. In this instance the default is to offer a networking device (e.g. virtual switch) that offers Layer 2 networking capabilities. Should a provider wish to offer Layer 3 capabilities (e.g. TCP/IP etc) then this can be provided through the IPNetworking Mixin.

- **Template support** for both operating systems and resource types. In the former, providers can offer templates, implemented through the Mixin mechanism, that specify various different OS offerings (e.g. Windows, Ubuntu etc). In the latter, again with the Mixin mechanism, template resources can be offered. This enables functionality similar to that offered by Amazon EC2 regarding various sizes of virtual machines.

- **Independence from provisioning system**. OCCI does not make any assumptions about the underlying provisioning systems and hypervisors.

The current release of OCCI is designed with additional models to IaaS in mind including, for example, PaaS. A book chapter [44] has been published detailing how various PaaS type offerings can be supported through the OCCI Core model.

OCCI also has substantial and growing open source software adoption, with many implementations [45] and a number of supporting tools [46]. SLA@SOI open source contributions include:

- **OCCI implementation**: The infrastructure service manager for Apache-Tashi is detailed in Section 6.6 and for OpenNebula in Section 6.9.

- **OCCI ANTLR Grammar**: This grammar’s generated lexer and parser enables the validation and extraction of values of OCCI requests and responses. More details are presented in Section 6.6.

- **OCCI JClouds client**: This is an OCCI provider that allows users of the common JClouds client Java library to interact with an OCCI service implementation. More details are provided in Section 6.5.
OCCI has been recommended by the UK G-Cloud initiative [47], is currently in the process of consideration by NIST [48] in the US and also supported by the SIENA [49] and EGI [50] initiatives in the European Union. OCCI has received significant contributions from European Commission FP7-funded projects including both RESERVOIR and SLA@SOI. Forthcoming extensions to the specification include:

- **Monitoring**: this extension is being developed by IOLanes [51] and SLA@SOI through collaboration. The current status of this collaboration can be found in the project’s Collaboration Report [52]. Briefly, the monitoring API is defined mainly as an OCCI Mixin and is very flexible in terms of configuration of monitored metrics. The work is on-going and will be sustained through IOLanes and possibly FI-Ware.

- **SLA capabilities**: This specification describes how OCCI can be extended to support mechanisms for Monitoring and SLA agreement negotiation. It is being developed through the German Government funded DGSI project [53] along with SLA@SOI contributions. It is a first step to creating extensions to the current version of the OCCI specification (Version 1.1) which enable OCCI based services to offer these features. The work is influenced by WS-Agreement, WS-Negotiation as well as the SLA models and negotiation work from SLA@SOI. This specification will leverage the monitoring specification currently under development. SLA templates are defined as OCCI resource templates, an agreement is modelled as an OCCI Kind, and once an agreement is established, that agreement and the related resource instances are linked using a variant of OCCI Link, called an AgreementLink. By association, the specification exposes this in a REST-ful manner and borrows approaches as described by Kübert and Katsaros [54].
6 Implementation

6.1 Infrastructure SLA Manager

The Infrastructure SLA Manager is an instance of the Generic SLA Manager with two infrastructure specific components, the Infrastructure Planning and Optimization Component (IPOC) and Infrastructure Provisioning and Adjustment Component (IPAC). These two components are detailed in the following sections.

6.2 Infrastructure Planning and Optimization Component

The infrastructure layer is the basis on which the project’s multi-layer SLA hierarchy for IT services is established. The infrastructure layer will typically compare the infrastructure resource requests within incoming SLA requests, with the actual infrastructure resources, and then make an optimal plan on what resources to commit to this SLA.

As per the achievements in the first two years of the project, the Infrastructure Planning and Optimization Component (IPOC) receives requests for infrastructure, queries the infrastructure service manager for potential provisioning solutions, selects and reserves the optimal one and requests the Infrastructure Provisioning and Adjustment Component (IPAC) to provision the selected plan as appropriate. If local resources cannot satisfy the request (e.g. due to lack of availability or specification discrepancies), the infrastructure planning and optimization component can attempt to outsource to third party providers, in order to satisfy the request.

However, the feasibility of advanced reservation in the IaaS scenario was not previously fully addressed. This functionality has been integrated in the third year of the project. We introduce an SLA-based advanced reservation methodology by using computational geometry, which is able to verify, record and manage the infrastructure resources efficiently. Based on that model, the service provider can easily verify the quality of service (resource availability and service time span). This allows the feasibility for satisfying the customer's request to be determined. Furthermore, a flexible alternative solution can be generated as a counter offer to the customer, in cases where the service provider lacks resources. Therefore the model on one hand increases the utilization of the resources and attempts to satisfy as many customers as possible, while on the other hand strengthens the reputation of the service provider via increased probability to honour the established SLAs.

Problem Statement

The problem that this work seeks to solve is how to efficiently represent the advanced reservations in the IaaS scenario in a way that:

- All reservations are represented in a context. This context varies dynamically when new reservations are set up and old reservations expire.
- A decision maker, i.e. Infrastructure SLA manager, can decide the feasibility of incoming requests, in the form of SLAs, for reservation of resources by taking service availability and reliability into account. If the request is deemed feasible, a new reservation is added into the context.
A search engine can offer alternative solutions by taking required availability, reliability as well as flexible time interval into account.

While settling down a reservation into the context, the fragments of the server are traced and managed.

**Problem Modeling**

The following notations will be adopted in later sections:

- $S_i$, where $i > 0$: It represents a physical server. It contains a certain number of CPUs, gigabyte(s) of memory and hard disk.
- $T_i$, where $i > 0$: A time point in the future on one server. Its unit could be either coarse-grained (e.g., by day) or fine-grained (e.g., by hour).
- $P_i$, where $i > 0$: It is a point that represents a time interval on one server which has the consistent infrastructure resource configuration. It is equal to a fragment.
- $R_i$, where $i > 1$: A request from a customer. It is a tentative point. If an agreement is finally reached by both parties, a request point will be distributed into one or more points.
- **Basic Unit (BU)**: We assume that $S_1$ has configuration of 4 Quad-Core Intel Xeon and 24 GB memory and 2400 GB hard disk. The quantification can be realized simply by setting a basic unit with fixed configuration, namely, each BU has 1 core, 1.5 GB memory and 150 GB hard disk. In this case, $S_1$ can provide, at a maximum, 16 BUs.

**Computational Geometry Representation**

Figure 10 (a1) and Figure 11 (b1), (c1) represent reservation histogram scheduling. Several requests from customers are scheduled on $S_1$, where x-coordinate indicates time $T_i$ and y-coordinate indicates quantification of $S_1$. Therefore, customers can customize the VM configuration by setting the number of BU according to their preferences.

Our approach uses concepts from computational geometry to represent the time intervals corresponding to periods as points on a plane, as illustrated in Figure 10 (a2) and Figure 11 (b2), (c2). The x-coordinate indicates ending time and the y-coordinate indicates starting time. Since the ending time of a service is greater than the starting time, all points will always be above the diagonal. The whole server is represented as a plane, in which points and segments lie. Each point indicates a period of time over which the server has a consistent and continuous available resource configuration. The configuration information is dynamically updated according to the new requests. The more the points in the plane, the higher is the degree of the server's partitioning. Therefore, fragmentation means that a server contains a number of time slices throughout its reservation time. The virtual resource configuration in each time slice differs from its neighbouring slices.
Figure 10: Computational geometry representation

In Figure 10 (a1), a customer sends request R1 for 4 BUs as VM configuration in a time span from T1 to T2. It is apparent that S1 is capable of satisfying R1. Meanwhile, S1 is partitioned to be three fragments with respective available resource quantities: fragment F1 with 16 BUs, F2 with 12 BUs and F3 with 16 BUs. We can map the reservation histogram in Figure 10 (a1) to a computational geometry representation in Figure 10 (a2). There, R1 is a tentative point when two parties reach an agreement and is finally merged to be P1, P2 and P3.

Then, in a time span from T2 to T3, R2 arrives for 9 BUs as VM configuration and is handled successfully by S1. This leads to an update of fragment F3 with available resources of 7 BUs and a new fragment F4 with 16 BUs. Consequently, in Figure 10 (a2), a new point P4 is created through R2. The similar approach for handling R3 in Figure 11 (b1), (b2) leads to a new point P5.

Figure 10 and Figure 11 illustrate how these simple VM requests can be mapped into a computational geometry context. A point on the plane represents a one-dimensional attribute, namely the number of available BU for one VM.

Our approach applies not only to the above situation but also to flexible VM configuration, because an accepted request will dynamically impact its related points, inside which the server information at that specific time interval will also be updated respectively. In this case, a point on the plane represents three-dimensional attributes, namely the available cores, memory and hard disk sizes.

Therefore, in our experimental part, SLA requests are generated randomly with various VM configurations.
Figure 11: Computational geometry representation of different plans

**Implementation**

In Figure 12, class diagram is given, where we can see how the context plane is constructed and maintained.

- `<Segment>` represents the segment between starting point and ending point.
- `<RequestPoint>` represents the details information about the VMs of customer; it is a tentative point in the context at the very beginning.
- `<ReservationPoint>` is a reserved point, which inherits all the features from `<RequestPoint>` and represents a valid service that will be delivered in the future to the customer. Both points contain a `<VirtualMachineConfig>`.

The logic part of the implementation is `<ResourceManagementPlane>`, it could add and delete the point, segment. When the context mismatches the request from customer with its own resource pool, it will try to move the point and get the alternative solutions.
In computational geometry context, finding an alternative solution is equal to finding a proper location for R4 in Figure 13 (d1) (d2). In (d2), an alternative solution planner is introduced that moves request point R4 on the track of the segment which goes through the centre of R4 and also parallel to the diagonal, because all points on that track always have the identical time interval as R4’s. We name such kind of segment as the target segment of request point. R4 keeps moving on the target segment in two directions one after another until the server is able to provision the service with resource availability of 100%. Then the service provider will send this counter-offer to the customer. The searching scope is controlled within certain time units. The complexity of this linear search methodology is $O(n)$, which depends on the searching time units in two directions.

Figure 12: Class diagram for computational geometry implementation

**Resource Availability**

Figure 13: Finding alternative solutions by moving the request point
**Virtual Fragments of Server**

We strive to merge the request point with available points while trying to avoid adding new points into the plane unless it is proved to be necessary.

Expired reservations (points) are removed from the context periodically, thus reducing and controlling the degree of the fragmentation.

**Experimental Verification**

To evaluate the model with regard to its validity, we established an online simulation scenario with specific functions to increase the resource utilization by searching alternative solutions and the number of satisfied requests. Resources under negotiation are VMs with CPU cores, memory and hard disk.

There are 4 types of servers and Table 6 illustrates the infrastructure capacity of each kind of server. The overall hardware infrastructure is: 10*S1, 10*S2, 10*S3, 10*S4.

<table>
<thead>
<tr>
<th>Table 6: Server capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Server</strong></td>
</tr>
<tr>
<td>S1</td>
</tr>
<tr>
<td>S2</td>
</tr>
<tr>
<td>S3</td>
</tr>
<tr>
<td>S4</td>
</tr>
</tbody>
</table>

There were 3 types of testing workload: the workload with 800 requests, with 1200 requests and with 1600 requests. During SLA negotiation customers can request from 1 to 8 CPU cores, 512 MB to 16 GB memory and 20 GB to 400 GB hard disk.

The requests array [800 requests, 1200 requests, 1600 requests] is evaluated for different searching scopes [+/- 0, +/- 10, +/- 20, +/- 30], therefore there are in total 12 combinations. The result for each combination is the rate of satisfied requests to all requests. In Table 7 we can see that the wider the searching scope available to the alternative solution planner, the more requests that can be satisfied. For 800 requests, the service provider can handle almost all the requests (95.38%) without finding alternative solutions. For 1600 requests, not every incoming request can be satisfied. For 1200 requests, by searching alternative solutions, almost 90% requests can be satisfied, which is quite close to service provider's capability (see Figure 15).

<table>
<thead>
<tr>
<th>Table 7: Satisfaction ratio of requests with different shifting steps</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workload</strong></td>
</tr>
<tr>
<td>800 requests</td>
</tr>
<tr>
<td>1200 requests</td>
</tr>
<tr>
<td>1600 requests</td>
</tr>
</tbody>
</table>
Figure 14 summarises fragment statistics of all the servers in 1200 requests, +/- 10. It illustrates both when fragmentation is controlled (green points) and not controlled (red points). By controlling fragmentation, there are on average 36 fragments in each server, which are 10 fragments less than the approach without considering fragmentation. Figure 15 is a summary of resources utilization of all the servers in [1200 requests, +/- 0] and [1200 requests, +/- 20] scenarios. We can see that by searching alternative solutions on one hand, significantly more customers requests can be satisfied (19.16% more), whilst on the other hand, the service provider has a higher utilization of resources. For example for CPU cores in (a), (79:55% - 70:30%) * (320 cores * 365 days) = 10804 core * day. 10804 means that 29.6 more cores can be rented to customers throughout the whole year.

![Figure 14: Fragment statistics](image-url)
Figure 15: Resources utilization of all servers
6.3 Infrastructure Provisioning and Adjustment

As described in some detail in the Year 2 version of Deliverable DA5.a [8], the Provisioning and Adjustment Component consists of two parts: a generic part whose class diagram is illustrated in Figure 16, and a non-generic part that must be customized before applying to a specific use case.

![Figure 16: PAC class diagram](image)

In the case of the I-SLAM, the ActionExecutionTask had to be re-implemented in order to properly invoke the Infrastructure Service Manager (I-SM). This task must be added to an agent. On system start-up this agent configures and instantiates itself, after which it starts the specified tasks by communicating with the I-SM.

Once the system is provisioned, the PAC starts listening to the event bus in order to receive messages informing it about the status of the service. These messages usually are domain-specific, so the schema and a parser to translate the xml message to Java classes needs to be provided and added to the configuration file. Monitoring events are fed into a Drools rule engine managed by an AnalysisTask.

For the infrastructure layer, specific rules have been written which analyse the different received events and trigger different actions: re-provisioning, re-starting of a Virtual Machine (VM) or allocation of extra resources.
For illustration purposes, two basic rules are shown here. Note that for the sake of simplicity, the use of constants has been avoided in these examples, being substituted by numerical examples:

```java
//Detects a violation indicating that the availability of a given VM does not fulfill the SLA, and restarts it.
rule "Violation on Availability"
  when
    $event: Event(type == EventType.MonitoringEventMessage, $sla : event.eventPayload.monitoringResultEvent.monitoredSLA);
    $violation : SLAViolationEvent( type == "availability", $infrastructureId : infrastructureId, $availability : value);
    $ism : InfrastructureServiceManager();
  then
    $ism.restart($infrastructureId);
    retract($violation);
end

//If more than 5 memory warnings arrive in 10 min, loads the original ProvisionRequest, resizes its memory, and triggers a reprovision
rule "Adjustment for memory warnings"
  when
    InfrastructureSLA($infrastructureID : infrastructureID);
    Number($counter : longValue > 5) from accumulate( $warning: SLAWarningEvent(type == "average memory", infrastructureId == infrastructureID) over window:time(10m), count( $warning ));
    $provision : ProvisionRequest(infrastructureID == $infrastructureID, $memory : memory);
    $ism : InfrastructureServiceManager();
  then
    modify($provision){setMemory(2*$memory)};
    $ism.reprovision($provision);
end
```

The first example shows a rule that acts upon the reception of a violation on the guaranteed availability of an SLA. In that case, the virtual machine is restarted and its behaviour is observed to see if the reset has improved its performance.

The second example presents a preventive action: after a given set of predictive events (warnings) alerting on high memory consumption for the same infrastructure element arrive in a given time window, then the system triggers a reprovisioning, modifying the amount of memory allocated for the virtual machine.

Rules have been provided that analyse the different received events and trigger different actions: re-provisioning, re-start of VMs or allocation of extra-resources. These actions have been requested by the ORC and the other B-line work packages.

### 6.4 I-SM Proxy

The I-SM Proxy is a component that runs inside the I-SLAM OSGI container. It understands how to communicate with the provisioning system being manipulated. That provisioning system is typically running outside the I-SLAM OSGI container.

As per the high-level architecture, the I-SM Proxy supports both the `<prepare_service>>` and `<manage_service>>` interactions employed by the I-SLAM.
By the end of the second year of the project, this I-SM Proxy made direct calls to an OCCI compatible provider. In the third year of the project the I-SM Proxy was refactored, with generic code removed and integrated into a new JClouds compatible OCCI API to simplify the coding required to manipulate OCCI providers.

The I-SM Proxy was also extended with support for concepts such as reservations, and querying available resources.

Figure 17: Relationship between OSGI, I-SLAM, I-SM Proxy, JClouds and OCCI I-SM for Apache Tashi
**ProvisionRequestType**

Infrastructure requests are passed to the I-SM in the form of a **ProvisionRequestType**. The I-SM Proxy contains the logic that maps from a **ProvisionRequestType** object to a series of OCCI HTTP REST requests.

A **ProvisionRequestType** contains a set of items, known as Kinds, each of which can contain an arbitrary amount of Compute, Network and Storage resources, known as Categories.

```java
ProvisionRequestType
kinds HashSet<E> (id=10203)
[0] Compute (id=10209)
  boot_vol_type null
categories HashSet<E> (id=10215)
[0] Category (id=10252)
  coreClass "mixin" (id=10256)
  id 0
  scheme "http://sla-at-soi.eu/occi/infrastructure/template#" (id=10257)
  term "large" (id=10258)
  title "Large Compute Resource Template"
[1] Category (id=10253)
  coreClass "mixin" (id=10261)
  id 0
  scheme "http://sla-at-soi.eu/occi/templates#" (id=10263)
  term "ubuntu_9-10" (id=10265)
  title "Base Ubuntu 9.10 LTS operating system"
[2] Category (id=10254)
  coreClass "kind" (id=10229)
  id
  scheme "http://schemas.ogf.org/occi/infrastructure#" (id=10267)
  term "compute" (id=10268)
  title null
  version 0
[1] Service (id=10212)
  categories HashSet<E> (id=10222)
  [0] Category (id=10227)
    coreClass "kind" (id=10229)
    id 0
    scheme "http://sla-at-soi.eu/occi/infrastructure#" (id=10230)
    term "service" (id=10231)
    title null
    version 0
  extras Hashtable<K,V> (id=10216)
    resourceId null
    uniqueid UUID (id=10219)
    serviceId UUID (id=10222)
    type null
    monitoringRequest ""
    name null
    provide null
    provStartTime Date (id=10204)
    provStopTime Date (id=10204)
    cdate GregorianCalendar (id=10224)
    fastTime 1309174855153
```

**Figure 18: Example provision request type**

The **ProvisionRequestType** in Figure 18 contains the provision specification for one instance of an Infrastructure service that contains one Compute resource of size large hosting an image whose operating system is Ubuntu 9.10.

The attribute ‘large’ has a domain specific meaning for the actual size of the virtual machine in terms of memory, number of cores, and speed. This is exposed and defined in the SLA Template.
The Service has ‘extra’ parameters associated with it including start and stop times for the service to be provisioned, and monitoring configuration details if required.

<<prepare_service>> Implementation

Whilst the initial version of the OCCI specification did not support reservations, the latest version of the specification supports reservation semantics and so temporary reservation logic has been removed from the I-SM proxy. query, reserve, release and commit methods are now all supported by the implementation.

- **query**: returns a list of types that can be created, and the quantity of these types that there is capacity for.
- **reserve**: reserves the resources for a particular service request. The reservation will expire after a pre-determined period of time.
- **release**: releases an existing reservation.

The <<prepare_service>> interaction only deals with querying and reservations. To complete the provisioning of a reservation, it is necessary to call the commit method on the <<manage_service>> interaction.

<<manage_service>> Implementation

The <<prepare_service>> interaction deals mainly with the provisioning of resources and their runtime management. The architecture specifies the following interfaces: commit, provision, getDetails, reprovision, startResource, stopResource, stop.

- **commit**: commits a reservation to be provisioned. This method is called after the I-SLAM has successfully completed negotiation with the end user.
- **provision**: provisions a specified service. This method is designed to support the on-demand provisioning scenario where a reservation has not been made.
- **getDetails**: Obtains details of a provisioned service. This method calls the I-SM and retrieves information related to the service.
- **reprovision**: this method updates a provisioned service with the new provisioning details.
- **startResource**: signals a resource to start.
- **stopResource**: signals a resource to stop.
- **stop**: stops all the resources in a provisioned service.

In order to facilitate the integration of the I-SM Proxy and the I-SM with other SLA@SOI components, the I-SM proxy interface also provides various helper methods. These include methods to getMonitoringFeatures supported by the provisioning system, createProvisionRequestType to simplify the creation of a ProvisionRequestType, and createComputeConfiguration to simplify the creation of a Compute resource.

I-SM Proxy Implementation Walkthrough

The following example will illustrate step by step how to define, provision, manage and stop a provisioning that contains only one Compute specification (one Virtual Machine) using the currently available interfaces. This example is mainly related to the <<manage_service>> interaction. However for <<prepare_service>> the process is very similar.
The main steps performed are:

1. Create a **Compute** resource instance with only one VM description
2. Create a **ProvisionRequestType** and add the **Compute** instance to it
3. Provision the service.
4. Manage the provisioned service.
5. Stop the provisioned service.

### 1 - Create a **Compute** resource configuration

```java
// We create a Compute Resource first
Category computeCategory = new Category();
computeCategory.setTerm(Terms.COMPUTE);
computeCategory.setScheme(Schema_Constants.OCCI_INFRASTRUCTURE_SCHEME);
computeCategory.setCoreClass(Terms.OCCI_KIND);
Compute vmComputeConfiguration = new Compute();

Set<Category> categories = new HashSet<Category>();
categories.add(computeCategory);

Category osCategory = osReg getCategoryByID(osID);
if (osCategory == null) {
    logger.error("Could not find an Operating System matching a Category - "+ osID);
    osCategory = osReg.getDefaultCategory();
}
categories.add(osCategory);
Category sizeTemplateCategory = metricReg.getCategoryByID(sizeTemplateID);
if (sizeTemplateCategory == null) {
    Logger
        .error("Could not find a Size Template matching a Category - ");
    sizeTemplateCategory = metricReg.getDefaultCategory();
}
categories.add(sizeTemplateCategory);

Category locTemplateCategory = locReg.getCategoryByID(locTemplateID);
if (locTemplateCategory == null) {
    Logger
        .error("Could not find a Location Template matching a Category - ");
    locTemplateCategory = locReg.getDefaultCategory();
} else {
    categories.add(locTemplateCategory);
}
vmComputeConfiguration.setCategories(categories);
String extrasCopy = new HasTable<String, String> extrasCopy = new HasTable<String, String> extrasCopy.putAll(extras);
vmComputeConfiguration.setExtras(extrasCopy);
```
logger.debug("CreateComputeConfiguration result: "+ vmComputeConfiguration);
return vmComputeConfiguration;

2 – Create ProvisionRequestType and add the Compute Kinds Set
ProvisionRequestType provisionRequestType = new ProvisionRequestType();
Date startTime = new Date(System.currentTimeMillis());
Date stopTime = startTime;
provisionRequestType.setProvStartTimestamp(startTime);
provisionRequestType.setProvStopTimestamp(stopTime);

provisionRequestType.setNotificationUri(notificationURI);

Compute vmComputeConfiguration = createComputeConfiguration(osID, sizeTemplateID, locTemplateID,
new Hashtable<String, String>());
Set<Kind> kinds = new HashSet<Kind>();
kinds.add(vmComputeConfiguration);
kinds.add(GetDefaultServiceKind());
provisionRequestType.setKinds(kinds);

// We add the monitoringRequest
provisionRequestType.setMonitoringRequest(monitoringRequest);
logger.debug("createProvisionRequestType Result: "+ provisionRequestType);

3 – Provision the service
ProvisionResponseType provisionResponseType = null;
String infrastructureID = null;
try {
    provisionResponseType = infrastructure.provision(provisionRequestType);
    infrastructureID = provisionResponseType.getInfrastructureID();
}

} catch (DescriptorException e) {
    e.printStackTrace();
}
} catch (ProvisionException e) {
    e.printStackTrace();
}

4 – Retrieving details of the provisioned service
try {
    provisionResponseType = infrastructure.getDetails(infrastructureID);
} catch (UnknownIdException e) {
    e.printStackTrace();
}
List<EndPoint> endPoints = provisionResponseType.getEndPoints();
for (Iterator<EndPoint> iterator = endPoints.iterator();
iterator.hasNext();)
{    EndPoint endPoint = iterator.next();
    // We get the url of this resoruce
    URL url = endPoint.getResourceUrl();
6.5 **JClouds OCCI API**

JClouds is an open source library that simplifies coding required by Java developers to manipulate many cloud platforms and providers. Amazon, GoGrid, Azure, vCloud and Rackspace are all supported at the time of writing.

JClouds implements two generic abstractions: ComputeService for manipulating virtual machines, and BlobStore for accessing key-value data stores.

Three abstractions are available:

- **Providers** allow specific cloud providers to be manipulated, e.g. Amazon EC2
- **APIs** allow specific cloud technologies to be manipulated, e.g. vCloud or Openstack deployments at arbitrary locations
- **Drivers** offer support for different hosting platforms, e.g. Google AppEngine or Apache HC.

Intel has developed an OCCI API for JClouds to help Java developers manipulate OCCI-compatible cloud infrastructure. This API is used by the I-SM Proxy to communicate with an OCCI compliant REST-ful I-SM web server.

The JClouds API communicates via HTTP, passing OCCI payloads to the target system.

The methods in Table 8 have been implemented to date.

### Table 8: JClouds Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>listkeys</td>
<td>This method returns a list of compute resources or services, depending on the request. A list of compute resources or services is returned.</td>
</tr>
<tr>
<td>listAvailResources</td>
<td>This describes the type of resources the provisioning system can provision, and the amounts of those types of resources that can be requested to be provisioned.</td>
</tr>
<tr>
<td>createService</td>
<td>This method creates a new service. The service may contain several compute resources. Unique identifiers may be supplied for both the service and the individual compute resources. This is done by specifying X-OCCI-Attribute: occi.core.id= %unique id%</td>
</tr>
<tr>
<td>getService</td>
<td>This method returns complete details about a specific</td>
</tr>
</tbody>
</table>
service, including the operations (actions) that that service supports.

**getResource**
This method returns complete details about a specific compute resource, including the operations (actions) that that resource supports.

**postServiceAction**
Allows a supported action to be posted to a specific service. Typical actions include suspend, start, stop.

**removeService**
Deletes a previously provisioned or reserved service

**updateService**
One can update the configuration of a service or compute resource

The Java method call is translated into the appropriate OCCI call. An example OCCI call to return a list of IDs of compute nodes looks like:

```
GET /compute/ HTTP/1.1
Date: Mon, 27 Jun 2011 08:38:04 GMT
Authorization: pyocci_user Zm9v|1299618662|6e03b112eccd614f08baca3f545df6ae9c0c0731;
expires=Thu, 07 Apr 2011 21:11:02 GMT; Path=/:GZulLqwPEn4ec72avAXR05Cpfqg=
Host: localhost:8888
User-Agent: jclouds/1.0 java/1.6.0_16
Content-Length: 0
Connection: keep-alive
```

Whilst the corresponding result looks like:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Server: OCCI/1.1
Content-Type: text/plain
Content-Length: 1008
Date: Mon, 27 Jun 2011 08:38:23 GMT
X-OCCI-Location: http://localhost:8080/compute/590658e5-7b3c-4385-9927-2c8a5c0da0dd
X-OCCI-Location: http://localhost:8080/compute/8aa6c7dc-e3d7-4f77-b450-225eba3e125e
X-OCCI-Location: http://localhost:8080/compute/4376e9eb-b373-4227-8772-ae27f84597ab
X-OCCI-Location: http://localhost:8080/compute/dcef9ff1-2bd7-4c28-9e96-54a4003ceac5
```

This may be followed by a call to start one of these resources as follows:

```
POST /service/14f6fe8d-c223-4e0d-ae7c-f000cf627c6b?action=start HTTP/1.1
Date: Mon, 27 Jun 2011 09:37:46 GMT
Authorization: pyocci_user Zm9v|1299618662|6e03b112eccd614f08baca3f545df6ae9c0c0731;
expires=Thu, 07 Apr 2011 21:11:02 GMT; Path=/:E8WcowKHjM8QOP7UowUeklvy5j4=
Host: localhost:8888
User-Agent: jclouds/1.0 java/1.6.0_16
Content-Length: 0
Accept: text/html, image/gif, image/jpeg, *, q=1.0; */*; q=0.2
Connection: keep-alive
Content-type: application/x-www-form-urlencoded
```

A typical response of the above call would be:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Server: OCCI/1.1
Content-Type: text/html;charset=utf-8
Transfer-Encoding: chunked
Date: Mon, 27 Jun 2011 09:37:46 GMT
```

The translation to and from these HTTP calls is handled by the JClouds OCCI API.
6.6 OCCI I-SM for Apache Tashi

The Infrastructure Service Manager (I-SM) component is a web service that implements the OCCI standard. The web service implements a REST-ful interface as specified by the OCCI standard. The main responsibilities of this component are to accept queries related to resources, place reservations for future resource allocation (including the release of those reservations). These responsibilities map to the <<prepare_service>> interaction. Furthermore, the I-SM has the responsibility of creating (including commitment of resource reservations), updating, retrieving information on and deleting resources under the management of the I-SM. These correspond to the <<CommitRUD>> interaction.

The I-SM tracks all the requests associated with these responsibilities in the Infrastructure Landscape (IL). In terms of implementation, the IL represents the domain model of the I-SM and consequentially is very much aligned to the OCCI model (see Section 5).

The I-SM and also the IL are implemented using the web framework Grails [41], a rapid web development framework and environment that uses the dynamic language, Groovy [56] that is completely Java byte code compatible. This means that we leverage all the benefits of not only a rapid development environment and dynamic language but also the huge, diverse and rich Enterprise Java ecosystem. Groovy and consequently Grails can use any Java library ensuring easy integration and compatibility with any code from other SLA@SOI work packages. Grails, like many other web frameworks, implements the Model-View-Controller design pattern. For the purposes of the I-SM implementation it was not seen as a priority to implement a user interface (the View in MVC) using the frameworks available within Grails. As such there are currently no views, shown in the internal architecture (see Figure 19).

In order to ensure that requests received by the I-SM are OCCI compliant as per [OCCI_HTTP] definitions an ANTLR [19] grammar was defined. This allowed for the automatic generation of a validating lexer and parser. However, in order to extract the values of the validated input language-specific (Java in this case) code was required to be authored. The OCCI grammar [41] was contributed to the OCCI project and there are now a number of external OCCI-related projects using this. More details of this grammar can be found in a blog post on the OCCI website [58].

The I-SM internal architecture (Figure 19) shows the immediate components that are implemented in order to satisfy its responsibilities. Those are grouped as:

- **Controllers**: These implement stateless business logic. They are the service side logic that clients interact with depending on what function needs to be carried out. For example, if a client wants to create a new Compute resource on-demand, they will interact with the Compute controller using OCCI.

- **Model**: This implements the domain model of the IL. All operations that the I-SM carries out on the behalf of its clients are recorded here.
• **Services**: These implement stateful or long running business logic. For example the ReservationService. When clients request the reservation of infrastructure resources, the controller (Reservation) on their behalf makes an entry in the IL. When called upon to commit a lease-bounded reservation, the ReservationService looks up the reservation’s resource requirements and instructs the ProvisioningSystem (via the MessagingService) to make the reserved resources available.

![Diagram of OCCI I-SM for Apache Tashi](image)

**Figure 19: Internals of the OCCI I-SM for Apache Tashi**

Also shown in the internal architecture diagram are the Message Bus and ProvisioningSystem. The I-SM and the target provisioning system adopt an event-based architecture and are therefore cleanly separated by the use of messaging middleware (Message Bus). This brings great flexibility to the system. As the I-SM is loosely coupled to the ProvisioningSystem this enables many different deployment configurations. As the message formats that the I-SM sends (MessagingService) to the ProvisioningSystem are well understood, all that is required to support a different ProvisioningSystem is to write an associated message sink that has full access to management facilities of the provisioning system. Alternatively, the MessagingService can be extended or modified to adapt to the target ProvisioningSystem’s communication mechanism.
In our case, the I-SM in SLA@SOI uses Apache Tashi (Section 6.7) as a provisioning system. Our messaging middleware is pluggable and currently implements the open standards XMPP and AMQP. It was required that we write an XMPP message sink that would receive the XMPP or AMQP messages from the I-SM and duly process them. On both sides (I-SM and Tashi) it is relatively easy to add support for other messaging technologies. In the I-SM the MessageDispatcher and MessageHandler interfaces need to be extended. In Tashi the OCCI bridge daemon needs to be adapted. Once processing was complete a responding message was sent back to the I-SM. The I-SM then, on receipt of the provisioning system result, updated the IL state and sent notification of the result to the original requestor.

Another point of integration was in providing direct access to Tashi’s own internal resource pool. This was done by implementing a new Tashi data driver to write data to Redis, a key-value store. Tashi constantly updates this information in the store and the I-SM can access it using the Redis grails plugin.

### 6.7 Provisioning System – Apache Tashi

To better deliver a SLA-enabled infrastructure, two modules for the virtual machines management have been designed and developed. As a provisioning system for the SLA@SOI infrastructure management Apache Tashi was adopted. Apache Tashi manages a set of KVM and/or Xen hypervisors that run on a clustered set of physical machines. Both modules have been integrated in Apache Tashi, but can be further integrated in other IaaS platforms.

The first virtual machines management module is a scheduler that allocates the virtual machines to the physical machines. The allocations take into account Infrastructure SLA specifications. As described in the B4 Enterprise IT use case deliverable [11], server efficiency, energy consumption, user priorities and over-provisioning policies have also been integrated into the scheduler.

SLA-aware virtual machines provisioning and re-provisioning is the second extension for the virtual machines management and it provides functionality for a dynamic resource management (CPU, memory, network bandwidth, disk bandwidth) of the virtual machines.

#### 6.7.1 SLA-aware scheduler

SLA@SOI scheduler provides the following functionality:

- Find the appropriate server for a new virtual machine request and start the virtual machine with resources as specified in the Infrastructure SLA.
- Minimize the number of active physical servers.
- When some virtual machines are released, migrate virtual machines with the highest priority to the more efficient servers (servers are ranked by energy and utilisation efficiency).
- Avoid too many migrations of the virtual machines, because of the fact that live migrations require two instances of virtual machine in the process of migration and consequently some SLA violations can happen.

The scheduler flow is executed periodically and proceeds as follows:

- Checks if distribution of the virtual machines across the servers complies with the server policies and user priorities; triggers migrations if needed.
- Finds the most efficient available server for a new provisioning request (checks for available CPU speed, CPU cores, memory resources and other SLA parameters).
- If no available server was found, the most appropriate server for over-provisioning is searched on the base of CPU over-provisioning policy. CPU over-provisioning policy defines if virtual CPU speed and cores can exceed (and for how many times) the actual physical resources.

The scheduler flow is illustrated on the Figure 20. It is executed every “scheduleDelay” seconds (scheduleDelay is part of the scheduler configuration, which will be described later). The flow is managed from SchedulerManager, which is executing methods from the scheduler implementation instance. The scheduler implementation needs to implement the following methods (SchedulerInterface was defined and integrated into the scheduler module to make SLA@SOI modifications pluggable):

```python
def setVmsForMigration():
    def handleMigration():
        def determineHost():
            def getMostSuitableHost():
                def freeUpResources():
                    def checkAdaptedVms():
                        def reschedule():
                            def powerOffHosts():
                                def powerOnAppropriateHost():
                                    def guaranteedServersOn():
```

![Figure 20: Apache Tashi Scheduler flow](image)

---

**SLA@SOI** – FP7216556  
**D.A4a – SLA-Aware Infrastructure Management**  
**Page 65 of 116**
6.7.2 Scheduler and Infrastructure SLA

Overall, the responsibility of the scheduler is to guarantee the Infrastructure SLA terms, for example the following part of the SLA specifies that changes to the virtual machine disk have to be persistent.

```xml
<slasoi:VariableDeclr>
    <slasoi:Text/>
    <slasoi:Properties/>
    <slasoi:Customisable>
        <slasoi:Var>VM_PERSISTENCE_VAR</slasoi:Var>
        <slasoi:Value>true</slasoi:Value>
    </slasoi:Customisable>
    <slasoi:Datatype>http://www.w3.org/2001/XMLSchema#boolean</slasoi:Datatype>
</slasoi:VariableDeclr>

In order to meet the requirements of this particular part of the SLA, the scheduler has to take care of preparing enough images for a virtual machines to be started with a persistent disk and after this it has to start the virtual machine with the appropriate parameters (for a KVM that means to start a virtual machine with a properly set snapshot parameter). Other Infrastructure SLA terms are: VM_QUANTITY, VM_ISOLATION, VM_CORES, VM_CPU_SPEED, VM_MEMORY_SIZE, VM_IMAGE and others.

6.7.3 Scheduler algorithm

A search for the appropriate server for a new virtual machine request can be essentially viewed as a constraint satisfaction problem (CSP). Let \( h^i_{\text{cores}} \) denote the number of CPU cores on a physical server \( h^i \), \( h^i_{\text{cpu}} \) denote the CPU speed in MHz and let \( h^i_{\text{mem}} \) denote the amount of memory of \( h^i \) in megabytes. The number of physical servers is denoted by \( k \) and the number of virtual machines that are running on server \( h^i \) as \( n^i \). Request for a new virtual machine can be described as \( v = \{ v^i_{\text{cores}}, v^i_{\text{cpu}}, v^i_{\text{mem}} \} \), where CPU cores, CPU speed and memory are specified for a virtual machine request. Running virtual machines are described as \( v_{ij} = \{ v^j_{\text{cores}}, v^j_{\text{cpu}}, v^j_{\text{mem}} \} \), where \( v_{ij} \) denotes a running virtual machine on a server \( h^j \) and \( j < n^j \). Let \( \text{opr}_{\text{cores}} \) denote the over-provisioning rate for CPU cores and \( \text{opr}_{\text{cpu}} \) the over-provisioning rate for CPU speed (see cpuCoresOverrate and cpuSpeedOverrate configuration option). When over-provisioning variables are set to 1, the over-provisioning is not allowed. A problem of finding the most suitable server for over-provisioning can be illustrated as follows:
minimize \( f(0, a_{\text{cores}}^i, a_{\text{cpu}}^i) \), where 1 ≤ i ≤ k and:

\[
\sum_{j=1}^{n_k} v_{\text{cores}}^{ij} + v_{\text{cores}} \leq h_{\text{cores}}^i \times o_p r_{\text{cores}}^i
\]
\[
\sum_{j=1}^{n_k} v_{\text{cpu}}^{ij} + v_{\text{cpu}} \leq h_{\text{cpu}}^i \times o_p r_{\text{cpu}}^i
\]
\[
\sum_{j=1}^{n_k} v_{\text{mem}}^{ij} + v_{\text{mem}} \leq h_{\text{mem}}^i
\]

Regarding the guaranteedServers option - two scheduler implementations have been developed for SLA@SOI purposes - Consolidated and Balanced (see the scheduler configuration option). The configuration option guaranteedServers makes sense for the Consolidated type. To fully exploit the Balanced scheduler implementation the guaranteedServers option should be set to the number of all available servers.

**Balanced scheduler:** When VMs are provisioned, they are spread evenly across the number of available hosts. Hosts with highest efficiency rating are filled first. VMs priorities come into account when an unsuccessful provisioning request happens - in this case and if provisioning request is of a higher priority - resources are taken from the VMs with the lowest priority to provide resources for a request. When some resources are released the VMs with the highest priority are migrated to a better host if possible. When rescheduling is executed the scheduler tries to balance VMs' distribution across the servers. In general too many migrations are avoided, because migrations besides other effects also require two VMs instead of one (source and destination VM) at the time of migration.

**Consolidated scheduler:** When VMs are provisioned, each physical host is filled to its capacity before moving to the next one. Hosts with highest efficiency rating are filled first. VM priorities come into account when an unsuccessful provisioning request happens - in this case and if provisioning request is of a higher priority - resources are taken from the VMs with lowest priorities to provide resources for a request. When some resources are released the VMs with the highest priorities are migrated to better hosts if possible. When rescheduling is executed the scheduler tries to consolidate VMs' distribution across the servers. In general too many migrations are avoided, because migrations besides other effects also require two VMs instead of one (source and destination VM) at the time of migration.

The scheduler can be switched from Consolidated to Balanced mode when the a rise in the cloud utilization is foreseen, and the intent is to distributed workload across all nodes. A switch from Balanced to Consolidated mode could be made when a fall in the cloud utilization is predicted, and the goal is to reduce power consumption.

A detailed demonstration of the implemented scheduler is published on the SLA@SOI Sourceforge developer wiki [59].
6.7.4 VM resource allocation management via cgroups

There is a strong need for managing the allocation of various resources to virtual machines in the SLA@SOI framework in order to provide SLA-compliant behaviour. While physical memory and number of cores allocated to a virtual machine are enforced by the hypervisor itself, allocation of CPU and I/O bandwidth to virtual machines is implemented by leveraging the Linux Control Groups (cgroups) facility on the host operating system level.

Linux cgroups provide a general mechanism for grouping Linux tasks (processes and their threads) into hierarchically arranged groups with distinct, user-specified properties. A cgroup assigns a number of OS tasks a set of parameters that affect how the tasks are handled by the OS. Each parameter affects one of the OS subsystems (such as CPU bandwidth, assignment of tasks to individual CPU cores and memory nodes, I/O bandwidth etc.).

User-level code may establish, remove, inspect and manipulate cgroups by means of cgroups pseudo file-system, mounted at an arbitrary location. Each directory in this pseudo file-system represents an individual cgroup, and the files therein provide means for assigning tasks to the group and setting of the parameters associated with the group. A hierarchy of nested directories represents nested cgroups, with each child cgroup inheriting the parameters of the parent and further refining them.

It must be noted that cgroups allow only for imposing soft limits on resources allocated to tasks. Thus, a task with a limit imposed on the share of the CPU provided to it by the OS scheduler, may be given a share that exceeds this limit iff there are spare CPU cycles available (i.e. the host is not fully loaded). Limits on the CPU share will be respected when the host is fully loaded, though. Hard limiting CPU bandwidth (i.e. ensuring the task is never given more CPU cycles than the limit imposed states) would have been possible by employing specialized kernel patches for the CFS scheduler, such as the CFS bandwidth control patch or the CFS hard limits patch; as these patches are not part of mainline kernel, and we wanted to avoid introduction of specialized, patched kernels, we chose not to opt for these solutions.

The rest of this section describes how cgroups are used in our system, and how CPU and I/O bandwidth allocation to virtual machines is performed.
Setting Up the cgroup Subsystem

Upon starting the sla@soi services on a virtualization host, the cgroups pseudo file system is mounted to /dev/cgroup, enabling the CPU and block I/O subsystems:

```
mkdir /dev/cgroup
mount -o cpu,blkio -t cgroup /dev/cgroup
```

Then, a dedicated cgroup for all host processes (except for the VM-related ones) is created and all tasks are moved into it:

```
mkdir /dev/cgroup/host
for pid in `cat /dev/cgroup/tasks`; do
echo $pid >/dev/cgroup/host/tasks
done
```

In this manner, we can control what share of the total CPU power is set aside for the system processes and what share is left to be shared between the virtual machines.

For each virtual machine that will be created on the host, a separate cgroup will be created at /dev/cgroup/<vm-name> and the tasks related to that virtual machine will be moved to that cgroup.

**CPU Share Allocation**

CPU shares are allocated by entering the sharing weight number into the file /dev/cgroup/<group>/cpu.shares. A process belonging to cgroup C1 with cpu share weight W1 is given a share of the total CPU cycles equivalent to W1/sum(Wn).

Figure 21 shows the actual percentage of total CPU time given to each of the 32 processes with different weights on a fully loaded system. Processes (and their weights) are shown on the X axis and the share of CPU time in percent on the Y axis. The red line shows the ideal, expected CPU time share and the blue one the actual, measured CPU time share of each process. It is clear that the cgroups are able to enforce the CPU sharing policy quite well.
In order to explain how we map the CPU speed SLA term to the cgroups cpu.shares parameter, let us first introduce the relevant notation.

The number of cores of the host machine is denoted with $n_{cores}(host)$, and the clock frequency of the host cores (in MHz) with $clock(host)$. Both number of cores as well as the host CPU frequency are determined from `/proc/cpuinfo` pseudofile. When reading this file, the CPU frequency should be stable (i.e. frequency scaling should be disabled, or the scaling governor set to `performance`) in order to determine the maximum (unthrottled) clock frequency. From this parameters, we compute the total CPU cycles available (every second) on the machine as

$$t_{cpu}(host) = n_{cores}(host) \times clock(host)$$

A part of the CPU will be dedicated to execution of system processes. This is a user configurable parameter, and we denote it with $sys_{cpu}(host)$.

Thus, the total amount of CPU cycles available to the virtual machines started on a host is

$$guest_{cpu}(host) = t_{cpu}(host) - sys_{cpu}(host)$$

We denote virtual machine CPU speed SLA term with $clock(vm)$ and its number of cores (also a SLA term) with $n_{cores}(vm)$.

Thus, a virtual machine $vm$ with given CPU speed and core count SLA terms must be guaranteed

$$cpu(vm) = n_{cores}(vm) \times clock(vm)$$
CPU cycles per second. Note that sum of \(\text{cpu}(vm)\) over all virtual machines can be higher than \(\text{guestcpu}(host)\) iff the machine is overcommitted with regard to CPU speed SLA term.

Based on the way cgroups CPU allocation mechanisms function, we can use \(\text{syscpu}(host)\) as the \text{cpu.shares} weight for the host cgroup (system processes), and \(\text{cpu}(vm)\) as the \text{cpu.shares} weight for virtual machine \(vm\).

If a set of virtual machines \(\{vm_1 \ldots vm_n\}\) is scheduled to run on a host, the relevant virtual machine cgroups are established as the following pseudocode shows:

```bash
for vm in \{vm_1 \ldots vm_n\}; do
    mkdir /dev/cgroup/vm
    echo cpu(vm) >/dev/cgroup/vm/cpu.shares
done
```

With such a set-up, each virtual machine will receive at least \(\text{cpu}(vm)\) cycles per second, provided that \(\text{guestcpu}(host) \geq \text{sum}(\text{cpu}(vm))\) (i.e. the host is not overcommitted wrt. SLA-guaranteed CPU speed of all the virtual machines).

In case of overcommiting, CPU speed SLA term for a given virtual machine will be violated only if the sum of required CPU cycles of all virtual machines exceed the available CPU cycles in a given time interval. In this case, service degradation will be uniform over all the virtual machines whose required CPU cycles exceed their share of available CPU cycles.

This brings us to the final question: which tasks to put in each virtual machine's cgroup? It turns out this is a hypervisor-dependent issue. In case of KVM, a number of tasks (threads of a single process) is spawned for each virtual machine. The main KVM thread takes care of interaction with the host environment (hardware emulation, I/O etc.), and spawns one additional thread for every core of the virtual machine. These additional threads are the ones that perform actual guest computation, and are thus the only ones that should be put in the virtual machine's cgroup as the CPU speed SLA term only relates to CPU time received by guest computation. The main thread, on the other hand, should be put in the host cgroup. When a KVM process with PID \(P\) is started in order to host virtual machine \(VM\), the following pseudocode assigns the guest core threads to the appropriate virtual machine cgroup, and the main KVM thread to the host cgroup:

```bash
echo P >/dev/cgroup/host/tasks
for pid in /proc/P/task/*; do
    if [ \(\text{pid} != P\) ]; then
        echo pid >/dev/cgroup/VM/tasks
    fi
done
```

---

### 6.7.5 Tashi remote management

In order to integrate Tashi with the infrastructure service manager, a message sink process that receives messages from the I-SM and processes them was implemented. This process or daemon runs in the background on the Cluster Manager node of a Tashi deployment. It receives provisioning requests as XMPP messages, extracts the requirements and using the Tashi client, issues requests for VM creation. The daemon will not return a response immediately to the I-SM
but rather will notify the I-SM asynchronously once the provisioning process has completed. A technical description of the XMPP bridge and details on how a remote Tashi API can be extended is available on the SLA@SOI Sourceforge developer tutorial and wiki [59].

6.7.6 Tashi monitoring sensors

Tashi has been extended with monitoring sensors in order to provide detailed information about the system to the monitoring components. Sensors periodically send messages about physical servers and virtual machines to the Infrastructure Monitoring Agent. These messages contain information that is not available via Ganglia or Nagios or any other monitoring engine that is used by Infrastructure Monitoring. A more detailed description can be found in Section 6.8.

6.7.7 Tashi hosts auto-registration

Automatic host registration has been implemented in order to reduce the manual work when installing Tashi. Host is registered when nodemanager.py script is started – the information about underlying physical server is automatically retrieved from the system and inserted in to the Tashi data storage. Auto-registration is supported for SQL, Pickled and FromConfig data storage types in Apache Tashi.

6.8 Infrastructure Monitoring

The Infrastructure Monitoring Agent (IMA) is an entity responsible for infrastructure layer monitoring and verifying the compliance of infrastructure services with the SLA. It collects metrics data from external monitoring software (e.g. Ganglia), processes it, stores metrics history, computes QoS terms and verifies their compliance with the SLA. In case any violations are found IMA notifies higher-level components (the PAC) and stores violations history. If any QoS term is nearing the violation threshold IMA emits warning of potential SLA violation. IMA also provides an interface for other components to get monitoring data.

6.8.1 Architecture

The complete architecture of infrastructure monitoring, including the subcomponents of the IMA and interactions with other components, is shown in Figure 22.
Figure 22: Infrastructure monitoring architecture
The main architectural components of the IMA are:

- Service Registration Manager
- Metrics Gatherer
- QoS Terms Computation & SLA Compliance Evaluation
- Data Store
- Monitoring Data Provider
- Infrastructure Reporting

**Service Registration Manager**

The Service Registration Manager is responsible for registering new infrastructure services, unregistering services, updating service registration and providing monitoring features of the IMA. The corresponding interactions are `<<configure_monitoring>>` and `<<get_monitoring_features>>`. The communication goes through a publish/subscribe messaging protocol. Messaging is implemented based on Messaging SLA@SOI framework module which supports different protocols, e.g. XMPP and AMQP.

The Service Registration Manager listens on the configuration channel and receives monitoring configuration requests from the Infrastructure Service Manager component. Three kinds of requests are possible:

- register new infrastructure service and start monitoring it (RegisterService request)
- update registration of an already registered service (UpdateServiceRegistration request)
- stop monitoring of specific infrastructure service (StopMonitoring)

Service Registration Manager processes the request, stores/upDATES service data in the database and takes appropriate action.

Service Registration Manager also responds to the `get_monitoring_features` requests on the configuration channel that come from the Monitoring Manager and are passed via the Infrastructure Service Manager. Monitoring features specify the capabilities of the monitoring system associated to a service. This information enables the planning components to understand and reason about the monitorability of the guarantee terms as well as the capability to configure monitoring elements on the fly during service provisioning process.

**Metrics Gatherer**

The responsibility of the Metrics Gatherer is to periodically collect metrics from external monitoring software, cache metrics data in memory and makes that data available to other IMA components. Metrics Gatherer isn’t tightly coupled to specific external monitoring software. Instead a generic IMonitoringEngine interface is defined that specifies methods that all monitoring engines must implement, e.g. methods to retrieve specific host or VM metric value, to get the date when the metrics was collected. The IMonitoringEngine interface is as follows:

```java
define interface IMonitoringEngine {
define enum Status {
  OK,
  ERROR,
```
Metrics Gatherer interacts with monitoring engine only through the IMonitoringEngine interface. This way different monitoring engines (and external monitoring software) can be supported. Currently Ganglia Monitoring System in combination with a custom Tashi Sensor is fully supported, which is called GangliaTashiMonitor monitoring engine.

Initial integration has been done with the OpenStack as a cluster management software and Nagios as a monitoring tool. OpenStack has been extended with a custom sensor similar to the custom Tashi Sensor.

**GangliaTashiMonitor**

GangliaTashiMonitor is a monitoring engine that implements IMonitoringEngine interface and retrieves monitoring data from the Ganglia Monitoring System and Tashi Sensor. Because not all required monitoring data is provided by the Ganglia a custom Tashi Sensor was implemented which provides missing data. GangliaTashiMonitor periodically queries Ganglia and Tashi Sensor, metrics from both sources are combined into a whole.

To retrieve monitoring data from Ganglia, the GangliaTashiMonitor queries the Ganglia Meta Daemon (gmetad) through a TCP port (8651 by default) and retrieves a Ganglia report in XML format (GANGLIA_XML document). It parses the report and extracts relevant metrics of host and guest machines (VMs) that are being monitored.

Tashi Sensor is queried through XMPP point-to-point protocol. It provides two types of monitoring data:

- **VmLayout**: virtual machines layout, reserved system resources on hosts, virtual machine specifications
- **ClusterConfiguration data**: host performance, security, reliability and location information

In addition Tashi Sensor notifies the IMA about various Tashi events related to hosts and virtual machines. These kind of messages are of type AuditRecord and are sent by Tashi Sensor to IMA through XMPP point-to-point protocol by push method.

VmLayout and ClusterConfiguration messages are retrieved from Tashi sensor periodically on request, while AuditRecord messages are pushed from Tashi sensor. All communications between IMA and Tashi Sensor go through XMPP point-to-point protocol.
QoS Terms Computation & SLA Compliance Evaluation

The QoS Terms Computation & SLA Compliance Evaluation component is responsible for computation of QoS terms (Quality of Service terms) based on monitoring data obtained from the Metrics Gatherer and evaluation of computed QoS terms and checking if they are compliant with the SLA. QoS terms are SLA guaranteed terms and define the assurance on service quality associated with the infrastructure service described by the service definition terms. In addition the component computes some metrics which are not SLA-guaranteed terms but are needed for QoS terms computation (e.g. Service Availability Status) or for generating reports (e.g. VM CPU Speed Used).

Some simple QoS terms/metrics do not require any computation and can be directly obtained from the Metrics Gatherer component with the possible exception of unit conversion (e.g. VM Memory Size) while some more complex ones are computed by a specific procedure from the metrics obtained from Metrics Gatherer, metrics history retrieved from the database, etc (e.g. Service Availability). The computed values are optionally stored to the database as metric value history for the QoS terms/metrics where the history is needed (e.g. the history of Service Availability Status metric is needed for computation of the QoS term Service Availability, history of Service Availability is needed for generating reports). History values are stored for the time intervals (startTime, endTime) corresponding to metrics collection interval. If the metric has the same value over multiple intervals it will be stored only once and the interval will be widening. This method is very efficient for storing metrics history that don't change much over time, for example Service Availability Status has the same value (true or false) over longer time periods. For numeric metrics whose values change frequently (e.g. Memory Size Used) it is possible to reduce the history values frequency (to reduce space usage) with the calculation of average value over a longer time period.

After the QoS term value is calculated the component checks if the value is compliant with the SLA. For each QoS term a constraint expression is specified by the SLA to which metric values must correspond. In the opposite case a SLA violation occurs. SLA violations events are recorded in the database and published to the publish/subscribe event channel where the Provisioning and Adjustment (InfrastructurePAC) component listens for them. In some special cases it is possible that in spite of some individual QoS terms being violated the whole infrastructure service is still compliant with the SLA. This is possible if some special QoS terms are defined (Acceptable Service Violations, Service Availability Restrictions).

For some numeric QoS terms (e.g. Service Availability) it is possible and reasonable to generate a warning if a metric value is nearing the violation threshold. The IMA calculates the warning threshold based on the violation threshold of the specific QoS term, for example it multiplies the violation threshold with some factor. If metric value exceeds the warning threshold (from above or from below, depends on the metric) a warning is generated. The warning event is recorded in the database and also published to the event channel.

Besides that the QoS Terms Computation & SLA Compliance Evaluation component periodically publishes values of certain QoS terms for each service being monitored to the event channel for the Provisioning and Adjustment (InfrastructurePAC) component.
**Data Store**

The Data Store is responsible for persisting data like infrastructure services and provisioned virtual machines, metrics to monitor and their constraint expressions, metrics value history, violation and warning events, etc. It is implemented in the JPA EclipseLink technology, and data is stored in a MySQL database.

Schema of the infrastructure_monitoring_agent database is presented in Figure REF.

Database consists of following tables:

- **service**: represents an infrastructure service that is being monitored. Service contains one or more resources (virtual machines).
- **vm**: represents a virtual machine that is being monitored. Each virtual machine belongs to specific service.
- **metric**: represents a metric that is being monitored. Each metric is of specific type and belongs to specific virtual machine.
- **metric_type**: represents a type of a metric (including QoS terms).
- **metric_value**: represents a current value of a specific metric and its compliance with the SLA.
- **metric_value_history**: represents a history value of a specific metric. History value is defined by the metric value and time interval the metric had that value.
- **violation**: represents a SLA violation that occurred for a specific metric.
- **audit_record**: represents an audit record the IMA received from the cluster manager (Tashi, OpenStack) as a notification of some event.
Monitoring Data Provider

The Monitoring Data Provider component provides monitoring data to the frontend application. The communication goes through a publish/subscribe messaging protocol. The component listens on the monitoring data channel, accepts queries from the frontend application, retrieves required data from the Metrics Gatherer and Data Store components and responds with the appropriate response message over monitoring data channel.
Infrastructure Reporting

The Infrastructure Reporting component is responsible for generating various reports about infrastructure services. Reports are generated based on the data from the Infrastructure Monitoring Agent database and are exported in PDF format. Various reports are available, for example:

- **Service Summary**: presents summary and various statistics of the given infrastructure service, like date created, total uptime, total downtime, resources list, service and its resources availability, SLA compliance of the service and resources, total number of violations.

- **Service SLA Summary**: presents all QoS terms defined by the infrastructure SLA for given service and its resources. For each QoS term a current value is given, as well as constraint expression or violation threshold and whether the current value is compliant with the SLA. For the service and all resources an overall SLA compliance state is presented.

- **QoS Terms History**: presents history of selected QoS terms for given infrastructure service in graphical form

- **Violations History**: presents all SLA violations for given infrastructure service. For each violation a QoS term in violation is given, actual value, constraint expression and time period of the violation.

- **Resources Usage**: presents host machines resources usage (CPU, memory) by the selected infrastructure service separately for each VM and totally for all VMs.

Reports are generated on request initiated by for example frontend application.

6.8.2 Interactions

External interactions with the IMA are:

- `<<get_monitoring_features>>`
- `<<configure_monitoring>>`
- `<<get_ganglia_metrics>>`
- `<<get_tashi_metrics>>`
- `<<notify_tashi_event>>`
- `<<publish_violation_event>>`
- `<<publish_metric_values>>`
- `<<get_monitoring_data>>`
- `<<display_monitoring_event>>`
- `<<publish_diagnostic_message>>`

`<<get_monitoring_features>>`

The `<<get_monitoring_features>>` interaction allows the Monitoring Manager to get the monitoring capabilities of the IMA to be able to generate a Monitoring System Configuration (MSC) for the given SLA model. The Monitoring Manager doesn’t interact directly with the IMA but via the Infrastructure Service Manager (I-SM) which acts as a proxy. The `<<get_monitoring_features>>` interaction communication goes through a publish/subscribe messaging protocol. A client creates an instance of request message (MonitoringFeaturesRequest), serializes it
to JSON and publishes it to the configuration publish/subscribe channel. The IMA listens on the configuration channel and receives the request. It generates monitoring features (ComponentMonitoringFeatures[] object), creates response message (MonitoringFeaturesResponse), serializes it to JSON and publishes it to the same publish/subscribe channel.

<<configure_monitoring>>

The <<configure_monitoring>> interaction allows the calling component (i.e. the Infrastructure Service Manager) to configure monitoring performed by the IMA. Three operations are available:

- **RegisterService**: registers new infrastructure service and initiates monitoring of it.
- **UpdateServiceRegistration**: changes already registered service
- **StopMonitoring**: stops monitoring of specific service. Service data is preserved in the database and is still available for generating reports.

The <<configure_monitoring>> interaction communication goes through a publish/subscribe messaging protocol. The client creates an instance of a request message, serializes it to JSON by calling the toJson() method and publishes it to the configuration channel. IMA processes the request and returns response message serialized to JSON over the same channel.

**Operation RegisterService**

The operation RegisterService is used to register new infrastructure service and initiate its monitoring. An infrastructure service is a collection of resources (i.e. virtual machines) and corresponds to the infrastructure SLA. RegisterService request is sent by the Infrastructure Service Manager when it receives new infrastructure SLA. The request message contains data about the infrastructure service, list of service resources (i.e. virtual machines) and guaranteed QoS terms from the SLA for both the service and all resources. Guaranteed terms can be specified in two ways:

- as a list of AgreementTerm objects. Each AgreementTerm specifies the name of the guaranteed QoS term, violation constraint and the constraint unit in case of numeric terms.
- as an instance of MonitoringSystemConfiguration which is generated by the Monitoring Manager based on the SLA and monitoring features of the IMA. The IMA parses the MonitoringSystemConfiguration object and extracts QoS terms constraint expressions.

A sample of RegisterService request message with guaranteed terms specified by the first method (as a list of AgreementTerm-s) can be seen in Appendix D.

**Operation UpdateServiceRegistration**

The operation UpdateServiceRegistration is used to change (update) service registration in case service changes, for example guaranteed QoS terms’ constraints are changed or new guaranteed terms are defined or new resources are added. The request message has the same structure as RegisterService operation. The IMA checks for changes and registers them in the database.
Operation StopMonitoring

The operation StopMonitoring is used when SLA contract expires and the resources are deprovisioned. The monitoring of the service is stopped but the service data is still preserved in the database.

<<get_ganglia_metrics>>

The <<get_ganglia_metrics>> interaction allows the IMA to get monitoring data from the Ganglia. The IMA interacts with Ganglia Meta Daemon (gmetad) which aggregates metrics data from all Ganglia Monitoring Daemons (gmond) that run on each host machine. Gmetad listens on TCP port 8651 by default and returns Ganglia metrics report in XML format. The IMA periodically queries Ganglia and extracts metric values from the received report.

Ganglia Monitoring Daemon (gmond) runs only on host machines and not on guest machines (VM). This way it is not required for customers to install any monitoring software on virtual machines. All VM metrics are reported by the host machine on which VM resides. For this reason all VM metrics are given from the host point of view, e.g. memory usage means host memory used by the VM.

Standard Ganglia metrics reported by the host were extended with some custom ones which report usage of host resources by the guest machine (VM). The values of these metrics are reported by the cluster manager - Tashi. Their name has a form 'tashi_vm_<fqdn>_<metric>', where fqdn is fully qualified domain name of the VM and metric is type of the metric, for example 'tashi_vm_ExpenseSystemvm-1_cpuLoad'.

A shortened sample of Ganglia report:

```xml
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<!DOCTYPE GANGLIA_XML [ ...
]>   
<GANGLIA_XML VERSION="3.1.2" SOURCE="gmetad">
  <GRID NAME="Xlab" AUTHORITY="http://muca01/ganglia/" LOCALTIME="1308580592">
    <CLUSTER NAME="unspecified" LOCALTIME="1308579110" OWNER="unspecified"
LATLONG="unspecified" URL="unspecified">
      <HOST NAME="muca04.xlab.tst" IP="10.100.0.4" REPORTED="1308579108" TN="15"
TMAX="20" DMAX="0" LOCATION="unspecified" GMOND_STARTED="1303133370">
        <METRIC NAME="mem_free" VAL="3428672" TYPE="float" UNITS="KB" TN="39" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond">
          <EXTRA_DATA>
            <EXTRA_ELEMENT NAME="GROUP" VAL="memory"/>
            <EXTRA_ELEMENT NAME="DESC" VAL="Amount of available memory"/>
            <EXTRA_ELEMENT NAME="TITLE" VAL="Free Memory"/>
          </EXTRA_DATA>
        </METRIC>
        <METRIC NAME="disk_free" VAL="977.551" TYPE="double" UNITS="GB" TN="99" TMAX="180"
DMAX="0" SLOPE="both" SOURCE="gmond">
          <EXTRA_DATA>
            <EXTRA_ELEMENT NAME="GROUP" VAL="disk"/>
            <EXTRA_ELEMENT NAME="DESC" VAL="Total free disk space"/>
            <EXTRA_ELEMENT NAME="TITLE" VAL="Disk Space Available"/>
          </EXTRA_DATA>
        </METRIC>
        <METRIC NAME="tashi_vm_ExpenseSystemvm-1_cpuLoad" VAL="0.013" TYPE="float"
UNITS="" TN="269161" TMAX="60" DMAX="0" SLOPE="both" SOURCE="gmond">
          <EXTRA_DATA>
            <EXTRA_ELEMENT NAME="GROUP" VAL=""/>
            <EXTRA_ELEMENT NAME="DESC" VAL=""/>
            <EXTRA_ELEMENT NAME="TITLE" VAL=""/>
          </EXTRA_DATA>
        </METRIC>
      </HOST>
    </CLUSTER>
  </GRID>
</GANGLIA_XML>
```
The sample report contains metrics data for the host machine muca04.xlab.tst. Two host metrics are shown mem_free (Free Memory) and disk_free (Disk Space Available) and two VM related metrics: 'tashi_vm_ExpenseSystemvm-1_cpuLoad' and 'tashi_vm_ExpenseSystemvm-2.xlab.tst_cpuLoad'.

Host metrics reported by the Ganglia that are used by the IMA:

- **Free Memory** (mem_free): total amount of free physical memory in the system.
- **Total Memory** (mem_total): total amount of physical memory in the system.
- **CPU Speed** (cpu_speed): speed (clock frequency) of the CPU
- **CPU System** (cpu_system): percentage of CPU utilization that occurred while executing at the system level
- **CPU User** (cpu_user): percentage of CPU utilization that occurred while executing at the user level
- **CPU Idle** (cpu_idle): percentage of time that the CPU or CPUs were idle and the system did not have an outstanding disk I/O request
- **Disk Space Available** (disk_free): total free disk space
- **Total Disk Space** (disk_total): total available disk space
- **One Minute Load Average** (load_one): the average sum of the number of processes waiting in the run-queue plus the number currently executing over 1 minute time period.
- **Five Minute Load Average** (load_five): the average sum of the number of processes waiting in the run-queue plus the number currently executing over 5 minute time period.

VM related host metrics reported by the Ganglia that are used by the IMA:

- **Memory Usage** (tashi_vm_<fqdn>_rss): host memory used by the VM. It is measured by the virtualization software (QEMU) process memory usage. RSS stands for Resident Set Size and means the portion of a process's memory that is held in RAM.
- **CPU Load** (tashi_vm_<fqdn>_cpuLoad): cpu usage of QEMU process corresponding to the VM expressed as a ratio. The value 1 means that one CPU core is fully loaded.

<<get_tashi_metrics>>

The <<get_tashi_metrics>> interaction allows the IMA to get monitoring data from the Tashi. Communication between the IMA and Tashi runs through point-to-point XMPP protocol. The IMA periodically queries Tashi so that it sends appropriate request through XMPP protocol to the Tashi user which returns corresponding monitoring data.

Tashi provides two types of monitoring data:

- VmLayout
- ClusterConfiguration
VmLayout report

VmLayout report provides a list of all host machines and VMs running on them, total host resources (number of CPU cores, CPU speed and memory), host resources reserved by VMs, VM characteristics and state. The format of the report is XML.

A sample VmLayout report:

```
<?xml version="1.0" ?>
<Message time="2011-06-21T2:32:13.833841+2:00" type="VmLayout">
  <NetworkEntities>
    <NetworkEntity cpu_cores="8" cpu_speed="12768" fqdn="muca04.xlab.tst" memory="5980" name="muca04" type="host" up="1" used_cpu_cores="2" used_cpu_speed="2048" used_memory="256">
      <NetworkEntity cpu_cores="1" cpu_speed="1024" disk="[{'uri': 'lenny1.qcow2', 'persistent': 'False'}]" fqdn="ExpenseSystemvm-1.xlab.tst" memory="128" name="ExpenseSystemvm-1" state="Running" type="vm" userId="1006"/>
      <NetworkEntity cpu_cores="1" cpu_speed="1024" disks="[{'uri': 'lenny1.qcow2', 'persistent': 'False'}]" fqdn="ExpenseSystemvm-2.xlab.tst" memory="128" name="ExpenseSystemvm-2" state="Running" type="vm" userId="1006"/>
    </NetworkEntity>
    <NetworkEntity cpu_cores="8" cpu_speed="13966" fqdn="muca03.xlab.tst" memory="3960" name="muca03" type="host" up="0" used_cpu_cores="0" used_cpu_speed="0" used_memory="0"/>
  </NetworkEntities>
</Message>
```

Host metrics in the VmLayout report:
- **cpu_cores**: number of CPU cores
- **cpu_speed**: total CPU speed (sum of all cores)
- **memory**: total amount of physical memory in the system
- **up**: state of the machine (up/down)
- **used_cpu_cores**: number of CPU cores reserved by the VMs (sum of CPU cores of all guest machines)
- **used_cpu_speed**: CPU speed reserved by the VMs (sum of CPU speed of all guest machines)
- **used_memory**: memory size reserved by the VMs (sum of memory size of all guest machines)

VM metrics in the VmLayout report:
- **state**: state of the virtual machine (can be pending, activating, running, pausing, paused, unpausing, suspending, resuming, migratet prep, migratet trans, shuttingdown, destroying, orphaned, held, exited, not_responding)
- **cpu_cores**: number of CPU cores
- **cpu_speed**: CPU speed reserved by the VM
- **disks**: contains attributes
  - **uri**: VM image template from which the VM was created
  - **persistent**: tells if VM disk is persistent (stored on host machine disk) or transient (stored only in host machine RAM)
- **memory**: VM memory size
- **userId**: user ID of the owner of the VM
ClusterConfiguration report

ClusterConfiguration report provides some characteristics of the host machines:

- **Auditability**: tells whether auditability (audit trail) is enabled on the host machine
- **Location**: location of the host machine (ISO 3166-1 country code)
- **SAS70**: tells if host machine is compliant with SAS 70 auditing standard (Statement on Auditing Standards No. 70)
- **CCR**: tells if host machine is located in CCR region (Controlled Country Region). CCR region consists of controlled and embargoed countries.
- **DataClassification**: security level of data stored on the host machine (public, confidential or secret)
- **HWRedundancyLevel**: redundancy level of host machine hardware (standard, better or best)
- **DiskThroughput**: host machine disks throughput level (standard, better or best)
- **NetThroughput**: host machine network throughput level (standard, better or best)
- **DataEncryption**: tells whether data is encrypted on the host machine disks where VMs images are stored

A sample ClusterConfiguration report:

```xml
<Message time="2011-06-21T2:32:13.850828+2:00" type="ClusterConfiguration">
  <ClusterConfiguration fqdn="tashi.xlab.tst">
    <Host fqdn="muca04.xlab.tst" id="1">
      <Location>
        <CountryCode>SI</CountryCode>
      </Location>
      <Auditability>True</Auditability>
      <SAS70>False</SAS70>
      <CCR>False</CCR>
      <DataClassification>Public</DataClassification>
      <HWRedundancyLevel>Standard</HWRedundancyLevel>
      <DiskThroughput>Standard</DiskThroughput>
      <NetThroughput>Standard</NetThroughput>
      <DataEncryption>False</DataEncryption>
    </Host>
  </ClusterConfiguration>
</Message>
```
<<publish_tashi_event>>

Tashi notifies the Infrastructure Monitoring Agent about various events, for example VM creation, changing state of VMs, host registration, scheduling of VMs, etc. Tashi sends notification messages in XML format through point-to-point XMPP protocol. There are two types of messages according to event source:

- Tashi audit records
- Scheduler events

Infrastructure Monitoring Agent stores the Tashi events to the database for possible further queries from the frontend application. The IMA also publishes the events to the monitoring publish/subscribe channel to be immediately displayed in the GUI (<<display_monitoring_event>> interaction).

A sample Tashi audit record message:

```xml
<?xml version="1.0" ?>
<Message id="muca04.xlab.tst" time="2011-06-21T14:44:03.397073+2:00" type="auditRecord" userId="admin">
  <Action>VM ExpenseSystemvm-1 on host muca04.xlab.tst paused</Action>
</Message>
```

A sample scheduler event:

```xml
<?xml version="1.0" ?>
<Message time="2011-06-21T14:42:10.403874+2:00" type="scheduler">
  <Action>Scheduling instance ExpenseSystemvm-1 (128 mem, 1 cores, 1006 uid) on host muca04.xlab.tst</Action>
</Message>
```

<<publishViolationEvent>>

The <<publishViolationEvent>> interaction allows the IMA to inform the Provisioning and Adjustment (InfrastructurePAC) component when SLA violations are detected at infrastructure services being monitored. The communication goes through a publish/subscribe messaging protocol. The IMA publishes violation events messages to the event channel where the PAC listens. The IMA also informs PAC when a possibility of SLA violation is detected and publishes warning event message.

Violation and warning event messages are formatted in XML format according to the MonitoringEvent schema. A sample violation event message for a case of Memory QoS term violation is as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
  <ns3:EventID>
    <ns3:ID>7182340883285100550</ns3:ID>
    <ns3:EventTypeID>InfrastructureMonitoringAgent.SLA_VIOLATION</ns3:EventTypeID>
  </ns3:EventID>
  <ns3:EventContext>
    <ns3:Time>
      <ns3:Timestamp>2011-03-09T09:15:13.926</ns3:Timestamp>
      <ns3:CollectionTime>2011-03-09T09:15:13.926+01:00</ns3:CollectionTime>
      <ns3:ReportTime>2011-03-09T09:15:43.932+01:00</ns3:ReportTime>
    </ns3:Time>
  </ns3:EventContext>
</ns3:Event>
```
<ns3:Source>
  <ns3:InfrastructureLayer name="InfrastructureMonitoringAgent"/>
</ns3:Source>

<<publish_metric_values>>

The <<publish_metric_values>> interaction allows the IMA to inform the Provisioning and Adjustment (InfrastructurePAC) component about current values of QoS terms guaranteed by the SLA for all infrastructure services being monitored. The communication goes through a publish/subscribe messaging protocol. The IMA publishes metrics report messages to the event channel where the PAC listens.

Metrics report messages are formatted in XML format according to the MonitoringEvent schema. A sample shortened message for an infrastructure service with two resources is as follows:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns3:Event xmlns:ns2="http://www.slaatsoi.org/commonTerms"
          xmlns:ns4="http://www.slaatsoi.org/coremodel"
          xmlns:ns3="http://slasoi.org/monitoring/xml/eventformat">
  <ns3:EventID>
    <ns3:ID>7182340883285100550</ns3:ID>
    <ns3:EventTypeID>InfrastructureMonitoringAgent.METRICS_REPORT</ns3:EventTypeID>
  </ns3:EventID>
  <ns3:EventContext>
    <ns3:Time>
      <ns3:Timestamp>1299674984904</ns3:Timestamp>
      <ns3:CollectionTime>2011-03-09T13:49:44.904+01:00</ns3:CollectionTime>
    </ns3:Time>
    <ns3:Source>
      <ns3:InfrastructureLayer name="InfrastructureMonitoringAgent"/>
    </ns3:Source>
  </ns3:EventContext>
  <ns3:EventPayload>
    <ns3:MonitoringResultEvent>
      <ns3:MonitoringInfo>
        <InfrastructureService>
          <InfrastructureId>945b99c6-7f81-4738-bbc2-50db8662f2be</InfrastructureId>
          <Resources>
            <Resource>
              <ResourceId>46c3ceb0-1439-4433-9417-1410727e828a</ResourceId>
              <Metrics>
                <Metric type="Memory" unit="MB" value="210.08"/>
                ...
              </Metrics>
            </Resource>
            <Resource>
              <ResourceId>bd83jd93-5816-4843-1028-39032d82323</ResourceId>
              <Metrics>
                <Metric type="Memory" unit="MB" value="168.43"/>
                ...
              </Metrics>
            </Resource>
          </Resources>
        </InfrastructureService>
      </ns3:MonitoringInfo>
    </ns3:MonitoringResultEvent>
  </ns3:EventPayload>
</ns3:Event>
```
<<get_monitoring_data>>

The <<get_monitoring_data>> interaction is used by the frontend application to get monitoring data from the IMA and then present that information to the customer or administrator. The communication goes through a publish/subscribe messaging protocol. The frontend application sends request message to the monitoring data channel, the IMA receives it, prepares required data, generates the response message and sends it via the same channel back to the frontend application serialized in JSON format.

Several operations are available:

- **GetHostsInfo**: returns a list of all host machines with certain host metrics and a list of all guest machines (VMs) for each host with certain VM metrics
- **GetMetricHistory**: returns metric value history for specified metric and time interval of specified VM or infrastructure service. The density of values is limited by the maximum number of points.
- **GetSchedulerEvents**: returns scheduler events history
- **GetServiceEvents**: returns cluster manager audit records history related to given infrastructure service or user
- **GetServicesInfo**: returns a list of all infrastructure services, for each service state, some details and statistics and a list of resources. For each resource certain metrics are given.
- **GetServiceSLASummary**: returns SLA summary for given infrastructure service which contains a list of all service QoS terms and a list of all VM QoS terms for each resource. For each QoS term a current value, violation threshold and state (compliance with the SLA) is given.
- **GetServiceViolationsFrequency**: returns SLA violations and warnings frequency for given infrastructure service per reporting periods
- **GetServiceViolations**: returns a detailed list of SLA violations and warnings for given infrastructure service in the given time period
- **GetInfrastructureInfo**: returns monitored cluster infrastructure info
- **GetSysInfo**: returns details and metric values for given host or VM

<<display_monitoring_event>>

The <<display_monitoring_event>> interaction allows the IMA to inform the frontend application about certain monitoring events immediately after they were detected by the push method. The communication goes through a publish/subscribe messaging protocol. The IMA publishes notification messages serialized to the JSON format to the monitoring data channel.

A sample notification message:
The <<generate_report>> interaction is used by the frontend application to request various reports of infrastructure services being monitored from the IMA. The communication goes through a publish/subscribe messaging protocol. The frontend application sends request message to the monitoring data channel, the IMA receives it, generates requested report in the PDF format and returns it via the same channel packed in response message serialized to JSON format.

6.8.3 QoS Terms

Supported QoS terms

The IMA supports following core SLA model guaranteed terms (QoS terms):

- Service Availability (SERVICE_AVAILABILITY)
- Service Isolation (SERVICE_ISOLATION)
- VM Availability (VM_AVAILABILITY)
- VM Cores (VM_CORES)
- VM Cpu Speed (VM_CPU_SPEED)
- VM Image (VM_IMAGE)
- VM Memory Size (VM_MEMORY_SIZE)
- VM Persistence (VM_PERSISTENCE)
- VM Quantity (VM_QUANTITY)

In addition the IMA computes and records history of some metrics which are not SLA-guaranteed terms but are needed at QoS terms computation (e.g. Service Availability Status) or for generating reports (e.g. VM CPU Speed Used). These metrics are:

- Service Availability Status (SERVICE_AVAILABILITY_STATUS)
- Service SLA Compliance (SERVICE_SLA_COMPLIANCE)
- VM Availability Status (VM_AVAILABILITY_STATUS)
- VM CPU Speed Used (VM_CPU_SPEED_USED)
- VM Memory Size Used (VM_MEMORY_SIZE_USED)
- VM SLA Compliance (VM_SLA_COMPLIANCE)
**Service Availability**

Service Availability is ratio of time from the start of service monitoring expressed as a percentage when the service was available (up). The service is considered to be available if all of its resources (VMs) are available. VM is available if it is up and running. Service Availability is calculated based on the SERVICE_AVAILABILITY_STATUS metric history, which records the availability status (up or down) of the service through time. Service Availability is ratio of time the service was available divided by the whole time the service was monitored. The QoS term becomes violated if the value falls below the violation threshold. For this QoS term it is possible to generate warnings if the value is falling towards the violation threshold. Warning is emitted if value falls below the warning threshold (but is still above the violation threshold).

**Service Isolation**

Service Isolation term requires that all VMs which belong to the service run on separate physical host machines. Neither two of the VMs must not be provisioned on the same host machine and also must not be moved to the same machine by the scheduler. If the IMA detects that two VMs run on the same host machine the QoS term becomes violated. Note that VMs can share physical machines with other services.

**VM Availability**

VM Availability is ratio of time from the start of VM monitoring expressed as a percentage when the VM was available (up and running). The VM is considered to be available if the cluster manager reports that VM is in a running state. VM Availability is calculated based on the VM_AVAILABILITY_STATUS metric history, which records the availability status (up or down) of the VM through time. VM Availability is ratio of time the VM was available divided by the whole time the VM was monitored. The QoS term becomes violated if the value falls below the violation threshold. For this QoS term it is possible to generate warnings if the value is falling towards the violation threshold. Warning is emitted if value falls below the warning threshold (but is still above the violation threshold).

**VM Cores**

VM Cores is number of cores on the host machine reserved and available to the VM. The term becomes violated if actual number of cores reported by the monitoring engine is lower than the value agreed by the SLA.

**VM Image**

VM Image is an image template name by which the VM was created. The IMA checks if the actual image by which the VM was created corresponds to the image agreed by the SLA. In case it does not the term becomes violated.

**VM Memory Size**

VM Memory Size is amount of RAM on the host machine available to the VM. The IMA checks:

- if the cluster manager allocated enough memory to the VM
- if there is always enough free memory on the host machine that the VM can get the agreed amount of memory (the VM doesn't necessarily use all of the allocated memory). Memory available to the VM is calculated as
sum of memory currently used by the VM and the free memory on the host. Memory available to the VM should always be greater than the memory size agreed upon in the SLA.

If any of these conditions is not met the term becomes violated. If the memory available to the VM is near the agreed upon amount of memory (i.e. if it falls below the warning threshold), a warning is generated and the QoS term moves to the 'warning' state.

**VM Persistence**

VM is persistent if its virtual disk image is stored on the host hard disk. In case it is stored in host RAM then the VM is not persistent. The term is violated if actual persistence does not match the persistence agreed by the SLA, i.e. if the VM should be persistent according to the SLA and in fact is not and vice versa.

**VM Quantity**

VM Quantity is the number of VMs contained in the SLA agreement. The IMA checks if all VMs were successfully provisioned and are reported by the cluster manager. The term is violated if actual number of VMs reported by the cluster manager does not match the value specified by the SLA.

**Service Availability Status**

The metric Service Availability Status records the availability status (up or down) of the service through time. The service is considered to be available if all of its VMs are available. VM is considered to be available if cluster manager reports its state is running. The metric Service Availability Status is used for calculation of QoS term Service Availability and for generating reports.

**Service SLA Compliance**

The metric Service SLA Compliance records the service SLA compliance state through time. The state can be compliant (if service is compliant with the SLA) or not compliant. The metric history can be used for calculation of some special QoS terms like Mean Time To Violation (not in the core SLA model) and for generating reports.

**VM Availability Status**

The metric VM Availability Status records the availability status (up or down) of the VM through time. The VM is considered to be available if cluster manager reports its state is running. The metric is used for calculation of QoS term VM Availability and for generating reports.

**VM CPU Speed Used**

The metric VM CPU Speed Used records the host CPU speed usage by the VM and is expressed as a frequency. It is calculated by multiplying CPU usage of virtualization software process (qemu) corresponding to the VM expressed as a percentage with host CPU speed. The metric is used by frontend application for displaying CPU usage graph and for generating reports.

**VM Memory Size Used**

The metric VM Memory Size Used records the host memory usage by the VM. It corresponds to the Resident Set Size (RSS) of virtualization software process (qemu) corresponding to the VM. Resident Set Size is the portion of a process's
memory that is held in RAM. The metric is used by frontend application for displaying memory usage graph and for generating reports.

**VM SLA Compliance**

The metric VM SLA Compliance records the VM SLA compliance state through time. The state can be compliant (if VM is compliant with the SLA) or not compliant. The metric history can be used for calculation of some special QoS terms like Mean Time To Violation (not in the core SLA model) and for generating reports.

**Extending IMA with new QoS terms**

The IMA is designed as modular as possible to be easily extendable with new QoS terms. In order to implement a new QoS term, the following needs to be done:

1. Implement a class in the metrics package that extends the abstract Metric class and implement functionality for computing the QoS term value and validating it against the SLA.
2. Add the new QoS term to the MetricTypeEnum enum.
3. Add the new QoS term with required attributes to the metric_type database table.

**Implementing a new QoS term class**

Each metric or QoS term is implemented in a separate class, all of which are located in the metrics package. Create a new class for the new QoS term which extends abstract class Metric and implements following methods:

```java
void compute(IMonitoringEngine monitoringEngine);
```

The method `compute()` should compute the value of the QoS term/metric based on the monitoring data retrieved from the monitoring engine (given by the parameter `monitoringEngine`), values of other metrics and their history. In case the new metric is a SLA model guaranteed term the computed value should be validated against the SLA. The method should check if violation constraint is met and generate a violation in case it is not. If warning threshold is defined (for numerical metrics) it should be checked also.

```java
public String formatValue(Object value);
```

The method `formatValue()` should format the metric value into a form suitable for storing into the database (metric values are stored in the database as strings).

```java
public String getDisplayValue(Object value);
```

The method `getDisplayValue()` should format the metric value into a human-readable form suitable for displaying in the frontend application. The formatted value should include a metric unit if appropriate.

```java
public Object getViolationThresholdValue();
```

The method `getViolationThresholdValue()` should return metric violation constraint (threshold for numerical metrics) in a suitable type.

```java
public Object getWarningThresholdValue();
```

The method `getWarningThresholdValue()` should return metric warning threshold in a suitable type. Warnings are applicable only for certain numerical QoS terms, like Service Availability. If they are not applicable to the QoS term in question the method should return null.
A shortened sample of Service Availability QoS term implementation:

```java
@Entity
@DiscriminatorValue("SERVICE_AVAILABILITY")
public class ServiceAvailability extends Metric {

@Override
public void compute(IMonitoringEngine monitoringEngine) throws Exception {
    Service service = this.getService();
    SLAComplianceEnum prevSLACompliance = this.getPrevSLACompliance();
    Double availabilityRatio;
    try {
        Metric availStatusMetric = service.getMetric(MetricTypeEnum.SERVICE_AVAILABILITY_STATUS);
        availabilityRatio = MetricsComputationHelper.computeAvailabilityTotal(availStatusMetric);
    } catch (Exception e) {
        throw new Exception(String.format("Failed to compute metric SERVICE_AVAILABILITY of the service '\"%s\"', service.getServiceUrl(), e.getMessage()));
    }

    Date metricsTimestamp = monitoringEngine.getMetricsCollectedDate();
    MetricValue metricValue = storeMetricValue(availabilityRatio, metricsTimestamp);
    storeMetricHistoryValue(availabilityRatio, metricsTimestamp);

    // validate metric value against SLA
    if (this.getViolationThreshold() != null) {
        double violationThreshold = this.getViolationThresholdValue();
        SLAComplianceEnum slaCompliance = SLAComplianceEnum.COMPLIANT;
        if (availabilityRatio < violationThreshold) {
            slaCompliance = SLAComplianceEnum.VIOLATION;
            log.warn(...);
        } else if (this.getWarningThreshold() != null) {
            double warningThreshold = this.getWarningThresholdValue();
            if (availabilityRatio < warningThreshold) {
                slaCompliance = SLAComplianceEnum.WARNING;
                log.warn(...);
            }
        }
    }

    updateMetricSLACompliance(this, metricValue, slaCompliance, metricsTimestamp);
    ViolationHandler.handleViolation(this, metricValue, slaCompliance, prevSLACompliance);
}

@Override
public String formatValue(Object value) {
    if (value == null) {
        return null;
    } else {
        return String.format("%.4f", (Double)value);
    }
}

@Override
public String getDisplayValue(Object value) {
    if (value == null) {
        return null;
    } else {
        return formatValue(value) + " %";
    }
}

@Override
public Double getViolationThresholdValue() {
    if (this.getViolationThreshold() == null) {
        return null;
    } else {
    }
```

### 6.8.4 CPU Share Monitoring

Monitoring actual CPU usage by a virtual machine is performed by sampling process tick counters for that virtual machine's KVM process. A tick is a (rather long) interval, whose duration is system specific and can be determined by a libc call `sysconf(_SC_CLK_TCK)` and is commonly 1/100 of a second on most Linux kernels.

A process’ tick counters increase by one for every tick that a process spends in one of its possible states; the set of tick counters that are increased is determined by the state of the process (user space code execution, kernel space code execution, interrupt handling etc.) during that tick, and the counters for process with pid P are available in the file `/proc/P/stat`. Note that execution tick counters are increased on a per-core basis (tick counters of a process with four threads that all simultaneously execute on four cores, would increase by four on every tick).

As described in section 6.7.4, the KVM process does not perform solely guest computation but other work as well. Thus, it is important that we only measure ticks that the KVM process’ threads spend executing guest code. Fortunately, one of the tick counters available in recent Linux kernels measures precisely the ticks spent executing guest code: let us denote it with `guestticks(vm)` and its value at time $T$ as $guestticks(vm, T)$.

Furthermore, let us denote host tick frequency with `tickfreq`. We can compute the total number of ticks on a host in a time interval of duration $\text{int}$ as:

$$ticks(host, int) = \text{int} \times \text{ncores}(host) \times \text{tickfreq}$$

The share of ticks (and thus of CPU time) spent executing guest code of a virtual machine $vm$ in the last interval of duration $\text{int}$ at time $T$ can then be computed as:

$$\text{ticksshare}(vm, \text{int}) = (guestticks(vm, T) - guestticks(vm, T-\text{int}))/ticks(host, \text{int})$$

In this manner, sampling the `guestticks(vm)` value for a virtual machine with a period $\text{int}$ can provide us with the share of total system ticks spent executing guest code for a virtual machine, a relative measure of the computation performed on behalf of that virtual machine.

In order to compare this relative measure with the CPU speed SLA term reflected in the `cpu.shares` weight that we used to allocate CPU share to a virtual machine (see description of CPU allocation in section 6.7.4), we must first transform it to the same domain; recalling the `tcpu(host)` value computed there, the following is obvious:

$$mcpu(vm, \text{int}) = tcpu(host) \times \text{ticksshare}(vm, \text{int})$$

And therefore:
\[
mcpu(vm, int) = \frac{tcpu(host) \times (guestticks(vm, T) - guestticks(vm, T-int))}{ticks(host, int)} = \frac{clock(host) \times (guestticks(vm, T) - guestticks(vm, T-int))}{int \times tickfreq}
\]

Note that the expression does not explicitly refer to the core count of the host, as this is implied in the \textit{guestticks} measure.

The CPU share allocation scheme as described in section 6.7.4 guarantees that as long as the host is not fully loaded, the CPU speed SLA terms of all the virtual machines are implicitly satisfied. Therefore we need a mechanism to measure when the host is fully loaded.

The file /proc/stat contains the aforementioned tick counters for the whole system (instead of per-process ones we used before). However, as the "loaded" ticks are divided into multiple classes, it is simpler to sample idle tick counter and simply subtract those from the maximum \textit{ticks(host, int)} value:

\[
idleshare(vm, int) = \frac{idleticks(vm, T) - idleticks(vm, T-int)}{ticks(host, int)}
\]

As long as \textit{idleshare(vm, int)} > 0, all the SLA terms are implicitly satisfied.

When the opposite is true, i.e. \textit{idleshare(vm, int)} = 0, there are two possible scenarios.

If the host is not overcommitted, i.e. \textit{sum(cpu(vm))} ≤ \textit{guestcpu(host)} (see section 6.7.4), each virtual machine is always getting at least its required CPU share. This situation can only occur when the host is fully loaded due to unused cycles dedicated to guests, which are spent by the system processes.

Finally, in the remaining case, where the host is overcommitted – when \textit{sum(cpu(vm))} > \textit{guestcpu(host)} – and the share of idle ticks is 0, it is likely that one of the virtual machines is violating its SLA in terms of provided CPU speed. Analysing the relation between \textit{mcpu(vm)} and \textit{cpu(vm)} in this case, we can differentiate the following scenarios:

1. \textit{mcpu(vm)} ≥ \textit{cpu(vm)}: virtual machine is sufficiently (perhaps even over-) provisioned;
2. \textit{mcpu(vm)} < \textit{cpu(vm)}: virtual machine may be violating the CPU speed SLA term.

However, can we tell anything more about possible violations than the rather weak »may« in scenario (2) by merely observing the virtual machines from the host?

First of all, if at least one of the virtual machines is overprovisioned (that is, strict inequality holds in (1) above), no virtual machines on that host are in violation of their CPU speed SLA term: if any of the machines were underprovisioned (and thus violating SLA), the cgroups would allocate the extra CPU cycles from the overprovisioned virtual machine(s) to the underprovisioned one.

Now, in a situation where \textit{mcpu(vm)} ≤ \textit{cpu(vm)} is true for all virtual machines, assume the host is overcommitted by a factor K:

\[
sum(cpu(vm)) = K \times guestcpu(host)
\]

If all the virtual machines were consuming all the CPU cycles provided to them, the following would hold (cgroups logic distributes the CPU cycles according to cpu.shares weights) for each virtual machine vm:

\[
cpu(vm)/mcpu(vm) = K
\]
Thus, the machines with $\text{cpu(vm)/mcpu(vm)} < K$ are not violating the SLA (they do not even consume all the CPU cycles that would be allocated to them by cgroups if they needed them).

As far as the remaining virtual machines go, it is almost certain that they are in violation of the SLA (the only exception being that the virtual machine is getting exactly the number of CPU cycles it can consume by its computation, which is a highly improbable situation).

### 6.9 OCCI I-SM for OpenNebula

Although Apache Tashi was selected as the primary provisioning system for the reference implementation, the Infrastructure SLA Manager was designed to manipulate any provisioning system that supports the OCCI standard.

During the third year of the project, the Technical University of Dortmund identified resources that could help OCCI-enable the OpenNebula provisioning system, a system progressed by the RESERVOIR project with which SLA@SOI had collaborated.

As the Cloud provisioning system OpenNebula already had limited support for the OCCI standard in version 1.0 it was decided to update the implementation to fully support the new OCCI 1.1 standard and thus enable it to be used by the SLA@SOI framework.

Analysis of the existing implementation showed that a rewrite from scratch was easier to support the features newly introduced in both OpenNebula and OCCI.

In a first step, the architecture for the new implementation was designed as shown in Figure 24. The architecture consists of three parts, namely a webserver running the OCCI 1.1 service, the OCCI framework, and a backend specification to support arbitrary Cloud provisioning systems.

![Figure 24: Architecture of OCCI I-SM for OpenNebula](image)
The OCCI Framework itself is designed to be modular and extendible to support future versions of the OCCI standard as well as changes in the backend APIs. The main component is the Request Processing module. It carries out the following steps:

1. Receive OCCI request from webserver
2. Parse request using the OCCI ANTLR grammar into objects from OCCI Model (see Section 6.6)
3. Carry out REST specific functionality for GET, POST, PUT, DELETE by registering new objects to or retrieving information on existing objects from the Location Registry
4. Use Backend API to pass information to or retrieve information from a specific backend
5. Pass information to the Response Rendering module

If a request has been processed, the Response Rendering module uses the OCCI ANTLR parser again to render the information from the Request Processing according to the Rendering specification of the OCCI standard. This information is then passed to the Webserver to be delivered to the client.

The OCCI Framework has been developed in the Ruby programming language for two reasons. First, OpenNebula offers a Ruby and a Java API which helps implementing the backend part. Second, the Ruby language allows for rapid prototyping which was required due to the short remaining time of the SLA@SOI project.

The OpenNebula developer team gladly offered support in developing the OCCI Framework and agreed to make it an OpenNebula Ecosystem project. The source code has been released in the official OpenNebula repositories under an Apache 2.0 license [61] and documented on the OpenNebula wiki [62]. The license was chosen because OpenNebula already uses an Apache 2.0 license and because it is compatible with the BSD license used by the SLA@SOI project.

A test service has been setup by TU Dortmund for regression and interoperability testing. At both of the SNIA Cloud Plugfest [63] events held to date by the Storage Network Infrastructure Association, SLA@SOI members participated and interoperability tests have been carried out together with the other attendees. The progress on this component has recently presented to the "Workshop on Science Agency Uses of Clouds and Grids" during OGF 32 [64].
7 Deployment

The Infrastructure Management work package has deployed dedicated testbed infrastructure to support the SLA@SOI project. This includes both servers available on the internet to all partners and use cases within the consortium, and workstations on a local area network for ad hoc development and demonstration purposes.

The main external testbed has been hosted by Intel In year 3 this was complemented with an external testbed hosted by XLAB. This was used by the B4 Enterprise IT use case in particular. Intel also deployed an internal testbed for internal SLA@SOI development and testing purposes.

Rather than construct state-of-the-art testbeds with the latest hardware and software, a much more realistic approach was adopted: the testbeds were constructed using a mix of hardware platforms, with different CPUs and levels of hardware support for virtualisation technologies.

7.1 External Testbed – Intel

The external testbed hosted by Intel is the main testbed for the project and has been used by the B3 Open Reference Case, B4 Enterprise IT and B5 Service Aggregation work packages. In year three, the size of this test bed was scaled up from 11 physical hosts to 17. This allowed it to be used to facilitate development of the Data Centre Optimisation application designed to use IPMI commands to power on and off physical systems and record the power savings. The infrastructure was partitioned into 1 Apache Tashi cluster manager, 13 Tashi node managers, and 3 prediction processing nodes as illustrated in Figure 25.

![Figure 25: Intel external testbed](image)
The physical attributes of a selection of the nodes is listed in Table 9.

**Table 9: Intel external testbed physical node attributes**

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Up</th>
<th>Decayed</th>
<th>state</th>
<th>Memory</th>
<th>cores</th>
<th>Cpu speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>iricvtb002</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>32239</td>
<td>16</td>
<td>54256</td>
</tr>
<tr>
<td>2</td>
<td>iricbl010</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>3</td>
<td>iricvtb001</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>32240</td>
<td>16</td>
<td>54256</td>
</tr>
<tr>
<td>4</td>
<td>iricbl014</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>5</td>
<td>iricbl011</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>6</td>
<td>iricbl009</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>7</td>
<td>iricbl001</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12040</td>
<td>8</td>
<td>21328</td>
</tr>
<tr>
<td>8</td>
<td>iricbl002</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12040</td>
<td>8</td>
<td>21328</td>
</tr>
<tr>
<td>9</td>
<td>iricbl003</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>10</td>
<td>iricbl004</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>11</td>
<td>iricbl005</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>12</td>
<td>iricbl006</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>8000</td>
<td>8</td>
<td>14928</td>
</tr>
<tr>
<td>13</td>
<td>iricvtb013</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>12040</td>
<td>4</td>
<td>32256</td>
</tr>
</tbody>
</table>

In addition to the standard physical server attributes that Tashi tracks (as shown in Table 9), some of the nodes were assigned with additional attributes to support the B4 Enterprise IT use case in particular. This allowed measuring of adherence to SLA QoS terms such as location, whether the server was in a ‘controlled country, and the classification of the sensitivity of the information on the server. These supplementary attributes could be taken into account by the B4 Enterprise IT use case scheduler, as described in Section 6.7 and also Deliverable B4c [11].

**Table 10: Supplementary Intel external testbed physical node attributes**

<table>
<thead>
<tr>
<th>Location</th>
<th>CCR</th>
<th>Efficiency</th>
<th>Hw Redun</th>
<th>Disk Thruput</th>
<th>Net Thruput</th>
<th>SAS 70</th>
<th>Data Class</th>
<th>Pwr CAP</th>
<th>Pwr STAT</th>
<th>Intel TXT</th>
<th>Intel AES-NI</th>
</tr>
</thead>
<tbody>
<tr>
<td>GER</td>
<td>0</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>1</td>
<td>secret</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLO</td>
<td>0</td>
<td>12</td>
<td>standard</td>
<td>standard</td>
<td>0</td>
<td>public</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IRE</td>
<td>0</td>
<td>22</td>
<td>best</td>
<td>best</td>
<td>1</td>
<td>secret</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SLO</td>
<td>0</td>
<td>3</td>
<td>standard</td>
<td>standard</td>
<td>0</td>
<td>public</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ITA</td>
<td>0</td>
<td>6</td>
<td>standard</td>
<td>standard</td>
<td>0</td>
<td>public</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>CHI</td>
<td>1</td>
<td>9</td>
<td>standard</td>
<td>standard</td>
<td>0</td>
<td>public</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GER</td>
<td>0</td>
<td>11</td>
<td>better</td>
<td>better</td>
<td>1</td>
<td>confidential</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IRE</td>
<td>0</td>
<td>11</td>
<td>better</td>
<td>standard</td>
<td>0</td>
<td>public</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>IRE</td>
<td>0</td>
<td>9</td>
<td>standard</td>
<td>standard</td>
<td>1</td>
<td>public</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GER</td>
<td>0</td>
<td>9</td>
<td>standard</td>
<td>standard</td>
<td>0</td>
<td>public</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SLO</td>
<td>0</td>
<td>9</td>
<td>standard</td>
<td>standard</td>
<td>1</td>
<td>public</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>ITA</td>
<td>0</td>
<td>9</td>
<td>standard</td>
<td>better</td>
<td>1</td>
<td>confidential</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CHI</td>
<td>1</td>
<td>29</td>
<td>best</td>
<td>best</td>
<td>1</td>
<td>secret</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
7.2 External Testbed – XLAB

In year three of the project a second external testbed was deployed and made available by XLAB. This allowed the possibility for XLAB to research how best to interface the Tashi scheduler they had developed with a networked remote power controller (a Trendnet 8-outlet Remote Power Controller [REF42]) connected to their physical hosts. The powering on and off of their testbed nodes would not affect any users of the primary testbed.

The XLAB hosted testbed consists of 11 servers with two different types of configuration. Ten computers are configured as working nodes, the eleventh as a head node, requiring two Ethernet connections. Additional hardware provisioned includes a switch and the remote power controller, the latter allowing an array of energy-efficient algorithms to be deployed and tested. The complete testbed is illustrated in Figure 26.

![Figure 26: XLAB external testbed](image)

Debian Squeeze 6.0 was installed on each of the nodes, along with additional MPI libraries. Each node is booted via TFTP, and receives a read-only or read/write OS image. For provisioning nodes, Perceus software is used. The following IaaS provisioning system packages have been installed: OpenNebula, OpenStack and Apache Tashi.

**Configuration of working nodes (10 nodes):**
- Motherboard Gigabyte H67MA-02H-B3,
- RAM data AD3U1333B2G9-2 4GB (2GB×2) DDR3 1333 CL9,
- CPU INTEL Core i5 2400S 2.5GHz 6MB BOX LGA1155 (65W),
- disk WD WD20EARS 2TB 2.5GHz 6MB BOX LGA1155 (65W),
- disk WD WD20EARS 2TB 7.2/64M/S300 GreenPower,
- power supplier INTER-TECH IT-SL500 AC 115/230V, 50/60Hz, DC 3.3/5/±12V, 500W, Retail, Passive PFC, 1x120
Configuration of head node:
- Motherboard Gigabyte H67MA-02H-B3,
- RAM G.Skill F3-10666CL9Q-16GBXL Quad Kit 16GB (4x4GB) PC10600 DDR3,
- CPU INTEL Core i5 2400S 2.5GHz 6MB BOX LGA1155 (65W),
- 2 x disk WD WD20EARS 2TB 7.2/64M/S300 GreenPower
- 2 x network card 10/100/1000 UTP PCIe x1,
- power supplier INTER-TECH IT-SL500 AC 115/230V, 50/60Hz, DC 3.3/5/±12V, 500W, Retail, Passive PFC, 1x120

Switch:
- Cisco 24 Port 1 Gbps Switch (SGE2000-G5),

Remote power controller:
- Trendnet 8-Outlet Remote Power Controller TK-RP08
7.3 Internal Testbed

In the Intel Ireland Innovation Open Lab an internal testbed of 4 workstations (HP Compaq dc7800’s each with an Intel® Core™2 Duo E6750 CPU running at 2.66 GHz) have been deployed. This multiple computer setup allowed Intel to investigate virtualization and live migration technologies, as well as develop and test the evolving SLA@SOI framework, Ad-hoc demonstrator and ORC images.

Conveniently, these machines command a significant presence in the Showcase area of the Innovation Centre, and so are regularly introduced and demonstrated to delegations from academia, government and industry that are visiting the Intel Ireland facility. In early 2011 the Intel SLA@SOI team was delighted to showcase SLA@SOI to the CEO of Intel, Paul Otellini.

Figure 27: Intel’s internal testbed, hosting developers, VMs and Intel’s CEO Paul Otellini.
8 Conclusions

8.1 Contributions and Achievements

The SLA@SOI A4 Infrastructure Management work package has delivered a number of significant contributions and achievements.

The successful implementation of an SLA-enabled infrastructure layer is perhaps the most substantial contribution. It includes the development of significant infrastructure specific components:

- Infrastructure SLA Manager including advanced Infrastructure Planning and Optimisation Component and Infrastructure Provisioning and Adjustment Component.
- Infrastructure Service Manager on top of both Apache Tashi and OpenNebula, with software contributed to both open-source communities.
- Infrastructure Monitoring System including the ability to be dynamically configured, and with support for arbitrary instrumentation layers including Ganglia and Nagios.
- JClouds OCCI API, simplifying coding against OCCI-capable cloud providers and platforms for Java Developers, contributed to the open-source JClouds community.

This contribution realizes one of the key objectives of the project, the design and implementation of foundations for an adaptive SLA-aware infrastructure that allows for harmonized access to different virtualization technologies.

The co-chairing of the OGF’s Open Cloud Computing Interface is perhaps the most high-profile contribution, with OCCI V1.1 officially released in April 2011 and attracting significant interest in DMTF and ISO, key international organisations for the development of cloud standards. This contribution squarely addresses the project objective to develop standardized interfaces for adaptive infrastructures which allow for harmonized access to different virtualization technologies.

8.2 Lessons learned

Some interesting lessons were learned or reinforced through the activities in the Infrastructure Management work package. These include:

- The importance of regular, repeatable, automated integration in large-scale, distributed software development projects. The relatively complex technical environment adopted by SLA@SOI meant that the automated integrated tests were an invaluable aid to ensuring that new or updated code did not break anything else in the stack.
- The importance of ongoing, external collaboration when dealing with open-standards. Although standards can be notoriously slow to develop and publish, organisations such as the Open Grid Forum provide a relatively low-overhead forum in which to develop at least initial versions of standards which might then be adopted by more industry-wide organisations.
• The importance of developer-friendly tools. Inviting developers to embrace a world of machine-readable SLAs is being greatly facilitated by the development of an SLA editor.

• It is much more practical to select and contribute to appropriate open-source communities that already exist, than to attempt to start a new community from the ground up. Furthermore, for consortia that cease to exist at the end of a project, it may be more prudent to set-up and contribute to open-source projects that are component specific rather than consortia specific. The overhead of initializing multiple open-source projects needs to be balanced against the relative ease of maintaining a more focused open-source project after the consortium disbands.

8.3 Outlook

In the area of infrastructure management some interesting opportunities and challenges still exist and could be pursued in follow-on work. These include

• The adoption of the Year 3 SLA Model with the enhancements that that brings, including removing the need for coding changes to SLA Templates and models

• The SLA-enabling of OpenStack, a cloud provisioning platform receiving significant attention and support in industry at the time of writing

• The SLA-enabling of future cloud platforms including Contrail’s federated platform, and the core platform of the Future-Internet Public-Private-Partnership, FI-WARE.

• On the standards front, there is certainly interest in building on the ground-breaking progress made by OCCI, enhancing it with additional monitoring and SLA-enabling support, and integrating it into more industry-wide standards organisations and efforts such as DMTF and ISO.
9 References

[3] SLA@SOI Annex I - Description of Work
[34] Ganglia, available at http://ganglia.sourceforge.net/
[38] AMQP, available at http://www.amqp.org
[40] JClouds, available at www.jclouds.org


Appendix A: Glossary

The following list shows the most important entries of the SLA@SOI glossary. Note that terms that are specific for the current document and not part of the overall project wide glossary are marked with an asterix *.

Agreement Initiator
An agreement initiator is a party to a service level agreement. The initiator creates and manages an agreement on the availability of a service on behalf of either the service customer or service provider, depending on the domain-specific signalling requirements.

Agreement Offer
An offer is the description of the agreement relationship that is sent from agreement initiator to agreement responder during agreement creation, indicating the relationship which the initiator would like to form.

Agreement Responder
The agreement responder is a party to a service level agreement. The responder implements and exposes an agreement on behalf of either the service provider or service customer, depending on the domain-specific signalling requirements.

Agreement Template
An agreement template is an XML document used by the agreement responder to advertise the types of offers it is willing to accept.

Agreement Term
Agreement terms define the content of a service level agreement.

Business Service
A business service is exposed/invoked via at least some non IT elements.

Business Manager
A specialization of service provider: person that defines the SLATs of products and joins available services in a product.

External Service
External services are exposed across the boundaries of an organization, i.e. across at least two administrative domains.

Framework Administrator
A specialization of service provider: person that configures/adapts the SLA@SOI framework for a specific application.

Guarantee Term
Guarantee terms define the assurance on service quality associated with the service described by the service definition terms. They refer to the service description that is the subject of the agreement and define service level objectives, qualifying conditions and business value expressing the importance of the service level objectives.

Hybrid Service
A hybrid service is a set or bundle of other services where all these services are exposed to the customer but have different service interface types (e.g. an IT service and a business service).

Infrastructure Manager
A specialization of infrastructure provider: person/system that is interested to measure and control infrastructure properties.

Infrastructure Provider
A specific kind of service provider that focuses on the provisioning of infrastructure services.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infrastructure Service</td>
<td>An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.</td>
</tr>
<tr>
<td>Internal Service</td>
<td>Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.</td>
</tr>
<tr>
<td>IT Service</td>
<td>An IT service is exposed/invoked by means of information technology. Specific classes of IT services may be software services, infrastructure services or media services.</td>
</tr>
<tr>
<td>Offered Service</td>
<td>An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.</td>
</tr>
<tr>
<td>Operation Level Agreements</td>
<td>A specification of the conditions under which an internal service or a component is to be used by its “customer”.</td>
</tr>
<tr>
<td>Service</td>
<td>A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks.</td>
</tr>
<tr>
<td>Service Concreteness</td>
<td>The stage a service reaches over time from a fully abstract type to actually instantiated.</td>
</tr>
<tr>
<td>Service Consumer</td>
<td>Person(s) who actually consume/use the provided services. Typically they belong to the service customer.</td>
</tr>
<tr>
<td>Service Customer</td>
<td>Someone (person or group) who orders/buys services and defines and agrees the service level targets.</td>
</tr>
<tr>
<td>Service Description Term</td>
<td>Service Description Terms describe the functionality that will be delivered under the service level agreement. The agreement description may include also other non-functional items referring to the service description terms.</td>
</tr>
<tr>
<td>Service Exposure</td>
<td>Services can be exposed either internally (within the same administrative domain) or externally.</td>
</tr>
<tr>
<td>Service Implementation</td>
<td>A service implementation is a possible concrete realization of a given service type.</td>
</tr>
<tr>
<td>Service Instance</td>
<td>A concrete realization of an offered service which is ready for consumption by service users. It relies on the instantiations of all the resources required for a given service implementation.</td>
</tr>
<tr>
<td>Service Interface Type</td>
<td>Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface.</td>
</tr>
<tr>
<td>Service Level Consequence</td>
<td>An action that takes place in the event that a service level objective is not met.</td>
</tr>
<tr>
<td>Service Level Agreement</td>
<td>An agreement defines a dynamically-established and dynamically managed relationship between parties. The object of this relationship is the delivery of a service by one of the parties within the context of the agreement. The management of this delivery is achieved by agreeing on the respective roles, rights and obligations of the parties. The agreement may specify not only functional properties for identification or creation of the service, but also non-functional properties of the service such as performance or</td>
</tr>
</tbody>
</table>
availability. Entities can dynamically establish and manage agreements via Web service interfaces.

**Service Level Objective**  
Service Level Objective represents the quality of service aspect of the *agreement*. Syntactically, it is an assertion over the agreement terms of the agreement as well as such qualities as date and time.

**Service Provider**  
An organization supplying services to one or more internal customers or external customers.

**SLA Manager**  
A specialization of *service provider*: person/system that is responsible for managing SLATs and SLA relationships.

**Software Designer**  
A specialization of *software provider*: person that designs/develops the architecture and components of a specific SLA based application.

**Software Manager**  
A specialization of *service provider*: person that defines software-based services, takes care of their management and supports the SLA manager in creating appropriate SLA templates.

**Software Provider**  
An organization producing *software components* which might be used by a *service provider* to assemble actual services.

**Software Service**  
A software service is a specific *IT service* which is exposed/invoked by means of software entities such as Web services, user interfaces, or software-based business processes.

**Software Component**  
Software components are the entities produced at design-time by a *software provider*.

**Service Type**  
A service type (or abstract service) specifies the external interface of a service possibly including non-functional aspects. It does not specify any means (components, resources) which are needed for the actual provisioning of that service.
Appendix B: Abbreviations

AOP  Aspect Oriented Programming
BM   Business Manager
B-SLAM  Business SLA Manager
EMF  Eclipse Modelling Framework
ERP  Enterprise Resource Planning
IE   Interaction Event
FCR  Finite capacity regions
G-SLAM  Generic SLA Manager
I-SLAM  Infrastructure SLA Manager
I-SM  Infrastructure Service Manager
IoC  Inversion of Control
KPI  Key Performance Indicator
LLMS  Low Level Monitoring System
LQN  Layered Queuing Networks
MA  Manageability Agent
MRE  Monitoring Result Event
MVC  Model View Controller
NFP  Non-functional property
ORC  Open Reference Case
OVF  Open Virtualization Format
QoS  Quality of Service
QPN  Queueing Petri Nets
PAC  Provisioning and Adjustment Component
POC  Planning and Optimization Component
POJO  Plain Old Java Objects
SaaS  Software as a Service
SE  Service Evaluation
SLA  Service Level Agreement
SLAM  SLA Manager
SLAT  Service Level Agreement Template
SM  Service Manager
SME  Small and Medium-sized Enterprise
SOA  Service Oriented Architecture
SW-SLAM  Software SLA Manager
SW-SM  Software Service Manager
TCO  Total Cost of Ownership
TOGAF  The Open Group Architecture Framework
Appendix C: Multiple VM SLA Model

```json
sla{
  agreedAt = Thu May 30 09:00:00 BST 2002
  effectiveFrom = Thu May 30 09:00:00 BST 2002
  effectiveUntil = Thu May 30 09:00:00 BST 2002
  templateId = 42
  uuid = TwoVMTypeTemplate
  sla_model_version = sla_at_soi_sla_model_v1.0

/* ---- PARTY DESCRIPTIONS ---------------------------------------- */

// G-SLAM_epr = 129.217.211.139:7070
party{
  id = ID-OF-PROVIDER-PARTY-GOES-HERE
  role = provider
}

// G-SLAM_epr = 129.217.211.140:7070
party{
  id = ID-OF-PROVIDER-PARTY-GOES-HERE
  role = customer
}

/* ---- INTERFACE DECLARATIONS------------------------------------ */

interface_declr{
  id = VM_Access_Point
  provider_ref = http://www.slaatsoi.org/slamodel#provider
  interface_resource_type{
    name = VirtualMachine
  }
}
```
/* ---- VARIABLE DECLARATIONS------------------------------------------*/

/* ---- AGREEMENT TERMS----------------------------------------------------*/

agreement_term{
   id = OverAllAvailability
   guaranteed_state{
      id = AvailabilityState
      availability(VM_Access_Point) > "95" percentage
   }
}

agreement_term{
   id = autogen
   VM_0 is subset_of(VM_Access_Point)
   START_TIME is "effectiveFrom" = "2002-05-30T09:00:00" xsd:dateTime
   END_TIME is "effectiveUntil" = "2002-05-30T09:00:00" xsd:dateTime
   VM_QUANTITY_VAR is = ("10" xsd:integer)
   VM_ISOLATION_VAR is = ("false" xsd:boolean)
   VM_LOCATION_VAR is = ("IE" xsd:string)
   VM_CORES_VAR is = ("8" xsd:integer)
   VM_CPU_SPEED_VAR is = ("2.6" GHz)
   VM_MEMORY_SIZE_VAR is = ("1024" Mb)
   VM_PERSISTENCE_VAR is = ("true" xsd:boolean)
   VM_IMAGE_VAR is = ("ubuntu-9.10-32" xsd:anyURI)
   guaranteed_state{
      id = VM_COUNT
      count(VM_0) = VM_QUANTITY_VAR
   }
   guaranteed_state{
      id = VM_ISOLATION
      isolation(VM_0) = VM_ISOLATION_VAR
   }
   guaranteed_state{
      id = VM_LOCATION
      location(VM_0) = VM_LOCATION_VAR
   }
}
guaranteed_state{
  id = VM_CORES
  vm_cores(VM_0) = VM_CORES_VAR
}
guaranteed_state{
  id = VM_CPU_SPEED
  cpu_speed(VM_0) = VM_CPU_SPEED_VAR
}
guaranteed_state{
  id = VM_MEMORY_SIZE
  memory(VM_0) = VM_MEMORY_SIZE_VAR
}
guaranteed_state{
  id = VM_PERSISTENCE
  persistence(VM_0) = VM_PERSISTENCE_VAR
}
guaranteed_state{
  id = VM_IMAGE
  vm_image(VM_0) = VM_IMAGE_VAR
}
}

agreement_term{
  id = autogen
  VM_1 is subset_of(VM_Access_Point)
  START_TIME is "effectiveFrom" = "2002-05-30T09:00:00" xsd:dateTime
  END_TIME is "effectiveUntil" = "2002-05-30T09:00:00" xsd:dateTime
  VM_QUANTITY_VAR is =("8" xsd:integer)
  VM_ISOLATION_VAR is =("true" xsd:boolean)
  VM_LOCATION_VAR is =("ie" xsd:string)
  VM_CORES_VAR is =("4" xsd:integer)
  VM_CPU_SPEED_VAR is =("2.6" GHz)
  VM_MEMORY_SIZE_VAR is =("512" Mb)
  VM_PERSISTENCE_VAR is =("true" xsd:boolean)
  VM_IMAGE_VAR is =("UBUNTU_9_10" xsd:anyURI)
  guaranteed_state{
    id = VM_COUNT
    count(VM_1) = VM_QUANTITY_VAR
  }
{
  
  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_ISOLATION} \\
    \text{isolation}(\text{VM}_1) &= \text{VM\_ISOLATION\_VAR} \\
    \end{align*}
  \}

  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_LOCATION} \\
    \text{location}(\text{VM}_1) &= \text{VM\_LOCATION\_VAR} \\
    \end{align*}
  \}

  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_CORES} \\
    \text{vm\_cores}(\text{VM}_1) &= \text{VM\_CORES\_VAR} \\
    \end{align*}
  \}

  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_CPU\_SPEED} \\
    \text{cpu\_speed}(\text{VM}_1) &= \text{VM\_CPU\_SPEED\_VAR} \\
    \end{align*}
  \}

  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_MEMORY\_SIZE} \\
    \text{memory}(\text{VM}_1) &= \text{VM\_MEMORY\_SIZE\_VAR} \\
    \end{align*}
  \}

  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_PERSISTENCE} \\
    \text{persistence}(\text{VM}_1) &= \text{VM\_PERSISTENCE\_VAR} \\
    \end{align*}
  \}

  \textit{guaranteed\_state}\{
    \begin{align*}
    \text{id} &= \text{VM\_IMAGE} \\
    \text{vm\_image}(\text{VM}_1) &= \text{VM\_IMAGE\_VAR} \\
    \end{align*}
  \}

  \}

  \}

  \}
Appendix D: Example RegisterService request

The following request to a Monitoring Agent is serialised in JSON format.

```json
{
  "serviceUrl": "slasoi://xlab.si/Service/PayrollSystem",
  "recipient": "InfrastructureMonitoringAgent",
  "vmList": [{
    "fqdn": "PayrollSystemvm-1.xlab.tst",
    "agreementTerms": [{
      "violationThreshold": "96",
      "guaranteedTerm": "VM_AVAILABILITY",
      "unit": "%"
    }, {
      "violationThreshold": "1",
      "guaranteedTerm": "VM_CORES",
      "unit": "%"
    }, {
      "violationThreshold": "1.5",
      "guaranteedTerm": "VM_CPU_SPEED",
      "unit": "GHz"
    }, {
      "violationThreshold": "false",
      "guaranteedTerm": "VM_DATA_ENCRYPTION",
      "unit": "%"
    }, {
      "violationThreshold": "STANDARD",
      "guaranteedTerm": "VM_DISK_THROUGHPUT",
      "unit": "%"
    }, {
      "violationThreshold": "tashi1.img",
      "guaranteedTerm": "VM_IMAGE",
      "unit": "%"
    }, {
      "violationThreshold": "STANDARD",
      "guaranteedTerm": "VM_NET_THROUGHPUT",
      "unit": "%"
    }, {
      "violationThreshold": "256",
      "guaranteedTerm": "VM_MEMORY_SIZE",
      "unit": "MB"
    }, {
      "violationThreshold": "2048",
      "guaranteedTerm": "VM_MEMORY_SPEED",
      "unit": "MHz"
    }, {
      "violationThreshold": "false",
      "guaranteedTerm": "VM_PERSISTENCE",
      "unit": "%"
    }, {
      "violationThreshold": "SI",
      "guaranteedTerm": "VM_LOCATION",
      "unit": "%"
    }
  },
  "clusterName": "tashi.xlab.tst"
  }, {
    "fqdn": "PayrollSystemvm-2.xlab.tst",
    "agreementTerms": [ ...
  },
  "clusterName": "tashi.xlab.tst"
}
}```
{"violationThreshold": "true",
"guaranteedTerm": "SERVICE_AUDITABILITY",
"unit": ""
},
{"violationThreshold": "WORKING_DAYS",
"guaranteedTerm": "SERVICE_AVAILABILITY_RESTRICTIONS",
"unit": ""
},
{"violationThreshold": "99",
"guaranteedTerm": "SERVICE_AVAILABILITY",
"unit": "%"
},
{"violationThreshold": "1",
"guaranteedTerm": "ACCEPTABLE_SERVICE_VIOLATIONS",
"unit": ""
},
{"violationThreshold": "PUBLIC",
"guaranteedTerm": "SERVICE_DATA_CLASSIFICATION",
"unit": ""
},
{"violationThreshold": "STANDARD",
"guaranteedTerm": "SERVICE_HARDWARE_REDUNDANCY_LEVEL",
"unit": ""
},
{"violationThreshold": "NOT_IMPORTANT",
"guaranteedTerm": "SERVICE_ISO_27001",
"unit": ""
},
{"violationThreshold": "24",
"guaranteedTerm": "SERVICE_MTTF",
"unit": "hours"
},
{"violationThreshold": "15",
"guaranteedTerm": "SERVICE_MTTR",
"unit": "minutes"
},
{"violationThreshold": "12",
"guaranteedTerm": "SERVICE_MTTV",
"unit": "hours"
},
{"violationThreshold": "DAY",
"guaranteedTerm": "SERVICE_REPORTING_INTERVAL",
"unit": ""
},
{"violationThreshold": "NOT_IMPORTANT",
"guaranteedTerm": "SERVICE_SAS70_COMPLIANCE",
"unit": ""
},
{"violationThreshold": "CCR_REGION_ALLOWED",
"guaranteedTerm": "SERVICE_CCR",
"unit": ""
},
{"violationThreshold": 2,
"guaranteedTerm": "VM_QUANTITY",
"unit": ""
},
"serviceName": "PayrollSystem"}