**Project no.** FP7- 216556  
**Instrument:** Integrated Project (IP)  
**Objective ICT-2007.1.2:** Service and Software Architectures, Infrastructures and Engineering

**D.A4.a SLA Aware Infrastructure Management**  
*(full lifecycle)*

**Keywords:**  
Infrastructure, Management, Service Level Agreement, Service-Oriented Infrastructure, Cloud-Computing, Virtualisation, OCCI

**Due date of deliverable:** 31st July 2010  
**Actual submission to EC date:** 10th September 2010

**Start date of project:** 1st June 2008  
**Duration:** 38 months

**Lead contractor for this deliverable:** Intel  
**Revision:** V.1.0 (8th September 2010)

---

Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th>PU</th>
<th>Public</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 216556.

**Document Status**

<table>
<thead>
<tr>
<th>Deliverable Lead</th>
<th>John Kennedy, Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviewer 1</td>
<td>Juan Lambea, TID</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>6th July 2010, 1st September 2010</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>19th July 2010</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>6th September 2010</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>6th September 2010</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>7th September 2010</td>
</tr>
</tbody>
</table>

**Contributors**

<table>
<thead>
<tr>
<th>Partner</th>
<th>Contributors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel</td>
<td>John Kennedy, Andrew Edmonds, Victor Bayon</td>
</tr>
<tr>
<td>UDO</td>
<td>Costas Kotsokalis</td>
</tr>
<tr>
<td>TID</td>
<td>Beatriz Fuentes</td>
</tr>
<tr>
<td>XLAB</td>
<td>Miha Vuk, Miha Stopar</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 2010 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](https://creativecommons.org/licenses/by/3.0/).
<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>28th June 2010</td>
<td>All WP partners</td>
<td>First draft of year 2 deliverable</td>
</tr>
<tr>
<td>0.2</td>
<td>6th July 2010</td>
<td>All WP partners</td>
<td>Updates to all sections prior to internal review</td>
</tr>
<tr>
<td>0.3</td>
<td>13th August 2010</td>
<td>John Kennedy</td>
<td>Feedback from 1st reviewer incorporated</td>
</tr>
<tr>
<td>0.4</td>
<td>6th September 2010</td>
<td>John Kennedy</td>
<td>Feedback from 2nd reviewer incorporated</td>
</tr>
<tr>
<td>1.0</td>
<td>9th September 2010</td>
<td>John Kennedy</td>
<td>Final edits 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 progress made in the Infrastructure Management work package in SLA@SOI since the beginning of the project. It builds upon the achievements in the first year of the project documented in the Year 1 deliverable [1] and thus focuses on the advancements and effort in this, the second year of the project.

Achievements include the contribution to, co-chairing, editing and releasing of OGF’s ground-breaking Open Cloud Computing Interface standard. A reference implementation of an SLA-enabled Infrastructure layer as per the evolved architecture has also been delivered. The internet-accessible testbed has been expanded, and collaborations with key fellow-travellers have both been started and reinforced.

With SLAs at its core, SLA@SOI Infrastructure Management continues to have a focus unique to the industry. As the cloud computing movement matures, this work package will continue to strengthen its engagement with the community, promoting SLA-aware infrastructures where appropriate to help realise the vision of SLAs empowering a dependable service economy.
# Table of Contents

1 Introduction ........................................................................................................... 10
  1.1 Context ............................................................................................................. 10
  1.2 Scope ............................................................................................................... 11
  1.3 Overall Process and Approach ...................................................................... 12
  1.3.1 Year 1 ........................................................................................................... 13
  1.3.2 Year 2 ........................................................................................................... 13
  1.4 About this Document ....................................................................................... 14

2 Contribution Overview ......................................................................................... 16
  2.1 Key Innovations ............................................................................................. 16
  2.2 Framework Contributions .............................................................................. 17
  2.3 Task-level Activities ..................................................................................... 17
  A4.1 Abstract Infrastructure Framework Definition ........................................... 17
  A4.2 Harmonized Interface for Heterogeneous Virtualised Infrastructure Services .................................................................................. 17
  A4.3 Infrastructure Foundations, Resource Specification and Management .... 18
  A4.4 Monitoring, Logging and Alerting Services ................................................ 18
  A4.5 Dynamic Provisioning & Re-provisioning .................................................... 18
  A4.6 Test Bed Implementation ............................................................................. 18
  A4.7 SLA Service Provisioning ............................................................................ 18

3 Architecture ....................................................................................................... 19
  3.1 Infrastructure Requirements ......................................................................... 19
    3.1.1 High Level Requirements ........................................................................ 19
    3.1.2 Infrastructure SLA Requirements ............................................................. 20
    3.1.3 Interface and Management Requirements ............................................. 20
  3.2 Monitoring Requirements ............................................................................ 21
    3.2.1 Use Case Requirements ........................................................................ 21
  3.3 State of the Art .............................................................................................. 22
    3.3.1 Infrastructure Models ............................................................................... 22
    3.3.2 Infrastructure Virtualization ..................................................................... 23
    3.3.3 Infrastructure Platforms .......................................................................... 24
    3.3.4 Infrastructure Monitoring ......................................................................... 25
    3.3.5 Infrastructure Messaging ......................................................................... 26
  3.4 Overall Architecture ..................................................................................... 26
  3.5 Infrastructure SLA Manager .......................................................................... 28
  3.6 Infrastructure Service Manager .................................................................... 30
  3.7 Infrastructure Monitoring ............................................................................. 33
  3.8 Deployment Scenarios .................................................................................... 33

4 Infrastructure SLAs .......................................................................................... 36
  4.1 Infrastructure Vocabulary .............................................................................. 36
  4.2 Example Infrastructure SLAs ......................................................................... 37
    4.2.1 ORC Gold SLA: Single VM Provisioning ............................................... 37
    4.2.2 Multiple VM (Dynamic Provisioning) ...................................................... 39

5 Infrastructure Interface ....................................................................................... 42
  5.1 Introduction ..................................................................................................... 42
  5.2 OCCI ............................................................................................................... 42
    5.2.1 History ....................................................................................................... 43
    5.2.2 OCCI Model ............................................................................................. 44
    5.2.3 OCCI Serialisation Formats ..................................................................... 57
    5.2.4 OCCI and the Web ................................................................................... 63
    5.2.5 Extending OCCI ....................................................................................... 67
    5.2.6 Open Issues with OCCI .......................................................................... 68
    5.2.7 Current Status and Future ....................................................................... 68
6 Implementation ........................................................................................................ 69
  6.1 Infrastructure SLA Manager ........................................................................... 69
  6.1.1 Infrastructure Planning and Optimization ................................................. 69
  6.1.2 Infrastructure Provisioning and Adjustment .............................................. 70
  6.2 Infrastructure Service Manager: Proxy (OCCI) ........................................... 71
  6.2.1 <<prepare_service>> and <<manage_service>> Interactions ..................... 72
  6.2.2 ISM Proxy OCCI Implementation .............................................................. 73
  6.2.3 Component Reuse: ISM Proxy Standalone and OSGI Integration .......... 76
  6.2.4 ISM Proxy Implementation Walkthrough ................................................ 77
  6.3 Infrastructure Service Manager: Allocation and Management ................. 80
  6.4 Tashi Implementation .................................................................................... 83
  6.4.1 Tashi XMPP Daemon .............................................................................. 83
  6.4.2 Management of CPU and Memory Resources ........................................ 83
  6.4.3 Tashi Reprovisioning .............................................................................. 87
  6.4.4 CPU Speed .............................................................................................. 87
  6.4.5 Automatic Hosts Registration ................................................................. 87
  6.4.6 Tashi Scheduler Abstraction .................................................................... 88
  6.5 Infrastructure Runtime Prediction Integration Implementation ............... 89
  6.5.1 Consuming Prediction Data ..................................................................... 90
  6.6 Infrastructure Monitoring ............................................................................ 92
  6.6.1 Instrumentation ...................................................................................... 92
  6.6.2 Data Collection ...................................................................................... 93
  6.6.3 Data Processing ...................................................................................... 94
  6.6.4 SLA Monitoring ..................................................................................... 95
  6.6.5 Monitoring of Available Resources ....................................................... 96

7 Deployment ........................................................................................................... 97
  7.1 External Testbed .......................................................................................... 97
  7.2 Internal Testbed .......................................................................................... 100

8 Conclusions ........................................................................................................ 101

9 References ......................................................................................................... 102

Appendix A: Glossary .......................................................................................... 105
Appendix B: Abbreviations ................................................................................ 108
Appendix C: Multiple VM SLA Model ............................................................. 110
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Conceptual Architecture of SLA@SOI</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>High level Cloud Computing Taxonomy (Intel)</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>Cloud Computing Focus Areas (Intel)</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Refined Infrastructure Management Architecture</td>
<td>27</td>
</tr>
<tr>
<td>5</td>
<td>Top level Architecture</td>
<td>28</td>
</tr>
<tr>
<td>6</td>
<td>ISLAM Architecture</td>
<td>29</td>
</tr>
<tr>
<td>7</td>
<td>Infrastructure Services Manager</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>Layers of the Monitoring System</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>High-level deployment scenarios supported by the architecture</td>
<td>34</td>
</tr>
<tr>
<td>10</td>
<td>Implementation approaches supported by the architecture</td>
<td>34</td>
</tr>
<tr>
<td>11</td>
<td>A potential deployment scenario with RESERVOIR</td>
<td>35</td>
</tr>
<tr>
<td>12</td>
<td>OCCI &amp; Architectural Positioning</td>
<td>43</td>
</tr>
<tr>
<td>13</td>
<td>OCCI Core Resource Types</td>
<td>44</td>
</tr>
<tr>
<td>14</td>
<td>SLA@SOI OCCI Link Extensions</td>
<td>47</td>
</tr>
<tr>
<td>15</td>
<td>The Core OCCI Model in UML</td>
<td>49</td>
</tr>
<tr>
<td>16</td>
<td>OCCI Infrastructure Hierarchy</td>
<td>50</td>
</tr>
<tr>
<td>17</td>
<td>OCCI Compute Action State Diagram</td>
<td>52</td>
</tr>
<tr>
<td>18</td>
<td>OCCI Storage Action State Diagram</td>
<td>54</td>
</tr>
<tr>
<td>19</td>
<td>Network Action State Diagram</td>
<td>56</td>
</tr>
<tr>
<td>20</td>
<td>The Complete OCCI Model – Core &amp; Infrastructure</td>
<td>57</td>
</tr>
<tr>
<td>21</td>
<td>An example of a XHTML/RDFa Instance Single-mode Serialisation</td>
<td>60</td>
</tr>
<tr>
<td>22</td>
<td>An example of a XHTML/RDFa Instance Plural-mode By Reference Serialisation</td>
<td>61</td>
</tr>
<tr>
<td>23</td>
<td>An example of a XHTML/RDFa Instance Plural-mode By Value Serialisation</td>
<td>62</td>
</tr>
<tr>
<td>24</td>
<td>Infrastructure POC internals</td>
<td>69</td>
</tr>
<tr>
<td>25</td>
<td>Problem solved by the ISLAM planning algorithm</td>
<td>70</td>
</tr>
<tr>
<td>26</td>
<td>PAC class diagram</td>
<td>71</td>
</tr>
<tr>
<td>27</td>
<td>Relationship between Infrastructure SLA Manager, ISM Proxy and ISM</td>
<td>72</td>
</tr>
<tr>
<td>28</td>
<td>Example Provision Request Type</td>
<td>74</td>
</tr>
<tr>
<td>29</td>
<td>Structure of the ISLAM in terms of Spring Beans</td>
<td>77</td>
</tr>
<tr>
<td>30</td>
<td>IAM Internal Architecture</td>
<td>81</td>
</tr>
<tr>
<td>31</td>
<td>Infrastructure Landscape Domain Model</td>
<td>82</td>
</tr>
<tr>
<td>32</td>
<td>IAM to Provisioning System Communication</td>
<td>83</td>
</tr>
<tr>
<td>33</td>
<td>Functions to manage Cgroups</td>
<td>85</td>
</tr>
<tr>
<td>34</td>
<td>Rendering of prediction data in XML</td>
<td>91</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Explicit Use Case Requirements ................................................................. 21
Table 2: ORC Gold SLA ......................................................................................... 37
Table 3: Agreement Term for VM_0 ........................................................................ 39
Table 4: Mandatory attributes of OCCI Resource Type Kind ........................... 45
Table 5: Mandatory attributes of OCCI Resource Type Link ............................ 46
Table 6: Mandatory attributes of OCCI Resource Type Category .................. 47
Table 7: Per-user Category Attributes ................................................................. 48
Table 8: Mandatory attributes of SLA@SOI OCCI Resource Type Task .......... 49
Table 9: Mandatory Core Category Assignments .............................................. 50
Table 10: Mandatory Attributes of OCCI Compute ........................................... 51
Table 11: Mandatory Actions of OCCI Compute ............................................... 52
Table 12: Mandatory Attributes of OCCI Storage ............................................. 53
Table 13: Mandatory Actions of OCCI Network ............................................... 53
Table 14: Mandatory Attributes of OCCI Network ............................................ 55
Table 15: Mandatory Actions of OCCI Network ............................................... 56
Table 16: Core Mandatory HTTP Verbs and OCCI Usage ............................... 65
Table 17: Supplemental HTTP Verbs and OCCI Usage .................................. 65
Table 18: Summary of External Test bed Capacities ....................................... 97
Table 19: Use Case Server Hardware Details .................................................... 98
1 Introduction

1.1 Context

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

To distil such a large scope, several key actors, components, and the information flows between them have been identified and illustrated in the conceptual architecture illustrated in Figure 1 below.

Figure 1: Conceptual Architecture of SLA@SOI
The base layer of this conceptual architecture is infrastructure, the layer of hardware (including both real and virtual) and software on which Services are deployed and executed as per their SLAs. The realisation of this layer has been assigned to a dedicated Work Package (WP) within the project, WP A4 Infrastructure Management.

1.2 Scope

Formally, the scope of the Infrastructure Management work package is defined in the SLA@SOI Description of Work [2]. It defines the objectives of this work package as being:

- 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 focuses on the SLA-aware management of infrastructural resources such as computers, networks and storage, be they real or virtual. It seeks 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). The 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 [3]. In Intel’s Cloud computing taxonomy, SLA@SOI Infrastructure management work package primarily maps to Infrastructure as a Service concerns.

Figure 2: High level Cloud Computing Taxonomy (Intel)
Some of the key issues to be addressed by Infrastructure Management include:

- 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 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 ongoing monitoring of them
- Alerts, monitoring and logging – enabling an infrastructure to cope with failure and to meet SLA requirements.

The work package also provides 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.

1.3 Overall Process and Approach

The Infrastructure management work package has been closely following the iterative approach described in the Description of Work requiring a prototype to
be developed by end of year 1, a lab demonstrator to be implemented by end of year 2, and a field demonstrator to be realised and evaluated by end of year 3.

1.3.1 Year 1

Year 1 progress in the Infrastructure Management work package followed a largely conventional software development process.

The initial phase of the work package was spent performing an initial state-of-the-art analysis in Infrastructure Management, and included the installation of and experimentation with various virtualisation technologies and provisioning platforms.

In parallel, an overall SLA@SOI architecture was developed, potential adhoc demonstrator scenarios were described, and formal explicit requirements were solicited from the various SLA@SOI Use Cases.

As the requirements, both implicit and explicit, began to emerge, the Infrastructure Management architecture began to be formed and a preliminary proof of concept prototype was developed. Of particular note, at this stage it was realised that core functionality concerning the SLA's of the infrastructure layer for the first year of the project, the responsibility of the team from the University of Dortmund, could be addressed independently of the core infrastructure provisioning layer. This decoupling allowed high-level SLA modelling, management and negotiation concerns to be progressed largely independently of the evolution of the low-level infrastructure management components.

When the overall architecture and requirements of the project stabilising, and initial prototyping and experimentation completing, the Infrastructure architecture was solidified and the Year 1 reference deployment designed and implemented.

This implementation largely followed an agile approach, with best-of-breed quality assurance tools and processes promoted to encourage rapid, continuous, integration, and incremental improvement.

Finally, throughout these phases it is worth noting the Infrastructure Management work package began to share its initial observations and findings via public blog posts on the SLA@SOI website [4], as well as via more direct channels such as position papers and collaboration events. This effort helped initiate dialog with the external cloud computing community, forming relationships that are continuing to develop.

1.3.2 Year 2

Following the implementation of the Year 1 prototype and Open Reference Case demonstrator, in Year 2 the consortium set about refining the architecture to take various learning’s onboard and the Infrastructure Management work-package contributed the infrastructure perspective to this evolution, as well as participated on the project’s Architecture Governance Board.

In parallel with this effort, the infrastructure models and heterogeneous interfaces developed in year 1 were of sufficient maturity and interest for Infrastructure Management to be invited to participate in the start-up of OGF’s Open Cloud Computing Interface (OCCI) working group [5]. Our state of the art analysis, review of existing interfaces and models in the public domain, and outputs of Year 1 SLA@SOI efforts were all contributed to this working group, and Andy Edmonds was subsequently honoured to be invited to co-chair this standards group.
Throughout the year the OCCI standard was drafted and edited, and at the time of writing its publication is imminent.

From a lower-level implementation point of view, it was clear after our Year 1 efforts that it was not appropriate or necessary for Infrastructure Management to develop and extends its own open-source hypervisor management software framework, with all the plethora of concerns that such a system would need to address, and so a decision was made to adopt the Apache Tashi [6] framework, and use it for further generic infrastructure prototype developments. This allowed the work package to focus more on Infrastructure SLA Management concerns rather than reinvent the generic concerns that all Infrastructure management frameworks need to tackle. To this end, partners have been working to enhance and extend Tashi in several key areas, including scheduling, monitoring and re-provisioning, and conversations are underway with the owners of Tashi to explore if and how these enhancements may be integrated into their source tree.

As the architecture stabilised during Year 2, our initial prototype implementation was reworked and adjusted accordingly, with new components developed to realise the new Infrastructure SLA Manager and Infrastructure Service Manager. The former included a dedicated infrastructure Planning and Optimisation Component (IPOC) and Infrastructure Provisioning and Adjustment Component (IPAC), whilst the latter included compatibility with OCCI, additional services such as prediction, and integration with Apache Tashi. The Ganglia-based [7] infrastructure monitoring framework developed in Year 1 was also enhanced and integrated into the implementation.

### 1.4 About this Document

This document describes the progress towards constructing the SLA-aware Infrastructure layer in the second 12 months of the SLA@SOI project.

Chapter 2 describes the key contributions of the work at a high level. Key Innovations, Framework Contributions 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 is explained in some detail. 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 details the infrastructure models and external interfaces. In particular, this chapter presents the OCCI interface standard which Infrastructure Management has not just contributed to but also co-chaired.

Chapter 6 describes the implementation of the Infrastructure SLA Manager and Infrastructure Service Manager as they currently exist, illustrating how the revised architecture has been adopted.

Chapter 7 focuses on the deployment of the Infrastructure Manager layer. This chapter details the internal and external testbeds which have been expanded during year 2 of the project, as well as a description of the deployment plans for the infrastructure layer in the various industrial use cases and the open reference case.
Finally, Chapter 8, Conclusions, summarises the key outcomes and learning’s from this work package at this stage of the project, and outlines some of the Future Work planned for the remainder of the project.
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, 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 [8], no current infrastructure providers provide holistic, automated SLA management. From a service consumer point of view it is not possible to advertise supported SLAs in a machine readable fashion, support automatic negotiation with a customer, and automatically inform them of violations. From a service provider point of view, a framework for automatically managing a landscape of provisioned infrastructure based on their individual SLAs is not yet available. These areas are all being addressed by SLA@SOI Infrastructure Management. Following the architecture managed by work package A1 Architecture and Integration, and with key contributions from work package A5 SLA Management and Foundations, A4 Infrastructure Management has delivered a reference Infrastructure layer. With the initial version built on top of Tashi, other OCCI compatible infrastructure layers can easily be managed, whilst Infrastructure Service Manager implementations for other third-party frameworks such as RESERVOIR’s [10] OpenNebula [11] are on the road map for Year 3.

A separate and significant innovation from this work package has been the harmonized interface to access heterogeneous infrastructure resources. Work package A4 has played a leadership role in the creation of OGF’s Open Cloud Computing Interface standard [5], with the publication of the first release of this standard imminent at the time of writing. 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 being invited to co-chair 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 [8], existing open infrastructure monitoring frameworks typically support a centralised approach to managing infrastructure. This innovation sees low level monitoring components automatically provisioned intelligently, depending on the needs of the consumers of the monitoring information. These distributed components are also managed intelligently, 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 is scalable to very large scale 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 a reference implementation of the Infrastructure SLA Manager, as well as a reference Infrastructure Service Management built on top of Apache Tashi, and a 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 (IPAC), and an Infrastructure Provisioning and Adjustment component (IPOC). 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 OCCI-compatible interface, and an Infrastructure Allocation and Management layer on top of Apache Tashi. A prediction engine has also been implemented, to allow the performance of various prediction algorithms that have been implemented to be explored.

The Low Level Monitoring System is a monitoring layer built on top of Ganglia that allows the automated configuration, deployment and management of monitoring components to meet the client’s needs. It uses an open messaging framework - currently XMPP – to avoid communications bottlenecks.

2.3 Task-level Activities

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

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 to the OGF OCCI standard documented as described in Chapter 5 of this document, Infrastructure Interfaces.

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, and have helped come up with the OCCI standard described in Chapter 5, and whose implementation in the Infrastructure Service Manager is 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 in our case 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 help it better deliver SLA-enabled infrastructure. These enhancements included an improved, and discussions are now underway with Apace Tashi as to how our current and future enhancements to Tashi may be integrated into the project codebase. The enhancements to 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 and complementing the evolved architecture. 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.

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. This infrastructure is now available to all use cases, and 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. This task has also enhanced the SLA Model to accommodate infrastructure concerns, helping to realize the SLA templates and models described in Chapter 4.
3 Architecture

This chapter describes at a high level the architecture of the Infrastructure Manager. It begins with an analysis of the requirements for infrastructure, is followed by a summary of the state-of-the-art for 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 [2], 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 [8]. These implicit requirements have been the main driver for the architecture, implementation and reference implementations to date.

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:

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

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

6. The SLA@SOI software must be open source, and its license and the license of third party dependencies be legally compatible
7. Existing or maturing standards should be used where possible, with extensions proposed to the relevant standards organisations if appropriate
8. The resulting framework must be scalable, potentially supporting tens of thousands of nodes, if not more.
9. The results 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.
10. 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:

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

Infrastructure SLAs should ultimately allow:

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

## 3.1.3 Interface and Management Requirements

Infrastructure Management must expose the following functionality to consumers:

1. Preregistering of software images for potential deployment in the future. These images include operating system, middleware, end-user services and configuration scripts
2. Negotiation of provisioned infrastructure – including the ability to reserve infrastructure for provisioning in the future
3. 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
4. Support a hierarchy of deployed infrastructure, e.g. clustering of resources and the federation of these clusters

Infrastructure Management should also

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

3.2 Monitoring Requirements

Infrastructure Monitoring in SLA@SOI must

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

Ideally, monitoring should be

7. 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 formal explicit requirements, the various SLA@SOI Use Cases have to date registered the explicit requirements listed in Table 1.

Table 1: Explicit Use Case Requirements

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>Dynamic / unforeseen re-allocations should be possible</td>
</tr>
<tr>
<td>85</td>
<td>Monitoring metrics must include memory / processor performance, response time, failure rate</td>
</tr>
<tr>
<td>86</td>
<td>Aggregate monitoring metrics must be supported</td>
</tr>
<tr>
<td>97</td>
<td>Hardware virtualization (VMware Server, Xen) must be available</td>
</tr>
<tr>
<td>105</td>
<td>It must be possible to dynamically provision and manage physical and virtual machines as well as resource bundles, data storage and software components</td>
</tr>
<tr>
<td>190</td>
<td>Customers must be able to select basic or higher compute unit.</td>
</tr>
<tr>
<td>191</td>
<td>Customer must be able to request for additional data storage (other than the basic compute unit).</td>
</tr>
<tr>
<td>194</td>
<td>Customer must be able to specify the numbers of hours to be made available for the compute unit each day, either during peak hours or off peak hours.</td>
</tr>
</tbody>
</table>
Java Management Extensions (JMX) must be supported as a management technology

Customer must be able to request for extra compute power after initial deployment, and this must be made available by giving reasonable advance notice period (e.g. 2 days notice).

In particular, infrastructure resources might need negotiation as a complete landscape (i.e. a structured layout of related infrastructure resources)

SLA Applications must support event-based or some other asynchronous communication involved

The delivery of these requirements will be tracked throughout the lifecycle of the project. For more context and details please refer to the SLA@SOI Scientific Evaluation Report [9].

3.3 State of the Art

An extensive state of the art analysis in infrastructure management was undertaken at the state of this project. This analysis has been updated as the project has matured to reflect more recent developments in this area. As SOIs can be considered synonymous with the growing Cloud Computing paradigm, 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 [8].

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) [12] issued by the Distributed Management Task Force (DMTF) [13], Open Virtual Machine Format (OVF) [14], and Open Grid Forum’s (OGF) [15] GLUE v. 2.0 [16].
**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, 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 meets all the needs of SLA@SOI. OVF was considered to be the most relevant model, and indeed SLA@SOI has the potential to propose extensions to OVF to enable it support non-functional parameters that are so core to 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.

At this stage of the project the SLA@SOI architecture has been designed to support the recommendations of this analysis. For bandwidth reasons it was decided to de-prioritise potential contributions to OVF, however with our ongoing collaboration with RESERVOIR [10] (who are also contemplating contributions to OVF) the opportunity remains open to influence where appropriate. Instead, our efforts have gone into making OCCI as useful a standard as possible, and it has been designed in such a way as to support OVF descriptors.

### 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 paravirtualization, 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 location.

In operating system virtualization, the virtualisation is done within the operating system, it providing isolated virtualised operating system environments 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 [17] is an open source API that provides a generic way to interact with different types of open source virtualization technologies (including Xen [18] and KVM [19]). It allows the complete life cycle of these VMs to be managed independent of the underlying virtualisation technology, allowing a project like SLA@SOI to focus on higher level concerns.

OpenNebula [11] 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.

At this stage of the project the SLA@SOI architecture has been designed to support arbitrary virtualisation technologies, with the OCCI interface specifically conceived to provide a generic interface into heterogeneous virtualisation providers. As part of our ongoing collaboration with RESERVOIR we have explicit plans for a proof of concept to demonstrate the SLA@SOI framework SLA-enabling OpenNebula.

### 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, Apachi 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.

Finally, after refreshing our state of the art analysis in Year 2, it is interesting to note that it remains that none of the platforms reviewed offer comprehensive SLA-awareness. Indeed several conversations have started with third party providers such as SensibleCloud [20] to help them keep abreast of our progress and look for opportunities to build on our open research.

### 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, Groundwork, MonALISA and Zabbix all have interesting aspects, from a low-level infrastructure point of view the most relevant framework continues to be Ganglia [7].

Infrastructure monitoring brings its own unique challenges – 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 of course as the potentially massive distribution and scale of modern cloud computing infrastructures. This is an environment that Ganglia has been optimised for. It 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 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.
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 [21], 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 [22], 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 [23], 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 has evolved as requirements emerged and the overall architecture of the SLA@SOI framework matured. Deliverable D.A4a [1] describes in detail the development of the architecture in the first 12 months of the project, ultimately resulting in the infrastructure architecture illustrated in Figure 4.
Based on the learnings from the implementation of the Year 1 demonstrator, and the maturing of the requirements from the use cases, the overall top level architecture was refined, with generic functionality explicitly abstracted. The resulting top level architecture is illustrated in Figure 5 and detailed in Deliverable D.A1a Framework Architecture [24].
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, and the functionality captured in the Year 1 architecture was reorganised to these components. The individual architectures of these components are described in the following sections.

### 3.5 Infrastructure SLA Manager

The **Infrastructure SLA Manager** (ISLM) is an instantiation of the **Generic SLA Manager** (GSLAM) as described in Deliverable D.A5a, tailored to infrastructure services. It corresponds to task TA4.7. The ISLM does not try to address each exotic feature of each possible infrastructure service, rather focuses on the most important features typically offered by *Infrastructure as a Service* (IaaS).

In brief, the GSLAM includes all those features necessary for the full lifecycle management of SLAs. It supports:
- 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
- The flexible definition of per-SLA monitoring frameworks, through a generic Monitoring Manager.

The GSLAM 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 are expected to be implemented by interested parties, and replace the default placeholders. The ISLAM is, ultimately, a GSLAM with custom POC & PAC, targeting the least common denominator of IaaS. Figure 6 illustrates a high-level overview of the ISLAM architecture, as an extension of a GSLAM (more information about the latter can be found in D.A5a).

Figure 6: ISLAM Architecture

As it can be seen from this Figure, the ISLAM interacts externally with:

- The Business (SLA) Manager from WP A2 (described in detail in Deliverable D.A2a), 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 (as described in the following section), which provides resource information, and controls resources in general (reservations, initialization, runtime management).

The POC is responsible for the planning and optimisation of Infrastructure SLAs. It receives requests for infrastructure, queries the Infrastructure Service Manager for potential provisioning solutions, selects and reserves the optimal one and
requests the PAC to provision the selected plan as appropriate. If local resources cannot satisfy the request (e.g. due to lack of resources or specification discrepancies), the POC can attempt to outsource to third party providers to satisfy the request.

The PAC is responsible for the provisioning and adjustment of Infrastructure SLAs. It directs the Infrastructure Service Manager to provision as per the plan supplied by the POC. It also decides on any adjustments required, to avoid potential SLA violations.

Details of the implementation of the ISLAM’s custom POC and PAC are provided in Chapter 6 as well as in Deliverable D.A5a (Chapter 7, “Planning” and Chapter 9, “Provisioning and Adjustment”).

### 3.6 Infrastructure Service Manager

The InfrastructureServiceManager (ISM) is responsible for the creation, lifecycle management, internal optimisation and manipulation of infrastructure resources. These infrastructure resources are broadly group into the functional areas of:

- **Compute** - an entity that performs computation
- **Network** - an entity that links together 2 or more resources
- **Storage** - an entity that allows for state to be persisted

The ISM communicates with the actual provisioning system(s) that can provision such resources and is not aware of SLA concerns. The ISM exposes its functionality to clients through an interface and an associated data model and both abstract the low-level details of the provisioning system supported. This interface and model are now standardised, implementing the Open Cloud Computing Interface (OCCI) [5] as described in Chapter 5. The ISM is a composition of a number of components as illustrated in Figure 7.
Figure 7: Infrastructure Services Manager

The main architectural components of the Infrastructure Services Manager are:

- **InfrastructureAllocationAndManagement (IAM)** providing the top-level abstract interface into the ISM and is implemented using the OCCI standard. It processes requests from the InfrastructureSLAManager, making calls to subcomponents as appropriate. This includes querying the InfrastructureLandscape, HistoricalDB and Policies. Resource reservations are to be made, released, and committed to. Provisioned resources can be queried, updated and disposed of. If re-provisioning is requested, the IAM component can 'spot-the-difference' between the new and original provisioning request, and make the appropriate calls to update to the new request. This may simply require adjusting an existing resource in real-time or the reallocation of a new resource. The IAM component is responsible for the overall management and dispatching of provisioning requests and
resource manipulation requests to the target provisioning system asynchronously through the <<publish_event>> interaction. Resultant actions are then propagated back (via <<publish_event>>>) to the IAM through the asynchronous interaction, <<subscribe_to_event>>. When a provisioning has been successful, it is recorded in the InfrastructureLandscape. The provisioning identifier is recorded against the provisioning system's resource identifiers along with the provisioning system type (for later extensibility).

- **InfrastructureLandscape (IL)** contains the state of all physical resources, virtual resources and provisioning requests and represents the state through the OCCI data model associated with the top-level OCCI interface as well as the provisioning-system-specific data model. The state may not reflect the actual current status precisely: it relies on timely updates from the provisioning system and its tracked resources. The IL has scope to includes the most recent predictions for future resource status, based on analysis by the prediction component.

- **InfrastructureReporting (IR)** provides infrastructure related data to the use case GUI enabling presentation of infrastructure status and its history. It essentially provides an API to access the InfrastructureLandscape and all historical monitoring data stored in the HistoricalDB.

- **Policies** includes details of all policies relevant to the Infrastructure being managed. These policies are taken into account when provisioning, optimisation and adjustment decisions are being taken.

- The **OptimisationAgent** component reviews policies, the current status of provisioned results and the predicted status as stored in the InfrastructureLandscape, and passes any possible optimizations to the InfrastructureAllocationAndManagement component.

- **PredictionAgent** employs various algorithms to predict the value for parameters of interest based on the historical data. Predictions are stored in the InfrastructureLandscape.

- **MonitoredEventChannel** is a shared communication channel to which components can publish and / or subscribe. It is used to exchange monitoring observations, notifications and interesting events. It comprise of several subchannels each designated for a specific purpose.

- **ProvisioningSystem** is the infrastructure system being managed. It typically comprises of a set of hypervisors together with the hypervisor manager that provisions the resources and maintains their lifecycle. Examples of provisioning systems include TASHI which manages a set of KVM and/or Xen hypervisors that run on a clustered set of physical machines, or OpenNebula that manages a hybrid set of cloud computing resources.
3.7 Infrastructure Monitoring

The scope and position of Monitoring in the general architecture is described in Deliverable D.A1a [24]. Its internal structure is shown in Figure 8.

![Figure 8: Layers of the Monitoring System](image)

Infrastructure monitoring spans all of these layers but focuses on infrastructure resources. All blue components are generic and can also be used in other domains.

Besides SLA monitoring related to the services provisioned, infrastructure monitoring also addresses the monitoring of available resources. This information is used in infrastructure provisioning and adjustment.

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.

High level deployment scenarios are illustrated in Figure 9. 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.
The Infrastructure Management workpackage Year 2 prototype implements scenario (a).

The architecture is also flexible enough to allow various implementation approaches to be adopted as illustrated in Figure 10. 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 Year 2 Infrastructure Management prototype implements approach (a). The ISM and its target provisioning system adopt an event-based architecture and as
a result are cleanly separated by the use of messaging middleware. This brings great flexibility to the system. As the ISM is loosely coupled to the provisioning system this enables many different deployment configurations. As the message formats that the ISM sends to the provisioning system are well understood, all that is required to support a different provisioning system is to write an associated message sink that has full access to management facilities of the provisioning system. In our reference implementation, the ISM in SLA@SOI uses Apache Tashi as a provisioning system. Our messaging middleware implements the open standard XMPP and so an XMPP message sink is required that can receive the XMPP messages from the ISM and duly process them. Once processing is complete a responding message is sent back to the ISM. The ISM then, on receipt of the provisioning system result, updates the IL state and sends notification of the result to the original requestor.

The reference implementation also implements the OCCI standard at the ISM interface level, thus facilitating the SLA-enabling of other OCCI-compliant provisioning systems. This is one approach that could be taken to SLA-enable RESERVOIR’s provisioning system for example, and is being actively explored as part of our collaboration with that project, as presented in a joint technical report authored by SLA@SOI and RESERVOIR partners [26]. Indeed, OCCI opens up the possibility of an interesting brokerage scenario for the Infrastructure SLA Manager should it be aware of multiple OCCI-compliant infrastructure services. This topology is illustrated in Figure 11.

![Diagram](image-url)

**Figure 11: A potential deployment scenario with RESERVOIR**
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 [25].

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 uses the BNF form of SLAs. XML versions are available in the appendices.

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 2: ORC Gold SLA

```plaintext
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 = SLASOIPrivate
    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 = ""
      mtr(VM_Access_Point) < "3" hrs
    }
    guaranteed_state{
  }
```
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 VMs configurations from a pre-determined range of VMs and the amount (count) of VMs that will be provisioned.

The Agreement Term defined in Table 3 defines an agreement terms for a VM type VM_0.

<table>
<thead>
<tr>
<th>agreement_term</th>
</tr>
</thead>
<tbody>
<tr>
<td>id = autogen</td>
</tr>
<tr>
<td>VM_0 is subset_of(VM_Access_Point)</td>
</tr>
<tr>
<td>START_TIME is &quot;effectiveFrom&quot; = &quot;2002-05-30T09:00:00&quot; xsd:dateTime</td>
</tr>
<tr>
<td>END_TIME is &quot;effectiveUntil&quot; = &quot;2002-05-30T09:00:00&quot; xsd:dateTime</td>
</tr>
<tr>
<td>VM_QUANTITY_VAR is =(&quot;10&quot; xsd:integer)</td>
</tr>
<tr>
<td>VM_ISOLATION_VAR is =(&quot;false&quot; xsd:boolean)</td>
</tr>
<tr>
<td>VM_LOCATION_VAR is =(&quot;IE&quot; xsd:string)</td>
</tr>
<tr>
<td>VM_CORES_VAR is =(&quot;8&quot; xsd:integer)</td>
</tr>
<tr>
<td>VM_CPU_SPEED_VAR is =(&quot;2.6&quot; GHz)</td>
</tr>
<tr>
<td>VM_MEMORY_SIZE_VAR is =(&quot;1024&quot; Mb)</td>
</tr>
<tr>
<td>VM_PERSISTENCE_VAR is =(&quot;true&quot; xsd:boolean)</td>
</tr>
<tr>
<td>VM_IMAGE_VAR is =(&quot;ubuntu-9.10-32&quot; xsd:anyURI)</td>
</tr>
<tr>
<td>guaranteed_state</td>
</tr>
<tr>
<td>id = VM_COUNT</td>
</tr>
<tr>
<td>count(VM_0) = VM_QUANTITY_VAR</td>
</tr>
</tbody>
</table>
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 ISM. To this end, the correct infrastructure resource types and features were identified and exposed at an abstract level. This allowed for a harmonised, implementation technology neutral interface into infrastructure to be defined. More dynamic with high temporal variations and characteristics, for example I/O read and write speeds, network latency are more suitably dealt with the work related to monitoring and the related historical database. Infrastructure Managements’ abstract interface and model is embodied in OCCI.

5.2 OCCI

The Open Cloud Computing Interface, OCCI, will according to the working group charter “deliver an API specification for remote management of cloud computing infrastructure, allowing for the development of interoperable tools for common tasks including deployment, autonomic scaling and monitoring. The scope of the specification will be all high level functionality required for the life-cycle management of virtual machines (or workloads) running on virtualization technologies (or containers) supporting service elasticity” Open Cloud Computing Interface, available at http://www.occi-wg.org. The current specification itself is currently split into three complimentary documents:

1. Core [28] – this defines the OCCI model, common resource types and shared attributes.
2. Infrastructure [29] – this defines the infrastructure domain resource types, the required attributes for each and the actions that can be taken on each.
3. HTTP Header Rendering [30] – this defines how the OCCI model can be communicated and thus serialised using HTTP headers.

OCCI is a boundary protocol/API that acts as a service front-end to a provider’s internal infrastructure management framework. The following diagram shows OCCI’s place in a provider’s architecture.
5.2.1 History

In an effort to make this work package’s contributions as wide-ranging and to have the most impact, the early outputs of this work were contributed to a forming standard working group within the Open Grid Forum (OGF) [15]. This working group assumed the name of the Open Cloud Computing Interface and was initially lead by co-chairs Thijs Metsch (then SUN Microsystems), Alexis Richardson (RabbitMQ), Ignacio Martin Llorente (UCM, Madrid) with Sam Johnston (Australian Online Solutions) as secretary. This particular working group was recognised by this work package as the only developing open standard in cloud computing at the time and so SLA@SOI began engaging with the team. With the departure of Ignacio Martin Llorente, Andy Edmonds a member of the consortium was invited to take the position of co-chair. The working group soon had a membership of over 200 members and included numerous industry and academic parties. Some of these members include:

- **Industry**: Rackspace, Oracle, GoGrid, Cisco, Flexiscale, ElasticHosts, CloudCentral.
- **Academia & Research**: University of Madrid, SLA@SOI, RESERVOIR.
- **Service Providers**: RabbitMQ, CohesiveFT.

The reasons driving the development of OCCI were identified as:

- **Interoperability** - Allow for different IaaS providers work together without:
  - Data schema format translation
  - Facade/proxying between APIs
  - Understanding & dependency on multiple APIs
• **Portability** - No technical/vendor lock-in and allow for services to move between IaaS providers (Inter-Cloud)
  - Easily allow clients switch between providers based on business objectives (e.g. cost)
  - Minimal technical cost
  - Enables and fosters competition

• **Integration**: Implementations of the specification can be implemented with those with the latest or legacy infrastructure

The OCCI working group’s first goal, after setting out its charter, was to assemble a number of use cases in the cloud computing domain. SLA@SOI contributed to this effort and those results can be seen in [31]. Once the use case document was complete all attention was turned to creating the specification that would form the backbone of the standard. Again, SLA@SOI contributed heavily here with work that had been done prior to the inception of OCCI. This influencing work included work that was part of the Year 1 Infrastructure Management deliverable [1]. What was of particular importance was to settle upon the OCCI model in order to frame and focus the specification.

### 5.2.2 OCCI Model

The core of the OCCI model is a simple one. The central abstract resource type is Kind and contains a number of common attributes that domain-specific resource types inherit. These resource types are complemented by another two types, Link and Category. These are central to understanding the OCCI model and all core resource types must be implemented by those wishing to be OCCI compliant.

#### Core

This section of the specification defines the foundations of the OCCI model, including common resource types, linkages and attributes. OCCI resource types can be modelled so as to show the inherited relationships. In fact, the OCCI model is more suited and better represented by the style of modelling that is the basis for the Semantic Web and linked data however that does not preclude other modeling techniques such as UML.

![Figure 13: OCCI Core Resource Types](image)

**Client-side Attribute Mutability**

In the following sections, resource type attributes are noted to be either client mutable or client immutable – see the Client Mutability column. If an attribute is noted to be mutable this must be interpreted as the client can create (via HTTP POST) a resource instance that is parameterised by the attribute. Likewise, if an attribute is mutable, a client can update that resource instance’s mutable attribute value and the server side must support this.
If a client attempts to send an immutable attribute as part of a request, the server must indicate the error by returning a HTTP 400, bad request, code.

If an attribute is marked as immutable, it indicates that the server side implementation must manage these - they are not modifiable by clients under any circumstance.

**Kind**

The resource type Kind is the base abstract resource type and is common to all inheriting domain-specific resource types. It enforces domain specific resource types to inherit a number of common attributes through Kind. The attributes that the Kind resource type must implement to be compliant are shown below. The most important attribute is id, which must be a unique URI, with that URI being unique within the service provider's namespace.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>URI</td>
<td>1</td>
<td>Immutable</td>
<td>Immutable, unique identifier of the instance</td>
<td>Compliant</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>0...1</td>
<td>Mutable</td>
<td>Display name for the instance</td>
<td>Compliant</td>
</tr>
<tr>
<td>summary</td>
<td>String</td>
<td>0...1</td>
<td>Mutable</td>
<td>General summary description of the instance</td>
<td>Compliant</td>
</tr>
<tr>
<td>categories</td>
<td>Category[]</td>
<td>1</td>
<td>Mutable</td>
<td>This is a set of associated categories with Kind. There must be at minimum one Category associated (See Table 9)</td>
<td>Compliant</td>
</tr>
<tr>
<td>tasks</td>
<td>Task[]</td>
<td>0...1</td>
<td>Immutable</td>
<td>This is a set of associated tasks with Kind.</td>
<td>Compliant</td>
</tr>
<tr>
<td>links</td>
<td>Link[]</td>
<td>0...1</td>
<td>Mutable</td>
<td>This is a set of associated Links with Kind</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

Table 4: Mandatory attributes of OCCI Resource Type Kind

**Link & Action**

With common attributes established, the possible linkages between resource types must be defined. This is done using the Link resource type and for compliance it must be implemented. The Link resource type is heavily influenced by other standardisation work [33] presently underway and so aims to reuse

\footnote{2 See Client-side Attribute Mutability.}
attributes defined by those efforts. The attributes that the Link resource type **must** implement to be compliant are detailed below.

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>class</td>
<td>Enumeration {link, action}</td>
<td>0...1</td>
<td>Mutable</td>
<td>This denotes the type of Link</td>
<td>Compliant</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>0...1</td>
<td>Mutable</td>
<td>Display name for the Link</td>
<td>Compliant</td>
</tr>
<tr>
<td>rel</td>
<td>Enumeration {first, next, previous, last, help, icon, search, self}</td>
<td>0...1</td>
<td>Mutable</td>
<td>The type of Link</td>
<td>Compliant</td>
</tr>
<tr>
<td>target</td>
<td>Kind</td>
<td>1</td>
<td>Mutable</td>
<td>The Kind to which the Link points to.</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

Table 5: Mandatory attributes of OCCI Resource Type Link

The Link resource type is also used for performing **Actions** upon a target resource type. In general, these actions modify state, such as rebooting or pausing a virtual machine. As Links **must** be implemented, so are Actions by association. Information related to the Action is relayed through particular attributes of the Link resource type. The valid actionable states that a particular domain-specific resource must support are discussed in the following section on infrastructure.

For SLA@SOI purposes, the Link has been extended depending on what resource type is being linked to another. This was done as the current OCCI specification of Link cannot accommodate certain infrastructure-related attributes that are best associated with a domain-specific Link sub-class, rather than overloading the Link resource type with additional domain-specific attributes (e.g. device or mount point).
As these specialised resource types fall outside of the specification, these will be offered to the OCCI working group as suggested changes.

**Category**

With a mechanism to link related resource types in place, it remains for a means to categorise and assign type information to each particular domain-specific resource type. This is done through the use of a core resource type **Category** (See Category). Below shows the attributes that the Category resource type **must** implement to be compliant. A category can be uniquely identified by the concatenation of scheme and term (e.g. http://user.com/category#term)

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>String</td>
<td>1</td>
<td>Immutable</td>
<td>Target definition within the scheme</td>
<td>Compliant</td>
</tr>
<tr>
<td>scheme</td>
<td>URI</td>
<td>1</td>
<td>Immutable</td>
<td>A resource that defines the model of the referred term</td>
<td>Compliant</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>0...1</td>
<td>Immutable</td>
<td></td>
<td>Compliant</td>
</tr>
<tr>
<td>related</td>
<td>Category[]</td>
<td>0...1</td>
<td>Immutable</td>
<td>A set of related categories</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

Table 6: Mandatory attributes of OCCI Resource Type Category

Categories must be queried through the query interface (See Discovery and Query). The Category resource type is a very flexible concept. Not only can it be used to supply type information to resource types but they can also be used to organise resource types in a fashion not unlike folksonomies [34]. Multiple categories **must** be assignable to resource types.

Categories are not emphasised within the current OCCI specification, but it is the intention of SLA@SOI's implementation to provide a working concept of not only...
folksonomic (flat) categorisation but also that of taxonomic (hierarchical) categorisation.

**Category & Collections**

The Category type groups related resource types. Therefore it can be argued that such a group is a collection. These collections are operated on by Link Actions, using the relevant Link (See Link & Action) relation "rel" value. Collections in OCCI operate much like how Atom [37] collections work. To assign a resource type instance as part of a collection is merely a matter of associating that instance with a Category.

**Provider and User Categories**

Other than the required categories (See Table 4), providers can offer their own custom categories. These custom categories must use a scheme namespace other than those used by OCCI. Typically, this namespace is one managed by the provider (e.g. http://provider.com/schemes/). The client mutability of these provider specific categories will remain as specified above (See Table 6).

Providers may also offer per-user defined categories to allow for per-user organisation of their instances. The namespace for per-user categories may be managed by the provider (e.g. scheme="http://provider.com/schemes/users/userA"), however a user can manage the namespace themself (e.g. scheme=http://user.name/schemes/myScheme). In the case that a provider chooses to support user categories then the client mutability of the category resource type changes:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>String</td>
<td>1</td>
<td>Mutable</td>
<td>Target definition within the scheme</td>
<td>Compliant</td>
</tr>
<tr>
<td>scheme</td>
<td>URI</td>
<td>1</td>
<td>Mutable</td>
<td>A resource that defines the model of the referred term</td>
<td>Compliant</td>
</tr>
<tr>
<td>title</td>
<td>String</td>
<td>0...1</td>
<td>Mutable</td>
<td>Compliant</td>
<td></td>
</tr>
<tr>
<td>related</td>
<td>Category[]</td>
<td>0...1</td>
<td>Mutable</td>
<td>A set of related categories</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

Table 7: Per-user Category Attributes

**Task**

Task is a SLA@SOI extension to OCCI. Task is used to track the progress of operations upon other OCCI resource types. This resource types provides support for long-running operations whose owning client is notified of the result asynchronously.
Table 8: Mandatory attributes of SLA@SOI OCCI Resource Type Task

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>statusId</td>
<td>UUID</td>
<td>1</td>
<td>Immutable</td>
<td>An identifier to uniquely identify the status object</td>
<td>Compliant</td>
</tr>
<tr>
<td>percentage</td>
<td>Integer (0...100)</td>
<td>0...1</td>
<td>Immutable</td>
<td>The fraction of work done</td>
<td>Compliant</td>
</tr>
<tr>
<td>message</td>
<td>String</td>
<td>0...1</td>
<td>Immutable</td>
<td>A message describing the status</td>
<td>Compliant</td>
</tr>
<tr>
<td>state</td>
<td>Enumeration,</td>
<td>1</td>
<td>Immutable</td>
<td>The current action of the status</td>
<td>Compliant</td>
</tr>
<tr>
<td></td>
<td>{queued, running,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>completed, in_progress, ok, warning, violated}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>targetUri</td>
<td>URI</td>
<td>1</td>
<td>Immutable</td>
<td>The resource instance being operated upon</td>
<td>Compliant</td>
</tr>
<tr>
<td>notificationUri</td>
<td>URI</td>
<td>0...1</td>
<td>Immutable</td>
<td>An endpoint to call once the status item is completed</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

Task should be used when executing an Action against a particular instance to track the action’s progress, especially if clients require notification of an Action’s completion.

**Core Model UML**

Figure 15: The Core OCCI Model in UML
Infrastructure

The infrastructure domain resource types within OCCI, including their specification definitions, are:

- **Compute**: Information processing resources.
- **Network**: Interconnection resources.
- **Storage**: Information recording resources.

These infrastructure resource types inherit all the attributes of **Kind**. OCCI implementors **must** implement these. This inheritance (See Figure 16) is useful to know in the case where other resource types are required in order to extend OCCI (e.g. a load balancer or network appliance). Instances of these resource types are interacted with using RESTful [36] communications (See OCCI and the Web). Simply put, those instances, identified by a URL, can be operated upon by the standard set of HTTP verbs. This makes OCCI an ideal boundary interface between the web and the internal resource management system of infrastructure providers. The ISM is an excellent example of the loose coupling and relationship of OCCI to a provider-specific resource management system (More details in section 6.3).

![Figure 16: OCCI Infrastructure Hierarchy](image)

Every resource type instantiated that is a subclass of Kind **must** be assigned a Category (See Category) that identifies the instantiated resource type. These Categories, the core categories, **must** always remain immutable to any client.

<table>
<thead>
<tr>
<th>Kind</th>
<th>Term</th>
<th>Scheme</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>compute</td>
<td><a href="http://purl.org/occi/kind#">http://purl.org/occi/kind#</a></td>
<td>Any provider-defined string</td>
</tr>
<tr>
<td>Storage</td>
<td>storage</td>
<td><a href="http://purl.org/occi/kind#">http://purl.org/occi/kind#</a></td>
<td>Any provider-defined string</td>
</tr>
<tr>
<td>Network</td>
<td>network</td>
<td><a href="http://purl.org/occi/kind#">http://purl.org/occi/kind#</a></td>
<td>Any provider-defined string</td>
</tr>
</tbody>
</table>

Table 9: Mandatory Core Category Assignments
With the main infrastructure resource types defined, further collaborative efforts and consultation with the OCCI community allowed consensus to be reached on how each of these domain specific resource types would be described through their attributes, allowable linkages and the valid actions.

The following sections on Compute, Storage and Network detail the attributes that must be part of each resource type, along with the actions (via Links) that are valid.

**Compute**

When creating a Compute resource type, the “compute” Category must be assigned (See Table 9). The attributes that must be exposed by an instance of the Compute resource type are as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>occi.compute.</td>
<td>Enumeration</td>
<td>1</td>
<td>Mutable</td>
<td>CPU Architecture of the instance.</td>
<td>Compliant</td>
</tr>
<tr>
<td>architecture</td>
<td>, {x86, x64}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>occi.compute.</td>
<td>Integer</td>
<td>1</td>
<td>Mutable</td>
<td>Number of CPU cores assigned to the instance.</td>
<td>Compliant</td>
</tr>
<tr>
<td>cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>occi.compute.</td>
<td>String</td>
<td>1</td>
<td>Mutable</td>
<td>DNS hostname for the instance.</td>
<td>Compliant</td>
</tr>
<tr>
<td>hostname</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>occi.compute.</td>
<td>Float, 10^9</td>
<td>1</td>
<td>Mutable</td>
<td>CPU Clock frequency (speed) in gigahertz.</td>
<td>Compliant</td>
</tr>
<tr>
<td>speed</td>
<td>(giga)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>occi.compute.</td>
<td>Float, 10^9</td>
<td>1</td>
<td>Mutable</td>
<td>Maximum RAM in gigabytes allocated to the instance.</td>
<td>Compliant</td>
</tr>
<tr>
<td>memory</td>
<td>(giga)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>occi.compute.</td>
<td>Enumeration</td>
<td>1</td>
<td>Immutable</td>
<td>Current status of the instance.</td>
<td>Compliant</td>
</tr>
<tr>
<td>status</td>
<td>, {active,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>inactive,</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>suspended}</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 10: Mandatory Attributes of OCCI Compute

Actions can be performed upon instances of the Compute resource type. The set that must be supported are as follows:
<table>
<thead>
<tr>
<th>Action</th>
<th>Target State $^3$</th>
<th>Parameters</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://purl.org/occi/action#start">http://purl.org/occi/action#start</a></td>
<td>active</td>
<td>None</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#stop">http://purl.org/occi/action#stop</a></td>
<td>inactive</td>
<td>Enumeration {graceful, acpioff, poweroff}</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#restart">http://purl.org/occi/action#restart</a></td>
<td>active</td>
<td>Enumeration {graceful, warm, cold}</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#suspend">http://purl.org/occi/action#suspend</a></td>
<td>suspended</td>
<td>Enumeration {hibernate, suspend}</td>
<td>Not compliant</td>
</tr>
</tbody>
</table>

Table 11: Mandatory Actions of OCCI Compute

Figure 17 illustrates the state diagram based on the actions defined above.

![Figure 17: OCCI Compute Action State Diagram](image)

**Storage**

When creating a Storage resource type, the “storage” Category must be assigned (See Table 9). The attributes that **must** be exposed by an instance of the Storage resource type are as follows:

---

$^3$ See the attribute occi.compute.status
<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Mutability</th>
<th>Description</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>occi.storage.size</td>
<td>Float, 10^9 (giga)</td>
<td>1</td>
<td>Mutable</td>
<td>Storage size in gigabytes of the instance</td>
<td>Compliant</td>
</tr>
<tr>
<td>occi.storage.status</td>
<td>Enumeration, {online, offline, degraded}</td>
<td>1</td>
<td>Immutable</td>
<td>Current status of the instance.</td>
<td>Compliant</td>
</tr>
</tbody>
</table>

**Table 12: Mandatory Attributes of OCCI Storage**

Actions can be performed upon instances of the Storage resource type. The set that must be supported are as follows:

<table>
<thead>
<tr>
<th>Action</th>
<th>Target State⁴</th>
<th>Parameters</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://purl.org/occi/action#online">http://purl.org/occi/action#online</a></td>
<td>online</td>
<td>None</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#offline">http://purl.org/occi/action#offline</a></td>
<td>offline</td>
<td>None</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#backup">http://purl.org/occi/action#backup</a></td>
<td>none</td>
<td>None</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#snapshot">http://purl.org/occi/action#snapshot</a></td>
<td>none</td>
<td>None</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#resize">http://purl.org/occi/action#resize</a></td>
<td>none</td>
<td>Float size 10^9</td>
<td>Not compliant</td>
</tr>
</tbody>
</table>

**Table 13: Mandatory Actions of OCCI Network**

The state diagram in Figure 18 is based on the above defined actions.

⁴ See the attribute occi.storage.status
Figure 18: OCCI Storage Action State Diagram
**Network**

When creating a Network resource type, the "network" Category must be assigned (See Table 9). The attributes that **must** be exposed by an instance of the Storage resource type are as follows:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Multiplicity</th>
<th>Client Immutability</th>
<th>Description</th>
<th>SL@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td>occi.network.vlan</td>
<td>Integer, 0-4095</td>
<td>0...1</td>
<td>Mutable</td>
<td>802.1q VLAN Identifier (e.g. 4095)</td>
<td>Compliant</td>
</tr>
<tr>
<td>occi.network.label</td>
<td>Token</td>
<td>0...1</td>
<td>Mutable</td>
<td>Tag based VLANs (e.g. external-dmz)</td>
<td>Compliant</td>
</tr>
<tr>
<td>occi.network.address</td>
<td>IPv4 or IPv6 Address, CIDR notation</td>
<td>1</td>
<td>Mutable</td>
<td>Internet Protocol(IP) network address (e.g. 192.168.0.1/24, fc00::/7)</td>
<td>Compliant</td>
</tr>
<tr>
<td>occi.network.gateway</td>
<td>IPv4 or IPv6 Address, CIDR notation</td>
<td>1</td>
<td>Mutable</td>
<td>Internet Protocol (IP) network address(e.g. 192.168.0.1/24, fc00::/7)</td>
<td>Compliant</td>
</tr>
<tr>
<td>occi.network.allocation</td>
<td>Enumeration, {auto, dhcp, manual}</td>
<td>1</td>
<td>Mutable</td>
<td>Address mechanism: <strong>auto</strong> use the provider default policy</td>
<td>Compliant</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>dhcp</strong> use the dynamic host configuration protocol</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>manual</strong> use user supplied static network configuration</td>
<td></td>
</tr>
</tbody>
</table>

**Table 14: Mandatory Attributes of OCCI Network**
Actions can be performed upon instances of the Storage resource type. The set that must be supported are as follows:

<table>
<thead>
<tr>
<th>Action</th>
<th>Target State</th>
<th>Parameters</th>
<th>SLA@SOI Compliance</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://purl.org/occi/action#up">http://purl.org/occi/action#up</a></td>
<td>active</td>
<td>None</td>
<td>Not compliant</td>
</tr>
<tr>
<td><a href="http://purl.org/occi/action#down">http://purl.org/occi/action#down</a></td>
<td>inactive</td>
<td>None</td>
<td>Not compliant</td>
</tr>
</tbody>
</table>

**Table 15: Mandatory Actions of OCCI Network**

The state diagram based on the above defined actions is illustrated in Figure 19.

![Network Action State Diagram](image-url)
Complete OCCI Model – Core & Infrastructure

Figure 20: The Complete OCCI Model – Core & Infrastructure

5.2.3 OCCI Serialisation Formats

Having a model is the prerequisite before communicating service requests to a provider. Using the conceptual model, OCCI defines efficient and suitably expressive serialisation formats. The serialisation format is the content that is passed "on-the-wire" between client and service. This particular conversation within the community was highly contested and provided excellent input to directions on this topic. In the end, two main formats were selected to be supported in OCCI. Those formats are listed and described in the following sections.

HTTP Header Serialisation/Rendering

The HTTP header serialisation is a rendering of the OCCI model using HTTP key-value pairs in the header [37] of a HTTP request or response. This rendering must be supported by all OCCI implementations. This is a very simple format and can express the basic needs of the OCCI model. In general, HTTP headers are arbitrary strings, however their general format is of the key followed by a colon and space and then the value referred to by the key. Below we show an example request that creates a compute resource.

```
1 POST /compute HTTP/1.1
1 Host: example.com
1 Authorization: Basic xxxxxxxxxxxxxxxxxxx
1 User-Agent: occi-client/1.0 (linux) libcurl/7.19.4 OCCI/1.0
```
The general method of specifying the three infrastructure domain specific resource types in a HTTP headers style request is:

1. supply required standard HTTP headers
2. supply the required OCCI Category that matches the resource type being requested
3. optionally, supply customising Category headers
4. optionally, supply the OCCI Link header that will link the resource to be instantiated with an existing one
5. optionally, supply the resource type attributes to customise the instance to be created

Although rich enough to support basic use cases, it has been found that this approach is not rich enough for certain use cases especially in the case of an atomic request containing multiple resource creation requests. There are also limitations related to payload size that must be respected when using HTTP headers.

**XHTML5/RDFa Serialisation/Rendering**

This serialisation is a novel aspect of OCCI, however it is currently under development (July 2010) and scheduled to appear in the next version of OCCI. There are three aspects to this serialisation format.

**RDFa**

RDFa [40] provides a set of XHTML attributes to demarcate machine-readable semantic content and hence compliments visual display of related data in a typical standards compliant web browser. RDFa is similar to microdata [38] in that it exposes content that is structured and machine-readable, however RDFa allows for easy creation of custom vocabularies whereas to do the same with microdata requires a submission process. RDFa uses the power of the W3C’s RDF [39] standard to embedded RDF triplets as attributes, hence the 'a' in RDFa. RDF is the standard used for interoperable machine-readable linked data. It forms the foundation to the semantic web and is powering the notion of Linked Data [32] which is a 'a term used to describe a recommended best practice for exposing, sharing, and connecting pieces of data, information, and knowledge on the Semantic Web using URIs and RDF'. Using RDFa allows to demarcate structured semantic data within XHTML documents and allows such data to be easily extracted and machine-readable.

**XHTML5**

XHTML5 (Extensible Hypertext Markup Language) is an XML-based and thus well-formed syntax of HTML5. XHTML5 is required rather than HTML5, as documents
in HTML5 format do not have to be well-formed. Adopting XHTML5 ensures that documents can be readily parsed - essential for enabling machine-readability.

**RDF Ontology**

The Resource Description Framework (RDF) is a means to describe a graph-oriented model that consists of nodes, which can either be subject or object and edges, which are the predicate. The subject can be compared to a noun in a sentence, the object to the subject of a sentence and the predicate to the verb of a sentence. For example: Ms. Piggy has a snout. Here the subject is "Ms. Piggy", "has" is the predicate and "snout" is the object. The combination of these 3 elements is known as a triplet. Many of these triplets grouped together form an ontology. The ontology itself provides a way for organising information and also defines a shared vocabulary and meanings of terms with respect to other terms. Ontologies are described using various syntaxes the Resource Description Framework (RDF) is the most relevant for OCCI purposes. RDF is to ontology modelling what UML is to object-oriented modelling. Using RDF we can completely describe the OCCI model.

**OCCI & RDF**

As noted above, RDF can completely describe the OCCI model and so therefore the OCCI model can be serialised as RDF-XML. This in itself is quite useful as there are many interested people within the OCCI working group that need a format that is XML-like. Importantly, once an RDF ontology is established, instances of the OCCI model represented as RDF can be expressed as RDFa statements within a XHTML5 document. These expressions within the XHTML5 document can be verified and validated against the reference ontology. Each triplet can have its meaning extracted by referring to the ontology. The alternative to expressing RDFa within XHTML5, i.e. serialisation as RDF-XML, does not have these advantages.

**OCCI, RDFa & XHTML5**

Having established the requirement for a structured RDF ontology on which to base our RDFa expressions, it is necessary to discuss how triplets in various combinations can be expressed in XHTML5.

Let us recall that there are currently three infrastructure domain resource types in OCCI: Compute, Storage and Networking. Users interact with these in either a singular or plural fashion. These resource types need to be expressed using XHTML5 & RDFa in these modes.

**Singular Mode**

In order to serialise the representation of one infrastructure domain resource type, its triplets need to be structured in XHTML5 in the following way:
In order to serialise the representation of a multiple infrastructure domain resource instances, their triplets can be serialised using XHTML5 in two fashions:

1. **By Reference.** In this fashion, a triplet that refers to each resource instance in the generated set is serialised. The triplet does not contain detailed information such as the instances attributes. The clients receive a set of references to instances that can be dereferenced so that further information regarding the target instance can be received. It is analogous to using pointers in programming language to refer to values. With that specified, below is a simple set of three instances serialised By Reference using XHTML5 & RDFa.

```xml
<doc>
  <head>
    <title>OC1I Resource</title>
  </head>
  <body>
    <h1>An OCCI Resource</h1>
    <form method="POST" action="#"/>
    <div about="#" typeof="occi-Iri:Instance">
      <h2>The title</h2>
      <h3>The hostname</h3>
      <h4>The architecture</h4>
      <h5>The CPU speed</h5>
      <h6>The number of CPU cores</h6>
      <h7>The total system memory</h7>
      <h1>A member of the category</h1>
      <h2>A linked kind (Network)</h2>
    </div>
  </body>
</doc>
```

**Figure 21: An example of a XHTML/RDFa Instance Single-mode Serialisation**

**Plural Mode**

In order to serialise the representation of a multiple infrastructure domain resource instances, their triplets can be serialised using XHTML5 in two fashions:

1. **By Reference.** In this fashion, a triplet that refers to each resource instance in the generated set is serialised. The triplet does not contain detailed information such as the instances attributes. The clients receive a set of references to instances that can be dereferenced so that further information regarding the target instance can be received. It is analogous to using pointers in programming language to refer to values. With that specified, below is a simple set of three instances serialised By Reference using XHTML5 & RDFa.
2. **By Value.** This fashion is the same as the "By Reference" fashion only not only is the reference to the instance included but also all the immediate details of that instance must be serialised. Linked instances related to the instance to be serialised must only be serialised by reference. This fashion is analogous to passing copies of values in programming languages. With that specified, below is a simple set of three instances serialised By Value using XHTML5 & RDFa.

---

```xml
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RDFa 1.0//EN" "http://www.w3.org/MarkUp/DTD/xhtml-rdfa-1.dtd">

<html lang="en">
  <head>
    <link rel="stylesheet" type="text/css" href="example.css"/>
  </head>
  
  <body>
    <div><span about="compute" type="ocl:Category">compute</span>
      <span about="compute" property="ocl:term" content="compute"/>
      <span about="compute" property="ocl:scheme" content="http://purl.org/occi#compute"/>
      <span about="compute/123-123-123" type="ocl:Compute"></span>
      <span about="compute/124-124-124" type="ocl:Compute"></span>
      <span about="compute/125-125-125" type="ocl:Compute"></span>
      <span about="compute/126-126-126" type="ocl:Compute"></span>
    </div>
  </body>
</html>
```

---

**Figure 22: An example of a XHTML/RDFa Instance Plural-mode By Reference Serialisation**
Figure 23: An example of a XHTML/RDFa Instance Plural-mode By Value Serialisation
**OCCI, RDFa & XHTML5 and write-class operations**

OCCI maps conveniently to both read-class (e.g. HTTP HEAD and GET) and write-class (e.g. HTTP PUT, POST and DELETE) operations when using HTTP header serialisation or indeed, although not dealt with, XML-RDF.

The mapping is not so straightforward when using the XHTML5/RDFa serialisation, however, especially in the case of write-class operations.

If the client was a plain programmatic one then the client could simply use a write-class operation to send back a modified model in the serialisation format received. This, however, is not so well defined when the client is the humble and ubiquitous web browser. The web-browser is expected to be the most common platform that will need to interact with OCCI-based services using XHTML/RDFa documents. This particular aspect is an open topic currently (July 2010) being addressed by the working group. It is foreseen that the use of client-side JavaScript and draft RDFa APIs [41] will be used together to form a solution.

### 5.2.4 OCCI and the Web

OCCI is a specification which has to deal with incoming requests from the world wide web as this is the environment in which cloud services including cloud computing "live". It is HTTP and its underlying protocols (including the self-healing properties of TCP) that is the core fabric of the world wide web and as such OCCI has adopted HTTP as the bearer transport for itself. OCCI also employs a Resource Oriented Architecture (ROA), where the system is modelled as a set of related resources. Interaction with the system is by inspection and modification of its state, be it on the complete state or a sub-set. ROA's use Representation State Transfer (REST) to cater for client and service interactions. In these interactions, clients request to perform operations on the state of an individual or set of resources managed by the service. Those resources within the system can be and must be uniquely identified and operated upon by a limited set of operations that do not leak the resource model semantics in their signature. HTTP is an ideal protocol to use in ROA systems as it provides the means to uniquely identify individual resources through URLs as well as operating upon them with a set of general-purpose operations, the HTTP verbs. These HTTP verbs map loosely to the resource related operations of Create (POST), Retrieve (GET), Update (POST/PUT) and Delete (DELETE). HTTP also provides a number of other verbs within the core specification [37] but their use is not widespread. In order to implement OCCI in such a way to be a ROA and satisfy the REST model a number of aspects need to be specified. Certain categories can be excluded from results by prefixing with "-" and schemes may be specified within braces "{}", allowing for query by resource type scheme.

**Discovery and Query**

OCCI Clients need to discover what categories a provider support before interacting with that provider. Indeed, some providers may support other domain-specific resource types other than Compute, Storage and Network. To determine such information the clients can use the Category management endpoint. This endpoint is discovered by issuing a HTTP HEAD request against the root URL of the provider (e.g. http://example.com/). The HEAD request must be accepted and a Link to the Category management URL returned within the
response. This should be, relative to the root URL, ‘/’/. This Category management URL allows the client to get a listing of all categories supported by the provider. Should these providers allow and support client-created categories then this URL endpoint must support the creation of user categories. Use of the Category management endpoint, from the perspective of the client, is optional but **must** be supported by the provider.

### Identification of Resources

Each and every resource type instance within an OCCI system **must** be uniquely identified by URLs. The structure of these URLs is opaque and the system should not assume a static, pre-determined scheme for their structure. It is the OCCI Category (see page 47) system coupled with the query interface (see page 63) that allows for such dynamic URL structured schemes.

### Interaction with Resources

As OCCI adopts a ROA, REST-based architecture and uses HTTP as the foundation protocol the means of interaction with all resource instances is through the four main HTTP verbs. OCCI service implementations **must**, at a minimum, support these verbs.

<table>
<thead>
<tr>
<th>HTTP Verb</th>
<th>RFC [37] Definition</th>
<th>Usage in OCCI Context</th>
</tr>
</thead>
</table>
| POST      | "The POST method is used to request that the origin server accept the resource enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line" | This is the verb that will be used most often when creating resources.  
- POST can be used to create an resource type within a collection. In this case the target URL is one pointing to the Category (collection).  
- POST can also be used to create a sub-resource of an existing resource type. This is used in the case of issuing an Action against a resource type. In this case the target URL is one pointing to the resource type instance.  
This verb **must** be implemented. |
| GET       | "The GET method means retrieve whatever information (in the form of an resource) is identified by the Request-URI" | Retrieving a resource using GET will result in a representation of the target resource, specified by the request URL, in a serialisation format specified by the HTTP Accept header.  
- Clients issuing GETs must send the Accept header.  
- If the service does not support the requested serialisation format, it must respond with the HTTP code 406, not acceptable. This verb **must** be implemented. |
| PUT       | "The PUT method requests that the enclosed resource be stored under the supplied Request-URI" | This is the verb that will be used most often when modifying/updating instances.  
- PUT must be used to modify an resource type instance. In this case the target URL is one pointing to the resource type instance.  
- PUT can also be used to modify the contents of a collection. An example is adding an existing instance to a Category (collection). |
In this case the target URL is one pointing to the Category (collection). This verb must be implemented.

| DELETE | "The DELETE method requests that the origin server delete the resource identified by the Request-URI." | This verb is used to destroy resource instances. To destroy an instance, the target URL must be the URL to the specific resource type Category (See Table 9) and the identifier to the instance (e.g. http://example.com/compute/123-123). A client can also specify only the URL to the specific resource type Category. In this case ALL the user’s resource instances will be destroyed. The DELETE verb can also be used to remove one or ALL resource instance(s) associations from a particular non-mandatory Category. To remove one the target URL must be the URL to the Category (See Table 9) and the identifier to the instance (e.g. http://example.com/mywebapplication/123-123). If the target URL is only the URL to the Category then ALL instances associated with the Category will be removed. This verb must be implemented. |

Table 16: Core Mandatory HTTP Verbs and OCCI Usage

Supplemental HTTP Verbs

As previously mentioned, there are other HTTP verbs that can be supported. There are two HTTP verbs that are useful for retrieving certain meta-data of resource representations and the system.

<table>
<thead>
<tr>
<th>HTTP Verb</th>
<th>RFC [37] Definition</th>
<th>Usage in OCCI Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEAD</td>
<td>&quot;The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response&quot;</td>
<td>For example, this is useful in the case that you are solely relying on the HTTP header serialisation format where you want to avoid the possibility of any HTTP body being returned in the response. This verb may be implemented. If it is not implemented and a HEAD request is received a HTTP 405, method not allowed must be issued.</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>&quot;The OPTIONS method represents a request for information about the communication options available on the request/ response chain identified by the Request-URI.&quot;</td>
<td>The OPTIONS verb may be implemented. If it is not implemented and an OPTIONS request is received a HTTP 405, method not allowed must be issued. The OPTIONS verb is an interesting one. This might provide a means to provide information about certain features the OCCI implementation supports e.g. serialisation formats, authentication methods.</td>
</tr>
</tbody>
</table>

Table 17: Supplemental HTTP Verbs and OCCI Usage

Common Required Response Handling

Other than the specific response handling to some of the verbs in the previous section, there are common response handling behaviours that must be considered.
If a request succeeds the HTTP code 200, OK, **must** be returned along with the appropriate content.

If a request succeeds but not complete, in the case of delayed, asynchronous processing, the HTTP code 202, accepted, **must** be returned. The HTTP Location header **must** also be set to a resource, be it the target or intermediate resource, that should provide status information. The body of the response **should** contain appropriate content.

If a request is malformed, the service **must** respond with HTTP code 400, bad request. The body of the response **should** contain appropriate content. A request is deemed to be malformed if the service receives unexpected content such as an unrecognised attribute.

If there is an error in processing the request, the service **must** respond with HTTP code 500, internal server error. The body of the response **should** contain appropriate content.

**Security & Authentication**

OCCI does not require that an authentication mechanism be used nor does it require that client to service communications are secured. It does recommend that an authentication mechanism be used and that where appropriate, communications are encrypted using HTTP over TLS. The authentication mechanisms that can be used with OCCI are those that can be used with HTTP and TLS, for example Basic [42], Digest [42] and OAuth [43]. If an OCCI service requires authentication the response to a request that must be authenticated **must** be a HTTP 401 code that indicates the request is unauthorised. In response to authenticate the client must set a WWW-Authenticate header field and through this indicate the authentication mechanism.

**Versioning**

Versioning information pertaining to what version of OCCI is supported by a provider must be advertised to a client using the HTTP header, Server. The Server field should include the value OCCI/X.Y, where X is the major version number and Y is the minor version number of the implemented OCCI specification.

Mirroring the service side behaviour of an OCCI implementation, a client should indicate to the OCCI service implementation the version it expects to interact with. For the clients that implement this, the information must be advertised in the HTTP header User-agent. The User-agent field must include the same value (OCCI/X.Y) as specified for the Server HTTP header (above).

If a server receives a request from a client that supplies a version number higher than the service supports, the service must respond back to the client with the HTTP code 501, not implemented.

**Integration with Other Standards**

A guiding principle in OCCI is not to reinvent the wheel and always reuse standards and specifications where appropriate.
**SNIA/CDMI**

OCCI and the Storage Networking Industry Association's (SNIA) [44] Cloud Data Management Interface (CDMI) [45] working groups have collaborated together so that both specifications are interoperable with each other. From [46] "The SNIA Cloud Data Management Interface (CDMI) is the functional interface that applications will use to create, retrieve, update and delete data elements from the cloud. As part of this interface the client will be able to discover the capabilities of the cloud storage offering and use this interface to manage containers and the data that is placed in them. In addition, metadata can be set on containers and their contained data elements through this interface". The manner in which CDMI and OCCI can be integrated and interoperate is described in [47].

**DMTF OVF**

OCCI and the Distributed Management Task Force's (DMTF) [13] Open Virtualisation Format (OVF) [14] can be easily integrated through the use of the resource type Link. Where a provider wishes to supply an OVF representation of a client's resource instance(s), they can do so by associating the instance(s) with a mirror representation. The serialisation format used is then OVF.

**IETF Web Linking - Link Headers**

OCCI uses the IETF Web Linking draft specification [33] to express relationships between OCCI and non-OCCI resources.

**5.2.5 Extending OCCI**

OCCI is easily extendable through scheme and namespace imports. For individuals and organisations wishing to extend the standard, they are encouraged to submit the extension for peer review. When formulating their extensions they must use unique namespaces. The review process will validate those namespaces.

**How to Extend an Existing Resource Type**

In SLA@SOI, we required certain additions to the model in order to support SLA-aware infrastructures. To support asynchronous notifications of completed provisionings, SLA@SOI added to the Compute resource type an attribute that was a URI to which a message or request could be issued to. The first step was to establish our unique namespace. This was done using similar techniques used when structuring a company's Java packaging namespace. For our unique namespace we chose "eu.sla-at-soi". As the change was to affect the Compute type, we then defined an attribute named "eu.sla-at-soi.compute.notification-uri". To this attribute we set its type to be a string. No other changes to the OCCI model were required to support this attribute. The client and service implementations, however, were required to have their implementations updated.
5.2.6 Open Issues with OCCI

OCCI is a draft standard at the time of writing, and several open issues have been noted and are being worked. Some of these issues have been directly discovered by SLA@SOI Infrastructure Management, during our implementation effort. Others have been observed by the RESERVOIR project that is also contributing to the standard. Best-practices and workaround for these issues were documented in a shared document titled "Implementer & Integrator HOWTO" [48]. The purpose of that document is to explain OCCI in terms of implementation and integration phases of an OCCI client and OCCI provider, and to address issues that arise during those phases. Some of the key open issues are listed below.

- **Atomic provisionings**: Owing to the nature of HTTP headers it is inelegant to specify several resource interactions in a single atomic request. HTTP headers are inherently flat in structure - they are nothing but key-value pairs - and so are inefficient for attempting to describe hierarchical groupings of information. SLA@SOI has worked around this issue by hiding the lack of this feature in the client library so that it appeared to users of the client library that a request containing multiple resources requests was issued atomically. A more permanent solution is currently being discussed and is based on RDF.

- **Collections**: Somewhat related to the issue of atomic provisionings and certain weaknesses in HTTP headers, collections in OCCI are somewhat underspecified in the current public issue of the specification. This needs to be strengthened to give implementers better direction on this issue. The current discussions regarding RDF look promising and SLA@SOI hopes that their contributions are helpful and received well.

- **Monitoring**: SLA-aware infrastructure services require interfaces for both infrastructure provisioning and infrastructure monitoring at runtime. Whilst OCCI addresses the former, monitoring is not yet addressed. Ideally a standardised interface should be developed to identify and subscribe to a particular metric on a particular resource or group of resources. SLA@SOI Infrastructure Management are currently investigating how we can integrate such a monitoring API within the TCloud API [49], which was an API that was first introduced to the work package during our collaboration with RESERVOIR (more details of that collaboration can be found in the B8 deliverable on collaboration and in the technical paper on Using Cloud Standards for Interopability Frameworks [26]).

5.2.7 Current Status and Future

At the time of writing the standard is being finalised and is now awaiting to be released as a peer-reviewed specification and OGF recommendation. Along with the specification are four early reference implementations. Those implementations come from SLA@SOI, Project Claudia, OpenNebula and INFC. In parallel, an update to specification is being worked on and is scheduled for release in October 2010.
6 Implementation

6.1 Infrastructure SLA Manager

As discussed earlier in Section 3.5, the implementation of the ISLAM can be essentially decomposed into the implementation of a POC and a PAC custom to the infrastructure. These two infrastructure-specific components are elaborated in detail in Deliverable D.A5a [25], in the context of tasks TA5.5 (Planning and Optimization) and TA5.7 (Provisioning and adjustment). Nevertheless, we provide here a quick overview, for reasons of completeness of the document.

6.1.1 Infrastructure Planning and Optimization

Figure 24 depicts the internal architecture of the ISLAM POC:

The core component is the “Runner”, which executes the (usually more complex) tasks of evaluating SLA offers, or creating new SLA offers, according to some utility measure. The “MonitorabilityChecker” communicates SLAs to the MonitoringManager, so that the capability to monitor a SLA is confirmed. The “PlanManager” integrates with the PAC by means of submitting to it the plans that “ResourceProcessManager” creates. Then, “ProvisionStatus” receives information from the PAC, so that if some provisioning fails, other options/alternatives can be evaluated.

The algorithm implemented essentially solves the following problem:
Minimize the cost of implementing an SLA for infrastructure resources of specific type and quantity;

Given a base cost that may be increased by the provider to achieve better quality (without the customer having requested to do so) as a means to improve QoS and therefore, in the long run, reputation;

Taking into account mass-purchase reductions, and the cost of implementing some increased QoS that the user requested;

Ensuring that the standard profit, according to implementation costs, minus extra costs (voluntarily taken up by the provider) exceeds some threshold value; and

Keeping failure probability (as it results from prediction based on models, monitoring data or otherwise) lower than some threshold value.

Figure 25 illustrates this problem definition.

\[
\begin{align*}
\min & \sum_{i} (1 + \beta^i) \cdot N^i \cdot \sigma^i \cdot C^i_B + C^i_E \\
\sum_{i} \left[ g^i(C^i_I, C^i_E) - \beta^i \cdot C^i_I \right] & \geq F^* \\
h^i(\beta^i, T^i) & \leq P^i_{V^*} \\
0 & \leq \beta^i, \forall i
\end{align*}
\]

Figure 25: Problem solved by the ISLAM planning algorithm

The result of solving this problem is a selection of resources to be outsourced if they are not available locally (represented by an external implementation cost), the additional QoS measures for the provider, and eventually, a price quotation to be handed to the customer – that is, the value to be minimized in the first equation.

6.1.2 Infrastructure Provisioning and Adjustment

The Provisioning and Adjustment Component has a generic part (illustrated in Figure 26 in the form of a class diagram), and a non-generic part that must be customized before applying to a specific use case.

In the case of the ISLAM, the ActionExecutionTask had to be re-implemented, in order to invoke properly the Infrastructure Service Manager (ISM). This task must be added to an agent; on system start-up, this agent configures and instantiates itself, finally then starts the specified tasks by communicating with the ISM.

Once the system is provisioned, the PAC starts listening to the event bus in order to receive messages informing about the status of the service. These messages usually are domain-specific, so the format and a parser to translate the xml message to Java classes should be provided and added to the configuration file.

Monitoring events are fed into the drools rule engine managed by an AnalysisTask. For the infrastructure layer, specific rules have been written which analyze the different received events and trigger different actions: reprovisioning, re-start of VMs or allocation of extra-resources.
As is the case with Section 6.1.1 and the POC, much more elaborate implementation details can be found in Deliverable D.A5a.

### 6.2 Infrastructure Service Manager: Proxy (OCCI)

One of the main requirements of the ISM is to provide a standardised means of accessing different types of infrastructure service providers in a generic manner. By using a common interface, an infrastructure SLA Manager could interact in a consistent way with multiple infrastructure service providers that implement different types of access methods and protocols to provision infrastructural resources.

Accessing infrastructural services will typically involve calling the remote service that provides the infrastructure provisioning capabilities on the infrastructure provider side.
The generic SLA Manager, and by inheritance the infrastructure SLA Manager, allows by default to swap any of the components that integrate the SLA Manager either by configuration during design and deployment time, and even in principle it should be possible to change at runtime based on the OSGI component implementation.

From an abstraction point of view, different ISM Proxies could be implemented to support different standards and/or different infrastructure service providers. The implementation provided during Y2 of the project has focused on developing a Client-Side (Proxy) OCCI reference implementation for an OCCI compatible infrastructure provider. Figure 27 illustrates this conceptually.

The infrastructure SLA Manager is in principle agnostic to the type of service provider that is implemented, which is only determined by the type of ISM proxy that it implements. Figure 27 illustrates how the ISM Proxy is architecturally located within the infrastructure SLA Manager. In principle, the ISM Proxy is a domain specific component, whereas the Infrastructure SLA Manager is an SLA Manager that is specific to infrastructure, but domain agnostic with regards to service providers.

Figure 27: Relationship between Infrastructure SLA Manager, ISM Proxy and ISM

6.2.1 <<prepare_service>> and <<manage_service>> Interactions

The architecture specifies that the ISM should support mainly two types of interactions

- <<prepare_service>>
- <<manage_service>>

The <<prepare_service>> interaction has a series of operations that are related to the reservation of resources such as query(), reserve() and release(). A typical scenario where these operations are used is in a Grid Computing setting, whereby resources need to be reserved, scheduled and placed in a queue for
provisioning for running non-interactive jobs simulations, or cpu intensive operations.

Although basic reservations are supported based on memory usage for computational resources, for the current implementation we have prioritised the development of an on-demand provisioning facility with the <<manage_service>> (provision()) over the <<prepare_services>>, as requested by the requirements of the SLA@SOI use cases. The guiding principle for the current implementation has been to assume that infrastructure service provider will have enough resources to meet up with demand, and that resources are instantiated via an on-demand resource provisioning in a “cloud” like manner rather than a “grid” like manner.

The differences between Cloud computing and Grid computing are often subtle [50], [51], [52], [53]. However as SLA@SOI is oriented towards business applications rather than batch processing/scientific grids type of workloads, the on demand provisioning approach has been prioritised.

6.2.2 ISM Proxy OCCI Implementation

To support the OCCI specification, we have developed a first reference implementation that allows the creation of a provisioning request data model. The ISM Proxy contains the logic that maps from a ProvisionRequestType object to a series of OCCI HTTP REST requests.

ProvisionRequestType is the basic data model used internally by the ISMProxy. This data model is an extension of the Infrastructure Landscape domain model that is used by the ISM, as illustrated in Figure 31.

A ProvisionRequestType contains a set of Kinds, of which they can contain an arbitrary amount of Compute, Network or Storage Categories (Resources). OCCI requires to issue a provision request per specified resource. For example, if the ProvisionRequestType specifies 10 compute resources (10 Virtual machines), then it is necessary to group the resources around the same provisioning ID, and issue 10 requests to the OCCI compatible service provider.

Figure 28 illustrates the content of a ProvisionRequestType serialized to XML using XSTREAM [54].
Figure 28: Example Provision Request Type

This `ProvisionRequestType` contains the provision specification for five Compute resources (only one Compute resource Kind is shown) with the following Kind:

One Virtual Machine with the following specification:

- **Location = Ireland**
- **SLA = Silver**
- **cpu_cores = 4**
- **cpu_speed = 2.6 Ghz**
- **Memory_size = 4096 MB**

The **SLA = Silver** attribute has a domain specific meaning for the actual implementation of the ISM. The `ProvisionRequestType` also specifies a notification URI that can be used within the framework to get a message when
the provisioning is done, and also for any other type of notifications between the
ISM and the infrastructure SLA Manager. The request also contains a provisioning
start and stop time, used for indicating when to start/stop the provisioning, and a
monitoringRequest (specified as an URI) that indicates the location of a
monitoring configuration that will be used for the provisioning of the monitoring.

**<<prepare_service>> Implementation**

One limitation of the current OCCI specification is that it caters only with on-
demand provisioning and therefore it does not support reservations. To allow the
infrastructure SLA Manager to interact with the resources directly, the
functionality of the ISM has been extended to support alternative means of
querying and reserving the resources. **<<prepare_service>>** has the following 4
methods: **query**, **reserve** and **release**:

- **query**: The current implementation of query will compute the total
  amount of memory required by the specified resource configuration
  specified by a **ProvisionRequestType**. If there are enough resources
  available, it will return the same or a similar resource configurations to the
  one provided. If resources are not available, the ISM might return an
  empty configuration or a set of different resource configurations. With
  virtualization, the main physical limitation for server capacity is memory,
  with the assumption that most of the provisioned virtual machines will not
  use 100% of other resources such as cpu or io. Currently time bound
  queries are not supported, only on-demand provisioning is supported.
- **reserve**: reserves the resource configurations. Currently time bound
  reservations are not supported, only on demand provisioning is supported.
  The reservation will expired after a pre-determined period of time.
- **release**: releases an existing reservation.

The **<<prepare_service>>** 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**: provisions a reservation. This method is called after the
  preparation of the service. As more than one resource could be
  provisioned, this method runs inside a transaction, implemented by a 2
  phase-commit. The transaction will release any provisioned resources if
  there are any problems completing the request.
- **provision**: provisions the specified resources where o reservations
  required. It also runs in a transaction. This method is designed to support
  the on-demand provisioning scenario required for the cloud computing
  approach.
- **getDetails**: Obtains details of a running provision. This method calls the
  ISM and tries to retrieve information related to the provision.
- **reprovision**: this method tries to update a running provisioning with the
  specified provisioning.
- **startResource**: signals a resource to start.
• **stopResource**: signals a resource to stop.
• **stop**: stops the resources.

In order to facilitate the integration of the ISM Proxy and the ISM with other SLA@SOI components, the ISM proxy interface, also provides helper methods to create resource specifications such as Compute, and to create `ProvisionRequestType` objects from sets of Kinds. Rather than create a Compute Kind manually, it is possible to use the helper classes to do it.

### 6.2.3 Component Reuse: ISM Proxy Standalone and OSGI Integration

The ISM Proxy is available in two forms, as a standalone component and as an OSGI [55] bundle. The OSGI version is provided by another component that wraps the basic functionality of the standalone ISM Proxy and configures it to be able to run as an OSGI service.

With regards the SLA@SOI framework and OSGI, is also required that all its dependencies are converted to OSGI bundles (or made available to the OSGI container as bundles) and a Pax-Runner [56] profile needs to be created to manage any dependencies that the ISM Proxy has. Pax-Runner is the preferred OSGI bundle provisioning framework chosen by the project.

It is possible to run ISM Proxy, integrated with other components, standalone as for example a Spring application. Internally, the ISM Proxy has been implemented as a collection of Spring beans [57], allowing the replacement of any of the bean implementation for different ones, allowing a (design time) pluggable architecture for the modules themselves.

For example, it could be the case that the OCCI implementation required for a use case such as B4 needs to be extended to support extra features such as private cloud authentication.

Figure 29 illustrates the wiring of a basic SLA Manager using Spring, that utilises some of the components of the SLA Manager as simple Spring beans, such as the SLARegistry (from the Generic SLAM), the PAC (from Infrastructure SLAM) or the POC (also from the Infrastructure SLAM).

The ISM Proxy also integrates 3 different types of registries: SLAType, Location and Image which are also implemented as Spring beans.

These registries will provide domain specific mappings for the actual ISM to register what it can offer, such as locations (which countries), SLA Types (Gold, Silver etc). Currently these registries are static.
6.2.4 ISM 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 manly related to the \texttt{<<manage\_service>>} interaction. However for \texttt{<<prepare\_service>>} the process is very similar.

The main steps required:

1. Create a Compute resource instance with only one VM description
2. Create a ProvisionRequestType and add the Compute instance to it
3. Call provision to perform the actual provisioning.
4. Manage the provisioning (getDetails).
5. Stop the provision.

1. \texttt{Create Compute resource configuration}

    // *******************************************
    // Using the ISM Proxy

---

Figure 29: Structure of the ISLAM in terms of Spring Beans

---

Spring Beans

Beans Overview

Select an element to edit its details.
// Steps for provisioning a Virtual Machine (VM)
// 1 - Create a Compute representation of the VM
// 2 - Create a ProvisionRequestType and add the Compute Representation
// 3 - Call provision()
// 4 - getDetails() of or provision
// 5 - stop() our provision
// ***************************************
// We create a Compute Resource first
Category computeCategory = new Category();
computeCategory.setTerm(Terms.COMPUTE);
computeCategory.setScheme(Schemas.OCCI);
computeCategory.setLabel(Labels.COMPUTE_RESOURCE);

// Appliance Category (What image type we should use)
Category applianceCategory = imageregistry.getCategoryByID("Ubuntu 10.4 x86_64");

// SLA Under what SLA Type we should provision this resource (Domain Specific)
Category slaTypeCategory = slatyperegistry.getCategoryByID("GOLD");

// Location
Category locationCategory = locationregistry.getCategoryByID("Ireland");

// We create the actual Compute configuration
Compute computeConfiguration = new Compute();
computeConfiguration.setCpu_cores(Cores.FOUR);
computeConfiguration.setCpu_speed(CpuSpeed.Medium);
computeConfiguration.setMemory_size(Memory.Large);
computeConfiguration.setHostname(hostName);

// We add the Categories to a Set
Set<Category> categories = new HashSet<Category>();
categories.add(applianceCategory);
categories.add(computeCategory);
categories.add(slaTypeCategory);
categories.add(locationCategory);

// We associate the categories to the compute configuration
computeConfiguration.setCategories(categories);

// We create a set of compute configurations and add
// our Compute Configuration
Set<Compute> computeConfigurations = new HashSet<Compute>();
computeConfigurations.add(computeConfiguration);

2 – Create ProvisionRequestType and add the Compute Kinds Set
// *****************************************************
// We create a ProvisionRequestType and add the Compute representation
ProvisionRequestType provisionRequestType = new ProvisionRequestType();
provisionRequestType.setKinds(computeConfigurations);

// Start now, run forever
Date startTime = new Date(System.currentTimeMillis());
Date stopTime = startTime;
provisionRequestType.setProvStartTime(startTime);
provisionRequestType.setProvStopTime(stopTime);

// Notification URI for Infra. Provisioning and Adjustment Component
provisionRequestType.setNotificationUri(iPAC.notificationURI);

// Monitoring configuration
provisionRequestType.setNotificationUri(MonitoringConfiguration.DEFAULT);

3 – We perform the actual provisioning

// ******************************************************
// ******************************************************
// We call the ISM and try to provision
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 – Manage our provision

// ******************************************************
// ******************************************************
// We try to getDetails of our provisioning
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 resource
        URL url = endPoint.getResourceUrl();
    }
6.3 Infrastructure Service Manager: Allocation and Management

The Infrastructure Allocation and Management (IAM) component is a web service that implements the OCCI standard (discussed in detail in Chapter 5). The web service implements a RESTful [36] interface due to its compliance with 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 (see Section 6.2.1). Furthermore, the IAM has the responsibility of creating (including commitment of resource reservations), updating, retrieving information on and deleting resources under the management of the IAM. These correspond to the <<CommitRUD>> interaction.

The IAM 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 IAM and consequently is very much aligned to the OCCI model (see Section 5.2.2).

The IAM and also the IL are implemented using the web framework Grails [58], a rapid web development framework and environment that uses the dynamic language, Groovy [59] 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 [60]. For the purposes of the IAM 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 30).

The IAM internal architecture only shows the immediate components that are implemented in order to satisfy its responsibilities.
Figure 30: IAM Internal Architecture

- **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 IAM carries out on the behalf of its clients are recorded here.
- **Services**: These implement stateful or long running business logic. For example the SchedulerService. When clients request the reservation of infrastructure resources, the controller (Scheduler) on their behalf makes an entry in the IL. The SchedulerService then selects reservations that are due to be executed and when that due date occurs instructs the ProvisioningSystem (via the MessagingService) to make the reserved resources available.

Shown in the diagram above is the IL and as stated already is very much aligned to the OCCI model. This model when viewed as an object-oriented (UML) model rendering looks like the following.
Details related to the attributes of the types shown above can be found in Section 5.2.2.

Also shown in the internal architecture diagram are the Message Bus and ProvisioningSystem. The IAM, hence ISM, and the target provisioning system adopt an event-based architecture and as a result are cleanly separated by the use of messaging middleware (Message Bus). This brings great flexibility to the system. As the IAM is loosely coupled to the ProvisioningSystem this enables many different deployment configurations. As the message formats that the IAM 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/modified to adapt to the target ProvisioningSystem’s communication mechanism.

In our case, the ISM in SLA@SOI uses Apache Tashi as a provisioning system. Our messaging middleware implements the open standard XMPP and so it was required that we write a XMPP message sink that would receive the XMPP messages from the ISM and duly process them. Once processing was complete a responding message was sent back to the ISM. The ISM then, on receipt, of the provisioning system result updated the IL state and then sent notification of the result to the original requestor.
6.4 Tashi Implementation

6.4.1 Tashi XMPP Daemon

In order to integrate Tashi with the infrastructure service manager, a message sink process that receives messages from the ISM 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. Besides the provisioning requests, the daemon is also listening to the requests for reprovisioning and user registration. The daemon will not return a response immediately to the ISM but rather will notify the ISM asynchronously once the provisioning process has completed.

6.4.2 Management of CPU and Memory Resources

There is a strong need for managing the performance of virtual machines in the SLA@SOI framework. To achieve this we decided to use Cgroups, which stands for Control groups, essentially the resource manager for Linux containers. With Cgroups it is possible to assign shares of CPU, memory and I/O bandwidth to processes according to their importance. Unlike some other scheduling techniques that set proportions of resources to processes based on weights, Cgroups can set minimums, allowing guaranteed resources to be assigned to the process, but the process can also use more resources if available.

Cgroups in Tashi

If we want to manage a process, we have to set it to a particular Control group and specify a share of the resources for it. If one group is specified to have 30% of the CPU resources and another group 70%, then if the more privileged group does not need its full 70%, the other group can use what is left.
Cgroups are organized in a virtual filesystem, so we can control them by reading and writing files. Because we are starting virtual machines within Tashi, Cgroups need to be integrated inside it. We have to take care that virtual machine processes are attached to a specific Control group and we have to specify its share. Later on, in the runtime we can increase or decrease the share of each Control group if we detect or predict violations.

When a node manager receives a request for a new virtual machine from cluster manager, it starts the provisioning via VmControlInterface, which triggers the Qemu or Xen command, depending on which we are using. For example in the Qemu implementation of the VmControlInterface the startVm is the method that needs to be modified to make use of the Cgroups in Tashi. We had to add two functions which are invoked from the startVm method: qemuProcess and qemuParentProcess. In short, we had to insert one additional process between Tashi node manager process and VM process and attach this process to a new Cgroup.

```python
def qemuProcess(qemu, cmd, pipe_r, pipe_w):
    log.info("The qemu process with %s pid started" % os.getpid())
    os.close(pipe_r)
    os.dup2(pipe_w, sys.stderr.fileno())
    for i in [sys.stdin.fileno(), sys.stdout.fileno()]:
        try:
            os.close(i)
        except:
            pass
    for i in xrange(3, os.sysconf("SC_OPEN_MAX")):
        try:
            os.close(i)
        except:
            pass
    os.execl(qemu, *cmd)

def qemuParentProcess(vmName, q, qemu, cmd, cpu_share, memoryLimit, pipe_r, pipe_w):
    """ This function is used to insert one additional process between nodemanager process and Qemu process. On this process Cgroups are applied to ensure that all Qemu processes for this VM are inside this Cgroup. """
    log.info("qemuParentProcess started...")
    pid = os.getpid()
    log.info("The qemu parent process with %s pid started" % pid)
    os.setpgid(pid, pid)
    if not os.path.exists("/dev/cpu/tasks"): 
        print "mounting Cgroups CPU..."
        subprocess.call("sudo mkdir -p /dev/cpu", shell=True)
        subprocess.call("mount -t cgroup -ocpu cpu /dev/cpu", 
                        shell=True)
    # create a cgroup for this virtual machine:
    subprocess.call("sudo mkdir -p /dev/cpu/%s" % vmName, 
                    shell=True)
    # create a cgroup for this virtual machine:
    subprocess.call("sudo mkdir -p /dev/cpu/%s" % vmName, 
                    shell=True)
    print "CPU Cgroup %s created" % vmName
    # specify a share for this group:
    subprocess.call("echo %s | sudo tee /dev/cpu/%s/cpu.shares" % 
                    (cpu_share, vmName), shell=True)
    subprocess.call("echo %s | sudo tee /dev/cpu/%s/tasks" % (pid, 
```

In the code snippet above we can see how the command for starting a new virtual machine is generated in the `startVm` and then passed to the `qemuParentProcess` function. What we need to do there is to execute Cgroups commands after we start the virtual machine. We do this by using python functions from the subprocess packages.
First we have to mount Cgroups CPU container:

```python
if not os.path.exists("/dev/cpu/tasks"):
    subprocess.call("sudo mkdir -p /dev/cpu", shell=True)
    subprocess.call("mount -t cgroup -ocpu cpu /dev/cpu", shell=True)
```

If we now create a directory under /dev/cpu it will define a new Control group for CPU management.

```python
subprocess.call("sudo mkdir -p /dev/cpu/%s" % vmName, shell=True)
```

After a Control group for a virtual machine is created, the share for this group is specified as follows:

```python
subprocess.call("echo %s | sudo tee /dev/cpu/%s/cpu.shares" % (cpu_share, vmName), shell=True)
```

Now the processes where the virtual machine is running has to be attached to the newly created Control group:

```python
subprocess.call("echo %s | sudo tee /dev/cpu/%s/tasks" % (pid, vmName), shell=True)
```

We do the same for memory then.

```python
if not os.path.exists("/dev/memory/tasks"):
    subprocess.call("sudo mkdir -p /dev/memory", shell=True)
    subprocess.call("mount -t cgroup none /dev/memory -o memory", shell=True)
    # create memory Cgroup for this virtual machine
    subprocess.call("sudo mkdir -p /dev/memory/%s" % vmName, shell=True)
    print "Memory Cgroup %s created" % vmName
    subprocess.call("echo %sM | sudo tee /dev/memory/%s/memory.soft_limit_in_bytes" % (memoryLimit, vmName), shell=True)
    subprocess.call("echo %s | sudo tee /dev/memory/%s/tasks" % (pid, vmName), shell=True)
```

Each Control group has specified CPU and memory shares as defined in provisioning request. Later on, when a violation is detected or predicted, the share can be modified by calling reprovisioning functions, for example `adaptCpuShare` or `adaptMemoryLimit`, which were added to the `VmControlInterface`. 
6.4.3 Tashi Reprovisioning

Reprovisioning functions were written and added to the VmControlInterface to enable the dynamic reallocation of the VM CPU and memory resources. These functions simply modify the share of the Control group to which the VM is attached. The Qemu implementation of the adaptMemoryLimit function is reproduced below.

```python
def adaptMemoryLimit(self, vmName, limit):
    subprocess.call("echo %sM | sudo tee /dev/memory/%s/memory.soft_limit_in_bytes" % (limit, vmName),
        shell=True)
    log.info('Memory limit %sM given to the Cgroup %s' %
        (limit, vmName))
```

6.4.4 CPU Speed

The CPU speed notion was added and integrated in the Tashi. This was needed because CPU speed is part of the provisioning request and the original Tashi does not have CPU speed included in its scheduling algorithm, it contains only CPU cores number and memory size. The python objects that represent VMs and physical servers contain now CPU speed property and it can be managed by provisioning and reprovisioning functions. The adaptCpuShare for example changes the CPU share of a specific VM. Note that the share term is used, because it applies to the Cgroups terminology.

```python
def adaptCpuShare(self, instanceId, share):
    instance = self.data.acquireInstance(instanceId)
    hostname = self.data.getHost(instance.hostId).name
    try:
        self.proxy[hostname].adaptCpuShare(instance.vmId, share)
    except Exception:
        self.log.exception('adaptCpuShare failed for host %s vmId %d' %
            (instance.name, instanceId))
        raise
    instance.cpuShare = share
    self.data.releaseInstance(instance)
    return
```

6.4.5 Automatic Hosts Registration

The automatic host registration was implemented when we needed to integrate the CPU speed property and made some changes related to physical servers representation. Now when installing Tashi you don't need to manually add hosts in the database. Just start the nodemanager and the host will be registered automatically if it was not already.
def registerHost(self):
    cm = ConnectionManager(self.username, self.password, self.cmPort)[self.cmHost]
    hostname = self.getHostname()
    now = datetime.datetime.now()
    version = now.strftime("%Y-%m-%d")
    memory = self.getTotalMemory()
    cores = self.getNumOfCores()
    cpuSpeed = self.getCPUSpeed()
    cm.registerHost(hostname, memory, cores, cpuSpeed, version)

6.4.6  Tashi Scheduler Abstraction

In order to simplify the further development of the Tashi scheduler and to make our modifications pluggable, the SchedulerInterface was defined and integrated inside the Tashi implementation.

class SchedulerInterface(object):
    def __init__(self, client):
        if self.__class__ is SchedulerInterface:
            raise NotImplementedError
        self.client = client

    def setVmsForMigration(self, hosts, instances, freedHostId, load):
        raise NotImplementedError

    def handleMigration(self, instances, freedHostId):
        raise NotImplementedError

    def determineHost(self, hosts, load, instances, inst, targetHost, allowElsewhere):
        raise NotImplementedError

    def reschedule(self, inst, instances, hosts, load):
        raise NotImplementedError

SchedulerInterface specifies a set of methods which should be implemented by each scheduler. A small refactoring was performed to modify the existing Tashi scheduler to implement the SchedulerInterface.

Once the SchedulerInterface was integrated inside Tashi, we could use it for the EnterpriseIT use case and also for the general SLA@SOI framework. For example ECFScheduler implementation of the SchedulerInterface was developed for the EnterpriseIT needs, where the scheduler needs to obey the ECF (Enterprise Capability Framework) policy.
6.5 Infrastructure Runtime Prediction Integration Implementation

From an A4 WP perspective, the main usage of prediction data is to make prediction metrics available to the other components of the framework. Depending on deployment, prediction can publish its data to the LLMS components developed A4 using two different approaches:

- Using Ganglia Custom metrics injection [7]. As the underlying instrumentation of LLMS is Ganglia, it is possible to issue from each resource (physical or virtual) that is instrumented a command that will add prediction metrics the local resource, and Ganglia will take care of propagating it through the system.
- Using directly the LLMS API. The LLMS provides an API that allows the publication of metrics directly to the LLMS, by registering the metric or metrics and sending observations.

The choice of what approach to use will depend on the requirements of the particular prediction deployment.

By default, prediction data is associated directly with physical servers or virtual machines such as the ones that are running in a provisioning cluster (for example in a Tashi cluster). Therefore, any component that requires the use of runtime prediction must also have low level access to the ISM and internal knowledge about the infrastructure and also knowledge about the relationships between the VM, SLAs and users.

For each physical server or virtual machine, prediction data can take the following generic format:

\[
\text{PhysicalServer } [1...N] \ \text{[PastMetrics, PresentMetrics, FutureMetrics]} \\
\text{VM } [1...N] \ \text{[PastMetrics, PresentMetrics, FutureMetrics]}
\]

In short, prediction represent its data as [Past, Present, Future] vectors. The different parts of these vectors can be used by the SLA@SOI components to implement different planning, optimisations and/or adjustments strategies that can act upon historical and future information in a unified way. These vectors are derived from the pre-processing of raw metrics which are heavily customised for the intended use, such as deploying prediction for a use case.

For example, based on the history (past) of provisioned SLAs, together with the current instant load (present) of the cluster and an estimation of future load of a particular server (future), a planning implementation could locate a server or group of servers that are suitable for provisioning an SLA with known workloads.
6.5.1 Consuming Prediction Data

To access the prediction can be done in two ways: via the prediction service directly, or via the LLMS with its query capabilities. The main differences between the two approaches are that prediction data is offered via a REST service interface (pull) whereas the LLMS query facility is message-based (push). Prediction service can also offer higher level of pre-processing or post-processing of the data, in order to suit the needs of the use cases. The following section will illustrate how prediction data is queried and rendered from the SLA@SOI B4 use case perspective.

Data Query and XML Rendering

Consumers of data can query the prediction service using a domain specific REST call that needs to be customised to the specific requirements of the data consumer. The following example illustrates a query and rendering developed for the SLA@SOI B4 use case: the REST call performs a request for a resource (VM) with a specific id, for the last 4 minutes, for a cpu related metric:

http://predictionService:port/instrumentation?id=545200457a4c&metricName=cpu&minutes=4

The XML rendering of prediction data is illustrated in Figure 34.
Figure 34: Rendering of prediction data in XML

For this particular XML rendering, the data takes the following format:

[historytotal, lasthour, lastMinute, prectionerror, last, prediction]

Where the metrics are defined as:

historytotal: The average value measured for the metric (cpu) since monitoring of this service began. This metric is useful to determine the service utilization over time. For example an adjustment algorithm could look at this metric and optimize its usage by means of making sure this value is maximised. This metric changes slowly over time.
**lasthour**: The average value of this metric for the last minute. For example a planning component could evaluate this metric for a service and provision only on services (physical servers) that have a medium utilization.

**lastMinute**: The average value of this metric for the last minute. For example planning and optimization components could look at this metric and determine the services with lowest utilization and perform a provisioning that requires a high instant utilization (based on knowing the usage profile of the service that needs to be provisioned). This metric can change very rapidly.

**predictionError**: The calculated error for prediction, based on the training stage. This value is only estimation.

**last**: The average value of this metric for the training data set and time window of prediction use for training prediction. In general, this metric changes slowly and depends on the period of time considered for training.

**prediction**: The predicted average value for the configured period of time that prediction is calculated.

### 6.6 Infrastructure Monitoring

#### 6.6.1 Instrumentation

An Apache Tashi cluster is at the core of the infrastructure platform in Year 2. Its management information is accessible on a Cluster Manager Node while the runtime data is accessible through a tightly coupled Ganglia installation.

A Ganglia server (gmetad) collects data sent by Ganglia daemon (gmond) from other nodes (hypervisors and VMs). Current state of the cluster is accessible by querying the TCP port 8651. A custom-developed Ganglia connector queries in short periods and sends new observations (according to observation timestamp) to the Low Level Monitoring System (LLMS).

Any observation sent to LLMS represents the last value of a given metric. Metrics are identified by a **unique URL** and described by:

- Name
- Unit
- MetricClass (fluent, event, alarm)
- Data type

The format of metric URL is typically domain specific\(^5\), but should explicitly include resource(s) it belongs to. Metrics collected from Ganglia are registered under the following URL:

**slasoi://<service>/<cluster_name>/<host_name>/FLUENT/<metric_name>**

---

\(^5\) Violations, warning and other event and alarms are also stored as a metric, but using a different URL format.
6.6.2 Data Collection

Ganglia connector and custom sensors use the following API to register metrics and send observations.

**IServiceInstance2Monitoring:**
- registerMetric(MetricEx metric)
- unregisterMetric(String metricURL)
- isRegisteredMetric(String metricURL)
- storeMetricObservation(String metricURL, Date time, String value)
  - send observation

The API is implemented by local stub object. It takes care of marshaling, sending over messaging bus and receiving response synchronously. A dedicated pub-sub channel is used for observation exchange.

The observations are received by the LLMS Manager, running on some central node. To allow for scalability, several instances of LLMS Manager can run in parallel acting as one, but is not yet fully supported in Y2.

After reception observations are stored in HistoryInfoStore, while the metric definitions are stored in MetricRegistry that is a part of Infrastructure Landscape.

All these data are exposed to other components through the following API implemented by local stub⁶.

**IMonitoringDataAndHistory:**
- getMetric(String metricURL)
  - get metric details for the given metric URL.
- getMatchingURLs(String pattern)
  - get list of registered metricsURLs that match the given pattern
- getMetricValue(String metricURL)
  - get the current/last value of specific Metric
- getMetricValueHistory(String metricURL, Period period)
  - get the list of history values of specific Metric (or Event) for the given period.
- queryHistory(String query)
  - query the Monitoring HistoryInfoStore SQL database

Beside pull access to data other components can also subscribe to receive them in a push manner.

---

⁶ Other components call local Java calls.
6.6.3 Data Processing

In the processing phase data from HistoryInfoStore are aligned with the set of rules from ConfigStore. Each rule has a condition and action. When the condition is met the action is triggered. This is a data driven process. Old data are taken into effect only when explicitly used in condition.

The processing is done by RuleEngine using Drools Business Logic integration Platform capable of complex rules, event processing and temporal reasoning allowing invocation of arbitrary Java code from rule action statement.

Typical rules used in LLMS:

- Calculating derived metrics\(^7\) (averages, response times, etc.) from basic observation.
- Sending violations/warnings and other data based events.
- Sending specific data to subscribers.

```java
// send a CPU overflow event, whenever observation with CPU > 95% arrives
rule "CPU overflow"
  when
    o : Observation(metric.metricURL ==
      "slasoi://myManagedObject.company.com/TravelService/
      VMs/VM1/fluent/CPU",
    $value : value, Integer.valueOf($value) > 95)
  then
    // store an event
    manager.storeMetricObservation(cpu1OverflowEvent, new Date(), $value);
    // send event
    manager.sendGenericNotification(channelName, cpu1OverflowEventNotification, new Date(), o.metric.metricURL);
end
```

**Figure 35: Example Monitoring Configuration**

Initial set of rules is loaded from ConfigStore at framework startup while additional rules can be provisioned in runtime using the following API (implemented by local stub).

**IConfigureMonitoring:**

- provision(MonitoringRequest request)
- reprovision(Object requestID, MonitoringRequest newRequest)
- free(Object requestID)

\(^7\) Calculated derived metric value is stored in LLMS as an observation of derived metric. It is accessible in through IMonitoringDataAndHistory and can also be used in subsequent rules.
6.6.4 SLA Monitoring

An SLA is composed of a set of SLOs (SLA Objectives). Each defines the criteria for service acceptability. Whenever any of these conditions is violated, the SLA is violated. When an SLA might be violated soon, it is an SLA Warning.

Each SLO uses a formal logic (written in SLA Model format) to constrain the QoS terms (used variables). These QoS terms are calculated by LLMS as a derived metric.

The structure of the SLA Manager is described in Section 3.5.

In the negotiation phase the Monitoring Manager (MM) (part of the Infrastructure SLA Manager (ISLAM)) receives the SLA and checks whether it is monitorable. The SLA is monitorable if all required QoS terms are implemented (in service configuration) and if SLO constraints can be reasoned by deployed Reasoning Engine(s).

In the provisioning phase the MM divides the SLA into smaller sets of SLOs and delegates them to Reasoning Component Gateways (RCG) of appropriate Reasoning Engines. RCGs are required as the Reasoning Engines do not support the SLA Model syntax used for SLOs. Further RCGs interact with LLMS to regularly fetch the required QoS terms, feed them into Reasoning Engines and send SLA Violations when constraints are violated.
6.6.5 Monitoring of Available Resources

Provisioning and Adjustment require information of available resources potentially useful for service (re)provisioning. Note that this information is required prior to service (re)provisioning.

Using a hierarchical resource related structure of metric URL, available resources can be easily discovered in explored by IMonitoringDataAndHistory interface.

In scope of Infrastructure typical usage is finding the available hypervisors, checking their utilization and capabilities.
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 external testbed is most relevant to the B4 Enterprise IT Use Case, also led by Intel, and so has been largely shaped and driven by B4 Enterprise IT Use Case requirements.

7.1 External Testbed

In advance of installing any newly developed service in a lab environment for demonstrative purposes, it is important to consider several factors before deciding which hardware configuration best serves the requirements of the project.

When making these determinations about the hardware composition, the macro-level fundamentals of where the service might be applied in real-world implementations is an important element. Essentially, this equates to replicating a small-scale environment which ideally should possess the pertinent limitations which the new service or solution is addressing. In the case of an established business landing a new, but mature, service or solution, then it is obviously best practice to locate these on the most recent and most powerful hardware available. However, doing that in this use case would take away the opportunity to showcase the concepts around server efficiencies and investment governance which are important aspects of the Enterprise IT work package. To showcase these aspects of the use case it was important to include a mix of heterogeneous hardware as might be the case in existing data centres. This added to the substance of the use case, albeit with the drawback that the underlying hardware layer may not contain chipsets that support the latest virtualisation optimisations.

Table 18 summarizes the specification of the server hardware deployed in the external test bed.

Table 18: Summary of External Test bed Capacities

<table>
<thead>
<tr>
<th>Server Count</th>
<th>Cores</th>
<th>Raw Storage</th>
<th>Memory (RAM)</th>
<th>Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>64</td>
<td>6.8 TB</td>
<td>140 GB</td>
<td>External Internet: 10 MB/sec</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Internal Intranet: 1Gb/sec</td>
</tr>
</tbody>
</table>

The external test bed consists of eleven physical servers with a total of 64 CPU cores and 140 Gigabytes (GB) of random access memory (RAM). Five of the
servers are blades and six of the servers are classic rack mounted models. All ten servers use Intel® Xeon® processors. The raw disk capacity of the servers is 6.8 Terabytes (TB). The following table shows how this is broken down between the ten servers, and includes their high level role in the cloud.

<table>
<thead>
<tr>
<th>Server Name</th>
<th>Role</th>
<th>Form Factor</th>
<th>Processor Model</th>
<th>RAM (GB)</th>
<th>Raw Disk Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRICBL001</td>
<td>Tashi Node Manager</td>
<td>Blade</td>
<td>2x Quad-Core Intel Xeon®, 2.6 GHz</td>
<td>12</td>
<td>4x 72GB disks</td>
</tr>
<tr>
<td>IRICBL009</td>
<td>Tashi Node Manager</td>
<td>Blade</td>
<td>2x Quad-Core Intel Xeon®, 1.86 GHz</td>
<td>8</td>
<td>2x 72GB disks</td>
</tr>
<tr>
<td>IRICBL010</td>
<td>Tashi Node Manager</td>
<td>Blade</td>
<td>2x Quad-Core Intel Xeon®, 1.86 GHz</td>
<td>8</td>
<td>2x 72GB disks</td>
</tr>
<tr>
<td>IRICBL011</td>
<td>Tashi Node Manager</td>
<td>Blade</td>
<td>2x Quad-Core Intel Xeon®, 1.86 GHz</td>
<td>8</td>
<td>2x 72GB disks</td>
</tr>
<tr>
<td>IRICBL014</td>
<td>Tashi Node Manager</td>
<td>Blade</td>
<td>2x Quad-Core Intel Xeon®, 1.86 GHz</td>
<td>8</td>
<td>2x 72GB disks</td>
</tr>
<tr>
<td>IRICVTB001</td>
<td>Tashi Node Manager</td>
<td>Rack</td>
<td>4x Intel Xeon® dual-core 3.4GHz</td>
<td>32</td>
<td>8x 146GB disks</td>
</tr>
<tr>
<td>IRICVTB002</td>
<td>Tashi Node Manager</td>
<td>Rack</td>
<td>4x Intel Xeon® dual-core 3.4GHz</td>
<td>32</td>
<td>8x 146GB disks</td>
</tr>
<tr>
<td>IRICVTB003</td>
<td>Prediction Processing &amp; iSCSI (Y3)</td>
<td>Rack</td>
<td>2x Intel Xeon® MP 3.66GHz</td>
<td>8</td>
<td>5x 300GB disks</td>
</tr>
<tr>
<td>IRICVTB004</td>
<td>Tashi Cluster Manager</td>
<td>Rack</td>
<td>2x Intel Xeon® single-core 3.06GHz</td>
<td>8</td>
<td>5x 146GB disks</td>
</tr>
<tr>
<td>IRICVTB005</td>
<td>Prediction Processing &amp; iSCSI (Y3)</td>
<td>Rack</td>
<td>2x Intel Xeon® single-core 3.06GHz</td>
<td>8</td>
<td>5x 146GB disks</td>
</tr>
<tr>
<td>IRICVTB006</td>
<td>Prediction Processing &amp; iSCSI (Y3)</td>
<td>Rack</td>
<td>2x Intel Xeon® single-core 3.06GHz</td>
<td>8</td>
<td>5x 146GB disks</td>
</tr>
</tbody>
</table>

The lab environment consists of several distinct server roles as can be seen in Table 19 above. The Tashi Cluster Manager (CM) is installed on the primary management node within the cloud. This node has ultimate responsibility for Tashi operations. In addition it hosts the web user interface (UI), handles the scheduling of provisioning requests, runs monitoring and ensures policy enforcement. There are also six Tashi Node Managers (NM’s). The NM’s role is to control operation of the individual node within the cloud. Finally, there are three nodes who share responsibility for prediction processing and shared storage over iSCSI. iSCSI is an abbreviation for internet Small Computer System Interface which essentially is a standard which allows SCSI to be sent over IP network packets. The underlying operating system on all these servers is Ubuntu version 9.10.
Figure 37 illustrates how all this is put together. The Tashi Cluster Manager acts as a bridge-head server with an external interface which is accessible over the corporate intranet. The remaining servers share a high capacity 1Gb/sec Ethernet backbone and are also accessible over the corporate intranet. In this way, any hosted services can be reached provided the firewall allows access to the system.

The servers have been deployed in a DMZ area within an Intel data centre facility. A block of external internet IP addresses has been reserved and assigned to the machine, enabling multiple VMs to be externally accessible.

**Figure 37: Year 2 Testbed Infrastructure**
7.2 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 allows 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. 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.

Figure 38: SLA@SOI testbed in Intel IT Innovation Centre Showcase area
8 Conclusions

Substantial contributions have been made by the Infrastructure Management work package in this the second year of the project.

Efforts in infrastructure modeling and interfaces have contributed to the OCCI standard, now released to the public, and a representative of this work package is playing a leadership role as co-chair of this standard group.

A complete SLA-enabled infrastructure layer has been developed as per the updated architecture, dedicated Infrastructure SLA Manager, Infrastructure Service Manager and Infrastructure monitoring successfully implemented and integrated.

The Infrastructure SLA Manager includes a reference Infrastructure POC and Infrastructure PAC, supporting SLAs that include both single and multiple VM provisioning requests, with example functional and non-functional parameters.

The Infrastructure Service Manager has an OCCI-compliant interface, is built upon the Apache TASHI provisioning system, and has included initial enhancements to TASHI that we have the opportunity to contribute back to the community.

The Infrastructure Monitoring components deliver all the functionality needed to configure, deploy, monitor, gather, process, publish and manage the monitoring needs of the offered Infrastructure SLAs. They have been integrated into Tashi, built on top of Ganglia, and are managed via an innovative distributed management topology.

Important future work remains however. In the coming months the important milestone of going open source will be reached, opening up the consortium to new ideas and possibly new approaches. The collaboration with RESERVOIR will provide a very useful and hopefully powerful validation of the SLA@SOI models, architecture and reference implementations developed to date. The potential drafting of an SLA-aware Distributed Computing Infrastructure protocol could provide for a formal standard that directly leverages the core of the project. However, perhaps most importantly, the deployment and evaluation of the framework in each of the four use cases as well as the Open Reference Case will deliver comprehensive feedback on our solution in a variety of grounded deployments. The Infrastructure Management work package partners look forward to our SLA-enabled infrastructure layer delivering real value, solving real problems in the real world.
9 References


[2] SLA@SOI Annex I - "Description of Work"


[41] RDFa API - An API for extracting structured data from Web documents, B. Adrian, M. Sporny, M. Birbeck, available at http://www.w3.org/TR/rdfa-api/

[48] OCCI Implementer and Integrator Guide, A. Edmonds, available at https://docs.google.com/Doc?docid=0AS0AExvzzYt7YWQ2enFodmt6czIfMzRNm25mMnJmZA&hl=en
[57] Spring, available at http://www.springsource.org
Appendix A: Glossary

The following list shows the most important entries of the SLA@SOI glossary.

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.

Infrastructure Service: An infrastructure service is a specific IT service which exposes resource/hardware-centric capabilities.

Internal Service: Internal services are exposed within the boundaries of an organization, i.e. within one administrative domain.

IT Service: 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.

Offered Service: An abstract service (more precisely: service type) which is offered by a specific Service Provider to its Service Customers.

Operation Level Agreement: A specification of the conditions under which an internal service or a component is to be used by its “customer”.

Service: A means of delivering value to customers by facilitating outcomes customers want to achieve without the ownership of specific costs and risks.

Service Concreteness: The stage a service reaches over time from a fully abstract type to actually instantiated.

Service Consumer: Person(s) who actually consume/use the provided services. Typically they belong to the service customer.

Service Customer: Someone (person or group) who orders/buys services and defines and agrees the service level targets.

Service Description Term: 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.

Service Exposure: Services can be exposed either internally (within the same administrative domain) or externally.

Service Implementation: A service implementation is a possible concrete realization of a given service type.

Service Instance: 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.

Service Interface Type: Describes the nature of an actually exposed service, i.e. about the nature of his invocation interface.

Service Level Consequence: An action that takes place in the event that a service level objective is not met.

Service Level Agreement: 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 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

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMQP</td>
<td>Advanced Message Queuing Protocol</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect Oriented Programming</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BM</td>
<td>Business Manager</td>
</tr>
<tr>
<td>B-SLAM</td>
<td>Business SLA Manager</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DMZ</td>
<td>Demilitarised Zone</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modelling Framework</td>
</tr>
<tr>
<td>ERP</td>
<td>Enterprise Resource Planning</td>
</tr>
<tr>
<td>IAM</td>
<td>Infrastructure Allocation and Management</td>
</tr>
<tr>
<td>IE</td>
<td>Interaction Event</td>
</tr>
<tr>
<td>IL</td>
<td>Infrastructure Landscape</td>
</tr>
<tr>
<td>FCR</td>
<td>Finite capacity regions</td>
</tr>
<tr>
<td>FP</td>
<td>Functional Property</td>
</tr>
<tr>
<td>GSLAM</td>
<td>Generic SLA Manager</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IAM</td>
<td>Infrastructure Allocation and Management</td>
</tr>
<tr>
<td>IPAC</td>
<td>Infrastructure Provisioning and Adjustment Component</td>
</tr>
<tr>
<td>IFOC</td>
<td>Infrastructure Planning and Optimisation Component</td>
</tr>
<tr>
<td>ISLAM</td>
<td>Infrastructure SLA Manager</td>
</tr>
<tr>
<td>ISM</td>
<td>Infrastructure Service Manager</td>
</tr>
<tr>
<td>IoC</td>
<td>Inversion of Control</td>
</tr>
<tr>
<td>KPI</td>
<td>Key Performance Indicator</td>
</tr>
<tr>
<td>LLMS</td>
<td>Low Level Monitoring System</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queueing Networks</td>
</tr>
<tr>
<td>MA</td>
<td>Manageability Agent</td>
</tr>
<tr>
<td>MM</td>
<td>Monitoring Manager</td>
</tr>
<tr>
<td>MRE</td>
<td>Monitoring Result Event</td>
</tr>
<tr>
<td>MVC</td>
<td>Model View Controller</td>
</tr>
<tr>
<td>NFP</td>
<td>Non-functional property</td>
</tr>
<tr>
<td>OCCI</td>
<td>Open Cloud Computing Interface</td>
</tr>
<tr>
<td>OGF</td>
<td>Open Grid Forum</td>
</tr>
<tr>
<td>ORC</td>
<td>Open Reference Case</td>
</tr>
<tr>
<td>OVF</td>
<td>Open Virtualization Format</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>QPN</td>
<td>Queueing Petri Nets</td>
</tr>
<tr>
<td>PAC</td>
<td>Provisioning and Adjustment Component</td>
</tr>
<tr>
<td>POC</td>
<td>Planning and Optimization Component</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Objects</td>
</tr>
<tr>
<td>PS</td>
<td>Provisioning System</td>
</tr>
<tr>
<td>RCG</td>
<td>Reasoning Component Gateway</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SE</td>
<td>Service Evaluation</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SLAM</td>
<td>SLA Manager</td>
</tr>
<tr>
<td>SLAT</td>
<td>Service Level Agreement Template</td>
</tr>
<tr>
<td>SLMS</td>
<td>SLA Level Monitoring System</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>SLO</td>
<td>Service Level Objective</td>
</tr>
<tr>
<td>SM</td>
<td>Service Manager</td>
</tr>
<tr>
<td>SME</td>
<td>Small and Medium-sized Enterprise</td>
</tr>
<tr>
<td>SOA</td>
<td>Service Oriented Architecture</td>
</tr>
<tr>
<td>SOI</td>
<td>Service Oriented Infrastructure</td>
</tr>
<tr>
<td>SW-SLAM</td>
<td>Software SLA Manager</td>
</tr>
<tr>
<td>SW-SM</td>
<td>Software Service Manager</td>
</tr>
<tr>
<td>TCO</td>
<td>Total Cost of Ownership</td>
</tr>
<tr>
<td>TOGAF</td>
<td>The Open Group Architecture Framework</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>WP</td>
<td>Work-package</td>
</tr>
<tr>
<td>XMPP</td>
<td>Extensible Messaging and Presence Protocol</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language</td>
</tr>
</tbody>
</table>
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 ------------------------------------------*/

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

    // gslam_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
}
guaranteed_state{
    id = VM_ISOLATION
    isolation(VM_1) = VM_ISOLATION_VAR
}
guaranteed_state{
    id = VM_LOCATION
    location(VM_1) = VM_LOCATION_VAR
}
guaranteed_state{
    id = VM_CORES
    vm_cores(VM_1) = VM_CORES_VAR
}
guaranteed_state{
    id = VM_CPU_SPEED
    cpu_speed(VM_1) = VM_CPU_SPEED_VAR
}
guaranteed_state{
    id = VM_MEMORY_SIZE
    memory(VM_1) = VM_MEMORY_SIZE_VAR
}
guaranteed_state{
    id = VM_PERSISTENCE
    persistence(VM_1) = VM_PERSISTENCE_VAR
}
guaranteed_state{
    id = VM_IMAGE
    vm_image(VM_1) = VM_IMAGE_VAR
}
}