D.A4.a SLA-Aware Infrastructure Management

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

Due date of deliverable: 31st May 2009
Actual submission to EC date: 31st May 2009

Start date of project: 1st June 2008
Duration: 36 months

Lead contractor for this deliverable: INTEL
Revision: V.1.0 (28th May 2009)
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>Samuel Kounev, FZI</td>
</tr>
<tr>
<td>Reviewer 2</td>
<td>Luciano Baresi, PMI</td>
</tr>
<tr>
<td>PMT Reviewer</td>
<td>Ramin Yahyapur, UDO</td>
</tr>
<tr>
<td>Complete version submitted to reviewers</td>
<td>May 11, 2009</td>
</tr>
<tr>
<td>Comments of reviewer 1 received</td>
<td>May 14, 2009</td>
</tr>
<tr>
<td>Comments of reviewer 2 received</td>
<td>May 15, 2009</td>
</tr>
<tr>
<td>Revised deliverable submitted to PMT</td>
<td>May 25, 2009</td>
</tr>
<tr>
<td>PMT Approval</td>
<td>May 28, 2009</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 Molino, Jacek Maza</td>
</tr>
<tr>
<td>XLAB</td>
<td>Primoz Hadalin</td>
</tr>
<tr>
<td>BeSC</td>
<td>Yih Leong Sun</td>
</tr>
<tr>
<td>UDO</td>
<td>Costas Kotsokalis</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 2009 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](http://creativecommons.org/licenses/by/3.0/).

### Document History

<table>
<thead>
<tr>
<th>Version</th>
<th>Date</th>
<th>Author</th>
<th>Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>20th April 2009</td>
<td>John Kennedy</td>
<td>Initial draft / collation of all inputs</td>
</tr>
<tr>
<td>0.2</td>
<td>25th May 2009</td>
<td>John Kennedy</td>
<td>Various updates based on internal review feedback</td>
</tr>
<tr>
<td>1.0</td>
<td>28th May 2009</td>
<td>John Kennedy</td>
<td>Various updates based on internal PMT review feedback</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 over the first 12 months of the project.

Achievements include the completion of a comprehensive state-of-the-art review, together with the gathering of requirements, both implicit and explicit. A flexible, scalable architecture for Infrastructure Management has been defined, and a reference implementation created for demonstration purposes. An internet-accessible testbed has been deployed, and is used to host various components of the prototype. Software development best-practices have been promoted, and proliferated across the project.

Technical highlights including defining flexible external and internal models, documenting a harmonized interface, realising a distributed agent-based architecture, implementing a highly scalable XMPP-based messaging layer and realising powerful, distributed monitoring. Already this work is beginning to influence external standards initiatives: feedback from Infrastructure Management blog posts have ultimately leaded to the receipt of an invitation to co-chair the newly-formed OVF working group Open Cloud Computing Interface (OCCI).

With SLAs at its core, SLA@SOI Infrastructure Management has a focus unique to the industry. As the cloud computing movement matures, this work package will strengthen its engagement with the community, promoting SLA-aware service oriented infrastructures, helping to realise the vision of the project.
# Table of Contents

1  Introduction ........................................................................................................... 8  
  1.1  Context .............................................................................................................. 8  
  1.2  Scope .................................................................................................................. 9  
  1.3  Overall Process and Approach ......................................................................... 10  
  1.4  About this Document ......................................................................................... 11  
2  State of the Art ..................................................................................................... 12  
  2.1  Introduction ........................................................................................................ 12  
  2.2  Infrastructure Models ....................................................................................... 12  
  2.3  Infrastructure Virtualization ............................................................................ 13  
  2.4  Infrastructure Platforms ................................................................................... 14  
  2.5  Infrastructure Monitoring ................................................................................ 14  
  2.6  Infrastructure Messaging ................................................................................ 15  
3  Requirements ....................................................................................................... 16  
  3.1  High Level Requirements ............................................................................... 16  
  3.2  Infrastructure SLA Requirements ................................................................... 16  
  3.3  Interface and Management Requirements ..................................................... 17  
  3.4  Monitoring Requirements .............................................................................. 17  
  3.5  Use Case Requirements .................................................................................. 18  
4  Architecture ......................................................................................................... 19  
  4.1  Overall Architecture ....................................................................................... 19  
  4.2  Negotiation ....................................................................................................... 27  
  4.3  Agent-Based Architecture .............................................................................. 28  
  4.4  Provisioning ...................................................................................................... 29  
  4.5  Monitoring ....................................................................................................... 30  
  4.6  Adjustment and Reprovisioning .................................................................... 34  
  4.7  Application Framework .................................................................................. 35  
5  Models .................................................................................................................. 36  
  5.1  Introduction ...................................................................................................... 36  
  5.2  Infrastructure Model ....................................................................................... 36  
  5.3  Resource States ............................................................................................... 43  
  5.4  Future Enhancements ...................................................................................... 43  
6  Implementation .................................................................................................... 44  
  6.1  Introduction ...................................................................................................... 44  
  6.2  Infrastructure interface .................................................................................... 45  
  6.3  Integration infrastructure class ....................................................................... 46  
  6.4  common.messaging ......................................................................................... 48  
  6.5  common.agent ................................................................................................. 52  
  6.6  infrastructure.management ............................................................................... 54  
  6.7  infrastructure.model ......................................................................................... 56  
  6.8  infrastructure.webservice ............................................................................... 57  
  6.9  infrastructure.webservice-proxy ...................................................................... 58  
  6.10 infrastructure.planner .................................................................................... 61  
  6.11 infrastructure.provisioning ............................................................................ 63  
  6.12 infrastructure.landscape ............................................................................... 68  
  6.13 infrastructure.resource ................................................................................ 71  
  6.14 infrastructure.monitoring ............................................................................ 76  
  6.15 Quality Assurance ......................................................................................... 80  
7  Reference Deployment .......................................................................................... 81  
  7.1  Introduction ...................................................................................................... 81  
  7.2  Testbed Infrastructure ..................................................................................... 81  
  7.3  Virtualisation Platform .................................................................................. 82  
  7.4  Default Operating System Services ................................................................ 83  
  7.5  Network Configuration .................................................................................. 83
7.6 Storage Management ............................................................. 84
7.7 Open Reference Case Images .................................................. 85
7.8 Automation and Management ................................................ 85
8 Conclusions................................................................................. 87
8.1 Summary................................................................................ 87
8.2 Outlook on Future Work......................................................... 88
9 References................................................................................ 89
Appendix A. Glossary ................................................................. 91
Appendix B. Abbreviations .......................................................... 92
Appendix C. Review of IaaS API Data Schemas .............................. 93
  Scope .................................................................................. 93
  Amazon EC2 ......................................................................... 93
  Sun Cloud API ..................................................................... 94
  Flexiscale ........................................................................... 96
  ElasticHosts ........................................................................ 97
  GoGrid ............................................................................... 98
  Enomaly .............................................................................. 99
  OpenNebula ....................................................................... 100
  SliceHost .......................................................................... 102
  Apache Tashi ...................................................................... 103
  Eucalyptus, Globus Nimbus .................................................. 103
  AppNexus .......................................................................... 103
Appendix D. IaaS API Analysis .................................................... 104
  Scope ................................................................................ 104
  Amazon EC2 ....................................................................... 104
  The Sun Cloud API ............................................................ 105
  Flexiscale .......................................................................... 107
  ElasticHosts ...................................................................... 108
  GoGrid .............................................................................. 109
  Enomaly - Enomalism .......................................................... 110
  OpenNebula ..................................................................... 112
  Slicehost .......................................................................... 113
  Globus Nimbus .................................................................. 114
  Eucalyptus ........................................................................ 114
  AppNexus .......................................................................... 114
  F5.com .............................................................................. 114
  Apache Tashi (Incubator) ...................................................... 115
  CohesiveFT ....................................................................... 116
  Providers not considered ...................................................... 117
Appendix E. Open Virtualisation Format Model ............................. 118
Table of Figures

Figure 1: Conceptual Architecture of SLA@SOI .......................................................... 8
Figure 2: A Cloud Computing Stack – 2009 (Sam Johnston) .................................. 9
Figure 3: Initial Conceptual Architecture .................................................................. 19
Figure 4: SLA Negotiation Components ................................................................... 20
Figure 5: SLA Infrastructure Realisation Components ........................................... 20
Figure 6: SLA Enforcement Components .................................................................. 21
Figure 7: Overall Negotiation and Provisioning-time Functional View .................... 21
Figure 8: Overall Run time Functional View ................................................................ 22
Figure 9: High-level Infrastructure Management Architecture ............................... 23
Figure 10: Expanded Infrastructure Management Architecture ............................. 24
Figure 11: Current Infrastructure Management Architecture .................................... 25
Figure 12: Provisioning Architecture ........................................................................ 30
Figure 13: Monitoring Architecture .......................................................................... 32
Figure 14: Impact of Live Migration on Network Performance .............................. 35
Figure 15: Infrastructure Meta-model ....................................................................... 37
Figure 16: Infrastructure Model ............................................................................... 37
Figure 17: Infrastructure Landscape Meta-model .................................................... 41
Figure 18: Infrastructure Landscape Model ............................................................... 42
Figure 19: Resource State Diagram .......................................................................... 43
Figure 20: High-level design of the Infrastructure Management layer .................... 45
Figure 21: Infrastructure Interface Class Diagram ................................................... 46
Figure 22: Integration Infrastructure Class Diagram ................................................ 46
Figure 23: XMPP PubSub Class Diagram .................................................................. 49
Figure 24: PubSubMessage Class Diagram .............................................................. 50
Figure 25: XMPP Messaging Class Diagram ............................................................ 51
Figure 26: Agent Class Diagram ............................................................................... 52
Figure 27: Task Class Diagram ................................................................................ 53
Figure 28: Infrastructure Management Class Diagram ............................................. 54
Figure 29: Internal Infrastructure Data Model ......................................................... 56
Figure 30: Infrastructure Webservice Class Diagram .............................................. 57
Figure 31: Webservice Layers .................................................................................. 57
Figure 32: Webservice Endpoint Class Diagram ..................................................... 58
Figure 33: Infrastructure Webservice Proxy Class Diagram .................................... 58
Figure 34: Webservice Proxy Layers ........................................................................ 59
Figure 35: Webservice Client Class Diagram ............................................................ 59
Figure 36: Deployment Planner Class Diagram ....................................................... 61
Figure 37: Overall Provisioning Sequence Diagram ................................................ 62
Figure 38: Provider Manager Class Diagram ............................................................ 63
Figure 39: Register Physical Resource Sequence Diagram ..................................... 63
Figure 40: Infrastructure Provisioning Sequence Diagram .................................... 65
Figure 41: Provider Management Listeners .............................................................. 66
Figure 42: Provisioning Job Management Class Diagram ........................................ 66
Figure 43: Infrastructure Landscape Data Model ...................................................... 68
Figure 44: Infrastructure Landscape Class Diagram ................................................ 69
Figure 45: Resource Agent Class Diagram ............................................................... 72
Figure 46: Information gathered by Monitoring ...................................................... 73
Figure 47: Contents of the Infrastructure Landscape ................................................. 74
Figure 48: Monitoring Agents Architecture ............................................................... 76
Figure 49: Monitoring a Virtual Machine ................................................................. 80
Figure 50: SLA@SOI testbed in Intel IT Innovation Centre Showcase area ............. 82
Figure 51: Network Configuration of a Virtual Machine ......................................... 84
Figure 52: Open Reference Case deployed across three VMs ............................... 85
Figure 53: Simultaneous Machine Management ..................................................... 86
List of Tables

Table 1: Explicit Use Case Requirements .......................................................... 18
Table 2: Request Parameters ........................................................................ 38
Table 3: Response Parameters ..................................................................... 38
Table 4: Internal Parameters ...................................................................... 40
Table 5: Infrastructure Landscape Methods ............................................... 69
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.

![Conceptual Architecture of SLA@SOI](image)

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 [1]. 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 an SOA. The framework manages the resource issues associated with the specification, negotiation, fulfilment and enforcement of SLAs. With respect to a modern cloud computing stack, Infrastructure Management’s efforts are concentrated in the red layer in Sam Johnston’s 2009 Cloud Computing Stack [2] as reproduced in Figure 2.

![Figure 2: A Cloud Computing Stack – 2009 (Sam Johnston)](image)
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 will also provide test bed resources together with workload synthesis tools for this and other relevant work packages, supporting in particular the SLA@SOI Adhoc Demonstrator and appropriate SLA@SOI Industrial Use Cases.

### 1.3 Overall Process and Approach

Year 1 progress in the Infrastructure Management work package has 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 [3], as well as via more direct channels such as position papers and collaboration events. These efforts helped initiate dialog with the external cloud computing community, forming relationships such as bringing SLA@SOI onboard for the launch of the OGF[8] Open Cloud Computing Interface Working Group, a group which we have just recently been invited to co-chair.

1.4 About this Document

This document describes the progress towards constructing the SLA-aware Infrastructure layer in the first 12 months of the SLA@SOI project. The document progresses through the traditional software life-cycle. The next section, Section 2, State of the Art, describes important current technologies together with the latest research considered relevant to SLA-aware infrastructure.

Section 3, Requirements, then describes the requirements foreseen for this layer, taking both the general industrial and research views and expectations into account as well as the explicit requirements emerging from the various SLA@SOI Use Cases.

Section 4, Architecture, describes the evolution of the Architecture of the Infrastructure Management layer, taking the requirements into account.

Section 5, Design and Implementation, documents the actual design and implementation of the architecture as it exists at Month 12 of the project.

Section 6, Reference Deployment, explains some of the more practical details of the initial ad-hoc demonstrator, including the automating of the images and the testbed on which it is hosted.

Section 7, Quality Assurance, details the measures taken towards producing the highest quality software. This included promoting an integrated build system, providing tools to track unit testing, and encouraging coding-style consistency.

Finally, Section 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 State of the Art

2.1 Introduction

This section presents a summary of an extensive State of the Art review that was performed into various aspects of infrastructure management for Service Oriented Infrastructures. As SOIs can be considered synonymous with the emerging Cloud Computing paradigm, this analysis includes many references to current Cloud Computing offerings and research initiatives.

To help structure this analysis, this section is organised into the following key subsections:

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

To avoid repetition, this section does not go into detail on any particular technology, platform or product. For detailed reviews refer to Appendices C and D for details of various Infrastructure as a Service (IaaS) data models and APIs respectively, and to the overall SLA@SOI Scientific Evaluation Report [4].

2.2 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 were Common Information Model (CIM) [5] issued by the Distributed Management Task Force (DMTF) [6], Open Virtual Machine Format (OVF) [7], and Open Grid Forum (OGF) [8] GLUE 2.0.

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 Appendix E.

OGF GLUE 2.0 is a specification, in draft at the time of writing, 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 Appendix C. General requirements for the SLA@SOI data models that were revealed by this analysis are documented in Section 3.

In summary, it was observed that no current or draft model currently exists that meets all the needs of SLA@SOI. OVF is 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, 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.

### 2.3 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 is 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 [9] is an open source API that provides a generic way to interact with different types of open source virtualization technologies (including Xen [10] and KVM [11]). 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 [12] 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.

### 2.4 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 in Appendix D a review of the APIs of the major infrastructure providers has been undertaken. It includes Amazon EC2, Sun Cloud API, Flexiscale, ElasticHosts, GoGrid, Enomaly – Enomalism, OpenNebula, Slicehost, Globus Numbus, Eucalyptus AppNexus, F5.com, Apache Tashi and CohesiveFt are provided in Appendix D.

This review of the IaaS platform providers has helped identify key generic requirements of the API that SLA@SOI Infrastructure Management should expose, as well as gain an understanding of some of the explicit implementations that SLA@SOI should be able to integrate with. It has helped inform the decisions made during the design and implementation phases of the project and whilst this analysis focused on the API and functional offerings of these infrastructure providers, it was interesting to note that none of the platforms offered comprehensive SLA-awareness.

### 2.5 Infrastructure Monitoring

A key part of any SLA-aware system is the monitoring of the performance of the system – without it there is no awareness. The state-of-the-art analysis for monitoring technologies involved a review of several key frameworks and technologies in this area including EVEREST, Ganglia, Nagios, Groundwork, MonALISA and Zabbix. Although Nagios, Groudwork, MonALISA and Zabbix all have interesting aspects, the two most relevant frameworks by far were identified to be EVEREST [13] and Ganglia [14].

EVEREST, an Event Reasoning Toolkit developed by CITY University, one of the SLA@SOI partners, is a flexible monitoring engine solution built on Event Calculus that is being developed and extended into the backbone for overall monitoring of SLAs in SLA@SOI.

However, 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.
2.6 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 [15], 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 [16] and IO-Data XEPs [17]. 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 [18], 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 [19]. 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 [20], 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. It is still in active development – version 0.9 is the latest stable version and implementations have not yet received widespread acceptance and adoption.

Based on the analysis it was determined that XMPP offered the most appropriate messaging solution for the time being. However, with the maturing of AMQP the messaging layer in SLA@SOI should be implemented in such a fashion that the messaging protocol can be easily replaced.
3 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 [1] 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. These implicit requirements have been the main driver for the architecture, implementation and reference deployment in the first year of the project.

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

3.1 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 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 resulting framework must not mandate the use of particular operating systems, middleware, hypervisors or external infrastructure providers. Generic interfaces and abstractions should be implemented where possible.
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.

3.2 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

### 3.3 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.

### 3.4 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.

### 3.5 Use Case Requirements

The SLA@SOI project includes five Use Case work packages dedicated to
championing the SLA@SOI project in realistic deployment scenarios. These use
cases span ERP Hosting, Enterprise IT, Service Aggregation in the

The B1 work package on Business Requirements has gathered formal explicit
requirements for the SLA@SOI framework from these use cases, and the current
list of requirements assigned to Infrastructure Management is listed in Table 1.

<table>
<thead>
<tr>
<th>#</th>
<th>Description</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>Dynamic / unforeseen re-allocations should be possible</td>
<td>blocker</td>
</tr>
<tr>
<td>85</td>
<td>Monitoring metrics must include memory / processor performance, response time, failure rate</td>
<td>blocker</td>
</tr>
<tr>
<td>86</td>
<td>Aggregate monitoring metrics must be supported</td>
<td>blocker</td>
</tr>
<tr>
<td>97</td>
<td>Hardware virtualization (VMware Server, Xen) must be available</td>
<td>blocker</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>
<td>blocker</td>
</tr>
<tr>
<td>190</td>
<td>Customers must be able to select basic or higher compute unit.</td>
<td>blocker</td>
</tr>
<tr>
<td>191</td>
<td>Customer must be able to request for additional data storage (other than the basic compute unit).</td>
<td>Blocker</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>
<td>blocker</td>
</tr>
<tr>
<td>106</td>
<td>Java Management Extensions (JMX) must be supported as a management technology</td>
<td>critical</td>
</tr>
<tr>
<td>192</td>
<td>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).</td>
<td>critical</td>
</tr>
<tr>
<td>202</td>
<td>In particular, infrastructure resources might need negotiation as a complete landscape (i.e. a structured layout of related infrastructure resources)</td>
<td>critical</td>
</tr>
<tr>
<td>94</td>
<td>SLA Applications must support event-based or some other asynchronous communication involved</td>
<td>minor</td>
</tr>
</tbody>
</table>

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 [4].
4 Architecture

4.1 Overall Architecture

The architecture for the software in Infrastructure Management evolved as requirements emerged.

From implicit requirements available at the start of the project, a bottoms-up architecture was constructed. It identified three core responsibilities: SLA Negotiation, SLA Infrastructure Realisation and SLA Enforcement, connected as illustrated in Figure 3.

Each of these responsibilities was broken down further into constituent parts:

- **SLA Negotiation** requiring SLA Infrastructure Mapping and a Runtime Predictor as illustrated in Figure 4.
- **SLA Infrastructure Realisation** requiring SLA Translation, Infrastructure Deployment, Providers and Event Monitoring as illustrated in Figure 5.
- **SLA Enforcement** requiring Event Correlation, Autonomic Predictive Management and SLA Monitoring as illustrated in Figure 6.
Figure 4: SLA Negotiation Components

Figure 5: SLA Infrastructure Realisation Components
In parallel with this internal effort, a project-wide top-down architecture was emerging as illustrated in Figure 6 (Negotiation and Provisioning time) and Figure 7 (Run time).
Extracting the SLA Negotiation responsibilities (being managed and implemented by Work package 5), introducing a message bus to allow the monitoring system communicate with external components, and merging the bottoms-up (Figure 3 to Figure 6) and top-down (Figure 7 and Figure 8) architectures already introduced resulted in a merged high level Infrastructure Management Architecture as illustrated in Figure 9.
This merged architecture continued to have three core functional blocks.

**Infrastructure SLA Provisioning and Negotiation** includes all functionality that is required prior to actual resource allocation. Its responsibilities include the need to translate from potentially high-level Infrastructure Requirements (e.g. in the form of business-level SLA terms) to technical requirements; it needs to map the resulting technical requirements to appropriate bundles of resources, and it needs to verify the actual deployment is possible based on the infrastructure actually available. These steps are all carried out as part of an Infrastructure Negotiation Workflow, after which the requestor may or may not decide to proceed with the deployment. If the decision is made to proceed, the Resource Allocation and Management functionality is then invoked.

**Resource Allocation and Management** performs the actual provisioning. Hardware, either physical or virtual, is provisioned via a Provider Management Interface. This provides an abstract interface into arbitrary internal and external infrastructure providers as may be appropriate. It also supports the provisioning of required infrastructural software, if appropriate. This functional component also includes an Autonomic Management service to identify if and when reprovisioning should occur to best accommodate any over-arching business rules or infrastructure policies that may be defined, or SLA conditions that are in danger of being violated.

Once infrastructure is provisioned, appropriate event logging, correlation and monitoring is instantiated in the **Infrastructure Reporting** component to allow any relevant conditions or events to be identified and escalated, either internally back to Infrastructure SLA Provisioning and Negotiation for immediate disposition,
or externally to the party that has requested this infrastructure in the first place, in the case of for example the unavoidable violation of a SLA.

Introducing these various discrete functions into the diagram gives rise to the architecture in Figure 10.

![Expanded Infrastructure Management Architecture](image)

**Figure 10: Expanded Infrastructure Management Architecture**

As scalability and messaging aspects were investigated, this architecture was further refined into the current infrastructure management architecture illustrated in Figure 11.
The responsibilities of the various components, their associated tasks and the partners involved in their development can be summarised as follows.

<table>
<thead>
<tr>
<th>Component</th>
<th>Responsibilities</th>
<th>Task</th>
<th>Partners</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Infrastructure Manager</strong></td>
<td>Provides interfaces for Registration, Redeployment, Provisioning and Management.</td>
<td>Not explicit</td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Abstract Infrastructure Framework Model</strong></td>
<td>An abstract model for describing required infrastructure resources. It can accommodate infrastructure constraints defined in higher-level business SLAs.</td>
<td>A4.1</td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Infrastructure SLA Translation</strong></td>
<td>Converts provisioning request into an Abstract Infrastructure Framework Representation. Could potentially support requests in multiple formats.</td>
<td>A4.7 (input)</td>
<td>UDO, Intel</td>
</tr>
<tr>
<td><strong>Infrastructure Deployment Planner</strong></td>
<td>Analyses requests and converts them into individual VM requirements and corresponding software images. Checks if these resources can be provisioned. Reserves these resources for a short duration.</td>
<td>A4.5</td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Infrastructure Negotiation Workflow</strong></td>
<td>This workflow uses the Infrastructure SLA Translation and Infrastructure Deployment Planner</td>
<td>A4.7</td>
<td>UDO</td>
</tr>
</tbody>
</table>

**Figure 11: Current Infrastructure Management Architecture**
to see if resources can be provisioned. The customer may or may not decide to proceed with the provisioning.

<table>
<thead>
<tr>
<th><strong>Provider Management</strong></th>
<th>Provides a plug-in management system for communicating and controlling resource providers using a consistent abstracted interface. Performs the provisioning and re-provisioning as required.</th>
<th>A4.2 (Interface)</th>
<th>BeSC, Intel</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A4.3 (Implementation)</td>
<td></td>
</tr>
<tr>
<td><strong>Internal Providers</strong></td>
<td>Resource providers completely hosted by the Infrastructure Provider - arbitrary monitoring and support services are supported.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>A4.3</td>
<td>BeSC</td>
</tr>
<tr>
<td><strong>External Providers</strong></td>
<td>3rd party resource providers which may not be able to support arbitrary monitoring requirements.</td>
<td></td>
<td>BeSC</td>
</tr>
<tr>
<td><strong>Autonomic Management (Optimisation)</strong></td>
<td>Requests the Deployment Planner to perform redeployment pre-emptively based on potential SLA violations identified by Monitoring.</td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Monitoring</strong></td>
<td>Receives events from internal or external resource providers, standardises them and stores them in a historical repository. Reviews the historical repository, correlates raw events, identifies escalations including potential and actual SLA violations, and forwards them to subscribers. It will store this information in an historical repository and may expose it depending on the SLA.</td>
<td></td>
<td>XLAB</td>
</tr>
<tr>
<td><strong>Prediction Services</strong></td>
<td>Will be used by Infrastructure Deployment Planner to predict the actual resources required based on historical and any other information available.</td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Testbed</strong></td>
<td>Physical and virtual infrastructure on which all development SLA@SOI software can be deployed and tested</td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td><strong>Infrastructure Landscape</strong></td>
<td>A representation of all physical and virtual infrastructure currently running that is under the control of the infrastructure provider. All attributes of registered infrastructure can be queried. Physical and virtual infrastructure must be registered here upon activation.</td>
<td></td>
<td>Intel</td>
</tr>
</tbody>
</table>

Whilst this subsection has summarised the overall Infrastructure Management architecture, there were several key technical and implementation decisions taken that at this phase of the project that affect the architecture in various ways. The
areas concerned were Negotiation, Agent-Based Architectures, Provisioning, Messaging and Management, and are discussed in some more detail in the remainder of this section.

### 4.2 Negotiation

As elaborated further in Deliverable D.A5a, it was decided that negotiation for infrastructure resources would best take place in the “Negotiation” module being implemented by Work package 5, SLA Management and Foundations, for Year 1. This allowed us to decouple the concern of provisioning infrastructure resources from that of negotiating for those resources. The fact that negotiation in SLA@SOI in this first year is following a one-off model, without counter-offers, assists in applying this to the infrastructure and the current infrastructure architecture.

Within the Negotiation module, infrastructure appears just as any other service offering SLA functionality. There is a template for reserving resources (i.e. establishing agreements), which is customised to size the infrastructure provisioning request according to application requirements. In WS-Agreement notation, a simple template looks like the following:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsag:Template wsag:TemplateId="SLAT_Infrastructure_Service_v1"
    xmlns:terms="http://www.slaatsoi.org/commonTerms"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:coremodel="http://www.slaatsoi.org/coremodel"
    xmlns:wsag="http://schemas.ggf.org/graap/2007/03/ws-agreement"
    xsi:schemaLocation="http://schemas.ggf.org/graap/2007/03/ws-agreement_types.xsd">
    <wsag:Name>SLAT_Infrastructure_Service</wsag:Name>
    <wsag:Context>
        <wsag:ServiceProvider>AgreementResponder</wsag:ServiceProvider>
    </wsag:Context>
    <wsag:Terms>
        <wsag:All>
            <wsag: GuaranteeTerm wsag:Name="InfrastructureGuarantee" wsag:Obligated="ServiceProvider">
                <wsag:ServiceLevelObjective>
                    <wsag: KPITarget>
                        <wsag: KPIName>ProvisioningRequest</wsag:KPIName>
                        <wsag: CustomServiceLevel>
                            <terms: ProvisionRequestType>
                                <resource xsi:type="terms:VirtualMachine">
                                    <appliance diskImage="DB Image ID"/>
                                    <component xsi:type="terms:CPUComponent">
                                        <parameter xsi:type="terms:CPUSpeed">
                                            <frequency unit="GHz" value="2.0"/>
                                        </parameter>
                                        <parameter xsi:type="terms:NumCPUs">
                                            <number long="4"/>
                                        </parameter>
                                    </component>
                                    <component xsi:type="terms:MemoryComponent">
                                        <parameter xsi:type="terms:Memory">
                                            <size unit="GB" value="4.0"/>
                                        </parameter>
                                    </component>
                                </resource>
                                <resource xsi:type="terms:VirtualMachine">
                                    <appliance diskImage="App Server Image ID"/>
                                    <component xsi:type="terms:CPUComponent">
                                        <parameter xsi:type="terms:CPUSpeed">
                                            <frequency unit="GHz" value="1.0"/>
                                        </parameter>
                                    </component>
                                </resource>
                            </terms: ProvisionRequestType>
                        </wsag:CustomServiceLevel>
                    </wsag: KPITarget>
                    </wsag:ServiceLevelObjective>
                </wsag: GuaranteeTerm>
            </wsag:All>
        </wsag:Terms>
</wsag:Template>
```
This template defines the necessary constructs, based on the SLA@SOI core model, without setting any constraints on the requested values for the infrastructure. The template includes default values which would typically be modified as part of an agreement offer. It is up to the infrastructure planner to decide whether the values submitted as part of an agreement offer (that was produced based on this template) are acceptable or not.

This offer would not reach the infrastructure in this format though. The Negotiation module would extract the necessary terms (which, in this case, is straightforward as we only have a single Service Level Objective (SLO) to model the complete agreement) and submit it as an argument to the provisioning request. Depending on accepting or rejecting this request, a SLA would be established or the offer would be similarly rejected.

In multi-round negotiation, where counter-offers may exist, the infrastructure will need to provide reasonable alternatives to provisioning requests which cannot be satisfied. This may take place by adjusting some of the terms in the request, by adding extra terms, or by removing some of the existing ones. As an example, one may consider the case of a provisioning request of 2 VMs with CPUSpeed set to 10 GHz (the Negotiation module does not filter the requests based on their semantics, but rather only on the constraints of the templates used). This request would of course fail due to unrealistic CPUSpeed requirement, and it would be up to the infrastructure to facilitate negotiation by replying with a counter-offer (e.g. 2 VMs with CPUSpeed = 1.5 GHz), instead of just rejecting the request.

### 4.3 Agent-Based Architecture

As the requirements for the Infrastructure Management layer emerged, and the architecture evolved, it became clear that the system as a whole needed to be highly scalable. Given the distinct responsibilities of the various components within the architecture, it became clear that an agent-based approach may be very suitable.

Wooldrige describes an agent as “an encapsulated computer system that is situated in some environment and that is capable of flexible, autonomous action in that environment in order to meet its design objectives.” [21]
Implementing various Infrastructure Management components as autonomous agents, giving them certain goals that react to inputs based on the current state of the agent and their environment, would allow for a massively distributed system, decentralized and highly-scalable.

It was decided to implement a generic agent-based architecture, where an abstract agent could be assigned custom tasks, e.g. to monitor particular events, to manage a physical resource or to host an instance of the Infrastructure Landscape. By using a shared agent platform, generic activities such as starting, stopping, and upgrading the agent could be implemented just once.

To allow agents communicate, a generic cross module communications channel would be required. For ultimate flexibility it was decide to integrate a messaging bus into the architecture. To allow communications to route to individual agents, and to distinguish between arbitrary message types, it was observed that assigning a message-bus channel per message type was appropriate [22].

4.4 Provisioning

Architecturally, Infrastructure Management needs to be able to interact with arbitrary sources of infrastructural resources. These may be local installed physical machines, which may or may not support some form of virtualisation. Alternatively, the infrastructure may be provided by remote providers such as Amazon EC2. An extendable plugin architecture has been adopted to support this broad range of internal and external providers.

The Provider Management component provides this plugin management system for communicating with and controlling resource providers using a consistent abstracted interface. Each resource, either local or remote, is manipulated via an appropriate Resource Agent. The resource agent is customised for the type of resource being controlled, be it a local hypervisor, or a remote infrastructure provider. To ensure scalability, the Provider Management communicates with the Resource Agents via a messaging bus.

This architecture is illustrated in Figure 12.
Initially, Provider Management will expose the following functionality to the Deployment Planner:

1. **Find** – to find suitable resources that can satisfy the resource requirements. The Deployment Planner will pass the resource requirements to Provider Management in this function. Provider Management will search the Infrastructure Landscape and return a list of suitable resources to Deployment Planner.

2. **Reserve** – to reserve a (typically) virtual resource on a physical resource. The Deployment Planner selects one physical resource from the list found in the “Find” operation. The Deployment Planner will instruct Provider Management to reserve the virtual resource with the appropriate VM image name. The Deployment Planner will make multiple calls to Provider Management if multiple copies of virtual resources with a different VM image name are required. To maintain the infrastructure provisioning request state, an Infrastructure Identifier will be passed between the Deployment Planner and Provider Management. Provider Management will ‘mark’ the resources on which resources are reserved with a short-term lease.

3. **Instantiate** – to instantiate and start the virtual resource. After the virtual resource is reserved successfully on the physical resource, the Deployment Planner will instruct Provider Management to instantiate (i.e. start) the virtual resource(s). The Deployment Planner will make multiple calls to Provider Management if multiple copies of the virtual resources are required.

4. **Stop** – to stop the infrastructure provisioning. The Deployment Planner will instruct Provider Management to stop all virtual machines associated with an infrastructure provisioning request.

### 4.5 Monitoring

An essential part of an SLA-aware infrastructure is a scalable and self-sufficient monitoring system capable of monitoring large distributed systems, in real-time. The monitoring system must support two mutually exclusive perspectives arising from the Service Level Agreement, namely the customer’s perspective and the
infrastructure/service provider’s perspective. The former is interested in the SLA alone, while the latter needs to be able to optimise the utilisation of the infrastructure.

To help process and manage the volume and variety of monitoring data, a multi-layer monitoring architecture has been proposed by Infrastructure Management.

### 4.5.1 Multi-layer monitoring architecture

The distributed multi-layer monitoring architecture may be comprised of as many layers as necessary to support the monitoring of the underlying infrastructure. However, these layers have been divided into three logical layers, according to their primary purpose, amount of input and output events, and degree of processing. The lowest layer of the hierarchy, the data collection layer (L0), is mainly used for the collection of raw input data. Basic filtering and pre-processing of collected information can also be applied at L0 to reduce network traffic. However, processing on Layer 0 should be kept to a minimum to limit the monitoring resource usage.

The second logical layer is the event evaluation layer (L1) that supports the integration of monitors into a cascade of increasingly more complex monitors, ranging from simple metric checks to composed monitors. Composed monitors re-use other monitoring agents to process complex rules, e.g. monitoring of an entire cluster, taking the relationship between nodes in a cluster into account.

The top-most layer, named the service layer (L2), configures as well as defines the meaning of monitoring events generated in lower layers of the architecture. The architecture prevents top-level monitors from connecting to data collection layer and bypassing the event evaluation layer. L2 layer is a collection of conceptually similar functions that provide services used by any service dealing with infrastructure and receives inputs from layers below it.
The following subsections describe the logical layers of the architecture in further detail.

### 4.5.2 Data collection layer (L0)

Layer 0 represents the data-collecting monitoring agents producing low-level and (mostly) unprocessed events. These agents are wrappers around specialised data collectors, like Ganglia 4 or Munin 4 at the infrastructure layer. They can even use probes into Virtual Machine Monitor (xentop), /proc, kernel, or middleware components (web server, application server, database).

Data collection monitors support three types of operation, timed push, conditional push and pull. Timed push publishes configured metrics in uniformly timed intervals regardless of the value change since the last published value. Agents support an arbitrary number of timed pushes, i.e., different metrics can be requested at different intervals. Conditional push provides a basic mechanism for the reduction of network load. Metric values are pushed on the channel only when the last published value is exceeded by specified threshold value or percentage. Agents from higher layers can also opt to query metrics from data collectors only when needed in their own calculations. For this purpose, L0 agents also support...
metrics to be pulled on request. Each agent can be configured to work in one or more of these modes simultaneously.

4.5.3 Evaluation layer (L1)

The event evaluation layer is a dynamic network of distributed agents. A dedicated infrastructure node may be used to deploy these agents to reduce the overhead of nodes offered to customers/users. Agents are all subscribed to a single configuration channel to which monitoring requests are published by service layer monitors. Monitoring requests are represented as rules that L1 monitors are expected to validate. Every L1 monitor can verify whether it supports validation of requested rules and whether it has sufficient resources to accept additional monitoring. Agents willing to start monitoring solicited rules notify the requester that, based on these responses, decides which monitor to send configuration to. It is also possible to select several monitors for the same validation rule.

4.5.4 Service layer (L2)

Every request for monitoring has to enter through one or more service layer (L2) monitors that form the boundary of the entire monitoring architecture. These monitors’ activities are composed according to the users’ requests. Each L2 monitor implies a certain configurations of the L1 monitors, which are managed dynamically. Users of L2 monitors may be infrastructure providers, service providers or even service customers requiring immediate notification that the agreement was breached. Events from L1 monitors are used as triggers for execution of required actions.

Infrastructure Warning/Violation events are issued by L2 layer and are known as an InfrastructureMonitoringEventType defined by the monitoring XSD schema.

Examples of service layer tasks are:

- **Auditing task** - logging and monitoring customer’s usage of resources.
- **Accounting task** - producing information used for billing.
- **Autonomous management** - providing self re-organisation of the infrastructure in order to improve the utilization without breaking any SLA.
- **Notification task** - may be used to notify various stakeholders (service provider, customer) of a broken rule.
- **Historical information repository task** - is used to store various monitoring information, ranging from raw data about infrastructure and/or services to events raised by different monitors.

4.5.5 Monitoring Virtual Machines

In SLA@SOI there is a basic need to be able to monitor provisioned virtual machines. There are several potential sources for this L0 data, but each has its own advantages and disadvantages.

One is to monitor the hypervisor hosting the virtual machines, but this only allows the data exposed by the hypervisor to be monitored. Different hypervisors expose different data.
A second approach is to run a generic instrumentation framework like Ganglia inside each virtual machine, and the resource agent on the hypervisor communicates to it via the virtual network. This requires the customer to install, configure and run Ganglia inside their virtual machine, something that will consume some of their resources.

The third approach is to install a complete SLA@SOI specific resource monitoring agent including instrumentation framework inside each virtual machine. This would allow maximum control over what is monitored, and where it is passed to, but also places maximum inconvenience on the customer, and consumes more of their virtual machine than the previous approach.

In Year 1 the second approach has been pursued. This is described in more detail in Section 6.14.

### 4.6 Adjustment and Reprovisioning

A key benefit of having an Infrastructure Management layer is being able to adjust and reprovision the infrastructure as required. This may be following a request from the customer, or following some internal analysis and detection of an opportunity for consolidation or avoiding an SLA violation.

The type of adjustment and reprovisioning supported depends on the type of infrastructure technologies being used, and the architecture of the application or service being hosted. Typical scenarios include ‘imaging’ a virtual machine and booting it up in an alternative virtual machine, perhaps on different hardware; adjusting aspects of the virtual machine, e.g. the CPU allocation, dynamically; and live migration, where a virtual machine can be transferred from one physical machine to another, without any downtime for the users. If the application is partitioned according to the Model/View/Controller approach, reprovisioning could simply require the instantiation of additional Views (web servers) or Models (database servers) depending on which component is under the most stress. Scalr [23] is an example of such an “elastic” architecture implemented on top of Amazon EC2.

#### 4.6.1 Live Migration

Live migration is a particularly interesting form of reprovisioning as it requires no downtime for the customer. For example, if the Autonomic Management component detects an opportunity for consolidation of services without affecting customer SLAs, live migration could be used to relocate the virtual machines seamlessly, if the providers being used support live migration.

However, live migration does not come for free and can temporarily reduce the performance of hosted services during the migration process. The time required to live-migrate can be significant, depending on size of the virtual machine and network bandwidth.

Some experimentation has been undertaken in order to profile the extra impact that a live migration can have within the running time of virtual machines and the infrastructure. An example of live-migration is illustrated in Figure 14.

From a host machine running a virtual image, a command can be issued indicating which virtual machine to migrate, and to where (e.g. “xm migrate...”, see top right of the screenshot). Assuming that the user doing the migration has
the correct credentials, the machine will be migrated. If the configuration of the systems is correct, live migration will take as long as the memory of the running virtual machine is transferred from one system to the other. In our example, a virtual machine with 512Megs of memory was migrated in approximately 49 seconds (48.895 seconds) over a 100 Mbps network with almost no visible impact on the running services inside the virtual machine.

In Figure 14 it is also possible to see in the impact of the live migration on the network. The “Network History” plot on the left hand side of the image illustrates that the available uplink bandwidth on one of the network interfaces has been maxed-out during the migration.

![Figure 14: Impact of Live Migration on Network Performance](image)

### 4.7 Application Framework

From a software point of view, following cross-workpackage deliberations it was decided to adopt Spring [24] as the application framework for all relevant SLA@SOI components. Spring provides a consistent development model for software components, allowing cross-cutting concerns such as security, logging, configuration and deployment options to be manipulated dynamically via changes to XML-based Spring configuration files, rather than source code changes requiring recompilation, repackaging and reinstallation.
5 Models

5.1 Introduction

Significant attention has been paid to modeling within Infrastructure Management as models can allow a shared understanding across multiple components, both internal to Infrastructure Management and external. This effort has been performed as part of Task TA4.1, where an abstracted representation or model of infrastructure based on the underlying fine-grained and heterogeneous resources, be they virtualised or not, is being defined.

After some investigation it was identified that two models were appropriate: one, the Infrastructure model, to model infrastructure from an external point of view, the other, the Landscape model, for internal management purposes. This separation allowed the internal model to be refined without affecting the simpler and hence more stable external model being used by external components.

Central to both models is the concept of resource.

Resources are the pieces of infrastructure that Infrastructure Management can provision for customers. Examples of resources include:

- **Compute Resources** - providing users of the system a means to perform tasks such as computation, and execute business logic
- **Network Resources** - providing a service that allows entities such as storage and compute resource communicate effectively and in a scalable way.
- **Storage Resources** - providing a service that allows entities such as the compute resource persist data in a performant, secure and reliable way

Both models were built around the concept of a resource, designed to accommodate the requirements described in Section 3. These requirements reflected the learning’s from the state of the art analysis (Section 2) and review of Data Schemas and APIs from external Infrastructure as a Service providers (Appendices C & D respectively). It should be noted that a review of the internal data models used by third partners was not always possible as this information is often deemed sensitive. What was apparent, however, was that the internal models were typically a superset of the external models.

5.2 Infrastructure Model

The Infrastructure Model, designed for external use, provides enough richness to allow provisioning requests to be constructed, details of infrastructure provisioned to be extracted, and reprovisioning requests to be specified. It has been realised by extending the core SLA@SOI model. Thus all modules and components of SLA@SOI can create and navigate data based on the Infrastructure Model.

The basic Infrastructure meta-model is illustrated in Figure 15, and the current version of the Infrastructure model in Figure 16.
A customer's request for infrastructure is encapsulated by the infrastructure entity. This is a group in terms of the meta-model and allows a number of functional and non-functional parameters to be associated with the request. The infrastructure entity can then contain a number of resources. These resources, in year one, are compute resources (i.e., virtual machines), and have an appliance (i.e., pre-packaged application) associated with them.
In order to define the required attributes (non-functional parameters and functional parameters) of a compute resource entity, it can be associated with an arbitrary amount of parameters. Common functional parameters include CPU speed, installed memory and disk capacity whilst non-functional parameters represent the contextual attributes of physical and virtual machines. These would include geographic location, isolation level, security concerns and availability required. The particular attributes that have been implemented in year one are listed in the following table.

**Table 2: Request Parameters**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Occurrence</th>
<th>Permission</th>
<th>Resource</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>Infrastructure (reference)</td>
<td>0...*</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>architecture</td>
<td>string</td>
<td>0...1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td>not implemented for requests</td>
</tr>
<tr>
<td>cpuNum</td>
<td>int</td>
<td>0...1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>cpuSpeed</td>
<td>double</td>
<td>0...1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td>in GHz</td>
</tr>
<tr>
<td>mem_total</td>
<td>double</td>
<td>0...1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
</tbody>
</table>

For consistency, the Infrastructure model is also used when customers request details of the infrastructure already provisioned for them. In this case what is returned to the customer contains more details that the original request for the respective infrastructure. The attributes that can be returned for each resource in such a response include the following.

**Table 3: Response Parameters**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Occurrence</th>
<th>Permission</th>
<th>Resource</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>boottime</td>
<td>string</td>
<td>1</td>
<td>read-only</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>cpuIndex</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual &amp; Physical</td>
<td>Calculated as: (cpuSpeed * cpuNum)</td>
</tr>
<tr>
<td>ipAddress</td>
<td>string</td>
<td>1</td>
<td>read-only</td>
<td>Virtual &amp; Physical</td>
<td>TODO: should support more than 1 IP address per resource.</td>
</tr>
<tr>
<td>osRelease</td>
<td>string</td>
<td>1</td>
<td>read-only</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>timeStamp</td>
<td>string</td>
<td>1</td>
<td>read-only</td>
<td>Virtual &amp; Physical</td>
<td>when this model was last updated</td>
</tr>
<tr>
<td>Metric</td>
<td>Type</td>
<td>Size</td>
<td>Access</td>
<td>Type 1</td>
<td>Type 2</td>
</tr>
<tr>
<td>--------------</td>
<td>----------</td>
<td>------</td>
<td>--------</td>
<td>-----------------</td>
<td>------------</td>
</tr>
<tr>
<td>cpu_user</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>cpu_idle</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>cpu_system</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>cpu_wio</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>mem_buffers</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>mem_shared</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>mem_cached</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>mem_free</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>load_one</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>load_fifteen</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>load_five</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>disk_free</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>disk_total</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>diskUsage</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>swap_total</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>swap_free</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>swapUsage</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td>Physical</td>
</tr>
<tr>
<td>pkts_in</td>
<td>double</td>
<td>1</td>
<td>read-only</td>
<td>Virtual</td>
<td></td>
</tr>
</tbody>
</table>
There are also a number of resource parameters that are tracked but are only used and are viewable by internal infrastructure management. Those parameters are as follows.

**Table 4: Internal Parameters**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Occurrence</th>
<th>Permission</th>
<th>Resource</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtualResources</td>
<td>VirtualResource (reference)</td>
<td>0...*</td>
<td>read/write</td>
<td>Physical</td>
<td>Calculated as: (cpuSpeed * cpuNum)</td>
</tr>
<tr>
<td>physical</td>
<td>boolean</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>hypervisor</td>
<td>string</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td>TODO: should support more than 1 IP address per resource.</td>
</tr>
<tr>
<td>hypervisorURI</td>
<td>string</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>hypervisorVersion</td>
<td>string</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>macAddress</td>
<td>string</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>jabberID</td>
<td>string</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
<tr>
<td>machineType</td>
<td>string</td>
<td>1</td>
<td>read/write</td>
<td>Virtual &amp; Physical</td>
<td></td>
</tr>
</tbody>
</table>

**5.2.1 Infrastructure Landscape Model**

The infrastructure landscape records and manages all running and live provider resources. These include all physical resources, whether they are hosting anything for customers or not, together with all virtual resources that have been provisioned.

Again, a resource may not just be a physical server. It could be any form of infrastructure, and may be part of a hierarchy. Collections of servers could be
organised into clusters, if appropriate, or racks and data-centres, if that is how they need to be managed. The infrastructure landscape model allows flexibility in how resources can be organized into collections. Where collection type entities have additional attributes (e.g. rack location, geographic location etc.), these can be used help automate the grouping should that be appropriate.

Whilst the Infrastructure Landscape stores the state of all infrastructure, it does not capture real-time monitoring data. As this is very dynamic in nature, this information is stored separately in the historical information database. This helps avoid any queues of update that may arrive in bulk at the infrastructure landscape.

For scalability reasons, ideally management of each resource entry in the infrastructure landscape should be managed by the respective resource. Furthermore, a distributed landscape would help share the load. Whether it makes sense to decentralise the infrastructure landscape is not yet clear. This will be investigated within the remainder of the project.

The Infrastructure Landscape Meta-model and Model are illustrated in Figure 17 and Figure 18 respectively.
When customer requests are provisioned these are recorded in the infrastructure as either virtual or physical resources depending on what the deployment planner deems the most suitable based on the customer requirements. The Infrastructure entity is again included in this model as a way to referring to a customer’s provisioned resources. Effectively the whole executed infrastructure request model is recorded in the infrastructure landscape, the only core difference being that the type of resource type is particular to the type of machine; physical or virtual. These core entities both can have a number of non-functional and functional parameters associated with them (see Table 2, Table 3 & Table 4 for a list of currently supported parameters). A physical resource, unlike the virtual one, has a link to a number of appliances that can possibly be hosted by a hypervisor. A virtual resource only has a link to one as virtual resources can only run one appliance at any one time. These entities (physical/virtual) are represented as resources in the infrastructure landscape meta-model, the various parameter types as parameters and the infrastructure entity as a group. It is important to link the customers (an owner in the meta-model) infrastructural SLA (iSLA) to the group of compute resources contained by the infrastructure entity. This is done via the iSLA entity, a SLA entity in the meta-model.

Lastly, all resources can be grouped into provider specific clusters and then those clusters grouped into domains. Each of these administrative grouping constructs are of the group type in the meta-model. Each of these group types can be associated with a number of non-functional and functional parameters. Examples of these would include rack identifier and geographic location.

### 5.2.2 Model Integration

At the time of writing the Infrastructure model is relatively stable has been integrated into the Core SLA@SOI Model. The Infrastructure Landscape model continues to evolve, and this model is stand-alone for the time being.
Other models have been created in other work packages, and in the next phase of the project it is hoped that some or all of these models can be integrated, at least to some extent. In particular it could be very useful if the Infrastructure Landscape model could refer to elements of the Software Landscape, or vice versa, allowing a mapping between software services and the individual virtual machines on which they are provisioned to be maintained.

Although SLA@SOI uses its own models for now, it is intended that standard models such as OVF will be supported if not natively then via plugin adapter layers. OVF does not yet support non-functional parameters, so extensions to OVF will need to be proposed before this can be fully realised. Infrastructure Management is now actively engaged in the OVF OCCI working group [25] and will look to pursue this when appropriate.

### 5.3 Resource States

To help monitor resources, an exercise in defining the possible states of resources was carried out by SLA@SOI Infrastructure Management in cooperation with the OGF’s OCCI working group [25]. This has resulted in the following state model of a resource being defined.

![Resource State Diagram](image-url)

**Figure 19: Resource State Diagram**

### 5.4 Future Enhancements

To date we have defined an external request model that is SLA@SOI-specific but as we also drive to the want of adopting standards we will aim to support OVF requests sent to the external A4 infrastructure provisioning interface. Future plans include:

- supporting storage and networking resources
- allowing for arbitrary customer groupings
- integrating with the software landscape (work package A3), using their software and service concepts in infrastructure models
- influencing DMTF through suggested enhancements to OVF
- integrating with the core meta model
- decentralising the infrastructure landscape
6 Implementation

6.1 Introduction

Whilst the previous sections have described an architecture, interfaces and models to support the requirements identified for the Infrastructure Management layer of SLA@SOI, this section focuses on the actual implementation of the components of the infrastructure layer that have been developed by month 12 of the project.

These components include:

- **Infrastructure interface** – defining the interface into the Infrastructure Management layer.
- **Integration infrastructure class** – the Spring-based class that wraps the Infrastructure Management layer for the overall SLA@SOI Adhoc Demonstrator.
- **common.messaging** – a generic messaging framework and utility classes for use by all components.
- **common.agent** – a generic agent framework and utility classes for use by all components.
- **infrastructure.management** – a layer that wraps the implementation of the infrastructure layer, providing a single point of entry to the functionality.
- **infrastructure.model** – a data-model for use inside the Infrastructure Management layer.
- **infrastructure.webservice** – a webservice wrapper on top of infrastructure.management, for users that wish to communicate with Infrastructure Management through the SOAP protocol.
- **infrastructure.webservice-proxy** – a proxy for the webservice interface to infrastructure.management, allowing users to simply code against a Plain Old Java Object (POJO). The POJO translates any invocations into their SOAP equivalent and invokes them in turn against the webservice.
- **infrastructure.planner** – a module that performs initial planning for any provisioning that is required.
- **infrastructure.provisioning** – a module responsible for provisioning (or reprovisioning) requests.
- **infrastructure.landscape** – a persistent data store that maintains the run-time state of all infrastructure, and infrastructure requests.
- **infrastructure.resource** – an agent that resides on and controls a particular physical resource.
- **infrastructure.monitoring** – a comprehensive monitoring framework that can monitor and respond to arbitrary states and occurrences within the infrastructure layer.

These modules and their interrelationships are illustrated in Figure 20 below. Both Monitoring and Resource are implemented as agents. A typical deployment would have multiple Resources – one per each physical resource.
The implementation of these components is now described in some detail. For more complete information, please refer to the appropriate JavaDoc documentation.

### 6.2 Infrastructure interface

The Infrastructure Interface defines the external interface into the Infrastructure Management layer. The interface is located in a dedicated module so that the implementation does not have to be distributed with the interface itself. This allows customers program against the interface, binding with the actual implementation only at runtime.
Four methods have been defined for the first year of the project:

Provision() receives a provision request, and returns a provision response, typically a unique infrastructure ID assuming the provision request was satisfied successfully.

Reprovision() is similar to provision(), but receives the infrastructure ID of an existing deployment as part of its parameters. This method attempts to alter or else replace this existing deployment with the details of the new provision request.

Stop() is provided for convenience purposes, allowing all provisioned virtual machines associated with a provision request to be stopped, and the infrastructure be reclaimed.

Finally, getDetails() returns the complete details surrounding a provisioned request corresponding to a particular infrastructure ID. This allows a customer to identify or confirm the resources assigned to a current request.

### 6.3 Integration infrastructure class

The Infrastructure Integration component delegates all calls to the wired infrastructure implementation. It is provided as a wrapper class within the Spring-based Integration module to assist in the integration of infrastructure with the other modules as part of the ad-hoc demonstrator.

A sample Spring configuration file for the Integration infrastructure class is presented below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:tx="http://www.springframework.org/schema/tx"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
```
<bean id="providerManagement"
class="org.slasoi.infrastructure.provisioning.ProviderManager">
  <constructor-arg value="pmp-configuration.xml"/>
  <property name="infrastructureLandscape" ref="landscape"/>
</bean>

<bean id="deploymentPlanner"
class="org.slasoi.infrastructure.planning.DeploymentPlannerImpl">
  <property name="providerManagement" ref="providerManagement"/>
</bean>

<bean id="entityManagerFactory"
class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="dataSource"/>
  <property name="jpaVendorAdapter">
    <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
      <property name="database" value="HSQL"/>
      <property name="showSql" value="true"/>
      <property name="generateDdl" value="true"/>
    </bean>
  </property>
</bean>

<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:mem:slasoi"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="landscapeDao"
class="org.slasoi.infrastructure.landscape.LandscapeDaoJPAImpl">
  <property name="entityManagerFactory" ref="entityManagerFactory"/>
</bean>

<bean id="landscape"
class="org.slasoi.infrastructure.landscape.LandscapeJPAImpl">
  <property name="landscapeDao" ref="landscapeDao"/>
</bean>

<bean id="infrastructureProxy"
class="org.slasoi.infrastructure.management.InfrastructureImpl">
  <property name="deploymentPlanner" ref="deploymentPlanner"/>
</bean>
6.4 common.messaging

As messaging in SLA@SOI has been implemented using the publish / subscribe pattern, the term pubsub is used extensively throughout this subsection.

6.4.1 Realisation

Because messaging can be realised with many different technologies e.g. XMPP [15], JMS [18], AMQP [20] and so on, the **factory creational design pattern** is used to ensure support for different technologies through a unified interface, and to allow easy switching between these technologies.

Currently both XMPP and JMS have been implemented though XMPP is the protocol generally used. In the future the implementation of AMQP will also be considered because is it known for its reliability.

6.4.2 Messaging interface

The main class used for messaging is **PubSubManager** - it exposes the interface. The implementation of the concrete PubSubManager has to implement these methods which basically represent the interface:

- connect() - connects to the pubsub server
- getId() - returns the ID of the connection
- createChannel(Channel) - creates a new pubsub channel to which messages are published
- isChannel(String) - return true if channel exists
- deleteChannel(String) - deletes the pubsub channel
- publish(PubSubMessage) - publishes the message
- subscribe(String) - subscribes to the pubsub channel
- unsubscribe(String) - unsubscribes from the channel
- close() - closes the connection with the pubsub server
6.4.3 Pubsub message

The class that encapsulates the message being sent over pubsub is PubSubMessage. The most notable properties of PubSubMessage are the channel to which the message is published and the payload which contains the actual content.
Messages get serialized before they are sent over pubsub, and are deserialized after arrival. The encoding strategies currently supported are Base64 and XML.

### 6.4.4 Point to point messaging

Besides the pubsub type of messaging, point to point messaging is also supported. It can be used for direct communication between two end points. The interface is as follows:

- `sendMessage(Message)` - sends a message
- `addMessageListener(MessageListener)` - adds a message listener
- `getAddress()` - returns the address of the user
- `close()` - closes the connection

Currently XMPP point to point messaging is supported.
6.4.5 Example Usage of PubSubManager

A typical usage of PubSubManager would be to first define the settings needed to connect to the pubsub server. This includes user name, password, server address and so forth. Settings can be read from a properties file with these properties:

```java
#PubSub engine
pubsub=xmpp

# XMPP
xmpp_username=user1
xmpp_password=pass
xmpp_host=192.168.0.23
xmpp_port=5222
xmpp_service=virtuoz
xmpp_resource=user1
xmpp_pubsubservice=pubsub.virtuoz
```

The second step is to create an instance of PubSubManager using the factory:

```java
PubSubManager pubSubManager = PubSubFactory
    .createPubSubManager("user1.properties");
```

After that we can create and subscribe to channels, publish messages and add message listeners.

```java
// Add message listener.
pubSubManager.addMessageListener(new MessageListener() {
    public void processMessage(MessageEvent messageEvent) {
        System.out.println("Message received: "+
                        messageEvent.getMessage().getPayload());
    }
});
```
6.5 common.agent

6.5.1 Agent

Agent is essentially a thread pool of tasks. It also implements the Runnable interface so it can be executed in its own thread.

Agent has an instance of PubSubManager (explained in Section 6.4, common.messaging) which connects the agent to the messaging bus. Every agent is subscribed to the agent configuration channel from which it receives configurations as well as commands e.g. to shutdown, restart, etc. Another use of the agent configuration channel is to receive tasks to run.

When the agent is started it reads the configuration from the properties file. The configuration must contain the agent configuration channel, messaging settings and optional tasks configurations. The latter is used to add tasks to the agent. Agent ID is defined by the messaging connection ID.

Figure 26: Agent Class Diagram
Agent uses an extended version of PubSubMessage with the following additional properties:

- **MessageType** (agent configuration, data, ...)
- **Sender Agent ID**
- **Receiver Agent ID**
- **Sender Task ID**
- **Receiver Task ID**

### 6.5.2 Task

Task encapsulates a specific functionality. It's an abstract class with the following interface:

- `configure(TaksConfiguration)`: configures the task with specified the configuration
- `createInstance()`: creates an instance of the task
- `run()`: runs the task
- `shutdown()`: properly shuts down the task

When created it receives an instance of an IPublisher object which wraps PubSubManager functionality which enables the task to be connected to the messaging bus.

![Figure 27: Task Class Diagram](image-url)
6.6 infrastructure.management

The Infrastructure Management module acts as an entry point to the infrastructure subsystem. It implements the Infrastructure interface and thus exposes all implemented operations, i.e. provision, reprovision, stop and getDetails.

Figure 28: Infrastructure Management Class Diagram

The **provision(ProvisionRequestType)** method generates a unique request identifier (infrastructureID), generates host names for each resource (virtual machine) and converts the request into the infrastructure model form. Internally, the request, along with the infrastructureID and host names, is passed down to the Deployment Planner for further processing.

The **reprovision(string, ProvisionRequestType)** method alters (or replaces) virtual machines that have already been provisioned. The incoming provision request needs to contain the resource IDs of the virtual resources to be modified. The IDs are required to identify virtual resources in the landscape, and are retrieved by calling getDetails().

The **stop(String)** method stops all virtual machines provisioned as part of the supplied infrastructureID. The Infrastructure Management module calls the Provider Manager and passes infrastructureID accordingly.

The **getDetails(string)** method retrieves the provision request object from the landscape and returns it in the Infrastructural (coremodel) form. The result looks similar to the initial provision request but also includes the unique IDs of the various virtual resources provisioned.

A sample Spring configuration file for the Infrastructure Management component is presented on the following page.
6.6.1 Example client application

The sample code below creates a request to provision a resource with the following requirements:

Request
  location: Germany
  Resource1
    • CPU speed: 2GHz
    • CPU number: 4
    • memory: 4GB
    • image name: DB Services Image

```java
CoremodelFactory coremodelFactory = CoremodelFactory.eINSTANCE;
TermsFactory termsFactory = TermsFactory.eINSTANCE;

Location location = termsFactory.createLocation();
location.setCountry("Germany");

ProvisionRequestType provisionRequest =
  termsFactory.createProvisionRequestType();
provisionRequest.getParameter().add(location);

Frequency frequency = coremodelFactory.createFrequency();
frequency.setValue(2.0);
frequency.setUnit(FrequencyUnitKind.GHZ);
CPUSpeed cpuSpeed = termsFactory.createCPUSpeed();
cpuSpeed.setFrequency(frequency);

DataSize size = coremodelFactory.createDataSize();
size.setValue(4.0);
size.setUnit(DataSizeUnitKind.GB);
Memory memorySize = termsFactory.createMemory();
memorySize.setSize(size);

Number number = coremodelFactory.createNumber();
nrumber.setLong(4);
NumCPUs numberOfCPUs = termsFactory.createNumCPUs();
numberOfCPUs.setNumber(number);
```
Appliance appliance = termsFactory.createAppliance();
appliance.setDiskImage("DB Services Image");

CPUComponent cpu = termsFactory.createCPUComponent();
cpu.getParameter().add(cpuSpeed);
cpu.getParameter().add(numberOfCPUs);

MemoryComponent memory = termsFactory.createMemoryComponent();
memory.getParameter().add(memorySize);

VirtualMachine resource = termsFactory.createVirtualMachine();
resource.setAppliance(appliance);
resource.getComponent().add(cpu);
resource.getComponent().add(memory);

provisionRequest.getResource().add(resource);
...
ProvisionResponse response =
infrastructure.provision(provisionRequest);

6.7 infrastructure.model

The Infrastructure Model defines the data structures for information flowing between different components within infrastructure. It contains information about requests, available physical resources, and the association of which virtual resource is hosted by which physical resource.

Figure 29: Internal Infrastructure Data Model

Information such as CPU speed or memory size is stored as resource parameters. The parameters are strongly typed to allow operators such as '<', '>', 'contains(text)' etc to be used in search operations.
6.8 infrastructure.webservice

The Infrastructure Webservice module exposes the Infrastructure Management component as a web service. Together with the Infrastructure Webservice Proxy module it provides means to deploy the Infrastructure Management on a remote server but yet invoke it transparently from the client.

Figure 30: Infrastructure Webservice Class Diagram

The Infrastructure Webservice is implemented as a web module and can be deployed on any servlet 2.4 capable web container. The webservice endpoint implementation is built on top of the Spring Remoting module which allows exposing any Spring managed component as a web service via JAX-RPC.

The Infrastructure Webservice takes an incoming provision request which comes as xml data, deserializes it to the java object and passes it down to the infrastructure management implementation. Once the response java object is retrieved, it serializes the response object to equivalent xml and sends it back to the webservice client. All java-to-xml and xml-to-java operations are performed with the EMF API, so no changes in the code are required if the infrastructure model is updated.

Figure 31: Webservice Layers

The webservice endpoint class extends the Spring’s ServletEndpointSupport class.
As the InfrastructureEndpoint inherits spring support from its parent class, it is not a spring managed bean itself. It accesses the InfrastructureImpl class as follows:

```java
infrastructure = (Infrastructure)
applicationContext.getBean("infrastructureImpl");
```

Therefore, the InfrastructureImpl class must be defined as infrastructureImpl managed bean as follows:

```xml
<bean id="infrastructureImpl"
class="org.slasoi.infrastructure.management.InfrastructureImpl" />
```

### 6.9 `infrastructure.webservice-proxy`

The Infrastructure Webservice Proxy module provides the Infrastructure Management interface that refers to the implementation deployed on a remote server.
The Infrastructure Webservice Proxy wraps an Apache Axis webservice client that refers to the infrastructure endpoint and exposes the logic using the Infrastructure interface. It could be used as an alternative to direct bindings to the InfrastructureImpl implementation in a distributed deployment.

The Infrastructure Webservice Proxy takes an incoming provision request, serializes it to the xml and passes to the webservice client. Once the response xml is retrieved, it deserializes the xml to the response java object and returns it back to the client. All java-to-xml and xml-to-java operations are performed with EMF API, so no changes in the code are required if request/response model is updated.

![Figure 34: Webservice Proxy Layers](image1)

The class diagram for the webservice client is illustrated in Figure 34.

![Figure 35: Webservice Client Class Diagram](image2)
A sample Spring configuration file for the Webservice Client component is presented below:

```xml
<bean id="infrastructureEndpointServiceLocator" class="org.slasoi.infrastructure.webserviceclient.InfrastructureEndpointServiceLocator">
    <property name="infrastructureServiceEndpointAddress" value="http://remotehost/webservice/services/InfrastructureService"/>
</bean>

<bean id="infrastructureEndpoint" factory-bean="infrastructureEndpointServiceLocator" factory-method="getInfrastructureService"/>

<bean id="infrastructure" class="org.slasoi.infrastructure.webserviceclient.InfrastructureWebServiceProxy">
    <property name="infrastructureEndpoint" ref="infrastructureEndpoint"/>
</bean>
```
6.10 *infrastructure.planner*

### 6.10.1 Deployment Planner

In Year 1, the main responsibility of the Deployment Planner component is to process incoming provision requests and call the infrastructure provisioning module (Provider Manager) accordingly.

The class diagram of the Deployment Planner is illustrated in Figure 34.

![Deployment Planner Class Diagram](image)

**Figure 36: Deployment Planner Class Diagram**

The `doProvision(ProvisionRequest)` method determines whether requested resources could be provisioned, and if so, initiates provisioning. Firstly, the Infrastructure Landscape is searched for suitable resources. Secondly, the resources are reserved to be provisioned and finally the provision action is issued. Further details of how the Deployment Planner interacts with the Provider Manager are provided in the following subsection.
6.10.2 Interactions with Provider Manager

The onEventReceive method is called by a thread (via asynchronous callback) so it must be synchronized with the main thread. The current implementation uses the wait-notify approach to achieve this goal. The main thread calls the wait(timeout) method on a synchronization object so that it continues either the callback thread called notify(), or the timeout elapses.

A sample Spring configuration file for the Deployment Planner component is presented below:

```xml
<bean id="providerManagement" ...
...
</bean>

<bean id="deploymentPlanner"
class="org.slasoi.infrastructure.planning.DeploymentPlannerImpl">
  <property name="providerManagement" ref="providerManagement"
</bean>
```
6.11 infrastructure.provisioning

Provider Manager is the main class in the infrastructure provisioning module. This class implements the ProviderManagement interface, which defines the find, reserve, instantiate and stop operations.

During the initialisation of the Provider Manager, the Provider Manager subscribes to the relevant channels of the XMPP messaging bus.

```
ProviderManager

- find(resource : VirtualResource, params : PMParameters) : List<PhysicalResource>
- instantiate(virtualResource : VirtualResource, params : PMParameters) : void
- stop(params : PMParameters) : void

PMParameters
```

Figure 38: Provider Manager Class Diagram

The Resource Provider sends the physical resource registration message to the channel "Landscape". The Provider Manager picks up this message from the messaging bus and registers the resource to the Infrastructure Landscape.

```
: Physical Resource
: ProviderManager
: Messaging
: InfrastructureLandscape

1: send registration message
2: pick up registration message
3: addPhysicalResource(physicalResource)
```

Figure 39: Register Physical Resource Sequence Diagram

For each provisioning request, the Deployment Planner passes an Infrastructure Identifier along with other necessary parameters (e.g. image name, etc) to the Provider Manager. This Infrastructure Identifier is the unique global identifier for the infrastructure provisioning request.

On receiving the “Find” request from the Deployment Planner, the Provider Manager searches the Infrastructure Landscape and returns a list of suitable physical resources to the Deployment Planner.
The Deployment Planner chooses one of the physical resources from the list. On receiving the “Reserve” request from the Deployment Planner, the Provider Manager sends the “Reserve” command to the messaging bus with the relevant messenger ID of the selected physical resource. The physical resource picks up this command from the messaging bus and validates if it can provision the request. The physical resource then sends the result to the messaging bus. The Provider Manager picks up the “Reserve” result, updates the association relationship between the physical resource and the virtual resource in the Infrastructure Landscape, and returns the result to the Deployment Planner via the event callback handler.

On receiving the “Instantiate” request from the Deployment Planner, the Provider Manager sends the “Instantiate” command to the messaging bus with the relevant messenger ID of the physical resource associated with this virtual resource. The physical resource picks up this command from the messaging bus and starts the virtual resource. The physical resource then sends the result to the messaging bus. The Provider Manager picks up the “Instantiate” result, updates the Infrastructure Landscape, and returns the result to the Deployment Planner via the event callback handler.

On receiving the “Stop” request from the Deployment Planner, the Provider Manager retrieves all the relevant virtual resources from the Infrastructure Landscape. For each virtual resource provisioned, the Provider Manager sends the “Stop” command to the messaging bus with the relevant messenger ID of the physical resource associated with this virtual resource. The physical resource picks up this command from the messaging bus and stops the virtual resource. The physical resource then sends the result to the messaging bus. The Provider Manager picks up the “Stop” result, updates the Infrastructure Landscape, and returns the result to the Deployment Planner via the event callback handler.
Figure 40: Infrastructure Provisioning Sequence Diagram

To receive the result of each operation (i.e. reserve, instantiate, stop, etc), the Deployment Planner implements the Provider Management action listener class, PMActionListener, and subscribes to the Provider Management events.
6.11.1 Job Manager

The Provider Manager includes a Job Manager which is responsible for executing and monitoring each operation command (i.e. reserve, instantiate, stop). Each operation command is defined as an Action, which implements the Job interface.

This Job Manager is implemented as a Queue and Worker Pool mechanism. On initialisation, the Job Manager creates a list of Job Worker in the Worker Pool. Every new Job submitted is put into the Job Queue which is implemented as a Java BlockingQueue. A Job Supervisor is used to monitor the execution of every Job. The Job interface defines the execute, onCompleted and onTimeout methods. The Job Worker in the Worker Pool picks up the Job from the Job Queue and runs the job by calling the execute method. On completion (i.e. on receipt of result from the messaging bus), the onCompleted method is called. If the Job result is not received within a given time frame, the Job Supervisor expires the Job and calls the onTimeout method.

![Figure 42: Provisioning Job Management Class Diagram](image)
6.11.2 Example Code

To initialise the Provider Manager:

```java
PropertyConfigurator.configure("log4j/appender.properties");
String configurationFile = "pmp-configuration.xml";
ProviderManager provider = new ProviderManager(configurationFile);
```

To find a suitable physical resource that matches the SLA requirements:

```java
int infrastructureId; // the infrastructure identifier
String imageName; // the virtual machine image name

PMParameters params = new PMParameters();
params.setValue(ResourceCommandMessage.PARAM_NAME.INFRASTRUCTURE_ID.toString(), infrastructureId);
params.setValue(ResourceCommandMessage.PARAM_NAME.IMAGE_NAME.toString(), imageName);
List<PhysicalResource> physicalResources = provider.find(virtualResource, params);
```

To reserve a virtual resource:

```java
// pick a physical resource from the list
Random random = new Random(System.currentTimeMillis());
int index = random.nextInt(templates.size());
physicalResource = physicalResources.get(index);
provider.reserve(physicalResource, virtualResource, params);
```

To instantiate and start the virtual resource:

```java
provider.instantiate(virtualResource, params);
```

To stop the infrastructure provisioning:

```java
provider.stop(params);
```

To subscribe to the Provider Manager events:

```java
provider.subscribeEvent(new PMReserveActionListener() {
    public synchronized void onEventReceive(VirtualResource virtualResource, ActionParameters params) {
        //do necessary action here
    }
});

provider.subscribeEvent(new PMInstantiateActionListener() {
    public synchronized void onEventReceive(VirtualResource virtualResource, ActionParameters params) {
        //do necessary action here
    }
});
```
6.12 infrastructure.landscape

In Year 1 the infrastructure landscape will be implemented based on the Java Persistence API (JPA) [27]. The infrastructure landscape provides an interface that allows operations such as persisting, searching and updating entities defined in the Infrastructure Model (see Section 6.7). The Infrastructure Model is stored in the relational database according to the following schema.

Figure 43: Infrastructure Landscape Data Model
The implementation is based on a DAO pattern that allows changing the persistent storage mechanism in the future with a minimum of coding effort.

The infrastructure landscape interface exposes the following methods.

**Table 5: Infrastructure Landscape Methods**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;PhysicalResource&gt; getAllRegisteredResources()</td>
<td>returns list of all available physical resources</td>
</tr>
<tr>
<td>List&lt;PhysicalResource&gt; findPhysicalResources(double cpuSpeed, int cpuNum, int memorySize)</td>
<td>searches for physical resources which meets given criteria: cpu speed (GHz), number of CPUs, memory size (MB)</td>
</tr>
<tr>
<td>PhysicalResource addPhysicalResource(PhysicalResource physicalResource)</td>
<td>registers new physical resource</td>
</tr>
<tr>
<td>removePhysicalResource(String resourceGUID)</td>
<td>remove physical resource from the landscape</td>
</tr>
<tr>
<td>PhysicalResource addTargetPhysicalResource(VirtualResource virtualResource)</td>
<td>associates a virtual resource with given physical resource and updates physical resource attributes accordingly (amount of memory available etc)</td>
</tr>
<tr>
<td>PhysicalResource removeVirtualResource(targetPhysicalResource(VirtualResource virtualResource)</td>
<td>removes association between virtual resource and given physical resource and updates</td>
</tr>
</tbody>
</table>
Sample spring configuration based on hibernate [2] JPA provider and MySQL database.

A sample Spring configuration file based on a hibernate JPA provider [28] and a MySQL database [32] is presented below:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
         http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">
  <bean id="entityManagerFactory"
        class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />    
    <property name="jpaVendorAdapter">
      <bean
        class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
        <property name="database" value="MYSQL" />    
        <property name="showSql" value="false" />    
        <property name="generateDdl" value="true" />
      </bean>
    </property>
  </bean>
  <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://databasehost/databasename" />
    <property name="username" value="username" />
    <property name="password" value="password" />
</beans>
```
6.13 infrastructure.resource

Using the general principles and guidelines defined in the architecture section of this document and the more detailed description of the different abstraction layers and modules that a Physical Machine and a Virtual Machine are composed of, we have implemented a prototype that integrates the functionality of the Hardware Abstraction Layer (HAL) and all the necessary components to provide basic infrastructure provisioning capabilities.

This is a first implementation that has focused on integrating the basis for messaging, monitoring and provisioning of Virtual Machines. As this is an ongoing development process, it is important to note that the code and the architecture will be subject to change, refactoring and improvements over the coming months and years as the functionality gets refined; more advanced concepts are researched, evaluated and implemented; the state of the art updated and new requirements discovered.

6.13.1 Hypervisor Abstraction Layer Implementation

Figure 43 illustrates the initial class diagram for the Hardware Abstraction Layer (HAL). The HAL is mainly defined by the Provisioning interface. The main class that implements the Provisioning interface is ProvisioningAdapter. ProvisioningAdapter adapts the HAL provisioning interface to the specifics of different hypervisor technologies including the means of executing commands. The Driver interface is simply modeled as a typical Unix command, whereby input is given (in the forms of parameters) including the command type and the output is captured with ProvisioningAdapter configuring the subsystems of each virtualization technology during runtime.
PhysicalMachine has an instance of the ProvisioningAdapter and it is here where the provisioning commands are routed to the underlying implementation. PhysicalMachine also has a Monitor instance which at runtime is used to detect the type of hypervisor that PhysicalMachine has installed as well as all the functional properties of the machine such as CPU architecture/speed, total amount of memory, OS type and version, network interfaces and many more.
functional software and hardware properties.

Figure 44: Information gathered by Monitoring

Each Virtual Machine running under the Physical Machine host will take a “slice” of its resources such as CPU, storage, network bandwidth, and memory. Memory being the most constrained one as each memory allocation for a Virtual Machine is fixed during its lifetime. As memory is the least flexible resource when it comes to sharing in virtualization, there are some approaches to dynamically allocate/reallocate resources such as memory to virtual machines as needed to try to overcome these shortcomings. Progress in these techniques such as Transcendent memory will be monitored for their applicability to SLA@SOI Infrastructure Management.

CPU will be shared among Virtual Machines and the host Physical Machine by means of a combination of the host Physical Machine standard process scheduler (such as the Linux process scheduler [32]) and the scheduling policies applied by the hypervisor such as the approach of KVM of managing Virtual Machines as a normal user process [33] or XENs approach of replacing the host Physical Machine scheduler with its own scheduling approach [34], [35].

Another potential approach to sharing resources among virtual machines is to partition and configure the schedulers by means of only giving what the Virtual Machine needs (up to a level) by predicting its behaviour. An example of such approach is the use of Kalman filters to hypervisors CPU schedulers [36].
6.13.2 Agent, Tasks and Messaging

Once the Physical Machine has started, loaded all the required software/libraries and configured itself, it is ready to receive provisioning requests. Rather than the Physical Machine starting this process, it is the agent, by means of executing a Task to start up the Physical Machine thread. Following the agent-based architectural approach (see Section 4.3) the agent will be responsible for starting the Physical Machine and forwarding any messages (such as provisioning messages) to the Physical Machine.

Via the agent, it can also be possible to send messages to the Physical Machine to change its configuration, such as the rules for scheduling or the policy for provisioning during runtime.

One aspect of Physical Machine is that when its initialization is completed, a message should be sent to the Infrastructure Landscape to notify that it is ready to take provisioning requests. Any changes of state within Physical Machine will also be notified to the Infrastructure Landscape so that the Deployment Planner can take informed decisions on receipt of provisioning requests.

In a pool of hundreds of Physical Machines, the Deployment Planner will have to decide which Physical Machines are the potential candidates for deploying the different Virtual Machines according to the rules and policies of deployment, and also according to the prediction models implemented in work package task A6.3.

Figure 45 illustrates the contents of an instance of the Infrastructure Landscape (as a resource pool) with two Physical Machines registered (Resource 0 and 1). When an external command arrives, the Deployment Planner will select one or more Physical Machines that can potentially satisfy the request. For this first prototype, the policy for deployment is “Best Effort”.

![Figure 45: Contents of the Infrastructure Landscape](image)

With the Agent executing independent tasks without further dependencies, we are also pre-empting within the design the possibility that each task could be loaded or unloaded dynamically during runtime. As these agents and tasks will be
running potentially in hundreds or thousands of Physical Machines, a mechanism to control their lifecycle and provide updates including software updates with minimum downtime to the agents and the tasks during runtime is required. With the current architecture that separates agents from their tasks and the way they are managed, this functionality could be achieved by using a component based approach such as OSGI [37] to dynamically unload, update and reload the Agent and Tasks libraries.

6.13.3 The Leasing Approach to Resource Sharing

As described earlier each Virtual Machine has been modelled as a lease that takes resources off the Physical Machine and gives them back when it is finished with them.

There is a further aspect to the lease and it is of reservations. As we have a large number of Physical Machines that registers with the Infrastructure Landscape as provisioning targets, the management layers (Provider Management and Deployment Planner) use the Infrastructure Landscape to retrieve some of the information with regards to the state of the Physical Machines such as which ones have spare capacity (and requirements) to accommodate the requirements of Virtual Machines.

One shortcoming of the design of the asynchronous messaging based architecture is that there is no guarantee that Infrastructure Landscape is in a 100% “up-to-date” or consistent state. However, the Infrastructure Landscape could be, at some point in time, “Eventually Consistent” [38]. For this reason, the exact state of a Physical Machine might not be consistent in the Infrastructure Landscape and for each provisioning requests against a Physical Machine, a transaction has to be made with 2 stages (inspired by the 2 phase commit protocol. The two stages are reservation and provisioning.

The main principle is that provisioning can only happen when a reservation has been placed against the Physical Machine. When a Physical Machine has first been identified as a candidate for provisioning via the Infrastructure Landscape, a reservation has to be made first in order to mark the resources as reserved. These reservations could be short-lived (such as in the case of on-demand and immediate access to the Virtual Machine) or could point to a provisioning in the future. With regards to planning, the scheduling method that keeps track of the reservations is fully distributed. At runtime, future reservations can be moved from one Physical Machine to another as required. We are currently looking at modelling this approach using memory management techniques (such as Garbage Collection) to identify a novel technique. When live migration is integrated, the same techniques could be applied to reservations or running instances of Virtual Machines.

Within Physical Machine a reservation takes the form of a temporal lease over those resources, which are physically committed only when the provisioning request arrives after a reservation request. If a provisioning request does not arrive within a specified time frame, the lease expires and the resources are released back into the Physical Machine pool.
### 6.13.4 Libvirt

For this first prototype implementation, we have focused on integrating libvirt [9] as the means to provide physical virtualization capabilities. Libvirt by itself already provides several means of abstraction to different open source virtualization technologies, allowing the project to experiment with different hypervisors such as Xen or KVM. Libvirt is a very popular and actively developed open-source framework, and it has also been integrated within other projects as a means of abstracting virtualization [39].

One of the benefits of libvirt for the initial phase of the project has been to facilitate an iterative development process and experimentation with the virtualization technologies. The initial Open Reference Case (ORC) images have been built using libvirt and images for XEN and KVM have been provided and made available to the consortium. It is expected that during the duration of the project other types of hypervisors will be integrated, such as proprietary ones.

### 6.14 infrastructure.monitoring

Infrastructure monitoring is provided by agents with specific monitoring tasks which are distributed around the infrastructure. There are several layers of monitoring tasks.

![Monitoring Agents Architecture](image)

**Figure 48: Monitoring Agents Architecture**
6.14.1 Data collection layer (L0)

Data collection, also referred to as gathering, is at layer 0 of the monitoring architecture. The monitoring tasks procure low-level and (mostly) uncompressed monitoring data. Data can be gathered using different monitoring engines like Ganglia, Munin, Nagios, and xentop. Currently the Ganglia [14] monitoring engine is supported. The interface to the gathering is as follows:

- `getStatus() : HashMap<Metric, Value>` - get values of all metrics from the monitored host
- `getStatis(Metric) : Value` - get a value of a single metric from the monitored host

Agents with the L0 monitoring task are deployed on every machine in the infrastructure and get configured by the task on the higher layer (L1).

Although an operation for pulling data on request is supported, typically a timed push operation is used which publishes configured metrics according to timed intervals. The configuration for timed push contains a list of metrics to get status from, the interval between two status reports, the pubsub channel to which monitoring data is published and the L1 task subscribed to.

6.14.2 Evaluation layer (L1)

In general the function of the evaluation task is to evaluate monitoring data received from L0 tasks. The evaluation monitoring task automatically configures L0 tasks to receive needed metrics. Tasks above layer 1, also referred to as service tasks, can use the evaluation task for the validation of rules.

The current implementation of the evaluation task supports validation of simple rules referring to single metric values e.g. CPU, memory, I/O, and network usage as well as aggregated values (i.e. average, sum). Logical operators are also supported which enables an arbitrary number of rules to be combined.

The configuration of the evaluation task contains the rule to validate and a channel to send the result of validation. The evaluation task is configured to send the validation result only when the rule validation succeeds so there no network overhead.

The rule syntax is presented in the Java CUP [30] parser notation. The rule syntax is as follows:

```java
terminal SEMI, COMMA, LPAR, RPAR, LBRICK, RBRICK;
terminal AND, OR;
terminal GT, LT, GE, LE, EQ, NEQ;
terminal PLUS, MULT, MINUS, DIV;
terminal AVG, SUM;
terminal Double NUMBER;
terminal String IDENT;
non terminal exprList;
non terminal IBoolNode command;
non terminal IBoolNode exprBool;
non terminal IBoolNode exprCheck;
non terminal INumericNode exprNumeric;
```
non terminal INumericNode exprNumericOperation;
non terminal INumericNode exprNumericAggregation;

precedence left AND;
precedence left OR;

precedence left GT, LT, GE, LE, EQ, NEQ;
precedence left PLUS, MINUS;
precedence left MULTIPLY, DIV;

command ::= exprBool:e SEMI {:RESULT = e;:};

exprBool ::= exprBool:left AND exprBool:right
{: RESULT = new LogicNode (BoolOperation.AND, left, right); :}
| exprBool:left OR exprBool:right
{: RESULT = new LogicNode (BoolOperation.OR, left, right); :}
| LPAR exprBool:e RPAR
{: RESULT = e;:}
| LBRICK exprCheck:e RBRICK
{: RESULT = e;:}

exprCheck ::= exprNumeric:left GT exprNumeric:right
{: RESULT = new CheckNode (ComparasionOperator.GT, left, right); :}
| exprNumeric:left LT exprNumeric:right
{: RESULT = new CheckNode (ComparasionOperator.LT, left, right); :}
| exprNumeric:left GE exprNumeric:right
{: RESULT = new CheckNode (ComparasionOperator.GE, left, right); :}
| exprNumeric:left LE exprNumeric:right
{: RESULT = new CheckNode (ComparasionOperator.LE, left, right); :}
| exprNumeric:left EQ exprNumeric:right
{: RESULT = new CheckNode (ComparasionOperator.EQ, left, right); :}
| exprNumeric:left NEQ exprNumeric:right
{: RESULT = new CheckNode (ComparasionOperator.NEQ, left, right); :}

exprNumeric ::= NUMBER:i
{: RESULT = new NumberNode(i); :}
| IDENT:ref
{: RESULT = new ReferenceNode (ref); :}
| LPAR exprNumeric:e RPAR
{: RESULT = e; :}
| exprNumericOperation:e
{: RESULT = e; :}
| exprNumericAggregation:e
{: RESULT = e; :}

exprNumericOperation ::= exprNumeric:left PLUS exprNumeric:right
{: RESULT = new SimpleNumericNode(NumericOperator.PLUS, left, right);:}
| exprNumeric:left MINUS exprNumeric:right
{: RESULT = new SimpleNumericNode(NumericOperator.MINUS, left, right);:}
| exprNumeric:left MULTIPLY exprNumeric:right
{: RESULT = new SimpleNumericNode(NumericOperator.MULTIPLY, left, right);:}
| exprNumeric:left DIV exprNumeric:right
{: RESULT = new SimpleNumericNode(NumericOperator.DIVIDE, left, right);:}
Here is a list of rule examples with their explanation:

- \([\text{cpu} > 0.10]\): simple rule that checks whether the CPU is above given value.
- \([\text{cpu} > 0.10] \&\& [\text{mem} > 2]\): rule fires only when both conditions are satisfied.
- \([\text{cpu}\{\text{host1,2s}\} > 0.10] \&\& [\text{mem}\{\text{host1,3s}\} > 2] || [\text{mem}\{\text{host2}\} < 3]\): this rule uses metrics from different hosts as well as different sampling intervals.
- \(! ([\text{cpu} \times 3.14 > (2.714 \times 2) / (\text{mem} / 1024)] \&\& [\text{mem} > 2])\): rule is fired when the expression inside the parentheses is NOT true.
- \([\text{AVG}(\text{cpu}\{\text{host1,1s}\}, 10s) > \text{AVG} (\text{cpu}\{\text{host2,1s}\}, 10s)\]: this example shows the use of aggregation functions.
- \([\text{AVG}(\text{cpu}\{\text{host1,1s}\}, 10s) > 50] \implies [\text{ram}\{\text{host1}\} > 20]\): logical implication might also be used in the definition of the rule.

### 6.14.3 Service Layer (L2)

Every request for monitoring has to enter through one or more service layer (L2) monitors that form the boundary of the entire monitoring architecture. Each L2 monitoring task implies a certain configuration of the L1 monitors, which are managed dynamically.

The service layer task implemented is used for communication with layer 1 tasks using instant messaging (IM) which is convenient for debugging and infrastructure monitoring demonstration purposes.

### 6.14.4 Monitoring Message Payload Format

For monitoring message payload the JSON [31] data-interchange format is used. It is relatively fast for machines to parse and generate and it's easy for humans to read thus making it easier to debug.

The message payload format (JSON) should not be confused with the message serialization for which XML and Base64 is used. More on message serialization can be found in Section 6.4 of the deliverable.

### 6.14.5 Monitoring Virtual Machines

As discussed in Section 4.5.5, there are various approaches to extracting L0 data from virtual machines. In the year one implementation it was decided to install Ganglia within the virtual machine, and have it capture raw data for a Monitoring Agent associated with the physical machine. This minimises the overhead inside the virtual machine, whilst exposing more data than a hypervisor could provide. A screenshot of a virtual machine being monitored is provided in Figure 49.
6.15 Quality Assurance

Regarding the implementation of Infrastructure Management, it should be noted that significant effort was made to ensure code developed is of a high quality. In particular:

- Maven has been promoted and configured to manage the build process, dependencies and support continuous integration
- Various best practices such as using CPD (Copy Paste Detector) and PMD (commonly understood to stand for Programming Mistake Detector) reports have been embraced to maximise code quality
- A consistent coding style has been adopted
- JavaDocs have been provided
- A suite of Maven-generated project websites have been created.

Deliverable D.A1a M12 Framework Architecture describes these quality assurance tools and techniques in more detail.
7 Reference Deployment

7.1 Introduction

In parallel with the development of the software components of the Infrastructure Management work package, significant effort has been put into delivering a functional reference deployment to support general experimentation as well as an overall, integrated, SLA@SOI Adhoc Demonstrator.

Ultimately, this reference deployment has allowed the Adhoc Demonstrator User Interface to be used to negotiate and provision a required set of Open Reference Case software images, the dynamic deployment of which is automatically monitored and adjusted as appropriate to minimise the violations of agreed SLAs. The entire infrastructure landscape can be administered through a scalable message-based interface.

Within the first year of the project, the Infrastructure Management reference deployment has considered the following technical areas:

- Testbed Infrastructure
- Virtualisation Platform
- Default Operating System Services
- Network Configuration
- Storage Management
- Open Reference Case Images
- Automation and Management

7.2 Testbed Infrastructure

Infrastructure Management has deployed dedicated testbed infrastructure to support the SLA@SOI project. This includes a server available on the internet to all partners in the consortium, and workstations on local area networks for ad hoc development and demonstration purposes. Additional hardware will be added to the testbed as and when required.

The current internet accessible testbed is a high performance HP* Proliant DL380 G5 Series server in a 2U form factor (Rack). The machine specification includes:

- CPU: 4 x Intel® Xeon® processor 5500 series at 2.66GHz,
- Memory: 12 GB of Ram, expandable to 64 GB of RAM
- Storage: 8 SFF Hot plug to support Serial-attached SCSI (SAS) and Serial ATA (SATA).
- Network: Two (2) Embedded NC373i Multifunction Gigabit Network Adapters with TCP/IP Offload Engine

The server has been built with CentOS 5.2 X64_86 edition with XEN as the default virtualization hypervisor. The server has been preloaded with the SLA@SOI ORC images and can access the SLA@SOI svn repository on the internet to download, compile, install and run the development versions of different software components of SLA@SOI. To support the SLA@SOI messaging bus, an OpenFire XMPP server [40] has been installed with all relevant network ports configured and open. The server is also enabled to host arbitrary VMs that SLA@SOI developers may require.
The server has been deployed in a DMZ area within an Intel data center facility. A block of external internet IP addresses has been reserved and assigned to the machine, enabling multiple VMs to be externally accessible.

Various partners contributing to Infrastructure Management have also deployed internal testbed infrastructure for local development, testing and demonstration purposes.

In the Intel Ireland IT Innovation Centre 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. SLA@SOI deployments on both the CentOS and Ubuntu operating systems, employing Xen and KVM hypervisors respectively, have been successfully demonstrated on these machines. 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 50: SLA@SOI testbed in Intel IT Innovation Centre Showcase area](image)

### 7.3 Virtualisation Platform

There are numerous virtualisation platforms currently available. For license, cost, flexibility and ease-of-configuration reasons it was decided to use Linux-based hypervisors to support the initial reference deployment.
The SLA@SOI framework has been successfully demonstrated on two operating systems to date: CentOS with Xen as the hypervisor, and Ubuntu (both server and desktop editions) with KVM as the hypervisor.

**CentOS** [41] is a derivative of the official Redhat Linux Enterprise distribution. Using the rpm package manager [42], CentOS 5.2 integrates Xen as its default hypervisor with support for paravirtualization and full-virtualization out of the box. CentOS is mainly a server/enterprise oriented distribution.

**Ubuntu** [43], based on Debian [44], supports KVM as its default hypervisor and official virtualization technology. Less conservative in its release policy than CentOS or Debian, Ubuntu often ships with up-to-date packages, making it a convenient workstation platform and ideal for experimenting with the latest releases of operating system and hypervisor technologies.

### 7.4 Default Operating System Services

Several Operating System services need to be installed and configured on the virtualisation platforms to support the reference deployment.

**Libvrt** [9] is used to provide a generic interface into both hypervisors employed. Both CentOS and Ubuntu come integrated with libvirt, as well as virt-manager, a GUI-based tool for creating and managing virtual machine images. Libvirt runs as a service (libvird) which provides facilities for local and remote management of virtual images by means of connecting to the service provider.

**Ganglia** [14] is the generic instrumentation monitoring framework on which Infrastructure Management depends. As the .rpm and .deb official distributions of Ganglia for both Centos and Ubuntu were not the most up-to-date, including necessary functionality for Infrastructure Management, Ganglia needed to be compiled and installed from source.

Other operating system services which will be integrated into the reference deployment in the future include DHCP and DNS servers.

### 7.5 Network Configuration

Virtual machines need to have their network settings automatically configured to be accessible over the network. The initial reference deployment implements the following approach to network configuration:

- A network bridge is used to allow the virtual machines to connect to the real network. The virtual machines are not under a NAT configuration and will pick up their network configuration from a local DHCP server.
- The MAC addresses that are assigned to the VMs are managed and registered with the DHCP server. The MAC addresses of the VMs are known before provisioning time.
- The DHCP server is configured to map the virtual machine MAC address to a fixed IP address during boot time. Following this process it is possible to go from a known MAC address to a known IP address, ensuring particular VMs receive particular IP addresses.

This setup allows virtual machines to be accessible over a LAN, or over the internet if pre-allocated external IP addresses are used.
Figure 51: Network Configuration of a Virtual Machine

Figure 51 illustrates various aspects of the network configuration employed. The bottom left of the image illustrates the MAC address assigned to the VM via the network bridge configuration. The right of the image illustrates the assignment of MAC address to IP address in the HTML administration interface of the external router. The upper left of the screen confirms that the virtual machine has received the assigned MAC and IP address.

This initial approach to Network Configuration is basic, limited, and requires manual administration. Future iterations of the reference deployment will implement more intelligent, automated configuration solutions. Examples of how to configure more advanced network features such as virtual public/private networks can be found in the network management implementation of OpenNebula [12]. Integration between OpenNebula and SLA@SOI Infrastructure Management is being explored as part of ongoing conversations between the SLA@SOI and RESERVOIR [45] projects.

7.6 Storage Management

Many modern server-based applications and services need to access enormous volumes of data. Modern data centres often include significant NAS or SAN infrastructure to deliver the data storage needs. The servers mount the appropriate infrastructure, and the applications and services use them rather than local hard disks. In a virtualised environment, virtual machines should also be able to access remote storage, either real or virtual, however, support for remote storage is not yet implemented in the reference deployment. The virtual images are self-contained, and are hosted purely by the physical host. It is planned to
extend the reference deployment to support live migration – the ability to configure remote storage will be added at that stage of the implementation.

### 7.7 Open Reference Case Images

As Infrastructure Management currently supports both KVM and XEN as hypervisors, virtual images of the Open Reference Case (ORC) that run in both hypervisors have been created and configured. Various deployment scenarios of the ORC have been supported by creating virtual machines with different combinations of the ORC components: database, services and BPEL engine.

The most complex deployment supported requires three virtual machines to be instantiated: one containing the database, one containing the services and one hosting the BPEL engine. Figure 50 illustrates the ORC deployed across three VMs (on the left) hosted by the KVM hypervisor. The services deployed in the BPEL engine and the list of all running VMs can also be seen.

![Open Reference Case deployed across three VMs](image)

To simplify the process of creating new images every time the ORC software is modified, scripts have been developed which automatically install relevant development tools such as SVN and Maven, get the latest version of the ORC source code, and compile, install and deploy the latest versions of the ORC components.

### 7.8 Automation and Management

Another important consideration in Infrastructure Management is the automation and management of the boot, installation and execution stages of the framework. This is of critical importance for scalable architectures.

Ideally, a new physical machine connected to the Infrastructure Provider LAN should be able to locate, download, install and boot the most recent version of the operating system and hypervisor required for provisioning. PXE boot is a
technology that can be used to automate the initial building of the physical machine.

The physical machine should then install and configure any operating system services required, as well as the latest version of the SLA@SOI Infrastructure Management software. In some cases the software may have to be compiled and deployed as part of this process. To this end, various scripts have been written to help automate these processes. When the script has completed and the image booted, the machine registers its availability on the Infrastructure Landscape.

Once up and running, Infrastructure Management should be able to easily manipulate physical machines – for example querying their uptime, reviewing the processes they contain, and triggering additional upgrades.

The messaging framework that has been implemented by Infrastructure Management enables such remote command execution, and publish/subscribe channels can be used to arbitrary numbers of machines with the one command.

![Figure 53: Simultaneous Machine Management](image)

Figure 53 illustrates 2 physical machines responding to a “uname –a” command executed in broadcast mode - all machines registered with the channel receive and process the message. The “uname –a” command in unix/linux displays several parameters including the node name, the operating system details and the hardware architecture of the machine. It is also possible to direct commands to individual machines if required.
8 Conclusions

8.1 Summary

The Infrastructure Management work package has made significant progress in the first twelve months of SLA@SOI:

- A significant state of the art review has been completed
- An initial set of requirements has been identified
- A flexible, scalable infrastructure management platform has been architected and a prototype implemented
- A reference adhoc demonstrator has been integrated and made available for experimentation.
- An internet-accessible testbed has been deployed.

The state of the art has been progressed in a number of areas including:

Abstract Infrastructure Framework Definition
Initial flexible technology-independent models, both internal and external, have been defined and implemented. Although the focus has been on supporting key functional parameters for the first 12 months, the model has been designed to accommodate non-functional parameters also.

Harmonized Interface for Heterogeneous Virtualised Infrastructure Services
Important abstract methods for the interface into Infrastructure Management have been defined, supporting the provisioning, re-provisioning and management of virtualised infrastructure. The harmonized interface is enabled by the external abstract model. Although the current implementation only manipulates local Xen and KVM hypervisors, the prototype has been architected to allow manipulation of arbitrary local hypervisors, as well as 3rd party cloud infrastructure. SLA@SOI Infrastructure Management was a founding member of the new OGF Open Cloud Computing Interface (OCCI) working group, and is helping drive the definition of a generic community-wide interface.

Monitoring, Logging and Alerting Services
A powerful monitoring infrastructure has been architected and demonstrated, showing how Ganglia can be used to instrument arbitrary physical (and virtual) hardware parameters, which are then collated and translated into higher level infrastructure events, which can ultimately be used to identify SLA violations. Monitoring is built on top of an XMPP communications infrastructure, ensuring scalability and making it implementation agnostic.

Dynamic provisioning and re-provisioning
Although prediction algorithms have not yet been integrated into the prototype, the infrastructure is now in place to allow the internal monitoring of events, the population of an infrastructure landscape, and the commanding of appropriate re-provisioning to avoid potential SLA violations.
8.2 Outlook on Future Work

Infrastructure Management is keen to address many interesting challenges in Years 2 and 3 of the project. Key technical topics to be tackled include

- Providing interfaces for SLATemplates to be offered and software images to be loaded and registered
- Exploring additional potentially useful technologies and protocols, including AMQP for messaging, Nagios and Munin for monitoring, and VMWare for virtualisation.
- Extending the models and implementation to support both additional functional and non-functional parameters. Network management and storage profiles need to be considered, and non-functional aspects such as performance bounds, isolation levels and geo-location need to be accommodated
- Integrating run-time prediction (task A6.3) into the Infrastructure Management layer to allow SLA violations to be predicted and avoided
- Integrating support for internal Infrastructure Provider policies
- Integrating with external cloud infrastructures, including Amazon EC2.

On a more general note, SLA@SOI Infrastructure Management is unique in the industry given the SLA-awareness which is at its core. As the industry matures, SLA-aware monitoring and management of infrastructure will become a significant topic that SLA@SOI will be well placed to influence and lead.

To this end, next steps include not just advancing the technical state-of-the-art, but also increasing our engagement with key players in the community to embed the learning’s of SLA@SOI into the Cloud Computing movement.


9 References

[1] SLA@SOI Annex 1 – “Description of Work”
[34] Xen Scheduling, available at http://wiki.xensource.com/xenwiki/Scheduling
Appendix A. Glossary

Please see Appendix A in Deliverable D.A1a M12 Framework Architecture for a comprehensive glossary of SLA@SOI terms.
Appendix B. Abbreviations

AMQP  Advanced Message Queuing Protocol
API   Application Programming Interface
CIM   Common Information
CPU   Central Processing Unit
CSV   Comma Separated Variable
DAO   Data Access Objects
DMTF  Distributed Management Task Force
HAL   Hardware Abstract Layer
IaaS  Infrastructure as a Service
IM    Instant Messaging
JMS   Java Messaging Service
JMX   Java Management Extensions
JPA   Java Persistence API
JSON  JavaScript Object Notation
MOM   Messaging Oriented Middleware
NAS   Network Attached Storage
Nfp   Non-functional property
OCCI  Open Cloud Computing Interface
OGF   Open Grid Forum
ORC   Open Reference Case
OVF   Open Virtual Machine Format
POJO  Plain Old Java Object
QoS   Quality of Service
RAM   Random Access Memory
SAN   Storage Area Network
SLA   Service Level Agreement
SLO   Service Level Objective
SOA   Service Oriented Architecture
SOAP  Simple Object Access Protocol
SOI   Service Oriented Infrastructure
UI    User Interface
URI   Uniform Resource Identifier
VM    Virtual Machine
WP    Work Package
WS    Web Service
WSDL  Web Service Description Language
XML   Extensible Mark-up Language
XMPP  Extensible Messaging and Presence Protocol
XSD   XML Schema Definition
Appendix C. Review of IaaS API Data Schemas

Scope
This appendix presents a summary of an analysis of the data models employed by various Infrastructure as a Service providers. To make the scope manageable, storage and networking aspects are not considered.

Amazon EC2

Resources available: Amazon EC2 provides compute resources. One resource is known as an Instance.

Supported Representation Rendering: XML conforming to WSDL Schema
As Amazon EC2 is an RPC driven API the follow entities of the model are passed to methods as individual parameters as opposed to a document containing all parameters to the method.

Compute Resource Request
The majority of the functional requirements of infrastructure contained in an EC2 request are embedded in the instanceType parameter. Further to those functional requirements are:

```
imageId, minCount, maxCount, keyName, groupSet, userData,
instanceType, placement, kernelId, ramdiskId, blockDeviceMapping
```

Compute Resource Representation
This model is returned when clients query for information about what instances are currently running.

```
amiLaunchIndex, dnsName, imageId, instanceId, instanceState,
instanceType, keyName, kernelId, launchTime, placement,
privateDnsName, productCodes, ramdiskId, reason
```
Sun Cloud API

Resources available: Sun provides compute resources. One resource is known as a virtual machine (VM).

Supported Representation Rendering: JSON

The Sun Cloud API is consistent in its request and resource representation models. A request is a subset of a resource's representation. This is as it uses the REST approach as its underlying architectural principle. In order for a new resource to be created, all one needs to do is to supply a VM model to the URI that allows the creation of VMs.

Compute Resource Request

See compute resource representation.

Compute Resource Representation

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th>Occurs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>String</td>
<td>1</td>
<td>Logical name of this VM, unique within the owning Cluster. [POST][PUT]</td>
</tr>
<tr>
<td>uri</td>
<td>URI</td>
<td>1</td>
<td>GET, PUT, and DELETE may be used to retrieve representations of the VM, to update it, and to delete it, respectively.</td>
</tr>
<tr>
<td>model_status</td>
<td>String</td>
<td>0..1</td>
<td>Current modeling status of this virtual machine (read only). FIXME - enumerate valid values beyond &quot;deployed&quot;, &quot;error&quot;, etc.</td>
</tr>
<tr>
<td>run_status</td>
<td>String</td>
<td>0..1</td>
<td>Current running status of this virtual machine (read only). FIXME - enumerate valid values beyond &quot;halted&quot;, &quot;running&quot;, &quot;paused&quot;.</td>
</tr>
<tr>
<td>from_template</td>
<td>String</td>
<td>0..1</td>
<td>On a Create VM request type, the URI of an existing VM template to initialize default values from. [POST]</td>
</tr>
<tr>
<td>from_vm</td>
<td>String</td>
<td>0..1</td>
<td>On a Create VM request type, the URI of an existing VM to initialize default values from. [POST]</td>
</tr>
<tr>
<td>description</td>
<td>String</td>
<td>0..1</td>
<td>Human friendly description of this virtual machine. [POST][PUT]</td>
</tr>
<tr>
<td>hostname</td>
<td>String</td>
<td>0..1</td>
<td>Fully qualified host name of this virtual machine.</td>
</tr>
<tr>
<td>os</td>
<td>String</td>
<td>0..1</td>
<td>Operating system running on the VM being modelled. FIXME - enumerate the legal values. [POST]</td>
</tr>
<tr>
<td>cpu</td>
<td>Integer</td>
<td>0..1</td>
<td>CPU speed in Mhz. [POST]</td>
</tr>
<tr>
<td>memory</td>
<td>Integer</td>
<td>0..1</td>
<td>Main memory size in GB. [POST]</td>
</tr>
<tr>
<td>boot_disk</td>
<td>Integer</td>
<td>0..1</td>
<td>Boot disk space to allocate in GB. [POST]</td>
</tr>
<tr>
<td>data_disk</td>
<td>Integer</td>
<td>0..1</td>
<td>Data disk space to allocate in GB. [POST]</td>
</tr>
<tr>
<td>temp_disk</td>
<td>Integer</td>
<td>0..1</td>
<td>Temporary disk space to allocate in GB. [POST]</td>
</tr>
<tr>
<td>Field</td>
<td>Type</td>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>--------</td>
<td>-------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>params</strong></td>
<td>{ }</td>
<td>0..1</td>
<td>Configuration parameters for this VM, keyed by parameter name. The list of system defined configuration parameters is TBD, but one of them will be &quot;user_params&quot;, whose value is a hash of arbitrary user defined configuration parameters. [POST][PUT]</td>
</tr>
<tr>
<td><strong>tags</strong></td>
<td>[ ]</td>
<td>0..1</td>
<td>Arbitrary values assigned by the user. These values have no semantic impact on the operation of the cloud. [POST][PUT]</td>
</tr>
<tr>
<td><strong>back_up</strong></td>
<td>URI</td>
<td>1</td>
<td>A POST of a Backup representation to this URI requests creation of a new Backup.</td>
</tr>
<tr>
<td><strong>attach</strong></td>
<td>URI</td>
<td>1</td>
<td>A POST of a PublicAddress representation to this URI requests connection to a Public Address. Or, a POST of a VNet representation to this URI requests connection to a VNet.</td>
</tr>
<tr>
<td><strong>detach</strong></td>
<td>URI</td>
<td>1</td>
<td>A POST of a PublicAddress representation to this URI requests disconnection from a Public Address. Or, a POST of a VNet representation to this URI requests disconnection from a VNet.</td>
</tr>
<tr>
<td><strong>backups</strong></td>
<td>Backup[]</td>
<td>0..1</td>
<td>Backup snapshots from this virtual machine.</td>
</tr>
<tr>
<td><strong>interfaces</strong></td>
<td>Interface[]</td>
<td>1</td>
<td>Network interfaces associated with this virtual machine. These are created automatically when VNets or Public Addresses are associated with VM.</td>
</tr>
<tr>
<td><strong>controllers</strong></td>
<td>{}</td>
<td>0..1</td>
<td>Hash of URIs which may be used to request state changes in the VM via POST requests (see next table), keyed by the type of state change being requested.</td>
</tr>
</tbody>
</table>
Flexiscale

Resources available: Flexiscale provides virtual compute resources. One resource is known as a server.

Supported Representation Rendering: XML conforming to WSDL Schema

Compute Resource Request

In order for a new server to be created, all one needs to do is to supply a server model to the URI that allows the creation of servers. See Compute Resource Representation.

Compute Resource Representation

```
<xsd:complexType name="Server">
  <xsd:all>
    <xsd:element name="server_id" type="xsd:int" minOccurs="0"/>
    <xsd:element name="server_name" type="xsd:string" minOccurs="0"/>
    <xsd:element name="status" type="xsd:string" minOccurs="0"/>
    <xsd:element name="package_id" type="xsd:int" minOccurs="0"/>
    <xsd:element name="processors" type="xsd:int" minOccurs="0"/>
    <xsd:element name="memory" type="xsd:int" minOccurs="0"/>
    <xsd:element name="operating_system_image" type="typens:OperatingSystemImage" minOccurs="0"/>
    <xsd:element name="disk_capacity" type="xsd:long" minOccurs="0"/>
    <xsd:element name="disk_ids" type="typens:ArrayOf_xsd_int" minOccurs="0"/>
    <xsd:element name="network_interfaces" type="typens:ArrayOf_xsd_int" minOccurs="0"/>
    <xsd:element name="initial_password" type="soapenc:string" minOccurs="0"/>
    <xsd:element name="uptime" type="xsd:long" minOccurs="0"/>
    <xsd:element name="ip_addresses" type="typens:ArrayOf_xsd_string" minOccurs="0"/>
    <xsd:element name="modified" type="xsd:boolean" minOccurs="0"/>
  </xsd:all>
</xsd:complexType>
```
**ElasticHosts**

**Resources available:** ElasticHosts provides virtual compute resources. One resource is known as a server.

**Supported Representation Rendering:** JSON, CSV

**Compute Resource Request**

In order for a new server to be created, all one needs to do is to supply a server model to the URI that allows the creation of servers. See Compute Resource Representation.

**Compute Resource Representation**

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>Server name.</td>
</tr>
<tr>
<td>parent</td>
<td>UUID of parent object, if any.</td>
</tr>
<tr>
<td>cpu</td>
<td>cpu quota in core MHz.</td>
</tr>
<tr>
<td>smp</td>
<td>number of virtual processors or 'auto' to calculate based on cpu.</td>
</tr>
<tr>
<td>mem</td>
<td>virtual memory size in MB.</td>
</tr>
<tr>
<td>ide:BUS[0-1]:UNIT[0-1]:media</td>
<td>Drive UUID to connect as specified device.</td>
</tr>
<tr>
<td>scsi:BUS[0]:UNIT[0-7]:media</td>
<td>Media type - set to 'cdrom' to simulate a cdrom, set to 'disk' or leave unset to simulate a hard disk.</td>
</tr>
<tr>
<td>boot</td>
<td>Device to boot, e.g. ide:0:0 or ide:1:0. Set to a list to make multiple devices bootable.</td>
</tr>
<tr>
<td>nic:0:model</td>
<td>Create network interface with given type (use 'e1000' as default value).</td>
</tr>
<tr>
<td>nic:0:dhcp</td>
<td>The IP address offered by DHCP to network interface 0. If unset, no address is offered. Set to 'auto' to allocate a temporary IP at boot.</td>
</tr>
<tr>
<td>vnc:ip</td>
<td>IP address for overlay VNC access on port 5900. Set vnc:ip='auto', to reuse nic:0:dhcp if available, or otherwise allocate a temporary IP at boot.</td>
</tr>
<tr>
<td>vnc:password</td>
<td>Password for VNC access</td>
</tr>
<tr>
<td>vnc:tls</td>
<td>Set to 'on' to require VNCrypt-style TLS auth in addition to the password. If this is unset, only unencrypted VNC is available.</td>
</tr>
</tbody>
</table>
GoGrid

Resources available: ElasticHosts provides virtual compute resources. One resource is known as a server.

Supported Representation Rendering: XML, JSON, CSV

Compute Resource Request

To create a server on GoGrid the following is supplied as a name-value tuple:

- api_key
- sig
- v
- name
- image
- ram
- ip
- description (opt)
- format (opt)

Compute Resource Representation

A server on the GoGrid is represented by a Server entity. A response containing server representations will be a list of servers along with summary containing total; the total number returned, start; the start index of the list (always 0) and a variable named returned which is equal to total. The attributes returned for server in the list are:

- id
- name
- description
- ip
- image
- ram
- state
- type
- os
Enomalism

**Resources available:** Enomalism provides virtual compute resources. One resource is known as a virtual machine.

**Supported Representation Rendering:** XML

Large dependence on Libvirt. Below is how machines are tracked (the landscape) within enomaly. What's shown is the a table of the enomaly system, in which machines running are stored. There are a number of attributes but most interesting is the "xmldesc" field. This field corresponds to the Libvirt XML descriptor of the virtual machine.

```sql
CREATE TABLE `machine` (
    `id` int(11) NOT NULL auto_increment,
    `ip_addr` text,
    `ip_port` int(11) default NULL,
    `rest_baseurl` text,
    `uuid` varchar(36) NOT NULL,
    `parent_id` int(11) default NULL,
    `suspend_data` varchar(512) default NULL,
    `machine_name` varchar(512) NOT NULL,
    `xmldesc` text,
    PRIMARY KEY (`id`),
    UNIQUE KEY `uuid` (`uuid`),
    UNIQUE KEY `machine_name` (`machine_name`),
    UNIQUE KEY `suspend_data` (`suspend_data`),
    KEY `machine_parent_id_exists` (`parent_id`)
)
```

**Compute Resource Request**

Virtual machines in Enomalism are invariably requested using the Libvirt XML descriptor. In order to create a virtual machine in enomaly a number of calls are required until one is up and running. This implies a certain amount of state is maintained throughout the calls that create the VM. This is counter to REST principles.

**Compute Resource Representation**

Are represented using the Libvirt XML descriptor.²

---

¹ [http://libvirt.org/format.html](http://libvirt.org/format.html)
OpenNebula

Resources available: OpenNebula provides virtual compute resources. One resource is known as a virtual machine (VM).

Supported Representation Rendering: Unclear

Compute Resource Request

To request the creation of a VM, a user has to supply a Virtual Machine Template string to the one.vmallocate RPC method. This string is a vector of name-value pairs. This is convenient as it allows easy extension. The current set of attributes is:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Name that the VM will get, for description purposes. If NAME is not supplied a name generated by one will be in the form of one-vm_id.</td>
</tr>
<tr>
<td>CPU</td>
<td>Percentage of CPU divided by 100 required for the Virtual Machine. Half a processor is written 0.5.</td>
</tr>
<tr>
<td>MEMORY</td>
<td>Amount of RAM required for the VM, in Megabytes.</td>
</tr>
</tbody>
</table>
| DISK | Image file to be used to deploy the Virtual Machine. Disk attributes:  
- IMAGE: Path of the image file.  
- DEVICE: Device associated to the image file. |
| BOOT | Tells the kernel what device it is going to boot from. |
| KERNEL | The kernel file that will be used to boot the VM's, normally located at /boot (i.e. /boot/vmlinuz-2.6.XX.X-xen). |
| RAMDISK | The initial ramdisk file that will be used to boot the VM's, normally located at /boot (i.e. /boot/initrd.img-2.6.XX.X-xen). |
| NIC | Network interface, attributes:  
- MAC: HW address associated with the network interface.  
- BRIDGE: Name of the bridge the network device is going to be attached to. |
| REQUIREMENTS | Boolean expression that rules out provisioning hosts from list of machines suitable to run this VM. |
| RANK | This field sets which attribute will be used to sort the suitable hosts for this VM. Basically, it defines which hosts are more suitable than others. |

What's also useful with this type of request representation is that logical operators can be used to narrow resource selection (e.g. CPUSPEED>1000) via the REQUIREMENTS vector name.

## Compute Resource Representation

A representation of a VM can be retrieved by issuing the command “onevm show <vm_id>”. This will return back a VM representation containing the following information:

<table>
<thead>
<tr>
<th>ID</th>
<th>ONE VM identifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>Name of the ONE</td>
</tr>
<tr>
<td>STAT</td>
<td>Status of the VM</td>
</tr>
<tr>
<td>CPU</td>
<td>CPU percentage used by the VM</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory used by the VM</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>Host where the VM is being or was run</td>
</tr>
<tr>
<td>TIME</td>
<td>Time since the submission of the VM (days hours:minutes:seconds)</td>
</tr>
</tbody>
</table>
**Slicehost**

**Resources available:** Slicehost provides virtual compute resources. One resource is known as a slice.

**Supported Representation Rendering:** XML

**Compute Resource Request**

Typically it is enough to specify a slice’s image, flavour and name (see below) when requesting a new provisioning.

**Compute Resource Representation**

<table>
<thead>
<tr>
<th>Field</th>
<th>Access</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>rw</td>
<td>A string to identify the host</td>
</tr>
<tr>
<td>flavor_id</td>
<td>rw</td>
<td>Flavor - name, price and amount of RAM available for this slice offering (slices increase by 256MB RAM intervals)</td>
</tr>
<tr>
<td>image_id</td>
<td>rw</td>
<td>Image - name: the name of the OS image to run</td>
</tr>
<tr>
<td>backup_id</td>
<td>rw</td>
<td>Backup - a pointer to a backup image of the slice its name and when it was last backed up</td>
</tr>
<tr>
<td>Status</td>
<td>w</td>
<td>Current status of the slice</td>
</tr>
<tr>
<td>Progress</td>
<td>w</td>
<td>Progress of the current action</td>
</tr>
<tr>
<td>bw_in</td>
<td>w</td>
<td>Total incoming bandwidth per Gigabyte for the slice in the current billing cycle</td>
</tr>
<tr>
<td>bw_out</td>
<td>w</td>
<td>Same as above but for outgoing</td>
</tr>
<tr>
<td>Addresses</td>
<td>w</td>
<td>Array of IP addresses assigned to the slice</td>
</tr>
<tr>
<td>ip_address</td>
<td>w</td>
<td>Deprecated</td>
</tr>
<tr>
<td>root_password</td>
<td>w</td>
<td>Shown only on creation of slice</td>
</tr>
</tbody>
</table>
Apache Tashi

Resources available: Tashi provides virtual compute resources. One resource is known as an Instance.

Supported Representation Rendering: Unclear

Compute Resource Request

To create an instance using Tashi, a call to the cluster manager interface is made. This is done calling the createVm method and passing to it an Instance object (see representation below).

Compute Resource Representation

Apache Tashi uses Thrift to model Instances. The Thrift structure that represents an instance is as follows:

```
struct Instance {
   1:i32 id,
   2:i32 vmId,
   3:i32 hostId,
   4:bool decayed,
   5:InstanceState state,
   6:i32 userId,
   7:string name, // User specified
   8:i32 cores, // User specified
   9:i32 memory, // User specified
   10:list<DiskConfiguration> disks, // User specified
   11:list<NetworkConfiguration> nics // User specified
   12:map<string, string> hints // User specified
}
```

To represent disk and network configuration the following structs are used:

```
struct DiskConfiguration {
   1:string uri,
   2:bool persistent
}
struct NetworkConfiguration {
   1:i32 network
   2:string mac
}
```

Eucalyptus, Globus Nimbus

See the review of Amazon EC2.

AppNexus

As AppNexus does not offer an easily accessible programmable API (only command line tools), it is not included in this review.
Appendix D. IaaS API Analysis

Scope
An analysis has been performed on all known infrastructure as a service providers that publish an API and corresponding schema (data model) that allows the creation of compute resources. Storage and networking aspects of APIs and schemas have not been considered in this review.

Amazon EC2

- **Communication Pattern:** Synchronous
- **Web Service Technology:** Access is primarily through a WSDL interface[^4] over HTTP-SOAP. There is an interface that can be used somewhat like a REST-style approach
- **SLA:** [http://aws.amazon.com/ec2-sla/](http://aws.amazon.com/ec2-sla/). The user is responsible for providing evidence of violations. SLA is manual.

<table>
<thead>
<tr>
<th>Compute-resource oriented operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>runInstances</strong></td>
<td>The runInstances operation launches a specified number of instances.</td>
</tr>
<tr>
<td><strong>terminateInstances</strong></td>
<td>The TerminateInstances operation shuts down one or more instances.</td>
</tr>
<tr>
<td><strong>rebootInstances</strong></td>
<td>Reboots one or more specified instances.</td>
</tr>
<tr>
<td><strong>describeInstances</strong></td>
<td>The DescribeInstances operation returns information about instance representations that you own.</td>
</tr>
<tr>
<td><strong>confirmProductInstance</strong></td>
<td>The ConfirmProductInstance operation returns true if the specified product code is attached to the specified instance. Related to windows installations and their licensing (to confirm)</td>
</tr>
</tbody>
</table>

Comments

- ( - ) EC2 interface explicates the types of machines that can be provisioned via the runInstances method. This does not allow for custom-defined machines such that we will have in SLA@SOI.
- ( - ) The describeInstances method specifies an instance’s representation in a non-standard fashion - this could perhaps return the instance representation using OVF.
- ( + ) An almost defacto standard in infrastructure as a service providers
- ( - ) **No means to specify non-functional requirements**
- ( + ) Document oriented interface
- **CPU/Mem Sizing:** Fixed by CPU/Mem combo, T shirt model (Small, Medium, Large), compute intense/mem intense

The Sun Cloud API

- **Communication Pattern:** Synchronous
- **Web Service Technology:** A REST interface over HTTP. The base URI to resources is `X.cloud.sun.com`, where `X` is an unique virtual data center ID. Provisioned resources are grouped in clusters.
- **SLA:** No SLA is currently available as there is only an API, not an implementation.

## Compute-resource oriented operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /clusters/clusterX</td>
<td>Post the VM representation to that cluster's URI. The VM representation does not need to be complete; only the core information should be present. Note at this stage, the resources are not deployed. The VM however will be associated with the cluster.</td>
</tr>
<tr>
<td>GET /clusters/123456</td>
<td>Gets information about a specific cluster.</td>
</tr>
<tr>
<td>PUT /clusters/123456</td>
<td>Updates information about a cluster.</td>
</tr>
<tr>
<td>DELETE /clusters/123456</td>
<td>Deletes a cluster and all the resources grouped by it.</td>
</tr>
</tbody>
</table>
| POST /cluster/deploy?name=XYZ           | Performs the control action "deploy" on the named cluster. The full list of actions that can be executed on a cluster is:
|                                         | • deploy, undeploy, start, stop                                             |
|                                         | Each action is associated with a state of the same name.                    |
| GET /vms/XYZ                            | Gets the representation of a particular VM, identified by XYZ that may be associated with a cluster. |
| PUT /vms/XYZ                            | Updates the named VM with new attributes.                                   |
| DELETE /vms/XYZ                         | Deletes the named VM                                                        |
| POST /vms/XYZ?control=start             | Performs the control action "start" on the named VM. The full list of actions that can be executed on a VM is:
|                                         | • deploy, start, stop, undeploy, reboot, hibernate, resume                  |
|                                         | Each action is associated with a state of the same name.                    |
Comments

- (+) The separation of requesting resources and deploying them (executing the request) is useful as this pushes control to the client and enables the client control when to deploy (future timed provisioning).
- (+) This API's notion of a cluster of VMs maps very well to the current SLA@SOI ProvisioningRequest, which is a collection of virtual machines, configured (network) in a particular manner.
- (-) Does not include any notion of machine readable SLAs
Flexiscale

- **API:** [http://api.flexiscale.com](http://api.flexiscale.com)
- **Communication Pattern:** Synchronous. Those methods that return a job ID can issue a call back check to wait notification of job completion (ref method: WaitForJobs(jobID)).
- **Web Service Technology:** WSDL interface\(^5\) over HTTP-SOAP
- **SLA:** "We offer 99.99% uptime." \(^6\)

### Compute-resource oriented operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ListServers</td>
<td>Lists customers servers and returns servers machine representations as a name-value map.</td>
</tr>
<tr>
<td>CreateServer</td>
<td>Creates a server based on a name-value map of hardware and OS parameters. Returns a Job ID.</td>
</tr>
<tr>
<td>StartServer</td>
<td>Starts a server with the given server ID. Returns a Job ID.</td>
</tr>
<tr>
<td>StopServer</td>
<td>Stops a server with the given ID and the stop method type (hard, soft). Returns a Job ID.</td>
</tr>
<tr>
<td>StopStartServer</td>
<td>Stops and starts a server with the given ID. Returns a Job ID</td>
</tr>
<tr>
<td>ModifyServer</td>
<td>Allows a server be reconfigured including the basics; ram, cpu, disk etc. The response is of the new machine representation. A StopStartServer call is then required to execute the new configuration</td>
</tr>
<tr>
<td>RebootServer</td>
<td>Reboots a server with the given server ID. Returns a Job ID</td>
</tr>
<tr>
<td>DestroyServer</td>
<td>Deletes a server, returning true or false to indicate success or not</td>
</tr>
</tbody>
</table>

### Comments

- ( - ) No inbuilt capability to deal with a grouping of servers in a request on their service side. This is possible through repeated calls to their interface.
- ( - ) The createServer/listServers specifies an server's representation in a non-standard fashion - this could perhaps return the instance representation using OVF.
- ( + ) Very simple API to use. No complex types.
- ( - - ) **No means to specify non-functional requirements**
- **CPU/Memory:** Proportional, more Memory means more CPU automatically. Storage is dealt with separately

---

\(^5\) [http://api.flexiscale.com/current/FlexiScale.wsdl](http://api.flexiscale.com/current/FlexiScale.wsdl)

\(^6\) [http://www.flexiscale.com/FlexiScale_FAQ.pdf](http://www.flexiscale.com/FlexiScale_FAQ.pdf)
ElasticHosts

- **API:** [http://www.elastichosts.com/products/api](http://www.elastichosts.com/products/api)
- **Communication Pattern:** Synchronous
- **SLA:** [http://www.elastichosts.com/benefits/peace-of-mind](http://www.elastichosts.com/benefits/peace-of-mind) (SLA = peace of mind that we know what we are doing)
- **Web Service Technology:** REST interface over HTTP supporting only GET and POST verbs. The base URL to resources is https://X.api.elastichosts.com, where ‘X’ is an availability zone.

<table>
<thead>
<tr>
<th>Operation resource</th>
<th>Name (operation, resource)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /servers/list</td>
<td>For each server, a list of {SERVER, CPU, MEM}.</td>
<td></td>
</tr>
<tr>
<td>GET /servers/SERVER-UUID/info</td>
<td>Key-value pairs that represent the server specs also the cumulative received byte count for NICs and the cumulative transmitted byte count for NICs.</td>
<td></td>
</tr>
<tr>
<td>POST /servers/create</td>
<td>Key-value pairs for server configuration and returns the same data representing the instantiation of requested server.</td>
<td></td>
</tr>
<tr>
<td>POST /servers/SERVER-UUID/set</td>
<td>Key-value pairs that represent the server attributes to change and the name of the server to make the changes to.</td>
<td></td>
</tr>
<tr>
<td>POST /servers/SERVER-UUID/destroy</td>
<td>Returns a HTTP 204 No Content. Will do a hard shutdown of said server and delete.</td>
<td></td>
</tr>
<tr>
<td>POST /servers/SERVER-UUID/shutdown</td>
<td>Returns a HTTP 204 No Content status code. Will then perform a soft ACPI shutdown.</td>
<td></td>
</tr>
<tr>
<td>POST /servers/SERVER-UUID/reset</td>
<td>Returns a HTTP 204 No Content and soft reboots the server.</td>
<td></td>
</tr>
</tbody>
</table>

Comments

- ( - ) No way to dynamically change your availability zone
- ( - ) No means to specify non-functional requirements
- ( - ) No elasticity per server - to have elastic behaviour one needs to utilise their API directly
- ( - ) some inconsistencies with REST-style architecture e.g. the HTTP verb DELETE should be used to destroy a server
- **CPU/MEN Sizing:** GUI based interface, allowing any size or predefined templates
GoGrid

- **Communication Pattern:** Synchronous
- **Web Service Technology:** REST interface over HTTP supporting only GET and POST verbs.
- **SLA:** GoGrid provides a SLA\(^7\) with hard values that their services aim to adhere to.

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid.server.list</td>
<td>Lists all the servers in the system.</td>
</tr>
<tr>
<td>grid.server.get</td>
<td>Retrieves one or many server representations from your list of servers.</td>
</tr>
<tr>
<td>grid.server.add</td>
<td>Adds a single server representation to your grid.</td>
</tr>
<tr>
<td>grid.server.delete</td>
<td>Deletes a single server representation from your grid.</td>
</tr>
<tr>
<td>grid.server.power</td>
<td>This call will issue a power command to a server object in your grid. Supported power commands are:</td>
</tr>
<tr>
<td></td>
<td>- power on (start), power off (stop), power cycle (restart)</td>
</tr>
</tbody>
</table>

**Comments**

- ( - ) The list method returns all servers available; however it doesn’t group them by application-grouping.
- ( - ) The delete method is inconsistent with a REST-style approach. It should use the HTTP DELETE verb.
- ( - ) Appropriate HTTP verbs for each method are not specified in the documentation

---

**Enomaly - Enomalism**

- **API:**

- **Communication Pattern:** Synchronous

- **Web Service Technology:** REST interface over HTTP

- **SLA:** Enomalism is a product and enomaly does not offer a service with a corresponding SLA

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET or POST /rest/machines or /rest/machine</td>
<td>Lists the machines associated with an account. Returns a list of machine IDs</td>
</tr>
<tr>
<td>GET or POST /rest/machine/&lt;UUID&gt;</td>
<td>Perform an update on the machine (configuration?) or retrieve its machine representation</td>
</tr>
<tr>
<td>GET or DELETE /rest/hypervisor/&lt;UUID&gt;</td>
<td>Perform an action on the named hypervisor</td>
</tr>
<tr>
<td>GET or POST /rest/machine/&lt;UUID&gt;/hypervisors</td>
<td>Gets a listing of all the hypervisors installed on a machine.</td>
</tr>
<tr>
<td>GET or POST /rest/machine/&lt;UUID&gt;/clusters</td>
<td>Gets a list of clusters to which the named machine belongs to.</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/unlock</td>
<td>Unlocks a locked machine</td>
</tr>
<tr>
<td>GET or POST /rest/machine/&lt;UUID&gt;/actions</td>
<td>Gets a list of actions available on a machine.</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/suspend</td>
<td>Suspends the named machine</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/resume</td>
<td>Resumes the named machine</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/shutdown</td>
<td>Shutdowns the named machine</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/poweroff</td>
<td>Powers off the named machine</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/reboot</td>
<td>Reboots the named machine</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/create</td>
<td>Creates the named machine</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/store_xml</td>
<td>Stores a machine definition file to be used by the create action</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/manage</td>
<td>Makes the machine &quot;manageable&quot;</td>
</tr>
<tr>
<td>POST /rest/machine/&lt;UUID&gt;/actions/totaldelete</td>
<td>Completely deletes the referenced machine</td>
</tr>
<tr>
<td>POST or PUT /rest/machine/&lt;UUID&gt;/actions/provision</td>
<td>Provisions the named machine</td>
</tr>
</tbody>
</table>
Comments

- From a REST point of view the machine listing action is inconsistent. The 2nd action listed is also inconsistent. In fact, the whole "REST" interface seems merely REST-like.
- Confusing interface - to create a VM; do multiple methods have to be called? store_xml, create and provision?
- Allows access to the hypervisor
- On a public facing API actions related to the hypervisor or virtual machine manager (marked in italics above) should not be visible.
- Interface is quite granular as a result of mixing hypervisor and resource management
- Heavily dependent on libvirt
OpenNebula

- **Communication Pattern**: Synchronous
- **Web Service Technology**: XML-RPC
- **SLA**: OpenNebula is a product and has no corresponding service with a SLA

### Compute-resource oriented operations

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>one.vmallocate</code></td>
<td>Allocates a virtual machine description from the given template string</td>
</tr>
<tr>
<td><code>one.vmdeploy</code></td>
<td>Initiates the instance of the given <code>vmid</code> on the target host.</td>
</tr>
<tr>
<td><code>one.vmaction</code></td>
<td>Submits an action to be performed on a virtual machine.</td>
</tr>
<tr>
<td><code>one.vmmigrate</code></td>
<td>Migrates one virtual machine (<code>vid</code>) to the target host (<code>hid</code>).</td>
</tr>
<tr>
<td><code>one.vmget_info</code></td>
<td>Gets information on a virtual machine.</td>
</tr>
<tr>
<td><code>one.hostallocate</code></td>
<td>Adds a host to the host list.</td>
</tr>
<tr>
<td><code>one.hostinfo</code></td>
<td>Retrieves the information of the given host.</td>
</tr>
<tr>
<td><code>one.hostdelete</code></td>
<td>Deletes a host from the host list.</td>
</tr>
<tr>
<td><code>one.hostenable</code></td>
<td>Enables or disables a given host.</td>
</tr>
</tbody>
</table>

### Comments

- ( - ) On a public facing API actions related to the hypervisor or virtual machine manager (marked in italics above) should not be visible.
  - Methods that start with `one.vm*` are the ones most suitable to be exposed as external methods to allow customers manage their resources. Those with `one.host*` are particular to internal infrastructure management from a provider's perspective.
- ( - ) How can "actions" be discovered at runtime?
- ( - ) Session tracking is used throughout all methods.
Slicehost

- **Communication Pattern**: Synchronous
- **Web Service Technology**: REST interface over HTTP. Capistran and ActiveResource (RoR) pattern. **HTTP PUT** verb and Ruby wrappers
- **SLA**: State a best effort type of SLA

<table>
<thead>
<tr>
<th>Operations (root api.slicehost.com)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>s = Slice.find(xxx)</td>
<td></td>
</tr>
<tr>
<td>/slices/xxxx/hard_reboot.xml</td>
<td>Hard Reboot</td>
</tr>
<tr>
<td>(s.put(:hard_reboot))</td>
<td></td>
</tr>
<tr>
<td>/slices/xxxx/reboot.xml</td>
<td>Soft Reboot</td>
</tr>
<tr>
<td>(s.put(:reboot))</td>
<td></td>
</tr>
<tr>
<td>/slices/xxxx/rebuild.xml?image_id=x (s.put(:rebuild, :image_id =&gt; x))</td>
<td>Rebuild Image</td>
</tr>
<tr>
<td>/slices/xxxx/rebuild.xml?backup_id=x (s.put(:rebuild, :backup_id =&gt; x))</td>
<td>Backup Image</td>
</tr>
<tr>
<td>slice = Slice({'image_id':1, 'flavor_id':1, 'name':'example'}).save()</td>
<td>Create a Slice and save it (ActiveResource pattern) (It actually &quot;persists it&quot;)</td>
</tr>
</tbody>
</table>

slice.name = 'example.org'
slice.save() | Update the name of the Slice |

**Comments**

- (+-) Xen Based
- (-+) Pure Restful approach
- (-+) No CPU choice, proportional to amount of memory (CPU "cycles" = 4 times the amount of memory)
- (+) Basic DNS management, private IP networking (slice 2 slice communications), Shared IP (HA)
- (+-) Authentication for the API uses standard HTTP Authentication which uses the customers unique password as part of the URL
- Software management: Capistrano is suggested for Slice management (software deployment) with a front-end with svn to manage software on the slices.

---

8 [http://www.slicehost.com/questions/#sla](http://www.slicehost.com/questions/#sla)
**Globus Nimbus**

- **API:** EC2 WSDL. See evaluation of EC2. A WSRF interface is also supported.

**Eucalyptus**

- **API:** EC2 WSDL. See evaluation of EC2.
- **SLA:** No SLA is offered as this is a software product.
- There are plans for Eucalyptus to be integrated with the next release of Ubuntu to simplify the deployment of services between private infrastructure and EC2 compatible providers (via Eucalyptus)\(^9\)

**AppNexus**

- **API:** [wiki.appnexus.com/display/documentation/APIs](http://wiki.appnexus.com/display/documentation/APIs)
- As AppNexus does not offer an easily accessible programmable API (only command line tools), it has been excluded from this analysis.

**F5.com**

  Note: requires free login.

**Apache Tashi (Incubator)**

- **API:** [http://opencirrus.intel-research.net/tashi](http://opencirrus.intel-research.net/tashi)
- **Communication Pattern:** Synchronous, RPC
- **Web Service Technology:** Thrift RPC\(^\text{10}\). Cluster management details not available. Functionality available for one client machine.
- **SLA:** Tashi is a product and as such does not offer a SLA.

### Compute-resource oriented operations (tashi-client)

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pauseVm</td>
<td>Pauses a virtual machine instance (supply instance ID)</td>
</tr>
<tr>
<td>suspendVm</td>
<td>--instance &lt;value&gt; --destination &lt;value&gt;</td>
</tr>
<tr>
<td>unpauseVm</td>
<td>Resumes a VM instance from a paused state (supply instance id)</td>
</tr>
<tr>
<td>getUsers</td>
<td></td>
</tr>
<tr>
<td>getHosts</td>
<td></td>
</tr>
<tr>
<td>getInstances</td>
<td></td>
</tr>
<tr>
<td>createVm</td>
<td>[--userId &lt;value&gt;] --name &lt;value&gt; [--type &lt;value&gt;] --disks &lt;value&gt; [--n...]</td>
</tr>
<tr>
<td>shutdownVm</td>
<td>--instance &lt;value&gt;</td>
</tr>
<tr>
<td>getUsers</td>
<td>[--userId &lt;value&gt;]</td>
</tr>
<tr>
<td>resumeVm</td>
<td>--name &lt;value&gt; [--type &lt;value&gt;] --disks &lt;value&gt; [--n...]</td>
</tr>
<tr>
<td>migrateVm</td>
<td>--instance &lt;value&gt; --targetHostId &lt;value&gt;</td>
</tr>
</tbody>
</table>

### Comments

- ( - ) Methods do not suit an external cloud API. Very much a framework to cater for internal management of VMs hosted within a provider.
- ( - ) Light on documentation
- ( + ) Lightweight implementation
- ( + ) KVM live migration (xen?)
- ( - ) Only supports KVM currently
- ( + -) Not clear how VM cpu usage managed (KVM style scheduling - 50-50, 33 - 33 -33,... )
- ( - ) Not much management functionality for clusters
- ( - ) clustering functionality not described (yet)
- ( + ) Usage of Thrift as "IDL" will make easy to integrate with other approaches via RPC

\(^{10}\) [http://developers.facebook.com/thrift/](http://developers.facebook.com/thrift/)
### CohesiveFT

**API:**

**Note:** For the management of a VM (Virtual Server) and not the host running the VM

**Communication Pattern:** Synchronous

**Web Service Technology:** SOAP/WSDL. For management of the VM

---

#### Management Operations. A Resource/Group of Resources (source)

<table>
<thead>
<tr>
<th>Operation Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2 getCredentials</td>
<td></td>
</tr>
<tr>
<td>1.3 getLoadAverages</td>
<td></td>
</tr>
<tr>
<td>1.4 getUptime</td>
<td></td>
</tr>
<tr>
<td>1.5 getMemInfo</td>
<td></td>
</tr>
<tr>
<td>1.6 getDiskInformation</td>
<td></td>
</tr>
<tr>
<td>1.7 getLog</td>
<td></td>
</tr>
<tr>
<td>1.8 getSysLog</td>
<td></td>
</tr>
<tr>
<td>1.9 restartRubymin</td>
<td></td>
</tr>
<tr>
<td>1.9.1 Networking Functions</td>
<td></td>
</tr>
<tr>
<td>1.10 getHostIPAddress</td>
<td></td>
</tr>
<tr>
<td>1.11 getNetworkInfo</td>
<td></td>
</tr>
<tr>
<td>1.12 setNetworkInfo</td>
<td></td>
</tr>
<tr>
<td>1.13 getNetworkStats</td>
<td></td>
</tr>
<tr>
<td>1.14 resetNetwork</td>
<td></td>
</tr>
<tr>
<td>1.15 getHostname</td>
<td></td>
</tr>
<tr>
<td>1.16 setHostname</td>
<td></td>
</tr>
<tr>
<td>1.16.1 Timezone Functions</td>
<td></td>
</tr>
<tr>
<td>1.17 getTimezoneList</td>
<td></td>
</tr>
<tr>
<td>1.18 setTimezone</td>
<td></td>
</tr>
<tr>
<td>1.19 getTimezone</td>
<td></td>
</tr>
<tr>
<td>1.19.1 Nameserver Functions</td>
<td></td>
</tr>
<tr>
<td>1.20 getNameservers</td>
<td></td>
</tr>
<tr>
<td>1.21 addNameserver</td>
<td></td>
</tr>
<tr>
<td>1.22 delNameserver</td>
<td></td>
</tr>
<tr>
<td>1.22.1 NTP Functions</td>
<td></td>
</tr>
<tr>
<td>1.23 addNTPServer</td>
<td></td>
</tr>
<tr>
<td>1.24 deleteNTPServer</td>
<td></td>
</tr>
<tr>
<td>1.25 setDefaultNTPServer</td>
<td></td>
</tr>
<tr>
<td>1.26 get DaytonaNTPServer</td>
<td></td>
</tr>
<tr>
<td>1.27 getNTPServerList</td>
<td></td>
</tr>
<tr>
<td>1.28 runNTPServer</td>
<td></td>
</tr>
<tr>
<td>1.29 setNTPUpdateTime</td>
<td></td>
</tr>
<tr>
<td>1.30 getNTPLog</td>
<td></td>
</tr>
<tr>
<td>1.31 clearNTPLog</td>
<td></td>
</tr>
<tr>
<td>1.31.1 Appliance/System Control Functions</td>
<td></td>
</tr>
<tr>
<td>1.32 shutdown</td>
<td></td>
</tr>
<tr>
<td>1.40 enableAppliance</td>
<td></td>
</tr>
<tr>
<td>1.41 disableAppliance</td>
<td></td>
</tr>
</tbody>
</table>

Managing one VM appliance (Virtual Server)
• 1.42 getApplianceStatus

• 1.33 stopAppliances
• 1.34 startAppliances
• 1.35 restartAppliances
• 1.36 resetAppliance
• 1.37 getEnabledApplianceList
• 1.38 getApplianceList
• 1.39 getEnabledApplianceList

Managing Several

Comments

• ( - ) Not really an IaaS player but a Virtual Server (Appliance) builder with their own package management for configuring the VMs with software bundles.
• ( - ) (+) "Middle man" providing software packaging inside VMs rather than infrastructure
• ( - ) Compute resource provisioning is handled by the UI, not the API
• ( + ) Different choice of Virtualization technologies
• ( + ) Possibility of downloading the VMs for running in own infrastructure
• ( - ) Deployment on external infrastructure (Amazon, Elastic Hosts)
• ( - ) Api seems evolving and the semantics of the operations does not appear finalised. Managing groups of resources does not appear to be addressed yet.
• ( + ) VPN-Cubed allows the weaving of several servers in different infrastructures into one unique network. (Different product)
• ( - ) No explicit SLA (up to the external providers)
• ( + ) Could be useful input to SLA@SOI A3 Software Management Work package

Providers not considered

The following providers have not been considered for evaluation as they do not offer a public programmable API. Instead, they typically offer a user interface to access, configure and create infrastructure.

• Joyent Accelerators
• Mosso
• Cassatt Active Response
• 3Tera
• NewServers
Appendix E. Open Virtualisation Format Model

The Open Virtualisation Format (OVF; DSP0243) is a schema for describing a virtual machine or a collection of virtual machines. The initiative is the results of efforts by the Distributed Management Task Force (DMTF) who, amongst other standards, are responsible for the Common Information Model (CIM). Recently OVF version 1.0 was released and with a number of products (Kensho, VMware, Virtualbox) supporting the standard, as well as backing from large vendors (Citrix, VMware, Intel), it is looking like a promising specification in an area that is quite barren of standards, namely cloud infrastructure as a service providers. OVF can be used as a means for customers of an IaaS provider to express their infrastructural needs. Currently within SLA@SOI we use our own schema for making requests against our infrastructure service which then in turn is translated into an internal representation. What this allows us to do is to support multiple schemas and we aim to support the OVF specification. In order to support the OVF specification, we’ve had to dig deep into OVF to assure ourselves that it can accommodate one important requirement of ours - non-functional requirements. By default, OVF does not support this but if you read section 7.3 of the specification you will find that the specification has made it possible to easily extend OVF. OVF is XML, so obviously hierarchical in nature. At the core of OVF is the Envelope element and it is this element that contains:

- **References** - this is a list of external files that are required to satisfy the OVF package. An OVF package does not need local virtual disks or remote ISO files to operate. They can be supplied within the OVF document by using the "ovf:href" attribute of a File element within the Reference section.

- **Core Meta-data** - this is a list of what are known as Sections. Section elements that are allowed at this level are:
  - **NetworkSection** - each OVF envelope can have only one or opt not to include the section (zero-or-one). It is here where textual names are given to networks. These work as logical grouping identifiers. Here the attribute of Network, name is required.
  - **DiskSection** - each OVF envelope can have only one or opt not to include the section (zero-or-one). This is a listing of disks used within the OVF document. Each disk can be referred throughout an OVF document by the mandatory "diskId" attribute. Here other attributes include:
    - **fileRef** - this is an optional reference to virtual disk content
    - **capacity** - this is required and is the virtual disks maximum capacity
    - **capacityAllocationUnits** - units in which space is allocated and is optional. By default it is bytes but this can be

---

11 [http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf](http://www.dmtf.org/standards/published_documents/DSP0243_1.0.0.pdf)
12 [http://www.dmtf.org/](http://www.dmtf.org/)
13 [http://community.citrix.com/display/xs/Kensho](http://community.citrix.com/display/xs/Kensho)
16 [http://www.citrix.com](http://www.citrix.com)
specified to any unit as listed in the DMTF DSP0004\(^{19}\) specification.

- **format** - this is a URI that points to a description of the virtual disk format in use
- **populatedSize** - this is the estimated populated (space used) size of disk in bytes and is optional
- **parentRef** - a reference to a parent disk and is optional

  - **DeploymentOptionSection** - each OVF envelope can have only one or opt not to include the section (zero-or-one). This section lists a set of configuration options for the contained resources within the OVF document. Each configuration has a textual identifier (e.g. `ovf:id"large VM") which can be related to a particular configuration within the **VirtualHardwareSection**. A default configuration can be specified by setting 'ovf:default="true"'.

- **Virtual Machine Specification** - the specification of virtual machines in OVF can be presented either as singular or multiple virtual machine specifications. This allows for not only the simple but complex multi-tier applications. A **VirtualSystem** element represents one virtual machine. A **VirtualSystemCollection** element represents a list of both **VirtualSystems** and **VirtualSystemCollections**. We'll look further into these two types later.

- **Message Bundles** - this is a list of message resource bundles for zero or more locales, used for the purpose of localisation and is denoted by the element of **Strings**

### Common Sections of VirtualSystem and VirtualSystemCollection

There are three common Sections used in both VirtualSystem and VirtualSystemCollection:

- **AnnotationSection** - this section contains any number of user-defined annotations that are relevant to the particular entity. These can be pieces of information that are displayed when opening an OVF document.

- **ProductSection** - this section specifies product information for an appliance, such as:
  - **Product** - name of product
  - **Vendor** - name of product vendor
  - **Version** - product version, short form
  - **FullVersion** - product version, long form
  - **ProductUrl** - URL resolving to product description
  - **VendorUrl** - URL resolving to vendor description
  - **AppUrl** - experimental: URL resolving to deployed product instance
  - **Icon** - experimental: display icon for product
  - **Property** - this is a property bag of name/value pairs that allow for additional configuration parameters specific to the particular product. These entries are required to specify the type of the value entered (using CIM types DSP0004) and can be flagged if they can be configured by the user at installation time (using the "userConfigurable" boolean attribute).

- **EulaSection** - this section contains the legal terms for using VirtualSystem/VirtualSystemCollection. This license should be shown during the deployment of the OVF package.

---

\(^{19}\) [http://www.dmtf.org/standards/documents/CIM/DSP0004.pdf](http://www.dmtf.org/standards/documents/CIM/DSP0004.pdf)
**VirtualSystem**

A **VirtualSystem** is the element that describes one virtual machine. Contained in the VirtualSystem are a number of Sections, including the shared Sections of VirtualSystem and VirtualSystemCollection, that are specific to the element:

- **VirtualHardwareSection** - each VirtualSystem can have only one or opt not to include the section (zero-or-one). The VirtualHardwareSection has an attribute "ovf:transport" that is required. This attribute describes how environment document information can be communicated to the guest software. A VirtualHardwareSection consists of the following elements:
  - **System** - this element identifies the hypervisor/virtualisation technology that is to be used with the particular VirtualSystem. An enumeration of these types can be found in DSP1042\(^\text{20}\) (for example vmx-4 corresponds to VMWare's 4th generation virtual hardware).
  - **Items** - there can be multiple Items in this section. Each item represents a virtual hardware component (e.g. Memory, CPU). The actual type information for each virtual hardware component can be found in DSP1042 in the *CIM_ResourceAllocationSettingData* class.

A very interesting aspect is use of **Ranges** within the VirtualHardwareSection. This allows for minimum and maximum ranges for hardware specifications. For example using Ranges an OVF document can describe that a provider should run a particular appliance with a minimum of 1GB of RAM yet allow the usage of RAM expand and grow to a maximum of 4GB. This allows OVF explicitly capture aspects of elasticity that is core to cloud infrastructure.

- **OperatingSystemSection** - each VirtualSystem can have only one or opt not to include the section (zero-or-one). This section details the type of operating system installed on the virtual system. The valid operating systems that can be specified here can be found in the *CIM_OperatingSystem.OsType*.

- **InstallSection** - each VirtualSystem can have only one or opt not to include the section (zero-or-one). If this section is specified then it signals that the virtual machine needs to be booted once in order to install and/or configure the guest software. The attribute "ovf:initialBootStopDelay" specifies a delay to wait before stopping the machine after configuration is complete. If there are many VirtualSystems with this section it is allowable to run these boot phases concurrently.

**VirtualSystemCollection**

A **VirtualSystemCollection** is the element that allows one or many virtual machines in different logical grouping. This element is a list of VirtualSystems and/or VirtualSystemCollections, supporting multi-tier applications and allowing for flexible grouping. Contained in the VirtualSystemCollection are a number of Sections, including the shared Sections of VirtualSytem and VirtualSystemCollection, that are specific to the element:

- **VirtualSystemCollection** - each VirtualSystemCollection can have many or opt not to include the section (zero-or-many).

- Or:
  - **VirtualSystem** - each **VirtualSystemCollection** can have many or opt not to include the section (zero-or-many).
  - **ResourceAllocationSection** - each **VirtualSystemCollection** can have only one or opt not to include the section (zero-or-one). This describes all resource allocation requirements of a VirtualSystemCollection. This could be used perhaps where a number of appliances are to be executed on the same physical host.
  - **StartupSection** - each **VirtualSystemCollection** can have only one or opt not to include the section (zero-or-one). This section specifies how a virtual machine collection is powered on and off and the sequence in which each VM should be powered on/off.