Deliverable D6.1 (DS2.3.1): Architecture Description: Dynamic Virtualised Packet Testbeds Service

Abstract
This document is a high-level description of how Testbeds as a Service (TaaS) is designed and functions. Its purpose is to provide the network researcher with a basic understanding of the service model and how it constructs the user-defined Testbed network, as well as to provide the service provider engineer with a basic knowledge to begin requirements planning for a rollout of similar services. The strategic goal of SA2 is to incrementally enhance the feature set, improve the production reliability, and extend the reach and resolution of the distributed service.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executive Summary</td>
<td>iv</td>
</tr>
<tr>
<td>1 Architectural Overview</td>
<td>2</td>
</tr>
<tr>
<td>2 The Testbed</td>
<td>4</td>
</tr>
<tr>
<td>3 Virtual Resources</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Testbed Interaction with the Physical World</td>
<td>8</td>
</tr>
<tr>
<td>3.2 Resource Control</td>
<td>10</td>
</tr>
<tr>
<td>4 Infrastructure and Virtualisation Layer(s)</td>
<td>11</td>
</tr>
<tr>
<td>4.1 Virtualisation Software Layer</td>
<td>11</td>
</tr>
<tr>
<td>4.2 Physical Infrastructure</td>
<td>12</td>
</tr>
<tr>
<td>5 Resource Database</td>
<td>17</td>
</tr>
<tr>
<td>5.1 TaaS Resources</td>
<td>17</td>
</tr>
<tr>
<td>5.1.1 Domain Specific Language</td>
<td>17</td>
</tr>
<tr>
<td>5.1.2 Resource Templates and Instances</td>
<td>20</td>
</tr>
<tr>
<td>5.1.3 Resource Templates and Class Descriptions</td>
<td>22</td>
</tr>
<tr>
<td>5.1.4 Reservation Dependencies</td>
<td>23</td>
</tr>
<tr>
<td>6 The Resource Lifecycle</td>
<td>26</td>
</tr>
<tr>
<td>6.1 Resource Reservations</td>
<td>26</td>
</tr>
<tr>
<td>6.2 Resource Activation</td>
<td>27</td>
</tr>
<tr>
<td>6.3 Resource Instance States</td>
<td>27</td>
</tr>
<tr>
<td>7 The Testbed Control Agent and the TaaS Core Resource Manager</td>
<td>29</td>
</tr>
<tr>
<td>8 The Testbed Control Primitives</td>
<td>31</td>
</tr>
<tr>
<td>8.1 The TaaS Testbed Control Protocol and Lifecycle of a Testbed/Resource Instance</td>
<td>31</td>
</tr>
<tr>
<td>8.2 The TaaS Testbed Control Protocol</td>
<td>33</td>
</tr>
</tbody>
</table>
Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>TaaS components</td>
<td>3</td>
</tr>
<tr>
<td>2.1</td>
<td>The TaaS architecture treats all Testbed networks as graphs</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Basic representation of Nodes, Links and Ports</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>A simple physical Testbed</td>
<td>7</td>
</tr>
<tr>
<td>3.1</td>
<td>TaaS Virtual Resource Model. Templates and their instantiation.</td>
<td>8</td>
</tr>
<tr>
<td>3.2</td>
<td>UAG resource within the Testbed</td>
<td>10</td>
</tr>
<tr>
<td>3.3</td>
<td>External Port diagram</td>
<td>10</td>
</tr>
<tr>
<td>4.1</td>
<td>GN3plus project TaaS virtualisation, management and control layers</td>
<td>13</td>
</tr>
<tr>
<td>4.2</td>
<td>SA2 service pod physical design</td>
<td>14</td>
</tr>
<tr>
<td>4.3</td>
<td>The TaaS distributed architecture across the GÉANT network core</td>
<td>14</td>
</tr>
<tr>
<td>4.4</td>
<td>Pod infrastructure platforms</td>
<td>15</td>
</tr>
<tr>
<td>4.5</td>
<td>Resource mapping: VMs to servers</td>
<td>15</td>
</tr>
<tr>
<td>4.6</td>
<td>The Infrastructure, virtualisation software, and resource layering</td>
<td>17</td>
</tr>
<tr>
<td>5.1</td>
<td>Resource templates and resource graph describing the simple Testbed.</td>
<td>19</td>
</tr>
<tr>
<td>5.2</td>
<td>Example of Groovy code</td>
<td>20</td>
</tr>
<tr>
<td>5.3</td>
<td>The user and project organisation within TaaS</td>
<td>22</td>
</tr>
<tr>
<td>6.1</td>
<td>Reservation lifecycle state machine</td>
<td>27</td>
</tr>
</tbody>
</table>
Executive Summary

SA2 Testbeds as a Service constitutes a new GN3plus production service to offer experimental network resources to the network research community for the express purpose of testing novel networking concepts, at scale and across a geographically realistic European footprint. Testbeds as a Service (TaaS) will initially provide a dynamically-configurable, packet-based, virtualised network environment, which is capable of fully autonomous and independent operation under the direction and control of the network researcher. Over time, TaaS will introduce a broad range of resources and span multiple technologies.

The TaaS Architecture describes the key functional components of the service and how they interact or are used to deliver the desired Testbed networks. The Service consists of a central TaaS Core Resource Manager, responsible for interacting with the users and managing the pool of virtual resources that are allocated to the users' networks. It also provides a domain-specific grammar for describing TaaS Testbeds and a set of control primitives are also defined that allow the user to control Testbed resources through the entire lifecycle, from inception to project completion.

TaaS offers a set of virtual “resources” that the user may reserve for inclusion in a Testbed network. These resources include objects, such as: Linux-based virtual machines (VMs), Ethernet-framed virtual circuits (VCs), and novel virtual switching devices such as OpenFlow forwarding fabrics [OpenFlow]. The Testbeds are modelled in the TaaS description language as resource graphs. The TaaS resource specifications are extensible, and allow users to construct complex, object-oriented, and Testbed resources from basic atomic resources offered by providers.

TaaS is a new offering from the current GÉANT project. There is no standard, off-the-shelf software or hardware for building this particular type of geographically distributed research capability. There are a number of research projects that have explored the concept, and some commercial products that address limited aspects of the service concept, but there is currently no best practice or standard service offering for the virtualised network environments that TaaS offers. The TaaS architecture is intended to provide a rigorous and scalable foundation for generalised (flexible) virtual networks, in a provably secure fashion. Because there is no precedent, a significant amount of software development is required. TaaS utilises existing tools, where practicable, and where they do not conflict with the underlying TaaS architecture. Existing tools typically pose substantial integration challenges however, as these tools often may hold different assumptions about the underlying infrastructure and information model. The TaaS development will be an ongoing, iterative process of feature implementation and user feedback over a period of two (or more) years, to evolve a full-featured service capability over time.
This document presents a high-level description of how the TaaS is designed and functions. Its purpose is to provide the network researcher with a basic understanding of the service model and how it constructs the user-defined Testbed network, as well as to provide the service provider engineer with sufficient knowledge to begin requirements planning for a rollout of similar services. The strategic goal of SA2 is to incrementally enhance the feature set, improve the production reliability, and extend the reach and resolution of the distributed service.

The reader is encouraged to review the SA2 Testbeds Service Specification document [TAASSPEC], to understand the purpose and general interactions between the user and the Testbeds service. Other documents, to be developed as the Activity progresses, will include a TaaS Resource Guide that will provide a technical specification for available Testbed resources and the TaaS Engineering Design and Deployment plan.
1 Architectural Overview

The virtualised network environment provided by Testbeds as a Service (TaaS) is capable of fully autonomous and independent operation under the direction and control of the network researcher. Before the Testbed is described in detail, it is important to first look at the key components of the TaaS architecture, as shown in Figure 1.1, and described below.

Figure 1.1: TaaS components

- **The Testbed object** – This is the experimental environment that is created for the user at the user’s request. The TaaS provides the Testbed object to users and so is referred to as the “service object”.

1. The researcher logs in and builds a testbed description via a web GUI to the Testbed Control Agent (TCA)

2. The testbed description document is fed to the Resource Manager (RM)

3. The RM parses the document, allocates resources and sets up the Testbed control pane

4. The testbed is activated and the user controls it via the GUI
- The **Domain Specific Language** (DSL) – Testbeds (and resources) are described using a tailored, domain-specific language.

- **Virtual Resources** – These are the objects (such as virtual machines, virtual circuits, etc.) that the service allocates to user Testbeds.

- **Infrastructure** – The underlying set of physical hardware and/or software facilities from which resources are created.

- The **Resource Database** – This persistent structure holds information relating to the state of virtual resources that are created for and allocated to the user Testbeds.

- The **Testbed Control Agent** (TCA) – A user agent that accepts user input and converts that input to control messages ("primitives") that are sent to the TaaS Resource Manager for processing.

- The **TaaS Core Resource Manager (RM)** – A set of software functions that collectively act as the “Provider” agent for the Service

- The **Resource Control Methods** (RCM) – Software functions that translate basic semantic control primitives to low-level command sequences that accomplish a specific function on a particular device. These components are described in detail below.
2 The Testbed

The most fundamental concept in the SA2 Testbeds as a Service (TaaS) is the notion of a Testbed. A Testbed is a network comprising switching/forwarding elements, end systems that act as sources and/or sinks of traffic and data transport links connecting these other objects to one another. It is this Testbed network that TaaS provides to users.

Figure 2.1: The TaaS architecture treats all Testbed networks as graphs
As shown in Figure 2.1, a network can be represented using a graph theoretical model. TaaS treats Testbed networks as graph objects. In a conventional network model, each forwarding element (or end system) in the network is represented as a Node or vertex in the graph. Likewise, each transport circuit that connects two network Nodes is represented as a Link or edge in the graph. It is often the case that networks may have many edges or Links converging at a single particular Node. Indeed, there may be multiple Links existent between the same pair of Nodes. Because of this, a Link cannot be uniquely identified by the vertices it connects. This is solved at each Node by enumerating all edges that converge at that Node. These enumerated identifiers at each Node are defined as Ports. It is possible to uniquely and topologically identify a particular data transport connection by specifying a Node:Port pair. This model has three key objects: Nodes, Links, and Ports, as shown in Figure 2.2.

**Figure 2.2: Basic representation of Nodes, Links and Ports**

This conventional model is simple and intuitive. However, it is not an accurate representation of real-world networks, where Nodes and network Links represent very sophisticated objects that are constructed from physical infrastructure assets with administrative and performance attributes and parameters that must be considered in the overall object model. This is also true in TaaS. There will also be other atomic, elementary functional objects that the TaaS service model will require to be complete and which must be represented within a Testbed service semantic (such as storage objects or input or output sensors/instruments). Therefore, in order to generalise a representational model of the Testbed service object, TaaS takes this conventional network graph concept one step further to represent networks as “derivative resource graphs”.

A *derivative resource graph* is derived from the conventional graph model by treating both Nodes and Links as generalised Resources, with a common resource representation. In the resource graph model, all tangible network components are represented as Resources. All Resources are defined to have a set of (enumerated) data flow interfaces called Ports – and this applies to both the forwarding elements as well as transport circuits.
Finally, in a derivative resource graph, the Resources are linked to one another by “adjacency relations”, dimensionless information elements that simply indicate the data flow Ports that are connected to one another.

Thus, the TaaS network model is a resource graph where the nodes are TaaS Resources – all with explicit data flow Ports, and the Links of the resource graph simply indicate which data flow Ports are mated. This model not only allows the TaaS software to generalise the management and control of Testbed Resources, it also provides a model where the topology can be more easily validated to ensure the data flow characteristics specified in a potentially large and complex Testbed are realisable in practice. This model allows TaaS to define composite or abstract Resources in terms of other resources, and thus, construct large Testbed networks in a scalable and manageable object-oriented fashion.

When taken as a whole, these Resources, their Ports, and their Adjacencies constitute the Testbed data plane. The Testbed data plane is the experimental facility available for the user to control.

The functional set of agents and protocols that allow the user to acquire and manage Testbed resources, i.e. to identify the types and required attributes of resources, to schedule resource so that the Testbed is available in a coordinated manner and at an appropriate time, to initialise, reinitialise, or to query the state of those resources, etc., all of these functions are aspects of the TaaS control plane. The control plane and the control primitives are provided by the Service, and may be used by the user to control his/her Testbed data plane, as illustrated in Figure 2.3 (see Section 8 for further details).

Figure 2.3: A simple physical Testbed
3 Virtual Resources

All data plane elements within the Testbed are represented as "virtual resources". This includes processing nodes (such as routers, switches, servers, etc.), as well as network transport links (such as circuits, waves, Ethernet Framed Transport Connections, SDH circuits, etc). In TaaS, it is important to emphasise that all objects in the user's Testbed are virtual resources, allocated, as required, from a pool of infrastructure components maintained by the service. This generalised virtual resource model allows a common set of control primitives and semantics to be used to manage all Testbed resources, regardless of their class. The virtualised resource model allows the Service to maintain authoritative control of all TaaS resources and infrastructure in order to make sure it is appropriately managed and secure. Equally important, the virtual resource model allows the Service to incorporate additional novel resource classes without changes to the service software or the architecture.

![Figure 3.1: TaaS Virtual Resource Model. Templates and their instantiation.](image)

Deliverable D6.1 (DS2.3.1): Architecture Description: Dynamic Virtualised Packet Testbeds Service
Document Code: GN3PLUS14-836-76
Virtualisation, in general, is a process whereby the service object presented to the user resembles a physical device in many (most) aspects, but through software, the virtual resource is not directly tied to any particular physical infrastructure. The virtualisation-layer software decides how to map the virtual object(s) to the underlying physical infrastructure. In some instances, the virtualisation layer may allow an almost direct mapping. For instance, each Bare Metal (BM) server resource in TaaS will be mapped to an entire physical server but the BM virtual resource class is defined such that a BM instance can be mapped to any available physical server platform of the dozens distributed across the GÉANT network footprint. Likewise, a resource class may be defined that allows multiple resource instances to be mapped to a single piece of infrastructure for efficiency or operational reasons (e.g. a dozen or more lightweight VMs may be mapped to one physical server in the infrastructure.) Further, the virtualisation layer ensures that the TaaS service retains authoritative control of the underlying infrastructure and resource management processes – this allows the service to deterministically recover or to reinitialise virtual resources between user reservations.

Although virtual in name, the TaaS virtual resources constitute very real computational, switching, and transport capability. There is potential for such a powerful Testbed capability to cause problems if it is allowed to interact with or impact other non-Testbed facilities. By default, a user’s Testbed is completely isolated from all other networks. This prevents the experiment from interfering with other testbeds or production services that may be sharing infrastructure. And this isolation, in turn, prevents other unrelated Testbeds from interfering with this user’s Testbed. However, this strict isolation limits access to the Testbed, even by the owner. In order to address this, TaaS provides several implicit capabilities for users to interact with their Testbeds.

### 3.1 Testbed Interaction with the Physical World

First, VMs are created with console ports which are accessible via a web-proxy interface. The console port allows the user to see the VM boot, and provides a basic means to manage OS configuration. This is the most basic means for a user to access and interact with his/her VMs.

Second, TaaS will activate a User Access Gateway (UAG) component within the Testbed. The UAG instantiates a conventional IPv4 subnet that is configured as part of the VM activation. The UAG will provide a private IPv4 network address for each VM, and map those addresses to the resource names the user defined in the DSL. A gateway router is instantiated, and a public URL is created that provides a VPN server to access the UAG subnet. Any resource instance may be part of the UAG subnet. In v1.0, only VMs and OpenFlow fabrics will have interfaces attached to the UAG. The UAG subnet must provide access to the Internet from inside the Testbed via Network Access Translation (NAT) [NAT]. As a result, access from outside will require users to Secure Shell (SSH) to the Gateway (GW) for authentication, and then SSH from the GW to particular VMs in the Testbed. A VPN server will also be available at the GW, allowing the user to VPN into the subnet for easier access from external labs.
The third means of Testbeds interacting with the physical world is to define external Ports as part of the Testbed description. This is a standard feature of resource descriptions, which allows the user to build particular testbed topologies. Similarly, the Testbed can be defined to have Ports that transit the Testbed boundary. These ports can be concatenated with other external services, such as BoD circuits, or manually provisioned subnets in the GÉANT network core. Like the UAG port, these user-defined external ports are closely monitored, as they pose a potential security risk to or from external facilities.

The UAG resource is a ‘best-effort’ service. Inside the TaaS infrastructure, the UAG subnets are provisioned over 10 G infrastructure. However, the interfaces of the individual devices attached to the subnet typically are limited.
to 1 Gbps or less. Further, as in conventional IP networks, the public GW interface linking the UAG subnet to the outside world is shared by all UAGs. The UAG subnet is available for the user to leverage within their experiment. For instance, a VM may be allocated for an OpenFlow Controller and the OpenFlow fabrics may use the UAG subnet address to access the controller.

This UAG GW is monitored closely by TaaS, and if it is determined that the experiment is (or might be) posing a threat to external facilities via the UAG, the UAG resource will notify the NOC for further action. The suspect interface can be throttled or closed completely until the issue is resolved.

TaaS will also provide access to network attached storage across this UAG subnet. A NAS server will be available, with approximately 50 Terabytes of storage. The NAS server [NAS] will offer Network File Systems (NFS) [NFS] that can be accessed externally by TaaS users to build VM images, as required. These file systems can then be mounted during the VM boot process to make user files immediately available to VMs. An ability for VMs to boot remotely from this NAS server will be introduced in future TaaS versions which will provide substantial new customisation capabilities for Testbeds.

Note that the UAG configuration is quite complex in its actual implementation. Thus the UAG may not be included in the initial v1.0 release, but will be introduced as part of the continuous release process following the launch (in v1.1 or similar).

3.2 Resource Control

The TaaS virtualised resource model allows the user, or the TaaS provider itself – to control the Testbed resources. TaaS will introduce increasingly more sophisticated “freeze/thaw/migrate” capabilities over time, in conjunction with basic control functions that will allow a controlling agent to halt a resource, or to freeze it in order to remap the Testbed to other infrastructure or to thaw that virtual resource and restart it later, or elsewhere. This capability may also be used operationally to migrate user Testbeds away from particular infrastructure so that maintenance actions may be performed or for more efficient mapping of TaaS resources to underlying infrastructure.

The resource virtualisation that TaaS provides is intended to generalise the service object so that TaaS can offer a highly flexible and reliable service to many users simultaneously, while ensuring a secure environment with appropriate control and management of the underlying infrastructure.
4  Infrastructure and Virtualisation Layer(s)

The physical equipment and capabilities that form the raw materials on which the TaaS virtual resources are deployed is referred to as the TaaS “infrastructure”. The physical server platforms installed in the GÉANT network PoPs that are used for VMs are part of the infrastructure, as are the switches that provide the OpenFlow [OpenFlow] fabrics. The waves provisioned between the TaaS Core PoPs over which the VCs are provisioned are also considered infrastructure.

The KVM hypervisor software [KVM] that runs on the physical servers that virtualise the hardware and the OpenStack software running on a separate control server that manages the VM creation is considered the virtualisation layer software. The virtualisation layer takes the raw materials of the infrastructure and produces and presents a consistent, virtual resource model to the users.

4.1  Virtualisation Software Layer

The software that supports the virtualisation of the infrastructure is referred to as the virtualisation layer. This layer of software is typically considered to exist “above” the infrastructure and simultaneously “below” or underlying the Resource services function. This virtualisation software layer includes:

- The KVM hypervisor software that runs on the physical servers that enables the hardware to be shared by multiple guest operating system instances.
- The OpenStack software [OpenStack] running on a separate control server that manages the VM creation.
- The OpenNSA provisioning system [OpenNSA] that manages the virtual circuit provisioning.

The TaaS virtualisation layer takes the raw materials of the infrastructure and produces and presents a consistent virtual resource model to users.

The TaaS virtualisation layer employs many other software tools in addition to the software specifically developed by the GÉANT project. TaaS uses the OpenStack portfolio of tools to manage VMs. TaaS could as easily use other tools, such as VMware or Parallels [VMWARE] [PARELLELS], to create virtual machines. The user sees only the virtual machine representation defined in the TaaS VM Resource Class Definition. The TaaS virtualisation software interacts with the underlying packages, as necessary, to configure the underlying infrastructure so as to present a consistent virtual resource model to the user. Thus, the user is never exposed to the practical details required to manage the underlying infrastructure.
The virtual circuit resources share a similar underlying infrastructure within TaaS. TaaS has dedicated network transport capacity allocated between all of its core PoPs. TaaS uses the OpenNSA circuit provisioning software to manage this transport capacity and to virtualise it to create Virtual Circuits (VCs). TaaS also has data transport switching infrastructure co-located with its other service infrastructure that provide TaaS with the ability to create user-defined virtual circuit resources connecting the user’s VM resources. The TaaS resource mapping software interfaces to the OpenNSA provisioning software, using a standard, NSI-CS v2.0 [NSI-CSv2.0], provisioning protocol. Because TaaS VC infrastructure uses the NSI standards-based interface for circuit provisioning, it will be able to easily interface to external provisioning systems that also use NSI, such as the GÉANT Bandwidth on Demand service, and other similar, globally deployed NSI Connection Services. The TaaS virtual circuit resource model views the NSI transport infrastructure as part of a global inter-domain architecture. For example, the data transport Resources will allow TaaS Testbeds to integrate VMs or other resources in other TaaS domains (such as TaaS service domains operating in the NRENs) into a single Testbed. Thus, the underlying infrastructure of the transport resources must be able to extend to infrastructure outside of TaaS. (This architectural aspect of TaaS VCs is also how TaaS Testbeds may link to real-world facilities, such as a user’s own campus lab, servers, or other external facilities.)

4.2 Physical Infrastructure

The TaaS physical infrastructure is geographically distributed with other GÉANT services in the GÉANT network Points of Presence (PoPs) around Europe. The TaaS infrastructure facilities comprise a set of “Service Pods” that contain the infrastructure components that make up the VMs, VCs, and other resources.
A Service Pod is a self-contained set of physical components that is placed as a group at different geographically locations. Pods may be substantially different in terms of number of components, but they all follow the same architectural design below.

![Diagram of Service Pod](image)

Figure 4.2: SA2 service pod physical design

The TaaS Pods are designed to be scalable. The various infrastructure components are integrated so that they can be interconnected by transparent data plane transport virtual circuits. The Service Pods can incorporate numerous and varied server components, limited only by physical port availability for connecting resources and the space and power constraints of the physical location.

Figure 4.3: The TaaS distributed architecture across the GÉANT network core

The TaaS Pods are designed to be scalable. The various infrastructure components are integrated so that they can be interconnected by transparent data plane transport virtual circuits. The Service Pods can incorporate numerous and varied server components, limited only by physical port availability for connecting resources and the space and power constraints of the physical location.
The physical infrastructure pool is always controlled by TaaS, and the user is never allowed to usurp that control. It is fundamental to the TaaS service that TaaS is the sole arbiter for resource mapping to that infrastructure. The user is provided with substantial capabilities and control through the resource virtualisation, and so where a researcher deems it necessary to control an object in his/her target experimental environment, s/he must request a resource that provides a user with that level of user control. For instance, TaaS will define a resource class that is an OpenFlow-capable switch fabric [OpenFlow]. These OpenFlow switching fabric resources are provisioned over hardware switching equipment and operating software that allows TaaS to partition a single physical switch...
into several independent OpenFlowFabric instances. The virtual OpenFlowFabric resource is defined such that there is no “console” access to the resource instance, i.e., the OpenFlowFabric resource does not provide console access to the infrastructure switch operating system – all access to the OpenFlowFabric resource is via OpenFlow protocol from the user assigned OpenFlow control server. If future virtualisation software for this resource is able to partition the switch operating software such that a user could be allowed to log into his/her particular OpenFlowFabric resource without access to other resources mapped to that same physical infrastructure, then the resource class may be updated to provide this console capability. Until then, the user must rely on OpenFlow protocol to interface to the fabric. In this case however, the resource virtualisation layer can define adjunct control primitives that could interact with the resource instance via the switch console. Since these adjunct primitives are run under control of the provider – who is administratively in control of the switch – they can run commands that can be limited to specific functions on specific switch objects and thus virtualise the console capabilities.

The TaaS infrastructure may be shared by many types of virtual resource. For instance, the deployment of blade servers could be viewed as infrastructure for both Virtual Machine resources, as well as Bare Metal server resources. Indeed, a Bare Metal Server resource instance could be Reserved and Activated by the VM resource virtualisation functions in order to dynamically create additional infrastructure for the VM resources if/when needed.

A simple analogy for this overall virtual service model is that of a factory. The Infrastructure objects are the raw materials or subassemblies that are input to the assembly line in the factory. The virtual resources are the product that is manufactured and output by the factory. And the manufacturing process, the assembly line itself, is analogous to the virtualisation process. The user never sees the input materials. The products generated by the factory may be used by a consumer, or they may be components that are shipped to another factory and used as raw materials for yet another manufacturing process and product line. Thus the notions of infrastructure and virtual resources are relative terms that relate to a particular virtualisation service layer. And so the layers can be stacked – with the resources produced by one virtualisation layer acting as infrastructure for a virtualisation layer above, and so on.

In the Testbeds Service Architecture, there is nothing that prevents this multi-layer virtualisation. Further, these virtualisation properties should not be confused with the object oriented recursive construction of composite resources. The virtualisation layer that produces an atomic resource might acquire a single resource from an underlying virtualisation service and add its own resource specific configuration or partitioning to produce the atomic resource, or it might as easily construct a complex underlying composite resource as infrastructure and simply present an opaque root object as the upper layer atomic resource. The upper layer agent that requested the atomic does not ‘know’, and therefore it does not ‘care’. This ability to stack these virtualisation layers and to hide or abstract the underlying infrastructure and virtualisation process will be key to the multi-domain Testbeds services where internal provisioning processes and policies may not be publicly shared.
In summary, TaaS offers virtualised resources to the user. Those virtual resources are constructed from the TaaS infrastructure facilities. There is a sophisticated layer of software that constitutes this virtualisation layer.

Figure 4.6: The Infrastructure, virtualisation software, and resource layering
5 Resource Database

TaaS must maintain a database of both the virtual resources it has created and allocated to users, and the infrastructure facilities that were used in this process. For instance, if a user requests a VM resource in Amsterdam, the TaaS Resource Manager (RM) must be able to determine if there is an available VM in Amsterdam at the desired time, and with the desired performance characteristics. If no VMs are available, then the TaaS RM must decide if additional VMs can be created from infrastructure in Amsterdam. TaaS keeps the state of the virtual resources in a Resource Database (RDB).

In addition to an inventory of infrastructure and resources, TaaS must be able to determine the specific allocation of resources to users, and apply policy enabling those users to manipulate specific resources, as required.

It is also important to understand that the RDB structure is directly related to the processes that create and manage the service objects. The following discussion describes the RDB and the processes that define and manipulate the data structures found in the RDB.

5.1 TaaS Resources

5.1.1 Domain Specific Language

TaaS has defined a Domain Specific Language (DSL) for describing Testbed networks. The TaaS v1.0 DSL is mapped on top of Groovy, an object-oriented programming language [GROOVY]. The TaaS DSL implements an object-oriented model for defining Testbeds. The DSL description of a resource class or type is called a “template” or simply a class description. Every resource belongs to some class, and the attributes of that class of virtual resource are explicitly defined in the class template. The template acts as a plan that TaaS uses to construct the virtual resource the user requested. A resource class template may describe a resource that contains other resources (this is a “composite” resource), and those other referenced resources must also have a resource template describing their salient features.
Figure 5.1: Resource templates and resource graph describing the simple Testbed.

The process of creating a resource instance from the class template is called *instantiation*. The user assigns a unique name to each resource instance created. It is important to understand the difference between the Class name, and a resource instance name. Multiple resources of the same class may be Reserved and instantiated within a single testbed. These instances may be individually referenced by their Instance identifier/name.

This basic model treats the user Testbed as a composite resource. This Testbed resource can incorporate other resources – either defined by the TaaS service or other user-defined composite resources. This object-oriented specification model allows the user to define increasingly sophisticated and complex resources from simpler, more atomic resources.

The DSL description contains the user constraints for each requested resource, i.e. the required attributes of the resource. This text description of the user Testbed is presented to the TaaS service agent for mapping and scheduling. The DSL for a user's Testbed can be created and edited offline and saved. Upon logging into the TaaS user portal, the DSL description can be imported and interactively processed by the TaaS service agent.

The following is a Groovy code snippet describing a simple triangle topology.

```groovy
testbed {
    name = "Hosts connected in a triangle topology."

    def hosts = []
    def links = []

    3.times { idx ->
        def h1 = host {
            id = "host$idx"
            cpuCores = 3 - idx
            port { id = "p1" }
        }

        links << link {
            src = h1
            dst = hosts[2]
            id = "l1"
        }

        links << link {
            src = hosts[1]
            dst = h1
            id = "l2"
        }
    }
}
```
port { id = "p2" }
}

hosts << h1

def l1 = link {
    id = "link$idx"
    bandwidth = 1000
    rtt = 50
    port { id = "l1"; mode = "bidirectional" }
    port { id = "l2"; mode = "bidirectional" }
}

links << l1

adjacency h1.p1, l1.l1

3.times { idx -> adjacency hosts[(idx + 1) % 3].p2, links[idx].l2 }

Figure 5.2: Example of Groovy code

The DSL resembles many other languages in terms of syntactic patterns and semantic structure. It provides an object-oriented definition of resource classes (types), and a recursive capability for parsing, reserving, and activating instances of those resources. Indeed, the user's Testbed description is itself treated as a user-defined composite resource type, comprising other user-defined or TaaS defined resources. The DSL provides "iterators", which enable concise, syntactic specification of large sets of resources and their relationships to one another. It provides scoping at the resource instance level and per-instance control of resources. Thus, a Testbed can be defined using any programming editor, the DSL can be parsed and reserved, and the user can control the activation (within the reservation windows) of individual resources.

The Groovy-based DSL interpreter is an interim implementation designed to speed the deployment of TaaS v1.0. It provides the basic functionality required, but is not always perhaps the easiest grammar for non-programmers to understand. The TaaS development effort will review the effectiveness and ease-of-use of this implementation by users and network engineers (network engineers are expected to be a major user community in addition to the other researchers). It is expected that a DSL more specifically customised to network topology descriptions will be desired in future TaaS releases, rather than one that is more inclined towards a programming paradigm. For v1.0, the user should view the DSL as a stepping stone for greater capabilities as the service scales up and matures. The GÉANT project TaaS service management team will work with the user community to ensure its requirements for the grammar are reflected in future versions, and that current V1.0 Testbed descriptions are easily migrated forward to future TaaS versions.
5.1.2 Resource Templates and Instances

TaaS resources are object-oriented objects. Thus, as resources are instantiated, a tree-structure is created with a user-defined composite Testbed resource description at the root, and atomic resources at the leaves. When the Testbed is instantiated by a Reserve command, the TaaS Core Resource Manager parses the Testbed description, recursively parsing composite resources templates that are referenced within the Testbed description. This recursive processing continues until an atomic resource description is encountered. The atomic resources become leaf nodes in the Testbed resource tree. Atomic resources are typically base resources offered by the TaaS core service – such as VMs or VCs. The TaaS Core RM creates a reservation for the atomic resource instance, and then returns up the tree, completing the reservation confirmation process as it goes.

This model implies two objects: A resource “template” and a resource “instance”. Templates are a symbolic representation of a resource’s structure and data plane and control interfaces. A template is a “plan” for building resources of that class. A composite Resource is simply a template that contains other resources as internal components. An atomic resource contains no other resources. Finally, a resource instance is a fully realised example of a resource class where all templates have been instantiated, resulting in a tree of resource instances.

This template model allows a user to describe multiple “Testbeds” or other resource classes that might assist in recursively constructing larger, more complex Testbeds. Likewise, there may be a need for several users to have access to these resource templates, and to any resource instances based on them. This multi-user visibility also means that there are implicitly other users that are not allowed to see or control resources allocated by users. Collaborating users are therefore grouped under a common ProjectID.

The ProjectID object acts as the root object for three, key data structures:

- The list of users that share a common Testbed environment.
- The list of resource class templates available to users within that Project.
- The list of all resource instances that have been created by users in the Project group. (TaaS implicitly defines the TaaS provider resource templates under each Project, so that users will always see the TaaS classes.)
In order for a resource to be reserved, its internal structure must be resolved. This requires that the Template must be parsed and all children resolved. The resulting, fully resolved tree structure is the “syntax tree” for the root resource. The resource class list and the resource instance list are actually lists of recursive, tree-structured objects. So these structures may become quite large as resources are reserved and activated.

Any user assigned to a ProjectID is allowed to see the class templates defined under that Project and to manipulate resource instances under that ProjectID. Note that this AAI model provides an initial, rudimentary and
rather stiff segregation and insulation of users and projects within the overall TaaS service. Future TaaS versions will explore requirements for a more flexible and generalised model as the TaaS multi-domain and federated AAI aspects are developed.

The parsing of the resource template does not create a resource instance – parsing simply walks the templates incorporating children templates so that a fully resolved syntax tree for the root resource class is constructed. This validates all required resource descriptions. A resource instance is only created when the class has been parsed and a syntax tree has been created and that root resource class is referenced in a Reservation request. A unique name is assigned by the user to each resource instance when the resource is referenced in the class description. The resource provider (TaaS RM) will also assign a unique Resource Instance ID (RIID) when the resource is actually reserved. This allows the user to differentiate multiple instances of the same resource class. Likewise, the provider-assigned resource Identifier is unique within the overall provider RDB and may also be referenced by the user to specify a particular resource instance.

Drilling down, there are two important data structures that must be defined: a) a resource “template” or class descriptor, and b) the resource “instance” object. These objects must be persistent in order for TaaS to maintain a Testbed in active service, even when there are no authorised users logged into the system, and/or to recover Testbeds across service interruptions.

5.1.3 Resource Templates and Class Descriptions

The resource class template is an internal data structure that defines the key components of all resources:

- **The name of the Type or Class of resource** (e.g. a “Type LinuxVM {} ”) This is the identifier used to refer to a particular class (*not* the resource instance identifier.)

- **The external Ports for a resource instance of that Type** Each Port has a port name and a set of attributes for those Ports, e.g. Port P1, P2, P3 { framing=Ethernet, directionality=bidirectional }

- **The Attributes of the resource class** (e.g. “CPUSpeed=2.3 GHz”, or “Capacity=10 Gbps”.)

- **Internal resources** If the resource template describes a composite resource, the template must specify the internal topology for a resource of this type. The internal topology is defined by specifying a set of internal resources and their port adjacencies. This internal topology only refers to other resources to be included (and does not actually define those resource classes here). These are called resource “references” during the parsing process. A resource reference must specify the Type of resource, a user-specified resource Name for the resource instance that will be reserved, and the necessary parameters that define or constrain the referenced resource instances (e.g. “LinuxVM Host1 { port eth0, eth1 {} attributes {cpuspeed=2.3 GHz, location=Amsterdam } ” )

- **Adjacencies among internal resources** If the resource template describes a composite resource, the template must specify the Adjacencies among the internal resources of that class. These adjacencies specify port pairs using the fully qualified port reference 2-tuples: <resource instance Name><Port name>. These adjacencies define the topology among the internal resources of a composite template. If the
template defines external linkages (via Ports defined at the template level), the Adjacency relations must use the template class Name to fully qualify an external Port. E.g. “Adjacency Host1.eth0, Testbed.externalPort1.”

- Finally, the resource template must define the set of control primitives that are available for manipulating resources of this class These are called Resource Control Methods (RCMs). At a minimum, all resources must define the five basic methods for managing a resource instance though its lifecycle. These are “Reserve”, “Activate”, “Query”, “Deactivate”, and “Release” primitives (see Section 8.2). A resource template may define additional methods that are available for that particular class. Note that in TaaS v1.0, user-defined composite resource templates are not allowed to specify user-defined Methods. There are a number of security issues that must be resolved before this can be implemented, so for now, such user-composite resources are given a generic set of resource control methods by the TaaS parsing process.

A Resource instance is allocated when its resource class is referenced from a Reserve control primitive. The Reservation processing will locate the resource class template and perform a post-order walk along the syntax tree for that class. This traversal will visit and allocate leaf resources (i.e. the atomic resources) first, and then incrementally construct instances of each intermediate composite resource class as it reserves its way back towards the root.

5.1.4 Reservation Dependencies

At each level of the resource syntax trees, the TaaS Intelligent Resource Mapper will inspect the internal topology of each composite resource to determine the reservation dependency ordering of the resource references. Dependency ordering recognises that there may be many ways a Testbed network topology might be mapped onto the overall available resource pool and still meet the user’s individual resource constraints. Further, there are dependencies between resources that may force or prefer that resources be allocated in a particular order. For example, a Virtual Circuit (VC) resource is dependent upon knowing where its endpoint Ports must be placed. So a VC must know where the adjacent resources are placed in order to define the endpoints associated with its own Src and Dst ports. The Port Adjacency information is used to find the neighbour resource where the adjacent Port is located. The two adjacent Ports are dependent upon one another. If one has been mapped to a physical port (or Service Termination Point), then the other is implicitly mapped to that same STP. However, if neither Port has been resolved to a specific topological location, then the respective resources must be resolved first and this requires resolving any other Port dependencies. Thus the dependency processing is key to a good network layout.

In a closed virtual network, there are no base dependencies – the Testbed topology can be mapped to any homomorphic infrastructure topology as long as both meet the user’s constraints. Theoretically, any arbitrary resource could be chosen, instantiated to resolve its placement, and then work outward from one resource to another, incrementally resolving each adjacent resource. However, such selection of arbitrary resources is not always a practical choice in the real world. For example, if a VC were first to be allocated, the user would arbitrarily choose end points for the ingress and egress ports, which then require adjacent VMs to have ports at those locations (theoretically possible, but not really practical.) Some of these potential mappings make more practical sense than others. For instance, it makes more sense to first allocate the Virtual Machines to define their port locations, and then resolve the adjacent Virtual Circuits’ port locations. Mapping the VMs first resolves the VM
port locations and pins them to specific topological locations. These topological locations can then be applied to resolve the adjacent VC ports, which then constitute the endpoints when creating the VC instance. Thus in this precedence model, the VC reservation is dependent upon the VM reservations being completed first. This is generalised by assigning a precedence to each resource class.

5.1.4.1 Resource Records

All virtual resources are represented in the RDB with a Resource Record. The resource record contains a Resource ID – an identifier that is unique among all resources provided by the present Testbeds Service Provider. The Resource record also contains the resource type, the user assigned resource instance name (unique with the scope of an encompassing composite resource), and any key/value pairs describing resource attributes. Resource records are created by a Reserve() primitive invoked to reserve a resource of a particular Type or Class. Since a testbed is in fact a composite resource, the Reserve() primitive will evaluate the resource template and recursively visit and evaluate and reserve any children resources reference from the composite. Thus a resource tree is constructed.

When a Reservation request is received for a particular resource class, the TaaS core Resource Manager (RM) will query the Resource Database for any static (i.e. pre-provisioned) virtual resources that meet the user’s constraints. This query is non-trivial when one considers there are multiple dimensions that define the constraint space: resource type and associated performance parameters that the user might specify, the user’s authorisation credentials (dictating which resources are visible or available to the user), and temporal scheduling windows.

If no existing, static resources meeting the user constraints are found in the RDB, the RM will attempt to dynamically create the required resource from underlying infrastructure. For instance, a user may request a VM with Mem=4GB in Amsterdam. If there are no such VMs found in Amsterdam, or none of those found are available at the time and for the duration requested, the RM will invoke a constructor method that will locate infrastructure for that class and will create a VM that meets the specified criteria. In this example, it would request that OpenStack defines and instantiates a VM with 4 GB of memory on the compute node (the VM platform) in Amsterdam. If the compute node does not have the memory available, the RM will look for an additional platform in Amsterdam that can be allocated to OpenStack and initialised as a compute node, and then, the RM will request a VM from that new piece of infrastructure. A similar process is invoked for Virtual Circuits when a VC is dynamically reserved across the infrastructure.

Upon creating a resource instance, the TaaS RM assigns a TaaS Resource Instance Identifier (RIID). The RIID is unique within the scope of a TaaS service. For example, within the SA2 TaaS service, no two RIIDs will ever be the same. (This uniqueness feature will be important in TaaS v2.x, when resources from multiple providers will be found within a single composite resource instance.) The Resource Instance record in the RDB specifies the as-built attributes of a resource instance. For example, the user’s Reservation request may have only specified a few of the possible constraints for a resource. By Reserving a conforming resource instance, the unspecified parameters will have been defined. These resulting parameters are then returned with the Reservation confirmation as the “as-built” attributes.
The Resource record in the RDB points at its parent Resource RIID. Therefore, all RIID records that point to the same parent RIID are the internal [children] resources of a composite resource instance of that parent RIID. The resource instance tree is constructed through the Resource instance records in the RDB.

Each resource is allocated on a scheduled basis. This time window is represented by a “reservation” record in the RDB. The Reservation record specifies a Resource Instance ID, the start time and date of the reservation, the end time and date of the reservation, the user ID that reserved the resource, and the project under which the resource is allocated.

A resource may be statically pre-allocated from infrastructure before it is requested. These are static resources. Other resources will be allocated dynamically, on demand, from the infrastructure. While the latter is more flexible, it often requires substantial time for initialisation. In order to reduce this set-up time, some resources may be pre-allocated and simply assigned to Testbeds when requested. Dynamic resources will be de-allocated when they are released, and so typically, there is a one-to-one relation between a dynamic resource instance and its reservation record. In contrast, a static resource is not de-allocated between reservations (although it is typically cleared or reinitialised to prevent information leakage from one user to another) so there may be multiple reservations pointing to a single, static RIID.

The Resource Instance record also contains the current lifecycle state: “Reserved”, “Active”, or “Released”. A resource may not be immediately activated upon the start of the reservation, therefore, at the start time and upon resource allocation, the resource is marked as “Reserved” and points to either a normal reservation record or is reserved to the “available” pool.

The resource record also contains pointers to internal infrastructure mapping information that is maintained by the virtualisation layer software. This information is not generally available to users, and is typically only used by the virtualisation methods and the service operations control and monitoring functions.

Each species of Resource class has a unique infrastructure table, and there is no standard mechanism for how the virtualisation layer will deal with each species. For instance, the infrastructure used for Virtual Machines may require different information, depending on the software managing the equipment. This is also the case for Virtual Circuit infrastructure. In general, new infrastructure is inserted using species-specific tools to add/modify/remove components into a particular infrastructure pool.
6 The Resource Lifecycle

6.1 Resource Reservations

A resource is allocated when it is reserved, although it generally is not actually instantiated until the data plane is Activated. This allows the resource instance and the required infrastructure associated with the resource instance to be accounted for in future resource management and scheduling, even though the infrastructure may be serving other resources at the same time. As a result, resources may not be actually instantiated until it is time to Activate them in a data plane.

During its existence, a resource can transition through several states: It is initially “available” when it is created and entered into the RDB – such as would be the case for pre-provisioned static resources. When the resource is selected and allocated to a particular user, a reservation record is created, stating the window for that reservation. Although a particular instance is individually identified in the RDB calendar, the actual virtual resource instance that the user will acquire may still not exist, as the resource does not actually belong to the user until the scheduled start time. The point at which the virtual resource is actually realised is left to the resource-specific methods, the implementation, local policy, and in some cases, user control. The resource reservation is created and exists from the point of reservation confirmation. The actual resource instance is only required to exist if/when the Activation primitive is successfully confirmed.

Figure 6.1: Reservation lifecycle state machine
There is a calendar that contains the start-time events for reserved resources. As the start time arrives, the resource is Activated, which will instantiate the resource (if not already existent) and moves it to the “Active” state. The resource must be activated by the user TCA by invoking an Activate() control method for that instance. If the Activate is received prior to the scheduled start time, the resource instance is marked for “auto-activate”, and will be automatically Activated when the start time arrives. If no Activate is received prior to the start time, the resource will remain in a reserved state until an Activate is received. If an Activate is received after the scheduled start time, the resource is immediately Activated. Essentially, a resource must be both a) activated by the TCA, and b) within its schedule service window, in order to become Active.

6.2 Resource Activation

The Activation process is not necessarily quick. For instance, a VM may take a minute or more to initialise (boot up). There is also no reliable means to predict the Activation time. The Activation process is not required to begin until both conditions are met. This means the user should expect a delay after the scheduled start time before the resource becomes active. Once the ActivateConfirm has been acknowledged to the user, the activation has been completed and the user will have use of the resource until the scheduled end time. Upon arrival of the scheduled end time – but not before – the resource instance will be unceremoniously Deactivated and Released by the TaaS scheduler. There is no warning or grace period provided, it is the user’s responsibility to gracefully close down and Deactivate the resource before the scheduled end time if an abrupt halt is problematic.

The user TCA may send a Query() control primitive at any time. Like every control primitive, the Query primitive is keyed to a specific Resource Instance ID. The Query() looks this RIID up in the RDB and returns the lifecycle state of the instance – “Reserved”, “Active”. If the RIID does not exist or is not allocated to the requesting user, the Query returns “Null” – unless the user’s AAI credentials are of sufficient power to allow them greater visibility. (Like all primitives, the Query() operation is governed by authorisation credentials. Therefore, only properly authorised agents may learn about resources that have not been allocated to their project. As a result, super-user operations or monitoring agents could see privileged information, whereas ordinary users will see only their own allocated resources.) Further, if the Query does locate the RIID, and it is in the Active state, the Resource Control Agent (RCA) will return a more detailed resource status block as defined for that specific resource type. The TCA may interpret the state according to published documentation for that resource. Note, that the Query does not enquire into the “internal” state of the resource – i.e. any state created by the user’s experiment. This is a user function and only becomes part of the control plane protocols if/when the user specifies a new resource class that has specific control methods defined to carry out in-depth interrogation.

6.3 Resource Instance States

A resource instance exists in one of three basic “stable” states: Unallocated, Reserved, and Active. TaaS informally implements a “Released” state for reservation in order to allow a user to Query an expired or released reservation and determine that terminal state. A Released reservation will be dropped from the RDB according to locally defined aging-out value.
A resource may be deactivated – taken out of service without releasing the reservation – by the Deactivate() control primitive. The resource instance reverts to the “Reserved” state when it is deactivated. The resource may subsequently be re-Activated, as long as it is still within the scheduled window, or it may be subsequently released altogether.

Once Activation is successfully completed, and the resource instance is marked as “Active”, the resource instance is then available to the user for his/her operation.

When the resource is no longer required by the Testbed, the TCA may issue a Release primitive that releases the resource back to the Available pool. The resource instance becomes undefined when it is released – it may be simply marked as Unallocated, or it may be completely de-materialised.

All reservation state transitions are logged to a history file. This log may be examined separately to reconstruct the chain of events or to provide usage accounting.
The Testbed Control Agent and the TaaS Core Resource Manager

The Testbed Control Agent (TCA) is the user agent that interacts with the TaaS Resource Manager to manage a Testbed. The TCA (broadly construed) is any agent representing the user, regardless if that agent is a GUI or simply repeating human directives or some more-intelligent, automated agent acting on its own volition. The GÉANT Testbeds Service provides a user TCA in the form of a web-based Graphical User Interface (GUI), where the user can login in order to define resources, to reserve/activate/query those resources, or to issue other control primitives defined for those resources.

The user TCA and the TaaS Resource Manager (RM) – are, in many ways, similar agents. Both agents must be able to parse DSL template descriptions. The TaaS provider side must be able to describe and build the resource classes it offers, many of which may be composite resources. Likewise, the user TCA must be able to parse resource templates, since this is the way the user defines a Testbed. Further, the user may define many resource classes in order to simplify complex Testbed descriptions. Thus, both the user TCA and the TaaS RM must be able to parse DSL and construct syntax trees, and then walk these trees, as necessary, to manage them.

There are two, key differences between the user TCA and the TaaS RM. First, the user TCA typically has no actual resources of its own, and does not act as a resource provider to other user agents. Conversely, the TaaS Resource Manager has an array of defined Resource classes – some atomic, some composite – and the associated virtualisation software and underlying infrastructure. The provider’s primary task is to accept resource requests from other (user) agents. Second, the user TCA has a user interface – perhaps a GUI or an interactive CLI or shell interface. The provider side has no such interface (or only uses such an interface to define the offered resource classes or to monitor the infrastructure and resources).

In the basic process of creating a Testbed, the user defines at least one composite resource class – his/her Testbed. And in that same basic model, the TaaS provider agent offers a selection of atomic resource classes, such as VMs or VCs. In the basic model, the user defines a user-side composite Testbed class that contains a number of resource instances that are classes offered by the TaaS core RM. When the user commands the TCA to Reserve an instance of the Testbed class, the TCA will parse the user’s Testbed class template and Reserve all the specified internal “children” resources. The TCA recursively processes each child resource until it encounters a class that has no children defined, i.e. an atomic resource template. The TCA then issues a Reserve request to the provider agent. The provider processes the Reserve request just like the TCA processed the user’s initial Reserve request. The provider allocates the resource requested and returns a Resource Instance Identifier (RIID). The similarities between the user agent and the provider agent are more interesting. Both must process Reserve() requests in this recursive fashion, which means that a provider agent may receive a Reserve request...
for resources for which it does not have all of the resources necessary to fulfil the request. The situation may be that the provider does not have enough infrastructure to create resources to meet the request, or the resource requested was a composite resource that contained children classes the local provider does not offer. These last scenarios are the basis for “multi-domain” Testbed services.

The multi-domain extensions for the Testbeds will allow a provider to acquire needed resources from another secondary provider. The secondary resources can be used as infrastructure to the primary virtualisation services – thus the primary provider can “re-sell” resources acquired elsewhere to the user requesting those resources. There are a number of issues that must be addressed in the multi-domain case that we can gloss over in the single domain environment.

TaaS 1.0 does not yet offer multi-domain services, therefore the TaaS 1.0 service can make some simplifying assumptions about the resource processing. First, when the user RM is instructed to Reserve an instance of a resource class, the RM must be able to locate a class template for that class within a user local library. This class template should indicate the provider(s) associated with that class. In the multi-domain scenario, this requires a directory of providers and the classes offered by those various providers, which, in turn, requires a scalable process for constructing that directory. A mechanism allowing provider RMs to announce class templates that it will offer is required, followed by a scalable authorisation process for acquiring those resources, etc. These multi-domain issues will be addressed in TaaS v2.0.

For TaaS v1.0, a number of decisions have been made in order to simplify the process. First, the user RM only interacts with one provider RM – the TaaS core Resource Manager – and a pointer to the TaaS RM web service is statically configured in the user TCA when it starts.

Second, as the resources may have a large catalogue of defined control methods, the TCA is responsible for formatting those primitives and other unsolicited notifications (such as a failure of a resource). The GÉANT Testbeds Service provides a user TCA that can perform these functions. A user is free, and indeed encouraged, to develop his/her own agents to interact with the TaaS provider. For example, such an agent might provide a Network Operations Centre monitoring and control GUI that is tailored to the particular Testbed being managed.
8 The Testbed Control Primitives

Each resource class has a well-defined set of web service Methods that define the specific, high-level functions that a user can perform on a resource. These Methods are defined as part of the resource class specification. For instance, the Reserve() function is a control primitive, as is Activate() and Deactivate(), etc.

In order for TaaS to provide a minimum uniform functionality for all resources that allow them to be scheduled and turned up or down, or to be queried for their respective state, TaaS requires that all resource classes must implement five basic Methods: Reserve(), Activate(), Query(), Deactivate(), Release(). Each class of resource is free to define other additional primitives that may be unique to its class. User classes are not currently allowed to define new Methods (as these may create a security hole).

8.1 The TaaS Testbed Control Protocol and Lifecycle of a Testbed/Resource Instance

A Testbed is described by a textual DSL document. This document uses a domain specific language to describe the network, and contains information about a) resources that comprise the Testbed network, b) the data flow ports associated with each of those resources, and c) the adjacency relations among those data flow ports that defines the Testbed topology.

The resources are organised in a tree structure, with the user Testbed as the root “Resource”. Typically, the root Testbed is constructed incrementally through a sequence of interactions between the human researcher and a GUI front end to the TCA.

As introduced in Section 6, the root Testbed resource, and the other resources recursively included, goes through a sequence of states from beginning to end, known as the resource “lifecycle”. The following describes the basic sequences and states that the Testbed may transition. Note that a “Testbed” network and a “Resource” are essentially the same type of objects. A “Testbed”, being a term applied to the user’s top-level composite resource defined and instantiated under a particular project. Testbeds and other composite resources may be saved as a resource class and referenced from other Testbeds. As a result, the lifecycle of a Testbed also describes the lifecycle of its constituent resources.

In order to understand the lifecycle of the Testbed, it is useful to have an overview of the Testbed Control Protocol (TaaS.TCP). The control protocol is a set of primitive functions between the various agents that allow these agents to effectively manage a group of resources.
The TCA creates a root Testbed object. From the user’s perspective, this is where the process begins. In a typical exchange, a user may define one (or several) Testbed objects from the GUI front end. This Testbed object is simply an empty resource description. Through interaction with the user, this Testbed object will be populated with a set of resource descriptions, drawn from a well-known library of resources. This will define the data flow port relationships. At any point, the user may ask that the Testbed as a whole be Reserved() or Activated(), or that specific resources be reserved and/or activated. Thus the lifecycle process of the Testbed is the same as the lifecycle process of any other resource instance.

The lifecycle of a resource instance is characterised by a set of states that the resource reservation may be in at any time, and a set of control primitives that move the resource from one state to another. These reservation states are managed by the Resource Manager. The states consist of “stable” states that a reserved resource may occupy for an arbitrary period of time, and “transient” states that indicate a transition is occurring from one state to another, and are expected to be short lived. (Some transitions may incur a time delay sufficient to be observable by the TCA or other automated agents.)

As previously mentioned in Section 6.2, these stable states are:

- **Null** – This is a state maintained by the TCA, which indicates that the resource instance described in the Testbed has not yet been reserved. From the RM’s perspective, if a resource in the RDB is “null” it means it is available.

- **Reserved** – In the TCA Testbed description, Reserved means the resource has been presented to a Resource Manager as part of a reservation request, and a resource instance has been selected and allocated to the Testbed. In the RM space, Reserved means a resource has been allocated to a TCA.

- **Active** – The resource has been successfully placed into service and is available for use by the Testbed.

Transient states include:

- **Reserving** – The resource is in the process of being reserved. Since some reservation requests (particularly composite resources) may take a significant amount of time to complete (or to fail), a status query for such a resource should be able to reflect a slow, but nevertheless transient, condition.

- **Activating** – The provisioning or initialisation of a resource instance may take several minutes, thus tracking such a transient state is necessary.

- **Deactivating** – De-activating a resource (transiting from Active to Reserved).

- **Releasing** – Termination of a reservation.

The Resource Manager, which owns the resources that it allocates, is authoritative for the state of the resource reservation. However, once a reserved resource has been instantiated and placed into service (the resource is in the “Active” state), the detailed resource specific state of that instance is managed by Resource Control Methods (RCM). The detail states of a resource class may vary substantially from one resource class to another, but all resources will transit the standard reservation lifecycle. Thus, any agent wishing to learn the status of a
resource should first query the RM to learn the state of the resource reservation. If/when the resource is active, the query will return the detailed state of the resource.

8.2 The TaaS Testbed Control Protocol

The control protocol consists of a set of basic functions (“primitives”) that occur between the TCA, the RM, and the RCAs. The basic control protocol consists of request/response messages that move resources through their lifecycle, detailed, as follows.

Reserve() The TCA sends this primitive to the provider’s RM to request that resources meeting the accompanying constraints be reserved. The primary information elements passed with the Reserve() request is a set of selection constraints for the RM to apply to the RDB to identify one or more suitable resources that will meet the user’s requirements. The Reserve() request will also carry AAI information that identifies the client entity on whose behalf the resources are being requested – i.e. the user.

The Reserve() request does not need to specify selection constraints for all resource attributes (for instance, the “location” constraint may not have been specified by the user) and thus, the RM is free to define (“fill in”) a location value as a default or provider-specified selection constraint to the user’s primary constraints. Once selected and reserved, a resource’s attributes are fully defined (the fully resolved attributes of the reserved resource are the “as-built” attributes). After conforming resources are identified within the RDB, the RM will select/allocate the appropriate resource and mark it as allocated to the requesting user, set the state to “Reserving”, and invoke the Reserve() function defined for that specific resource type for each of the selected resource instances. Upon completion, the RM will update the resource state to “Reserved” and pass the as-built information back to the TCA.

ReserveResponse() When the RM has completed processing of the Reserve() request, the RM sends a ReserveResponse() message back to the requesting TCA. The ReserveResponse() contains the RIID, the Reserve status/error code, the reservation state, and the “as-built” attributes for each individual resource successfully reserved. Each selected resource will have a unique Resource Instance ID (RIID) that identifies the resource instance within the scope of that Resource Manager. The RIID will be used in subsequent control primitives to indicate the target resource instance. If some resources were unable to be reserved, a “Null” resource entry is returned that contains a null RIID, a Null state, and the set of selection constraints that were searched but that did not produce a quantity sufficient to completely fill the request. (Note that the Null resource does not indicate why there were not enough resources to meet the request – just that it was unable to fill the quantity desired. The null entry simply contains the exact same constraints that the TCA specified and the RM used to search the RDB. Nothing more.) If the full quantity is met, the null descriptor is omitted from the response.

The Reserve() function acts like a constructor to instantiate the resource instance if it does not already exist. If a resource is a composite resource, then any subordinate resources will also be recursively reserved.

Activate() The Activate() primitive is sent from the TCA to the RCA, when the TCA wishes to place a resource in service. Since resources are scheduled, the resource cannot be instantiated for this TCA until the reservation start time has arrived. For example, a Testbed may require that all or some specific resources be allocated and successfully reserved before the potentially time-consuming process of instantiating them should begin (time
The Testbed Control Primitives

needed to boot-up VMs, etc.). This will help to ensure that, at the scheduled start time, a coordinated set of resources has been allocated and can be activated to place them into active service. During the activation process, a resource should show a state of “Activating”, and upon successful completion, the state of the resource should show “Active”.

Note that the capabilities that a resource exhibits when Active are dependent upon the nature of the resource type as it was defined. The Active state means that the resource is now under control of the TCA (i.e. the user). The Resource Control Methods may implement additional, resource-specific primitives. These resource-specific primitives will be defined in the Resource Guide document that describes the TaaS Resource capabilities.

**ActivateResponse()** This message is sent from the RCA to the requesting TCA, and returns a resource entry for each resource that was requested to be activated. The resource entry contains the RIID, the Activation status code (used to indicate an error code if the activation was unsuccessful), the resulting Reservation State (“Active” if successful), and the communications address of the RCA.

**Deactivate()** This primitive simply removes the resource from active state, but retains the reservation. It does not release the resource back to the RM's available pool. Upon completing the deactivation process, the resource is marked “Reserved”. This is not the same as halting a server VM or Releasing the resource. This primitive places the resource back in the Reserved state, and it may be reactivated at a later time. A resource may be deactivated for operational or maintenance reasons (particularly if the resource is reserved for weeks or months) without triggering alarms in the monitoring system. If/when re-Activated, a resource will be reinitialised, as if it were activated for the first time.

**DeactivateResponse()** From the RM to the TCA, this message carries the RIID, the Deactivate() status code (for errors), and the resulting Reservation state (should be “Reserved”).

**Release()** This is the complementary primitive to Reserve(). It releases the resource instance back to the RM’s available pool and marks the reservation as released. The release functionality is expected to clean up any hanging conditions, including a recursive decent of the resource tree to properly release and clean up all subordinate resources. Upon completion, any previously allocated resources in the tree should be back in the RDB, their scheduled reservation removed from the calendar, and ready to be re-allocated.

The Release() function is the euphemistic “nuclear option”. It is expected to be able to recover any resource from any state back to a known “null” state (available) in the RDB. Thus, if any agent cannot otherwise gracefully resolve an unexpected condition with a resource instance, a Deactivate/Activate sequence can reinitialise the instance. If this is unsuccessful, a total Release() should be able to destroy the reservation and any dynamic instance information, and return that instance to the RDB in a pristine state.

**ReleaseResponse()** Sent from the RM to the requesting agent, this response contains the RIID, and the Release status/error code.

While the Release() should never actually fail, this cannot be guaranteed. Any problems should not be presented to the TCA, but rather to the RM or the RM operations management for resolution. So a Release() fail should throw an operational warning to the service management team.
**Query()** This primitive is used to request certain resource states to be returned. The simplest form of Query would return the reservation state of a resource – Reserved, Active, etc., but more sophisticated Queries could return a more detailed dataset tailored to the particular resource instance. The Query() function should be prepared to walk the resource tree of any composite resource instances. However, this may be sensitive to authorisation policy.

**QueryResponse()** This message contains the RIID, the Query status code, the reservation state, and a Query status block. The QSB contains any internal status data structures returned from a query, and reflects (as appropriate) the recursive visibility of internal status blocks for subordinate resource instances.

The five primitives (Reserve, Activate, Deactivate, Release, and Query) constitute the common control protocol for the Testbed service. A resource may define additional primitives that are implemented by the Resource Control Method(s) (for instance a Halt() primitive or a Reboot() for a Virtual Machine) that allows the master Testbed Control Agent to perform specialised control functions for particular resources. However, since these functions are specialised, it is up to the TCA to understand what these primitives do. In addition, these resource-specific control primitives should be well documented as part of the Resource Specification and their introduction into the RDB.

A typical lifecycle would comprise the TCA creating a Testbed object, and then, via interactive human guidance, iteratively reserve resources for the Testbed. The user could then indicate which resources will connect to other resources to define the Testbed topology by defining the port adjacencies. The user would then direct the TCA to Activate the Testbed. The TCA would iteratively resolve any necessary dependencies in the topology graph, and issue Activate() primitives for each resource. As the root TCA, and the Testbed editor GUI, the TCA can simply display the resulting Activation status for each resource. The user may then direct the TCA to acquire additional resources, define the interface adjacencies, and then have those new resources Activated. Or the user could Release some resources that are no longer needed.
The Testbeds Service architecture is a continuing work. The overall vision of a virtualised environment is well understood and has been presented in the preceding material. However, while the high-level concepts are well understood, how these are best implemented functionally is perhaps less straightforward. For instance, the DSL to describe testbeds is a necessary feature of v1.0, but it need not be the final form of a DSL for describing testbed networks. There are many other existing tools, frameworks, and DSLs that have evolved from similar efforts. The use of one of these over the others is not critical, as long as there is a common service semantic underlying their use. As a result, the present DSL for v1.0 may be complemented by others over time.

In light of this, the initial implementation of the GÉANT Testbeds Architecture will focus on the key architectural components, and some of the more specific features may be delayed until a later update or the next major revision.

The Testbeds Service 1.0 will provide the following key features:

- A Graphical User Interface (GUI) that allows the user to navigate the testbed control functions and to interactively control the user’s testbed(s).
  - A graphical testbed editor will be forthcoming but not in v1.0 (The Testbeds development team believe this is an important usability feature and is a high priority development.).

- A Command Line Interface for issuing control primitives to the Testbed Service.

- A basic set of Resource classes and attributes (these are described in detail in the Testbeds Service Resource Guide). The basic resources are:
  - Host – a virtual machine running Linux with specific memory and disk attributes.
  - Link – a virtual circuit that can terminate at any Port within the Testbeds service.
  - Composite resources – contain references to other resources.
  - Host and OpenFlow resources will be available in Copenhagen, Bratislava, Amsterdam, and Ljubljana. Other locations will follow in v1.1+.
  - All resources will support multiple Ports – some resources may offer a fixed number of Ports as part of v1.0.
  - All resources will be scheduled. In addition, reservations are to be reliable to a level consistent with other production services.
  - Link resources will be Port-based in v1.0 (i.e. VLAN based Ports will be introduced in v1.1+). The “port based” Link resources will allow the user to create VLANs within the Links that will be carried transparently within the Testbeds Service. This allows the VMs to tag frames as necessary, and allows OpenFlowFabrics (OFF) to have free access to VLAN tags within their [Port-based] flowspace.

- The Testbeds DSL will provide the following basic functionality:
  - Define resource classes (i.e. testbeds, and other resources).
  - Reserve resources (testbeds).
○ Attribute specification of resources at reservation time.
○ Iterators

- The basic common API will consist of the following “Gang of Five” functions:
  ○ Reserve
  ○ Activate
  ○ Query
  ○ Deactivate
  ○ Release

- The Service will allow basic users to sign in essentially anonymously to define and experiment with small-scale testbeds.
- The Service will register more advanced users and allow these users to define and hold large more persistent testbeds. Specific resource allocation policy is to be determined in the near future.
- An “admin” functionality will be provided to manage users and projects.
- Initial v1.0 total user resources will be limited to approximately eight virtual Hosts per location (~40 to 60 VMs overall), and approximately five OpenFlow Fabrics per location (25 to 35 OFFs overall.) This capacity will be increased over time as software is refined, more VM flavours are defined, and additional hardware infrastructure is deployed.
References

[BareMetal] https://wiki.openstack.org/wiki/Baremetal
[OpenFlow] https://www.opennetworking.org/sdn-resources/onf-specifications/openflow

Glossary

AAI Authentication and Authorisation Infrastructure
AMS Amsterdam
BM Bare Metal
BRA Bratislava
BoD Bandwidth on Demand
CPU Central Processing Unit
CRM Core Resource Manager
CSF Central Server Facilities
DSL Domain Specific Language
FRA Frankfurt
GB Gigabyte
GN3 GÉANT Network 3, a project part-funded from the EC's Seventh Framework Programme under Grant Agreement No.238875
GN3plus GÉANT Network 3 plus, the GÉANT project following GN3
### Glossary

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>GW</td>
<td>Gateway</td>
</tr>
<tr>
<td>IPv4</td>
<td>Version 4 of the Internet Protocol (StB IETF)</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>L1</td>
<td>Level 1</td>
</tr>
<tr>
<td>LJU</td>
<td>Ljubljana</td>
</tr>
<tr>
<td>NAS</td>
<td>Network-Attached Storage</td>
</tr>
<tr>
<td>NAT</td>
<td>Network Address Translation</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>OFF</td>
<td>OpenFlow Fabrics</td>
</tr>
<tr>
<td>PoP</td>
<td>Point of Presence</td>
</tr>
<tr>
<td>RCA</td>
<td>Resource Control Agent</td>
</tr>
<tr>
<td>RCM</td>
<td>Resource Control Methods</td>
</tr>
<tr>
<td>RDB</td>
<td>Resource Database</td>
</tr>
<tr>
<td>RIID</td>
<td>Resource Instance Identifier</td>
</tr>
<tr>
<td>RM</td>
<td>Resource Manager</td>
</tr>
<tr>
<td>SA</td>
<td>Service Activity</td>
</tr>
<tr>
<td>SDH</td>
<td>Synchronous Digital Hierarchy</td>
</tr>
<tr>
<td>SSH</td>
<td>Secure Shell</td>
</tr>
<tr>
<td>SW</td>
<td>Software</td>
</tr>
<tr>
<td>TaaS</td>
<td>Testbeds as a Service</td>
</tr>
<tr>
<td>TACACS</td>
<td>Terminal Access Controller Access Control System</td>
</tr>
<tr>
<td>TCA</td>
<td>Testbed Control Agent</td>
</tr>
<tr>
<td>UAG</td>
<td>User Access Gateway</td>
</tr>
<tr>
<td>VC</td>
<td>Virtual Circuit</td>
</tr>
<tr>
<td>VLAN</td>
<td>Virtual Local Area Network</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machine</td>
</tr>
<tr>
<td>VOX</td>
<td>Virtual OpenFlow Switch</td>
</tr>
<tr>
<td>VPLS</td>
<td>Virtual Private LAN Service</td>
</tr>
<tr>
<td>VPN</td>
<td>Virtual Private Network</td>
</tr>
<tr>
<td>VR</td>
<td>Virtual Router</td>
</tr>
</tbody>
</table>

### Key Testbed Terminology:

**Active**

The resource has been successfully placed into service and is available for use by the Testbed.

**Activate()**

The Activate() primitive is sent from the TCA to the RCA when the TCA wishes to place a resource in service.

**ActivateResponse()**

This message is sent from the RCA to the requesting TCA and returns a resource entry for each resource that was requested to be activated.

**Activating**

The provisioning or initialisation of a resource instance may take several minutes, thus tracking such a transient state is necessary.

**Adjacencies**

Dimensionless information elements that indicate which data flow Ports are connected to one another.
Deactivate() This primitive simply removes the resource from active state, but retains the reservation. It does not release the resource back to the RM’s available pool. Upon completing the deactivation process, the resource is marked “Reserved”.

DeactivateResponse() From the RM to the TCA, this message carries the RIID, the Deactivate() status code (for errors), and the resulting Reservation state (should be “Reserved”).

Deactivating De-activating a resource (transiting from “Active” to “Reserved”).

Link Each transport circuit that connects two network Nodes.

Nodes Enumerated identifiers.

Null This is a state maintained by the TCA, which indicates that the resource instance described in the Testbed has not yet been reserved.

Ports All edges that converge at a Node.

Query() Used to request certain resource states to be returned.

QueryResponse() This message contains the RIID, the Query status code, the reservation state, and a query status block.

Release() This is the complementary primitive to Reserve(). It releases the resource instance back to the RM’s available pool.

ReleaseResponse() Sent from the RM to the requesting agent, this response contains the RIID, and the Release status/error code.

Releasing Termination of a reservation.

Reserve() The TCA sends this primitive to the provider’s RM to request that resources meeting the accompanying constraints be reserved.

Reserve(d) The resource has been presented to a Resource Manager as part of a reservation request, and a resource instance has been selected and allocated to the Testbed.

ReserveResponse() When the RM has completed processing of the Reserve() request, the RM sends a ReserveResponse() message back to the requesting TCA.

Reserving The resource is in the process of being reserved.

Resource Objects allocated to the service by the user.

Resource Control Software functions that translate basic semantic control primitives to low-level command sequences.

Methods Accomplish a specific function on a particular device that accomplish a specific function on a particular device.

Resource Instance Identifier A unique ID assigned to a resource instance.

Resource Manager Authoritative for the state of the resource reservation.

Testbed Network comprising switching/forwarding elements, end systems that act as sources and/or sinks of traffic, and data transport links connecting these other objects to one another.

Testbed Control Agent A user agent that accepts user input and converts that input to control messages sent to the TaaS Resource Manager for processing.

Unallocated Marking for a resource instance that is no longer required.