29-04-2015

Deliverable D6.2 (DS2.3.2): Architecture Description: GÉANT Testbeds Service, Version 2

Deliverable D6.2 (DS2.3.2)

Contractual Date:	30-04-2015
Actual Date:	29-04-2015
Grant Agreement No.:	605243
Work Package/Activity:	6/SA2
Task Item:	Task 3
Nature of Deliverable:	R (Report)
Dissemination Level:	PU (Public)
Lead Partner:	NORDUnet
Document Code:	GN3PLUS14-836-123

Author:	Susanne Naegele-Jackson (DFN), Pietrzak Błażej (PSNC), Michal Hazlinsky (CESNET), Fabio Farina (GARR), Oscar Moya (i2CAT), Peter Szegedi (GEANT Association), Jakub Gutkowski (PSNC), Robert Szuman (PSNC), Tamas Varga (NIIFI), Markus Krogh (NORDUnet), Kim Kramaric (NORDUnet), Milosz Zdrakovic (AMRES), Aleksander Weinert (PSNC), Nikos Kanakis (GRNET), Andreas Papazois (GRNET), Andrew Mackarel (HEAnet), Hakan Calim (DFN), Buelent Arlsan (DFN), Nicolai Iliuha (RENAM), Jerry Sobieski (NORDUnet)

© GEANT Limited on behalf of the GN3plus project.

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7 2007–2013) under Grant Agreement No. 605243 (GN3plus).

Abstract

This document provides an update to the high-level description of how the GÉANT Testbeds Service (GTS) 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 regional or local 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.

GÉANT

Table of Contents

Executive Summary 1

1 Architectural Overview 3

2 Resources 6

2.1 The GTS API 9

2.2 Provider Agent 10

2.3 User Agent 11

2.4 API Primitives 11

2.4.1 Lifecyle Primitives and the Resource Lifecycle 11

2.4.2 Extension Primitives 12

2.4.3 Administrative Primitives 13

2.5 Extensibility 14

3 The GTS Core Resource Manager 16

3.1 Single-Domain Interaction 16

3.2 Multi-Domain Interaction 17

3.3 One-Stop Shopping 20

3.4 The GTS Domain Specific language 22

4 Virtual Resource Model 26

4.1 Testbed Interaction with the Physical World 28

5 Infrastructure and Virtualisation Layer(s) 34

5.1 Virtualisation Software Layer 34

5.2 Physical Infrastructure 36

6 The GTS Resource Database 41

6.1 RDB Data Structure 42

6.2 Resource Templates and Class Descriptions 44

6.2.1 Resource Records 45

7 The Resource Lifecycle 48

7.1 Resource Reservations 48

7.2 Resource Activation 49

7.3 Resource Instance States 50

8 The Resource Control Primitives 51

8.1 The GTS Resource Control Primitives and Lifecycle of a Testbed/ Resource Instance 51

8.2 The GTS Application Programming Interface Protocol 53

References 56
Glossary 56

Table of Figures

Figure 1.1: GTS workflow illustrating how the testbed network can test a researcher's idea 5
Figure 2.1: The GTS architecture treats all testbed networks as *graphs* 7
Figure 2.2: Basic representation of Nodes, Links and Ports 8
Figure 2.3: User-provider interaction 10
Figure 2.4: A simple physical testbed 15
Figure 3.1: User-Provider protocol relations among Multi-Domain GTS agents. 18
Figure 3.2: One-Stop Shopping 20
Figure 3.3: Example of the GTS DSL 23
Figure 3.4: Example of Groovy code 24
Figure 4.1: GTS object oriented virtual resource model. The DSL diagrams on left, the resolved integrated resource diagram on the right. 27
Figure 4.2: Testbed Edge Ports are linked to external facilities via separately provisioned virtual circuits 30
Figure 4.3: External facilities can be defined as external domain classes and manipulated like other GTS resources. 31
Figure 4.4: IAGW within the testbed 32
Figure 5.1: GÉANT GTS virtualisation, management and control layers 35
Figure 5.2: The GTS pod physical design 36
Figure 5.3: The GTS distributed architecture across the GÉANT core network 37
Figure 5.4: Pod infrastructure platforms 38
Figure 5.5: Resource mapping: VMs to servers 38
Figure 5.6: The GTS infrastructure, virtualisation software, and resource layering 40
Figure 6.1: The Project object and its structure within the Resource Database 42
Figure 6.2: The user and project organisation within GTS 43
Figure 7.1: Reservation lifecycle state machine 49

GÉANT

Executive Summary

This document describes the Architectural structure of the GÉANT Testbeds Service, Version 2.0. This second version of the service, originally introduced in GN3plus as "Testbeds as a Service", has been renamed to the GÉANT Testbeds Service (GTS). This architecture document describes the key functional components of the service and how they interact or are used to deliver the desired testbed networks.

GTS provides virtualised network resources to the network research community for the expressed purpose of testing novel networking concepts, at scale, across a geographically realistic European footprint. GTS creates user-defined, dynamically reconfigurable, virtualised network environments that are capable of fully autonomous and independent operation under the direction and control of the network researcher. GTS provides networks consisting of user-specified "resources" such as virtual machines, virtual circuits, and/or virtual switching or forwarding elements. These GTS networks are isolated from one another and provide performance guarantees where possible and appropriate. Users may create or delete these networks and may customise them and orchestrate them to address their research objectives. The GTS service architecture assumes a GTS service "domain" consisting of a geographically distributed "infrastructure" under the control of a single authoritative provider agent. The provider agent produces "resources" that are assigned to and comprise the users' testbed environments. Through a simple Application Program Interface (API), requesting user agents interact with a GTS provider agent to acquire and control testbed resources. The architecture neither stipulates nor is it dependent upon any specific geographical footprint or particular physical hardware or technology to function. The GÉANT-specific software or hardware engineering/implementation of these architectural aspects may employ certain specific technologies or standards, but these are generally not required by the architecture.

The most notable new architectural feature in GTS Version 2.0 is the multi-domain (MD) capability. "Multi-domain" GTS produces virtual networks ("testbeds") that are able to span multiple (GTS) service domains. The MD GTS constitutes a superset of functionality from Version 1.x. Associated with the MD capability is an integrated "One-Stop Shopping" feature, whereby a GTS service provider will locate and acquire resources from other GTS service providers in other domains on behalf of a single user request.

This document is intended to describe the *architecture* of the GÉANT Testbeds Service, not a particular *implementation* of GTS. However, since the architecture is currently only implemented in one software/hardware suite, some aspects of this particular implementation are described as explanation or example. It is expected that this implementation will be deployed generally in multiple, other domains (e.g. the GÉANT NRENs in the most immediate future). Some particular implementation aspects may provide generalisations that blur the distinction between architecture and implementation. At present, the GTS implementation and the GTS architecture is synchronised in Version 2.0, but these may diverge in the future. New features that do not constitute an

architectural aspect, or architectural aspects that are not yet supported by the implementation, will be described in a separate release guide. Such capabilities may be briefly mentioned, where relevant, in the following sections, but the main focus of this document is on the architectural concepts: functional components and interactions, scalability issues, control issues, etc.

The GTS architecture addresses a broad set of long-term goals, such as global scalability, security/authorisation/privacy, extensibility/maintainability, and user control. The software-defined networking and virtualisation technologies that make up the GTS architecture, the GTS software, and the GÉANT GTS service itself are all still in their infancy. Thus, some critical architectural elements may not be fully (or rigorously) implemented at present due to realities of development timelines, hardware or software inter-dependencies and feature priorities. As the system and service grows, and new issues come to light, the architecture will be enhanced to address these issues in a manner consistent with principles of scalability, security, reliability, and maintainability, as well as aligned with global collaboration and standards.

1 Architectural Overview

The virtualised network environments provided by the GÉANT Testbeds Service (GTS), i.e. the testbeds, are fully autonomous and independent networks operating under the direction and control of the user. For a better understanding of the context, several key terms and components of the architecture must be introduced.

- **The Resource**. The GTS service makes *resources* available to users. Resources are virtualised objects that are an abstracted representation of some network element or other device or function. Resources can be allocated to and controlled by the user. Current GTS resources include hardware analogues: virtual machines, virtual circuits, virtual switching/forwarding fabrics, etc. Resources may be grouped to form more complex “composite” resources. It is these composite resources that define a “testbed”.

- **The Infrastructure**. Infrastructure, in the GTS context, refers to the underlying facilities managed by a GTS service domain from which the resources are produced. One can think of infrastructure as raw materials that are fed into a virtualisation function to produce resources as output. Since virtualisation is a *layered* concept, resources from one virtualisation function - or “layer”- may be requested and treated as infrastructure by another virtualisation function/layer. Thus resources and infrastructure are related objects. It is only at the lowest layer of this stacked virtualisation model where the infrastructure consists of real hardware or other physical devices.

- **The Domain Specific Language (DSL)**. A DSL is a textual description of the user’s desired testbed. The DSL *grammar* is an object-oriented data flow description language designed specifically to address GTS services. It describes the computational, switching, and transport resources that make up a testbed network, and the topological arrangement of these resources and their I/O capabilities. The DSL representation of a resource/testbeds is used by the API to carry resource specifications between a user agent and the GTS provider agent. In addition, the DSL can be used to save a testbed description after editing or for later inclusion in more complex testbeds or resource class libraries (see Section 3.3).

- **The GTS Applications Program Interface (API)**. The API is a set of basic functional commands or “primitives” used to reserve and manage a resource or other GTS networks through its lifecycle. The API primitives are passed between the User Agent (UA) and the Provider Agent (PA). These “agents” are actually roles an agent assumes within the API protocol and do not refer to specific software modules. (See Section 8.2). The DSL describing desired resources or a testbed network is a primary parameter sent between user and provider agents.

- **The GTS Core Resource Manager (RM)**. The RM is a software module that is the main active component of the GTS service. The Resource Manager acts as the GTS provider agent (PA) to external requesters, and in Version 2.0 of the MD GTS, the resource manager acts as a Multi-Domain Provider Agent (MDPA) which can also assume the role of a requesting user agent to other GTS provider agents. The RM includes a DSL parsing function that converts DSL textual descriptions into an internal "resource tree" used for the actual management of the resources.

- **The Resource Database (RDB)**. The RDB contains all the information about the services provided by a GTS domain. This includes the projects and users known to the GTS domain, the inventory of infrastructure components managed by the service, the virtual resources allocated to those projects and users, resource reservations, and the state of all of these objects. The RDB is persistent and is intended to be a distributed facility (for increased reliability).

- **The GTS Graphical User Interface (GUI)**. The GUI is an important part of the GTS software implementation and an important part of the GÉANT service workflow, but, strictly speaking, it is only relevant to the architecture in that it is an example of a User Agent. The GUI addresses two practical service requirements: a) it provides an intuitive point-and-click and drag-n-drop interaction with human users, and b) it converts these graphical interface actions into testbed/resource lifecycle management interactions with GTS resource Provider Agent(s). It is this latter function – the testbed control function – that is relevant to the architecture because this function is an instance of the User Agent referenced in the GTS API protocol.

- **The GTS Workflow Model**. The basic workflow model for GTS essentially describes the lifecycle of a GTS resource (and by extension, a testbed). The workflow begins with a user with an idea for a network. The user describes the resource/network in a DSL document and submits this document as part of a Reservation request to a GTS service. The GTS service parses the DSL description of the resources and/or network, and selects and allocates appropriate resources to the user. The user, via API control primitives, is able to move the resources from a Reserved state to an Active state, perform the necessary experiments or other functions within the virtual network, and ultimately Release the resource or testbed when it is no longer needed.

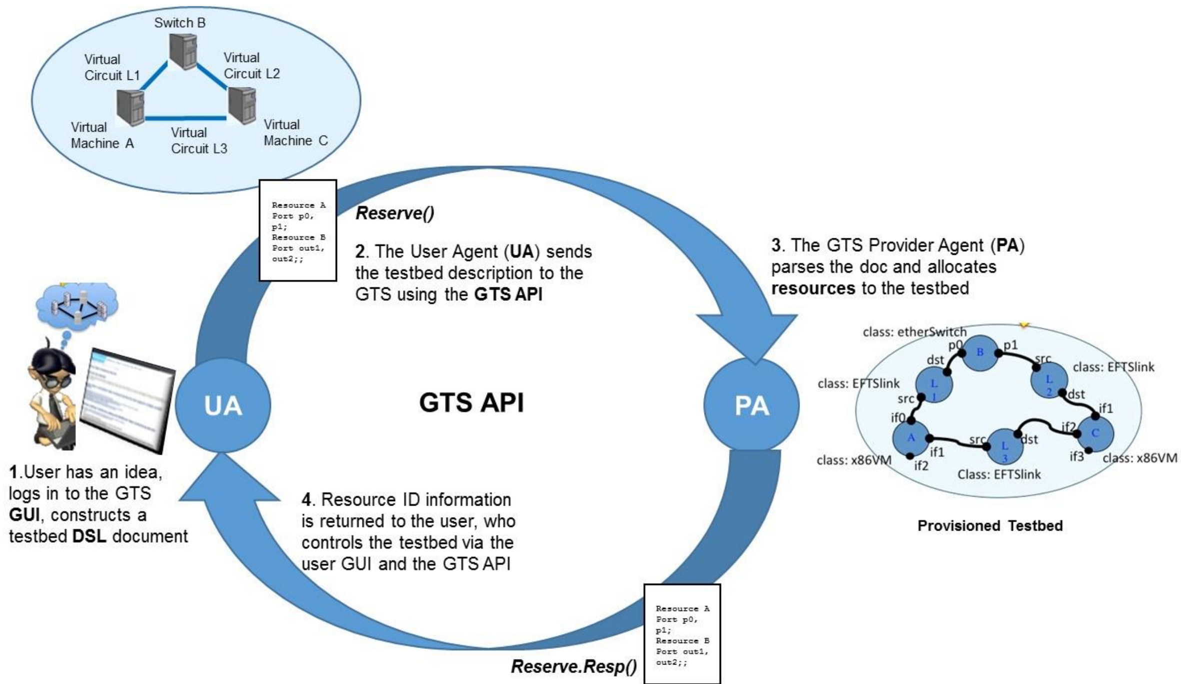


Figure 1.1: GTS workflow illustrating how the testbed network can test a researcher’s idea

Each of these concepts and components is described in greater detail in the following sections.

2 Resources

The most fundamental concept in the GÉANT Testbeds Service is the notion of a “Resource”. Resources are individualised virtual objects. These objects are the components that make up a “testbed”. For instance, a virtual machine is a resource, or a virtual circuit is a resource.

There are a number of properties associated with resources that define their behaviour or define how information transits these resources.

Resources can be grouped according to their type, or *class*. A resource class defines a common set of properties that all instances of that class share. For instance, a Virtual Circuit class might define a property called “Capacity” that could take any value between 0 and 10,000. For a particular instance of the Virtual Circuit resource class, this property would have a particular value, e.g. Capacity=1000.

Resource properties include:

a) **Attributes** – “attributeName=value” pairs identify an attribute and define a value for that attribute, e.g. Capacity=1000, or cpuCount=2, or Memory=2000.

b) **Ports** – *Ports* enumerate data flow interfaces associate with a resource instance. Ports define interfaces where data may enter or leave the resource. A given resource instance may have multiple ports, each with a unique port name, and each with a set of attributes that define its operational characteristics. For example, a port may have a name “P1” with attributes of “macAddr=BADBADB0”, and/or “qTag=1784”.

c) **Resources** – A resource may contain internal resources. A resource class that is defined to contain other resources is called a *composite* resource. A resource class that contains no other resources is an *atomic* resource.

d) **Adjacencies** – An adjacency defines a relationship between two ports that indicates the two ports are bound to one another, i.e. data flowing out one port in an adjacent pair is considered to be flowing into the adjacent port. Adjacencies define the topology of a group of resources.

e) **Control primitives** – Control primitives are class-specific control functions that can be performed on resource instances of that class. All resource classes must provide the five, basic lifecycle control primitives: Reserve(), Release(), Activate(), Deactivate(), and Query().

f) Other resource specific primitives may exist to perform specialized functions applicable to a particular resource class. Administrative primitives also exist generally to allow users to manage their project and users.

The GTS testbeds are established by defining a composite resource containing the specific testbed resources and the port adjacencies among those resources to establish the desired topology.

Resources are defined by a template that fully specifies the attributes associated with that resource. A template formally defines a type, or *class*, of resource. Each class of resource performs a unique function and has a unique set of attributes that describe or constrain the behaviour of instances of that class.

The testbed network that GTS provides to users consists of switching/forwarding elements, end systems that act as sources and/or sinks of traffic (or possibly switching or protocol stack platforms), and data transport links connecting these objects to one another.

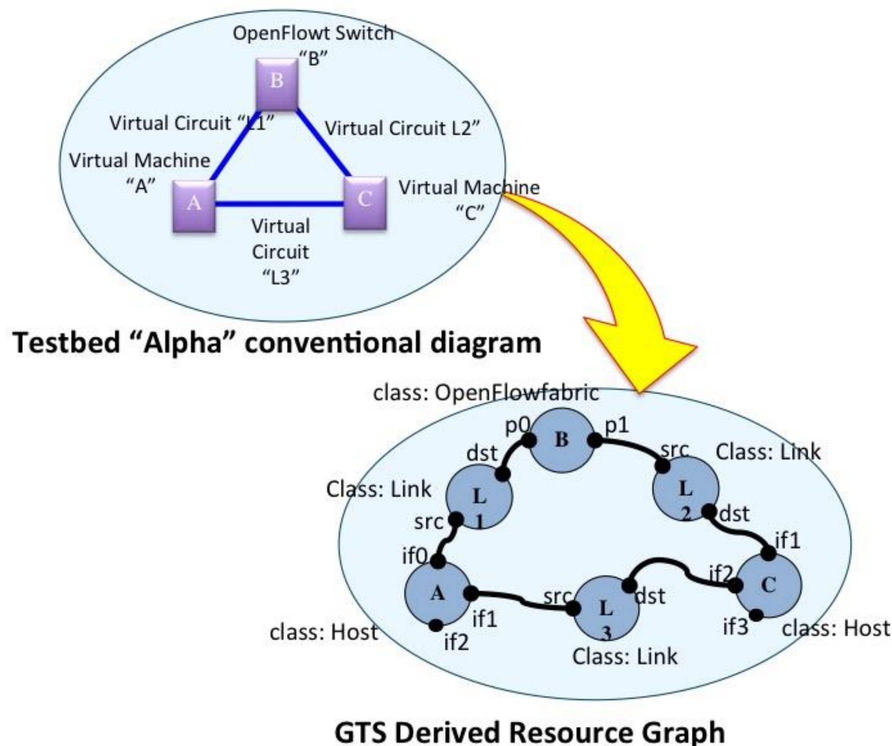


Figure 2.1: The GTS architecture treats all testbed networks as *graphs*

As shown in Figure 2.1, a network can be represented as a graph. In a conventional network model, each forwarding element or end system in the network is represented as a vertex in the graph. These vertices are the network elements, or “nodes”. Likewise, each transport circuit that connects two network nodes is represented as an edge in the graph. And similarly, these edges are referred to as “links” in the context of network topologies. It is often the case that networks may have multiple links converging at a single particular node. Indeed, there may be multiple Links existent between a pair of nodes. Because of this, a link cannot be uniquely identified by the two nodes it connects. This is solved at each Node by enumerating the links that converge at that node. These enumerated identifiers at each node are represent “ports”. It is possible then to uniquely identify any particular data transport connection by specifying its <node:port> two-tuple. Thus, this conventional graph model has three key objects: Nodes, Links, and Ports, as shown in Figure 2.2.

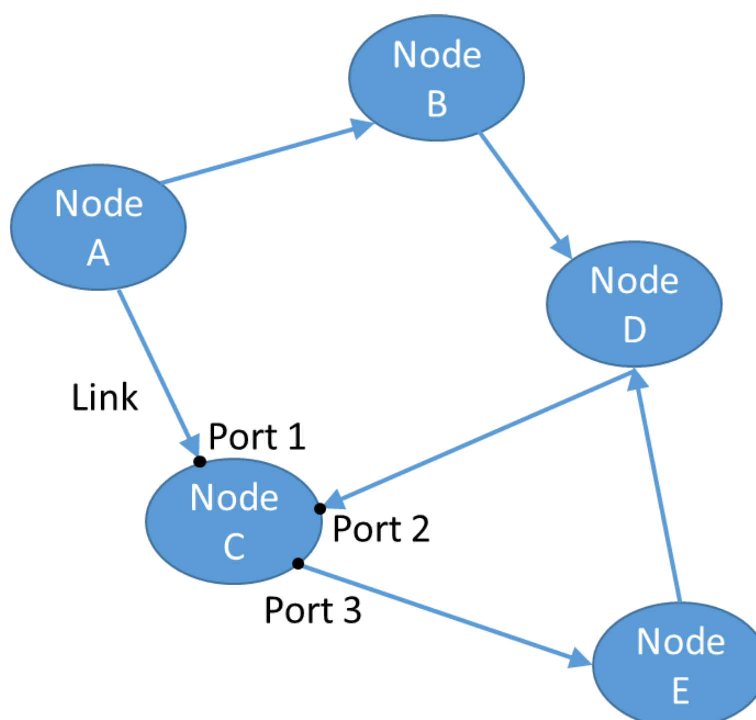


Figure 2.2: Basic representation of Nodes, Links and Ports

This conventional model is simple and intuitive. However, it is not an accurate representation of real-world networks, where Nodes and network Links represent technologically complex objects that are constructed from other underlying or hidden physical infrastructure assets with administrative and performance attributes and parameters that must be considered in the overall object model. Therefore, in order to generalise a representational model of a testbed, GTS takes this conventional network graph concept one step further to represent networks as “derivative resource graphs”.

A *derived resource graph* is a derivative of the conventional graph model, but it treats both nodes *and links* as generalised “resources” with a common resource representation. In the resulting resource graph, all network components are represented as *resources*. This means that conventional switching elements, servers, and the transport circuits between these other network elements are all defined as their own resource types.

In order to organise this collection of individual resources into a network, we must specify how these resources are arranged into a topology, i.e. how each resource exchanges data with other resources. Since the GTS model sees only generic resources, and those resources each have explicitly defined data flow ports, we can define the topology by indicating which ports are connected to, or are *adjacent* to, one another. The connectivity among these resources is specified by *adjacency* relations indicating which resource ports are attached to one another.

Thus, in the GTS network diagram, the vertices represent resource objects (e.g. switches, servers, circuits, etc.) each with defined and enumerated data flow ports, and the edges represent the data flow adjacency relations between resource ports. This model allows the GTS software to generalise the management and control of all testbed resources. It also provides a model where the network topology is specified independently of the network resource attributes and can be more easily validated to ensure the data flow characteristics specified in a potentially large and complex testbed are realisable in practice.

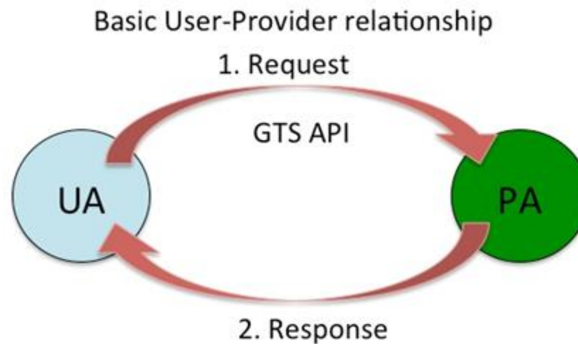
GTS can group resources together into a “composite” resource. A composite resource is simply a container for other resources. The composite may also contain adjacency relations among the internal resources thus describing an internal topology. And composites itself may also define data flow ports that describe the I/O capability of the composite resource itself. These external data flow ports can be set adjacent to data flow ports of internal resources to indicate how data enters or exits the internal network. Thus a user “testbed” is in fact just an instance of a user defined composite resource class.

Composite resources can be defined and used in an object-oriented and recursive manner to define large or complex testbeds from simpler, more atomic resource constructs. Thus the notion of a testbed itself is simply reduced to a user-defined composite resource.

When taken as a whole, these resources, along with their ports and their adjacencies, define a testbed data plane. The data plane is the experimental facility available to the user and under user control.

2.1 The GTS API

The user interacts with GTS through the GTS API. The API is a simple two-party messaging protocol. In general, the two parties assume a user-provider (or requester−provider) service model. The user is actually a “user agent”, i.e. a software component that represents the actor wishing to acquire and manage resources. Similarly, the provider is a software agent representing the service provider that produces resources from that infrastructure and makes those resources available to users. The API is intended to be a symmetric protocol, i.e. either agent may send messages to the other agent at any time. This allows the provider to send messages to the user notifying him/her of various downstream events. In practice, however, the User Agent can normally manage resources through their entire lifecycle without needing to implement a server-side interface to listen for unsolicited primitives from other agents (this server side functionality is typically only used by multi-domain provider agents).



In the Multi-Domain model, the Multi-Domain Provider Agent plays both roles:
Provider to upstream User Agents, and User to downstream Provider Agents

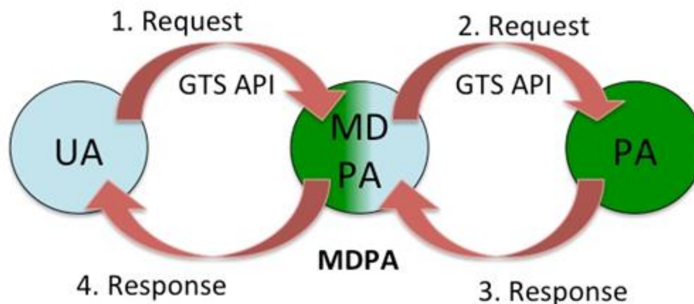


Figure 2.3: User-provider interaction

2.2 Provider Agent

The GTS API consists of a set of resource control *primitives*. These primitives are user commands associated with a particular resource instance and so assume a “user-provider” protocol relationship. In other words, when an agent determines that it must acquire a resource from another agent, the requesting agent assumes the role of “User Agent”, and will send an unsolicited resource reservation primitive to another agent that assumes the role of a “Provider Agent”. Thus, the respective protocol roles of these two agents is thereby fixed in relation to the particular resource instance that is being managed.

This resource relative relationship is important in the multi-domain functionality introduced into GTS v2.0. In the MD GTS the two agents may both be service providers that are brokering resources from each other to serve their respective users. Provider A may request a resource of type X from Provider B, and Provider B may request resources of type Y from Provider A. Each agent acts as a requesting User Agent on some primitive exchanges and acts as a Provider Agent on other primitive exchanges.

These API primitives manage the lifecycle of resources and provide other administrative functions for managing GTS projects and user AAI information.

2.3 User Agent

Typically, a user agent would be a software tool, such as a graphical user interface that converts graphical point and click human interactions into appropriate API control primitives, which get sent to a provider agent to effect some action on a resource. The researcher is free to create any type of software contol agent to manage testbed resources via the API. GTS also provides command line tools that can be used within scripts to similarly submit user initiated control primitives to a provider.

2.4 API Primitives

The GTS API consists of three key types of primitive control functions: Lifecycle primitives, extension primitives, and administrative primitives. Each is dicussed in the following sections.

2.4.1 Lifecyle Primitives and the Resource Lifecycle

Lifecyle primitives mange the lifecycle of GTS resources. These primitives move a resource from its initial creation through several reserved or active states until it is released back into the infrastructure pool. There are five primitives that move the resource through its lifecycle, and two that are used to register a user defined resource description with the provider agent. These seven primitives are as follows:

- Define() – Enters resource class description into the project's class library.
- Undefine() – Removes a resource class from the project's class library.
- Reserve() – Instantiates a resource and a reservation for that resource.
- Release() – Releases a resource back to the available pool and deletes the reservation.
- Activate() – Configures the infrastructure to place a resource in service to the user.
- Deactivate() – Takes a resource out of service but holds the reservation.
- Query() – Returns the state and attributes associated with a resource instance.

The GTS API primitives are designed to move a resource instance through its "lifecycle". See Figure 1.1 for the resource lifecycle. The life of a resource begins when the class is defined. A *class* definition gives name to a group of properties that are shared by all resource instances of that class. A resource *instance* is created, or *instantiated*, by the Reserve() primitive. Instantiation means that a reservation record has been created for the resource and that the associated virtualisation layer has mapped the resource instance to the appropriate underlying infrastructure components. The corporeal resource object may not have been actually created at this point, but the raw materials that will be required when it is created have been reserved and accounted for. Thus,

the resource manager is able to allocate a resource ID to the instance and so, for the purposes of the API, the resource has been instantitated. Thus, the resource instance lifecycle begins once the instance is reserved.

The Activate() primitive realises the resource instance. Activation causes the underlying infrastructure to be reconconfigured, or *provisioned*, to initialise the resource and place it in service. A virtual circuit is realised when all the network elements along its designated path in the infrastructure have been reconfigured to switch the circuit through end to end. Similarly, a virtual machine is provisioned when the resources on the compute node server have been assigned and the hypervisor boots up the guest operating system. Activation may take several minutes, depending upon the type or complexity of the resource being activated. The resource instance is not considered in service to the user until the activation process has been successfully completed. The Activate() primitive will be rejected if it is invoked before the reservation start time or after the reservation end time for that instance. Once active and in service, the resource may then remain active until the reservation end time.

The Deactivate() primitive is used to take a resource instance out of service. This shuts down all functions of the instance without saving state. The resource reservation is not affected, so the instance may be re-activated later. Note: Reactivating a resource reinitialises the resource. Therefore, if the resource is expected to recall state from before the deactivation, the user must save such state prior to deactivating the resource and then restore that state after restart.

The Release() primitive will cause all aspects of the resource instance to be released and returned to their infrastructure pool, and the reservation is deleted. The reservation is written to an archive for long-term storage, but it is no longer part of the active service.

These seven resource lifecycle primitives are defined for all resources. The Resource Control Agents that produce the resources from the underlying infrastructure must implement these seven functions, regardless of the type of resource they may be managing.

2.4.2 Extension Primitives

Different resource types may have operations that are unique to that class of resource, or can only be performed by a privileged agent, or may need to violate virtualization in order to perform the function. Such extension primitives can be defined and performed by the RCA as a proxy in order to insure that a user request is done within the context of the virtualized resource model. For instance, the OFX resource class (OpenFlow switching fabric) would implement a setDatapathID() primitive to set a user specified datapath id for the openflow instance. This function can only be performed through the console interface and a privileged user. A user performing this function directly would be able to inadvertantly (or maliciously) reconfigure other openflow instance on the hardware switch. In order to prevent this security breech, the GTS RCA-VC agent that is part of the GTS service provider would perform this function on behalf of the user.

A resource class may have any number of extension primitives.

2.4.3 Administrative Primitives

Administrative primitives are functions that modify the Project or Users associated with a Project. These primitives can be performed by users that are designated as "admin" for a project. The project admin can add or delete users associated with a project or adjust key attributes. The super user admin is able to add or delete projects and designate users as admins.

These administrative primitives are very basic. GTS will be implementing a more sophisticated authentication and authorization model in a future version. The GTS design team is evaluating the eduGAIN federation model, and evaluating other federating models implemented within other testbed frameworks.

Examples of administrative primitives include:

- addProject() – Create a new project with the provider. Admin priviledge.
- delProject() – Delete a project. Admin privilege.
- addUser() – Add a new authorised user to a Project. Admin priviledge.
- modUser() – Change information associated with a user (such as loginid or passwd).
- delUser() – Remove a user. Admin priviledge.

The addProject() primitive creates a new project within the provider's resource database. The Project object is a root node of a tree data structure that contains all aspects of a particular project. This includes users that are authorised to manipulate resources under the project, the resource classes defined under that project, and resources allocated to this project. This primitive can be called only by users with "admin" privileges.

The add/mod/delUser() primitives perform the associated function under a particular project. User name, login, passwords, etc. information is captured to create a new user authorised to manipulated testbeds (resources) under that project.

The admin user can create a new project, and define users under that project. The users designated as admin are allowed to manipulate other user entries under that ProjectID. Non-admin users are allowed to reset their passwords and to manipulate resources assigned to the project using the resource control primitives.

The API primitives, their arguments and returned information, and functional internal information are detailed in a separate GTS User and Resource Guide. The abbreviated functional descriptions provided above are sufficient for this GTS Architecture guide.

These primitives are interpreted by the Provider Agent. The provider agent may be a multi-domain provider agent the interprets and forwards a primitive to a Local Only provider, or it may be the local provider agent directly. The Provider Agent sends a response back upstream to the requesting user agent upon completion of the primitive. It should be noted that some primitives may be completed very quickly, such as a reservation of a local resource. Other primitives, or particular attribute permutations of some primitives, may take considerably longer. For instance, an Activate() primitive on a VM may take several minutes as the VM must be booted up. In such a case, the Provider will respond back upon initiating the activation process. The User Agent can poll the state of the

resource with a Query() primitive to determine when the resource has transitioned from Activating state to the Active state.

2.5 Extensibility

An important feature of the GTS architecture is its extensibility, or consideration of future growth. New resource types can be introduced into the GTS service portfolio by defining the class properties for that new resource class and implementing the five, basic lifecycle management primitives for that class: Reserve(), Release(), Activate(), Deactivate(), and Query(), and also implementing any extension primitives that may be required. The five life cycle primitives are internally mapped by the provider agent to specific software functions that implement the associated task for each class of resource. Thus a Reserve() primitive for a resource of class “host” would map internally to a software function x() that allocates and reserves a virtual machine. Similarly, a Reserve() primitive for a class “Link” would map to a different internal function y() that allocates and reserves virtual circuit resources. For the other resource control primitives (Activate, Release, etc), which specify a resource instance and do not directly specify the resource class, the provider agent can look up a specific resource instance in the resource database to obtain the resource class, and then map the (primitive, class) tuple to the appropriate resource-specific software function.

The basic five resource lifecycle control primitives described above are defined for all resources. The other two lifecycle primitives Define() and Undefine() are not resource specific, so they are generalised and performed by all multi-domain Provider Agents. The class templates are associated with the user’s ProjectID and stored in the provider’s RDB. The remaining five primitives are “resource oriented” – i.e. they manipulate particular resource instances. This “Gang of Five” primitives (Go5) manage the lifecycle sequencing of resources. The Go5 primitives may each need to perform different infrastructure tasks for each resource class. While the virtualisation layer presents a consistent set of base resource primitives northbound towards the resource manager, the southbound tasks are resource specific, and so each resource will have its own basic software control functions. These resource-specific control agents (RCAs) are unique to atomic resources.

As new resource classes are defined, one can imagine that there will be functions required of a resource or operations performed on that resource that would violate the abstracted virtualisation model for the resource or might cause a security breach. For instance, modifying the BIOS of a VM server. Since such actions pose a security risk, these actions must be performed by a privileged proxy. For example, the resource-specific functions are implemented as resource specific extension control primitives (in addition to the gang of 5) in the API and will be performed by the Provider Agent after proper auditing of parameters and authorisation has been approved. Thus, the creation of a new atomic resource class requires the implementation of the basic Go5 lifecycle primitives, and any extension primitive unique to that class of resource. These extension primitives pose a serious security issue therefore GTS does not allow users to develop their own primitives at this time. If new resource classes are required, or if an extension primitive to an existing resource class is necessary, the user should contact GTS Support to explore the concept, and the GTS development team can develop the new primitive.

For composite resources, GTS defines a set of basic, generic lifecycle primitives. The generic primitives essentially walk the resource tree defined by the composite DSL, and recursively perform the associated lifecycle functions. For instance, a Reserve() of a composite resource recursively Reserve()’s all the internal (child) resources and resolves the internal adjacencies before returning.

Figure 2.4: A simple physical testbed

3 The GTS Core Resource Manager

The GTS service is embodied in the core Resource Manager (RM). The RM is a major function of the Provider Agent. The Resource Manager interprets and processes the primitives received from User Agents. In general, the RM will allocate resources from its own “local” inventory of resources. In MD GTS, the RM may also acquire resources from other providers and present those resources to the requesting User Agent as if those resource were local. The RM implements the server side of the API – the Provider Agent – when interacting with upstream user requests, and it implements the client side – the User Agent – when requesting resources from other downstream provider agents.

3.1 Single-Domain Interaction

The first key function performed by the RM is the DSL Parser. The Define() primitive passes a DSL description of a resource class to the Provider Agent. The RM parses that class description and creates an internal *template* tree for that class description under the User’s Project ID. Multiple class templates may be defined under a given project, which act as a library of resource classes available to the user. A class template must exist in the project library before a resource of that class can be Reserve()’d. Note: The base set of atomic classes “Host”, “Link”, and “OFX” are implicitly defined in every project library. Thus, the user does not need to define these atomics and can immediately reference them from the API or a composite class description. The formal DSL description of these classes, as well as a detailed description of all API administrative primitives, lifecycle control primitives, and extension primitives suitable for software development can be found in the GTS User and Resource Guide [GTSWEB].

A class template is an internal tree representation of a DSL class description. In general, the user is only able to define composite classes that contain atomic resources or other user defined composites. The template tree represents any internal topology associated with the class. The root of the template tree represents the class itself. Each child node represents a resource instance that is contained within the parent composite. When found within a class description, these child resources are called resource “references” because they refer to a name of an instance, but do not actually instantiate the resource until the parent class is instantiated by a Reserve() primitive.

A Reserve() primitive requests the RM to instantiate a resource of a particular class. The desired resource class is indicated in the Reserve() parameters, as are any attributes or other class properties specified by the user. Upon receiving the Reserve() primitive, the RM will locate the specified class template in the user’s project library. The RM walks the class template, recursively Reserve()’ing each child resource and processing its associated

adjacencies. When all children have been Reserve()'d, the Reserve() function returns a unique ResourceID for the completed parent resource instance.

Each resource instance created by this process generates a resource record in the Resource Database. The resource record contains the ResourceID and the resource class of the new instance, the ResourceID of the parent instance, and the other values or user specified attributes or properties of the instance. Taken as a group, these instance records, with their ResourceID as key and the Parent ResourceID as a pointer, define a Resource instance tree that represents the user's fully instantiated testbed.

The instantiation process and processes associated with the other standard lifecycle control primitives for atomic classes are typically highly class specific. For instance, the allocation process for a Virtual Machine is substantially different than the allocation process for a Virtual Circuit (as is also the case for Activate(), or Query(), etc.). While all resources must support the same basic primitives, each primitive is overloaded with a different function, depending upon the target resource class. The actual mapping from the primitive to the class-specific function is an implementation issue.

Unlike other atomic classes, the instantiation of a composite resource is essentially a bookkeeping function that walks the template tree and creates the resource record, but without the need of any specialised allocation or management of infrastructure. Therefore, the composite resource control primitives are implemented internally as a single set of generic functions within the MDPA/RM agent. This architecture will allow user-defined resource control primitives in the future, but for security and RDB integrity reasons, these are not currently allowed.

3.2 Multi-Domain Interaction

With the multi-domain capability in GTS Version 2.0, the basic User-Provider protocol of the API plays a greater role in the distributed GTS system. While the API protocol is itself unchanged, the function of the Provider Agent incorporates additional functionality that allows the Provider to act as a broker or reseller of resources besides its own. As a result, the Multi-Domain Provider Agent (MDPA) was introduced. The MDPA acts as a conventional Provider Agent when interacting with upstream User Agents (clients), and acts as a conventional User Agent when attempting to acquire resources from downstream Provider Agents.

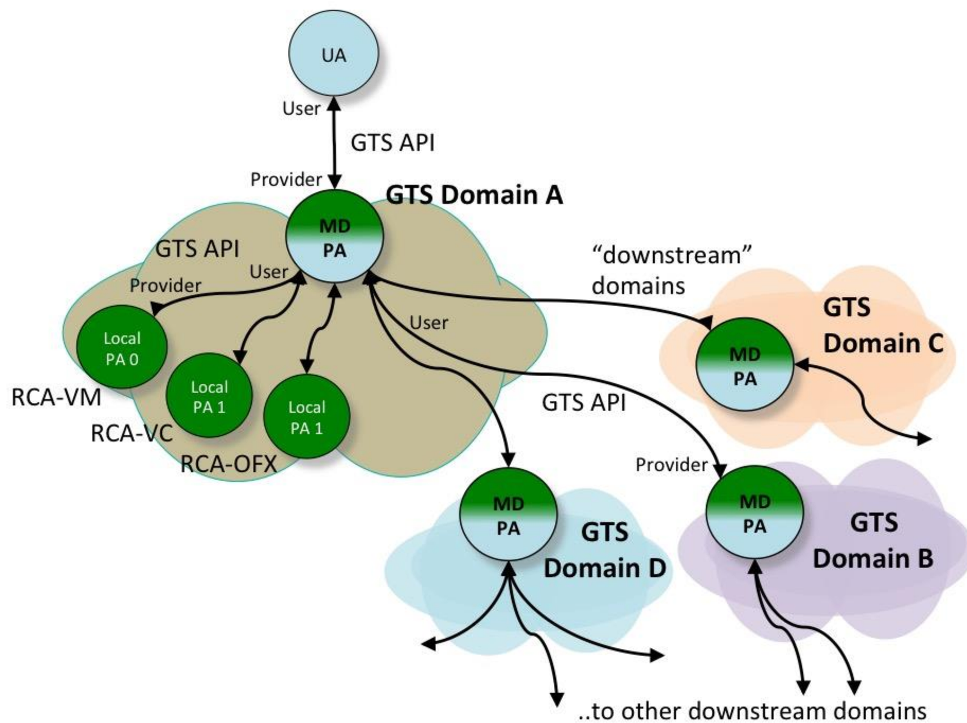


Figure 3.1: User-Provider protocol relations among Multi-Domain GTS agents.

If one closely considers this arrangement, the MDPA does not need to have any resources of its own – it can acquire all of its resources from downstream providers and present them as its own to the upstream client. Given this possible approach, the agents responsible for provisioning local atomic resources can, in fact, be configured as Provider Agents themselves. Each of these local PAs manage (their own) local resources only and do not look downstream for other providers. Thus the local agents are Local Only and function as leaf nodes in the distributed search process that results from the GTS Multi-Domain resource discovery and reservation process. The Local Only PAs (LOPAs) can be class specific, i.e. each LOPA configured to be responsible for a single, particular class of resource (For instance, in the GTS 2.0 implementation the LOPAs are named from their v1 implementation as Resource Control Agents (RCAs). So in v2.0 architecture the LOPAs are named RCA-VM, RCA-VC, etc. and each manages one particular resource class, although in v2.0, these agents are promoted to full GTS API protocol agents).

The MDPA is the externally facing agent for a GTS service domain. It is the only protocol agent that needs to be publically advertised. User Agents can submit all API primitives to the MDPA, and the MDPA will process the administrative primitives and composite resources itself, and it will forward other lifecycle primitives to the appropriate downstream PA. The Multi-Domain Provider Agent serving a GTS domain may be configured to inter-operate with multiple other MDPAs. This interoperability requires each agent to know the service address of the other PAs and have appropriate authorisation credentials to use those PAs. Similarly, in order for the relationship to be symmetric, each remote PA should also know the service address of the local MDPA and also have proper credentials.

The multi-domain aspects of Version 2.0 enable the MDPA to look for and acquire resources from its own local provider agents, or, if local resources are unavailable, to look for and acquire needed resources from other GTS service domains. Because this multi-domain behaviour is present in all GTS domains, it is important to consider how the behaviour of each agent – if duplicated in all agents – will define the emergent behaviour of the distributed system.

First, this MD reservation process is actually a constraint-based, depth-first search of a spanning tree rooted at the initial MDPA. The Reserve() primitive will be sent to be processed by as many PAs as necessary until the requested resource is acquired. First, local PAs will be searched, then immediate neighbour PAs, and then more distant PAs, until the entire GTS peering region is contacted. When this multi-domain region is small, with only a few peering GTS domains, this behaviour can quickly locate a resource class and allocate an instance. However, if the number of peering domains is large, this search may be long. And the search may be longer if resources are generally scarce, causing the search to visit a greater number of domains.

Second, loops may occur. For example, Domain A forwards to Domain B, which forwards to Domain C, which forwards to Domain A again – not knowing A has already been visited. Loops can occur innocently in a normal search, or they may be injected deliberately by a bad actor. Since all agents that may be in the loop are not known, they cannot be relied upon to detect and remove loops. An MDPA's own actions are the only actions the MDPA can rely upon to protect itself against loops or other errant behaviour of other domains. Therefore, a simple means of killing a loop is to only allow a fixed number of open requests from a particular agent to be active at any time. Received requests in excess of this ceiling are immediately rejected.

Third, composite reservation requests can be processed in two basic techniques; how an MDPA pre-processes the primitive request can be an issue. For instance, should the MDPA forward a composite request downstream or should the MDPA process all composites locally and only forward atomic resource requests downstream? GTS 2.0 has adopted a policy of processing composites locally and only forwarding atomic requests downstream. This not only reduces the messaging load of the overall system dramatically, it also improves its responsiveness.

A fourth potential issue is that the resulting (recursive) depth-first search may locate and acquire resources in domains quite distant from the originating user agent. This measure of near or far is the number of hops in the control plane – which may or may not correlate to a physical distance in the data plane or geographically. A "time to live" (TTL) reservation attribute is under consideration (not yet approved) that will limit the depth of the search: each PA will check this attribute received with an upstream Reserve() request and, if it is equal to zero, no further downstream domains will be searched. If the TTL is greater than zero, the TTL is decremented and the Reserve() request will be sent to downstream GTS domains.

These issues will be monitored closely as MD GTS is deployed and more first-hand experience is gained. Experience will dictate which issues are of most concern and why. In the interim, implementations should limit the number of simultaneous Reserve() requests accepted from any particular upstream client in order to deterministically prevent infinite loops. If architectural changes are necessary, they will be reflected in a future GTS Architecture update.

GTS also provides an Internet Access Gateway (IAGW). The IAGW is usability feature for testbeds that allows users to easily reach the elements in their testbed. The IAGW consists of (among other things) a Layer2 VLAN that is set up by the service provider and is available to all resources in a testbed. If resources are acquired from

other downstream domains, then this Layer2 access network should also extend into these other domains. Under the current construction of the IAGW, replicating the access network in downstream domains becomes problematic, e.g. VLAN assignments must be coordinated, isolation and privacy engineering must be preserved, etc.) Therefore, in GTS v2.0, downstream reservations are constrained to atomic resource instances, and a standard virtual circuit is implicitly created to link the downstream resource back to the originator's IAGW in the home domain. This issue will be addressed more formally in a later release when "Layer2 broadcast domains" are introduced as a formal resource.

3.3 One-Stop Shopping

The multi-domain downstream forwarding of resource requests allows GTS 2.0 to provide One-Stop Shopping to clients. The User Agent can submit a testbed class description to any first-hop MDPA, and that MDPA has the ability to acquire resources from any peered GTS domain in the contiguous GTS region. This transitive behaviour is designed to enable the first-hop MDPA to construct the entire composite facility, potentially from multiple contributing downstream domains, and present the entire montage of resources back to the user as a single, integrated testbed domain, controlled through a single comprehensive provider interface. This One-Stop Shopping concept extends beyond simply the initial allocation of the testbed. One-Stop Shopping includes ongoing responsibility for the operational availability of the testbed. The first-hop MDPA confirmed the facility to the user as part of meeting their requirements, therefore, that first-hop MDPA also becomes the one-stop shop responsible for resolving any performance or operational issues. As far as the user is concerned, the first-hop provider sourced all of the resources from its own domain.

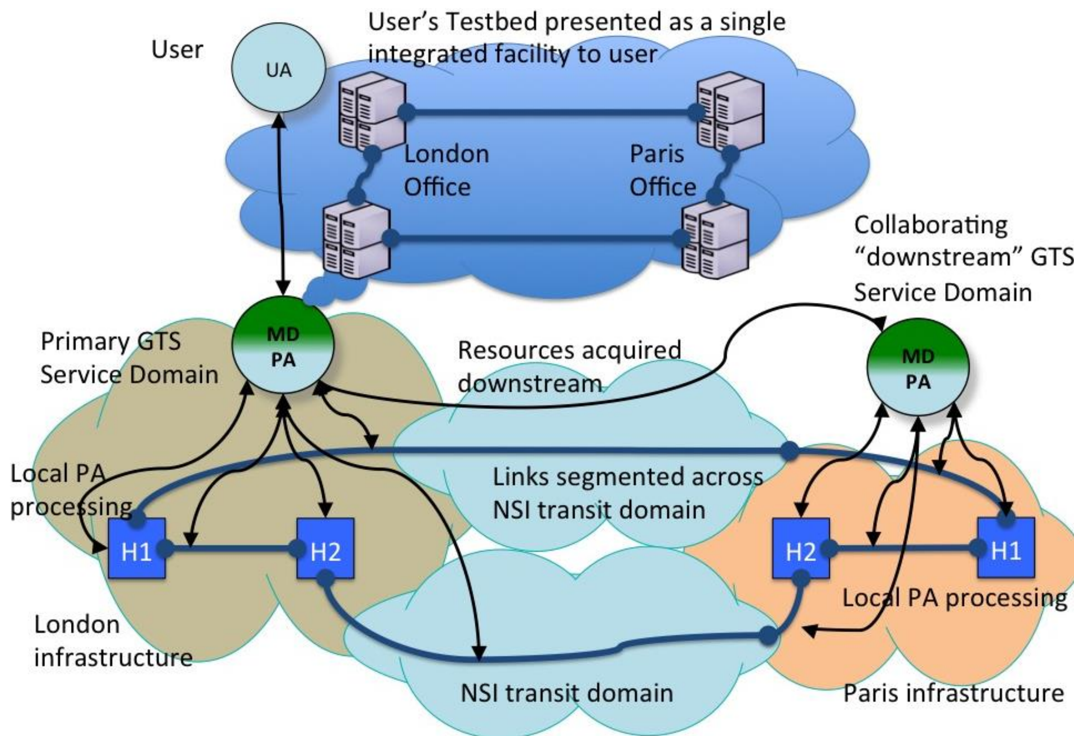


Figure 3.2: One-Stop Shopping

In order to present a single, integrated testbed environment to the user, the MDPA must present all resources to the user as if they were sourced from its own local infrastructure. In the resource instance tree, the MDPA must create a resource instance record that contains its own local ResourceID, and it must record the downstream MDPA and ResourceID from which it acquired the instance. This means the MDPA will transparently act as a proxy for resource control primitives it receives from upstream and forward them downstream to the appropriate provider. For example, when the upstream client sends a Query() request to the first-hop MDPA, the request will contain a ResourceID that belongs to the first-hop domain. The first-hop MDPA will locate that local ResourceID up in the local resource data base, and see that it was acquired downstream from another MDPA. The MDPA will take the associated downstream ResourceID and generate a Query() to the indicated downstream MDPA referencing the downstream ResourceID. The results obtained from the downstream request will be translated and returned upstream reflecting the virtualised resources identifiers of the local MDPA.

By mutual agreement, the UA and PA may have arranged one or more authorisation credentials that may be used in allocating resources. It is up to the UA to request resources using appropriate credentials recognised by the PA. Authorisation credentials are presented by the UA to the PA along with each request. The request is authorised by the first-hop PA using the presented credentials. The User Agent is responsible for all resources acquired under any of its pre-arranged credentials – i.e. all resources are assigned to the requesting UA, not to any third party. The credentials can be used to apply different policies or for accounting purposes, but ultimately, the allocation of resources is between the PA and the UA – no one else. The MD-GTS protocol assumes the immediate upstream requesting agent is always the customer of record for all resources it has requested.

When the MDPA elects to forward a request downstream in search of appropriate resources, the MDPA assumes the role of User Agent to the downstream PAs. In addition, the MDPA uses authorisation credentials established bilaterally between itself (the local MDPA) and the respective downstream PAs.

“Atomic” resource classes in this MD-GTS context are not necessarily truly atomic – they are simply “opaque”. This makes them appear to the User Agent as devoid of any obvious internal structure. However, the provider PA may privately have a different – possibly more complex – description of that class with dynamic internal structure. This possibility allows opaque “atomic” classes to be advertised that can/will be treated as a single black-box instance by the users, but may constitute a much richer or more complex facility when reserved by the provider. It could be argued that GTS already supports private class descriptions in that a Provider Agent could easily instantiate an “atomic” resource using a complex internal structure that was not visible to the client. Such complexity hiding may be useful to the user, but the Resource Control Agents in an implementation – in order to support the virtualisation model – must act as an intelligent proxy for the multiple API requests/responses that could be generated downstream. The GEANT GTS v2 implementation does not support this feature at this time, but it does not preclude it from later versions of the GTS.

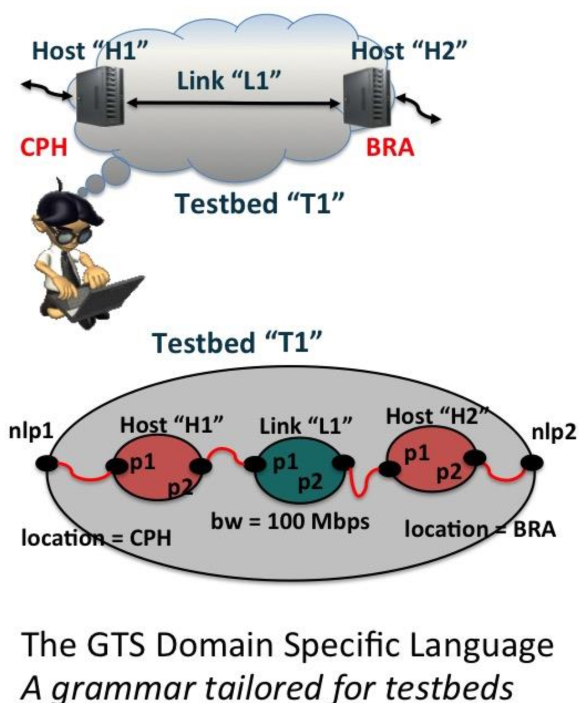
The order in which an MDPA will contact downstream PAs is configurable, but consideration should be given to the emergent behaviour of the distributed system that could result given different approaches. For instance, if all MDPAs try to acquire and re-allocate downstream resources before using their own local resources, a new Reserve() request would in theory recursively visit all peered downstream domains before a leaf node (with no other peer domains) would allocate its own resources. Such a tactic would be slow, and result in far flung resources. Therefore, it is considered an early best practice that MDPAs prioritise local PAs (LOPAs) above other downstream MDPAs.

The Resource Manager includes one simple heuristic optimisation when walking a class template to reserve resources: The RM will visit and Reserve() nodal resources (e.g. VMs or OFXs) and composite resources first leaving Link (virtual circuit) resources until last for each set of sibling nodes. This simple optimisation allocates resources with limited Port assignment flexibility first which allows those Ports to be resolved and mapped to specific Service Termination Points within the GTS infrastructure. The Link resources, by definition, can place their Ports wherever they are needed, and so Links are allocated last. After each VM or OFX resource is successfully Reserve()'d and its Ports are mapped to termination points in the infrastructure (STPs), the RM can walk the Adjacency relations to copy the STP information from the newly mapped VM Ports to the (unmapped) VC Ports of the adjacent resource. Thus, when the Link resources are finally Reserve()'d, the infrastructure end points for those Links are known.

Upon successfully instantiating and reserving a resource instance, a resource instance record is entered in the RDB that contains the ResourceID of the instance, and the Resource ID of the parent along with other appropriate instance info. Port records are also created, which correspond to the Ports defined in the new resource instance. These records comprise the "Resource Instance Tree" and are stored in the RDB.

3.4 The GTS Domain Specific language

GTS has defined a Domain Specific Language (DSL) for describing Testbed networks. The GTS v2.0 DSL is mapped on top of Groovy, an object-oriented programming language [GROOVY]. The GTS 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 GTS 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.




```
Class Testbed {
    def H1 {
        host {
           location = CPH
        }
    }
    def H2 {
        host {
           location = BRA
        }
    }
    def L1 {
        link {
           bandwidth = 100
        }
    }
    adjacency H1.p2, L1.p1
    adjacency H2.p1, L1.p2
    port { id = "nlp1" }
    port { id = "nlp2" }
    adjacency nlp1, H1.p1
    adjacency nlp2, H2.p2
}
Def T1 {
    Testbed {
        start = NOW
        end = Jul 1, 2014 08:00 GMT
    }
}
```

Figure 3.3: Example of the GTS DSL

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 GTS 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 GTS user portal, the DSL description can be imported and interactively processed by the GTS service agent.

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

```
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" }
               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 3.4: 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 GTS 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 conventional text 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 development and deployment of GTS. This remains the case for GTS v2.0. However, as the base functionality is deployed and more nuanced capabilities become the focus of the development effort, it is expected that the DSL will require two basic enhancements:

- First, the grammar will diverge from its Groovy base and develop into more of a data flow paradigm from its original Groovy based programming paradigm. The grammar will remain syntactically similar to the current object oriented DSL, but will require additional dynamic data description features.

- Second, the process of editing the DSL text will be replaced by a graphical drag-n-drop and point-n-click interface. Such a graphical construction process will allow network researchers and other network engineering personnel to define their network facilities in a more conceptual manner, consistent with many other widely used applications and GUI functions.

The GTS v2.0 DSL grammar is sufficient to construct what could be called "first generation" virtual network environments (testbeds). These are scheduled and dynamically provisioned virtual networks, but they are somewhat static. "Second generation" networks will deliver more advanced features such as dynamic in-situ modification, i.e. the ability to modify an active network without tearing down the entire existing testbed instance

and re-reserving a whole new instance. Such modifications involve changes to class definitions and parsing (e.g. versioning of class instances) of the DSL specifications. There remain some subtle semantic issues related to the DSL naming, how these relate to underlying infrastructure, and how such names ought to be seen/advertised in a globally scalable system.

For GTS v2.0, the user should view the DSL as a stepping stone for greater capabilities as the service scales up and matures. The GÉANT GTS service management team will work with the user community to ensure its requirements for the grammar are reflected in future versions, and that current V2.0 testbed descriptions are easily migrated to future versions of the GTS.

4 Virtual Resource Model

All allocatable objects within the GTS service model 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 LSPs, SDH circuits, etc.[EthernetFrame] [SDH]) In GTS, it is important to emphasise that all objects in the user’s testbed are *virtual* resources, created from a set of infrastructure components maintained by the GTS 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 particular type or class. The virtualised resource model enables the service to maintain consistent control of all GTS infrastructure in order to make sure it is appropriately managed and secure. Equally important, the virtual resource model allows the service to incorporate other novel resource classes without changes to the core management software or the architecture.

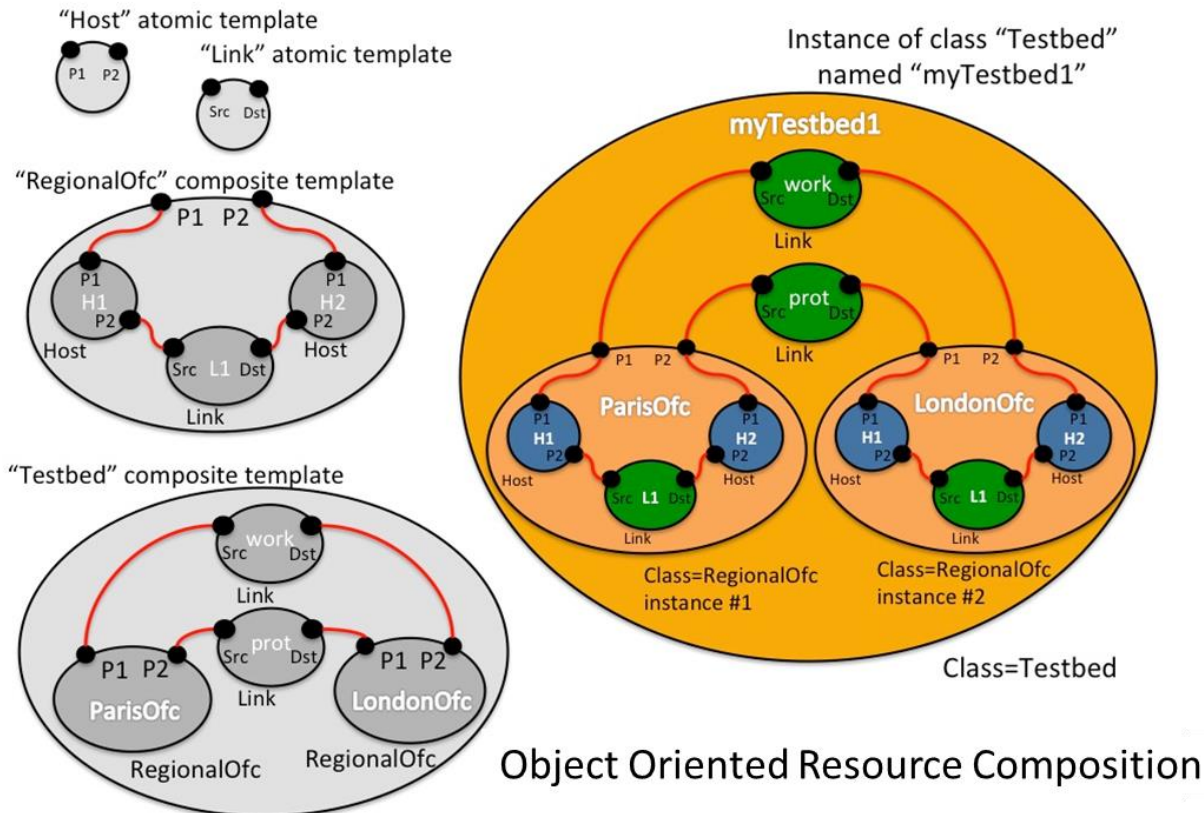


Figure 4.1: GTS object oriented virtual resource model. The DSL diagrams on left, the resolved integrated resource diagram on the right.

Virtualisation, in general, presents an *abstracted* service object to the user. This abstracted service object, such as a virtual machine or virtual switch, looks and feels to the user like the real thing, but is no longer tied to specific physical hardware. Software and often certain hardware features are combined to create the *abstraction layer* (or *virtualisation layer*) that provides a service instance (a resource) that has all the salient properties of the physical object, but is not directly tied to all or even part of a particular physical device. The virtual resource may be emulated in software, or – as is the case with GTS – it may be mapped to similar hardware but is no longer mapped to specific hardware and/or may only be allowed to utilise a portion of the performance capability of the hardware platform to which it is mapped. The virtualisation layer software decides how to map the virtual object(s) to the underlying physical infrastructure. Stated another way, the physical hardware forms the infrastructure raw materials that the abstraction layer manages in order to produce the virtual resources delivered to the user.

In some instances, the virtualisation layer may allow an almost direct mapping. For instance, each Bare Metal Server (BMS) resource in GTS will be mapped to an entire physical server, but the BMS virtualisation is defined such that a BMS resource can be mapped to any available physical server platform of the dozens distributed across the GÉANT footprint as long as that underlying platform meets the user's constraints [BareMetal]. Also, the virtualisation model for a resource such as BMS must ensure that the GTS service is always able to recover the resource when the resource reservation expires, or if the resource fails or goes rogue for some reason. In

the case of the BMS, for instance, there is a "lights-out" interface available to the GTS resource manager that allows the GTS software to interrupt the BMS resource and halt it or completely reinitialise the physical server platform should it be necessary. This management interface is part of the infrastructure layer, and is not accessible by the user as part of the virtual BMS resource. 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.) The virtualisation layer ensures that the GTS service retains authoritative control of the underlying infrastructure and resource management processes – this allows the service to reliably recover or to reinitialise virtual resources to a known state between user reservations.

It should be noted that true virtualisation allows the layers to be stacked without limits. This is not always the case in the GTS implementation. For instance, the BMS virtualisation only supports a single BMS virtualisation layer as there are no plans to promote the lights-out remote access and control interface to the virtual resource layer, which would be necessary to make the resource a truly virtual resource. In this case, we have implemented more accurately a "delegation" layer. This is similar to OpenFlow Fabric (OFX) resources, where the user is able to designate a physical port to act as the control port for an OFX instance in the infrastructure layer, but that ability is not promoted to the virtual resource model due to third-party software limitations. GTS provides virtual resources, but these virtual resources are not always complete abstractions of the underlying hardware that they mimic. Still, the virtualisation provided allows a powerful dynamic system that can substantially reduce set-up time and accelerate the research or other operational objectives.

GTS virtual resources constitute very real computational, switching, and transport facilities. In general, GTS virtual resources are mapped to real hardware and are not emulated. Virtual machines and bare metal servers provide real computational horsepower, and virtual circuits can move 10s of gigabits of data each second. There is potential for such a powerful testbed capability to cause problems if it is allowed to interact with other testbeds or to impact other non-testbed facilities. The GTS resource model keeps all data transport circuits internal to the testbed unless there is a data flow port explicitly defined at the testbed boundary. Thus any data traffic entering or leaving the testbed must transit these ports.

By default, a user's testbed is completely isolated from all other networks. This prevents the experiment from interfering with other experiments or production services that may be sharing infrastructure. In turn, this isolation prevents other unrelated testbeds from interfering with a user's experiments. However, this strict isolation limits access to the testbed, even by the researchers that requested it. GTS provides several features that enable users to reach and interact with their testbeds and/or the individual virtual resources that make up their testbed.

4.1 Testbed Interaction with the Physical World

In general, the user is allowed to access the resources in the testbed. "Access" normally means Internet access such that the user can log in to the resource to configure the resource or otherwise control its behaviour. However, the degree of access and the means of access vary depending upon the resource class and its virtualisation model and the specific requirements of the researcher. The computational resources are the resources that most often require user accessibility. These are virtual machines and other classes, such as Bare Metal Servers These resources tend to have mature virtualisation support, which limits user access to functions within the virtual context of the VM or BMS. The user is unable to break out of that resource context. Resources such as switches

or routers would generally have access as well, but these depend more upon the virtualisation support that is integrated into the devices that make up that infrastructure. Granting the user access to a console port on a virtual router that allows the user to see and potentially modify the hardware could pose a major security problem to other services or testbeds sharing the same hardware device. An invariant rule is that access to the virtual resource cannot allow the user to subvert the virtual context of the resource and take over, or otherwise access, features in the hardware that are not explicitly part of the resource abstraction model. Other resources, such as virtual circuits, completely lack a concept of user access.

If a resource model allows a functional capability within the context of a resource instance, but that function requires user privileges that violate virtualisation or otherwise compromise security or isolation, then that function must be performed by a GTS proxy. These privileged functions can be defined as API extension primitives for the given resource class, and the GTS core resource manager will perform the function on behalf of the user.

Strictly speaking, the user should indicate where or how all resources are accessed from other resources or from external facilities. However, for most resources, GTS recognises that simply allocating the resource without enabling some means of easy, but secure, access to allow the user to configure the resource to his/her needs would pose a major obstacle to productivity. Therefore, GTS facilitates practical access to resources.

Console Access

The most commonly acquired resources are VMs. VMs are instantiated with console ports, which are accessible via a web-proxy interface. This console link is presented in the GTS GUI. The console port allows the user to see the VM boot, and provides a means to login and perform basic OS configurations. This is the simplest means for a user to access and interact with his/her VMs.

Edge Ports

Testbeds can interact with the external world by defining edge ports that transit the testbed boundary as part of the testbed description. In the same way as ports can be defined between resources within a composite, the composite can define ports that allow data to enter or leave the testbed. These edge ports can be connected to other external services, such as BoD circuits, or static tunnels to other testbeds. Like the IAGW port, these user-defined external ports are closely monitored, as they pose a potential security risk to or from external facilities.

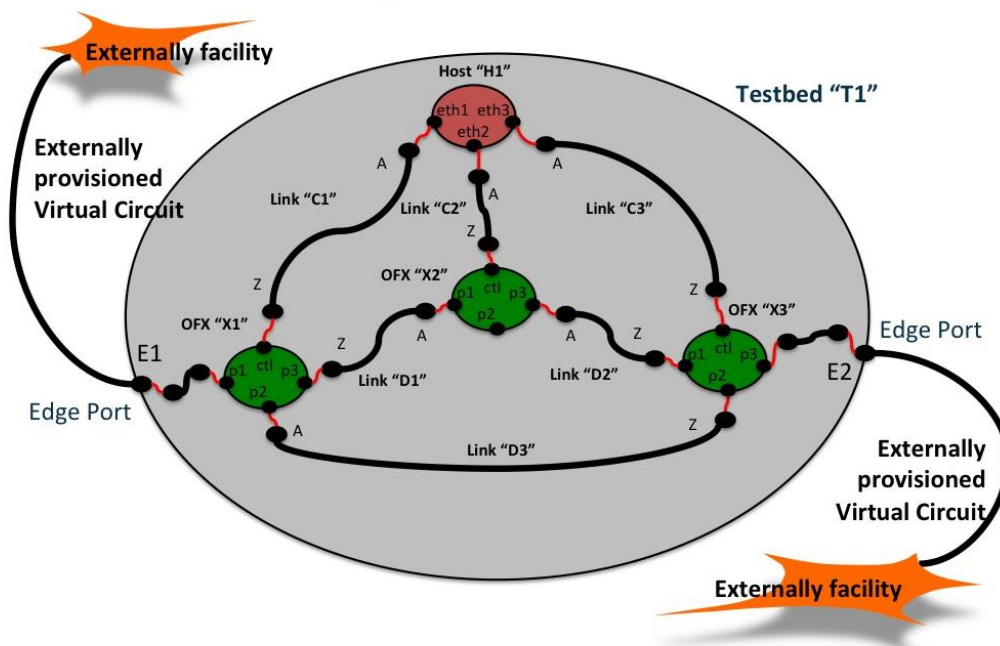
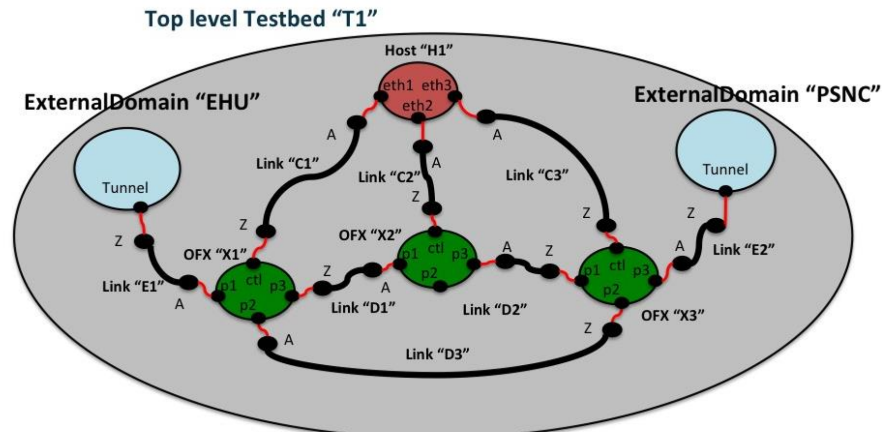


Figure 4.2: Testbed Edge Ports are linked to external facilities via separately provisioned virtual circuits

External Domain Classes

GTS supports "External Domain" (ED) resources that appear in a GTS bubble diagram as just another resource, similar to a Host. External Domains are a class, and the instances are predefined and associated with a preconfigured tunnel endpoint. In this manner, a static circuit (tunnel) could be manually set up to link from the GTS domain to some particular external facility (such as a university lab or some other testbed) and the user can include that ED resource in his/her testbed description.

External Domains as GTS Resources



When defined as GTS External Domains, external facilities can be manipulated within the testbed DSL just like other resources.

Figure 4.3: External facilities can be defined as external domain classes and manipulated like other GTS resources.

Internet Access Gateway

The GTS also provides an Internet Access Gateway (IAGW). The IAGW is established when the user creates a new project within the GTS service. The IAGW consists of a VPN gateway server that has a public Internet interface. This VPN server is linked to an internal VLAN assigned and dedicated to the user's project. Any resources assigned to the project are allowed access to this internal, general-purpose VLAN. A private IPv4 and IPv6 address range is assigned for DHCP allocation on the internal VLAN, and NAT services and a default route to the commodity internet are provided. The user is assigned a login for the IAGW VPN server. When the user successfully logs in to the VPN, s/he becomes part of the internal IAGW subnet. As resources are reserved and assigned to the project, those resource are able to connect to this internal IAGW subnet. This internal IAGW subnet is available to all resources assigned to the project – not just those in a particular testbed. It is important to note that since testbeds are simply composite resources, a single project may have multiple composite resources – testbeds – reserved and/or active at any given time. However, resources assigned to other projects, and therefore different VLANs, cannot access resources allocated under other projects without prior explicit data flow engineering.

For VMs, the Activate() primitive will extend the IAGW VLAN to the virtual eth0 interface of the VM, and the VM will configure the appropriate sub-interface to be part of the IAGW IP subnetwork during the initialisation of the VM. IPv4 or IPv6 can be used within this subnet.

For OpenFlow fabrics (OFXs), the control port can be placed in the IAGW subnet. This can simplify access to OFX fabrics within a testbed because all fabrics can join the same subnet as the controller. A Layer2 Broadcast Domain resource class will be introduced in a future release as a formal GTS a resource class. This will simplify many network configurations and enables the user to directly engineer network features.

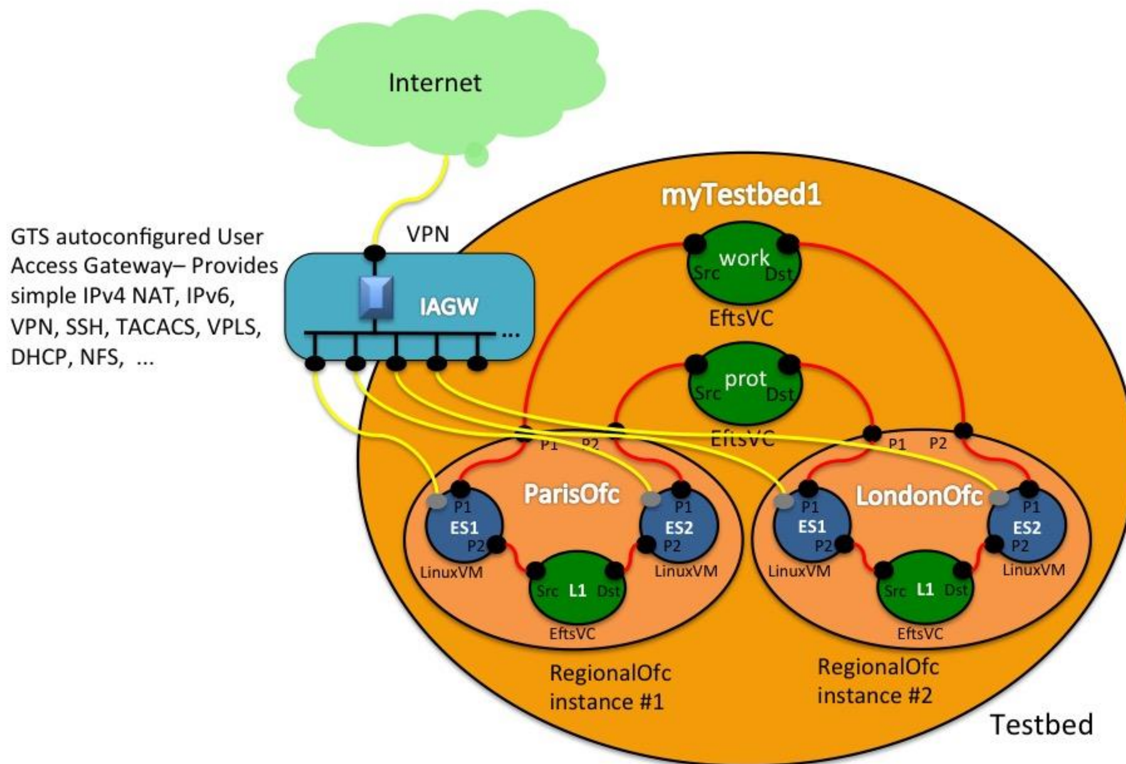


Figure 4.4: IAGW within the testbed

The IAGW is a 'best-effort' service and is implemented using a virtual routing forwarding table (VRF) across the GÉANT IP core. This is essentially a VLAN that appears in each Pod and is bridged to appropriate VMs during activation. 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 interface linking the IAGW subnet to the outside world is shared by all projects – i.e. all IAGWs for all the projects managed by a GTS domain will typically share the same total capacity to the Internet (this is not an architectural requirement, but is typical, and currently all IAGWs share a single 1Gbps port). Inside the GTS infrastructure, the IAGW subnets are provisioned over 10Gbps infrastructure but they should be constrained to a useable but capped bandwidth so that no single IAGW can monopolise the shared external interface and interfere with other projects. The IAGW subnet is a general purpose subnet and is available for the user to leverage within his/her experiment. It is not solely for external access.

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

Network Attached Storage

The GTS will also provide access to network-attached storage (NAS) across this IAGW subnet. The NAS server [NAS] will offer Network File Systems (NFS) [NFS] that can be accessed externally by GTS users to build VM

images, to stage other data sets, to capture a testbed state, or as generally required by experiments. These file systems can then be mounted during the VM boot process to make user files immediately available to VMs.

.

5 Infrastructure and Virtualisation Layer(s)

The physical hardware from which the GTS creates the virtual resources is called the “infrastructure”. The physical server platforms installed in the GÉANT Points of Presence (PoPs) that are used for VMs are part of the infrastructure, as are the switches that provide the OpenFlow fabrics (OFXs). The optical wave services provisioned between the GTS Core PoPs over which the VCs are provisioned are also considered infrastructure.

The KVM hypervisor software [KVM] that runs on the physical servers (that set up and supervise the context switching of the CPU for each virtual machine) and the OpenStack software that runs on a separate control server (that manages the VM creation) is all considered part of the virtualisation layer. The virtualisation layer uses the raw materials of the infrastructure to produce and present a consistent, virtual resource model to users.

Put simply, the GTS infrastructure is the underlying raw materials that are input to and managed by the virtualisation layer software to produce virtual resources. The output resources of a virtualisation layer may be acquired by other virtualisation layers – in which case, the resources produced by the first layer become infrastructure to the upper-layer software. For instance, rack mount servers or blade servers may act as infrastructure to the virtualisation layer that generates Bare Metal Server resources. Another virtualisation layer module may use Bare Metal Server resources as input infrastructure to produce Virtual Machines as output resources. This abstraction means that “infrastructure” and “resources” can be used to refer to the same objects but imply a relationship to the virtualisation process.

5.1 Virtualisation Software Layer

The software Resource Control Agents that generate the virtual resources are – when taken as a group – referred to as the virtualisation layer. This layer of software is typically considered to exist “above” the infrastructure layer and simultaneously “below” the resource layer. This virtualisation layer is accessed using the same API that allow the user agent to interact with the Multi-Domain Provider Agent to interact with all of the GTS virtualisation functions. The virtualisation software includes other third-party software that is not typically visible to or accessible by the GTS user. This third-party software includes:

- The KVM hypervisor software that runs on the physical servers.
- The OpenStack software [OpenStack].

- The OpenNSA provisioning system [OpenNSA] that manages the virtual circuit provisioning.

The GTS virtualisation layer may employ many other software tools or packages in addition to the software specifically developed by the GÉANT project [GN3plus]. The GEANT GTS implementation uses the OpenStack portfolio of tools to manage VMs. Other GTS implementations could as easily use other tools, such as VMware or Parallels [VMWARE] [PARELLELS] to virtualise the hardware facilities and present the virtual machines upward as resources. As long as the virtualisation model for the resource class is consistently honoured and presented across all providers (i.e. all properties defined in the class definition are presented to the user in a consistent manner) it does not matter which underlying tools are used to produce that virtual model. The user sees only the virtual machine representation defined in the GTS "Host" resource class definition. The GTS 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 Machine resource control agent (RCA-VM) adapts the GTS API to appropriate OpenStack interactions to allocate and initialise virtual machines. The OpenStack software suite was selected for GTS internal use in order to facilitate access to independent large scale data centres in the future to complement the VMs in the GEANT Core. This feature, when fully implemented, could make tens of thousands of VM resources available in geographically important regions that GÉANT can leverage for research or production services.

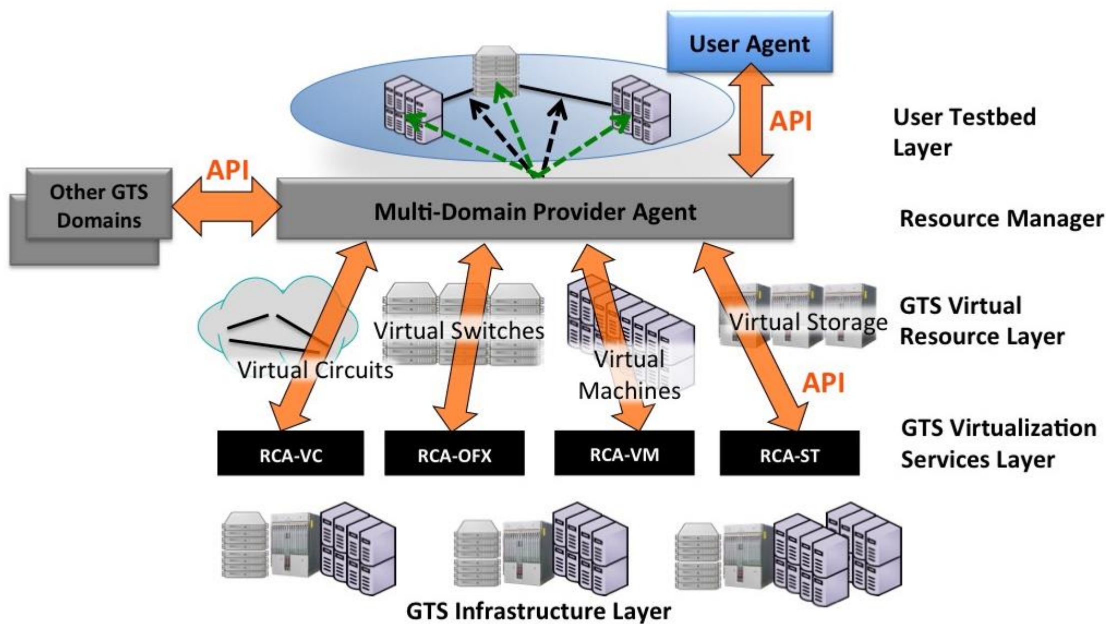


Figure 5.1: GÉANT GTS virtualisation, management and control layers

The virtual circuit (VC) resources employ a similar underlying infrastructure within GTS. GTS has dedicated network transport capacity (specifically 10Gbps waves) set up between all of its core PoPs. GTS uses OpenNSA circuit provisioning software in conjunction with MPLS capable hardware switching devices and their operating software to manage this transport capacity and to create VC resource instances. (The GÉANT implementation currently employs Juniper MX routers and JunOS operating software.) The GTS resource manager interacts with the OpenNSA provisioning software using standard NSI-CS v2.0 [NSI-CSv2.0], provisioning protocol. Because

GTS VC virtualisation uses the NSI standards for circuit provisioning, it will easily interface to external provisioning systems that also use NSI, such as the GÉANT Bandwidth on Demand service. The GTS virtual circuit resource model views the NSI transport infrastructure and the NSI protocol standards as part of a global, inter-domain architecture. This will allow GTS testbeds to integrate VMs or other resources in other GTS domains (such as the GTS service domains operating in the NRENs) into a single user environment spanning multiple underlying service domains.

5.2 Physical Infrastructure

The GTS physical infrastructure is geographically distributed with other GÉANT services in the GÉANT PoPs around Europe. The GTS physical infrastructure is assembled into a set of equipment "pods" that contain the hardware components that make up the VMs, VCs, and other resources.

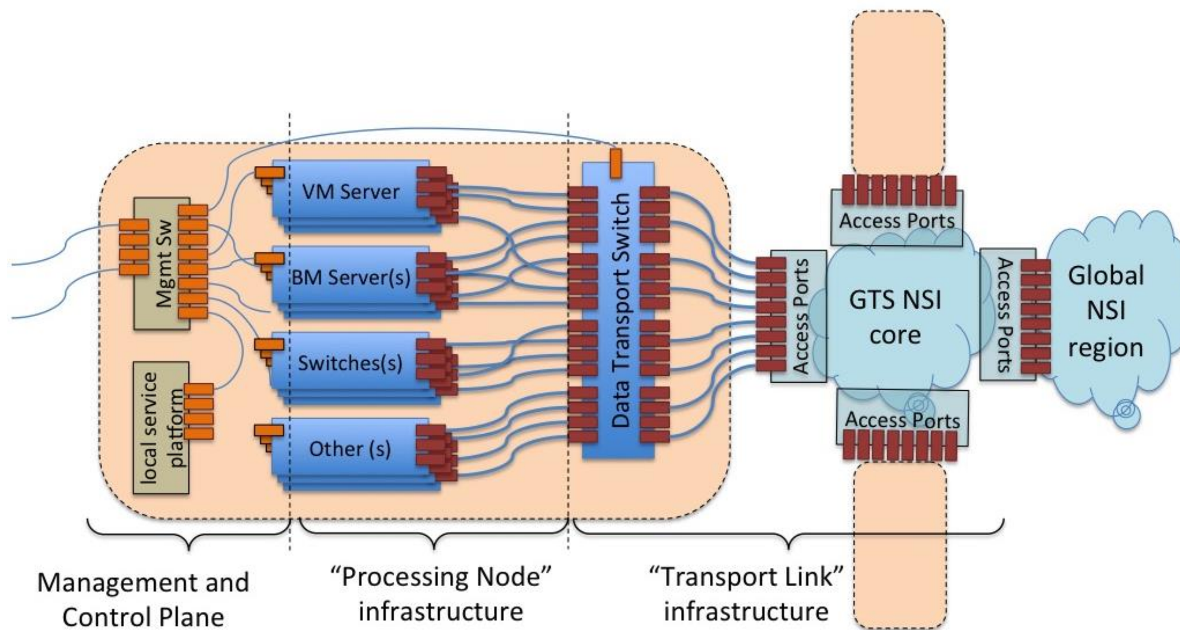


Figure 5.2: The GTS pod physical design

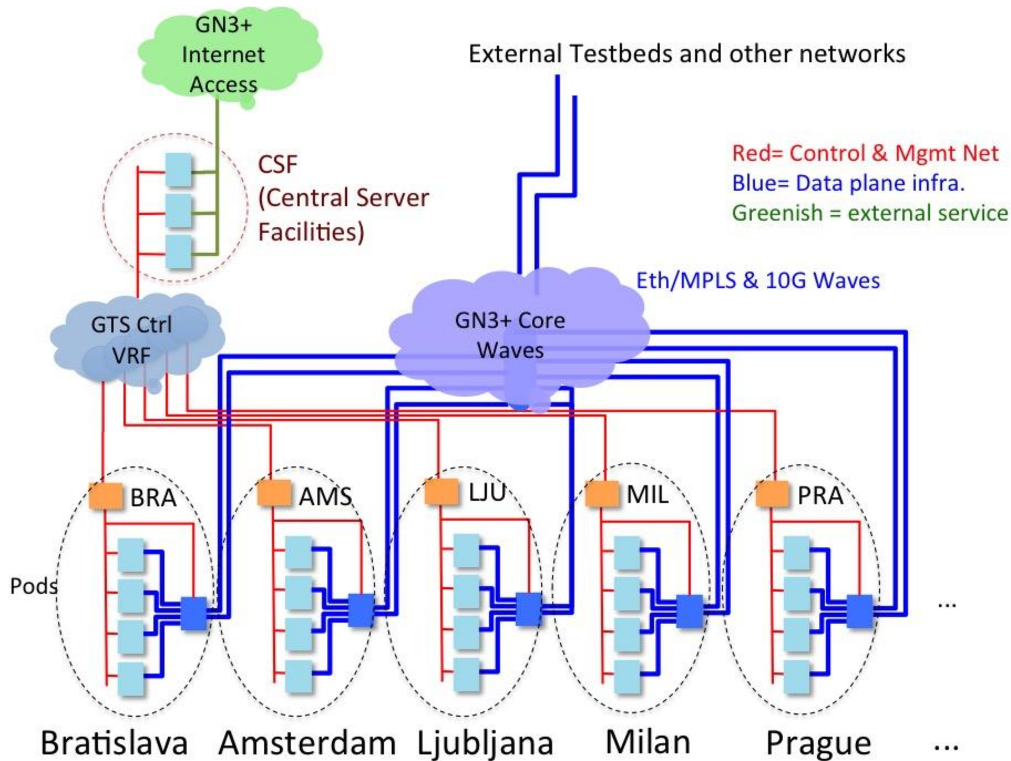


Figure 5.3: The GTS distributed architecture across the GÉANT core network

GTS pods are designed to be scalable. The various infrastructure components are engineered so that they can be interconnected by transparent data plane transport circuits. 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.

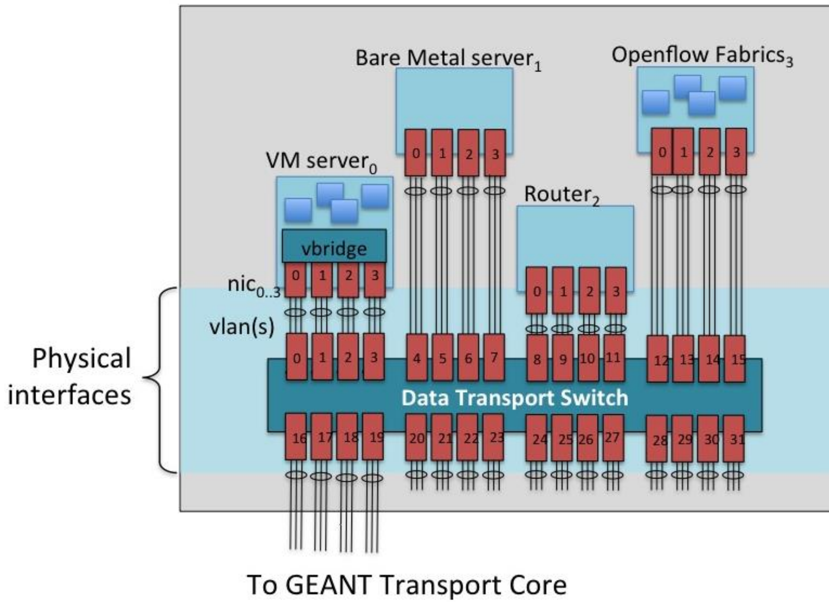


Figure 5.4: Pod infrastructure platforms

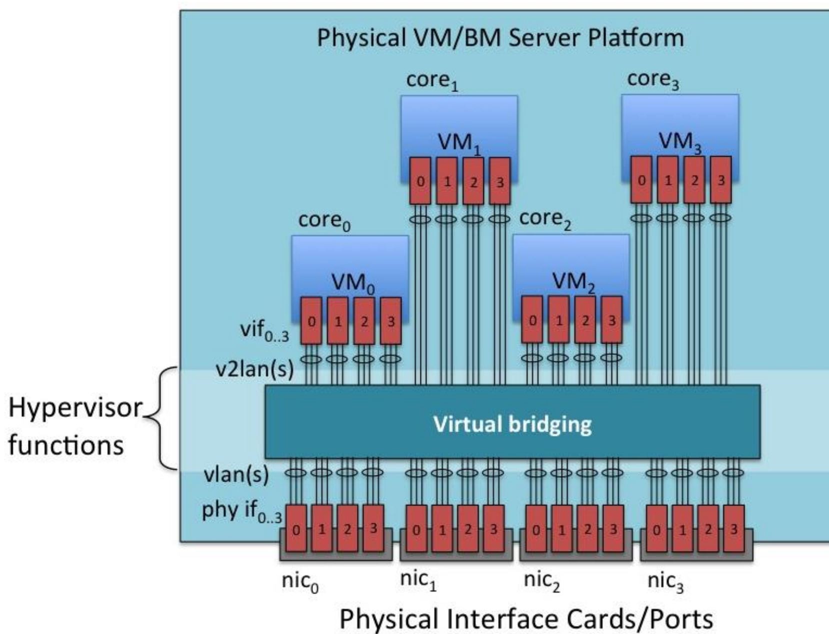


Figure 5.5: Resource mapping: VMs to servers

The physical infrastructure pool is always controlled by GTS, and the user is never allowed to usurp that control. It is fundamental to the GTS service that GTS 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, GTS 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 GTS 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 physical device's operating system – all (user) 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 extension control primitives that could interact with the underlying hardware via a GTS proxy. The GTS resource control agent will ensure the user's request does not violate virtualisation and will not impact other virtual instances on the same device, and then GTS will perform the requested function and return the results to the user agent. 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 provide a limited virtualisation of the console capabilities.

The GTS 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.

It should again be noted that the virtualised model presented by each resource is not necessarily a complete model of the underlying hardware. It may not always be possible to stack the current virtualisation layers in the current implementation. For instance, with Bare Metal Servers, full virtualisation would present a virtualised maintenance interface for the resource instance that would allow another Bare Metal Server virtualisation layer to be stacked on top of the base layer. This is not presently implemented, and so a second Bare Metal Server layer would not be able to recover the device from a user without releasing the underlying resource altogether – which breaks the virtual model of a BMS. This capability is not seen as critical in the near term, although a more thorough virtualisation model would, in theory, offer further options for future services.

In the GTS architecture, there is nothing that prevents this multi-layer virtualisation, albeit particular resource models or implementations may have limitations. In addition, 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 testbed services where internal provisioning processes and policies may not be publicly shared.

In summary, GTS offers virtualised resources to the user. Those virtual resources are constructed from the GTS infrastructure facilities. There is a sophisticated layer of software that constitutes these virtualisation layers.

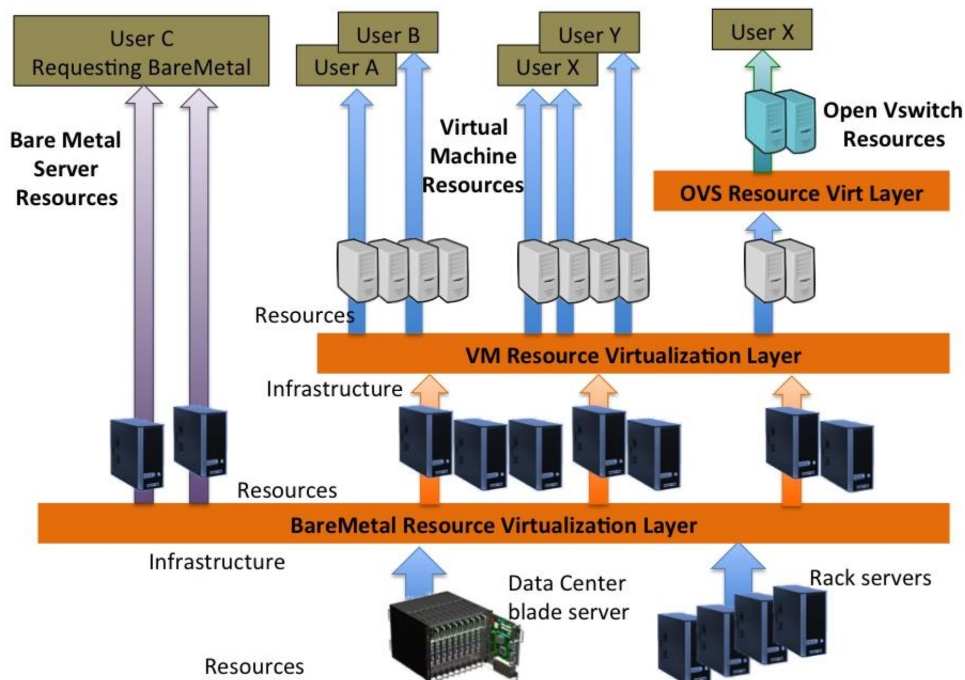


Figure 5.6: The GTS infrastructure, virtualisation software, and resource layering

6 The GTS Resource Database

The GTS must maintain a database of information about:

- Users.
- Projects to which the users belong.
- Class definitions (DSLs) for the resources and testbeds associated with the project(s).
- Virtual resources that have been allocated.
- Scheduling information and requested and proffered attributes associated with each resource instance.
- Present operational state of all resources.
- Domain from which those resources were acquired.
- Underlying infrastructure elements from which those resources were created.
- Software tools that produced the resources from the infrastructure.

All of this information must be persistent and must remain harmonised and congruent across multiple layers of management services for each class of resource. This information is stored in the Resource Database (RDB).

There are a number of key data structures that are critical to the functioning of the core GTS service, such as the project data structure, list of users, list of resource class templates and list of resource instances allocated to the project. These data objects and the way they are created and internally managed are presented below.

6.1 RDB Data Structure

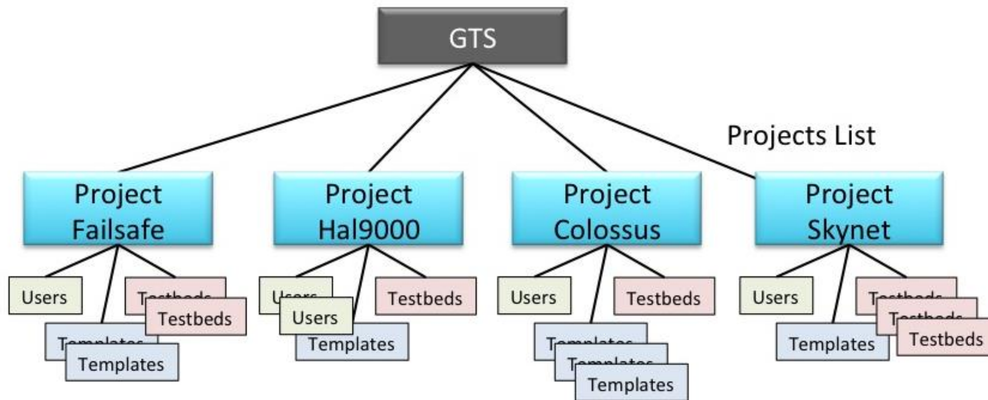


Figure 6.1: The Project object and its structure within the Resource Database

The "project" object (further called ProjectID) is the key element of the GTS internal information model. The project data structure (as shown in Figure 6.1 and Figure 6.2) functions as a basic accounting object and is the root of a tree of other various information about a testbed and its users. The project data structure contains three basic constructs: 1) a list of users that are authorised to request or manipulate resources allocated to the project, 2) a list of resource class templates that users belonging to that project have defined, and 3) a list of all resource instances that have been allocated to the project.

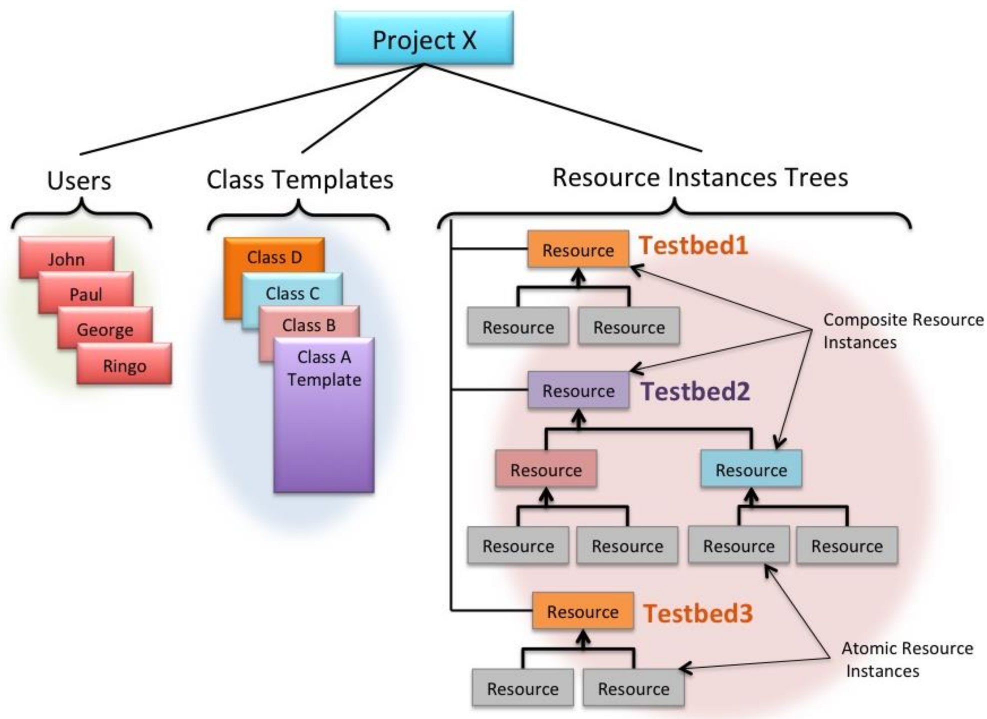


Figure 6.2: The user and project organisation within GTS

Users assigned to a particular project are allowed to reserve, or otherwise manipulate, resources assigned to that project. When a user reserves a resource instance, the resource class template is processed by the GTS provider agent to create an internal “resource tree” that describes the fully resolved resource instance (Figure 6.2).

For instance, if a user requests a VM resource in Amsterdam, the GTS 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 existing VMs are available, then the GTS RM has several choices: a) it could search other GTS domains to acquire conforming resources from other providers and repackage these resources as its own and allocate them to the user, b) it could possibly check to see if additional VMs could be dynamically created, locally, or c) it can terminate and reject the Reserve() request. If VMs are ultimately acquired, the RM will mark them appropriately as allocated, and keeps the state of those resources in the Resource Database (RDB).

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 a specific project.
- The list of all resource instances that have been created by users in the project group.

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 GTS service. Future GTS versions will explore requirements for a more flexible and generalised model as the GTS multi-domain and federated AAI aspects are developed.

As mentioned in Section 3, the parsing of the resource template does not create a resource instance – parsing simply walks the class definition incorporating child 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. This user-assigned “local name” is unique within the scope of the DSL resource class description, but this local name may not be unique within the testbed as a whole once all classes have been resolved. Therefore, the resource provider (GTS RM) will also assign a unique Resource (instance) ID 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 internally by GTS software modules to specify a particular resource instance.

Drilling down, there are two important data structures that must be defined:

- A resource "template" or class descriptor.
- The resource "instance" object.

These objects must be persistent in order for GTS 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.

6.2 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 "Define LinuxVM {} "). This is the identifier used to refer to a particular class (*not* an instance of that class). One can think of the class as a name for some arbitrary group of resource properties.
- **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" in the context of 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). As previously mentioned in Section 2.6, at a minimum, all resources must define the five basic methods (Go5) for managing a resource instance though its lifecycle: Reserve(), Release(), Activate(), Deactivate(), and Query(). A resource template may define additional methods that are available for that particular class.

> Note that in GTS v2.0, user-defined composite resource templates are not allowed to specify user-defined Methods. There are a number of security issues relating to data integrity and database coherence that must be resolved before user written RCMs can be dynamically incorporated. So for now, such user-composite resources are mapped to a generic set of resource control methods by the GTS 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.

6.2.1 Resource Records

All resource instances are represented in the RDB by a Resource Record. The resource record contains a Resource ID – an identifier that is unique among all resources provided by the local Provider. The Resource record also contains the resource class, the user assigned resource instance name (unique with the scope of an encompassing composite resource), and any key/value pairs describing resource attributes.

In some cases, resource instances may be created before they are requested by a user. This can substantially reduce provisioning time for many types of resources. These pre-provisioned resources are referred to as “static” resources in that they are not dynamically created as they are requested. For instance, VM resources can be pre-provisioned and booted up and left running in an idle state (or a hibernated state) until they are ready to be activated. They are reserved like any resource. When an Activate() request goes to place them in service, the resource is woken up and some user-specific initialisation is performed (e.g. mount a particular network file system, or register the project’s users within the host OS, etc.). This user initialisation can be done quite quickly and so the resource is ready in a few seconds rather than a few minutes. Large-scale testbeds can then provide substantial speed up during Activation. When static resources are released, they are re-initialised and replaced on the shelf ready for the next reservation. Static resources must be sterilised between activation to ensure privacy of user information – this may extend so far as to re-formatting disks to make sure data from one session cannot be acquired by a subsequent session. Within GTS, static resources are assigned to a “GTS_available” Project in between activations. When an Activate() request is received, the RM re-assigns the resource to the target project and completes the user-specific initialisation. A static resource instance may have multiple reservation records in the RDB. Static resources are most useful for highly generalised or highly flexible resource requirements, i.e. many of the properties of static resources cannot be adjusted during the lifetime of the instance. In order to allocate a static resource to a request, the resource instance properties must be able to match a wide variety of resource requests. The instance must either be highly general, or the request highly unspecified so that free (unspecified) attributes can be matched by the fixed attributes of the static resource.

“Dynamic” resources are created on-demand. In general, this means there is no resource record associated with the resource instance until a reservation is made. For these resources, the Activate() request cannot begin until the reservation start time has arrived, and so the full infrastructure provisioning and cold initialisation is required at start time. Upon release, all infrastructure assigned to this resource instance is released and the RDB

resource/reservation records are archived and removed from the RDB. A dynamic resource instance has only a single reservation record in the RDB.

When a Reserve() request is received for a particular resource class, the GTS core Resource Manager (RM) passes the Reserve() request to the Local Provider Agent – a Resource Control Agent - for that class. The RCA will query the local 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 attributes that the user might specify, the user's authorisation credentials (dictating which resources are reservable by the user), and temporal scheduling windows.

If no existing, static resources meeting the user constraints are found in the RDB, the RCA 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 RCA 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, the RCA-VM would request that OpenStack create 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 RCA will look for an additional platform in Amsterdam that can be (or already is) allocated to OpenStack as a compute node, and then, the RCA-VM will request a VM from that new piece of infrastructure. If resources are not available from the local domain, the RCA-VM will reject the request back to the RM (The MDPA). The RM then has the option (under local policy control) to search other collaborating domains to acquire a suitable resource from one of those domains.

Upon creating a resource instance, the RCA assigns a GTS Resource Identifier (RIID). The RIID is unique within the scope of a GTS service provider domain. For example, within the GÉANT GTS service, no two RIIDs will ever be the same. Along with the RIID, the Resource Manager will also write the Provider ID of the downstream provider agent along with the downstream ResourceID returned by the downstream PA. Thus, the Resource Record contains a) the local Resource ID generated by the local RCA that will be presented to the upstream requester, and the downstream PA and downstream RIID. The resource record also contains the set of attributes requested by the upstream user agent, and the fully resolved attributes of the resource class that were assigned or defaulted by the downstream PA. These fully resolved values are called the "As-Built" attributes. The Resource record contains the Project ID under which it is assigned, and the User that reserved the instance.

The Resource record in the RDB points at the parent Resource RIID in which this instance resides. Therefore, all RIID records that point to the same parent RIID are the internal (child) resources of that parent composite resource instance. The resource instance tree can therefore be discovered by recursively querying the RDB for appropriate parent ResourceIDs.

Each resource is allocated on a scheduled "book ahead reservation" basis. This time window is represented by a reservation record in the RDB. The Reservation record specifies a Resource Instance ID to which it applies the start time and date of the reservation, the end time and date of the reservation. As some resources may exist "on the shelf" without any reservation, or alternatively, a given statically allocated resource may be delegated in turn to different users/projects, a separate Reservation record is stored in the RDB for each window in which the resource is assigned.

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 visible or available to user agents, 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.

7 The Resource Lifecycle

7.1 Resource Reservations

A resource is *allocated* when it is reserved, although it generally is not actually instantiated until the data plane is Activated. Allocation 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 in the meantime. 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 start/end time and date window for that reservation. Although a particular instance is individually identified in the RDB calendar, and the infrastructure raw materials necessary to realise that instance have been allocated, 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, as does the resource “instance” itself (the “spirit” of the virtual object.) But the resource instance is only required to exist once the Activation primitive is successfully confirmed.

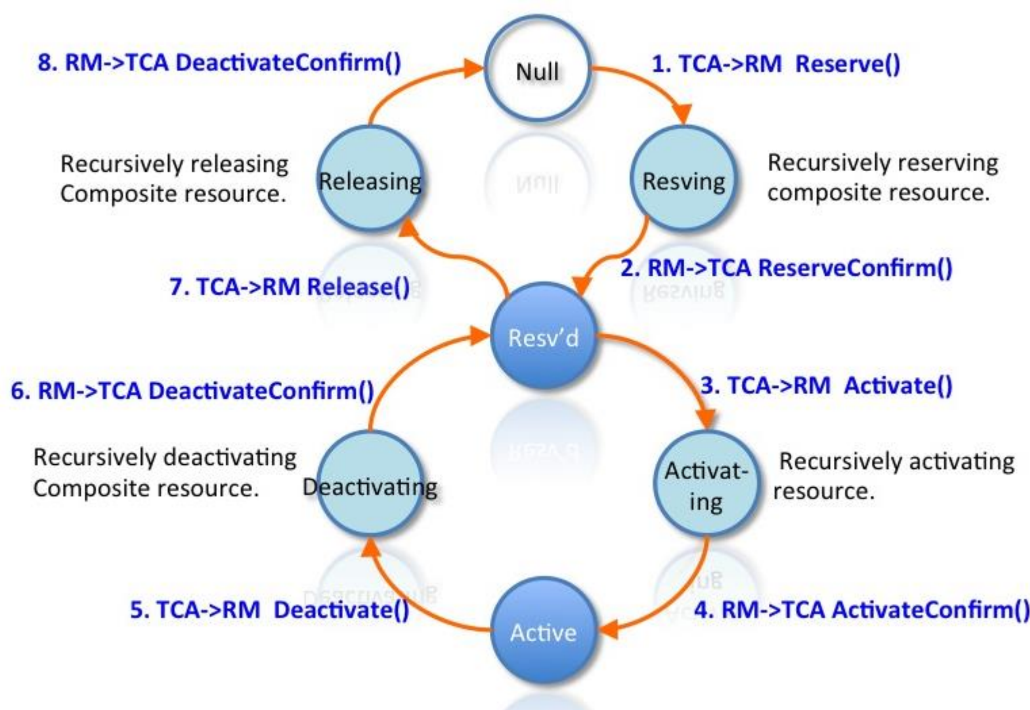


Figure 7.1: Reservation lifecycle state machine

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 agent 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 UA, and b) within its schedule service window, in order to become Active.

7.2 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 GTS 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. A future enhancement to GTS will provide Notification() primitives that could alert the User Agent to impending events such as a Deactivate() possibly due to a hardware failure requiring some temporary interruption of service, or a Release() heralding the arrival of the reservation End Time. These are not currently implemented in GTS 2.0.

The User Agent 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", or "Active", or in a transient state such as "Activating". 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 document as defined for that specific resource class. The UA may interpret the state and As Built information according to published documentation for that resource class. 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, application-specific interrogation.

7.3 Resource Instance States

A resource instance exists in one of three basic "stable" states: Unallocated, Reserved, and Active. GTS 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 UA 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 or forensic information.

8 The Resource 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.

As previously stated, in order for GTS 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, GTS 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 GTS Resource Control Primitives and Lifecycle of a Testbed/Resource Instance

A testbed is described by a text-based DSL document. This DSL 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 testbed resource instances are organised in a tree structure, with the user's overall testbed as the root "Resource". The root testbed resource instance, and the other resource instances 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" is the 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 GTS API. The API asserts a control protocol consisting of a set of primitive functions between the User Agent and the Provider Agent(s) that allow these agents to effectively manage a group of resources.

The UA 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 description as a whole be sent to a Provider Agent and entered into the class library under the User's ProjectID. A Reserve() action may be subsequently sent from the UA to the PA to allocate and instantiate the Testbed. A reservation action may specify a start time of "Now" (the default start time if unspecified) or an End Time of "Never" (ridiculously far in the future, not actually infinite.) Policy is applied to all Reservation requests.

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. The states consist of "stable" states that a resource instance may occupy for an infinite period of time, and "transient" states that indicate a transition is occurring from one state to another that will be allowed only a finite amount of time to complete before some recovery action is initiated. Transient states are typically short – a few seconds is normal, a few minutes at most.

As stated in Section 7.2, 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 detailed information associated with each state 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 immediate downstream provider agent 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 GTS Application Programming Interface Protocol

The API protocol consists of a set of basic functions ("primitives") that occur between the UA, the Multi-Domain Provider Agent(s) (MDPAs), and the Local Only Provider Agents (LO-PA, or RCAs). The reader is directed to the User/Resource Guide for a detailed description of all API administrative primitives, lifecycle control primitives, and extension primitives suitable for software development. The basic control protocol consists of request/response messages that move resources through their lifecycle, as follows.

Define(). The UA sends a DSL description to the PA. The PA will parse the DSL and add the class(es) to the Project Library to which the user belongs.

DefineResponse(). From the PA to the UA, this response indicates if the DSL was successfully parsed, or alternatively if errors or other warning were detected.

Reserve(). The UA sends this primitive to the PA 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 PA 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 identities the client entity and ProjectID 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 resolved" (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 PA sends a ReserveResponse() message back to the requesting UA. 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 PA. 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 UA to the PA, when the UA wishes to place a resource into service. Since resources are scheduled, the resource cannot be realised and activated 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 activating them should begin (time 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 and placed 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 UA (i.e. the user). The Resource Control Agents (the Local Only resource managers) may implement additional, resource-specific primitives. These resource-specific primitives will be defined in the Resource Guide document that describes the GTS Resource capabilities.

ActivateResponse(). This message is sent from the PA to the requesting UA, 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). As some Activate() requests may take some time, the Activate() may return prior to completing the Activation of all resources covered by the request. The returned state will be “Activating”, and the UA may periodically poll the resource instance to determine when the instance has entered the “Active” state.

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. Therefore, it is the user’s responsibility to save the internal user state of the resource before Deactivate() ing an instance.

DeactivateResponse(). From the PA to the UA, 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. It is likewise expected to clear any trail in the infrastructure or underlying resource specific databases or configuration state. 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 the RDB in a pristine state.

ReleaseResponse(). Sent from the PA to the requesting UA, 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 UA, 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 Quick Search Box (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(), Query(), Deactivate(), Release()) constitute the common control protocol for the testbed service. A resource may define additional primitives that are implemented by the Resource Control Agent(s) (for instance a Halt() primitive or a Reboot() for a Virtual Machine) that allows the User Agent to perform specialised control functions for particular resources. However, since these functions are specialised, it is up to the UA to understand what these primitives do. In addition, these resource-specific control primitives should be well documented as part of the Resource Guide.

References

[BareMetal] https://wiki.openstack.org/wiki/Baremetal
[EthernetFrame] https://communities.netapp.com/blogs/ethernetstorageguy/2009/09/12/anatomy-of-an-ethernet-frame
[GN3plus] http://www.geant.net/Pages/default.aspx
[GROOVY] http://groovy.codehaus.org/
[GTSWEB] http://services.geant.net/GTS/Pages/Home.aspx
[KVM] http://www.linux-kvm.org/page/Main_Page
[NAS] http://en.wikipedia.org/wiki/Network-attached_storage
[NAT] http://en.wikipedia.org/wiki/Network_address_translation
[NFS] http://en.wikipedia.org/wiki/Network_File_System
[NSI-CSv2.0] http://www.ogf.org/pipermail/nml-wg/2014-January/001368.html
[OpenFlow] https://www.opennetworking.org/sdn-resources/onf-specifications/openflow
[OpenNSA] https://github.com/jeroenh/OpenNSA/wiki/Getting-Started
[OpenStack] http://www.openstack.org/
[PARALLELS] http://www.parallels.com/
[SDH] http://www.rfc-editor.org/rfc/rfc4842.txt
[TAASSPEC] https://intranet.geant.net/SA2/_layouts/15/DocIdRedir.aspx?ID=GN3PLUS13-836-57
[**VMWARE**] http://www.vmware.com/uk

Glossary

AAI Authentication and Authorisation Infrastructure
AMS Amsterdam
API Applications Program Interface
BM Bare Metal
BMS Bare Metal Server
BRA Bratislava
BIOS Basic Input/Output System
BoD Bandwidth on Demand
CPU Central Processing Unit
CRM Core Resource Manager
CSF Central Server Facilities

DHCP	Dynamic Host Configuration Protocol
DSL	Domain Specific Language
ED	External Domain
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
Go5	Gang of Five
GTS	GÉANT Testbeds Service
GUI	Graphical User Interface
GW	Gateway
IAGW	Internet Access Gateway
IPv4	Version 4 of the Internet Protocol (StB IETF)
KVM	Kernel-based Virtual Machine
L1	Level 1
LJU	Ljubljana
LOPA	Local Only Provider Agent
MD	Multi-Domain
MDPA	Multi-Domain Provider Agent
NAS	Network-Attached Storage
NAT	Network Address Translation
NFS	Network File System
NOC	Network Operations Centre
NSI	Network Service Interface
OFX	OpenFlowFabric
OS	Operating System
PA	Provider Agent
PoP	Point of Presence
QSB	Quick Search Box
RCA	Resource Control Agent
RCM	Resource Control Methods
RDB	Resource Database
RIID	Resource Instance Identifier
RM	Resource Manager
SA	Service Activity
SDH	Synchronous Digital Hierarchy
STP	Service Termination Points
SSH	Secure Shell
SW	Software
TaaS	Testbeds as a Service
TACACS	Terminal Access Controller Access Control System
TCA	Testbed Control Agent
TTL	Time To Live
UA	User Agent

UAG	User Access Gateway
VC	Virtual Circuit
VLAN	Virtual Local Area Network
VM	Virtual Machine
VOX	Virtual OpenFlow Switch
VPLS	Virtual Private LAN Service
VPN	Virtual Private Network
VR	Virtual Router

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 bloc.
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 GTS Resource Manager for processing.
Unallocated	Marking for a resource instance that is no longer required.