TNOVA

NETWORK FUNCTIONS AS-A-SERVICE
OVER VIRTUALISED INFRASTRUCTURES

GRANT AGREEMENT NO. 619520

Deliverable D3.2

Infrastructure Resource Repository

Editor Michael J. McGrath (Intel)

Contributors Giuseppe Petralia, Vincenzo Riccobene (Intel), Jordi Ferrer Riera, Josep Batallé (i2CAT), José Bonnet, (PTIN), Marco Trubian (UMIMI), Francesco Liberati (CRAT), Marco Di Girolamo (HP), George Xilouris (NCSRD), Thomas Pliakas (CLDST)

Version 1.0

Date July 31st, 2015

Distribution PUBLIC (PU)

Executive Summary

The infrastructure repository is a key subsystem of the T-NOVA Orchestration layer which provides infrastructure related information collected from the VIM and NFVI components of the IVM layer. This subsystem is comprised of a number of key elements and capabilities including (i) data model; (ii) resource infrastructure repository; (iii) access mechanisms to the infrastructure repository; (iv) enhancement of the default information resources provided by cloud and SDN environments and (v) a resource discovery mechanism.

Analysis of the infrastructure information from the technologies selected to implement the VIM (namely OpenStack and OpenDaylight) revealed a significant deficit in available infrastructure information. An analysis of the various potential implementations was carried out and a candidate was selected for a prototype implementation. This prototype was used first to investigate if the necessary requirements could be supported and secondly to identify new requirements which had not previously been captured during the architectural design activities carried out work package WP2.

Using both the learnings from the prototype and the requirements, the final design of the infrastructure repository subsystem was developed. This design comprised of 5 key components. The first component is an enhanced platform awareness agent which runs on the compute nodes and collects platform specific information. This component was implemented as framework of libraries, commands and script to collect and aggregate a rich set of compute node information. The second component is a set of listener services. One listener is dedicated for EPA agent messages and a second one is dedicated to OpenStack related messages. The third component is the EPA controller which coordinates with listener services to process and persist updates to the repository database using data files received from the EPA agents or OpenStack infrastructure landscape change notifications. The fourth component is the infrastructure repository database which is responsible for storing the infrastructure related information and the relationships between the stored information. The database was implemented as a graph database in order to support the encoding of the relationships between the components of the NFVI. This approach also provided a convenient mapping of the system structures within the NFVI and the node structures of the graph database. The final component is a middleware API layer which provides a common OCCI compliant REST based interface to the Orchestrator components that need to retrieve information from the repository. The middleware layer also features a database to support the storage of NFVI PoP ingress and egress endpoints and associated parametric data for the links. The middleware implementation also provides support for multiple instances of the PoP level resources repository databases ensure appropriate scalability of the subsystem. All components have been successfully implemented and integrated to deliver a fully functional infrastructure repository subsystem.

Table of Contents

Figures

Tables

1. INTRODUCTION

The primary goal of Task 3.2 is to design and implement a resource discovery and repository subsystem as part of the T-NOVA Orchestration layer. The infrastructure repository subsystem is responsible for the collection and persistence of infrastructure data available from the infrastructure virtualisation and management (IVM) layer. The repository subsystem is also required to provide an interface to support requests from the Orchestration layer functional components (such as the resource mapping module) for infrastructure resource information as shown in Figure 1-1. The Orchestration layer utilises this information to reason over what collection of resource types need to be provisioned by the IVM for different types of VNFs within the T-NOVA system. The Orchestration layer then sends requests to the IVM to provision the required VM resources.

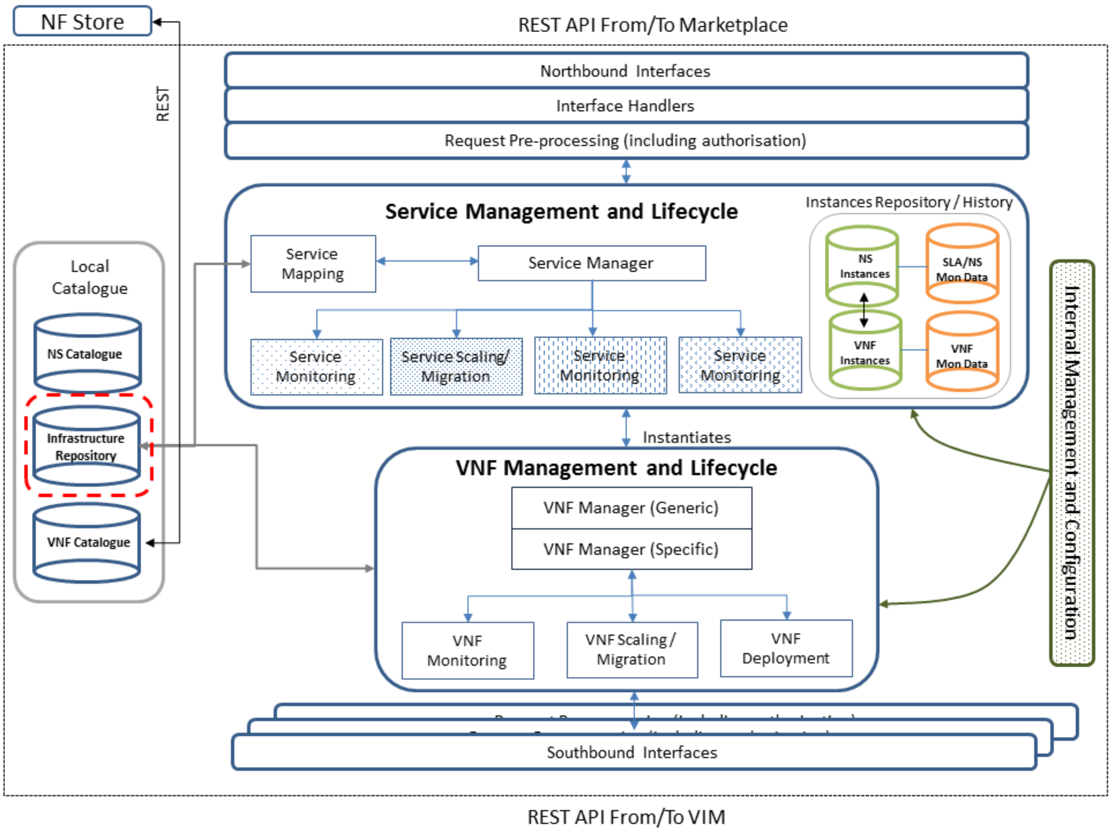


Figure 1-1 T-NOVA Orchestrator Architecture

The design of the repository subsystem addresses the challenges of assimilating infrastructure related information from sources within the IVM, namely the cloud infrastructure and data centre network environments. This subsystem comprises a number of key elements including a data model, resource information repositories, access mechanisms to the information repositories. The subsystem also augments the information provided by cloud and SDN environments through a resource discovery mechanism.

The IVM layer of the T-NOVA system comprises of a virtual infrastructure manager (VIM) and network function virtualisation infrastructure (NFVI). The design phase of the repository focused initially on identifying infrastructure related information available from the selected technologies to implement the VIM and NFVI. OpenStack and OpenDaylight were selected to implement the VIM while standard X86 high volume compute nodes and SDN enabled switches together with various support technologies were selected for the NFVI implementation.

An analysis of the service databases within OpenStack (Juno release) identified that only limited infrastructure related information was available. For example in the NOVA database the only CPU information available beyond manufacturer and speed was CPU status flags. Addressing these shortcomings was a key goal for the repository subsystem design and implementation.

Using learnings collected from the analysis of the information sources and the requirements identified in the architecture related deliverables in WP2 a number of approaches to the design of the subsystem were identified. The respective pros and cons of each implementation were evaluated. A candidate design was selected for a prototype implementation. This prototype was used to evaluate and refine the proposed architecture of the subsystem and its components in cooperation with other T-NOVA tasks such as T3.1, T3.3, T3.4, T4.1 and T4.4. This initial prototype implementation was presented in deliverable 3.1.

Learnings from the prototype implementation were used to collect additional requirements and to define the final architecture of the resource repository subsystem. The final subsystem design comprises of enhanced platform (EPA) agents running on the compute nodes in the NFVI which collect and report detailed platform information. When platform updates are available from the EPA agents, notification messages are sent to a controller via a specific listener service. Upon receipt of messages from the EPA listener service, data files sent by the EPA agents to a storage directory are processed by the controller and used to update the central repository databases. A listener service is also used to intercept and pipeline infrastructure related messages in OpenStack and to update the repository database via the controller. The infrastructure repository database is implemented as a graph database. The graph database provides a hierarchical relationship in the form of semantically relevant connections between the nodes stored as a link. An Open Cloud Compute Interface (OCCI) [1] compliant middleware layer provides a common interface to the resource information stored in the graph databases. Additionally the middleware API provides an abstracted single access point to physical network information available from OpenDaylight through its REST API's. The middleware layer also features a graph database which is used to store the endpoint information of OpenStack services for each NFV-PoP under the control of the T-NOVA Orchestrator. The database also supports storage of inter-PoP WAN connection information which can be inserted via a REST PUT API call.

The task also defined a number of sample visualisation use cases. The use cases focused on scenarios where the resource information stored in the repository subsystem could be used to support specified problems or operational needs.

2. Requirements Updates

Initial requirements for the T-NOVA Orchestrator and the IVM were previously documented in deliverable 2.31. As an initial step these requirements were analysed to identify the ones with a direct mapping to necessary infrastructure repository functionalities and features. Table 2.1 provides a listing of the relevant requirements identified and outlines how they are addressed in the design and implementation of the infrastructure repository subsystem.

Table 2.1 Infrastructure Repository Requirements

Requirement	Infrastructure Repository Support.
NFVO.20 Resources Inventory tracking	The repository provides specific fields for tracking the resource allocation, relying on existing fields in OpenStack API (referring to CPU, disks, RAM usage, etc.). Additionally, the repository provides tracking of resources currently not identified by OpenStack (e.g. GPUs, NICs, DPDK libraries etc.) via its EPA agents.
NFVO.17 Mapping Resources	The infrastructure repository stores all available infrastructure related information from the physical and virtualised resources within an NFVI-PoP and exposes this information through a unified set of middleware APIs to the resource mapping functional entity.
Or-Vi.04 Retrieve infrastructure usage data	Data related to dynamic infrastructure resource allocations to VM instances is stored in the repository e.g. number of vCPUs allocated etc.
Or-Vi.05 Retrieve infrastructure resources metadata	Infrastructure metadata is stored in the infrastructure repository for example CPU instruction sets, NIC support for DPDK etc.
VIM.1 Ability to handle heterogeneous physical resources	Heterogeneous infrastructure information e.g. PCIe co-processor card, GPU etc. is collected by EPA agents running on the physical compute nodes in the NFVI-PoP and stored in the infrastructure repository database.
VIM.4 Resource abstraction	The infrastructure data stored in the repository database is structured in a hierarchical graph based schema. This approach supports configurable abstraction of the resource details via the middleware API.
VIM.7 Translation of references between logical and physical resource identifiers	The infrastructure repository uses as resource tags: • hostname+type – physical resources. • OpenStack UUID for virtual resources • OpenFlow ID for physical network

	resources.
VIM.9 Control and Monitoring	Specific event related information such as Resize, Create etc. is stored for VMs.
VIM.20 Query API and Monitoring	Hypervisor related information e.g. number of available vCPUs, available disk size etc. is collected and stored in the infrastructure repository database.
VIM.23 Hardware Information Collection	Hardware information is collected by EPA agents running on the NFVI compute nodes and persisted to the infrastructure repository database. Physical network resource information is exposed through the API middleware layer via OpenDaylight's REST interface.
C.7 Compute Domain Metrics	Information regarding capacity of hardware resources e.g. disk size, RAM size etc. is persisted into the infrastructure repository database. Dynamic metrics such as CPU utilisation are out of scope for task 3.2 and are addressed specifically by task 4.4.
H.7 Platform Features Awareness/Exposure	Hardware-specific features are collected by EPA agents running on the NFVI compute nodes and persisted to the infrastructure repository database. Platform information is exposed via middleware APIs.

2.1. Additional Requirements

As the infrastructure repository is a core subsystem of the Orchestration layer its implementation has direct dependencies upon a number of WP3 and WP4 tasks, as previously described in deliverable 3-1 (see section 3.9). As outlined later in section 3.3 a prototype implementation of the infrastructure repository was developed to act as a 'concept car' and was used with the dependent tasks to determine if the existing requirements were appropriately fulfilled as well as to identify additional requirements. The new requirements identified are outlined in Table 2.2 and were incorporated into the final design phase of the infrastructure repository.

Table 2.2 Additional Infrastructure Repository Requirements

ID	Requirement	Infrastructure Repository Support
IR01	Resource information **SHALL** be stored using a standard unique identifier.	Resource information is stored in the subsystem database using the OpenStack's UUID.
IR02	The infrastructure repository **SHALL** be able to store information about the PoP Ingress and Egress endpoints.	The infrastructure repository provides a REST PUT call in the middleware API which supports the insertion of WAN connection information into the middleware database.

IR03	The infrastructure repository **SHALL** support more than one NFVI-PoP instance	An instance of the infrastructure repository runs at each NFVI-PoP. A single point of access to all instances of information repositories is provided via middleware layer to the Orchestrator.
IR04	The infrastructure repository **SHALL** provide a common interface to the Orchestrator functional entities	The infrastructure repository provides a single interface through an API middleware layer for consumers of the information stored in the repository.
IR05	The infrastructure repository **SHALL** minimise the overhead it places on its data sources.	A listener service was developed which intercepts messages from the OpenStack notification.info queue and updates the resource repository via the EPA Controller with changes in the resource landscape relating to NOVA, Neutron and Cinder.
IR06	The infrastructure repository **SHALL** provide an interface which abstracts its implementation	The middleware layer API's implementation is based on an OCCI [2] compliant specification which provides full abstraction of the infrastructure repository implementation.
IR07	The infrastructure repository **SHALL** use a common authentication mechanism for all API calls	All calls to the middleware layer APIs are authenticated using the T-Nova Identity/Authorisation micro-service (GateKeeper)
IR08	The infrastructure repository **SHALL** store the relationships between resources	The infrastructure repository database is implemented as a graph database which is used to store the relationships between resources in a hierarchical manner.

A total of 20 specific requirements were considered in the architectural design of the infrastructure repository. The requirements listed in Tables 2-1 and 2-2 were also used to evaluate the final repository implementation to ensure the available functionalities and capabilities fully satisfied the identified requirements.

3. Infrastructure Repository Design

An iterative and incremental process was adopted in the design and development of the repository subsystem [3]. This approach can be described as a combination of both an iterative design method and an incremental build model for application software development. The development lifecycle was composed of several iterations in sequence. The initial iterations of the infrastructure repository were previously described in Section 3.7 of deliverable 3-1. The respective pros and cons of each identified design option were also presented. A key influence in the design of the repository subsystems was the targeted ability to provide Enhanced Platform Awareness (EPA) style information. Today's compute platforms with their rapidly evolving technologies embedded in processors and chipsets, integrated on server boards, and installed in PCIe slots, offer a rich set of capabilities which provide significant performance benefits to specific workload types if appropriately utilised. However cloud environment such as OpenStack, have not being taking full advantages of these enhancements. This is a particular acute problem for NFV type workloads whose performance can be significantly influenced by platform technology features. Therefore offering EPA type information as part of the infrastructure repository subsystem was an important design goal.

In this section a brief summary of the initial iterations is presented together with the final design of the repository subsystem. The key learnings from these early iterations were used to inform the final design implementation of the repository subsystem which is described in section 3.3. The final design comprises five functional entities, namely: EPA agents, infrastructure repository database, listener services (EPA and OpenStack), EPA Controller, middleware layer API's and database.

3.1. Overview of Infrastructure Data Sources

The main sources of infrastructure information based on the software platforms selected to implement the functional entities of the IVM layer (namely the VIM and NFVI-PoP) are OpenStack and OpenDaylight. The Kilo release of OpenStack and Helium release of OpenDaylight were selected as the base platform releases for the implementation of the T-NOVA VIM. From a hardware perspective standard X86 high volume servers from Hewlett Packard (HP) were selected during the design and testing phases of the repository subsystem.

As OpenStack is a modular platform, each module has its own database to manage the resources and information relevant to functions of that module. In the context of the T-NOVA infrastructure repository, the databases of primary interest are the Nova and Neutron DB's. Detailed information on the databases can be found in Section 3.4 of deliverable 3-1.

3.1.1. NOVA Database

OpenStack NOVA database can be implemented using any SQL Alchemy-compatible database. For T-NOVA the default MySQL implementation is used. The nova-conductor service is the only service that writes to the database. The other NOVA compute services access the database through the nova-conductor service. The NOVA database is relative complex, containing in excess of 100 tables. These tables were examined to identify which ones contained infrastructure data that was potentially useful to Orchestration layer related operations such as resource mapping. The *compute_nodes* table contains the most useful physical hosts information including information on the hypervisor, the number of virtual CPUs, available/used main memory (RAM), available/used disk space, CPU details (such as vendor, model, architecture, CPU flags, the number of cores, etc.).

Information on virtual machine instances is stored in the *instances* table. An instance dataset can include fixed IPs, floating IPs, volumes, virtual interfaces that provide network access, an instance type, and an image.

3.1.2. Neutron Database

Neutron is the OpenStack component that enables network virtualisation and provides "Networking as a Service". The service is based on a model of virtual networks, subnets and port abstractions to describe the networking resources. The primary tables of interest are *ports*, *routers*, *networks*, *subnets* and *ml2_port_bindings*. A subnet is a block of IP addresses that can be assigned to the VMs. A port is a virtual switch connection point. Each VM can attach its virtual Network Interface Controller (vNIC) to a network through a port. A port has a fixed IP address taken from the address subset of the related subnet. Routers are local entities that work at Layer-3, enabling packets routing between subnets, packets forwarding from internal to external networking, providing Network Address Translation (NAT) services and providing access instances from external networks through floating IPs.

3.1.3. OpenDaylight

OpenDaylight provides a set of base functions which are supported through a set of managers and components. The relevant ones from a network infrastructure perspective are [4]:

- **Topology Manager** – responsible for storing and handling the interconnection configuration of managed network devices. It creates the root node in the topology operational subtree during controller start-up and actively listens for notifications that require necessary updates to the subtree, including all discovered switches and their interconnections.
- **Switch Manager** – provides network nodes (switches) and node connectors (switch ports) details.
- **Inventory Manager** – Maintains the concurrency of the inventory database by querying and updating switch and port information managed by OpenDaylight.

The information stored by these managers is exposed via REST interfaces. The REST interfaces of interest from an infrastructure perspective are:

- *OpenFlow Nodes*: extends the top-level inventory node with OpenFlow (OF) - specific features that allow retrieving and programming of OF-specific state, such as ports, tables, flows, etc.
- *Base Topology* - list of all topologies known to the controller

3.2. Enhanced Platform Awareness

The purpose of EPA is to detect platform capabilities through the discovery, tracking, and reporting of enhanced features in the CPU and PCIe slots [5]. OpenStack Juno and the recent Kilo release offer some EPA platform information, for example PCIe aware NUMA pinning. Future releases of OpenStack will further increase the richness of the EPA data available. However within the context of the T-NOVA project timelines current EPA support within OpenStack was considered to be insufficient; therefore in the design of the repository it was necessary to consider appropriate functionality ensuring the availability rich platform information from the subsystem database. It is also important to note that EPA extends beyond simply capturing platform information. In order to use this information in a cloud environment such as OpenStack, filtering and matching of available platforms with the specific capabilities to an instance type requesting the desired features needs to be considered. Finally scheduling and installing the instance onto the selected platform with the enabled features is required. While these latter two requirements are out of scope for this task they will be considered by other T-NOVA tasks such as 4.5 and 7.1.

3.3. Repository Prototype Design

The initial prototype design focused on the use of existing OpenStack and OpenDaylight APIs to expose NFVI-PoP infrastructural information to the T-NOVA Orchestration layer. This approach provided advantages as the API’s available are standardised and concurrent information is always available. However there are a number of key disadvantages to this approach. OpenStack provides over one hundred REST APIs which increases the potential complexity of Orchestration interactions, for example multiple APIs could be required to retrieve an information set of interest. Also the infrastructure specific information available from the services databases is limited in nature.

Three potential designs where initially identified each with respective pros and cons (see section 3.3 deliverable 3-1). The prototype design is shown in Figure 3.1. This design addressed the issues relating to lack of platform information, i.e. enhanced platform awareness by utilising agents. These agents running on the NFVI compute nodes collect detailed platform information and persist the information to a central database. Information stored in the database is exposed via a REST API to consuming components.

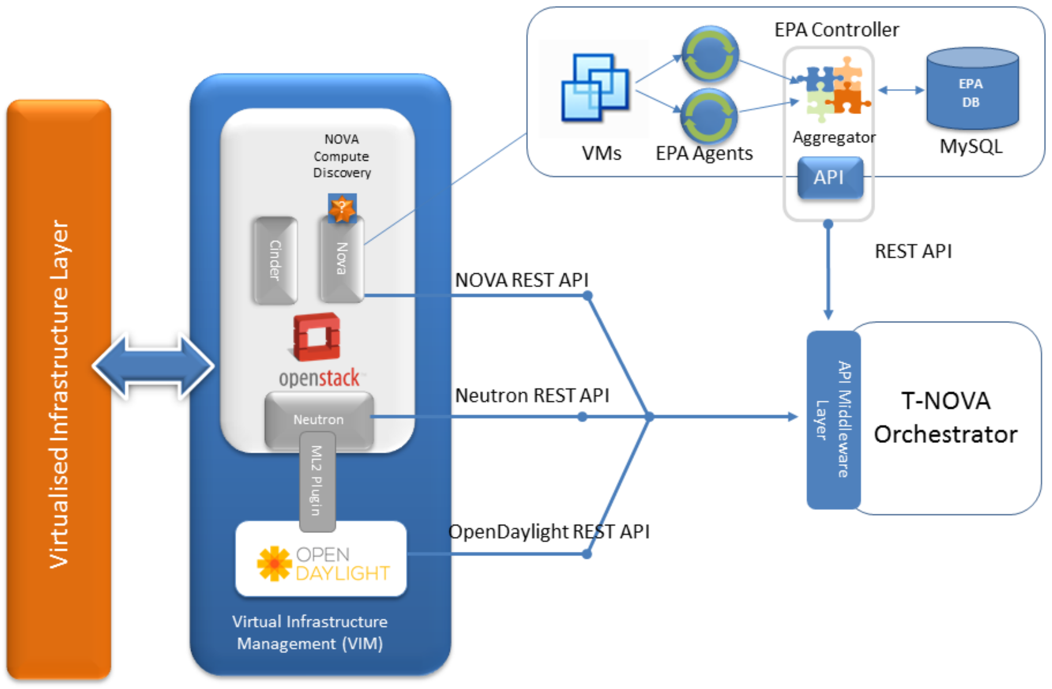
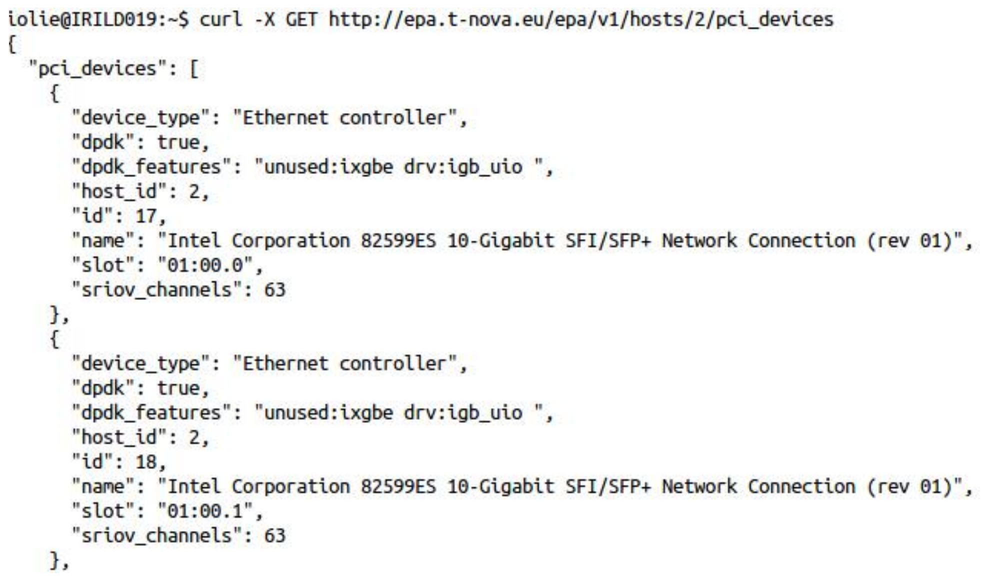


Figure 3-1 Prototype infrastructure repository architecture

The REST API design of the repository prototype was based on the same structure as the existing OpenStack API, and from a user perspective they appear as a simple extension of them. An example of a REST API call is shown in Figure 3-2, which returns a list of the PCIe devices available from a specified host. The call takes the form of:

```
GET /epa/v1/hosts/[host_id]/pci_devices
```



```
iolie@IRILD019:~$ curl -X GET http://epa.t-nova.eu/epa/v1/hosts/2/pci_devices
{
  "pci_devices": [
    {
      "device_type": "Ethernet controller",
      "dpdk": true,
      "dpdk_features": "unused:ixgbe drv:igb_uio ",
      "host_id": 2,
      "id": 17,
      "name": "Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01)",
      "slot": "01:00.0",
      "sriov_channels": 63
    },
    {
      "device_type": "Ethernet controller",
      "dpdk": true,
      "dpdk_features": "unused:ixgbe drv:igb_uio ",
      "host_id": 2,
      "id": 18,
      "name": "Intel Corporation 82599ES 10-Gigabit SFI/SFP+ Network Connection (rev 01)",
      "slot": "01:00.1",
      "sriov_channels": 63
    },
```

Figure 3-2 Sample of compute node PCIe devices

This prototype was implemented to be fully functional and was used as a 'concept vehicle' to elicit feedback and to identify new requirements for the tasks dependent on the repository subsystem. For example requirements IR03 – IR05 were identified as result of this process. Analysis of the implementation revealed that the implementation did not adequately support requirements IR02 and IR03. In a multi-PoP scenario complex and ineffective interactions between the T-NOVA Orchestrator and the various OpenStack and OpenDaylight instances would be required; furthermore, there was no provision for storing the ingress and egress network endpoints of the PoP's (i.e. WAN connections).

The prototype implementation supported performance evaluation of the implemented functional requirements, and also helped to identify limitations in available functionality.

The key learnings and the additional requirements identified in cooperation with the dependent tasks on the infrastructure repository were used to inform the final design of the subsystem which is outlined in the following section.

3.4. Final Resource Repository Subsystem Architecture

The key inputs into the final design of infrastructure repository were the prototype implementation as outlined above and in deliverable 3-01, the requirements defined in the WP2 deliverables and the new requirements (IR01 – IR08) identified during the prototyping phase. The final architecture design of the infrastructure repository subsystem is shown in Figure 3-3. The functional components of the architecture are as follows:

- **EPA Agents** – Python based software agent running on the compute nodes of the NFVI-PoP. A central EPA controller service provides aggregation of data from each agent and persists the data to a central database.
- **Infrastructure Repository Database** – Collected infrastructure data is stored in a graph database where resources are represented as nodes with associated properties. Edges between the nodes store information on the relationship between nodes.
- **Listener Services**- Two separate listener services are specified within the architecture. The OpenStack Notification listener service is designed to intercept messages from the OpenStack notification.info queue and to provide notifications to the controller. The EPA agent listener service intercepts EPA agent messages and notifies the controller of the messages in order to trigger processing of received data files and to use the data to carry out an update database action.
- **EPA Controller –** The controller is responsible for updating the infrastructure database based on information received from the listener services and data files sent by the EPA agents. One instance of the controller runs in each NVFI-PoP. The service runs on a compute node within the NFVI.
- **API Middleware Layer** – Provides a common set of API calls that can be used by all the T-NOVA Orchestration layer functional entities.

The key changes from the initial prototype implementation were as follows:

- The infrastructure repository database is implemented using a graph database. The prototype implementation used a MySQL relational database. The rational for this significant design change was to support encoding of the relationship between resources (see requirement IR08) and to arrange resources into logical layers which represented the stack of physical, virtual resources and workloads.

- Multi NFVI-PoPs are supported with a repository instance per PoP. All repository instances are accessible through a common middleware API (see requirement IR03).
- The API middleware layer provides a common interface for all T-NOVA Orchestrator functional entities (see requirement IR04).
- The API middleware layer database provides support for storing information on the PoP ingress and egress endpoints within the T-NOVA system (see requirement IR02).
- The API middleware layer is designed to be OCCI compliant, which provides abstraction from the underlying implementation (see requirement IR06)
- An OpenStack listener service provides interception of infrastructure related messages and updates to the infrastructure repository database (see requirement IR05). This approach reduces the overhead on the OpenStack service databases by eliminating the need to poll all the databases on a recurring basis.
- The API middleware layer uses the T-NOVA Gatekeeper service for authentication of API calls (see requirement IR07).

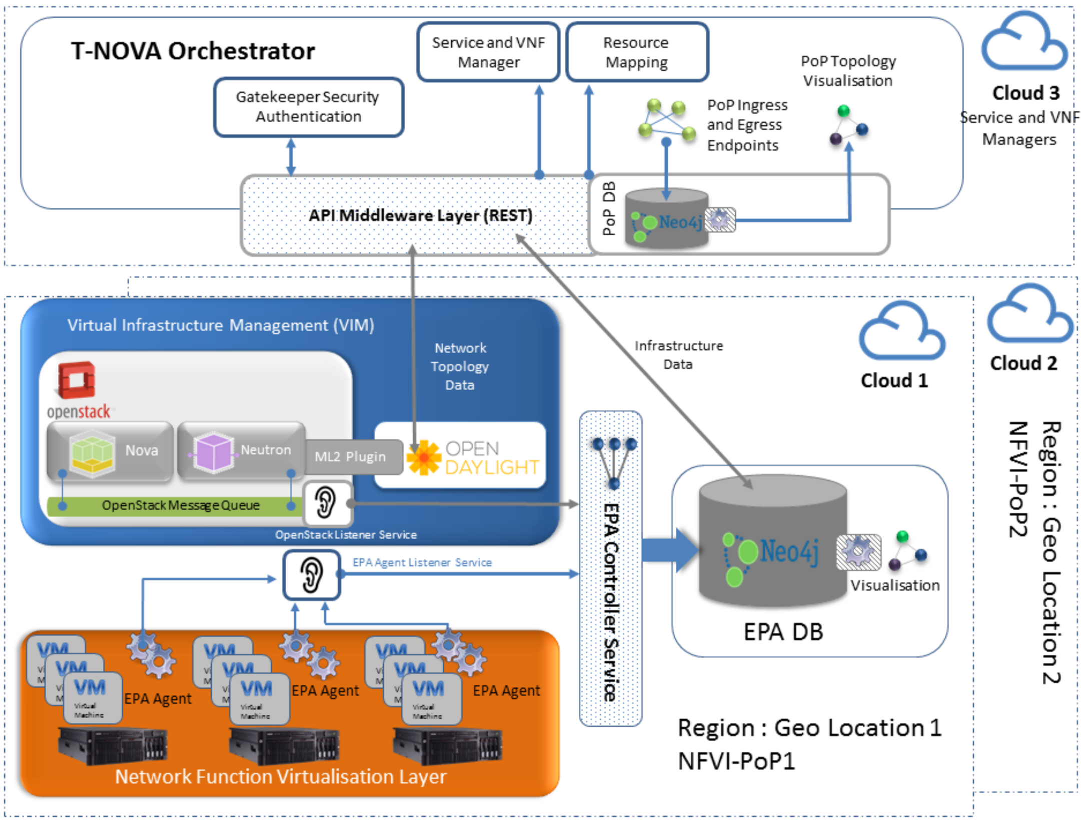


Figure 3-3 Infrastructure Repository Sub System Architecture

3.4.1.1. Enhanced Platform Awareness Agent Design

As previously outlined in deliverable 3-1, analysis of the NOVA database revealed that limited compute platform information was stored in the Juno OpenStack release. For example a significant gap was the absence of information relating to the available PCIe devices such as network cards for a given compute platform. In order to improve the richness of the available platform information beyond what was accessible in the NOVA database; direct interrogation of the NFVI resources was required in order to fulfil requirements VIM23 and H7.

Two important considerations in the design of the component were scalability and support for enhanced platform awareness. In order to support scalability, a per compute node agent based approach was selected as shown in Figure 3-4. Secondly the agent was designed to provide complete platform information at installation time. Thirdly a software agent approach was selected to meet some key design goals. These goals were as follows:

- Runs and collects host information upon host start-up.
- Does not require interaction with a user.
- Invokes necessary support tasks such as communication functions.

For an operational perspective the design of the agent includes the following assumptions:

- An agent runs each time the node is started or rebooted.
- The EPA agent image is included in the VM images used by the T-NOVA VIM.
- The configuration of NFVI nodes is considered to be stable. As new nodes are added, an EPA runs at start-up capturing the platform details of the new node. Hardware upgrades are considered to require a compute reboot which will allow the agent to capture hardware upgrades or changes.

Another design consideration was execution flexibility. For the purposes of the current T-NOVA implementation, agent execution is bounded to the compute node boot cycle. However in environments where low frequency updates are insufficient, the agents can be configured to run on scheduled basis as a system cron job. In this way the concurrency of the compute node information can be maintained for highly dynamic environments.

In order to address EPA needs, the information collected from each compute node in the NFVI PoP should have high granularity. Key types of required information included NUMA nodes, PCIe devices such as network cards and coprocessors etc. The design of the agent therefore needs to enable the capture of both detailed platform information and the hierarchical relationship between the components. Additionally the design of the agents needs to accommodate both the identification of resources and in some cases the configuration details of a device, e.g. the number of SR-IOV channels allocated on a NIC and the number of free channels available. Therefore implementing a single mechanism to retrieve all the information of interest is challenging. As a result, a framework based approach where utilities, scripts and commands are used collectively under the control umbrella of an agent to retrieve the necessary information was adopted.

All information generated by the agents should be persisted to a file format which can be sent to a common aggregation service in an asynchronous manner. Design details of the aggregation service are outlined in section 3.4.1.3.

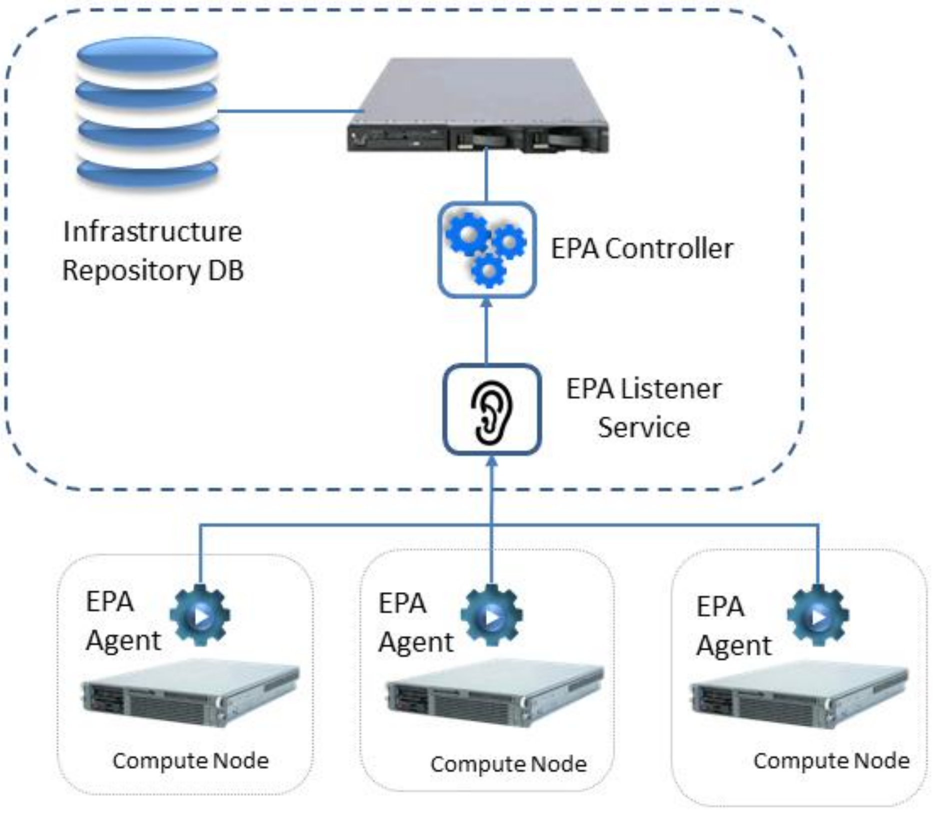


Figure 3-4 High Level EPA Agent Architecture

3.4.1.2. Infrastructure Repository Database Design

The previous prototype implementation used a standard MySQL relational database for the storage of resource information. However in order to encode the relationship between the resources and associated parameters (see IR08) in an efficient manner a graph database approach was adopted for the final database design. Graph databases are NoSQL (Not only SQL) database systems which commonly use a directed acyclic graph (DAG) to store data relationships. SQL based databases store and retrieve information stored in tabular relationships while graph databases use a graph data model for storage and processing of data. Graph databases bring application specific advantages such as simpler design and horizontal scaling. NoSQL databases are finding popularity for big data and real-time web applications. From the perspective of the infrastructure repository database design the rational for the use of a graph database approach is that it allows you to find interconnected data much faster and in a more scalable manner that in a relational data model [6]. For example traversal type queries, which would be commonly used for identifying specific resource types in server nodes, run up to 10 times faster with a graph database in comparison to SQL [7].

A graph database stores information using vertices/nodes and edges/relations. A graph structure supports:

- Representing data in a natural way, without some of the distortions of the relational data model
- Apply various types of graph algorithms on these structures. This will help service mapping algorithm, providing data in a way that is already optimised for its computation. The graph structure permits the navigation of nodes following explicit pointers that connect the nodes and to identify the paths between nodes.

From a database design perspective, using a graph database follows a different design approach in comparison to relational database designs. The initial focus is on identifying the nodes within the graph. The NFVI can be decomposed into either physical or virtual resources which can be mapped directly to a node structure. The relationships of virtual-virtual, physical-physical and virtual-physical are captured in the relevant connections between the resources. Virtual resources have an implicit dependency on physical resources, i.e. a virtual resource cannot exist without a physical host. Therefore in a graph construct, virtual resources must at some point in the graph be connected to a physical resource.

The use of graphs also maps conveniently to the hierarchical structure of compute, storage and network elements within the NFVI. Two approaches can be adopted in design of the graphs, namely a top down or a bottom up approach. A top down approach was adopted given that a server is the key autonomous unit within the NFVI. A server can then be broken down into its constituent components, while maintaining the relationship among components using directed acyclic graphs. For example in Figure 3-5 a simple graph for a server is shown, where the server has two sockets, each socket has a CPU and each CPU has multiple cores.

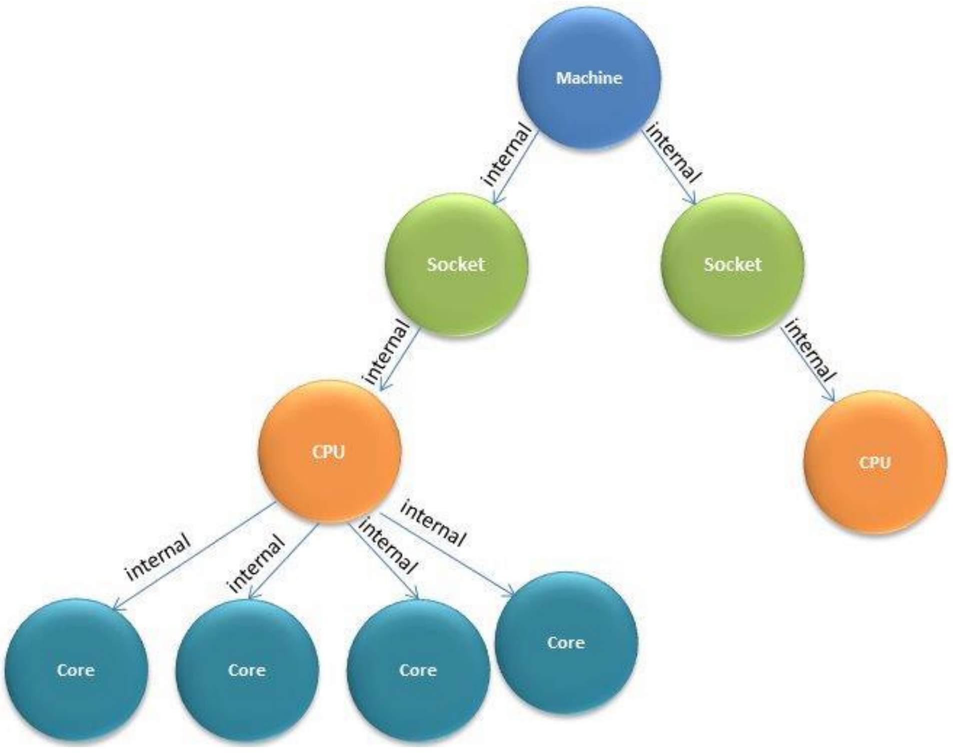


Figure 3-5 Simple server resource graph.

A similar process is adopted for virtual resources which have a similar hierarchical construct.

As shown in Figure 3-5, nodes in a graph database have explicit relationships between themselves. The relationship indicates the directed, semantically relevant connections, e.g. "has a" "runs on" "on network" etc. between node-entities. A relationship comprises of a direction (indicated by the direction of the arrow), a type, a start node, and an end node.

3.4.1.3. Listener Services Design

Listener services are processes that receive or intercept specific messages of interest and carry out some predefined action on the message, such as forwarding the message to another service or location. In order to ensure flexibility, the design of a listener service needs to utilise a configuration file in order to adapt the behaviour of the service to evolving system designs and upgrades. In the design of the repository subsystem the need for two listener services was identified. The listener services required in the design are as follows:

- EPA Agent listener service
- OpenStack Message Queue listener service

The function of the EPA agent listener service is to receive messages with platform information updates from the EPA agents running on the compute nodes. One listener service is required per NFVI. All EPA agents running in the NFVI need to be able to communicate with the listener service.

The OpenStack listener service is designed to intercept messages from the OpenStack notification.info queue and to notify the EPA Controller that a change in the infrastructure landscape of the NFVI has occurred which must be reflected in the infrastructure repository database. The design of the listener service plays a central role in maintaining the concurrency of the information stored in the resource repository data. It is designed in a flexible and scalable manner by implementing handlers for each specific category of message (e.g. compute.*, volume.* etc.) relating to resource updates such as the creation of a new VM etc.

3.4.1.4. EPA Controller Design

The goal of the controller component is to provide a centralised actuation point for listener service notifications as shown in Figure 3-6. A controller resides on each PoP in the T-NOVA system.

The main design goals of the controller are:

- Persistence and consistency of infrastructure information
- Requires no user interaction
- Provides asynchronous response to listener notifications to support scalability
- Processes data contained in different file formats which are published by the EPA agents.

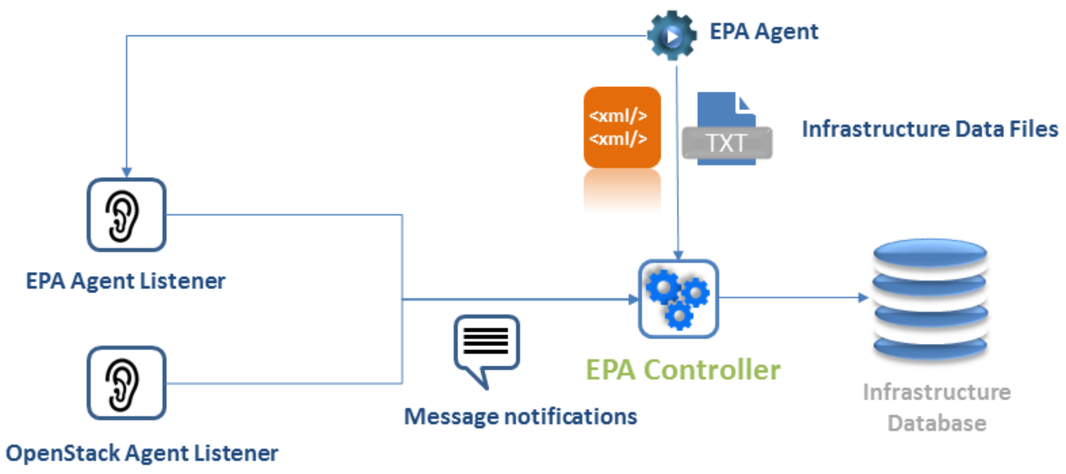


Figure 3-6 Controller system overview

To ensure persistence and consistency, the controller is responsible for managing the connection to the infrastructure repository database. The controller is also responsible for initialising the infrastructure repository database at start-up before starting the listener services so that any update to the database committed by the listener services will be consistent with current state of the infrastructure landscape. The controller also has responsibility for processing files containing infrastructure information received from the EPA agents with varying formats. After starting the Controller, it has responsibility for orchestrating in an autonomous manner its components without the need for user interaction.

3.4.1.5. Middleware API Layer Design

The middleware layer is the infrastructure repository subsystem component that provides the Northbound REST API to the functional components of the T-NOVA Orchestration layer such as the Orchestrator manager, resource mapping module etc. Its design is primarily driven by analysis of the requirements outlined in section 2.

The middleware layer needs to provide a common interface to all the PoP level databases within the T-NOVA system as shown in Figure 3-7. From the perspective of a component using the interface the location of the data and the underlying complexity in forming the query response is abstracted as per requirement VIM4 (see Table 2-1). In order to support common access to all PoPs, the relevant service endpoints need to be stored within the middleware layer. Secondly the middleware layer needs to store information regarding the network ingress and egress endpoints of the PoPs comprising the T-NOVA system and parametric data relating to the links e.g. available bandwidth available. Therefore the inclusion of a database was considered a necessary element in the design. Additional API calls which support the creation of new PoP entries or updating existing entries are required as per requirement IR02.

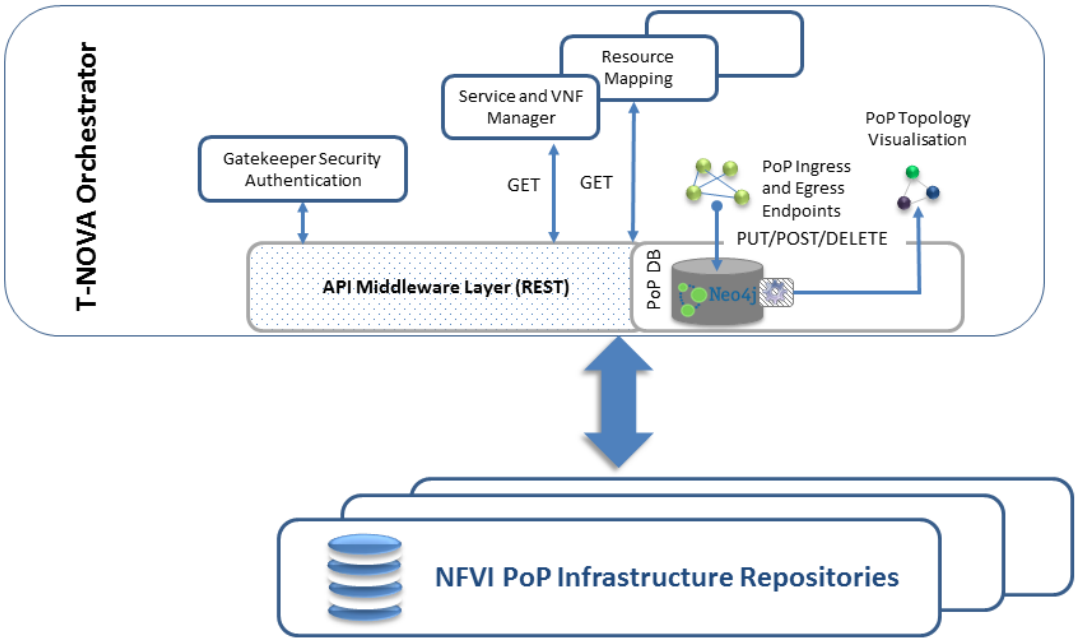

T-NOVA Orchestrator
Resource Mapping
Service and VNF Manager
Gatekeeper Security Authentication
GET
GET
PoP Topology Visualisation
PoP Ingress and Egress Endpoints
PUT/POST/DELETE
API Middleware Layer (REST)
PoP DB
Neo4j
NFVI PoP Infrastructure Repositories

Figure 3-7 Middleware layer design

The primary function of the middleware APIs is to support retrieval of information from the repository databases located at each NFVI-PoP. The interface does not support other actions on the PoP level resource repository databases such as inserting, updating or deleting information in the NFVI-PoP level databases. To be compliant with the design decisions of Task 3-1 a REST type approach to the design of the interfaces was required. However additional requirements in the interface design were also considered. The middleware API also provides an agnostic repository implementation interface to the dependent Orchestrator components as per requirement IR06. The design of the interfaces therefore considered approaches such as OCCI to fulfil this requirement.

Another requirement that the design considered was exposing hardware capabilities collected at PoP level by the agents as per requirement H.7 (see table 2-1). This requirement necessitates the discovery of the features and functionality provided by resources (compute, accelerators, storage and networking) and exposing this information to the Orchestration layer.

Finally the middleware design needed to support the common service authentication mechanism (Gatekeeper) used by the T-NOVA Orchestration layer. All API calls received by the middleware layer must be authenticated before execution.

4. INFRASTRUCTURE REPOSITORY SUB-SYSTEM IMPLEMENTATION

This section describes the implementation details of the infrastructure repository sub-system and its functional components. The key interactions between the EPA Agent, listener services (OpenStack and EPA Agent), EPA Controller, repository database and API middleware layer components are shown in Figure 4-1. The key information flows between components and the other T-NOVA Orchestrator subsystems such as the mapping service are also shown. The components represented as grey blocks are open source software components used to build the VIM at each T-NOVA NFVI-PoP. The controller is designed to use these components for retrieving virtual resources information. The repository sub-system components are presented as blue blocks. EPA agents running on each compute node of the NFVI reports hardware data to the controller via a RabbitMQ broker[1]. The controller, which is subscribed to messages of interest including those from the EPA agents, intercepts the messages via the dedicated listener services and uses the messages to trigger updates to the repository database via the EPA Controller. The dependent Orchestrator subsystem components interact with the repository via the middleware API layer. In particular the T-NOVA Orchestration layer retrieves infrastructure information to support both deployment and management decisions with respect to either new or existing network services. The T-NOVA Orchestration layer can also use the middleware APIs to support storage of NFVI-PoP ingress and egress endpoint information via the middleware layer graph database. Specific API calls are provided to support all required actions. New PoPs with their WAN link information can be added; existing PoP information relating to connection attributes can be either updated or deleted. A description of connections between the infrastructure sub-system components is provided in Table 4-1.

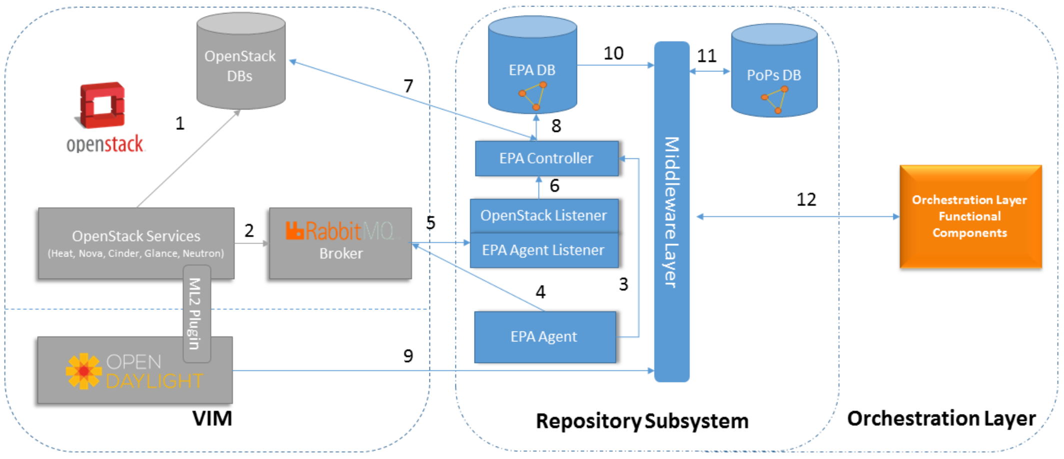


Figure 4-1 Infrastructure Repository – Key information flows

[1] https://www.rabbitmq.com/

Table 4.1 EPA Data Flows

#	Description
1	OpenStack services store information in their respective MySQL DBs
2	OpenStack services send notification messages to the RabbitMQ broker containing virtual resources updates
3	EPA Agent sends hardware features data to the Controller
4	EPA Agent sends a notification to the RabbitMQ broker when new hardware data is collected.
5	Listener Services intercept message of interest from EPA agents or OpenStack updates
6	EPA Controller consumes notifications from listener services to trigger resource repository DB updates
7	EPA Controller retrieves virtual resources information by querying OpenStack MySQL services DBs
8	EPA Controller persists infrastructure information to the resource repository DB
9	Middleware retrieves network topology information from OpenDaylight Controller.
10	Middleware retrieves infrastructure information from the EPA DB based on the API call used
11	Middleware stores and retrieve PoPs information using the PoPs DB
12	T-NOVA Orchestrator retrieves infrastructure information and stores PoP and connections information using the Middleware API

Each component is implemented as a Python module and configured using standalone configuration files. A detailed description of each component is provided in the following sub sections.

4.1. Enhanced Platform Awareness Agent Implementation

An EPA agent runs on each compute node within the NFVI-PoP. The agent is responsible for collecting information relating to the hardware features of the physical compute node hosts, and sending that information to the EPA Controller which persists the received information to the repository database. The majority of the compute node information is collected using the open source *hardware locality* software package [8]. This package provides an abstraction of the hierarchical topology of a compute node's architecture. It gathers various system attributes like cache and memory information, as well as information regarding I/O devices such as network interfaces, GPUs etc. (see Figure 4-2). The tool supports most of the modern operating systems ensuring good interoperability.

```
 <topology>
  <object type="Machine" os_index="0">
    <info name="DMIProductName" value="ProLiant DL380 Gen9"/>
    <info name="DMIProductVersion" value=""/>
    <info name="DMIChassisVendor" value="HP"/>
    <info name="DMIChassisType" value="23"/>
    <info name="DMISysVendor" value="HP"/>
    <info name="Backend" value="Linux"/>
    <info name="OSName" value="Linux"/>
    <info name="OSRelease" value="3.13.0-44-generic"/>
    <info name="OSVersion" value="#73-Ubuntu SMP Tue Dec 16 00:22:43 UTC 2014"/>
    <info name="Architecture" value="x86_64"/>
    <distances nbobjs="2" relative_depth="1" latency_base="10.000000">
      <latency value="1.000000"/>
      <latency value="2.100000"/>
      <latency value="2.100000"/>
      <latency value="1.000000"/>
    </distances>
    <object type="NUMANode" local_memory="33609957376">
      <page_type size="4096" count="8205556"/>
      <page_type size="2097152" count="0"/>
      <object type="Socket">
        <info name="CPUModel" value="Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz"/>
      </object>
      <object type="Bridge" os_index="0" bridge_type="0-1" depth="0"
bridge_pci="0000:[00-13]">
          <info name="PCIVendor" value="Intel Corporation"/>
          <info name="PCIDevice" value="Wellsburg PCI Express Root Port #5"/>
          <object type="PCIDev" name = "Broadcom Corporation NetXtreme BCM5719
Gigabit Ethernet PCIe" pci_busid="0000:02:00.0">
            <info name="PCIVendor" value="Broadcom Corporation"/>
            <info name="PCIDevice" value="NetXtreme BCM5719 Gigabit Ethernet PCIe"/>
            <object type="OSDev" name="em1" osdev_type="2">
              <info name="Address" value="c4:34:6b:b8:52:d0"/>
            </object>
            ...
      </object>
    </object>
   ...
  </object>
</topology>
```

Figure 4-2 Hardware locality data extract

Additional cpu specific information is collected (for Linux machines only) using the output of the command:

```
cat /proc/cpuinfo
```

The output generated and collected by the EPA agent is shown in Figure 4-3.

```
processor	: 41
vendor_id	: GenuineIntel
cpu family	: 6
model		: 63
model name	: Intel(R) Xeon(R) CPU E5-2697 v3 @ 2.60GHz
stepping	: 2
microcode	: 0x2b
cpu MHz		: 1200.976
cache size	: 35840 KB
physical id	: 0
siblings : 28
core id : 14
cpu cores	: 14
apicid  : 29
initial apicid	: 29
fpu		: yes
fpu_exception : yes
cpuid level	: 15
wp		: yes
flags		: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp
lm constant_tsc arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc
aperfmperf eagerfpu pni pclmulqdq dtes64 monitor ds_cpl vmx smx est tm2 ssse3
fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt
tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm abm ida arat epb xsaveopt
pln pts dtherm tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 avx2
smep bmi2 erms invpcid
bogomips	: 5194.05
clflush size	: 64
cache_alignment	: 64
address sizes	: 46 bits physical, 48 bits virtual
power management:
```

Figure 4-3 Output of CPUInfo command

The agent is also designed to detect the presence of DPDK compatible NICs using a script from the DPDK library (dpdk_nic_bind.py). The output of this script is shown in Figure 4-4.

```
PCI addr
0000:05:00.1
0000:05:00.0
```

Figure 4-4 Output of DPDK script

The agent also identifies the presence of SR-IOV capable NICs and the number of allocated and unallocated virtual functions[2] for each SR-IOV NIC as shown in Figure 4-5. The NIC in this example is dual channel NIC which can support up to 63 virtual functions or channels. However one channel has been configured to support only 25 virtual functions while the second channel is configured to support no virtual functions.

```
PCI addr      numvfs   totvfs
0000:05:00.1    25      63
0000:05:00.0    0      63
```

Figure 4-5 Output of SR-IOV script showing total number of SR-IOV NIC virtual functions

SR-IOV capabilities are discovered using a custom Python script. First the script parses the output of the list PCI devices command (lspci) and extracts (if any) the pci address of each SR-IOV device. Then the script extracts additional information about the SR-IOV card (number of available/allocated SR-IOV virtual functions) using the following two commands:

```
cat /sys/bus/pci/devices/0000\: + pci_address + /sriov_numvfs
cat /sys/bus/pci/devices/0000\: + pci_address + /sriov_totalvfs
```

Information collected by hwloc [9] is written to an XML file while information from cpuinfo, dpdk_nic_bind.py and the SR-IOV script are written to text files. All files are sent to the EPA controller using a secure shell (ssh) connection. When a file is sent to the controller the agent also sends a message to the controller using RabbitMQ to notify the controller via the EPA listener service a new file has been sent, so the controller can parse the file and update the infrastructure repository database. Notifications are sent to the RabbitMQ broker of the VIM's OpenStack instance using Pika[3]. Pika is a Python implementation of the AMQP 0-9-1 protocol. The execution life cycle of the agent is shown in Figure 4-6.

[2] Virtual Functions (VFs) are simple PCIe functions that contain all the resources necessary for I/O but have minimal set of configuration resources.

[3] https://pika.readthedocs.org/en/0.9.14/

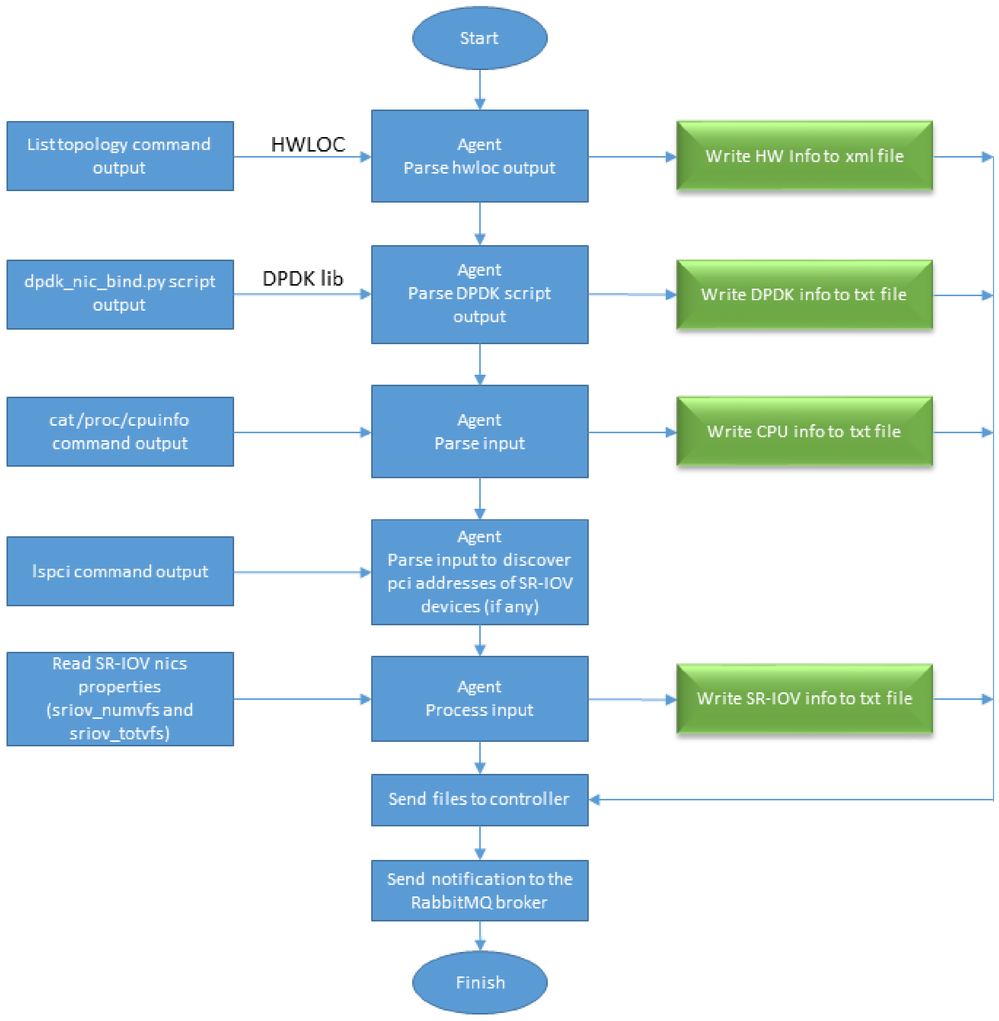

Start
List topology command output
HWLOC
Agent Parse hwloc output
Write HW Info to xml file
dpdk_nic_bind.py script output
DPDK lib
Agent Parse DPDK script output
Write DPDK info to txt file
cat /proc/cpuinfo command output
Agent Parse input
Write CPU info to txt file
lspci command output
Agent Parse input to discover pci addresses of SR-IOV devices (if any)
Read SR-IOV nics properties (sriov_numvfs and sriov_totvfs)
Agent Process input
Write SR-IOV info to txt file
Send files to controller
Send notification to the RabbitMQ broker
Finish

Figure 4-6 EPA Agent execution flow diagram

When the controller receives a message notification via the EPA listener service from an EPA agent, it parses the received files and updates the information stored in the infrastructure repository database.

4.2. Infrastructure Repository Database Implementation

The repository database is implemented as a graph database using Neo4j[4]. The 2.1.7 community version of the database was used for implementation purposes. Neo4j is an ACID-compliant database (**Atomicity**, **Consistency**, **Isolation**, and **Durability**). All changes in the database must be performed in a transaction, which checks for data validity before storing. For example it will check relationship consistency ensuring that the specified start and end nodes exist. The isolation property ensures that parallel transactions do not influence each other. The query language used to retrieve data from the database is called Cypher. It is a declarative graph query language that allows for expressive querying of the graph database.

Let's assume that in the repository database information relating to an OpenStack VM stack is stored. The stack has a UUID=7fe39371-1379-4162-9deb-e904c4f2dc43,

[4] http://neo4j.com

composed by one VM and a virtual network and you want to know on which host the VM has been deployed. Using the OpenStack API, this query would be complicated, requiring interacting with multiple API's from different services. However using the repository database with the following query the required information can be retrieved:

```
Match stack-[r]->resources-[s]->hypervisor-[u]->host where
stack.openstack_uuid='7fe39371-1379-4162-9deb-e904c4f2dc43' return stack,
resources, hypervisor, host
```

The query returns the following graph:

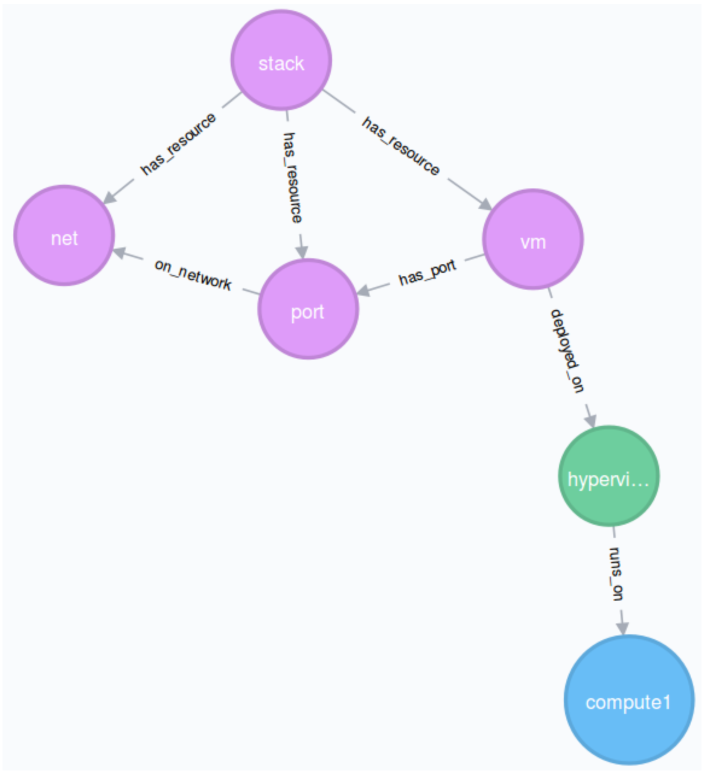


Figure 4-7 Cypher Query result

The graph, shown in Figure 4-7, is composed by:

- Nodes that represent the virtual resources (violet nodes). These virtual resources comprise a stack which is connected to a virtual network and to a VM on that network. The VM is connected to the network through a neutron port that represents its tap interface.
- The hypervisor where the VM has been deployed (green nodes).
- The physical host, having a hostname called compute1, where the hypervisor is running (blue nodes).

The controller and the middleware layer APIs access the infrastructure repository database using Py2neo[5] which is a client library that enables Python applications to work with Neo4j. In particular the controller and middleware share a Python module

[5] http://py2neo.org/2.0/

called neo4j_resources that wraps the Py2neo API to perform required actions before storing or updating new nodes or relationships in the database. All resources in the repository database are indexed using its UUID and label (node category) to boost performance.

The methods implemented are outlined in Table 4-2. Note: A graph_db is an instance of the Graph class from the py2neo library.

Table 4.2 Methods for Repository Database updates

Method	Description
create_index(**graph_db, label)**	Create an index in the database for the given label.
add_node(**graph_db, index, timestamp, properties**=**None**)	Add a new node with the given index, the given timestamp and optionally the given properties.
update_node(**graph_db, index, timestamp, properties**=**None**)	Update a node having the given index, with the new timestamps and optionally properties
delete_node(**graph_db, index, node**=**None**)	Delete the node with the given index. If you have already an instance pointing to the node, you can pass it. In this case index will be ignored.
add_edge(**graph_db, start_node, end_node, timestamp, label, properties=None**)	Add a relation between the start node and the end node, using the given label and timestamp and optionally properties.
delete_edge(**graph_db, start_node, end_node)**	Delete the relation between start_node and end_node
update_edge(**graph_db, start_node, end_node, timestamp, label, properties**=**None**)	Update a relation between the start node and the end node, using the given label and timestamp and optionally properties.
get_edges_by_node(**graph_db, index, node=None**)	Retrieve a list of relations for the given node, specified by index or by an instance of the node itself.
get_neighbours(**graph_db, index, node=None**)	Retrieve a list of nodes linked to the given node both with an ingress or egress relation. The node can be specified by index or by an instance of the node itself.
remove_neighbours(**graph_db, index, node=None, neighbour_type=None**)	Delete nodes linked to the given node both with an ingress or egress relationship. The node can be specified by index or by an instance of the node itself.
get_node_by_index(**graph_db, index)**	Retrieve a node given its index
get_node_by_property(**graph_db, label, property_key, property_value)**	Retrieve the first node with the given property and label.

get_edge**(graph_db, start_node, end_node)**	Retrieve the first relation between start and end node.
remove_nodes_by_property**(graph_db, label, property_key, property_value)**	Delete all nodes having the given label and property.

All functions can be called multiple times on the same data and will return the same output. In this way the design of components that use the library has been simplified.

With a graph database each resource is characterised by multiple links with other resources. For example when a user deploys a new stack composed by two VMs, this will be represented in the database as a Stack node connected to two VMs where each VM is connected to a port that in turn is connected to a virtual network. The two VMs will also be connected to the hypervisor where the VMs are running on. Each hypervisor is connected to a physical machine. If a VM requiring an SR-IOV NIC is deployed, the related port will be connected to the physical network card that supports SR-IOV.

In order to support consistency between the resource references used in the infrastructure repository and those used by the metric monitoring system being developed by the task 4.4 in WP4, the following conventions were adopted.

- Physical resources are identified by a combination of OpenStack hostname + kind (see Table 4.3).
- Virtual resources are identified by OpenStack UUID.
- Physical network resource by OpenFlow ID.

Use of these conventions ensures that the Orchestration layer can correlate metrics to the corresponding resources in the repository and vice versa.

Additionally, to standardise the naming convention of the T-NOVA PoPs the following convention was adopted.

- Country Code - Two letter code as per ISO 3166 (http://www.iso.org/iso/home/standards/country_codes.htm).
- City location code as per UN/LOCODE (http://www.unece.org/cefact/locode/service/location.html) and a 4 digit datacentre number (this could be increased if it makes sense).

An example of the convention applied to the Intel data centre in Leixlip, Ireland is:

e.g. IE-LEX-0001

The rational for this standardisation was to have short but meaningful names in order to facilitate shorter response strings to resource GET API calls where the source PoP is included in the response.

4.3. Listener Services Implementation

To ensure data concurrency in the repository database, updates by the EPA Controller are initiated via the Events Listener Service module. This module is connected to the RabbitMQ broker and consumes messages from the OpenStack "notifications.info" queue. An Event in OpenStack represents the state of an object in an OpenStack service (such as an Instance in Nova, or an Image in Glance) at a point in time when something of interest has occurred. In general, Events let you know when something has changed about an object in an OpenStack system, such as the resizing of an instance, or the creation of an image. Events are primarily created via the notifications system in OpenStack. OpenStack services, such as Nova, Glance, Neutron, etc. send notifications in a JSON format to the message queue when a notable action is taken by that system. The Events Listener consumes these notifications from the message queue, and processes them. To enable the notifications service in OpenStack the service configuration file must be updated as shown in Figure 4-8.

```
File: nova.conf (controller and computes)
[DEFAULT]
default_notification_level=INFO
notification_topics=notifications
notification_driver=nova.openstack.common.notifier.rpc_notifier
notify_on_state_change = vm_and_task_state
instance_usage_audit=True

File: cinder.conf (controller)
[DEFAULT]
default_notification_level=INFO
notification_topics=notifications
notification_driver=cinder.openstack.common.notifier.rpc_notifier

File: glance-api.conf (controller)
[DEFAULT]
default_notification_level=INFO
notification_topics=notifications
notification_driver=glance.openstack.common.notifier.rpc_notifier

File: heat.conf (controller)
[DEFAULT]
default_notification_level=INFO
notification_topics=notifications
notification_driver=heat.openstack.common.notifier.rpc_notifier

File: neutron.conf (controller)
[DEFAULT]
default_notification_level=INFO
notification_topics=notifications
notification_driver =
neutron.openstack.common.notifier.rpc_notifier
```

Figure 4-8 OpenStack Notification Configurations

After adding these configurations, all the OpenStack services must be rebooted to start producing notifications. All the notifications have a field called *event_type* which is based on a composite string with a dot delimiter defining what event has occurred. For example the notifications produced by Nova related to virtual machines' events will have event_type compute.instance.* (e.g. compute.instance.create.end, compute.instance.delete.end, compute.instance.update etc.)

The general architecture of the OpenStack Events Listener is shown in Figure 4-9. A component called Notifications consumer is responsible for:

- Creating the connection to the RabbitMQ Broker.
- Registering itself as a consumer for the messages in the notification queue.
- Providing registration functionalities to permit the handlers to register themselves for a specific event type patterns.

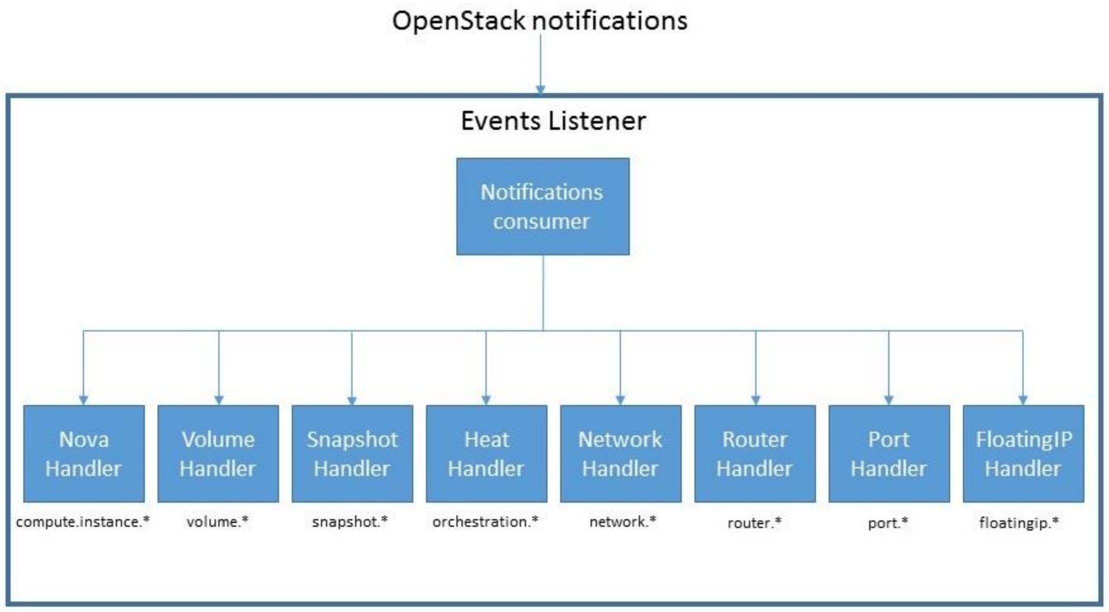


Figure 4-9 Events Listener Architecture

After the handler is registered for a specific pattern, it starts to receive the desired events. A new decorator[6] [10] called *register_handler* was also defined. The decorator takes the function, stores a reference to the function in a hash, using events as the key of the hash. The reference is then used whenever an event occurs connected to the key against which the function reference was stored. For example a Nova event handler would have the following format:

```
CREATE_EVENT = [
            "'compute.instance.create.end'
]
UPDATE_Events = [

        'compute.instance.resize.revert.end',
        'compute.instance.finish_resize.end',
        'compute.instance.rebuild.end',
        'compute.instance.update',
        'compute.instance.exists'
]
```

[6] *"A decorator is the name used for a software design pattern. Decorators dynamically alter the functionality of a function, method, or class without having to directly use subclasses or change the source code of the function being decorated."*

```
DELETE_EVENTS = [
'compute.instance.delete.end'
]

Class NovaHandler(OpenStackHandler):
        @register_handler(UPDATES_EVENTS)
        def handle_instance_update(self, graph_db, body):
        # Processing VMs update events
        @register_handler(CREATE_EVENTS)
        def handle_instance_create(self, graph_db, body):
        # Processing VMs create events
        @register_handler(DELETE_EVENTS)
        def handle_instance_delete(self, graph_db, body):
        # Processing VMs delete events
```

The common mechanisms for processing an event are:

- Querying OpenStack Service database to retrieve additional information about the resource that generated the event. This functionality was implemented using MySQL Connector/Python[7] which an open source API that is compliant with the Python Database API Specification v2.0. It is written in pure Python and does not have any dependencies except for the Python Standard Library.
- Update the repository database.

The EPA Agent Listener is implemented in a similar manner to the OpenStack listener service. The EPA agent listener is however specifically subscribed to the *agents.info* queue. This queue is configured in the RabbitMQ broker to specifically handle messages sent by the EPA agents. The queue is created during the installation process of EPA agent listener. The listener when subscribed to the queue waits for hardware information about new compute nodes added to the PoP or any updates relating existing nodes in the form of hardware changes or upgrades. The messages sent by the EPA Agents have an event type field that can have values *agent.new* or *agent.update*. The message also has a field where the path to the files sent by the agent are to the EPA Controller are stored. Once the EPA Agent Listener receives a new message, it informs the EPA Controller that new files require processing and provides the EPA Controller with the location of the files that must be processed. The specifics of how the EPA Controller service is implemented are outlined in the next section.

4.4. EPA Controller Service Implementation

The EPA controller service manages and orchestrates the infrastructure repository database, EPA agent and OpenStack notification listeners. At start-up the controller is responsible for initialising the infrastructure repository database by removing all

[7] http://dev.mysql.com/doc/connector-python/en/

previously present nodes and relationship entries (if any). The controller then populates the database by extracting data from the OpenStack service databases. When this step is completed the database contains all nodes and relations that represent the current OpenStack view of the datacentre, i.e. a screenshot of the current infrastructure. This establishes a ground truth state which can then be updated with new data produced by the two listeners (EPA agent and OpenStack). The set of operations performed in the initialisation of the EPA database are shown in Figure 4-9. Once the EPA database is initialised, the controller starts the Agent notification listeners.

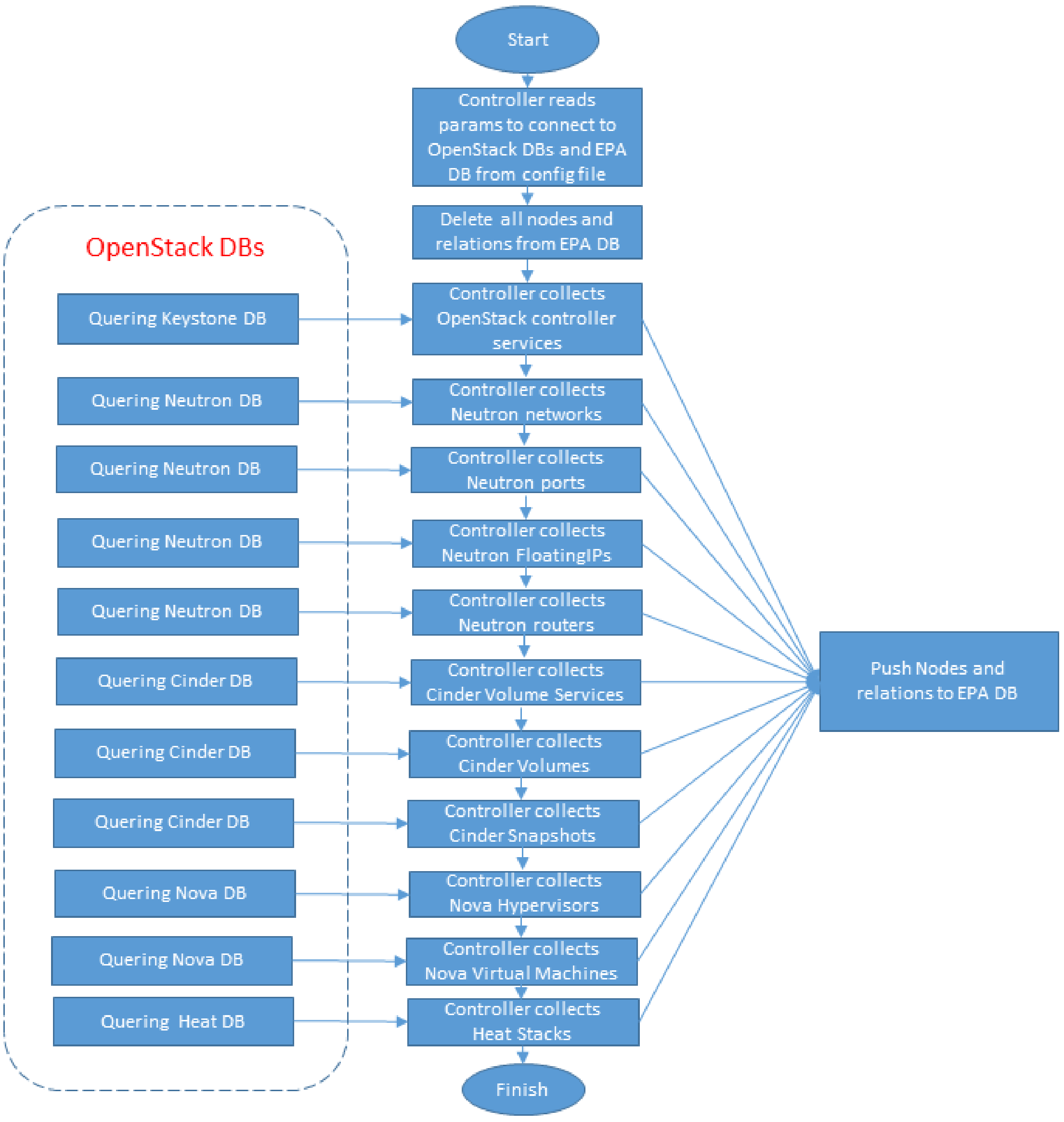

Start
Controller reads params to connect to OpenStack DBs and EPA DB from config file
Delete all nodes and relations from EPA DB
OpenStack DBs
Quering Keystone DB
Quering Neutron DB
Quering Neutron DB
Quering Neutron DB
Quering Neutron DB
Quering Cinder DB
Quering Cinder DB
Quering Cinder DB
Quering Nova DB
Quering Nova DB
Quering Heat DB
Controller collects OpenStack controller services
Controller collects Neutron networks
Controller collects Neutron ports
Controller collects Neutron FloatingIPs
Controller collects Neutron routers
Controller collects Cinder Volume Services
Controller collects Cinder Volumes
Controller collects Cinder Snapshots
Controller collects Nova Hypervisors
Controller collects Nova Virtual Machines
Controller collects Heat Stacks
Push Nodes and relations to EPA DB
Finish

Figure 4-9 EPA database initialisation flow diagram

When a new Agent notification is received, the controller takes the following actions:

1. Collects the files sent by the EPA Agent
2. Delete all nodes and relations from the database related to the hardware resources of the host that produced the notification.

3. Parses the HW locality file and converts the xml structure to a graph structure (for example see Figure 5-3).
4. Updates the Process Unit nodes with information contained in the CPU information file.
5. Updates the OS Network device with information contained in the DPDK information file.
6. Updates the OS Network device with information contained in the SR-IOV information file.

At the end of that process, the infrastructure repository database contains all the hardware information with respect to the host that sent the notification. Step 2 ensures that if a host sends multiple notifications, the database will not contain duplicate nodes for resources belonging to the same host. After starting the EPA Agent notification listener, the Controller starts the OpenStack Notification listener which keeps the information stored in the repository database up to date, using the mechanism outlined in Section 4.3.

4.5. Middleware API Layer Implementation

A key requirement for the infrastructure repository is to create a unified view of the T-Nova infrastructure environment composed by multiple Points of Presence (PoPs). Each PoP is a datacentre managed by a VIM based on OpenStack for compute and storage resources and OpenDaylight for the physical network topology. All infrastructure information is stored in the repository database except for the network topology which is retrieved directly from OpenDaylight via its REST API's. T-Nova uses a sharing-nothing approach, which means that the OpenStack and OpenDaylight instances are completely isolated. To follow this approach, each PoP has its own infrastructure database, where the infrastructure information for that PoP is stored (at PoP level). To achieve a unified view of the infrastructure information among multiples PoPs, the infrastructure repository implements a middleware layer.

The main responsibilities of the middleware layer are:

- Defining a common view for all information sources (OpenStack, EPA Agents and OpenDaylight);
- Dispatching user requests to the required PoP.

The middleware layer also has a database (at the middleware layer) where information relating to the PoPs in the T-NOVA system and the network links between them are stored.

Each PoP entry in the middleware layer database contains the following information:

- Name
- ID
- EPA database URL
- OpenDaylight URL
- OpenDaylight username
- OpenDaylight password

- Longitude
- Latitude

Each PoP entry also contains the URLs of the two sources of information at the PoP level (resource repository database URL and OpenDaylight URL).

The middleware layer exposes the API calls to manage the PoPs (Create, Retrieve, Update, Delete). When a request is sent to the middleware, the ID of the PoP that the user wants to access must be specified. In this way the middleware layer can retrieve the URLs and query the appropriate PoP level information sources.

The consistency of the OpenStack and EPA agent information contained in the infrastructure database is maintained using the listener services and EPA controller previously described. This approach cannot be used for physical network information, as OpenDaylight does not provide an equivalent messaging mechanism. For this reason when a T-NOVA Orchestration layer component requires information related to the physical network topology, OpenDaylight's (at the PoP level) REST API is called, however the specifics of the OpenDaylight API call are abstracted by the OCCI interface from the component making the request. From a requestor perspective the call for the physical network information appears to be the other middleware GET API calls. The database at the middleware layer also contains information regarding the connections between PoPs. For that reason, a graph database based on NEO4j was again selected for the implementation of this database. The PoPs link information can contain parameters such as currently available bandwidth and total link bandwidth when available. This information can for example used by the Service Mapping together with other PoP information to determine which PoP a VNF or network service should be deployed on, considering the network flows between multiple VNFs that belong to a given Network Service (NS).

The middleware is implemented in Python as a standalone application. The information required to install it are:

- Neo4j database URL.
- Neo4j database credentials.
- Port number used by the middleware to expose the service.

4.5.1. OCCI Compliant API's

The middleware layer exposes an OCCI[8] compliant interface to dependent functional entities within the T-NOVA Orchestrator. An OCCI approach was adopted as it builds on work carried out by the Mobile Cloud Networking (MCN) FP7 project which utilised OCCI interfaces in their system design and implementation [11]. It also provides abstraction of the underlying implementation of the infrastructure repository thus supporting easier reuse of the Orchestrator components by third parties with an alternative repository solution if required.

OCCI is a RESTful protocol and API for various kinds of management tasks. OCCI was originally initiated to create a remote management API for IaaS model-based services. These APIs support the development of interoperable tools for common

[8] http://occi-wg.org/about/specification/

tasks including deployment, autonomic scaling and monitoring [1]. The middleware interface was implemented using the pyssf package[9]. In accordance with the OCCI specification each resource in the repository is characterised by a *kind*. The kind is defined by a category in the OCCI model. This kind is immutable and specifies a resource's basic set of characteristics. This includes its location in the hierarchy, attributes, and applicable actions. The kinds exposed by the middleware layer are outlined in Table 4.3.

Table 4.3 Middleware API Kinds

Kind	Endpoint url	Description	Actions
PoP	/pop/	Point of presence	GET, POST, PUT, DELETE
PoP link	/pop/link/	Link between two POPs	GET, POST, PUT, DELETE
Stack	/pop/{pop_id}/stack/	OpenStack Stack	GET
Stack link	/pop/{pop_id}/stack/link/	Link between a stack and its resources	GET
VM	/pop/{pop_id}/vm/	Virtual Machine	GET
VM link	/pop/{pop_id}/vm/link/	Link between a vm and its resources (Volume, Port etc.)	GET
Volume	/pop/{pop_id}/volume/	Cinder volume	GET
Volume link	/pop/{pop_id}/volume/link/	Link with cinder volume service and snapshot	GET
Net	/pop/{pop_id}/net/	Neutron network	GET
Port	/pop/{pop_id}/port/	Neutron port	GET
Port link	/pop/{pop_id}/port/link/	Link between port, networks, floating IP, pci device (in case of PCI passthrough)	GET
Snapshot	/pop/{pop_id}/snapshot/	Cinder snapshot	GET
Floating IP	/pop/{pop_id}/floatingip/	Neutron Floating IP	GET
Floating IP link	/pop/{pop_id}/floatingip/link/	Link between floating ip and its network	GET
Router	/pop/{pop_id}/router/	Neutron virtual router	GET
Router link	/pop/{pop_id}/router/link/	Link between router	GET

[9] http://pyssf.sourceforge.net/

		and its interfaces	
Controller Service	/pop/{pop_id}/controller-service/	OpenStack Service (Nova, Glance, Cinder, Heat, Neutron)	GET
Controller Service link	/pop/{pop_id}/controller-service/link/	Link between OpenStack service and the machine where is hosted	GET
Hypervisor	/pop/{pop_id}/hypervisor/	Hypervisor used by Nova Compute	GET
Hypervisor link	/pop/{pop_id}/hypervisor/link/	Link between hypervisor and the machine where it is running on	GET
Cinder Volume	/pop/{pop_id}/cinder-volume/	Cinder Volume service	GET
Cinder Volume link	/pop/{pop_id}/cinder-volume/link/	Link between Cinder Volume service and the machine where it is running on	GET
Machine	/pop/{pop_id}/machine/	Physical machine	GET
Machine link	/pop/{pop_id}/machine/link/	Link between Machine and NUMA node (if NUMA architecture) or Bridge and Socket (if No NUMA architecture)	GET
NUMA Node	/pop/{pop_id}/numanode/	NUMA node	GET
NUMA node link	/pop/{pop_id}/numanode/link/	Link between NUMA node and Bridge or Socket	GET
PCI Bridge	/pop/{pop_id}/bridge/	PCI Bridge	GET
Bridge link	/pop/{pop_id}/bridge/link/	Link between PCI bridge and PCI devices connected to it	GET
PCI Device	/pop/{pop_id}/pcidev/	PCI Device	GET
PCI Device link	/pop/{pop_id}/pcidev/link/	Link between PCI Device and respective OS device	GET

OS Device	/pop/{pop_id}/osdev/	OS Device (allowed type: Compute, Network, Storage)	GET
OS Device link	/pop/{pop_id}/osdev/link/	Right now only Network device can have connection to SDN physical switch	GET
Socket	/pop/{pop_id}/socket/	Socket	GET
Socket link	/pop/{pop_id}/socket/link/	Link between socket and cache node	GET
Cache	/pop/{pop_id}/cache/	Cache	GET
Cache link	/pop/{pop_id}/socket/link/	Link to other cache nodes of lower level or to the Core	GET
Core	/pop/{pop_id}/core/	Physical Core	GET
Core link	/pop/{pop_id}/core/link/	Link to Process Units node	GET
PU	/pop/{pop_id}/pu/	Processing unit	GET
Switch	/pop/{pop_id}/switch/	Physical SDN switch	GET
Switch link	/pop/{pop_id}/switch/link/	Link between Switch and its interfaces controlled by ODL	GET
Switch Interface	/pop/{pop_id}/switch-interface/	Switch interface controlled by ODL controller	GET
Switch Interface link	/pop/{pop_id}/switch-interface/link/	Link between Switch Interface and Network card of Physical Node	GET

A complete API reference document is available particularly for partners working on integration of Orchestration layer components with infrastructure repository subsystem. Most of kinds allow only the retrieve action, as the infrastructure repository is updated automatically.

Only PoP WAN related information is fully managed by the Orchestrator. To add a new PoP the Orchestration layer uses the following call:

```
POST http://middleware_url:<middleware_port>/pop/

--header "Accept: application/occi+json"
--header "Content-Type: text/occi" --header 'Category: pop;
scheme="http://schemas.ogf.org/occi/epa#";class="kind"'
-d 'X-OCCI-Attribute: occi.epa.pop.name = "GR-ATH-0001"
```

```
   X-OCCI-Attribute: occi.epa.pop.lat = 53.3720513
   X-OCCI-Attribute: occi.epa.pop.lon = -6.5130686999999625
   X-OCCI-Attribute: occi.epa.pop.graph_db_url
"http://neo4j:intel_tnova@demokritos.com:7474/db/data/"
   X-OCCI-Attribute: occi.epa.pop.odl_url =
                                        "http://demokritos.com:9001/restconf/operational/"
   X-OCCI-Attribute: occi.epa.pop.odl_name = "admin"
   X-OCCI-Attribute: occi.epa.pop.odl_password="admin"'
```

For each kind there are least two calls available:

- One to retrieve the list of resources of the given kind
- One to retrieve a single resource and its attributes.

For example:

Retrieving a list of virtual machines:

(Note: The PoP ID must be included in the request of the call)

```
GET http://middleware_url:<middleware_port>/pop/55ef7cce-
1e9b-4b8f-9839-d40ceeb670f4/vm/
--header "Accept: application/occi+json"
```

Extract from response:

```
[
 {
   "actions": [],
   "attributes": {},
   "identifier": "/vm/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa",
     "title": "Virtual Machine"
   },
   "links": [
     {
       "actions": [],
       "attributes": {},
       "identifier": "/vm/link/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa->f18a3c74-e3de-
4271-9284-e47af46471ba",
       "source": "/vm/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa",
       "target": "/port/f18a3c74-e3de-4271-9284-e47af46471ba"
     },
     {
       "actions": [],
       "attributes": {},
       "identifier": "/vm/link/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa->hypervisor-2",
       "source": "/vm/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa",
       "target": "/hypervisor/hypervisor-2"
     }
   ],
   "mixins": []
 },
 {
   "actions": [],
   "attributes": {},
   "identifier": "/vm/cf3365f6-ee18-4f54-9acd-38f8855249ab",
   },
   "links": [
     {
       "actions": [],
       "attributes": {},
       "identifier": "/vm/link/cf3365f6-ee18-4f54-9acd-38f8855249ab->3f98d78e-a817-
418a-aeee-44b0edf14169",
       "source": "/vm/cf3365f6-ee18-4f54-9acd-38f8855249ab",
       "target": "/port/3f98d78e-a817-418a-aeee-44b0edf14169"
     },
     {
       "actions": [],
       "attributes": {},
       "identifier": "/vm/link/cf3365f6-ee18-4f54-9acd-38f8855249ab->hypervisor-2",
       "source": "/vm/cf3365f6-ee18-4f54-9acd-38f8855249ab",
       "target": "/hypervisor/hypervisor-2"
     }
   ],
   "mixins": []
 },

....]
```

Retrieving the details of a single virtual machine:

```
GET http://middleware_url:<middleware_port>/pop/55ef7cce-1e9b-4b8f-9839-
d40ceeb670f4/vm/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa
--header "Accept: application/occi+json"
```

Extract from response:

```
{
 "actions": [],
 "attributes": {
   "occi.epa.attributes": "{\"vm_state\": \"active\", \"internal_id\": null,
\"availability_zone\": \"nova\", \"ramdisk_id\": \"\", \"instance_type_id\": \"2\",
\"cleaned\": 0, \"vm_mode\": null,
\"reservation_id\": \"r-r1fob1wj\", \"disable_terminate\": 0, \"user_id\":
\"d719c3652ff64911a3c896e9c11f53e3\", \"default_swap_device\": null,
\"hostname\": \"test-ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa\", \"launched_on\":
\"controller\", \"display_description\": \"test\",
\"power_state\": 1, \"default_ephemeral_device\": null, \"progress\": 0,
\"project_id\": \"b6488d1a9ff34bcfb3f95d0d4399b0b3\", \"root_device_name\":
\"/dev/vda\", \"node\": \"controller\",
\"ephemeral_gb\": 0, \"access_ip_v6\": null, \"access_ip_v4\": null, \"kernel_id\": \"\",
\"key_name\": \"odl-keypair\", \"image_id\": \"83da27be-a376-4920-ab3a-
812473258cfd\", \"host\": \"controller\",
\"ephemeral_key_uuid\": null, \"task_state\": null, \"shutdown_terminate\": 0,
\"cell_name\": null, \"root_gb\": 20, \"locked\": 0, \"locked_by\": null,
\"launch_index\": 1, \"memory_mb\": 2048, \"vcpus\": 1,
\"architecture\": null, \"auto_disk_config\": 1, \"os_type\": null, \"config_drive\": \"\",
\"ports\": [\"f18a3c74-e3de-4271-9284-e47af46471ba\"]}",
   "occi.epa.category": "compute",
   "occi.epa.hostname": "controller",
   "occi.epa.name": "test-ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa",
   "occi.epa.pop": "IR-LEX-0001",
  "occi.epa.pop_id": "55ef7cce-1e9b-4b8f-9839-d40ceeb670f4",
  "occi.epa.resource_type": "vm",
  "occi.epa.timestamp": 1434536853.240091
 },
 "identifier": "/vm/ee3fa7b8-ad1f-46c7-8944-b7dc2640dcaa",

...}
```

5. Information Resources

The infrastructure repository database contains information on the virtual and physical resources of a T-NOVA PoP. Each resource is represented in the database as a node with links to others nodes. The following tables outline the key information stored with respect to the physical resources (Table 5.1), virtual resources (Table 5.2) and physical network resources (Table 5.3).

Table 5.1 Physical resources

Type	Description	Main Attributes
Physical Machine	Each host of the OpenStack cluster	Operating system information, hostname, architecture
Bridge	PCI Bridge	Type of bridge
Socket	CPU socket	CPU model supported
Core	Core	
PU	Process unit	Cache size, bogomips, model, cpu speed
NUMA node	Group of CPU in NUMA architecture	NUMA node index, local memory size
Cache	Node representing a cache memory	Cache type (data or instruction), size, cache line size
PCI device	PCI device corresponding to an OS device. Information relates to how the device is seen by the OS	PCI vendor, pci_type (storage, network, compute), name
Storage OS Device	Storage device like disk	Name
Network OS device	Network device like network card	Name, mac address, SR-IOV information, DPDK information

Table 5.2 Virtual Resources:

Type	Description	Main Attributes
Network	Virtual network	Status, DHCP agent, subnets information, network type (GRE, VLAN, VXLAN)

Port	Neutron ports	Driver (OpenDaylight, openvswitch), mac address, floating ips, IP address
Floating IP	Neutron floating IP associated to a Neutron port	Router ID, Fixed Port, IP
Router	Neutron Virtual Router	Gateway, status, l3 agent information
VM	Nova virtual machine	Image, flavour, IP addresses, hostname,
Volume	Cinder Volumes	Size, mount point, attachment information
Stack	Heat stack	Template, list of associated resources, status
Hypervisor	Hypervisor used by Nova	Information about running VMs, supported features and architecture
Cinder Volume	OpenStack service for Volume management	Status
Snapshot	Cinder volume snapshot	Status, original volume information
Glance service	Glance API service (image management)	Endpoints
Heat service	Heat API service (orchestration management)	Endpoints
Cinder service	Cinder API service (volume management)	Endpoints
Nova service	Nova controller service (VMs management)	Endpoints
Neutron service	Neutron controller service (Network management)	Endpoints

Table 5.3 Physical network resources

Type	Description	Main attributes
Physical Switch	Physical SDN switch controlled by OpenDaylight	Manufacturer, switch features, management IP address, software version
Switch interface	Physical switch interfaces, eventually connected to the hosts network cards	Name, Interface features, status, received/transmitted packets, mac address

To augment the information contained in Tables 5.1-5.3, detail descriptions of the all the resources types and their attributes have been documented as outlined in Table 5.4.

Table 5.4 Infrastructure database information resource documentation

Document	Document #
Enhanced Platform Awareness Database - Physical Resource Description	T-NOVA WP3-T2-001
Enhanced Platform Awareness Database - Virtual Resources Description	T-NOVA WP3-T2-002
Enhanced Platform Awareness Database – Physical Network Resources Description	T-NOVA WP3-T2-003
Enhanced Platform Awareness Database – Resources Links Description	T-NOVA WP3-T2-004

5.1. Infrastructure Repository Data Model

The purpose of the infrastructure repository data model is to define and organise how the data elements are extracted from the NFVI-PoP resources and to define how the data elements relate to one another. In approaching a graph data model the key question to consider is what domain knowledge will be extracted from the graph. A key advantage of a graph data model is that it is good at showing how resources are related to each other; it is also helpful in formulating the likely questions that will be asked. A graph database takes a different approach to “connection” relationships in comparison to traditional SQL approaches. The richness and expressive nature of the relationships between nodes are as important as the actual nodes. This combination of features provides a convenient mapping to complex systems such as those found in a NFVI-PoP which can be broken down into layers and inter layer relationships.

The infrastructure repository data model can be broken down into four primary layers, namely workloads (i.e. virtualised network functions and network services), virtual resources (e.g. virtual machines, networks), resource virtualisation (e.g. hypervisor) and finally physical resources (compute, storage and network) as shown in Figure 5-1. The model also comprehends the relationship between the nodes in the layers. For example, the connection between VM and hypervisor is “deployed on” which encapsulates the explicit relationship between the two node types.

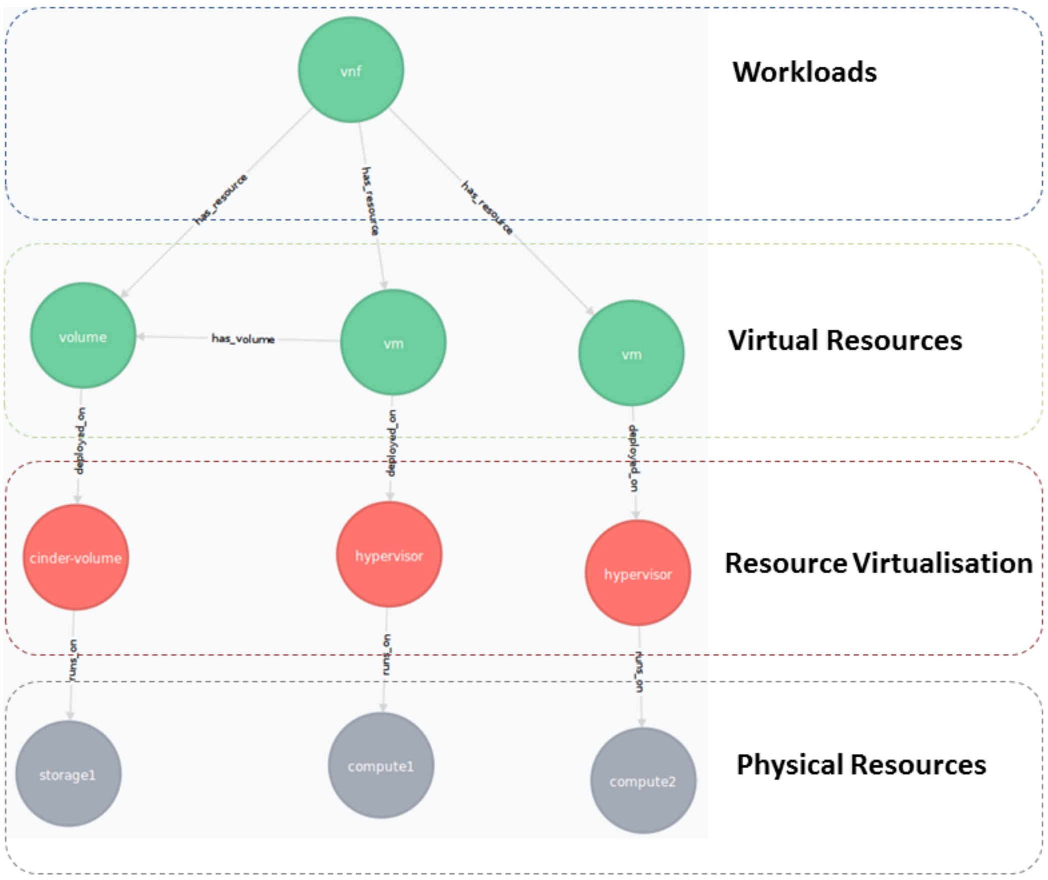


Figure 5-1 High level data model for infrastructure repository

In Figure 5-2 the relationship between computes nodes and the physical network is illustrated. The relevant components in a compute node that are involved in providing a network connection are shown in the model.

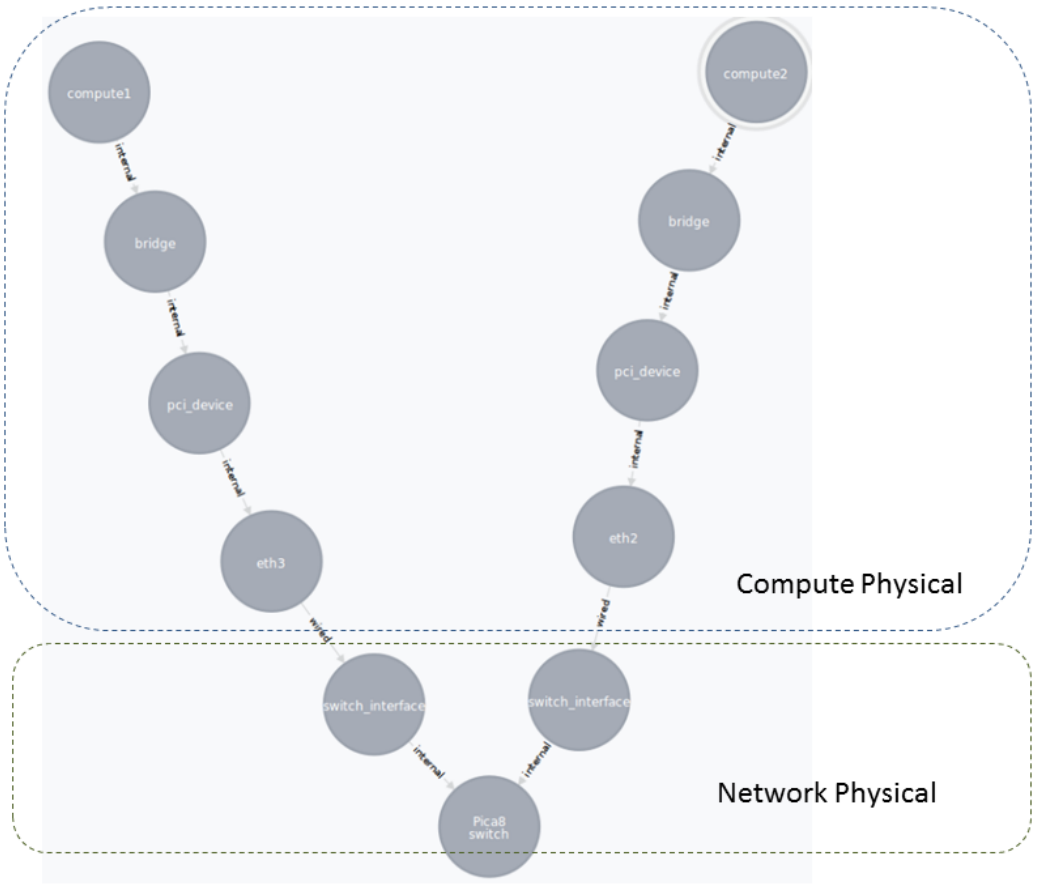


Figure 5-2 Data model for compute and physical network related resources

In Figures 5-3 and 5-4 detail models of both the physical and virtual resources are presented. In Figure 5-3 the model relates to a two socket server node with a NUMA implementation.

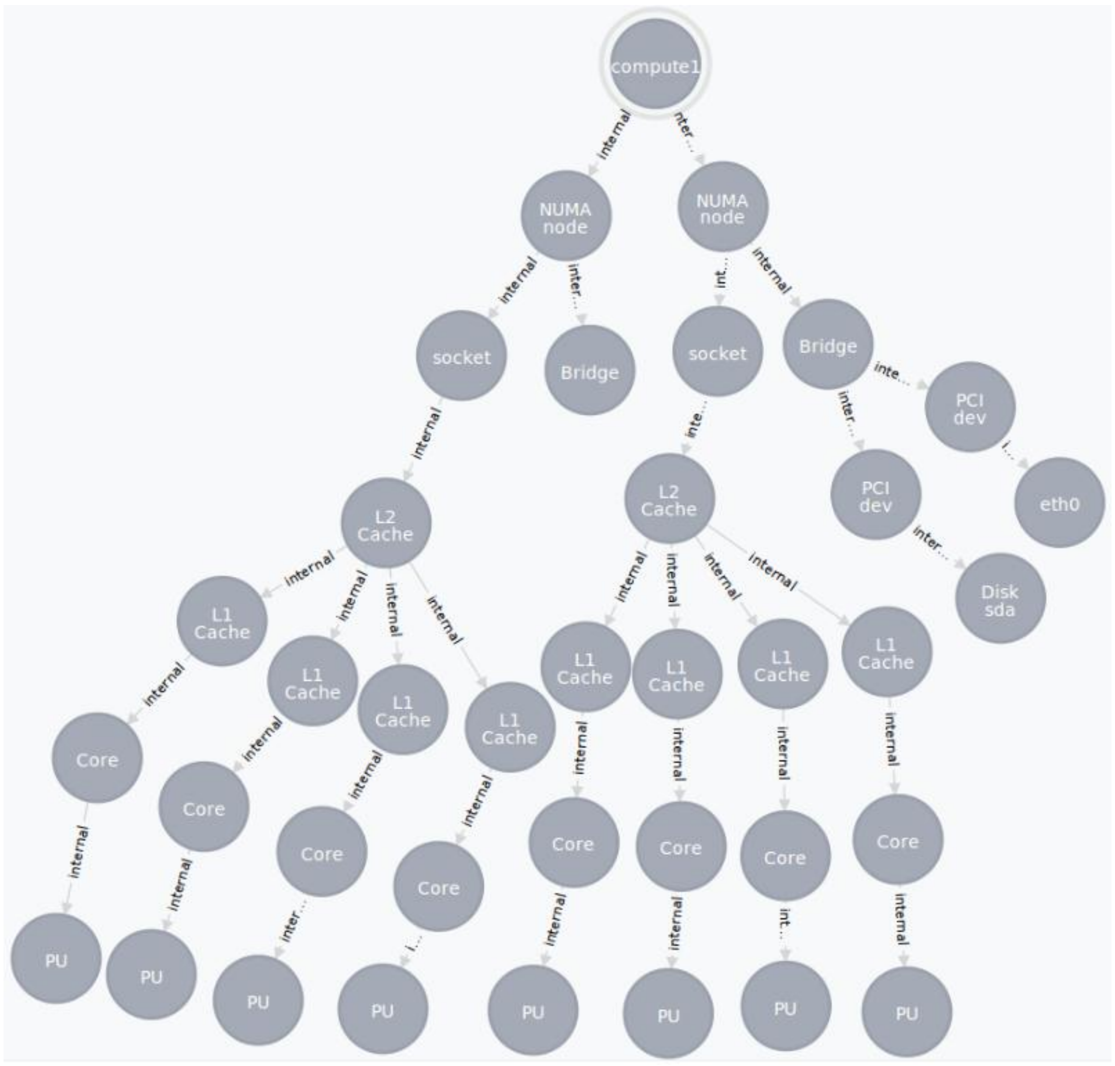


Figure 5-3 Data model for compute resources

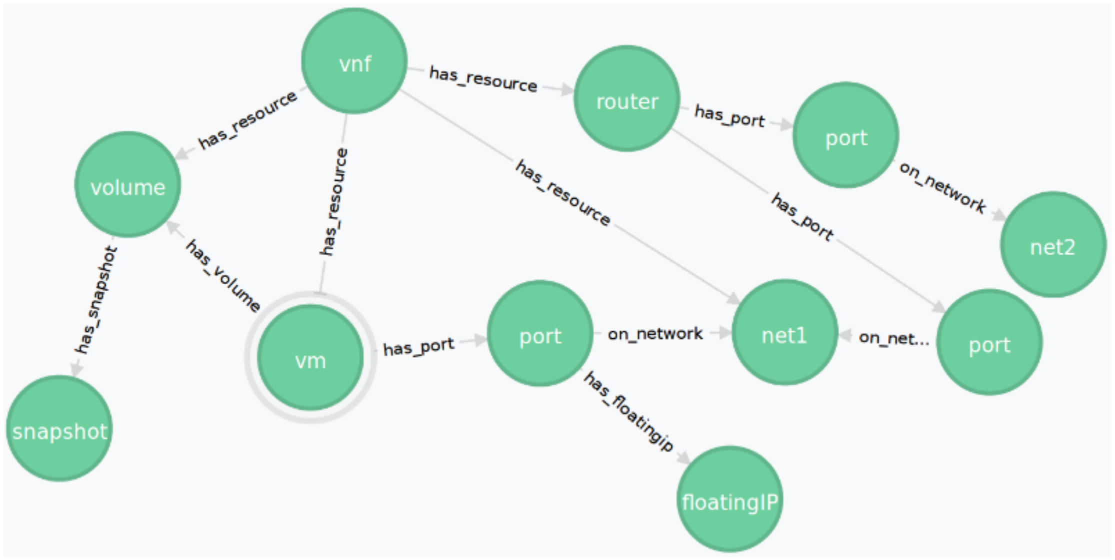


Figure 5-4 Data model for virtual resources

5.2. Resource Visualisation

The infrastructure repository provides a rich set of data which can be used by various 'users' or components within the T-NOVA orchestration layer. An expected use of the data is to provide use case specific visualisation of the data repository status. While the generalised visualisation of the data will be implemented as part of the Orchestrator management interface, a number of sample use cases were identified to validate the usefulness of the data stored in the repository, and the relationships defined in the connections between the nodes. The four use cases identified are not intended to represent an exhaustive investigation of all the potential visualisation uses cases, but rather to provide indicative use cases. These use cases represent real world operational scenarios which can be supported by the repository data and demonstrate how visualisation of the relevant data provides meaningful added value. The four sample use cases identified are:

- Visualisation of NFVI-PoPs in T-NOVA System
- VNF Resource Allocation
- Physical Network Topology
- Network Port Failure Identification

5.2.1. Use Case 1

Use Case ID	**IRS-VIS-001**
Use Case Name	Visualisation of NFVI-PoPs in T-Nova System
Actors	Orchestrator Administrator
Purpose	Visualisation used by the Orchestrator Administrator to view the status of the WAN links between the NFVI-PoP.
Description	Orchestrator inserts WAN endpoint information and associated statistics into the middleware PoP database. The administrator determines there is an issue with the configuration of a WAN connection. The administrator uses the visualisation to investigate configuration of the links between the PoP's and determine the root cause of issue. The administrator requests a WAN link configuration change e.g. increase bandwidth allocation to resolve the issue.
Assumptions	The middleware database has been updated by the T-NOVA Orchestrator or other entity within the T-NOVA system with PoP ingress and egress endpoints and associated parametric data.

Figure 5-5 shows the set of actions within the infrastructure repository subsystem that are required to retrieve the necessary information to fulfil the use case. Figure 5-6 shows the visualisation using Alchemy.js.

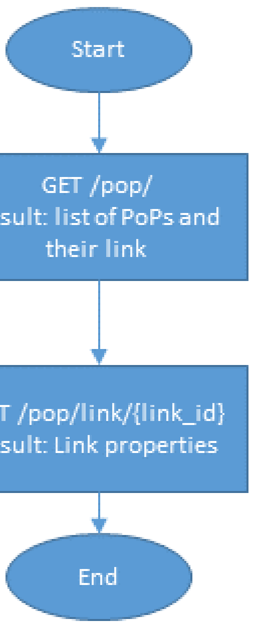
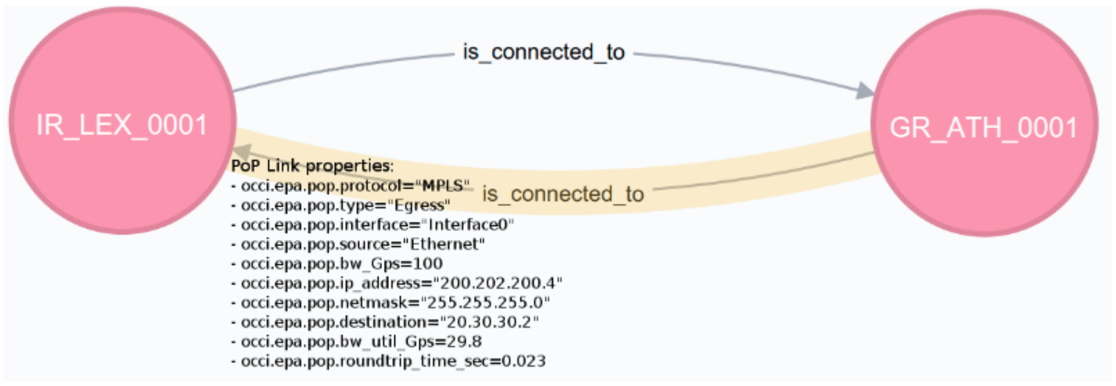


Figure 5-5 Retrieve PoPs Topology Flow Diagram



```
PoP Link properties:
- occi.epa.pop.protocol="MPLS"
- occi.epa.pop.type="Egress"
- occi.epa.pop.interface="Interface0"
- occi.epa.pop.source="Ethernet"
- occi.epa.pop.bw_Gps=100
- occi.epa.pop.ip_address="200.202.200.4"
- occi.epa.pop.netmask="255.255.255.0"
- occi.epa.pop.destination="20.30.30.2"
- occi.epa.pop.bw_util_Gps=29.8
- occi.epa.pop.roundtrip_time_sec=0.023
```

Figure 5-6 Example of a T-Nova PoPs topology

5.2.2. Use Case 2

Use Case ID	**IRS-VIS-002**
Use Case Name	VNF Resource Allocation
Actors	Customer
Purpose	Allows a customer to determine if requested resource allocation for purchased VNF has been correctly performed
Description	The customer purchases a Network Service with a specific resource allocation. From a menu the customer selects their NS service to visualise. The customer uses the visualisation to inspect if the VNFs composing up the service have been deployed in accordance with their purchase request. The high level details of the NS service stack deployment are

	visualised. The customer clicks on the icon to display detailed information on a specific component. If the customer is unhappy with the deployment configuration they contact the SP for resolution.
Assumptions	Assumes that OpenStack resource information has been updated correctly within the repository.

Figure 5-7 shows the set of actions within the infrastructure repository subsystem that are required to retrieve the necessary information to fulfil the use case. Figure 5-8 shows the visualisation using Alchemy.js.

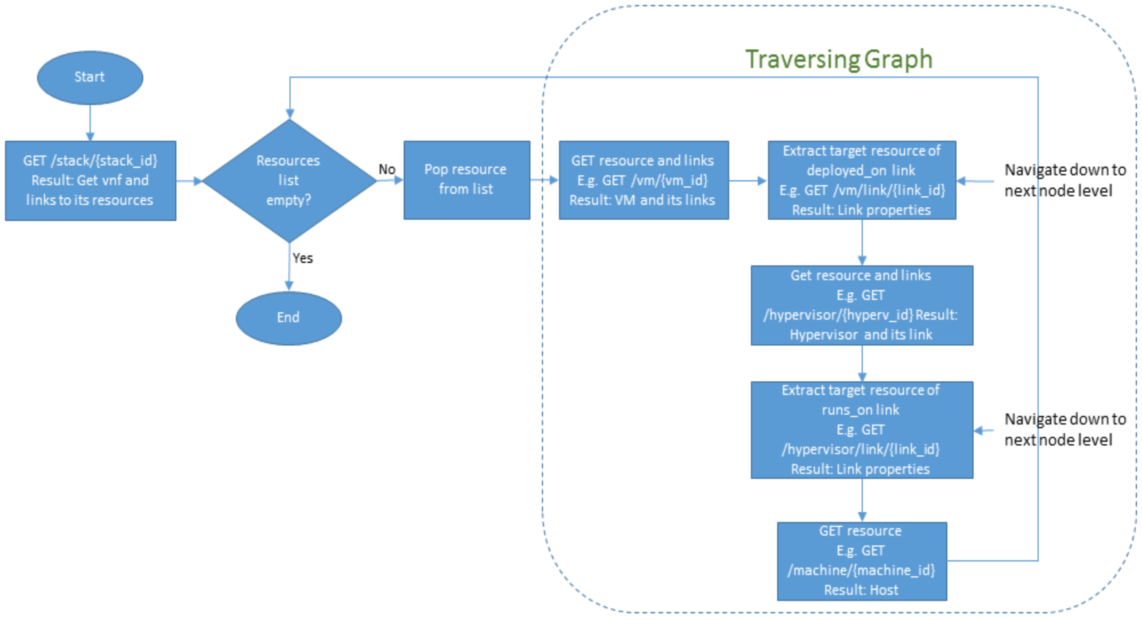

Traversing Graph
Start
GET /stack/{stack_id} Result: Get vnf and links to its resources
Resources list empty?
No
Yes
End
Pop resource from list
GET resource and links E.g. GET /vm/{vm_id} Result: VM and its links
Extract target resource of deployed_on link E.g. GET /vm/link/{link_id} Result: Link properties
Navigate down to next node level
Get resource and links E.g. GET /hypervisor/{hyperv_id} Result: Hypervisor and its link
Extract target resource of runs_on link E.g. GET /hypervisor/link/{link_id} Result: Link properties
Navigate down to next node level
GET resource E.g. GET /machine/{machine_id} Result: Host

Figure 5-7 Retrieve VNF allocation Flow Diagram

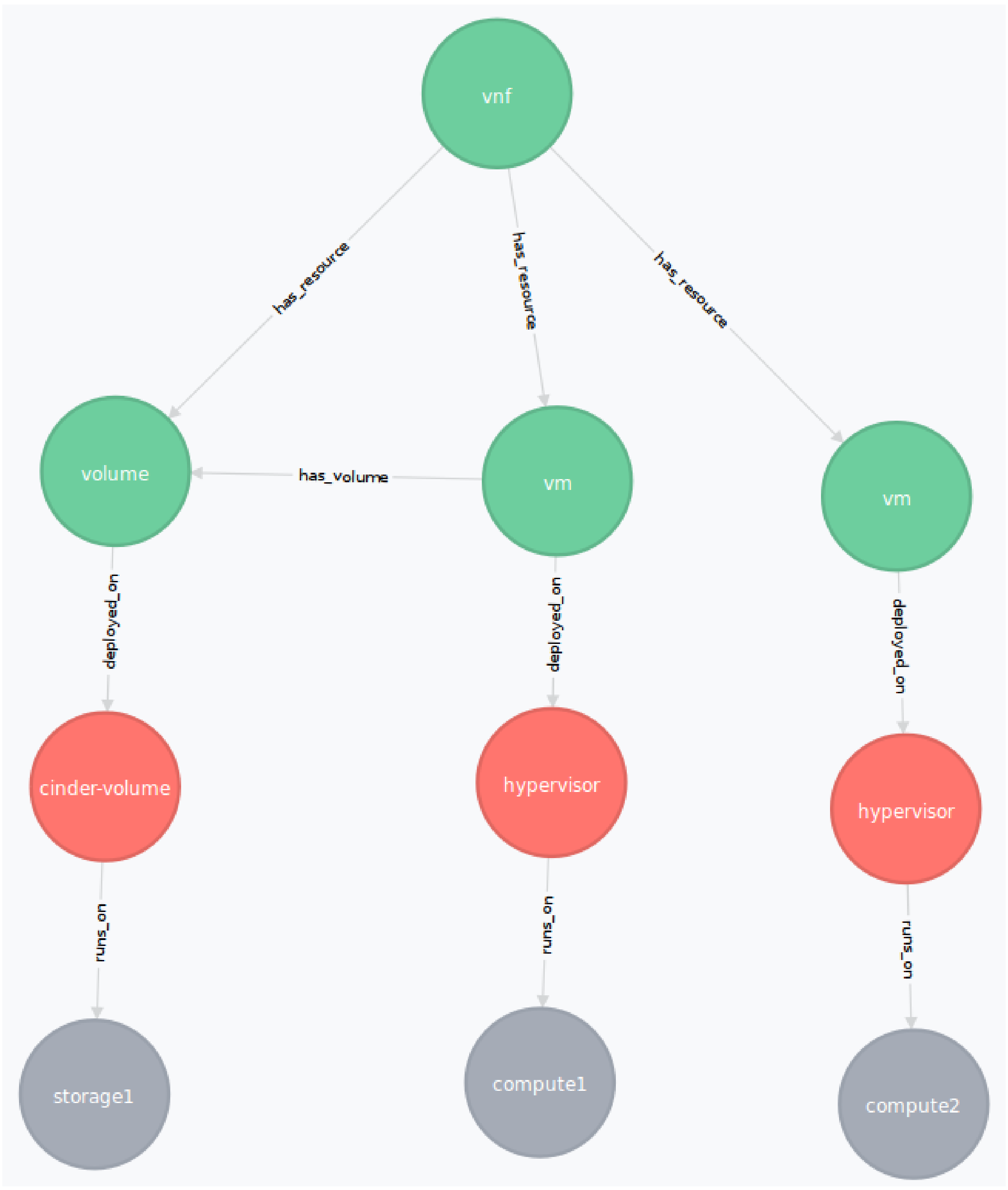


Figure 5-8 VNF Resources allocation graph

5.2.3. Use Case 3

Use Case ID	**IRS-VIS-003**
Use Case Name	Physical Network Topology
Actors	DC Administrator
Purpose	Allows a data centre administrator visualisation the complete or a subsection of the physical network topology to support redesign activities.
Description	The Administrator enters the name of the NFVI-PoP.

	The physical network topology of the DC is visualised showing the physical OpenFlow enable switches, the connections between the switches and the physical nodes attached to the switches. Additional information is available on the ports of the physical nodes connected to the switches. Information is available on the both switches and ports on the switches. The administrator can filter the visualisation to a subset of the physical network topology based on selections such as switch only.
Assumptions	The OpenDaylight controller's REST APIs are available.

Figure 5-9 shows the set of actions within the infrastructure repository subsystem that are required to retrieve the necessary information to fulfil the use case. Figure 5-10 shows the visualisation using Alchemy.js.

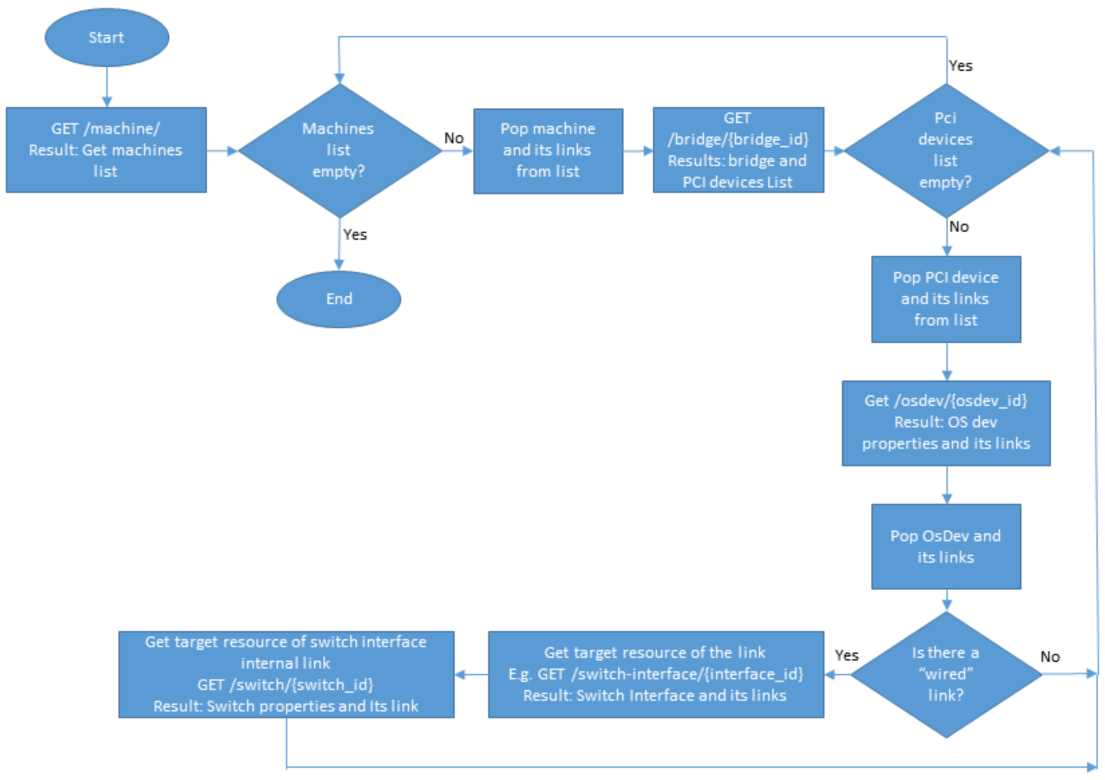

Start
GET /machine/ Result: Get machines list
Machines list empty?
No
Yes
End
Pop machine and its links from list
GET /bridge/{bridge_id} Results: bridge and PCI devices List
Pci devices list empty?
Yes
No
Pop PCI device and its links from list
Get /osdev/{osdev_id} Result: OS dev properties and its links
Pop OsDev and its links
Is there a "wired" link?
Yes
No
Get target resource of the link E.g. GET /switch-interface/{interface_id} Result: Switch Interface and its links
Get target resource of switch interface internal link GET /switch/{switch_id} Result: Switch properties and Its link

Figure 5-9 Retrieve Network Topology Flow Diagram

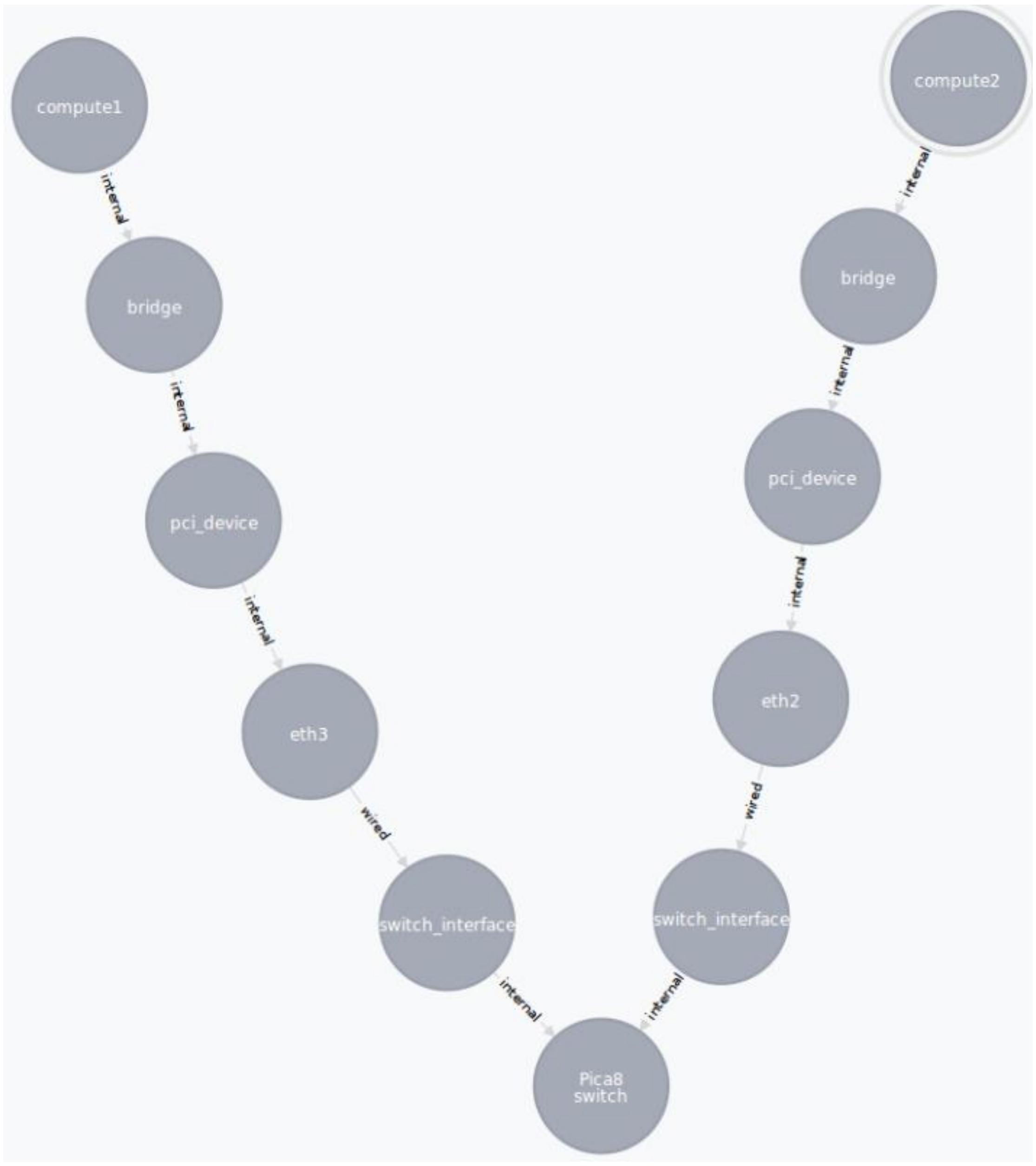


Figure 5-10 Physical Network Topology

5.2.4. Use Case 4

Use Case ID	**IRS-VIS-004**
Use Case Name	Network Port Failure Identification
Actors	DC Administrator
Purpose	Allows an administrator to troubleshoot the loss of network connectivity to a VNF or set of VNFs running on compute node connected to an SDN switch.
Description	The Administrator enters the NFVI PoP name The Administrator then enters the name of a virtual network.

	The active and inactive ports are displayed. The administrator uses the visualisation to identify VM's which are affected by a network port failure on a physical switch or NIC or vSwitch.
Assumptions	The OpenDaylight controller's REST APIs are available The compute node information in the infrastructure repository is up to date.

Figure 5-11 shows the set of actions within the infrastructure repository subsystem that are required to retrieve the necessary information to fulfil the use case. Figure 5-12 shows the visualisation using Alchemy.js.

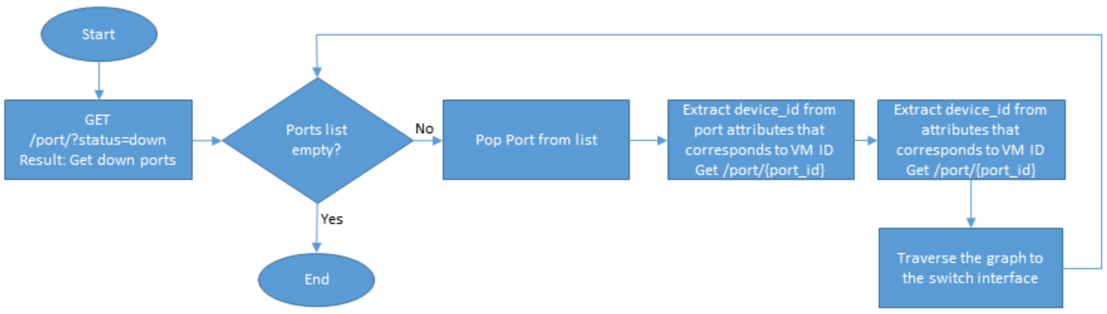

Start
GET /port/?status=down Result: Get down ports
Ports list empty?
No
Yes
End
Pop Port from list
Extract device_id from port attributes that corresponds to VM ID Get /port/{port_id}
Extract device_id from attributes that corresponds to VM ID Get /port/{port_id}
Traverse the graph to the switch interface

Figure 5-11 Flow diagram illustrating the steps in the retrieval of network ports in failed state

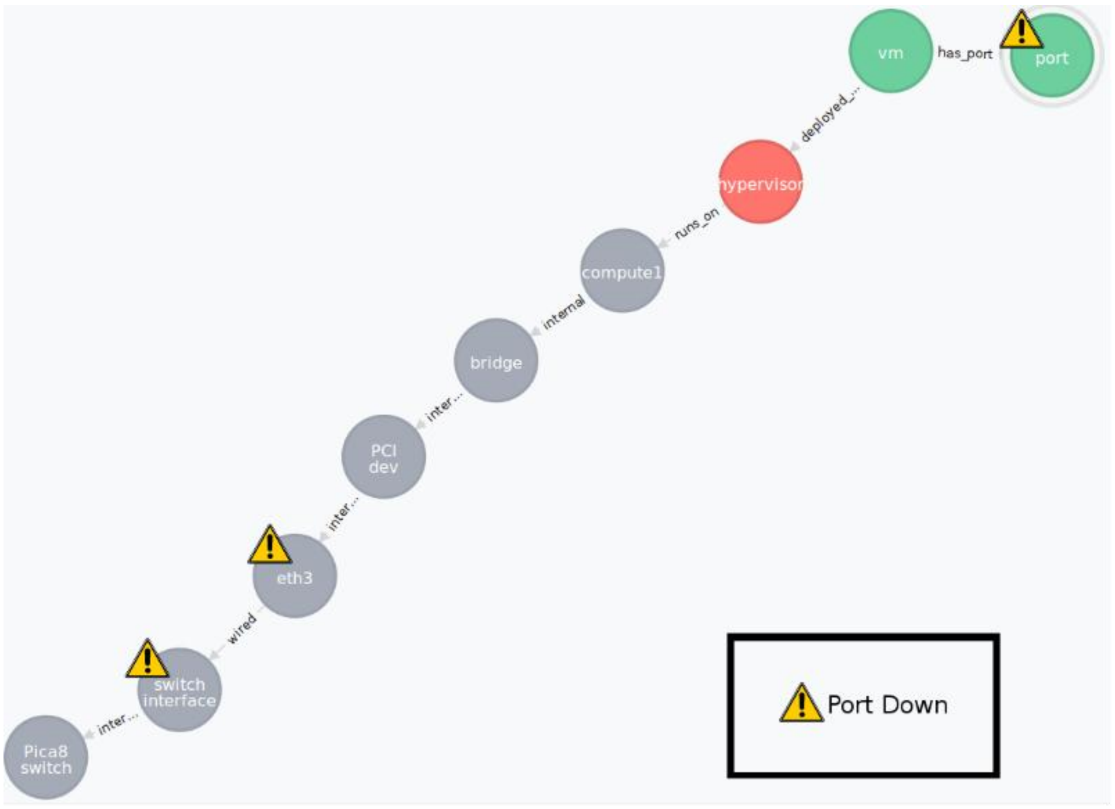


Figure 5-12 Visualisation Network Port failure between a switch and compute node

6. INFRASTRUCTURE REPOSITORY SUBSYSTEM INTEGRATION

The following sections describe the key T-NOVA system integration points with the infrastructure resource repository subsystem, or the repository's dependence on system functional features, namely security and authentication.

6.1. Resource Mapping Algorithm

The Service Mapping (SM) component of T-NOVA focuses on the optimal assignment of Network Service (NS) chains to servers hosted in an NFVI-PoP. The candidate hardware apparatus for a mapping, i.e. servers and links within each PoP and links between couples of PoP, have to be able to support the performance requirements of the VNFs composing the NS. In particular, a feasible solution of the service mapping problem must respect the following three requirements.

Node Requirements: A set of node resource types, say *NT*, is associated to the VNFs composing the NS and the PoP in the NFVI. Each member of the *NT* set represents a particular resource (e.g. CPU power need, number of cores, number of hardware and software accelerators, number of GPUs, etc.), which can be required by a *VNF*, since it could be required by some of its component. A numerical value, say RR_h^t, is associated to each VNF *h*, with $t \in NT$. It represents the amount of aggregate resource of type *t* required by the VNF *h*. A numeric value, say RA_u^t, is associated to each PoP *u*, with $t \in NT$. It represents the amount of aggregate resource of type *t* available in the PoP *u*.

For each resource type *t* present in a VNF, the SM algorithm needs to compute the aggregate value of that resource type available in each PoP of the NFVI, since for each PoP node *u* and resource type *t*, the sum of the aggregate resource needs of all VNFs mapped to it cannot exceed the aggregate available resource RA_u^t.

Link Requirements: A set of link resource types, say *LT*, is associated to the links in the NS chain and to those connecting different PoP. Each member of the *LT* set represents a particular resource (e.g. bandwidth) which can be required by an arc (*h*,*k*) in the *NS* chain. A numerical value, say RR_{hk}^t, is associated to each arc (*h*,*k*) in the *NS* chain, with $t \in LT$. It represents the amount of resource of type *t* required by the arc (*h*,*k*). A numeric value, say RA_{uv}^t, is associated to each arc (*u*,*v*) in NFVI, with $t \in LT$. It represents the amount of resource of type *t* available in the arc (*u*,*v*): for each arc (u,v) and each resource type $t \in LT$, the sum of the RR_{hk}^t, values of NS arcs mapped to paths including (u,v) cannot exceed RA_{uv}^t.

SLA: Each NS request corresponds to a set, say P, of paths connecting pairs of VNFs. Each path, say $\pi \in P$ is a sequence of arcs in the NS chain. A maximum allowed delay, say $\Delta\pi$, is associated to each path π in P. An actual delay δpq is associated to each arc (p,q) in the NFVI. For each path $\pi \in P$, the sum of the δpq of all the arcs (p,q) in the NFVI belonging the paths used for connecting all the links belonging to π, cannot exceed $\Delta\pi$.

The APIs implemented by the resource repository middleware layer represents the source of information needed by the SM algorithm to have visibility to the level of available infrastructure resources. In this way, the solutions computed by the Service Mapping algorithm are guaranteed to be aligned with the actual state of the infrastructure, respecting resource limits at each PoP. In particular, Use Case 1 shows how the SM algorithm can use "GET PoP" API to retrieve link information (GET http://.../pop/, GET http://.../pop/link) and Use Case 3 shows how the SM algorithm can use "GET resource" to retrieve resource information (GET http://.../machine/, GET http://.../core/, ...).

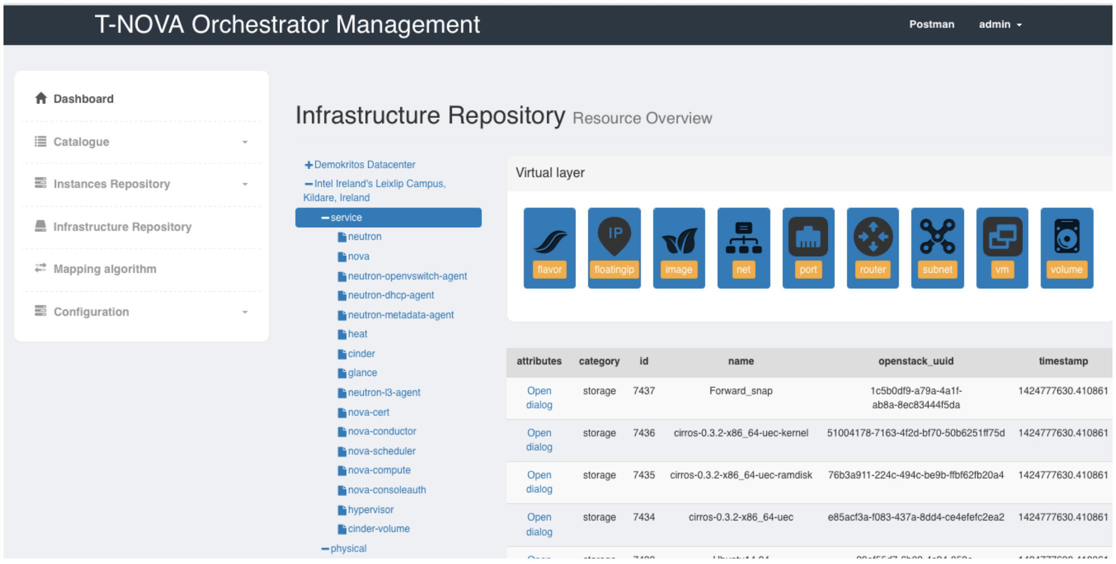
Full details on T-NOVA SA algorithm design and implementation will be presented in T-NOVA deliverable "Deliverable 3.3 Service Mapping", due on month 24.

6.2. Orchestrator Integration

The T-NOVA Orchestrator comprises different modules requiring infrastructure information from the repository. As the Orchestrator is built following the micro-service pattern, different modules can independently access the infrastructure repository through the middleware APIs. Table 6-1 outlines the core micro-services of the Orchestrator which can connect to the infrastructure resource repository.

Table 6.1 Orchestrator micro service dependencies

Micro-service	Description
NS Manager	Entry-point of the Orchestrator. It works as the internal coordinator of the orchestrator at the service level. It connects with the infrastructure repository in order to enable external entities (if authenticated and authorised) to obtain the corresponding information of the infrastructure repository.
NS Provisioning	NS Provisioning is the micro-service responsible for the deployment and instantiation of any given network service, starting with the software instantiation itself and ending with the deployment of the corresponding VNFs on the NFV Infrastructure. The NS provisioning, in coordination with the service mapping micro-service, is responsible for generating a notification when a service is successfully deployed, so that the infrastructure repository can update the capacity and information of the NFVI-PoP accordingly. This micro-service also consumes information from the infrastructure repository in order to be aware of the different services deployed and its status on the infrastructure.
Management UI	The management UI component utilises the repository, to support various use cases such as ones outlined in section 5.2. The management UI, through the NS manager, consumes information from the repository in order to enable a human administrator to have an overall view of the topology and the different resources available in the NFVI-PoP. Figure 6-1 shows an example of the infrastructure repository visualisation through the management UI.



Infrastructure Repository Resource Overview

- Demokritos Datacenter
- Intel Ireland's Leixlip Campus, Kildare, Ireland
  - service
    - neutron
    - nova
    - neutron-openvswitch-agent
    - neutron-dhcp-agent
    - neutron-metadata-agent
    - heat
    - cinder
    - glance
    - neutron-l3-agent
    - nova-cert
    - nova-conductor
    - nova-scheduler
    - nova-compute
    - nova-consoleauth
    - hypervisor
    - cinder-volume
  - physical

attributes	category	id	name	openstack_uuid	timestamp
Open dialog	storage	7437	Forward_snap	1c5b0df9-a79a-4a1f-ab8a-8ec83444f5da	1424777630.410861
Open dialog	storage	7436	cirros-0.3.2-x86_64-uec-kernel	51004178-7163-4f2d-bf70-50b6251ff75d	1424777630.410861
Open dialog	storage	7435	cirros-0.3.2-x86_64-uec-ramdisk	76b3a911-224c-494c-be9b-ffbf62fb20a4	1424777630.410861
Open dialog	storage	7434	cirros-0.3.2-x86_64-uec	e85acf3a-f083-437a-8dd4-ce4efefc2ea2	1424777630.410861

Figure 6-1 Integration repository in T-NOVA Orchestrator management UI

6.3. Orchestration Layer Interfaces

The interfaces exposed by the middleware layer are considered to be internal to the T-NOVA Orchestration layer. The basic features of the interfaces which need to be considered by the middleware API's in the context of the general goals of Orchestration layer's internal interfaces are as follows:

- Flexibility to enable new information resources to be defined for the NFVI.
- Low latency, to minimise the response times to queries from Orchestrator components.
- Scalability to support multi NFVI-PoP's
- Resiliency to (infrastructure) failure or performance degradation (due to failure or overload),

The implementation of the middleware layer considered these goals in the design and implementations of the API's. As previously outlined the middleware layer OCCI compliant REST API's makes that information (JSON over HTTP) available to T-NOVA Orchestrator components. The middleware layer API's is compliant to the general design approach for the Orchestration layer interfaces i.e. Northbound, Southbound and internal which are based on either Web Service SOAP- or REST, while data is structured in either XML or JSON formats.

6.4. Service Visualisation Module

T-NOVA Service Visualisation module is part of the T-NOVA service orchestration solution. It provides insights to network management and assurance operations across most of the segments/parts of a network service. This main functions provide by the module include a common inventory, and monitoring dashboard which shows the status of the network build and highlights possible alarms and key faults, with the ability to localise faults/alerts to a specific location. It runs as an independent modular microservice inside the orchestrator «eco-system» and scales up/down based on the network topology and number of network services that need to be visualised. The visualisation module uses the infrastructure repository as a key source of input data with integration provided via the repositories middleware API layer.

6.4.1. Features and Benefits

The Service Visualisation module provides the following features:

- *RESTful API,* supporting integrated into the Orchestrator's management user interface.
- Custom based views – Zoomable maps are supported, based on Google maps technology, to display end-to-end topology of the network service.
- *Performance and Scalability* - The microservice software architecture utilised provides robust, scalable and configurable software, to support large networks and a significant number of transactions per minute.

6.4.2. Service Visualisation Architecture

Figure 6-2 shows the internal architecture of Service Visualisation microservice. The module consists of the following main components:

1. Northbound Service Visualisation RESTful API – A RESTful API is provided to support integrated with the T-NOVA orchestration ecosystem.
2. Service Visualisation Adapter component – It converts specific network service data, to the appropriate data format required to support visualisation of the network service.
3. Management Functions component - Through the management functions, you can define the authentication/authorisation strategy, scaling metrics and configuration of the RESTful clients, in order to retrieve required information, from other orchestration layer modules. The Management Functions can be configured, using T-NOVA management UI, through the RESTful API of the management console.

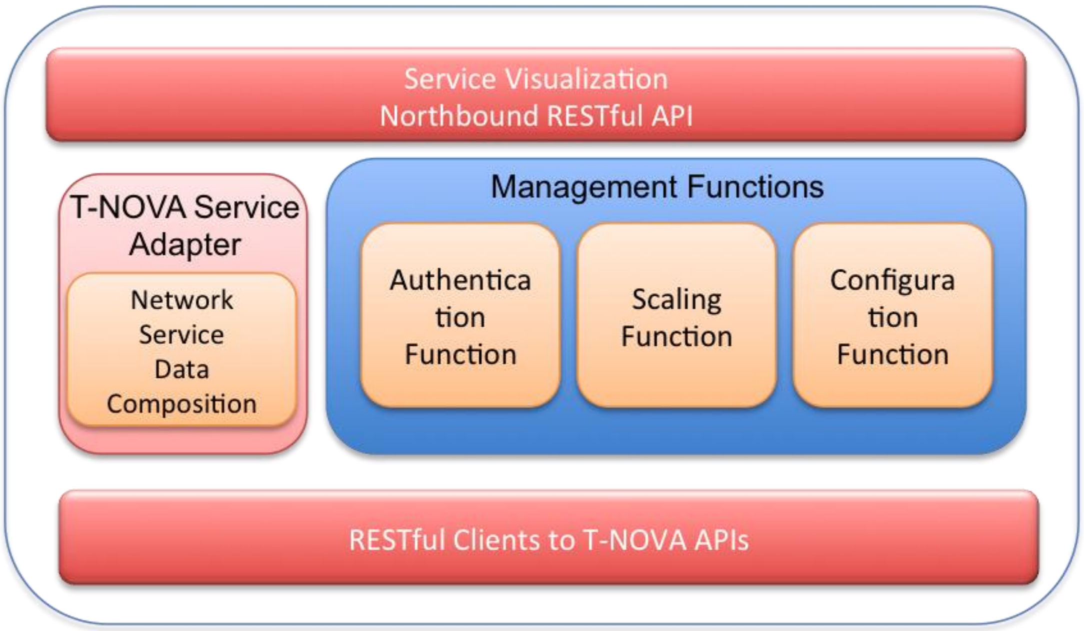


Figure 6-2 Service Visualisation Module Architecture

6.5. Gatekeeper Integration

Gatekeeper is the T-Nova identity/authorisation micro-service that allows simple actions such as user registration/authentication and inter-services authorisation based on tokens. It exposes a set API as outlined in Table 6.2.

Table 6.2 Gatekeeper API

URI	Headers	HTTP Type	Body	Purpose
/	None	GET	None	Discovery API, list of supported URIs
/admin/user/	X-Auth-Token, X-Auth-Uid	GET	None	List of registered users
/admin/user/	X-Auth-Token, X-Auth-Uid	POST	Required	New user registration endpoint
admin/user/{user-id}	X-Auth-Token, X-Auth-Uid	GET	None	Details on the user
/admin/user/{user-id}	X-Auth-Token, X-Auth-Uid	PUT	Required	Update an existing user
/admin/user/{user-id}	X-Auth-Token, X-Auth-Uid	DELETE	None	Delete an existing user
/auth/{user-id}	X-Auth-Password	GET	None	Authentication request, list of any valid token(s) are returned
/token/	X-Auth-Uid, X-Auth-Password or X-Auth-Token	POST	None	Generate a new token.
/token/{token-uuid}	X-Auth-Uid, X-Auth-Token	GET	None	Details of an existing token.

/token/{token-uuid}	X-Auth-Uid, X-Auth-Token	DELETE	None	Revoke an existing token.
/token/validate/{token-uuid}	X-Auth-Service-Key or X-Auth-Uid	GET	None	Validate an existing token.

To integrate the middleware layer APIs with Gatekeeper, an API Proxy was implemented. For authentication purpose, two new headers are added to requests to the middleware layer. The two headers are X-Auth-uid and X-Auth-Token. They are used to authenticate the request using the Gatekeeper call GET /token/validate/{token-uuid}. The steps required to authenticate requests are shown in Figure 6-3.

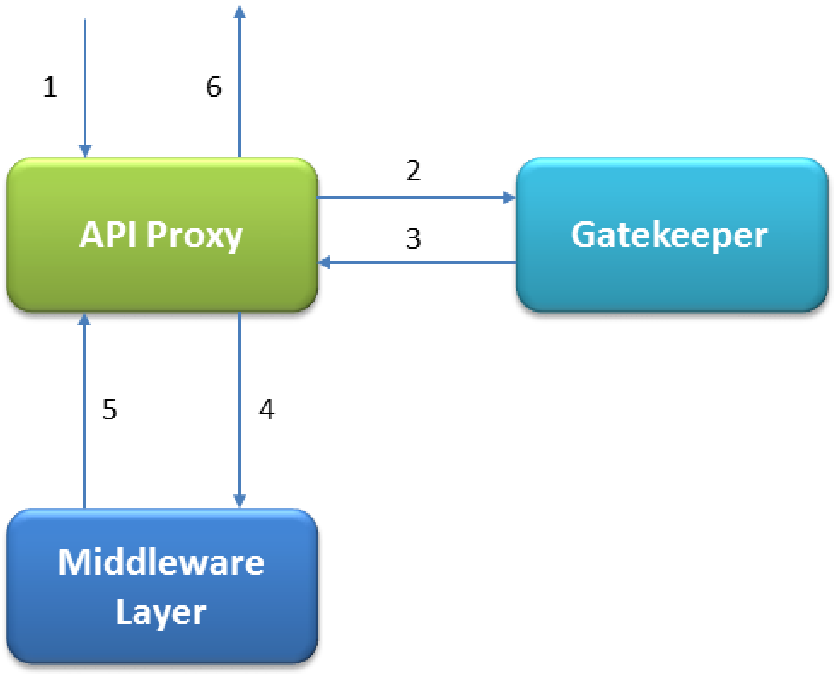


Figure 6-3 Middleware Layer - Gatekeeper integration

For example if a user wants to request the list of PoPs the user (USER_UUID) must be registered with Gatekeeper and possess a valid token (TOKEN_UUID).

As outlined in Section 4.5.1 the request to the Middleware layer, without authentication mechanism, would as follows:

```
GET http://middleware_url/pop/
--header "Accept: application/occi+json"
```

Using the API proxy the request becomes (steps numbers relate to Figure 6-1):

```
(1)
GET http://api_proxy_url/pop/
--header "Accept: application/occi+json"
--header "X-Auth-Uid:USER_UUID"
--header "X-Auth-Token:TOKEN_UUID"
```

Internally the API Proxy calls Gatekeeper to validate the token:

```
(2)
GET http://<gatekeeper_url>/token/validate/TOKEN_UUID
--header "Accept: application/occi+json"
--header "X-Auth-Uid:USER_UUID"
--header "X-Auth-Token:TOKEN_UUID"
```

Gatekeeper's response (3.) will be:

- 200 if the Token is valid;
- 406 if the token is not valid.

In the case of invalid token all following steps will be skipped and Gatekeeper's response will be provided to the user (6.)'

In the case of valid token the API Proxy will continue serving the request redirecting the call to the middleware layer:

```
(4)
GET http://<middleware_url>/pop/
--header "Accept: application/occi+json"
```

The Middleware layer's response (5.) will be provided to the user (6.) The same process is repeated for each call to the middleware layer. The Middleware layer is configured to respond only if requests come from the API Proxy, ensuring that all the calls are authenticated before the action.

7. Infrastructure Repository Distribution Package

As outlined in section 6.2 the T-NOVA Orchestration layer is being developed using a microservices architecture. The application comprises of a suite of small services, each running in its own process and communicating using a HTTP resource API.

The common requirements of the microservices are:

- They are relative small.
- They are independently deployable to improve fault isolation
- They are written using a variety of languages, frameworks, and framework versions.
- They can be composed by multiple service instances for throughput and availability purposes.
- Services are scalable.
- Services are isolated from one another.
- The manager should be able to constrain the resources (CPU and memory) consumed by a service.
- The manager needs to monitor the behaviour of each service instance
- Services should be deployable as cost-effective as is possible.

To satisfy these requirements a deployment a solution based around a container image (Docker) for infrastructure repository was developed. This approach supports the deployment of each service instance as a container. The benefits of this approach include:

- Agile service scale up/down if service by changing the number of running container instances.
- The container encapsulates the details of the technology used to build the service. All services are, for example, started and stopped in exactly the same way.
- Each service instance is isolated
- A container imposes limits on the CPU and memory consumed by a service instance
- Containers are extremely fast to build and start. For example, it's 100x faster to package an application as a Docker container than it is to package it as an AMI. Docker containers also start much faster than a VM since only the application process starts rather than an entire OS.

The components encapsulated in containers are as follows:

- The EPA controller and Listener services are packaged in a single container.
- The middleware database is packaged in a container
- The middleware layer is packaged in a container
- The resource repository database is packaged in a container.
- The API Proxy for Gatekeeper integration is packaged as a container

For each container a Dockerfile is provided to set up the components for a given deployment. The main parameters that an administrator installing the repository subsystem needs to provide when configuring each component are:

For the EPA controller:

- OpenStack controller IP
- OpenStack controller hostname
- OpenStack MySQL DBs credentials
- RabbitMQ broker credentials and endpoint (Host and Port)
- EPA database endpoint and credentials
- PoP ID (e.g. IR_LEX_001)

For the Middleware Layer:

- PoP database endpoint and credentials
- Middleware layer port

For the PoP database

- Neo4j credentials

For the EPA DB

- Neo4j credentials

For the API Proxy

- Gatekeeper endpoint
- Middleware layer endpoint

The agent is distributed as a Python application that needs to be installed on each compute node of a NFVI PoP. It is configured to run at boot time, and optionally it can be scheduled to run periodically as described in Section 3.4.1.1. The configuration parameters required by the agent are:

- EPA Controller IP
- EPA Controller ssh credentials
- RabbitMQ broker credentials and endpoint (Host and Port)

Further details on the installation of the repository subsystem components can be found in the installation guide.

8. Conclusions

The infrastructure repository is a key subsystem of the T-NOVA Orchestrator. It provides a rich set of infrastructure related information from the software and hardware components that comprise the NFVI and VIM of the T-NOVA IVM layer. Analysis of the software components selected to implement the VIM (namely OpenStack and OpenDaylight) revealed that limited infrastructure related information was available. Different options were identified and analysed to implement the repository. One option was selected for prototype implementation, in order to test the feasibility of the general design and act as vehicle to test fulfilment of requirements identified in WP2 plus identifying new requirements based on discussions with tasks dependent on the resource repository.

Using the learnings from the prototype implementation and the complete set of requirements captured, the final design of the infrastructure repository subsystem was developed. The design comprises 5 key functional components. The first component is an enhanced platform awareness agent which runs on the compute nodes and collects platform specific information. The second component is the listener services. One listener is dedicated to handling messages generated by the EPA agent, and the second one is dedicated to OpenStack related messages. The third component is the EPA controller which coordinates with listener services to process and persist updates to the repository database from files received from the EPA agents or OpenStack infrastructure landscape change notifications. The fourth component is the infrastructure repository database which is responsible for storing the infrastructure related information and the relationships between the stored information. The database was implemented as a graph database in order to support the encoding of the relationships between the components of the NFVI. This approach also provided a convenient mapping of the systems structures within the NFVI and node structures of a graph database. The final component is a middleware API layer which provides a common REST based interface to Orchestrator components that want to retrieve information from the repository. This REST interface was implemented in an OCCI compliant manner to provide abstraction from the underlying implementation. The middleware layer also features a database to support the storage of NFVI PoP ingress and egress endpoints and associated parametric data for the links. The design and implementation of the middleware layer API's also supports multiple NFVI-PoP infrastructure repository database instances thus allowing the subsystem to scale across multiple PoP as necessary.

A number of sample visualisation use cases based the information which is stored in the repository database were developed. The use cases focused on scenarios where the resource information stored in the repository subsystem could be used to support specified problems or operational needs in a value added manner. Visualisation of the repository information will be functionality incorporated into the management UI of the T-NOVA Orchestration layer. Additionally the Service Visualisation Module a component within the broader orchestration ecosystem will also use information from the infrastructure repository to support network specific visualisation.

All components have been successfully implemented and integrated to deliver a fully functional infrastructure repository subsystem. A Docker container based approach was been adopted for packaging and distribution of the repository subsystem components. This package will be used to support deployments activities required in Task 4.5 and WP7.

9. List of Acronyms

Acronym	Explanation
API	Application Program Interface
DAG	Directed Acyclic Graph
DB	Database
CPU	Central Processing Unit
EPA	Enhanced Platform Awareness
GPU	Graphics Processor Unit
HTTP	Hypertext Transfer Protocol
HW	Hardware
IaaS	Infrastructure as a Service
IVM	Infrastructure virtualisation and management
JSON	JavaScript Object Notation
NAT	Network Address Translation
NFVI	Network Function Virtualisation Infrastructure
NIC	Network Interface Card
NS	Network Service
NUMA	Non-uniform memory access
OCCI	Open Cloud Compute Interface
ODL	OpenDaylight (SDN Controller)
PCIe	Peripheral Component Interconnect Express
PoP	Point of Presence
REST	Representational State Transfer
SDN	Software Defined Networking
SR-IOV	Single Root Input/Output Virtualisation
SSH	Secure Shell
UI	User Interface
URL	Uniform Resource Locator
UUID	Universally unique identifier (
vCPU	Virtual Central Processing Unit
vNIC	Virtual Network Interface Card

VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtualised Network Function
WAN	Wide Area Network

10. References

[1] A. Edmonds, T. Metsch, A. Papaspyrou, and A. Richardson, "Toward an Open Cloud Standard," *Internet Computing, IEEE,* vol. 16, pp. 15-25, 2012.

[2] OCCI. (2015). *The Open Cloud Computing Interface*. Available: http://occi-wg.org/

[3] C. Larman, "Iterative & Evolutionary," in *Agile & Iterative Development - A Managers Guide*, ed Boston, MA, USA: Addison-Wesley Professional 2004, pp. 9-24.

[4] L. Efremova and D. Andrushko. (2015, 22nd June). *What's In OpenDaylight*. Available: https://www.mirantis.com/blog/whats-opendaylight/

[5] Intel, "OpenStack Enhanced Platform Awareness," 2015.

[6] A. Sagar. (2013). *What is the advantage of using a graph database over a relational database for recommendations?* Available: http://www.quora.com/What-is-the-advantage-of-using-a-graph-database-over-a-relational-database-for-recommendations

[7] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: a data provenance perspective," presented at the Proceedings of the 48th Annual Southeast Regional Conference, Oxford, Mississippi, 2010.

[8] Open-MPI. (2015). *Portable Hardware Locality (hwloc)*. Available: http://www.open-mpi.org/projects/hwloc/

[9] B. Goglin, "Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc)," presented at the International Conference on High Performance Computing & Simulation (HPCS 2014), Bologna, Italy, 2014.

[10] Python. (2012). *Python Decorators*. Available: https://wiki.python.org/moin/PythonDecorators

[11] A. Edmonds, T. M. Bohnert, T. Metsch, P. Harsh, G. Carella, L. Ferreira, A. Gomes, G. Katsaros, S. Khatibi, A. Marcarini, J. Muller, N. Nikaein, S. Ruffino, S. Ruiz, and G. Toffetti, "Final Overall Architecture Definition, Release 2", 2015.