Grant Agreement No.: 604590
Instrument: Large scale integrating project (IP)
Call Identifier: FP7-2012-ICT-FI

eXperimental Infrastructures for the Future Internet

D3.7: XIFI Infrastructure Network Adaptation Mechanisms API v2

Work package	WP 3
Task	Task 3.1
Due date	31/12/2014
Submission date	09/03/2015
Deliverable lead	Telefónica I+D (TID)
Authors	Luis M. Contreras (TID), Giuseppe Cossu (CNET), Matteo Gerola (CNET), José I. Aznar (I2CAT), Jose Gonzalez (UPM)
Reviewers	Sean Murphy (ZHAW) , Sergio Morant (ILB)

Abstract	This document updates the description enclosed in M12's Deliverable 3.4 with the latest developments for the XIFI Network Controller.

Keywords	Network Controller, Orchestrator, SDN, QoS

Document Revision History

Version	Date	Description of change	List of contributor(s)
V1.0	16/02/2015	Document creation	Luis M. Contreras (TID)
V1.1	09/03/2015	Internal revision	Luis M. Contreras (TID), Jose Gonzalez (UPM)

Disclaimer

This report contains material which is the copyright of certain XIFI Consortium Parties and may only be reproduced or copied with permission in accordance with the XIFI consortium agreement.

All XIFI Consortium Parties have agreed to publication of this report, the content of which is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License1.

Neither the XIFI Consortium Parties nor the European Union warrant that the information contained in the report is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using the information.

Copyright notice

© 2013 - 2015 XIFI Consortium

Project co-funded by the European Commission in the 7th Framework Programme (2007-2013)		
Nature of the Deliverable:	P (Prototype)	
Dissemination Level		
PU	Public	✓
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to bodies determined by the XIFI project	
CO	Confidential to XIFI project and Commission Services	

[1] http://creativecommons.org/licenses/by-nc-nd/3.0/deed.en_US

EXECUTIVE SUMMARY

Deliverable D3.7 updates and completes the description enclosed in M12's Deliverable 3.4 [13] with the latest advances related to the XIFI Network Controller. This controller is in charge of creating the connectivity between the virtual resources deployed by the end user in the XIFI infrastructure federation. This network controller orchestrates the required to provide end-to-end connection.

The different components that have been fulfilled within WP3's Task 3.1 are detailed in this documentation, providing the vision about basic functionality, usage and installation guidelines. Those changes and enhancements with respect to the first version are reflected here, providing the basis for future steps.

TABLE OF CONTENTS

EXECUTIVE SUMMARY 3

TABLE OF CONTENTS 4

LIST OF FIGURES 7

LIST OF TABLES 8

ABBREVIATIONS 9

1 INTRODUCTION 10

1.1 Scope 10

1.2 Intended audience 10

1.3 Reading suggestions 11

2 ABNO MODULE 12

2.1 Summary 12

2.2 Updates to ABNO Module from previous release 12

2.2.1 VM of-ports in local OpenStack instance 12

2.2.2 GRE tunnel of-ports in local OpenStack instance 12

2.2.3 Integration in ABNO of the scripts to retrieve of-port information 13

2.2.4 Main interactions 13

2.3 Installation Manual 15

2.3.1 Software Repository 15

2.3.2 Setup Guidelines 15

2.4 User Manual 18

2.4.1 API Specification 18

3 OPENNAAS MODULE 20

3.1 Summary 20

3.2 OpenNaaS enhancements for XIFI project 21

3.2.1 Identified OpenNaaS limitations during XIFI project 21

3.2.2 OpenNaaS general enhancements 21

3.3 OpenNaaS interfaces development: 22

3.3.1 OpenNaaS Northbound Interface: 22

3.3.2 OpenNaaS Southbound Interface: 23

3.3.3 OpenNaaS interface to XIFI monitoring module 23

4 RYU CONTROLLER 25

4.1 Setup guidelines 25
4.2 Ryu rules injection 25

5 QOS MECHANISMS IN OPENSTACK NETWORKS 26
5.1 Summary 26
5.2 Component Responsible 26
5.3 Design choices 26
5.4 System Architecture 28
5.5 QoS Neutron API 30
5.6 QoS Neutron Database 31
5.7 Installation and configuration procedure 32

6 SHORT TERM EVOLUTION 34
6.1 Development of capabilities internal to the controller 34
6.1.1 OpenNaaS integration 34
6.2 Integration with elements external to the controller 34
6.2.1 Connection to MD-VPN transport infrastructure 34
6.2.2 GRE Tunnels 35

7 CONCLUSIONS 37

REFERENCES 38

APPENDIX A INTERACTION WITH RYU AS LOCAL SDN CONTROLLER 39

APPENDIX B MONITORING MODULES 40
B.1 Summary 40
B.2 Component Responsible 40
B.3 Motivation 41
B.4 User Stories Backlog 41
B.5 Architecture Design 42
B.5.1 The main component of OpenFlow controller (in ryu/controller subfolder): 42
B.5.2 The Ryu libraries (in ryu/lib subfolder) provide: 43
B.5.3 The Third party libraries (in ryu/contrib subfolder) are composed of: 43
B.6 Basic actors 44
B.7 Main interactions 44
B.8 Release Plan 46
B.9 Installation Manual 46
B.9.1 Installation Requirements 46
B.9.2 Software Repository 46

© 2013 - 2015 XIFI Consortium Parties

B.9.3 Setup Guidelines 46
B.10 User Manual 48
B.10.1 API Specification 49

LIST OF FIGURES

Figure 1: XIFI Network Controller architecture 10
Figure 2: Workflow detailing ABNO components interactions 13
Figure 3: Screenshot for point-to-point connectivity request 19
Figure 4: Screenshot for point-to-point connectivity acknowledgment 19
Figure 5: Communication between the XIFI Network Controller and the different nodes 21
Figure 6: OpenStack Data Network 28
Figure 7: Neutron Private Network 29
Figure 8 - QoS policy applied per port 29
Figure 9: Neutron and Ryu Plugin Cooperation 30
Figure 10: QoS module's new database tables 32
Figure 10: GRE Tunnels 35
Figure 11: Deployment of GRE tunnels 36

LIST OF TABLES

Table 1: Component Responsible 26
Table 2: QoS policies 28
Table 3: New API calls added in Neutron to support Quality of Service 31

ABBREVIATIONS

ABNO	Application-Based Network Operations
API	Application Program Interface
DC	Data Centre
GE	Generic Enabler
GRE	Generic Routing Encapsulation
L2-VPN	Layer 2 VPN
L3-VPN	Layer 3 VPN
MAC	Media Access Control address
MD-VPN	Multi Domain Virtual Private Network
NaaS	Network as a Service
NBI	NorthBound Interface
NREN	National Research and Education Network
OF	Open Flow
PCE	Path Computational Element
PCEP	Path Computational Element Protocol
PN	Programmable Network.
PoC	Proof of Concept
QoS	Quality of Service
REST	Representational state transfer
SBI	SouthBound Interface
SDN	Software-Defined Networking
UC	Use Case
VIF	Virtual Network Interface
VM	Virtual Machine
VPN	Virtual Private Network

© 2013 - 2015 XIFI Consortium Parties

1 INTRODUCTION

1.1 Scope

This deliverable updates D3.4 [13] with regard to the XIFI Network Controller, including its main components and the interfaces existing among them.

The XIFI Network Controller plays the role of a network orchestrator instructing local SDN controller per domain in order to achieve end-to-end connectivity. Here it is briefly introduced for completeness.

The XIFI Network Controller has been designed combining both the ABNO (Application Based Network Orchestration) [1] and OpenNaaS [7] architectures. Figure 1 presents the modular architecture of the XIFI Network Controller including the NaaS component (green colour) and the specific components of the ABNO framework (red colour) considered within the scope of XIFI. From the whole set of functional components proposed in ABNO, only some of them are needed.

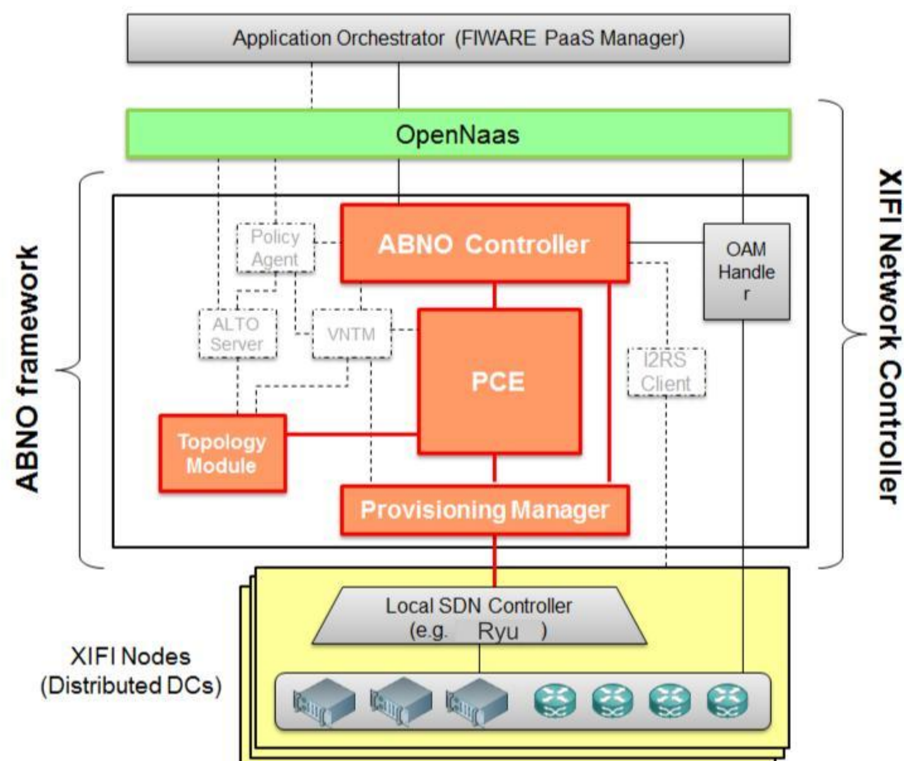


Figure 1: XIFI Network Controller architecture

1.2 Intended audience

The target audience of this deliverable is:

- The nodes of the XIFI federation (including original and new nodes).
- Experts and technical personnel providing deployment support and end-user support activities. These activities will be fulfilled by the support entity of the XIFI federation.

1.3 Reading suggestions

The document is divided into six chapters. They report the status of the XIFI Network Controller and describe the APIs and the interfaces between the different software modules.

Chapter 1: Introduction

- Provides the information required to understand the rest of the document

Chapter 2: Update on ABNO module

- Update of the ABNO component pertaining to the XIFI Network Controller

Chapter 3: Update on NaaS module

- Update on the NaaS component used for the first public release of the XIFI Network Controller

Chapter 4: QoS mechanisms

- Description of the QoS mechanisms implemented in the XIFI Network Controller

Chapter 5: Situation at M22

- Reports the status at M22 of the XIFI Network Controller.

Chapter 6: Test and use cases in progress

- Introduces the progress of the XIFI Network Controller in the short term for both the internal development of the controller and the external integration with other components of XIFI.

© 2013 - 2015 XIFI Consortium Parties

2 ABNO MODULE

2.1 Summary

This component composes the end-to-end connectivity by instructing local SDN controllers in each of the XIFI nodes to populate forwarding rules on the OpenFlow (OF) [6] enable switches in a coordinated way, thus setting up the connectivity path (stitching the local connectivity to the transport capabilities in the WAN).

The XIFI project is a representative of an SDN controlled federation of distributed data centres. The delivery of distributed cloud services implies the configuration of various capabilities across the networks involved in the service (both in the distinct data centres and in the WAN domains connecting them). Within a data centre, the configuration of a cloud service is done by using cloud management software to create virtual machines (VMs). Once the VMs are created in each of the data centres and are connected internally, a network connectivity service has to be invoked for connecting the overlay networks through the WAN. The Open vSwitch (OVS) [5] routers to be controlled are those ones deployed by OpenStack on the compute nodes within each infrastructure.

The architecture requires a logically centralized entity maintaining a comprehensive view of all the network resources available in order to orchestrate them. The ABNO component plays such role.

2.2 Updates to ABNO Module from previous release

In order to obtain the specific ports on the Open vSwitch connecting each VM in each instance of OpenStack (in the separated data centres) and the ports of the GRE Tunnel defined for end-to-end connection via MD-VPN, some scripts has been developed to facilitate this work. The commands used in these scripts will be explained below.

2.2.1 VM of-ports in local OpenStack instance

The first step is to know and to obtain the identifier of the VM from the VM name.

```
nova list | grep $VM_name | grep -o '\<[0-9a-z\.-]*\>' | head -1
```

With this VM identifier, it can be discovered the port identifier in which the VM is connected.

```
neutron port-list --device_id "$id_VM" | grep -o '\<[0-9a-z\]*\>' | head -3
| grep -o '\<........\>'
```

Then, it will be necessary this command line request in order to obtain the name of the correspondent port.

```
ifconfig | grep "$id_port_VM" | grep qvo | grep -o '\<[0-9a-z\.-]*\>' |
head -1
```

Finally, with the previous "qvo_name" of the port, it can be found the port of the VM wanted.

```
sudo ovs-vsctl list Interface "$qvo_VM" | grep ofport | grep -o '\<[0-9]\>'
```

2.2.2 GRE tunnel of-ports in local OpenStack instance

Once we have VM ports, it is possible to obtain the GRE Tunnel port by the following command.

```
sudo ovs-vsctl list Interface $GRE_name | grep ofport | grep -o '\<[0-9]\>'
```

2.2.3 Integration in ABNO of the scripts to retrieve of-port information

All this process to obtain the VM and GRE tunnel of-ports is integrated within the ABNO software; more specifically affects the ABNO Controller module.

There is a specific workflow for this task which launch a *ssh* connection and execute the following command in the remote machine in order to obtain the desired of-port identifier. The parameter "param" refers to the VM o GRE tunnel for which the of-port is needed.

```
sudo ovs-vsctl list Interface "+param+"| grep ofport | head -1 | tr -d
[ofport] | tr -d [:punct:] | sed 's/[[:space:]]*$//' | sed
's/^[[:space:]]*//'
```

In the future the access to obtain the of-port identifier will require a token authentication obtained via Keystone OpenStack [4] component in order to avoid the direct *ssh* connections.

2.2.4 Main interactions

The following diagram (Figure 2) presents the interactions between the actors of the ABNO component. ABNO receives the request from OpenNaaS, and then process the request through the different ABNO components and instructs the Ryu [11] rules as can be seen in the following figure.

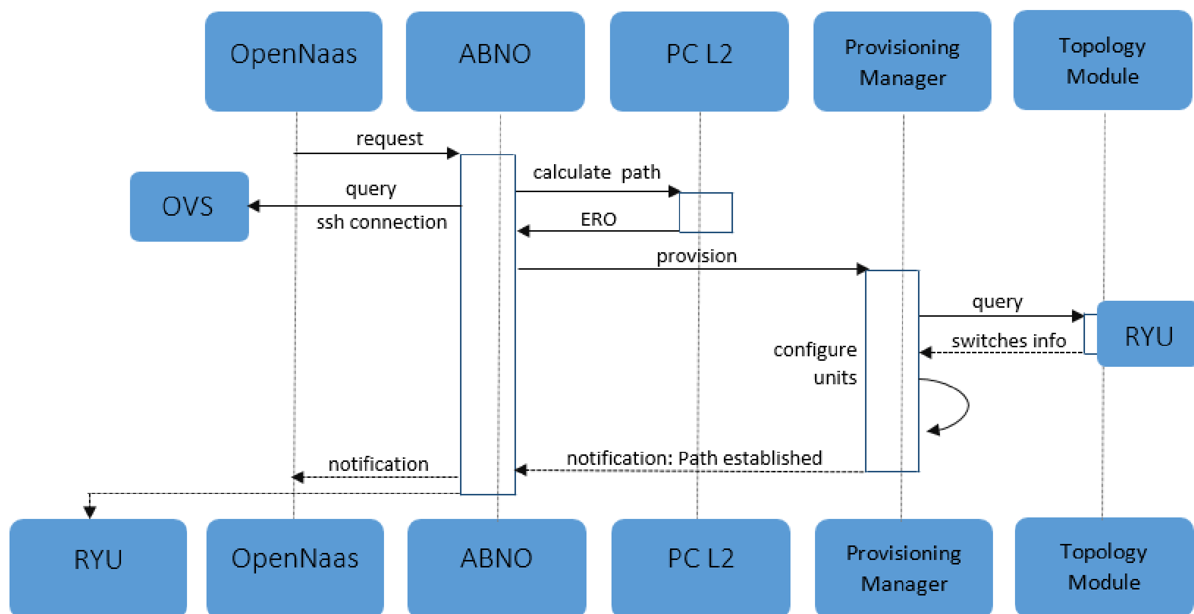


Figure 2: Workflow detailing ABNO components interactions

The following steps are required to provision a point-to-point link using the ABNO:

- OpenNaaS sends the required information to the ABNO controller including the source and destination regions, the source and destination MACs and id port of the VM through which the hosts are attached to the switch.

  - This interface to the ABNO controller is, for example, assumed to be an HTTP/JSON interface.
- When the ABNO controller receives the request, it obtains the information required through configuration files. With this information establish a ssh connection in order to obtain the OVS of-ports.
  - The configuration files are source and destination regions.
  - The command to obtain the of-ports is the mentioned in the of-ports integration in ABNO.
- ABNO asks the PCE [1] for a path between the two hosts.
  - PCEP is used in this interface.
  - Some extensions are required to the current protocol definitions so some information such as the switch identifiers which are made of 10 bytes are coded into newly defined ERO sub-objects.
- The PCE replies to the ABNO controller with a PCEP response.
  - The PCEP protocol is extended in order to fill the PCEP response.
- The ABNO controller creates a PCEP initiate packet from the response and sends it to the Provision Manager (PM).
  - This step is done using PCEP.
  - The ERO inside the Initiate is the same as the one inside the response so the PCEP protocol is extended in the same way as in the previous step.
- The Provisioning Manager queries the Topology Module for the description of each node.
- Depending on the technology and configuration mode selects a different protocol to complete the request. OF controller is queried.
  - This time technology will always be OF but the configuration mode can vary from node to node
- OF controller configures OF switches.
- Once the path has been established it notifies the ABNO controller.
  - This interface is PCEP.
- Similarly, the ABNO controller advertises OpenNaaS if everything is correct or there is a problem. ABNO also instructs Ryu flows.
  - This interface is JSON.
  - The answers to OpenNaaS follow these structures.
    i. Code: 400. SC_BAD_REQUEST in case OpenNaaS sends incorrect parameters to ABNO.
    ii. Code: 500. SC_INTERNAL_SERVER_ERROR in case ABNO was not able to establish the path or to instruct the Ryu flows. An example can be: “No Open VSwitch available. Not possible to provide a path"

iii. Code: 200. SC_OK in case everything goes well.

The mentioned OF controller refers to the local SDN controller deployed in the XIFI node (Ryu controller in XIFI).

2.3 Installation Manual

2.3.1 Software Repository

For the time being, the software will be delivered via a Google Drive file. It is foreseen to host the software in a Git repository once the software is available as open source software.

A public Google Drive link has been created from where the component can be downloaded:

https://drive.google.com/file/d/0B3en-dR7u9UPd3VYYnRpRHdid0k/view?usp=sharing

2.3.2 Setup Guidelines

This documentation assumes the file README.md inside the root of the TID installation package has also been read.

- Install Java JRE environment (mandatory to ensure proper installation):

```
http://www.liberiangeek.net/2012/04/install-oracle-java-runtime-jre-7-in-ubuntu-12-04-precise-pangolin/
```

- OpenStack instances are supposed to be deployed via XIFI project resources, even though Devstack deployment has been used for testing process.

  An example of localrc file is shown for completeness:

```
SERVICE_HOST=10.0.35.1
HOST_IP=10.0.35.1
disable_service n-net
enable_service neutron q-svc q-agt q-l3 q-dhcp q-meta ryu n-cpu n-api
FLOATING_RANGE=10.0.35.0/24
PUBLIC_NETWORK_GATEWAY=10.0.35.1
#FIXED_RANGE=192.168.0.0/24
#FIXED_NETWORK_SIZE=256

Q_PLUGIN=ryu
#Q_AGENT_EXTRA_AGENT_OPTS=(tunnel_type=gre)
#Q_AGENT_EXTRA_OVS_OPTS=(tenant_network_type=gre)
#Q_SRV_EXTRA_OPTS=(tenant_network_type=gre)

ENABLE_TENANT_TUNNELS=True
```

```
Q_HOST=$SERVICE_HOST
MYSQL_HOST=$SERVICE_HOST
RABBIT_HOST=$SERVICE_HOST
GLANCE_HOSTPORT=$SERVICE_HOST:9292
KEYSTONE_AUTH_HOST=$SERVICE_HOST
KEYSTONE_SERVICE_HOST=$SERVICE_HOST
RYU_API_HOST=$SERVICE_HOST
RYU_OFP_HOST=$SERVICE_HOST

MYSQL_PASSWORD=admin
RABBIT_PASSWORD=admin
SERVICE_TOKEN=admin
SERVICE_PASSWORD=admin
ADMIN_PASSWORD=admin

RYU_APPS=ryu.app.gre_tunnel,ryu.app.quantum_adapter,ryu.app.rest,ryu.app.re
st_conf_switch,ryu.app.rest_tunnel,ryu.app.tunnel_port_updater,ryu.app.rest
_quantum,ryu.app.qos_cn

# Images
IMAGE_URLS+=",https://launchpad.net/cirros/trunk/0.3.0/+download/cirros-
0.3.0-i386-disk.img"

# Branches
KEYSTONE_BRANCH=stable/icehouse
NOVA_BRANCH=stable/icehouse
GLANCE_BRANCH=stable/icehouse
CINDER_BRANCH=stable/icehouse
HORIZON_BRANCH=stable/icehouse
NEUTRON_BRANCH=stable/icehouse

NEUTRON_REPO=https://github.com/SmartInfrastructures/neutron
NEUTRON_BRANCH=stable/icehouse
RYU_REPO=https://github.com/SmartInfrastructures/ryu.git
RYU_BRANCH=qos

NEUTRONCLIENT_REPO=https://github.com/SmartInfrastructures/python-
neutronclient.git
NEUTRONCLIENT_BRANCH=qos
LIBS_FROM_GIT=python-neutronclient
```

© 2013 - 2015 XIFI Consortium Parties

The password should be replaced with the one you choose.

- Launch ABNO service

Run these commands:

```
./unstack.sh

./stack.sh
```

- Configure `init_setup_2.sh`, e.g.,

```
STACK_DIR=devstack
ABNO_DIR=ABNO
ASO_DIR=pip
PORT=eth0
LOG_FILE=setup.log
KILL_TAG=quantum-server
```

You should generally only need to change the PORT variable. This is the interface through which the physical host will be connected to other OpenStack instances, not the management IP of the physical host.

- Run Ryu service: run Ryu after configuring `init_setup_2.sh`, e.g.,

```
./init_setup_2.sh -RYU
```

- Checking of modules

And check if it is running and the IP of the controller with

```
ps aux | grep ryu

sudo ovs-vsctl get-controller br-int
```

- Configuration of local Open vSwitch instance to be controlled by local Ryu controller

```
sudo ovs-vsctl set-controller br-int  tcp:127.0.0.1:663
```

If you want to see the output of Ryu execution visit the following file:

```
vi nohup.out
```

- Configuration files for ABNO

Before run ABNO, config files must be correctly configured. Be sure the files

ABNO/scenarios/XIFI_scenario/nodeConfig/regionX.xml

are correctly updated before installation with the appropriate information of the controllers in each region, as well as GRE tunnel information. There will be a file per region following this structure:

```
<?xml version="1.0" encoding="UTF-8"?>

<Madrid>

        <Region>

                <ip>10.0.78.2</ip>

                <port>8080</port>
```

```
                <type>RYU</type>
        </Region>
        <Destination>
                <dest_region>Trento</dest_region>
                <tunnel>gre0</tunnel>
        </Destination>
</Madrid>
```

After that, you can run the ABNO:

```
./init_setup_2.sh -ABNO
```

2.4 User Manual

2.4.1 API Specification

Create unicast path

	URI
GET	`source_region=<?>&destination_region=<?>&source_mac=<?>&dest_mac=<?>&source_interface=<?>&destination_interface=<?>&operation_type=<?>&id_operation'`

Action:

Creates a unicast unidirectional link between two hosts identified by their MAC addresses.

Parameter Description:

- **Source and destination Region** (mandatory). These variables contain the name of the regions where Ryu controllers are.
- **Source and destination MACs of the VMs** (mandatory): the MACs from the VMs that are the origin and the destination of the LSP.
- **Source and destination interface** (mandatory): are the id ports of the source and destination VMs, necessary to correctly create the Ryu rules.
- **Operation**: this string is mapped to the operation that the ABNO has to execute to configure the link. For this workflow the value is: "`WLAN_PATH_PROVISIONING`".
- **Operation_Type** (mandatory): this string is mapped to the workflow that the ABNO controller has to execute. For this workflow the value is: "*`XIFIWF`*".
- **ID_Operation** (not mandatory): is the identifier for the specific operation in order to trace back and reference this specific action in the network.

Response:

Returns a JSON containing the configuration parameters.

Normal Response Code: 200

Error Response Code: any other.

Example:

A request could be done in the following way:

```
curl
"localhost:4445?source_region=Madrid&dest_region=Trento&source_mac=fa:16:3e
:a6:05:ab&dest_mac=fa:16:3e:ec:e2:cb&source_interface=b5a9b842-5262-4cff-
a051-e41fefce2132&destination_interface=de10aebe-ff6f-4532-935d-
25984554106e&operation=WLAN_PATH_PROVISIONING&Operation_Type=XIFIWF&ID_Oper
ation=1234"
```

And the response could be:

```
{
"ID_Operation":"1234"
"Operation Type":"XIFIWF",
"Result": "LINK_CONFIGURED"
"Error Code": "NO_ERROR"
}
```

The following figure shows a query to the ABNO asking for this workflow to be executed.

```
550 2.458414  127.0.0.1  127.0.0.1  HTTP  434 GET /?source=10:00:2c:59:e5:66:ed:00&destination=10:00:2c:59:e5:5e:2b:00&sourc
1243 4.026092 127.0.0.1  127.0.0.1  HTTP  333 HTTP/1.1 200 OK  (text/plain)
Frame 550: 434 bytes on wire (3472 bits), 434 bytes captured (3472 bits)
Linux cooked capture
Internet Protocol Version 4, Src: 127.0.0.1 (127.0.0.1), Dst: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: 55523 (55523), Dst Port: upnotifyp (4445), Seq: 1, Ack: 1, Len: 366
Hypertext Transfer Protocol
 GET /?source=10:00:2c:59:e5:66:ed:00&destination=10:00:2c:59:e5:5e:2b:00&source_mac=aa:bb:cc:dd:ee:ff&dest_mac=ff:ee:dd:cc:cc:aa&source_interface=5&destin
 User-Agent: curl/7.22.0 (x86_64-pc-linux-gnu) libcurl/7.22.0 OpenSSL/1.0.1 zlib/1.2.3.4 libidn/1.23 librtmp/2.3\r\n
 Host: localhost:4445\r\n
 Accept: */*\r\n
 \r\n
 [Full request URI [truncated]: http://localhost:4445/?source=10:00:2c:59:e5:66:ed:00&destination=10:00:2c:59:e5:5e:2b:00&source_mac=aa:bb:cc:dd:ee:ff&dest
```

Figure 3: Screenshot for point-to-point connectivity request

Additionally, the following capture shows a response from the ABNO where the point to point path has been configured correctly.

```
550 2.458414  127.0.0.1  127.0.0.1  HTTP  434 GET /?source=10:00:2c:59:e5:66:ed:00&destination=10:00:2c:59:e5:5e:2b:00&sourc
1243 4.026092 127.0.0.1  127.0.0.1  HTTP  333 HTTP/1.1 200 OK  (text/plain)
Hypertext Transfer Protocol
 HTTP/1.1 200 OK\r\n
 Content-Type: text/plain;charset=ISO-8859-1\r\n
 Content-Length: 149\r\n
 Server: Jetty(8.y.z-SNAPSHOT)\r\n
 \r\n
Line-based text data: text/plain
 <html><body><p> {"Result":"LINK_CONFIGURED","Operation Type":"BANDWIDTH_PROVISIONING","ID_Operation":7350,"Erro Code":"NO_ERROR"} </p></body></html>\n
```

Figure 4: Screenshot for point-to-point connectivity acknowledgment

3 OPENNAAS MODULE

The OpenNaaS module is required in the XIFI Network Controller to provide connectivity service awareness across the federation. It is in charge of collecting and maintaining the information about the resources committed to the end user from the connectivity point of view, either if the resources are local to just one XIFI node or if they are spread in different nodes in the federation.

This module is used to maintain the notion of connectivity service end to end by associating VMs located in separated data centres with the required network connection among them.

3.1 Summary

OpenNaaS [7] is an open-source service and network control and management platform which provides with a suitable set of tools for the management and operation of XIFI network connectivity services.

From a conceptual perspective OpenNaaS modules manage the service logic. This means that OpenNaaS holds the service awareness abstracting the underlying network complexity. Upon users' requests, this allows the possibility to easily deploy and operate provisioning network services between XIFI federated nodes and guarantee the agreed service Quality of Service (QoS).

Thus, OpenNaaS module constitutes the entrance point of the XIFI Network Controller. It receives the service requests and turns the requirements into network comprehensible parameters, so that the ABNO is able to establish the E2E connection.

OpenNaaS as the module which maintains the service logic of the XIFI network controller: OpenNaaS platform, as part of the service logic, enables to abstract the underlying network complexity allowing the possibility to easily deploy and operate customized advanced provisioning network services according upon requests. Service cares of handling end-user requests, organize the request inputs and map them into a network request to be understood by the ABNO module. OpenNaaS monitors the connectivity service and keeps the overall service performance with guaranteed QoS.

ABNO listens to E2E service connectivity requests from the OpenNaaS [3] by means of the north bound interface and selects the appropriate workflow to follow in order to connect the required network resources in within the XIFI nodes, matching the service logic.

ABNO as the module which maintains the network logic of the XIFI Network Controller: The ABNO is responsible of the network logic orchestration for provisioning the connectivity service requests by invoking the necessary components in the right order according to specific workflows. OpenNaaS communicates with ABNO by means of a REST API.

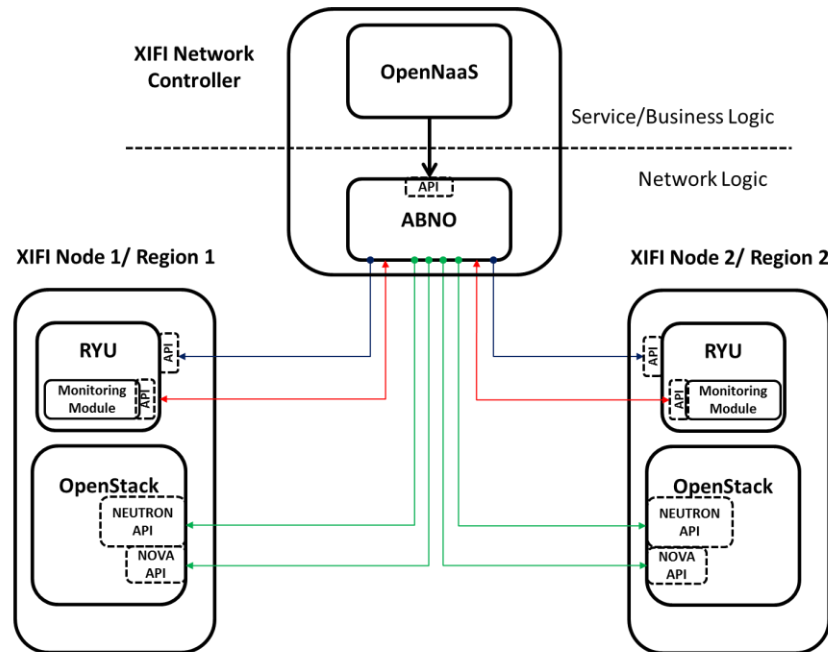


Figure 5: Communication between the XIFI Network Controller and the different nodes

The coordination between these two entities results crucial for a suitable implementation of the overall XIFI connectivity service. Remaining XIFI network controller, SW components (OpenStack APIs) are part of the network logic to achieve the network service provisioning.

3.2 OpenNaaS enhancements for XIFI project

3.2.1 Identified OpenNaaS limitations during XIFI project

In order to compose the XIFI Network Controller, several limitations were identified. Most relevant ones comprise:

- The architecture of the core was not flexible enough to integrate with the other XIFI controller SW modules.
- The authentication system is very limited. Role-based permission can only be assigned to services represented in the API and has to be configured manually. This does not scale while integrating with other SW modules and platforms which require authentication to share information.

3.2.2 OpenNaaS general enhancements

- To minimize the time necessary when deploying a new feature and to enrich the management possibilities of the platform, the core was completely redesigned using the main existing core concepts.
- The interfaces a developers needs to implement to deploy new functionality are simplified and enhanced. All automatable processing is taken over by the platform, without limiting the development possibilities. This has eased the NBI and SBI OpenNaaS development.

© 2013 - 2015 XIFI Consortium Parties

- Adaptation of the core's resource tree and the capabilities offered by OpenNaaS to tailor the platform exactly to the XIFI Network Controller needs.

- The network abstraction model has been enhanced as well in order to be able to keep the overall service view and the topology map. Thus OpenNaaS framework enables to abstract the underlying network complexity allowing the possibility for easily deploy and operate customized advanced network services according to PaaS connectivity services requests. These capabilities allow also the extension of Cloud management platforms (i.e. OpenStack) to integrate IT and network environments.

3.3 OpenNaaS interfaces development:

OpenNaaS has implemented all required functionalities and interfaces for the XIFI Network Controller.

3.3.1 OpenNaaS Northbound Interface:

The entry point of the PaaS manager to request a network connectivity service in the XIFI federation is the northbound interface. OpenNaaS has enabled a REST API interface to collect the required parameters that allows the XIFI Network Controller to establish the E2E path in the federation.

```
POST
http://{OpenNaaS_host}:{OpenNaaS_port}/OpenNaaS/XIFI/{XIFI_resource}/e2e/co
nnectEnds
```

where:

- {OpenNaaS_host} is the host where OpenNaaS instance is located (IP address or host name.
- {OpenNaaS_port} is the TCP port where OpenNaaS is listening. Typically 8888.
- {XIFI_resource} is the name of the OpenNaaS XIFI resource. For the use case it will be 'XIFI1'.

with this mandatory headers:

```
Content-Type: application/xml
Authorization: {auth}
```

where:

- {auth} is the HTTP Basic Authentication digest. In this case, the default user 'admin', with the default password '123456' produces digest 'Basic YWRtaW46MTIzNDU2'.

With this mandatory body as XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<ns2:connectEndsRequest xmlns:ns2="OpenNaaS.api">
    <sourceInstance>{srcVM}</sourceInstance>
    <sourceRegion>{srcRegion}</sourceRegion>
    <destinationInstance>{dstVM}</destinationInstance>
    <destinationRegion>{dstRegion}</destinationRegion>
</ns2:connectEndsRequest>
```

where:

- {srcVM} is the source OpenStack instance name.
- {srcRegion} is the configured name for the source XIFI region.
- {dstVM} is the destination OpenStack instance name.
- {dstRegion} is the configured name for the destination XIFI region.

It would produce a '204 No content' response if it was processed successfully. Otherwise, it would fail with a generic '500 Server error' response.

3.3.2 OpenNaaS Southbound Interface:

OpenNaaS southbound interface triggers the ABNO functionality so that it later establishes the network connectivity. OpenNaaS retrieves the information from other Software modules (OpenStack instances of the XIFI nodes, Network monitoring module and provides it to the ABNO by means of this interface.

As OpenNaaS communicates with ABNO on its SBI, the SBI is defined by ABNO (as ABNO API).

3.3.3 OpenNaaS interface to XIFI monitoring module.

Within the OpenNaaS platform, it has also been implemented the required interfaces to integrate with a monitoring module, which has been proposed in WP6 for the development of the showcases. The monitoring module triggers an alarm to OpenNaaS when congestion in the provisioned E2E path is detected so that the XIFI Network Controller is able to react accordingly to reduce congestion in the path or move the provisioned service to other non-congested path. OpenNaaS implements the interface to attack the following monitoring module REST API:

- To create an alarm:

```
curl -X POST -d '{ "threshold":"10", "host":"192.168.122.201", "port":"8080", "url_prefix":"/XIFI/raise_alarm
```

where:

- threshold is the threshold of the alarm
- host is the host to send the alarm (it should be OpenNaaS host)
- port is the port of the REST API to send the alarm on OpenNaaS host
- url_prefix is the url of the REST API to send the alarm
- dpid is the datapath ID of the OpenVswitch of the monitored port
- port_no is the port number in the OVS of the monitored port

To delete an alarm:

```
curl -X DELETE http://192.168.122.201:8080/XIFI/delete_alarm/{dpid}/{port_no}
```

where:

- dpid is the datapath ID of the OpenVswitch of the monitored port
- port_no is the port number in the OVS of the monitored port

To use the monitoring module, here are the applications configured in my ryu.conf file:

```
app_lists=ryu.app.gre_tunnel,ryu.app.quantum_adapter,ryu.app.rest,ryu.app.r
est_conf_switch,ryu.app.rest_tunnel,ryu.app.tunnel_port_updater,ryu.app.res
t_quantum, ryu.app.ofctl_rest,ryu.app.XIFI
```

4 RYU CONTROLLER

This chapter describes how to configure RYU to be controlled by ABNO and hence OpenNaaS above.

Each data centre is considered to be an independent SDN domain. Every domain has its own controller, with control responsibilities over physical and virtual network resources within the data centre.

The deployed controller is Ryu [11]. Only the functionalities of pushing the forwarding rules and discovering the topology information are used from the controller.

4.1 Setup guidelines

It is necessary to have installed Python 2.7 or higher.

There are two alternatives for the Ryu installation.

The first one is through the command pip.

```
sudo pip install ryu
```

The other way is through the Git Ryu reposition, executing the following command in the folder wanted.

```
git clone https://github.com/osrg/ryu.git
```

To check the installation has gone well and to see the connection is good

```
ryu-manager --verbose
```

NOTE: File **lldp.py** located in the path: */ryu/ryu/lib/packet/ has been modified in order to avoid a particular problem associated to an *out of range* issue. The changes were focused on changing the way going through the loop and adding a counter.

4.2 Ryu rules injection

The way to introduce Ryu flows is very easy. In this example it can be seen how introduce a flow.

```
curl -X POST -d '{"cookie": "1","dpid": "68831857340750", "match":
{"in_port":"10"}, "actions": [{"type": "OUTPUT", "port": "33"}]}'
http://$CONTROLLER_IP:8080/stats/flowentry/add
```

Where **dpid** parameter corresponds with the decimal dpid value of the ovs, and in_port and OUTPUT parameters refer to the input and output ports, respectively, for flowing traffic.

Finally, in order to see the flows configured for the ovs we can execute the following command.

```
sudo ovs-ofctl dump-flows br-int
```

On the other hand, to delete Ryu flows it is only necessary the decimal ovs dpid as it can be seen below.

```
http://$CONTROLLER_IP:8080/stats/flowentry/delete/68831857340750
```

© 2013 - 2015 XIFI Consortium Parties

5 QOS MECHANISMS IN OPENSTACK NETWORKS

5.1 Summary

In packet switched networks, as the XIFI cloud infrastructure, Quality of Service (QoS) can be defined as the ability to provide different priorities to different applications, users, or data flows, or to guarantee a certain level of performance to a data flow. For example, a required bit-rate or delay may be guaranteed. Quality of service guarantees are important if the network capacity is insufficient, especially for real-time streaming applications such as Voice-over-IP (VoIP), online games and IP-TV, since these often require fixed bit rate and are delay sensitive.

Currently, while it is possible for an OpenStack administrator to enforce some policies related to the virtual machines (VMs), such as assigned virtual cores, amount of RAM and disk, there are no open-source mechanisms to set and monitor the amount of traffic generated by a VM, and to configure a specific priority to some flows crossing the core network. So, it's possible for an user to flood the cloud network with malicious traffic without any restriction, and to impair other applications running in the same facility. For this main reason, and leveraging on the network architecture envisioned in XIFI, we decided to introduce some mechanisms to enforce QoS policies in OpenStack, extending the current open-source software suite (e.g. Neutron and Ryu OpenFlow controller) with new features.

5.2 Component Responsible

Developer	Contact	Partner
Matteo Gerola	mgerola@create-net.org	CNET
Giuseppe Cossu	giuseppe.cossu@create-net.org	CNET

Table 1: Component Responsible

5.3 Design choices

We defined these main requirements to be part of the QoS framework design:

- **The user can change the running policy at runtime**. Each customer can switch from a policy to another available one at runtime, without rebooting the VM. The changes are then applied by the framework in few milliseconds to the physical network.
- **QoS is applied at the edge of the network**. To ensure that each customer's traffic is tagged and shaped with the right rules, each policy is applied directly at the connection point between the VM and the cloud network.
- **Three QoS rules are supported**, but the framework is easily extendible with more constraints:
  - Maximum ingress bandwidth. The traffic going from the network to the VM (e.g. downstream) is limited to an upper bound defined by the policy.
  - Maximum egress bandwidth. The traffic flow sent out by each VM (upstream) is shaped to an upper limit defined by the network administrator and configured in the policy.

- Differentiated Services Code Point (DSCP) tagging. Contextually with the rate limiting, each traffic flow is tagged with a specific DSCP value, leveraging on the IETF RFC 2474 standard,that defines the DiffService field in the IPv4 and IPv6 packet headers. All the switches and router (both SDN and standard) in the XIFI network use this field to classify the packets in each port queue, thus prioritizing the traffic with an higher DSCP value.

Two different stakeholders can benefit from this functionality:

- **Network administrator**: the admin is the main responsible for the QoS configuration and management. He/she can:
  - Create/delete a new policy defined by a name, a description, and some rules that will be associated to that policy (e.g. max-bandwidth-ingress, max-bandwidth-egress, DSCP value).
  - Set the policy as default. The default policy (only one is allowed in the network) is automatically associated to all the newly created virtual machines.
  - Set the policy as shared. In order to reduce the manual work, the administrator can have a set of policies that are visible, and thus configurable, by all the users. These policies (obviously the default one is part of this subset) are usually the ones with less privileges, but with specific settings required by standard applications (e.g. web-server, multimedia streaming server).
  - Associate a policy with specific users/tenants. For the policies that are not shared, the administrator can manually configure each of them to be visible only by specific users. Generally this operation is applied only to specific users that require stronger Service Level Agreements.
- **Cloud user (tenant)**: while the network administrator is the one responsible for the correct configuration of the QoS, the users are the ones really affected by this feature. In order to reduce the intricacy of the setup (we cannot assume that each user is aware of the implications related to QoS), we let the admin to associate complex rules to user-friendly self-explained QoS policies. For example, the policies in the following table can be good examples. All the fields in white are visible by the users, while the real complex QoS rules are set by the administrator and are visible only to him/her.

Name	Description	Default	Shared	DSCP	Ingress-BW (kbps)	Egress-BW (kbps)
Bronze	Default policy, to be used will all the client VMs	YES	YES	15	50	100
Silver	Policy optimized for web-services	NO	YES	30	100	200
Gold	Policy for real-time services (e.g. video streaming)	NO	NO	45	1000	1000

© 2013 - 2015 XIFI Consortium Parties

Table 2: QoS policies

With the policies defined by the administrator, the user can:

- List all the available policies. After starting a VM, a user can see the running QoS policy, as well as the list of all the policies that are visible by him/her. Referring to the previous table, all the users could see the Bronze and Silver ones, while the Gold policy appears only to specific users.
- Assign the preferred policy to a VM or a network. After looking at the list, each user can then change at runtime the running policy with another one belonging to the list. Moreover, he can apply the choice to a single VM or to the whole network to which his/her VMs are attached. The change is processed by the system and applied in few milliseconds.

5.4 System Architecture

OpenStack Neutron configures the “data network” in order to allow the communication between VMs. The main process of Neutron is the so-called *Neutron-server* that exposes the API of the OpenStack Networking part, while the *Neutron-plugin* is the component in charge of the back-end implementation. Typically the plugin, such as Ryu, configures Open VSwitch (OVS) on all compute and controller nodes in order to create virtual networks for each tenant.

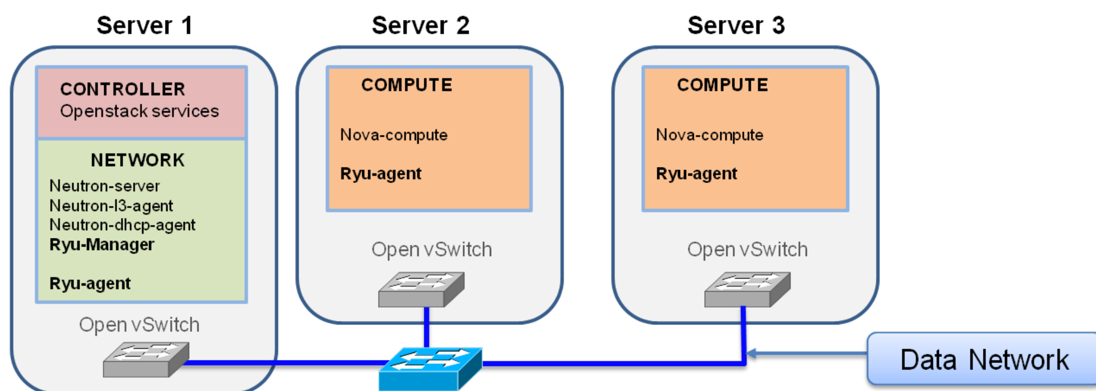


Figure 6: OpenStack Data Network

In particular, all the VMs attach their Virtual InterFace (VIF) to a local OVS bridge called *br-int*. That bridge is dynamically configured by the plugin using an agent process (e.g. Ryu-agent). The following figure shows how a tenant private network is configured upon an OVS. For the sake of simplicity is depicted one tenant private network on a single OVS switch.

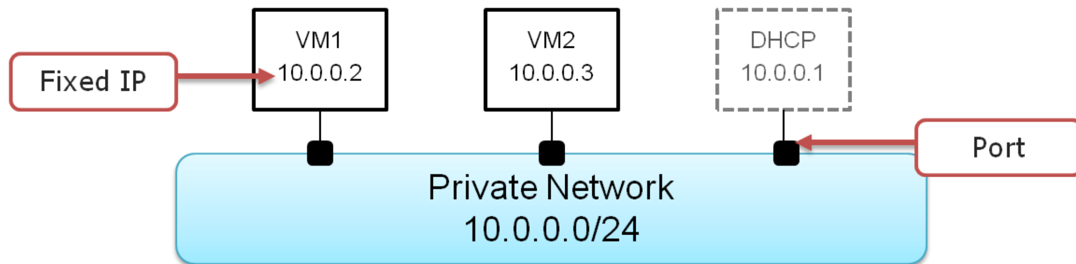


Figure 7: Neutron Private Network

Let us look into the Figure 7. Using the OpenStack Networking component each tenant can create his own *private network*, that is an isolated layer-2 network, and define a specific subnet (such as 10.0.0.0/24). The VIFs are mapped in Neutron using a *port*: a virtual switch port that defines *Fixed IP* and MAC address assigned to the VM interface. The tenant networks are created configuring OVS switch in order to allow/deny traffic and create the tenant *private network* as explained before.

The QoS policies are applied per *port* (although they can be set for a network), that is the Neutron resource attached to the VM virtual interface. In this way it is possible to control the incoming/outcoming traffic from the tenant port as explained in the previous section.

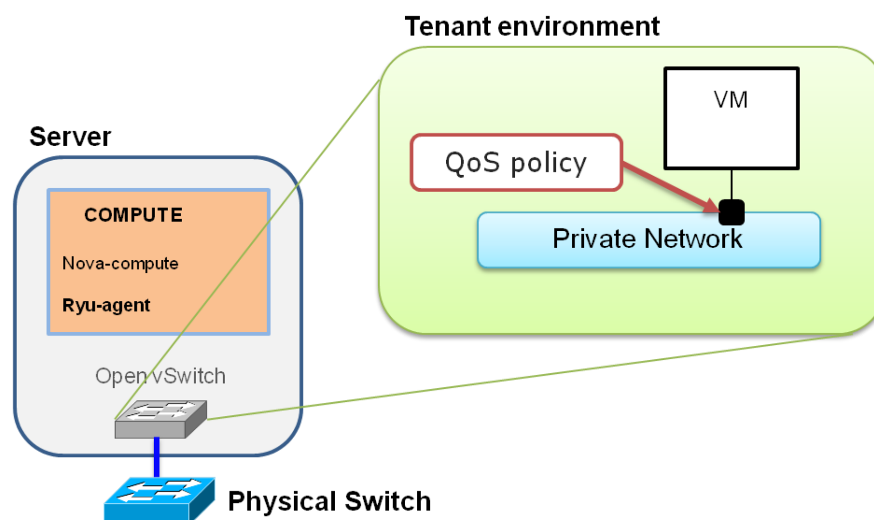


Figure 8 - QoS policy applied per port

As already mentioned the Neutron ports are mapped in OVS that provides the mechanism to create the private networks. As depicted above, the policies are applied updating the already present rules in the OVS switch using OpenFlow rules and OVSDB protocol. In order to introduce the QoS in OpenStack, therefore to apply the defined policies in Neutron, extensions have been developed in the following components:

- Neutron Server
  - API and Database
- Ryu Plugin
  - Communication with Neutron Server, OVSDB and OpenFlow rules on OVS
- Neutron Client
  - Command Line Interface (CLI)

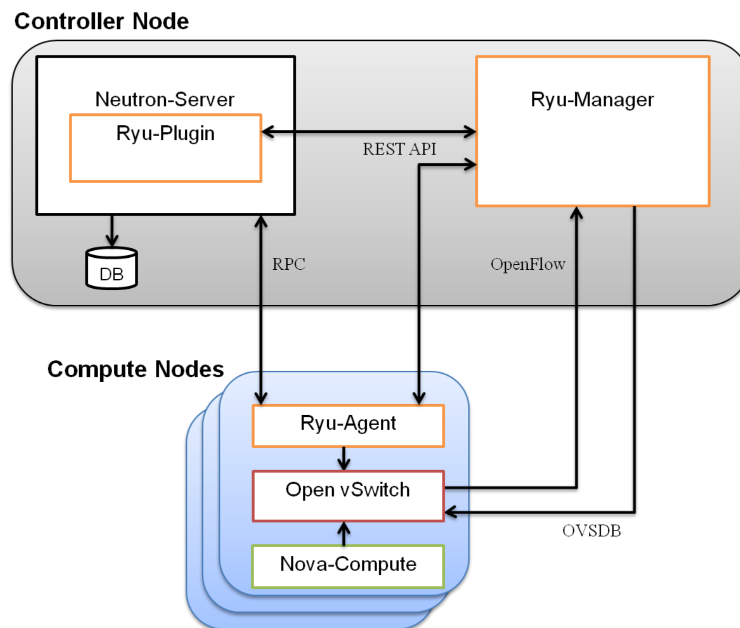


Figure 9: Neutron and Ryu Plugin Cooperation

First of all, QoS API and Data Model for MySQL have been defined (reported in the following sections). These are been implemented as Neutron extensions: *Neutron-server* manages the QoS API and exchanges all information *Ryu-manager* needs by means of *Ryu-plugin*.

In this way, the plugin can use different techniques and technologies to create isolated virtual networks, such as SDN [13]. Ryu-manager acts as local SDN Controller and configures the OVS switch using OVSDB Protocol.

Now we will describe what happens when a VM instance is created: Neutron port is created and then OVS port as well (note: VIF creation in managed by OpenStack Nova [8]). Ryu-agent is in charge of detect the port creation and to set up a first initialization (there are 2 steps of OVS initialization) with the basic configuration. The information is retrieved through Ryu-Plugin from Neutron. If additional configurations for OVS are needed, these are directly performed by Ryu-Manager as depicted in the figure above. Here the QoS parameters are set up: Ryu-Manager has been extended in order to create and update an OVS port with the QoS parameters defined above. In particular, the default QoS is set up in the first initialization phase whereas the other QoS are set up when the Neutron port in updated with a new QoS policy.

In order to use the developed extensions with the CLI, the *Neutron-client* has been modified in order to accept the QoS parameters and to communicate with Neutron server using REST architectural style.

5.5 QoS Neutron API

The table below summarizes the new API calls added in Neutron to support Quality of Service. With this northbound API, each OpenStack module (with the right privileges) can interact with Neutron to monitor and set the QoS policies.

© 2013 - 2015 XIFI Consortium Parties

QoS Neutron API call	Parameters	Mandatory	Authorized Users
neutron qos-create	policies	YES	Admin only
	description	NO	
	name	YES	
	default	NO	
	visible	NO	
neutron qos-list	tenant_id	NO	Admin only
neutron qos-show	id	YES	Admin only
neutron qos-delete	id	YES	Admin only
neutron qos-update	id	YES	Admin only
	description	NO	
	name	NO	
	policies	NO	
	default	NO	
	visible	NO	
neutron qos-tenant-associate	id	YES	Admin only
	tenant_id	YES	
neutron qos-tenant-disassociate	id	YES	Admin only
	tenant_id	YES	
neutron port-update	port_id	YES	Admin and tenant
	qos	NO	
neutron network-update	network_id	YES	Admin and tenant
	qos	NO	

Table 3: New API calls added in Neutron to support Quality of Service

5.6 QoS Neutron Database

Similarly to the Neutron northbound API, the QoS framework needs some extensions also to the Neutron database. The python QoS module add four new tables to the database, without modifying existing tables, in order to be back compatible with standard deployment, and also to be enabled and disabled on demand without modifying the behaviour of the other components. Two Neutron tables (ports and tenant) are used as foreign keys to maintain relationship between the configured policies and both ports and users.

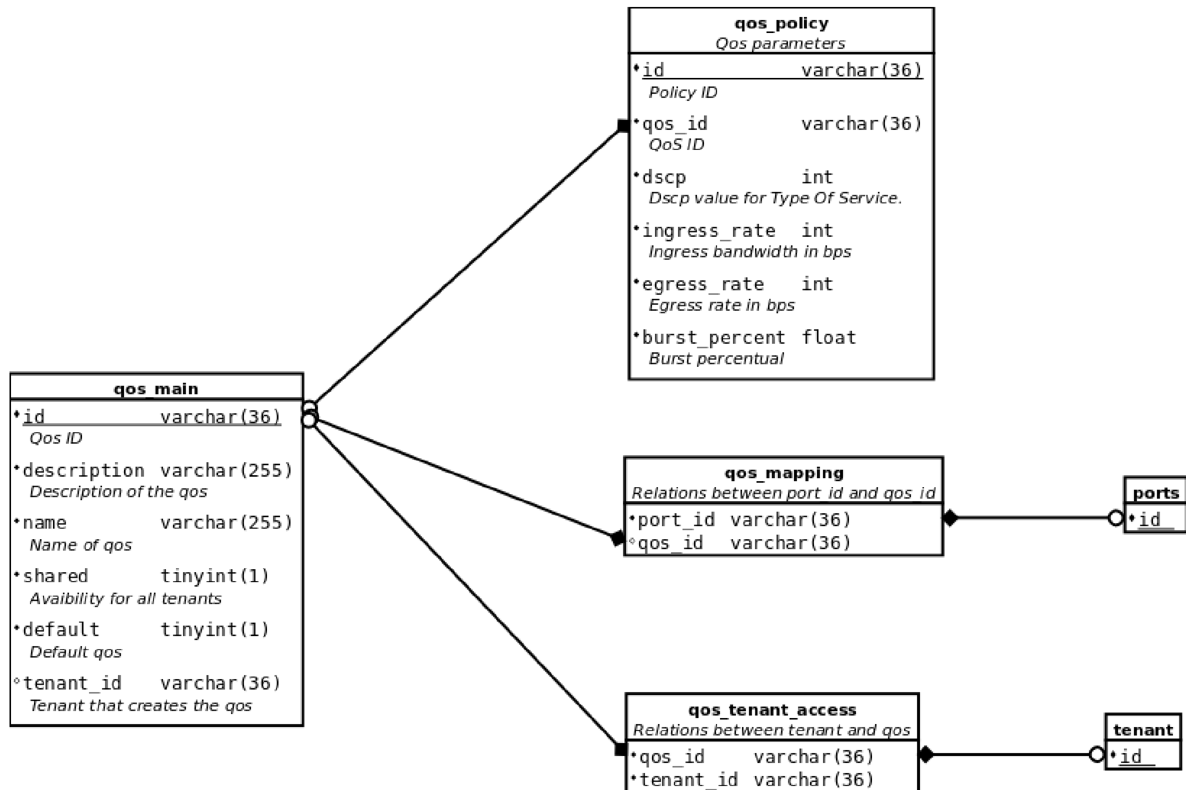


Figure 10: QoS module's new database tables

The tables are:

- **qos_main** is the main table for QoS. *Name* and *description* are fields used to describe policy itself. *Default* is false "by default", and only one policy can be true at a certain time. *Shared* states if the policy is visible to all the users or if the administrator intends to limit its visibility only to specific customers.
- **qos_policy** defines the QoS rules associated to the policy. Having the rules in a separated table allows adding easily new rules without changing the main QoS design. Regarding the current supported rules:
  - *Ingress_rate* and *egress_rate* are expressed in bit per seconds (bps).
  - *Dscp* is the Type of Service value sets in the IP header that can be used to differentiate the service. It has to be multiple of 4 and less than 65.
  - *Burst_percent* specifies the burst rate associated to the *ingress_rate* and *egress_rate*, in percentage.
- **qos_tenant_access** stores the associations between policies and tenants (e.g. user's projects).
- **qos_mapping** stores the current policy running on each port.

5.7 Installation and configuration procedure

The pieces of software involved in the QoS OpenStack extension are:

- Neutron: https://github.com/SmartInfrastructures/neutron
- Python-neutronclient: https://github.com/SmartInfrastructures/python-neutronclient
- Ryu OpenFlow Controller: https://github.com/SmartInfrastructures/ryu

© 2013 - 2015 XIFI Consortium Parties

These software bundles are forked from the community master branches:

- https://github.com/openstack (for Neutron and Python-neutronclient)
- https://github.com/osrg/ryu (for Ryu OpenFlow Controller)

Each repository has three main branches, with specific README.qos files (new features details, how-to install and use the software):

- *master*: the original master branch updated every week with the upstream repositories
- *qos:* the main QoS branch, that should be used in every production environment
- qos-dev: new features development, testing and bug fixing, later merged in the qos branch

These requirements have to be met in order to correctly install and configure the software:

- use *Icehouse* version for all Openstack components
- use the same branch for all three software components (to guarantee same features database and APIs)
- where *ryu-manager* runs load *qos_cn* app. Typically this is done in the configuration file "/etc/ryu/ryu.conf"
- redeploy the machines

For installation details see section 2.3.

© 2013 - 2015 XIFI Consortium Parties

6 SHORT TERM EVOLUTION

This chapter introduces the current activities being carried out in Task 3.1 which will impose some evolution steps for the XIFI Network Controller in the short term. Two types of actions are being taken, one of them referred to the internal evolution of the Network Controller, and the other one related to the external interaction with other components that are part of the XIFI Project.

6.1 Development of capabilities internal to the controller

The XIFI Network Controller is a modular software element. This fact permits that every specific component can take its own independent evolutionary path, minimizing the impacts on the rest of the components comprising the XIFI Network Controller. Additionally, new components can be integrated in the overall XIFI Network Controller by defining integration interfaces with the rest of the modules.

This section describes the modules and capabilities currently being developed and integrated in the XIFI Network Controller.

6.1.1 OpenNaaS integration

OpenNaaS is one of the original software components considered in the XIFI Network Controller design. Its capabilities were already reported in Deliverable 3.1 [12].

Essentially the OpenNaaS component will maintain the service awareness in the XIFI Network Controller architecture. It will keep information related to the resources associated to tenants on a per DC basis, including some identifiers that allow connectivity service tracking. For instance, data like MAC addresses of the VMs, ports where the VMs are attached to the Open vSwitch virtual device, local identifiers of the network in each DC assigned by the local instance of OpenStack, etc. Some other data complements the local information obtained from the resources deployed in each DC, and it is essential for the connectivity service, like the regions where the tenants have deployed their resources, or proper identifiers of the end-to-end connectivity service.

OpenNaaS – ABNO integration has been tested. However the real integration of the complete system is yet pending.

6.2 Integration with elements external to the controller

This section addresses the activities being currently done in the integration process with external capabilities external to the XIFI Network Controller.

6.2.1 Connection to MD-VPN transport infrastructure

The XIFI deliverable 5.2 [14] reports the XIFI backbone implementation for the initial nodes of the federation. The backbone connectivity is based on the MD-VPN transport service offered by the GÉANT community. The purpose of the XIFI Network Controller is to provide an orchestrated connectivity between the edges (i.e. the XIFI nodes or DCs) by stitching that local connectivity to the XIFI backbone. The XIFI Network Controller will instruct the local SDN controllers to compose the service by pushing the appropriate OF rules in the devices, either virtual (e.g. those created in the servers by OpenStack) or physical (e.g. those acting as demarcation points before entering the NREN infrastructure).

Then it is essential to integrate the local connectivity with the backbone transport. Due to the diversity of topological options it is required to clearly identify the service requirements to find commonalties

© 2013 - 2015 XIFI Consortium Parties

and identify particularities for each of them.

6.2.2 GRE Tunnels

6.2.2.1 Installation Manual

The process to configure GRE Tunnels follows the steps detailed below, which must be repeated on both servers in br-ex and br-int Open vSwitch virtual bridges created as result of OpenStack installation.

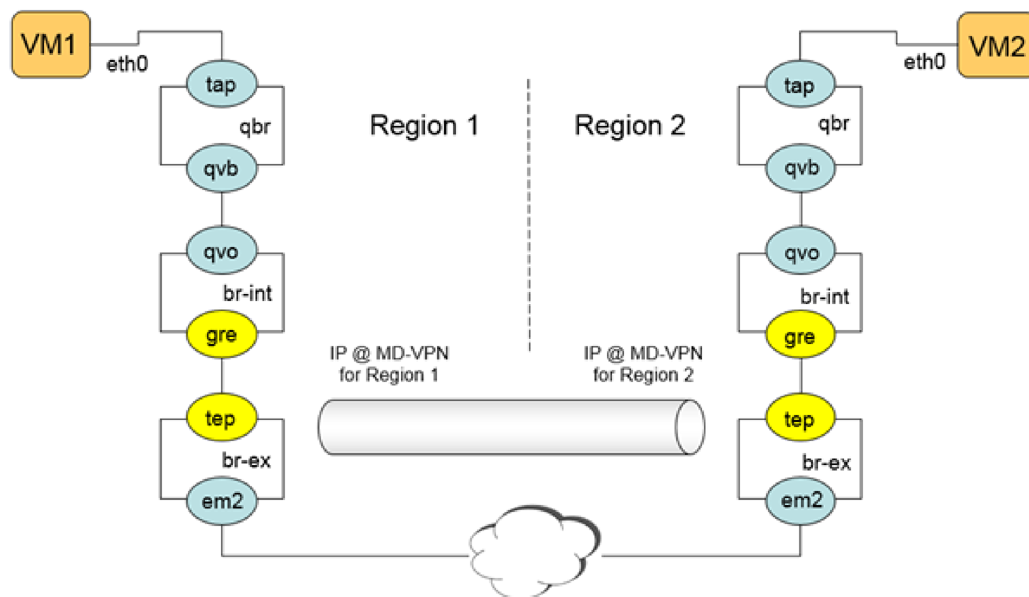


Figure 11: GRE Tunnels

6.2.2.2 Configuring the GRE Tunnel Endpoint on br-ex

The GRE tunnel endpoint (TEP) will act as endpoint for the GRE tunnel on each end. This TEP will be assigned with an IP address of the MD-VPN to allow connectivity to other data centers in the federation.

To create the internal interface, use this command:

```
ovs-vsctl add-port br-ex tep0 -- set interface tep0 type=internal
```

Note.- A planning of the IDs (e.g., tep0, tep1, etc) to be assigned to the TEPs should be defined for the whole federation.

Once the internal path is established, assign it with an IP address of the MD-VPN using *ifconfig* or *ip*, whichever you prefer.

Note.- Separated IP ranges of the MD-VPN should be identified for inter-DC communication.

Note.- The IP address to be assigned should be different from the one assigned to the physical NIC on the server.

Then:

```
ifconfig tep0 192.168.100.2 netmask 255.255.255.0
```

Once it is established the GRE tunnel endpoint on each server, test connectivity between the endpoints using *ping* or a similar tool.

6.2.2.3 Establishing the GRE Tunnel

Use this command to add a GRE interface to the br-int on each server:

```
ovs-vsctl add-port br-int gre0 -- set interface gre0 type=gre
options:remote_ip=<GRE tunnel endpoint on other hypervisor>
```

Once a service is deployed in separate data centres, the traffic to be interchanged between VMs residing in each data centre will be forwarded through it.

6.2.2.4 Deployment of GRE Tunnels in XIFI infrastructure.

Up to now, three potential cases have been identified, as shown in the figure below.

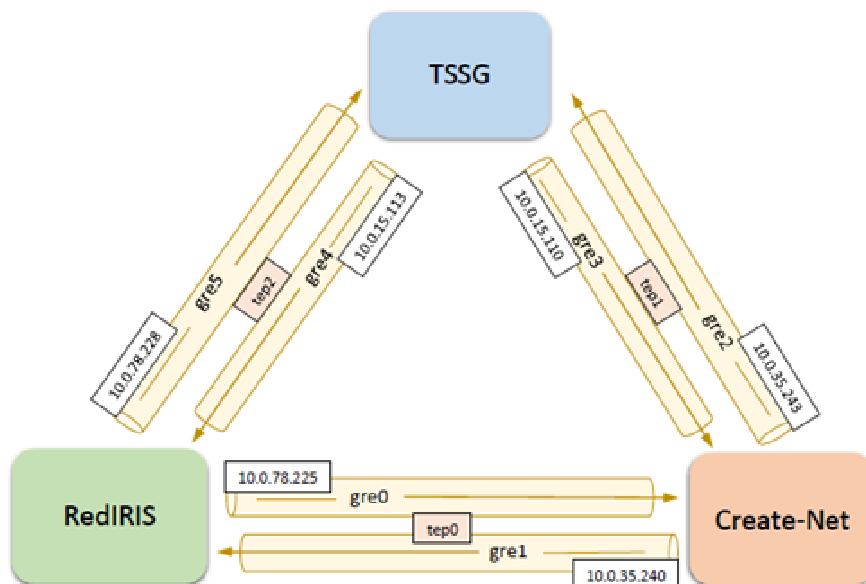


Figure 12: Deployment of GRE tunnels

© 2013 - 2015 XIFI Consortium Parties

7 CONCLUSIONS

This deliverable provides the second software release of the XIFI Network Controller described in the framework of WP3. The outcomes enclosed in this document define the basis of Task 3.1 - Networks and virtual networks Infrastructure Adaptation mechanisms.

Although some activities are still being performed at the time of writing, especially in terms of deployment and validation, the documentation provided remarks the innovation overcome by the project in the field of federated network management.

Meaningful outcomes have been fulfilled with the release and/or integration of the following software modules:

- Real implementation of the reference architecture based on the ABNO principles
- Integration with the system for Network-as-a-Service OpenNaaS, enabling a full-fledged solution
- Installation and configuration procedures of the industry-based network controller Ryu
- Innovative mechanisms to manage the Quality-of-Service in OpenStack environments

© 2013 - 2015 XIFI Consortium Parties

REFERENCES

[1] D. King, A. Farrel, “A PCE-based Architecture for Application-based Network Operations”, draft-farrkingel-pce-abno-architecture-07 (work in progress), February 2014.

[2] I. Bueno, J.I. Aznar, E. Escalona, J. Ferrer, J.A. García-Espín, “ An OpenNaaSOpenNaaS based SDN Framework for Dynamic QoS control,” SDN for Future Networks and Services (SDN4FNS), (November 2013) http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6702533

[3] J.I. Aznar, .; Jara, M., ; Rosello, A., ; Wilson, D., ; Figuerola, S.,” OpenNaaSOpenNaaS Based Management Solution for Inter-data Centres Connectivity, “ IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom), December, 2013. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6735399

[4] Keystone OpenStack. http://docs.openstack.org/developer/keystone/

[5] Open vSwitch. http://openvswitch.org/

[6] OpenFlow - Open Networking Foundation. https://www.opennetworking.org/sdn-resources/openflow

[7] OpenNaaSOpenNaaS. : http://opennaas.org/http://opennaas.org/

[8] OpenStack Nova. https://wiki.openstack.org/wiki/Nova

[9] QoS in Neutron OpenStack. https://github.com/SmartInfrastructures/neutron/wiki

[10] Ryu 3.18 Documentation. http://ryu.readthedocs.org/en/latest/index.html

[11] Ryu OpenFlow Network Controller. http://osrg.github.io/ryu/

[12] XIFI Deliverable D3.1 XIFI infrastructure adaptation components API open specification", http://wiki.fi-XIFI.eu/Public:D3.1https://bscw.fi-XIFI.eu/pub/bscw.cgi/d44705/XIFI-D3.1-XIFI_infrastructure_adaptation_components_API_open_specification.pdf

[13] XIFI Deliverable D3.4 "XIFI infrastructure network adaptation mechanisms API", https://bscw.fi-XIFI.eu/pub/bscw.cgi/d64447/XIFI-D3.4-XIFI_Infrastructure_Network_Adaptation_Mechanisms_API.pdf

[14] XIFI Deliverable D5.2 “Report on XIFI Core Backbone Deployment”, https://bscw.fi-XIFI.eu/pub/bscw.cgi/d64414/XIFI-D5.2-XIFI_Core_Backbone.pdf

© 2013 - 2015 XIFI Consortium Parties

Appendix A Interaction With RYU As Local SDN Controller

The XIFI Network Controller orchestrates local SDN controllers, one per each XIFI node. The reason for that is because each XIFI node forms a separated SDN domain.

The Ryu OpenFlow controller [11] has been considered as the local SDN controller for the new release of the XIFI Network Controller. This Annex collects a number of queries/interactions used for the communication between the XIFI Network Controller (the Provisioning Manager module in the ABNO component) and the Ryu controller. Ryu API is documented in [11].

Some useful queries are the following:

- Get the list of all switches:

```
curl http://$CONTROLLER_IP:8080/stats/switches
```

- Retrieve topology:

```
curl http://$CONTROLLER_IP:8080/v1.0/topology/links
```

- Clear switch rules:

```
curl                              -X                              DELETE
http://10.0.78.2:8080/stats/flowentry/clear/$SWITCH_DPID
```

- Push a flow

```
curl  -X  POST  -d  '{"cookie":  "0","dpid":  "68831857340750",  "match":
{"in_port":"10"},  "actions":  [{"type":  "OUTPUT",  "port":  "33"}]}'
http://$CONTROLLER_IP:8080/stats/flowentry/add
```

Appendix B Monitoring Modules

The monitoring modules described in this section have been specifically developed to fit the use case scenario of the XIFI Network Controller. This means, these modules are not integrated in the XIFI Federation Architecture, and do not follow the same means.

B.1 Summary

Le monitoring is a component is an additional function developed inside the RYU SDN controller chosen to control the internal XIFI node network. It allow to monitor the network traffic using the OpenFlow protocol to communicate with the OpenFlow switches and provide an external API to register alarm function of the network real time QoS. In particular, it is used to monitor the bandwidth between the XIFI nodes in order to trigger a new bandwidth allocation action by OpenNaaS component.

Reference Scenarios	MG2(old UC6 SDN Traffic engineering)
Reference Stakeholders	Users: those who request a connected infrastructure comprising one or more XIFI nodes. Infrastructure owners and operators: those who want to connect resources to the XIFI federation.
Type of Ownership	Development and extension
Original tool	Ryu
Planned OS license	None at the moment
Reference OS Community	None at the moment

B.2 Component Responsible

Developer	Contact	Partner
Bruno VIDALENC	Bruno.vidalenc@thalesgroup.com	TCS
Mathieu BOUET	Mathieu.bouet@thalesgroup.com	TCS

B.3 Motivation

The monitoring module has been developed to improve Quality of Service in cloud network. Indeed SDN network present in cloud networks allow more flexibility, adaptability in order to better react function of network state in order to provide the best Quality of service possible. The network central SDN controller is able to continuously modify the OpenFlow rules deployed in the switches in order to maintain the QoS. But the first step in this process is to know with precision the state of the network. That is for what the monitoring modules has been developed. Observe the network stats and in particular its QoS parameters (bandwidth, latency, etc...), and trigger a reaction to maintain a certain QoS. For that purpose, the monitoring module, provide an external API to register an alarm on a specific QoS parameter (for example QoS) in order to be notified when the QoS is degraded.

Intra-datacenter networks are currently often overprovisioned, and so, more rarely subject to congestion or degraded, even if this problem will surely occur when datacenter will become even more complex that today. But for a federation, network QoS is a critical part for its success. Indeed, the interconnections between XIFI nodes have a limited capacity, and it is very important to exploit them finely. For the XIFI success, it is important that VM distributed across multiple XIFI nodes must be transparent to the user, as its VMs were in the same datacenter. For that purpose, the weak link is the interconnection between XIFI nodes, and the management of its link.

Fortunately, the GEANT network and its NREN network interlinking the XIFI nodes, provides some features that allow to dynamically managing its interconnection like the Bandwidth and Demand feature. Thanks to the BoD service, it is possible to modify in real time the QoS and in particular the bandwidth of the inter-XIFI nodes connection. Combined with the flexibility, adaptability of OpenFlow network, the BoD service, the monitoring module allow to build a federation of cloud like the XIFI federation with network performance close to a unique (gigantesque) cloud.

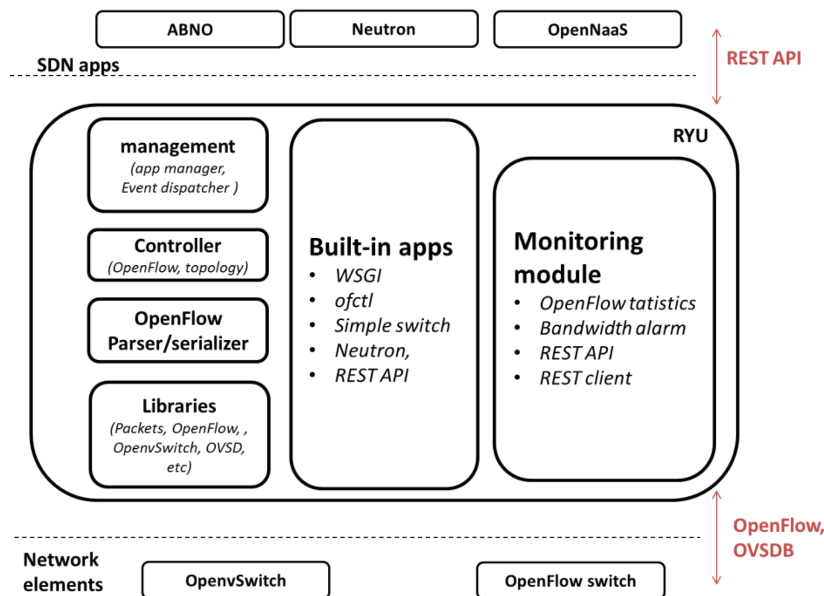
Following the SDN principle, the monitoring module is instantiated in the SDN controller, i.e. in each XIFI nodes. The XIFI SDN controller choice being Ryu, the implementation has been done as a tiers application in this application. We will describe in the following section how the monitoring module is architectured and implemented into Ryu.

B.4 User Stories Backlog

Id	User Story Name	Actors	Description
1	Connect dispersed infrastructure	Infrastructure owners	The infrastructure owners provide infrastructure (computing and network) resources to the federation to interconnect them
2	Connect VMs	User	The user creates a VM in one or more XIFI nodes
3	Add new VMs to an existing service	User	The user deploys additional VMs in a different XIFI node
4	Total service removal	User	The user removes all the deployed VMs, and the connectivity

B.5 Architecture Design

The monitoring module is part of the RYU controller. The following figure tries to represent the main component of the RYU SDN controller.



The central management of Ryu applications (in ryu/base subfolder):

- Load Ryu applications
- Provide *contexts* to Ryu applications
- Route messages among Ryu applications

B.5.1 The main component of OpenFlow controller (in ryu/controller subfolder):

- Handle connections from switches
- Generate and route events to appropriate entities like RYU applications
- Manage switches. (Planned to be replaced by a new module for switch and link discovery from ryu/topology).
- OpenFlow event definitions.
- Basic OpenFlow handling including negotiation

- A. (in ryu/topology subfolder)

B.5.2 The Ryu libraries (in ryu/lib subfolder) provide:

- The Ryu packet library.
- Decoder/Encoder implementations of popular protocols like TCP/IP
- ovsdb interaction library.
- OF-Config implementation.

B.5.3 The Third party libraries (in ryu/contrib subfolder) are composed of:

- Open vSwitch python binding. Used by ryu.lib.ovs.
- Oslo configuration library. Used for ryu-manager's command-line options and configuration files
- Python library for NETCONF client. Used by ryu.lib.of_config.

The OpenFlow wire protocol encoder and decoder (in ryu/ofproto subfolder) are responsible of :

- OpenFlow definitions
- Decoder/Encoder implementations of OpenFlow.

The built-in applications (in ryu/app subfolder) include:

- Application for openflow switches management (ofctl)
- An OpenFlow L2 learning switch implementation (simple_switch)
- Application for Openstack integration. Listen OpenFlow port status change notifications from switches. Consult ovsdb to retrieve the corresponding port uuid. Notify relevant parties, including quantum (via Ryu plug-in) and Ryu applications. (via Ryu Events) (quantum_adapter)
- A set of REST API used for Openstack integration like network registration and end-point port management (rest); switch configuration (rest_conf_switch); for Interface registration and maintaining interface association to a network (rest_quantum).

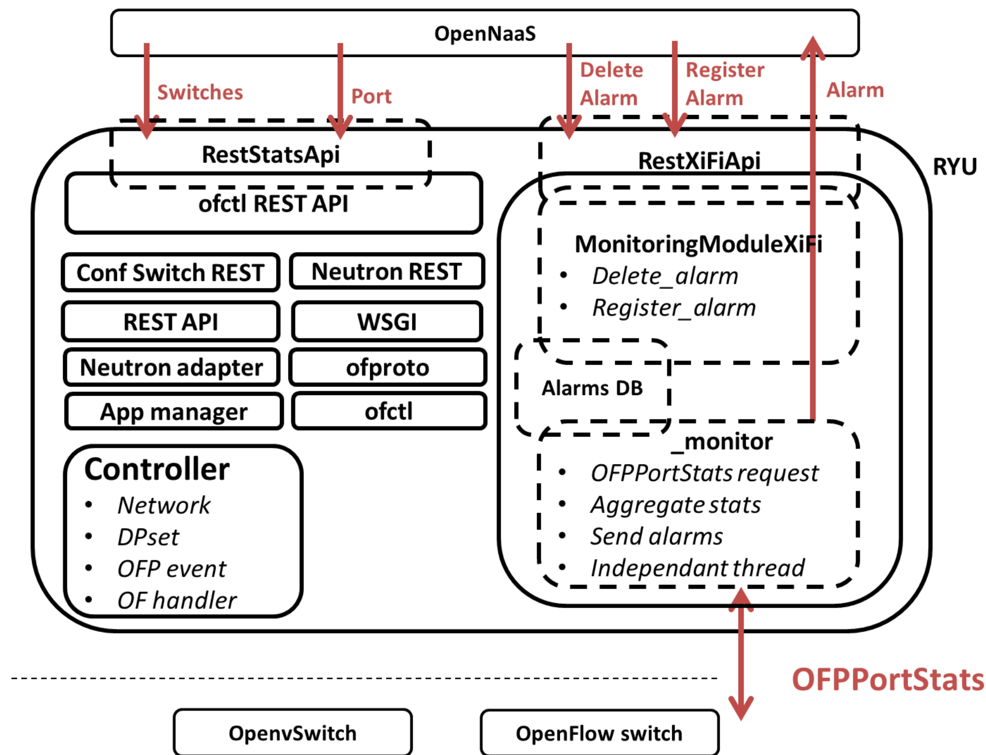
The monitoring module has been developed inside this in architecture, relying on the RYU tools and libraries. The monitoring module is composed of two part represented on the following figure.

On part is responsible of exposing the REST API to the outside and recording of deleting the alarms.

The alarms are stored in a unique database in order to be available for the second subcomponent that is responsible of the monitoring. This component is an autonomous thread that periodically request OpenFlow statistics to the switches functions of the alarms that are active at the moment.

If it detects that the real-time bandwidth is higher that the threshold of the alarm, it sends an HTTP request to the REST API defined by the alarm.

The alarms that can be recorded with the monitoring modules are currently focused on the bandwidth of one virtual port. Consequently an alarm is identified by the switch ID (datapath ID) and the port number in this switch. To retrieve this information, the ofctl application and its REST API provided by RYU are required. The other components present in the architecture are the module required to Openstack and Neutron to create the internal network.



B.6 Basic actors

The basic actors are the IaaS, the OpenNaaS component, the ABNO controller, the Ryu SDN controllers in each XIFI nodes and the OpenFlow switches.

In this overall mechanism OpenNaaS acts as orchestrator to request bandwidth between the different XIFI nodes. Its action is mainly triggered by the IaaS. The ABNO controller is responsible of the inter-XIFI node connectivity. The Ryu SDN Controller in collaboration with Openstack/neutron are responsible of the intra-XIFI node communication.

B.7 Main interactions

The monitoring module is a feature added to the network central controller to monitor the traffic in

real time.

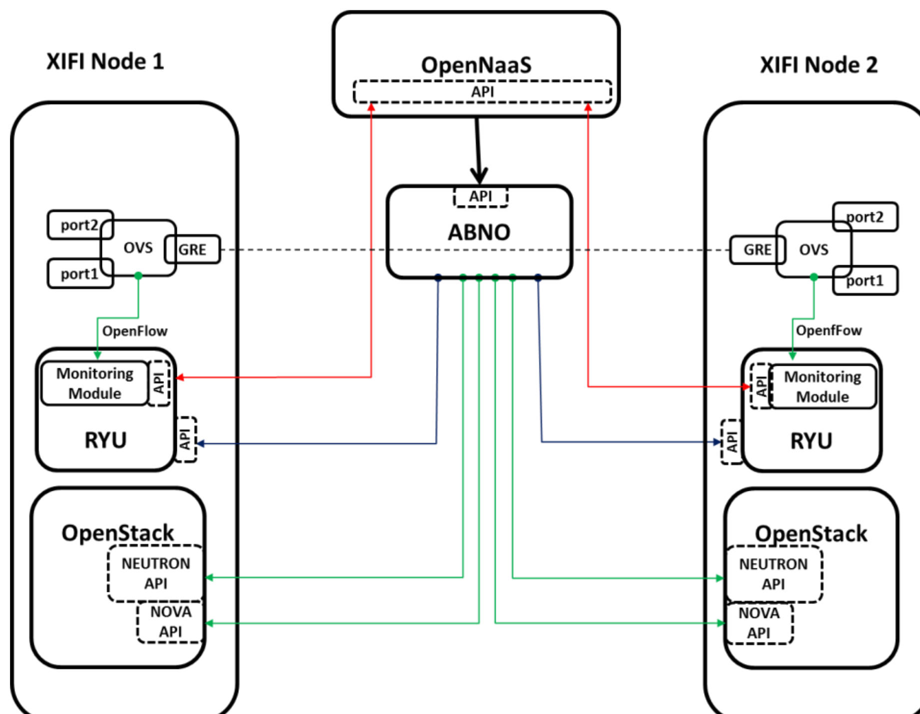
This function periodically send OpenFlow request to the network element in charge of carrying the traffic. The goal of this function is to collect the traffic statistics and aggregate them to have the status of the network bandwidth. The monitoring module is instantiated inside the RYU controller. It can monitor the traffic of both the intra and inter XIFI node communication

Leveraging on this knowledge, the monitoring module provides an external REST API to other components such as OpenNaaS in order to register bandwidth alarms. These bandwidth alarms are raised if the traffic in more important that a specific threshold. Each alarm is specific to one port and one OpenFlow switches. Once the alarm activated, the monitoring module will send an alarm containing the affected tuple switch/interface to OpenNaaS, only if the traffic become higher that the registered threshold.

Although this functionality can be exploited on any port of any switches, its main purpose is to monitor the output link toward the other XIFI nodes, in order to trigger the action of increasing the inter-XIFI node bandwidth.

Indeed, OpenNaaS can request and configure more bandwidth between the XIFI nodes if necessary. But this operation must rely on the real time network usage, that only the monitoring module is aware.

The REST API, allow OpenNaaS to register an alarm that is send when the inter-XIFI nodes communication are close to the maximum allocated bandwidth. With this alarm the OpenNaaS component can request more bandwidth between the XIFI nodes in order to provide to the users the bandwidth function of their needs. This feature is the trigger, of the action of adapting the bandwidth to the users' usage, without which it will be impossible to decide or not to request bandwidth to the BoD service. The following figure represents the different actor and their interactions.



B.8 Release Plan

The features have been partially implemented into the former XIFI SDN controller Floodlight, and then have been reimplemented and completed into the final XIFI SDN controller: RYU.

Version ID	Milestone
1.0	M12
2.0	M24

B.9 Installation Manual

B.9.1 Installation Requirements

Here is the software that have been tested and validated with the monitoring module.

- OS: Ubuntu 14.04 LTS with kernel 3.13.0-24-generic x86_64 into a vm executed in kvm
- Openstack: icehouse branch with Devstack icehouse branch (commit 1921be9226315b175ad135f07aeb0a715aaffb24)
- Neutron icehouse branch (commit 7f13cbca1c007623313025ecfeb8e7cae9e7e365)
- Ryu: 3.14 (commit 5aa14c61a1f31ca277cdc7e978cd45c7f4b9cbe4)
- Open vSwitch 2.0.2
- KVM: QEMU emulator version 2.0.0

B.9.2 Software Repository

Currently the software has been delivered via email. It is foreseen to push the software along with the RYU software used in the XIFI project.

B.9.3 Setup Guidelines

The monitoring modules has been developed as a separate component, as a not intrusive manner.

It uses the internal Ryu API as well as the functionality bringed by the ofctl application. Moreover, the alarm registration is identified by the datapath ID and the Port number. These ID can be retrieved through the ofctl rest API application of RYU. We recommend running the ofct_rest application in conjunction with our monitoring module.

The monitoring module is implemented in a single python file call XIFI.py. This file must be placed in the Ryu folder into the application folder. In general, this folder is in /opt/ryu/ryu/app/

The monitoring module, must be started with RYU, either by adding the XIFI.py file in the command line or adding the ryu.app.XIFI application into the configuration file.

© 2013 - 2015 XIFI Consortium Parties

Using the command line, if you use mininet with only the ofctl application and without your monitoring module, we used to start Ryu with the following command line:

```
Cd /opt/ryu && bin/ryu-manager --verbose --observe-links ryu/app/ofctl_rest.py ryu/app/simple_switch.py
```

To start RYU with the monitoring module, just add the XIFI.py as an additional file to start:

```
Cd /opt/ryu && bin/ryu-manager --verbose --observe-links ryu/app/ofctl_rest.py ryu/app/XIFI.py
ryu/app/simple_switch.py
```

If you prefer to use a configuration file (/etc/ryu.conf), you have to add the application in the list of application to start. If we take the example of the configuration file used to work in an openstack/neutron use case. Without the monitoring module, the configuration file should look like this:

```
cat /etc/ryu/ryu.conf

[DEFAULT]

app_lists=ryu.app.gre_tunnel,ryu.app.quantum_adapter,ryu.app.rest,ryu.app.rest_conf_switch,ryu.app.rest_tunne
l,ryu.app.tunnel_port_updater,ryu.app.rest_quantum

wsapi_host=192.168.122.201

wsapi_port=8080

ofp_listen_host=192.168.122.201

ofp_tcp_listen_port=6633

neutron_url=http://192.168.122.201:9696

neutron_admin_username=neutron

neutron_admin_password=password

neutron_admin_tenant_name=service

neutron_admin_auth_url=http://192.168.122.201:35357/v2.0

neutron_auth_strategy=keystone

neutron_controller_addr=tcp:192.168.122.201:6633
```

To use the monitoring module, you must add the XIFI application and we also recommend to add the ofctl_rest api in order to easily retrieve the datapath ID required by the alarm registration. In consequence, the new ryu.conf file sould look like this:

```
cat /etc/ryu/ryu.conf

[DEFAULT]

app_lists=ryu.app.gre_tunnel,ryu.app.quantum_adapter,ryu.app.rest,ryu.app.rest_conf_switch,ryu.app.rest_tunne
l,ryu.app.tunnel_port_updater,ryu.app.rest_quantum,ryu.app.ofctl_rest,ryu.app.XIFI

wsapi_host=192.168.122.201

wsapi_port=8080

ofp_listen_host=192.168.122.201

ofp_tcp_listen_port=6633

neutron_url=http://192.168.122.201:9696

neutron_admin_username=neutron

neutron_admin_password=password

neutron_admin_tenant_name=service

neutron_admin_auth_url=http://192.168.122.201:35357/v2.0
```

© 2013 - 2015 XIFI Consortium Parties

```
neutron_auth_strategy=keystone
neutron_controller_addr=tcp:192.168.122.201:6633
```

B.10 User Manual

The monitoring module must be loaded with RYU. To start it, add the monitoring module (XIFI.py file) with ryu. Here is an example where the monitoring module is run along the simple L2 application built-in RYU.

```
i2cat:(master)~/tid/ryu$ bin/ryu-manager --verbose --observe-links ryu/app/ofctl_rest.py ryu/app/XIFI.py ryu/app/simple_switch.py
loading app ryu/app/ofctl_rest.py
loading app ryu/app/XIFI.py
loading app ryu/app/simple_switch.py
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app ryu/app/XIFI.py of RestXIFIAPI
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu/app/ofctl_rest.py of RestStatsApi
instantiating app ryu/app/simple_switch.py of SimpleSwitch
 (19581) wsgi starting up on http://0.0.0.0:8080/
```

To test the monitoring module, the easiest way is to create a virtual network using mininet. Here is an example to build an simple OpenFlow network with three switches in line and attach them to RYU.

```
i2cat:~$ sudo mn --topo linear,3 --mac --controller=remote,ip=127.0.0.1 --switch ovsk
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s2) (h3, s3) (s1, s2) (s2, s3)
*** Configuring hosts
h1 h2 h3
*** Starting controller
*** Starting 3 switches
s1 s2 s3
*** Starting CLI:
```

To monitor an interface you need to know the datapath ID of the OVS instance concerned by the

monitoring and its port number. For that, you can rely on the REST API provided by the ofctl built-in application of RYU.

If we want send traffic from h1 to h2 and monitor the output traffic of the first switch, here is an example to retrieve this information using the RYU REST API:

```
i2cat:(master)~/tid/ryu/ryu/app$ curl -X GET 127.0.0.1:8080/stats/switches
[1, 2, 3]11:42:24
i2cat:(master)~/tid/ryu/ryu/app$ curl -X GET 127.0.0.1:8080/stats/port/1
{"1": [{"tx_dropped": 0, "rx_packets": 7, "rx_crc_err": 0, "tx_bytes": 3906, "rx_dropped": 0, "port_no": 1, "rx_over_err": 0, "rx_frame_err": 0, "rx_bytes": 558, "tx_errors": 0, "collisions": 0, "rx_errors": 0, "tx_packets": 49}, {"tx_dropped": 0, "rx_packets": 42, "rx_crc_err": 0, "tx_bytes": 1116, "rx_dropped": 0, "port_no": 2, "rx_over_err": 0, "rx_frame_err": 0, "rx_bytes": 3348, "tx_errors": 0, "collisions": 0, "rx_errors": 0, "tx_packets": 14}, {"tx_dropped": 0, "rx_packets": 8, "rx_crc_err": 0, "tx_bytes": 3906, "rx_dropped": 0, "port_no": 65534, "rx_over_err": 0, "rx_frame_err": 0, "rx_bytes": 648, "tx_errors": 0, "collisions": 0, "rx_errors": 0, "tx_packets": 49}]}
```

Knowing the dapapath ID is 1 and the port number 2, we can register the alarm in the monitoring modules. Please refer to the API section for more information.

If we want to define a threshold of 200000 bytes/s, (i.e. between 1Mbites/s and 2Mbites/s) and assume that opennaas is running on the same computer (127.0.0.1 on port 8080), Here is the command line to register an alarm:

```
curl -X POST -d '{ "threshold":"200000", "host":"127.0.0.1", "port":"8080", "url_prefix":"/opennaas/test_alarm" }' http://127.0.0.1:8080/XIFI/register_alarm/1/2
```

If the RYU log is activated, you must see the following output:

```
127.0.0.1 - - [29/Jan/2015 11:46:02] "POST /XIFI/register_alarm/1/2 HTTP/1.1" 200 115 0.001919
```

To test the module, use iperf to generate traffic with mininet. If we generate only 1Mbits/ of traffic from H1 to H2, there is no alarm:

```
mininet> h1 iperf -c 10.0.0.2 -u -b 1M -t 10
```

But if we generate 2Mbits/s of traffic from H1 to H2, an alarm is sent by the monitoring module:

```
mininet> h1 iperf -c 10.0.0.2 -u -b 2M -t 10
```

If the logs are activated, a message displaying the alarm is present:

```
127.0.0.1 - - [29/Jan/2015 11:48:55] "POST /opennaas/test_alarm/1/2 HTTP/1.1" 200 115 0.000531
alarm sent for port 2 on datapath ID 1 with current bandwidth at 257457.494282
```

The alarm will be periodically sent until the traffic become lower than the defined threshold or the alarm deleted with the following command:

```
i2cat:(master)~/tid/ryu/ryu/app$ curl -X DELETE  http://127.0.0.1:8080/XIFI/delete_alarm/1/2
```

B.10.1 API Specification

Request to create an alarm:

- Threshold is the threshold of the alarm in bytes/seconds
- Host is the host to send the alarm, so, in our case the, OpenNaaS location

- Port is the tcp port where the OpenNaaS Rest API is located
- Url_prefix is the url where OpenNaaS Rest API is located
- Dpid is the datapath ID of the OVS instance where the virtual port will be monitored (this ID is easily available with the openflow REST API provided by Ryu)
- Port_no is the port number in the ovs instance (in our case br-int) of the virtual port you want to monitor (also available with the ofctl rest api of Ryu)
- All field are mandatory and of string type

Example:

```
curl -X POST -d '{
"threshold":"10",
"host":"192.168.122.201",
"port":"8080",
"url_prefix":"/XIFI/raise_alarm/"
}‘ http://RyuIP:8080/XIFI/register_alarm/{dpid}/{port_no}
```

The alarm will be well registered if the return HTTP value is 200.

To update an alarm, you have to do exactly like creating an alarm register an alarm. If an alarm already exists, it will be updated with the new value.

When an alarm is raise a POST message is send to the url: http://host:port/XIFI/url_prefix/{dpid}/{port_no}.

This request will be done with the same period that the monitoring module use to poll the statistics until the bandwidth will become lower than the threshold of the alarm, or until the alarm is deleted, or updated with a different threshold value.

Request to delete an alarm:

```
curl -X DELETE  http://192.168.122.201:8080/XIFI/delete_alarm/{dpid}/{port_no}
```

To delete an alarm, you need to identify it with the tuple dpid and port_no:

- Dpid is the datapath ID of the OVS instance concerned by you alarm
- Port_no is the port number in this OVS instance where you want to remove the alarm.