Deliverable-6.3

Consolidated software for the use cases and final report on the use cases trials and business impact

Deliverable Editor: Roberto Riggio, CREATE-NET, Jose Enrique Gonzalez Blazquez, ATOS

Publication date:	30-June-2016
Deliverable Nature:	Report
Dissemination level (Confidentiality):	PU (Public)
Project acronym:	PRISTINE
Project full title:	PRogrammability In RINA for European Supremacy of virTualIsed NEtworks
Website:	www.ict-pristine.eu
Keywords:	DIF, management, system, RIB, proof-of-concept

Synopsis: This document reports on the last round of experiments carried out using the PRISTINE SDK and combining several research directions. Results show promising technical progresses in all areas. From the business impact perspective, a qualitative analysis for the three reference PRISTINE use cases has been carried out.

The research leading to these results has received funding from the European Community's Seventh Framework Programme for research, technological development and demonstration under Grant Agreement No. 619305.

List of Contributors

Deliverable Editor: Roberto Riggio, CREATE-NET, Jose Enrique Gonzalez Blazquez, ATOS
CN: Roberto Riggio, Kewin Rausch
ATOS: Juan Vallejo, Jose Enrique, Javier Garcia, James Ahtes, Miguel Angel Puente
i2CAT: Francisco Miguel Tarzan Lorente, Eduard Grasa
TSSG: Miguel Ponce de Leon, Ehsan Elahi, Micheal Crotty
iMINDS: Dimitri Staessens, Sander Vrijders
FIT-BUT: Matej Gregrl, Ondrej Rysavy, Ondrej Lichtner
NEX: Julien Muchembled, Jean-Paul Smets
NXW: Vincenzo Maffione
PNSol: Peter Thompson
TRT: Hamid Asgari, Sarah Haines
UPC: Sergio Leon Gaixas

Executive Summary

The second and last round of experiments carried out as part of the validation work of the PRISTINE SDK has shown promising results. The activities performed during this phase followed two parallel axes. On one hand, we carried out experiments aimed at validating the usefulness of the PRISTINE SDK as far as RINA benefits are concerned. On the other hand, we combined several research directions, designing and executing more complex scenarios that help validate the SDK and RINA at a larger scale. In particular we defined several experiments converging the requirements of the three PRISTINE use cases, making sure that the simultaneous use of policies belonging to different research (such as routing, congestion control, security, scheduling, etc.) areas do not interfere with one another, but rather show at least the same level of performance they did when they were tested individually. These experiments helped in identifying development and integration issues that have been promptly corrected.

Several of the RINA benefits demonstrated during this cycle of experiments apply to all three use cases. For example, multipath routing policies have been used in both the data-centre and in the network service provider scenarios. Similarly, the experiments on security and key management naturally apply to any realistic networking scenario as well as aggregated congestion and performance isolation.

Moreover we also extended the experimentation to the network service providers scenarios testing several use cases related to network function virtualization. During this second phase we also developed and tested new RINA software tools, including an application, NFV Over RIna (NORI), aimed at transporting legacy IP traffic over a RINA network. NORI has been successfully demonstrated at the SDN World Congress.

The development, debugging, testing, and integration of policies associated to this cycle of experiments uncovered a number of bugs in the RINA implementation as well as some changes required in the SDK (which have already been addressed and released in the final version of the SDK, pristine-1.5). There are still a few open issues currently reported. However, most of them are related to design improvements or non blocking bugs. From the business impact perspective, a qualitative analysis for the three reference PRISTINE use cases has been carried out.

Table of Contents

List of Figures

List of acronyms

AES	Advanced Encryption Standard
CACE	Common Application Connection Establishment
CBR	Constant Bit Rate
CDAP	Common Distributed Application Protocol
CRC	Cyclic Redundancy Check
DC	Data Centre
DAF	Distributed Application Facility
DFT	Directory Forwarding Table
DIF	Distributed-IPC-Facility
DMS	Distributed Management System
DTCP	Data Transfer and Control Protocol
DTP	Data Transfer Protocol
ECN	Explicit Congestion Notification
EFCP	Error and Flow Control Protocol
ECMP	Equal Cost Multi-Path
IP	Internet Protocol
IPC	Inter-Process Communication
IPCP	Inter-Process Communication Process
IPCM	Inter-Process Communication Manager
ISP	Internet Service Provider
LFA	Loop Free Alternates
MA/MAD	Management Agent (Daemon)
MLS	Multi-Layer Security
NM	Network Management
NMS	Network Management Service
NSM	Name-Space Manager
NSP	Network Service Provider
PDU	Protocol Data Unit
RED	Random Early Detection
RIB	Resource Information Base
RINA	Recursive Inter-Network Architecture
RMT	Relaying and Multiplexing Task
RTT	Round Trip Time
SDU	Service Data Unit
TCP	Transmission Control Protocol

ToR	Top of Rack
TTL	Time To Live
VLAN	Virtual LAN
VM	Virtual Machine
VNF	Virtual Network Function
VPN	Virtual Private Network

1. Introduction

1.1. Outcomes

During the second and final phase of the project, WP6 carried out several experiments covering different individual research areas. The PRISTINE SDK has been integrated and tested in the three reference uses cases, namely Data-centre Networking, Internet Service Provider, and Distributed Cloud. Ultimately, during the final phase of the project significant efforts have been put towards the completion of a number of tools for researchers and practitioners. These tools include the IRATI demonstrator, aimed at fostering the adoption of RINA while at the same time providing the foundation for reproducible experiments, and the Pristine Configurator, aimed at simplifying the deployment of RINA on the Virtual Wall Testbed.

The experiments carried out during the final phase of the project, include the validation of the PRISTINE SDK into several research areas, that is multi-path routing, security, and management. At the same time, a set of joint experiments, integrating selected policies that are implemented using the PRISTINE SDK, have also been successfully carried out. As for the first phase, these experiments allowed to isolate blocking issues which have been reported and fixed, and provide directions for design enhancements which have been included in the SDK. In relation to the business aspects, a technological-economical analysis has been provided for the three use cases stated above.

1.2. Experiment Coordination and Methodology

This deliverable includes the experiments done during the final phase of the project and reports on experimentation tools developed in this period. This round of experiments has been designed with integration in mind, allowing to test the performance of the combination of different RINA policies in the defined use case scenarios. These experiments build up from the results presented in deliverable D6.2 in order to test their integration in more complex conditions. Additionally, several other individual experiments per research area have been carried out to evaluate new policies and configurations coming from other WPs after the feedback received in the first experimentation phase.

Coordination among partners has been the key for realisation of joint experiments. In order to find the best matching policies to build complex but realistic tests, the following criteria have been taken into account, similar to the principles of the first round of experiments:

- **Relation to use cases and requirements**: Again, the main focus of WP6 is to evaluate results in use cases defined in the project. For that, combination of policies is required to cover as much as possible the real conditions in the proposed scenarios. Moreover, those experiments that have come as the output of research areas from other WPs, were also performed in the scope of the use cases. All of these were required to provide an analysis of the potential technical and business impact.
- **Experiment goals**: Focused on the evaluation of more complex experiments in the use case scenarios, the result of the combination of several policies already validated individually and the development of new policies.
- **Metrics**: For evaluation, the required metrics must be highly relevant to be measured in real scenarios based on PRISTINE use cases.
- **Process**: Reproducibility is essential for external users to replicate the experiments so the detailed explanation of the experimental steps is required. Additionally, special focus has been put on making the environment deployment and configuration as easy as possible.
- **Configuration**: Variables and parameters defined for the experiments, in terms of both RINA policies and scenarios environment, were also necessary.

In addition to the second and final round of experiments gathered in this deliverable, this document also includes feedbacks and recommendations for the future development of RINA as an independent Open Source project.

Finally, to finalise the activities of WP6, this deliverable includes both technical and business impact analysis as a result of the conclusions extracted from each of experiements.

1.3. Organization of Deliverable

This deliverable is organized as follows:

Section 2 describes the tools developed and used for setting up the experimentation environments.

Section 3 covers a summary of policies used in the experiments for each of the defined use cases.

Section 4 provides details for individual experiments per research are. It is based on second cycle developments and feedback from phase one experimentation.

Section 5 gathers the information realted to all of the joint experiments carried out during the second phase experimentation, classified according to the use case in which they were evaluated.

Finally, Section 6 provides a consolidated overview of the second and final phase of experimentation results together with both technical and business impact analysis with the aim of providing assessments and guidelines for any future developments.

2. Testbed Setup and Experimentation Tools

In this section we report on a set of tools and utilities that can be used by researchers and practitioners to experiment with the PRISTINE SDK and with RINA. Such tools include:

- The PRISTINE Configurator, used to quickly deploy large experiments on the Virtual Walls;
- The IRATI Demonstrator, used to test the IRATI stack in a realistic multi-node scenario;
- The RINA Traffic Generator, used to inject synthetic traffic into a RINA network;
- NFV Over Rina, used to create a NFV overlay over a RINA network and to implement VNF Service Function Chains;
- RINA ioq3 demonstrator, a port of the ioquake3 game.

2.1. Configurator

The configurator was developed to be able to quickly deploy larger scale experiments on the iLab.t virtual wall. It was first described in Deliverable 6.2[1].

The purpose of the configurator is to be able to bootstrap a complete RINA network on the iLab.t Virtual Wall facility. The configurator takes as input the following:

- topology.xml: Contains the physical topology of the network. The different nodes in the network should be provided, and the links between them.

```
<?xml version="1.0"?>
<topology>
 <node id="m"/>
 <node id="n"/>
 <node id="o"/>

 <link id="link0">
   <from node="m"/>
   <to   node="n"/>
```

[1] https://wiki.ict-pristine.eu/WP6/d62/d62-configurator

```
 </link>
 <link id="link1">
   <from node="n"/>
   <to   node="o"/>
 </link>
 <link id="link2">
   <from node="m"/>
   <to   node="o"/>
 </link>
</topology>
```

- difs.xml: Contains the different DIFs to be deployed in the network. For every DIF, the type has to be given. In the case of normal DIFs, the DIF template, as defined for the IRATI stack, has to be specified, and also supplied in that directory. In the case of a shim IPC process, the physical (or virtual) link has to specified, on top of which a shim DIF has to be overlayed.

```
<?xml version="1.0"?>
<difs>
 <dif id="normal" type="normal-ipc" template="default.dif" />
 <dif id="eth.shim0" type="shim-eth-vlan" link="link0"/>
 <dif id="eth.shim1" type="shim-eth-vlan" link="link1"/>
 <dif id="eth.shim2" type="shim-eth-vlan" link="link2"/>
</difs>
```

- ipcps.xml: Contains the placement of the IPCPs on the nodes. For every IPCP, the DIF it will be a part of has to be provided. If the IPCP has to use the IPC services provided by other DIFs, it has to specify this in the element “register-dif”.

```
<?xml version="1.0"?>
<ipcps>
 <node id="m">
   <ipcp ap-name="test-eth" ap-instance="1" dif="eth.shim0"/>
   <ipcp ap-name="test-eth" ap-instance="2" dif="eth.shim2"/>
   <ipcp ap-name="ipcp.m" ap-instance="1" dif="normal">
     <register-dif name="eth.shim0"/>
     <register-dif name="eth.shim2"/>
   </ipcp>
 </node>
 <node id="n">
   <ipcp ap-name="test-eth" ap-instance="1" dif="eth.shim0"/>
   <ipcp ap-name="test-eth" ap-instance="2" dif="eth.shim1"/>
```

```
   <ipcp ap-name="ipcp.n" ap-instance="1" dif="normal">
     <register-dif name="eth.shim0"/>
     <register-dif name="eth.shim1"/>
   </ipcp>
 </node>
 <node id="o">
   <ipcp ap-name="test-eth" ap-instance="1" dif="eth.shim1"/>
   <ipcp ap-name="test-eth" ap-instance="2" dif="eth.shim2"/>
   <ipcp ap-name="ipcp.o" ap-instance="1" dif="normal">
     <register-dif name="eth.shim1"/>
     <register-dif name="eth.shim2"/>
   </ipcp>
 </node>
</ipcps>
```

- vwall.ini: Contains configuration parameters to access the iLab.t Virtual Wall facility. This includes the wall to use (wall 1 or wall 2), the user's credentials (logging in with an SSH key is also supported), the project and experiment name and the image to deploy on every node.

```
[vwall_config]
wall = wall2.ilabt.iminds.be
username = sander
password = <password>
proj_name = PRISTINE
exp_name = 3nodes
image = PRIST-bee1658-NOLOG
```

- apps.xml (optional): Contains the mapping of certain applications to a DIF in order to force them to only use that DIF. A better way is to give this as a parameter to the application at runtime.

```
<?xml version="1.0"?>
<apps>
   <app ap-name="rina.apps.echotime.server" ap-instance="1">
     <node name="m">
       <register dif-name="normal.DIF"/>
     </node>
   </app>
</apps>
```

Once this input is successfully parsed, a new experiment is created on the Virtual Wall based on the specified physical topology. If the experiment

already exists, it is not re-created. To achieve this goal, configurator generates an NS script, which is required by the Virtual Wall when creating a new experiment:

```
set ns [new Simulator]
source tb_compat.tcl
set m [$ns node]
tb-set-node-os $m PRIST-dc38de9f-NOLOG
set n [$ns node]
tb-set-node-os $n PRIST-dc38de9f-NOLOG
set o [$ns node]
tb-set-node-os $o PRIST-dc38de9f-NOLOG
set link0 [$ns duplex-link $m $n 1000Mb 0ms DropTail]
set link1 [$ns duplex-link $n $o 1000Mb 0ms DropTail]
set link2 [$ns duplex-link $m $o 1000Mb 0ms DropTail]
$ns run
```

This will create the following topology:

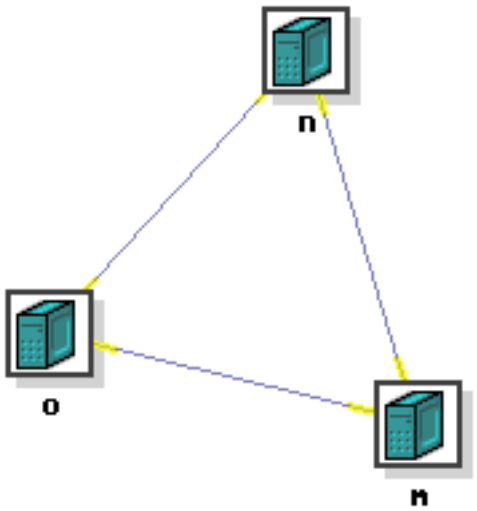


Figure 1. 3-node experiment

Then, the experiment is swapped in. Once again, if it is already swapped in, configurator just continues to the next step. After it is swapped in, configurator will access every node to obtain the OS specific name of the interface for a particular link. As an example, for link0 node m's specific OS name of the interface is eth56. This is needed in order to be able to auto-generate the DIF templates for the shim DIF for Ethernet. Configurator

also assigns a VLAN id to every link, since this is required for the shim DIF for Ethernet. It then creates the correct VLAN interface for every VLAN on the appropriate nodes. The kernel modules that are required by the IRATI stack are also inserted on every node. Next, for every node, the configuration files are generated and copied to the configuration directory of the IRATI stack on every node. Finally, on every node, the IPC Manager is started.

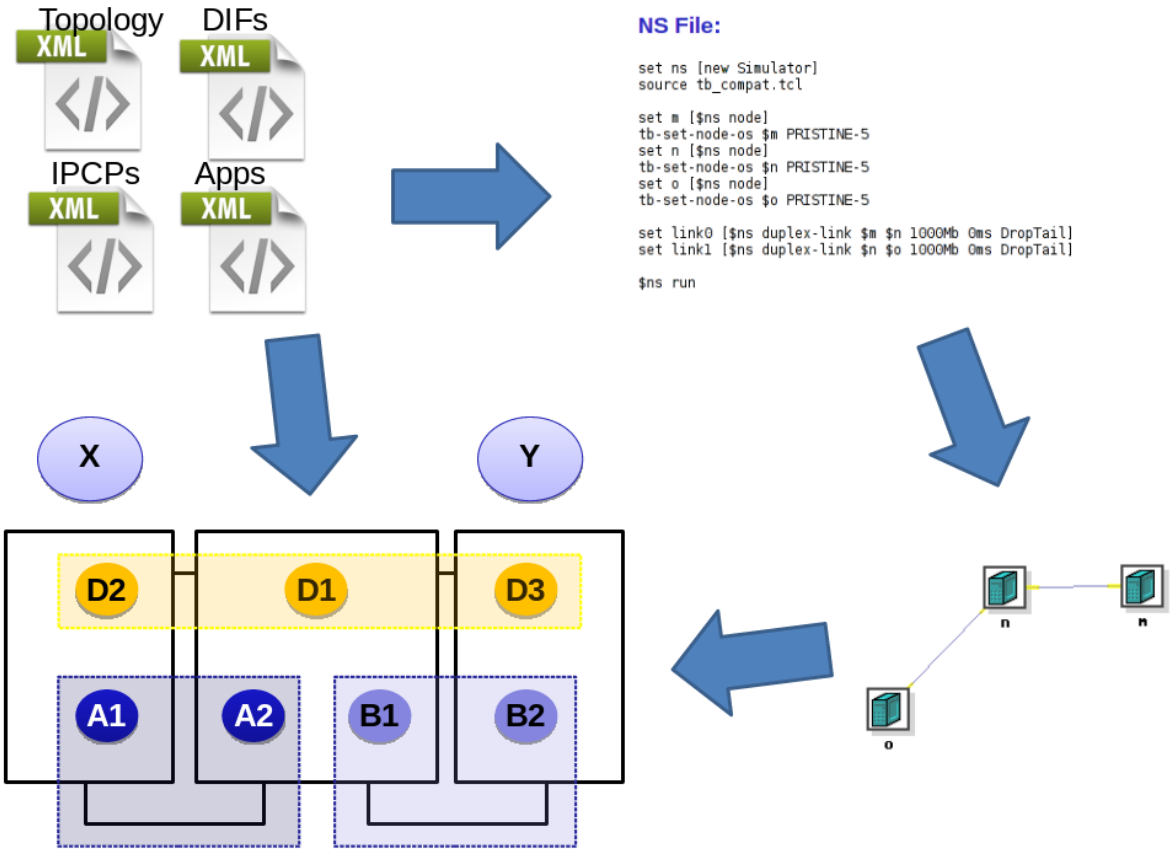


Figure 2. Configurator tool

To build "configurator", it is necessary to execute the following commands in the main directory:

```
./bootstrap
./configure --prefix=/path/to/irati/
make
```

To run it, one has to either install it (make install) or go into the src/ directory and run the confgen script. Confgen takes the following parameters by command line:

```
[sander@Faust src]$ ./confgen --help
usage: confgen [-h] [--topology FILE] [--ipcps FILE] [--apps FILE]
               [--difs FILE] [--vwall-conf FILE] [--output-dir DIR]
               [--prefix FILE] [--input-dir DIR]

RINA Configuration Generator

optional arguments:
 -h, --help          show this help message and exit
 --topology FILE     the topology XML filename (default: ['topology.xml'])
 --ipcps FILE        the IPC Process XML filename (default: ['ipcps.xml'])
 --apps FILE         the applications XML filename (default: ['apps.xml'])
 --difs FILE         the DIFs XML filename (default: ['difs.xml'])
 --vwall-conf FILE   the Virtual Wall INI filename (default: ['vwall.ini'])
 --output-dir DIR    the output dir of the XMLs (default: ['configs'])
 --prefix FILE       the prefix of the name of the output files (default:
                     ['ipcmanager'])
 --input-dir DIR     the input dir of the config files (default:
                     ['inputs/2nodes'])
```

Configurator has the following restrictions:

- Enrollment is not yet automated. You have to perform enrollment yourself once the IPCMs are started.
- Only the shim-eth-vlan is supported. Extending the support to also include the shim-tcp-udp would require minimal effort. On the other hand, adding support for the shim-hv it is a more complicated process, since it would also mean instantiating new virtual machines on the Virtual Wall.

Subsequent updates were done to align the configurator with the integrated version of the IRATI prototype.

These changes include

- Updating the configurator to comply with the latest version of the configuration files for the IRATI prototype. The DIF configurations used by the configurator have also been updated.
- Provide shell scripts for performing enrollment for the integrated scenarios.

2.2. IRATI Demonstrator

2.2.1. Overview

This IRATI demonstrator is an open source command-line tool which allows the user to easily try and test the IRATI stack in a multi-node scenario. This tool comes as a consistent evolution of the all-in-one-machine testbed reported in Deliverable 6.2[2]. This section is meant to illustrate the purpose and features of the demonstrator, and therefore to act as an user manual for the tool, as it is at the end of PRISTINE. For this reason, the description reported here is complete and self-contained, and it is not related to the description of the all-in-one-machine tool in D6.2, which is to be considered obsolete. In any case, in order to illustrate the effort that has been put in the development of the demonstrator, a short summary of the changes with reference to the all-in-one-machine testbed is reported at the end of the section.

The purpose of the demonstrator is twofold:

- Allow people interested in RINA to trial the IRATI stack, which includes the PRISTINE Software Development Kit.
- Help IRATI developers and software release engineers to carry out integration and regression tests.

For the first kind of users, no knowledge about how to compile and install IRATI is required. Everything the user need is self-contained in this repository and is explained in this document.

Each node is emulated using a Virtual Machine (VM), run under the control of the QEMU hypervisor.

All the user has to do is to prepare a configuration file which describes the scenario to be demonstrated. This requires the user to specify all the Layer 2 connections between the nodes and all the DIFs which lay over this L2 topology. A DIF can be stacked over other DIFs, and arbitrary levels of recursion is virtually supported by the tool (be aware that the IRATI stack may place restrictions on the recursion depth, so the scenario bootstrap may fail if one uses too many levels of recursion).

[2] https://wiki.ict-pristine.eu/WP6/d62/d62-AI1TB

The syntax of the configuration file is detailed in Section 2.2.4.

Using the -g option, the tool is able to generate an image depicting the graph of all normal DIFs, showing the lower-level DIFs connecting them, as showed in Figure 3

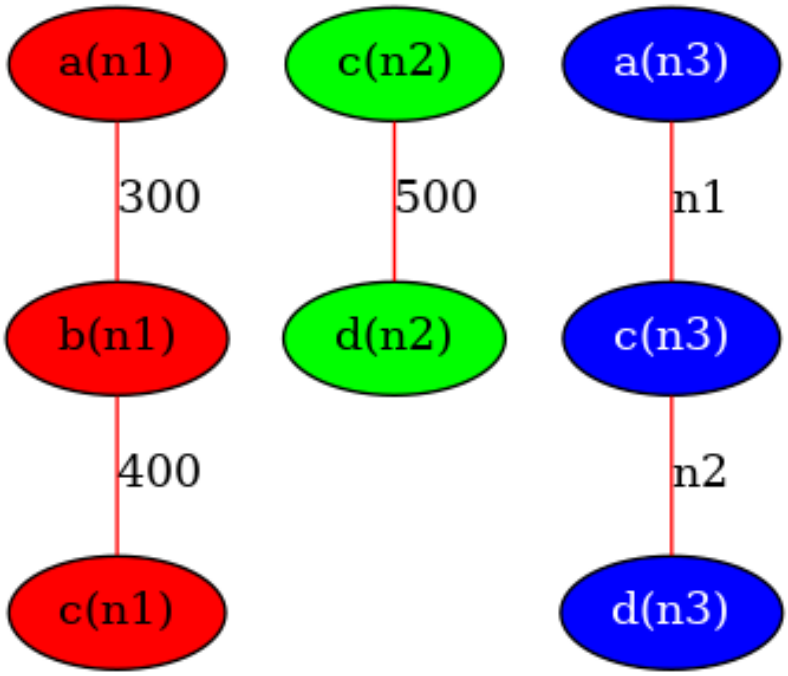


Figure 3. Image produced by the demonstrator representing the DIF graphs.

In this example (that can be found in the examples/two-layers.conf file in the demonstrator repository), the topology includes 4 nodes, named `a`, `b`, `c` and `d`. Each subgraph depicted represents a different normal DIF, with all the nodes that take part to the DIF. The `n1` DIF is depicted with red nodes, the `n2` DIF with the green nodes, and the `n3` DIF with the blue nodes. The links represent the N-1-DIFs used to perform the enrollments, that are shown as link labels.

Run

```
$ ./gen.py -h
```

to see all the available options.

The code is publicly available at https://github.com/irati/demonstrator, as part of the IRATI GitHub organisation.

During the last 6 months of PRISTINE, the demonstrator has also taken an important role in the dissemination of project results. It has been used

by PRISTINE partners to show the IRATI stack in action, with specific PRISTINE policies, in non-trivial scenarios (up to 40 nodes, three levels of DIFs). In particular, two of the demos shown to the audience during the final PRISTINE workshop where based on the demonstrator. The success of this tool is due to (i) its simplicity of use, i.e. the concision of its configuration file; and (ii) its portability, since the VM image used to run the IRATI stack are contained in the repository itself. PRISTINE partners can easily distribute complex demonstrator scenarios by publishing a small demonstrator configuration file.

2.2.2. Workflow

Once the configuration file has been prepared, the user can invoke the tool

```
$ ./gen.py -c /path/to/config/file
```

which will generate two bash scripts: up.sh and down.sh.

Running the up.sh script will bootstrap the specified scenario, which involves the following operations:

- Create TAP interfaces and linux software-bridges to emulate the specified L2 topology.
- Run the VMs emulating the nodes.
- Bootstrap the IRATI stack on each node, with proper configuration (IPCM configuration, DIF templates, DIF Allocator map, ...).
- Perform all the enrollment, at all DIF layers, respecting the dependency order.

The up.sh script reports verbose information about ongoing operations. If everything goes well, one should be able to see the script reporting about successful enrollment operations right before terminating.

Once the bootstrap is complete, the user can access any node and play with them (e.g. running the rina-echo-time test application to check connectivity between the nodes).

The tool can work in two different modes: buildroot (default) and legacy. The way one accesses the nodes depends on the modes. In any case, unless

one knows what he/she is doing, the default buildroot mode is preferable. More information about the legacy mode can be found in Section 2.2.6.

To undo the operations carried out by the up.sh, the user can run the down.sh script (this will terminate all VMs).

The user can run up.sh/down.sh multiple times, and use gen.py only when he/she wants to modify the scenario. Before generating a new scenarios it is important to run the down.sh script, otherwise a full system reboot may be necessary in order to clean-up leftover artifacts.

By default every VM is assigned 128 MB of memory, so with 4GB of memory it is possible to execute up to 32 nodes. It is also possible to reduce the per-VM memory to achieve higher density.

2.2.3. Hardware and software requirements

- [HW] An x86-64 processor with hardware-assisted virtualization support (e.g. Intel VT-X or AMD-V).
- [SW] Linux-based Operating System.
- [SW] QEMU, a fast and portable machine emulator.
- [SW] brctl command-line tool (usually found in a distro package called bridge-utils or brctl).

2.2.4. Scenario configuration file syntax

The configuration file for gen.py contains a list of declarations, with one declaration per line. Lines starting with '#' are ignored by the tool so that they can be used for comments.

There are a few types of declarations:

- **eth**, to specify Layer 2 connections between nodes.
- **dif**, to specify how normal DIFs stack onto each other.
- **policy**, to specify non-default policies for normal DIFs.
- **appmap**, to specify static application-to-DIF mappings for normal DIFs.
- **overlay**, to specify a per-node directory to be overlaid on the node file system.

- **netem**, to specify a per-node, per-shim-dif link emulation features (delay, loss, duplicate packets, etc.).
- **enroll**, to specify manual enrollment, in case the automatic enrollment strategies do not meet the user requirements.

Each type of declaration may occur many times. Note that nodes are implicitly declared by means of **eth** and **dif** lines: there don't have an separate explicit declaration type.

The repository contains some example of scenario configuration files:

```
gen.conf, examples/star.conf, examples/two-layers.conf, ...
```

with comments explaining the respective configuration.

The following example (two-layers.conf) is taken from the `examples` directory of the demonstrator repository.

```
# Three links connecting four nodes
eth 300 0Mbps a b
eth 400 0Mbps b c
eth 500 0Mbps c d

# DIF n1 spans a,b and c and runs over the shims
dif n1 a 300
dif n1 b 300 400
dif n1 c 400

# DIF n2 spans c and d and runs over the shims
dif n2 c 500
dif n2 d 500

# DIF n3 spans over n1 and n2
dif n3 a n1
dif n3 c n1 n2
dif n3 d n2

policy n2 security-manager pubkey alg=RSA
policy n1 rmt.pff lfa
```

This simple configuration specifies a four nodes linear topology, with two levels of normal DIFs. Two non-default policies are specified for DIFs n1 and n2.

eth declarations

An **eth** declaration is used to specify an L2 (Ethernet) connection between two or more nodes. A star topology - one with a central L2 switch - is used to connect together the specified nodes. Consequentely, each **eth** declaration corresponds to a separate Ethernet L2 broadcast domain.

Each L2 domain is identified by a different VLAN number, which will be used by the IRATI stack as a name for the Shim DIFs over 802.1q, and that will be used to configure the VLAN on the nodes' interfaces.

The syntax for the **eth** declaration is as follows:

```
eth VLAN_ID LINK_SPEED NODE_NAME...
```

where

- VLAN_ID is an integer between 1 and 4095, identifying the L2 domain.
- LINK_SPEED indicates the maximum speed for this L2 domain (e.g. 30Mbps, 500 Kbps, 1Gbps, ...), which is implemented by rate-limiting the TAP interfaces, outside the VMs. If 0Mbps is specified, no rate-limiting is used.
- NODE_NAME is an identifier for a node, which can be any non-space character. Two or more node can be specified, separated by spaces.

dif declarations

A **dif** declaration gives information about how a single node participates in a specific N-DIF. It is used to specify what N-1 (lower) DIFs are used by the node to participate in the N-DIF.

Consequently, for each normal DIF there will be a separate **dif** declaration for each node taking part in that DIF.

The syntax for the **dif** declaration is as follows:

```
dif DIF_NAME NODE_NAME LOWER_DIF_NAME...
```

where

- DIF_NAME is the name of the N-DIF under specification.

- NODE_NAME is the name of the node that takes part in DIF_NAME.
- LOWER_DIF_NAME is the name of an N-1-DIF (either Shim or normal) which is used by NODE_NAME to connect to its neighbors in the N-DIF. One or more N-1-DIFs can be specified, separated by space. Having more N-1-DIFs usually happens when a node has multiple neighbors.

policy declarations

A **policy** declaration is used to instruct the tool to setup a particular non-default *policy-set* for a single IRATI component in a specific DIF. The policy-set is an IRATI-specific object that groups together all the policies belonging to a certain IRATI component (which maps to a component of the RINA architecture). A policy-set is logically associated with an instance of an IPCP component in a certain DIF.

Consequently, for each normal DIF there will be a separate **policy** declaration for each IRATI DIF component that needs a non-default policy-set.

The syntax for the **policy** declaration is as follows:

```
policy DIF_NAME ( * | NODE_NAME_1,NODE_NAME2,... ) COMPONENT_PATH
 POLICY_SET_NAME [PARAM1=VALUE1 PARAM2=VALUE2 ...]
```

where

- DIF_NAME is the name of the DIF under specification.
- The second argument can be '*' or a list of comma-separated node names. In the former case, the policy is deployed on all nodes in the DIF, while in the latter the policy is deployed only in the listed nodes.
- COMPONENT_PATH is a string identifying the DIF component under specification. The string syntax is the same one used by the PRISTINE SDK to identify components (e.g. "rmt.pff" to indicate the PDU Forwarding Function component).
- POLICY_SET_NAME is the name of a policy-set to use for the specified component in the specified DIF. The name must correspond to the one declared in some plugin's manifest file.
- A list of name/values couples is used to specify policy-set parameters.

appmap declarations

An **appmap** declaration is used to specify a static mapping of an application name to a normal DIF. This static configuration is used by the DIF allocator during the flow allocation procedure.

The syntax is as follows:

```
appmap DIF_NAME AP_NAME AP_INSTANCE
```

where

- DIF_NAME is the name of the DIF that the applications name maps to.
- AP_NAME is the application process name of the mapped application.
- AP_INSTANCE is the application process instance of the mapped application.

overlay directive

An **overlay** directive is used to specify a directory on the host file system to be overlaid on the file system of a specific node. This directive overrides the behaviour of the --overlay option, because the --overlay option is processed (i.e. the files copied on the node file system) before the directive is processed.

The syntax is as follows:

```
overlay NODE_NAME OVERLAY
```

where

- NODE_NAME is the name of the node to which the overlay applies.
- OVERLAY is the path (absolute or relative to the path of the demonstrator command line tool) of the overlay directory tree on the host file system.

netem directive

An **netem** directive allows link emulation, using the **netem** features of the Linux traffic control framework. Any valid netem command can be

used (e.g. see http://man7.org/linux/man-pages/man8/tc-netem.8.html). However, it is not recommended the use of the term netem rate command unless one knows what he/she is doing. The recommended way to add rate limiting is to specify a non-zero link speed value in the **eth** directive.

The syntax is as follows:

```
netem SHIM_NAME NODE_NAME NETEM_COMMAND
```

where

- SHIM_NAME is the name of the Shim DIF where link emulation applies.
- NODE_NAME is the name of the node for which the link must be emulated for the specified link.

Example to add delay and packet loss to node xyz in shim wan.DIF:

```
netem wan.DIF xyz delay 100ms loss random 0.1%
```

enroll directive

A list of **enroll** directives may be specified when the user does not want to use the automatic enrollment strategies supported by the demonstrator (e.g. minimal, full-mesh, ...). If the user does not know how to use it properly, it is recommended to not use the **enroll** directive and to rely on the automatic (default) strategies.

The syntax of a directive is as follows:

```
enroll DIF_NAME NODE_NAME NEIGH_NODE_NAME N_1_DIF
```

where

- DIF_NAME is the name of the DIF where the enrollment should happen.
- NODE_NAME is the name of the enrollee node.
- NEIGH_NODE_NAME is the name of the enroller node.

- N_1_DIF is the name of the N-1-DIF in common between enrollee and enroller, to be used for the enrollment procedure.

2.2.5. Buildroot mode

When using the tool in buildroot (default) mode, things are straightforward.

Once the user has run up.sh, he/she can access the node named "XYZ" using:

```
$ ./access.sh XYZ
```

Each node runs in a minimal VM environment, which is an RAM filesystem (initramfs) created using the buildroot framework (https://buildroot.org/). All the IRATI software and its dependencies (except for the kernel image) are packed into about 30 MB.

A snapshot of both the kernel image and the file system the are available in the buildroot directory of this repository. The SHA identifier of the IRATI stack built into the current image is 12d9cf0da612b57597581d16b0bfdc466947cc0a..

Be aware that any modifications done on the VM filesystem are discarded when the scenario is torn down.

IRATI mini-tutorial

This document is **not** a tutorial on how to use the IRATI stack. However, this section describes some basic tests one can do in order to check that things are working as they are supposed to work.

Generate a simple three-node scenario, with two Shim DIFs over 802.1q and one normal DIF laying over those:

```
$ ./gen.py -c gen.conf
```

Run the up.sh script to boostrap the scenario

```
$ ./up.sh
```

waiting for it to finish (it may take tens of seconds).

Access node "a" and run rina-echo-time in server ping mode

```
$ ./access.sh a
# rina-echo-time -l
```

Using a different terminal, access node "c" and run rina-echo-time in client ping mode, sending 10 packets to the server.

```
$ ./access.sh c
# rina-echo-time -c 10
```

Once the experiment terminates, use CTRL-C to stop the rina-echo-time server, and issue the "exit" command on both terminals to exit from the nodes.

Run the down.sh script to tear down the scenario

```
$ ./down.sh
```

That's it, you successfully ran a ping application in the RINA world!

2.2.6. Legacy mode

The use of this mode is not recommended.

When using the legacy mode, the user is not using the buildroot-generated kernel and filesystem, already available in the repository. Instead, the user must build his/her own VM disk image, which must contain the IRATI kernel and user-space software.

The disadvantage of this approach is that it is hard to build such an image using less than 4 GB. See Section 2.2.5 for a better approach.

In addition to the requirements specified in Section 2.2.3, you need a QEMU VM image containing:

- The IRATI stack installed.
- The bootloader configured to boot the IRATI kernel.

- Python.
- sudo enabled for the login username on the VM (referred to as ${username} in the following), with NOPASSWD, e.g. /etc/sudoers should contain something similar to the following line:

```
% wheel ALL=(ALL) NOPASSWD: ALL
```

Instructions, to be followed in the specified order

1. Edit the gen.env file to set the IRATI installation path (on the VM filesystem), the path of the QEMU VM image (on your physical machine file system) and the login username on the VM (${username}).
2. Specify the desired topology in gen.conf.
3. Run ./gen.py to generate bootstrap and teardown script for the topology specified in (2).
4. Use ./up.sh to bootstrap the scenario.
5. VMs are accessible at localhost ports 2223, 2224, 2225, etc. e.g. ssh -p2223 ${username}@localhost.
6. Perform the tests on the VMs using ssh (5).
7. Shutdown the scenario (destroying the VMs) using ./down.sh.
8. VMs launched by ./up.sh have a non-persistent disk which means that any change will be lost when the node is reboot or shutdown. To make persistent modifications to the VM image (e.g. to update the PRISTINE software), run ./update_vm.sh and access the VM at ssh -p2222 ${username}@localhost. Don't try to run ./update_vm.sh while the test is running (i.e. you've run ./up.sh but still not run ./down.sh).

Automatic login to the VMs

In order to automatically login to the VMs, it's highly recommended to use SSH keys[3]. In this way it is possible to avoid inserting the password again and again (during the tests):

```
$ ./update_vm.sh
```

[3] http://serverfault.com/questions/241588/how-to-automate-ssh-login-with-password

```
$ ssh-keygen -t rsa  # e.g. save the key in /home/${username}/.ssh/
pristine_rsa
$ ssh-copy-id -p2222 ${username}@localhost
$ shutdown the VM
```

Now one should be able to run ./up.sh without the need to insert the password.

2.2.7. Summary of improvements over the previous version

The demonstrator tool is a feature-rich demonstrator tool for the IRATI stack and PRISTINE policies, which goes far beyond the primitive all-in-one-machine testbed.

The following list contains some of the features added with reference with the previous version:

- Support for Buildroot-generated VM images containing the IRATI stack and related tools. Buildroot images are small enough that they can be stored in the demonstrator repositories, so that the user does not need to build an image from scratch.
- Completely automated workflow and user-friendly interface. The user does not need anymore to manually copy any file inside the VM image.
- Support for arbitrary stacking of DIFs (the previous tool only supported a single normal DIF on top of shim DIFs over ethernet).
- Simplified specification of the physical topology. The user does not need to explicitly declare nodes, links and bridges. The **eth** directive is able to describe the same information more synthetically.
- Support for different enrollment strategies: minimum spanning tree, full-mesh, manual
- Support for custom PRISTINE policies, plugged in leveraging on the SDK.
- Integration with the graphviz tool, used to draw the graph of each DIF specified by the configuration file.
- Support for auto-generated parametric topologies (e.g. a ring topology with N nodes), for which the user does not even need to specify a configuration file.

- Support for different emulated NICs in the VMs (e1000, virtio-net, virtio-net with vhost acceleration).
- Ability to specify the IRATI log level for all the VMs.
- Ability overlay per-VM or global user-specified directory trees, that are automatically copied from the host file system to the running VM images. This is useful to overlay configuration files, key material, etc.

2.3. RINA traffic generator tool

The RINA traffic generator rina-tgen was first described in Deliverable 6.2[4].

The RINA traffic generator (rina-tgen) was inherited from the GN3+ IRINA[5] OC project. PRISTINE experimentation sets new requirements for rina-tgen, for which it has undergone some changes:

- During PRISTINE, a change in the IPC API was included that has the flow allocation function return a `<port_id>` instead of a `<Flow *>`. rina-tgen was updated for compatibility with this new API.
- A major code refactoring was conducted to make the tool more extensible. It includes function prototypes for registering the application with multiple DIFs (multi-homing).
- Statistics are now reported per port_id, over a timed interval and a packets per second (p/s) metric was added
- An option was added to write output as comma-separated-values (.csv) for easy parsing and analysis, which generates a `.csv` file per connected client and per performed test for easily tracking multiple concurrent clients connected to a single server.
- The build system was improved with dependency detection of the BOOST C++ libraries.
- The client and server were extended with non-blocking I/O options for improved accuracy in the server application and the handling of packet loss. To include this functionality, the non-blocking I/O in IRATI needed revision, performed in WP2.

[4] https://wiki.ict-pristine.eu/WP6/d62/d62-tgen
[5] http://www.geant.org/Projects/GEANT_Project_GN4/Pages/Home.aspx

The RINA traffic generator is available at https://github.com/IRATI/traffic-generator

```
USAGE:

   ./rina-tgen  [-o <string>] [-l] [--interval <unsigned integer>] [-c
                <unsigned integer>] [--duration <unsigned integer>] [--rate
                <unsigned integer>] [--timeout <unsigned integer>] [-s
                <unsigned integer>] [--distribution <string>]
                [--poissonmean <double>] [--qoscube <string>] [-d <string>]
                [--client-api <string>] [--client-apn <string>]
                [--server-api <string>] [--server-apn <string>] [-r]
                [--sleep] [--] [--version] [-h]


Where:

   -o <string>,  --output-path <string>
     Write csv files per client to the specified directory, default = no
     csv output.

   -l,  --listen
     Run in server (consumer) mode, default = client.

   --interval <unsigned integer>
     report statistics every x ms (server), default = 1000.

   -c <unsigned integer>,  --count <unsigned integer>
     Number of SDUs to send, 0 = unlimited, default = unlimited.

   --duration <unsigned integer>
     Duration of the test (seconds), 0 = unlimited, default = 60 s IF count
     is unlimited.

   --rate <unsigned integer>
     Bitrate to send the SDUs, in kb/s, 0 = no limit, default = no limit.

   --timeout <unsigned integer>
     Time for a test to timeout from client inactivity (ms), default =
     10000 ms

   -s <unsigned integer>,  --size <unsigned integer>
     Size of the SDUs to send (bytes), default = 500.

   --distribution <string>
     Distribution type: CBR, poisson, default = CBR.
```

```
--poissonmean <double>
  The mean value for the poisson distribution used to generate
  interarrival times, default is 1.0.

--qoscube <string>
  Specify the qos cube to use for flow allocation, default = unreliable.

-d <string>,  --dif <string>
  The name of the DIF to use, empty for any DIF, default = empty (any
  DIF).

--client-api <string>
  Application process instance for the client, default = 1.

--client-apn <string>
  Application process name for the client, default =
  traffic.generator.client.

--server-api <string>
  Application process instance for the server, default = 1.

--server-apn <string>
  Application process name for the server, default =
  traffic.generator.server.

-r,  --register
  Register the application with the DIF, default = false.

--sleep
  sleep instead of busywait between sending SDUs, default = false.

--,  --ignore_rest
  Ignores the rest of the labeled arguments following this flag.

--version
  Displays version information and exits.

-h,  --help
  Displays usage information and exits.
```

2.4. NORI: Nfv Over RIna

2.4.1. Description

NFV over RINA (NORI) is a software developed in order to run legacy IP applications over RINA. Notice however that, the general concept which

lies at the base of NORI is quite generic and can be easily extended to accommodate other types of traffic (e.g., Ethernet one). NORI allows to transport IP traffic over a RINA network without requiring any change to the original applications. This feature is particularly interesting to support Network Function VIrtualization (NFV) use cases where legacy IP —aware Virtual Network Functions (VNFs) are to be interconnected using a RINA transport network. In such a scenario any NFV—based network service can easily take advantage of any RINA policy currently developed by PRISTINE. An high level representation of the NORI architecture is sketched in Figure 4.

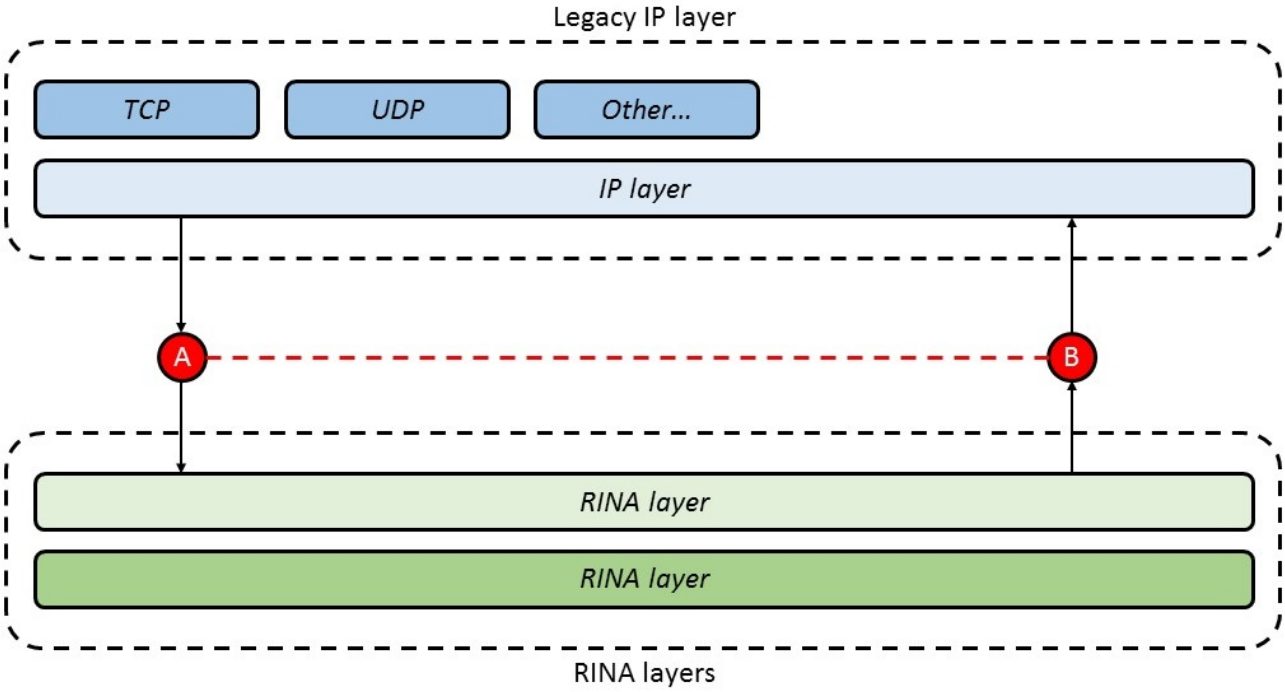


Figure 4. How NORI link upper legacy IP layers with lower recursive architecture logic.

NORI does not require any change to the existing TCP/IP stack and presents itself to RINA (**south-bound** interface) as an Application Entity which needs to exchange information with a single or multiple other instances of NORI (depending on how the software is instructed). This is completely transparent to the IP layer which is unaware on what will happen to the traffic flowing across that interface. At the same time the IRATI stack does not need to know that type of traffic being exchanged between the AEs.

NORI **north-bound** interface is represented by a TUN/TAP device which is created in the moment the application instance starts. Such interface can

be configured by the network administrator. Once the interface has been configured, IP applications can start to use it as a normal network interface.

What is left for the communication to succeed is a translation between the two different naming systems, namely RINA and IP. This is possible thanks to a set of rules that are feed to NORI during its initialization. Such rules allows the administrator to specify which AEs a precise portion of the flowspace shall be routed across.

The rules are applied only to outgoing traffic, since we assume that all the packets exchanged between these APs are generated by other NORI instances. The entire process is sketched in Figure 5.

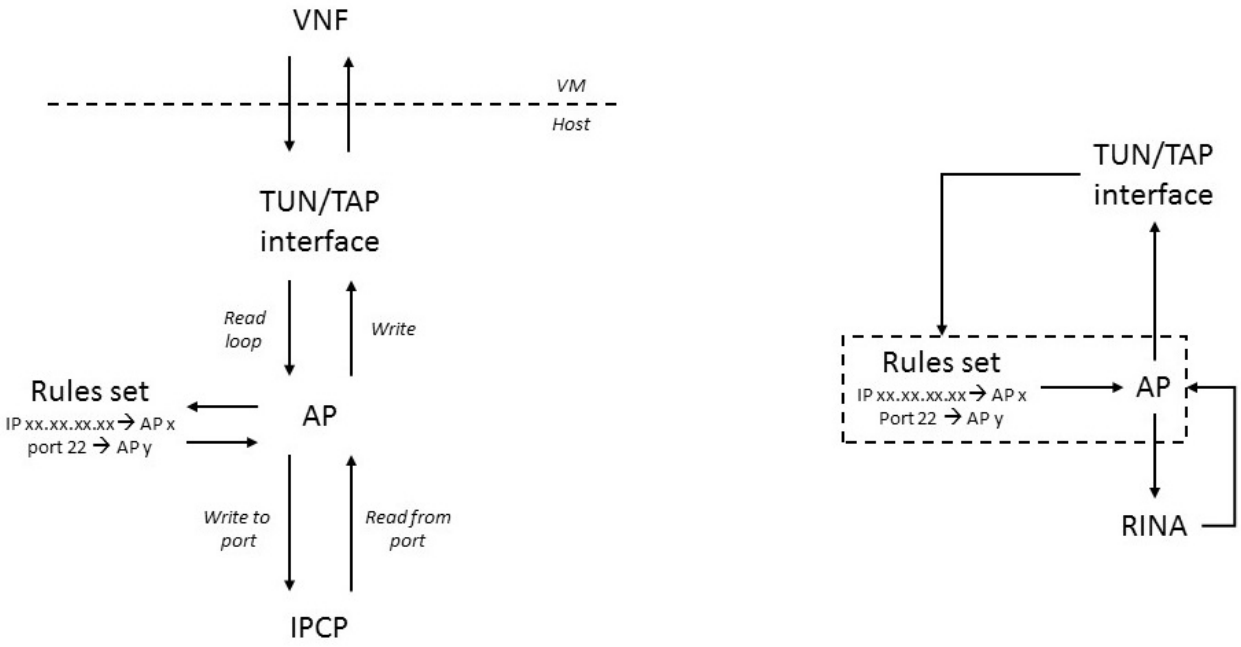


Figure 5. How NORI translates between the two naming system to determine where the packet should go.

2.4.2. Dictionary

The NORI forwarding rules are stored in a dictionary following a very simple syntax. The dictionary currently support only IPv4 traffic and allows matching traffic flows using the following fields: source IP address, destination IP address, transport protocol type (TCP or UDP), and transport source and destination ports. An additional rule called 'default' is used to match all the traffic. NORI applies the rules in the order they are specified in the dictionary. If a default rule is specified as last entry then all the traffic that does not match the previous rules will be treated using the default rule.

If no default rule is specified then all the traffic that does not match the rules in the dictionary will be dropped.

Here follows an example of a NORI dictionary:

```
ip src 192.168.1.1 s3,1
ip dst 127.33.151.248 s4,1
port dst UDP 12345 s5,1
port src UDP 12345 s6,1
port dst TCP 2234 s7,1
port src TCP 2234 s8,1
default si s2,1
```

The quick reference for the supported rules is:

- default rule, `default <si,rr> <AE_name, AE_instance>*`
- port rule, `port <src|dst> <UDP|TCP> <number> <AE_name, AE_instance>`
- ip rule, `ip <src|dst> <address> <AE_name, AE_instance>`.

2.4.3. Run NORI

NORI can be run as any other command line application from a terminal. The application needs to have root access in order to be able to create the tun/tap device. Invoking the application without any command line arguments will trigger the help. The NORI syntax is the following:

```
./nori <AE_name> <AE_instance> <DIF_name> <rule_dictionary>
```

This will create a NORI instance, which blocks the terminal if the command is not issued with `&` token at the end. To stop the application in the proper way you need to issue a `CTRL-C` signal to the NORI console (and wait for NORI to gracefully exit). If the application is successfully created, then the user will be able to see a new `tunX` interface (the number depend on the system) together with the other network interfaces of the machine. The run interface must then be configured with an ip address and a netmask otherwise the received traffic will be discarded by the IP network stack.

2.5. RINA-ioq3

2.5.1. Introduction

rina-ioq3 is a IRATI-capable port of ioquake3, which is an Open Source community effort to continue improving the GPLv2 licensed source code of the classic multiplayer game Quake 3 Arena. rina-ioq3 was originally developed as a proof of concept by Addy Bombeke at Ghent University as part of his master thesis on RINA.

PRISTINE decided to adopt the software as a demonstration tool, and subsequent development and stabilisation of the proof-of-concept prototype to a stable demo-ready version was carried out as part of WP6.

2.5.2. ioq3

ioquake3 is a free software first person shooter engine based on the Quake 3: Arena and Quake 3: Team Arena source code. The source code is licensed under the GPL version 2, and was first released under that license by id software on August 20th, 2005. The goal is to create the open source Quake 3 distribution upon which people base their games, ports to new platforms, and other projects. The engine runs Quake 3: Arena, Team Arena, and all popular mods on modern platforms like Linux, Windows 7, 8, and 10, Mac OS X Mavericks and Mac OS X Yosemite. http://ioquake3.org.

2.5.3. rina-ioq3

The objective of the master thesis was to investigate the difficulty of porting existing TCP/IP applications (POSIX sockets) to the IRATI IPC API. Our choice fell with ioquake3 since it was Open Source, GPL licensed and an initial feasibility assessment of the code turned out favourable (the networking code was well structured and contained within a few source files).

A nice overview of how Quake 3 is structured can be found at http://fabiensanglard.net/quake3/

At the time of the master thesis, the following features needed implementation.

On the IRATI side:

- A wrapper for C, since Quake3 was written in C (and a special language called QuakeC for the game engine) and librina is written in C++.

On the rina-ioq3 side:

- Wrapping the IRATI `read_sdu` calls (which at the time were always blocking) in a non-blocking API.
- Creating the correct calls within the game server and client to make use of the librina API.

The master thesis delivered a proof-of-concept prototype that was demonstrated at the Pisa plenary meeting. The code was contributed to the IRATI repository in-kind (this development was outside of the PRISTINE project).

2.5.4. rina-ioq3: PRISTINE developments

Following the demonstration of the tool, PRISTINE decided to adopt the software for public demonstration purposes. While the master thesis developed a working version of the game, there were instabilities in the code that did not allow it to be used for public demonstration purposes. Among others:

- Incomplete handling of exceptions (C doesn't allow C++ style Exceptions) caused crashes (both predictable and unpredictable)
- PRISTINE developed native support for non-blocking I/O in the IRATI prototype, which is more stable than the implementation that was present in ioq3 based on the IRATI blocking I/O.
- The split of functionality between the C wrapper and ioq3 internals needed revision and most functions were moved to the C wrapper.

2.5.5. PRISTINE C Wrapper

The PRISTINE C Wrapper was redesigned to serve as a basis for future ports and a POSIX-style API for RINA prototypes exposing file descriptors to the application process. We will explain its functions using the header file:

```
#ifndef LIBRINA_C_H
#define LIBRINA_C_H
```

```
#ifdef __cplusplus
extern "C"
{
#endif

#include <stdint.h>

#define FLOW_O_NONBLOCK 00004000
```

The above option sets a flow to non-blocking. It is based on the UNIX fcntl (file control) call, which has the same value for setting file descriptors to non-blocking access (O_NONBLOCK).

```
// TODO: Extend QoS spec
struct qos_spec;
```

This is a parameter to specify a certain QoS for a flow, it is currently not implemented.

```
/* Returns identifier */
int     ap_reg(char * ap_name, char ** difs, size_t difs_size);
int     ap_unreg(char * ap_name, char ** difs, size_t difs_size);
```

Application registration and unregistration functions allow to register an application name in a set of DIFs to make them reachable as a server or peer by other (client or peer) applications. The unregister call allows removing the name from a DIF.

```
/* Returns file descriptor (> 0) and client name(s) */
int     flow_accept(int fd, char * ap_name, char * ae_name);
int     flow_alloc_resp(int fd, int result);
```

The flow_accept call allows a server or peer application to receive incoming flows. It is somewhat comparable to a POSIX listen() call. It returns a descriptor for a flow (similar to a file descriptor, but technically it's not a file descriptor since it only identifies a RINA flow and not a general UNIX file).

After a an application receives an incoming request, it either accepts or rejects that incoming flow using the flow_alloc_resp() call. The convention is that 0 is accept, all other values are reject.

```
/* Returns file descriptor */
int     flow_alloc(char * dst_ap_name, char * src_ap_name,
                   char * src_ae_name, struct qos_spec * qos,
                   int oflags);

/* If flow is accepted returns a value > 0 */
int     flow_alloc_res(int fd);
```

On the client side, flow allocation is started using the flow_alloc() call, which can be compared to a combination of the POSIX socket() and connect() calls. The call is non-blocking. In order to know the result of a flow_allocation call, flow_alloc_res is called, which will block until the flow is either established or rejected.

```
int     flow_dealloc(int fd);
```

Flow deallocation can be done using a the flow_dealloc() call, which will free the resources associated with a flow (specified by the descriptor).

```
/* Wraps around fnctl */
int     flow_cntl(int fd, int oflags);
```

Control of the IO parameters of the flow. This call was previously implemented a non-blocking I/O behaviour by having a thread that constantly reads from a flow (blocking), queuing these SDU's, and then allow a non-blocking read from this queue. The new C wrapper now uses the flow_cntl syscall added to the IRATI prototype by PRISTINE.

```
ssize_t flow_write(int fd, void * buf, size_t count);
ssize_t flow_read(int fd, void * buf, size_t count);

#ifdef __cplusplus
}
#endif

#endif //LIBRINA_C_H
```

Reading and writing to a RINA flow takes as parameters the descriptor of the flow, a buffer with the SDU to write or space for the SDU to be read into, and the length of the buffer that has to be written, or the maximum

length of the provided buffer in the read case. The return value specifies an error if negative, or the number of bytes read.

2.5.6. rina-ioq3 rina calls

With all librina calls wrapped in the C wrapper, it was fairly easy to port the ioquake3 application. The most important restriction is that the server must be a dedicated server, the RINA_Init call will register the application if it is a server and start a thread listening for RINA SDUs.

The RINA_Recvfrom call mimics a POSIX select() call to receive SDUs from multiple clients in an asynchronous way. This call exposed a bug in the IRATI prototype that was subsequently fixed: https://github.com/IRATI/stack/pull/956

```
#include "net_rina.h"

#include <stdio.h>
#include <pthread.h>
#include <unistd.h>
#include <stdlib.h>

#include <librina-c/librina-c.h>

#define SERV_NAME "server.ioq3"
#define CLI_NAME "client.ioq3"
#define DIF_NAME "*"
```

This is the entry point of RINA in the ioq3 application. The C librina header is included. RINA mandates application names, so default names for the server and client are set here. The application will register itself in all available DIFs in the system ("*").

```
#define FDS_SIZE 255

int fds[FDS_SIZE];

static void add_fd(int fd)
{
        int i;
        for (i = 0; i < FDS_SIZE; i++) {
                if (fds[i] == -1) {
                        fds[i] = fd;
                        break;
```

```
                }
        }
}
```

We maintain an array of descriptors for the RINA flows of the application. Since the ioq3 server allows only 8 clients, 255 FDs is plenty.

```
void RINA_Resolve(const char * s, netadr_t * a)
{
        int fd;
        int result;

        a->type = NA_RINA;

        fd = flow_alloc(SERV_NAME, NULL, NULL);
        if (fd < 0) {
                printf("Failed to allocate flow\n");
                return;
        }

        result = flow_alloc_res(fd);
        if (result < 0) {
                printf("Flow allocation refused\n");
                flow_dealloc(fd);
                return;
        }

        flow_cntl(fd, FLOW_F_SETFL, FLOW_O_NONBLOCK);

        a->fd = fd;

        add_fd(fd);
}
```

ioq3 maintains its connections in a (proprietary) netadr_t structure. It supported types for IPv4 (and specific ones for loopback (local) addresses and bot support), and we added a new netadr_t type NA_RINA. This type is very simple as the only data it needs to maintain is the descriptor of the flow. The IP structure need to manage addresses and ports, for instance. The RINA_resolve function maintains flows through this netadr_t adapter, when a flow is established, it waits for the server response and sets the flow to non-blocking I/O before adding the descriptor to the set of active flows.

```
void RINA_Sendto(int length, const void * data, netadr_t * to)
```

```
{
        flow_write(to->fd, (void *) data, length);
}

int RINA_Recvfrom(msg_t * msg, netadr_t * from)
{
        ssize_t count = 0;
        int i         = 0;

        for (i = 0; i < FDS_SIZE; i++) {
                if (fds[i] == -1)
                        break;

                count = flow_read(fds[i], msg->data, msg->maxsize);
                if (count < 0) {
                        continue;
                }

                if (count > msg->maxsize) {
                        printf("Oversized packet received");
                        return 0;
                }

                from->type = NA_RINA;
                from->fd = fds[i];
                msg->cursize = count;
                msg->readcount = 0;

                return count;
        }

        return 0;
}
```

The sending of SDUs is straightforward through the RINA_Sendto function which simply wraps flow_write(). Currently, the PRISTINE/ IRATI prototype does not have a select() type function, so a similar function had to be implemented. This is the RINA_Recvfrom() function. Currently it just loops over all flows and returns the first packet it finds. We acknowledge that this call is not scalable to a large number of flows, but since ioq3 has 8 clients at most that send at low rate on the order of tens of kilobits per second, this is an acceptable solution until IRATI has a scalable asynchronous I/O call implemented.

```
void * RINA_Server_Listen(void * server_fd)
```

```
{
        int serv_fd = (intptr_t) server_fd;
        int client_fd;
        char * client_name = NULL;

        for (;;) {
                client_fd = flow_accept(serv_fd,
                                        &client_name, NULL);
                if (client_fd < 0) {
                        printf("Failed to accept flow\n");
                        continue;
                }

                printf("New flow from %s\n", client_name);

                if (flow_alloc_resp(client_fd, 0)) {
                        printf("Failed to give an allocate response\n");
                        flow_dealloc(client_fd);
                        continue;
                }

                flow_cntl(client_fd, FLOW_F_SETFL, FLOW_O_NONBLOCK);

                add_fd(client_fd);
        }

}
```

The listen call for the server is an (endless) for-loop that runs in its own thread and responds to flow allocation requests. It currently accepts all incoming flows without distinction (the flow_alloc_resp response is always 0). It also sets the flow to non-blocking and adds the descriptor to its set of flows.

```
void RINA_Init(int server)
{
        char * dif = DIF_NAME;
        pthread_t listen_thread;
        int server_fd;
        int i = 0;

        for (i = 0; i < FDS_SIZE; i++) {
                fds[i] = -1;
        }

        if (server) {
```

```
                if (ap_init(SERV_NAME)) {
                        printf("Failed to init.\n");
                        return;
                }

                server_fd = ap_reg(&dif, 1);
                if (server_fd < 0) {
                        printf("Failed to register AP.\n");
                        return;
                }

                pthread_create(&listen_thread,
                               NULL,
                               RINA_Server_Listen,
                               (void *) (intptr_t) server_fd);
                pthread_detach(listen_thread);
        } else {
                if (ap_init(CLI_NAME)) {
                        printf("Failed to init.\n");
                        return;
                }
        }
}
```

The RINA_Init does some basic configuration and management. In the client case, it is a single call that basically bootstraps the communication with the IRATI IPC manager. In the server case, apart from bootstrapping the communication with the IPC manager, it register the application in the DIFs and starts the thread that listens for incoming flows.

```
void RINA_Fini(int server)
{
        char * dif = DIF_NAME;

        if (server) {
                if (ap_unreg(&dif, 1)) {
                        printf("Failed to unregister application\n");
                }
        }

        ap_fini();
}
```

The RINA_Fini call unregisters the application if it is a server and cleans up the RINA resources when the application shuts down.

2.5.7. Conclusion

The work to port ioq3 showed that adding RINA support to an application is definitely feasible. The way ioq3 was adapted even allows both sockets (TCP/IP) operation and RINA operation at the same time. In other words, it's perfectly possible for clients over IP to play against clients over RINA.

Under PRISTINE, the proof-of-concept port turned into a stable tool for demonstrating RINA, which was first demonstrated at Geant The Network Conference (TNC) 2016.

3. Integration

In this section we report the final system integration. Each subsection accounts for one of the PRISTINE use cases, namely distributed cloud, datacenter networking, and network service provider. For each use case, the list of policies that have been integrated is reported.

3.1. Distributed Cloud

The VIFIB cloud currently relies on IPv6 in order to interconnect all nodes, but in many cases such IPv6 access is unreliable or even absent. To solve this, an overlay network has been created: re6stnet[6]. RINA, as an alternative to the re6st overlay, would enable different overlays to different customers, with custom characteristics: availability, levels of service, security and so on.

The availability of RINA on a global infrastructure is also important to increase adoption of RINA, and for this, performance and maintainability is taken into account. It has been decided in a first step to have RINA on top of an existing re6st network, for the following reasons:

- Reimplementing all algorithms of re6st inside IRATI would take more time before seeing first results, and it would increase maintenance costs, instead of taking advantage of further re6st improvements.
- This also avoids the maintenance of a separate network with only nodes that support RINA.
- Gateways for RINA applications and IP applications inside the cloud are easier to deploy.

Figure 6 sketches the adopted configuration.

[6] https://lab.nexedi.com/nexedi/re6stnet/

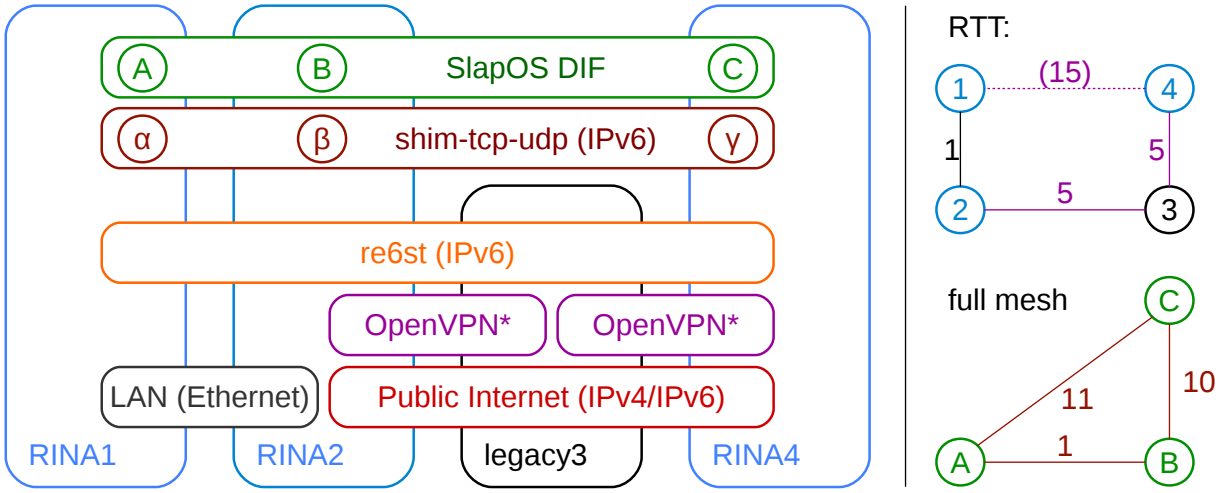


Figure 6. Distributed cloud DIF configuration.

- A single shim-tcp-udp DIF on top of an existing re6st network (that also contains nodes without RINA support);
- a single normal DIF is used for the whole cloud;
- and each RINA node is a neighbour of every other RINA nodes (enrollment).

In other words, the routing via intermediate re6st nodes is done at IP level, and this is the most efficient solution to begin with, because latencies can be reduced by routing via non-RINA nodes, as shown above.

In any case, all this is only possible if nodes can join/leave the network dynamically and attention was mainly focused on this aspect.

3.1.1. Overview of re6st.net

re6stnet creates a resilient, scalable, ipv6 network on top of an existing ipv4 network, by creating tunnels on the fly, and then routing targeted traffic through these tunnels.

re6stnet can be used to:

- guarantee connectedness between computers connected to the internet, for which there exists a working route (in case the direct route isn't available).
- create large networks
- give ipv6 addresses to machines with only ipv4 available

Building an ipv4 network is also supported if one has software that does not support ipv6.

How it works

A re6stnet network consists of at least one server (re6st-registry) and many nodes (re6stnet). The server is only used to deliver certificates for secure authentication in established tunnels, and to bootstrap new nodes.

re6stnet can detect and take into account nodes present on the local network.

- Resilience: re6stnet guarantees that if there exists a route between two machines, traffic will be correctly routed between these two machines. Even if the registry node is down, the probability that the network is not connected is very low for big enough networks (more than a hundred nodes).
- Scalability: Since nodes do not need to know the whole graph of the network, re6stnet is easily scalable to tens of thousand of nodes.

3.1.2. Integration with re6st

Inside a re6st network, each node is given a x509 certificate that contains 2 fields. When appended to the network prefix, they give the allocated IPv4/IPv6 ranges. The normal IPCPs are named after the IPv6 *node prefix* because it simplifies the implementation. However, only the IPv4 *node prefix* is small enough to serve as an IPCP address.

The re6st daemon knows everything about the network, its allocated prefixes and that of other nodes, so it configures everything dynamically, and the IPCM configuration is minimal:

- normal DIF: the template does not define any IPCP address (empty *knownIPCProcessAddresses*)
- shim-tcp-udp DIF: no hostname/expReg/dirEntry

re6st only expects the shim DIF to be created with a specific name. Then it takes care of:

- configuring the shim DIF (hostname/expReg)
- assigning the shim DIF

- creating/assigning/registering the normal DIF

And in order to do enrollments, it also updates directory entries (shim DIF) and set IPCP addresses.

Only 1 RPC between re6st daemons was extended, to discover nodes that implement RINA. The x509 certificates are already used to secure packets so the peer prefix (hence the IPCP addresses) is already known once a connection is successful.

In any case, re6st first queries the state of the IPCPs to determine which actions must be taken. In particular, this handles the following cases:

- re6st restarted
- IPCM restarted
- IPCP crashed

3.1.3. Developments in the IRATI stack

- IPv4 support in re6st is optional and currently disabled in VIFIB cloud. re6st provides primarily an IPv6 network, and shim-tcp-udp was extended to support IPv6.
- Most communications with the IPCM is done via the console.

The following features had to be developed in order to support the distributed cloud use case, which is currently awaiting to be merged into the main branch of the IRATI stack:

- Communication between re6st and the kernel is done via a sysfs writable attribute, so that the shim-tcp-udp DIF can be configured dynamically (PR #997[7]). See Specification for the Shim DIF over IPv4/UDP with Domain Name System (DNS) Support[8] for a proper solution.
- re6st exposes a UNIX socket that is queried by IPCM/IPCP to get IPCP addresses (PR #996[9]).

[7] https://github.com/IRATI/stack/pull/997
[8] https://wiki.ict-pristine.eu/wp4/d43/d43-parallel-flows#providing-parallel-flows-over-the-shim-difs_specification-for-the-shim-dif-over-ipv4-udp-with-domain-name-system-dns-support
[9] https://github.com/IRATI/stack/pull/996

3.1.4. Deployment

Nodes runs the *master* branch of https://github.com/jmuchemb/irati-stack, which is mainly a merge of the *pristine-1.5* branch and the features described in the previous section.

The kernel, librina and rinad components are installed as Debian packages (PR #1001[10]), including development libraries. We have few softwares in Nexedi that require system-wide installation, and for them we use the Package Management System of the Operating System. This is important for maintainability because, contrary to an installation from sources, packages enable:

- faster installation, thanks to binary packages
- clean update or removal, by proper tracking of installed files

As explained above, some changes in re6st itself were required, and they have been merged into the *master* branch. We also deploy it as a system package, and we use Open Build Service[11] in order to support many OS/architectures. Packages on OBS[12] have been regenerated, so that any node can easily install (or upgrade to) a version of re6st that works with the IRATI stack.

The installation of machines in our cloud is automated with Ansible[13], which provides an API to abstract partially the differences in the way each OS is configured. Our playbooks are maintained at https://lab.nexedi.com/nexedi/slapos.package/tree/master/playbook, and we added one that can be used to install and configure the IRATI stack for use in our cloud: see *rina.yml* and the *rina* role.

For the moment, a single DIF spans the whole cloud: instances are given unique identifiers that can be used as application process instance.

With all the above developments, RINA is now available inside any SlapOS cloud and users can now describe/deploy services that use a RINA stack.

[10] https://github.com/IRATI/stack/pull/1001
[11] https://build.opensuse.org/
[12] https://build.opensuse.org/package/show/home:VIFIBnexedi/Re6stnet
[13] https://www.ansible.com/

3.2. Datacentre Networking

The datacentre networking joint experiment uses the following policies:

- Simple multi-path forwarding policy (hash-threshold based, like ECMP). Policy of the PDU Forwarding Policy.
- ECN marking policies for the Relaying and Multiplexing Task: RED (based on Random Early Detection) and CAS (based on Jain's binary feedback scheme).
- Flow control policies that are ECN-aware for the Error and Flow Control Protocol: TCP-like (with ECN) and CAS (based on Jain's binary feedback scheme).

Policies were upgraded to the last version of the RINA code developed by PRISTINE (pristine-1.5 branch), which required the update of some APIs implemented by the policies. In order to verify the correct behaviour of the policies in *pristine-1.5*, the simplified version of the datacentre experiment shown in Figure 7 was set up using the demonstrator.

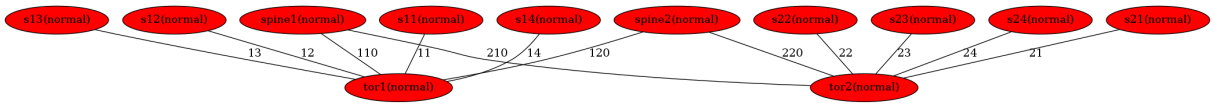


Figure 7. Integration tests demonstrator scenario, single DIF

The IPCPs in the top of rack switches (labeled *tori*) were configured with the multipath and ECN marking policies; the servers (labeled *sij*) were configured with EFCP ECN-aware flow control policies and the spines (labeled *spinei*) were configured with ECN marking policies. Two *rina-tgen* sessions were established between each pair of servers in different racks (*s1i* to *s2i*). The correct behaviour of the policies was verified, leaving a more detailed analysis of its behaviour to the experimentation activities per se.

The most challenging part of the integration activity was to integrate the Manager in the demonstrator. First of all the demonstrator was extended with a new option (*-manager*). When instructed with this option the demonstrator automatically creates:

- An extra virtual machine to run the Manager
- An extra bridge in the host running the demonstrator

- An extra virtual Ethernet interface in each virtual machine in the experiment, connected to the host's "management bridge"
- A management DIF that connects all the systems in the experiments to the Manager and allows it to configure them

Demonstrator virtual machines run Linux images, which are conveniently generated automatically using a tool called buildroot (they can be generated manually by other means, but it is a cumbersome procedure). Therefore the Manager had to be able to run in a buildroot-generated image, which was challenging because buildroot didn't support Java 8 out of the box. A *Java 8* binary package for buildroot was created, which solved the problem. Since *Java 8* is quite a big package and the goal of the demonstrator VMs is to be as small as possible, the demonstrator was extended with another option to specify two different images: one for "regular" machines running the IRATI stack and another one for the machine running the Manager.

3.3. Network Service Provider

The network service provider joint experiment uses the following policies:

- SDU protection policies to ensure exchange of data between IPCPs in a secure way over an untrusted network.
- Authentication policies to establish a DIF between trusted IPCPs.
- TTL (time-to-live) policy to avoid PDU from flooding the network.

Policies are compatible with the last version of PRISTINE stack source code (branch 1.5). The experiment takes place on a simplified network, where the Service Provider has four IPC Process which provides access to its services. Such IPCPs are considered to be located on different physical locations, so security must be applied to the data passing through them, as the lower DIF which provides the connectivity is not trusted.

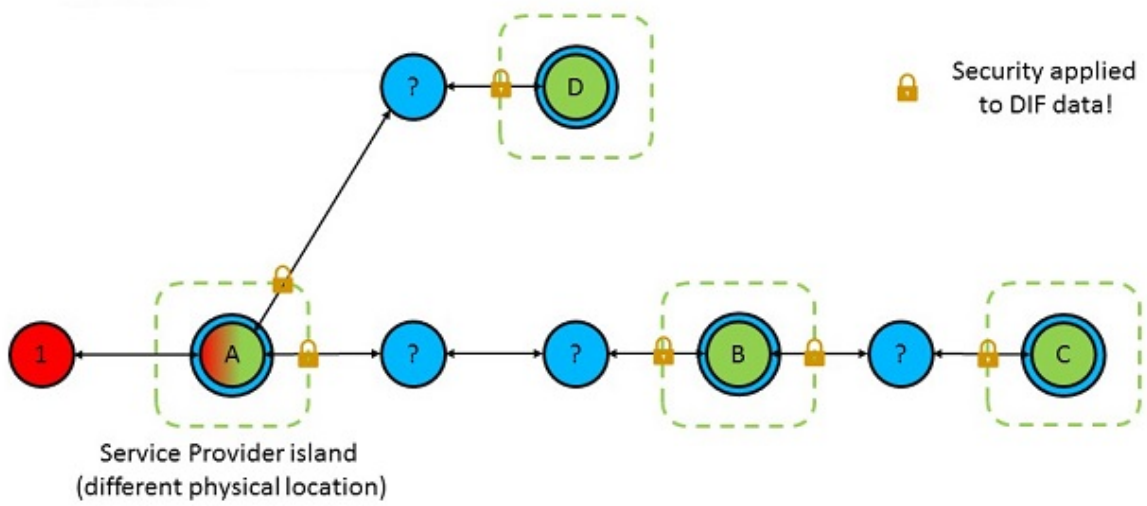


Figure 8. Setup of the join experiment.

The Network Service Provider, in this scenario, has different "computational islands" which are dedicated to certain jobs; for example the first node (A) provides the entry point in its network, and have routing and firewalling capabilities. This is necessary to allow just authorized traffic to flow within the NSP network. The following node (B) provides filtering functionalities, this can be done with specialized software/hardware in order to minimize the latency and increase the performances. Nodes C and D provide the service main servers (configured with different policies of hosted on hardware with different performances), which are differentiated for paying users and free users. Flows which aims for the first service have less hops and can follow an low-latency QoS, while the ones going to the free service must pass through the filtering servers and can experience lag or similar issues (since they follow a low priority QoS).

The experiment runs on real nodes on a remote testbed, with links connecting each node at 1Gb/s. Such nodes are connected between each other on a shim over Ethernet, but thanks the layering nature of RINA it can be placed over other layer without changing its configurations.

4. Experiments Per Research Area

In this section we report on the stand-alone experiments carried out during the second phase of the project. These experiment have been carried out to test specific PRISTINE policies in isolation. Notice how this section accounts for both experiments carried out using the IRATI stack as well as for experiments implemented using the simulated testbed.

4.1. Multi-level Security

Testing of the Multi-level Security (MLS) implementations focuses on the Boundary Protection Component (BPC) that has been implemented as a Java application and integrated with Librina to operate over the RINA's IRATI stack. The BPC implementation at DAF-level is described in detail in D4.3[14]. The experiment focuses on component-level verification of the BPC. This experiment is aimed at evaluating whether or not the implementation operates without error and according to its specification in the Network Service Provider (NSP) use-case scenario. This is to prove the correct functionality of the implementation. Component-level verification of the BPC at the DAF-level was performed to prove the correct functionality of the implementation and reported in D4.3[15].

4.1.1. BPC Configuration

The BPC is configured with a policy that determines the classification of the sending and receiving applications from their registered DIFs. The policy searches the data of SDUs for sensitive keywords and, if any are found, classifies the data as High, or Low otherwise. The BPC policy then applies the rules in Table 1 to make a decision whether to allow the SDU to be forwarded to the destination.

Table 1. Boundary Protection Policy Rules

Rule ID	Data Classification	Sender Classification	Receiver Classification	Decision	BPC Action
1	Low	High or Low	High or Low	Accept	Forward SDU
2	High	Low	High or Low	Indeterminate	Block SDU

[14] https://wiki.ict-pristine.eu/wp4/d43/d43-mls
[15] https://wiki.ict-pristine.eu/wp4/d43/d43-mls

3	High	High	High	Accept	Forward SDU
4	High	High	Low	Reject	Block SDU

'Accept' means that the SDU is at a suitable classification for the destination, so the SDU is forwarded to the intended destination application. 'Reject' means that the classification of the SDU is above that of the destination, so the SDU is blocked. 'Indeterminate' means that the classification of the sender is lower than that of the SDU, so the SDU is blocked. This should not occur, since the sender should not have data that is above its classification level.

4.1.2. Experimentation Environment

The experimentation environment used for the verification tests is shown in Figure 9. It consists of three VMs running the PRISTINE SDK, which are connected via two VLANs: 110 and 111. Nodes 1 and 2 are configured to have a classification level of either High or Low, depending on the requirements of the verification test.

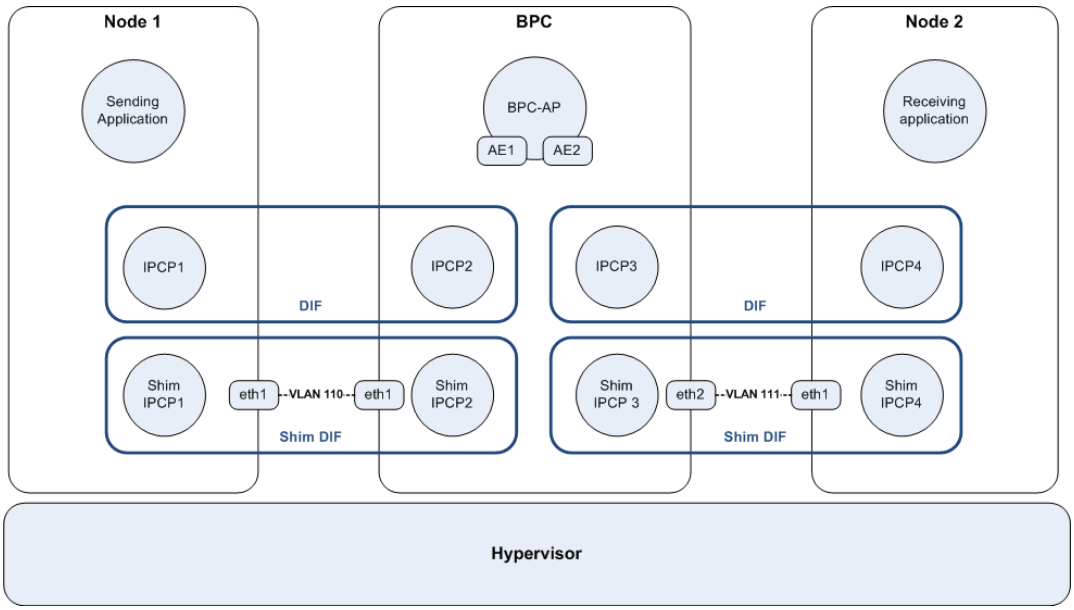


Figure 9. Experimentation Environment

4.1.3. Verification Tests

The verification tests are designed to exercise the BPC rules in Table 1. For each verification test, the classification of the Node 1 (sender), Node 2 (receiver) and the Data are each set to either High or Low according to the configuration in Table 2. For each verification test the sending application

is configured to send 5 SDUs to the BPC application to be forwarded to the receiving application. High SDUs contain sensitive keywords, while Low SDUs do not. The sending and receiving applications each record the number of SDUs sent and received respectively in their log files, while the BPC records its policy decision for each SDU. At the end of each test the log files from all three applications are inspected to verify that the BPC: i) correctly classified the SDUs as either High or Low; ii) made the correct policy decision and iii) took the correct action of either blocking or forwarding the SDUs.

Table 2. Verification Test Configuration

Test ID	Rule Under Test	Node 1 Classification	Node 2 Classification	Data Classification
1	4	High	Low	High
2	1	High	Low	Low
3	2	Low	High	High
4	1	Low	High	Low
5	3	High	High	High
6	1	High	High	Low
7	2	Low	Low	High
8	1	Low	Low	Low

The results of the verification tests are shown in Table 3.

Table 3. Verification Test Results

Test ID	BPC Data Classification	Expected BPC Decision	Actual BPC Decision	BPC Action	SDUs received at Destination
1	High	Reject	Reject	Block SDU	0
2	Low	Accept	Accept	Forward SDU	5
3	High	Indeterminate	Indeterminate	Block SDU	0
4	Low	Accept	Accept	Forward SDU	5
5	High	Accept	Accept	Forward SDU	5
6	Low	Accept	Accept	Forward SDU	5
7	High	Indeterminate	Indeterminate	Block SDU	0
8	Low	Accept	Accept	Forward SDU	5

The results in Table 3 verify that the BPC correctly classified the SDUs, made the correct policy decision and took the correct action of either blocking or forwarding SDUs. The BPC therefore functions correctly according to its configured policy rules.

4.2. Load Balancing and Optimal Resource utilization

The load balancing mechanism presented in this section targets specifically DC scenarios and has been implemented in the RINASim[16] which is an open source OMNET++ based implementation of RINA. The load distribution implementation is available in TSSG-LB[17] branch of the RINASim[18] implementation. The main load distribution algorithm can be seen in stats.cc[19] and stats.h[20] files.

The simple implemented algorithm is shown below:

```
std::string Stats::getBestApp(std::string srcApp, std::string dstApp,
 std::string allApps)
    {
        if(!allApps.compare("AppErr"))
            return dstApp;
        std::istringstream    lineStream(allApps);
        std::string app, availableApp = dstApp;
        int minLoad = MAX_LOAD;
        int appLoad = 0;
        int appCount = 0;

        while( std::getline( lineStream, app, ',' ) )
        {
            app = trim(app);
            if (!srcApp.compare(app))
            {
                continue;
            }
            else
            {
                appLoad = getLoad(app);
                //appsMap[app] = appLoad;
```

16 https://github.com/kvetak/RINA/
17 https://github.com/kvetak/RINA/tree/TSSG-LB
18 https://github.com/kvetak/RINA/
19 https://github.com/kvetak/RINA/blob/TSSG-LB/src/DAF/AE/Stats.cc
20 https://github.com/kvetak/RINA/blob/TSSG-LB/src/DAF/AE/Stats.h

```
                appCount++;
                EV <<"Load on the current App: " << appLoad;
                if(minLoad > appLoad)
                {
                    minLoad = appLoad;
                    availableApp = app;
                }
            }
        }
        EV << " Min load:" << minLoad << " at the App:"<<
availableApp<<std::endl;
        return availableApp;
    }
```

In order to create a simulation setup and topology, six very large files need to be created. These files include network definition (.ned), simulation initialization (.ini), connection set (.xml), directory entries (.xml), qoscube (.xml), and stats (.txt). The manual creation of these files is very time consuming and there is very high probability of errors therefore we decided to automate the topology generation process. The script written in topology generator[21] takes the following input and generates the desired topology along with all the associated files:

- Number of total participating Servers in the data center topology
- Number of total participating clients in the cloud
- Number of PODs in the data center
- Number of TORs per POD
- Number of Aggregation switches per POD
- Number of Edge or Core switches
- Number of Server Application Processes (AP) per server machine
- Number of Server AP Instances per AP
- Number of client AP per client machine
- Number of client AP Instances per AP
- AP name
- Number of Servers per POD

[21] https://wiki.ict-pristine.eu../../uploads/filewrite.cc

- Link properties such as BW, Delay, Jitter, BER, PER etc within data center
- Maximum and minimum SLA details for clients. For example the client can be given maximum 1Gbps connection and minimum 10Mbps. The script will randomly subscribe each client between these extremes.

The number of lines in most files created by this script are usually more than 10,000. The topology used in initial experiment can be seen in Figure 10. In this experiment we created a fat tree like data center topology with 4 PODs and 8 server machines per POD thus 32 total server machines. There are 30 clients having random connection subscription between 1Mbps to 2Gbps. Here a new node type is introduced which is named as cloud in order to emulate the Internet cloud in the simulation setup. This node can hold, drop, add errors and forward the PDUs like the Internet. The behaviour of this node is to emulate each client is many hops away from the servers. The properties of links between client machines and the cloud nodes reflects the subscription of each client with its ISP. This simulation setup is available in the LB example[22] for reference and result regeneration purposes.

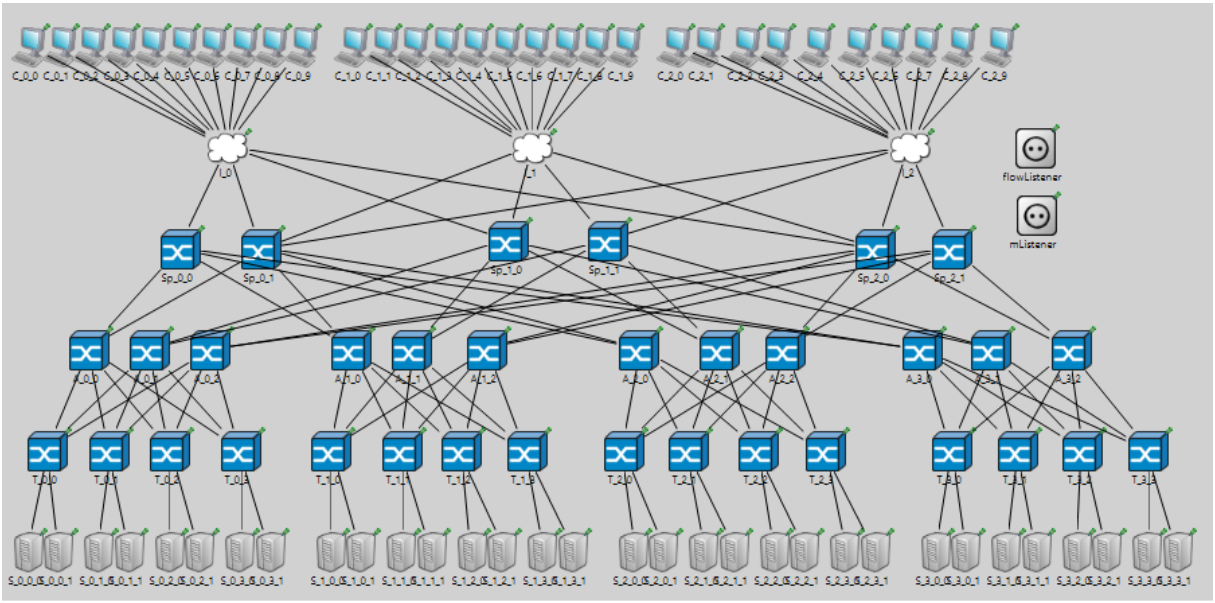


Figure 10. Simulation Topology

A continuous stream type of traffic is generated between client and server APs. Each client generates a stream of PDUs for 5 seconds at random

[22] https://github.com/kvetak/RINA/tree/merginthingsTSSG/examples/LB/DDC1

intervals between 1-100 msec. Each experiment is repeated 10 times and the average of each traced value is used for result and analysis purposes.

In the first experiment as can be seen in the Figure 11, each server machine is set to process 22,000 PDUs per second. Beyond this limit the next PDUs will be forwarded to the next AP instance. The result shows that 6 out of 32 servers run at their full limits while 26 servers remained idle causing under utilization of resources. On the other hand when we configure our load balancing algorithm in such a way that when one server is running at 1/4th of its capacity or beyond that, then the next incoming load is diverted to the next available server. The result in Figure 11 shows an even distribution of load across all the available resources.

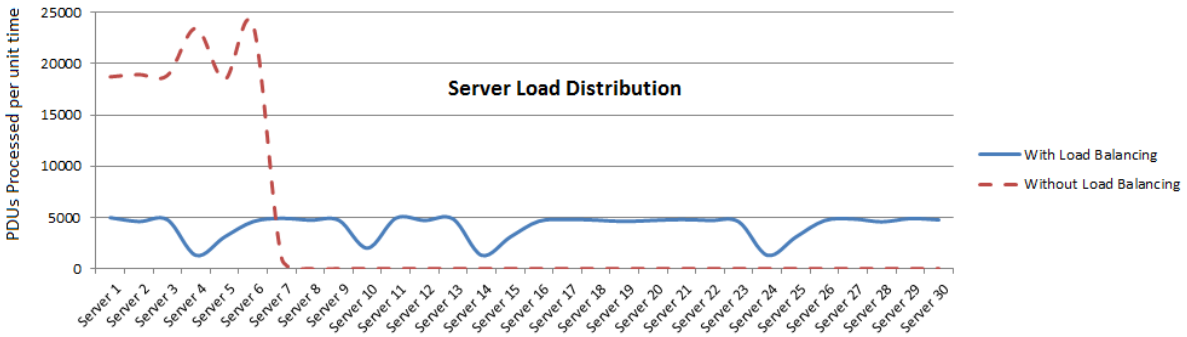


Figure 11. Server Load Distribution with and without Load Balancing

In the next experiment, the two load distribution approaches; Random Load Distribution and Min-hop Selection as discussed in [d4.3] are compared to each other. Figure 12 shows that flows experienced more average delay in the random selection method than the min-hop selection method. This is very obvious that in random selection, the load balancing algorithm forwards the flow request to any available server instance irrespective of how far it is located from the client. Each client has it's own connection speed, generates PDUs at random intervals therefore each client would experience different delays. Furthermore the peaks in the graph shows that these clients experienced more delays than the others because of there distance from servers, connection speed etc. The load of each client is different. Each client has independent connection with the servers therefore the delay experienced by each client is independent to the others However it is apparent that min-hop selection method offers less average delays than the random selection method.

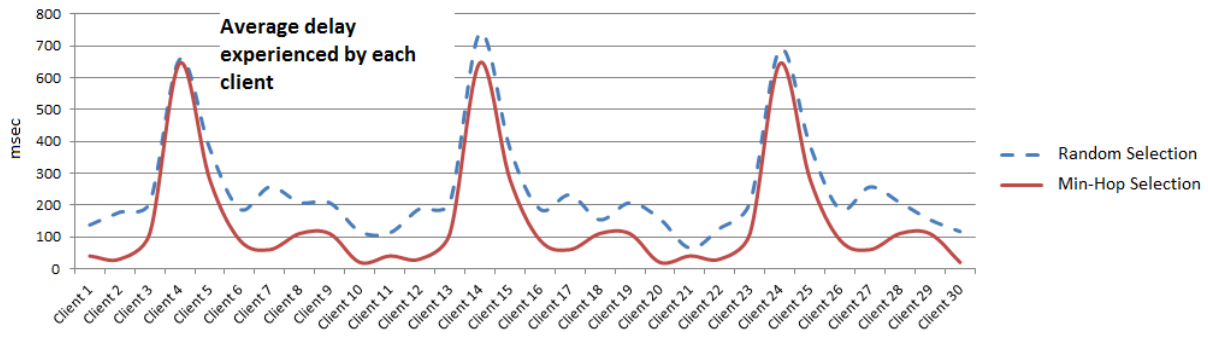


Figure 12. Average Delay (mSec) per flow random instance selection vs min-hop instance selection

Figure 13 shows the average throughput gained by each client while using both methods respectively. In random selection method, the average throughput for each flow merely exceeds 200Mbps while with the use of min-hop selection method, the same flow with same properties, can achieve near 2Gbps of throughput. It is worth noting here that each client has independent traffic load and flow characteristics therefore throughout gained by each client would be independent to each other.

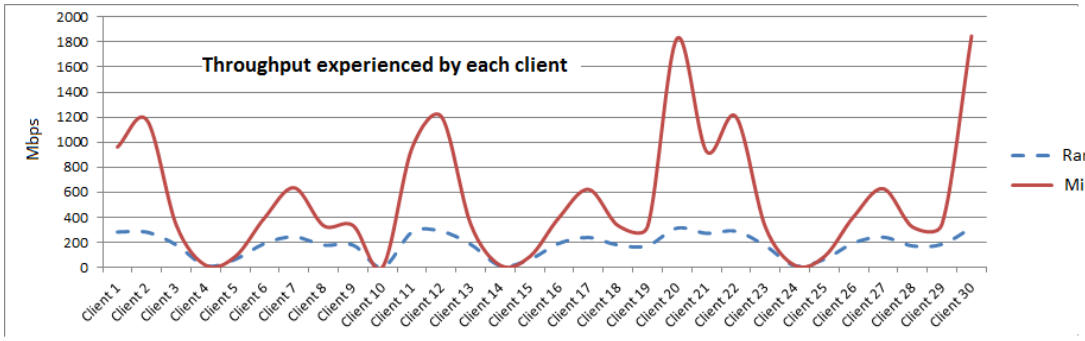


Figure 13. Average Throughput (Mbps) per flow random instance selection vs min-hop instance selection

With random selection method, the selection algorithm picks one server instance randomly from a list of available server instances without bothering with how far the selected instance is located from the client instance. This method results in an overall increase in latency and number of dropped packets as well as in a reduced throughput (See Figure 14).

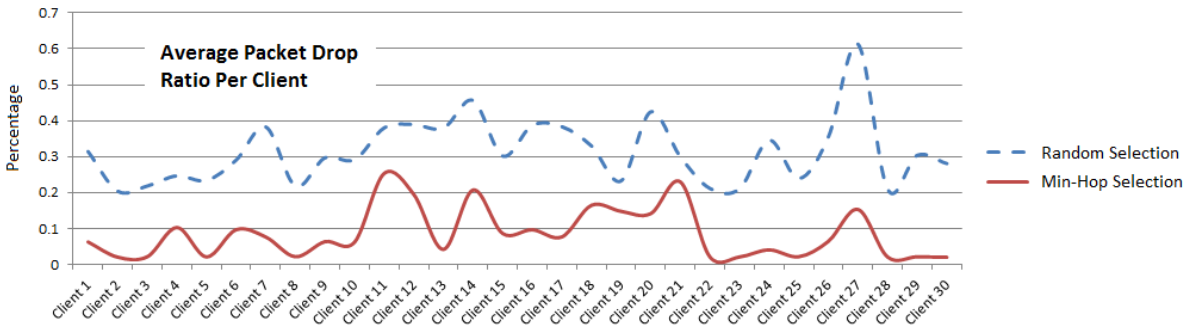


Figure 14. Average packet drop ratio (%) per flow random instance selection vs min-hop instance selection

4.3. Policies for Performance Isolation in Multi-Tenants Data-Centres

4.3.1. Introduction

The experiments described in this section have been carried out using the Virtual Wall(vWall) test bed. Four experiments have been executed:

- **I** experiment, which identifies the case where two or more tenants communicate from the same source node to the same destination node.
- **Y** experiment, which identifies the case where two or more tenants communicate from different source nodes to the same destination node.
- **XS** experiment, which identifies the case where two or more tenants communicate from different source nodes to different destination nodes (under a common Top Of Rack switch).
- **XD** experiment, which identifies the case where two or more tenants communicate from different source nodes to different destination nodes (under different Top Of Rack switches).

Each experiment has been run in two modes, namely:

- **Macro-flow**, which considers flows with an high Minimum Granted Bandwidth (MGB, see Minimum Granted Bandwidth definition[23]) requirement. During this experiment two tenants, called Alpha and

[23] https://wiki.ict-pristine.eu/wp3/d32/D32-data-centre-cc#policies-for-performance-isolation-in-multi-tenant-data-centres_data-centre-organization-assumptions_minimum-granted-bandwidth

Bravo, exists with different MGB. One of the two tenants has an intermittent behavior, periodically communicating only for 30 seconds every 60 seconds.

- **Micro-flow**, which considers flows with a low MGB requirement. During this experiment 4 tenants, called Alpha, Bravo, Charlie and Delta, exists with different low MGB. Two of them have an intermittent behavior with different periods of activity.

Finally, to appreciate the impact of routing and forwarding, every experiment has been repeated with different Equal Cost Multi Path policies, namely:

- **Static hash** (provided by ATOS). This policy uses a hash of the connection block of the PDU header to map a flow to a port. As a result, all packets sharing the same PCI will always go trought the same path.
- **Random pick** (provided by CREATE-NET). This policy randomly shuffles the flows across the available paths. Notice that, once a flow has been assigned a given path, this path will be used for the entire length of the experiment.
- **Port load** (provided by Create-Net). This policy forwards each flow across the least loaded port.

4.3.2. Experiment setup

As we said, the Virtual Wall environment from iMinds will be used to perform the experiment (iMinds testbed[24]).

This scenario consists of 20 machines organized in a fat-tree topology (see Figure 15). In particular we consider a configuration with only two PODs (See definition for Datacentre organization[25]). Each POD consists of 4 Servers, 2 Top of Rack switches and 2 Aggregator switches. Four additional core routers provide the POD with backbone connectivity.

[24] https://www.wall1.ilabt.iminds.be/
[25] https://wiki.ict-pristine.eu/wp2/d22/D22-RefArch-Datacentre-Networking

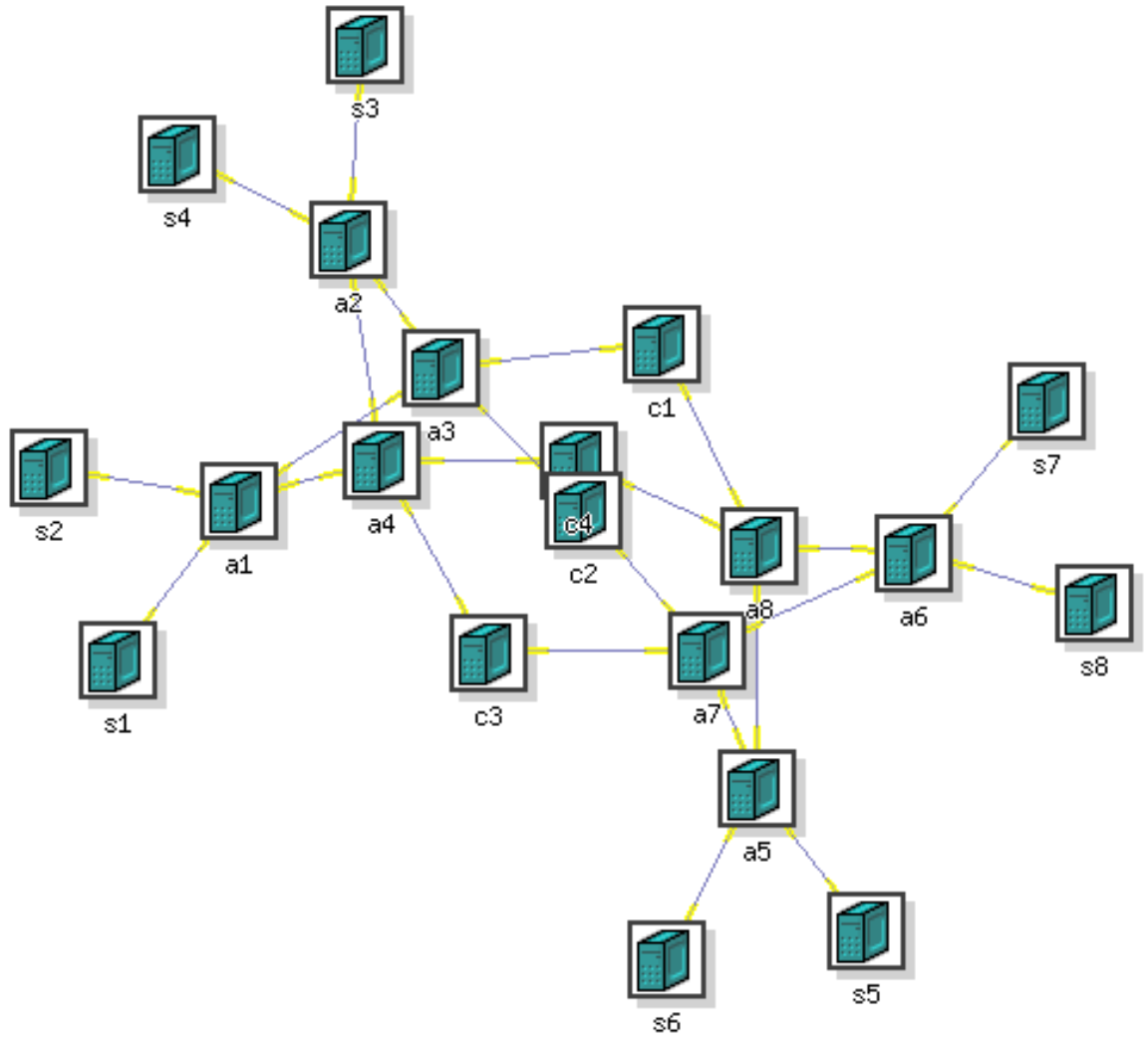


Figure 15. Topology used for the experiments. Nodes marked with's' are servers, nodes marked with 'a' are aggregator switches, nodes marked with 'c' are core routers.

Each link in the topology has a physical line rate of 1 Gb/s.

4.3.3. Experiment nodes setup

In order to correctly configure the vWall machines the following steps must be performed:

1. Swap a single machine with a standard Debian image. This machine will be used as a base to install IRATI necessary software.

2. Follow the IRATI tutorial(Wiki for IRATI configuration[26]) in order to install the kernel stack and user space components. At the end of this procedure you will have a working IRATI stack on the machine.
3. It is also needed to ensure that some additional prerequisites are installed on such machines, like `ethtools`, which will be used to limit the link rate to 100 Mb/s.
4. Once IRATI and all the additional software has been installed it is possible to create a base OS image that can used for all the experiments, thus avoiding repeating the full setup procedure for every node in the experiment.

At this point one should have an OS image with a bootable system with the IRATI stack installed. This is the minimum in order to have a quickly deployable experiment in the vWall. After having performed these steps one needs to set up all the machines in order to prepare IRATI to be run. At the end of this configuration stage one will end up having a set of configuration files which are compatible with the IPC Manager daemon parser, and that will instruct it on how to configure the machine at bootstrap of the system. These files are:

- A set of `.conf` files, which are used by the IPCM of the nodes to know how many IPC to run at the begin and at which DIF they belong.
- A set of `.dif` files, which are used by shim IPC Processes to know what kind of configuration they must follow when they adopt a DIF.
- One `dc.dif` file, which summarizes the characteristics of the base DC DIF which provides the base connectivity layer between each element of the DC. This is where the custom policies for congestion control are applied.
- One `da.map` file, which is left empty and contains static mapping for Application entities and DIFs.

Such files must be deployed on every node in the experiment, since we do not assume the use of the DMS manager to handle the creation of DIFs. The DC base DIF will already be present on the machines when the operations of bootstrap finishes.

[26] https://github.com/IRATI/stack/wiki

4.3.4. Data Center DIF configuration

This DIF provides the connectivity between each node in the data center network, and is placed directly above the Ethernet Shim IPC Processes. The layer is configured to use a set of 4 policies which, altogether, provides the desired multi-tenant congestion control mechanism (see performance isolation for multi-tenants datacentre[27]). The base idea behind this is to assign to every upper DIF (the tenant ones) a QoS cube which have the `averageBW` field set to the desired MGB. This values is used by the DC DIF to set such threshold of granted bandwidth for that particular flow.

Since every top IPC can be considered as an AE, we use this simplification in order to reduce the configuration overhead of IRATI. Our tenants traffic so will be executed by AEs placed directly on the top of the data center DIF (see Figure 16).

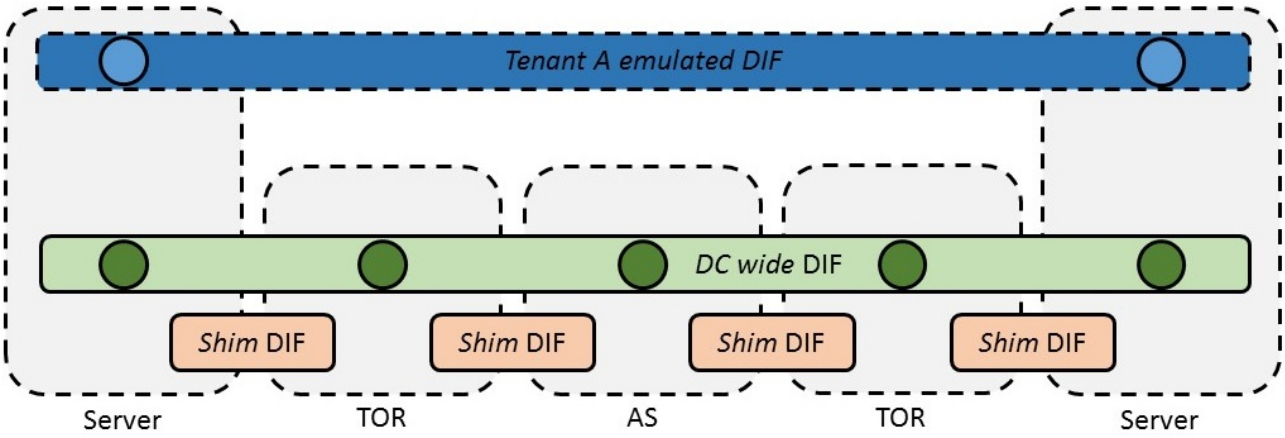


Figure 16. Tenant's DIFs are implemented as AEs.

4.3.5. Experiment 1: Static hashed forwarding

During this experiment we expect that congestion will be resolved by adapting the flow rate according to the given policies logic. No decision is made at forwarding level, and the flow path can be predicted by knowing the header connection block hash and the list of ECMP ports to the destination.

[27] https://wiki.ict-pristine.eu/wp3/d33/D33-data-centre-cc

Macro flows

Macro flows experiment analyze the behavior of having Tenants with high MGB requirements. During this experiment setup we expect that Tenants are always granted to have at least their MGB as accessible resources. If more bandwidth is available the two tenants will share the remaining resources equality.

I scenario

Figure 17 shows the two tenants exchanging data and consuming the DC resources. The communication takes place from 's6' to 's1'(see Figure 15). Since they are placed in the same source node, both Tenants end up sharing the resources of the same outgoing link. This will trigger the congestion control mechanism once the Alpha tenant wakes up, causing both tenants to reduce their transmission speed to the nominal MGB rate.

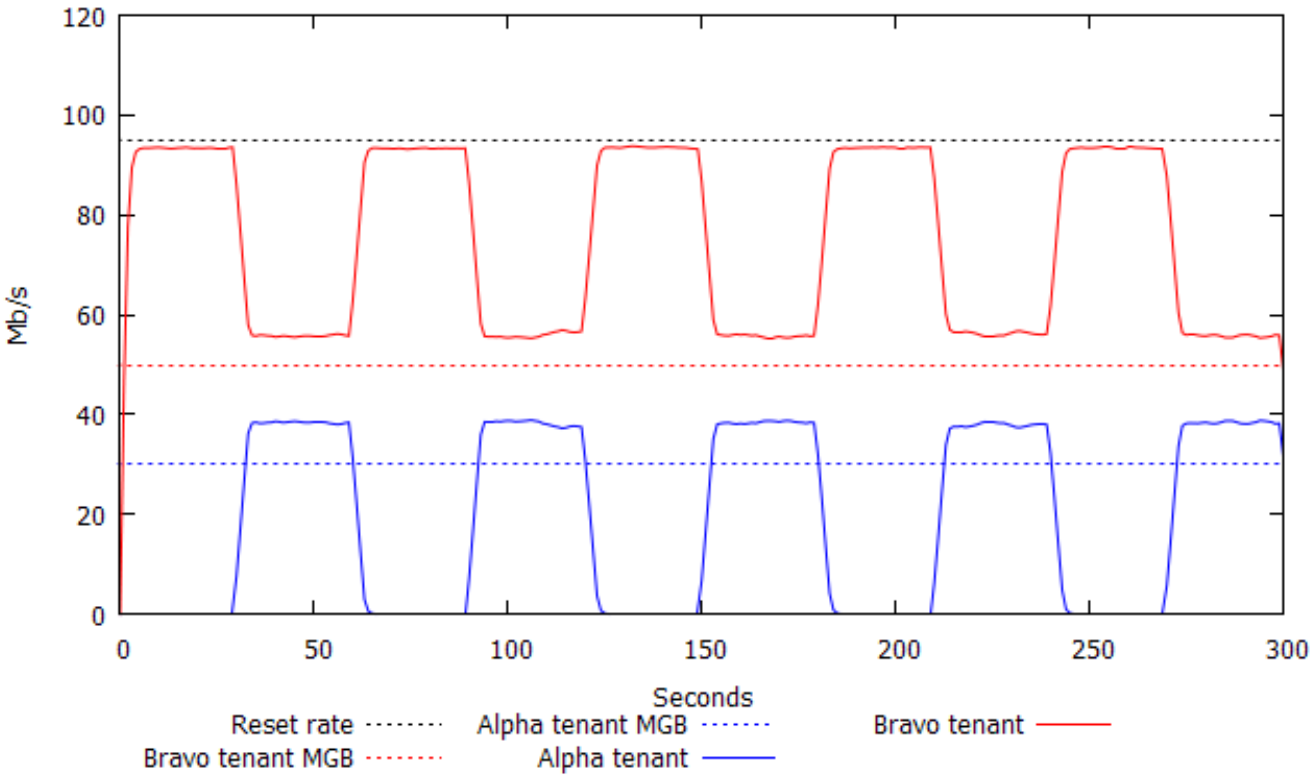


Figure 17. Two Tenants, Alpha and Bravo, concur for resources from the same source to the same destination nodes.

Y scenario

Figure 18 shows the two tenants exchanging data from a different source machine. Communication now is from 's6' and 's5' to 's1'(see Figure 15). The

behavior is more or less the same as the one in the previous experiment because having the same destination introduce a bottleneck in the last hop in the path.

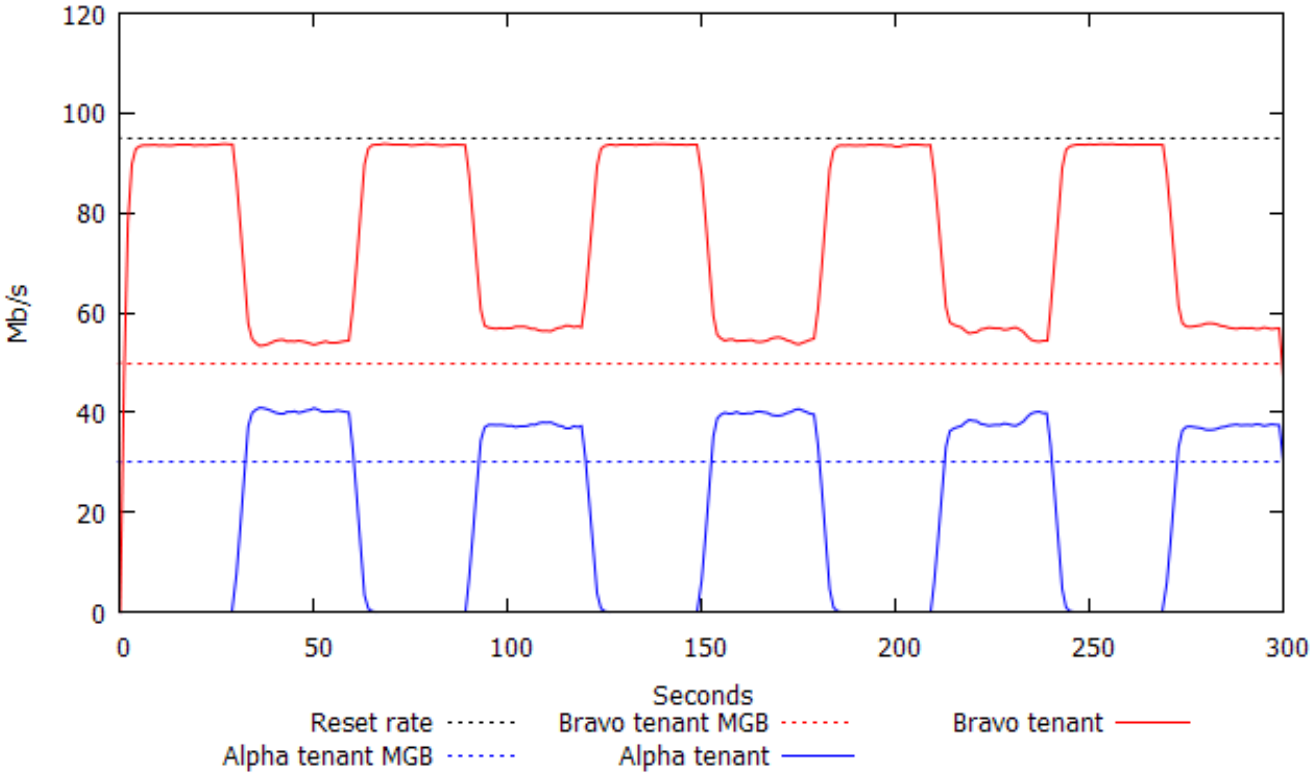


Figure 18. Tenants, Alpha and Bravo, concurs for resources from different sources to the same destination nodes.

In addition to this, also the forwarding policy plays a decisive role here. Figure 19 shows that the aggregator switch *5* (on port 3) is the one which triggers the congestion signal (since it ss the only one with a growing queue). This is because the PDU forwarding policy decided to route both flows on the same direction, rather than using one of the alternative paths. Switch a1 does not detect any congestion, since this is already resolved due the feedback from a*5*.

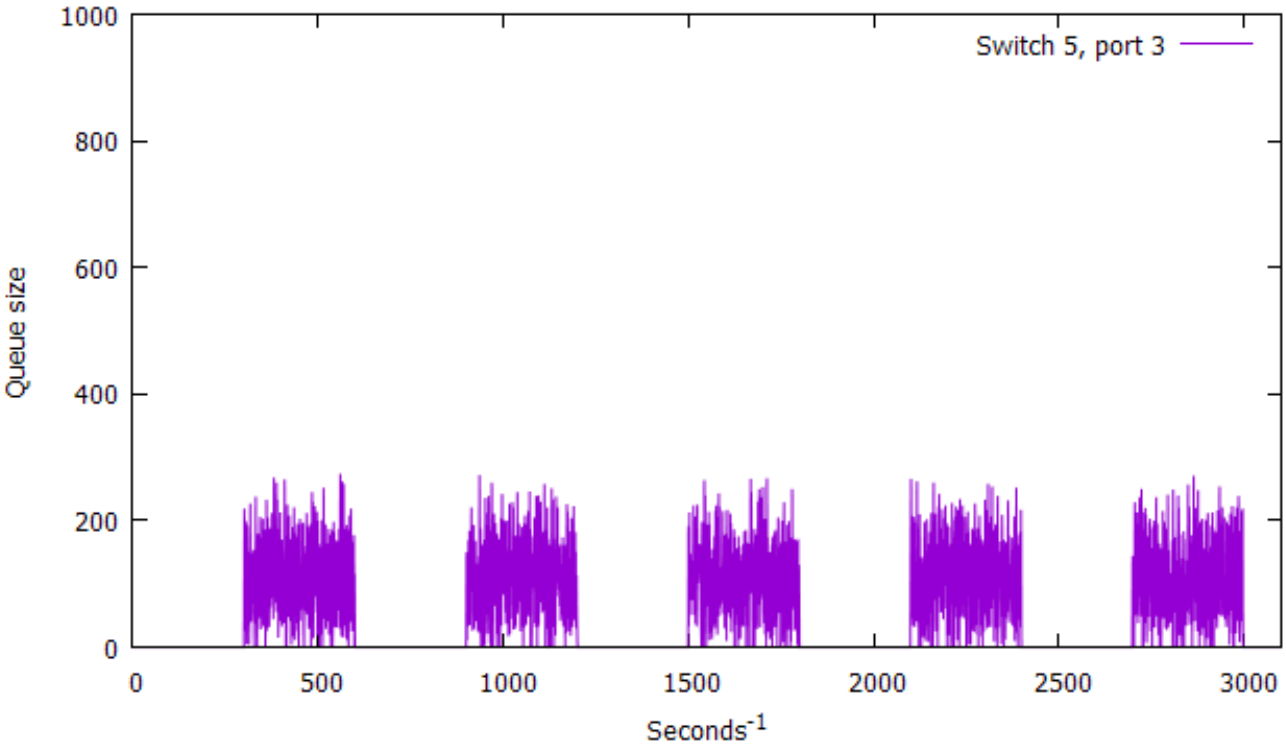


Figure 19. The status of the queues of the switches during the experiment.

XS scenario

Figure 20 shows the two tenants exchanging data and consuming the data center resources. The communication takes place from 's6' and 's5' to 's1' and 's2' (see Figure 15). Since we are using a full bisection bandwidth topology and since the two flows are forwarded by the PDU forwarding policy on different routes, both Tenants can exploit the full link rate bandwidth.

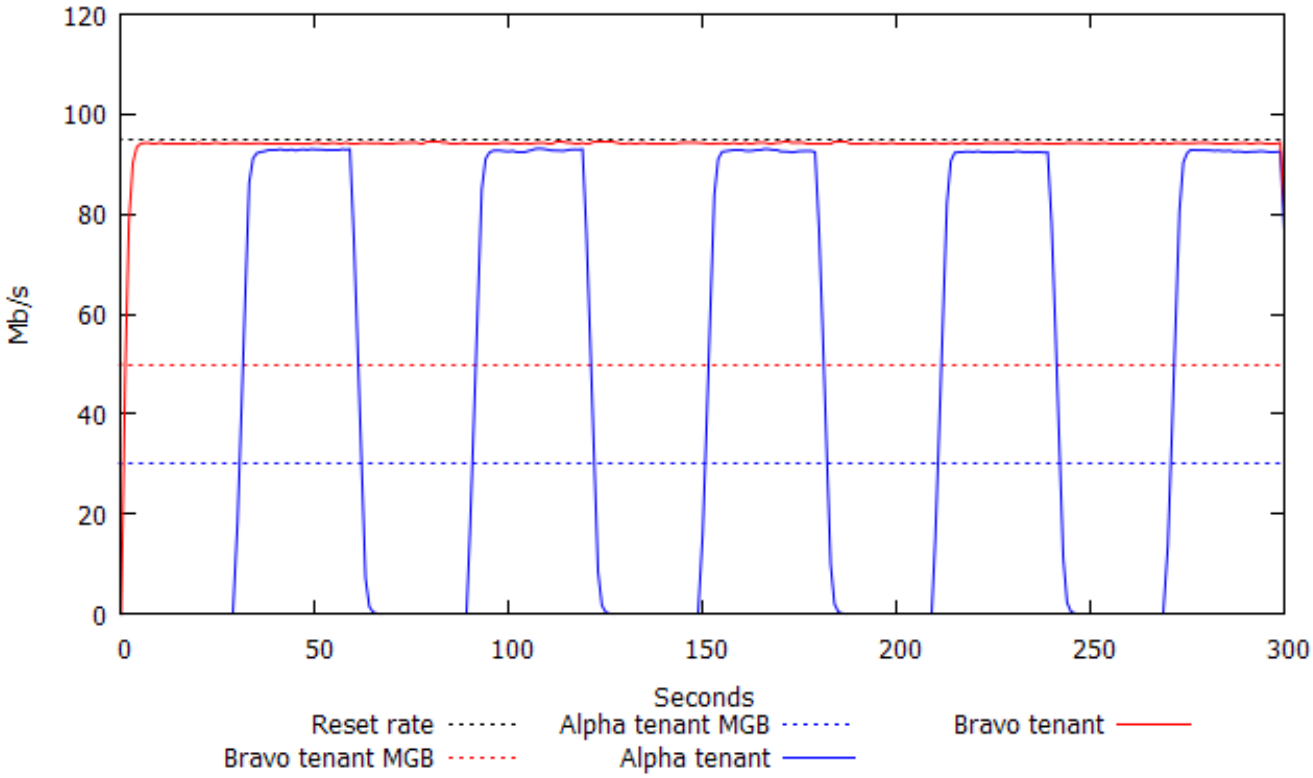


Figure 20. Tenants, Alpha and Bravo, concur for resources from different sources to different destinations nodes, under the same Top of Rack switch.

XD scenario

Figure 21 shows two tenants exchanging data and consuming the DC resources. The communication takes place from 's6' and 's7' to 's1' and 's3' (see Figure 15). In this experiment, the forwarding policy routed the flows through two different paths which do not share any hop. This results in the possibility for both Tenants to achieve the full line rate speed.

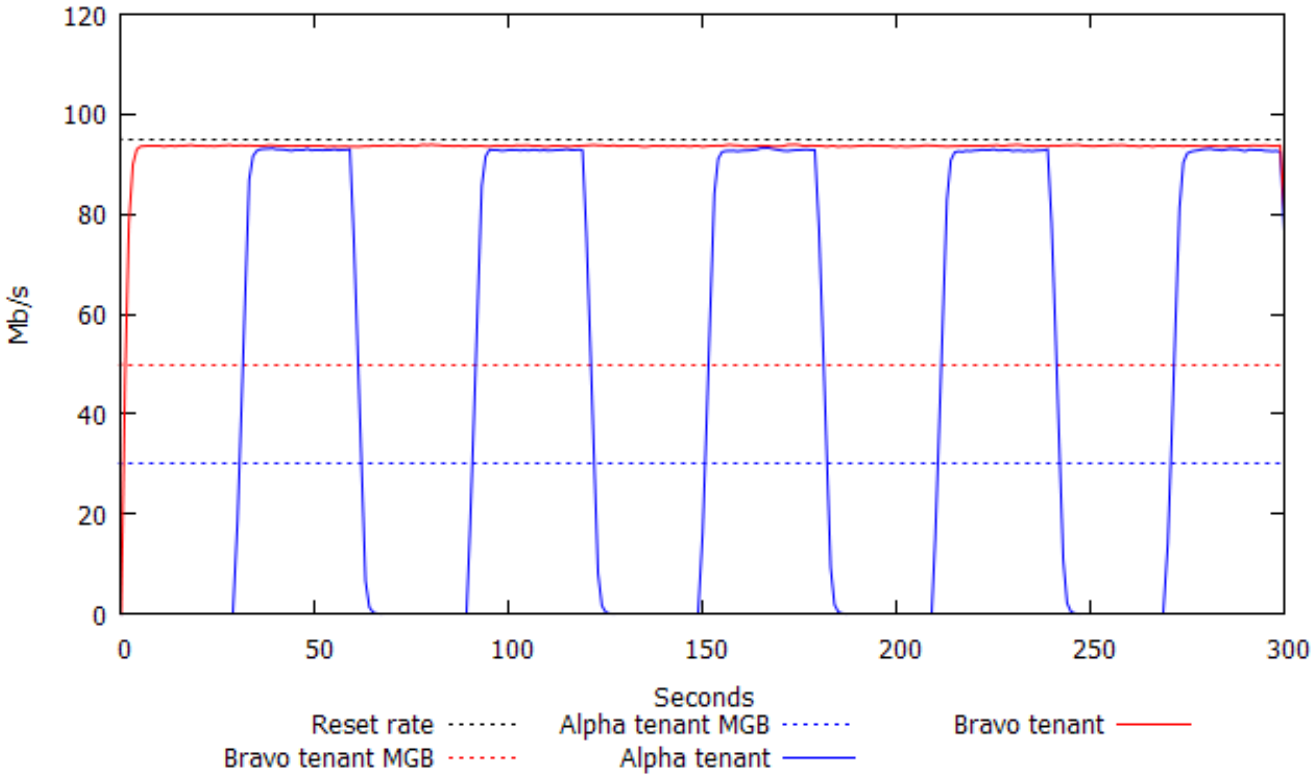


Figure 21. Tenants, Alpha and Bravo, concurs for resources from different sources to different destinations nodes.

Micro flows

During this experiment setup we expect that Tenants are always granted to have at least their MGB as accessible resources. If more bandwidth is available the two tenants will share the remaining resources equality.

I scenario

Figure Figure 22 shows the four tenants exchanging data and consuming the DC resources. The communication takes place from 's6' to 's1'(see Figure 15). Since they are all placed in the same source node, all Tenants end up sharing the resources of the same outgoing link. This will triggers the congestion control mechanism once Alpha and Charlie tenants wake up (they have an intermittent behavior), causing all the flows to be scaled back to their nominal MGB. Since the nominal MGB is very low, all the tenants end up sharing equality the remaining bandwidth.

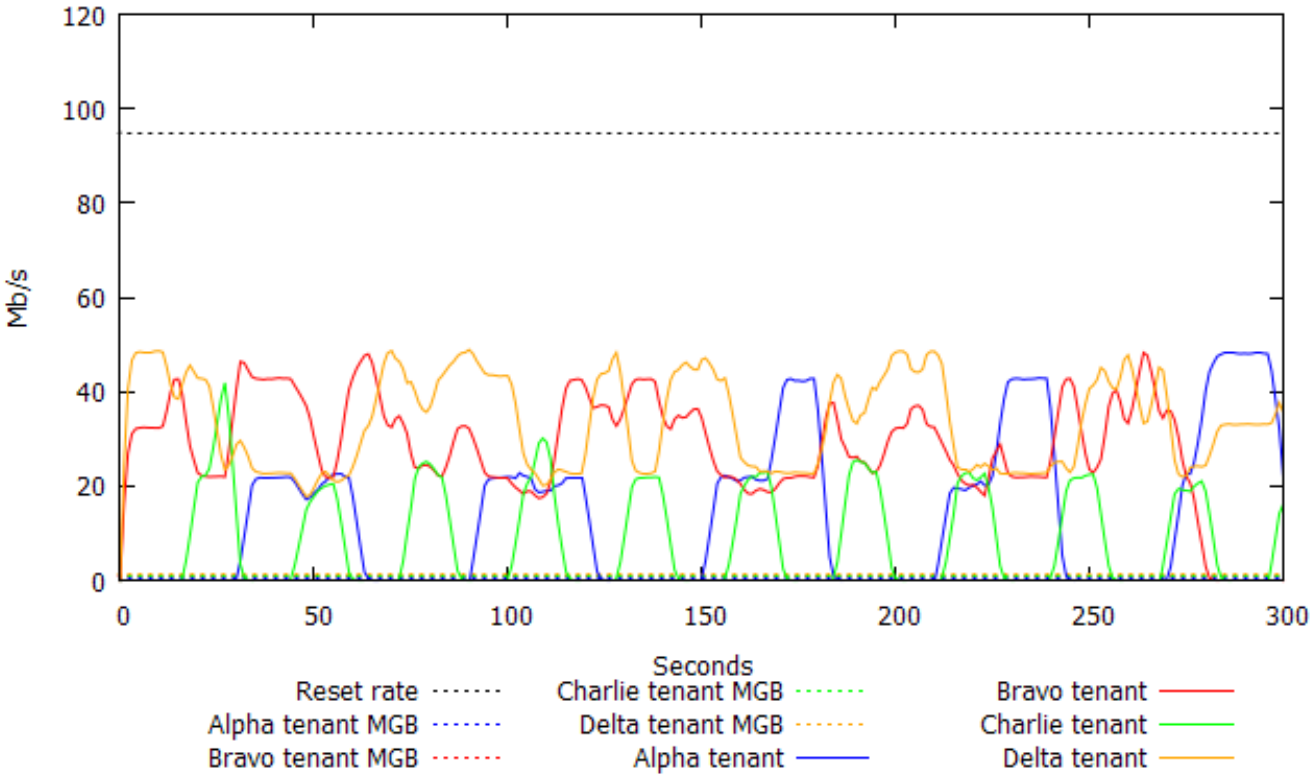


Figure 22. Four Tenants communicating from the same source server to the same destination server.

Y scenario

Figure Figure 23 shows the four tenants exchanging data and consuming the DC resources. The communication takes place from 's5' and 's6' to 's1'(see Figure 15). Now every different source node shares a continuous active Tenant and an intermittent one.

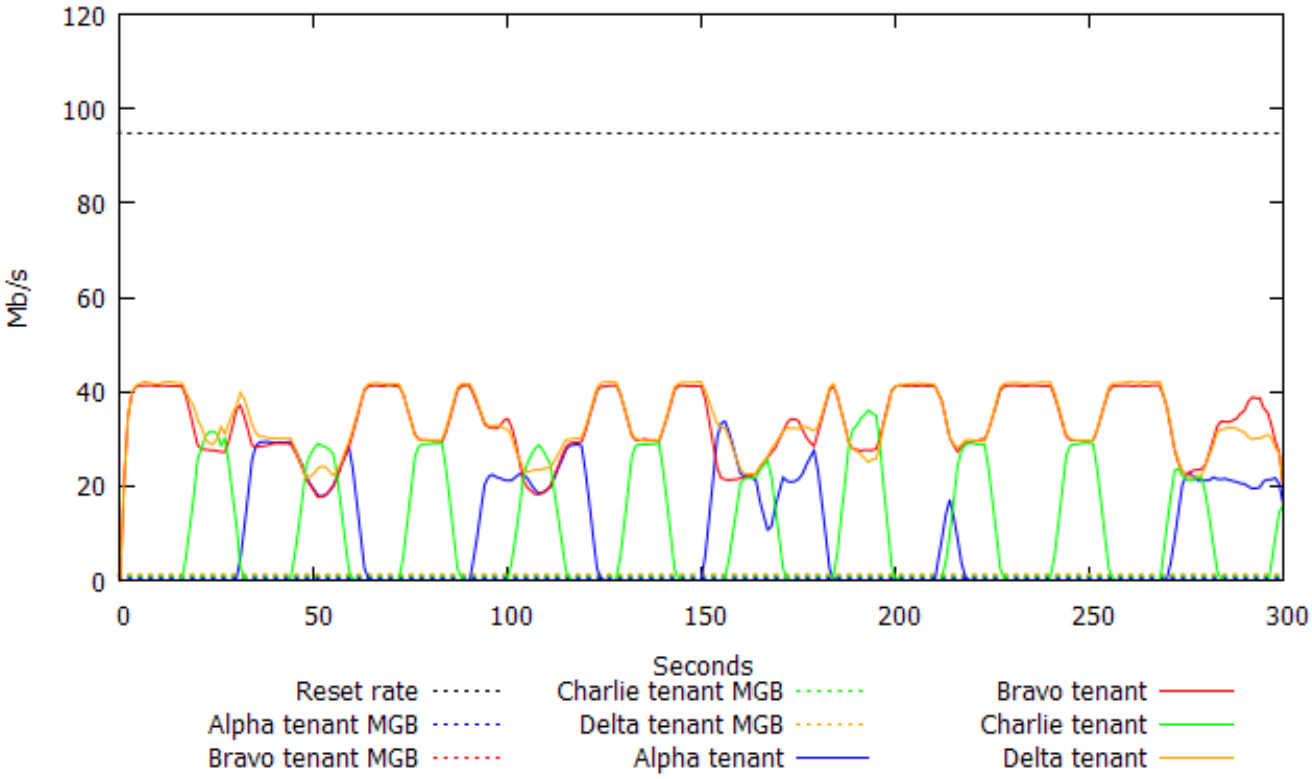


Figure 23. Four tenants communicates from the same source to the same destination.

As it can be observed in Figure 24, now the PDU forwarding policy successfully selects the correct equal-cost path to reach the destination, and the last link is the only one suffering from congestion. In particular we can notice that the queue reaches an higher occupancy when all the tenants are active and try to communicate at full rate.

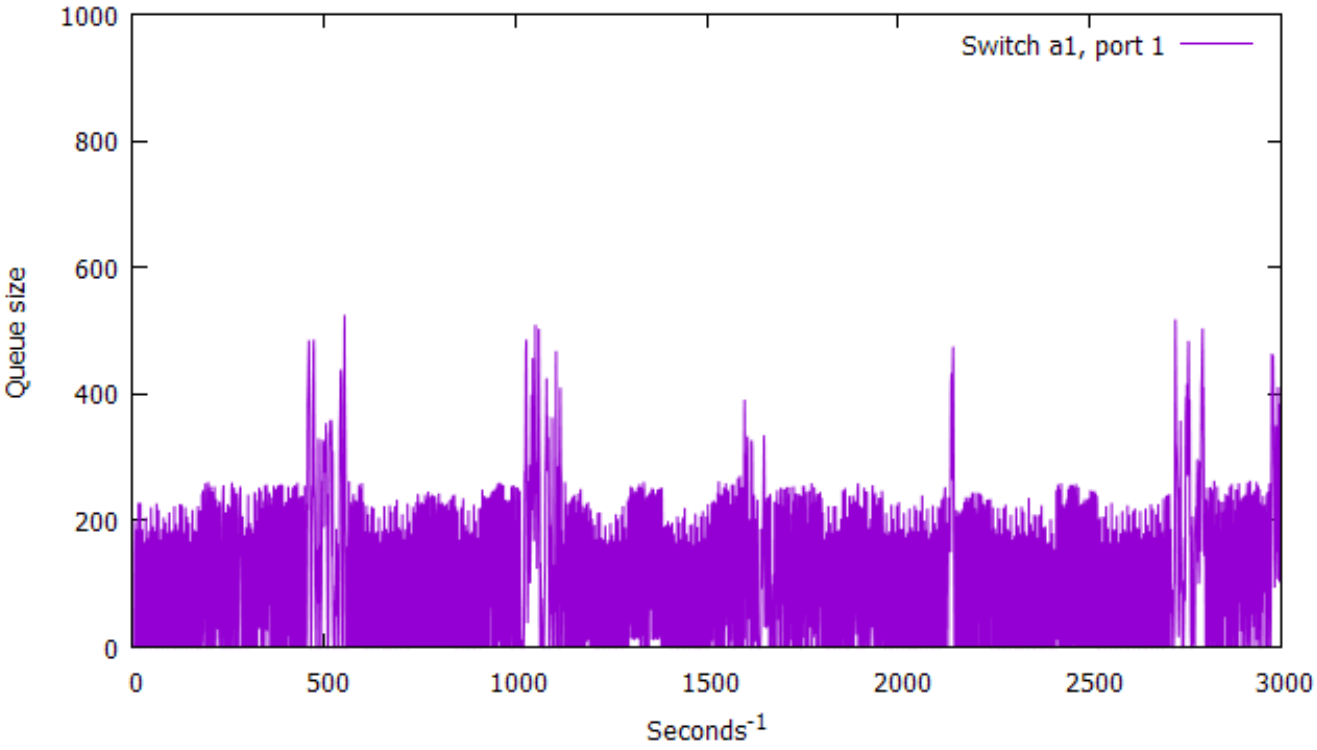


Figure 24. Four tenants communicates from the same source to the same destination.

XS scenario

Figure Figure 25 shows the four tenants exchanging data and consuming the DC resources. The communication takes place from 's5' and 's6' to 's1' and 's2'(see Figure 15). Now every different source node shares a continuous active Tenant and an intermittent one. This pattern is also followed at destination side, where every destination node receives traffic from a continuous and an intermittent Tenant.

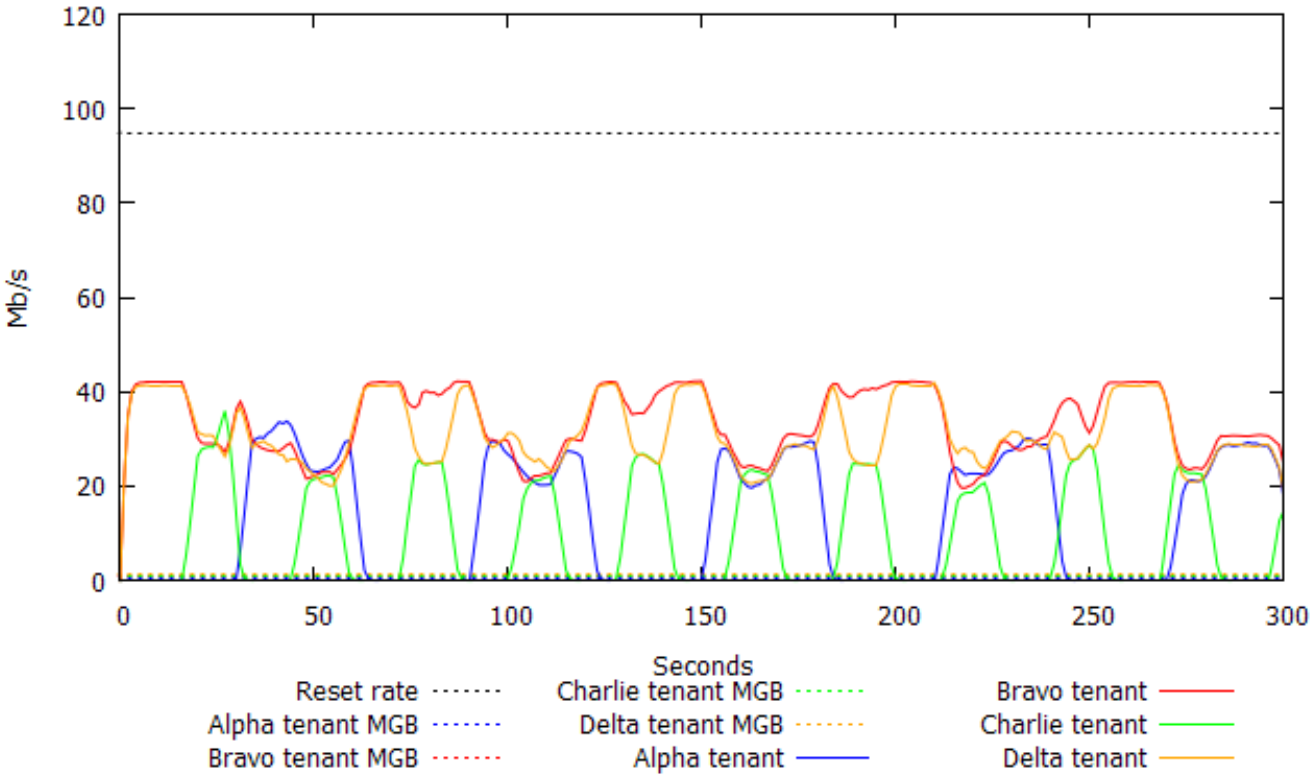


Figure 25. Four tenants communicates from the different sources to the different destinations, in couple of continuous and intermittent flows.

We expected that the continuous flow could reach the full rate and share it when the intermittent flow appears, but instead the forwarding policy redirected the traffic of all four tenants over the same path (see Figure 26).

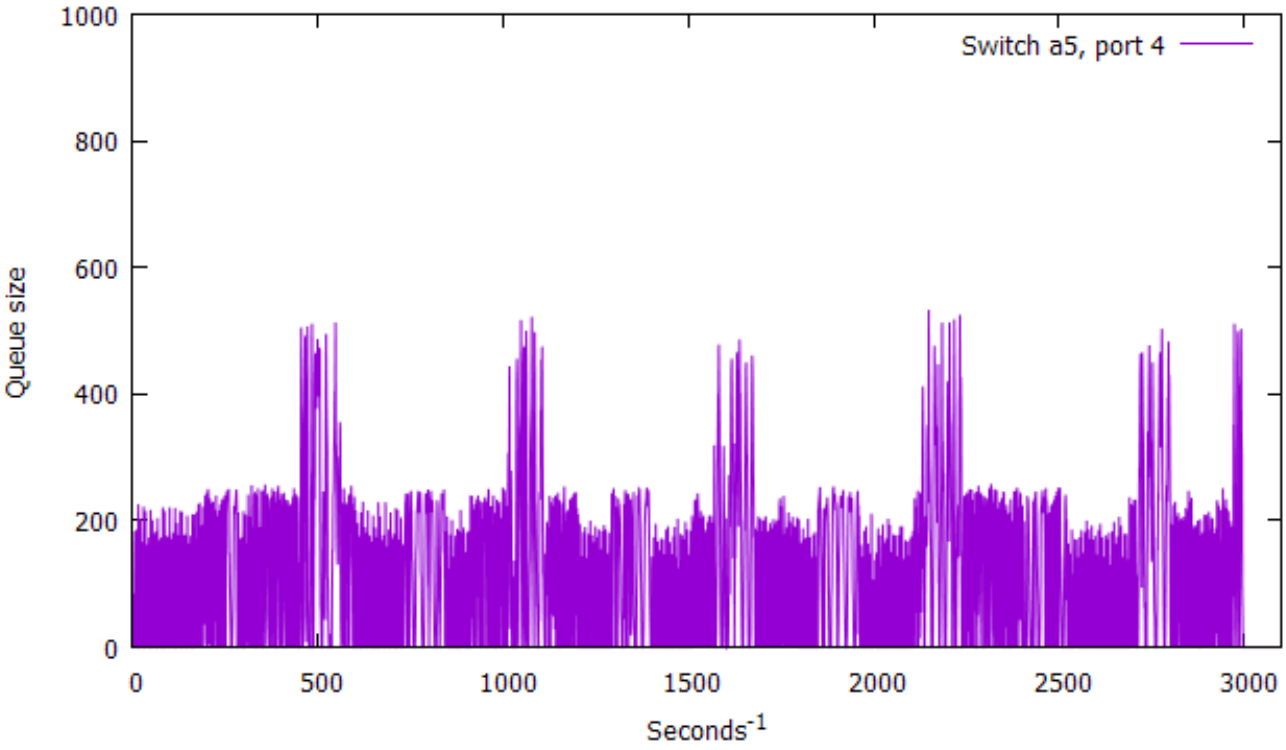


Figure 26. The traffic of all four tenants is redirected on the same path to their destinations.

XD scenario

Figure 27 shows the four tenants exchanging data and consuming the DC resources. The communication takes place from different sources to different destinations for every in the DC DIF (see Figure 15).

- Alpha, on 's5', communicates with 's1'.
- Bravo, on 's6', communicates with 's2'.
- Charlie, on 's7', communicates with 's3'.
- Delta, on 's8', communicates with 's4'.

Since we are using a full bisection bandwidth topology, it should be possible for every tenant to be placed on a different route to its destination, but it is not the case in this experiment.

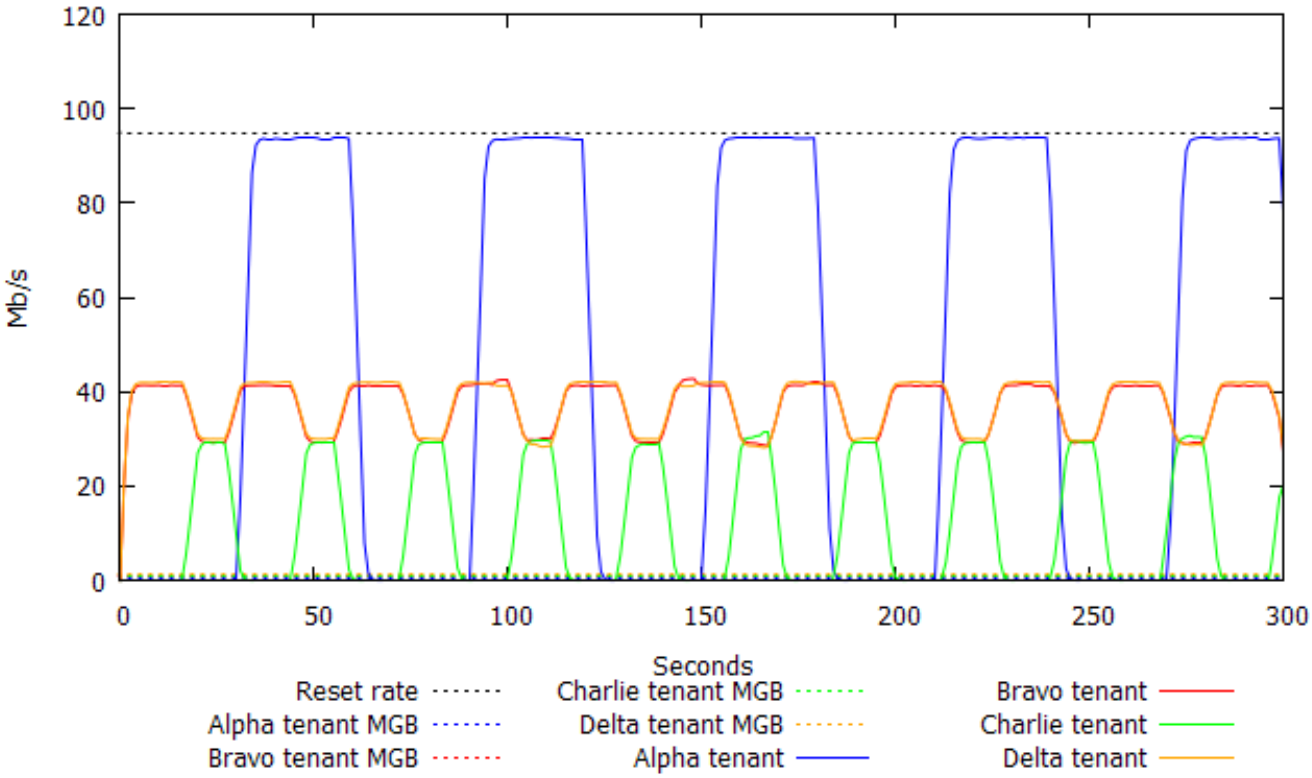


Figure 27. Four tenants communicates from the different sources to the different destinations.

Only Tenant Alpha is able here to communicate at full speed. This is due the fact that tenants Charlie and Delta are redirected from aggregator switch 6 on the same route to switch 7, and then both flows join tenant Bravo's traffic (see Figure 28). In the meantime tenant Alpha's traffic is routed on a completely different path by aggregator switch 5 or 7, which allows it to run on a link without any concurrency with the other tenants.

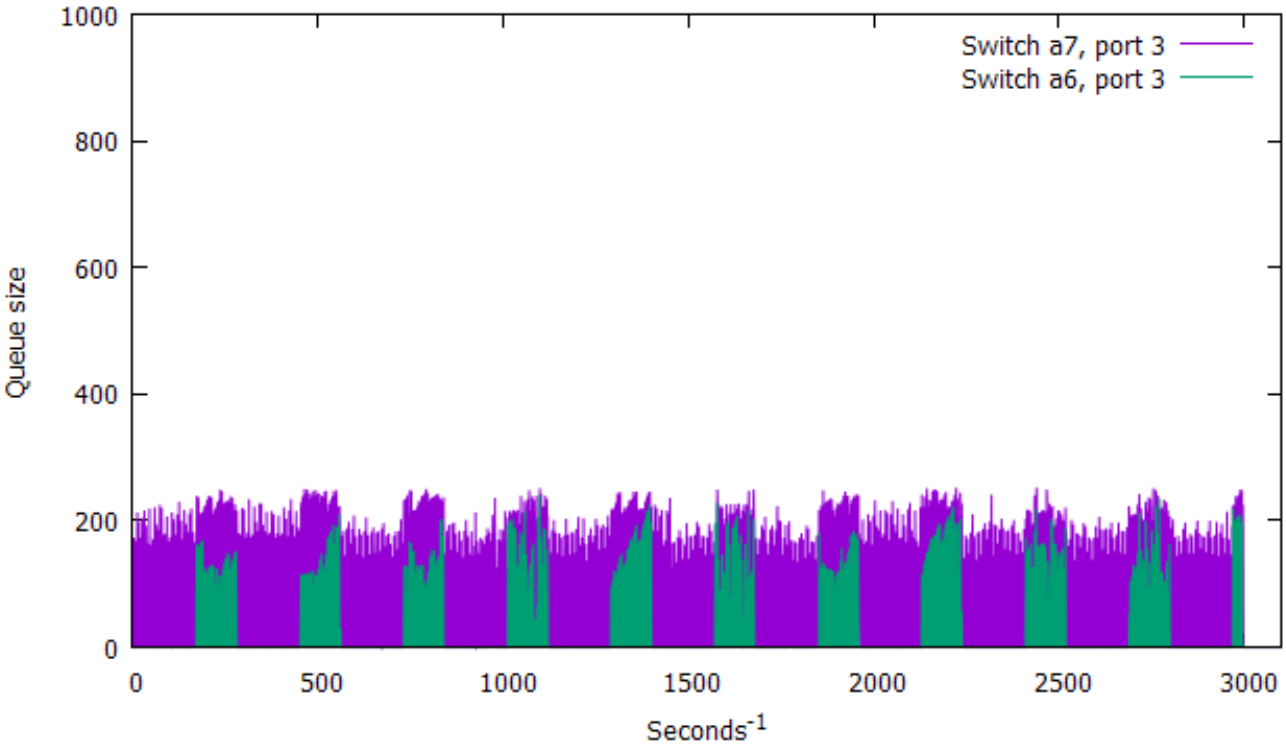


Figure 28. The status of the queue of aggregator switches during the experiment.

4.3.6. Experiment 2: Random port pick forwarding

Experiments with random port-picking forwarding policy are conducted as for the previous ones. The difference with the previous scenario is that having the same connection identifier does not result in using the same path. Instead, outgoing ports are randomly selected for a given source destination pair and maintained for the entire duration of the experiment.

Macro flows

The macro flows experiment analyzes the behavior of having Tenants with high MGB requirements. During this experiment setup we expect that Tenants are always granted to have at least their MGB as accessible resources. If more bandwidth is available the two tenants will share the remaining resources equality.

I scenario

Figure 29 shows the two tenants exchanging data and consuming the DC resources. Figure 30 provides a better probabilistic view of this scenario. The communication takes place from 's6' to 's1'(see Figure 15). Since they

are placed in the same source node, both the Tenants end up sharing the resources of the same outgoing link. This triggers the congestion control mechanism once Alpha tenant wakes up, causing both tenants to be scaled down to their nominal MGB rate.

Regardless of the changed forwarding decision taken at aggregator switches level, the behavior remains the same as for the previous Macro I scenario (the one with static-hash forwarding decision).

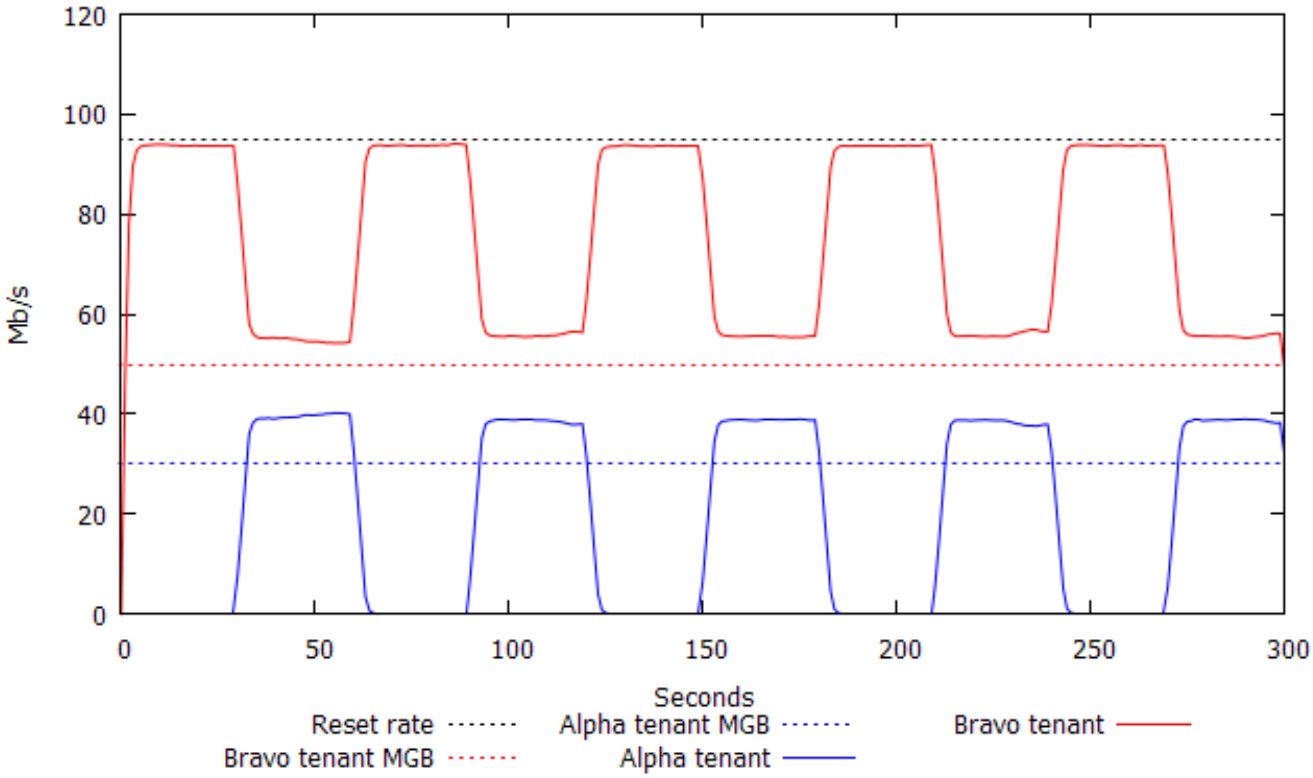


Figure 29. Two Tenants, Alpha and Bravo, concur for resources from the same source node to the same destination node.

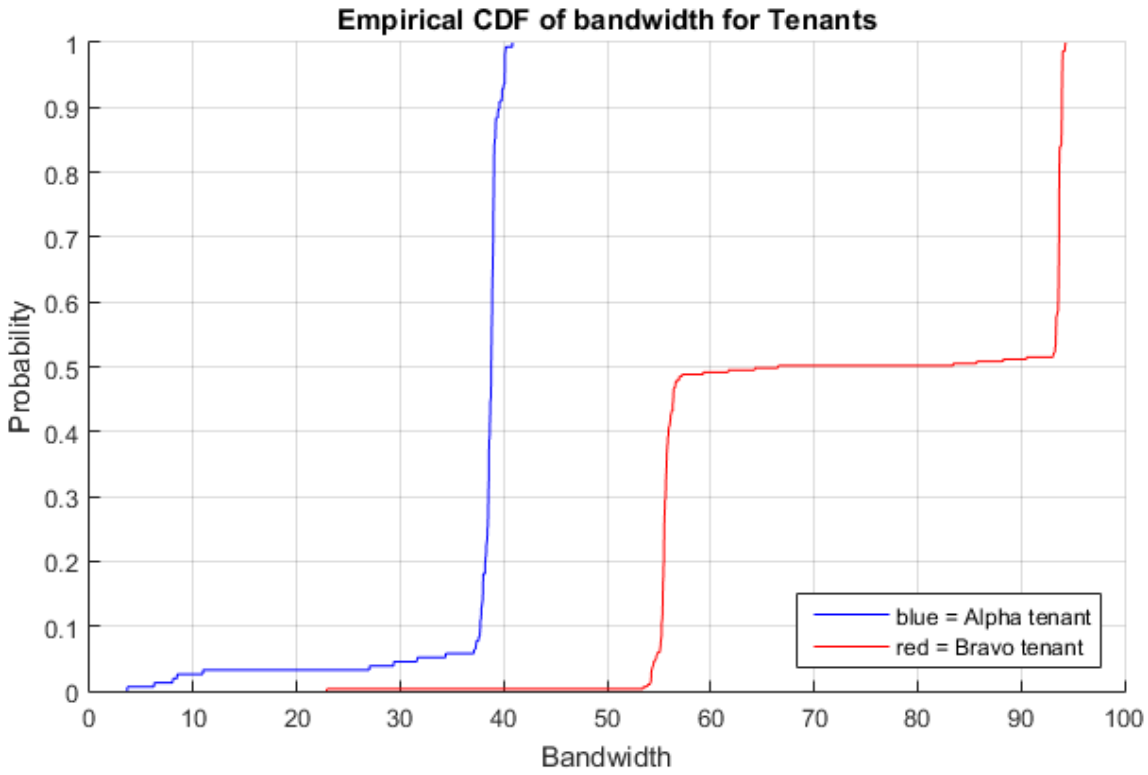


Figure 30. Empirical Cumulative Distribution Function of the bandwidths achieved by the tenants.

Y scenario

Figure 31 shows the two tenants exchanging data and consuming the DC resources. The communication takes place from 's5' and 's6' to 's1'(see Figure 15). Figure 32 shows the distribution of the bandwidth samples of the tenants during the experiment.

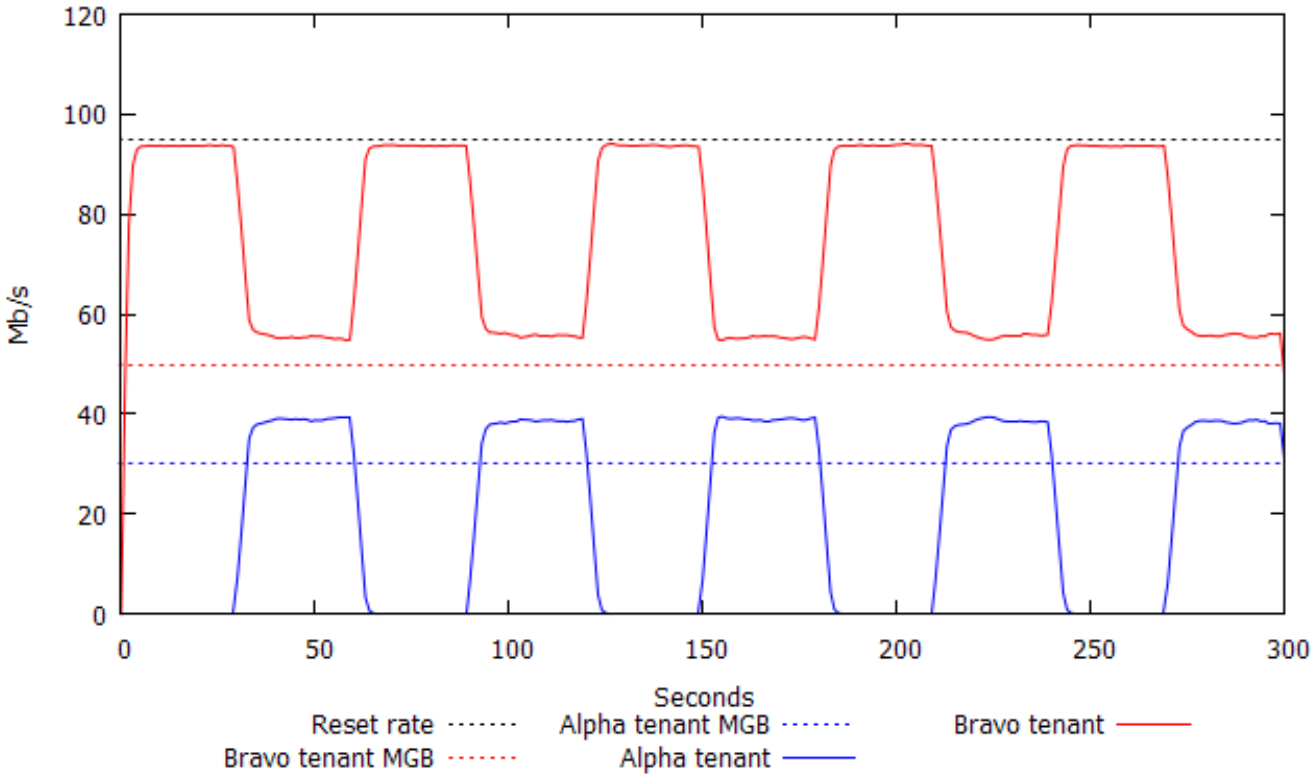


Figure 31. Alpha and Bravo tenants communicate from different source nodes to the same destination node.

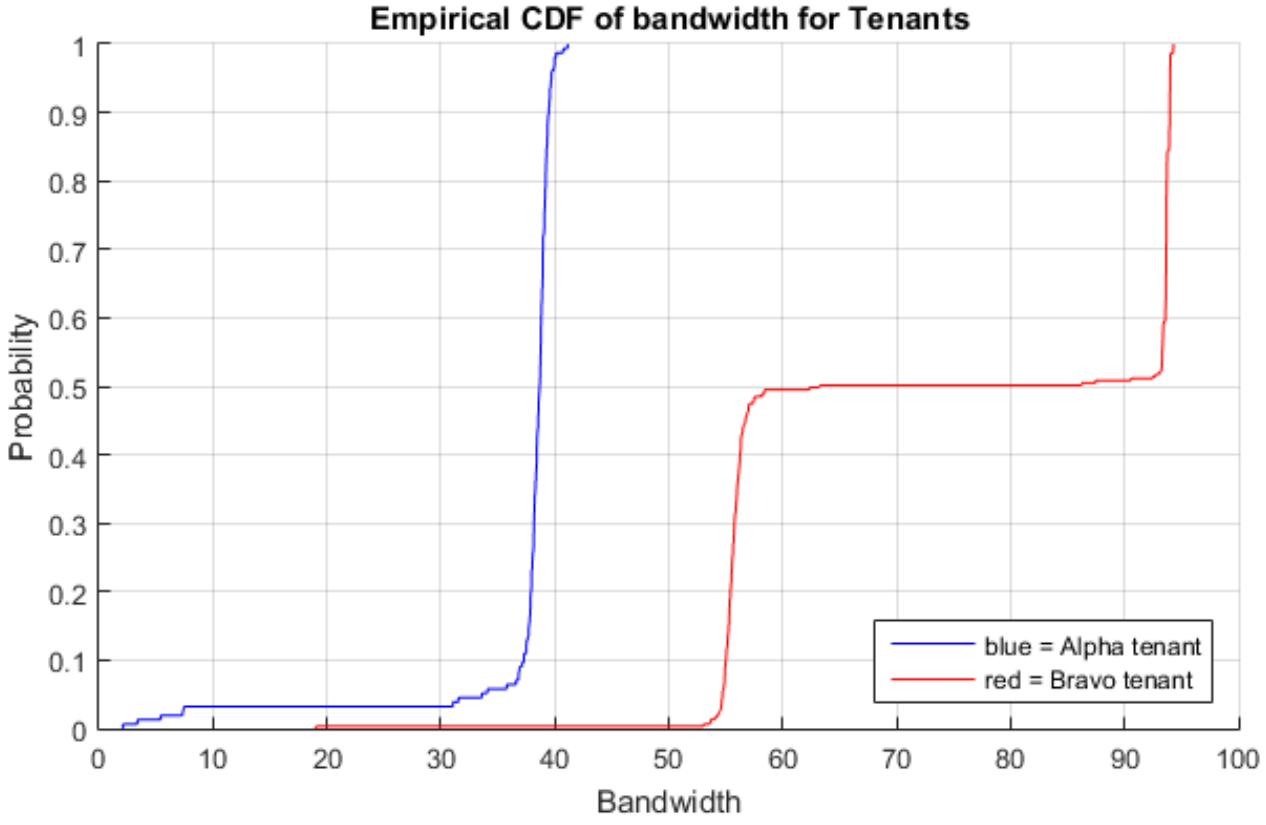


Figure 32. Empirical Cumulative Distribution Function of the bandwidths achieved by the tenants.

As it can be observed in Figure 33, the status of the queue of switch 5 is similar to the previous Y scenario with static hash routing. This is due to the fact that, at some point during the communication, the flows converge to the same link in order to reach the common destination node. This creates a bottleneck (the location depends on the forwarding strategy) which causes the congestion.

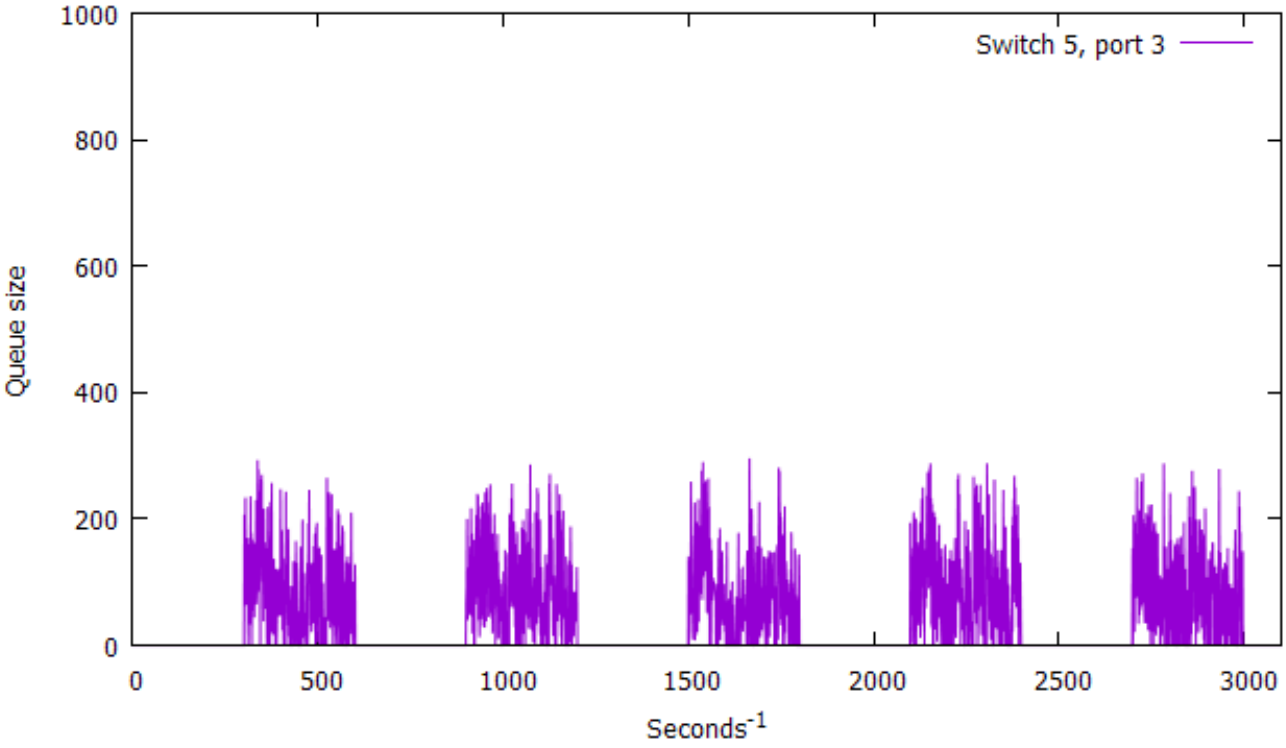


Figure 33. Status of the queues in the aggregator switches during this scenario.

XS scenario

Figure 34 shows the two tenants exchanging data and consuming the DC resources. The communication takes place from 's6' and 's5' to 's1' and 's2' (see Figure 15). Since we are using a full bisection bandwidth topology, we expect them to be routed on different paths, having the ability to communicate at near link speed. However, the random port-pick policy performed a bad selection during this experiment and both flows ended up being forwarded over the same port thus causing congestion.

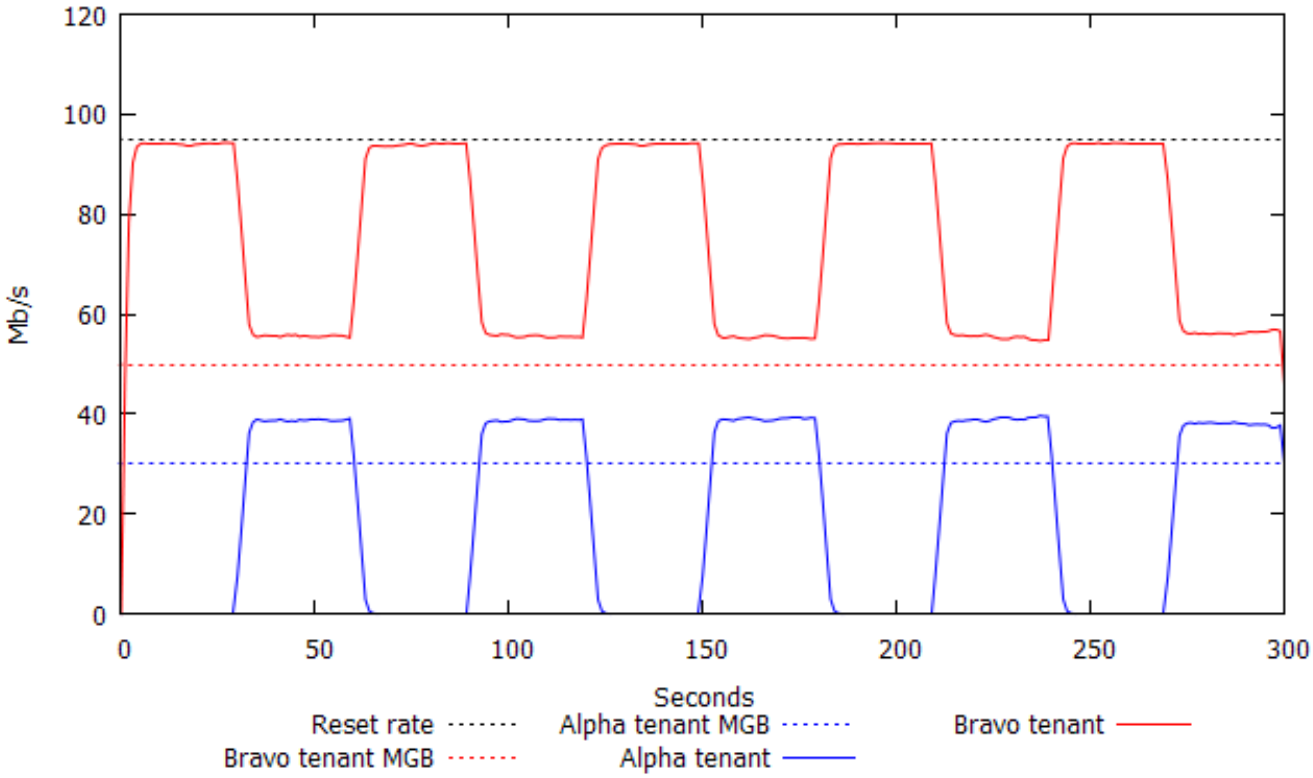


Figure 34. Tenants, Alpha and Bravo, concur for resources from different source node to different destinations nodes but placed under the same Top of Rack switch.

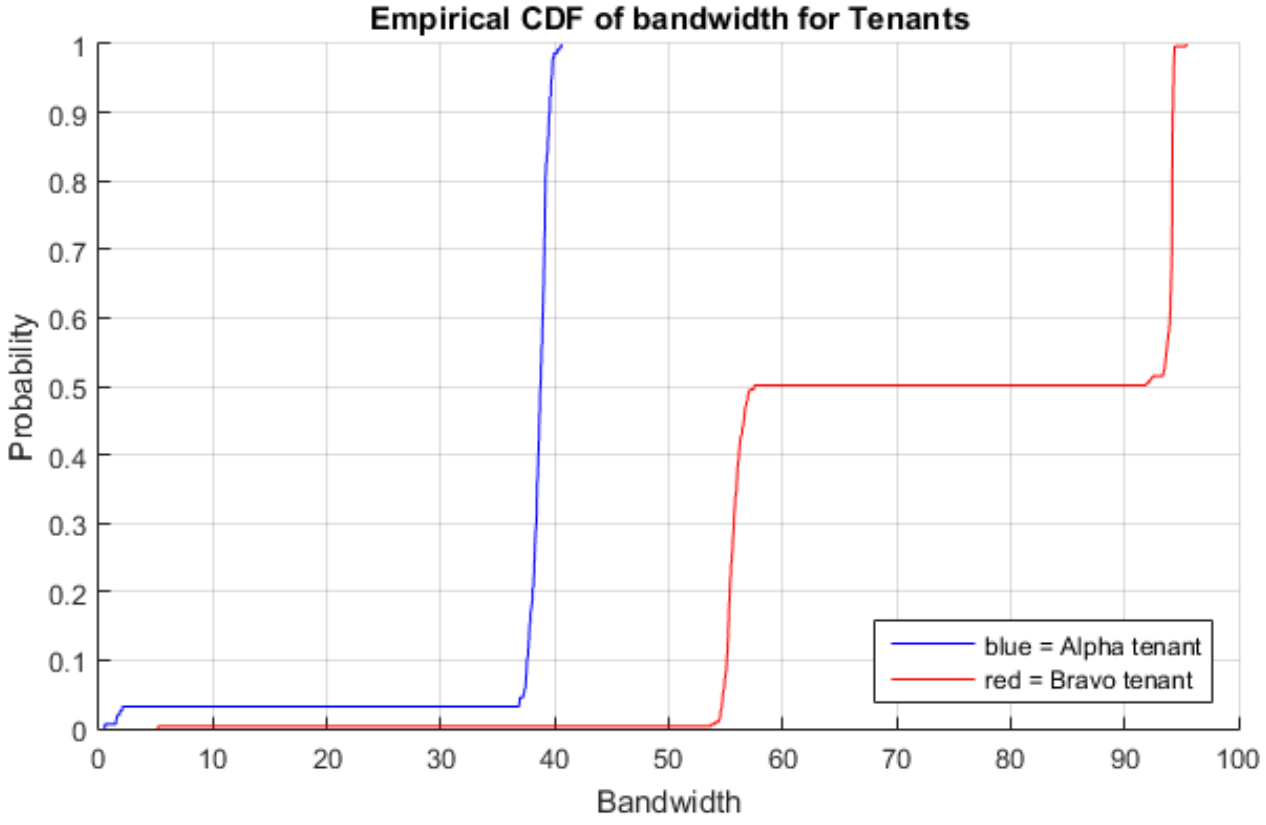


Figure 35. Empirical Cumulative Distribution Function of the bandwidths achieved by the tenants.

We can analyze the situation of the queues of the aggregator switches, and see that the switch *5* is chosen to route both flows. Since again a bottleneck is introduced, we got similar performances of the previous scenario. This is a possible drawback when using a random pick for ports.

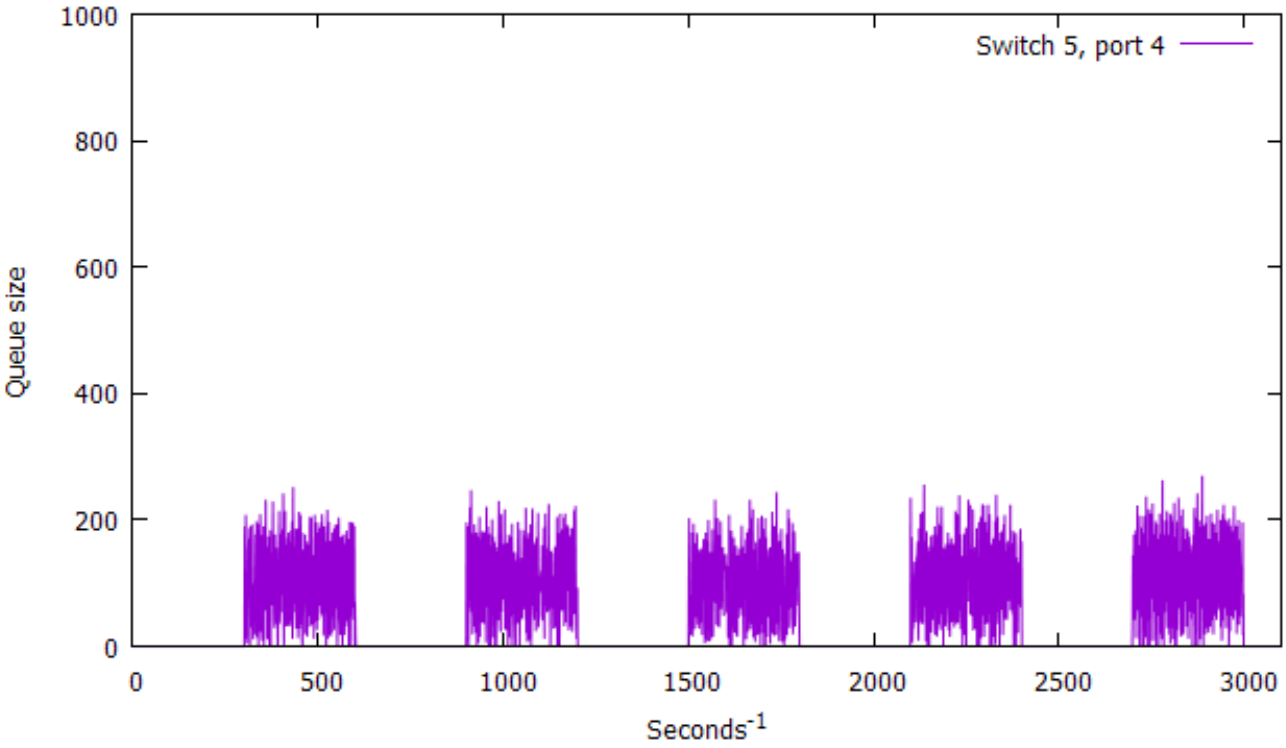


Figure 36. Status of the queues in the aggregator switches during this experiment.

XD scenario

Figure 37 shows the two tenants exchanging data and consuming the DC resources. The communication takes place from 's6' and 's7' to 's1' and 's3' (see Figure 15). Now both source and destination nodes are connected to different Top of Rack switches.

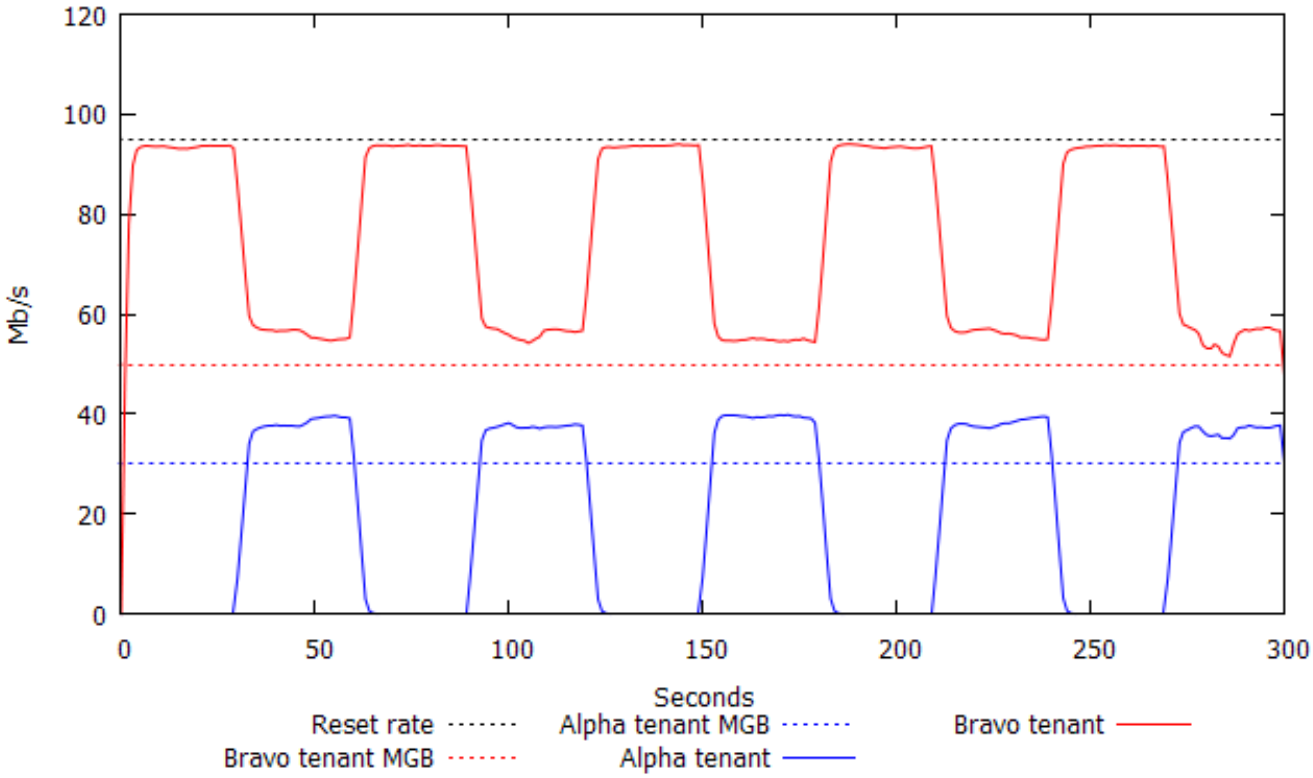


Figure 37. Tenants, Alpha and Bravo, concur for resources from different sources nodes to different destinations nodes.

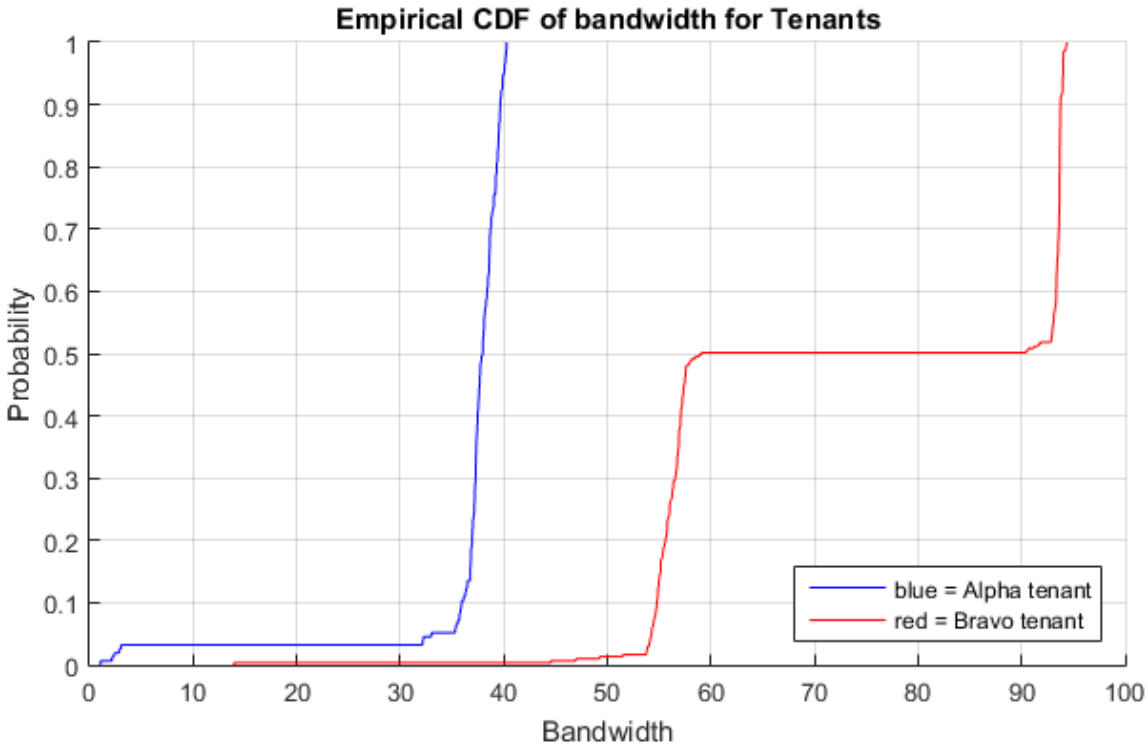


Figure 38. Empirical Cumulative Distribution Function of the bandwidths achieved by the tenants.

Loking at this particular scenario, you can notice that the flow starting from server 6 is routed by switch 6 to switch 7. On such switches near the core router we still have two possible path to the destination, so the probability of picking the one which is not used is 50%. During this experiment switch 7 routed both flows on the same port thus causing congestion.

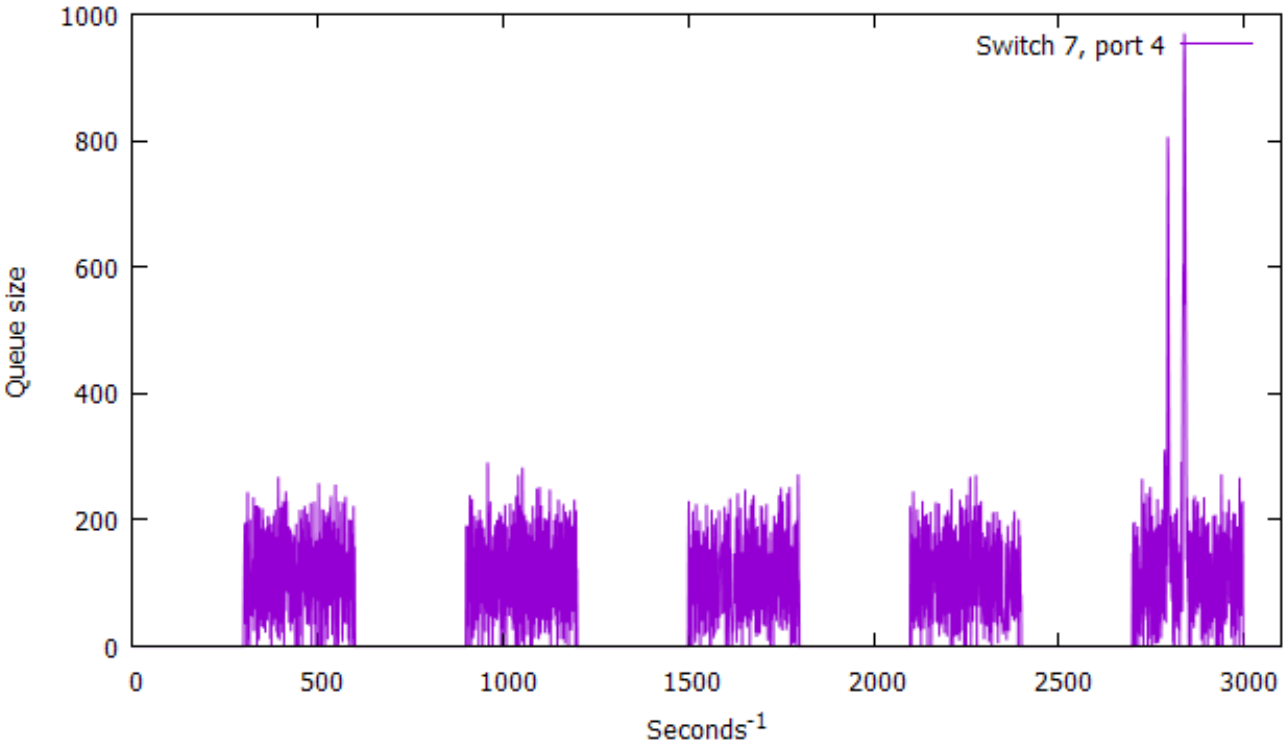


Figure 39. Status of the queues in the aggregator switches during this experiment.

Micro flows

During this experiment setup we expect that Tenants are always granted to have at least their MGB as accessible resources. If more bandwidth is available the two tenants will share the remaining resources equality.

I scenario

Figure 40 shows the four tenants exchanging data and consuming the DC resources. The communication takes place from 's6' to 's1'(see Figure 15). Since they are placed in the same source node, all the Tenants end up sharing the resources of the same outgoing link. This triggers the congestion control mechanism on the source node, which limits the flows from reaching the link rate, and equally splits the resources between the tenants.

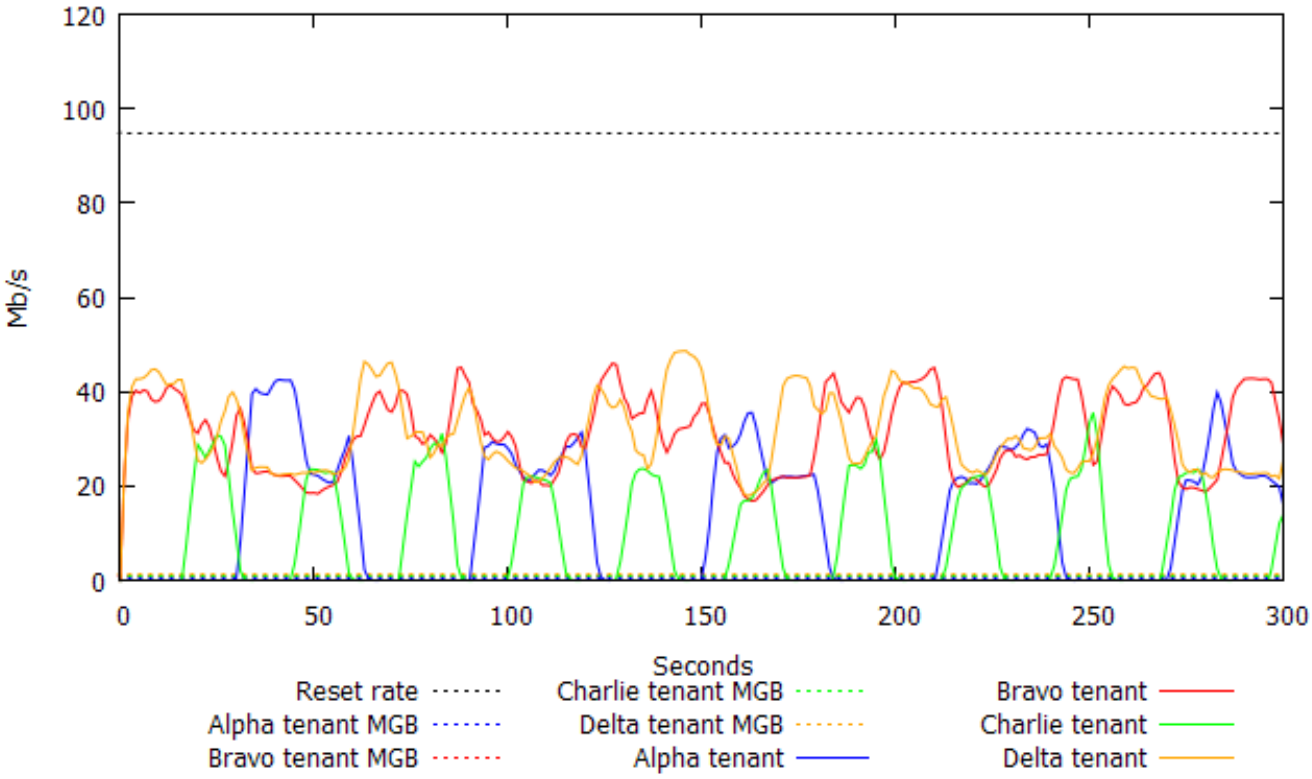


Figure 40. The four tenants concurring for resources in the scenario where the same source and destination node are used.

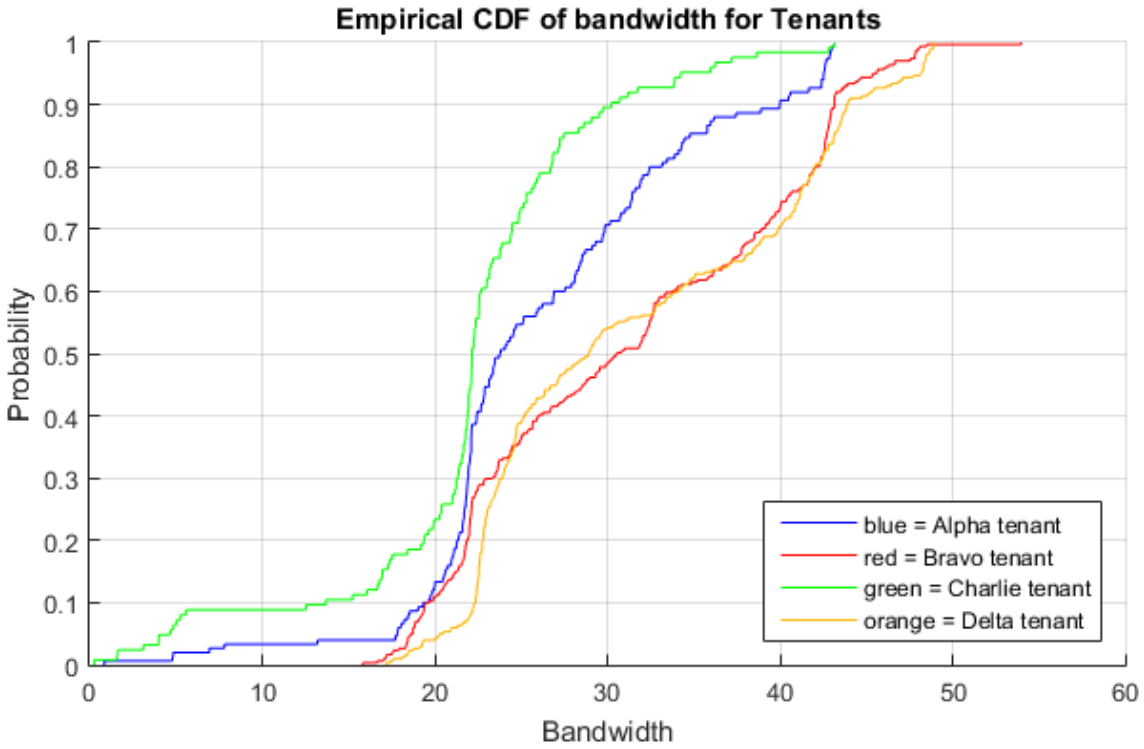


Figure 41. Empirical Cumulative Distribution Function of the bandwidths achieved by the tenants.

Y scenario

Figure 42 show the four tenants exchanging data and consuming the DC resources. The communication takes place from 's5' and 's6' to 's1'(see Figure 15). In this setup every source node has a continuous Tenant and an intermittent Tenant, while the destination remains the same for every tenant. Congestion is now happening in different part of the topology, as it can be seen in Figure 44, but the major aggregator for the flows is switch *5*, which is the Top of Rack switch of the server *5* and server 6.

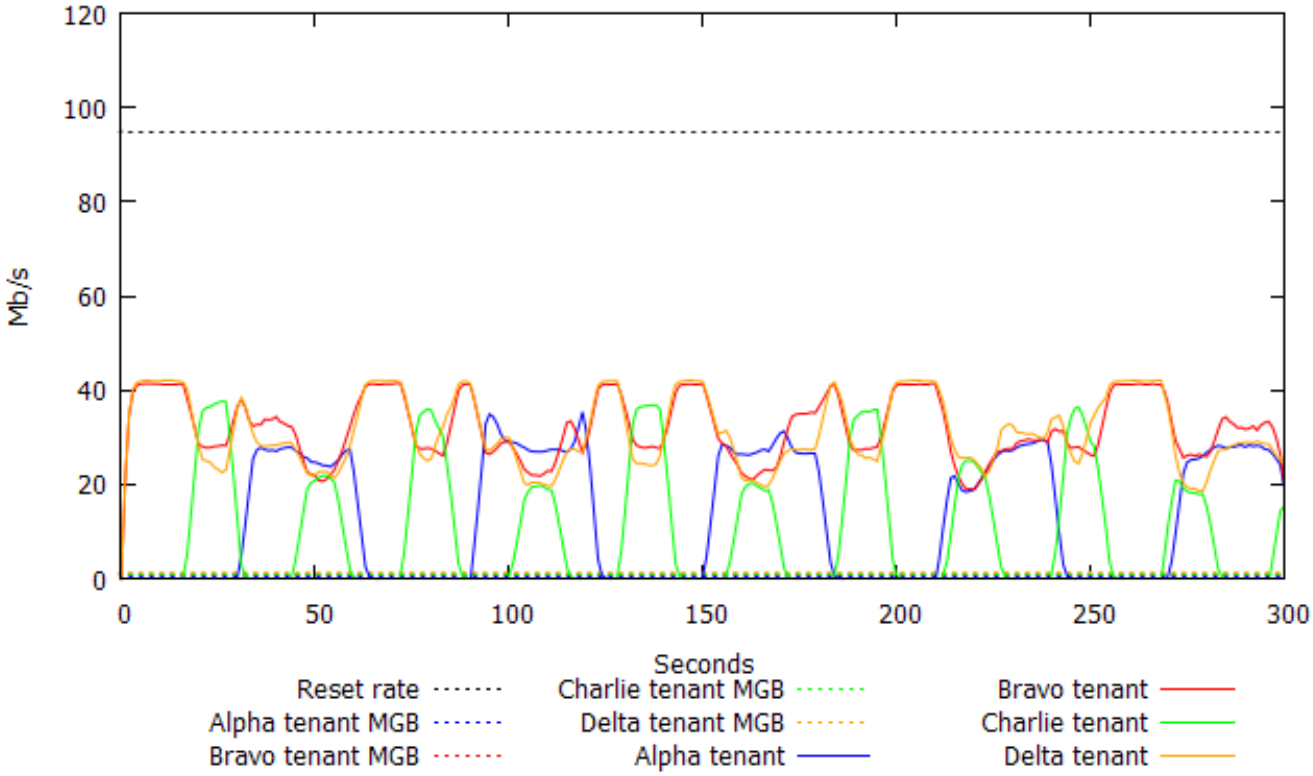


Figure 42. Four tenants communicate from the two different sources to the same destination node.

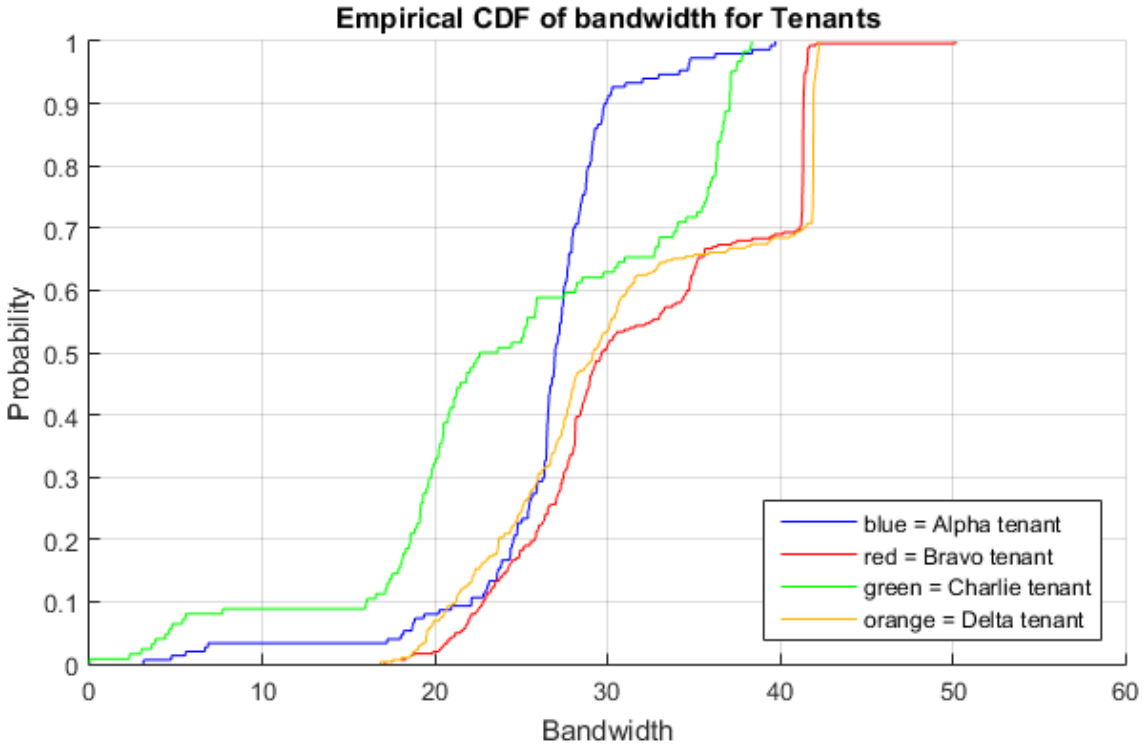


Figure 43. Distribution of bandwidth between tenants.

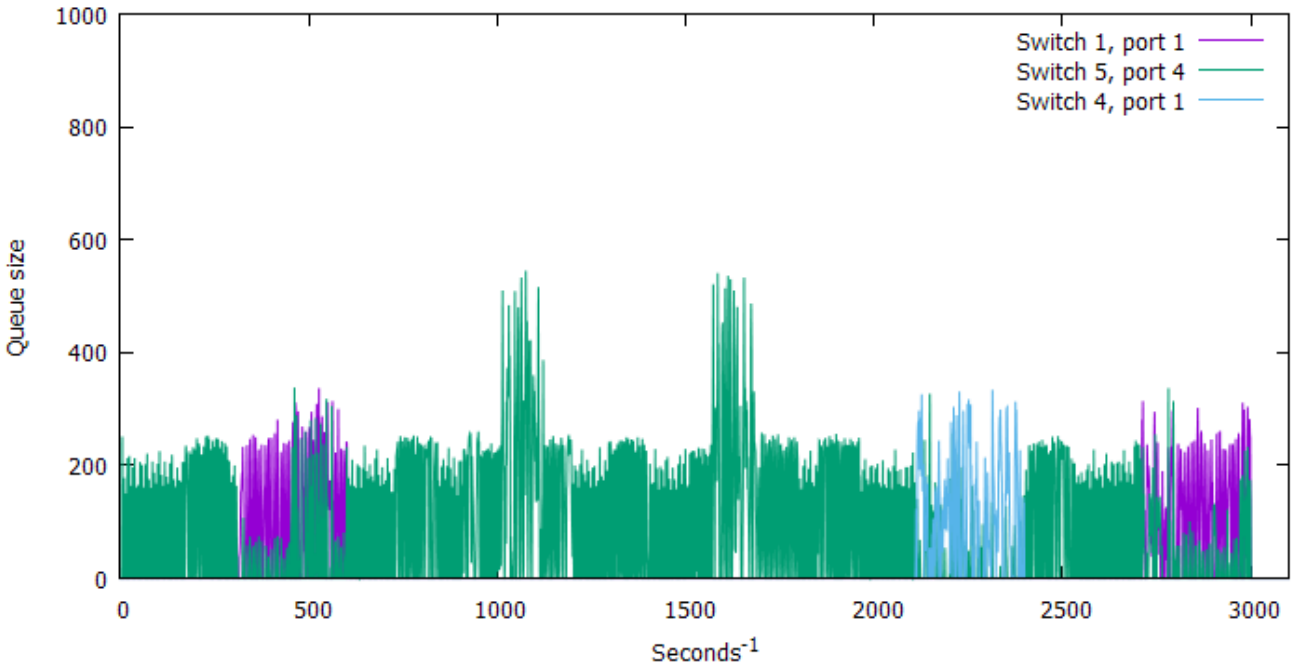


Figure 44. Status of the queues in the aggregator switches.

XS scenario

This scenario gives us an idea of what is happening when multiple tenants communicate from different sources to different destinations. In this setup,

two tenants, one intermittent and the other continuous (Alpha and Bravo), communicate with the same destination, and the remaining ones (Charlie and Delta) with the other. As for the other scenarios, the random-pick forwarding strategy took sub-optimal choices resulting in a non-efficient routing configuration. Figure 45 shows the overall bandwidth achieved by all tenants.

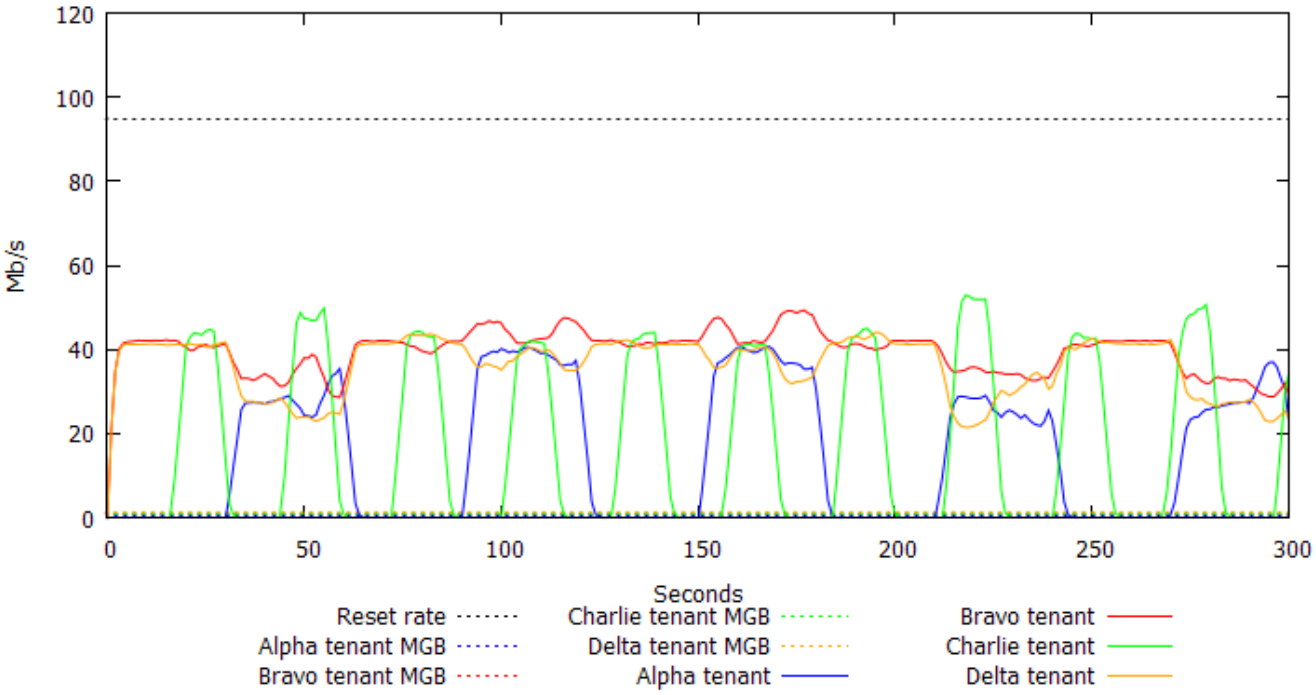


Figure 45. Status of the queues in the aggregator switches.

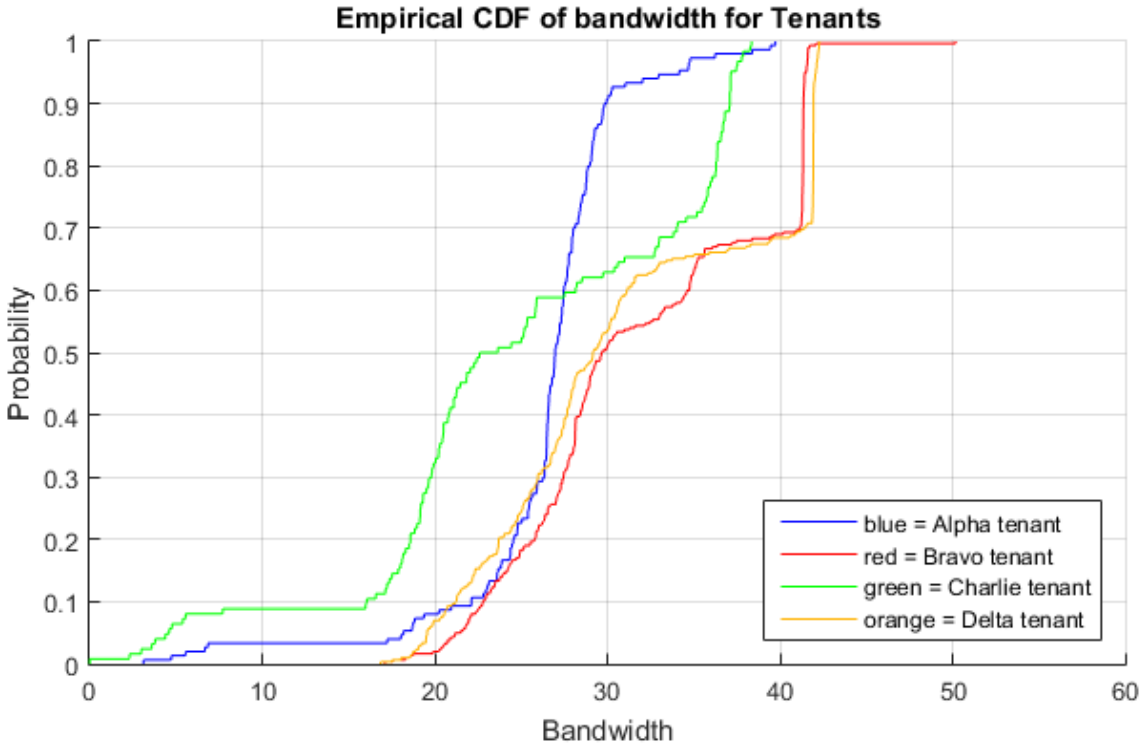


Figure 46. Distribution of bandwidth between tenants.

Once again, at switch *5*, the policy randomly picked the same path for all the flows (see Figure 47). Under the given conditions the bandwidth is still acceptable, but the alternative paths with equal cost are not used.

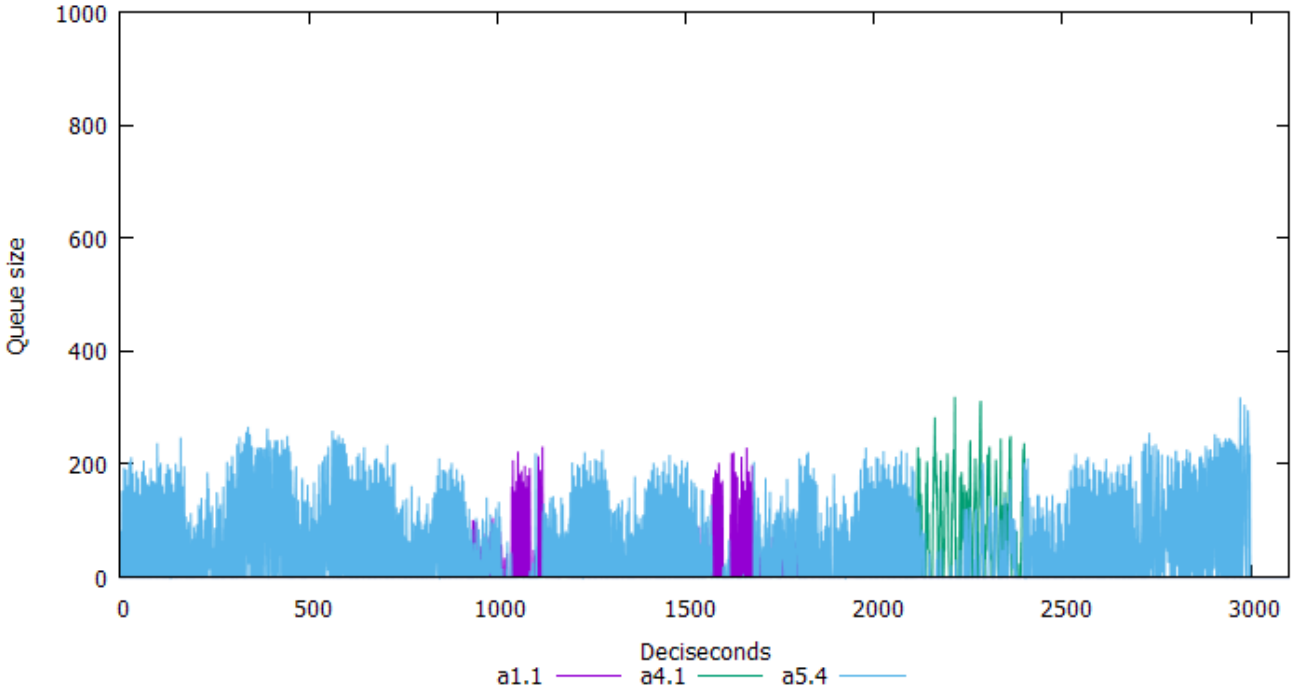


Figure 47. Status of the queues in the aggregator switches.

XD scenario

In this scenario all the tenants have a different destination, while the next hop is still randomly picked. During the initial classification of the flows, the forwarding policy put tenants Alpha and Bravo on the same route, while Charlie and Delta have been put on another one. In Figure 48 we can clearly observe the result of this selection in terms of queue occupancy when the intermittent tenants are active.

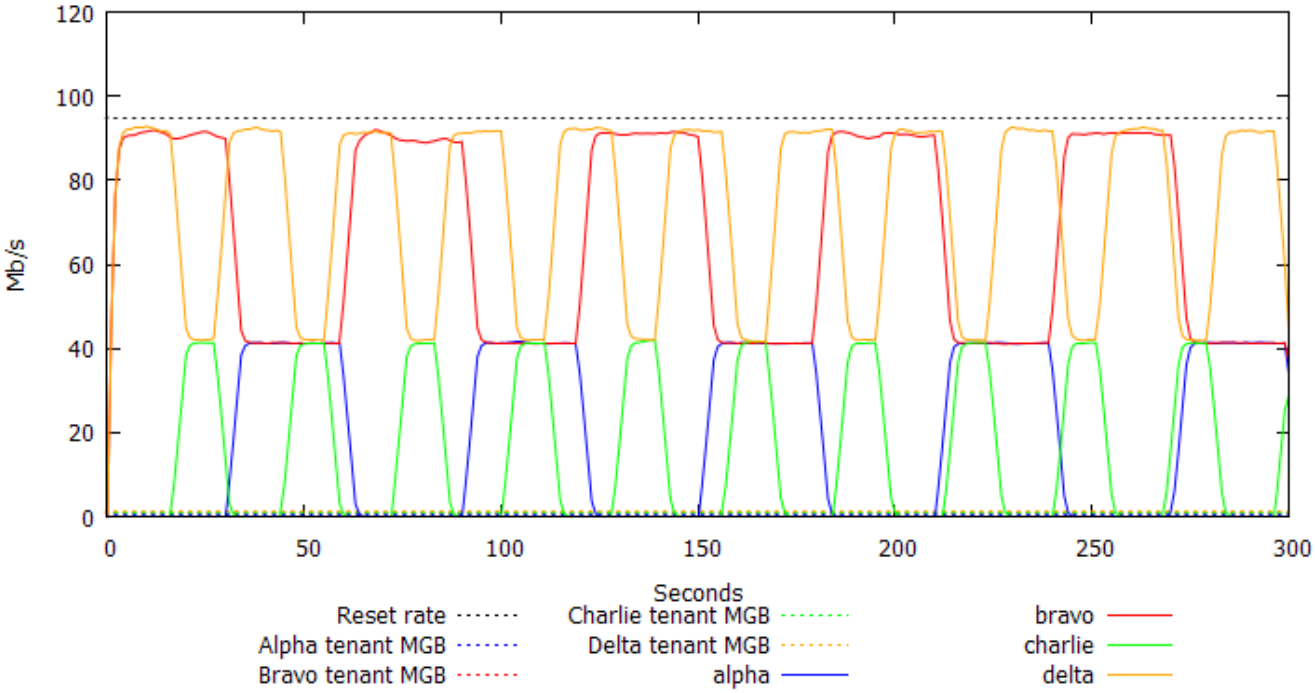


Figure 48. Status of the queues in the aggregator switches.

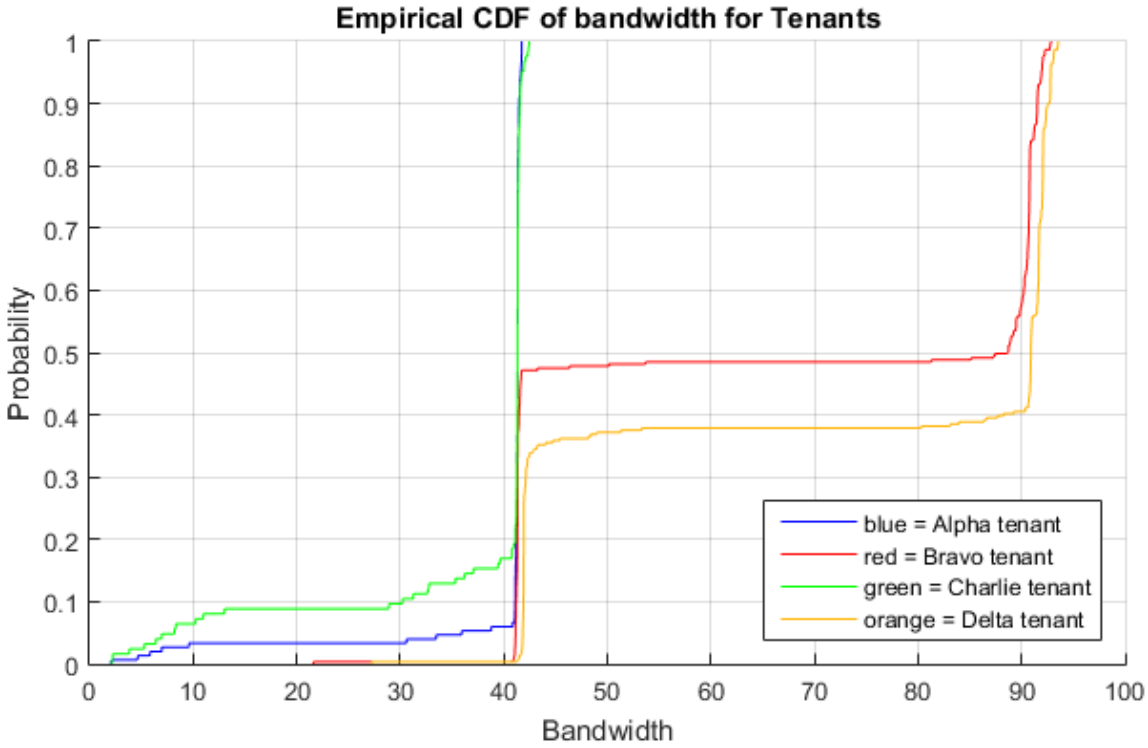


Figure 49. Distribution of bandwidth between tenants.

Here the switches which took the sub-optimal decision are switch 6 and 5. In Figure 50, the report of the switches queue occupancy clearly highlight this problem.

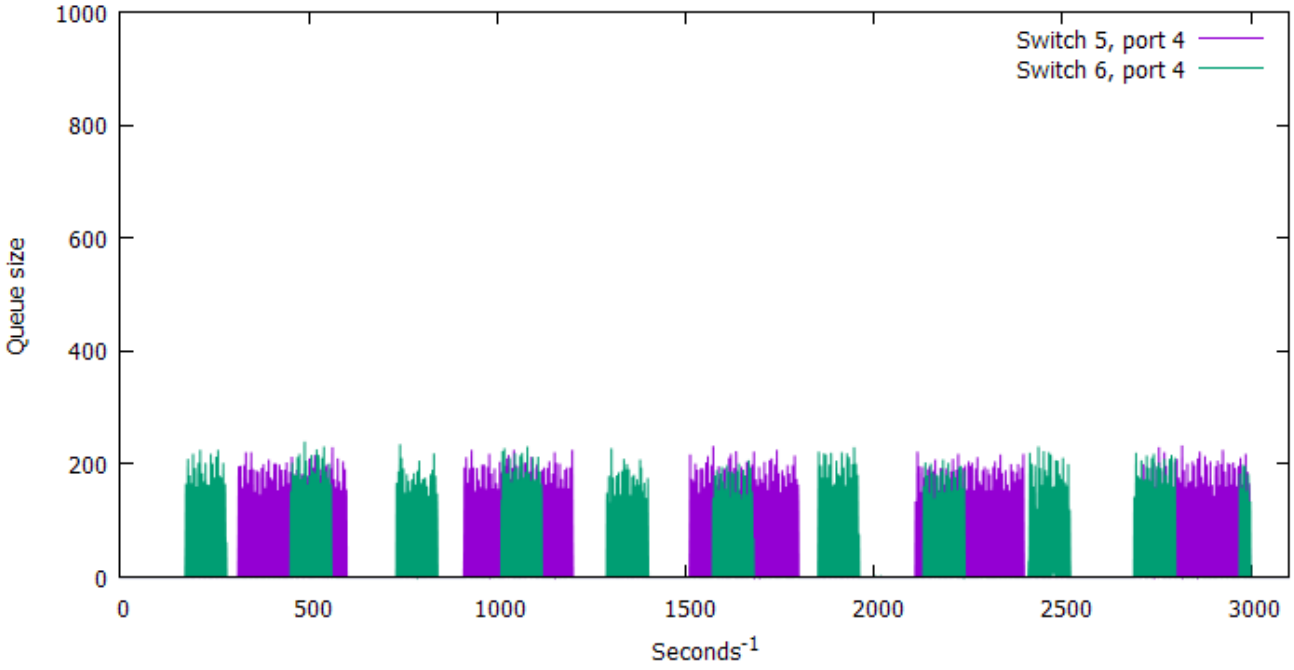


Figure 50. Status of the queues in the aggregator switches.

4.3.7. Experiment 3: Port load pick forwarding

Finally our last set of experiment will be done using the port load pick forwarding policy. This policy selects the outgoing port taking into account the load of all available ports. Notice how this is still a greedy approach based on local knowledge (i.e. there is no global optimization).

Macro flows

This experiment analyzes the behavior of having Tenants with high MGB requirements. During this experiment we expect that Tenants are always granted at least their nominal MGB. If more bandwidth is available, the two tenants will share the remaining resources equality.

I scenario

This scenario, as we expected, does not differs from the previous ones (see Figure 51), since having the same source creates a bottleneck at the source node. Both tenants are then scaled down to their nominal MGB and then share the remaining bandwidth.

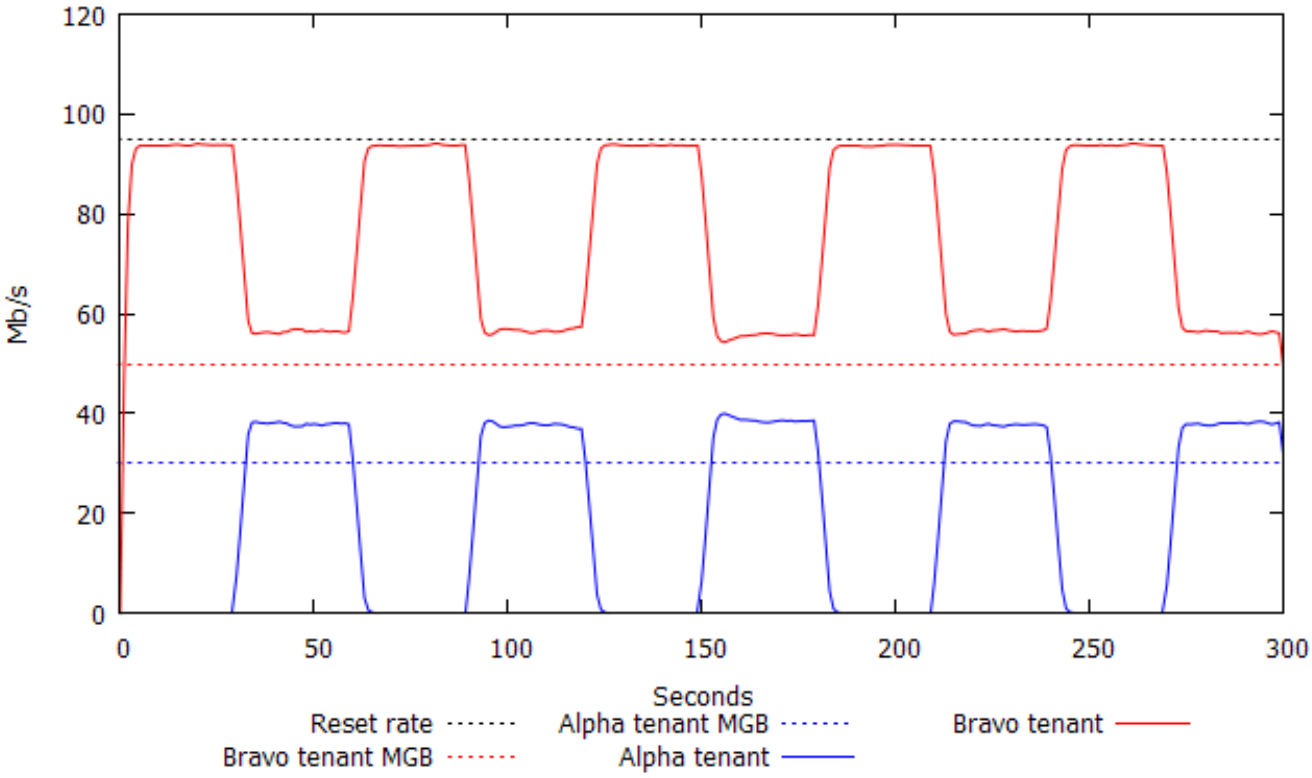


Figure 51. Two tenants which communicate from the same source node to the same destination node.

Y scenario

This scenario also does not different from the previous one leveraging on the random pick forwarding strategy (see Figure 52).

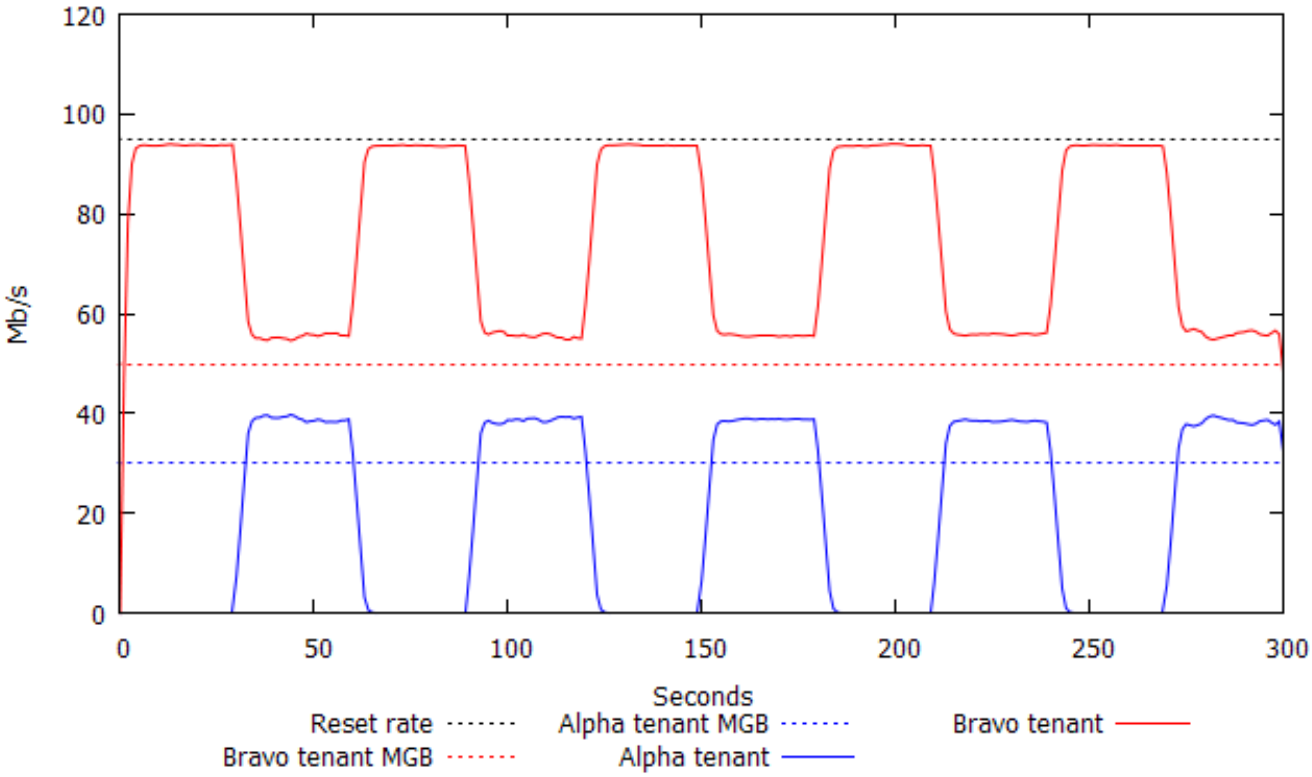


Figure 52. Two tenants which communicate from different sources nodes to the same destination node.

What changes from the previous scenarios, is where the congestion takes place (switch 3 in this case, see Figure 53).

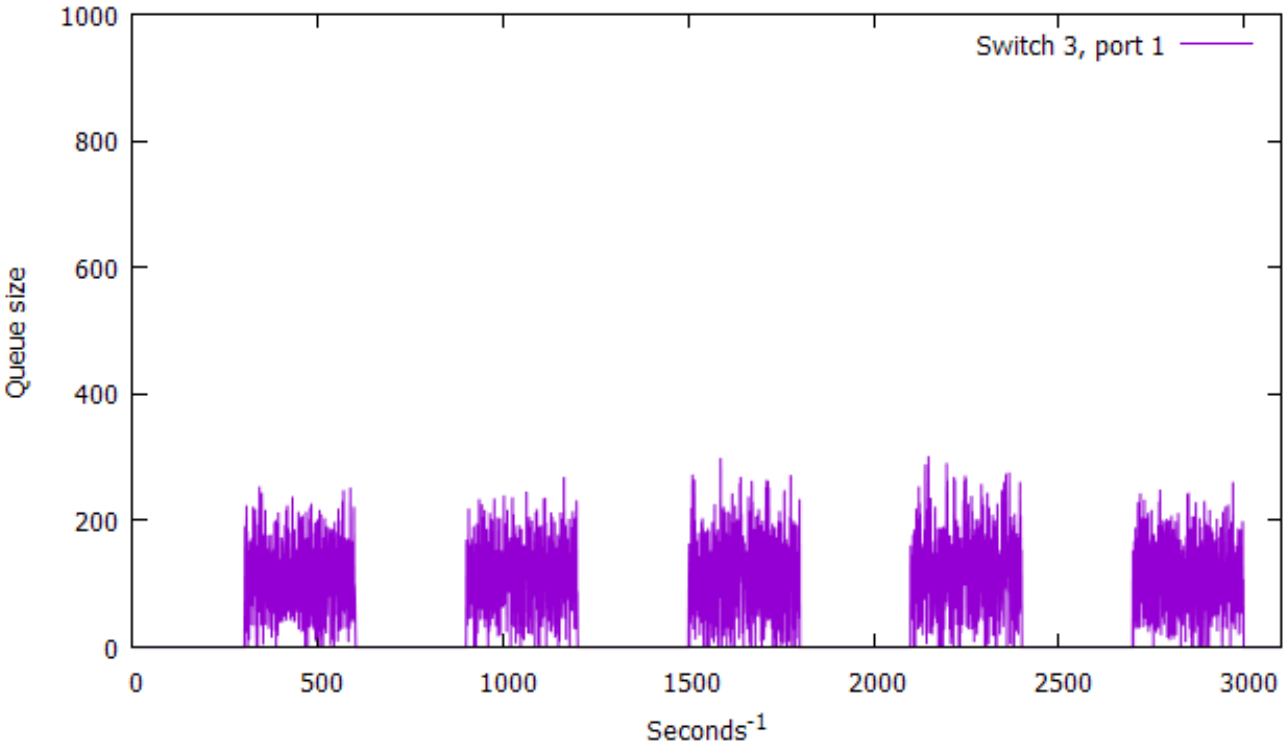


Figure 53. The status of the queues of the switches during the experiment.

XS scenario

As expected in this scenario, since there are multiple destinations (even under the same Top of Rack) and the communication starts from different sources, it should be possible to achieve full line rate speed. Figure 54 shows that expectations are respected, and that no congestion problems arise during the experiment.

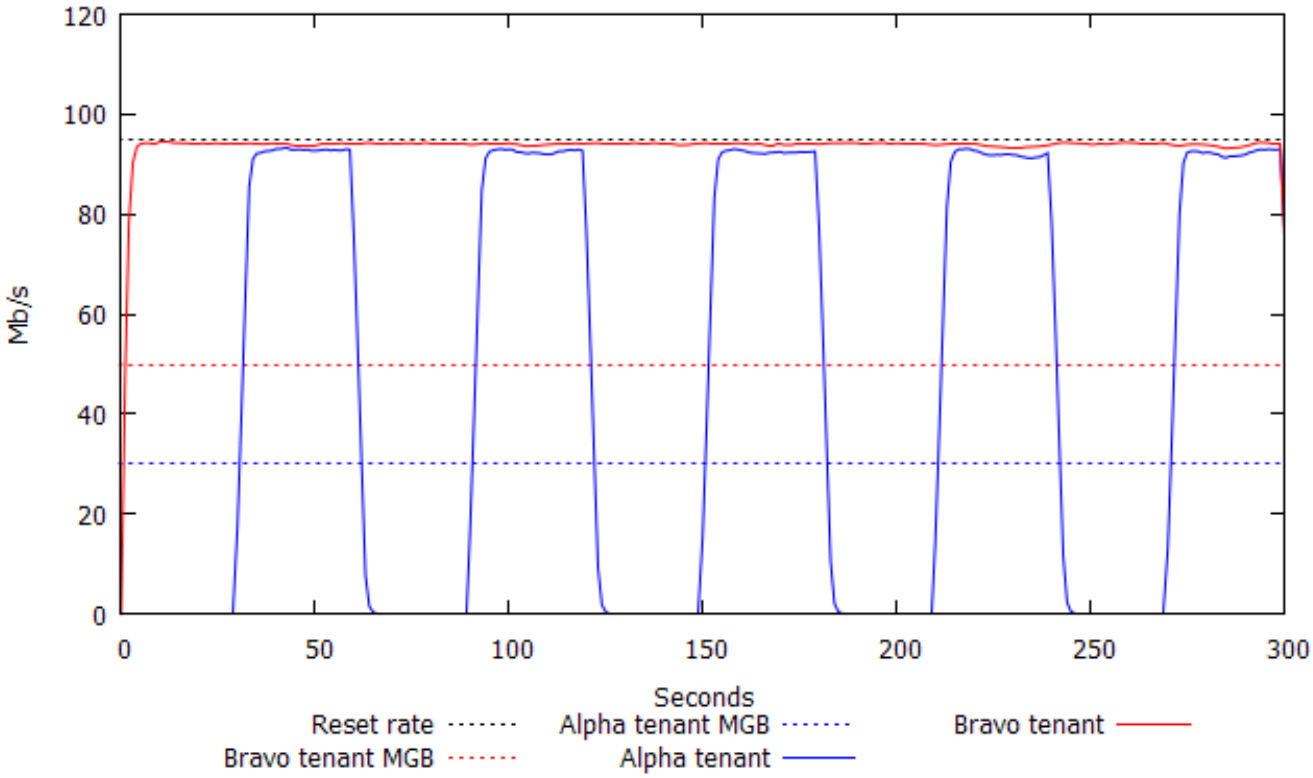


Figure 54. Two tenants which communicate from different source nodes to different destinations nodes under the same Top of Rack.

XD scenario

In this scenario, where tenants on different sources nodes communicates to different destinations, we also expect to exploit the full bisection bandwidth. Figure 55 shows that this is indeed the case.

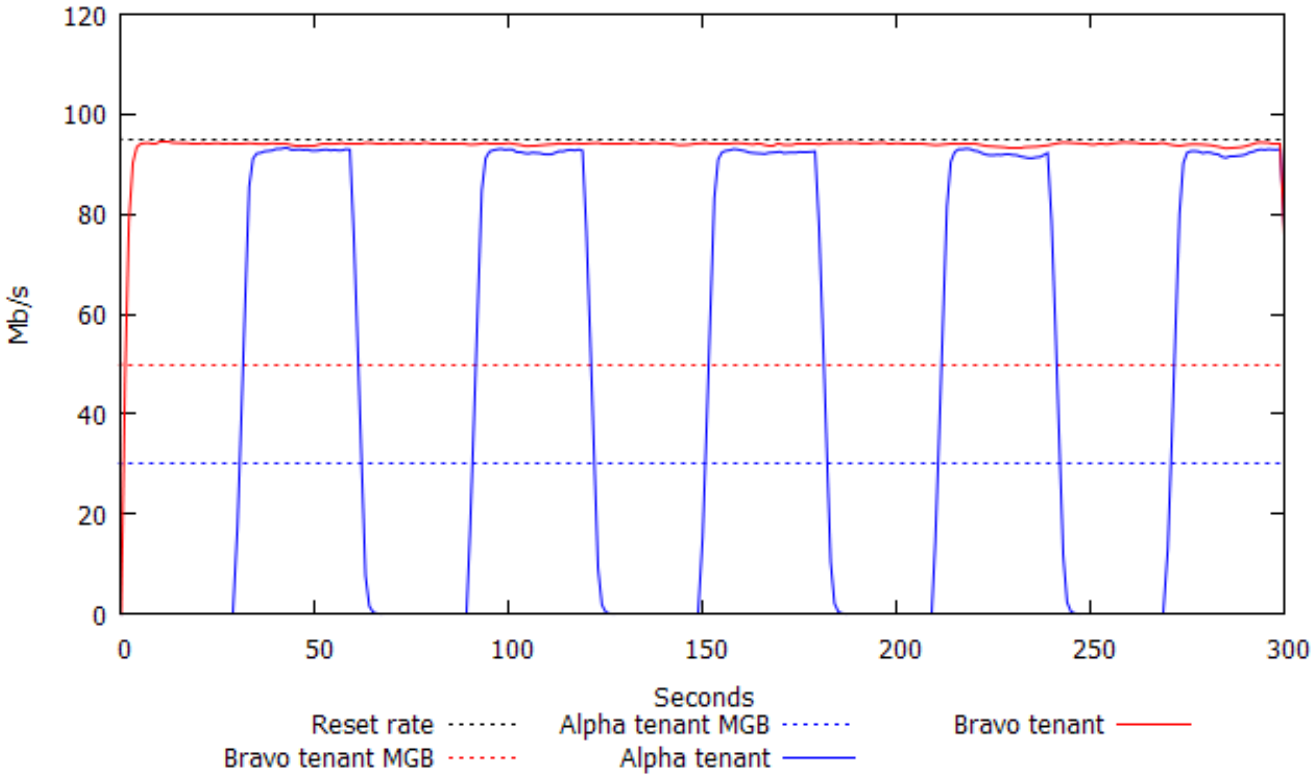


Figure 55. Two tenants which communicate from different source nodes to the different destination nodes.

Micro flows

During this experiment we expect that Tenants are always granted at least their nominal MGB. If more bandwidth is available, the two tenants will share the remaining resources equality.

I scenario

This scenario, as it can be seen in Figure 56, does not change compared to the one already seen in the previous experiments. Again, since the source node for all the tenants is the same, the network capacity of the source node becomes the bottleneck of the communication.

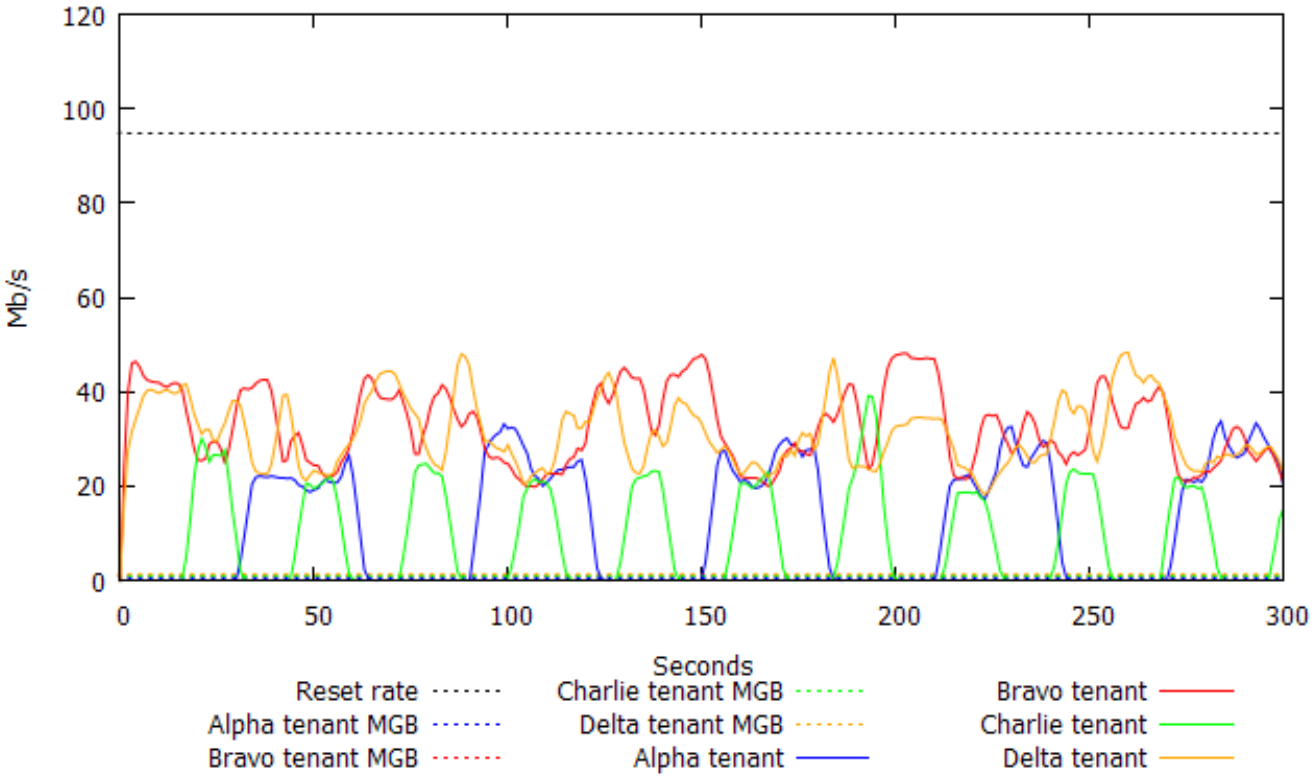


Figure 56. Four tenants which communicate from the same source node to the same destination node.

Y scenario

In this scenario tenants are placed on two different source nodes but share the same destination node. Figure 57 show that in terms of bandwidth we have similar performances of what we got with all the others micro-flows experiments.

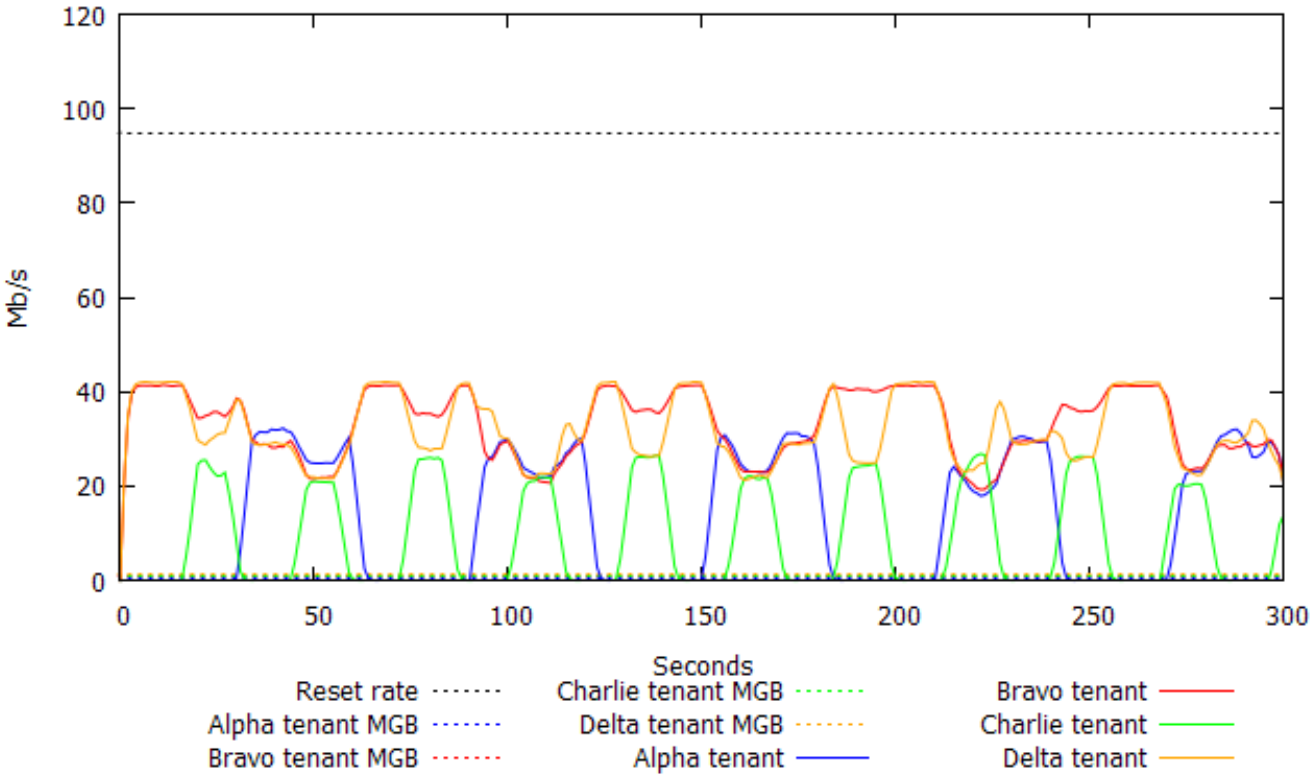


Figure 57. Four tenants which communicate from the different sources to the same destination.

What changes here, again, is where the congestion happens in the network (see Figure 58). Notice how when an intermittent tenant wakes up, switch 1 (the one immediately above the destination node) starts to report a congestion, thus becoming the one indirectly signalling the congestion to all tenants.

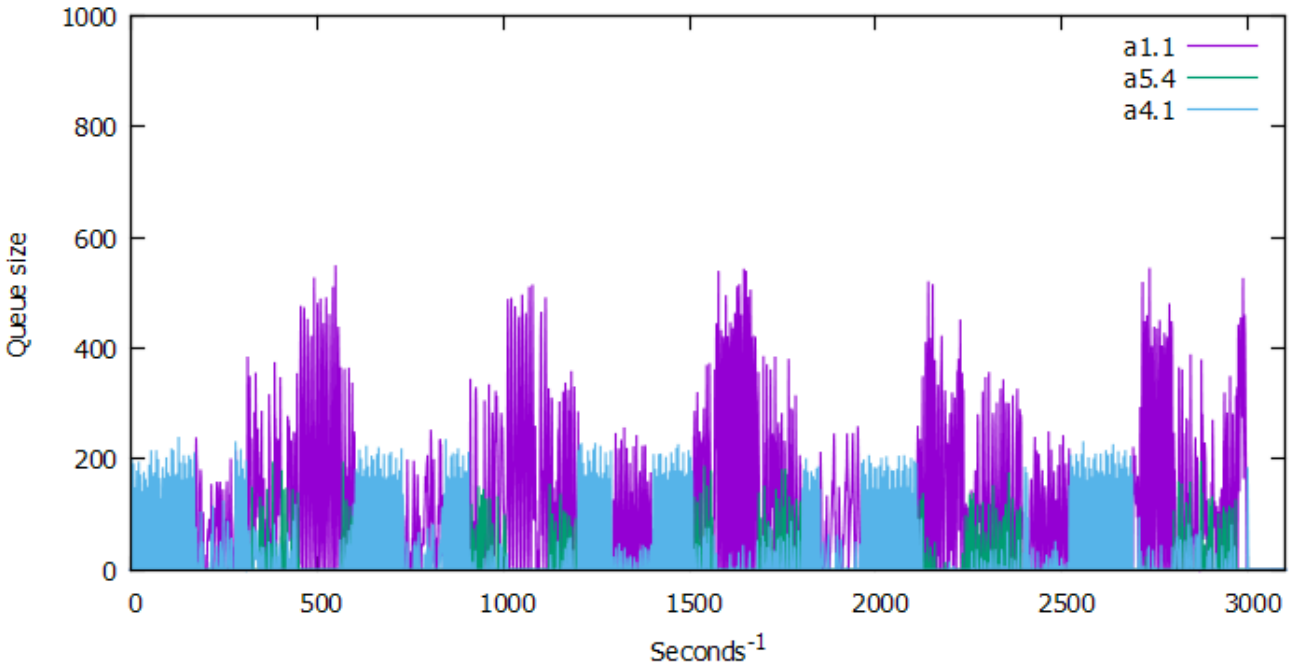


Figure 58. Situation of the queues in the aggregator switches.

XS scenario

In this scenario we will see how this forwarding policy behaves when different source nodes communicate with different destination. As is can be seen in Figure 59, the bandwidths are redistributes along the possible paths to reach the different destinations. In this configuration, since Bravo and Charlie aims for the same destination they directly affect each other.

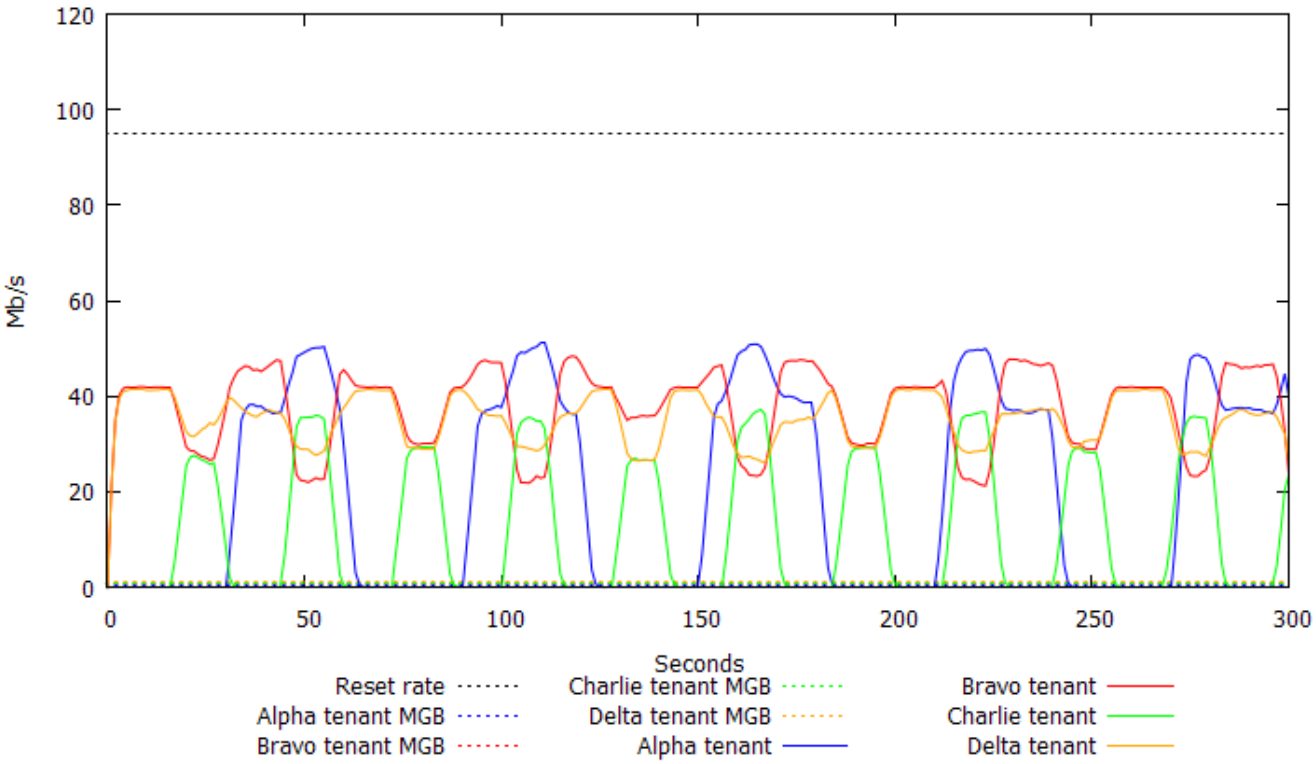


Figure 59. Four tenants which communicate from the different sources to the different destinations.

The situation of the queues in Figure 60 show us that the policy aggregates three tenants in such way that their traffic merges in switch 3, and leave tenant Alpha on a privileged path. Still, since Alpha is aiming at a destination node in use from another tenant, they will create a congestion on the port 3 of the switch 1, which is the one leading to server 1.

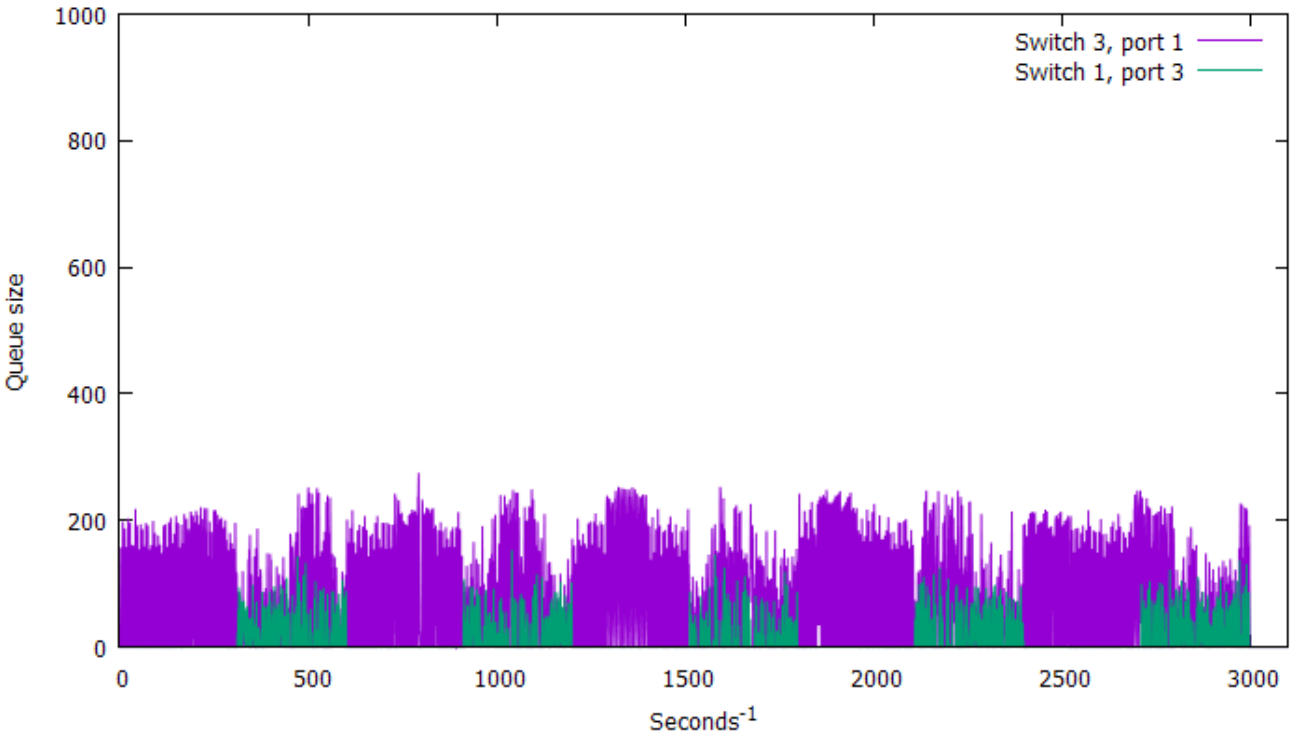


Figure 60. Situation of the queues in the aggregator switches.

XD scenario

Finally this last scenario will show a possible configuration of the flows when all the tenants have different source and different destinations. As we say the forwarding policy is a little more intelligent, but applies a greedy chose on the flow, so can be that the decision taken are good for the local switch, but triggers congestion in another one (which has no alternative paths). Figure 61 shows that every tenant successfully achieve the full bandwidth for its communication, since the forwarding policy organize their flows in such way that a port is never overloaded.

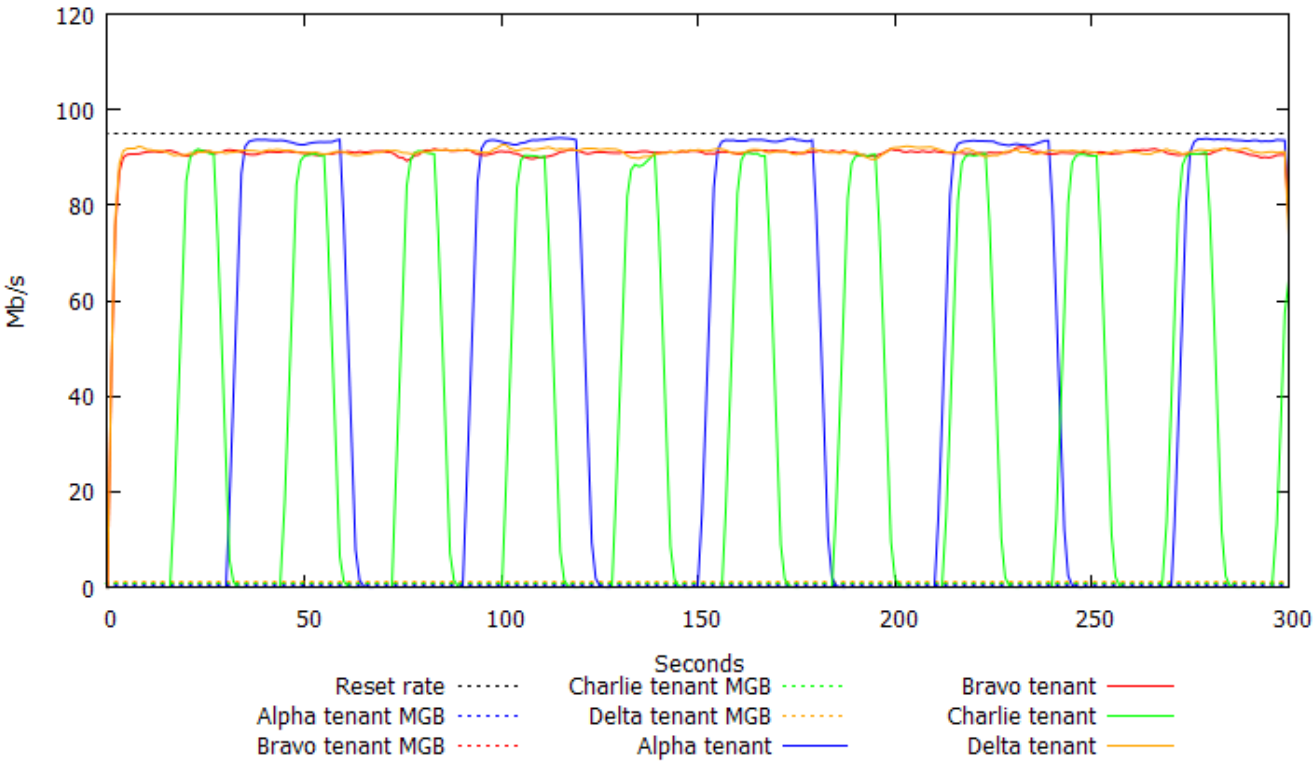


Figure 61. Four tenants which communicate from the different sources to the different destinations.

4.4. Policies for Data Center TCP-like behavior

4.4.1. Introduction

One of the biggest challenges for network administrators in DCs is how to build highly available and highly performant networking infrastructure to address different requirements of data center applications. Especially machine to machine communication between servers in a data center can be severely impacted if specific requirements are not met. These requirements are as follows: 1) low latency for short flows, 2) high burst tolerance and 3) high utilization for long flows.

The first two requirements stem from the partition/aggregate scheme that is often used by data center applications. In the partition/aggregate scheme, a request from an application is broken (partitioned) into several pieces and these pieces are passed to several workers. The responses of these workers are aggregated to produce a result. The latency is the key metric here and the application is typically allocated a few hundred of milliseconds to answer the request. To satisfy the requirement, worker nodes are assigned tight deadlines, usually less than 100ms. When a node

misses its deadline, the request cannot be completed or the quality of the result is decreased. The high utilization for long flows requirement stems from the background machine to machine communication that is used to continuously update internal data structures and databases of data center applications.

It is common to use low-cost switches at the top of the rack to provide connections for data center servers. These commodity switches are often shallow buffered as it is expensive to built a high port density switch with large memory per port. Unfortunately, using these shallow switches can cause several performance problems with TCP Incast as the very severe one.

TCP Incast is a TCP throughput collapse. The main issue is that synchronized workload can result in packets overfilling the buffers of the data center switch, resulting in many packet drops. This behavior can be caused by the Partition/Aggregate scheme as the request from an application creates several responses from workers usually at the same time. If there is a packet loss, TCP thinks there is congestion, it lowers the throughput and can also experience a timeout (determined by the TCP minimum retransmission timeout). The retransmission timeout is usually around 200 ms. This value works well for the communication over Internet but is very large for Data center networking where round trip time between servers in data center rack is less than 1 ms.

One of the solutions for above-described problems can be Data Center extension for TCP protocol (DCTCP) [flo93]. The goal of DCTCP is to achieve high burst tolerance, low latency, and high throughput, with commodity shallow buffered switches. DCTCP achieves these goals primarily by reacting to congestion in proportion to the extent of congestion. DCTCP requires active queue management scheme at switches. If the buffer occupancy exceeds a fixed small threshold at the switch, the switch marks the packet as Congestion Experienced. The DCTCP source reacts by reducing the window by a factor that depends on the fraction of marked packets: the larger the fraction, the bigger the decrease factor.

4.4.2. TCP policy in RINA

RINA has a benefit that different communication behaviors can be implemented using own policies. It is, thus, possible to implement a policy that behaves as a whole new transport protocol or has behavior similar to TCP or UDP. A somewhat analogous to TCP protocol in RINA is RED TCP[28] policy plugin implemented by Leonardo Bergesio (I2CAT). The RED policy plugin implements the basic TCP behavior such as slow-start for finding the maximum link speed and congestion avoidance phase by cutting the window size (credit size), etc. The RED TCP plugin implements policies for Relaying/Multiplexing Task (RMT) and Data Transfer Control Protocol (DTCP). Both policies are implemented as a kernel plugin.

RMT RED policy

The RED policy for RMT reuses RED (Random Early Detection) mechanism [flo93]. When a new packet arrives, the average queue length is calculated according to the following formula (for details see the source code[29]):

```
avg = (1-W) × avg + W × current_queue_len
```

If `(avg > th_max)` , the PDU is marked with ECN flag or dropped. If `(avg < th_min)` the PDU is passed and if `(th_min < avg < th_max)` the probability of marking is calculated according the following formula:

```
Pb = max_P * (avg - th_min)/(th_max-th_min)
```

The recomended value for `max_P` is between 0.01 - 0.02. It is possible to set different parameters using configuration file. The following output is an example of RED configuration.

```
"rmtConfiguration": {
   "policySet": {
      "name": "red-ps",
         "parameters": [ {
               "name"  : "qmax_p",
```

[28] https://github.com/IRATI/stack/tree/master/plugins/red
[29] http://lxr.free-electrons.com/source/include/net/red.h

```
                "value" : "600"
              }, {
                "name"  : "qth_min_p",
                "value" : "32"
              }, {
                "name"  : "qth_max_p",
                "value" : "128"
              }, {
                "name"  : "Wlog_p",
                "value" : "9"
              }, {
                "name"  : "Plog_p",
                "value" : "8"
              } ],
          "version": "1"
    }
}
```

The parameteres are the following:

```
qth_min   - should be < qth_max/2
qth_max   - should be at least 2*qth_min and less limit
Wlog      - bits (<32) log(1/W).
```

Plog is related to max_P by formula:

```
max_P = (qth_max-qth_min)/2^Plog
```

DTCP RED policy

The DTCP policy acts as a simple TCP implementation. It begins with slow-start where the credit is increased by one with each received PDU. The receiver maintains the credit and informs the sender that its credit is increased. If the receiver receives PDU with ECN mark set, the credit is set to half and the sender is informed that its credit was decreased. We can see, that the RED policy relies on the information from RMT and the congestion is detected only if RMT marks PDU with ECN flag. Note that sender does not detect if there is a congestion using timeout as in TCP - the flow control is currently entirely at receiver in the IRATI RINA implementation.

4.4.3. DCTCP policy in RINA

The purpose of this experiment is to accommodate the similar behavior as DCTCP for TCP/IP in RINA architecture and to show how the congestion policy can be easily changed and adapted in RINA. We created policies for RMT and DTCP that mimic the behavior of Data Center TCP and show the similar results as the DCTCP for TCP/IP.

The policy for RMT behaves similarly as a switch in Data Center network. If RMT queue exceeds a small fixed threshold, the RMT starts to mark PDU with explicit congestion flag. Policy for DTCP behaves similarly as TCP with DCTCP extension with some RINA specific differences.

The main difference is that in DCTCP, the sender maintains an estimate of the fraction of packets that are marked. The sender can react to a congestion and lower the speed. In other words, sender controls the flow speed and appropriately acts if it sees a congestion. RINA, however, maintains the flow control on the receiver and the receiver controls the speed of the sender by lowering or increasing sender's credit. Thus, the computation for DCTCP-like policy in RINA must be implemented on the receiver.

Both policies are implemented as a kernel plugin. For testing purposes, we recommend using RINA's buildroot[30] to compile an image with DCTCP support. The necessary changes that need to be done in order the DCTCP plugin works:

In buildroot/package/rinad-plugins/rinad-plugins.mk file it is necessary to add the plugin support:

```
RINAD_PLUGINS_MODULE_SUBDIRS = plugins/dctcp
RINAD_PLUGINS_DIRNAMES = dctcp
```

If the stack is recompiled with update-irati script, the policies will be available for RMT and DTCP.

RMT DCTCP policy

The DCTCP policy for RMT is simple. It is possible to configure a queue occupancy and a threshold. If queue occupancy exceeds the threshold, the

[30] https://github.com/IRATI/buildroot

RMT starts to mark PDUs with explicit congestion flag (ECN). It is possible to set the parameters using the following configuration for a DIF.

```
"rmtConfiguration": {
    "policySet": {
       "name": "dctcp-ps",
          "parameters": [ {
              "name": "q_threshold",
              "value": "10"
           }, {
              "name": "q_max",
              "value": "100"
           } ],
       "version": "1"
    }
}
```

The configuration sets the queue threshold to 10 and length of the queue to 100. Thus, if the length of the queue exceeds 10 PDUs, RMT will mark subsequent PDUs with ECN flag. The following configuration can be used for RINA demonstrator, assuming that n1 is the name of the DIF.

```
policy n1 rmt dctcp-ps q_threshold=10 q_max=100
```

DTCP DCTCP policy

The DCTCP policy for DTCP maintains an estimate of the fraction of packets that are marked with ECN flag. This value is updated with every PDU according to the following formula:

```
α ← (1−g) × α + g × F
```

where F is the fraction of marked packets in the flow and g is Estimation Gain. The estimation gain g must be chosen small enough to ensure the exponential moving average in the previous formula “spans” at least one congestion event. If the receiver receives a PDU with ECN flag sets, it reacts by reducing the sender’s credit size. In classical TCP and RED policy for RINA, the receiver cuts the credit size by a factor of 2. However, the credit size in DCTCP policy is reduced according to the following formula:

```
new_credit ← new_credit × (1−α/2)
```

Thus, when α is near 0 (low congestion), the window is only slightly reduced. If the fraction of marked packets is higher (high congestion), the windows is reduced more.

The following configuration for a DIF can be used to test DCTCP policy for DTCP.

```
"dtcpConfiguration": {
    "dtcpPolicySet": {
        "name": "dctcp-ps",
            "parameters": [ {
                "name": "shift_g",
                "value": "4"
            } ],
    "version": "1"
}
```

The following configuration can be used for RINA demonstrator, assuming that n1 is the name of the DIF.

```
policy n1 efcp.*.dtcp dctcp-ps shift_g=4
```

It is possible to alter the g parameter using the shift_g variable. According the DCTCP paper [aki10], the g value should be small enough and all experiments in the paper use g = 0.0625 (1/16). Thus, the shift_g = 4 is (2^4 = 16).

4.4.4. Results

The DCTCP and RED policies were tested in the following testbed. We used RINA bootstrap mechanism to build own image extended with DCTCP and RED plugins as described above. RINA demonstrator was used to building a simple topology as depicted in Figure 62 and for the basic configuration.

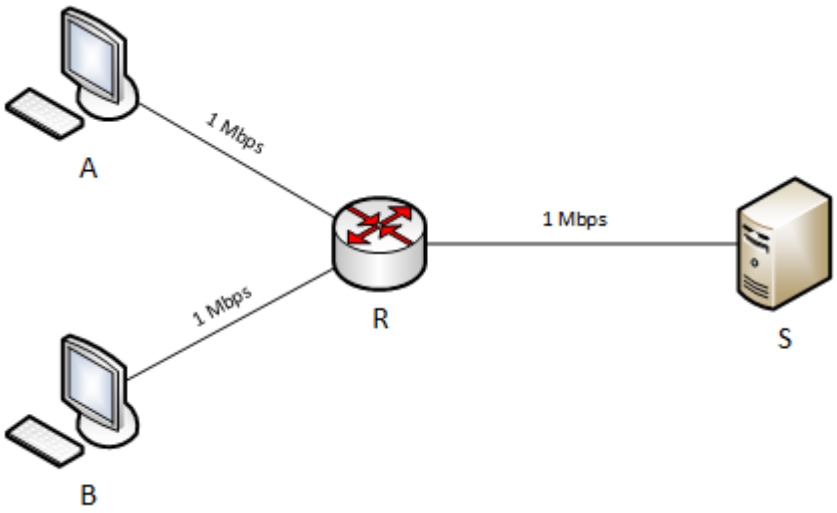


Figure 62. Topology used for experiments.

To validate the policies' behaviors, we need to limit the speeds between the nodes. It should be possible to limit the speed of the links using `demonstrator`. We were able to set links speeds between nodes and `rina-tgen` and `rina-echo` applications showed the right (limited) bandwidth with the default DIF configuration. However, we faced a few difficulties when we tried to adapt this approach for DCTCP policy testing while also keep gathering data from the DCTCP policy. The main issue was that the "full utilization of the link" feedback signalizing was not getting propagated from N-1 DIF (shim-eth DIF) to N DIF. In other words, the IPCPs in the N DIF did not see links as fully utilized, therefore they skipped the queuing policies of the DCTCP policy. We overcame the problem by altering the behavior of the eth-shim DIF and limiting the bandwidth there. The theoretical speed for our testbed is 1000 kbit/s. However, it is a bit less in practice. If we use multiple streams, the throughput is around 600 - 800 kbit/s. As our goal was not to test the maximum achievable speeds, but to validate the DCTCP policy behavior, we deem these modifications as not very important.

The router R has the RMT DCTCP plugin loaded with the following parameters: q_length=100, q_threshold=10. The same applies for RED plugin which has been loaded with the following parameters: qmax_p=600, qth_min_p= 32, qth_max_p=128, Wlog_p=9, and Plog_p=8.

In the case of DCTCP, the RMT policy behavior is the following: If the length of the queue for the link between router R and server S exceeds the threshold, all subsequent PDUs will be marked with ECN flag set. At the

beginning, senders A and B send as many packets as the initial value of their credit. If server S does not see any congestion, the credit is for A and B is increased - this is a variant of TCP slow-start mechanism. Whenever the queue size on R exceeds threshold, RMT process on R starts to mark PDUs with ECN flag. This behavior is depicted in Figure 63.

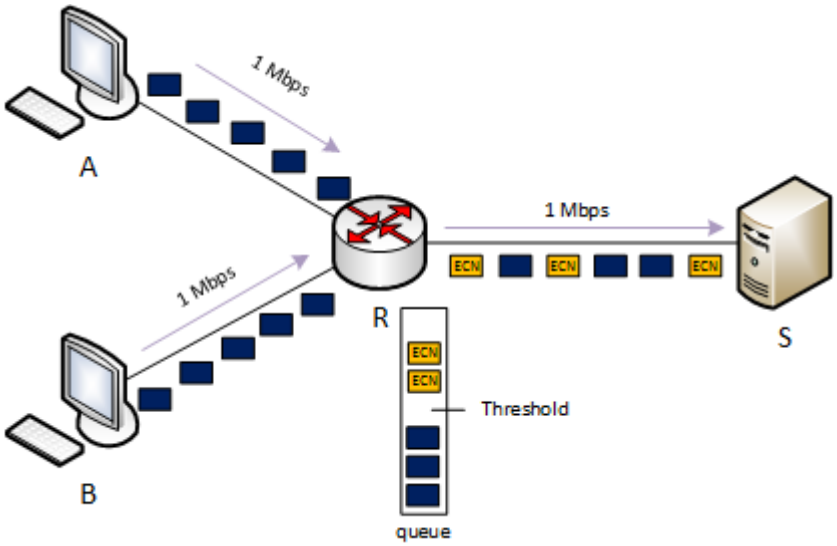


Figure 63. Basic DCTCP plugin behavior.

If the PDU marked with ECN flag arrives on device S, the device recomputes credit for the sender according to the formula described in the previous section and sends back the new credit value. Using this approach, the senders (A or B) lower their speed, thus eliminate the congestion.

In the case of RED, the RMT policy behavior is slightly different: If the length of the queue for the link between router R and server S exceeds the qth_min_p, the PDU can be marked with some probability. If the queue exceeds the qth_max_p, all subsequent PDUs will be marked. If the PDU marked with ECN flag arrives on device S, the device recomputes credit for the sender by cutting the credit size to half. The server S sends the new credit value back.

The following QoS Cube was used for testing DCTCP policy:

```
"qosCubes": [ {
        "efcpPolicies": {
            "dtcpConfiguration": {
                "dtcpPolicySet": {
                    "name": "dctcp-ps",
                    "parameters": [ {
                            "name": "shift_g",
```

```
                        "value": "4"
                    } ],
                    "version": "1"
                },
                "flowControl": true,
                "flowControlConfig": {
                    "rateBased": false,
                    "windowBased": true,
                    "windowBasedConfig": {
                        "initialCredit": 10,
                        "maxClosedWindowQueueLength": 10
                    }
                },
                "rtxControl": true,
                "rtxControlConfig": {
                    "dataRxmsNmax": 100,
                    "initialRtxTime": 1000
                }
            },
            "dtcpPresent": true,
            "dtpPolicySet": {
                "name": "default",
                "version": "0"
            },
            "initialATimer": 0
        },
        "id": 1,
        "maxAllowableGap": 0,
        "name": "reliablewithflowcontrol",
        "orderedDelivery": true,
        "partialDelivery": false
    }
]
```

Notice that this QoS cube uses DCTCP plugin for DTCP with shift_g parameter set to 4. Using these parameters, the length of the RMT queue should oscillate around 10. If RED policy is used, it is necessary to change the DTCP Policy Set with the following config:

```
"dtcpPolicySet": {
   "name": "red-ps",
   "version": "1"
}
```

The following Figure 64 and Figure 65 show the size of queues on R during the experiment with RED and DCTCP policies.

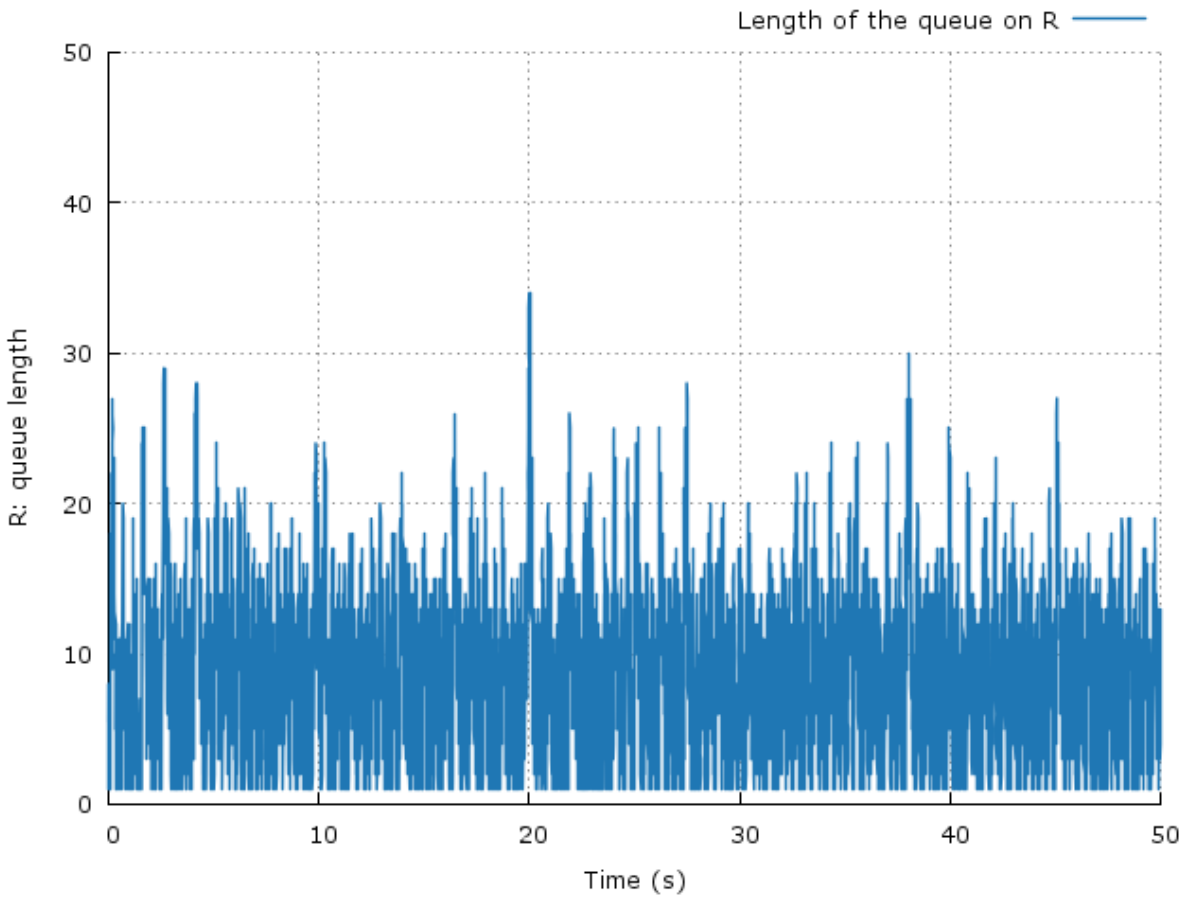


Figure 64. Queue length on R during the experiment with DCTCP policy.

We can see that the queue occupancy oscillates around 10 PDUs. Figure 65 shows the same queue occupancy with RED policy. Using the RED configuration described above, we can see, that the queue occupancy is much higher and the average queue length oscillates between `qth_min_p` and `qth_max_p` with some burst.

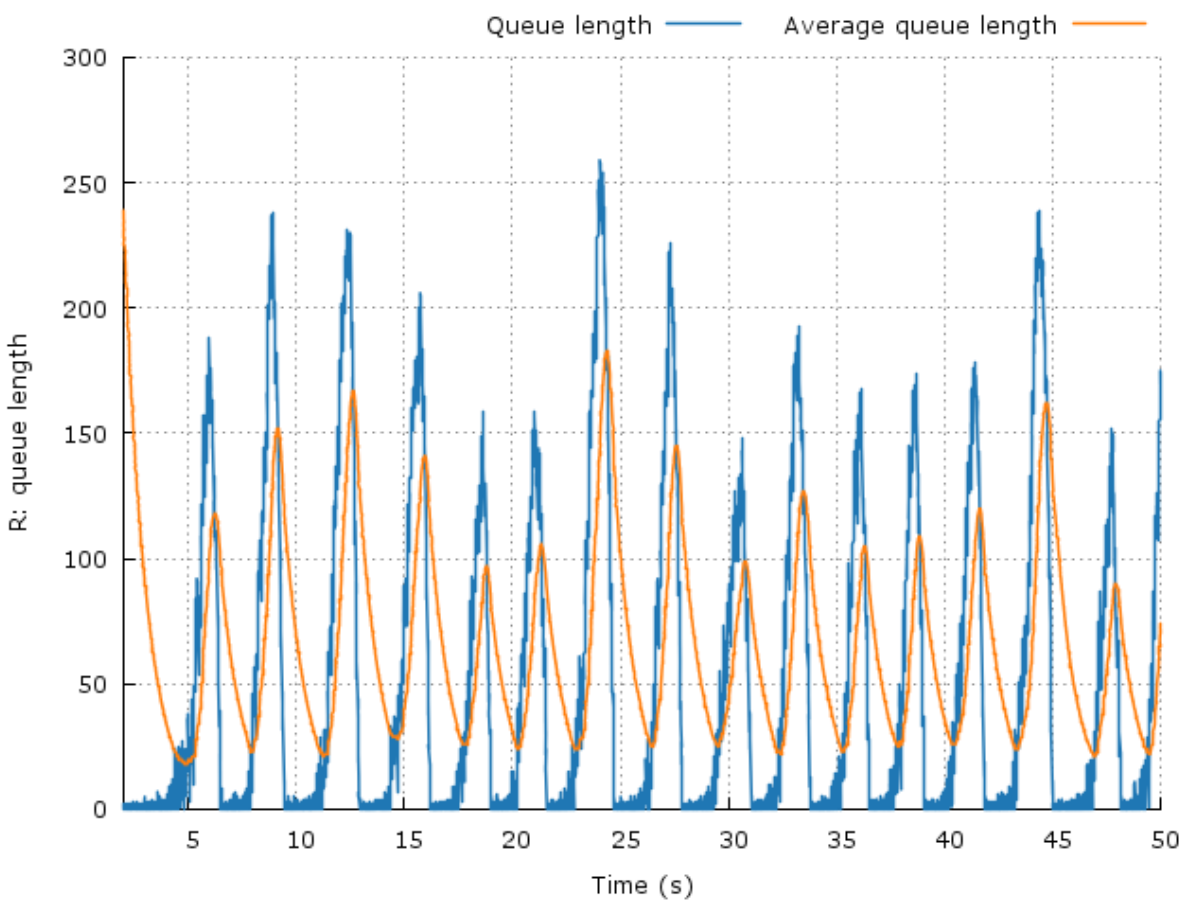


Figure 65. Queue length on R during the experiment with RED policy.

Figure 66 depicts CDF function of the queue size and compares it with the RED policy that simulates the classical TCP behavior in RINA. The figure confirms that DCTCP policy manages small queue length, the queue length for RED policy is larger. This has several consequences especially for incast scenarios where DCTCP can better handle bursts and Partition/Aggregate style communication due to smaller queue size utilization.

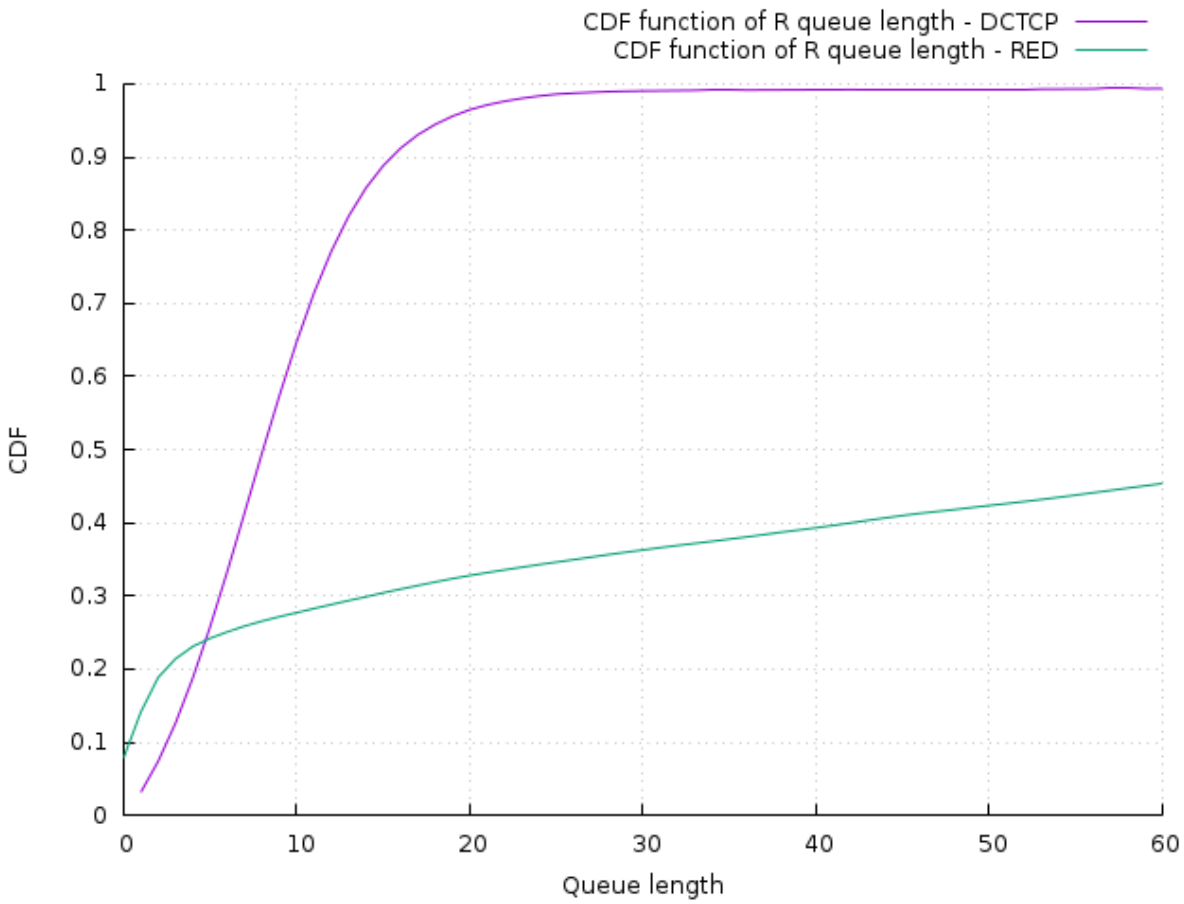


Figure 66. CDF function length on R during the experiment with RED policy.

The differencies between RED and DCTCP policies are also visible in Figure 67 and Figure 68 that show measured throughput for hosts A and B. If we use small measurement interval (0.5s in this case), we can observe how policy affects throughputs for hosts A and B.

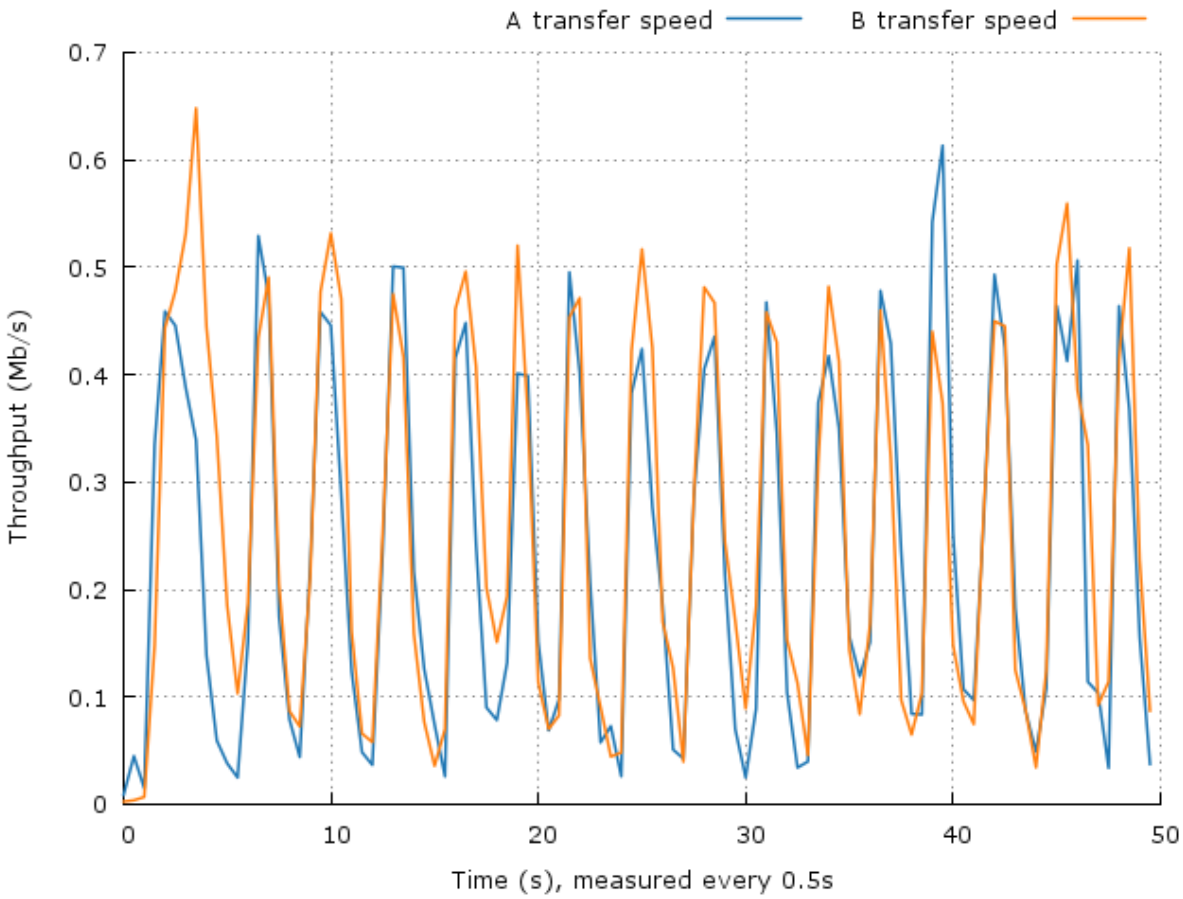


Figure 67. Measured throughput for hosts A and B using RED policy.

If RED policy is used (Figure 67), the throughput varies between 0.1 Mbps and 0.5 Mbps. The reason for the oscillation is that RED policy halves sender's window if congestion is reached.

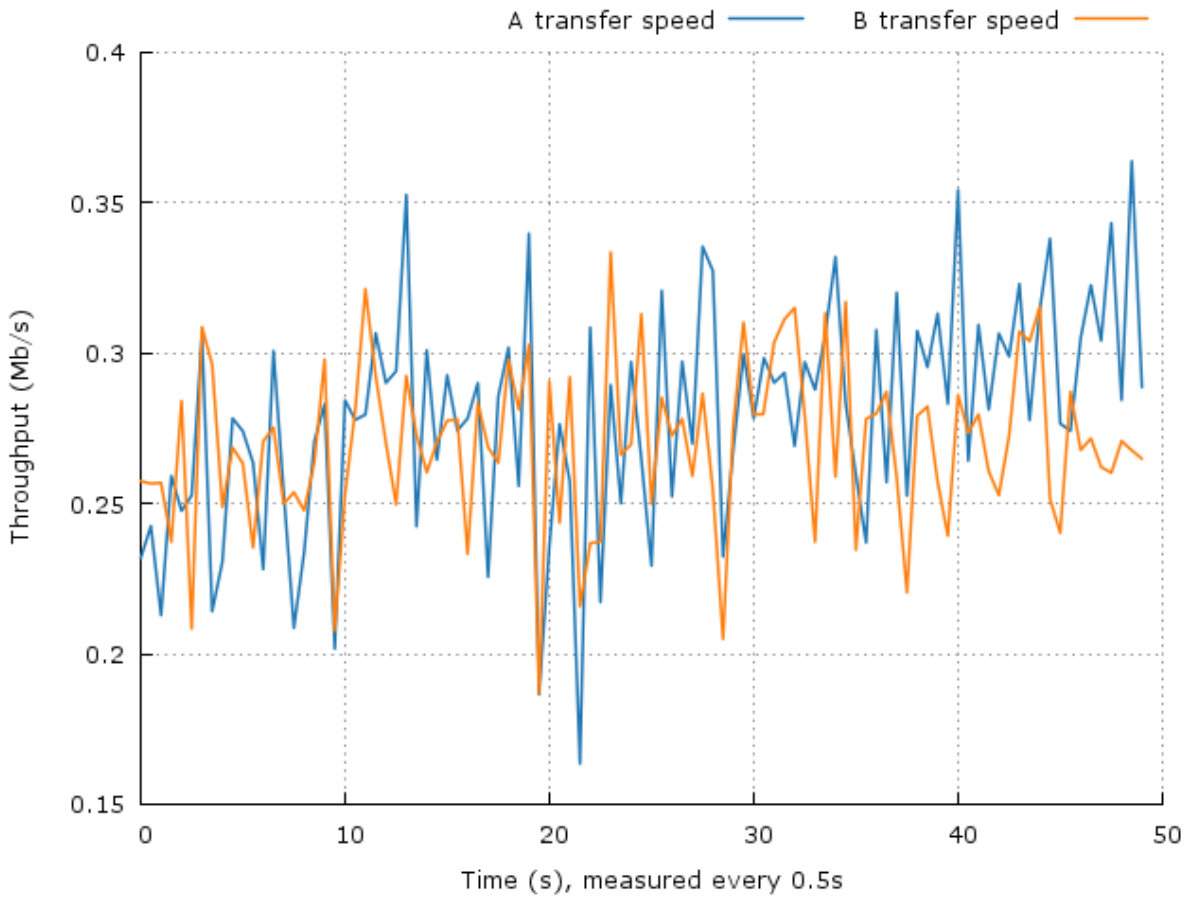


Figure 68. Measured throughput for hosts A and B using DCTCP policy.

The DCTCP policy tries to adapt the sender's speed to maintain small R's queue occupancy, thus the speed is more stable during the experiment (see Figure 68).

4.5. Resiliency

We implemented the LFA routing policy[31] into the IRATI prototype. IPCP components related to management are implemented in userspace, whereas components related to data transfer are implemented in the kernel. As far as routing is concerned, the RIB and PDU Forwarding Table Generator are located in userspace, and the PDU Forwarding Table is located in kernel space.

In the first phase of the implementation, we react mainly to the failure of an interface. When the interface is brought down, for instance because the UTP cable has been disconnected, the IRATI stack is notified. The shim IPC Process for IEEE 802.1Q, which wraps the IEEE 802.1Q standard (VLANs)

[31] https://wiki.ict-pristine.eu/wp4/d43/d43-resilient-routing

with the IPC API, listens to this particular event. It then notifies the IPCPs using its services that the port-id is down. A normal IPCP that uses the LFA routing policy can then react to this event by switching to a Loop Free Alternate (LFA). If there is no LFA available, it can mark its own port-ids as down and propagate the event upwards. Higher level DIFs may then react to this event and so on, effectively leveraging the recursive nature of RINA.

We performed a test on the Abilene network graph. Each node is physically connected to every other node by a 1 Gbps link. We setup VLANs on these links so that we can use the shim DIF for IEEE 802.1Q. On top of these shim DIFs we overlay a single normal DIF. Solving resiliency in lower level DIFs may make more sense than solving it in higher level DIFs, since the time for detection may be higher in higher level DIFs.

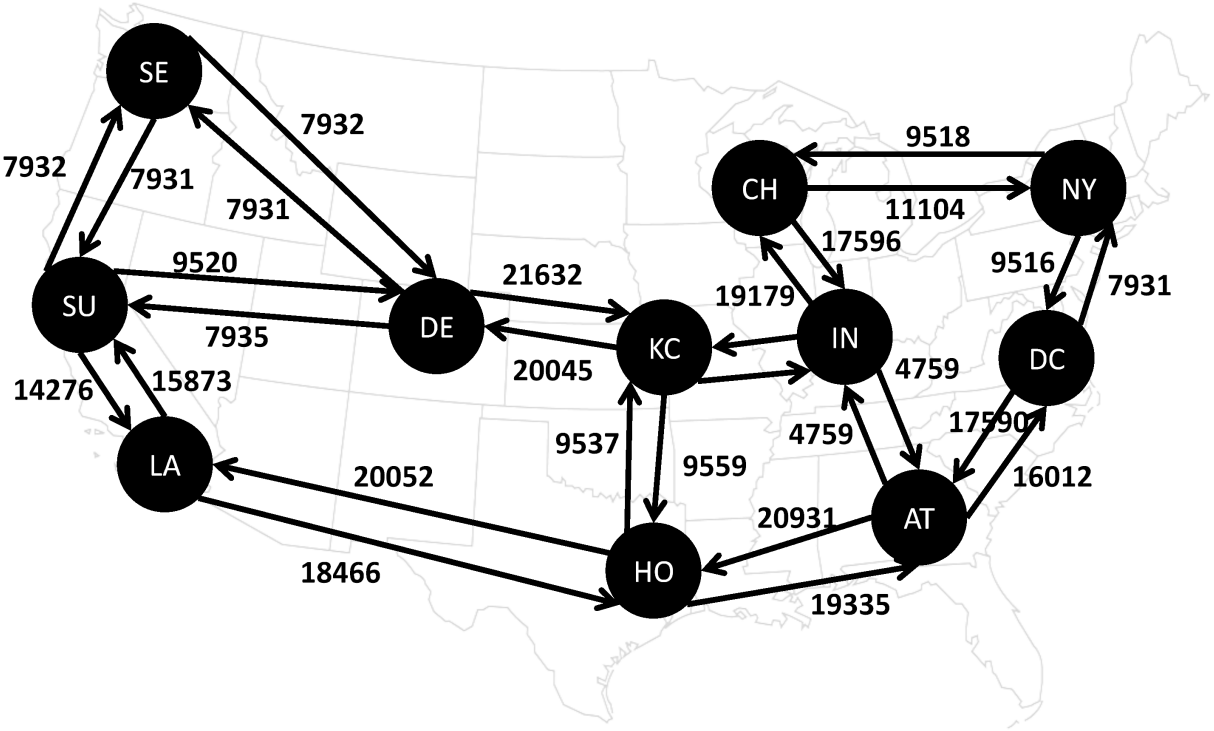


Figure 69. Results in normal operating conditions

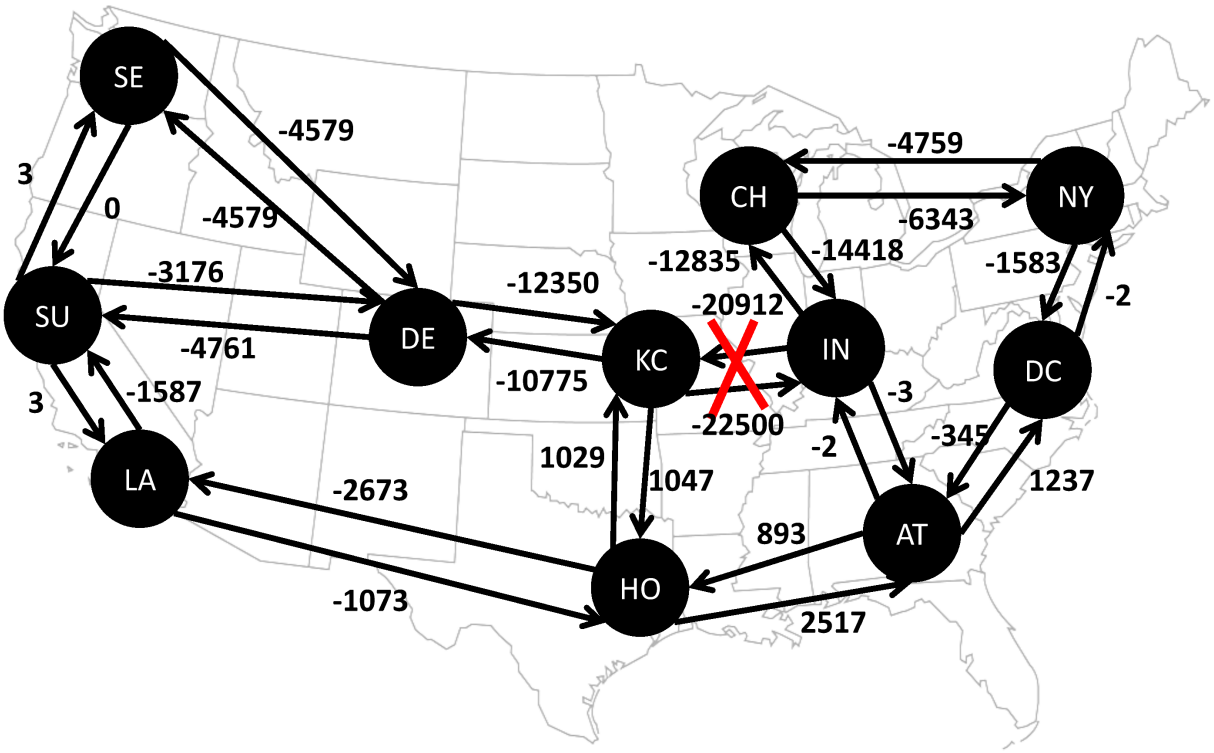


Figure 70. Results with link failure and link state routing

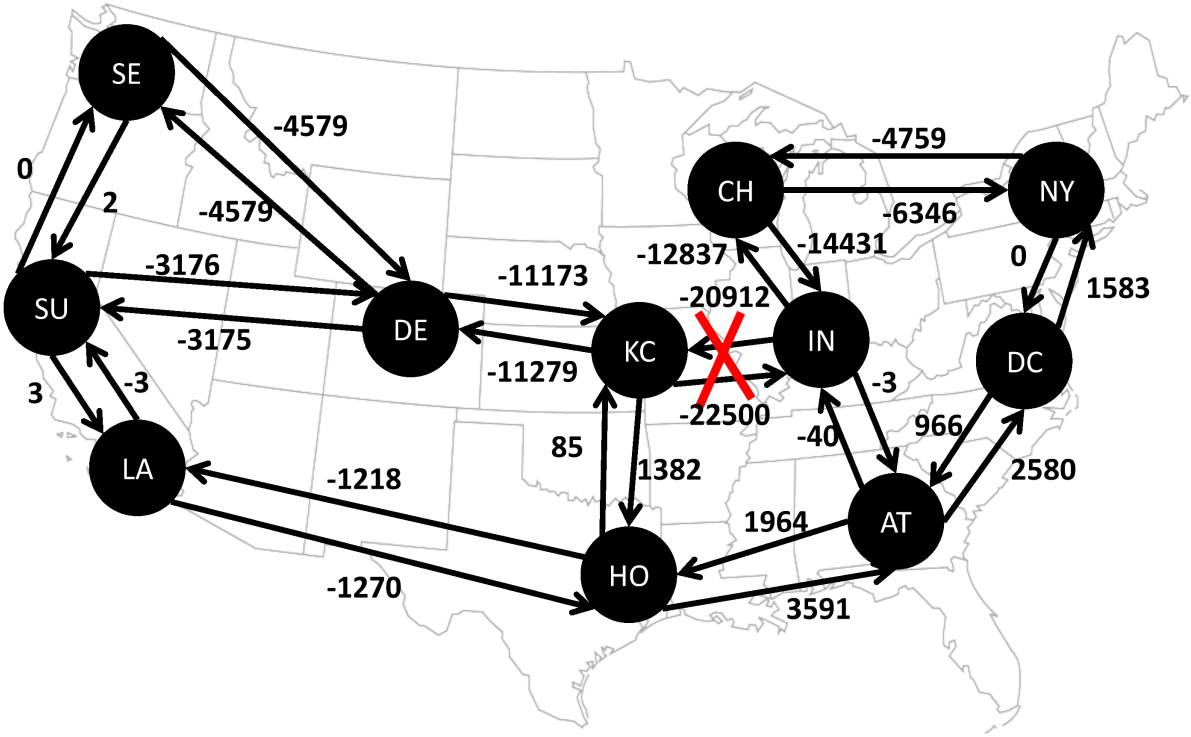


Figure 71. Results with link failure and loop free alternates policy

Table 4. Results on the Abilene network

	Normal conditions	95% CL	Failure without LFA	95% CL	Failure with LFA	95% CL
AT → HO	20931	66	21825	136	22896	133
HO → AT	19335	37	21853	65	22926	72
AT → IN	4760	32	4758	79	4719	62
IN → AT	4759	13	4757	23	4756	18
AT → DC	16012	47	17250	92	18593	72
DC → AT	17590	21	17245	66	18557	68
CH → IN	17596	27	4761	9	4759	17
IN → CH	19179	46	4760	28	4748	55
CH → NY	11104	16	4761	4	4758	3
NY → CH	9518	15	4759	15	4759	13
DC → NY	7931	13	7930	74	9515	20
NY → DC	9516	15	7933	25	9516	19
DE → SE	7931	13	3172	3	3173	3
SE → DE	7933	11	3173	9	3173	11
DE → SU	7935	17	4759	19	4758	17
SU → DE	9520	25	4759	65	6345	74
DE → KC	21633	31	9283	51	10459	92
KC → DE	20045	25	9270	105	8765	111
HO → KC	9537	46	10566	89	9622	126
KC → HO	9559	43	10607	87	10941	255
HO → LA	20053	44	17380	72	18834	53
LA → HO	18466	34	17393	54	17196	54
IN → KC	20913	36	0	0	0	0
KC → IN	22501	44	0	0	0	0
LA → SU	15873	37	14285	105	15870	57
SU → LA	14276	19	14279	42	14280	27
SE → SU	7931	11	7931	26	7933	22
SU → SE	7932	14	7930	72	7933	49
Total	**380273**	**66**	**257379**	**136**	**269784**	**255**

After the complete network has been bootstrapped, e.g. all IPCPs instantiated on all nodes and enrolled in their respective DIFs, we launched the rina-tgen application. Rina-tgen is a traffic generator for RINA. On every node, we run an instance of rina-tgen in server mode. Next, on every

node we launch a client instance to every server instance in the network. The clients will send at 1 Mbps, at packet size 125 bytes, so 1000 packets per second. Traffic is flowing between every pair of nodes. This situation is depicted in Figure 69. The throughput going out of every interface is shown in kbps.

After some time, we brought down the virtual interface on nodes KC and IN, e.g. a link cut. When using the simple link state routing policy, lots of flows will be disrupted for several seconds. We calculated the difference between the throughput coming out of the interfaces under normal running conditions and when the link is cut. The result of this is shown in Figure 70. As can be seen, most vertices have a negative weight. The vertices that hold a positive weight can be explained by the stability of the prototype. The prototype currently gives inconsistent results between different runs. Further research to find the source of this inconsistency is being performed. Nevertheless, one can already observe that on the whole most vertices now have a negative weight because a lot of traffic is lost due to the cutting of the link.

We repeated this test, but this time the IRATI stack was running the LFA policy. When we cut the link between KC and IN, the LFA policy on these nodes successfully switched to the Loop-Free Alternate, if available. Again we calculated the difference between the throughput coming out of the interface under normal conditions and when the link is cut. This result is displayed in Figure 71. A lot of vertices still contain negative weights, but the damage was mitigated, since a LFA was used when available.

The throughput (in kbps) together with their respective 95% confidence level in all the different cases are shown in Table 1 for completeness sake. The totals are also presented. Under normal operating conditions the total throughput is 380273 +- 66. When a link is cut and a regular link state routing policy is used, the total is only 257379 +- 136. When a link is cut and the LFA policy is used, this is 269784 +- 255. This means that at a rate of 1 Mbps per flow, 10 flows are recovered. This may seem a rather low number. Note however that the Abilene graph is not the best one for Loop-Free Alternates. The suitability of the LFA policy depends heavily on the network graph.

We then looked at the recovery time of the LFA policy. We first brought down the physical interface on nodes KC and IN, which took approximately

400 ms to recover from the failure when using the LFA policy. We also performed another experiment where we only brought down the virtual interface that the shim DIF for 802.1Q was using. In this case it took 1.7 ms to recover. After some investigation, we found the reason for this huge difference in recovery time. When a physical interface is brought down, first the driver is contacted to actually bring the interface down, meanwhile dropping any new packets that are trying to use the Network Interface Card (NIC). On our test machine, it takes the driver 400 ms to perform this, which is exactly the difference between the two recovery times. Using a different NIC may decrease this time. So in conclusion, the failure detection time on our test setup is 400 ms, whereas the time to recover is 1,7 ms.

We tried to repeat these tests at a later time, with more nodes. However, when we tried to setup the flows between all the traffic generator instances, some scalability problems arose. We are currently looking into the source of these problems. An issue description[32] has been created for the IRATI prototype.

4.6. Key management demonstration scenario

4.6.1. Context

The Key Management system consists of:

Key Managers (KMs)

A Key Manager is responsible for managing a Key Store, containing private key material. A RINA system may contain several Key Managers under different administrative control. Each Key Manager communicates with (and controls) a set of Key Management Agents via a private DIF within the Management DAP. A Key Manager connects to the RINA system via the Central Manager. The Manager would interact with the Key Manager via CDAP operations on its RIB.

Key Management Agents (KMAs)

A Key Management Agent is a process on a RINA node that is associated with both a Management Agent (on that node) and a Key Manager. It

[32] https://github.com/IRATI/stack/issues/1027

responds to requests from the Management Agent (some of which may in turn have come from a local IPCP). The Management Agent would interact with the Key Management Agent via CDAP operations on its RIB. The KMA and the KM are visible as DAPs (application processes) in the Management DAF. Direct KMA-KM communication occurs over one of the DIFs supporting the Management DAF: KMA would have to allocate a flow to KM - or vice versa, then they mutually authenticate and setup SDU protection as adequate, and can exchange PDUs.

Key Stores (KS)

A Key Store is a secure repository of key material under some particular administrative control. It is managed by a Key Manager and may be distributed across KMAs controlled by the Key Manager. It may contain references to Key Containers in the RIB.

Key Containers (KCs)

A Key Container is an object in the Key Management RIB. It can contain 0, 1 or 2 keys. Each key has a state; may have public key material; and may have references into the Key Store.

4.6.2. Objective

Show an IPCP authenticating with another, using key material distributed from a central key store in a way that keeps the private key material secret from the IPCP and the management agents.

The scenario starts from the assumptions:

- The management DIF is alive.
- The keys are [previously generated | or generated on demand] (potentially outside the RINA environment altogether) but the Central Manager's RIB does not know about them yet.
- The Key Manager and Key Management Agent have established an authenticated connection.

Steps

Scene 1

In which the reference to the private key material is made available.

The Central Manager decides that a new RSA key pair is needed (either because it has been asked for one by a Management Agent, or in advance to hide latency). In a multi-tenant environment, there may be a choice of key managers to access, the choice of which one will depend on the intended use of this key pair.

From the manager perspective, keys are kept in a special portion of its own RIB. It creates objects called 'key containers' to hold the relevant information [33]. Note, however, that private key material is not stored in the RIB, but held in the key manager; the key container holds a unique ID that can be used to reference the key material held in the key manager.

The Central Manager creates the key containers for new keys to be used by IPCP A and B. Consequently, the Key Manager RIB becomes populated with associated material, in particular:

- key state
- public key matter
- the ID of the private key held by the Key Manager

We are assuming that the relevant Management Agents have access to the appropriate portions of the RIB.

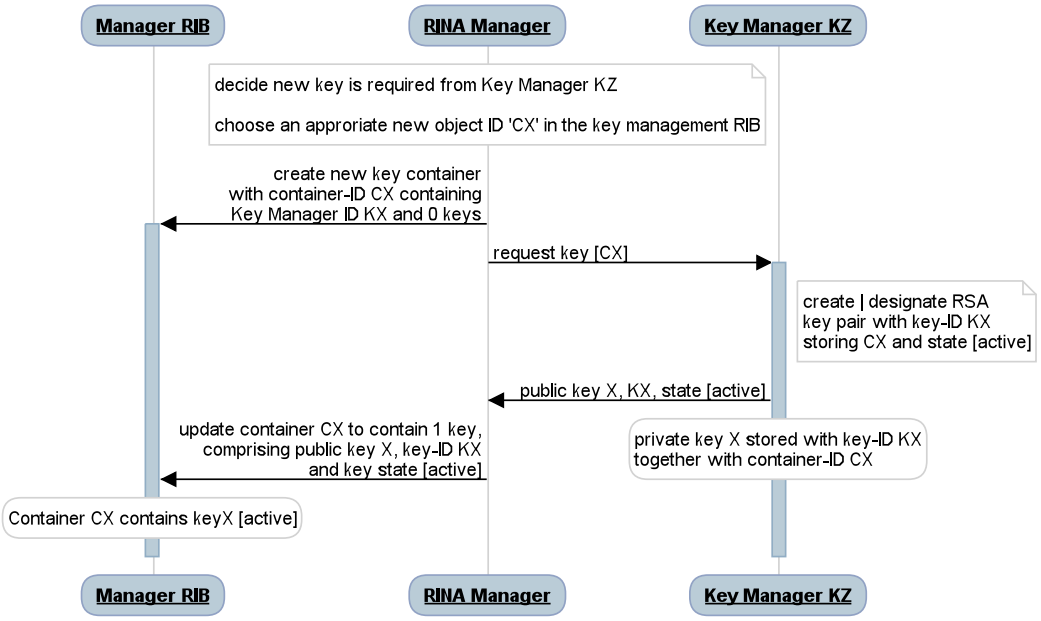


Figure 72. Creation and population of key containers

[33] Key containers can contain 0, 1 or 2 keys.

This process is illustrated in Figure 72 above.

Scene 2

In which the private key material is distributed to the Key Management Agents

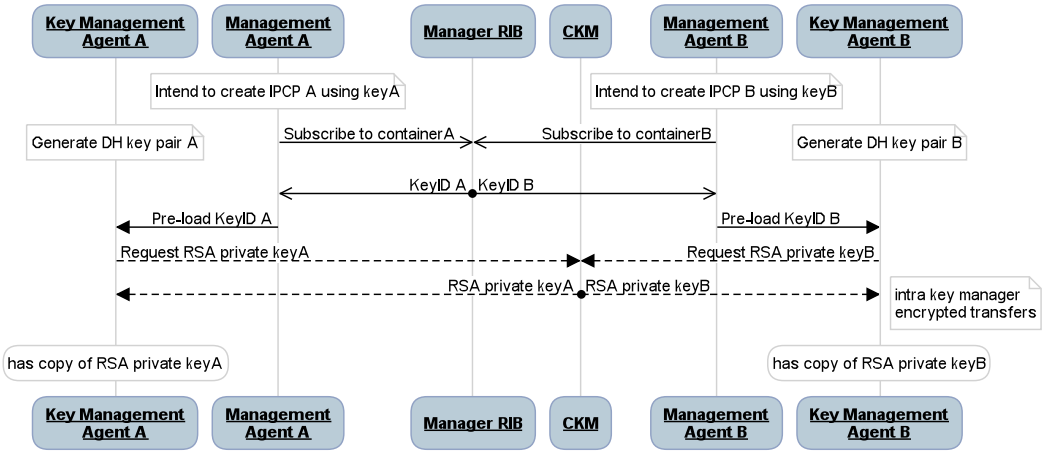


Figure 73. Distribution of private key material

Each MA, intending to use the corresponding key, subscribes to the relevant key container. Finding that this contains a key store handle, it passes this handle to the appropriate local KMA, which then communicates with its Key Manager to obtain a copy of the private key.

Figure 73 also shows both KMAs generating a Diffie-Hellman key pair. This could be done on demand, but as this is a computationally expensive process, it would be advisable for the KMAs to do this as a background task to reduce the latency of the authentication process. It will be a matter of the local security policy to decide how many such keys to pre-generate.

Scene 3

In which the private key material is used to establish an authenticated connection.

Management Agent A creates IPCP A, incorporating the identity of its authentication key, which it obtains from the Central Manager; Management Agent B does likewise.

- IPCP A wishes to authenticate with IPCP B (In Management Agent B's domain).
- Then, ICPCP A starts authentication (using AuthNAsymmetric Key protocol).

- After this, IPCP A and IPCP B have an authenticated connection.

Figure 74 shows the first part of this sequence, in which IPCP A interacts with the local key management agent and the RIB to obtain all the information it needs to start the connection with IPCP B.

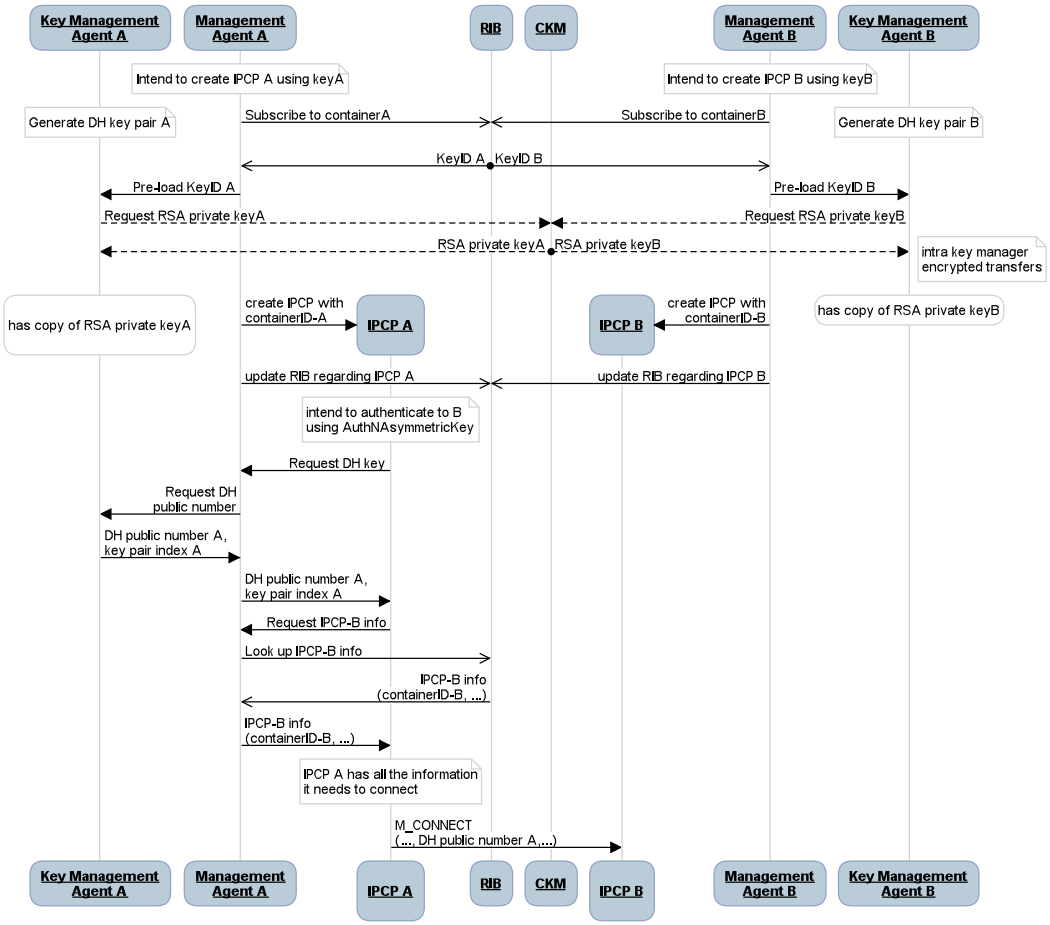


Figure 74. First part of the AuthNAsymmetricKey authentication

Figure 75 shows the second part of the authentication sequence, in which IPCP B uses information in the RIB and interactions with its local key management agent to respond to the connection request. Note that the fetching of the public keys occurs ahead of when they are needed; this hides latency, but is not essential, in which case this would occur later in the sequence.

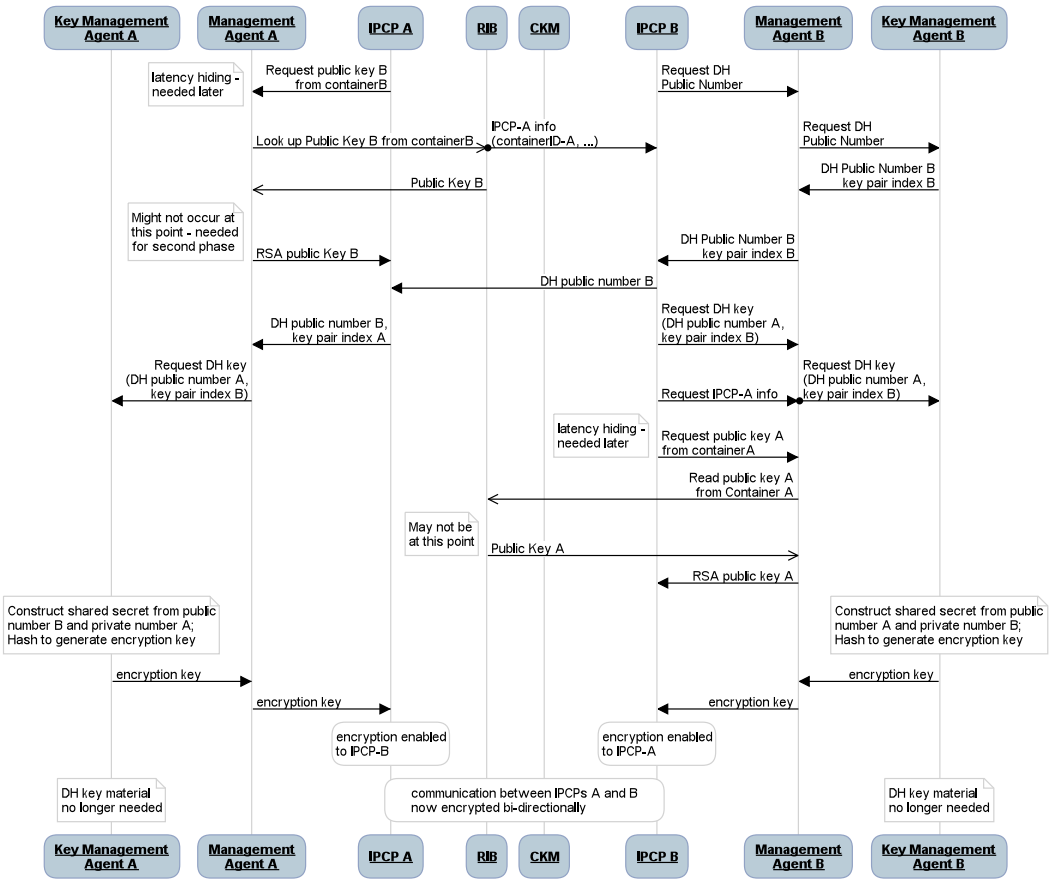


Figure 75. Second part of the AuthNAsymmetricKey authentication

The final part of the sequence, shown in Figure 76, does not involve the Central Key Manager or the RIB. For diagrammatic clarity, the distinction between Management Agents and Key Management Agents is dropped.

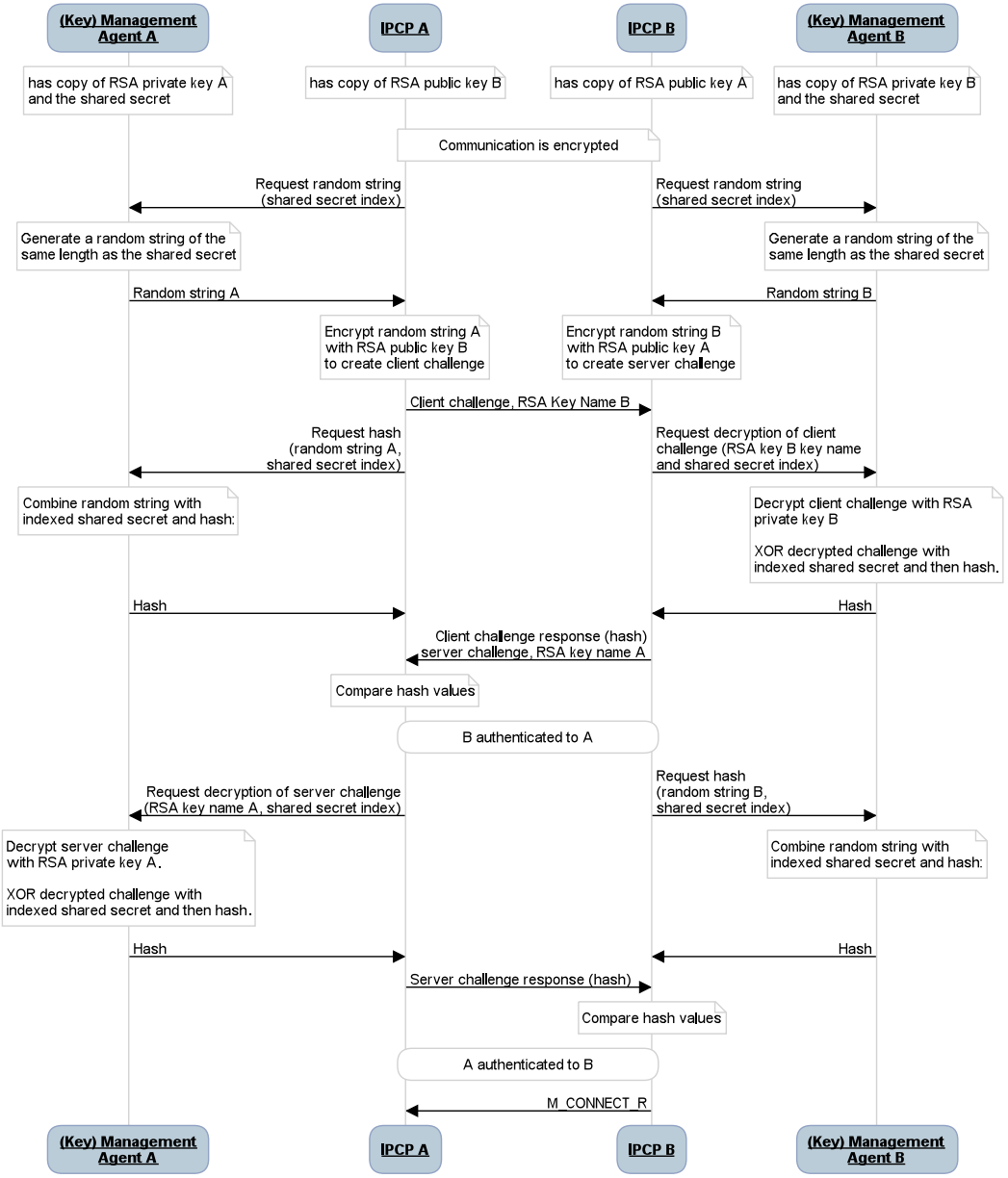


Figure 76. Third part of the AuthNAsymmetricKey authentication

5. Joint Experiments

In this section we report the joint experiments carried out during the second phase of the project. These experiments have been carried out using two or more PRISTINE policies at the same time. The goal here is to demonstrate the modular nature of the RINA architecture while at the same time providing valuable feedback to developers.

5.1. Datacentre Networking

Multi-tenant Datacentres (DCs) are one of the core use cases of PRISTINE. This experiment demonstrates a small-scale version of a RINA DC using the RINA implementation within the demonstrator. Congestion management and multipath forwarding policies are used in conjunction in order to guarantee an efficient use of the diverse paths in the DC fabric, as well as proper sharing of performance bottlenecks. The Management System developed by WP5 is also used to configure the different DIFs that compose the DC network.

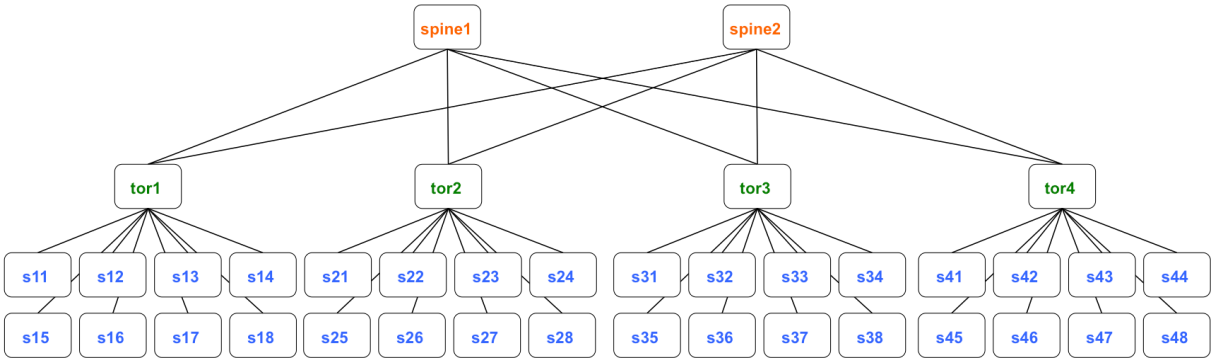


Figure 77. Physical layout of the systems in the DC experiment

Figure 77 shows the physical layout of the systems in the DC experiments. There are 4 racks of 8 servers, with a Top-of-Rack (ToR) router in each rack that connects all servers together. Then all ToRs are connected by two equal cost paths through 2 spine routers, following a leaf-spine configuration. Figure 78 shows the organization of DIFs within the DC. A DC fabric DIF connects together all ToR and spine routers. This DIF supports a number of VPN DIFs on top, which are use to create private computing domains for different tenants.

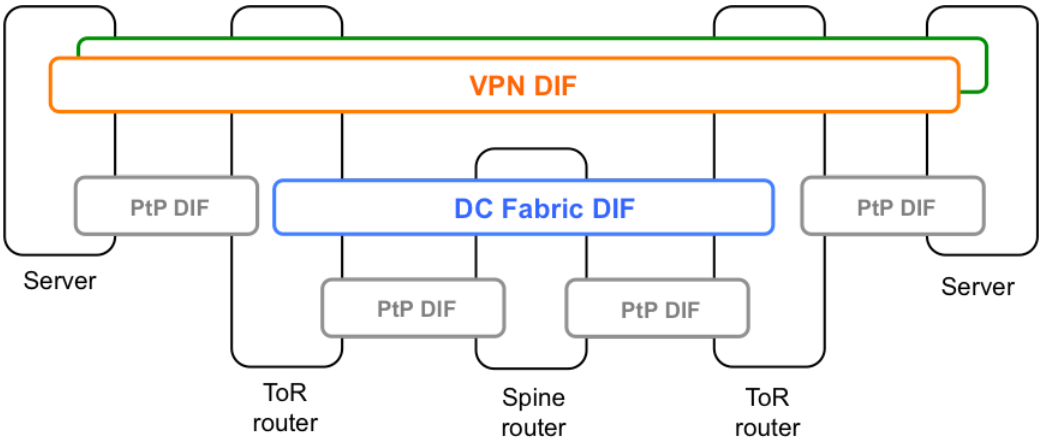


Figure 78. DIFs in the DC experiment

5.1.1. Manager experiment

The goal of this experiment is to validate the behaviour of PRISTINE's Management System implementation in a moderate-scale deployment consisting of 38 physical systems, and 5 DIFs organized in two levels (four tenant DIFs overlaid over the DC fabric DIF). Figure 79 shows the connectivity graph of the different DIFs (IPC Processes and N-1 flows linking them).

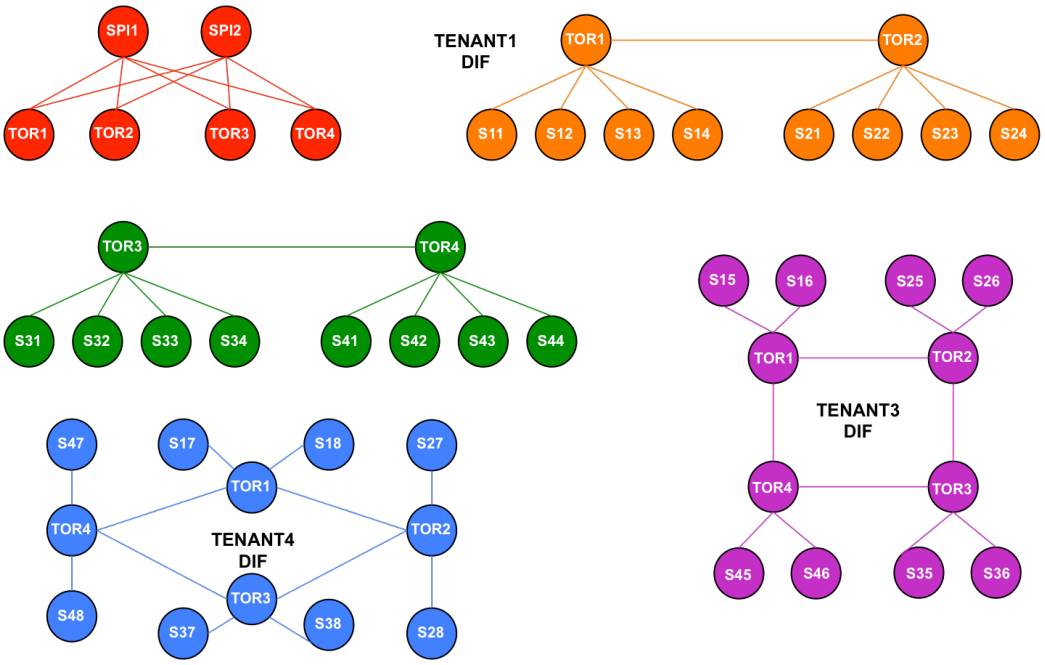


Figure 79. Connectivity graph of the different DIFs in the management experiment

To setup the experiment in the demonstrator we use the following configuration file, which just describes the physical systems (the DIFs will be created by the Manager). When generating the script to setup the scenario we specify the *-manager* option, which will create an extra machine

running the Manager and a *Management DIF* that will allow the Manager to reach the Management Agents of all the systems in the experiment.

DC Configuration Management Experiment description file

```
eth 110 100Mbps tor1 spine1
eth 120 100Mbps tor1 spine2
eth 11 25Mbps s11 tor1
eth 12 25Mbps s12 tor1
eth 13 25Mbps s13 tor1
eth 14 25Mbps s14 tor1
eth 15 25Mbps s15 tor1
eth 16 25Mbps s16 tor1
eth 17 25Mbps s17 tor1
eth 18 25Mbps s18 tor1
eth 210 100Mbps tor2 spine1
eth 220 100Mbps tor2 spine2
eth 21 25Mbps s21 tor2
eth 22 25Mbps s22 tor2
eth 23 25Mbps s23 tor2
eth 24 25Mbps s24 tor2
eth 25 25Mbps s25 tor2
eth 26 25Mbps s26 tor2
eth 27 25Mbps s27 tor2
eth 28 25Mbps s28 tor2
eth 310 100Mbps tor3 spine1
eth 320 100Mbps tor3 spine2
eth 31 25Mbps s31 tor3
eth 32 25Mbps s32 tor3
eth 33 25Mbps s33 tor3
eth 34 25Mbps s34 tor3
eth 35 25Mbps s35 tor3
eth 36 25Mbps s36 tor3
eth 37 25Mbps s37 tor3
eth 38 25Mbps s38 tor3
eth 410 100Mbps tor4 spine1
eth 420 100Mbps tor4 spine2
eth 41 25Mbps s41 tor4
eth 42 25Mbps s42 tor4
eth 43 25Mbps s43 tor4
eth 44 25Mbps s44 tor4
eth 45 25Mbps s45 tor4
eth 46 25Mbps s46 tor4
eth 47 25Mbps s47 tor4
eth 48 25Mbps s48 tor4

overlay mgr overlays
```

Once all the systems are up and running, we start the Manager and wait briefly for all the Management Agents to register, as shown in the snippet below. Management Agents register to the Manager by allocating a flow to the Manager application name and establishing a CDAP connection.

Manager shell, registration of Management Agents

```
dms @ ws-server on ws://localhost:8887: listMas dialect:dif
dms @ ws-server on ws://localhost:8887: dms: Response from <DMS_MANAGER>

List of connected MAs
.............................
3    {s38.mad,1}    Connected
4    {s27.mad,1}    Connected
29    {s14.mad,1}    Connected
1    {s22.mad,1}    Connected
2    {tor4.mad,1}    Connected
26    {s36.mad,1}    Connected
25    {s43.mad,1}    Connected
28    {s32.mad,1}    Connected
27    {tor2.mad,1}    Connected
9    {s41.mad,1}    Connected
7    {s45.mad,1}    Connected
8    {s16.mad,1}    Connected
5    {s23.mad,1}    Connected
6    {s11.mad,1}    Connected
11    {spine2.mad,1}    Connected
33    {mgr.mad,1}    Connected
10    {s34.mad,1}    Connected
32    {s21.mad,1}    Connected
13    {s28.mad,1}    Connected
35    {s44.mad,1}    Connected
12    {s12.mad,1}    Connected
34    {s37.mad,1}    Connected
31    {tor3.mad,1}    Connected
30    {s48.mad,1}    Connected
19    {s42.mad,1}    Connected
18    {s24.mad,1}    Connected
15    {s35.mad,1}    Connected
37    {s15.mad,1}    Connected
14    {s46.mad,1}    Connected
36    {s26.mad,1}    Connected
17    {tor1.mad,1}    Connected
39    {spine1.mad,1}    Connected
16    {s17.mad,1}    Connected
38    {s33.mad,1}    Connected
22    {s47.mad,1}    Connected
```

```
21    {s13.mad,1}    Connected
24    {s18.mad,1}    Connected
23    {s25.mad,1}    Connected
20    {s31.mad,1}    Connected
```

After all Management Agents are registered to the Manager, the Manager can start configuring the network. The current Manager implementation provides the capability of scripting a set of operations that automate the configuration of multiple systems. In this experiment we have created a script for each of the 5 DIFs under consideration (the DC Fabric DIF and the tenant DIFs). Each script will instruct the different systems that have presence in the DIF to create IPC Processes, register them to N-1 DIFs and trigger neighbor IPCP discovery. The Manager parses the script and sends the corresponding CDAP messages to the required Management Agents.

Manager script to instantiate and configure the *DC Fabric* DIF

```
createIPCP dialect:dif, appName:tor1.dcfabric, appInst:1,
 difName:dcfabric.DIF, difTemplateName:dcfabric, maName:tor1.mad, ipcpAddr:1,
 difN1NameList:110;120
createIPCP dialect:dif, appName:tor2.dcfabric, appInst:1,
 difName:dcfabric.DIF, difTemplateName:dcfabric, maName:tor2.mad, ipcpAddr:2,
 difN1NameList:210;220
createIPCP dialect:dif, appName:tor3.dcfabric, appInst:1,
 difName:dcfabric.DIF, difTemplateName:dcfabric, maName:tor3.mad, ipcpAddr:3,
 difN1NameList:310;320
createIPCP dialect:dif, appName:tor4.dcfabric, appInst:1,
 difName:dcfabric.DIF, difTemplateName:dcfabric, maName:tor4.mad, ipcpAddr:4,
 difN1NameList:410;420
wait 1000
createIPCP dialect:dif, appName:spine1.dcfabric, appInst:1,
 difName:dcfabric.DIF, difTemplateName:dcfabric, maName:spine1.mad,
 ipcpAddr:5, difN1NameList:110;210;310;410, neighbors:tor1.dcfabric/110/
dcfabric.DIF;tor2.dcfabric/210/dcfabric.DIF;tor3.dcfabric/310/
dcfabric.DIF;tor4.dcfabric/410/dcfabric.DIF
wait 2000
createIPCP dialect:dif, appName:spine2.dcfabric, appInst:1,
 difName:dcfabric.DIF, difTemplateName:dcfabric, maName:spine2.mad,
 ipcpAddr:6, difN1NameList:120;220;320;420, neighbors:tor1.dcfabric/120/
dcfabric.DIF;tor2.dcfabric/220/dcfabric.DIF;tor3.dcfabric/320/
dcfabric.DIF;tor4.dcfabric/420/dcfabric.DIF
```

The listing above shows the steps required to configure the different systems so that the *dcfabric DIF* is fully instantiated. Below there is an excerpt of the state of the *spine1* system before and after the execution of

the Manager script. The Manager instantiated a new IPC Process, registered it to N-1 DIFs *120*, *220*, *320* and *420* and allocated flows to neighbor IPC Processes in systems tor1, tor2, tor3 and tor4.

State of IPC Processes in system *spine1* before the creation of the *dcfabric DIF*

```
IPCM >>> list-ipcps
Management Agent name: spine2.mad-1--
Management Agent active connections  ( Manager name | via DIF )
        rina.apps.manager-1-- | NMS.DIF

Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 120 | - | -
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 220 | - | -
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 320 | - | -
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 3456 |
 NMS.35.IPCP-1-- | 1
    5 | eth.5.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 420 | - | -
    6 | NMS.35.IPCP:1:: | normal-ipc | ASSIGNED TO DIF NMS.DIF | - | 2
```

State of the IPC Processes in system *spine1* after the creation of the *dcfabric DIF*

```
Management Agent name: spine2.mad-1--
Management Agent active connections  ( Manager name | via DIF )
        rina.apps.manager-1-- | NMS.DIF

Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 120 |
 spine2.dcfabric-1-- | 3
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 220 |
 spine2.dcfabric-1-- | 4
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 320 |
 spine2.dcfabric-1-- | 5
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 3456 |
 NMS.35.IPCP-1-- | 1
    5 | eth.5.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 420 |
 spine2.dcfabric-1-- | 6
    6 | NMS.35.IPCP:1:: | normal-ipc | ASSIGNED TO DIF NMS.DIF | - | 2
    7 | spine2.dcfabric:1:: | normal-ipc | ASSIGNED TO DIF dcfabric.DIF | - |
 -

IPCM >>>
```

After creating the *DC fabric DIF*, we now proceed to instantiate the *tenant 1* and *tenant 2* DIFs. Since both DIFs have an equivalent composition, we just show the Manager script to instantiate the *tenant 1* DIF - provided in the following listing. Below we can also see the state of the system *tor1* before and after instantiating the *tenant 1* DIF. We can observe that a new IPC Process belonging to DIF *t1* was created, registered to N-1 DIFs l1, l2, l3, l4 and dcfabric and enrolled to neighbors s11, s12, s13, s14 and tor2.

Manager script to instantiate and configure the *t1* DIF

```
createIPCP dialect:dif, appName:s11.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s11.mad, ipcpAddr:1, difN1NameList:11
createIPCP dialect:dif, appName:s12.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s12.mad, ipcpAddr:2, difN1NameList:12
createIPCP dialect:dif, appName:s13.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s13.mad, ipcpAddr:3, difN1NameList:13
createIPCP dialect:dif, appName:s14.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s14.mad, ipcpAddr:4, difN1NameList:14
createIPCP dialect:dif, appName:s21.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s21.mad, ipcpAddr:5, difN1NameList:21
createIPCP dialect:dif, appName:s22.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s22.mad, ipcpAddr:6, difN1NameList:22
createIPCP dialect:dif, appName:s23.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s23.mad, ipcpAddr:7, difN1NameList:23
createIPCP dialect:dif, appName:s24.t1, appInst:1, difName:t1.DIF,
 difTemplateName:t1, maName:s24.mad, ipcpAddr:8, difN1NameList:24
wait 1000
createIPCP dialect:dif, appName:tor1.t1, appInst:1,
 difName:t1.DIF, difTemplateName:t1, maName:tor1.mad, ipcpAddr:9,
 difN1NameList:11;12;13;14;dcfabric.DIF, neighbors:s11.t1/11/
t1.DIF;s12.t1/12/t1.DIF;s13.t1/13/t1.DIF;s14.t1/14/t1.DIF
wait 3000
createIPCP dialect:dif, appName:tor2.t1, appInst:1,
 difName:t1.DIF, difTemplateName:t1, maName:tor2.mad, ipcpAddr:10,
 difN1NameList:21;22;23;24;dcfabric.DIF, neighbors:s21.t1/21/
t1.DIF;s22.t1/22/t1.DIF;s23.t1/23/t1.DIF;s24.t1/24/t1.DIF;tor1.t1/
dcfabric.DIF/t1.DIF
```

State of IPCPs in system *tor1* before instantiating the *t1* DIF

```
IPCM >>> list-ipcps
Management Agent name: tor1.mad-1--
Management Agent active connections  ( Manager name | via DIF )
        rina.apps.manager-1-- | NMS.DIF
```

```
Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 11 | - | -
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
 tor1.dcfabric-1-- | 3
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 12 | - | -
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 120 |
 tor1.dcfabric-1-- | 4
    5 | eth.5.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 13 | - | -
    6 | eth.6.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 14 | - | -
    7 | eth.7.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 15 | - | -
    8 | eth.8.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 16 | - | -
    9 | eth.9.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 17 | - | -
    10 | eth.10.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 18 | - | -
    11 | eth.11.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 3456 |
 NMS.36.IPCP-1-- | 1
    12 | NMS.36.IPCP:1:: | normal-ipc | ASSIGNED TO DIF NMS.DIF | - | 2
    13 | tor1.dcfabric:1:: | normal-ipc | ASSIGNED TO DIF dcfabric.DIF | - |
 -

IPCM >>>
```

State of IPCPs in system *tor1* after instantiating the *t1* DIF

```
IPCM >>> list-ipcps
Management Agent name: tor1.mad-1--
Management Agent active connections  ( Manager name | via DIF )
        rina.apps.manager-1-- | NMS.DIF

Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 11 | tor1.t1-1-- | 5
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
 tor1.dcfabric-1-- | 3
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 12 | tor1.t1-1-- | 6
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 120 |
 tor1.dcfabric-1-- | 4
    5 | eth.5.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 13 | tor1.t1-1-- | 7
    6 | eth.6.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 14 | tor1.t1-1-- | 8
    7 | eth.7.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 15 | - | -
    8 | eth.8.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 16 | - | -
    9 | eth.9.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 17 | - | -
    10 | eth.10.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 18 | - | -
    11 | eth.11.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 3456 |
 NMS.36.IPCP-1-- | 1
    12 | NMS.36.IPCP:1:: | normal-ipc | ASSIGNED TO DIF NMS.DIF | - | 2
    13 | tor1.dcfabric:1:: | normal-ipc | ASSIGNED TO DIF dcfabric.DIF |
 tor1.t1-1-- | 9
```

```
    14 | tor1.t1:1:: | normal-ipc | ASSIGNED TO DIF t1.DIF | - | -

IPCM >>>
```

Finally we proceed to instantiate the DIFs *t3* and *t4* using their respective scripts (shown in the listing below). We now check the state of the system *tor3* before and after the instantiation of DIF *t3*, and observe the changes. The Manager has instantiated a new IPC Process, registered it to N-1 DIFs 36, 46 and dcfabric and connected the IPC Process to its peers in systems s36, s46, tor1 and tor3.

Manager script to instantiate and configure the *t3* DIF

```
createIPCP dialect:dif, appName:s15.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s15.mad, ipcpAddr:1, difN1NameList:15
createIPCP dialect:dif, appName:s16.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s16.mad, ipcpAddr:2, difN1NameList:16
createIPCP dialect:dif, appName:s25.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s25.mad, ipcpAddr:3, difN1NameList:25
createIPCP dialect:dif, appName:s26.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s26.mad, ipcpAddr:4, difN1NameList:26
createIPCP dialect:dif, appName:s35.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s35.mad, ipcpAddr:5, difN1NameList:35
createIPCP dialect:dif, appName:s36.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s36.mad, ipcpAddr:6, difN1NameList:36
createIPCP dialect:dif, appName:s45.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s45.mad, ipcpAddr:7, difN1NameList:45
createIPCP dialect:dif, appName:s46.t3, appInst:1, difName:t3.DIF,
 difTemplateName:t3, maName:s46.mad, ipcpAddr:8, difN1NameList:46
wait 1000
createIPCP dialect:dif, appName:tor1.t3, appInst:1,
 difName:t3.DIF, difTemplateName:t3, maName:tor1.mad, ipcpAddr:9,
 difN1NameList:15;16;dcfabric.DIF, neighbors:s15.t3/15/t3.DIF;s16.t3/16/
t3.DIF
wait 3000
createIPCP dialect:dif, appName:tor2.t3, appInst:1,
 difName:t3.DIF, difTemplateName:t3, maName:tor2.mad, ipcpAddr:10,
 difN1NameList:25;26;dcfabric.DIF, neighbors:s25.t3/25/t3.DIF;s26.t3/26/
t3.DIF;tor1.t3/dcfabric.DIF/t3.DIF
wait 3000
createIPCP dialect:dif, appName:tor3.t3, appInst:1,
 difName:t3.DIF, difTemplateName:t3, maName:tor3.mad, ipcpAddr:11,
 difN1NameList:35;36;dcfabric.DIF, neighbors:s35.t3/35/t3.DIF;s36.t3/36/
t3.DIF;tor2.t3/dcfabric.DIF/t3.DIF
wait 3000
createIPCP dialect:dif, appName:tor4.t3, appInst:1,
 difName:t3.DIF, difTemplateName:t3, maName:tor4.mad, ipcpAddr:12,
```

```
 difN1NameList:45;46;dcfabric.DIF, neighbors:s45.t3/45/t3.DIF;s46.t3/46/
t3.DIF;tor3.t3/dcfabric.DIF/t3.DIF;tor1.t3/dcfabric.DIF/tor3.DIF
```

State of IPCPs in system *tor3* before instantiating the *t3* DIF

```
IPCM >>> list-ipcps
Management Agent name: tor3.mad-1--
Management Agent active connections  ( Manager name | via DIF )
        rina.apps.manager-1-- | NMS.DIF

Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 31 | tor3.t2-1-- | 5
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 310 |
 tor3.dcfabric-1-- | 3
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 32 | tor3.t2-1-- | 6
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 320 |
 tor3.dcfabric-1-- | 4
    5 | eth.5.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 33 | tor3.t2-1-- | 7
    6 | eth.6.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 34 | tor3.t2-1-- | 8
    7 | eth.7.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 3456 |
 NMS.38.IPCP-1-- | 1
    8 | eth.8.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 35 | - | -
    9 | eth.9.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 36 | - | -
    10 | eth.10.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 37 | - | -
    11 | eth.11.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 38 | - | -
    12 | NMS.38.IPCP:1:: | normal-ipc | ASSIGNED TO DIF NMS.DIF | - | 2
    13 | tor3.dcfabric:1:: | normal-ipc | ASSIGNED TO DIF dcfabric.DIF |
 tor3.t2-1-- | 9
    14 | tor3.t2:1:: | normal-ipc | ASSIGNED TO DIF t2.DIF | - | -
```

State of IPCPs in system *tor3* after instantiating the *t3* DIF

```
Management Agent name: tor3.mad-1--
Management Agent active connections  ( Manager name | via DIF )
        rina.apps.manager-1-- | NMS.DIF

Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 31 | tor3.t2-1-- | 5
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 310 |
 tor3.dcfabric-1-- | 3
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 32 | tor3.t2-1-- | 6
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 320 |
 tor3.dcfabric-1-- | 4
    5 | eth.5.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 33 | tor3.t2-1-- | 7
    6 | eth.6.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 34 | tor3.t2-1-- | 8
```

```
   7 | eth.7.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 3456 |
NMS.38.IPCP-1-- | 1
   8 | eth.8.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 35 | tor3.t3-1-- |
10
   9 | eth.9.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 36 | tor3.t3-1-- |
11
   10 | eth.10.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 37 | - | -
   11 | eth.11.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 38 | - | -
   12 | NMS.38.IPCP:1:: | normal-ipc | ASSIGNED TO DIF NMS.DIF | - | 2
   13 | tor3.dcfabric:1:: | normal-ipc | ASSIGNED TO DIF dcfabric.DIF |
tor3.t2-1--, tor3.t3-1-- | 9, 12, 13
   14 | tor3.t2:1:: | normal-ipc | ASSIGNED TO DIF t2.DIF | - | -
   15 | tor3.t3:1:: | normal-ipc | ASSIGNED TO DIF t3.DIF | - | -

IPCM >>>
```

To conclude, the configuration management features of PRISTINE's Management System work as expected; the whole experiment (instantiating and configuring the 5 DIFs over the 38 systems) can be executed in less than 30 seconds. Since all the layers have the same state and configuration model, the structure of DIF templates and Manager scripts for different layers are identical; greatly facilitating the task of managing the configuration of the whole network and reducing the number of lines of code required to implement the Management System.

5.1.2. Multi-path performance evaluation experiments

The goal of the multi-path performance evaluation experiment is to evaluate the behaviour of the simple multipath policy-set in a realistic datacentre configuration, such as the one depicted in Figure 77. The experiment is divided in two parts: the first one considers the DC fabric DIF in isolation; while the second one involves the DC fabric DIF supporting multiple VPN DIFs devoted to different tenants.

Experiment 1: Multipath policies in DC fabric DIF with 2 and 3 spines

The goals of this experiment are to achieve close to 100% utilization in the DC fabric DIF by exploiting multiple, redundant paths. In concrete, the Key Performance Indicators measured in this experiment are:

- Distribution of N-flows over N-1 flows (both in terms of number of flows and capacity)
- N-1 flow utilization

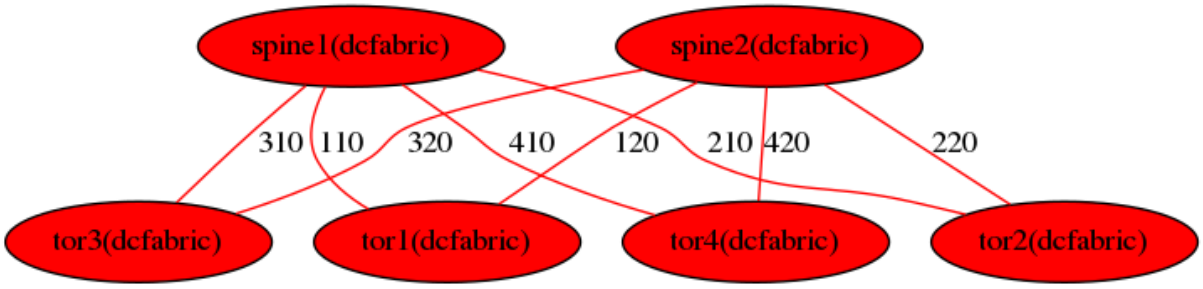


Figure 80. DC Fabric DIF in Experiment 1, with 2 spines

Figure 80 shows the graph of the DC fabric DIF with 2 spines. Multipath forwarding policies are running on TORs, each of which has 2 equal cost paths to all the other TORs. The aggregate outbound capacity at each TOR is 200 Mbps, which can only be reached if each TOR equally uses the two N-1 flows to balance the N-DIF traffic. At each experiment run N parallel flows are setup between tor1 and tor3, and another N flows between tor2 and tor4. The offered load at each TOR is always constant and 200 Mbps (each flow has a constant rate of 200/N Mbps). During each experiment we measure the flow distribution per N-1 port at TORs, as well as the throughput (Mbps) through each N-1 port. Demonstrator configuration file for 2 spines shows the demonstrator configuration file that can be used to reproduce the experiment.

Demonstrator configuration file for 2 spines

```
eth 110 100Mbps tor1 spine1
eth 120 100Mbps tor1 spine2
eth 210 100Mbps tor2 spine1
eth 220 100Mbps tor2 spine2
eth 310 100Mbps tor3 spine1
eth 320 100Mbps tor3 spine2
eth 410 100Mbps tor4 spine1
eth 420 100Mbps tor4 spine2

# DIF dcfabric
dif dcfabric tor1 110 120
dif dcfabric tor2 210 220
dif dcfabric tor3 310 320
dif dcfabric tor4 410 420
dif dcfabric spine1 110 210 310 410
dif dcfabric spine2 120 220 320 420

#Policies
#Multipath FABRIC
policy dcfabric rmt.pff multipath
policy dcfabric routing link-state routingAlgorithm=ECMPDijkstra
```

```
#Apps
appmap dcfabric traffic.generator.server 1
appmap dcfabric traffic.generator.server 2
```

We have repeated the experiment a second time with the scenario shown in Figure 81, in which the DC-fabric has 3 spines and therefore there are 3 equal cost paths between all TORs. This way we observe the behaviour of the multipath policy when it has to balance the load between 3 N-1 ports. Demonstrator configuration file for 3 spines shows the demonstrator configuration file required to reproduce the experiment.

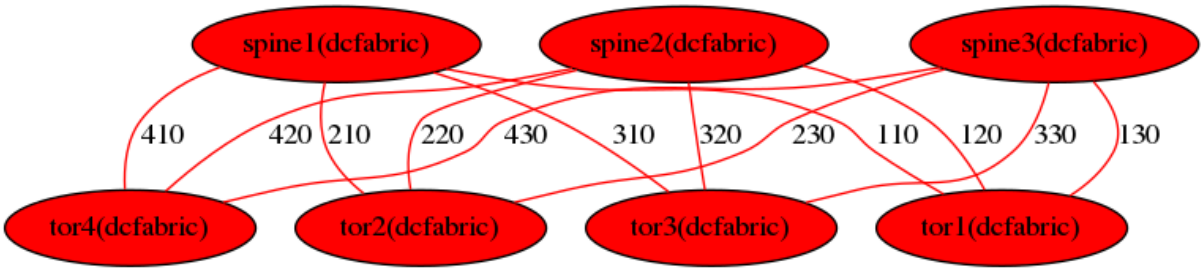


Figure 81. DC Fabric DIF in Experiment 1, with 3 spines

Demonstrator configuration file for 3 spines

```
eth 110 67Mbps tor1 spine1
eth 120 67Mbps tor1 spine2
eth 130 67Mbps tor1 spine3
eth 210 67Mbps tor2 spine1
eth 220 67Mbps tor2 spine2
eth 230 67Mbps tor2 spine3
eth 310 67Mbps tor3 spine1
eth 320 67Mbps tor3 spine2
eth 330 67Mbps tor3 spine3
eth 410 67Mbps tor4 spine1
eth 420 67Mbps tor4 spine2
eth 430 67Mbps tor4 spine3

# DIF dcfabric
dif dcfabric tor1 110 120 130
dif dcfabric tor2 210 220 230
dif dcfabric tor3 310 320 330
dif dcfabric tor4 410 420 430
dif dcfabric spine1 110 210 310 410
dif dcfabric spine2 120 220 320 420
dif dcfabric spine3 130 230 330 430

#Policies
```

```
#Multipath FABRIC
policy dcfabric rmt.pff multipath
policy dcfabric routing link-state routingAlgorithm=ECMPDijkstra

#Apps
appmap dcfabric traffic.generator.server 1
appmap dcfabric traffic.generator.server 2
```

The graphs in Figure 82 to Figure 85 show the results of the experiment for the configurations with 2 and 3 spines. Vertical axis shows either the number of flows per N-1 port or the N-1 port utilization in Mbps, while the horizontal axis shows the number of concurrent flows between each pair of TORs.

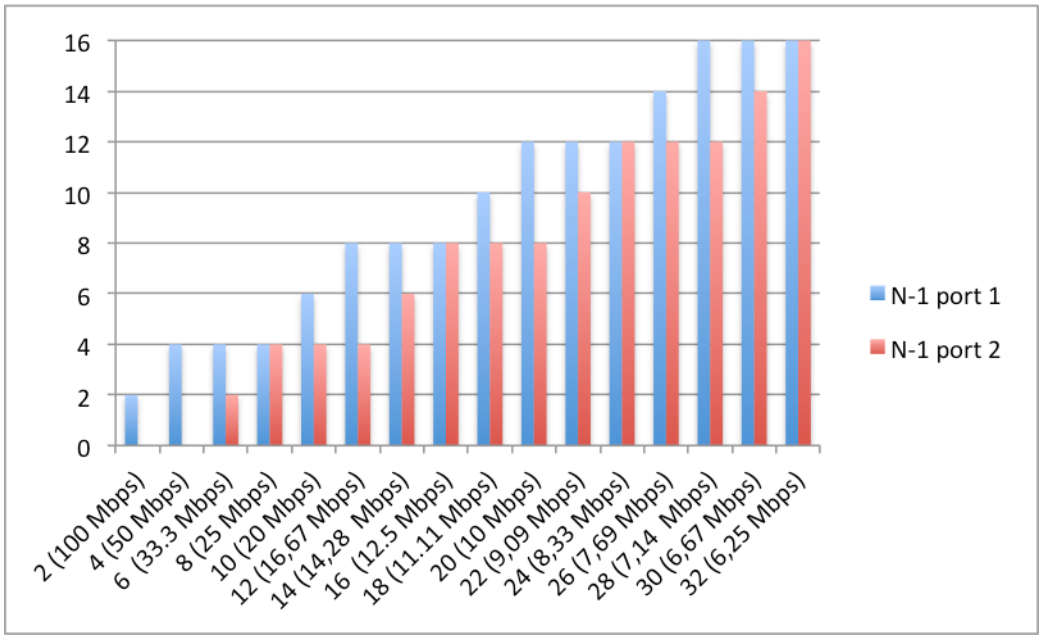


Figure 82. Distribution of flows per N-1 port, for a different number of concurrent flows (2 spines)

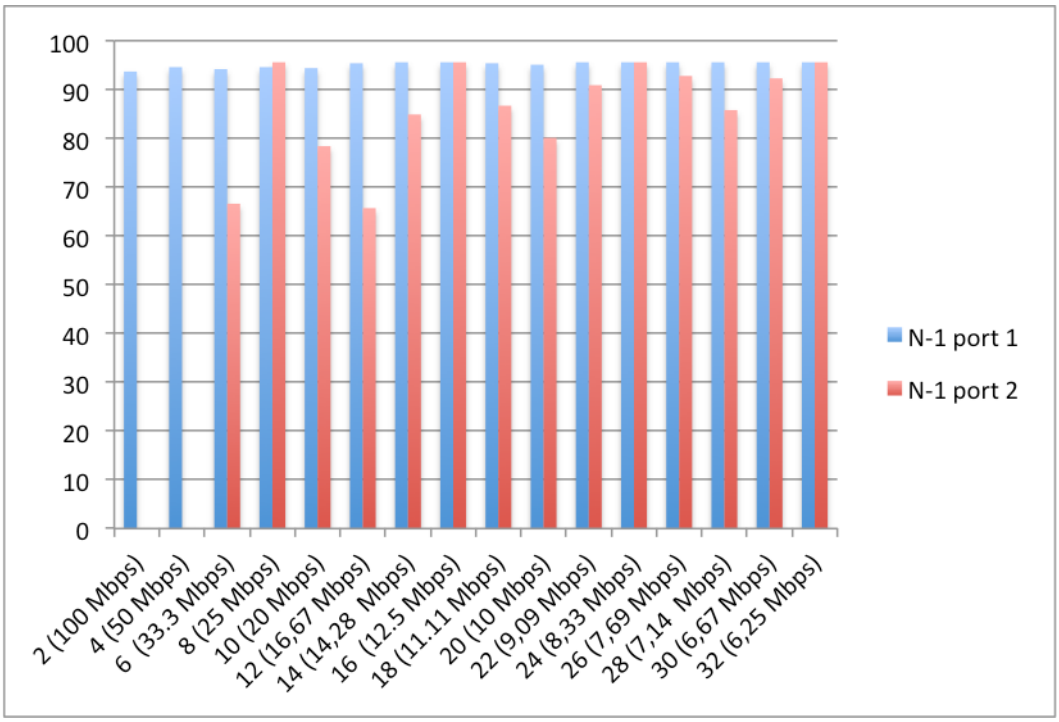


Figure 83. Throughput (Mbps) per N-1 port, for a different number of concurrent flows (2 spines)

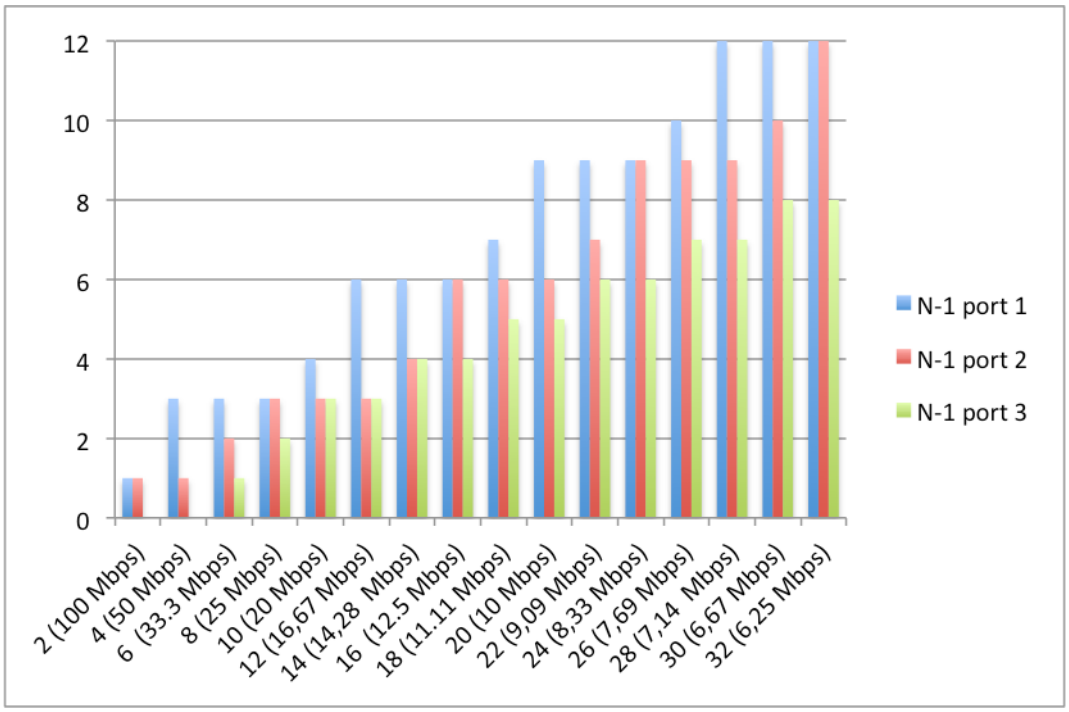


Figure 84. Distribution of flows per N-1 port, for a different number of concurrent flows (3 spines)

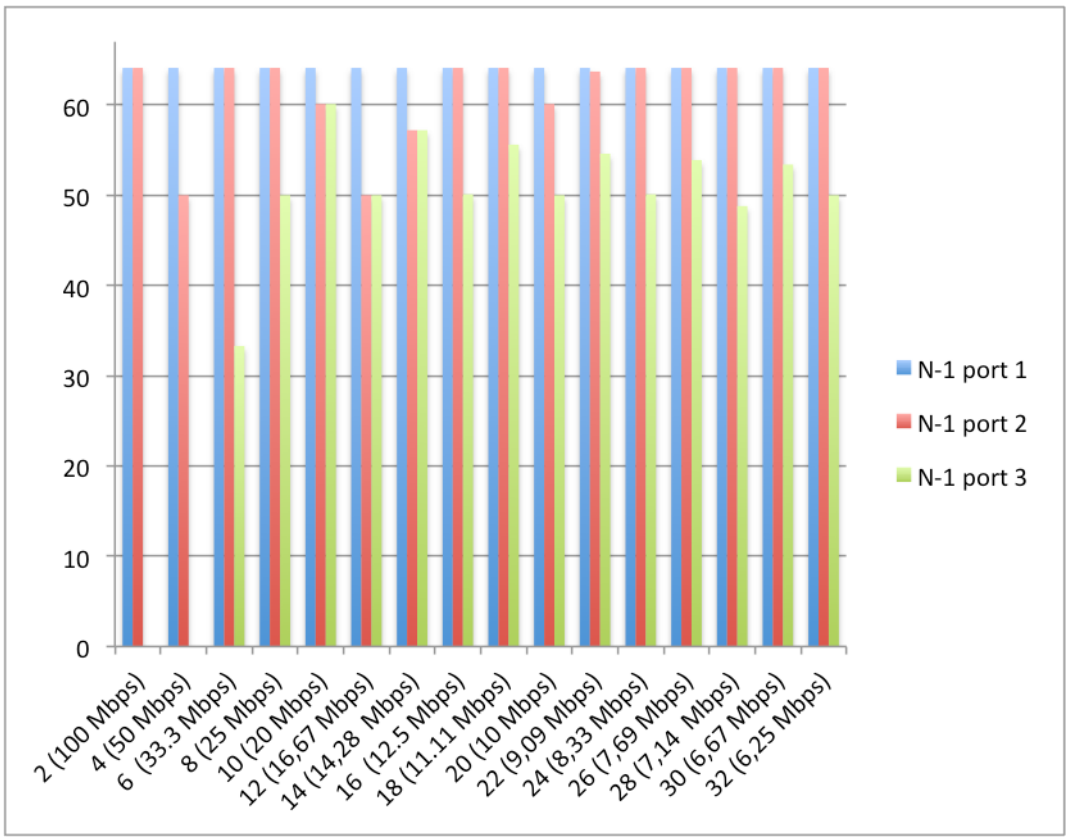


Figure 85. Throughput (Mbps) per N-1 port, for a different number of concurrent flows (3 spines)

The current algorithm to chose the path is based on a hash-threshold approach (like ECMP) that has a low computational cost and preserves the order of the packets in a flow. As expected, it doesn't achieve perfect balancing of flows amongst N-1 ports, resulting in a not ideal N-1 port utilisation and a higher packet loss than other approaches that work without preserving the flow order or storing per N-flow state. However, in environments with a relatively large number of medium or small flows in relation to the capacity of the N-1 ports (such as the environment found in many data-centers), the balancing of the load is a good approximation of the ideal one.

Other factors that could improve the load-balancing behaviour of this multipath policy are the choice of the hash algorithm and the assignment of cep-ids. The current policy uses a CRC16 algorithm on the values of the PCI header. Other algorithms that have a larger computational cost but better balancing properties might be applicable to different situations. Last but not least, the assignment of connection-endpoint-ids is done sequentially; so the values in the PCI header for two consecutive flows are very close. If cep-ids were assigned randomly, PCIs of consecutive flows would have

more entropy and would facilitate that the hash algorithm calculated values belonging to different regions (and therefore different N-1 ports).

Experiment 2: Setup DC fabric DIF and 16 VPN DIFs

Once the multipath forwarding policy behaviour has been validated in a single DIF, we proceed to check its correct operation in the full DC scenario. To do that the creation of VPN DIFs is included in the demonstrator configuration file, resulting in the creation of the 16 VPN DIFs depicted in Figure 86.

Figure 86. VPN DIFs in experiment 2 (there are 16 VPN DIFs on top of the DC fabric one)

```
eth 110 200Mbps tor1 spine1
eth 120 200Mbps tor1 spine2
```

```
eth 11 25Mbps s11 tor1
eth 12 25Mbps s12 tor1
eth 13 25Mbps s13 tor1
eth 14 25Mbps s14 tor1
eth 15 25Mbps s15 tor1
eth 16 25Mbps s16 tor1
eth 17 25Mbps s17 tor1
eth 18 25Mbps s18 tor1
eth 210 200Mbps tor2 spine1
eth 220 200Mbps tor2 spine2
eth 21 25Mbps s21 tor2
eth 22 25Mbps s22 tor2
eth 23 25Mbps s23 tor2
eth 24 25Mbps s24 tor2
eth 25 25Mbps s25 tor2
eth 26 25Mbps s26 tor2
eth 27 25Mbps s27 tor2
eth 28 25Mbps s28 tor2
eth 310 200Mbps tor3 spine1
eth 320 200Mbps tor3 spine2
eth 31 25Mbps s31 tor3
eth 32 25Mbps s32 tor3
eth 33 25Mbps s33 tor3
eth 34 25Mbps s34 tor3
eth 35 25Mbps s35 tor3
eth 36 25Mbps s36 tor3
eth 37 25Mbps s37 tor3
eth 38 25Mbps s38 tor3
eth 410 200Mbps tor4 spine1
eth 420 200Mbps tor4 spine2
eth 41 25Mbps s41 tor4
eth 42 25Mbps s42 tor4
eth 43 25Mbps s43 tor4
eth 44 25Mbps s44 tor4
eth 45 25Mbps s45 tor4
eth 46 25Mbps s46 tor4
eth 47 25Mbps s47 tor4
eth 48 25Mbps s48 tor4

# DIF dcfabric
dif dcfabric tor1 110 120
dif dcfabric tor2 210 220
dif dcfabric tor3 310 320
dif dcfabric tor4 410 420
dif dcfabric spine1 110 210 310 410
dif dcfabric spine2 120 220 320 420

# DIF VPN1
```

```
dif vpn1 tor1 11 dcfabric
dif vpn1 s11 11
dif vpn1 tor2 21 dcfabric
dif vpn1 s21 21

# DIF VPN2
dif vpn2 tor1 12 dcfabric
dif vpn2 s12 12
dif vpn2 tor2 22 dcfabric
dif vpn2 s22 22

# DIF VPN3
dif vpn3 tor1 13 dcfabric
dif vpn3 s13 13
dif vpn3 tor2 23 dcfabric
dif vpn3 s23 23

# DIF VPN4
dif vpn4 tor1 14 dcfabric
dif vpn4 s14 14
dif vpn4 tor2 24 dcfabric
dif vpn4 s24 24

# DIF VPN5
dif vpn5 tor1 15 dcfabric
dif vpn5 s15 15
dif vpn5 tor2 25 dcfabric
dif vpn5 s25 25

# DIF VPN6
dif vpn6 tor1 16 dcfabric
dif vpn6 s16 16
dif vpn6 tor2 26 dcfabric
dif vpn6 s26 26

# DIF VPN7
dif vpn7 tor1 17 dcfabric
dif vpn7 s17 17
dif vpn7 tor2 27 dcfabric
dif vpn7 s27 27

# DIF VPN8
dif vpn8 tor1 18 dcfabric
dif vpn8 s18 18
dif vpn8 tor2 28 dcfabric
dif vpn8 s28 28

# DIF VPN9
```

```
dif vpn9 tor3 31 dcfabric
dif vpn9 s31 31
dif vpn9 tor4 41 dcfabric
dif vpn9 s41 41

# DIF VPN10
dif vpn10 tor3 32 dcfabric
dif vpn10 s32 32
dif vpn10 tor4 42 dcfabric
dif vpn10 s42 42

# DIF VPN11
dif vpn11 tor3 33 dcfabric
dif vpn11 s33 33
dif vpn11 tor4 43 dcfabric
dif vpn11 s43 43

# DIF VPN12
dif vpn12 tor3 34 dcfabric
dif vpn12 s34 34
dif vpn12 tor4 44 dcfabric
dif vpn12 s44 44

# DIF VPN13
dif vpn13 tor3 35 dcfabric
dif vpn13 s35 35
dif vpn13 tor4 45 dcfabric
dif vpn13 s45 45

# DIF VPN14
dif vpn14 tor3 36 dcfabric
dif vpn14 s36 36
dif vpn14 tor4 46 dcfabric
dif vpn14 s46 46

# DIF VPN15
dif vpn15 tor3 37 dcfabric
dif vpn15 s37 37
dif vpn15 tor4 47 dcfabric
dif vpn15 s47 47

# DIF VPN16
dif vpn16 tor3 38 dcfabric
dif vpn16 s38 38
dif vpn16 tor4 48 dcfabric
dif vpn16 s48 48

#Policies
```

```
#Multipath FABRIC
policy dcfabric rmt.pff multipath
policy dcfabric routing link-state routingAlgorithm=ECMPDijkstra
```

To carry out the experiment we setup a rina-tgen session in each VPN DIF. We the measure the load going through each N-1 port at each ToR, obtaining the results depicted in Figure 87. Each ToR load balances the 8 flows belonging to the 8 VPNs that go through it, achieving a perfect flow split between the two N-1 ports (4:4).

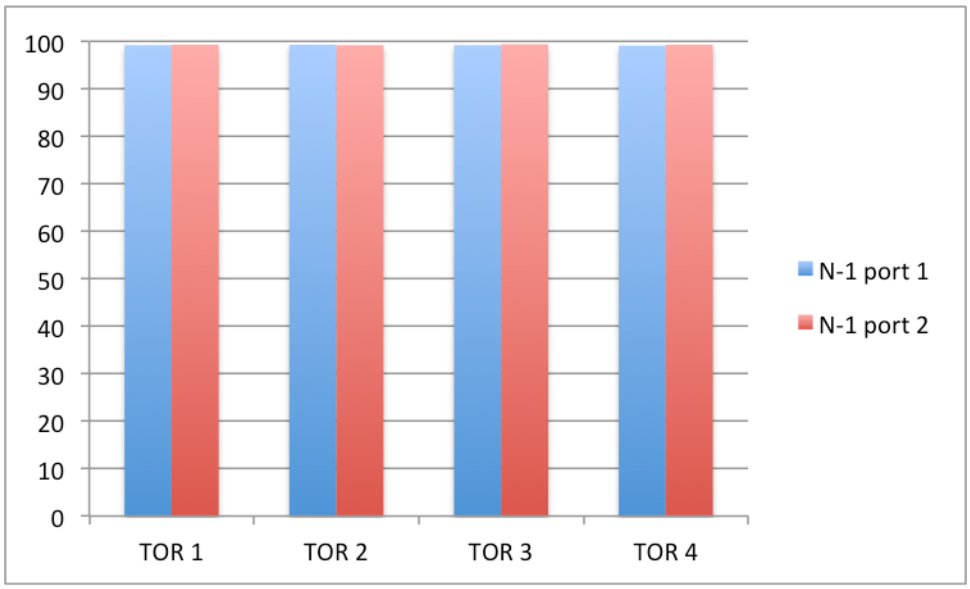


Figure 87. Throughput (Mbps) measured at all N-1 ports of the four ToRs in the experiment

5.1.3. Congestion Management experiments

These set of experiments evaluates the behaviour of the congestion management policies in the DC fabric. The key metrics to be observed are the *occupation of the RMT queues at bottleneck links*, *accumulated packet loss* and the *throughput of the flows that are competing for the bottleneck resources*. Two sets of experiments have been performed: one set with a single DIF, to verify the correct operation of the congestion management policies; and another set with two levels of DIFs (DC fabric and VPNs). In each experiment we have use three sets of policy sets, as shown in Table 5.

Table 5. Policy sets used in the congestion management experiments

Policy-set name	RMT ps	EFCP (dtcp) ps
default	Default RMT ps, drops PDUs when qlength > qmax	Flow control with fixed window size, no retransmissions
red-ps	Perform PDU dropping and marking as described in the RED algorithm	TCP-like flow control with ECN support, no retransmissions
cas-ps	Mark PDUs when average queue length > 1, drop when qlength > qmax	Follow Jain's binary feedback scheme to increase/decrease window size based on ECN feedback, no retransmissions

Experiment 1: single DIF with incast and barbell configurations

Figure 88 illustrates the first scenario with an incast configuration. All N-1 flows have 50 Mbps capacity except for the one between *r1* and *s21*, which is 40 Mbps. The experiment consists in running 4 parallel rina-tgen sessions between 4 clients at *s11* - *s14* and a rina-tgen server at *s21*. The rina-tgen clients sends data at the maximum rate available, therefore the oversubscription at the bottleneck link is of 5 to 1 (4 * 50 Mbps of incoming traffic and only 40 Mbps of outgoing traffic). At IPCP *r1* we monitor the RMT queue that sits on top of the N-1 flow to *s21*. All the RMT queues at all IPCPs are configured with a maximum length of 600 PDUs, and initial EFCP window sizes are 200 PDUs.

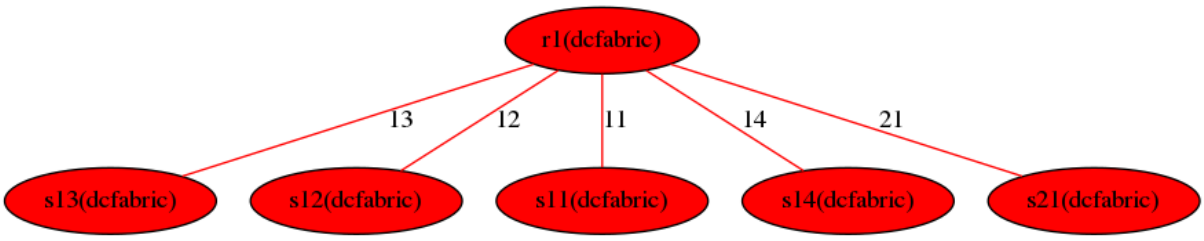


Figure 88. Single DIF incast configuration

Demonstrator configuration file

```
eth 11 50Mbps s11 r1
eth 12 50Mbps s12 r1
eth 13 50Mbps s13 r1
eth 14 50Mbps s14 r1
eth 21 40Mbps s21 r1
```

```
# DIF dcfabric
dif dcfabric s11 11
dif dcfabric s12 12
dif dcfabric s13 13
dif dcfabric s14 14
dif dcfabric s21 21
dif dcfabric r1 11 12 13 14 21

#Policies
#Default Policies
#policy dcfabric rmt default q_max=600

#CAS
#policy dcfabric rmt cas-ps q_max=600
#policy dcfabric efcp.*.dtcp cas-ps

#RED
#policy dcfabric rmt red-ps qmax_p=600 qth_min_p=17 qth_max_p=179 Wlog_p=7
 Plog_p=12
policy dcfabric rmt red-ps qmax_p=600 qth_min_p=10 qth_max_p=30 Wlog_p=10
 Plog_p=9
policy dcfabric efcp.*.dtcp red-ps
```

First we will focus on the queue lengths and dropped PDUs at the RMT queue. Figure 89 shows the behaviour of the default RMT policy set. Since in this case the RMT is not providing any indication of the congestion back to the EFCP senders and these always use the same window size, the queues are full most of the time and dropped PDUs grow linearly with the experiment time. Figure 90 shows the behaviour of the RED policy set. At the beginning there is some loss of PDUs due to the fact that EFCP senders have a large window. However, the policy set reacts to marking and dropping PDUs, and EFCP receivers adjust their rate accordingly. No more PDUs are lost during the experiment. Finally Figure 91 depicts the behaviour of the CAS policy set. In this case PDU loss is avoided even at the beginning of the experiment, since the marking policy acts earlier than in the RED case.

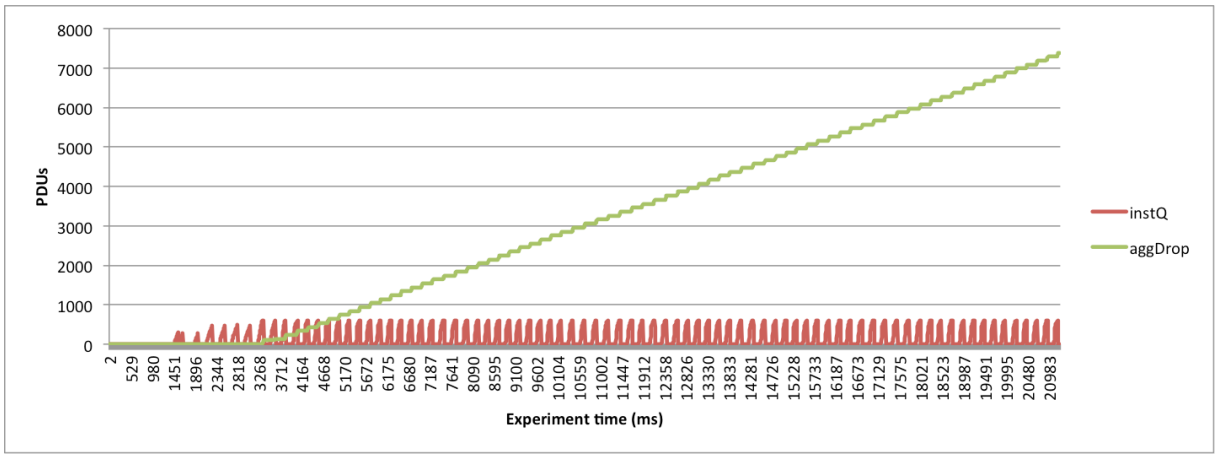


Figure 89. RMT queue length and accumulated dropped PDUs vs. experiment time, default policy-set

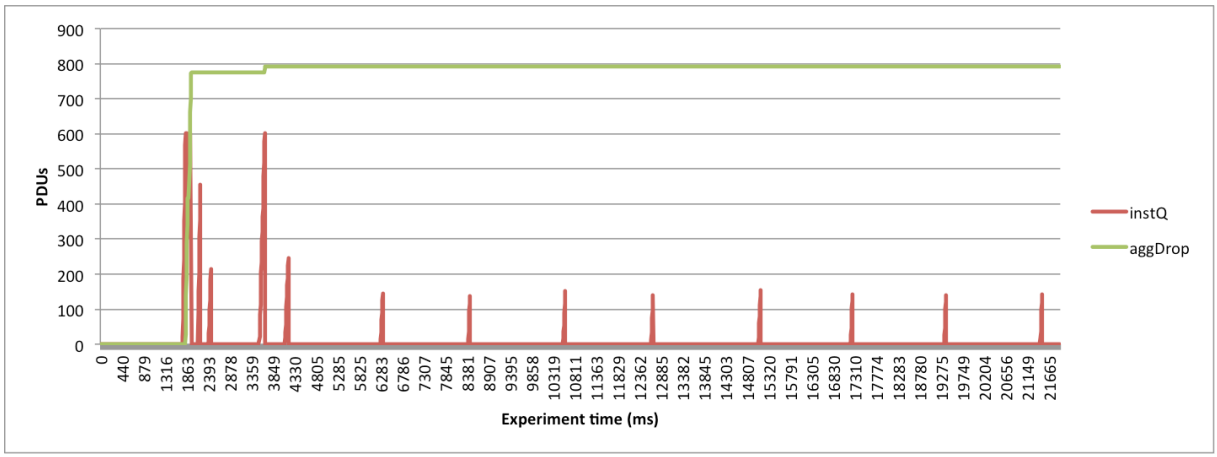


Figure 90. RMT queue length and accumulated dropped PDUs vs. experiment time, RED policy-set

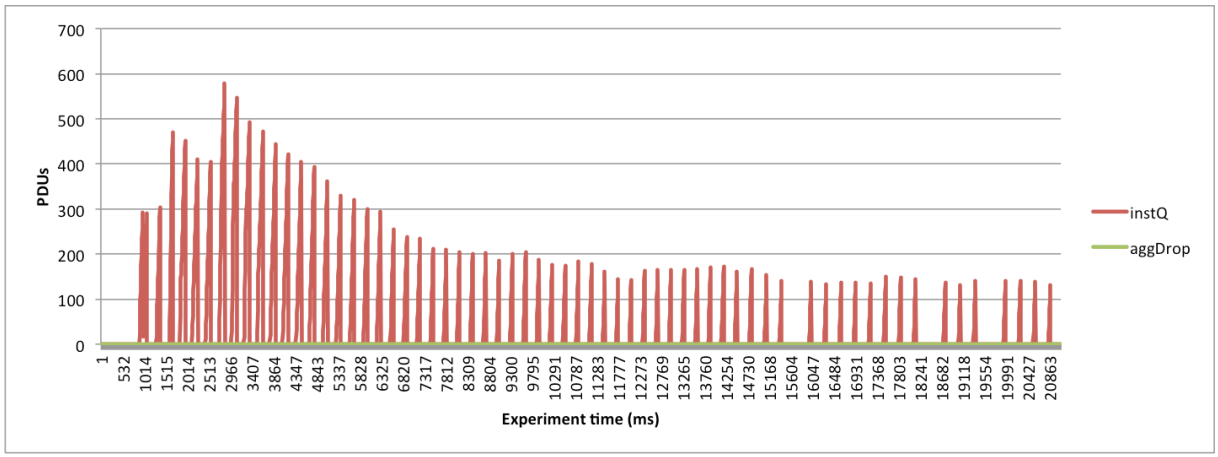


Figure 91. RMT queue length and accumulated dropped PDUs vs. experiment time, CAS policy-set

Regarding the throughput achieved by each individual flow across the bottleneck link, the graphs below show that, as expected, the default policy achieves the ideal result: all flows are able to send at the maximum rate,

close to 10 Mbps (Figure 92). This is due to the fact that they all use the same window size all the time. Of course this comes with the price of huge packet loss, which makes this configuration not practical. The other two policy sets try to approximate this behaviour while keeping the RMT queues at the bottleneck link under control; therefore the window sizes of EFCP senders vary through time. As it can be seen in Figure 91 and Figure 90, the cas policy set uses a smoother controller compared to RED, but both controllers do a similar job in sharing the resources of the bottleneck link between the competing flows.

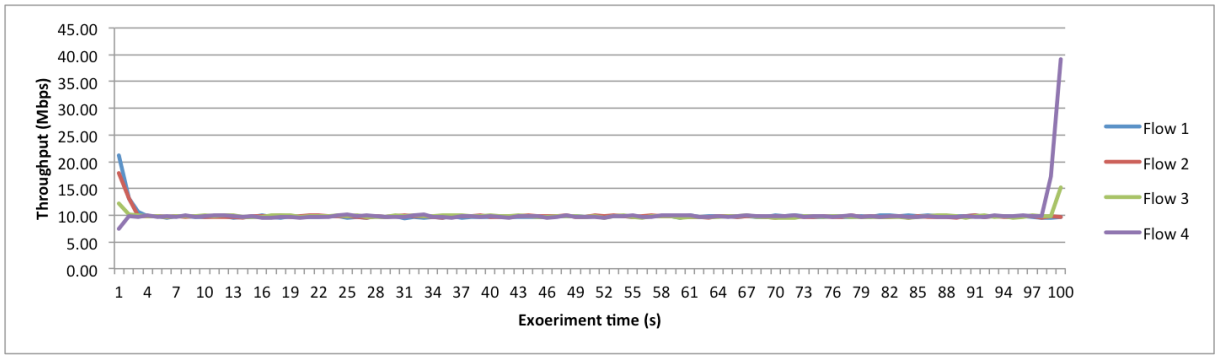


Figure 92. Throuhput of the different EFCP flows vs. experiment time, default policy-set

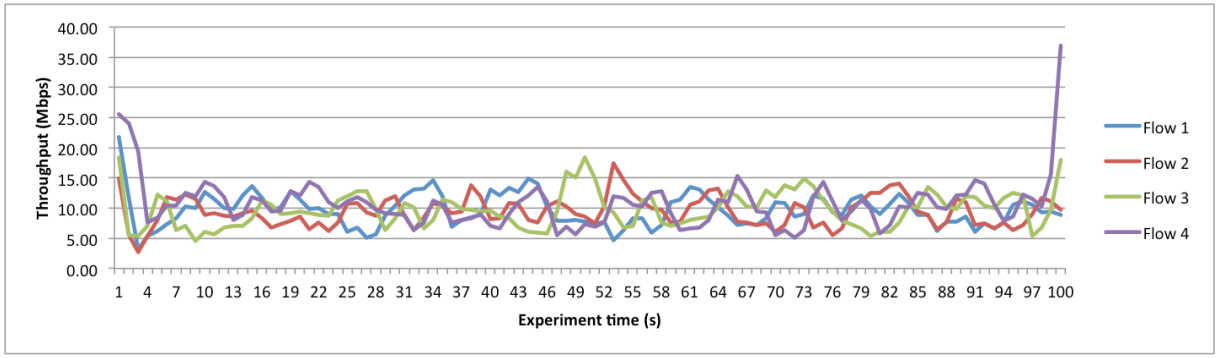


Figure 93. Throuhput of the different EFCP flows vs. experiment time, RED policy-set

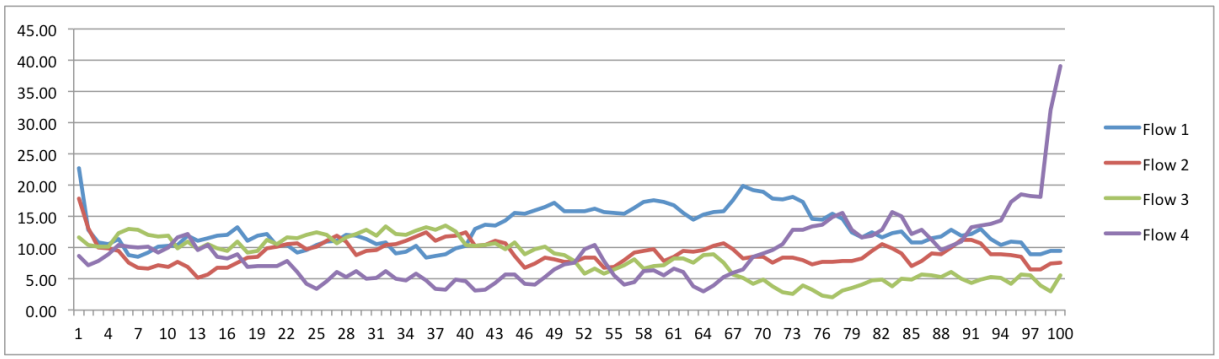


Figure 94. Throuhput of the different EFCP flows vs. experiment time, CAS policy-set

Figure 95 illustrates the barbell configuration scenario. N-1 flows between servers and routers (*r1* and *r2*) are of 50 Mbps, while the bottleneck link

between *r1* and *r2* has a capacity of 40 Mbps. In this setup we execute four parallel rina-tgen flows between servers *s1i* and *s4i*, resulting in an oversubscription ratio of 5 to 1 for the bottleneck link as in the incast configuration. We observe the RMT queue at IPCP *r1* sitting on top of the N-1 flow to *r2*, and the throughput achieved by each rina-tgen session.

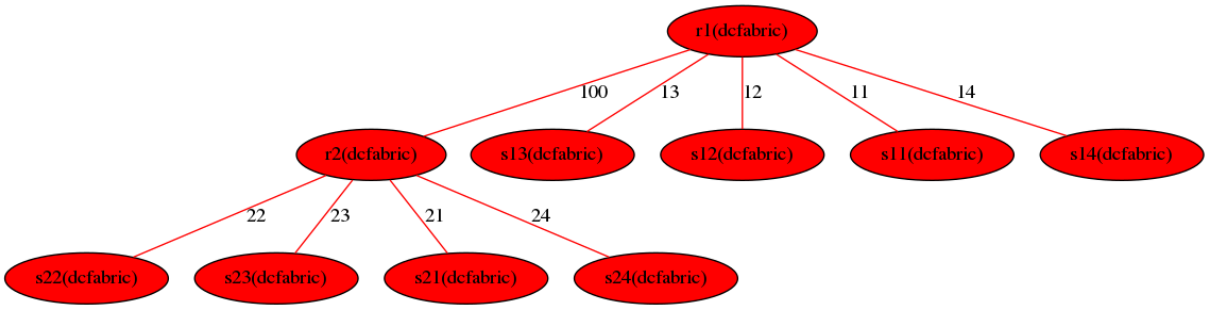


Figure 95. Congestion management experiments, single DIF barbell configuration

Demonstrator configuration file

```
eth 11 50Mbps s11 r1
eth 12 50Mbps s12 r1
eth 13 50Mbps s13 r1
eth 14 50Mbps s14 r1
eth 21 50Mbps s21 r2
eth 22 50Mbps s22 r2
eth 23 50Mbps s23 r2
eth 24 50Mbps s24 r2
eth 100 40Mbps r1 r2

# DIF dcfabric
dif dcfabric s11 11
dif dcfabric s12 12
dif dcfabric s13 13
dif dcfabric s14 14
dif dcfabric r1 100 11 12 13 14
dif dcfabric r2 100 21 22 23 24
dif dcfabric s21 21
dif dcfabric s22 22
dif dcfabric s23 23
dif dcfabric s24 24

#Policies
#Default Policiy
#policy dcfabric rmt default q_max=600

#CAS
#policy dcfabric rmt cas-ps q_max=600
#policy dcfabric efcp.*.dtcp cas-ps
```

```
#RED
policy dcfabric rmt red-ps qmax_p=600 qth_min_p=10 qth_max_p=30 Wlog_p=10
 Plog_p=9
policy dcfabric efcp.*.dtcp red-ps
```

The graphs below show the behaviour of the RMT queue at *r1* for the different policies. As before, PDU drop grows linearly with time in case of the default policies, while it keeps stable and with zero PDU drop in the case of the CAS and RED policy sets.

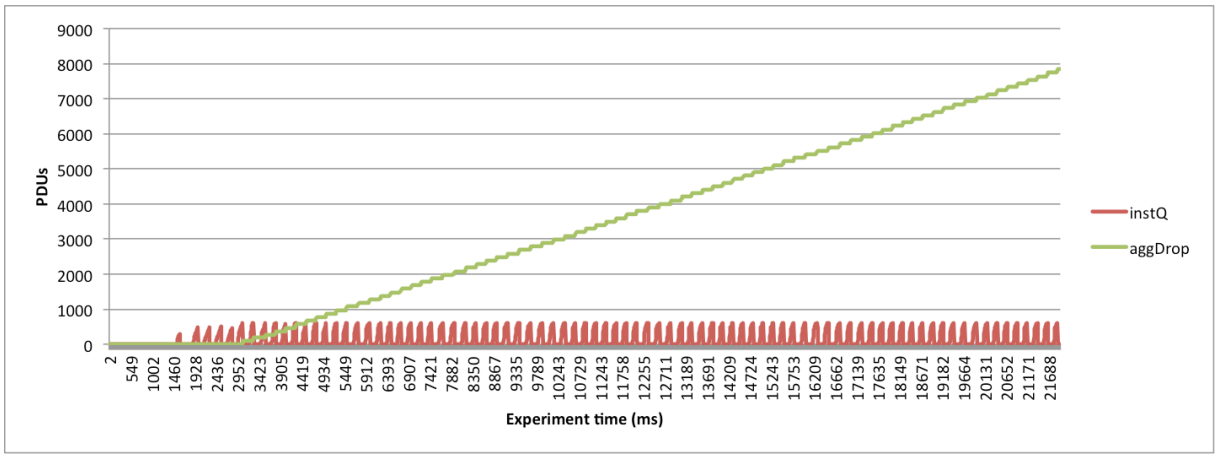


Figure 96. RMT queue length and accumulated dropped PDUs vs. experiment time, default policy-set

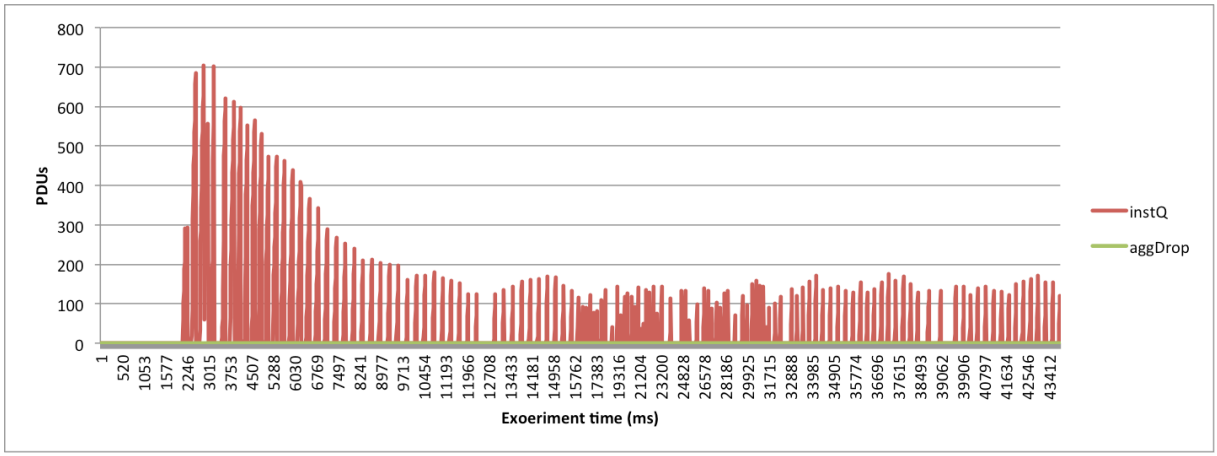


Figure 97. RMT queue length and accumulated dropped PDUs vs. experiment time, RED policy-set

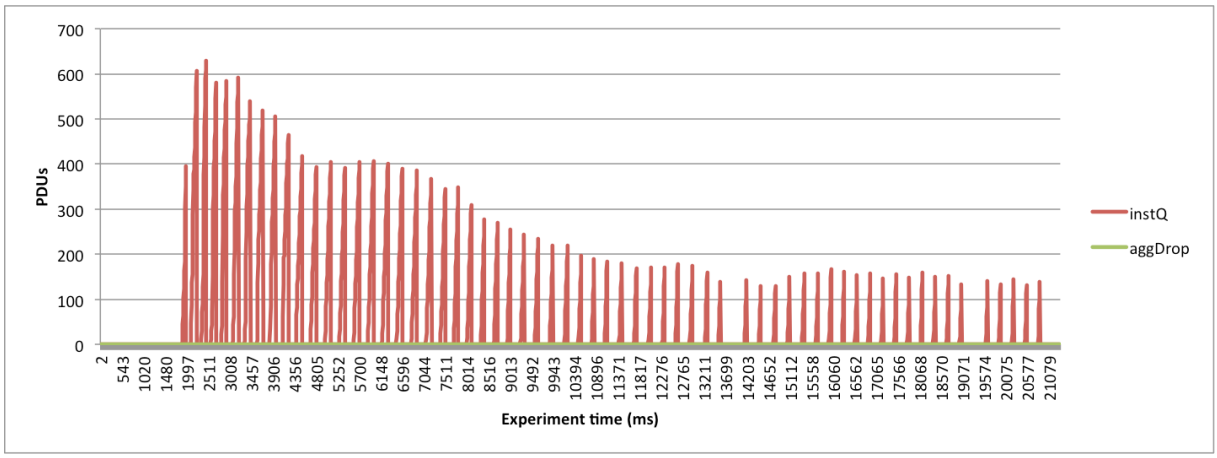


Figure 98. RMT queue length and accumulated dropped PDUs vs. experiment time, CAS policy-set

Flow throughput is shown in the graphs below, showing similar behaviour for the RED and CAS policy sets as before.

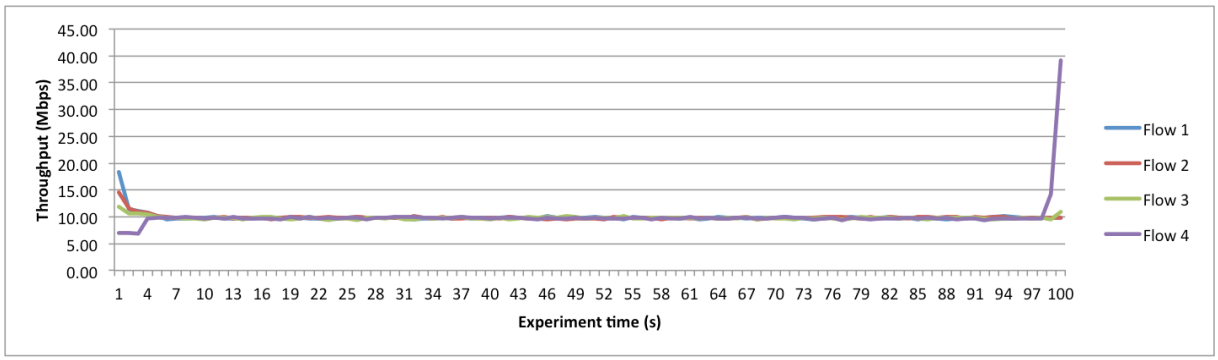


Figure 99. Throughput of the different EFCP flows vs. experiment time, default policy-set

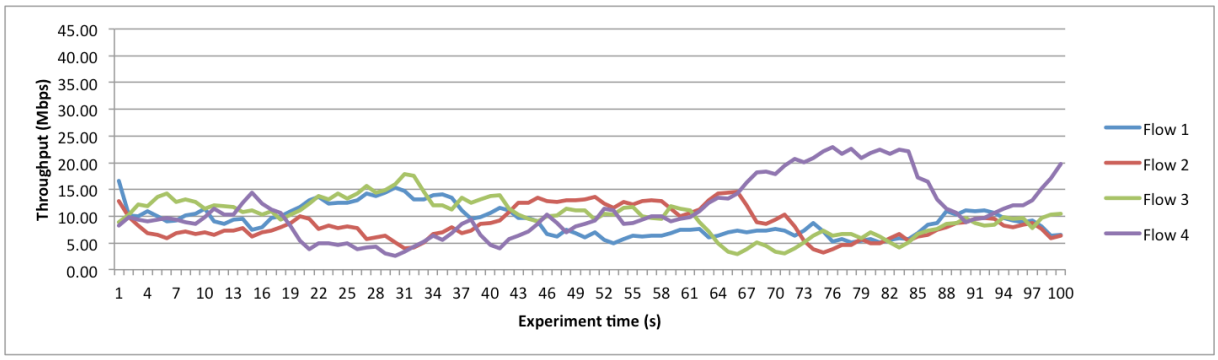


Figure 100. Throughput of the different EFCP flows vs. experiment time, RED policy-set

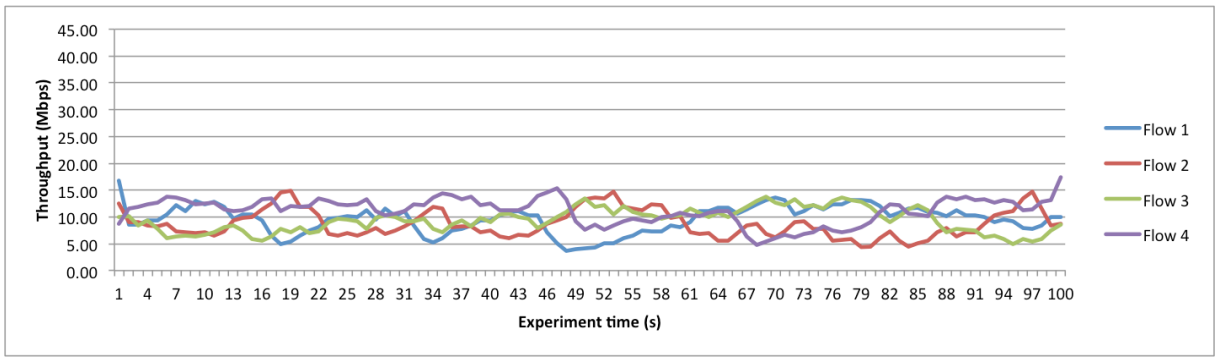


Figure 101. Throughput of the different EFCP flows vs. experiment time, CAS policy-set

This experiment has shown that the RINA implementation with the PRISTINE SDK provides a very good vehicle for experimenting with different congestion management policies within a DIF. Different controllers and RMT marking policies can be plugged in and out, mixed and matched while keeping all the other EFCP and RMT code the same; thus facilitating the observation of the effects of the different policy sets and the comparison between them.

Experiment 2: DC fabric and VPN DIFs

Having verified the correct operation of the policies within one DIF, now we move to the datacentre scenario. The second experiment uses the VPN DIFs depicted in Figure 86 over the datacentre fabric DIF. The DC fabric DIF has been configured with multi-path forwarding policies and the RED and CAS congestion management policies (in different runs of the experiment).

One rina-tgen session is setup at each VPN DIF, which will result in traffic between different racks (since each VPN DIF connects together two servers of different racks). In each experiment we observe the queues of the N-1 ports of TORs facing the spines in the DC fabric, and in the throughput reported by the rina-tgen application. The results in the scenario with multipath forwarding policies are provided in the following graphs.

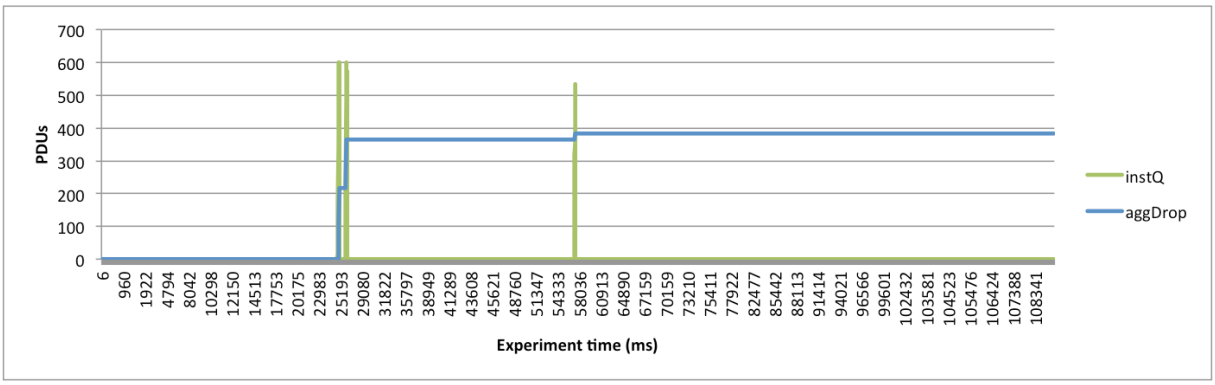


Figure 102. RMT queue length and accumulated dropped PDUs vs. experiment time, N-1 port 1, RED policy-set

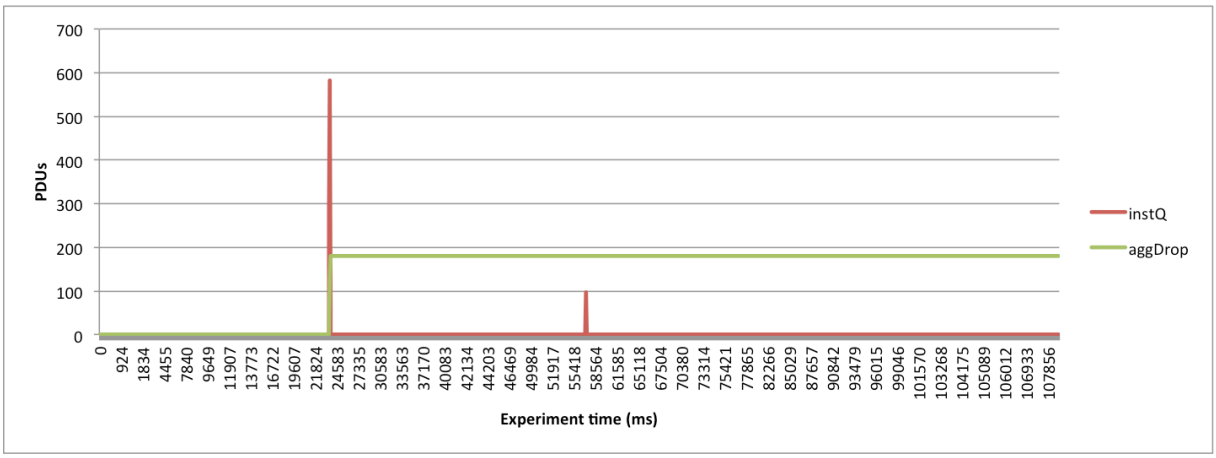


Figure 103. RMT queue length and accumulated dropped PDUs vs. experiment time, N-1 port 2, RED policy-set

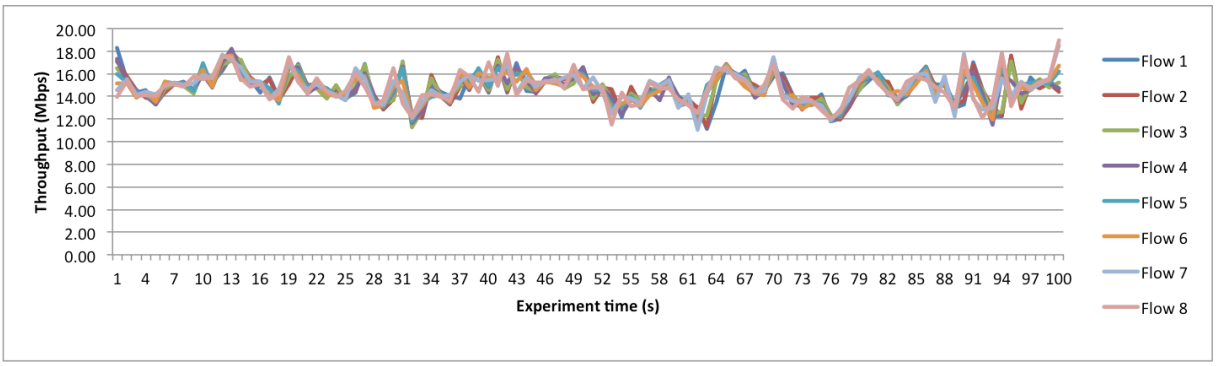


Figure 104. Throughput of the different EFCP flows vs. experiment time, RED policy-set

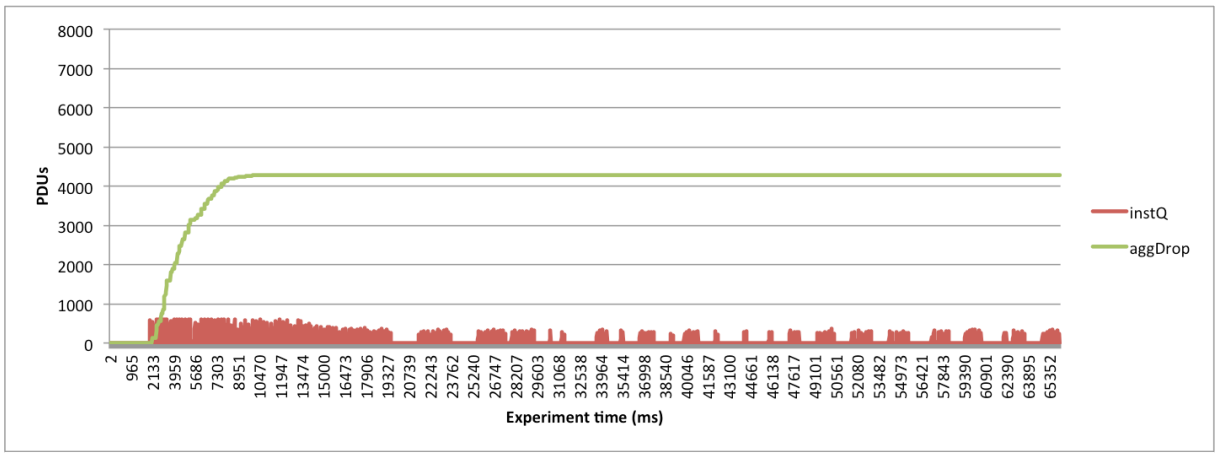


Figure 105. RMT queue length and accumulated dropped PDUs vs. experiment time, N-1 port 1, CAS policy-set

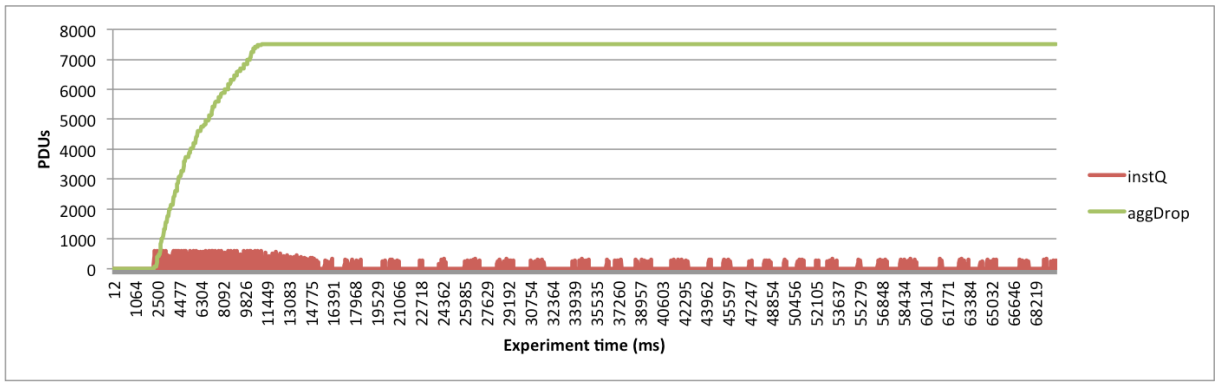


Figure 106. RMT queue length and accumulated dropped PDUs vs. experiment time, N-1 port 2, CAS policy-set

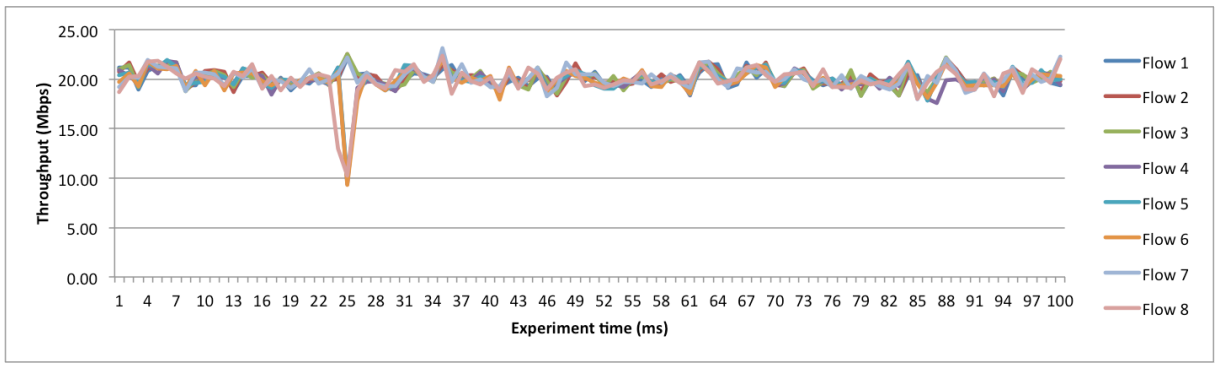


Figure 107. Throughput of the different EFCP flows vs. experiment time, CAS policy-set

Both congestion management policies keep the loss of PDUs in each of the N-1 port under control after a brief initial transient period. In this scenario the *RED* policy set worked faster generating less loss of PDUs than the *CAS* policy set. However, the throughput of the 8 flows is smoother and slightly higher in the *CAS* policy set case. Both *RED* and *CAS* policy sets achieve a similar degree of fairness in sharing the resources of the bottleneck between the different flows.

5.2. Network Service Provider: QoS-based multiplexing

The intensive use of network services and cloud computing by today's users requires that Network Service Providers be able to provide QoS guaranteed communications. This typically is done by overprovisioning resources, a strategy costly in economic terms and ineffective in regards of increasing requirements. A better appproach is to take advantage of the RINA programmability and ease of configuration to support QoS guaranteed differentiated services over multiple layers. This is the scenario tackled in this section, using the QTA Multiplexer, QTAMux for short, implemented in the IRATI stack with PRISTINE's SDK, as presented in deliverable D3.3. It will be shown how SDU drops and delays can be reduced using the QTAMux, while keeping intact applications's goodput.

5.2.1. Experimental setup

For this scenario the setup shown in Figure 108 has been used. From bottom to top, there are the shim DIFs based on the shim Ethernet VLAN, the normal DIF with the QTAMux scheduling policies, and the DAF formed by the rina-echo-time servers and clients. The shim DIF 300 is used to regulate the maximum bandwidth, while the shim DIF 400 is the one used to overwhelm the resources in the normal DIF. With 100 Mbps the ability of the RMT in the ISP system's IPC Process to deliver the traffic arriving from server 2 is compromised, since the shim DIF 300 only can deliver 10 Mbps, and so the buffers available in the RMT can be exhausted and resources must be managed to avoid large number of drops and delays. This is the point where QTAMux steps in.

The QoS cubes profiles used in the experiment are shown in the Cherish/ Urgency matrix, Table 6.

Table 6. QoS cubes profiles

Cherish\Urgency	More urgent	Less urgent
More cherished	QoS 1	QoS 3
Less cherished	QoS 2	QoS 4

The application used to measure the drop rate, delay, delay variance, and goodput was rina-echo-time. Trials were carried out in the following way: for each QoS cube, one rina-echo-time client requested a flow, asking

for the particular QoS cube to be used. Preliminar trials showed that there was no great difference if instead of one application per QoS cube several applications were used. After 5000 SDUs were sent by each client, they stopped; the rina-echo-time server then reported the results, SDUs received, goodput, and average time between packets for each QoS cube, which gives a measure of the delays experienced by the SDUs. These results were then averaged between several trials.

For comparison purposes, trials with a standard QoS cube were performed. In this case, there is only one queue, with a fixed maximum number of SDUs to be stored. One this number is exceeded, the RMT starts dropping SDUs. The setup was similar to the one described above: in this case also, the application used was the rina-echo-time. In each trial 4 rina-echo-time clients, using the only QoS cube available, requested a flow and sent 5000 SDUs.

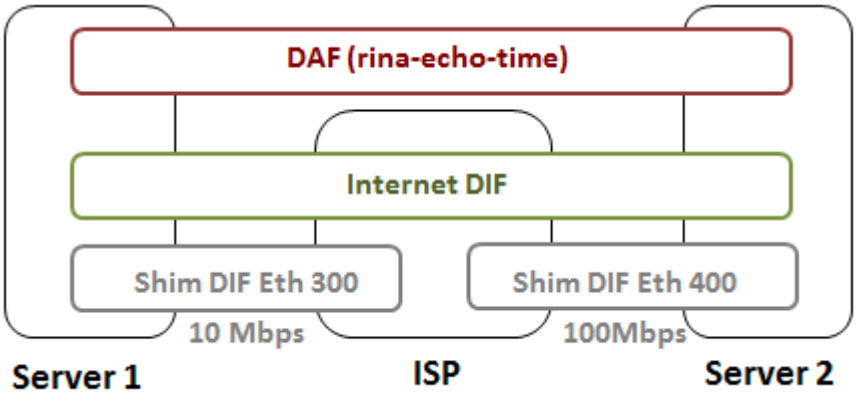


Figure 108. DIFs and DAF in the experimental setup in the Network Service Provider QTAMux experiment.

5.2.2. QTAMux and EFCP configuration

The QTAmux is made up of two main sets of components, the Policers/Shapers, and the Cherish-Urgency multiplexer, P/S and C/U-mux onwards. For this experiment, only the simplest P/S was used, with no shaping. Thus, only the C/U-mux need to be configured. Four traffic profiles were defined, with one QoS cube for each. These QoS cubes are grouped into 2 urgency levels, with 2 levels of cherish. The configuration of the QoS cubes can be found in Table 7:

Table 7. QoS configuration

QoS ID	Urgency class	Absolute threshold	Threshold	Drop probability	EFCP FC window size
1	1	200	140	5%	100
2	1	160	120	10%	80
3	2	130	90	15%	70
4	2	100	70	20%	60

Regarding the configuration of the EFCP, to avoid large amounts of SDUs dropped owing to initial credits too large, the maximum window size has been set up as the number of SDUs that could be accepted, with no drops, by each QoS cube. In Table 2 this corresponds to the values in column "EFCP FC window size". Thus, the configuration of the QoS cube in the .conf file should be like this:

```
"qosCubes" : [ {
"name" : "unreliablewithflowcontrol",
       "id" : 1,
       "partialDelivery" : false,
       "orderedDelivery" : true,
       "efcpPolicies" : {
            "dtpPolicySet" : {
              "name" : "default",
              "version" : "0"
            },
            "initialATimer" : 0,
            "dtcpPresent" : true,
            "dtcpConfiguration" : {
                 "dtcpPolicySet" : {
                   "name" : "default",
                   "version" : "0"
                 },
                 "rtxControl" : false,
                 "flowControl" : true,
                 "flowControlConfig" : {
                     "rateBased" : false,
                     "windowBased" : true,
                     "windowBasedConfig" : {
                       "maxClosedWindowQueueLength" : 10,
                       "initialCredit" : 100
                      }
                 }
            }
```

```
        }
]
```

This is the definition for the number QoS cube 1, the most urgent and most cherished. For the other QoS cubes, the only parameters that need change are the "id", which takes the value as in Table 2's column "QoS ID", and the "initialCredit", which should take the value according to the column "EFCP FC window size" in the same table.

The configuration of the QTAMUx policy set should be like this:

```
"rmtConfiguration" : {
    "policySet" : {
        "name" : "cher-urg-ps",
        "version" : "1",
        "parameters" : [{
              "name"  : "1.urgency-class",
              "value" : "1"
           },{
              "name"  : "1.drop-prob",
              "value" : "5"
           },{
              "name"  : "1.abs-th",
              "value" : "200"
           },{
              "name"  : "1.th",
              "value" : "140"
           },{
              "name"  : "2.urgency-class",
              "value" : "1"
           },{
              "name"  : "2.drop-prob",
              "value" : "10"
           },{
              "name"  : "2.abs-th",
              "value" : "160"
           },{
              "name"  : "2.th",
              "value" : "120"
           },{
              "name"  : "3.urgency-class",
              "value" : "2"
           },{
              "name"  : "3.drop-prob",
              "value" : "15"
           },{
```

```
            "name"  : "3.abs-th",
            "value" : "130"
         },{
            "name"  : "3.th",
            "value" : "90"
         },{
            "name"  : "4.urgency-class",
            "value" : "2"
         },{
            "name"  : "4.drop-prob",
            "value" : "20"
         },{
            "name"  : "4.abs-th",
            "value" : "100"
         },{
            "name"  : "4.th",
            "value" : "70"
         }]
      }
   }
```

For comparisons, as explained above, a simple RMT with a single, finite queue was used. The maximum length of this queue was chosen taking into account that in the trials carried out for this scenario the maximum value of the occupation of the RMT's queues was 170 (there is an example below of the occupation as a function of time). This was the value of the maximum queue length used when there was no QTAMux. Also, regarding the EFCP credit, the number of buffers used in the QTAMux policy were evenly assigned to the applications. Thus, 320 buffers in the EFCP for 4 applications meant that the credit for each application was 80.

5.2.3. Results

Figure 109 shows the average number of drops per QoS cube and the average number of drops in the case of no QTAMux. It shows how the use of no queueing policy in the RMT, only setting up a maximum queue size, even if it is quite large, leads to a number of drops that is not statistically different for the case with no QTAMux, a best effort QoS cube.

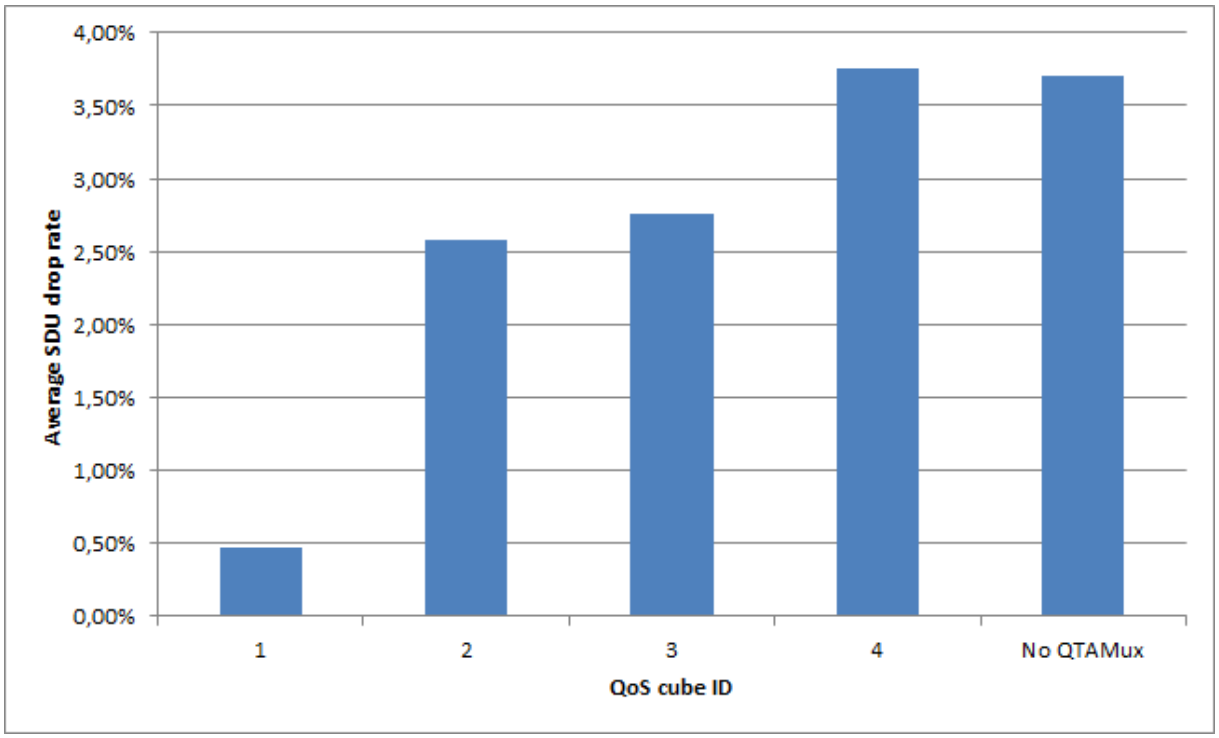


Figure 109. Average drop rate per QoS cube

Figure 110 shows the average delay for each QoS cube and, for comparison purposes, the case with no QTAMux. It can be seen how the average delay is lower for the more urgent and more cherished flows, while being comparable the average delays for the less urgent and less cherished to the case of no QTAMux, using a best effort QoS.

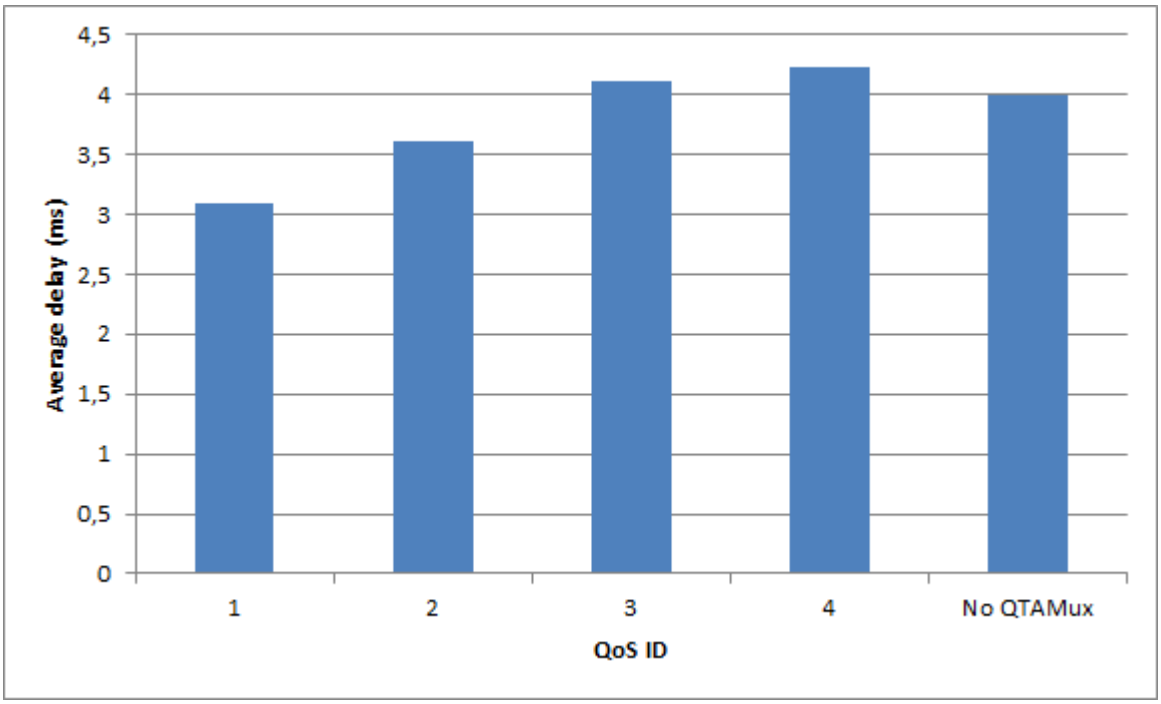


Figure 110. Average delay per QoS cube

In Figure 111, the delay's variance for each QoS cube is shown. As expected, the variance for the most cherished QoS cubes is lower than in the other cases, and much lower than for the case with no QTAMux.

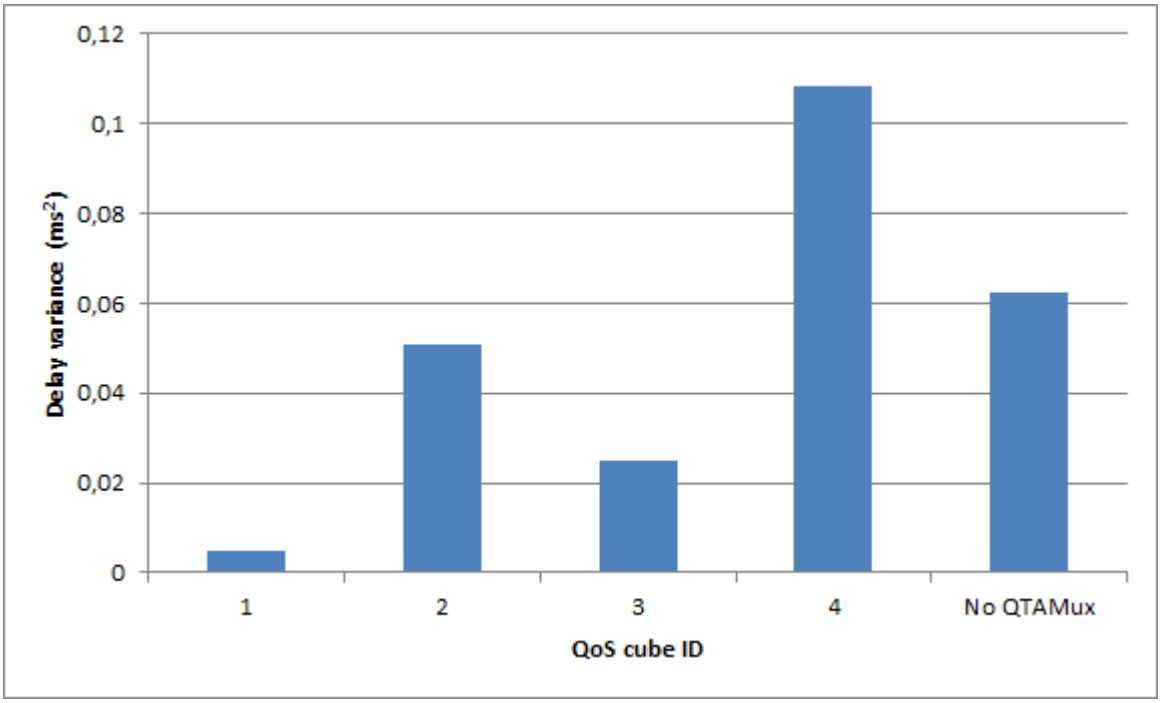


Figure 111. Delay variance per QoS cube

Figure 112 shows the goodput per QoS cube and for the case with no QTAMux. This figure is intended to show that the lower delays obtained with the QTAMux are not damaging the goodput. Indeed, the QTAMux allows for better goodputs than in the best effort case, with no QTAMux.

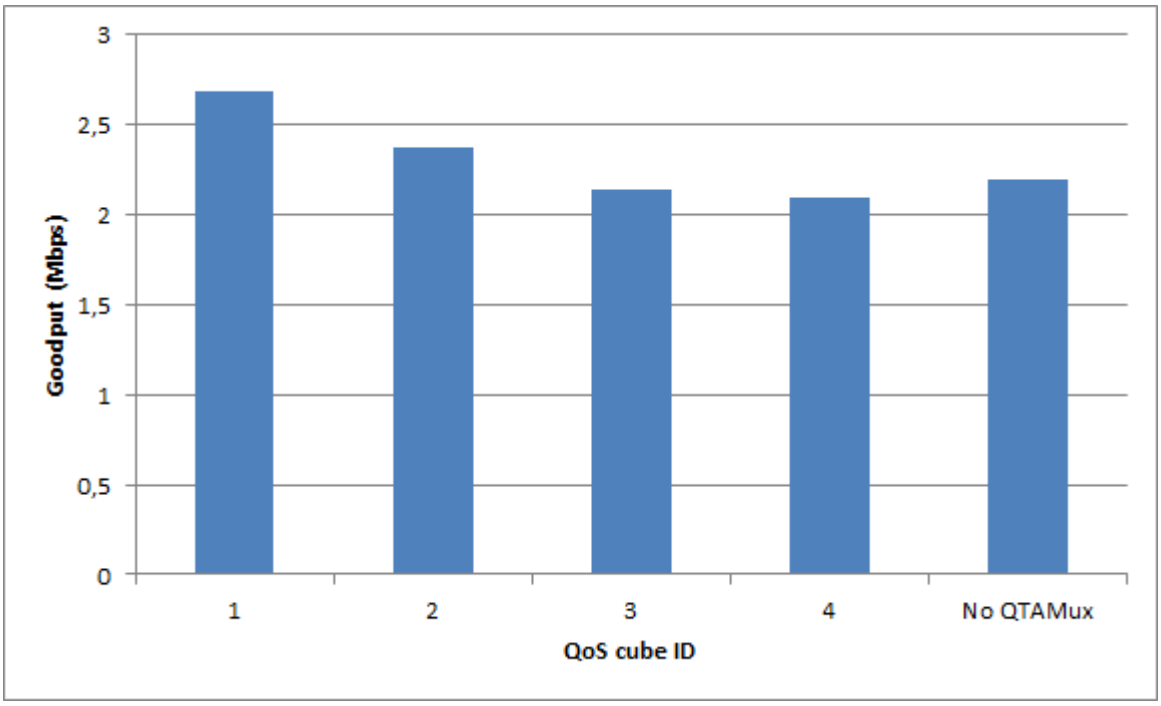


Figure 112. Average goodput per QoS cube

Figure 113 shows an example of the occupation of the different QoS cubes' queues as a function of time. The spikes in the occupation are due to the coupling of the limited capacity of the shims layer with the extension of credit by the receiver EFCP. The P/S module, not used in this scenario, would probably have limited these sudden rises in the occupation of the RMT's queues. This topic will be investigated in future work.

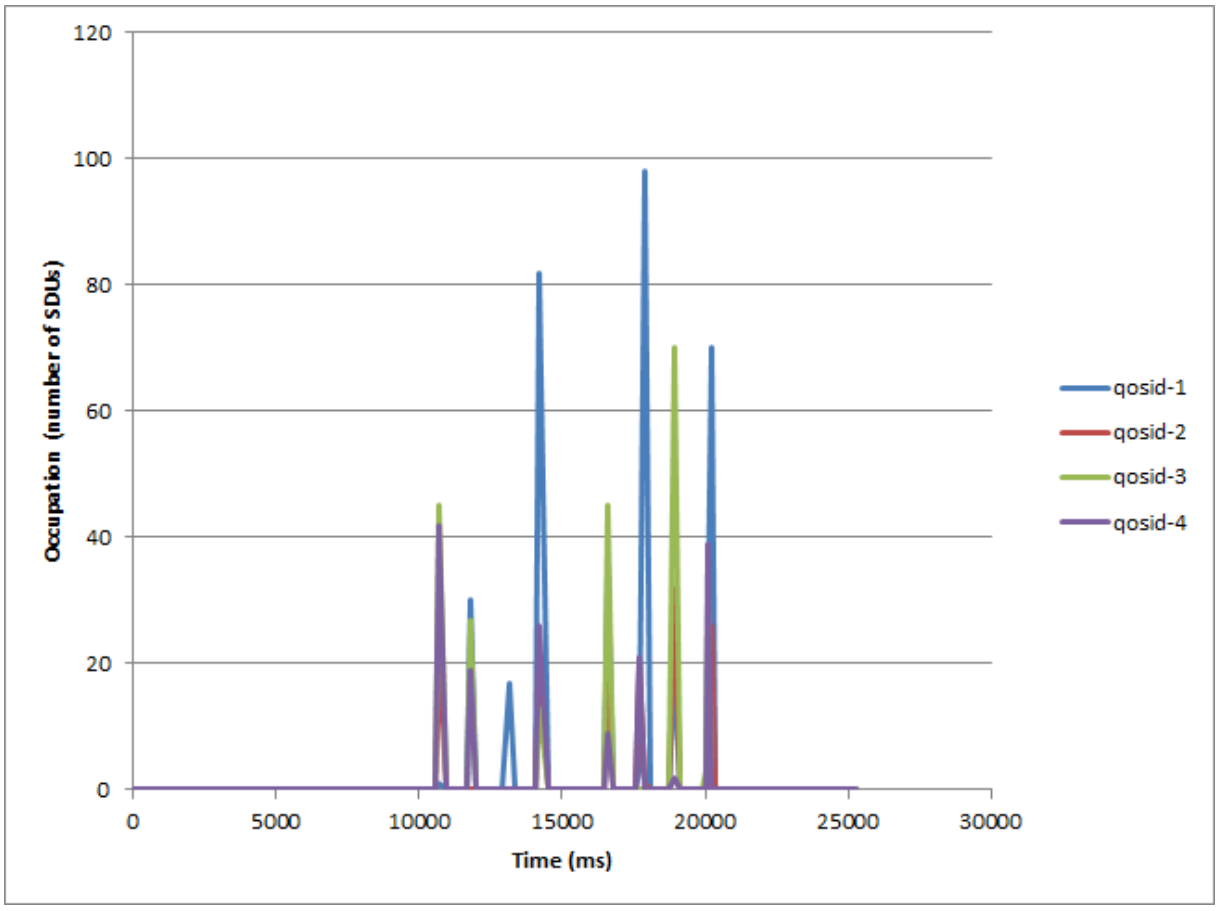


Figure 113. Occupation per QoS id as a function of time

5.3. Network Service Provider: NFV and Service Chaining

5.3.1. Introduction

What we want to show in this demo is the possibility to join the pros of IP network technologies with the newborn RINA ones; we do this in order to use mature software, which already has years of development and testing, with the flexibility and organization offered by the RINA network architecture. We are aiming to have IP applications running on the top of a RINA-organized network in a transparent way, such as top level user applications will not realize the changes which takes effect in the underlay network.

This is possible because RINA layered view can extends also to other kind of technologies; in our case the IP network will form the top-most layer used by Application Entities to exchange data. Since RINA layers allow to abstract over the type of traffic moving on every DIF, IP does not realize that RINA is moving its traffic under it, and on its side RINA network does not care about the type of bytes which are flowing through

it (see Figure 114). The only entities which realize the full situation are the Applications which provides the joint points between the two technologies.

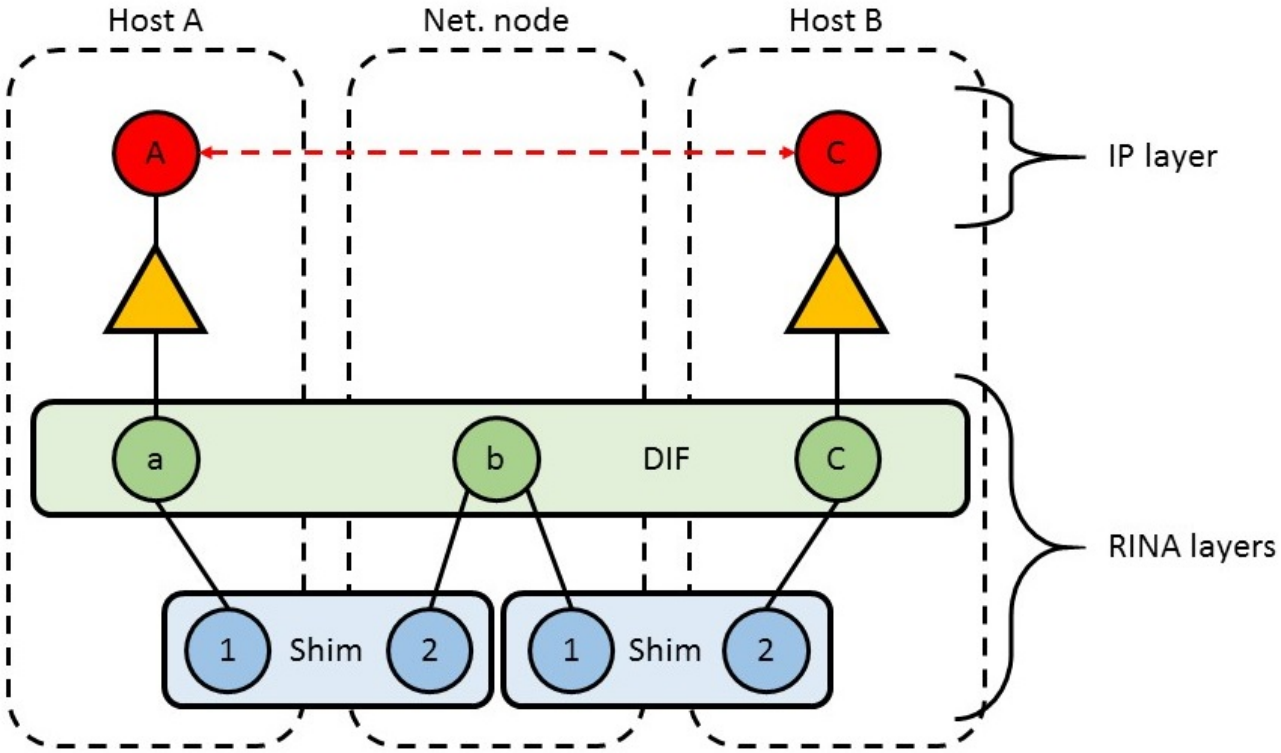


Figure 114. General layer organization in the demo. The yellow triangle is NORI software which provides the necessary service to move IP traffic over RINA.

The NORI tools (NFV over RINA) will be the central actor of this demo, and will provides the necessary elements to create such organization of IP over RINA. This allows us to use legacy IP software without any modification of their source code; you will see later in this document that only a configuration is necessary (as it is in their use over the simple IP), whereas such software usually need to use some IP address and TCP/UDP port.

5.3.2. Configuration

The demo will be setup on a five nodes topology: one will be the machine running outside the desired data center, while the remaining nodes will form what we consider the internal (data center point of view) network topology. The node which is running 'outside' will use a standard configuration with absolutely no RINA technology, and will connect remotely to the data center using clean IP services.

The border router node, which is the entry point of the communication, will process the incoming information and route them into the RINA

network, which provides the backbone for the inter-DC communication to reach other nodes in the topology.

In this particular demo the service chaining will be provided by VLC streaming instances. The provider will offers a video streaming service, and wants to differentiate the given service between free and premium users. Nothing changes on the user side, since the service will differentiate it in the data center. When the 'free' user (or the lowest paying one) connects to the streaming server it sees the video which is transcoded with a tag for the company advertising. When the remote user is classified as premium one, it is redirected on the premium servers which have no restriction nor advertise tag which limit the view of the content. This setup is shown by Figure 115.

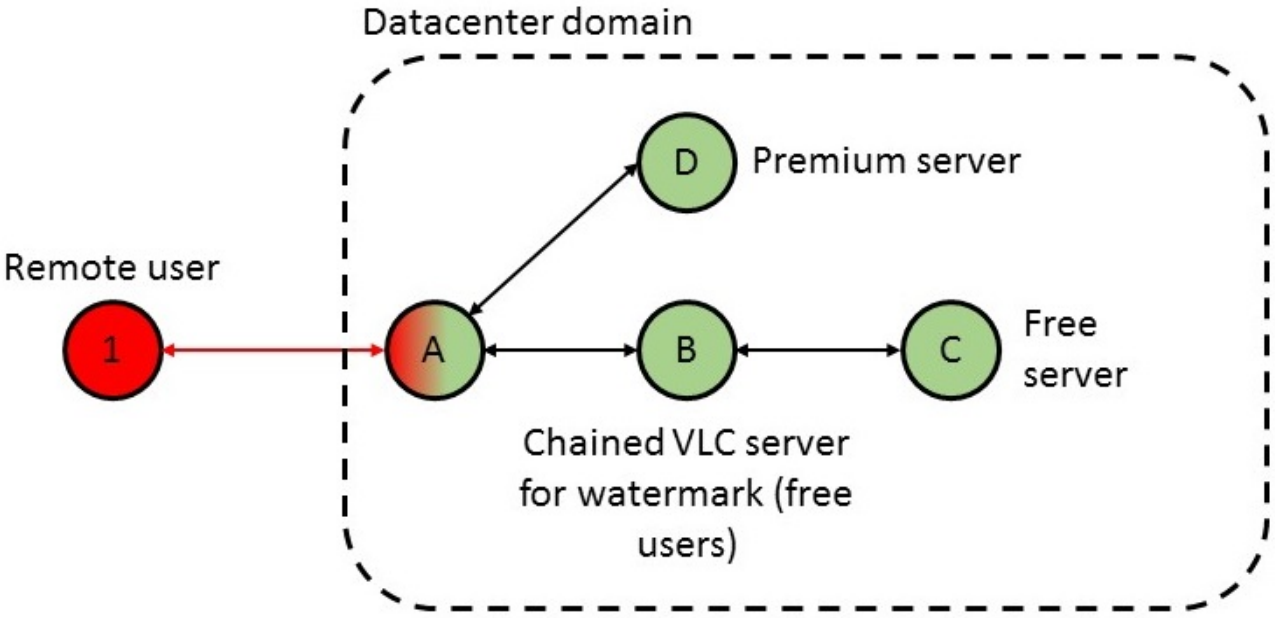


Figure 115. Setup of the Demo; red indicates pure IP communication while green indicates that data is handled by RINA.

Note that NORI allows us to create arbitrary long chains which can expose also different kind of functions, as for example a firewall of some sort, without having to modify anything in the IP software to adapt it to a new kind of technology (which is hidden at that level).

5.3.3. Restriction on the software

In this demo we are using VLC to provide an easy-to-setup demo with a basic service chaining, where an intermediate streaming server gets the

input from another streaming server. Take in account thus that, since we are operating with generic IP traffic on the "bottom" of an network interface, we have **no restriction** on the type of software we can run on the top. Your imagination is the only limit here.

You can setup other kind of software, which can be a set of rules using the `iptables` tools offered from Linux, or a more extensive firewall application which analyzes all the traffic flowing from that network device. The network device itself can be the outgoing/incoming traffic device for a Virtual Machine instance, which can offer a sandboxed and modular NFV software for your system. By handling the traffic without any additional assumption, we are able to widen a lot the range of solutions that can be built with this tool… and again: such modifications are invisible to the IP upper layer (depending on how fast you change you can have minimal/no service disruption) since the modifications are done on the lower layers.

We can also consider to apply different congestion control mechanism alongside with TCP and UDP, since we are controlling it from the bottom (and transparently). Every of such possibility while maintaining legacy IP software already developed by big internetworking companies.

5.3.4. Experiment setup

This scenario is run on the vWall testbed offered by iMinds, but can be easily adapted to work on a local scenario. There are some basic preparations that we assume have been already made before running this demo, which are:

- Machines of the service provider domain are already prepared with Pristine-1.5 branch of IRATI stack. Both user space and kernel space components must be installed and tested as working (using one of the user space tools already shipped with IRATI repository, like rina-echo-time).
- Configuration files for a simple setup of a single DIF over shims must be already in place.

Nodes 1, which represent the user, does not require any particular configuration nor IRATI stack installed; any distribution of Linux/ Windows can be used. OpenVPN is the only requirement (if you, like us, don’t have IPv6 support to reach vWall testbed, but this is a requirements

you don't have in other configurations) that such user equipment must have in order to reach the service. In order to use the offered streamed service we assume VLC (VideoLAN) player software is used.

As the first step nodes which provides the service (green nodes in Figure 115) must be enrolled with each other. This will create the Service Provider personal network which will handle packets in its own way (applying the most desired policy).

5.3.5. NORI setup

As the second step you must run NORI instances. This creates in the nodes which run it the additional virtual interface used for the communication between nodes over RINA (using IP). You will have to run the program on every node inside the vWall in order to enable the communication.

Every NORI application will have its own configuration in order to move packets to the right endpoint. This configuration will list every rule applied during the packet transaction, and can bound some characteristics of the packet flow to a specific endpoint. For node A this configuration will initially redirect everything on the Watermarking streaming server with:

Node A NORI command line:

```
./nori s1 1 dc s1.dict.txt
```

Node A dictionary:

```
default si s2,1
```

For node B the configuration must move every packet to the source stream service C, but also must allow eventual feedback to the service user connecting to the streaming service. This means that every packet with destination 192.168.200.1 must be shipped back to node A, while every other communication is isolated between node B and C.

Node B NORI command line:

```
./nori s2 1 dc s2.dict.txt
```

Node B dictionary:

```
ip dst 192.168.200.1 s1,1
default si s3,1
```

For node C the configuration is simple too and maintains the isolation between B and C. Any traffic, independently from any field in the packet headers, is redirected back to B.

Node C NORI command line:

```
./nori s3 1 dc s3.dict.txt
```

Node C dictionary:

```
default si s2,1
```

Node D hosts another streaming server which has the same characteristics as node C. This server has no traffic redirected to any filter and will give its feedback directly to the source of the communication, allowing a different path (with fewer hops) for the streaming services. Different QoS at the RINA level can be further configured in order to give different priorities to the traffic. The configuration of such node is as follows:

Node D NORI command line:

```
./nori s4 1 dc s4.dict.txt
```

Node D dictionary:

```
default si s1,1
```

At the end of this configuration you will have 4 instances of NORI serving 4 different IP devices which need to be configured in order to work correctly.

5.3.6. NORI IP interfaces setup

Since we are working with different domains (IP on the top while RINA on the bottom), we must ensure the coherency of the addresses used on both levels (see Figure 116). If random IP addresses are used, the underlay

RINA layer will still deliver the packet to the right node, but such traffic won't be recognized by the upper IP interface. As we pointed out the IP stack will be maintained untouched and fully functional, but the transport will be performed by RINA.

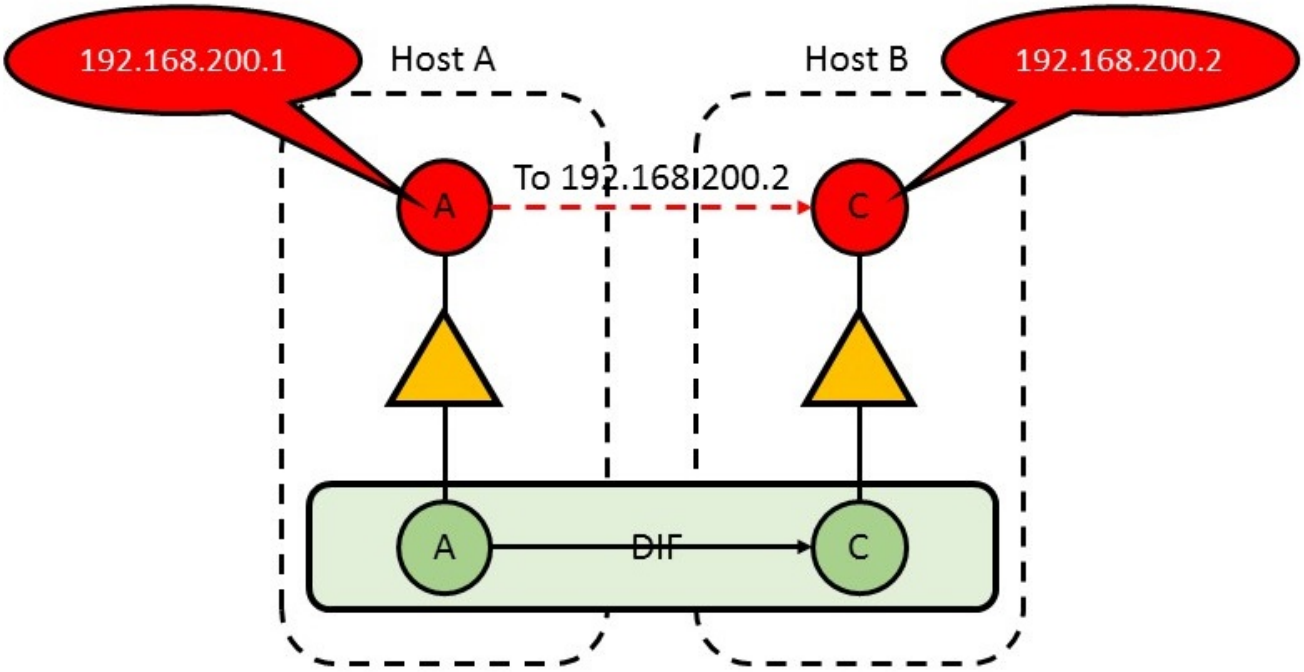


Figure 116. Not only RINA, but also IP addresses must be choosen in order for them to consider packets as valid.

Every node must run the following commands in order to correctly configure the IP interfaces:

```
sudo ifconfig tun0 192.168.200.x netmask 255.255.255.0
sudo ifconfig tun0 mtu 1400
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
```

IP forwarding must be enabled on the interface, in order for the IP layer to forward the packet to the right destination if the current one is not the receiver of the packet. This will allow the network device to process the packet and immediately forward it, giving the opportunity to NORI to perform again the next step and deliver the data to the next element in the chain. The MTU of the interface must be 1400 bytes, in order to leave some space for the additional headers (RINA ones) that will be append in front of the IP header without having the Shim over Ethernet reject them because they are too large. Finally every network interface must have a compatible IP.

The IP addresses(see Figure 117), node per node, are:

- Node A, `192.168.200.1`
- Node B, `192.168.200.2`
- Node C, `192.168.200.3`
- Node D, `192.168.200.2`

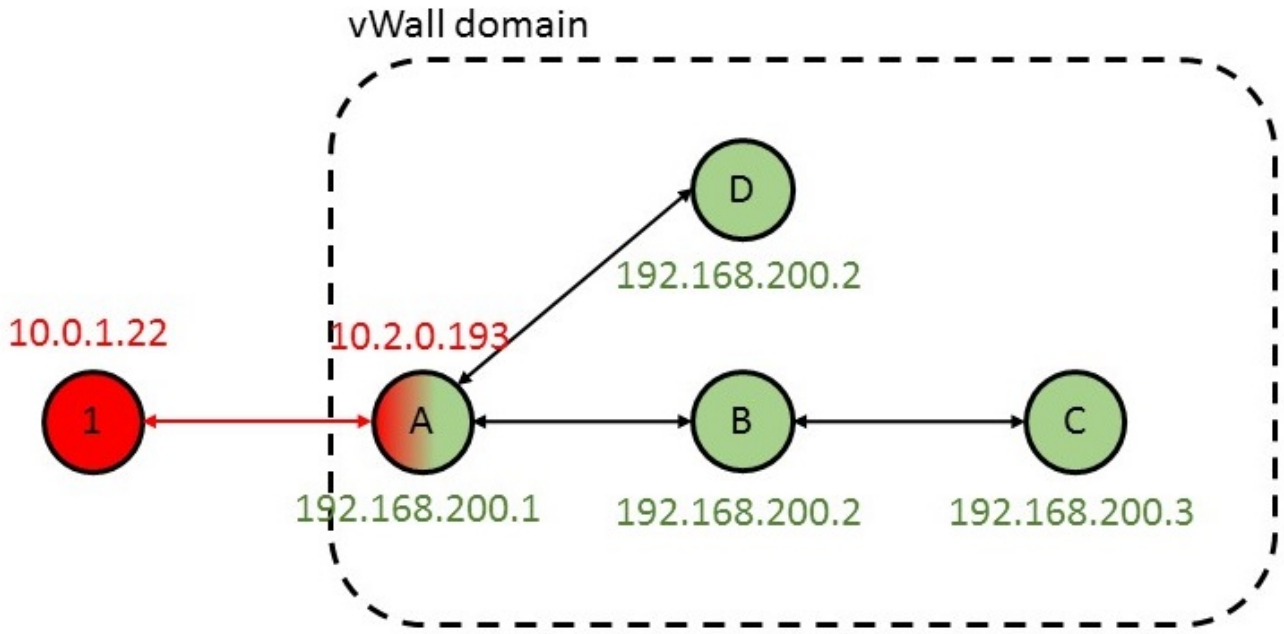


Figure 117. IP addresses assigned to every node; red are interfaces on the outer network which are purely IP, while green ones are network interfaces served by RINA.

As you can notice, nodes B and D have the same IP. This causes no conflicts in this setup, since NORI allows them to both coexist without interference (since it's NORI which decide where to send the packet). The IP configuration can maintain always the same target, but under the surface the service can be provided by several different physical machines.

5.3.7. Legacy IP to IP over RINA configuration

The last configuration instructs the IP stack to redirect traffic on IP over the RINA network, and takes place in node A, which is the "router" machine of the experiment. Traffic incoming from the 'outside' area must be redirected to the internal network, while traffic emerging from the internal network must be redirected 'outside' to the right. This is a kind of NATting service which is necessary for this setup to route traffic coming from the

command interface of vWall node to the internal network, which is usually isolated in order to have a clean testing network.

This configuration can be skipped if you have your own local testbed.

The first command is used to redirect all the incoming traffic for port 5001 by changing its destination before applying the routing decisions. This will move the traffic from whatever network to the internal one.

```
iptables -t nat -A PREROUTING -p tcp --dport 5001 -j DNAT --to-destination
 192.168.200.2
```

After the routing decision is taken, we need the destination to be able to give feedback and data to the source node. To do this we change its source IP as follows:

```
iptables -t nat -A POSTROUTING -p tcp --dport 5001 -j SNAT --to-source
 192.168.200.1
```

In the end, since we have only one client node, we edit all the traffic from address 192.168.200.2 (as we did for the first step) in order to be routed to the outside on the global network until our client instance. This rule can be applied also by targeting all the traffic with source port 5001.

```
iptables -t nat -A PREROUTING -s 192.168.200.2 -j DNAT --to-destination
 10.0.1.22
iptables -t nat -A POSTROUTING -s 192.168.200.2 -j SNAT --to-source
 10.2.0.193
```

5.3.8. Running the experiment

Now that everything is configured and ready to be used, we just have to start the streaming services. In order to do this nodes B, C and D must have VLC software installed. For such machines is not necessary to have an X server, since the server instances will be run using the terminal.

To start the C streaming service you have to issue on node C:

```
vlc <path_to_video> --sout
 '#standard{access=http,mux=ogg,dst=192.168.200.3:1212}' --loop
```

This will start an http streaming service on port 1212. This streaming service will be used by the VLC instance of node B as a source; an additional filter will be attached to it which adds a watermark in order to advertise whatever during the free access service (and will act as a visible mark of what NORI strategy is actually in place). This flow can also be configured with lower priority and lower bandwidth (using the desired DTCP policy), if desired.

To start the B streaming service you have to issue on node B:

```
vlc --logo-file="path_to_logo" http://192.168.200.3:1212 --sout
 '#transcode{vcodec=mp2v,sfilter=logo}:std{access=http,mux=ogg,dst=192.168.200.2:5001}'
 --loop
```

This will start the streaming service with a watermark filter which take a picture as a logo, and place it in the top-left corner of the video itself. This service is offered on port 5001, the one the connecting clients look for. Starting a VLC instance on node 1 and connecting to the streaming service (from the outside network the target is 10.2.0.193:5001) will result in seeing the desired video with a watermark applied on the stream.

To start the D streaming service you have to issue on node D:

```
vlc ~/m84_2.mpg --sout
 '#standard{access=http,mux=ogg,dst=192.168.200.2:5001}' --loop
```

This will start the streaming service at it is, without any filter in the middle.

5.3.9. Redirecting the traffic

Now you can open VLC (from the client node 1) and connect to a network stream using the `http://10.2.0.193:5001` address. This will let you show the desired video with a watermark on it. We can then show what happens if, for some reason, the client source address becomes "trusted" and gains access to a non-watermarked service; note that also the quality of the connection can change, and obtain a higher rate, or be redirected to more powerful servers. In order to operate the switch, since the actual version of NORI does not allow rules modification while running, you have to kill NORI on the node A and launch it again with an alternative rules set.

```
./nori s1 1 dc /nori/s1.alt.dic.txt &
sudo ifconfig tun0 192.168.200.1 netmask 255.255.255.0
```

```
sudo ifconfig tun0 mtu 1000
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
```

Where the alternative dictionary rules are:

```
default si s4,1
```

This will route all the traffic on the alternative server on node D. Note that no other reconfiguration must be done, since node D already has a valid IP.

5.3.10. Secure your layer

RINA architecture allows applying abstraction recursively over different layers with different scopes. It means that, for transmission of your data, usually you are dependent on another lower IPCP (unless you are the lowest one, which usually has a restricted scope). Since you might not have control over lower layers, it can mean that your data can move along insecure channels, exposing your network and your machines to external packet inspection.

To avoid such attacks, the data flow between the nodes of your DIF needs to be protected, usually using cryptographic methods. In RINA these are provided by the SDU Protection module, supported by an Authentication policy module that ensures the identity of the nodes joining a DIF and negotiates parameters used by SDU Protection. The SDU Protection module present in the IRATI stack supports cryptographic mechanisms based on the DTLS Record protocol. In combination with an Authentication policy, we can secure our DIF against attacks from lower layers.

Both SDU Protection and Authentication can be enabled by changing the configuration of the SecurityManager, which is a part of the IPCM. In this demo we used the TLS Authentication policy, configured to use self-signed X.509 certificates. In a real-world scenario, these can be replaced with certificates using the already existing certificate authority system if necessary, but self-signed certificates are sufficient for the purposes of this demo. The creation of a suitable self-signed certificate is described in D4.3[34]. Once done, the certificates need to be distributed to individual

[34] https://wiki.ict-pristine.eu/wp4/d43/d43-authentication

nodes of the DIF where we can change the IPCM configuration accordingly. Next, SDU Protection was configured to use the following cryptographic methods and algorithms:

- **compression** using the **DEFLATE** algorithm
- **message authentication** using the **SHA256** hash algorithm for a **HMAC** signature
- **encryption** using the **AES256** algorithm

With this, we've fully protected our demo DIF against attacks from lower layers. A possible further optimization of the security configuration would be skip cryptography for data transmitted over trusted lower layers and only secure data that leaves our trusted infrastructure. The reason behind this could be if we wanted to avoid performance loss of using cryptographic functions. For more information about the SDU Protection module, its implementation and configuration you can take a look at D4.3[35].

5.4. Network Service Provider: Integrated Security

5.4.1. Goals

The goal of this experiment is to provide a practical demonstration of the security properties provided by the RINA structure applied to the network of a service provider, as well as showing how each individual DIF that is part of the network can be configured with the appropriate authentication, access control and SDU protection policies to prevent the network from being compromised.

[35] https://wiki.ict-pristine.eu/wp4/d43/d43-encryption

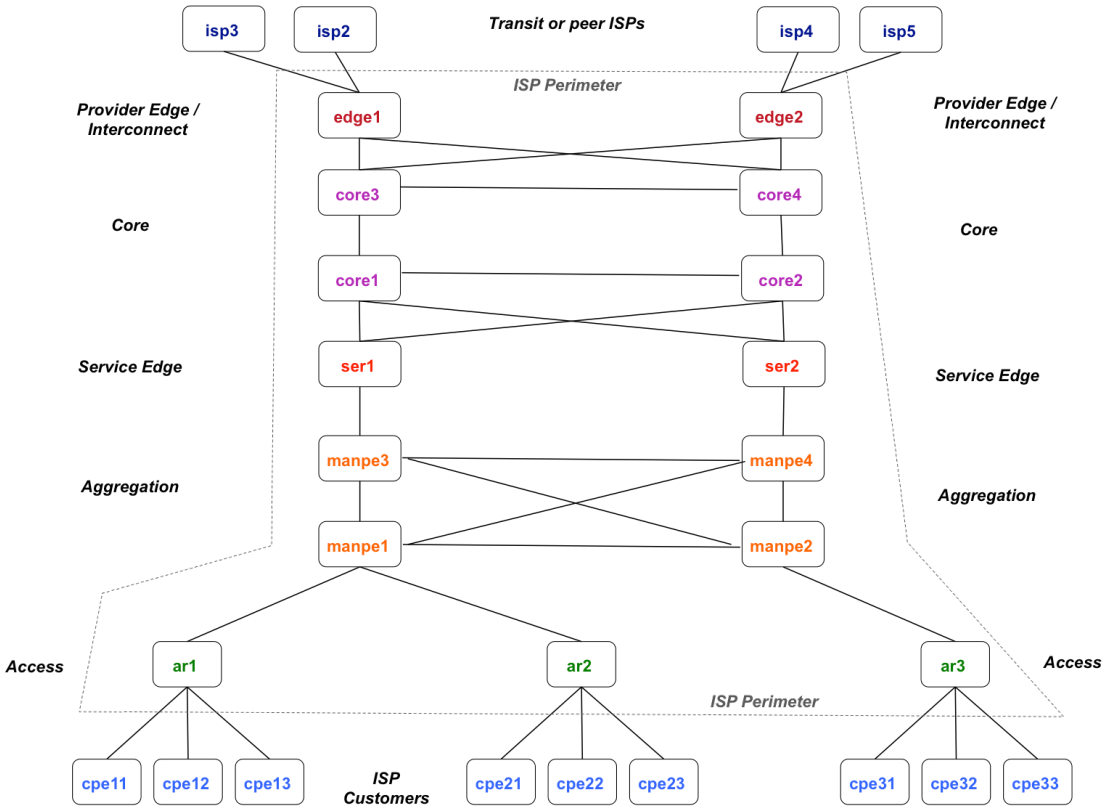


Figure 118. Physical layout of the systems in the ISP security experiment

Figure 118 shows the physical layout of the different systems present in the experiment. The service provider network is divided into different tiers: access (providing connectivity to customer CPEs), aggregation, service edge (to allow customers to connect to different services), core and provider edge (connecting to peer or transit providers). These systems implement a number of DIFs that follow the structure shown in Figure 119. E-mall DIFs provide customers with access to different types of applications; e-malls can be very generic such as an "Internet e-mall" or highly specialised to serve specific applications or providing isolated connectivity domains such as virtual private networks. E-mall DIFs may be offered by a single network service provider in isolation (partnering with CDNs or content providers) or via different providers who work together, such as the example e-mall DIFs shown in the Figure.

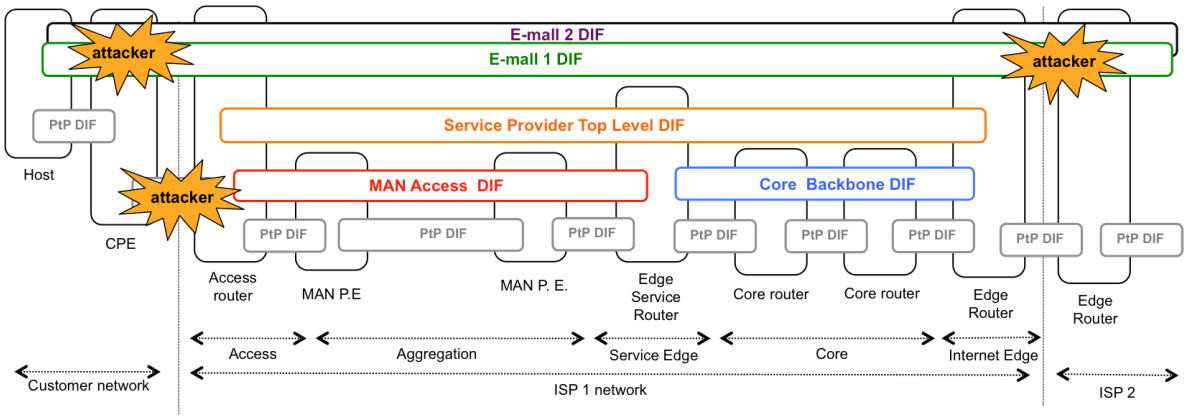


Figure 119. DIFs in the ISP security experiment: DIF stacking (side view)

Customer CPEs only have access to e-mall DIFs, provided via access DIFs that may be point to point (in the case of a wired DIF over copper or fibre) or multi-point (in the case of radio access DIFs). Therefore, attacks to the service provider infrastructure can only come from a rogue CPE (trying to spoof the identify of a valid customer) or from another system that can participate in the access DIF (by tapping a cable or scanning the radio spectrum). The service provider's internal network is hidden from customers or other providers; e-mall DIFs just have two hops over the provider network: access routers and provider edge routers.

Therefore in order to protect its network the service provider has to protect its perimeter, focusing on the systems and layers that are exposed to interactions with customers or other providers. RINA's recursive layering minimizes the attack surface of the ISP network, allowing the provider to expose very few systems and layers to the external world. For an external actor to access any information about the provider's internal network it would have to remotely compromise a provider's access router or a provider's edge router (here we are assuming that physical access to the provider's premises cannot be compromised; it is not part of the threat model).

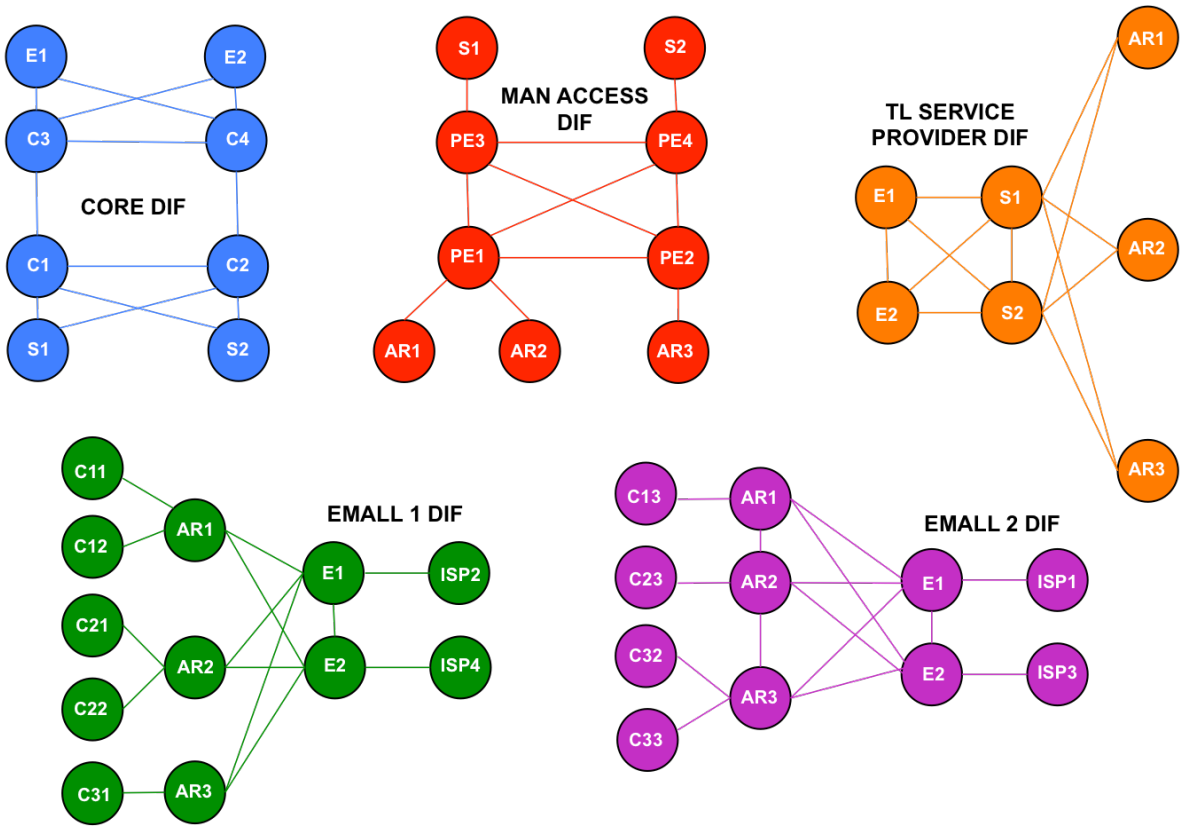


Figure 120. DIFs in the ISP security experiment: detailed view of each DIF

Figure 120 shows the detailed connectivity graph of the different DIFs in the service provider network. The top-level service DIF spans the whole ISP network, providing flows between access and edge routers that support the emall DIFs. Metro DIFs aggregate the traffic of several access routers into core Point of Presence locations, while core DIFs provide connectivity and performance between core PoPs and edge routers. The following bullet points summarize the main threats and countermeasures considered in this experiment:

- Unauthorized customer tries to join e-mall DIF → Cryptographic mutual authentication required to join e-mall DIF.
- System may get access to data sent over access DIF → Encrypt PDUs sent over access DIFs.
- System may tamper PDUs sent over access DIF → Apply HMAC code on PDUs sent over access DIFs.
- Customer tries to join provider's internal DIF → No IPCP can join a provider's internal DIF over access DIFs.
- Unauthorized peer/transit provider tries to join e-mall DIF → Cryptographic mutual authentication required to join e-mall DIF.

- Peer/transit provider tries to join provider's internal DIF → No IPCP can join a provider's internal DIF over DIFs shared with peer/transit providers.

5.4.2. Setup

This experiment is executed and configured using the demonstrator tool in a single physical machine. The configuration file for the demonstrator is provided below. Each machine needs to be configured - via the *overlay* directive - with a path to a folder that contains the credentials required for the SSH2 authentication policy to operate correctly (basically RSA keys). Key generation and distribution can be securely accomplished in real world RINA deployments following the design and procedure presented in the "Key distribution" section; for the experiment under consideration we have generated the keys using OpenSSL and placed the files in the right directories.

Demonstrator configuration file

```
eth 110 0Mbps cpe11 ar1
eth 120 0Mbps cpe12 ar1
eth 130 0Mbps cpe13 ar1
eth 210 0Mbps cpe21 ar2
eth 220 0Mbps cpe22 ar2
eth 230 0Mbps cpe23 ar2
eth 310 0Mbps cpe31 ar3
eth 320 0Mbps cpe32 ar3
eth 330 0Mbps cpe33 ar3
eth 100 0Mbps ar1 manpe1
eth 200 0Mbps ar2 manpe1
eth 300 0Mbps ar3 manpe2
eth 410 0Mbps manpe1 manpe2
eth 411 0Mbps manpe1 manpe3
eth 412 0Mbps manpe1 manpe4
eth 420 0Mbps manpe2 manpe3
eth 421 0Mbps manpe2 manpe4
eth 430 0Mbps manpe3 manpe4
eth 510 0Mbps manpe3 ser1
eth 520 0Mbps manpe4 ser2
eth 600 0Mbps ser1 core1
eth 610 0Mbps ser1 core2
eth 620 0Mbps ser2 core1
eth 630 0Mbps ser2 core2
eth 700 0Mbps core1 core2
eth 710 0Mbps core1 core3
```

```
eth 720 0Mbps core2 core4
eth 730 0Mbps core3 core4
eth 640 0Mbps core3 edge1
eth 650 0Mbps core4 edge1
eth 660 0Mbps core3 edge2
eth 670 0Mbps core4 edge2
eth 800 0Mbps edge1 isp2
eth 810 0Mbps edge1 isp3
eth 820 0Mbps edge2 isp4
eth 830 0Mbps edge2 isp5

# DIF core
dif core ser1 600 610
dif core ser2 620 630
dif core core1 600 620 700 710
dif core core2 610 630 700 720
dif core core3 640 660 710 730
dif core core4 650 670 720 730
dif core edge1 640 650
dif core edge2 660 670

# DIF access
dif access ar1 100
dif access ar2 200
dif access ar3 300
dif access manpe1 100 200 410 411 412
dif access manpe2 300 410 420 421
dif access manpe3 411 420 430 510
dif access manpe4 412 421 430 520
dif access ser1 510
dif access ser2 520

# DIF service
dif service ar1 access
dif service ar2 access
dif service ar3 access
dif service ser1 access core
dif service ser2 access core
dif service edge1 core
dif service edge2 core

# DIF emall1
dif emall1 cpe11 110
dif emall1 cpe12 120
dif emall1 cpe21 210
dif emall1 cpe22 220
dif emall1 cpe31 310
dif emall1 ar1 110 120 service
```

```
dif emall1 ar2 210 220 service
dif emall1 ar3 310 service
dif emall1 edge1 service 800
dif emall1 edge2 service 820
dif emall1 isp2 800
dif emall1 isp4 820

# DIF emall2
dif emall2 cpe13 130
dif emall2 cpe23 230
dif emall2 cpe32 320
dif emall2 cpe33 330
dif emall2 ar1 130 service
dif emall2 ar2 230 service
dif emall2 ar3 320 330 service
dif emall2 edge1 service 810
dif emall2 edge2 service 830
dif emall2 isp3 810
dif emall2 isp5 830

#Enrollments
enroll access ar1 manpe1 100
enroll access ar2 manpe1 200
enroll access ar3 manpe2 300
enroll access ser1 manpe3 510
enroll access ser2 manpe4 520
enroll access manpe1 manpe2 410
enroll access manpe1 manpe3 411
enroll access manpe1 manpe4 412
enroll access manpe2 manpe3 420
enroll access manpe2 manpe4 421
enroll access manpe3 manpe4 430

enroll core core1 core2 700
enroll core core1 core3 710
enroll core core2 core4 720
enroll core core3 core4 730
enroll core ser1 core1 600
enroll core ser1 core2 610
enroll core ser2 core1 620
enroll core ser2 core2 630
enroll core edge1 core3 640
enroll core edge1 core4 650
enroll core edge2 core3 660
enroll core edge2 core4 670

enroll service edge1 edge2 core
enroll service edge1 ser1 core
```

```
enroll service edge1 ser2 core
enroll service edge2 ser1 core
enroll service edge2 ser2 core
enroll service ser1 ser2 core
enroll service ar1 ser1 access
enroll service ar1 ser2 access
enroll service ar2 ser1 access
enroll service ar2 ser2 access
enroll service ar3 ser1 access
enroll service ar3 ser2 access

enroll emall1 cpe11 ar1 110
enroll emall1 cpe12 ar1 120
enroll emall1 cpe21 ar2 210
enroll emall1 cpe22 ar2 220
enroll emall1 cpe31 ar3 310
enroll emall1 ar1 edge1 service
enroll emall1 ar1 edge2 service
enroll emall1 ar2 edge1 service
enroll emall1 ar2 edge2 service
enroll emall1 ar3 edge1 service
enroll emall1 ar3 edge2 service
enroll emall1 edge1 edge2 service
enroll emall1 isp2 edge1 800
enroll emall1 isp4 edge2 820

enroll emall2 cpe13 ar1 130
enroll emall2 cpe23 ar2 230
enroll emall2 cpe32 ar3 320
enroll emall2 cpe33 ar3 330
enroll emall2 ar1 edge1 service
enroll emall2 ar1 edge2 service
enroll emall2 ar2 edge1 service
enroll emall2 ar2 edge2 service
enroll emall2 ar3 edge1 service
enroll emall2 ar3 edge2 service
enroll emall2 edge1 edge2 service
enroll emall2 isp3 edge1 810
enroll emall2 isp5 edge2 830

#Overlays
overlay ar1 overlays/ispsec/ar1
overlay ar2 overlays/ispsec/ar2
overlay ar3 overlays/ispsec/ar3
overlay cpe11 overlays/ispsec/cpe11
overlay cpe12 overlays/ispsec/cpe12
overlay cpe13 overlays/ispsec/cpe13
overlay cpe21 overlays/ispsec/cpe21
```

```
overlay cpe22 overlays/ispsec/cpe22
overlay cpe23 overlays/ispsec/cpe23
overlay cpe31 overlays/ispsec/cpe31
overlay cpe32 overlays/ispsec/cpe32
overlay cpe33 overlays/ispsec/cpe33
overlay edge1 overlays/ispsec/edge1
overlay edge2 overlays/ispsec/edge2
overlay isp2 overlays/ispsec/isp2
overlay isp3 overlays/ispsec/isp3
overlay isp4 overlays/ispsec/isp4
overlay isp5 overlays/ispsec/isp5
```

5.4.3. Discussion of observed behaviour

Threat: unauthorized customer tries to join e-mall DIF

Enrolling to any of the two e-mall DIFs of the experiment requires authentication of both IPCPs: the one at the customer's CPE and the one at the provider's edge router. The SSH2 authentication policy used in the experiment is based on the SSH2 authentication protocol and its use of public key cryptography: the provider IPCP at the edge router has access to the public RSA keys of clients. The provider IPCP performs cryptographic operations on a randomly generated array of bytes, and requests the joining IPCP to do the same on an encrypted version of the byte array (server challenge message). If the joining IPCP doesn't have the private key then it won't be able to properly decrypt the byte array and send the correct result to the provider's IPCP; thus authentication will fail.

If a rogue customer wants to join the emall DIF it either needs access to the private key or has enough computing power and time to break the RSA key. Right now the policy uses 2048 long RSA keys - which nowadays security agencies such as NIST consider safe enough from brute-force factorization attacks. Therefore the weakest point in the security chain is key generation and distribution. Designs and approaches such as the one presented in the "Key distribution" section of this document are key to keep networks as secure as possible.

Traces of authentication messages exchanged between IPCPs *cpe11* and *ar1* in DIF *emall1*

```
194(1475066763)#ipcp[8].rib-daemon (DBG): Received CDAP message from N-1 port
 7
Opcode: 0_M_CONNECT
```

```
Abstract syntax: 115
Authentication policy: Policy name: PSOC_authentication-ssh2
Supported versions: 1;

Source AP name: emall1.8.IPCP
Source AP instance: 1
Source AE name: Management
Destination AP name: emall1.1.IPCP
Destination AP instance: 1
Destination AE name: Management
Flags: 0
Invoke id: 1
Version: 1

194(1475066763)#ipcp[8].enrollment-task (DBG): M_CONNECT CDAP message from
 port-id 7
194(1475066763)#librina.timer (DBG): Timer with ID 14187488 started
194(1475066763)#ipcp[8].enrollment-task-ps-default (DBG): Created a new
 Enrollment state machine for remote IPC process: emall1.8.IPCP-1-Management-
194(1475066763)#ipcp[8].enrollment-task-ps-default (DBG): Authenticating IPC
 process emall1.8.IPCP-1 ...
194(1475066763)#ipcp (DBG): IPCPSecurityManager: update crypto state (under-
dif-name 110)
194(1475066763)#librina.security-manager (DBG): Initiating authentication for
 session_id: 7
194(1475066763)#librina.security-manager (DBG): Read RSA keys from keystore
194(1475066763)#librina.security-manager (DBG): Generated client encryption
 key of length 16 bytes: 1ecc82934da576ca8375609963e929b8
194(1475066763)#librina.security-manager (DBG): Generated server encryption
 key of length 16 bytes: 34a180717350235efbea5354f85c97ba
194(1475066763)#librina.security-manager (DBG):
 Generated client mac key of length 32 bytes:
 95e7b9172b2f43d0fc2946685dd59f56c600e22738b04b3e7ff3b9d99866265f
194(1475066763)#librina.security-manager (DBG):
 Generated server mac key of length 32 bytes:
 379b394990b052006209dc5a0947ffa06b0d850d25b459f7b737048d430cc3ff
194(1475066763)#ipcp (DBG): Requesting the kernel to update crypto state on
 port-id: 7
194(1475066763)#librina.nl-manager (DBG): NL msg RX. Fam: 25; Opcode:
 42_UPDATE_CRYPTO_STATE_RESP; Sport: 0; Dport: 194; Seqnum: 1475066562;
 Response; SIPCP: 8; DIPCP: 0
194(1475066763)#librina.nl-manager (DBG): NL msg TX. Fam: 25; Opcode:
 41_UPDATE_CRYPTO_STATE_REQ; Sport: 194; Dport: 0; Seqnum: 1475066562;
 Request; SIPCP: 8; DIPCP: 8
194(1475066763)#librina.core (DBG): Added event of type
 41_UPDATE_CRYPTO_STATE_RESPONSE and sequence number 1475066562 to events
 queue
```

```
194(1475066763)#ipcp[8].core (DBG): Got event of type
 41_UPDATE_CRYPTO_STATE_RESPONSE and sequence number 1475066562
194(1475066763)#librina.security-manager (DBG): Decryption enabled for port-
id: 7
194(1475066763)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU (330)
194(1475066763)#ipcp[8].rib-daemon (DBG): Sent CDAP message of size 412
 through port-id 7:
Opcode: 12_M_WRITE
Flags: 0
Object class: Ephemeral Diffie-Hellman exchange
Object name: Ephemeral Diffie-Hellman exchange
Scope: 0

194(1475066763)#ipcp (DBG): Requesting the kernel to update crypto state on
 port-id: 7
194(1475066763)#librina.security-manager (DBG): Encryption enabled for port-
id: 7
194(1475066763)#ipcp[8].enrollment-task-ps-default (DBG): Authentication in
 progress
194(1475066763)#rib (INFO): Bound port_id: 7 CDAP connection to RIB version 1
 (AE Management)
194(1475066763)#librina.syscalls (DBG): Invoking SYS_readManagementSDU (329)
194(1475066763)#ipcp[8].rib-daemon (DBG): Received CDAP message from N-1 port
 7
Opcode: 12_M_WRITE
Flags: 0
Object class: Client challenge
Object name: Client challenge
Scope: 0

194(1475066763)#librina.syscalls (DBG): Invoking SYS_writeManagementSDU (330)
194(1475066763)#ipcp[8].rib-daemon (DBG): Sent CDAP message of size 420
 through port-id 7:
Opcode: 12_M_WRITE
Flags: 0
Object class: Client challenge reply and server challenge
Object name: Client challenge reply and server challenge
Scope: 0

194(1475066763)#ipcp[8].enrollment-task-ps-default (DBG): Authentication
 still in progress
194(1475066763)#librina.syscalls (DBG): Invoking SYS_readManagementSDU (329)
194(1475066763)#ipcp[8].rib-daemon (DBG): Received CDAP message from N-1 port
 7
Opcode: 12_M_WRITE
Flags: 0
Object class: Server challenge reply
Object name: Server challenge reply
```

```
Scope: 0

194(1475066763)#librina.security-manager (INFO): Remote peer successfully
 authenticated
```

Threat: Unauthorized peer/transit provider tries to join e-mall DIF

This thread is equivalent to the rogue customer trying to access an e-mall DIF; it is not allowed to join. In the experiment we have used the same SSH2-based authentication policies, producing an equivalent output as in the provider customer's case.

Threat: rogue customer has joined e-mall DIF

In this case we assume that the rogue customer has joined the *e-mall DIF* because it gained access to a valid customer's RSA private key. When it enrolls to the DIF authentication will succeed and the provider's IPCP will consider the joining IPCP a valid member of the DIF. However, the damage that the rogue IPCP can inflict is limited.

First of all, it can only get information about the IPCPs belonging to the provider's border routers that participate in the *emall* DIF: all the provider inner DIFs and systems are invisible to it, as shown in the following caption. In this case the e-mall DIF distributes routing information from the DIF to the customers (equivalent to a customer connected via BGP today), but the provider could choose to exchange less routing information from the *emall DIF* to its customers (maybe just a default next hop). In any case, no information from lower DIFs is disclosed.

1. Routing table in the rogue IPCP RIB

```
Name: /resalloc/nhopt/key=17-0; Class: NextHopTableEntry; Instance: 156
Value: Destination address: 17; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=18-0; Class: NextHopTableEntry; Instance: 159
Value: Destination address: 18; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=19-0; Class: NextHopTableEntry; Instance: 162
Value: Destination address: 19; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=25-0; Class: NextHopTableEntry; Instance: 157
Value: Destination address: 25; QoS-id: 0; Cost: 1; Next hop addresses: 17/
```

```
Name: /resalloc/nhopt/key=27-0; Class: NextHopTableEntry; Instance: 160
Value: Destination address: 27; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=28-0; Class: NextHopTableEntry; Instance: 161
Value: Destination address: 28; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=30-0; Class: NextHopTableEntry; Instance: 163
Value: Destination address: 30; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=33-0; Class: NextHopTableEntry; Instance: 158
Value: Destination address: 33; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=34-0; Class: NextHopTableEntry; Instance: 164
Value: Destination address: 34; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=35-0; Class: NextHopTableEntry; Instance: 165
Value: Destination address: 35; QoS-id: 0; Cost: 1; Next hop addresses: 17/

Name: /resalloc/nhopt/key=37-0; Class: NextHopTableEntry; Instance: 166
Value: Destination address: 37; QoS-id: 0; Cost: 1; Next hop addresses: 17/
```

Let us assume that the rogue customer has somehow got access to provider's internal information and knows that there is a DIF called *service* that is also present in the access router. Even if the customer tries to enroll to this DIF, the access router will just reject this enrollment request because IPCPs in the access router are available via the N-1 DIF between the access router and the customer. The mere fact that a customer tries to join the *service DIF* is a clear indication that the customer may be an attacker; so the provider could quickly use these sort of events to detect attacks from illicit or compromised customers.

IPC Processes in access router. IPCPs in service DIF are only available via the *access DIF*.

```
Current IPC processes (id | name | type | state | Registered applications |
 Port-ids of flows provided)
    1 | eth.1.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 100 |
 access.1.IPCP-1-- | 1
    2 | eth.2.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 110 |
 emall1.1.IPCP-1-- | 7
    3 | eth.3.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 120 |
 emall1.1.IPCP-1-- | 8
    4 | eth.4.IPCP:1:: | shim-eth-vlan | ASSIGNED TO DIF 130 |
 emall2.1.IPCP-1-- | 4
    5 | access.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF access.DIF |
 service.1.IPCP-1-- | 2, 3
```

```
   6 | service.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF service.DIF |
emall2.1.IPCP-1--, emall1.1.IPCP-1-- | 5, 9
   7 | emall2.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF emall2.DIF | - | -
   8 | emall1.1.IPCP:1:: | normal-ipc | ASSIGNED TO DIF emall1.DIF | - | -
```

Last but not least, access control policies can limit the number of actions that the customer IPCP can perform in the *emall DIF*. As shown in [apcc2016], access control rules are expressed in terms of operations on the RIB; so the customer IPCP can be restricted to a basic subset of actions that don't compromise its proper operation. Actions such as creating flows whose target is an IPCP in the operator infrastructure, or advertising routes to next hops that don't belong to the customer's address space can be quickly rejected.

Threat: System may get access to or tamper data sent over physical media between customer and provider

An attacker may tap the cable between the customer and the provider or - more likely - be able to scan the radio spectrum to intercept wireless communications. To prevent or mitigate this threat cryptographic SDU protection policies are used in this experiment, in conjunction with HMAC codes. In the SSH2 policies, the first thing both IPCPs do is a Diffie-Hellman key exchange, resulting in the computation of a master secret that is then used to generate encryption and HMAC keys (one for each communication direction: rx and tx). As shown in Authentication messages exchanged between customer and provider IPCPs, captured with tcpdump, all messages are encrypted after the Diffie-Hellman exchange, therefore an attacker observing the physical media would have to break the Diffie-Hellman key exchange in order to be able to decrypt the information. The message authentication code (HMAC) protects PDUs in transit from tampering.

Authentication messages exchanged between customer and provider IPCPs, captured with tcpdump

```
14:46:07.257898 00:0a:0a:0a:09:01 (oui Unknown) > Broadcast, ethertype 802.1Q
 (0x8100), length 72:
0x0000:  0001 d1f0 0611 0001 000a 0a0a 0901 656d  ..............em
0x0010:  616c 6c31 2e39 2e49 5043 502f 312f 2fff  all1.9.IPCP/1//.
0x0020:  ffff ffff ff65 6d61 6c6c 312e 312e 4950  .....emall1.1.IP
0x0030:  4350 2f31 2f2f                           CP/1//
```

```
14:46:07.257982 00:0a:0a:0a:01:03 (oui Unknown) > 00:0a:0a:0a:09:01 (oui
 Unknown), ethertype 82.1Q (0x8100), length 72:
0x0000:  0001 d1f0 0611 0002 000a 0a0a 0103 656d  ..............em
0x0010:  616c 6c31 2e31 2e49 5043 502f 312f 2f00  all1.1.IPCP/1//.
0x0020:  0a0a 0a09 0165 6d61 6c6c 312e 392e 4950  .....emall1.9.IP
0x0030:  4350 2f31 2f2f                           CP/1//

14:46:07.267152 00:0a:0a:0a:09:01 (oui Unknown) > 00:0a:0a:0a:01:03 (oui
 Unknown) 02.1Q (0x8100), length 456:
0x0000:  0100 0019 0001 0000 0000 0040 00b6 0100  ...........@....
0x0010:  0000 0008 7310 0018 012a 0032 0038 0048  ....s....*.2.8.H
0x0020:  0050 0092 01c1 020a 1850 534f 435f 6175  .P.......PSOC_au
0x0030:  7468 656e 7469 6361 7469 6f6e 2d73 7368  thentication-ssh
0x0040:  3212 0131 1aa1 020a 0345 4448 1206 4145  2..1.....EDH..AE
0x0050:  5331 3238 1a06 5348 4132 3536 2207 6465  S128..SHA256".de
0x0060:  666c 6174 652a 8002 732b d76d 0fe4 9b7c  flate*..s+.m...|
0x0070:  b8d7 b40e 2ec5 7382 9a3d 3d3f 29ab 301b  ......s..==?).0.
0x0080:  baac ac5d 3c0a 94e8 c1aa 657f 5b72 31bd  ...]<.....e.[r1.
0x0090:  53ce 2fd2 3efc 6f84 41d1 88c4 6369 3418  S./.>.o.A...ci4.
0x00a0:  586f 2835 2bb2 0d78 442b 5f91 1b5d 9965  Xo(5+..xD+_..].e
0x00b0:  aed1 4bdb d623 1b20 fcd7 ccc2 2ca7 e452  ..K..#......,..R
0x00c0:  e80d a91f 8396 85fb 1fdb 83c5 4d91 b470  ............M..p
0x00d0:  c3b0 1a6b 3f32 9b4e e433 8d02 1a64 66a0  ...k?2.N.3...df.
0x00e0:  cef1 6435 d9a6 f877 f6f0 c8ca fc81 15fa  ..d5...w........
0x00f0:  1918 c29d 08b3 dd65 6223 1ca1 102c b4c2  .......eb#...,..
0x0100:  df0d c306 e108 7c5d 1fc3 d5af f9ca 9051  ......|].......Q
0x0110:  bd6d 67cd c5ea c607 6507 7add 241a c673  .mg.....e.z.$..s
0x0120:  7778 7a6e b442 bb13 c22e cc1a 5c92 843d  wxzn.B......\..=
0x0130:  6df5 a93b c65a 20d5 f7a7 4aff 166c 16fe  m..;.Z....J..l..
0x0140:  254c 0413 06a5 1683 c5a4 72db a238 bfaa  %L........r..8..
0x0150:  2337 8ab7 e06a 43ae 35db cabe b789 b020  #7...jC.5.......
0x0160:  9314 7792 eb0b a83a 9a01 00a2 010a 4d61  ..w....:......Ma
0x0170:  6e61 6765 6d65 6e74 aa01 0131 b201 0d65  nagement...1...e
0x0180:  6d61 6c6c 312e 312e 4950 4350 ba01 00c2  mall1.1.IPCP....
0x0190:  010a 4d61 6e61 6765 6d65 6e74 ca01 0131  ..Management...1
0x01a0:  d201 0d65 6d61 6c6c 312e 392e 4950 4350  ...emall1.9.IPCP
0x01b0:  da01 00e0 0101                           ......

14:46:07.282314 00:0a:0a:0a:01:03 (oui Unknown) > 00:0a:0a:0a:09:01 (oui
 Unknown), 802.1Q (0x8100), length 449:
0x0000:  0100 0011 0001 0000 0000 0040 00af 0100  ...........@....
0x0010:  0000 0008 0010 0c18 002a 2145 7068 656d  .........*!Ephem
0x0020:  6572 616c 2044 6966 6669 652d 4865 6c6c  eral.Diffie-Hell
0x0030:  6d61 6e20 6578 6368 616e 6765 3221 4570  man.exchange2!Ep
0x0040:  6865 6d65 7261 6c20 4469 6666 6965 2d48  hemeral.Diffie-H
0x0050:  656c 6c6d 616e 2065 7863 6861 6e67 6538  ellman.exchange8
0x0060:  0042 a402 32a1 020a 0345 4448 1206 4145  .B..2....EDH..AE
0x0070:  5331 3238 1a06 5348 4132 3536 2207 6465  S128..SHA256".de
```

```
0x0080:  666c 6174 652a 8002 2318 620c 1cdb 1f18  flate*..#.b.....
0x0090:  8c8e cb12 32f4 4626 4345 df9e 1521 cfca  ....2.F&CE...!..
0x00a0:  6272 f9bd 2cfb 4246 f44e 3792 e48c 3cf9  br..,.BF.N7...<.
0x00b0:  65fd af03 59b7 9e71 5a66 3121 87bd 4996  e...Y..qZf1!..I.
0x00c0:  020b 0d3d 4382 242e 0ef8 21ae e7ef 737b  ...=C.$...!...s{
0x00d0:  c25f 39da 449b 841d b1cd 58e0 e538 a1d0  ._9.D.....X..8..
0x00e0:  2466 8f1c 6028 cef3 3e7f e932 b513 09fa  $f..`(..>..2....
0x00f0:  e9cd 6f5a eab0 b0d7 f248 030a beb8 24af  ..oZ.....H....$.
0x0100:  a53c 467b 7443 e164 0ae9 8e8d e823 9965  .<F{tC.d.....#.e
0x0110:  4d74 2d32 9293 5f29 8478 81b8 f2f8 2542  Mt-2.._).x....%B
0x0120:  b3c8 2b88 150e 730d 4a6d 0151 ee27 7ff7  ..+...s.Jm.Q.'..
0x0130:  776f 0d3b f12a 476c b850 e8ac c33c a621  wo.;.*Gl.P...<.!
0x0140:  29a0 2c32 b380 7d20 067d e144 ff80 2f5f  ).,2..}..}.D../_
0x0150:  6e4e 3e0d 5a9f 56bc 78f6 1722 69dd 76a6  nN>.Z.V.x.."i.v.
0x0160:  a987 84e7 f965 4ad9 c43b 7345 d3fc 84ef  .....eJ..;sE....
0x0170:  09e6 6e61 0b25 360c d241 721e 0f55 b2eb  ..na.%6..Ar..U..
0x0180:  f073 5bae eb6a 339c 4800 5000 9201 020a  .s[..j3.H.P.....
0x0190:  009a 0100 a201 00aa 0100 b201 00ba 0100  ................
0x01a0:  c201 00ca 0100 d201 00da 0100 e001 00    ................

14:46:07.291367 00:0a:0a:0a:09:01 (oui Unknown) > 00:0a:0a:0a:01:03 (oui
 Unknown), 802.1Q (0x8100), length 434:
0x0000:  10c0 f8d8 ed45 eca3 0cf5 392b 46a5 5d05  .....E....9+F.].
0x0010:  6138 67c8 99f1 f120 1e7e d0ee 8c1e d6d4  a8g......~......
0x0020:  0d62 b513 038a 5118 f381 4caa 9a88 8f85  .b....Q...L.....
0x0030:  1542 6f0a bc73 ec0a 0e61 2bc9 509e 7c05  .Bo..s...a+.P.|.
0x0040:  1066 0786 e5ed 64b1 3dc9 9738 dace 877f  .f....d.=..8....
0x0050:  717e 02dd 7c55 de96 2eb9 8623 2e43 2b3b  q~..|U.....#.C+;
0x0060:  4e44 a462 b858 703b 46bc 3d07 5da2 889d  ND.b.Xp;F.=.]...
0x0070:  f7e0 fae0 31db 555e fc40 552a 4d76 a868  ....1.U^.@U*Mv.h
0x0080:  aafa 58e1 4308 3ed9 54d1 da62 7d78 08fc  ..X.C.>.T..b}x..
0x0090:  c874 32a4 abca 26f6 c161 69f1 6f2f a797  .t2...&..ai.o/..
0x00a0:  a368 90c4 23d1 e6c4 7b7e cba4 51b4 fe00  .h..#...{~..Q...
0x00b0:  4fea 0def a69f f0f8 c193 2eee 177f 20a2  O.............
0x00c0:  e689 c3e0 f8c6 0c4a a0fc 0975 dadf de10  .......J...u....
0x00d0:  825a 78dc 9e12 a1eb b9f0 b903 a5cb 0284  .Zx.............
0x00e0:  d6f1 92a4 c498 8d33 a8d5 c127 5396 9096  .......3...'S...
0x00f0:  de15 e54c 1c0a 8d73 8d0c 5cdf 4310 6f37  ...L...s..\.C.o7
0x0100:  5129 94ec 94ce eb8e 322b 5448 88f6 c7c9  Q).....2+TH....
0x0110:  9dcf a874 717f 80fd 7d23 a21c ae71 33aa  ...tq...}#...q3.
0x0120:  f2f4 d990 3cb9 71da 8f7c d9ac 1b67 0c33  ....<.q..|...g.3
0x0130:  a72b 42b0 6dc4 04bf 0d24 1857 3d97 fb0e  .+B.m....$.W=...
0x0140:  f4bc 4101 7037 3afc 5733 8c39 911f bb57  ..A.p7:.W3.9...W
0x0150:  f70c 28ec 7d23 79e1 63a1 ac88 df13 a66f  ..(.}#y.c......o
0x0160:  6e46 6d09 d5d6 95ab d7d4 a351 e18c a990  nFm........Q....
0x0170:  285d 7fc3 f124 cc2b 49b8 9103 7aae 319a  (]...$.+I...z.1.
0x0180:  d0ae e4b2 828d c04b 5d92 e555 95fb 3878  .......K]..U..8x
0x0190:  871d 0f66 5306 c63d 8476 e407 4067 186c  ...fS..=.v..@g.l
```

5.5. Distributed Cloud

The Distributed Cloud use case consists in the ability to perform the following 2 tasks:

- setup machines that become part of the cloud and that are ready to deploy DAFs
- actually deploy DAFs on such machines

This is now possible in any SlapOS cloud and in this experiment, we'll show how to do both tasks with examples inside VIFIB.

5.5.1. PaaS: Installation of a node

The installation of a node with RINA support differs very slighly from the normal procedure: at some point, you'll have to run the rina playbook[36] instead of the re6stnet one[37]. In addition to do what the re6stnet playbook does, this will perform the following actions:

- install packages (only Debian supported for the moment): kernel, librina, rinad and development libraries
- write IPCM configuration in */etc/ipcm-re6st*
- setup a systemd unit for the IPCM

IPCM is configured as follows:

- shim-tcp-udp.dif:

```
{
  "difType": "shim-tcp-udp"
}
```

- default.dif:
  - empty knownIPCProcessAddresses
  - no enrollment retry to avoid failures due to unknown IPCP addresses

At startup, we have:

[36] https://lab.nexedi.com/nexedi/slapos.package/blob/master/playbook/rina.yml
[37] https://lab.nexedi.com/nexedi/slapos.package/blob/master/playbook/re6stnet.yml

id	name	type	state	Registered applications	Port-ids of flows provided
1	re6st:1::	shim-tcp-udp	ASSIGNED TO DIF shim	-	-

Then the re6st daemon figures out that it has to do the following steps:

```
IPCM >>> create-ipcp re6st.917547.32 1 normal-ipc
IPC process created successfully [id = 2, name=re6st.917547.32]
```

```
# The following command causes the IPCM to ask re6st its own IPCP address
IPCM >>> assign-to-dif 2 normal.DIF default.dif
DIF assignment completed successfully
```

```
# At this point, re6st configures hostname/expReg of the shim
```

```
IPCM >>> register-at-dif 2 shim
IPC process registration completed successfully
```

Which gives:

id	name	type	state	Registered applications	Port-ids of flows provided
1	re6st:1::	shim-tcp-udp	ASSIGNED TO DIF shim	re6st.917547.32-1--	
2	re6st.917547.32:1	normal-ipc	ASSIGNED TO DIF normal.DIF	-	-

Later, when another node with RINA support is discovered:

```
# This causes the IPCP to ask re6st the remote IPCP address
IPCM >>> enroll-to-dif 2 normal.DIF shim re6st.917542.32 1
DIF enrollment succesfully completed in 1377 ms
```

re6st uses the following command to decide if any enrollment needs to be done:

```
IPCM >>> query-rib 2 Neighbor /difManagement/enrollment/neighbors/
```

```
Name: /difManagement/enrollment/neighbors/processName=re6st.917542.32; Class:
 Neighbor; Instance: 44
Value: Name: re6st.917542.32-1--; Address: 80; Enrolled: 1
; Supporting DIF Name: shim; Underlying port-id: 1; Number of enroll.
 attempts: 0
```

5.5.2. SaaS: Example of deployed DAF

With what has been done so far, it is already easy for a user to have access to a pool of machines for ad-hoc development/deployment of DAFs, for example by requesting development environment (e.g. webrunner) on any node, or KVMs that can join the cloud easily.

implementation

But we want to go further by defining DAFs that can be deployed by the cloud. As an example, we'll see an example of *Software Release* (SR) that just instantiates 2 *rina-echo-time* processes: 1 in server mode and the other in client mode. The example has been extended so that they are deployed on 2 different so-called *partitions*, that can be allocated on 2 different machines.

First, we define how to build the binaries. Here, it's just *rina-tools* and it was added to slapos.git as a component: https://lab.nexedi.com/nexedi/slapos/tree/master/component/rina-tools

Then, we actually define the SR: required binaries and configuration of services. It has been called hellorina[38].

We can see in particular the use of the *instance-guid* SlapOS value, as a unique identifier within the cloud. It is passed to *rina-echo-time* as *--server-api* parameter.

id	name	type	state	Registered applications	Port-ids of flows provided
1	re6st:1::	shim-tcp-udp	ASSIGNED TO DIF shim	re6st.917550.32-1--	
2	re6st.917550.32-1--	normal-ipc	ASSIGNED TO DIF normal.DIF	rina.apps.echotime.server-SOFTINST-71855--	

[38] https://lab.nexedi.com/nexedi/slapos/tree/master/software/hellorina

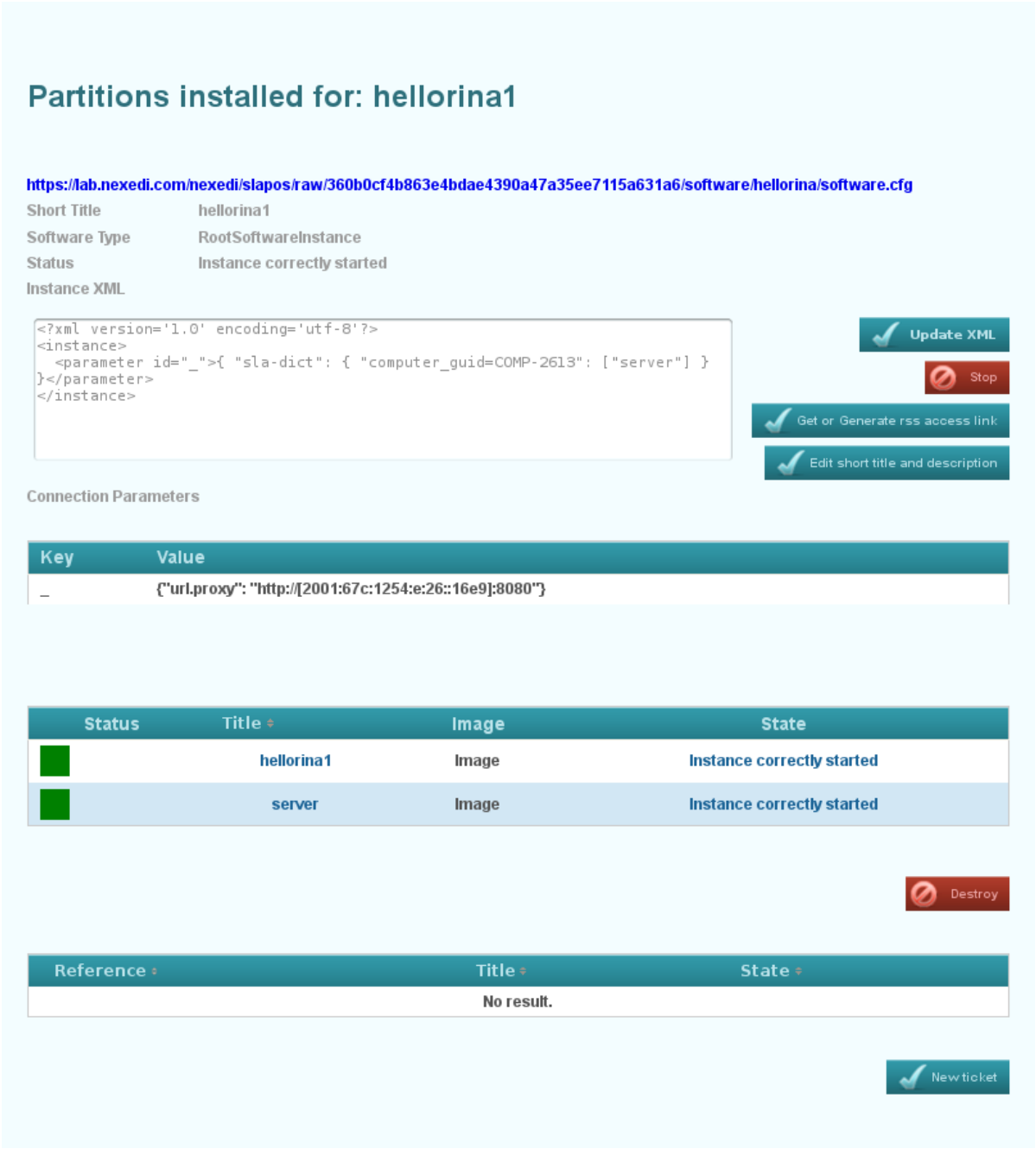
Note that the client *rina-echo-time* is currently a simple python HTTP app that spawns a client *rina-echo-time* at every request and returns its output in real-time.

deployment

An instance called *hellorina1* has been deployed in VIFIB cloud, on 2 machines *COMP-2613* and *COMP-2614*. The instance was actually requested on *COMP-2614* but with a parameter to place the server partition on the other machine: the above `list-ipcps` output is that of *COMP-2613*.

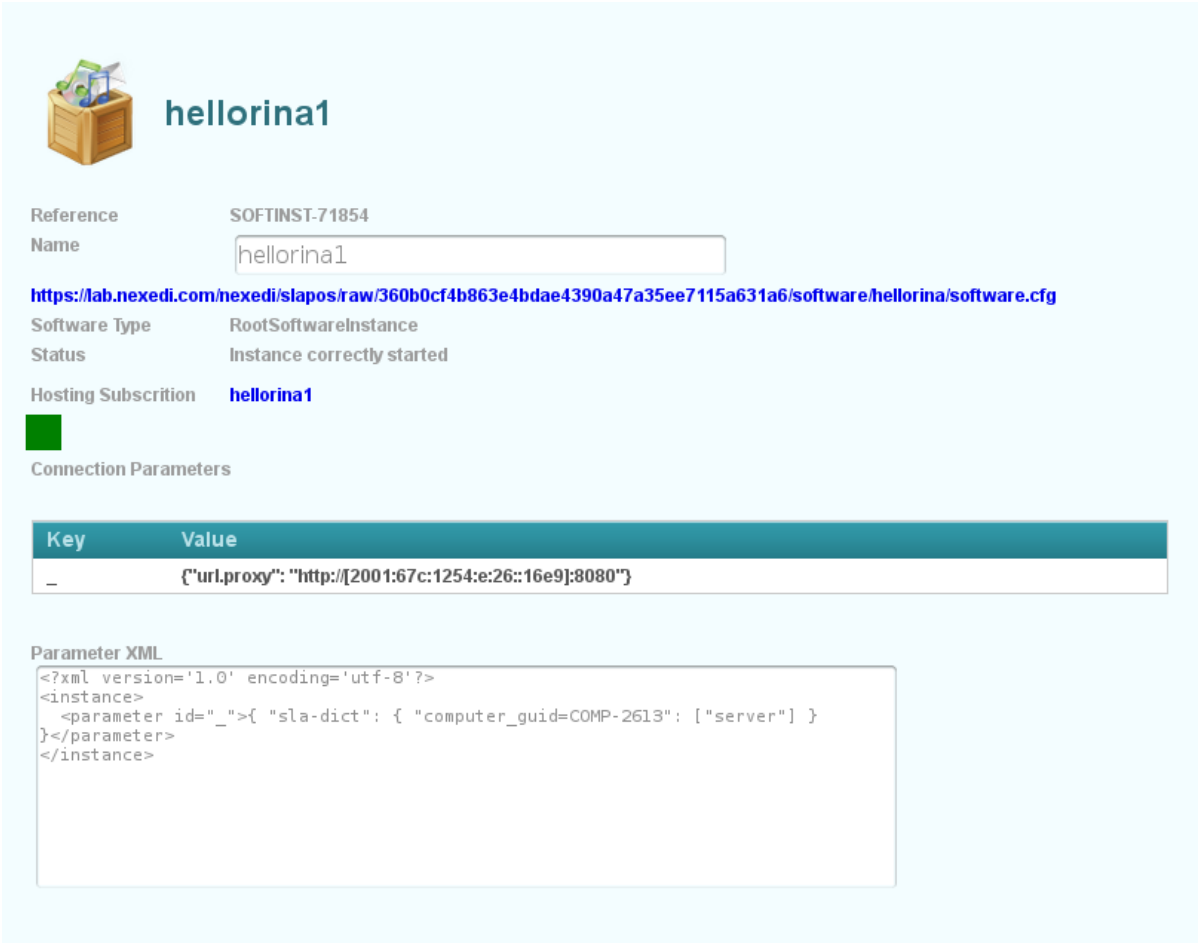
The following page in the management interface of the cloud shows the instance, and in particular:

- the *Instance XML*: here, there's only 1 paremeter to request the *server* partition on a specific computer
- the *Connection Parameters*: the only result of the instanciation is the URL to test the client *rina-echo-time*, at http://[2001:67c:1254:e:26::16e9]:8080/
- the list of partitions that make up the instance



The following pages show the status of the partitions:

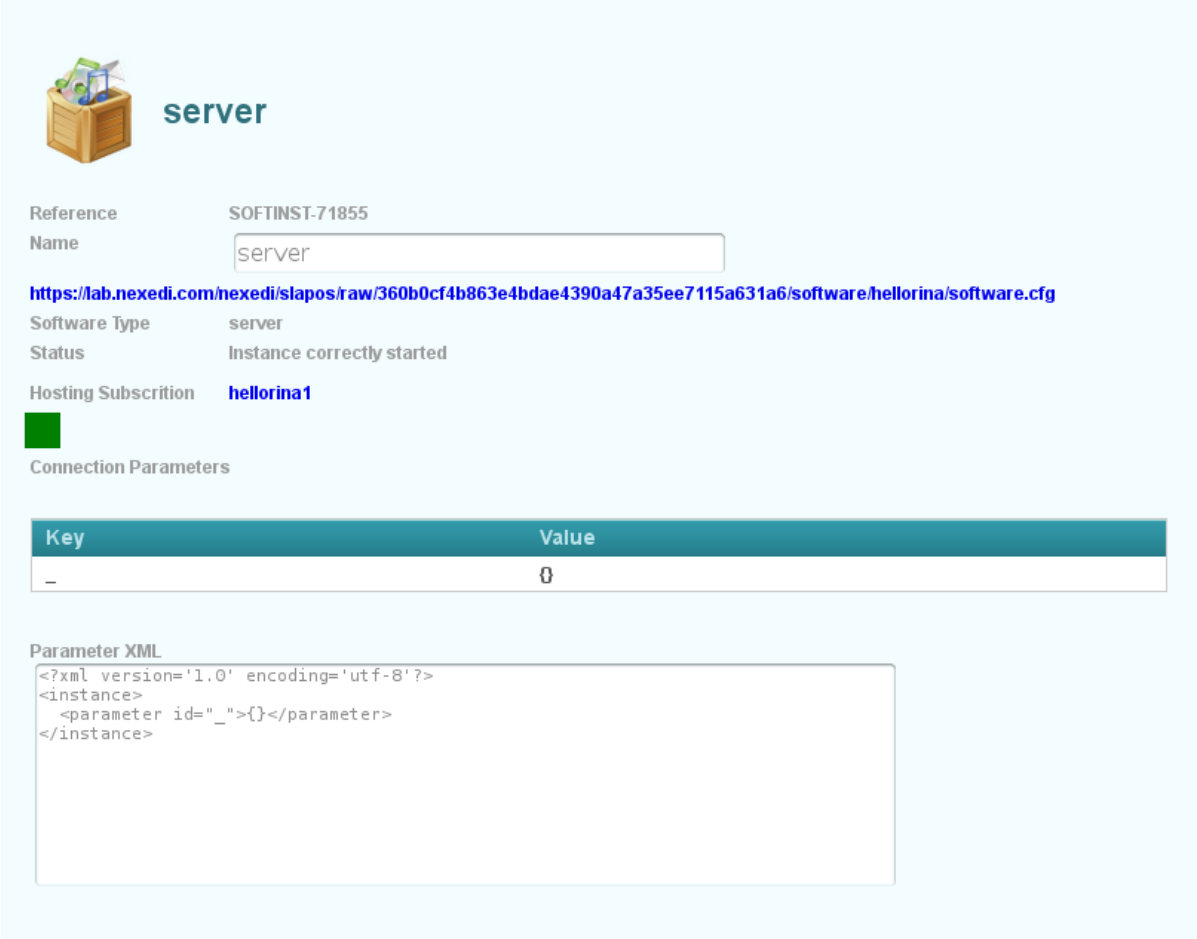
1. the root partition, on COMP-2614:



On the server itself:

```
# slapos node status slappart9:
slappart9:proxy-on-watch         RUNNING   pid 4527, uptime 1 day, 0:29:24
# ps 4527
  PID TTY      STAT   TIME COMMAND
 4527 ?        S      0:14 /opt/slapgrid/0c2b786ea377b73d412481ae19d7891a/
parts/python2.7/bin/python2.7 /opt/
slapgrid/0c2b786ea377b73d412481ae19d7891a/parts/proxy/proxy SOFTINST-71855
 2001:67c:1254:e:26::16e9 8080
```

2. the server partition, on COMP-2613, where you can see the unique identifier that is used (*SOFTINST-71855*):




```
# slapos node status slappart9:
slappart9:server-on-watch        RUNNING   pid 31249, uptime 1 day,
 1:16:00
# ps 31249
  PID TTY      STAT   TIME COMMAND
31249 ?        Sl     0:03 /opt/slapgrid/0c2b786ea377b73d412481ae19d7891a/
parts/rina-tools/bin/rina-echo-time -l --server-api SOFTINST-71855
```

5.5.3. Conclusion

The above examples are real resources inside VIFIB. The proxy URL http://[2001:67c:1254:e:26::16e9]:8080/ is public and you can test it.

In a SlapOS cloud, there is quite a lot of code to deal with the several IPs of a machine, to make sure that each service can communicate with each other. Not only in the core of SlapOS but also when defining Software Releases. Services in the same partition share the same IP and it is also a challenge to deal with port assignments.

RINA is huge improvement in this regard: for something equivalent, i.e. a DIF than spans the whole cloud, the implementation is much simpler, and thus easier to maintain.

5.6. Network Service Provider: Policies for QoS-aware Multipath routing

5.6.1. Introduction

What we intend to prove in this joint experiment are the advantages of using together the centralized resource reservation multipath strategy and the QTAMux scheduler. The idea is to use the centralized multipath to distribute flows between paths in such a way as to facilitate the task of scheduling and drop packets ensuring the minimum overall quality degradation.

What we want to achieve with each component is as follows:

- Multipath forwarding:
  - Even distribution of load avoiding bottlenecks.
  - Even distribution of load by QoS to prevent high priority traffic from accumulating in one link.
- QTAMux Scheduling:
  - Dropping polices that prioritize high cherished packets over low cherished ones.
  - Scheduling polices sthat prioritize high priority packets over low priority ones.

5.6.2. Centralized resource reservation multipath strategy

The centralized resource reservation multipath strategy is presented in D5.4[39]. The strategy consists of a forwarding policy, based in a central manager, which computes the path using all the information available in the network.

[39] https://wiki.ict-pristine.eu/wp5/d54/42-centralized-resource-reservation

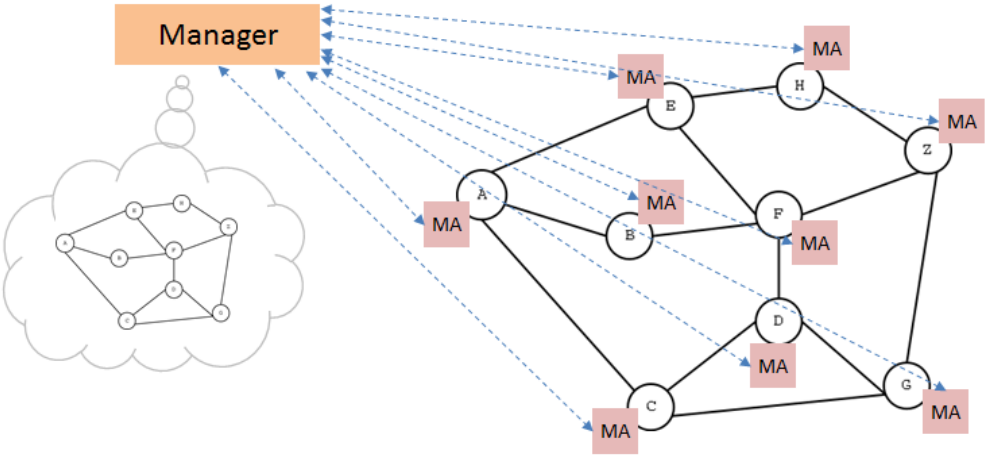


Figure 121. Centralized resource reservation multipath concept

Making use of the features included in RINA, it was possible to implement the strategy in a simple manner. First, the RINA architecture has built-in support of a central control manager to supervise the network that communicates with IPCPs using Management Agents. Second, the RIB acts as a distributed database between IPCPs of a DIF and the Manager, that contains all the objects required to perform any configuration change in the policies with a simple communication protocol (CDAP). Finally, as already mentioned for previous developed multipath policies, the integrated definition of QoS cubes to classify the flows is essential to achieve a good optimisation of the traffic distribution across multiple end-to-end paths. Results published in D5.4[40] proved a significant improvement in load balancing over the hop by hop strategies, avoiding bottlenecks and achieving levels of utilization near to 100%.

5.6.3. QTAMux scheduler

The QTAMux scheduler is a scheduling strategy presented in D3.3[41]. This strategy focuses in providing two levels of control in the degradation that flows suffers when waiting to be served at any RMT port. With the QTAMux scheduler, we are able to provide strong boundaries to the degradation of flows, both in terms of delay and drops, making it more consistent in real world networks, constrained by different SLAs. In addition to providing a great QoS differentiation, the usage QTAMux scheduling strategy is not limited to connection-oriented scenarios, but

[40] https://wiki.ict-pristine.eu/wp5/d54/42-centralized-resource-reservation
[41] https://wiki.ict-pristine.eu/wp3/d33/D33-delay-loss-mux

works well also with connectionless ones, making it a great candidate for a future RINA Internet's default scheduler policy.

5.6.4. Experiment configuration

The experiment will setup nodes forming a fat tree topology as the one showed in the Figure 122:

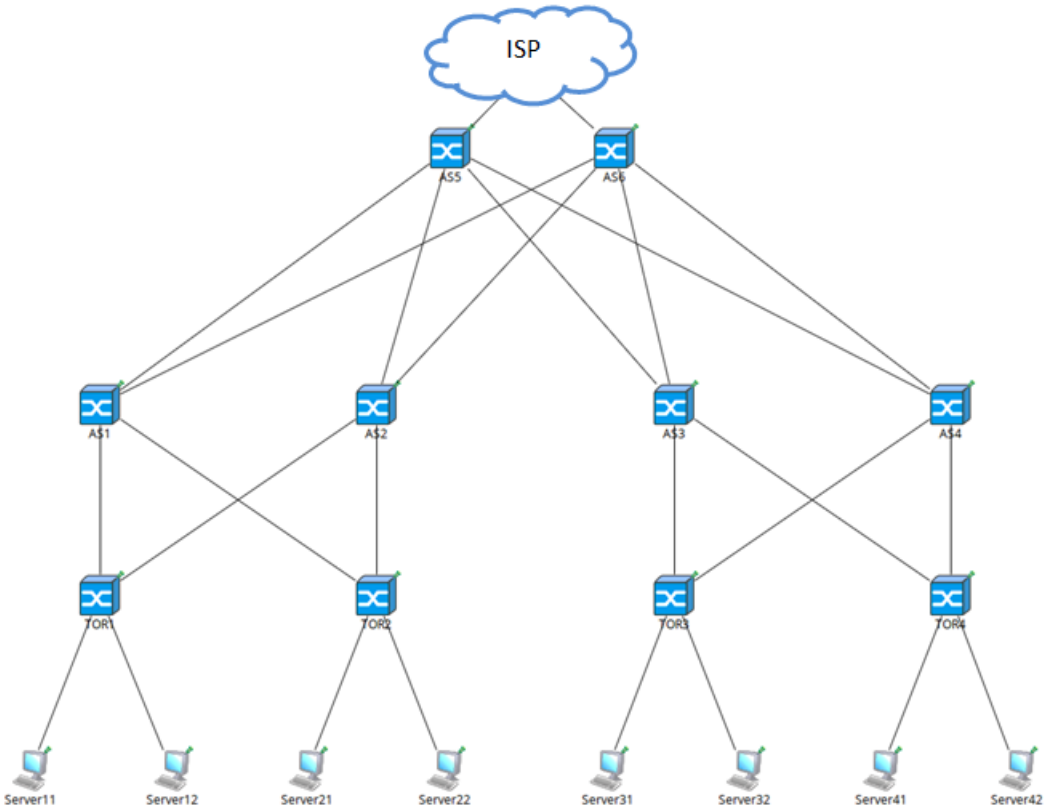


Figure 122. Fat tree topology.

As previously highlighted in multipath experiments, the fat tree topology provides redundant connectivity between servers and between servers and the ISP. Every link inside of the Data Center is 1Gbps, and for simplicity connexion with the ISP is considered infinite.

DIF configuration

The DIF configuration for the experiment is the one depicted by the following Figure 123:

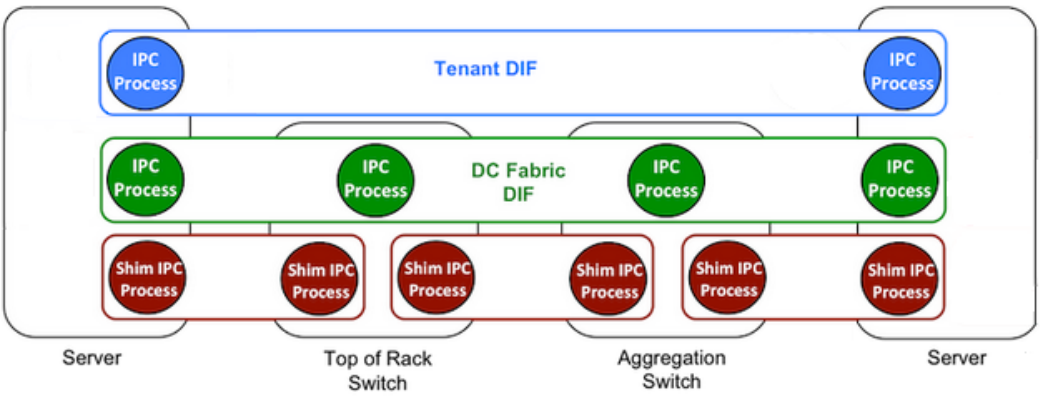


Figure 123. DIF configuration.

The DC Fabric DIF spans among all the routers of the network, and the Tenant DIF connects Serves between them and with the ISP.

Traffic classification

In order to test QTAMux capabilities of prioritize traffic when scheduling we defined several traffic types with different priorities. The traffic types and their characteristics are showed in Figure 124:

Traffic type	QoS priority	Drop preference	Retrasmission	Data rate	Traffic location
Video streaming	High	Low	No	High	External
Internet data	Medium	Medium	Yes	Medium	External
File transfer	Low	High	Yes	High	External
Management & signaling traffic	High	Very low	Yes	Low	Internal
VM migration	Low	High	Yes	Medium	Internal
Low-priority signaling	Medium	Very high	No	Low	Internal
Video on demand	Medium	Low	Yes	High	Internal

Figure 124. Traffic classification

To be able to simulate this test as realistic as possible, the traffic of each QoS was generated using the generic traffic type that best suits its requirements. In the following lines we describe the 2 different generic traffic types.

Type 1

This is the simplest traffic type, where retransmissions are not considered. Traffic pattern is composed by two periods, ON and OFF. During the ON period, traffic is sent in a constant PDU/s rate, but with random size PDUs. During OFF period no traffic is sent. The three parameters

are configurable, PDU size, ON period, and OFF period. With this type of traffic it is possible to simulate traffic patterns like constant flows or streaming with transmission windows. In this experiment was used to simulate: Video streaming and low priority signalling.

Figure 125 shows the departure of packets in number of SDU generated and its payload with a typical configuration.

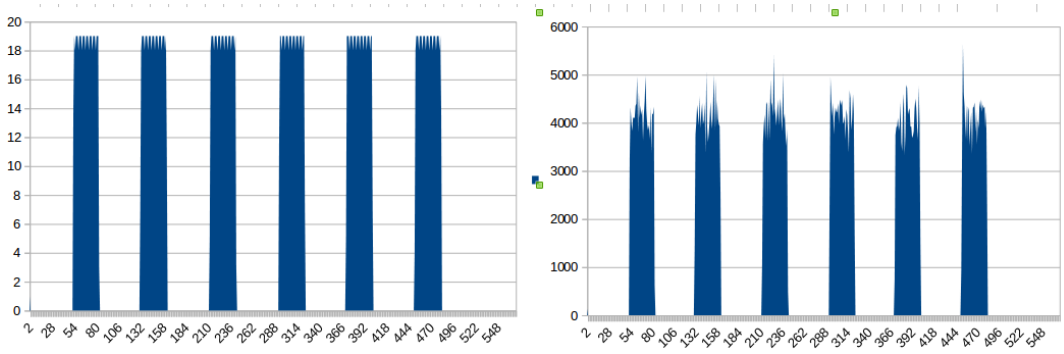


Figure 125. Departure of packets in number of SDU generated and its payload.

Type 2

This is a more complex traffic pattern with retransmission of lost PDU and request response behaviour. In intervals, if the previous request has been fulfilled, the client requests an amount of data to the server. After receiving the request, the server sends as data as requested. This traffic pattern can be used to simulate traffic sensible to losses, like file transfer. In this experiment this pattern was used to simulate: Internet data, file transfer, management, VM migration, Low-priority signalling and video on demand.

Figure 126 and Figure 127 show the graphically the traffic pattern.

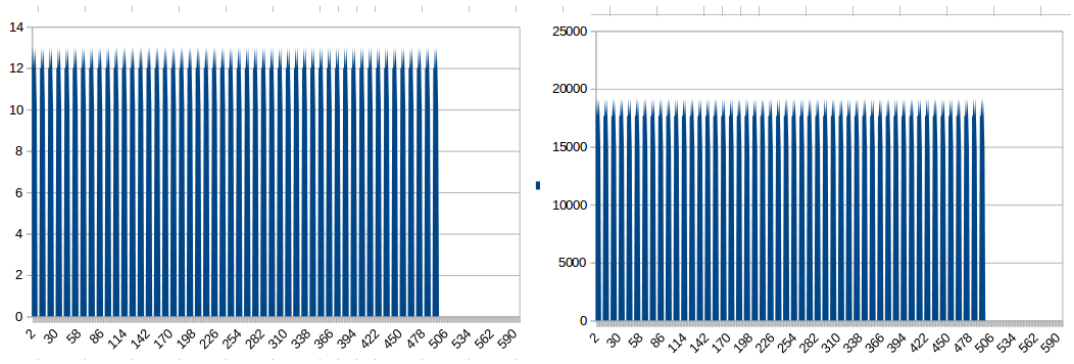


Figure 126. Departure of packets in number of SDU generated and its payload at the server.

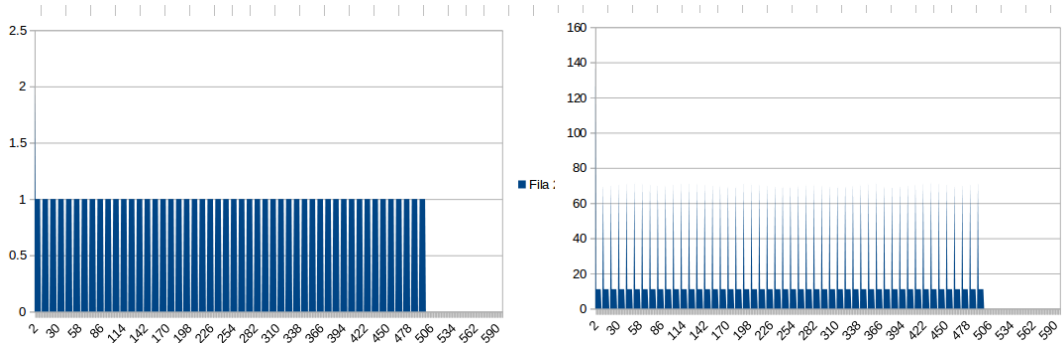


Figure 127. Departure of packets in number of SDU generated and its payload at the client.

QTAMux

For the QTAMux configuration it is necessary to map the previously presented traffic into the Urgency/Cherish matrix. The Cherish determines the dropping priority of the flow, being A the highest and D the lowest. The idea is to drop packets with low cherish preferentially to the ones with high cherish. On the other hand, urgency indicates the sensitivity of the packets to delay, being zero the higher and three the highest. The scheduling function will prioritize lower urgencies to reduce queueing time and deliver packets in time.

The used Urgency/Cherish matrix is as showed in Figure 128:

		Urgency			
		0	1	2	3
Cherish	A	MA			
	B		Streaming	VoD	VM migration
	C			Internet	
	D		Signalling		File transfer

Figure 128. Urgency/Cherish matrix.

Traffic distribution

Distribution of traffic is represented in Figure 129:

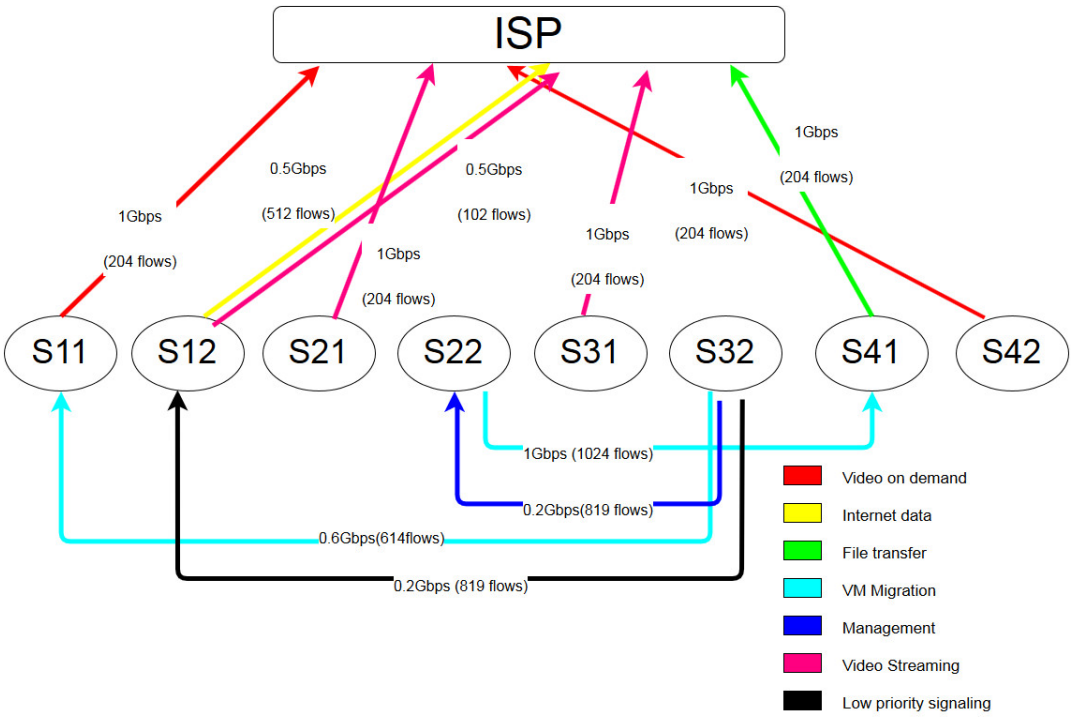


Figure 129. Distribition of traffic

It should be noted that this information is not previously known by the forwarding policy.

Results

For this experiment we are expecting results in three different areas. First, used bandwidth should be as evenly distributed as possible. To accomplish this, forwarding policy is configured to equally distribute both, total and per QoS bandwidth. Second, Flows marked with higher urgency are supposed to experience a lower delay. To that end, urgent packets are served first by the scheduler policy. Finally, Flows with higher cherish are less tolerant to losses, therefore they will have a lower dropping ratio. To achieve this, queues are individual for each QoS and the max length is associated with QoS's cherish.

Load balanding

The used method for testing the correct load balancing in the experiment is as follows. We study the utilization of symmetric links, taking into account both total and per QoS bandwidth. For example in this document we will show the results obtained in the pairs (AS2 → AS5, AS2 #AS6) and (TOR3- > AS3, TOR3 → AS4). Theoretically the links in the same pair have the same

load; this is so because AS2 and TOR3 can use indistinctly both links to reach every destination server or the ISP network.

Figure 130 and Figure 131 shows the difference between congestion levels in the nodes AS2 and TOR3 represented in absolute values.

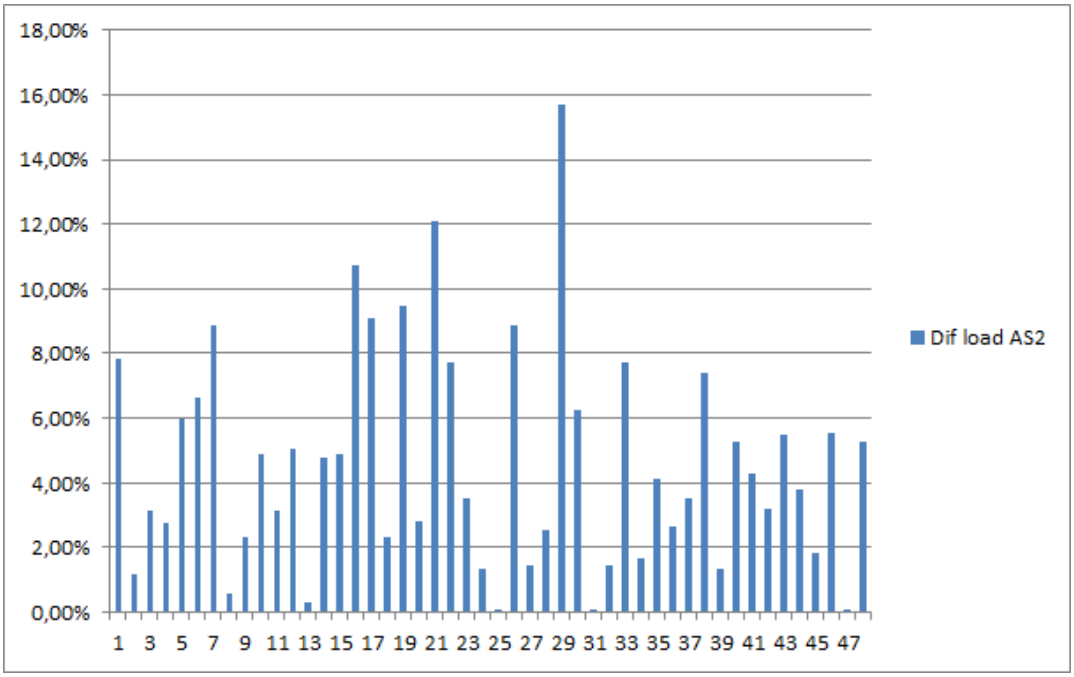


Figure 130. Total load balancing AS2.

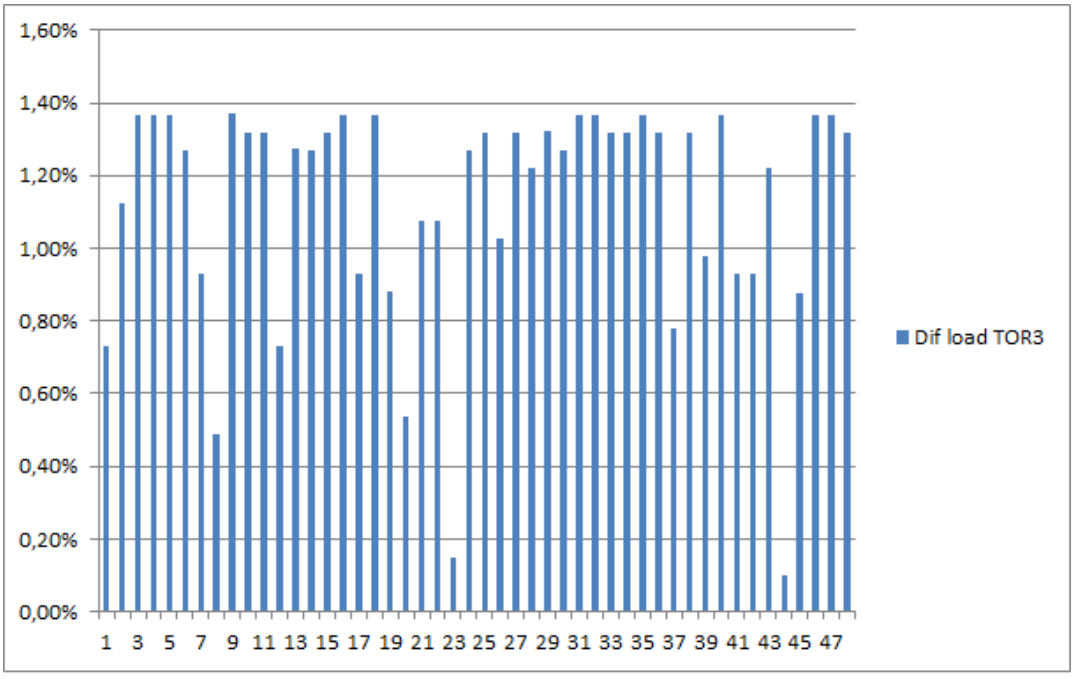


Figure 131. Total load balancing TOR3.

The results show a good distribution of bandwidth similar to the results obtained in D5.4[42] with the centralized resource reservation strategy. But now we will also show the results associated to each QoS. Figure 132 and Figure 133 shows the difference between the load associated to one QoS routed through a link and the load routed through the other.

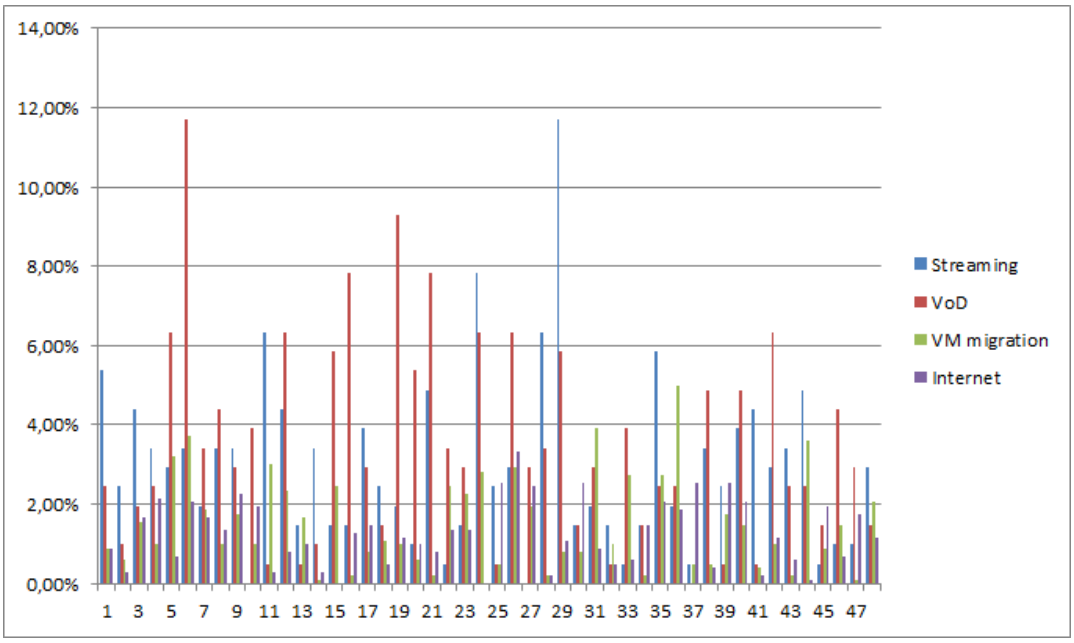


Figure 132. Difference in load balancing by QoS in AS2.

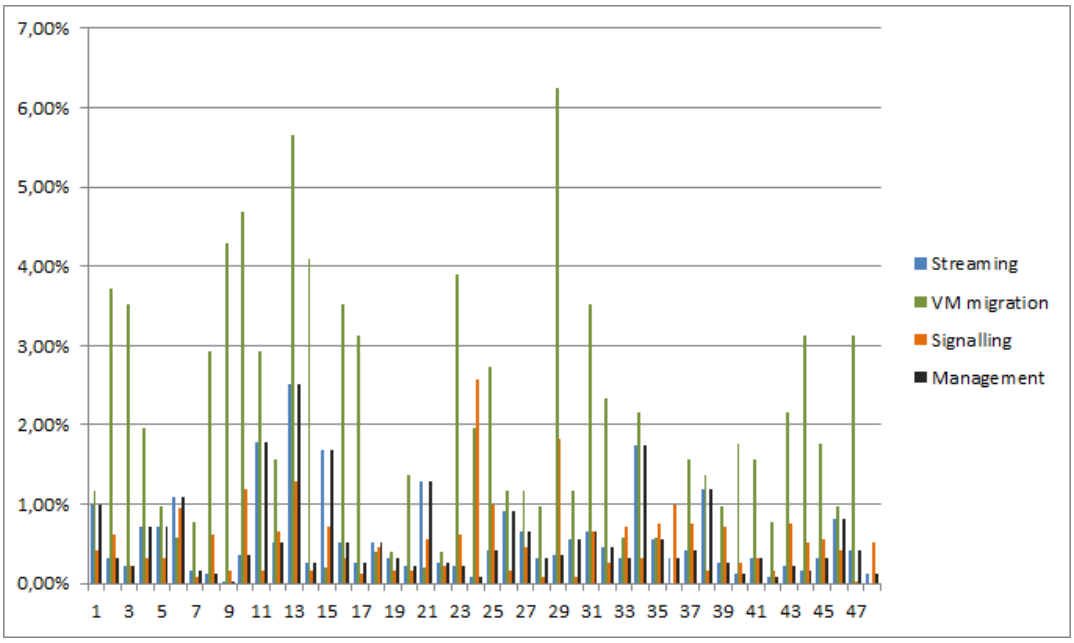


Figure 133. Difference load balancing by QoS in TOR3.

[42] https://wiki.ict-pristine.eu/wp5/d54/42-centralized-resource-reservation

The graphics show a good performance distributing the flows by QoS, being the difference mostly below to 5% of the link capacity and around 10% in worst case scenario.

Losses

For testing the dropping performance of the solution we will analyse the drop rate in the most congested nodes (AS1 and TOR1). The expected result is a lower dropping rate of high cherished packages over low cherished ones.

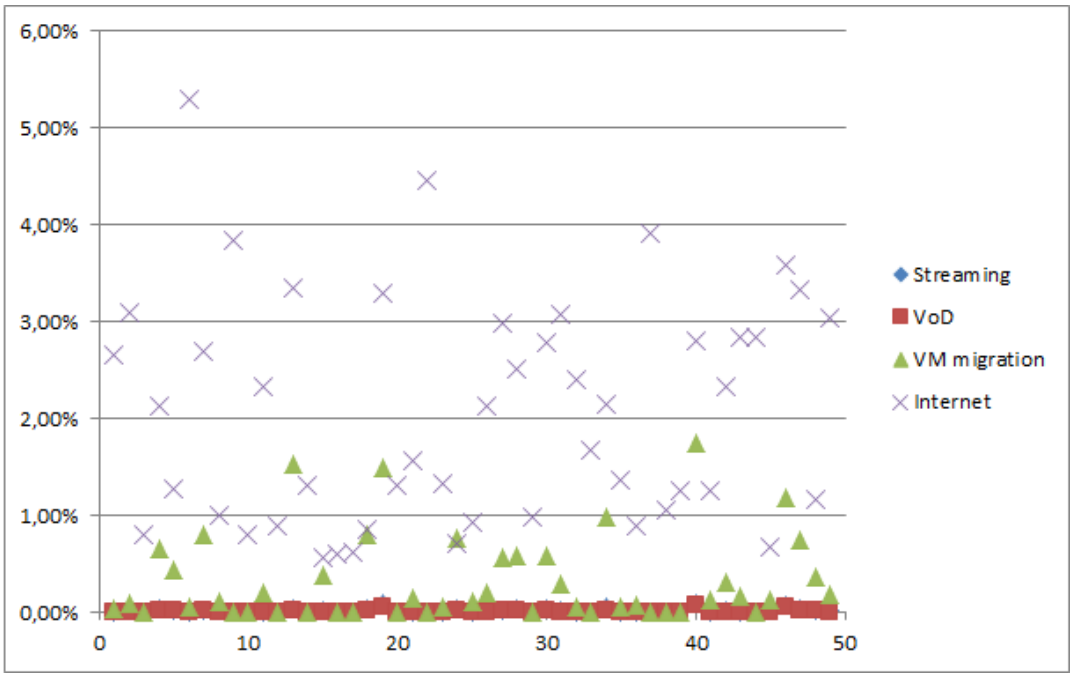


Figure 134. Package drop by QoS in TOR1.

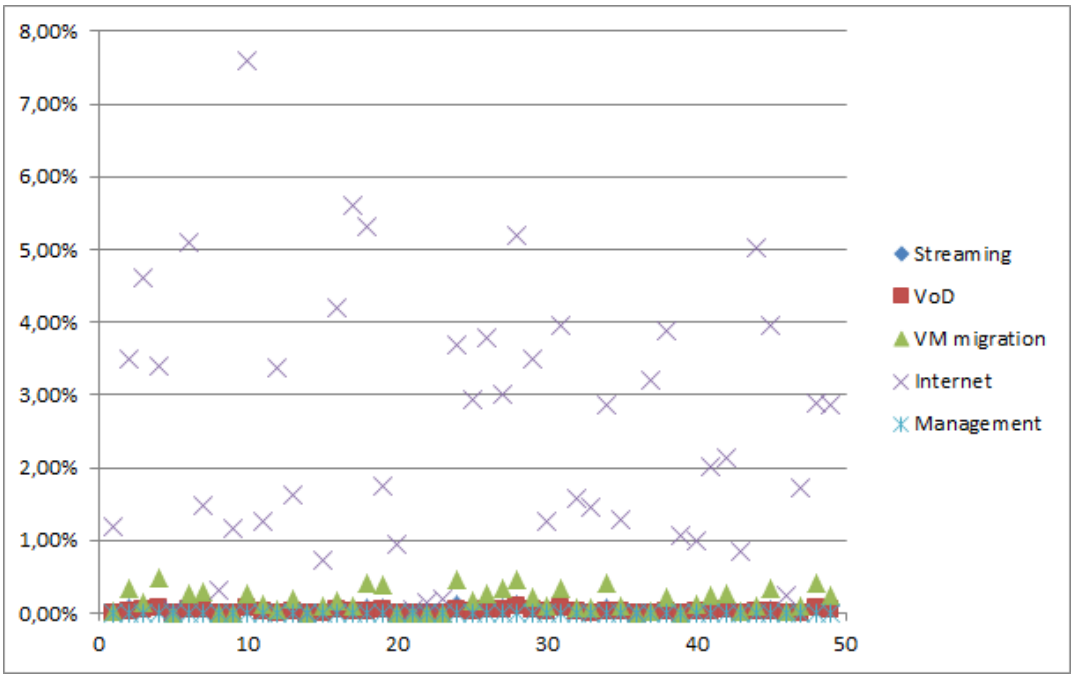


Figure 135. Package drop by QoS in AS1.

The results, showed in Figure 134 and Figure 135, are just as expected; more cherished QoS (Management, streaming, VM migration and VoD) are keeping a low dropping rate, while internet traffic is taking the most part of drops when necessary. To compare the results we repeated the experiment disabling the QTAMux scheduler obtaining the dropping rates showed in Figure 136 and Figure 137:

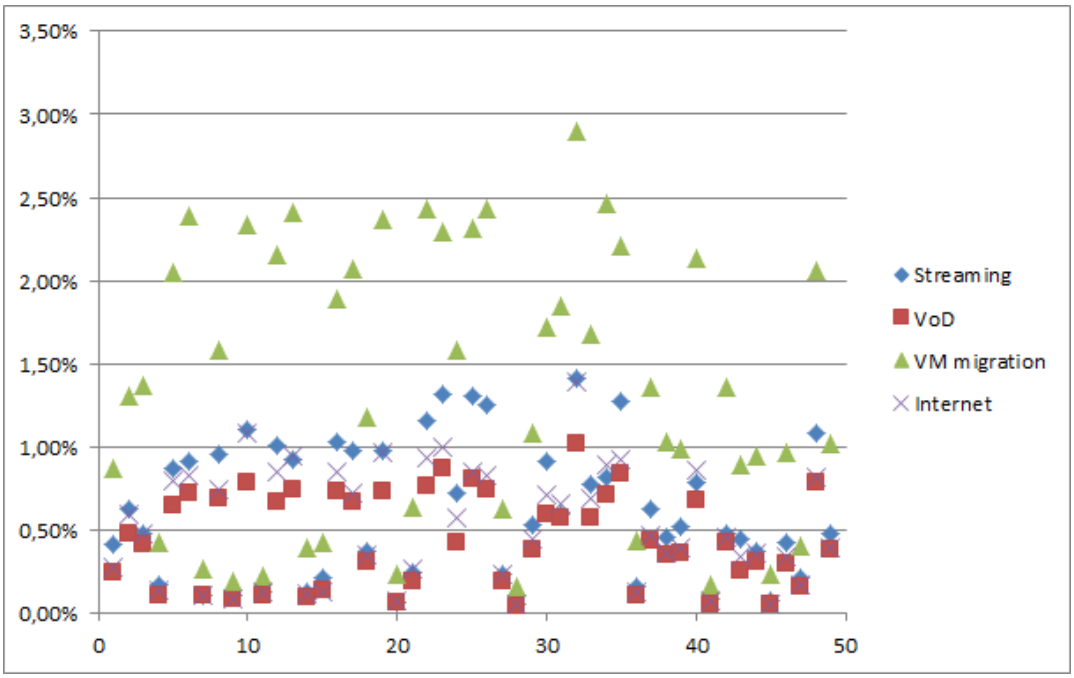


Figure 136. Package drop by QoS in TOR1 without QTAMux.

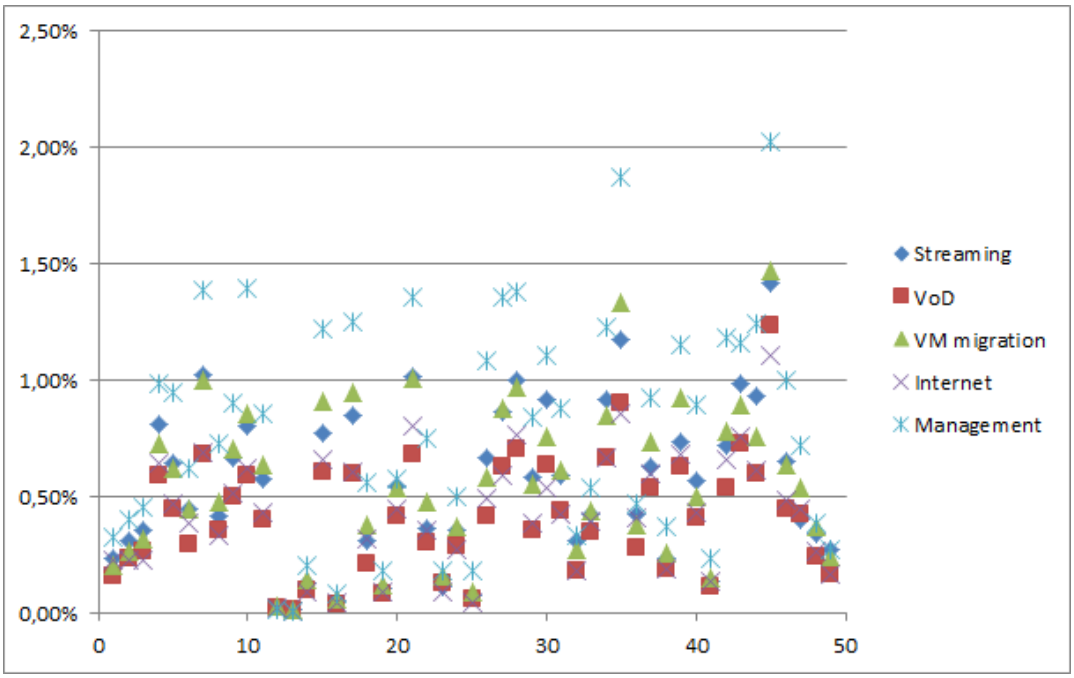


Figure 137. Package drop by QoS in AS1 without QTAMux.

For clarity we also include a table with average delay in the nodes TOR1 y AS1 in the fifty repetitions:

Traffic	Average losses
Streaming	0,0168%
VoD	0,0098%
VM migration	0,3287%
Internet	2,0566%

Figure 138. Drop rate in TOR1.

Traffic	Average losses
Streaming	0,0426%
VoD	0,0273%
VM migration	0,1748%
Internet	2,4268%
Management	0,0000%

Figure 139. Drop rate in AS1.

Delay

Finally we will study the delay experimented by each traffic type, for that propose we have obtained the average delay for each QoS. The results have been divided for internal and external traffic; the reason is that there is no

point in compare the delay of flows with different distance from source to destination. In the topology of the experiment the flows addressing the ISP network reach it in four hops, while internal traffic takes six hops to reach destination.

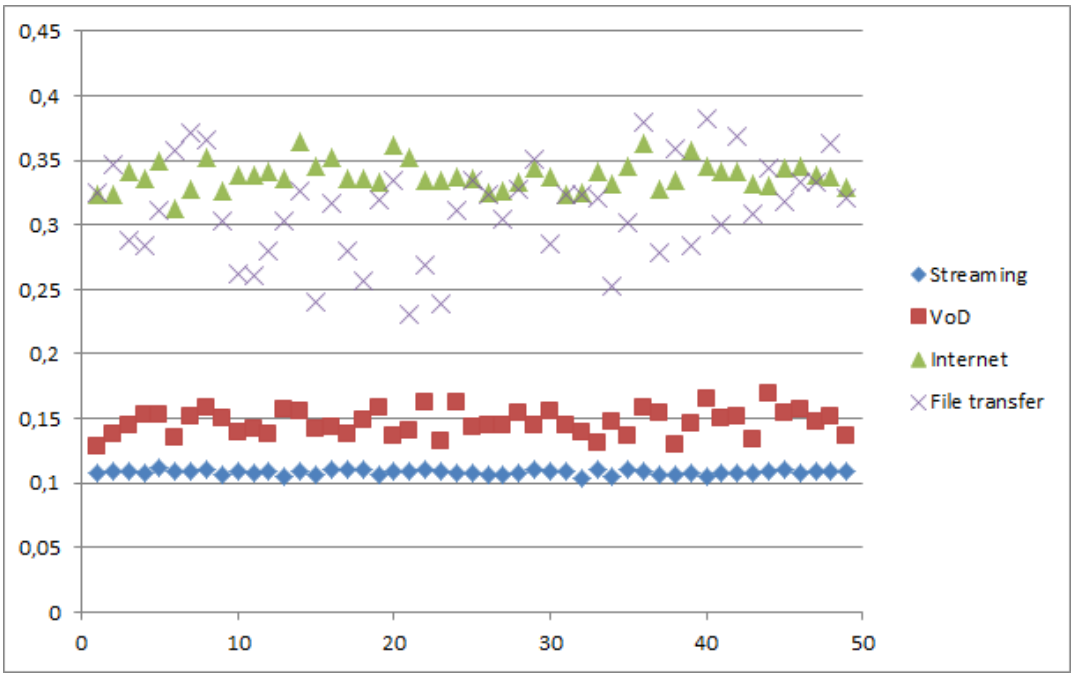


Figure 140. Delay of external traffic types in milliseconds.

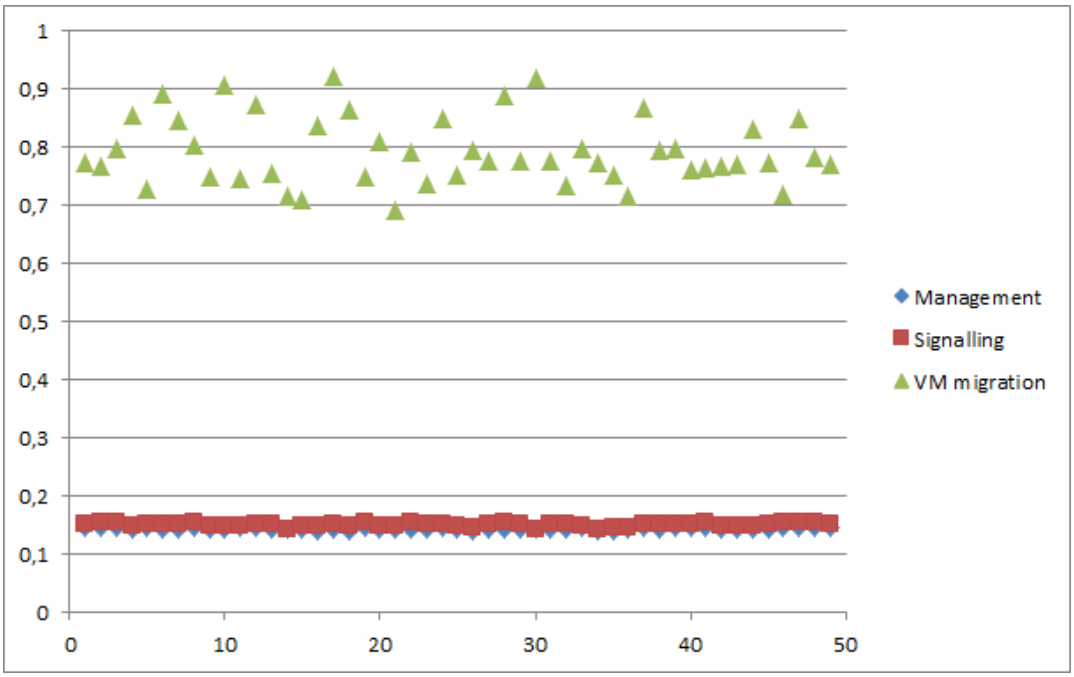


Figure 141. Delay of internal traffic types in milliseconds.

As was expected internal traffic suffers higher delay due to the higher number of hops from source to destination. Regarding the different types of traffic, the obtained results in Figure 140 and Figure 141 are reasonable,

higher priority traffic (management, streaming and signalling) clearly obtained lower values of delay. However what can seem unusual from the results is that file transfer traffic suffers similar delay than internet traffic, even when internet traffic has a higher priority. The reason is that internet traffic is sharing source server with streaming traffic that has a higher priority. This causes a higher queuing time than the file transfer traffic which is alone in the source server. To compare the results we also repeated the experiment without the QTAMux policy obtaining Figure 142 and Figure 143:

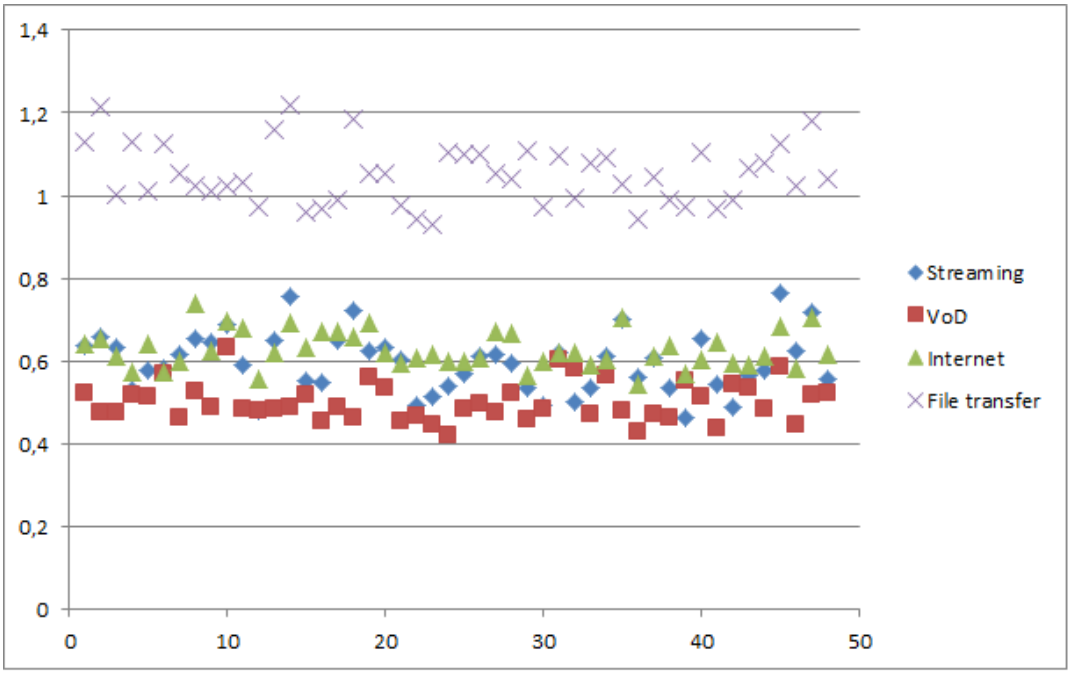


Figure 142. Delay of external traffic types in milliseconds without QTAMux scheduler.

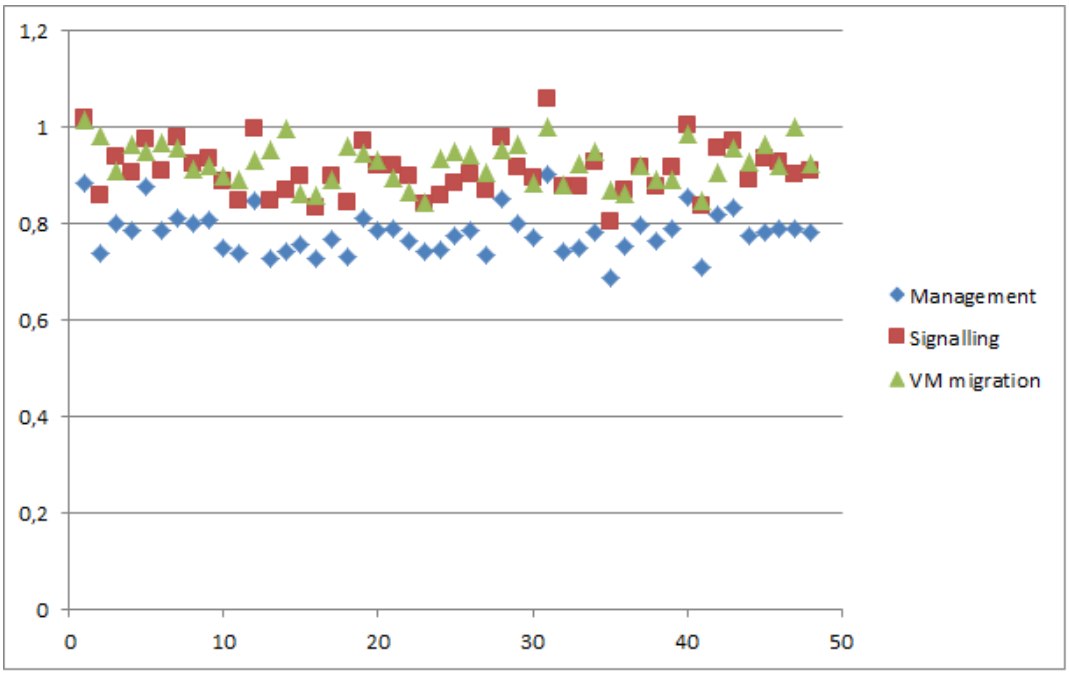


Figure 143. Delay of internal traffic types in milliseconds without QTAMux scheduler.

We also include a table with the average delay of the different types of traffic along the fifty repetitions.

Traffic	Delay
Streaming	0,108 ms
VoD	0,147 ms
Internet	0,338 ms
File transfer	0,312 ms
Management	0,143 ms
Signalling	0,150 ms
VM migration	0,794 ms

Figure 144. Delay by QoS.

6. Conclusions

6.1. Technical impact

The advanced experiments carried out during the second phase of the project have illustrated the benefits of the RINA architecture. Several of the RINA benefits demonstrated during this second cycle of experiments effectively apply to all three use cases. For example the performance isolation policies can effectively be used in both the data-centre and in the internet service provide use cases in order to enable safe and effective multi-tenancy.

6.1.1. Policies for Performance Isolation in Multi-Tenant Data-Centres

While resolving the congestion is a good feature for the network, because it allows to mitigate the effects of over-congested links, avoiding the congestion itself is another important objective for the network management. We showed how the same congestion reaction mechanism behaves on different condition and under different routing domains. Such routing policies can be made even smarter by improving the decision logic on the nodes, or by inserting a centralized actor which computes the best path for the destination. The IRATI stack allows us to quickly develop, configure and deploy such strategies in the network, allowing a flexibility that is not easily offered by the conventional network stacks.

6.1.2. Policies for Data Center TCP-like behavior

RINA allows us to implement our own version of a congestion avoidance mechanism and change the behavior of the nodes by changing a policy. We showed and measured the RED policy that is already available in the IRATI stack and compared RED with a DCTCP-like policy. The RED policy tries to mimics the TCP-like behavior in RINA. It implements slow-start and congestion avoidance mechanisms, thus, if there is congestion, the receiving node can signal back to the sender that the sender's transmission speed should be decreased. The DCTCP policy uses a similar approach with some modifications and enhancements. The window size is not halved, but it is decreased proportionally to the number of PDUs marked as congested. RMT tries also to control and keep occupancy of its queues at specified

threshold. This allows achieving higher burst tolerance, lower latency, and higher throughput.

6.1.3. Multi-Level Security

Multi-Level Security (MLS) is seen as a strong requirement for some organizations because without it organizations simply cannot perform their missions. MLS solutions for the current dominant networking architecture (e.g. IP), are well understood and developed, but with some issues regarding complexity and flexibility. Revolutionary networking architectures such as RINA are designed to migrate from IP-based networking, with the promise of less complexity and greater flexibility than is possible with IP. Embedding security controls (including MLS) in the network system architecture enables tight integration and inter-working of network functions and security enablers by dynamically configuring them. The MLS security control can be programmed to do a variety of protection functions through policies. This avoids, as of today's common practice the need of any further incremental updates and plug-ins of software and addition of extra hardware to the network for security reasons wherever it is required.

Here, the technical impact is specified qualitatively. That is, the protection and control of information are improved through use of SDU protection at any layer depending on the information granularity the network designer/ customer wishes to protect. There can be specific protection for each of the individual application flows using the same IPCP. Alternatively, the MLS solution is more scalable in terms of processing, as many application flows can be protected using the same IPCP flow.

RINA can sit and operate on top of existing networks (Ethernet, WiFi, TCP/IP, etc.) depending on the support of an appropriate shim DIF. This allows RINA to provide a flexible migration route for its use. The use of MLS can initially happen in Data Centres and in Service Providers' intra-domains, offering flexible and programmable MLS security controls. For proof of concept by using the developed SDK and IRATI stack, the MLS process has been implemented, integrated and verified in a real multi-node RINA testbed network using shim DIF over Ethernet. The tests verified that the proposed MLS solution complies with the expectation and design specification.

6.1.4. Key Management

Any security system is only as strong as its weakest link; without an effective, usable and scalable solution, key management can easily be that weakest link. Thus, an effective key management system meeting the objectives set out in D4.3 is essential to enable the implicit security benefits of RINA to be usable in large-scale real-world deployments. The demonstration shows that the key management architecture developed in WP4 is workable in practice, and makes credible RINA's claims to offer a network and distributed application security framework that is superior to current systems.

6.1.5. Datacentre networking

The datacentre (DC) experiment has provided a real use case of how RINA can be leveraged for slicing the DC resources and assigning individual partitions to the different tenants of the DC. These tenants get security and performance isolated connectivity domains, which brings together all the computer and storage resources under their control. For the DC provider RINA offers a great degree of flexibility via configurable policies (for multi-path forwarding, congestion management, resource allocation, etc) while still keeping a clean architecture with recursive building blocks that simplify the management and operation of the DC infrastructure. Virtualization and slicing in RINA are part of the architecture and therefore no additions have to be done in order to support these features.

6.1.6. Policies for QoS-aware Multipath Routing

These interesting results show how the use of the QTAMux improves the behaviour of the network in terms of limited packet loss and delay. This work will be continued with more complex traffic profiles, QoS cubes definitions, and taking full advantage of the possibilities offered by the Policer/Shapers during ARCFIRE.

6.1.7. NFV and Service Chaining

In the internet service provider use case we demonstrated how RINA can allow each DIF to support a network service composed of a variable number of Virtual Network Functions while also precisely specifying which portion of the traffic should be processed by a certain set of VNF and

in which order. In this scenario the policy-based nature of RINA allows each Network Service to use the routing, congestion control, and security policies that best suits the needs of the Network Service.

6.1.8. Integrated security

This experiment shows that DIFs are securable containers - different authentication and SDU protection policies can be plugged in to support authentication, confidentiality and integrity of data travelling through the DIF. Moreover, the clean design of RINA allows policies to be plugged to any DIF and evolve as attackers get stronger or more sophisticated (evolve to use stronger algorithms, longer keys, etc.). DIFs also provide isolation, allowing the provider to design a network that exposes the minimum set of layers and systems to customers and peers. Access control policies can be put in place to protect the operations on RIB objects as well as the ability to allocate flows to the members of a DIF/DAF. These access control policies can also be applied to any DIF, making the overall architecture very flexible and cost-effective.

All in all, the RINA architecture provides a foundation for a more effective and efficient network security; however it is not a silver bullet: properly implemented software (Operating Systems, applications and RINA implementations), security of physical systems and proper key material generation and distribution techniques are still critical to guarantee the security of a distributed computing environment.

6.1.9. QoS-aware multipath routing

With this joint experiment the QTAMux scheduler policy has proven to perfectly fit in a multipath datacenter scenario, allowing a graceful degradation of the traffic in the cases that the load overcomes network link capacity. The combined experiment has also shown that using QoS-aware techniques and QTAMux scheduling together an optimal use of link capacity can be achieved. Additionally, this results in a better SLA assurance, since it is possible to define different priorities of PDU delay and losses that can guarantee the compliance of such defined network KPIs. The simultaneous use of previous policies, although not initially conceived to work together, satisfy every QoS requirements defined in a datacenter to provide multiple services and traffic types. The carried out experiments were only possible thanks to the RINA policy-based architecture that allows

using policies developed as individual units together with the modification of a single line in a configuration file.

6.2. Feedback towards development

6.2.1. Improvements already implemented and released

The development, debugging and testing of policies associated to the second cycle of experiments uncovered a number of bugs in the RINA implementation as well as some changes required in the SDK. The following points summarize the feedback towards the RINA prototype development (which have already been addressed and released in the most up to date development branch, *pristine-1.3*[43]).

6.2.2. Open issues

Open issues currently reported but still not fixed:

- ECFP mechanism has been reported to have some misbehavior in certain extreme conditions. Such conditions are usually triggered by custom behaviors that have been introduced in some policies, like when introducing some kind of uncommon forwarding/routing behavior.
- Fragmentation capabilities should be introduced in the stack internal mechanism to overcome some issues which arise especially when using the shim over Ethernet. In the case where the PDUs are too big for the MTU, packets are dropped, thus making the communication impossible between two applications. This is also true when the management PDUs with RIB status are exchanged between IPCPs, and the update is so big to exceed the ethernet default limit of 1500 bytes.
- Under some conditions the stack stops responding and generates oops. Such events have been rarely reported and thus must be carefully examined in order to isolate why there is such behavior and how to mitigate it.

6.2.3. Design improvements

Some design improvements have been identified and integrated in the final iteration of the prototype in version 1.4:

[43] https://github.com/irati/stack/tree/pristine-1.3

- Policies have been improved and adapted to the new version of the stack prototype. Now they offer a more stable behavior and configuration thanks to sysfs.
- The tools which allows testing of the prototype have been improved and they are increased in number: now is also possible to use existing applications which run over IP on RINA with minimal reconfiguration, thus extending the applications compatible with IRATI from a few to all those are in the wild.
- Overall performance has improved by revisiting the locking and timing mechanisms inside the kernel part of the prototype, especially in the RMT and EFCP components. This applies to both virtualized and real environments.

6.3. Expected business impact

In the following section we report the final business impact. In the first subsection, the analysis process followed is explained. The following subsections account for the different PRISTINE use cases, namely distributed cloud, datacenter networking, and network service provider. For each use case, the expected business impact is described.

6.3.1. Analysis approach

To translate the above technical impact to its implications towards the business context, PRISTINE once again returned to its three industry use cases. An analysis process was adapted for a business impact analysis, adapted from Alex Osterwalder's "Business Canvas" but more focused on stakeholder/customer analysis and value proposition development:

1. Value Chain Analysis: Through market analysis, a validation of the value chain revolving around the three use cases is provided, identifying the various vendors, providers and supporting roles that follow the transfer of enabling and value-added products and services all the way to the end user. The primary goal is to identify a) who is the adopter/customer of a RINA solution in that use case in particular, b) who is the provider (service) or vendor (product) of the solution, and c) the other supporting actors that would experience added value, disruption, opportunity, etc.
2. Stakeholder Analysis and Customer Typology: Based on that value chain analysis, a more in-depth analysis is provided of relevant

stakeholders, focusing exclusively on the customer and provider of the RINA solution in the use case. In relevant cases, such as the Datacenter Networking use case, a typology is presented of the customer when significant differentiation exists.

3. Customer Jobs: The specific jobs of the identified customer stakeholder are examined. These customer jobs represent the priority activities that the customer carries out to ensure their own success metrics, whether customer-oriented (their customer, the end-user, etc.) or internal operational needs. This primarily includes “functional jobs”, in the sense of key activities that define their business objectives. However, “strategic jobs” are also included, where the focus is on positioning, partnerships, etc. that has a more intangible yet important role to play in their priorities. Finally, “supporting jobs” are at times significant to our exercise, such as those customer teams in charge of selecting and procuring new technology.

4. Customer Pains: The analysis then shifts towards recognizing the customer “pains” that manifest themselves from these priority jobs. As the end goal is to answer these problems with a solution and its value proposition, these must be tangible and measurable. If not, there is a risk of creating a solution that is vague in benefits, does not sufficiently tackle their problem, or does not apply in the manner that the customer measures it.

5. Customer Gains: Priority gains are then examined, covering what the customer is looking for, often in relation to their identified “pains”. These can cover those gains that are required, expected, desired or unexpected. For example, required for the most basic and core needs of that customer. Expected gains are a next step; beyond what is required but essential for a developed solution to answer. Desired gains refer to customer preferences that could prove important for a competitive solution. Finally, unexpected gains refer to what goes beyond customer preferences; perhaps related to an added value that they did previously recognize, but could so upon consideration.

6. Value Proposition: Understanding the above customer profile from the sections above, RINA’s value proposition in the use case can then be directly addressed. The value proposition(s) address directly the customer pains, customer gains, or both. Just as the aforementioned, the value proposition must be concrete and measurable. It should also link

and translate well from the technical impact of the D6.3's experiments and technical impact; in this case, in the context of each use case.

7. Market Feasibility: Market forecasts are then consolidated for the individual use case contexts. A Strength, Weakness, Opportunity and Threat (SWOT) analysis is provided for each use case, analyzing the feasibility of a RINA solution towards that scenario, based on value proposition, competing technology and forecasts in the market.

6.3.2. Distributed Cloud Use Case

During the PRISTINE project, Nexedi has gathered various ideas that can be used to extend current re6st network. We could solve as part of PRISTINE the problem of mesh scalability by using a recursive architecture directly inspired by RINA, even though it is not based on RINA itself. Thanks to this recursive architecture, re6st network can be deployed over much larger communities. VIFIB - Nexedi's wholly owned subsidiary - will thus provide before the end of 2016 experimental RINA connectivity service as well as RINA orchestration service.

Stakeholder Overview and Value Chain Analysis

The distributed cloud use case is an instance of the hyperconvergence movement that is transforming IT infrastructure business from a business of compartmented market segments (computer, networking, storage, monitoring, etc.) into a single converged market in which "**every thing is service deployed on generic hardware**". Networking service (LTE, IPv6, RINA, etc.) takes the form of routing daemon deployed on an server (x86, ARM). Storage service takes the form of a block storage daemon deployed on a generic server, etc.

The distributed cloud use cases is thus based on the following value chain:

- C1 - **Hosted server providers**
  - C1.1 - Self-hosting case
    - C1.1.1 - server hardware vendors
    - C1.1.2 - real-estate providers (office or datacenter)
    - C1.1.3 - Internet access or transit providers
  - C1.2 - Outsourced-hosting case: baremetal server or VM hosting companies

- C2 - **Hyperconvergence providers**
  - C2.1 - Hyperconvergence as a service providers
  - C2.2 - On-primise hyperconvergence providers
- C3 - **System integrators or end-users**

System integrators or end-users (C3) are the primary customers of the RINA solution that is deployed by hyperconvergence provides as a service on top of a generic infrastructure, based or not on RINA. Some hosted server providers (C1) may also be customers of the RINA solution, but this is not required. Some hyperconvergence providers (C2) may also be customers of the RINA solution, but this is not required.

In this value chain, system integrators or end-users (C3) rely on hyperconvergence providers (C2) to capture the value of hosted server providers (C1).

Value captured through hyperconvergence providers is shared between hyperconvergence providers (C2) and system integrators or end-users (C3).

The existence of an open source solution for hyperconvergence (SlapOS[44]) ensures that not all value will be captured by hyperconvergence providers (C2) since system integrators or end-users (C3) may deploy their hyperconvergence on premise (C2.2) if costs are lower than with hyperconvergence as a service (C2.1).

Due to competition traditional infrastructure, industries see their profits fall due to this evolution: server hardware vendors (C1.1.1), real-estate providers (C1.1.2), internet access or transit providers (C1.1.3) as well as outsourced hosting providers (C1.2) no matter whether they rely on baremetal or virtual machine.

Stakeholder Analysis and Customer Typology

System integrators or end-users are the primary customers of the RINA solution. They use RINA because they believe that RINA has some advantages over IPv6 or IPv4. Those advantages are currently unrelated to the distributed cloud use-case. However, in the future, we expect that using *RINA rather than IPv6 or IPv4 can bring as primary advantage the*

[44] http://community.slapos.org/

ability to customize routing policies per instance of mutually orchestrated services, something that is currently difficult with IPv6 routing, even with source based routing.

Hyperconvergence providers can differentiate by providing RINA service deployment by default, instead of forcing to system integrators or end-users to develop RINA deployment profiles. This is the same as what Amazon does with Amazon Machine Images (AMI).

Hosted server providers can differentiate by providing native RINA connectivity. This way, RINA deployment does not need to depend on encapsulation into IPv6, IPv4, Ethernet, etc. and can offer better performance or better implementation of policies.

Customer Jobs

System integrators or end-users are the primary customers of the RINA solution.

Functional Jobs

Functional jobs consist primarily of developers who define devops scripts that are used as input by hyperconvergence providers to deploy a set of mutually orchestrated services and RINA policies across a global distributed cloud.

Strategic Jobs

Strategic jobs consist of engineers looking for hosted servers or hyperconvergence providers that are ready to support and maintain the deployment of RINA network services.

Supporting Jobs

Supporting jobs consist of engineers looking for new applicable policies in certain business context and new suppliers of hosted servers that can support new or existing policies.

Customer Pains

The No 1 pain for customers is the absence of any form of maintenance or commercial support for a RINA compatible infrastructure, which puts

all the burden of creating Hyperconverged RINA service deployment profiles on the user or system integrator, with no help from the hosted server provider or the hyperconvergence provider. Lack of support increases deployment time to multiple days of hyperconverged profile development.

The No 2 pain for customers is the lack of resiliency of the RINA network over the distributed cloud. Lack of resiliency can cause frequent downtimes that break the orchestration of services and are simply incompatible with the use of a distributed cloud for mission critical applications.

The No 3 pain for customers is the absence of native support of RINA policies by the hosted server provider or the hyperconvergence provider. Lack of native support of RINA policies can prevent in some cases from deploying certain policies at all.

Pains are thus summarized with 4 tangible criteria:

- how long does it take to deploy a hyperconverged system that includes RINA network service?
- what is the average frequency and duration of RINA connectivity downtime?
- what is the percentage of routes that are not meeting the desired policy?
- which % of available RINA policies can be supported reliably by a hyperconverged deployment?

Customer Gains

Expected gains for the customer are:

- ability to deploy a hyperconverged system that includes RINA network service in less than 5 minutes
- average frequency of RINA downtime of less than once a month, average duration of less than 5 minutes
- 0% routes do not meet the desired policy for longer than 5 minutes
- 100% of available RINA policies supported by the hyperconverged system

Value Proposition

Nexedi's wholly owned subsidiary will provide before the end of 2016 experimental RINA connectivity service as well as RINA orchestration service. This new experimental service will provide a public platform to networking researchers willing to deploy and orchestrate services based on RINA.

VIFIB provides the following value:

- ability to deploy a hyperconverged system that includes RINA network service in less than 5 minutes
- besides IRATI's kernel bugs, downtime of less than once a month, average duration of less than 5 minutes
- support of all RINA policies that can be supported over Ethernet LAN

Market Feasibility

We believe that VIFIB, thanks to PRISTINE, can generate short term revenue for end-users looking for a RINA testbed:

- public institutions willing to conduct RINA-based research
- organisations looking for RINA deployment on bare-metal server
- organisations looking for RINA quick deployment and orchestration
- organisations willing to combine IPv6 and RINA on the same platform

We expect that PRISTINE can thus generate additional revenue for Nexedi group of at least 200 K€ yearly before 2018, and thus finance two additional full time employee.

VIFIB may generate long term revenue with system integrators, but no short term revenue.

VIFIB main competitors are public clouds (ex. Amazon), dedicated servers (ex OVH), self-hosted PC (ex. Intel NUC) and research teams that develop RINA and provide free testbed to other research teams. Market may prefer using those competitors rather than VIFIB.

Strengths	Weaknesses
• Automated RINA VM deployment	• Not all RINA policies supported

• Automated RINA bare-metal deployment • Automated RINA service orchestration • Native support of **all** public clouds • Native support of self-hosting • Native support of Ethernet LAN • Low latency networking • Routing resiliency • Scalable mesh architecture	• No resiliency to unfair routing attacks • Small pool of available servers
Opportunities	**Threats**
• R&D on fair routing	• No interest in any automated RINA deployment • Research teams prefer to work with research teams

The fair routing issue remains unsolved after PRISTINE project. We view it as an opportunity for future research on network resiliency based on VIFIB's RINA testbed.

6.3.3. Datacentre Networking Use Case

Stakeholder Overview and Value Chain Analysis

A simplified value chain is presented below, representing the main actors that would be involved in a RINA solution scenario for the Datacenter (DC) networking use case.

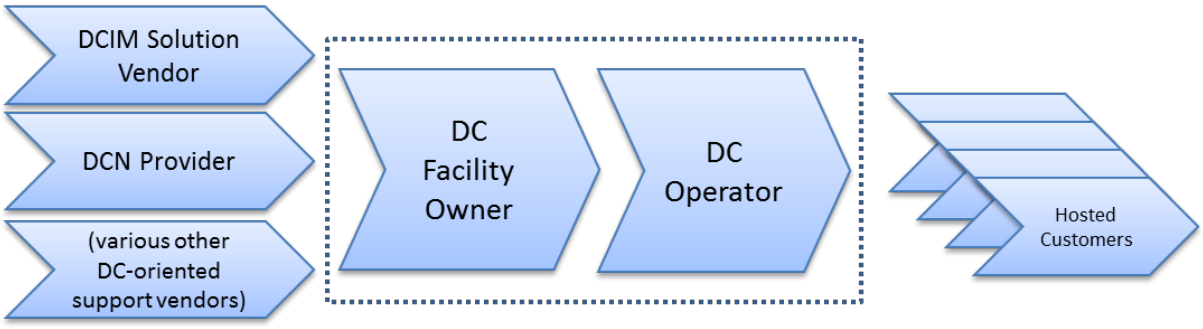


Figure 145. Datacentre Value Chain

- DC Operator: The operator of the datacenter, and the customer of a RINA solution.
- DC Facility Owner: In this simplified model, we assume the DC operator is the single tenant of the DC facility. However, in more complex models,

a facility could include colocation of several operators, renting space from the owner of such a facility. They support their tenants with the auxiliary cost items of their operations, such as facility cooling systems, networks, and sometimes even the actual hardware.

- Datacenter infrastructure management (DCIM) solution provider: supplying their DC operator customers with integrated software suites (e.g. full facility management and monitoring) or stand-alone solutions (e.g. power management solutions, sensor system, etc.). This actor could be a vendor of a RINA solution. An intermediary systems integrator could exist between vendor and DC operator, but for this model, it is assumed the integration will be done between vendor (DCIM Solution Vendor) and customer (DC Operator).
- Data Centre Network / Connectivity (DCN) Provider: Supplies the network architecture and operating bandwidth between interconnected data centers. This actor can also be the entry point for a RINA solution (see NSP use case).
- Hosted customers: The operator's serviced user, whether of the same organization or an external customer. Extremely diverse depending on the datacenter's profile, e.g. public cloud provider vs. a large enterprise's private DC. These could include anything from an application service provider with its deployed SaaS and end-user customer data, to the underlying infrastructure of a network operator.

Stakeholder Analysis and Customer Typology

As mentioned in the value chain summary above, the RINA DC Networking use case customer, the DC Operator, can be differentiated in the market.

We can distinguish 4 different types of DC operators:

- Enterprise Datacenter (internal): Enterprise data centers typically serve internal clients in a company behind a firewall. As opposed to Internet data centers which host a few applications for many users, enterprise data centers typically host many applications that serve an enterprise for a limited number of users.
- IT and Cloud Datacenters (public, outsourcing): Many of them consist of more than one datacenter located in different buildings and composed of thousands of servers. As such, datacenter networking interconnection

is a key priority, and QoS metrics for the customer are crucial. Generally speaking, these are the largest operators and often have the most innovative technology. However, as they are customer-facing and have high interconnectivity requirements between sites, it is also one of the more mission-critical scenarios that would require a longer roadmap for adoption.

- Datacenter Facility Owner/Operator (outsourcing): These companies, merging the roles DC Facility Owner and DC Operator, often rent part of their buildings for a co-location model. They generally are of very high quality but not as innovative as the ones mentioned above, relying more on a commodity model.
- Co-located operators, do not own their premises, but rent a complete section of the facility or only some square meters inside buildings owned by specialized DC Facility Owners.

Customer Jobs

To better understand the jobs and priorities of the DC Operator as a customer, we focus on the Enterprise Datacenter scenario. Although outsourced and cloud-oriented DCs is a rising trend, in RINA's roadmap a system integrator delivering a RINA-enabled datacenter holistic network solution to a single enterprise could be the most realistic at this point in time.

However, even taking into account differentiation, many of these customer (DC Operator) pains, jobs and priorities are common among different profiles.

Functional Jobs

A Functional Job is when the customer is trying to perform or complete a specific task, or solve a specific problem. This could be customer facing (provide good service), or internal (keep operations efficient). Examples for the DC Operator include:

- Ensure security and integrity of their customer's data
  - Get physical, control physical access to the datacenter.
    - Physical security may include:

        - fenced-off campus
        - badge access to the main building and datacenter
        - a guard who escorts visitors
        - key card admittance to rooms
        - video surveillance of the data center
        - and locked cages for servers, depending on the sensitivity of the data that they contain
    - Establish secure zones in the network
    - Lock down servers and hosts
    - Scan for application vulnerabilities
    - Coordinate communication between security devices for visibility into data flows.
- Ensure that the network access is always available
- Maintain optimized DC network inter-connection between sites.
- Ensure that the facilities hosting the servers meet the requirements in terms of physical access, cooling, etc.
- Provide service and SLAs to their customers that cover a wide range of performance metrics: e.g. availability, latency, etc.
- Ensure security and integrity of their customers' data.
- Offer backup solutions and disaster and recovery procedures.
- Provide added value offering (e.g. software) that for customers who lack high IT skills and resources.
- Marketing and operation of sales force.

Strategic Jobs

Strategic jobs refer to the customer activities that align to competitive power, status, strategic positioning, etc., including how they are perceived by others. For the DC Operator, examples include:

- Form and maintain strategic alliances with key stakeholders: storage vendors, network operators, etc. that position the DC Operator ahead in their sector.

- Adopting cutting-edge technology, using innovative solutions that helps position themselves among their peers and aids marketing towards their customers.
- "Going Green", for moral reasons and also to conform to their customers' environmental policies. In the case of the DC Operator, being green also means driving down OpEx, as energy consumption is a large factor of their carbon footprint.

Supporting Jobs

Supporting jobs are auxiliary jobs that help sustain the customer's main jobs and meeting success metrics. For the DC Operator, a relevant example for our use case include:

- Procurement of DCIM solutions: comparing between options and purchasing.

Customer Pains

Customer pains can refer to:

- Undesired outcomes, problems and characteristics: something does not work, negative side effects, lack of quality result or service, etc.
- Obstacles: prevents or slows down a job, e.g. causes inefficiency

Examples for the DC Operator include:

- Service interruption and SLA shortcomings (it can be measured in terms of % of unavailability, total time a service has been unavailable during a period, or number of individual interruptions). There are many reasons that can lead to this unavailability: failure in the air conditioning that makes the facilities overheated, power interruption, hardware failure in a server, etc. Fault tolerance is a large priority for the DC operator.
- Access network and DC interconnectivity interruptions, that affect their overall operations.
- Inefficient OpEx: anything related to the cost overrun of maintaining their operations. Capital costs account for nearly two-thirds of the purchasing decision for networking equipment. But over the life of the gear, the total cost of ownership (TCO) is dominated by ongoing operational costs – both administration and maintenance. Many TCO

models exhaustively look at all sources of expense, but it is also important to note the key drivers behind OpEx.

- Shortened hardware lifecycle: implying the need to upgrade or procure new hardware, raising their CapEx.
- Increasing infrastructure complexity: according to service providers and customers, datacenter technology is moving at a pace that is difficult to keep up with. IT commodity features are evolving so fast that users are not able to keep up and understand all the features in these products or how to use them.
- Performance issues: most data centers have bottleneck areas that impact application performance and service delivery to IT customers and users. Possible bottleneck locations include servers (application, web,file, email and database), networks, application software, and storage systems.
    - Data center performance bottleneck impacts include:
        - Under utilization of disk storage capacity to compensate for lack of I/O performance capability.
        - Poor quality of service (QoS) causing service level agreements (SLA) objectives to be missed. The performance metrics to be evaluated for a Data Center Operator are the following ones:
            - Reliability. Reliability means ability of the system to perform its task and function under stated conditions for a specified time. It is calculated by measuring Mean Time Between Failures (MTBF) and Mean Time To Repair (MTTR).
            - MTBF is typically represented in units of hours. The higher the MTBF is, the higher the reliability.
            - MTTR is the expected time to recover a system from a failure. The higher the MTTR, the worse off a system is.
            - Availability. Availability refers to the time or proportion of time for which a system is available and operable. It is represented by ratio of system uptime to total time:
                - Availability = System Uptime / Total Time (Uptime + Downtime)
            - Latency: The time between execution request and completion: <50ms
            - Jitter: Variations in delay of packet delivery: ≤ 3 ms

    - Packet loss: Too much traffic in the network causes the network to drop packets: ≤ 0.3 percent
- Premature infrastructure upgrades combined with increased management and operating costs.
- Inability to meet peak and seasonal workload demands resulting in lost business opportunity
- Data security/data loss. With the advent of cloud computing, rich Internet applications, service-oriented architectures and virtualization, data center operations are becoming more dynamic, with fluid boundaries. The shift toward a new computing environment adds layers of complexity that have broad implications for how information technology managers secure the components of a data center to protect data from malicious attack or compromise. Organizations need to monitor that all data center operations interact correctly and that each element of the data center is secure.
- Scalability. Scalability of the datacenter is the ability to construct and expand a datacenter with simple, repeatable designs that can accommodate increasing traffic or new devices without impacting applications and workflows. Scaling has to be linear for both performance and cost. Achieving this goal in the data center has several components:
    - Scaling up the performance of single systems, as this allows more price/performance linearity.
    - Extending or stretching the addressing of a network using open standards to embrace workload mobility across increased geographic distances and larger datacenter facilities.
    - Scaling the manageability and operational capacities of the network infrastructure in the datacenter network so that administrators can effectively provision and manage a larger number of systems.
    - Topologies must continue to self-organize and self-heal while converging quickly in the event of link or node failure.
- Power consumption. Cooling costs

Customer Gains

Whatever the service, data centers allow customers to better manage their IT landscape while:

- Reducing operating cost.
  - Resources on demand and according to the needs of the business, paying only for the resources that actually are in use (elasticity and flexibility).
  - Outsourcing services with a high level of technical support and team of experts in various areas: virtualization, operating system, security, database, storage, backup, and document management.
- Increasing safety, durability and reliability
  - Equipments and data in robust physical infrastructure, secure and resilient.
  - Data protection services aiming to ensure business continuity
- Agility and speed
  - Offering the ability to deploy services and applications rapidly and manage them easily.

Value Proposition

The direct adopter of RINA technology would be the DC operator.

As the chief stakeholder of the Datacenter Networking use case, DC operators, regardless of profile, look for common improvements in their costs, operation and service around a multitude of common issues. These include traffic management (within and between DCs), energy consumption, performance, reliability, hardware optimization and longevity, network scalability and security, all of which have spurred adoption for software-defined management of the network and a heavier reliance on virtualization.

RINA's value proposition for DC Operators core business is the following:

- Lower OpEx (operational expenditure) with better resource & network optimization, traffic management, etc.
- Lower OpEx and environmental footprint with better energy consumption (e.g. Power Usage Effectiveness, PUE). Energy is a huge cost of any DC's operating budget, as well as an environmental issue that receives pressure from regulation and industry certification, including Green IT.

- Lower CapEx (capital expenditure) with better use of existing hardware and future procurement of hardware with better longevity and modularity (e.g. more generic hardware with more reliance on software-defined management)
- Increase in performance, reliability and QoS towards both their hosted customers between other connected DCs, fulfilling internal or SLA-related metrics, and increasing competitive standing.

Market Feasibility

The feasibility of a a RINA solution in the market of datacentre networking is analyzed through an Strength, Weakness, Opportunity and Threat (SWOT) analysis:

Strengths

- Flexible migration. RINA can operate on top of existing networks. This will facilitate the adoption of the technology by the differente DC Operators.
- Scalability. DC Operators will be able to scale up or down their networks in a less complex and more flexible way than with IP technology.
- Explicit congestion notification and vertical pushback between layers enables a faster and more efficient response to congestion than the approach used currently.
- Optimal use of link capacity
- Better SLA assurance

Weaknesses

- Multiple domains, possibly based on different technologies, will be required to interoperate each other. This will demand the establishment of specific agreements.
- It is a new technology. It can be happen that some operators prefer to stay in their comfort zone and not to adopt it until it is more mature or others previously implanted it.
- DC Operators will need to invest in recruiting and training the staff.

Opportunities

- Develop new business cases and relationships between stakeholders.

- Performance isolation policies enable safe and effective multitenancy.
- Reconfigurability and fast network adaptation

Threats

- The potential appearance of competitive entirely new technologies (alternative to RINA) is always possible, however this is quite unlikely considering the required effort.
- Due to the fact that it is possible to share physical and virtual resources, security and privacy issues are key aspects to be guaranteed.
- The technical community with extensive knowledge in this new technology is restricted.

6.3.4. Network Service Provider Use Case

Stakeholder Overview and Value Chain Analysis

The goals of this use case are to investigate and trial the benefits of the use of RINA technology by a Network Service Provider (NSP), and to analyse RINA as a essential component of the Network Functions Virtualization (NFV) concept within an operator network.

The traditionally value chain would be that of equipment vendor supplies network hardware/software to the service provider, but in the world of network virtualization and the transition to "software networks", the lines between the vendor, systems integrator and service provider are becoming quite blurred. In fact it may not strictly be correct to call it a *value chain* as it is increasingly evolving into a value network consisting a series of inter-twined value chains with multiple entry and exit points, where some nodes are simultaneously involved in more than one value chain. The result is a highly complex and competitive telecommunications market, where companies compete with not only competitors in the conventional sense (the linear value chain), but also with companies from other industries operating under different value propositions and economics.

A value network is presented below, depicting the new shake-up of stakeholders as NFV and SDN technologies disrupt the classic model.

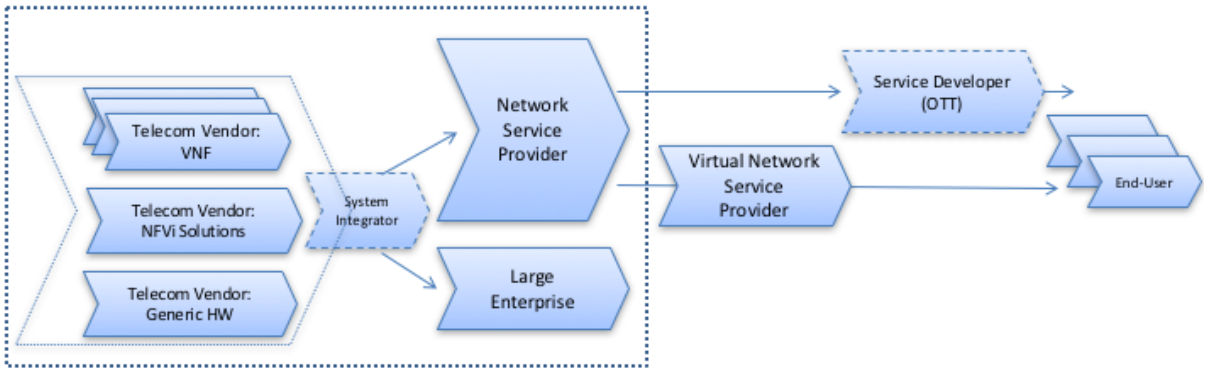


Figure 146. NSP Value Chain

- Telecom Vendors: What used to be the classic equipment vendor is now a more diverse portfolio or collection of competitors in the same ecosystem. A key disruption of the transition to software networks is the alleviation of vendor lock-in. In short, the network service providers that manage the network will be able to deploy software-based assets on top of generic hardware. This has the potential of a software stack with solutions integrated from various vendors. As that model implies, the following subset of vendors could also be a single large portfolio of a single vendor, or in fact broken down into even more subsets in the vendor-oriented area of the value chain.
  - Generic HW: As the name implies, this refers to the generic hardware equipment that would support the NFV Infrastructure (NFVI).
  - NFVi Solutions: This is simplified grouping, but a collection of diverse solutions that originate from one large portfolio, or an ecosystem or integration of several more focused software-based assets, such as a SDN Controller, NFVI, and NFV Management and Orchestration (MANO) platform. This is now an entity that provides the totality of all hardware and software components which build up the environment in which VNFs are deployed, managed and executed. The NFV MANO layer can even be more broken down into the Virtual Infrastructure Manager (VIM), VNF Manager (VNFM) and NFV Orchestrator (NFVO).
  - VNFs: The Virtual Network Function (VNF), a virtualized cousin of their physical counterparts, can come from various vendors. Currently developing NFV MANO solutions almost universally market their "multi-vendor" support, meaning that they can work with third-party VNFs to construct a network service (NS) chain.

These are the companies that supply virtual software application instantiations of a VNF, whether it be virtual Customer Premise Equipment (vCPE), virtual Evolved Packet Core (vEPC), virtual Mobility Management Element (vMME) or virtual Firewall (vFW). However the picture here is not so clear cut. There are also VNF Solution providers that gather together a suite of VNF vendor products together and provide this to a service provider as a complete catalog.

- System Integrator: Although this will often be done by the telecom vendor themselves, as part of their supporting services for their customers, the "softwarization" of networks means that a system integrator can become a larger actor in the value chain as they integrate solutions from various vendors.
- Network Service Provider: This refers to the telecommunications operator. They find themselves in a competition with "Over The Top" (OTT) service providers and look for a way for a return to form in competing with their own services, or monetizing an ecosystem involving them in a partner/competitor dynamic.
- Virtual Network Service Provider: This refers to companies that do not own the network infrastructure, but instead host the service operations and sales & marketing arms in offering content and applications to end customers. A RINA use case could be viable option for the (M)VNO, however only if it was a perceivable benefit in delivering differentiated value added services to customers.
- Converged fixed/mobile/content Players: It should also be noted that there are a number of converged fixed/mobile/content players (for example BT/EE and Sky) emerging in the market. This introduces a new (asymmetric) competitive dynamic compared to mobile players such as Telefonica. For example, converged players may be able to subsidise mobile elements of their offerings using non-mobile revenue streams, self-use network assets (core, backhaul and access) for both fixed and mobile services and strong media players (such as Sky) have significant “hero” content assets with strong appeal to customers. Content is increasingly important for customers and likely to increase in importance for customers when they choose a provider (whether a mobile offering or a multiple bundle).
- Large Enterprise: The vendor (or system integrator) can also deliver a NFV implementation to a large enterprise. It can be viewed as a

more realistic scenario for the RINA use case, where the scale is smaller and more apt to adopt additional disruptive technology. As network complexity, OSS legacy, etc. is a complexity in the SDN/NFV roadmap, RINA could be position as an additional facilitator for the enterprise

The value chain also includes actors further down the line, such as OTT service developers and end-users. OTT services and global players (such as Apple and Google) are an increasingly significant competitive factor, both in providing competing services and in exercising control over handset operating systems, home-screens and their integrated cloud platforms and app stores and the customer relationships they drive. New technology such as eSIMs and increasing Wi-Fi penetration, strengthens the opportunities of converged operators and OTT players. However, while the NSP use case for RINA would ultimately affect/benefit these actors, they would not experience the adoption or disruption themselves.

Examples of these stakeholders can be seen in the figure below.

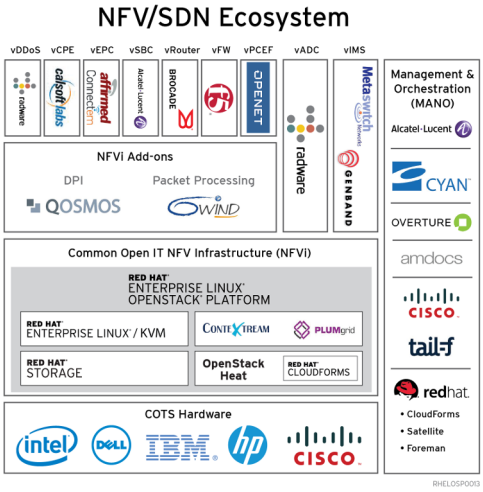


Figure 147. NFV ecosystem

Stakeholder Analysis and Customer Typology

For the purposes of this analysis we shall focus in the needs of the Network Service Provider (RINA consumer) and see how to link in the value network the System Integrator (RINA provider) offering a RINA solution. To get a sense of the value network from the perspective of the Network Service Provider, the diagram below is a submission by Telefonica UK to the UK regulator Ofcom.

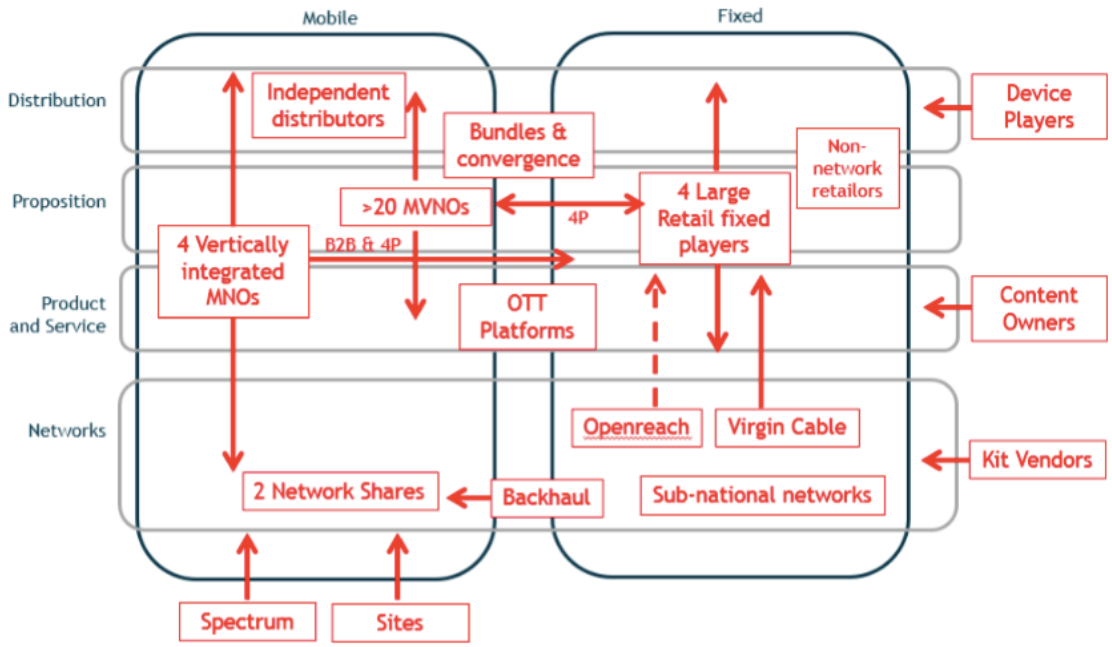
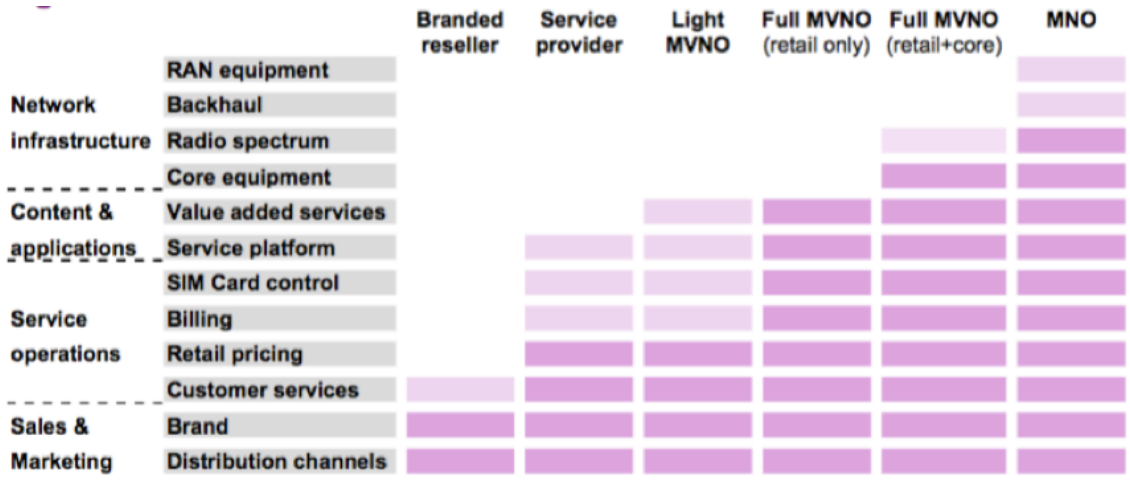


Figure 148. NFV stakeholders

In this diagram the Networks, Product/Service, Proposition, and Distribution value chain elements for both mobile and fixed line telecommunications are mapped out to the players that supply products and services in that network chain. The complexity of the value network is clear to see. The organisations we want to hone in on for this research are the vertically integrated mobile network operator (MNO), however we shall also consider the needs of a mobile virtual network operator (MVNO). It would be thought that the Retail Fixed Players would have the same types of needs as the MNO.



Source: Ofcom

Further to this, there are main shades of MNVO, but we are most interested in the Full MVNO, that's an MVNO that has both retail and core infrastructure.

Customer Jobs

In order to understand the needs of the MNO, we need to ask ourselves a set of trigger questions to see if we understand the jobs that the network operator is undertaking.

The one thing that the *MNO* must accomplishing everyday is to provide its customers with an available connectivity service, every hour of every day, or for want of a better term 5 nines availability (99.999%). While this may seem quite a simplistically stated goal, the stepping stones to achieve it are for from simple these days especially given the myriad of digital services that are accessed through the mobile network, and given the increased demand from bandwidth hungry devices and complexity of supporting the Internet of Things (IoT).

With the underlying infrastructure becoming more virtualised, it puts many different connectivity, performance and security contexts on the architecture that the *MNO* is managing towards 5 nines availability and the *MNO's* activities and goals change depending on these different contexts.

Since the connectivity between the user's handset and the services it is trying to connect to, be this a voice or data service, is no longer a single piece of copper wire with an electrically signal running through it, then the *MNO* needs to interact with each hardware and software vendor partner to ensure each element of the service chain from both a vertical and horizontal view point can provide 5 nines availability.

The *MNO* needs to manage and orchestrate the network infrastructure to provide for all the service contexts for which it will be used, and to continuously monitor this infrastructure to provide 5 nines availability.

The types of functional problems that the *MNO* is trying solve when carrying out this management and orchestration include:

- Network failures
- Internal application failures
- External service failures

- Network redundancy
- Active and passive infrastructure redundancy
- Storage architecture redundancy

However there are problems that the *MNO* is not even aware of. Existing QoS-aware multipath techniques are based on the knowledge of the characteristics of the traffic going through the network to efficiently select the best path for new flows. Current state of the art has covered this problem multiple times, for example IntServ through the use of RSVP (Resource Reservation Protocol) or automatically deploying MPLS tunnels. Although commonly used, these solutions have scalability problems because of the necessity of including additional layers on top of the traditional internet stack in order to support several levels of isolation. Moreover, such protocols face the problem of how to determinate the characteristics of each flow.

For end users, the *MNO's* customers, the characteristics of **their** flow is of super importance, whether it be a bandwidth hungry streaming service, or a low latency data gathering service.

In order for an *MNO* to deliver a network infrastructure that supports *5* nines availability with QoS-aware multipath techniques, then the job of setting up the service function chain, both for their own content & services, and those of the MVNO's that a MNO would support, has to be in the realm of simple and seamless.

Customer Pains

Service function chaining solidifies the relationship between NFV and Carrier Software Defined Networking (SDN), with a chain only comprising of devices which are not addressable through IP, that is, devices which are typically physically cabled into a path or are only locally addressable at Layer 2. This combined with an NFV orchestrator that manages the relationships between VNFs gives the *MNO* a service chain that consists of a set of network services, such as firewalls or application delivery controllers (ADCs) that are interconnected through the network to support service content and applications. SDN and NFV can make the service chain and application provisioning process a whole lot shorter and simpler.

Up till now building a service chain to support a new application or service took a great deal of time and effort. It meant acquiring network devices

and cabling them together in the required sequence. Each service required a specialised hardware device, and each device had to be individually configured with its own command syntax. The chance for error is high, and a problem in one component could disrupt the entire network. Moving network functions into software means that building a service chain no longer requires acquiring hardware.

Adding to the difficulty, application loads often increase over time, so building a chain that would not have to be immediately reconfigured meant estimating future demand and over-provisioning to support growth. Devices need to be sized to support the maximum level of demand, this extra capacity planning has meant extra capital investment for *MNOs*.

The effort required to construct a chain also meant that chains were often built to support multiple applications. As a result, data sometimes passed through unnecessary network devices or servers and consumed extra bandwidth and CPU cycles.

However the problems for network infrastructure deployment now lies with the virtual switching between VNF's in the data centre environment. Service chaining technologies provided by the OpenDaylight SFC implementation and intended to be integrated in the OPNFV platform currently include:

- OpenFlow programmed service chains for:
  - L2 VLAN encapsulation
  - MPLS encapsulation
- VxLan overlay based service chains for:
  - VxLan-GPE encapsulation with NSH headers
- Basic load balancing at SFC (planned for ODL Lithium)
- Programmatic service function selection algorithms
  - Round robin
  - Load balanced (choose the least loaded service function)
  - Random allocation

With future development looking to include hybrid service chains, support for layer 4-7 classification, additional encapsulations, additional load balancing and service function instance selection algorithms.

As has been found through the PRISTINE research, these solutions will not scale to support 5 nines availability with QoS-aware multipath techniques.

Notwithstanding, given that the service function chain is delivered via the software defined data centre, the customer pains as experienced in the the Datacenter (DC) networking use case will also be felt here in the network service provider use case.

Customer Gains

How to determine whether a new technology is worth the costs of adoption is a tricky question. One guideline is the 10X rule: if you can expect a return of 10 times the investment, then it's worth it.

However to understand how to put this rule into practice, it has to be remembered that the gains can come from any of several improvements, or a combination of improvements:

- Cost reduction,
- Service enhancement,
- Competitive edge in the environment,
- Efficiency or productivity improvement,
- Faster time to market.

The expected gains must be the sum total of all factors. If adopting a new technology like RINA provides an improvement in one factor but at the expense of another factor, it may not be worth adopting the technology. This is a type of return on investment (ROI) analysis carried out by network service providers, which frequently focuses only on financial factors.

Also of note the timeframe for the gains to be realised must also be included in the analysis. Some investments in new technology may require several years to begin to provide full value, although we do not think this is the case with RINA.

Value Proposition

The direct adopter of RINA technology in this value network would be the system integrator.

As the clear linkage point between the NFVi Solutions/VNFs and the Network Service Provider, the system integrator can provide the isolated network connectivity slices that RINA enables, which can guarantee the *MNO* performance quality assurance, with security and mobile per slice over existing (LTE) and future (5G) infrastructure towards their software defined data centres.

The RINA based solution will allow a mobile network service provider to deliver:

- Layer 2 Agnostic Networking.
- Global layer 2 vLANs
- Application Specific Name Spaces for Billions of IDs, truly supporting IoT.
- Inherent Application Security for Distributed Services.
- Isolated 'Network Slices' for End to End Services.
- Pro-Active Congestion Control.

Unlike current products which are all based on the TCP/IP paradigm, a RINA based product supersedes this old technology with simple policy-based programmable functions which: support congestion avoidance, provide protection/resilience, facilitate more efficient topological routing, and multi-layer management for handling configuration, performance and security.

Market Feasibility

The feasibility of a a RINA solution in the market of NFV for a network service provider is analysed through an Strength, Weakness, Opportunity and Threat (SWOT) analysis:

Strengths

- Network Dependability: As the Internet is becoming an increasingly critical infrastructure for governments, citizens and businesses in their day-to-day activities, the robustness of the network has to be a major consideration for any potential replacements. Networks can be

impaired for several reasons (e.g. physical or cyber attacks, etc.). RINA offers improvements in provisioning network connections that support required QoS, and security facilities. In particular, the enrolment process is very useful in identifying and authenticating network endpoints. When combined with the available resilience policies, this allows RINA to provide better guarantees for the dependability of the network, by controlling admission and optional authentication of communicating parties.

- Demand for improved security and inbuilt QoS: It is to be expected that once the consumer, regulatory bodies and industrial fora, become aware of RINA's technical benefits of improved security and inbuilt QoS, they may demand these features. There is a large demand for improvements in network technologies with regard to the security of the systems using it, recent DDoS attacks on popular social network sites attest to this. RINA offers an explicit enrolment stage on initial attachment, so allows for explicit access controls on who can connect to or gain access to the network. In addition it adds configurable packet protection on flows of information making it highly flexible for offering appropriately secured solutions for today's services. The inbuilt QoS support means that the delivery of these services is more consistent and reliable and it is expected consumers will like this.

Weakness

- Deployment costs; It is anticipated that there are some initial adoption costs that will be incurred in deployment of a clean-slate network architecture such as RINA. These adoption costs are varied and may impact on some or all the stakeholders involved. The deployment of RINA would require some form of software upgrade to the virtual routers and switches in the NFV environment. The most significant portion of the labour costs is the cost of training in the use of the RINA architecture. This can be regarded as a once off cost or as an investment in a future technology.
- Levels of education; The level of awareness of a nascent network technology within the network service providers, service/application providers and NFVi Solution providers will be a contributing factor. For a complete success, all stakeholders in the value network need to be aware of RINA and its relative advantages over the existing TCP/IP stack

and it may take some time to educate the stakeholders of the TCP/IP alternatives and relative benefits of RINA, as the leading contender.

Opportunity

- Network usage: In general terms the proportion of network data has been increasing year on year. However, Ericsson has been monitoring the proportion of traffic on its networks due to mobile data as opposed to voice. It is interesting to note that the proportion of voice data has only been slightly increasing in the last four years. However, mobile data, i.e. data used from smartphones has seen an exponential growth. This is reflected in other statistics and gives the overview that network traffic is increasing, it is becoming more mobile focussed, and the characteristics of the data are changing. RINA is in a good position with its inbuilt QoS and mobility support to take advantage of these trends.
- Mobility: Mobility, gives consumers the ability to roam, and still access the services they demand. There has been a huge growth in the demand for services that can be consumed on mobile devices. The complex mix of technologies that attempt to deliver on 4G are already adding delays in the perceived response time by the mobile consumer. In fact by 5G deployment time, the wireless interface will no longer be the bottle-neck when it comes to deliverying the service, it will in fact be the core network.

Within RINA, changing the point of attachment of a mobile device as it roams, is as simple as adjusting the routing table between two "layers" of the network stack. The consistency and simplicity of the approach will lead to faster switching between points of attachment and a better overall experience for the end user.

The market is also looking for:

- new ways to speed up service development and deployment,
- new ways to lower Capital and Operational expenditures,
- new business models,
- new ways to improve network reliability and mobility.

RINA is a technological architecture that can support these requirements.

Threat

- Resistance from incumbent technologies: Resistance to change is a serious, and often frustrating hurdle for technologists to overcome. Many of us are more adept at dealing with hard problems such as server infrastructure or network capacity. Shaping user behaviour is a soft problem that has more to do with psychological and social barriers to technology adoption. For a network service provider to risk **failure** is often synonymous with "looking stupid in front of someone". The safe option for most users is to avoid trying something as risky as new technology. As technologists, we often fail to see how intimidating technology can be to the user community. The introduction of new technologies, especially those that affect communication, is a stressful process for network service providers. These concerns must shape our strategy for gaining acceptance of any new technology. Firstly, with RINA, it must be evident to the user as potentially useful in making their life easier. Secondly, RINA must be easy to use to avoid rousing feelings of inadequacy. Thirdly, RINA must become essential to the user in going about their business. This **Three-E Strategy** if applied properly, has been at the core of every successful technology adoption throughout history, and should be applied again for RINA.
- Deployment risks: How to reduce the risk in deploying a new technology like RINA is a key question for those who may look to adopt it. What may help is that a plan for adoption can significantly reduce the risk were it is not necessary to have a full-blown deployment, but it's just as valid to have a plan to investigate new technology and how it can be applied to the business. A couple of suggested steps that can be taken to minimize the risk of adopting RINA can be summarised as:
  - Identify a business service that the technology addresses. Maybe there is a customer that needs to make their network more agile, with their current business being impacted by competitors that have more a more agile provisioning model.
  - Engage with a multi-functional team within the adopting *MNO* to investigate the technology. By engaging with staff members from different backgrounds, we gain the advantage of different perspectives, however it will be essential that the participating individuals are forward looking and open to change. Ideally, the individuals involved will have an understanding of the business,

the competitive environment, and be willing to make unpopular recommendations when needed.

- Engage with external advisors to the adopting *MNO*. This will enable a different viewpoint to that of the internal staff. A good advisor should have some experience with the new technology and be aware of the common problems to be encountered in its implementation.
- Identify the risks and quantify them. Include both the risk of adopting RINA and the risk of not adopting it.
- Document the costs and benefits of the new technology. Attempt to create a way to monitor the cost-benefit tradeoffs so that the *MNO* can tell if the new technology is resulting in a gain. With good controls, it can be easily determined if it is successful or not.
- Implement a proof-of-concept implementation. Starting with a small implementation early in the investigation process allows the organization to identify problems early when they are easier and less expensive to correct. It also makes it easy to start over.
- Processes and culture: Network service providers are very process orientated, and so we cannot overlook the soft costs of implementing a new technology like RINA. Both organizational and process changes may be required, and these should be captured early in the proof-of-concept phase.Procedural changes are very common with new technology. The process for provisioning for a telephone in 1990 has changed significantly over the years, and now looking at how triple play services are introduced and deployed in 2016 there is a world of difference. We should be prepared to develop changes to previous *MNO* processes and expect to encounter some resistance to this change.

References

[apcc2016] Ichrak Amdouni, Farma Hrizi, Anis Laouiti, Eduard Grasa, Hakima Chaouchi. “Exploring the flexibility of network access control in the Recursive InterNetwork Architecture“. APCC 2016, Yogyakarta, August 2016.

[flo93] Sally Floyd and Van Jacobson, "Random Early Detection Gateways for Congestion Avoidance", 1993, IEEE/ACM Transactions on Networking.

[aki10] Data Center TCP (DCTCP); Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan, SIGCOMM 2010.

[d4.3] wp4/d43/d43 Load Balancing, https://wiki.ict-pristine.eu/wp4/d43/d43-load-balancing