Defence Research and Development Canada
Recherche et développement pour la défense Canada

DRDC | RDDC
technologysciencetechnologie

Emulation- and hardware-based testbed setup and configuration for software-defined networking

Ming Li
DRDC – Ottawa Research Centre

Defence Research and Development Canada
Reference Document
DRDC-RDDC-2019-D149
November 2019

Canada

IMPORTANT INFORMATIVE STATEMENTS

This document was reviewed for Controlled Goods by Defence Research and Development Canada (DRDC) using the Schedule to the *Defence Production Act*.

Endorsement statement: This publication has been published by the Editorial Office of Defence Research and Development Canada, an agency of the Department of National Defence of Canada. Inquiries can be sent to: Publications.DRDC-RDDC@drdc-rddc.gc.ca.

Abstract

This Reference Document describes the setup of a Software-Defined Networking (SDN) testbed in a lab environment, aiming at providing a platform for exploring and practicing SDN operations. It serves a two-fold purpose: first as part of the research activity carried out at DRDC – Ottawa Research Centre for The Technical Cooperation Program (TTCP) C4I TP43 CP3 project, and secondly for the future investigation of the potential application of SDN techniques to support the CAF.

The testbed is designed and configured such that it includes the main components in the SDN architecture, representing key functionalities across the data plane, control plane and application plane. We construct the testbed with two configurations, one is based on an emulation network and the other uses an SDN switch hardware. After detailed description of the testbed setup, we present a simple test scenario to verify the testbed is performing SDN functions properly, and demonstrate some basic networking operations.

Résumé

Le document de référence décrit la mise en place d'un banc d'essai de réseaux définis par logiciel (*Software-Defined Networking* [SDN]) en laboratoire, le but étant de fournir une plateforme d'exploration et d'essai destinée aux opérations SDN. L'objectif est double : d'une part, appuyer les activités de recherche menées par RDDC Ottawa dans le cadre du projet CP3 du groupe technique 43 du C3IR du Programme de coopération technique; d'autre part, évaluer l'applicabilité des techniques SDN aux opérations des FAC.

Le banc d'essai est conçu et configuré de manière à reproduire les principaux composants de l'architecture SDN, c'est-à-dire les fonctionnalités essentielles des plans de données, de contrôle et d'application. Il se décline en deux configurations, l'une qui repose sur un réseau d'émulation, et l'autre, sur un dispositif de commutation SDN. La description détaillée de la mise en place du banc d'essai est suivie d'une simulation simple permettant de vérifier si celui-ci exécute correctement les fonctions SDN, ainsi que d'exemples d'opérations réseau de base.

Table of contents

List of figures

List of tables

1 Introduction

Software-Defined Networking (SDN) is an evolving networking paradigm that separates the control plane and data plane in the network architecture, and promotes (logically) centralized network control and management. SDN has been gaining momentum in recent years, from campus, data centre and enterprise to network service providers, see [1] for a comprehensive survey. The SDN techniques have also found potential applications in military operations, while The Technical Cooperation Program (TTCP) project C4I TP43 CP3 has been set up [2].

From a functional layer perspective, the SDN is primarily "the physical separation of the network control plane from the forwarding plane, and where a control plane controls several devices," as quoted from the Open Networking Foundation (ONF) website [3], where the SDN architecture is characterized to have the following properties: directly programmable, agile, centrally managed, programmatically configured, open standard-based and vendor-neutral. One of the cornerstone components in SDN is the southbound interface between control plane and data plane, with the OpenFlow protocol as the de facto standard, see [3] for the technical specifications.

The purpose of this Reference Document is to construct a lab testbed by integrating open source software and development toolkits that can be used to explore SDN functionalities and investigate the application of SDN techniques in military operations.

1.1 Testbed design considerations and architecture

The testbed is to be set up and configured with the following architectural and design considerations:

- Easy to setup with minimal requirements for expensive SDN devices;
- Using common and mature software for maintenance and sustainability;
- Compact, simple while representing key SDN features;
- Extensible for future function enhancement and feature implementation.

We use two approaches in the testbed setup: 1) based on the most popular SDN learning and experimental network emulation toolset known as mininet [4]; 2) based on an OpenFlow switch hardware Zodiac FX (ZFX) [5] from Northbound Networks.[1] We refer to them as emulation platform and hardware switch platform, respectively.

On both platforms we implement characteristic SDN functions at three layers: data plane, control plane and application plane. The SDN controller we chose to use is the Ryu SDN controller [6]. The SDN switch used in the testbed is the Open vSwitch [7] on the emulation platform and Zodiac FX (ZFX) on the hardware switch platform. The above implementation ensures that the southbound interface is fully OpenFlow compliant. The northbound interface is REpresentational State Transfer (REST) compliant with a simple Postman [8] Graphical User Interface (GUI) to validate the application plane functions.

[1] The vendor has stopped manufacturing SDN hardware. However, the testbed setup described in this document can be easily extended to working with other OpenFlow SDN switches.

The architecture and hardware/software requirements for setting up the SDN testbed are further summarized in the next two sub-sections.

1.1.1 Testbed on emulation platform based on mininet

This setup is implemented on a single Linux laptop; see Figure 1 for the architectural view. The detailed testbed build procedure is given in Section 2. The emulated network by mininet works seamlessly with the Open vSwitch (a.k.a. OVS, a virtual switch that has been integrated into most of the newer Linux releases) to provide an SDN enabled data network, which is controlled by the SDN controller Ryu. We have chosen Ryu among many available SDN controllers as it is one of the compact controllers that is widely used in the SDN community, being actively maintained and developed, and feature rich with a complete set of Python Application Programming Interfaces (APIs) that provide a good framework for our future adaptation and development. The network packet capture and protocol analyzing tool Wireshark [9] is also installed for traffic analysis and packet dissection, specifically to examine the OpenFlow protocol traffic between control plane and data plane.

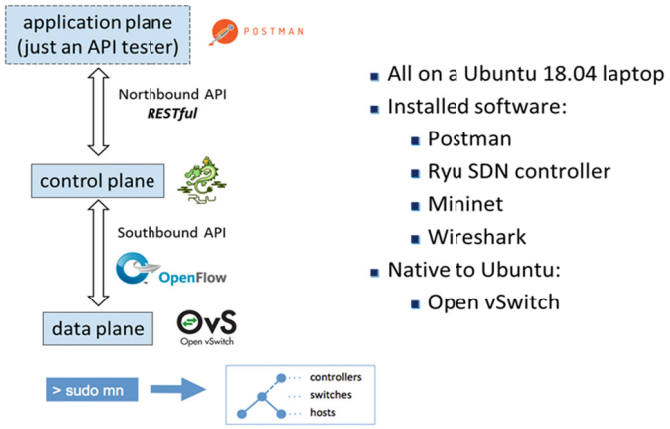

Figure 1: *Testbed architecture on emulation platform.*

1.1.2 Testbed on hardware switch platform based on ZFX

In this setup we attempt to set up a testbed that is as close as possible to the real world software-defined networks, including the following features: an independent data plane with the network data forwarding units; a hardware SDN switch; and a remote SDN controller (operating from a separate box). The entire testbed is still a light-weight system that can be hosted on two Linux laptops, as shown in Figure 2. One laptop is used to set the data plane which consists of three virtual Linux boxes, each of them is equipped with a dedicated USB Ethernet adapter for connection to the hardware switch ports on ZFX. This data plane is equivalent to a system of three separate Linux nodes each with its own wired connection to the hardware switch. The other laptop hosts a set of control plane applications and a specific virtual Linux box for the Ryu controller, which has a dedicated USB Ethernet adapter for connection to the management port on ZFX.[2]

[2] The Zodiac FX switch has three data ports (port 1, 2 and 3) connecting to data forwarding units and one management port (port 4) connecting to an SDN OpenFlow controller.

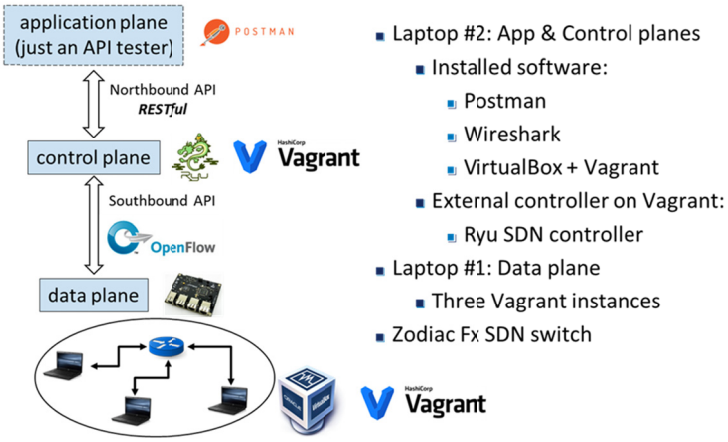

Figure 2: *Testbed architecture on hardware switch platform.*

The aforementioned four virtual Linux boxes (three for data units on laptop 1 and one for SDN controller on laptop 2 in Figure 2) are implemented on Oracle VirtualBox [10]. To maintain the configuration for the virtual machines we use Vagrant [11], a development and configuration tool to set up the virtual environment and configure the USB Ethernet adapters.

1.2 List of required hardware and software

The list of required hardware for SDN testbed setup:

- Two Linux laptops[3] (i5 or i7 processor, minimum 8G RAM, and enough hard disk space), with Ubuntu 18.04 installed and up-to-date;
- One Zodiac FX OpenFlow SDN switch [5];
- Four USB Ethernet adapters;
- One USB hub (if the data plane laptop cannot have three USB adapters plugged in);
- One Ethernet hub/switch (optional, to verify data plane connectivity).

The list of required software (with the software version at the time of testbed setup):

- Oracle VirtualBox [10], version 6.0.4 r128413 (Qt5.9.5);
- Vagrant [11], version 2.2.4;
- Mininet [4], version 2.3.0d5;
- Ryu SDN controller [6], version 4.30;
- Postman [8], version 1.8.8;
- Wireshark [9], version 2.6.6;

[3] For the emulation testbed only one Linux laptop is required.

- Open vSwitch [7], version 2.9.2;
- Minicom (for serial port communication with ZFX), version 2.7.1; and
- Other software packages and tools include Perl, Python 2.7, ifconfig, pip and git, which should be already packaged in the standard release.[4] Otherwise it is straightforward to install them following the standard Ubuntu package installation procedure, i.e., `sudo apt update; sudo apt install package_name`.

The rest of this document is organized as follows. In the next section we describe detailed procedure for testbed setup. In Section 3 we present a simple SDN scenario to verify that the testbed is fully functional. Some concluding remarks and notes for future work are given in the last section.

[4] The `ifconfig` command has been deprecated in Ubuntu 18.04 and can be installed by `sudo apt install net-tools`.

2 SDN testbed setup and configuration procedures

In this section we present step-by-step procedures for setting up the testbed. We assume the Ubuntu laptops are clean-installed Ubuntu 18.04 to have an unaltered baseline OS, and a direct internet connection is required for software installation and upgrade.[5] The order of installing the software components is not critical, although the steps presented here are recommended.

Many of the software packages can be installed using the simple Ubuntu software installer:

```
sudo apt update
sudo apt install package_name
```

However, the desired packages from official Ubuntu repository are often not updated in time. We may instead use the up-to-date compatible version directly from the source web site, and install the software either from the available .deb package or from the GitHub repository. In the latter case we would specify the git tag when applicable.

2.1 Emulation platform based on mininet

In this setup the testbed is implemented on a single Ubuntu 18.04 laptop, no other hardware is required.

2.1.1 Install software dependencies and tools

The current Ryu controller is based on Python 2.7, while the Ubuntu 18.04 defaults to Python 3. We install the required packages as follows (from now on we will omit the "sudo apt update" line prior to package installations). Note that some packages may have already been on the system, we list the steps below just for completeness.

```
sudo apt install git
sudo apt install python
sudo apt install python-dev python-setuptools python-pip
```

The Wireshark tool should be installed from the Wireshark repository for the latest stable version:

```
sudo add-apt-repository ppa:wireshark-dev/stable
sudo apt update
sudo apt install wireshark
(allow non-root users to run wireshark when prompted)
```

2.1.2 Install Ryu SDN controller

The installation should be done from the home directory (/home/wlab in our case).

```
git clone git://github.com/osrg/ryu.git
cd ryu
pip install .
```

To verify the successful installation and check the controller version:

```
ryu --version
(returns ryu 4.30)
```

[5] Once the laptops are completed with software installation, the testbed can be operated off-line.

```
ryu-manager --version
(returns ryu-manager 4.30)
```

Note: The Ryu official site [6] suggested building Ryu from source code using:

```
git clone git://github.com/osrg/ryu.git
cd ryu; python ./setup.py install
```

However, the above procedure had some package dependency issues in our testbed building. Using pip is the recommended way which is also specified on the Ryu GitHub site, and it brings an additional benefit that the executables (ryu and ryu-manager) are owned by the user wlab (hence no need to proceed the commands with "sudo").

2.1.3 Install mininet and test Ryu controller

The mininet emulation should also be installed from GitHub source under the home directory:

```
cd
git clone git://github.com/mininet/mininet
git tag
git checkout 2.3.0d5
cd mininet
util/install.sh -fnv

sudo mn --test pingall
sudo mn -c
```

Note 1*:* Using the -fnv option will also install the compatible OpenFlow implementation and the OVS packages, which would be sufficient for our testbed setup. Another install option is "-a" which will install all software for the mininet tutorials, including POX [12] (a Python based SDN software platform, however no active development now), the OpenFlow Wireshark dissector, the "oftest" framework, etc. See INSATLL section on the mininet GitHub site for detail.

Note 2*:* The penultimate line above is to verify the successful installation of mininet.[6] The last line with option "-c" is to clear up any environment changes and runaway processes during the mininet session, hence is strongly recommended after exit of each mininet run.

We now test the mininet with basic OpenFlow switch and Ryu controller functionalities. The Wireshark tool can be launched prior to running mininet to observe the dissected OpenFlow packets, with the command "sudo wireshark" and choosing interface "lo" and filter "openflow" in Wireshark GUI.

```
sudo mn --topo single,3 --mac --switch ovsk
mininet> dump
mininet> pingall
mininet> dpctl show
mininet> exit
sudo mn –c
```

In the above we start an emulated network of three nodes with designated MAC address, and the OVS switch. The sample commands under prompt "mininet>" illustrate the rich features of the mininet emulation. More details can be found in mininet web site [4]. Next, we re-run the above session but with an SDN controller included in the picture:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote
```

[6] The command line tool ifconfig is needed to run pingall test.

```
mininet> pingall
mininet> exit
sudo mn –c
```

In the above testing "pingall" would fail, as the network is now controlled by a remote SDN controller but we don't have one running yet. So we restart the mininet session as above, and run the following command in a separate terminal:

```
ryu-manager ryu.app.simple_switch
```

This command will start the Ryu controller and load a Python application to configure the OVS as a layer 2 learning switch (see /home/wlab/ryu/ryu/app/simple_switch.py for the Python code). The mininet "pingall" command should succeed now.

2.1.4 Install Postman and test northbound API

The Postman [8] tarball can be downloaded from https://www.getpostman.com/downloads/, then extracted to /home/wlab/Postman. For Ubuntu 18.04 we may need to install libgconf-2-4 package if it does not exist: sudo apt install libgconf-2-4.

It would be convenient to have a Ubuntu desktop launcher for Postman. For this purpose a new file named *Postman.desktop* should be created under /home/wlab/.local/share/applications with the following content:

```
[Desktop Entry]
Encoding=UTF-8
Name=Postman
Exec=/home/wlab/Postman/app/Postman %U
Icon=/home/wlab/Postman/app/resources/app/assets/icon.png
Terminal=false
Type=Application
Categories=Development;
```

The file should be set executable (chmod +x) then copied to the Ubuntu desktop. The first time Postman is launched it may need to be set as "trusted" in the launcher.

To test the Ryu controller northbound APIs, we start the mininet and Ryu controller on two different terminals similar to the last section:

```
sudo mn --topo single,3 --mac --switch ovsk --controller remote (terminal 1)

ryu-manager ryu.app.simple_switch ryu.app.ofctl_rest (terminal 2)
```

Note that when starting the Ryu manager we have loaded ofctl_rest module in addition, this is for the RESTful northbound APIs of the Ryu controller that will interact with Postman. The output from the Ryu controller terminal is as follows:

```
loading app ryu.app.simple_switch
loading app ryu.app.ofctl_test
loading app ryu.controller.ofp_handler
creating context dpset
creating context wsgi
instantiating app app ryu.app.simple_switch of SimpleSwitch
instantiating app app ryu.app.ofctl_rest of RestStatsApi
instantiating app app ryu.controller.ofp_handler of OFPHandler
(11678) wsgi starting up on http://0.0.0.0:8080
```

and on the mininet terminal we can find the OpenFlow datapath ID (dpid) by using "dpctl show" command:

```
mininet> dpctl show
*** s1 ----------------------------
OFPT_FEATURES_REPLY (xid=0x2): dpid:0000000000000001
N_tables:254, n_buffers:0
... ...
```

With the above Ryu controller web service port and mininet dpid information, we can start the Postman and enter the HTTP GET command to URL http://localhost:8080/stats/desc/1 to retrieve the controller description. More Postman testing examples are presented in Section 3.

2.2 Hardware switch platform based on ZFX

In this setup we implement several virtual machines built on VirtualBox [10] with Vagrant [11] for virtual environment configuration and maintenance, such a combination is referred to as a "vagrant box." We also choose to use USB Ethernet adapters which are bridged on the virtual network interfaces inside the vagrant boxes in the testbed. Since the vagrant boxes communicate with network peers through Ethernet cables instead of virtual communication links, we could simply replace each vagrant box by a Linux laptop to build the corresponding SDN network, in other words the migration of the testbed to a real-world physical network will be straightforward. This is one of key advantageous features of our testbed.

We will first describe the setup of three vagrant boxes on one Ubuntu laptop (the host machine), which constructs the data plane. The second laptop hosts a vagrant box as the SDN controller and the Postman for northbound API testing, as well as other auxiliary software.

2.2.1 Vagrant box configuration and data plane setup on laptop 1

We first install the up-to-date version of VirtualBox from its web site. Download the .deb file from https://www.virtualbox.org/wiki/Linux_Downloads (in our case we get version 6.0.4), then

```
sudo dpkg -i virtualbox-6.0_6.0.4-128413_Ubuntu_bionic_amd64.deb
sudo apt -f install
```

The second line above is as needed if any dependency issues are encountered. Next we download Vagrant .deb file from https://www.vagrantup.com/downloads.html and run

```
sudo dpkg -i vagrant_2.2.4_x86_64.deb
```

to install Vagrant on the host Ubuntu laptop. Each vagrant box should be configured in its own working directory. We search on https://app.vagrantup.com/boxes/search?provider=virtualbox to find a pre-built vagrant box, in our case we choose the official Ubuntu 16.04 LTS daily build box which is named ubuntu/xenial64. We create vagrant box 1 in a clean directory:

```
mkdir vgrt_node1
cd vgrt_node1
vagrant init ubuntu/xenial64
```

A new file *Vagrantfile* will be generated in the directory (by the vagrant init command), which maintains the configuration for the vagrant box. We need to make one-line change in the file:

```
config.vm.network "public_network", ip: "10.10.10.1"
```

This is to configure a virtual network interface in the vagrant box with IP 10.10.10.1, and bridge it to a physical network interface on the host. We repeat the above steps to create vagrant box 2 and 3 under directories vgrt_node2 & vgrt_node3, with private IP addresses 10.10.10.2 and 10.10.10.3, respectively. The next step is to connect three USB Ethernet adapters to the host (use a USB hub if there are no enough USB ports on the host laptop), then launch vagrant boxes from the corresponding working directory, for example:

```
cd vgrt_node1
vagrant up
```

Note that it will take some time to boot up the vagrant box for the first time as the ubuntu/xenial64 box needs to be downloaded from the Vagrant cloud. At the boot up we will see the following output (example):

```
... ...
===> default: Available bridged network interfaces:
1)  wlp3s0
2)  enxa0cec8d03945
3)  enxa0cec8d03b1e
4)  enpn0s25
5)  enxa0cec8d03933
===> default: When choosing an interface, it is usually the one that is
===> default: being used to connect to the internet
      default: Which interface should the network bridged to?
```

In our example we will enter USB adapter interfaces "2," "3" and "5" for vagrant boxes 1, 2 and 3, respectively. After the vagrant boxes are completely booted up, we can access the vagrant box using command:

```
vagrant ssh
... ...

vagrant@ubuntu-xenial:~$
```

and we can verify that the vagrant box has the defined IP address 10.10.10.x using command ifconfig or ip addr. To further validate the network connectivity we can attach USB cables to the three USB adapters and connect them to a network switch/hub. The vagrant boxes should be able to ping each other, the same behaviour as we connect three Linux boxes to the switch/hub.

The vagrant boxes as configured can be used in software development exactly as on a Linux box. To maintain a stable environment we suggest disabling automatic update of the vagrant box by changing one line in *Vagrantfile*:

```
config.vm.box_check_update = false
```

To exit from the vagrant box just issue "exit" command and get back to the host working directory. Other common vagrant commands (issued from the host machine vagrant directory) include:

```
vagrant -h  (print help message)
vagrant reload  (restart vagrant box upon any changes in the Vagrantfile)
vagrant halt  (stop the vagrant box, i.e., power off the corresponding virtual machine)
vagrant destroy  (to completely remove the virtual machine if no longer needed, cleaning up the disk space)
```

For further information please consult the documentation on the Vagrant web site [11].

2.2.2 The Ryu SDN controller on a vagrant box on laptop 2

The laptop 2 is the same one that is used in Section 2.1. Instead of using the existing Ryu controller installed in Section 2.1.2, we set up a vagrant box and install the Ryu controller on the box, which has a dedicated Ethernet connection to the ZFX. This way we have an independent platform for the Ryu controller application development that can be readily migrate to a separate Linux machine in future.

The Ryu controller vagrant box is set up via the procedure similar to the previous section, under directory vgrt_zfx, however with the following changes in its *Vagrantfile*:

```
config.vm.boc_check_update = false
… …
config.vm.network “forwarded_port”, guest: 8080, host: 8888, host_ip: “127.0.0.1”
config.vm.network “public_network”, ip: “10.0.1.8”
… …
config.vm.provider “virtualbox” do |vb|
   vb.memory = “2048”
end
```

in the above configuration, forwarded_port is for port forwarding between the vagrant box (guest) and the host, so that we can use URL http://localhost:8888 in Postman on the host machine to interact with the Ryu controller northbound API operated in the vagrant box, which would have been http://localhost:8080 inside the box if Postman is installed and run in the box as described in Section 2.1.4. In other words, we are simulating the scenario where the control plane (Ryu controller in the vagrant box) and application plane (Postman on the host machine) are hosted in different machines in the network. The IP assignment 10.0.1.8 for the public network is Zodiac FX specific, see next section for detail. We also increased RAM for the vagrant box to 2048M (default RAM is 1024M in vagrant generated *Vagrantfile*) for performance of the vagrant box. The complete form of the *Vagrantfile* is attached in Annex A.1, and the screen capture of the boot up process is depicted in Annex A.2.

The vagrant box has a default user vagrant. It’s a good practice to create a different user and install Ryu controller under that user’s home directory. To this end we ssh into the box and do the following:

```
sudo apt install python (install python 2.7 from the Ubuntu repository)

sudo adduser wlab  (or any other name, and enter selected password)
sudo usermod –aG sudo wlab (add user to sudo group)

su - wlab
(enter password for wlab)
(the following is done as user wlab)
sudo apt install python-dev python-setuptools python-pip
git clone git://github.com/osrg/ryu.git
cd ryu
pip install .

ryu-manager --version (validate ryu is installed successfully)
```

Note that in this setup, we should always switch to user wlab (“su - wlab”) after ssh into the vagrant box then run the Ryu controller.

2.2.3 ZFX management and other control plane software on laptop 2

The ZFX switch is powered by a USB cable connecting to the host laptop, which at the same time provides the serial port communication. We can access the ZFX by either the serial port or the

management Ethernet port (port 4 on the switch). For serial communication, we need to install a serial emulation program such as minicom (sudo apt install minicom). After we connect the ZFX switch to the laptop, we can access the switch via:

```
ls /dev/tty*  (should see /dev/ttyACM0 which refers to the ZFX)
sudo minicom --device /dev/ttyACM0
… …
Zodiac_FX# config
Zodiac_FX# show config
```

The default configuration of a ZFX is depicted in Figure 3. We see that the device is configured on 10.0.1.0/24 network[7] and the switch has the IP address 10.0.1.99. To access the device via Ethernet connection we need to assign an IP address of this network to the native Ethernet adapter of the laptop (enp0s2 in our case), for example we can set it to 10.0.1.5:

```
sudo ip addr add 10.0.1.5/24 dev enp0s2
```

after the IP is configured we can connect enp0s2 to ZFX port 4, then open a web browser and go to URL http://10.0.1.99 which presents the web GUI for the ZFX. The output display is shown in Figure 3 as well.

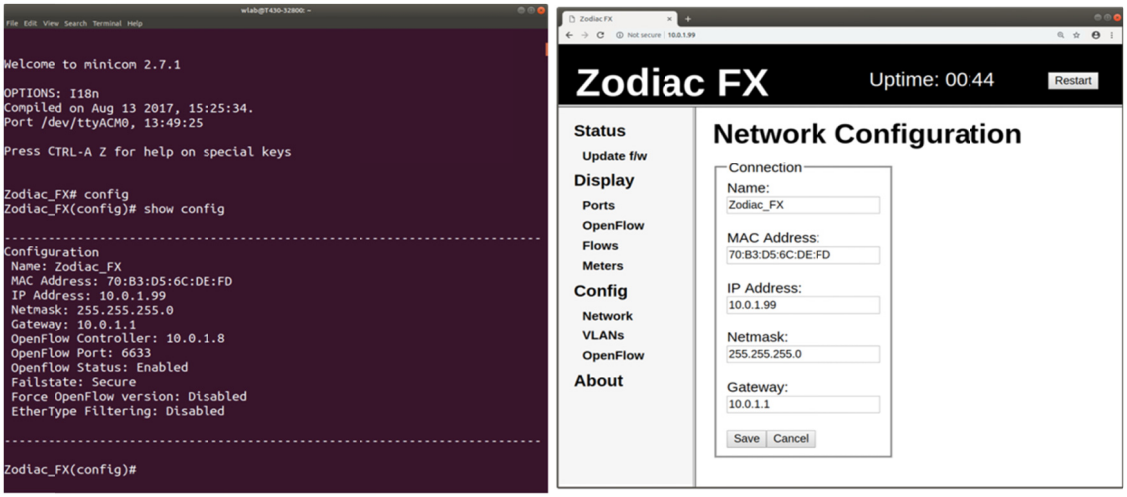

Figure 3: *Zodiac FX default configuration.*

We now re-connect the ZFX management port to the Ryu controller in vagrant box (as set up in the previous section) instead of directly to the host laptop. Attach a USB Ethernet adapter to the host laptop, connect it to ZFX port 4, then run vagrant reload under vgrt_zfx and choose the USB adapter (e.g., enxa0cec8d03994) for bridging. As described in the last section we have already configured the IP address in the *Vagrantfile* for this bridged interface to 10.0.1.8, which matches the OpenFlow controller address of the ZFX shown in Figure 3. If we ssh into the vagrant box, we should see a new virtual interface created in the box (in our case it's enp0s8) with IP address 10.0.1.8, and we can validate the connectivity between the Ryu controller and ZFX by pinging 10.0.1.99. Moreover, we can set up direct access from the host to ZFX via the USB adapter by a different IP in the private network:

```
sudo ip addr add 10.0.1.10/24 dev enxa0cec8d03994
```

[7] The private network for the ZFX is configurable; we will use the factory default settings in our testbed.

after this setup we should be able to ping 10.0.1.99 (ZFX) and 10.0.1.8 (Ryu controller) from the host laptop, and ping 10.0.1.99 (the ZFX) and 10.0.1.10 (the host machine) from within the vagrant box. If we start a browser on the host machine and go to URL http://10.0.1.99 we should see the same ZFX management GUI as in Figure 3 (note that previously we were connecting ZFX port 4 directly to the host network interface enp0s2).

We can now validate the ZFX hardware switch testbed. First we launch the Ryu controller in the vagrant box:

```
su - wlab
(enter password for wlab)
ryu-manager --verbose --ofp-listen-host=10.0.1.8 --ofp-tcp-listen-port=6633 ryu.app.simple_switch_13 ryu.app.ofctl_rest
```

in the above command line we specify the Ryu controller IP 10.0.1.8 and port 6633 that match the switch hardware configuration in Figure 3, and we invoke two Ryu modules: simple_switch_13 where "_13" stands for OpenFlow version 1.3 of the ZFX firmware; and ofctl_rest for interacting with RESTful northbound APIs.

***Note 1**:* Same as in Section 2.1 we can always start Wireshark before running the test to get detailed OpenFlow packets.

***Note 2**:* The above command will initiate an OpenFlow session between control plane (the Ryu vagrant box) and data plane (the ZFX switch and the attached data forwarding units). We can observe exchange of the OpenFlow protocol messages from the vagrant box command terminal (with the --verbose option above), and the flow statistics from the ZFX web GUI by clicking on "Display/Flows" in the left panel.

The last step is to verify the northbound API. Launch Postman and enter the HTTP GET command to URL http://localhost:8888/stats/desc/123917682138877 where port 8888 on the host machine corresponds to port 8080 in the vagrant box as explained in Section 2.2.2, and the dpid 123917682138877 is the decimal conversion of the MAC of the ZFX (70B3D56CDEFD). Sending this command to the Ryu controller will get the description of the ZFX switch.

In the next section we will run a simple scenario to further demonstrate the operations of our testbed.

3 A simple test scenario

In this section we demonstrate the operation of the testbed by a simple test scenario. We start from the default configuration for the OpenFlow switch (either the OVS or the ZFX) where the switch acts as a MAC-learning switch. Next we will manage the switch by the Ryu SDN controller, and see how the OpenFlow southbound communications kick in to populate the flow table on the switch when the data forwarding units exchange Internet Control Message Protocol (ICMP) echo request/reply messages (i.e., the ping command). Finally we modify the flow table by adding/deleting specific flow entries and observe the behaviour of the data plane has been altered accordingly.

We continue from the last section where we described procedures of setting up the testbed and basic verification steps that practically highlight the first half of the test scenario in this section. Further information and details for the flow table manipulation are presented below.

Note: We can always launch a Wireshark packet capture session before running the test cases to get detailed control messages and data flows in the data plane. However for brevity we will omit mentioning this step altogether in the rest of this section.

3.1 Emulation platform based on mininet

The first part of the test has been given in Section 2.1.3:

```
(first run: with OVS switch)
sudo mn --topo single,3 --mac --switch ovsk
mininet> dump
(list of network devices)
mininet> pingall
(success, 0% dropped)
mininet> dpctl show
(data path information)
mininet> exit
sudo mn –c

(second run: with a remote controller added)
sudo mn --topo single,3 --mac --switch ovsk --controller remote
… …
Setting remote controller to 127.0.0.1:6653
… …
mininet> pingall
(failed, 100% dropped)
```

This emulated mininet network consists of three hosts *h1, h2* and *h3*, an OVS switch *s1*, and an SDN controller *c0* (as seen from the dump command). In the first session above, the OVS switch is in default MAC-learning mode while the SDN controller is not controlling the network, hence the pingall command returned success. In the second session we specify that the network is to be controlled by the remote SDN controller, however there is no rules configured yet therefore the pingall command failed. Next we start the Ryu controller in a separate terminal:

```
ryu-manager --verbose ryu.app.simple_switch
```

by default the Ryu controller will communicate with the OpenFlow switch via host IP 127.0.0.1 (the loopback address) and IANA assigned OpenFlow port 6653, which matches the remote controller configuration in mininet, and loading of Ryu module simple_switch will configure the switch as a layer two

learning switch. If we run pingall again in mininet we will get a successful result. The detailed information and corresponding flow rules on the OVS switch can be found by the Open vSwitch command line (on a separate terminal):

```
sudo ovs-ofctl show s1
sudo ovs-ofctl dump-flows s1
```

To test the manipulation of flows via northbound APIs, we restart the mininet and Ryu controller sessions as described in Section 2.1.4:

```
(terminal 1, with remote Ryu controller
sudo mn --topo single,3 --mac --switch ovsk --controller remote
mininet> pingall

(terminal 2, note we loaded Ryu REST API module as well)
ryu-manager ryu.app.simple_switch ryu.app.ofctl_rest
```

then start a Postman session and we can get detailed switch information and flows (after running pingall in mininet to populate the flow table) using RESTful methods, some examples are listed in Table 1 below:

Table 1: *Sample RESTful HTTP methods.*

Method	HTTP URL	Purpose	Request body (JSON)	Response body
GET	localhost:8080/stats/switches	list all switches	none	one switch with dpid = 1
GET	localhost:8080/stats/desc/1	obtain detailed information for switch 1	none	description of the Open vSwitch
GET	localhost:8080/stats/flow/1	retrieve all flows on switch 1	none	list of all flows, 6 in total (in_port 1, 2, 3; out_port 1, 2, 3)
POST	Localhost:8080/stats/flow/1	retrieve flows that match criteria of out_port = 1	{ "out_port": 1 }	two flows, with in_port 1 and 3, respectively

Next we will use the HTTP POST command to modify the flow table by adding and deleting a flow entry, as depicted in Figure 4 and Figure 5 below. We intend to block any flows going through in_port 3 (which corresponds to the ingress direction of the Ethernet port on host 3). There used to be two flow entries in the table with dpid = 1 and match condition of in_port = 3, with priority value 32768 (the default priority value of the emulated network) and action [OUTPUT:1] and [OUTPUT:2], respectively, see the right subplot of Figure 5 for example. Now we issue the POST command to URL "http://localhost:8080/stats/flowentry/add" with the request body (in JavaScript Object Notation (JSON) format) as shown in the left subplot of Figure 4, where we add a flow entry with priority value 45000 and an empty action. The updated flow table is shown in the right subplot of Figure 4. As a result, the two flow rules for in_port = 3 will be overwritten (although still present in the table) due to the higher priority of the "no action" rule and subsequently any data packets coming to *h3* will be dropped. We can verify the changes at data plane by pingall command in mininet, we will see *h1* and *h2* can ping each other, but the ping commend to/from *h3* will fail.

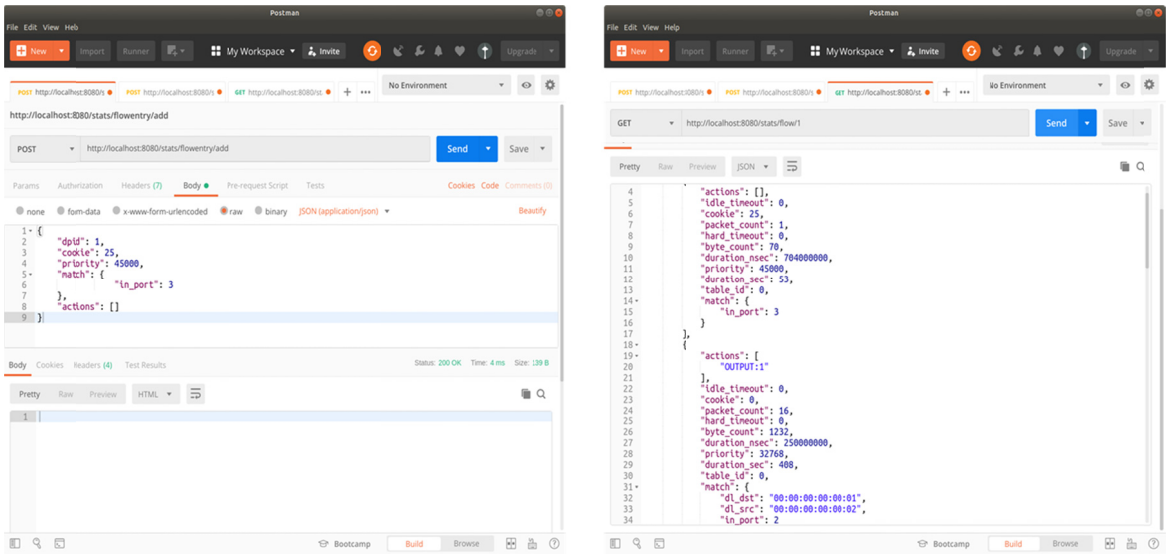

Figure 4: *Add a flow entry to block in_port 3 (mininet).*

Finally we resume the switch behaviour by deleting the added flow entry, the command is "POST http://localhost:8080/stats/flowentry/delete_strict" and the request body is shown in the left subplot of Figure 5, with the same JSON data as in Figure 4 (i.e., identify the same flow entry for deletion). We can verify the result by retrieving the flow table again (partly displayed in the right subplot of Figure 5) and run pingall in mininet.

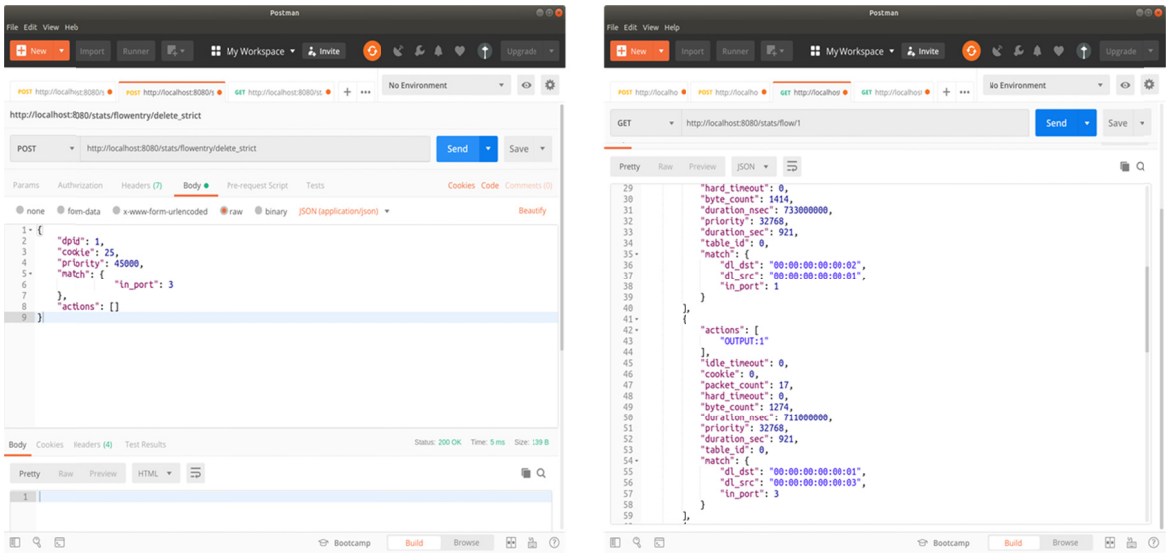

Figure 5: *Delete the flow entry to unblock in_port 3 (mininet).*

3.2 Hardware switch platform based on ZFX

For testing with ZFX, we first set up the data plane laptop as shown in Section 2.2.1, plug in USB adapters and launch three vagrant boxes (the data forwarding units), and validate the inter-connectivity by

connecting them to a network hub/switch. Next we follow the steps in Sections 2.2.2 and 2.2.3, connect the ZFX USB cable to the control plane laptop, and connect ZFX port 4 to the USB Ethernet adapter that is bridged to the Ryu vagrant box, then launch the Ryu controller box and confirm we can ping ZFX at 10.0.1.99 from within the Ryu vagrant box.

Next we move the Ethernet cables of the three data forwarding units from the network hub/switch to ZFX ports 1, 2 and 3. We can validate that there is no connectivity among them (command ping to other data forwarding units would fail) since the ZFX has not been configured yet. So we launch the Ryu manager inside the Ryu vagrant box:

- su - wlab
- (enter password for wlab)
- ryu-manager --ofp-listen-host=10.0.1.8 --ofp-tcp-listen-port=6633 ryu.app.simple_switch_13 ryu.app.ofctl_rest

then the three data forwarding units can ping each other as the ZFX behaves as a simple layer two switch now. Note we have also loaded the REST API module, so that we can manipulate the testbed network from Postman. We can issue the REST commands similar to those in Table 1, however the HTTP port should be 8888 and the dpid should be 123917682138877, as explained at the end of Section 2.2.3. In Figure 6 below we depict the HTTP GET results for ZFX description,[8] and a flow table entry that shows the flow for controller-switch communication.

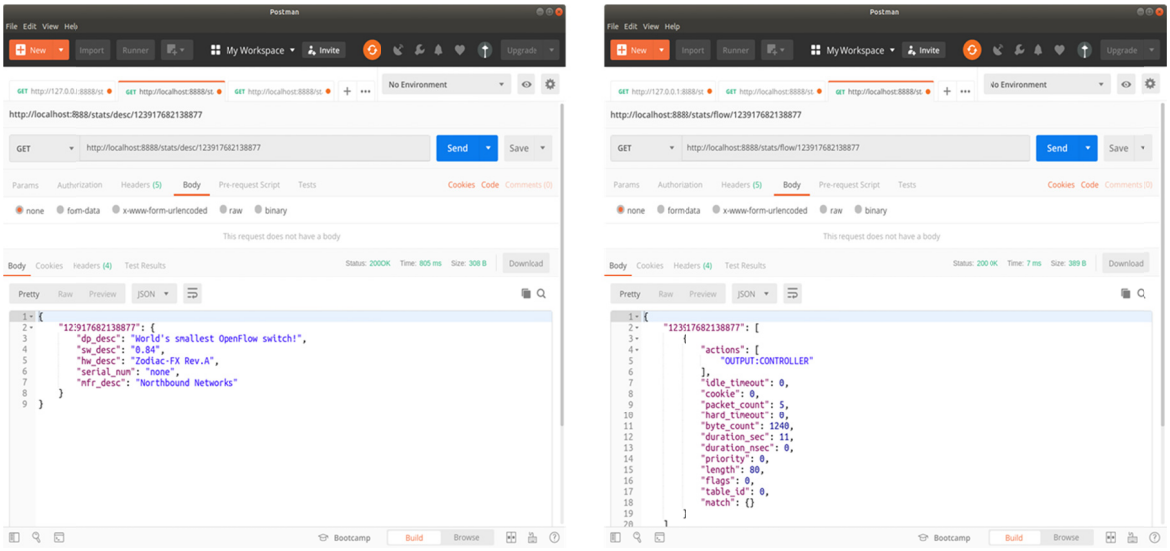

Figure 6: *ZFX description and flow between switch and controller.*

The next test is to add/delete a flow entry to modify the flow table on ZFX and verify the effect on the data plane networking behaviour, similar to the last section on the emulation platform. The Postman screen captures are depicted in Figure 7 and Figure 8, respectively.

[8] The display for switch description is inconsistent—if the GET command is sent again we may only get the dpid while the detailed information of ZFX is missing. This is possibly a bug in the ZFX firmware.

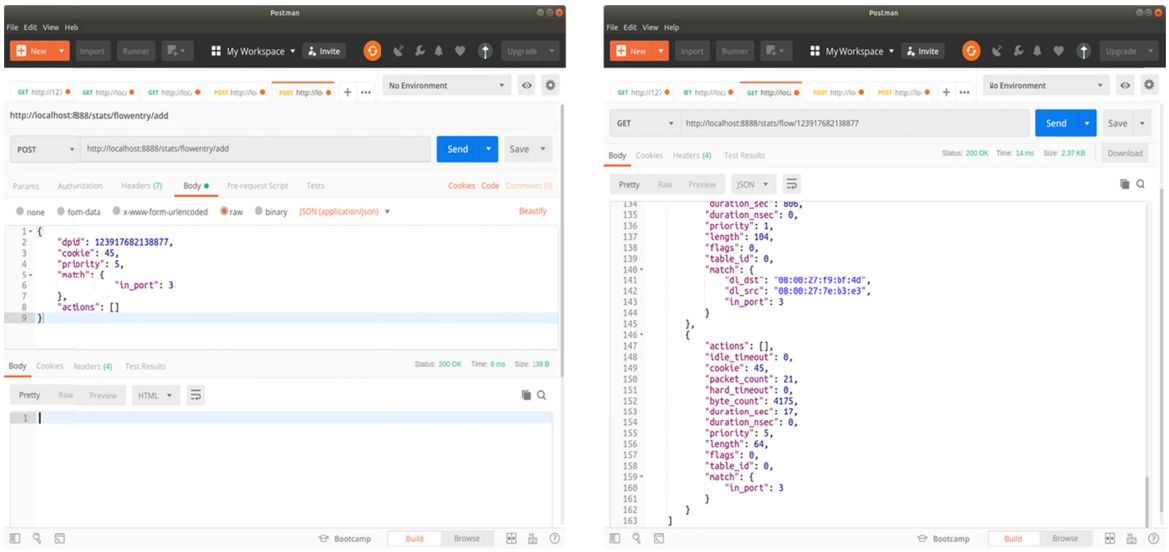

Figure 7: *Add a flow entry to block in_port 3 (ZFX).*

Note that for ZFX the flow table entries are slightly different from the emulation cases (e.g., the default priority value on ZFX is 1 for data flows and 0 for control flows), which reflect the real-world scenario of a hardware SDN switch. The validation of the flow manipulation effect on the data plane is done by pinging neighbors in the data forwarding unit vagrant boxes.

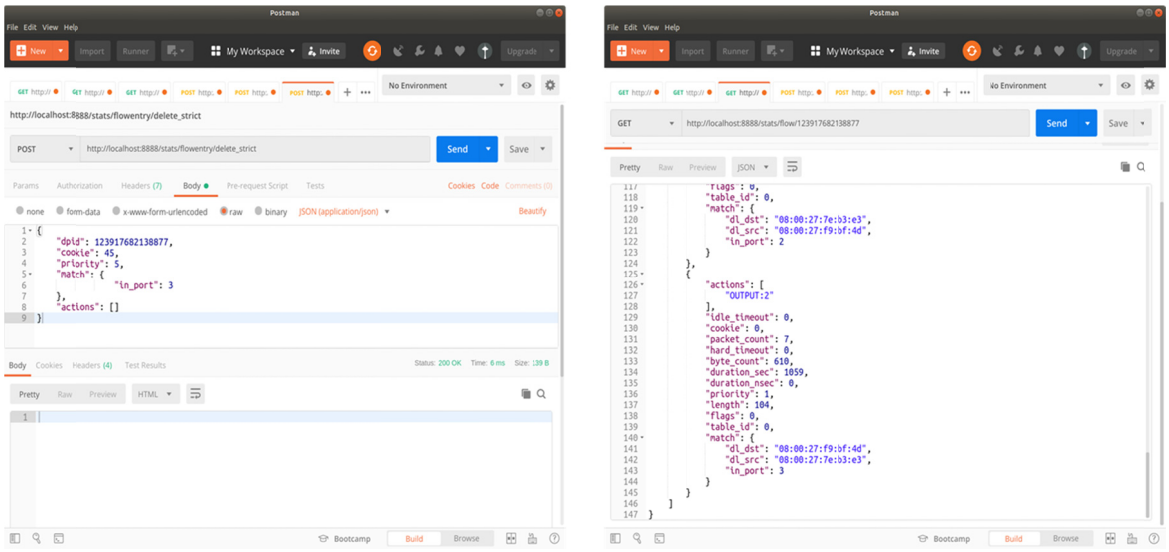

Figure 8: *Delete the flow entry to unblock in_port 3 (ZFX).*

3.3 Perl scripts for RESTful northbound applications

The Postman GUI is a good toolkit for SDN network manipulation and northbound API testing, however it has some limitations. For example, the GUI tool lacks a summary/statistics display for all flows, especially if the flow table has many entries, or in case of multiple tables. It is also not easy to pick up

specific table entries that we are interested in from a display window with so many lines.[9] There are many command line tools to carry out northbound API functionalities. During the testbed setup and testing we have written a set of Perl scripts to perform the following flow table manipulations: get all flows; get selected flow based on in_port; add a flow entry; and delete a flow entry. The scripts are presented in Annex B.

In the Perl scripts we have used some packages that may not come with the standard Perl installation on the host machine. The missing packages can be easily installed using the cpan tool, such as:

```
cpan
(cpan start up …)

cpan[1] install REST::Client
… …
cpan[2] install JSON
… …
cpan[3] quit
```

[9] We can use the POST command with proper JSON request to select flows, as depicted in the last entry of Table 1. However, the same method failed for ZFX, which is possibly another issue in the ZFX firmware.

4 Conclusion

In this Reference Document we described an SDN testbed with two configurations: on the emulation platform based on mininet and Open vSwitch; and on the hardware switch platform based on Zodiac FX. In the building of the testbed we implemented main functionalities across three layers in the SDN architecture: the data plane, the control plane and the application plane. We demonstrated some basic operations of software-defined networking by a simple test scenario.

Albeit simple and easy to set up, the testbed provides a fully functional platform to test SDN functionalities, and serves as a start point to design and develop SDN control software that would meet our specific requirements in the TTCP SDN project. We have chosen the components in the testbed setup and configuration so that they can be used as a framework for future research and development in software defined tactical networking.

The testbed is limited to wired networking at this stage. In the future work we will attempt to incorporate wireless mobile networking technologies into the SDN configuration, and investigate potential application of SDN techniques in military operations, particularly on the tactical edge.

References

[1] Diego Kreutz, Fernando M. V. Ramos, Paulo Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig, "Software-Defined Networking: A Comprehensive Survey," Proceedings of the IEEE, Vol. 103, No. 1, January 2015.

[2] Tricia Willink and Jon Spencer, "Software-Defined Networking," TTCP Technical Report TR-C3I-04-2106, January 2016.

[3] Open Networking Foundation and OpenFlow Technical Specifications, https://www.opennetworking.org/ Accessed April 2019.

[4] mininet, http://mininet.org/ Accessed April 2019.

[5] Zodiac FX switch, https://northboundnetworks.com/products/zodiac-fx Accessed April 2019.

[6] SDN controller Ryu, https://osrg.github.io/ryu/ Accessed April 2019.

[7] Open vSwitch, https://www.openvswitch.org/ Accessed April 2019.

[8] Postman API Development Platform, https://www.getpostman.com/ Accessed April 2019.

[9] Wireshark, https://www.wireshark.org/ Accessed April 2019.

[10] Oracle VirtualBox, https://www.virtualbox.org/ Accessed April 2019.

[11] Vagrant, https://www.vagrantup.com/ Accessed April 2019.

[12] POX, https://github.com/noxrepo/pox Accessed April 2019.

Annex A Vagrant box configuration and boot-up

A.1 Vagrantfile for Ryu controller

The *Vagrantfile* for the Ryu controller box is attached below for reference. Most of the content is generated by the vagrant init command, with some minor modifications for the Ryu controller configuration.

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

# All Vagrant configuration is done below. The “2” in Vagrant.configure
# configures the configuration version (we support older styles for
# backwards compatibility). Please don't change it unless you know what
# you're doing.
Vagrant.configure(“2”) do |config|
  # The most common configuration options are documented and commented below.
  # For a complete reference, please see the online documentation at
  # https://docs.vagrantup.com.

  # Every Vagrant development environment requires a box. You can search for
  # boxes at https://vagrantcloud.com/search.
  config.vm.box = “ubuntu/xenial64”

  # Disable automatic box update checking. If you disable this, then
  # boxes will only be checked for updates when the user runs
  # `vagrant box outdated`. This is not recommended.
  config.vm.box_check_update = false

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine. In the example below,
  # accessing “localhost:8080” will access port 80 on the guest machine.
  # NOTE: This will enable public access to the opened port
  # config.vm.network “forwarded_port”, guest: 80, host: 8080

  # Create a forwarded port mapping which allows access to a specific port
  # within the machine from a port on the host machine and only allow access
  # via 127.0.0.1 to disable public access
  # config.vm.network “forwarded_port”, guest: 80, host: 8080, host_ip: “127.0.0.1”
  # use the following to access ryu topology viewer,
  # or use guest firefox to 0.0.0.0:8080
```

```
  config.vm.network “forwarded_port”, guest: 8080, host: 8888, host_ip: “127.0.0.1”

  # Create a private network, which allows host-only access to the machine
  # using a specific IP.
  # config.vm.network “private_network”, ip: “192.168.33.10”

  # Create a public network, which generally matched to bridged network.
  # Bridged networks make the machine appear as another physical device on
  # your network.
  # config.vm.network “public_network”
  # for bridging to ZFX via USB ethernet adapter
  config.vm.network “public_network”, ip: “10.0.1.8”

  # Share an additional folder to the guest VM. The first argument is
  # the path on the host to the actual folder. The second argument is
  # the path on the guest to mount the folder. And the optional third
  # argument is a set of non-required options.
  # config.vm.synced_folder “../data”, “/vagrant_data”

  # Provider-specific configuration so you can fine-tune various
  # backing providers for Vagrant. These expose provider-specific options.
  # Example for VirtualBox:
  #
  config.vm.provider “virtualbox” do |vb|
  #   # Display the VirtualBox GUI when booting the machine
  #   vb.gui = true
  #
  #   # Customize the amount of memory on the VM:
  #   vb.memory = “1024”
      vb.memory = “2048”
  end
  #
  # View the documentation for the provider you are using for more
  # information on available options.

  # Enable provisioning with a shell script. Additional provisioners such as
  # Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the
  # documentation for more information about their specific syntax and use.
  # config.vm.provision “shell”, inline: <<-SHELL
  #   apt-get update
  #   apt-get install -y apache2
  # SHELL
end
```

A.2 Vagrant box boot-up screenshot

The Ryu vagrant box boot up process is depicted in Figure A.1 below. Note that network interface 2 refers to the USB Ethernet adapter which should be plugged in before starting the vagrant box.

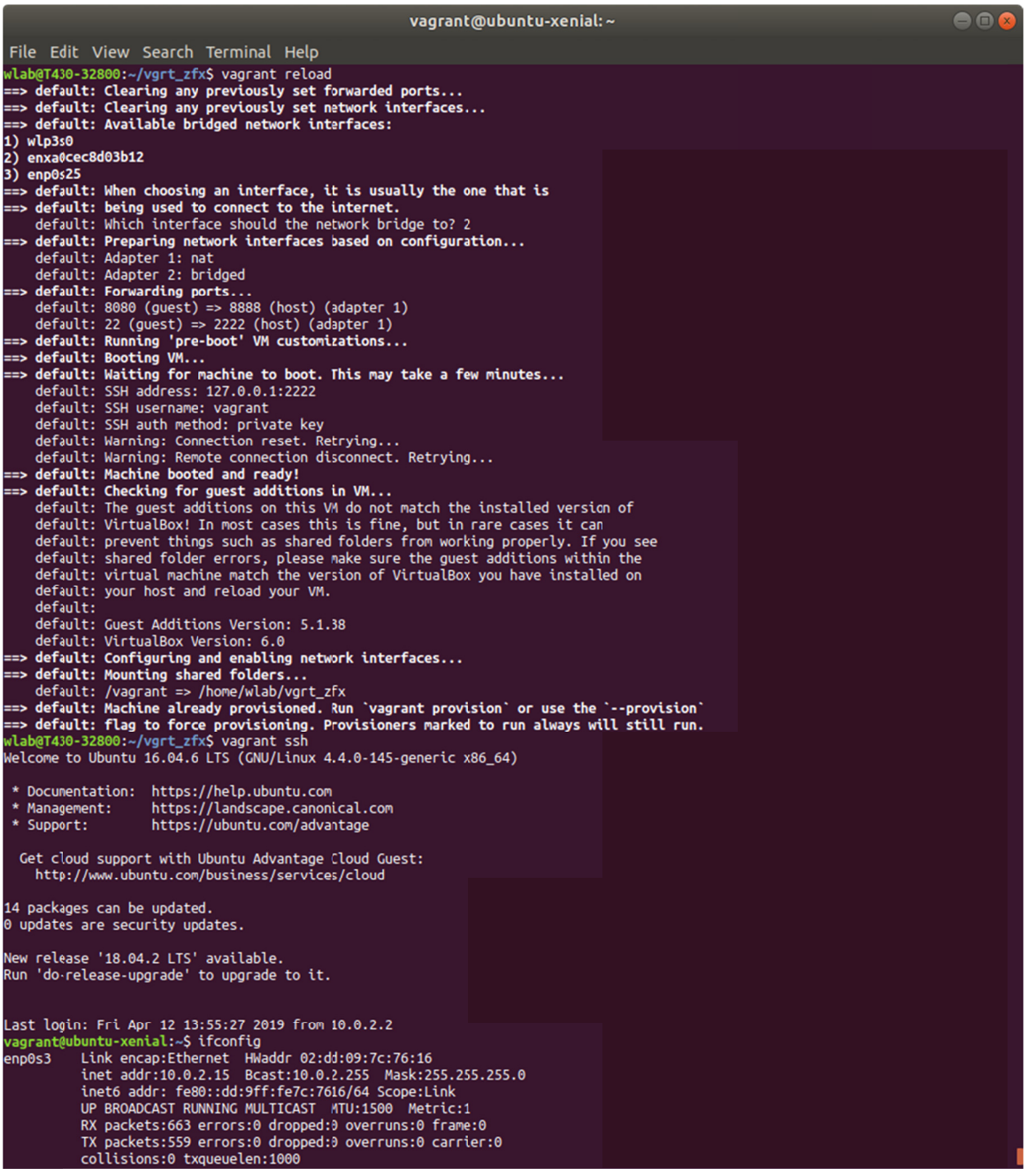

Figure A.1: *Boot up of the Ryu controller vagrant box.*

Annex B Perl scripts for RESTful APIs

The Perl scripts listed here are for interacting with the Ryu controller on the hardware switch testbed; hence the HTTP commands are issued to URL http://127.0.0.1:8888 as explained in Section 2.2.2. It is straightforward to apply the scripts in other cases with minor changes in the URL, for example to use the scripts in the emulation network in this document we just need to change the corresponding URLs to http://127.0.0.1:8080 (i.e., http://localhost:8080).

Note that we made no effort to optimize the Perl scripts, they are quoted in this document for illustration purpose only. The usage of each script can be found by the --help option in the command line:

```
script_name --help
```

and most of the input parameters are self-explanatory.

B.1 Get all flows: zfx_get_flows.pl

This script is used to query all flow entries on an SDN switch. Flow details are presented, with a statistical summary at the end of output.

```
#!/usr/bin/perl
#
use strict;
use warnings;
use locale;
use Carp;
use Getopt::Long;
use REST::Client;
use JSON;
use Data::Dumper;

use vars qw($DBG $WORK_DIR $help $dpid);
$DBG = 1;
$WORK_DIR = $ENV{PWD};
use vars qw($myurl $client $json_text $perl_scalar @Lines $line);
use vars qw($tmp $key $val $lbr $rbr %Flows);
use vars qw($table_id $in_port $out_port $cookie);
use vars qw($hard_tm $idle_tm $pkt_count);

usage() if ( @ARGV < 1 or
        !GetOptions('help|?'  => \$help,
                    'dpid=s'  => \$dpid)
                or defined $help
                or !(defined $dpid));
```

```
$client = REST::Client->new();

$client->addHeader('Content-Type', 'application/json');
$client->addHeader('charset', 'UTF-8');
$client->addHeader('Accept', 'application/json');

$myurl = “http://127.0.0.1:8888/stats/flow/$dpid”;
$client->GET($myurl);
# $client->GET('http://localhost:8080/stats/flow/1');
$tmp = $client->responseCode();
if ($tmp eq '200')
{ print “\nGET $myurl succeeded\n”; }
else
{
  print “\nGET $myurl failed, error code $tmp\n”;
  exit;
}

$json_text = $client->responseContent();

if ($DBG >= 1)
{ print “\n=== GET all flows (json) ===\n\n”, $json_text, “\n”; }

$perl_scalar = decode_json($json_text);
if ($DBG >= 1)
{
  print “\n=== All flows (Dumper) ===\n\n”;
  print Dumper($perl_scalar); print “\n”;
}
@Lines = split(/\n/, Dumper($perl_scalar));
# use left-brace and right-brace match to identify flow blocks
$lbr = 0; $rbr = 0;
# $DBG = 2;
$DBG = 0;
foreach $line (@Lines)
{
  $line =~ s/^\s+//; $line =~ s/\s+$//;
  print $line, “\n” if ($DBG >= 1);
  if ($line =~ /\'(\d+)\' => \[/)
  {
    $tmp = $1;
    print “get dpid: [$tmp]\n” if ($DBG >= 1);
```

```
    $tmp == $dpid or croak “dpid mismatch!”;
  }
  elsif ($line eq “{“)
  {
    # new flow block
    $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2);
    $key = ““; $val = ““;
    $in_port = 0; $out_port = 0; $table_id = 0;
    $cookie = 0; $pkt_count = 0; $hard_tm = 0; $idle_tm = 0;
  }
  elsif ($line eq “'match' => {“)
  { $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line eq “'match' => {},”)
  { $lbr ++; $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /^\}/)
  { $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /\'in_port\' => (\d+)/)
  {  $in_port = $1; }
  elsif ($line =~ /\'OUTPUT:(\d+)\'/)
  {  $out_port = $1; }
  elsif ($line =~ /\'packet_count\' => (\d+)/)
  {  $pkt_count = $1; }
  # the last statement in foreach
  if ($lbr == 2 and $rbr == 2)
  {
    $key = “in_port: “.$in_port.”, out_port: “.$out_port;
    $val = “pkt_count: “.$pkt_count;
    $Flows{$key} = $val;
    if ($DBG >= 1)
    { print “Flow entry $key pkt_count $val\n”; }
    $lbr = 0; $rbr = 0;
  }
}

print “\n=== Flow Summary: ===\n\n”;
foreach $key (sort keys %Flows)
{
  print $key, “ => “, $Flows{$key}, “\n”;
}
print “\n”;
exit;

sub usage
```

```
{
  print “Unknown options: @_\n” if (@_);
  print “usage: program [--dpid id] [--help|-?]\n”;
  exit;
}
```

B.2 Get selected flows: zfx_filter_flow.pl

This script is used to get selected flows based on in_port and/or out_port of the flow attribute.

```
#!/usr/bin/perl
#
use strict;
use warnings;
use locale;
use Carp;
use Getopt::Long;
use REST::Client;
use JSON;
use Data::Dumper;

use vars qw($DBG $WORK_DIR $help $dpid $inPort $outPort);
$DBG = 1;
$WORK_DIR = $ENV{PWD};
use vars qw($myurl $client $json_text $perl_scalar @Lines $line);
use vars qw(%request $tmp $key $val $lbr $rbr %Flows @getFlow @tmpFlow);
use vars qw($table_id $in_port $out_port $cookie);
use vars qw($hard_tm $idle_tm $pkt_count);

# dpid is mandatory (=) and in_port etc. are optional (:)
# (should default to 0 but seems not working so need to initialize)

$inPort = -1; $outPort = -1;
usage() if ( @ARGV < 1 or
        !GetOptions('help|?'  => \$help,
                    'dpid=i'  => \$dpid,
                    'in:i'    => \$inPort,
                    'out:i'   => \$outPort)
            or defined $help
            or !(defined $dpid));

if ($DBG >= 1)
{ print “GetOpt: dpid $dpid, in_port $inPort, out_port $outPort\n”; }
```

```
$client = REST::Client->new();

$client->addHeader('Content-Type', 'application/json');
$client->addHeader('charset', 'UTF-8');
$client->addHeader('Accept', 'application/json');

$myurl = “http://127.0.0.1:8888/stats/flow/$dpid”;

%request = ();
if ($outPort > 0)
{
  $request{out_port} = $outPort;
}
if ($inPort > 0)
{
  # this is the format of hash within hash accepted by encode_json
  $request{match} = {“in_port” => $inPort};
}
$json_text = encode_json(\%request);

if ($DBG >= 1)
{ print “JSON request:\n”, $json_text, “\n\n”; }

$client->POST($myurl, $json_text);

$tmp = $client->responseCode();

if ($tmp eq '200')
{ print “\nPOST $myurl succeeded\n”; }
else
{
  print “\nPOST $myurl failed, error code $tmp\n”;
  exit;
}

$json_text = $client->responseContent();

if ($DBG >= 1)
{ print “\n=== GET all flows (json) ===\n\n”, $json_text, “\n”; }

$perl_scalar = decode_json($json_text);
if ($DBG >= 1)
```

```
{
  print “\n=== All flows (Dumper) ===\n\n”;
  print Dumper($perl_scalar); print “\n”;
}
@Lines = split(/\n/, Dumper($perl_scalar));
# use left-brace and right-brace match to identify flow blocks
$lbr = 0; $rbr = 0;
@getFlow = (); @tmpFlow = ();
$DBG = 0;
foreach $line (@Lines)
{
  push (@tmpFlow, $line.”\n”);
  $line =~ s/^\s+//; $line =~ s/\s+$//;
  print $line, “\n” if ($DBG >= 1);
  if ($line =~ /\'(\d+)\' => \[/)
  {
    $tmp = $1;
    print “get dpid: [$tmp]\n” if ($DBG >= 1);
    $tmp == $dpid or croak “dpid mismatch!”;
  }
  elsif ($line eq “{“)
  {
    # new flow block
    $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2);
    $key = ““; $val = ““;
    $in_port = 0; $out_port = 0; $table_id = 0;
    $cookie = 0; $pkt_count = 0; $hard_tm = 0; $idle_tm = 0;
  }
  elsif ($line eq “'match' => {“)
  { $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line eq “'match' => {},”)
  { $lbr ++; $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /^\}/)
  { $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /\'in_port\' => (\d+)/)
  {  $in_port = $1; }
  elsif ($line =~ /\'OUTPUT:(\d+)\'/)
  {  $out_port = $1; }
  elsif ($line =~ /\'packet_count\' => (\d+)/)
  {  $pkt_count = $1; }
  # the last statement in foreach
  if ($lbr == 2 and $rbr == 2)
  {
```

```
      $key = “in_port: “.$in_port.”, out_port: “.$out_port;
      $val = “pkt_count: “.$pkt_count;
      $Flows{$key} = $val;
      if ($DBG >= 1)
      { print “Flow entry $key pkt_count $val\n”; }
      $lbr = 0; $rbr = 0;
      # check flow filter
      if ((($in_port == $inPort) or $inPort < 0) and
          (($out_port == $outPort) or $outPort < 0))
      {
        $tmp = “matching flow entry “.$key.”, pkt_count “.$val.”\n\n”;
        print “find match $tmp \n” if ($DBG >= 1);
        push (@getFlow, $tmp);
        push (@getFlow, @tmpFlow);
      }
      else
      { @tmpFlow = (); }
    }
}

print “\n=== Flow Summary: ===\n\n”;
foreach $key (sort keys %Flows)
{
   print $key, “ => “, $Flows{$key}, “\n”;
}
print “\n\n”;

print @getFlow;
print “\n”;
exit;

sub usage
{
  print “Unknown options: @_\n” if (@_);
  print “usage: program --dpid id [--in port] [--out port] [--help]\n”;
  exit;
}
```

B.3 Add a flow: zfx_add_flow.pl

This script is used to add a flow entry to the flow table. The entry is constructed by in_port, priority and cookie values.

```
#!/usr/bin/perl
#
use strict;
use warnings;
use locale;
use Carp;
use Getopt::Long;
use REST::Client;
use JSON;
use Data::Dumper;

use vars qw($DBG $WORK_DIR $help $dpid);
$DBG = 1;
$WORK_DIR = $ENV{PWD};
use vars qw($myurl $client $json_text $perl_scalar @Lines $line);
use vars qw(%request $tmp $key $val $lbr $rbr %Flows);
use vars qw($table_id $in_port $out_port $cookie $priority);
use vars qw($hard_tm $idle_tm $pkt_count);

# dpid is mandatory (=) and in_port etc. are optional (:)
# (should default to 0 but seems not working so need to initialize)

$in_port = -1; $cookie  = -1; $priority = -1;
usage() if ( @ARGV < 1 or
        !GetOptions('help|?'  => \$help,
                    'dpid=i'  => \$dpid,
                    'in=i'    => \$in_port,
                    'cki=i'   => \$cookie,
                    'pri=i'   => \$priority)
            or defined $help
            or !(defined $dpid));

if ($DBG >= 1)
{ print “GetOpt: dpid $dpid, in_port $in_port, cookie $cookie priority $priority\n”; }

$client = REST::Client->new();

$client->addHeader('Content-Type', 'application/json');
```

```
$client->addHeader('charset', 'UTF-8');
$client->addHeader('Accept', 'application/json');

$myurl = “http://127.0.0.1:8888/stats/flowentry/add”;

%request = (
            “dpid”    => $dpid,
            “actions” => []
          );
if ($cookie > 0)
{ $request{cookie} = $cookie; }
if ($priority > 0)
{ $request{priority} = $priority; }
if ($in_port > 0)
{
  # this is the format of hash within hash accepted by encode_json
  $request{match} = {“in_port” => $in_port};
}
$json_text = encode_json(\%request);

if ($DBG >= 1)
{ print “JSON request:\n”, $json_text, “\n\n”; }

$client->POST($myurl, $json_text);

$tmp = $client->responseCode();

if ($tmp eq '200')
{ print “\nPOST $myurl succeeded\n”; }
else
{
  print “\nPOST $myurl failed, error code $tmp\n”;
  exit;
}

# GET flows again for double check

$myurl = “http://127.0.0.1:8888/stats/flow/$dpid”;
$client->GET($myurl);
$json_text = $client->responseContent();

if ($DBG >= 1)
{ print “\n=== GET all flows (json) ===\n\n”, $json_text, “\n”; }
```

```
$perl_scalar = decode_json($json_text);
if ($DBG >= 1)
{
  print “\n=== All flows (Dumper) ===\n\n”;
  print Dumper($perl_scalar); print “\n”;
}
@Lines = split(/\n/, Dumper($perl_scalar));
# use left-brace and right-brace match to identify flow blocks
$lbr = 0; $rbr = 0;
$DBG = 0;
foreach $line (@Lines)
{
  $line =~ s/^\s+//; $line =~ s/\s+$//;
  print $line, “\n” if ($DBG >= 1);
  if ($line =~ /\'(\d+)\' => \[/)
  {
    $tmp = $1;
    print “get dpid: [$tmp]\n” if ($DBG >= 1);
    $tmp == $dpid or croak “dpid mismatch!”;
  }
  elsif ($line eq “{“)
  {
    # new flow block
    $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2);
    $key = ““; $val = ““;
    $in_port = 0; $out_port = 0; $table_id = 0;
    $cookie = 0; $pkt_count = 0; $hard_tm = 0; $idle_tm = 0;
  }
  elsif ($line eq “'match' => {“)
  { $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line eq “'match' => {},”)
  { $lbr ++; $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /^\}/)
  { $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /\'in_port\' => (\d+)/)
  {  $in_port = $1; }
  elsif ($line =~ /\'OUTPUT:(\d+)\'/)
  {  $out_port = $1; }
  elsif ($line =~ /\'packet_count\' => (\d+)/)
  {  $pkt_count = $1; }
  # the last statement in foreach
  if ($lbr == 2 and $rbr == 2)
```

```
  {
    $key = “in_port: “.$in_port.”, out_port: “.$out_port;
    $val = “pkt_count: “.$pkt_count;
    $Flows{$key} = $val;
    if ($DBG >= 1)
    { print “Flow entry $key pkt_count $val\n”; }
    $lbr = 0; $rbr = 0;
  }
}

print “\n=== Flow Summary: ===\n\n”;
foreach $key (sort keys %Flows)
{
  print $key, “ => “, $Flows{$key}, “\n”;
}
print “\n”;
exit;

sub usage
{
  print “Unknown options: @_\n” if (@_);
  print “usage: program --dpid id --in port --cki cookie --pri priority [--help]\n”;
  exit;
}
```

B.4 Delete a flow: zfx_del_flow.pl

This script is used to delete a flow entry with the same parameters as specified in script zfx_add_flow.pl.

```
#!/usr/bin/perl
#
use strict;
use warnings;
use locale;
use Carp;
use Getopt::Long;
use REST::Client;
use JSON;
use Data::Dumper;

use vars qw($DBG $WORK_DIR $help $dpid);
$DBG = 1;
$WORK_DIR = $ENV{PWD};
```

```
use vars qw($myurl $client $json_text $perl_scalar @Lines $line);
use vars qw(%request $tmp $key $val $lbr $rbr %Flows);
use vars qw($table_id $in_port $out_port $cookie $priority);
use vars qw($hard_tm $idle_tm $pkt_count);

# dpid is mandatory (=) and in_port etc. are optional (:)
# (should default to 0 but seems not working so need to initialize)

$in_port = -1; $cookie  = -1; $priority = -1;
usage() if ( @ARGV < 1 or
          !GetOptions('help|?'  => \$help,
                      'dpid=i'  => \$dpid,
                      'in=i'    => \$in_port,
                      'cki=i'   => \$cookie,
                      'pri=i'   => \$priority)
              or defined $help
              or !(defined $dpid));

if ($DBG >= 1)
{ print “GetOpt: dpid $dpid, in_port $in_port, cookie $cookie priority $priority\n”; }

$client = REST::Client->new();

$client->addHeader('Content-Type', 'application/json');
$client->addHeader('charset', 'UTF-8');
$client->addHeader('Accept', 'application/json');

$myurl = “http://127.0.0.1:8888/stats/flowentry/delete_strict”;

%request = (
            “dpid”          => $dpid,
            “actions”       => [],
            “idle_timeout”  => 0,
            “hard_timeout”  => 0,
            “tiable_id”     => 0
          );
if ($cookie > 0)
{ $request{cookie} = $cookie; }
if ($priority > 0)
{ $request{priority} = $priority; }
if ($in_port > 0)
{
  # this is the format of hash within hash accepted by encode_json
```

```
  $request{match} = {“in_port” => $in_port};
}
$json_text = encode_json(\%request);

if ($DBG >= 1)
{ print “JSON request:\n”, $json_text, “\n\n”; }

$client->POST($myurl, $json_text);

$tmp = $client->responseCode();

if ($tmp eq '200')
{ print “\nPOST $myurl succeeded\n”; }
else
{
  print “\nPOST $myurl failed, error code $tmp\n”;
  exit;
}

# GET flows again for double check

$myurl = “http://127.0.0.1:8888/stats/flow/$dpid”;
$client->GET($myurl);
$json_text = $client->responseContent();

if ($DBG >= 1)
{ print “\n=== GET all flows (json) ===\n\n”, $json_text, “\n”; }

$perl_scalar = decode_json($json_text);
if ($DBG >= 1)
{
  print “\n=== All flows (Dumper) ===\n\n”;
  print Dumper($perl_scalar); print “\n”;
}
@Lines = split(/\n/, Dumper($perl_scalar));
# use left-brace and right-brace match to identify flow blocks
$lbr = 0; $rbr = 0;
$DBG = 0;
foreach $line (@Lines)
{
  $line =~ s/^\s+//; $line =~ s/\s+$//;
  print $line, “\n” if ($DBG >= 1);
  if ($line =~ /\'(\d+)\' => \[/)
```

```
  {
    $tmp = $1;
    print “get dpid: [$tmp]\n” if ($DBG >= 1);
    $tmp == $dpid or croak “dpid mismatch!”;
  }
  elsif ($line eq “{“)
  {
    # new flow block
    $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2);
    $key = ““; $val = ““;
    $in_port = 0; $out_port = 0; $table_id = 0;
    $cookie = 0; $pkt_count = 0; $hard_tm = 0; $idle_tm = 0;
  }
  elsif ($line eq “'match' => {“)
  { $lbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line eq “'match' => {},”)
  { $lbr ++; $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /^\}/)
  { $rbr ++; print “== brace pair $lbr-$rbr ==\n” if ($DBG >= 2); }
  elsif ($line =~ /\'in_port\' => (\d+)/)
  {  $in_port = $1; }
  elsif ($line =~ /\'OUTPUT:(\d+)\'/)
  {  $out_port = $1; }
  elsif ($line =~ /\'packet_count\' => (\d+)/)
  {  $pkt_count = $1; }
  # the last statement in foreach
  if ($lbr == 2 and $rbr == 2)
  {
    $key = “in_port: “.$in_port.”, out_port: “.$out_port;
    $val = “pkt_count: “.$pkt_count;
    $Flows{$key} = $val;
    if ($DBG >= 1)
    { print “Flow entry $key pkt_count $val\n”; }
    $lbr = 0; $rbr = 0;
  }
}

print “\n=== Flow Summary: ===\n\n”;
foreach $key (sort keys %Flows)
{
  print $key, “ => “, $Flows{$key}, “\n”;
}
print “\n”;
```

```
exit;

sub usage
{
  print “Unknown options: @_\n” if (@_);
  print “usage: program --dpid id --in port --cki cookie --pri priority [--help]\n”;
  exit;
}
```

List of symbols/abbreviations/acronyms/initialisms

API	Application Programming Interface
CAF	Canadian Armed Forces
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
IANA	Internet Assigned Numbers Authority
ICMP	Internet Control Message Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LTS	Long Term Support (Ubuntu)
MAC	Media Access Control
ONF	Open Networking Foundation
OS	Operating System
OVS	Open vSwitch
RAM	Random Access Memory
REST	Representational State Transfer
SDN	Software Defined Networking
TTCP	The Technical Cooperation Program
URL	Uniform Resource Locator
USB	Universal Serial Bus
ZFX	Zodiac FX

DOCUMENT CONTROL DATA

*Security markings for the title, authors, abstract and keywords must be entered when the document is sensitive

1. ORIGINATOR (Name and address of the organization preparing the document. A DRDC Centre sponsoring a contractor's report, or tasking agency, is entered in Section 8.)

DRDC – Ottawa Research Centre
Defence Research and Development Canada, Shirley's Bay
3701 Carling Avenue
Ottawa, Ontario K1A 0Z4
Canada

2a. SECURITY MARKING (Overall security marking of the document including special supplemental markings if applicable.)

CAN UNCLASSIFIED

2b. CONTROLLED GOODS

NON-CONTROLLED GOODS
DMC A

3. TITLE (The document title and sub-title as indicated on the title page.)

Emulation- and hardware-based testbed setup and configuration for software-defined networking

4. AUTHORS (Last name, followed by initials – ranks, titles, etc., not to be used)

Li, M.

5. DATE OF PUBLICATION (Month and year of publication of document.)	6a. NO. OF PAGES (Total pages, including Annexes, excluding DCD, covering and verso pages.)	6b. NO. OF REFS (Total references cited.)
November 2019	43	12

7. DOCUMENT CATEGORY (e.g., Scientific Report, Contract Report, Scientific Letter.)

Reference Document

8. SPONSORING CENTRE (The name and address of the department project office or laboratory sponsoring the research and development.)

DRDC – Ottawa Research Centre
Defence Research and Development Canada, Shirley's Bay
3701 Carling Avenue
Ottawa, Ontario K1A 0Z4
Canada

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.)	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
05ab	

10a. DRDC PUBLICATION NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.)	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)
DRDC-RDDC-2019-D149	

11a. FUTURE DISTRIBUTION WITHIN CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

Public release

11b. FUTURE DISTRIBUTION OUTSIDE CANADA (Approval for further dissemination of the document. Security classification must also be considered.)

12. KEYWORDS, DESCRIPTORS or IDENTIFIERS (Use semi-colon as a delimiter.)

software defined networking; emulation; lab testbed

13. ABSTRACT (When available in the document, the French version of the abstract must be included here.)

This Reference Document describes the setup of a Software-Defined Networking (SDN) testbed in a lab environment, aiming at providing a platform for exploring and practicing SDN operations. It serves a two-fold purpose: first as part of the research activity carried out at DRDC – Ottawa Research Centre for The Technical Cooperation Program (TTCP) C4I TP43 CP3 project, and secondly for the future investigation of the potential application of SDN techniques to support the CAF.

The testbed is designed and configured such that it includes the main components in the SDN architecture, representing key functionalities across the data plane, control plane and application plane. We construct the testbed with two configurations, one is based on an emulation network and the other uses an SDN switch hardware. After detailed description of the testbed setup, we present a simple test scenario to verify the testbed is performing SDN functions properly, and demonstrate some basic networking operations.

Le document de référence décrit la mise en place d'un banc d'essai de réseaux définis par logiciel (*Software-Defined Networking* [SDN]) en laboratoire, le but étant de fournir une plateforme d'exploration et d'essai destinée aux opérations SDN. L'objectif est double : d'une part, appuyer les activités de recherche menées par RDDC Ottawa dans le cadre du projet CP3 du groupe technique 43 du C3IR du Programme de coopération technique; d'autre part, évaluer l'applicabilité des techniques SDN aux opérations des FAC.

Le banc d'essai est conçu et configuré de manière à reproduire les principaux composants de l'architecture SDN, c'est-à-dire les fonctionnalités essentielles des plans de données, de contrôle et d'application. Il se décline en deux configurations, l'une qui repose sur un réseau d'émulation, et l'autre, sur un dispositif de commutation SDN. La description détaillée de la mise en place du banc d'essai est suivie d'une simulation simple permettant de vérifier si celui-ci exécute correctement les fonctions SDN, ainsi que d'exemples d'opérations réseau de base.