SDN-Cockpit Manual

December 13, 2017

Contents

1 Introduction

This document contains the manual for the SDN-Cockpit, an open-source SDN teaching framework developed by the Institute of Telematics (Karlsruhe Insitute of Technology). The main goal of the tool is to provide an easy-to-use environment for fast pace SDN application development with special support for executing automated evaluation scenarios. It builds on the mininet emulator and the Ryu controller software and uses Vagrant to allow easy setup and reproducibility. Figure 1 highlights the general architecture and workflow. The tool consists of two separate views. The first view – denoted *Editor-View* in the figure – is a simple text editor that shows the code of one or multiple SDN applications. The second view (*Output-View*) contains all visual outputs of a running SDN environment in a compressed form. This includes, for example, status messages sent by the SDN controller or the SDN application, debug information about the topology or details about the traffic that is generated.

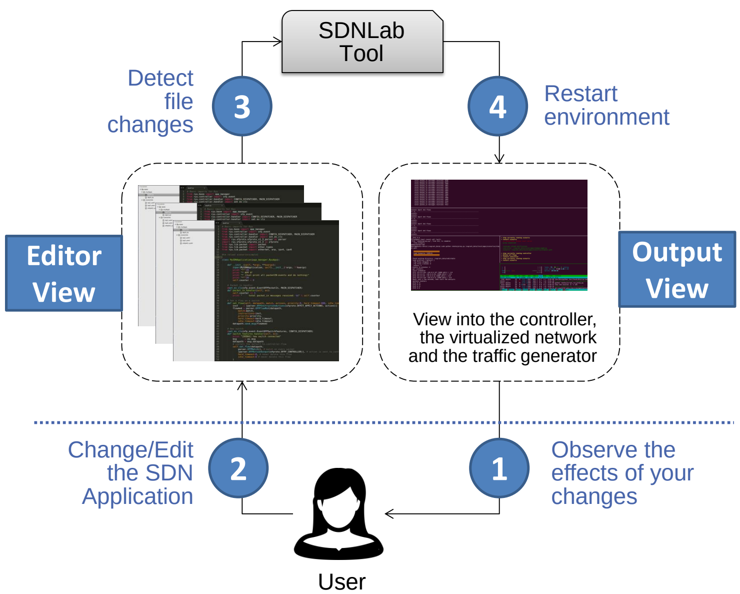

Figure 1: SDN-Cockpit: High Level View

For the best results, we suggest to provide a dual-monitor setup where the user can interact with both views at the same time. In the following, we briefly describe the high level workflow of the tool (visualized as the blue circles in the figure).

1. The user can interact with the Output-View in multiple ways. She gets a brief description about the currently loaded scenario and a short task description with a summary of the features for the to-be-developed SDN application. She also can directly interact with the virtualized network via the mininet command line interface, e.g., to start an iperf session between two virtual hosts. The Output-View further displays run-time information about the SDN application (such as print/echo commands) and the results of the automatic evaluation (if applicable). Currently, the Output-View is realized with tmux (see Section 4.4 for more details).

2. The information of the Output-View are used as a starting point for application development in the Editor View. The framework provides the user with several python files that contain empty SDN applications (Ryu controller applications). These files can be edited with any text-editor, e.g., gedit.

3. As soon as the user makes some changes to the application and saves the modified file, the framwork automatically detects the changes (Step 3) and restarts the SDN environment (Step 4). This includes a complete reset of the virtual network (mininet) and the traffic generation processes. The evaluation process is also restarted.

For a quick introduction of the main functions, please refer to Section 3.

2 Setup

The SDN-Cockpit can be installed on all major operating systems. The following sections show the installation process for Linux (section 2.1), macOS (section 2.2) and Windows (section 2.3). The SDN-Cockpit tool depends on git, vagrant (2.0.0) and virtualbox (v5.1).

2.1 Linux

Virtualbox can be optained either from your distributions packet repository or from the VirtualBox download page (`https://www.virtualbox.org/wiki/Download_Old_Builds_5_1`). We recommend the latter as only VirtualBox version 5.1 is supported. Select the entry corresponding to your distribution and install the package with your packet manager.

Vagrant can be optained from your distributions packet repository. Debian and Ubuntu provide outdated versions of Vagrant. As such, Vagrant should be downloaded directly from the Vagrant download page (`https://www.vagrantup.com/downloads.html`).

Open a new terminal session and continue with section 2.4.

2.2 macOS

First, install the Command Line Tools provided by Apple. For this, open a new terminal session by executing the `Terminal`-Application. It can be found under `Applications/Utilities/Terminal`. Enter `xcode-select --install` into the session. Confirm the installation by pressing `Install`. Then wait for the installation to complete.

Now install VirtualBox. Go to the VirtualBox download page (`https://www.virtualbox.org/wiki/Download_Old_Builds_5_1`) and select VirtualBox 5.1.30 for OS X hosts (Intel Macs). Open the downloaded image and follow the installation instructions.

Lastly, download Vagrant by going to its corresponding download page (`https://www.vagrantup.com/downloads.html`) and select Mac OS X 64-bit. Open the downloaded image and follow the installation instructions.

Open a new terminal session and continue with section 2.4.

2.3 Windows

First, download Git for Windows from the git download page (`https://git-scm.com/downloads`). Execute the installer and follow the installation instructions. Note that all options can be kept in their default configuration.

Then, install VirtualBox. Go to the VirtualBox download page (`https://www.virtualbox.org/wiki/Download_Old_Builds_5_1`) and select VirtualBox 5.1.30 for Windows hosts (x86/AMD64). Open the downloaded executable and follow the installation instructions.

Lastly, download Vagrant by going to its corresponding download page (`https://www.vagrantup.com/downloads.html`) and select Windows 64-bit. Open the downloaded executable and follow the installation instructions.

Additionally, we recommend to use notepad++ for editing the assignments as it handles UNIX-style line endings. It can be downloaded from (`https://notepad-plus-plus.org/download`).

Open a git bash session and change into the folder where SDN-Cockpit should reside by typing `cd <path>` where `<path>` is the path to this folder. Now clone the SDN-Cockpit repository by typing `git clone https://git.scc.kit.edu/internal/sdnlab.git`.

Enter the repository by typing `cd sdnlab`. As a linux virtual machine will access the repository, it needs to be checked out with UNIX-style line endings. Therefore type `git config core.autocrlf false && git checkout master` to disable Windows-style line endings. Create the virtual machine by typing `vagrant up` and wait for the command to complete (It might take a few minutes). The virtual machine is now running. Enter the virtual machine by typing `vagrant ssh`.

2.4 Common

Change into the folder where SDN-Cockpit should reside by typing `cd <path>` where `<path>` is the path to this folder. Now clone the SDN-Cockpit repository by typing `git clone https://git.scc.kit.edu/internal/sdnlab.git`.

Enter the repository by typing `cd sdnlab` and create the virtual machine by typing `vagrant up` and wait for the command to complete (It might take a few minutes). The virtual machine is now running. Enter the virtual machine by typing `vagrant ssh`.

3 Quick Start

After entering the virtual machine with the ssh command, you can start the framework by executing the run.sh script:

```
home@pc:$ vagrant ssh
ubuntu@ubuntu-xenial:/vagrant_data$ bash run.sh
```

Listing 1: Starting the framework

Note that the working directory is automatically changed during login. The VAGRANT_DATA folder should contain the project folder (i.e., the folder that was cloned from the git repository). Vagrant is configured to synchronize the project folder (outside of the VM) and the VAGRANT_DATA folder (inside the VM). Figure 2 shows the Output View of the framework. The view consists of four different panes:

Top left: This pane shows the execution of the SDN controller. The controller is restarted if the application is saved in the Editor View.

Top right: This pane shows additional information about the current task. Similar to the SDN controller view, the current task and the contents of this pane are updated, as soon as a task file (e.g., task21.py) is changed/saved.

Bottom left: This pane shows information about the scenario and the traffic generation process.

Bottom right: This pane gives the user access to the mininet terminal. Via the mininet terminal, it is possible to execute every command that is supported by the mininet CLI, e.g., pingall. See Section 4.1 for more details.

The four panes of the Output View are realized as a tmux session (tmux is a terminal multiplexer). After starting the framework, the mininet terminal (bottom right) should be selected as active pane. This means, that all inputs are directed to mininet. You can change the active pane by entering into the control mode (press CTRL + b) followed by one of the arrow keys. To scroll up inside a terminal, switch to control mode (again, CTRL + b) and press

Figure 2: Output-View of the framework

Page-UP/DOWN (in the right above the arrow keys). If you want to quit, press STRG+b followed by d.

4 Workflow

4.1 mininet

SDN-Cockpit is a tool built on top of mininet. It is responsible for building a virtual network topology which consists of hosts, SDN-switches as well as interconnections between those network elements. mininet further provides a command line interface to interact with the network. It is accessible from the SDN-Cockpit tool in the center-right pane. This section presents a basic overview of available commands.

Pairwise connectivity between all hosts can be tested with the `pingall` command. For every host it executes a ping request to every other host. This is useful to test basic reachability of all hosts and thus to detect potential bugs with the application. `pingall` prints a tabular view. Each row represents a sending host and each column a receiving host. A successfull ping request is denoted by the name of the receiving host. If the request was not successfull, “`X`” will be printed.

Furthermore, commands can be executed in the context of a host. This is achieved by prepending the host name to the command in question. Sending a `ping` request from host `h1` to host `h2` therefore translates to `h1 ping h2`.

The directional bandwidth between two hosts can be tested with `iperf`. Create an `iperf` server on one host with `<shost> iperf -s` where `<shost>`

is the name of this host. Then create an iperf client on another host with <chost> iperf -c <rhost> where <chost> is the name of this host. Data will be transmitted from <chost> to <shost> in order to estimate available bandwidth. When the transmission ends, a summary of the estimated bandwidth will be printed. Please note, that hosts in the SDN-Cockpit are currently named according to the autonomous systems which they represent, i.e., AS1, AS2022, ...

If you wish to inspect forwarding rules, which are currently active within your network, you can use the dpctl dump-flows command directly from the mininet console. This should help you verify that the forwarding rules, which you program into the network, are applied as expected.

4.2 Controller/Application

Throughout the lab we use the SDN controller implementation ryu, which is executed in the background, and refreshed whenever you save changes in your controller application. You can find the online documentation of the ryu API at this location: http://ryu.readthedocs.io/en/latest/api_ref.html

Interaction with the controller has been partially simplified by providing you with the abstraction class SDNApplication. To create a new SDN application navigate to the sync/local/apps/src folder in the SDN-Cockpit directory structure and have your controller class extend the SDNApplication located in controller.py through inheritance. This class offers you the two functions set_flow and send_pkt as convenience methods to simplify interactions with an SDN switch.

The function set_flow has the following synopsis:

```
def set_flow(self, datapath, match, actions,
    priority = 0, hard_timeout = 600, idle_timeout = 60)
```

The datapath variable is a reference to the switch, on which the controller wishes to install the new flow. This reference can be obtained from an OpenFlow event, that the controller receives when the configuration of a switch changes or when packets are forwarded to the controller (see the packet_in_handler function below). The match parameter expects an OFPMatch object, which can be constructed in the following way:

```
match = parser.OFPMatch(
    eth_type = ether_types.ETH_TYPE_IP,
    ipv4_src = src_ip
)
```

The constructor of a match object accepts numerous parameters, which represent the diffent match types that are supported by a generic SDN switch. For a full list of available parameters, please consult the ryu documentation under http://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html?highlight=OFPMatch#flow-match-structure. There is one caveat with ryu that you should keep in mind: ryu will silently discard any new flow rule if you fail to include a match on underlying protocol *types*. This is why in the example above an explicit match on the Ethernet type is performed (to ensure IP matching capabilites) in order to allow for a match on the IP source address.

The `actions` parameter expects a list of OpenFlow actions. Normally you will just want to drop packets or output them on a specific port. This can easily be achieved by passing a list of actions like this, with the variable `out_port` set accordingly:

```
[parser.OFPActionOutput(out_port)] # Output packet
[] # Drop packet
```

The parameter `priority` will determine the relative precedence of installed flow rules. You should always use a priority greater than 0 to allow the default flow rule to forward packets to the SDN controller when a table miss occurs. (This default flow rule is automatically installed with priority 0 by the SDN controller.) Finally, the `hard_timeout` and `idle_timeout` parameters determine the maximum lifetime, and the lifetime after a flow rule was last invoked (in seconds).

The second function provided by the `SDNApplication` class is `send_pkt`, which helps you to send packets via a specified port of an SDN switch from the controller. You can use this function to retransmit packets, which were redirected to the controller when a table miss occured.

```
def send_pkt(self, datapath, data,
    port = ofproto.OFPP_FLOOD)
```

Like the `set_flow` function, the `datapath` parameter determines the switch the controller wishes to interact with. The parameter `data` contains the actual packet data, which is to be transmitted, while the `port` parameter determines the output port, over which that packet will be sent. As indicated in the example, a special `OFPP_FLOOD` port can be specified to sent a packet over all ports of an SDN switch.

Furthermore, you should be aware of the function `packet_in_handler`:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev)
```

This function is invoked by the controller and delivers packets to your application when a table miss occurs in a switch. You should implement this function if you wish to extend or alter the current behaviour of the SDN controller. You can see the `demo.py` SDN application for an implementation example. Note, that through the `ev` parameter the application receives an event containing the available information on a packet that has been forwarded to the controller. For example the datapath, which connects the controller to the switch is a property of `ev` and can be used for subsequent flow processing as descibed above:

```
datapath = ev.msg.datapath
```

Finally, if you wish to perform any actions ahead of time and before individual messages are handled by the SDN controller, you can implement a handler with the following prototype:

```python
@set_ev_cls(ofp_event.EventOFPSwitchFeatures,
    MAIN_DISPATCHER)
def main_dispatcher_switch_features_handler(self,
    ev)
```

This will register the function `main_dispatcher_switch_features_handler` to `ryu` in such a way, that it will be invoked once a new switch has been detected and fully configured. You can use this function to install preemtive flow rules. The implementation corresponds to the `packet_in` function, except that no packet data will be available.

For more details on controller application programming see the `demo.py` application in the `sync/local/apps/src` folder of the SDN-Cockpit directory tree and the online `ryu` documentation.

If you should find a need to execute concurrent tasks within your controller application (e.g., polling flow rule statistics), you can use the ryu switching hub module to register a handler routine, which is executed in a concurrent fashion[1]. Periodic execution of your handler routine can be achieved by invoking the sleep method of the hub module from within the handler routine itself. The following gives an example of a simple monitoring solution, which periodically polls switch statistics.

```python
from ryu.lib import hub

def __init__(self, *args, **kwargs):
    # ...
    self.monitor_thread = hub.spawn(self._monitor)

def _monitor(self):
    while True:
        # Poll all switches for statistics ...
        hub.sleep(10)
```

You can find a more elaborate example at `https://osrg.github.io/ryu-book/en/html/traffic_monitor.html`.

4.3 Traffic Generator

In some of the tasks you will have to steer traffic streams in accordance with some given routing policy. These traffic streams will be automatically generated whenever you save changes to your application. You can confirm that your application performs routing as required by the routing policy by observing the output of the SDN-Cockpit tool. Whenever the traffic deviates from the expected behaviour you will be informed about how many packets were expected at a host and how many unexpected packets were recieved. This is displayed in the lower left window of the output view. If you accomplised a correct programming of the SDN application success will be indicated.

[1]Using the standard python concurrency features instead of the ryu switching hub may lead to problems.

4.4 tmux

tmux is a terminal multiplexer for the command line. SDN-Cockpit uses it to arrange the output of the various tools integrated into the development environment in an clear and compact fashion. By default only the mininet terminal is directly accessible, but tmux allows you to navigate through the multiple windows that are displayed on your screen. For your convenience a quick reference to the commands available in tmux is included in the follwing. Please note that the session handling commands must be executed from a command line.

list key bindings	`Ctrl-b ?`

Window Handling	
create a new window	`Ctrl-b c`
rename a window	`Ctrl-b ,`
move to next window	`Ctrl-b n`
move to previous window	`Ctrl-b p`
jump to a window by number	`Ctrl-b [number of window]`
find a window by name	`Ctrl-b f`
display menu of all windows	`Ctrl-b w`
close a window	`Ctrl-b &`

Working with Panes	
split window vertically	`Ctrl-b %`
split window horizontally	`Ctrl-b ¨`
cycle through panes	`Ctrl-b o`
move to upper pane	`Ctrl-b [Up]`
move to lower pane	`Ctrl-b [Down]`
move to left pane	`Ctrl-b [Left]`
move to right pane	`Ctrl-b [Right]`
cycle through layouts	`Ctrl-b [Space]`

Resize panes	
enlarge left pane	`Ctrl-b Alt-[Right]`
enlarge right pane	`Ctrl-b Alt-[Left]`
enlarge upper pane	`Ctrl-b Alt-[Down]`
enlarge lower pane	`Ctrl-b Alt-[Up]`
close active pane	`Ctrl-b x`

Session Handling	
Start tmux	`tmux`
list sessions	`tmux ls`
start a named session	`tmux new -s [session name]`
attach to existing session	`tmux attach -t [session name]`
kill a session	`tmux kill-session -t [session name]`
name window of new session	`tmux new -s [session name] -t [window name]`