SDN-Cockpit Manual

February 28, 2018

Contents

1 Introduction **1**

2 Setup **3**

2.1 Linux . . . 3
2.2 macOS . . . 3
2.3 Windows . . . 4
2.4 Common . . . 4

3 Quick Start **4**

4 Workflow **5**

4.1 mininet . . . 5
4.2 Application Development . . . 7
4.2.1 The `set_flow` Function . . . 7
4.2.2 The `send_pkt` Function . . . 8
4.2.3 Events . . . 9
4.2.4 Concurrency . . . 10
4.3 Traffic Generator . . . 11
4.4 The tmux Terminal Multiplexer . . . 11

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

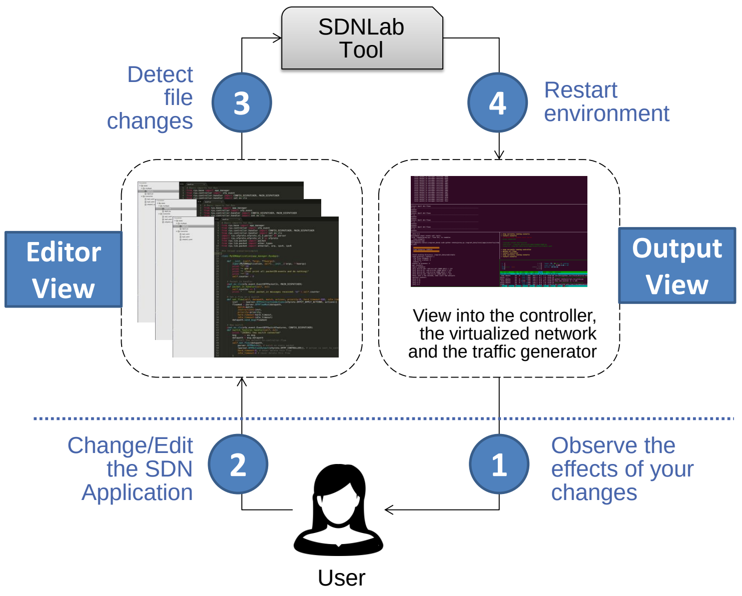

SDNLab Tool
Detect file changes
3
4
Restart environment
Editor View
Output View
View into the controller, the virtualized network and the traffic generator
Change/Edit the SDN Application
2
1
Observe the effects of your changes
User

Figure 1: SDN-Cockpit Tool: High Level View

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.

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 is 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 tool 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://github.com/KIT-Telematics/sdn-cockpit.git`.

Enter the repository by typing `cd sdn-cockpit`. 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 config core.eol lf && git reset --hard` 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://github.com/KIT-Telematics/sdn-cockpit.git`.

Enter the repository by typing `cd sdn-cockpit` 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
```

Note that the working directory is automatically changed during login. The folder `/vagrant_data` should contain the project folder (i.e., the folder that was cloned from the git repository). Vagrant is configured to synchronize this folder with the corresponding one outside of the VM. Any change to either one of the folders will be automatically performed on the counterpart.

Figure 2 shows the Output View of the framework. This view consists of four different panes:

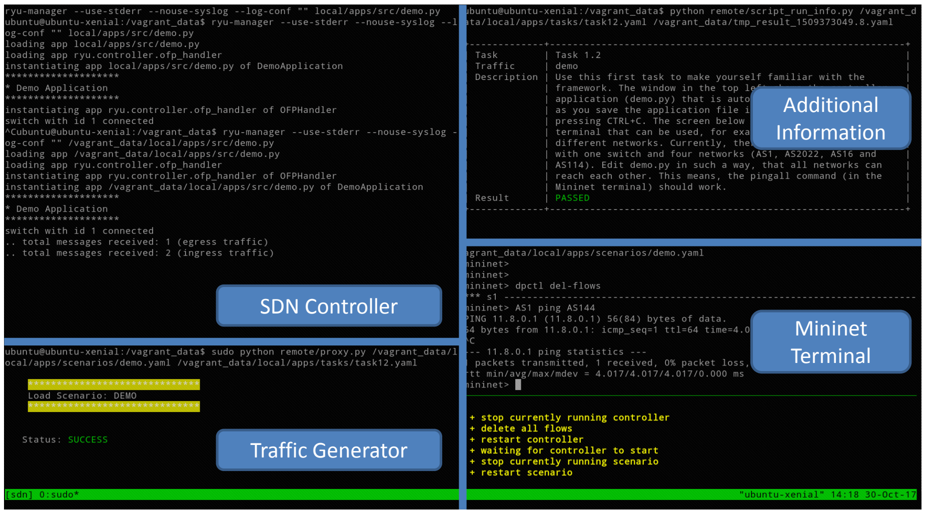
> **Top left:** This pane displays the output of the SDN controller. The controller is restarted whenever the application is saved in the Editor View to ensure that the latest changes to the application source code will take effect.
>
> **Top right:** This pane shows additional information about the 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. It displays the time-behaviour of the various traffic flows that are injected into the network.
>
> **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. For example, a user may use this console to manually test the connectivity between all hosts connected to the network with the `pingall` command. Section 4.1 provides a more detailed overview of commands available in the mininet network emulator.

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 (center-right) is automatically 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 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

To provide an authentic look and feel, we built SDN-Cockpit on top of the mininet network emulator. This emulator is responsible for building a virtual network topology which consists of hosts, SDN-switches as well as the links interconnecting the various network elements. mininet offers a command line interface for direct interaction with the network which is accessible from the SDN-Cockpit tool in the center-right pane. Here we present an overview of the relevant commands available in the mininet console.



```
ryu-manager --use-stderr --nouse-syslog --log-conf "" local/apps/src/demo.py
ubuntu@ubuntu-xenial:/vagrant_data$ ryu-manager --use-stderr --nouse-syslog --l
og-conf "" local/apps/src/demo.py
loading app local/apps/src/demo.py
loading app ryu.controller.ofp_handler
instantiating app local/apps/src/demo.py of DemoApplication
********************
* Demo Application
********************
instantiating app ryu.controller.ofp_handler of OFPHandler
switch with id 1 connected
^Cubuntu@ubuntu-xenial:/vagrant_data$ ryu-manager --use-stderr --nouse-syslog -
og-conf "" /vagrant_data/local/apps/src/demo.py
loading app /vagrant_data/local/apps/src/demo.py
loading app ryu.controller.ofp_handler
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app /vagrant_data/local/apps/src/demo.py of DemoApplication
********************
* Demo Application
********************
switch with id 1 connected
.. total messages received: 1 (egress traffic)
.. total messages received: 2 (ingress traffic)
```

```
ubuntu@ubuntu-xenial:/vagrant_data$ sudo python remote/proxy.py /vagrant_data/l
ocal/apps/scenarios/demo.yaml /vagrant_data/local/apps/tasks/task12.yaml

    ******************************
    Load Scenario: DEMO
    ******************************


   Status: SUCCESS

[sdn] 0:sudo*
```

```
ubuntu@ubuntu-xenial:/vagrant_data$ python remote/script_run_info.py /vagrant_d
ata/local/apps/tasks/task12.yaml /vagrant_data/tmp_result_1509373049.8.yaml

+-------------+------------------------------------------------------------+
 Task        | Task 1.2                                                   |
 Traffic     | demo                                                       |
 Description | Use this first task to make yourself familiar with the     |
             | framework. The window in the top le                        |
             | application (demo.py) that is auto                         |
             | as you save the application file i                         |
             | pressing CTRL+C. The screen below                          |
             | terminal that can be used, for exa                         |
             | different networks. Currently, the                         |
             | with one switch and four networks (AS1, AS2022, AS16 and   |
             | AS114). Edit demo.py in such a way, that all networks can  |
             | reach each other. This means, the pingall command (in the  |
             | Mininet terminal) should work.                             |
 Result      | PASSED                                                     |
+-------------+------------------------------------------------------------+
```

```
agrant_data/local/apps/scenarios/demo.yaml
ininet>
ininet>
ininet> dpctl del-flows
** s1 ----------------------------------------------------------------------
ininet> AS1 ping AS144
ING 11.8.0.1 (11.8.0.1) 56(84) bytes of data.
4 bytes from 11.8.0.1: icmp_seq=1 ttl=64 time=4.0
C
-- 11.8.0.1 ping statistics ---
 packets transmitted, 1 received, 0% packet loss,
tt min/avg/max/mdev = 4.017/4.017/4.017/0.000 ms
ininet> 

 + stop currently running controller
 + delete all flows
 + restart controller
 + waiting for controller to start
 + stop currently running scenario
 + restart scenario

                                          "ubuntu-xenial" 14:18 30-Oct-17
```

Figure 2: Output-View of the framework

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 Application Development

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. A comprehensive online documentation of the `ryu` API is available at this location: `http://ryu.readthedocs.io/en/latest/api_ref.html`. Applications use this API to interact with the network in various ways. Since `ryu` uses a Python-based programming interface some familiarity with this programming language is naturally required. As a straigtforward approach to familiarize yourself with the Python programming language we advise you to take a look at the comprehensive tutorial offered by the Python community: `https://docs.python.org/2/tutorial/`. You may want to pay special attention to classes, inheritance and instance variables, if you are unfamiliar with object oriented programming in Python. For a quick reference you may find the more aspect-oriented approach to Python at `https://www.tutorialspoint.com/python/index.htm` more convenient.

The Interaction with the `ryu` controller is already simplified since we provide you with an abstraction class called `SDNApplication`. To get started on a new SDN application first 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 functions to simplify interactions with an SDN switch.

4.2.1 The `set_flow` Function

The `set_flow` function is used to install a new flow, i.e., a rule that determines how packets should be processed, on a switch. This function has the following signature:

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

`ryu` uses the term `datapath` to refer to the connection between a controller and a switch. The datapaths you will need to use are usually identified by OpenFlow events, which the controller receives whenever the configuration of a switch changes or packets are forwarded from a switch to the controller (see the `packet_in_handler` function below). You can use these to identify the switch on which you need to install new flows.

The `match` parameter determines which packets should be handled by a flow. It expects an `OFPMatch` object as argument, which can be constructed in the following way:

```
match = parser.OFPMatch(
    eth_type = ether_types.ETH_TYPE_IP,
    ipv4_src = ('10.0.0.1', '255.255.255.0')
)
```

The constructor of a match object accepts numerous parameters, which represent the diffent matches that are supported by an OpenFlow switch. The following table lists some of the more commonly occurring matches that are supported by the OpenFlow protocol:

Argument	Type	Description
`in_port`	32bit integer	Switch input port
`eth_src`	MAC address	Ethernet source address
`eth_dst`	MAC address	Ethernet destination address
`eth_type`	16bit integer	Ethernet source address
`ip_src`	IP address	IP source address
`ip_dst`	IP address	IP destination address

Please note, that certain parameters actually accept a range of values in the form of a combination of a fixed base-value and a mask. This is for example required when you want a flow to match on an entire IPv4 subnet. Parameters like these are usually provided as tuples, as it is shown in the previous matching example for the IPv4 source address `10.0.0.1` and the subnet mask `255.255.255.0`. 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 previous example an explicit match on the Ethernet type is necessary to ensure that the match on the IPv4 source address will actually be performed.

The `actions` parameter of the `set_flow` function 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 to the number of the output port:

```
[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 they do not match any of the flows in a flowtable. (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 (with their default values set to 60 and 600 seconds respectively).

4.2.2 The `send_pkt` Function

The second function provided by the `SDNApplication` class is `send_pkt`, which lets you send packets from the controller via a specific port of an SDN switch. You can use this function to retransmit packets, which were delivered to the controller.

```
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. This is usually the entire packet as it was received by a controller (see below on how to obtain these messages). Finally, 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. The specifc OpenFlow protocol `ofproto` (and its version) used in the interaction with a switch can be obtained from the OpenFlow messages recevied by a controller (see below).

4.2.3 Events

To allow reactive flow programming the `ryu` controller generates an event whenever a new packet arrives at the controller. Furthermore, it offers a way to designate a function to handle these events. You can implement the following function to add any specific packet handling logic you wish to perform:

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

You can take a look at the `demo.py` application provided in the SDN-Cockpit folder `sync/local/apps/src` 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 a switch is a property of `ev` and can be used for subsequent flow processing as described above:

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

It also contains the contents of the packet that has been forwarded to a controller. This data is accessible through the following property:

```
data = ev.msg.data
```

To make sure you interact with a switch through the correct OpenFlow protocol version the reveived event `ev` also contains this information when a packet arrives at the controller. You can obtain it by accessing the following property:

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

It is for example useful, when you want to specify an `OFPOutputAction` in a flow to forward packets over a specific port.

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 for this specific purpose:

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures,
    CONFIG_DISPATCHER)
def switch_features_handler(self, ev)
```

This will register the function `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 proactive flow rules. The implementation corresponds to the `packet_in_handler` 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.

4.2.4 Concurrency

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. A switching hub allows to register a handler routine, which is executed in a concurrent fashion[1]. This handler routine can then perform periodic tasks. The following gives an example of a simple monitoring solution, which polls switch statistics in an interval of 10 seconds:

```
from ryu.lib import hub

class Monitoring(SDNApplication):
    def __init__(self, *args, **kwargs):
        # Spawn a new hub instance
        self.monitor_thread = hub.spawn(self.monitor)

    def monitor(self):
        datapath = # obtain a datapath reference...
        while True:
            # Poll all switches for statistics ...
            datapath.send_msg(
                parser.OFPFlowStatsRequest(datapath)
            )
            hub.sleep(10)

    @set_ev_cls(ofp_event.EventOFPFlowStatsReply,
        MAIN_DISPATCHER)
    def handle_flow_stats(self, ev):
        # Stat processing logic
```

First of all the new handler routine `monitor` is registered via the `hub.spawn` function during the exection of the constructor `__init__`. This handler will

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

be executed immediately. The `monitor` function then proceeds to sending `OFPFlowStatsRequest` messages to a switch, thereby instructing a switch to send its collected statistics to the controller. The `hub.sleep` function limits the execution of this step to every 10 seconds. When the switch has sent the collected statistics, the `ryu` controller generates an `EventOFPFlowStatsReply` event. We can designate a function to handle exactly this type of events, so we can ensure, we are only dealing with flow stats within that method. You can see how to register such a function in the example above: take a look at the `handle_flow_stats` function. This function finally serves to process the collected statistics. 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.

4.4 The tmux Terminal Multiplexer

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.

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`