Міністерство освіти і науки України

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

ЗВІТ

з лабораторної роботи №5

з навчальної дисципліни «Програмування комп'ютерних та віртуальних мереж»

Виконав:

студент групи ІМ-31мп

Королюк Я. О.

Київ – 2023

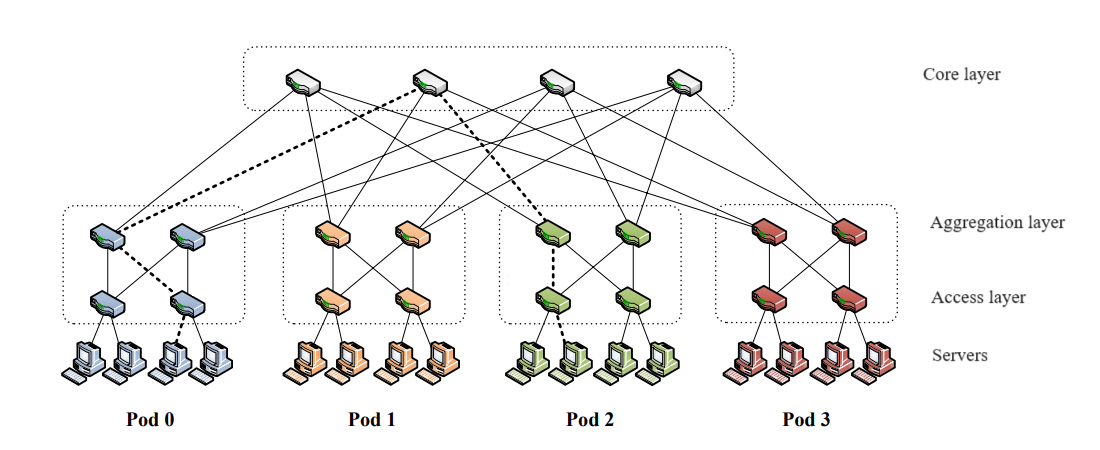
**Мета**: налаштувати керування продуктивністю каналів передачі повідомлень в залежності від добового часу та дня тижня у SDN мережі центру обробки даних (Data Center).

**Завдання**: підключити зовнішній контролер, що реалізує логіку продуктивності каналів передачі повідомлень в залежності від добового часу та дня тижня у SDN мережі, запрограмувати роботу Data Centre з розробленою Вами топологією.

**Номер у списку групи**: 10

**Хід роботи**

Насамперед необхідно розробити топологію центру обробки даних. За топологію було обрано товсте дерево (fat tree), адже вона забезпечує декілька з’єднань між елементами мережі, що робить її більш стресостійкою. Обрана топологія має наступний вигляд:



Ось приклад для k = 4, де k - кількість подів. Для створення топології в mininet було оформлено окремий скрипт. Він коректно працює для різної кількості подів, однак для подального дослідження було обрано 4 поди в топології. Скрипт топології (fat\_tree\_topology.py):

from mininet.topo import Topo

from mininet.net import Mininet

from mininet.node import OVSKernelSwitch, RemoteController

from mininet.log import setLogLevel

from mininet.cli import CLI

class FatTree(Topo):

switch\_counter = 0

host\_counter = 0

def \_\_init\_\_(self, k):

self.k = k

super().\_\_init\_\_()

def build(self):

core\_switches = []

for i in range(self.k):

core\_switches.append(self.\_\_create\_switch())

pods = []

for i in range(self.k):

aggregation\_switches = []

for j in range(self.k // 2):

aggregation\_switches.append(self.\_\_create\_switch())

access\_switches = []

for j in range(self.k // 2):

access\_switches.append(self.\_\_create\_switch())

for access\_switch in access\_switches:

for aggregation\_switch in aggregation\_switches:

self.addLink(aggregation\_switch, access\_switch)

for j in range(self.k // 2):

host = self.\_\_create\_host()

self.addLink(host, access\_switch)

pods.append(aggregation\_switches)

for i in range(self.k):

for j in range(0, self.k // 2):

self.addLink(pods[i][0], core\_switches[j])

for j in range(self.k // 2, self.k):

self.addLink(pods[i][1], core\_switches[j])

def \_\_create\_switch(self):

self.switch\_counter += 1

return self.addSwitch(f's{self.switch\_counter}')

def \_\_create\_host(self):

self.host\_counter += 1

return self.addHost(f'h{self.host\_counter}')

def create\_fat\_tree\_topo(k):

topo = FatTree(k)

net = Mininet(topo=topo, switch=OVSKernelSwitch, controller=RemoteController)

net.start()

for switch in net.switches:

switch.cmd("ovs-vsctl set Bridge {} protocols=OpenFlow13".format(switch.name))

switch.cmd("ovs-vsctl set-manager ptcp:6632")

CLI(net)

net.stop()

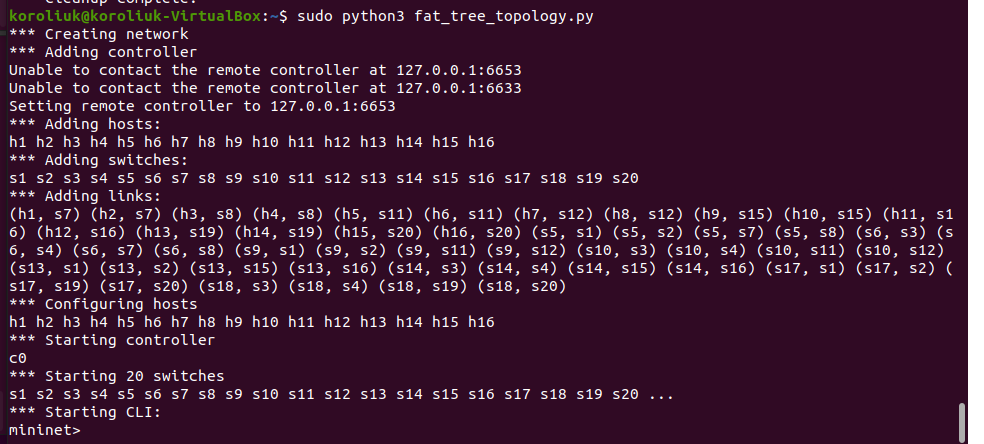
if \_\_name\_\_ == '\_\_main\_\_':

setLogLevel('info')

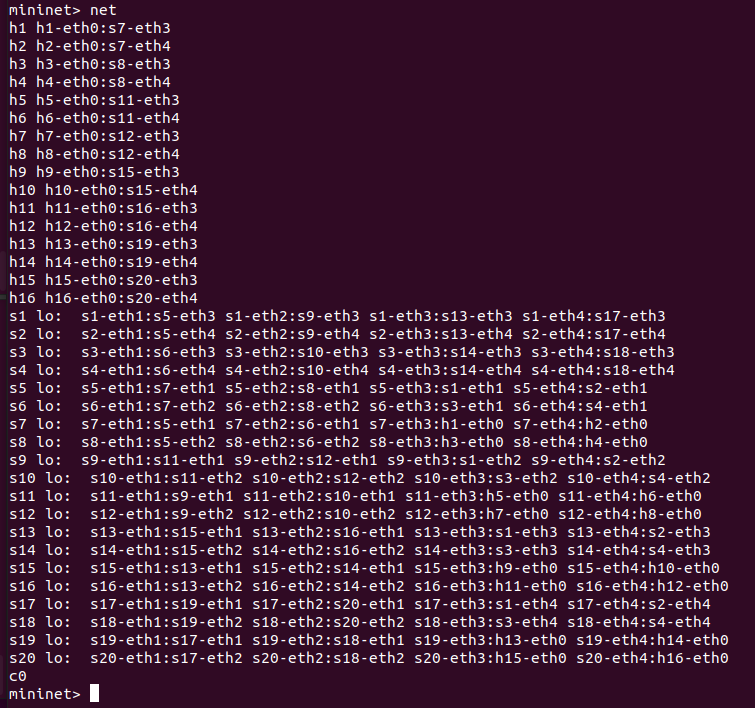
create\_fat\_tree\_topo(4)

Як бачимо було створено окремий клас *FatTree* для опису життєвого циклу топології в mininet. Цей клас при ініціалізації створює хости, комутатори та зв’язки між ними. Властивості зв’язків є дефолтними, їх в подальшому буде змінювати контролер. Функція *create\_fat\_tree\_topo()* приймає параметр кількості подів і власне створює клас топології. Також вона на кожному комутаторі встановлює версію OpenFlow протоколу, що буде використовуватись, та налаштовує з’єднання із OVSDB. В скрипті також вказується, що контролер - зовнішній. Крім цього, запуск та припинення командного рядка mininet, після того як топологія створення, також опрацьовується цієї функцією.

Скриншот запуску скрипта:



Можемо пересвідчитись в коректності роботи скрипта за допомогою команди *net*. Вона засвідчує, що скрипт створює топологію товстого дерева, яка має надлишкові зв’язки для забезпечення стійкості при локальних поломках:



Наступним кроком необхідно підключити контролер. Було обрано RYU контролер, так як POX не обробляє цикли в топології за замовчуванням. RYU має певні напрацювання, які працюють при циклах в топології. Однак наявні приклади треба редагувати, щоб забезпечити коректне збереження встановленої пропускної здатності в майбутньому. Розроблений контролер має наступний код (custom\_controller.py):

from ryu.base import app\_manager

from ryu.controller import ofp\_event

from ryu.controller.handler import CONFIG\_DISPATCHER

from ryu.controller.handler import MAIN\_DISPATCHER

from ryu.controller.handler import set\_ev\_cls

from ryu.lib import dpid as dpid\_lib

from ryu.lib import stplib

from ryu.lib.packet import ether\_types

from ryu.lib.packet import ethernet

from ryu.lib.packet import packet

from ryu.ofproto import ofproto\_v1\_3

class SimpleSwitch13(app\_manager.RyuApp):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(SimpleSwitch13, self).\_\_init\_\_(\*args, \*\*kwargs)

self.mac\_to\_port = {}

@set\_ev\_cls(ofp\_event.EventOFPSwitchFeatures, CONFIG\_DISPATCHER)

def switch\_features\_handler(self, ev):

datapath = ev.msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

# install table-miss flow entry

#

# We specify NO BUFFER to max\_len of the output action due to

# OVS bug. At this moment, if we specify a lesser number, e.g.,

# 128, OVS will send Packet-In with invalid buffer\_id and

# truncated packet data. In that case, we cannot output packets

# correctly. The bug has been fixed in OVS v2.1.0.

match = parser.OFPMatch()

actions = [parser.OFPActionOutput(ofproto.OFPP\_CONTROLLER,

ofproto.OFPCML\_NO\_BUFFER)]

self.add\_flow(datapath, 0, match, actions)

def add\_flow(self, datapath, priority, match, actions, buffer\_id=None):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

inst = [parser.OFPInstructionActions(ofproto.OFPIT\_APPLY\_ACTIONS,

actions)]

if buffer\_id:

mod = parser.OFPFlowMod(datapath=datapath, buffer\_id=buffer\_id,

priority=priority, match=match,

instructions=inst, table\_id=1)

else:

mod = parser.OFPFlowMod(datapath=datapath, priority=priority,

match=match, instructions=inst, table\_id=1)

datapath.send\_msg(mod)

@set\_ev\_cls(ofp\_event.EventOFPPacketIn, MAIN\_DISPATCHER)

def \_packet\_in\_handler(self, ev):

# If you hit this you might want to increase

# the "miss\_send\_length" of your switch

if ev.msg.msg\_len < ev.msg.total\_len:

self.logger.debug("packet truncated: only %s of %s bytes",

ev.msg.msg\_len, ev.msg.total\_len)

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

in\_port = msg.match['in\_port']

pkt = packet.Packet(msg.data)

eth = pkt.get\_protocols(ethernet.ethernet)[0]

if eth.ethertype == ether\_types.ETH\_TYPE\_LLDP:

# ignore lldp packet

return

dst = eth.dst

src = eth.src

dpid = format(datapath.id, "d").zfill(16)

self.mac\_to\_port.setdefault(dpid, {})

self.logger.info("packet in %s %s %s %s", dpid, src, dst, in\_port)

# learn a mac address to avoid FLOOD next time.

self.mac\_to\_port[dpid][src] = in\_port

if dst in self.mac\_to\_port[dpid]:

out\_port = self.mac\_to\_port[dpid][dst]

else:

out\_port = ofproto.OFPP\_FLOOD

actions = [parser.OFPActionOutput(out\_port)]

# install a flow to avoid packet\_in next time

if out\_port != ofproto.OFPP\_FLOOD:

match = parser.OFPMatch(in\_port=in\_port, eth\_dst=dst, eth\_src=src)

# verify if we have a valid buffer\_id, if yes avoid to send both

# flow\_mod & packet\_out

if msg.buffer\_id != ofproto.OFP\_NO\_BUFFER:

self.add\_flow(datapath, 1, match, actions, msg.buffer\_id)

return

else:

self.add\_flow(datapath, 1, match, actions)

data = None

if msg.buffer\_id == ofproto.OFP\_NO\_BUFFER:

data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer\_id=msg.buffer\_id,

in\_port=in\_port, actions=actions, data=data)

datapath.send\_msg(out)

class SimpleSwitch13Stp(SimpleSwitch13):

OFP\_VERSIONS = [ofproto\_v1\_3.OFP\_VERSION]

\_CONTEXTS = {'stplib': stplib.Stp}

def \_\_init\_\_(self, \*args, \*\*kwargs):

super(SimpleSwitch13, self).\_\_init\_\_(\*args, \*\*kwargs)

self.mac\_to\_port = {}

self.stp = kwargs['stplib']

def generate\_fattree\_config(k):

num\_switches = k + k \*\* 2

config = {}

priority = 0x8000

for i in range(1, num\_switches + 1):

switch\_id = f"{i:016x}"

config[dpid\_lib.str\_to\_dpid(switch\_id)] = {'bridge': {'priority': priority}}

priority += 0x1000

return config

self.stp.set\_config(generate\_fattree\_config(4))

def delete\_flow(self, datapath):

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

for dst in self.mac\_to\_port[datapath.id].keys():

match = parser.OFPMatch(eth\_dst=dst)

mod = parser.OFPFlowMod(

datapath, command=ofproto.OFPFC\_DELETE,

out\_port=ofproto.OFPP\_ANY, out\_group=ofproto.OFPG\_ANY,

priority=1, match=match, table\_id=1)

datapath.send\_msg(mod)

@set\_ev\_cls(stplib.EventPacketIn, MAIN\_DISPATCHER)

def \_packet\_in\_handler(self, ev):

msg = ev.msg

datapath = msg.datapath

ofproto = datapath.ofproto

parser = datapath.ofproto\_parser

in\_port = msg.match['in\_port']

pkt = packet.Packet(msg.data)

eth = pkt.get\_protocols(ethernet.ethernet)[0]

dst = eth.dst

src = eth.src

dpid = datapath.id

self.mac\_to\_port.setdefault(dpid, {})

self.logger.info("packet in %s %s %s %s", dpid, src, dst, in\_port)

# learn a mac address to avoid FLOOD next time.

self.mac\_to\_port[dpid][src] = in\_port

if dst in self.mac\_to\_port[dpid]:

out\_port = self.mac\_to\_port[dpid][dst]

else:

out\_port = ofproto.OFPP\_FLOOD

actions = [parser.OFPActionOutput(out\_port)]

# install a flow to avoid packet\_in next time

if out\_port != ofproto.OFPP\_FLOOD:

match = parser.OFPMatch(in\_port=in\_port, eth\_dst=dst)

self.add\_flow(datapath, 1, match, actions)

data = None

if msg.buffer\_id == ofproto.OFP\_NO\_BUFFER:

data = msg.data

out = parser.OFPPacketOut(datapath=datapath, buffer\_id=msg.buffer\_id,

in\_port=in\_port, actions=actions, data=data)

datapath.send\_msg(out)

@set\_ev\_cls(stplib.EventTopologyChange, MAIN\_DISPATCHER)

def \_topology\_change\_handler(self, ev):

dp = ev.dp

dpid\_str = dpid\_lib.dpid\_to\_str(dp.id)

msg = 'Receive topology change event. Flush MAC table.'

self.logger.debug("[dpid=%s] %s", dpid\_str, msg)

if dp.id in self.mac\_to\_port:

self.delete\_flow(dp)

del self.mac\_to\_port[dp.id]

@set\_ev\_cls(stplib.EventPortStateChange, MAIN\_DISPATCHER)

def \_port\_state\_change\_handler(self, ev):

dpid\_str = dpid\_lib.dpid\_to\_str(ev.dp.id)

of\_state = {stplib.PORT\_STATE\_DISABLE: 'DISABLE',

stplib.PORT\_STATE\_BLOCK: 'BLOCK',

stplib.PORT\_STATE\_LISTEN: 'LISTEN',

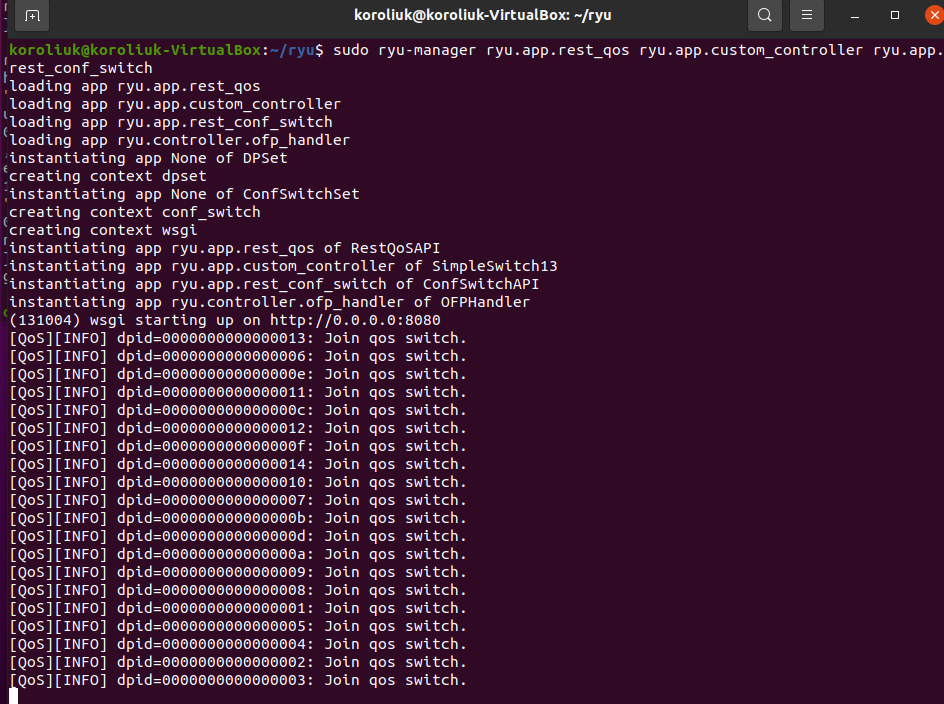
stplib.PORT\_STATE\_LEARN: 'LEARN',

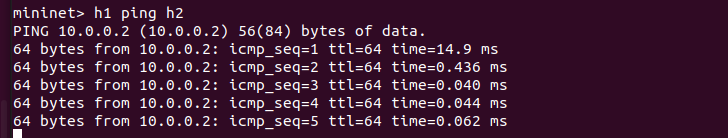
stplib.PORT\_STATE\_FORWARD: 'FORWARD'}

self.logger.debug("[dpid=%s][port=%d] state=%s",

dpid\_str, ev.port\_no, of\_state[ev.port\_state])

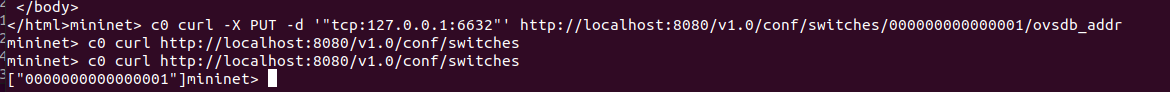
Розроблений контролер записує дані в OVSDB, коректно працює із топологіями, що мають цикли. Контролер містить функції, що обробляєть різні події по життєвому циклу взаємодії контролера із комутатором згідно до протоколу OpenFlow, зокрема додавання flow entry, видалення flow entry, дослідження шляхів та побудова дерева, обробка пакетів (у випадку коли ще немає встановленого шляху до якогось елемента топології), тощо. Також він проставляє dpid для комутаторів. Крім розробленого файлу, в команді запуску ryu менеджера будемо використовувати модулі *rest\_qos.py* та *rest\_conf\_switch.py*. Перший представляє REST обгортку, що дозволяє налаштовувати QoS для комутаторів, а другий реалізує доступ до статистики та оновлення певних властивостей комутаторів через REST запити, а також є залежністю *rest\_qos.py*. Підсумовуючи запуск контролера виглядає наступним чином:



Він працює коректно, можемо для перевірки виконати ping між хостами:  


Вкінці необхідно розробити компонент, який буде зчитувати в певному вигляді правила, що будуть визначати коли і яку пропускну здатність встановлювати. Крім цього, цей компонент має мати можливість симуляції руху часу або встановлення часу вручну, щоб перевірити коректність роботи.

Також необхідно додати доступ до OVSDB, наприклад для комутатора із dpid = 0000000000000001 це буде виглядати наступним чином:



Як бачимо із прикладу перший крок виконався успішно. Можна перевірити це за допомогою */v1.0/conf/switches*, який виводить список комутаторів доступних для взаємодії через контролер. У вікні контролера за допомогою *curl -X POST -d '{"type": "linux-htb", "queues": [ {"max\_rate" : "500000" }, { "min\_rate" : "800000" } ] } '* [*http://localhost:8080/qos/queue/0000000000000001*](http://localhost:8080/qos/queue/0000000000000001)можна налаштувати дві черги: одна із максимальної пропускною здатністю в 500000 біт, друга із мінімально пропускною здатністю в 800000 біт. Саме цей підхід буде використовуватись при реалізації останнього компонента. Його було реалізовано у вигляді flask додатку, що зчитує правила із yml файлу. Конфігураційний файл описує значення мінімальної та максимальної пропускної здатності за замовчуванням, а також список правил. Загалом його формат наступний:

networkBandwidth:

default:

minRate: 700000

maxRate: 800000

timeBasedRules:

- description: 'Rule 1 - Night time reduced bandwidth'

daysOfWeek: [Tuesday, Wednesday, Thursday, Friday]

timeRange:

start: '22:00'

end: '06:00'

bandwidth:

minRate: 100000

maxRate: 500000

- description: 'Rule 2 - Weekend increased bandwidth'

daysOfWeek: [Saturday, Sunday]

timeRange:

start: '08:00'

end: '23:59'

bandwidth:

maxRate: 200000

Кожне правило має свій опис, дні тижня коли застосовується, проміжок часу та значення пропускної здатності. Із мінімального та максимального значення лиш одне є обов’язковим для вказання. Якщо для якогось моменту часу доступними для застосування є декілька правил, то обирається те, що є ближчим до початку списку.

Код цього компонента має наступний вигляд:

import json

import subprocess

from flask import Flask, request, jsonify

from datetime import datetime, timedelta

import pytz

import yaml

from apscheduler.schedulers.background import BackgroundScheduler

app = Flask(\_\_name\_\_)

RYU\_CONTROLLER\_API = 'http://127.0.0.1:8080'

amount\_of\_switches = 20

is\_loading = True

time\_state = "REAL"

mock\_time = datetime.now(pytz.utc)

is\_moving = True

acceleration\_coefficient = 1

network\_bandwidth\_config = {}

bandwidth\_rules = []

def get\_switches\_ids(n):

return [f"{i:016x}" for i in range(1, n + 1)]

def load\_config():

global network\_bandwidth\_config, bandwidth\_rules

with open("net\_param\_config.yaml", 'r') as stream:

config = yaml.safe\_load(stream)

network\_bandwidth\_config = config.get('networkBandwidth', {}).get('default', {})

if not any(key in network\_bandwidth\_config for key in ['maxRate', 'minRate']):

raise ValueError("One of maxRate or minRate is mandatory in default settings")

for rule in config.get('networkBandwidth', {}).get('timeBasedRules', []):

if all(key in rule for key in ['daysOfWeek', 'timeRange']):

if not any(key in rule.get('bandwidth', {}) for key in ['maxRate', 'minRate']):

raise ValueError("One of maxRate or minRate is mandatory in each rule")

bandwidth\_rules.append(rule)

else:

raise ValueError("daysOfWeek and timeRange are mandatory in each rule")

def set\_switches(n):

global is\_loading

switches = get\_switches\_ids(n)

for switch\_id in switches:

connect\_ovsdb\_with\_switch(switch\_id)

set\_switch\_queue\_with\_default\_bandwidth(switch\_id)

is\_loading = False

def set\_switch\_queue\_with\_default\_bandwidth(switch\_id):

queue = {}

if 'minRate' in network\_bandwidth\_config:

queue['min\_rate'] = str(network\_bandwidth\_config['minRate'])

if 'maxRate' in network\_bandwidth\_config:

queue['max\_rate'] = str(network\_bandwidth\_config['maxRate'])

queue\_list = [queue]

set\_switch\_queue\_bandwidth(switch\_id, queue\_list)

def set\_switch\_queue\_bandwidth(switch\_id, queue\_list):

post\_data = {

"type": "linux-htb",

"queues": queue\_list

}

post\_base\_url = "http://localhost:8080/qos/queue/"

post\_url = f"{post\_base\_url}{switch\_id}"

post\_command = f"curl -X POST -d '{json.dumps(post\_data)}' {post\_url}"

subprocess.run(post\_command, shell=True)

def connect\_ovsdb\_with\_switch(switch\_id):

base\_url = "http://localhost:8080/v1.0/conf/switches/"

url = f"{base\_url}{switch\_id}/ovsdb\_addr"

put\_data = "'\"tcp:127.0.0.1:6632\"'"

command = f"curl -X PUT -d {put\_data} {url}"

subprocess.run(command, shell=True)

with app.app\_context():

global is\_loaded

load\_config()

set\_switches(amount\_of\_switches)

def get\_applicable\_rule(t):

current\_day = t.strftime('%A')

current\_hour\_minute = t.strftime('%H:%M')

for rule in bandwidth\_rules:

days\_of\_week = rule['daysOfWeek']

time\_range = rule['timeRange']

if current\_day in days\_of\_week:

start\_time = time\_range['start']

end\_time = time\_range['end']

if start\_time <= current\_hour\_minute <= end\_time or (

start\_time > end\_time and (current\_hour\_minute >= start\_time or current\_hour\_minute <= end\_time)):

return rule

return None

def apply\_bandwidth\_rule():

print("Job is running. Searching for rule...")

global is\_loading

if is\_loading:

return

current\_time = calculate\_time()

rule = get\_applicable\_rule(current\_time)

if rule is None:

for switch\_id in get\_switches\_ids(amount\_of\_switches):

set\_switch\_queue\_with\_default\_bandwidth(switch\_id)

else:

if 'bandwidth' in rule:

queue = {}

if 'minRate' in rule['bandwidth']:

queue['min\_rate'] = str(rule['bandwidth']['minRate'])

if 'maxRate' in rule['bandwidth']:

queue['max\_rate'] = str(rule['bandwidth']['maxRate'])

queue\_list = [queue]

for switch\_id in get\_switches\_ids(amount\_of\_switches):

set\_switch\_queue\_bandwidth(switch\_id, queue\_list)

scheduler = BackgroundScheduler()

scheduler.add\_job(func=apply\_bandwidth\_rule, trigger="interval", seconds=60)

scheduler.start()

def calculate\_time():

if time\_state == "REAL":

return datetime.now(pytz.utc)

else:

if is\_moving:

time\_diff = (datetime.now(pytz.utc) - mock\_time).total\_seconds()

adjusted\_time = mock\_time + timedelta(seconds=time\_diff \* acceleration\_coefficient)

return adjusted\_time

else:

return mock\_time

@app.route('/state', methods=['GET'])

def set\_state():

global time\_state

return jsonify({"state": time\_state}), 200

@app.route('/state', methods=['PUT'])

def update\_state():

global time\_state

data = request.json

if 'state' in data and data['state'] in ['REAL', 'MOCK']:

time\_state = data['state']

return jsonify({"message": "Time state updated successfully.", "state": time\_state}), 200

else:

return jsonify({"error": "Invalid state. Use 'REAL' or 'MOCK'."}), 400

@app.route('/mock-time', methods=['PUT'])

def set\_mock\_time():

global mock\_time, is\_moving, acceleration\_coefficient

data = request.json

try:

if 'dateTime' in data:

mock\_time = datetime.fromisoformat(data['dateTime']).astimezone(pytz.utc)

if 'isMoving' in data:

is\_moving = data['isMoving']

if 'accelerationCoefficient' in data:

acceleration\_coefficient = int(data['accelerationCoefficient'])

return jsonify({"message": "Mock time settings updated successfully."}), 200

except ValueError:

return jsonify({"error": "Invalid input."}), 400

@app.route('/time', methods=['GET'])

def get\_time():

return jsonify({"currentTime": calculate\_time().isoformat()}), 200

if \_\_name\_\_ == '\_\_main\_\_':

try:

app.run(host='0.0.0.0', port=9000)

except (KeyboardInterrupt, SystemExit):

scheduler.shutdown()

Цей скрипт є flask застосунком. Як тільки він запускається, зчитують правила зміни пропускної здатності із конфігураційного файлу, відбувається валідація даних. Далі, встановлюється зв'язок із OVSDB кожного комутатора.

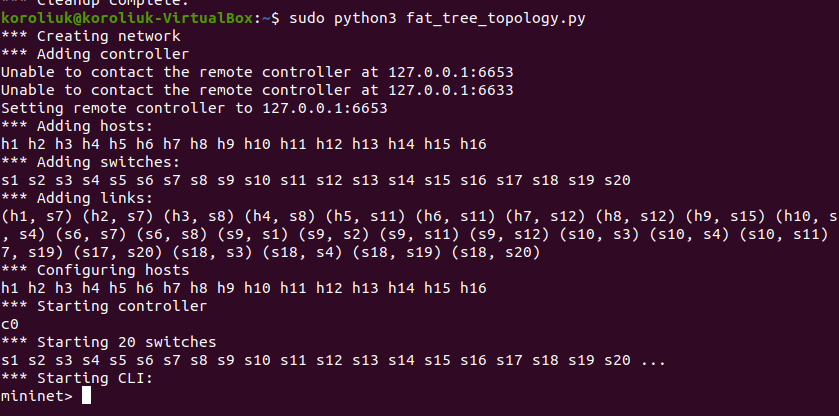
Сам механізм зміни пропускної здатності за потреби представлений джобою, яка тригериться за замовчуванням кожну 1 хвилину. Вона визначає чи є правило яке треба застосувати в певний момент часу. Якщо є правила немає, то поновлюються налаштування пропускної здатності за замовчуванням. Весь код тут спирається на вище описаний спосіб через REST ендпоінти контролера.

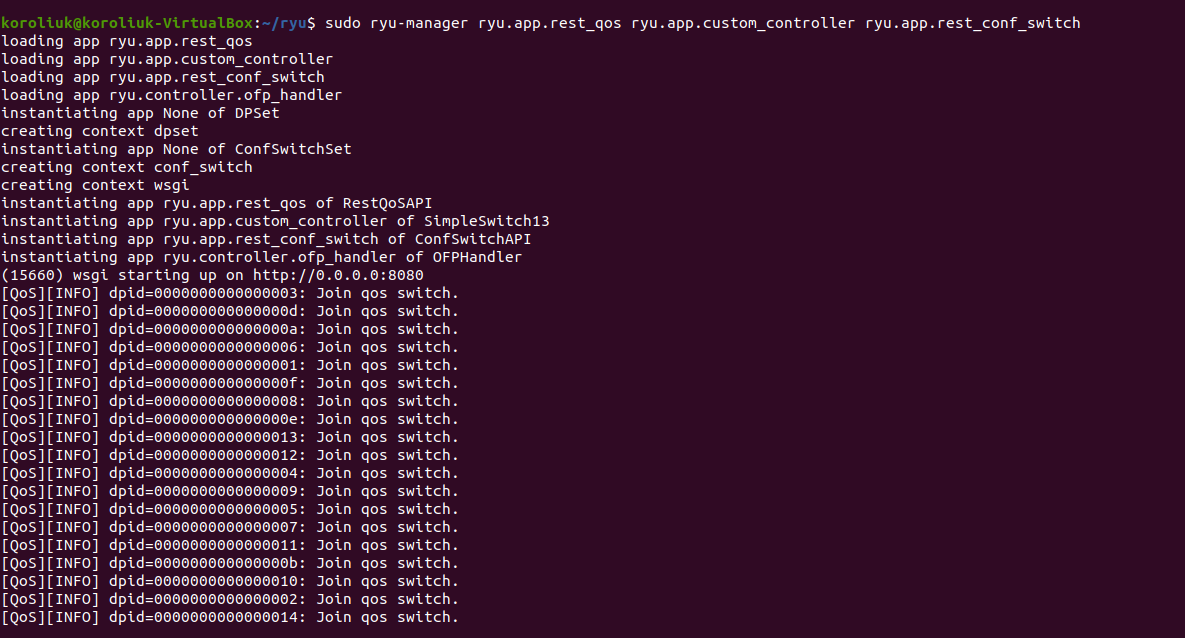
Крім цього, скрипт описує наступні ендоінти:

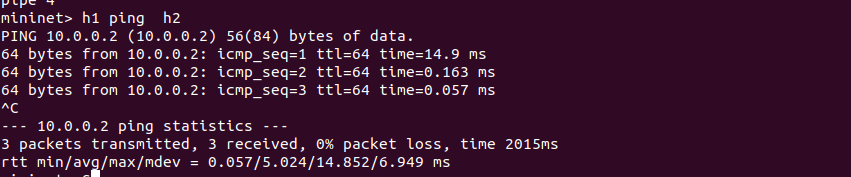
* GET /state - виводить стан часового функціоналу застосунку. Є два стани REAL та MOCK. При REAL стані джоба та /time ендпоінт беруть справжній час, при MOCK використовується штучно встановлене значення;
* PUT /state - оновлює стан;
* PUT /mock-time - встановлює моканий час, що буде використовуватись при стані MOCK. Тіло запиту містить базовий час, від якого починається відлік, прапорець чи необхідно починати відлік чи час заморожено, та коефіцієнт прискорення, щоб можна було прискорити рух часу в цілях тестування системи;
* GET /time - ендпоінт, що повертає час в даних момент, яким керується система.

**Приклад використання**

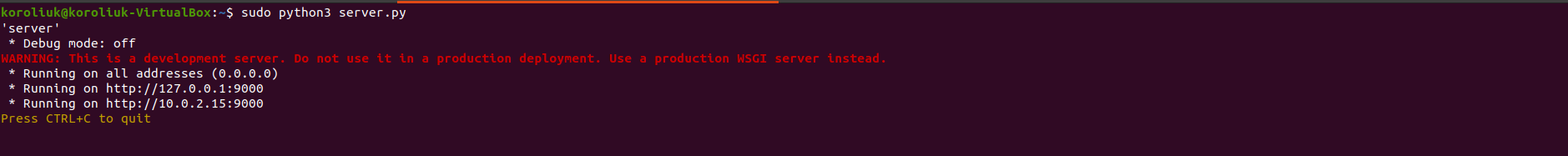
Запустимо скрипт топології та контролер:







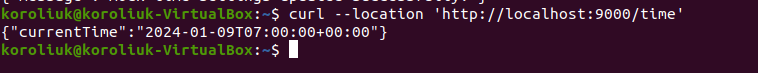
Далі, наш flask застосунок:



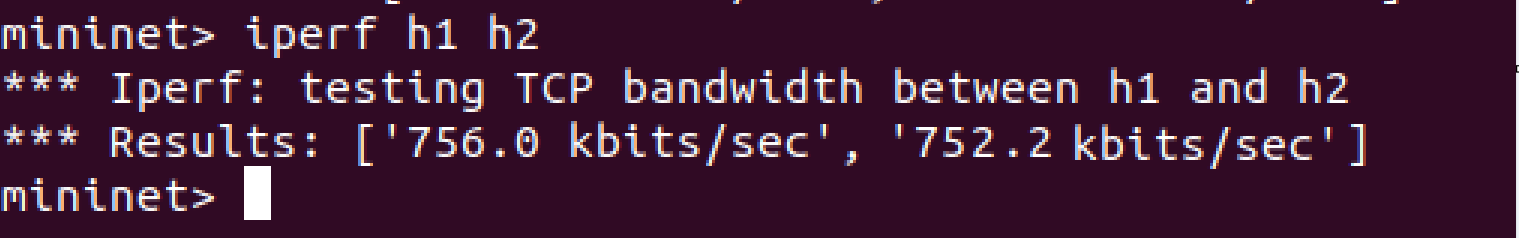
Конфігураційний файл містить такі правила:

| Дні | Проміжок часу дня | Мінімальне значення | Максимальне значення |
| --- | --- | --- | --- |
| сб - нд | 08:00 - 23:59 |  | 200000 |
| вт - пт | 00:00 - 06:00 та 22:00 - 24:00 | 100000 | 500000 |
| Весь час | Весь час | 700000 | 800000 |

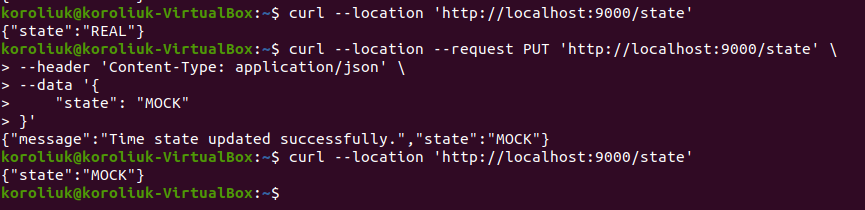
Перевіряємо за допомогою iperf пропускну здатність:



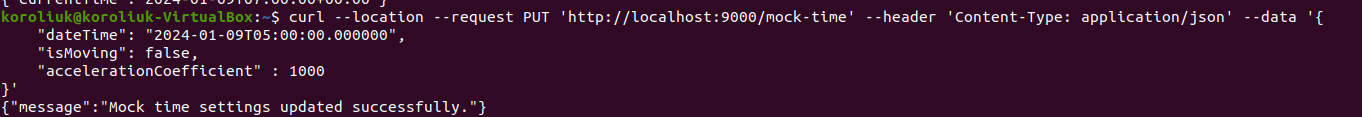
(9 - вівторок, тому в момент на скриншоті вище діють тільки дефолтні значення)



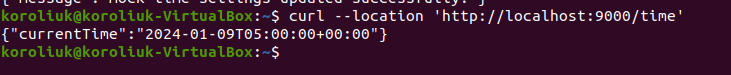
Бачимо, що вона очікувано в межах дефолтої із конфігураційного файлу. Далі, змінюємо стан системи на MOCK:



Встановлюємо моканий час:

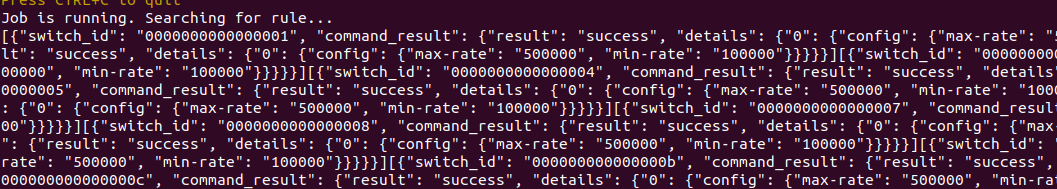


Перевіряємо викликаючи /time ендпоінт:

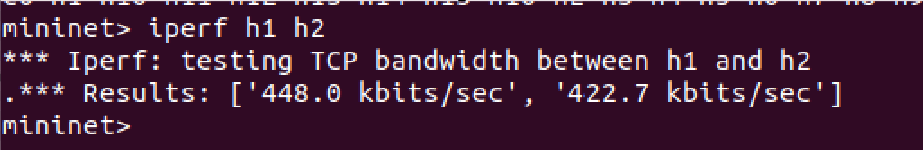


(9 - вівторок, але 5 година, тому застосовується друге правило)

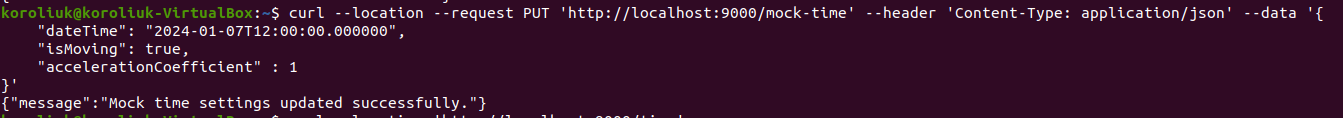
Чекаємо поки тригернеться джоба:



Перевіряємо пропускну здатність:



Бачимо, що вона в межах другого правила, як і очікувалось. Змінюємо час знову, тепер із рухом. Перевіряємо викликаючи /time:



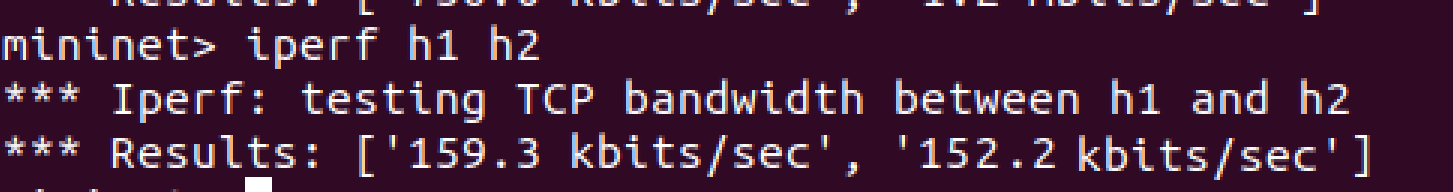


(7 - неділя, 12 година, тому застосовується перше правило)

Знову робимо запит на цей ендпоінт, щоб перевірити рух часу:



Перевіряємо пропускну здатність:



Як бачимо вона відповідає першому правилу.

**Висновок:** виконуючи лабораторну роботу, було створено застосунок, що дозволяє налаштовувати зміну пропускної здатності в мережі залежно від дня тижня та часу. Зокрема, було реалізовано скрипт, що створює топологію та запускає середовище mininet. Крім цього, було модифіковано контролер для запису в OVSDB використовуючи існуючі модулі RYU контролера та модифікуючи їх. Вкінці було розроблено компонент, який і відповідає за постійну перевірку того, чи є в поточний момент правило, яка треба застосувати. Цей компонент має також можливість мокати час та його рух для тестування. Він представлений у вигляді flask застосунку. Було наведено приклад роботи програми із скриншотами.