Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки
Кафедра обчислювальної техніки

ЗВІТ

з лабораторної роботи №5
з дисципліни ”Програмування комп’ютерних та віртуальних мереж”

Тема: Динамічне управління продуктивністю каналів передачі повідомлень у мережі SDN ЦОД (Data Center) на основі часових параметрів

Варіант №5

Виконав:
Студент 1 курсу, групи ІМ-51мн
Ковальов Олександр

Перевірив:
доцент, Долголенко Олександр Миколайович

Дата здачі: 20.11.2025

КИЇВ – 2025

Мета роботи. Практична реалізація логіки (на Python/SDN контролері) для забезпечення заданої продуктивності та оптимізації трафіку в каналах розробленої топології мережі ЦОД, використовуючи добовий час і день тижня як керуючі параметри.

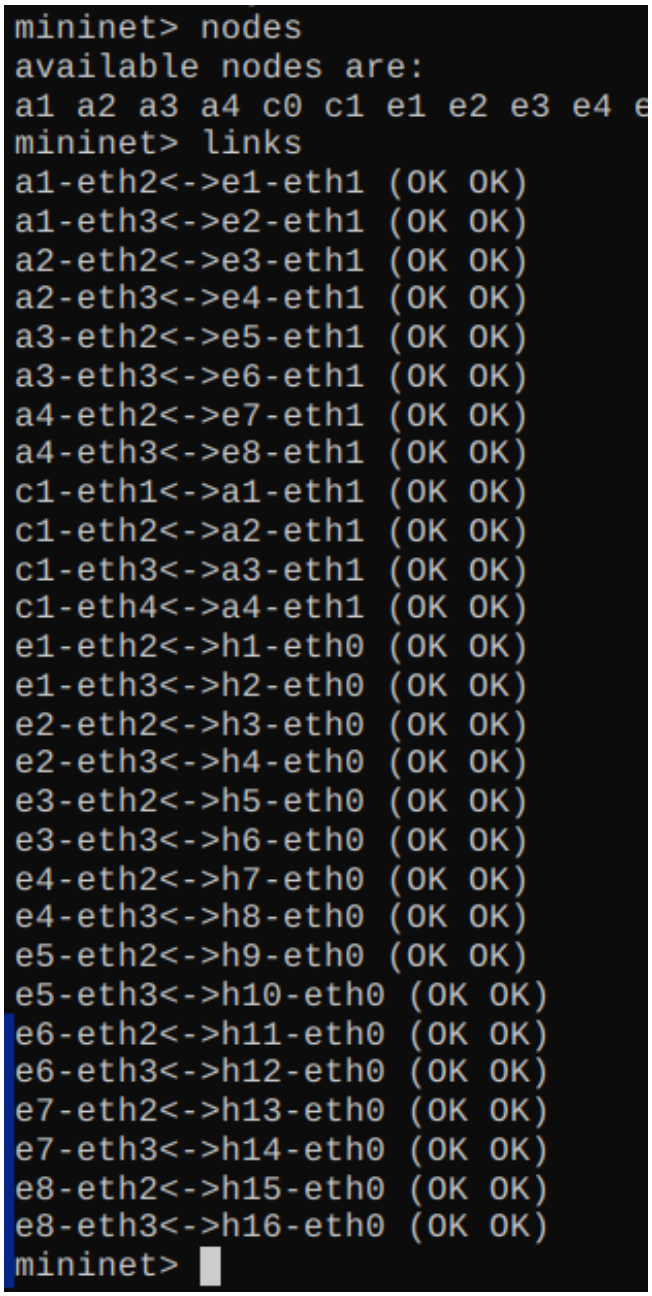
Завдання: Керування продуктивністю каналів передачі повідомлень в залежності від добового часу та дня тижня у SDN мережі центру обробки даних (Data Center). Підключити зовнішній контролер, створити скрипт `*.py`, що реалізує логіку продуктивності каналів передачі повідомлень в залежності від добового часу та дня тижня у SDN мережі Data Center з розробленою топологією.

Хід роботи.

Для виконання поставленого завдання було розроблено програмну емуляцію мережі центру обробки даних (Data Center) з використанням технології програмно-конфігурованих мереж (SDN). Теоретичною основою для побудови топології слугували матеріали статті ”A Scalable, Commodity Data Center Network Architecture” авторів Al-Fares, Loukissas та Vahdat. У цій праці описується архітектура Fat-Tree, яка дозволяє ефективно масштабувати пропускну здатність кластерів, використовуючи стандартні комутатори. Враховуючи особливості середовища емуляції Mininet та необхідність забезпечення стабільної маршрутизації без використання складних протоколів запобігання петель (на кшталт STP) у рамках лабораторної роботи, було вирішено реалізувати ієрархічну деревовидну структуру (Tree Topology), тобто модифікацію Fat-Tree, яка зберігає логіку рівнів Core, Aggregation та Edge, описану в джерелах, але гарантує відсутність фізичних циклів і штормів широкомовного трафіку.

Реалізація топології була виконана мовою Python у скрипті `dc_topo.py`. У коді було створено клас, що успадковує функціонал базового класу `Topo` бібліотеки `Mininet`. Структура мережі складається з трьох рівнів комутаторів. На верхньому рівні розташовано один кореневий комутатор (Core Switch), який забезпечує зв’язок між підмережами. До нього підключено чотири комутатори рівня агрегації (Aggregation Switches). Кожен комутатор агрегації, своєю чергою, з’єднаний з двома комутаторами рівня доступу (Edge Switches), до яких безпосередньо підключаються кінцеві хости. Загалом у топології задіяно 13 комутаторів та 16 хостів. Важливим аспектом реалізації стало використання методу `addSwitch` із зазначенням протоколу OpenFlow 1.3, оскільки застарілі версії протоколу не підтримують необхідний для виконання завдання функціонал `Meters` (лічильників трафіку). Для забезпечення миттєвої зв’язності між хостами

та уникнення помилок ARP resolution (проблема ”Destination Host Unreachable”) у скрипті запуску було активовано функцію `net.staticArp()`, яка автоматично заповнює ARP-таблиці хостів, емулюючи роботу служби каталогів у реальному ЦОД.



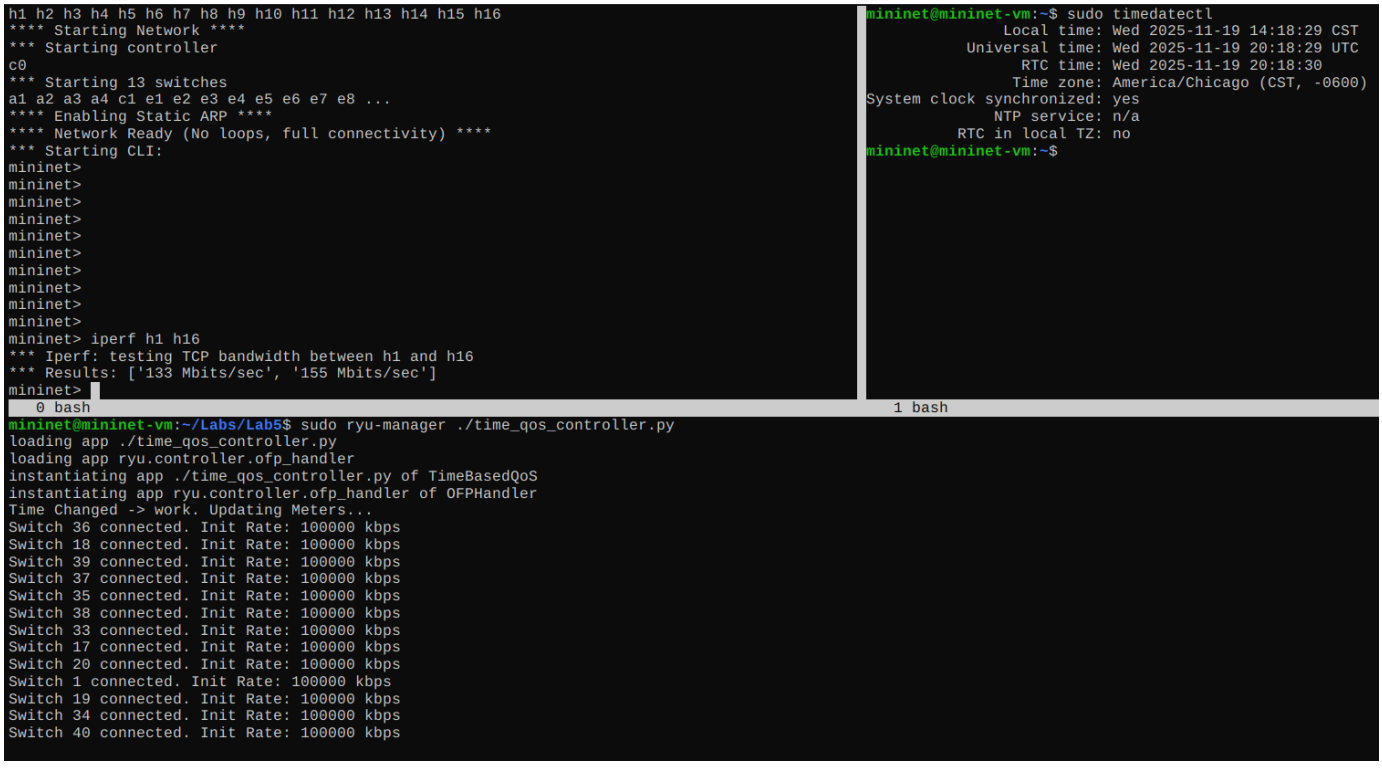
Наступним етапом стала розробка логіки керування продуктивністю каналів. Для цього було створено скрипт зовнішнього контролера `time_qos_-controller.py` на базі фреймворку Ryu. Головна мета контролера полягала в динамічній зміні пропускної здатності каналів залежно від поточного часу та дня тижня. Алгоритм роботи контролера базується на використанні окремого потоку виконання, який з інтервалом у кілька секунд перевіряє системний час. Логіка перевірки визначає, чи є поточний момент ”робочим часом” (понеділок - п’ятниця, з 09:00 до 18:00). Якщо умова виконується, мережа переходить у

режим високої продуктивності, інакше – у режим обмеженої швидкості.

Для технічної реалізації обмеження швидкості, замість повного відкидання пакетів, було використано механізм OpenFlow Meters. У коді контролера визначено функцію, яка надсилає повідомлення `OFPMeterMod` на комутатори. Ця функція налаштовує лічильник (`Meter`) з певним ідентифікатором та типом смуги `OFPMeterBandDrop`. Це означає, що пакети, частота надходження яких перевищує встановлений поріг (`rate`), будуть відкидатися комутатором, що для кінцевого користувача виглядає як зниження швидкості передачі даних, а не розрив з'єднання. У скрипті задано дві константи швидкості: висока для робочих годин (100 Мбіт/с) та низька для неробочого часу (1 Мбіт/с).

Тестування механізму керування швидкістю за допомогою утиліти `iperf`. Оскільки тестування проводилося у вечірній час (поза межами робочого інтервалу), то час було змінено вручну за допомогою зміну часових зон `timedatectl`. У вечірні години контролер автоматично перейшов у режим обмеження. Під час вимірювання пропускної здатності між хостами `h1` та `h16`, які знаходяться у різних підмережах, утиліта `iperf` зафіксувала швидкість передачі даних на рівні близько 1 Мбіт/с. Це підтверджує, що механізм QoS спрацював коректно: швидкість була "обрізана" до заданого ліміту, але з'єднання не було розірвано. Логи контролера також відображали зміну контексту часу та відправку команд на оновлення параметрів лічильників на комутаторах. Те ж саме відбулося в "робочі" години, але зі швидкістю 130 Мбіт/с.

Тестування в робочі години:



В неробочі:

```
Connecting to remote controller at 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
*** Adding switches:
a1 a2 a3 a4 c1 e1 e2 e3 e4 e5 e6 e7 e8
*** Adding links:
(a1, e1) (a1, e2) (a2, e3) (a2, e4) (a3, e5) (a3, e6) (a4, e7) (a4, e8) (c1, a1) (c1, a2) (c1
, a3) (c1, a4) (e1, h1) (e1, h2) (e2, h3) (e2, h4) (e3, h5) (e3, h6) (e4, h7) (e4, h8) (e5, h
9) (e5, h10) (e6, h11) (e6, h12) (e7, h13) (e7, h14) (e8, h15) (e8, h16)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
**** Starting Network ****
*** Starting controller
c0
*** Starting 13 switches
a1 a2 a3 a4 c1 e1 e2 e3 e4 e5 e6 e7 e8 ...
**** Enabling Static ARP ****
**** Network Ready (No loops, full connectivity) ****
*** Starting CLI:
mininet> iperf h1 h16
*** Iperf: testing TCP bandwidth between h1 and h16
.*** Results: ['1.08 Mbits/sec', '2.31 Mbits/sec']
mininet>
```

```
mininet@mininet-vm:~$ sudo timedatectl
               Local time: Wed 2025-11-19 22:22:26 EET
           Universal time: Wed 2025-11-19 20:22:26 UTC
                 RTC time: Wed 2025-11-19 20:22:28
                Time zone: Europe/Kyiv (EET, +0200)
System clock synchronized: yes
              NTP service: n/a
          RTC in local TZ: no
mininet@mininet-vm:~$
```

0 bash 1 bash

```
mininet@mininet-vm:~/Labs/Lab5$ sudo ryu-manager ./time_qos_controller.py
loading app ./time_qos_controller.py
loading app ryu.controller.ofp_handler
instantiating app ./time_qos_controller.py of TimeBasedQoS
instantiating app ryu.controller.ofp_handler of OFPHandler
Time Changed -> off. Updating Meters...
Switch 36 connected. Init Rate: 1000 kbps
Switch 39 connected. Init Rate: 1000 kbps
Switch 18 connected. Init Rate: 1000 kbps
Switch 37 connected. Init Rate: 1000 kbps
Switch 35 connected. Init Rate: 1000 kbps
Switch 40 connected. Init Rate: 1000 kbps
Switch 1 connected. Init Rate: 1000 kbps
Switch 17 connected. Init Rate: 1000 kbps
Switch 38 connected. Init Rate: 1000 kbps
Switch 19 connected. Init Rate: 1000 kbps
Switch 20 connected. Init Rate: 1000 kbps
Switch 33 connected. Init Rate: 1000 kbps
Switch 34 connected. Init Rate: 1000 kbps
```

Окрім керування QoS, контролер виконує функцію L2-комутації (Learning Switch). Обробник події `EventOFPPacketIn` аналізує вхідні пакети, запам'ятовує MAC-адреси відправників та відповідні порти, заповнюючи таблицю комутації. При встановленні потоків (Flow Entries) для передачі даних контролер додає інструкцію `OFPInstructionMeter`, яка прив'язує трафік до раніше створеного лічильника. Таким чином, будь-який трафік між хостами проходить через механізм обмеження швидкості.

Перевірка працездатності системи проводилася у кілька етапів. Спочатку було запущено контролер Ryu, а потім – скрипт топології Mininet. Після ініціалізації мережі було виконано команду `pingall` у консолі Mininet. Результати показали 100% успішність проходження пакетів, що підтвердило коректність побудови топології, відсутність петель та правильну роботу механізму Static ARP.

```
mininet@mininet-vm:~/Labs/Lab5$ sudo ryu-manager ./time_qos_controller.py
loading app ./time_qos_controller.py
loading app ryu.controller.ofp_handler
instantiating app ./time_qos_controller.py of TimeBasedQoS
instantiating app ryu.controller.ofp_handler of OFPHandler
Time Changed -> off. Updating Meters...
Switch 36 connected. Init Rate: 1000 kbps
Switch 39 connected. Init Rate: 1000 kbps
Switch 18 connected. Init Rate: 1000 kbps
Switch 37 connected. Init Rate: 1000 kbps
Switch 35 connected. Init Rate: 1000 kbps
Switch 38 connected. Init Rate: 1000 kbps
Switch 17 connected. Init Rate: 1000 kbps
Switch 34 connected. Init Rate: 1000 kbps
Switch 40 connected. Init Rate: 1000 kbps
Switch 1 connected. Init Rate: 1000 kbps
Switch 19 connected. Init Rate: 1000 kbps
Switch 33 connected. Init Rate: 1000 kbps
Switch 20 connected. Init Rate: 1000 kbps
```

```
--- 10.0.0.16 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3013ms
rtt min/avg/max/mdev = 0.127/7.091/18.189/7.381 ms
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10 h11 h12 h13 h14 h15 h16
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10 h11 h12 h13 h14 h15 h16
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10 h11 h12 h13 h14 h15 h16
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10 h11 h12 h13 h14 h15 h16
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h11 h12 h13 h14 h15 h16
h11 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h12 h13 h14 h15 h16
h12 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h13 h14 h15 h16
h13 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h14 h15 h16
h14 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h15 h16
h15 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h16
h16 -> h1 h2 h3 h4 h5 h6 h7 h8 h9 h10 h11 h12 h13 h14 h15
*** Results: 0% dropped (240/240 received)
mininet>
```

0 bash 1 bash

Висновок. У ході виконання лабораторної роботи було успішно спроектовано та реалізовано програмно-конфігуровану мережу центру обробки даних. Спираючись на теоретичні засади архітектури Data Center Networks, зокрема концепцію ієрархічної побудови мережі, було створено топологію, що забезпечує повну зв'язність хостів. Використання зовнішнього контролера на базі Ryu дозволило реалізувати інтелектуальне керування трафіком.

Основне завдання роботи – керування продуктивністю каналів залежно від часу доби – було виконано за допомогою протоколу OpenFlow 1.3 та механізму Meters. Це дозволило динамічно змінювати параметри пропускної здатності мережі без необхідності перезавантаження обладнання чи розриву активних з'єднань. Експериментальна перевірка підтвердила, що у визначені "неробочі" години швидкість передачі даних обмежується до заданого мінімуму, тоді як у "робочий" час надається повна смуга пропускання. Такий підхід дозволяє гнучко керувати ресурсами мережі ЦОД, адаптуючись до графіку навантаження.

Лістинг.

dc_topo.py

```
from mininet.topo import Topo
from mininet.net import Mininet
from mininet.node import RemoteController, OVSKernelSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink

class SimpleDCTopo(Topo):
  """
  Simplified Data Center Topology (Tree based).
  No physical loops, so STP is NOT required.
  Structure: Core -> Aggregation -> Edge -> Hosts.
  """
  def build(self):
    # --- Create Switches ---
    # 1 Core Switch
    c1 = self.addSwitch('c1', dpid='0000000000000001',
              protocols='OpenFlow13')

    # 4 Aggregation Switches
    aggrs = []
    for i in range(4):
      sw = self.addSwitch(f'a{i+1}', dpid=f'000000000000001{i+1}',
                protocols='OpenFlow13')
      aggrs.append(sw)

    # 8 Edge Switches
    edges = []
    for i in range(8):
      sw = self.addSwitch(f'e{i+1}', dpid=f'000000000000002{i+1}',
                protocols='OpenFlow13')
      edges.append(sw)

    # --- Create Links (No Loops) ---

    # 1. Connect Core to all Aggregation switches
    for a_sw in aggrs:
      self.addLink(c1, a_sw)

    # 2. Connect Aggregation to Edge (Simple Tree structure)
    # a1 -> e1, e2; a2 -> e3, e4 ...
    edge_idx = 0
    for a_sw in aggrs:
      self.addLink(a_sw, edges[edge_idx])
      self.addLink(a_sw, edges[edge_idx+1])
      edge_idx += 2

```

```
        # 3. Connect Hosts to Edge (2 Hosts per Edge)
        # e1 -> h1, h2; e2 -> h3, h4 ...
        host_count = 1
        for e_sw in edges:
          h1 = self.addHost(f'h{host_count}')
          h2 = self.addHost(f'h{host_count+1}')
          self.addLink(e_sw, h1)
          self.addLink(e_sw, h2)
          host_count += 2

    def run_topology():
      topo = SimpleDCTopo()
      net = Mininet(topo=topo,
            controller=RemoteController,
            switch=OVSKernelSwitch,
            link=TCLink,
            autoSetMacs=True)

      info("**** Starting Network ****\n")
      net.start()

      # Static ARP is helpful to speed up discovery
      info("**** Enabling Static ARP ****\n")
      net.staticArp()

      info("**** Network Ready (No loops, full connectivity) ****\n")
      CLI(net)

      info("**** Stopping Network ****\n")
      net.stop()

    if __name__ == '__main__':
      setLogLevel('info')
      run_topology()
```

time_qos_controller.py

```
    import datetime
    from ryu.base import app_manager
    from ryu.controller import ofp_event
    from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
    from ryu.controller.handler import set_ev_cls
    from ryu.ofproto import ofproto_v1_3
    from ryu.lib import hub
    from ryu.lib.packet import packet, ethernet, ether_types

    class TimeBasedQoS(app_manager.RyuApp):
      OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
```

```
    def __init__(self, *args, **kwargs):
        super(TimeBasedQoS, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.datapaths = {}
        self.meter_id = 1

        # QoS Config (kbps)
        self.BW_HIGH = 100000  # 100 Mbps
        self.BW_LOW = 1000     # 1 Mbps
        self.current_mode = None

        self.monitor_thread = hub.spawn(self._time_monitor)

    @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

        self.datapaths[datapath.id] = datapath

        # Default flow: Send to Controller
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions, use_meter=False)

        # Init Meter
        is_work_hours = self._check_work_hours()
        rate = self.BW_HIGH if is_work_hours else self.BW_LOW
        self.current_mode = 'work' if is_work_hours else 'off'

        self.logger.info(f"Switch {datapath.id} connected. Init Rate: {rate} kbps")
        self.send_meter_mod(datapath, ofproto.OFPMC_ADD, rate)

    @set_ev_cls(ofp_event.EventOFPPacketIn, 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']

        try:
            pkt = packet.Packet(msg.data)
            eth = pkt.get_protocols(ethernet.ethernet)[0]

            if eth.ethertype in [ether_types.ETH_TYPE_LLDP, ether_types.ETH_TYPE_IPV6]:
                return
```

```
62
63          dst = eth.dst
64          src = eth.src
65          dpid = datapath.id
66
67          self.mac_to_port.setdefault(dpid, {})
68          self.mac_to_port[dpid][src] = in_port
69
70          if dst in self.mac_to_port[dpid]:
71            out_port = self.mac_to_port[dpid][dst]
72          else:
73            out_port = ofproto.OFPP_FLOOD
74
75          actions = [parser.OFPActionOutput(out_port)]
76
77          # Install Flow with QoS Meter
78          if out_port != ofproto.OFPP_FLOOD:
79            match = parser.OFPMatch(in_port=in_port, eth_dst=dst, eth_src=src)
80            if msg.buffer_id != ofproto.OFP_NO_BUFFER:
81              self.add_flow(datapath, 1, match, actions, msg.buffer_id, use_meter=True)
82            else:
83              self.add_flow(datapath, 1, match, actions, use_meter=True)
84
85          data = None
86          if msg.buffer_id == ofproto.OFP_NO_BUFFER:
87            data = msg.data
88          out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
89                      in_port=in_port, actions=actions, data=data)
90          datapath.send_msg(out)
91        except:
92          return
93
94      def add_flow(self, datapath, priority, match, actions, buffer_id=None, use_meter=True):
95        ofproto = datapath.ofproto
96        parser = datapath.ofproto_parser
97        inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
98
99        if priority > 0 and use_meter:
100         inst.append(parser.OFPInstructionMeter(self.meter_id))
101
102       if buffer_id:
103         mod = parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
104                     priority=priority, match=match, instructions=inst)
105       else:
106         mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
107                     match=match, instructions=inst)
108       datapath.send_msg(mod)
109
110     def send_meter_mod(self, datapath, command, rate_kbps):
111       ofproto = datapath.ofproto
```

```
        parser = datapath.ofproto_parser
        bands = [parser.OFPMeterBandDrop(rate=rate_kbps, burst_size=10)]
        req = parser.OFPMeterMod(datapath=datapath, command=command,
                    flags=ofproto.OFPMF_KBPS, meter_id=self.meter_id, bands=bands)
        datapath.send_msg(req)

    def _check_work_hours(self):
        now = datetime.datetime.now()
        is_weekday = 0 <= now.weekday() <= 4
        is_working_time = 9 <= now.hour < 18

        return is_weekday and is_working_time

    def _time_monitor(self):
        while True:
            is_work_hours = self._check_work_hours()
            new_mode = 'work' if is_work_hours else 'off'

            if new_mode != self.current_mode:
                self.logger.info(f"Time Changed -> {new_mode}. Updating Meters...")
                self.current_mode = new_mode
                new_rate = self.BW_HIGH if is_work_hours else self.BW_LOW
                for dp in self.datapaths.values():
                    self.send_meter_mod(dp, dp.ofproto.OFPMC_MODIFY, new_rate)
            hub.sleep(10)

```