Topology and Service Slicing

Esposito Marcello M63001768, Fiumara Fabio M63001795

July 22, 2025

Abstract

This document presents the implementation of topology and service slicing in an SDN environment using Mininet, Ryu, and OpenFlow switches, to enforce traffic isolation and prioritize specific services, such as video streaming. Additionally, it encompasses the development of various extensions to enhance traffic classification and visualization.

Chapter 1

Project Overview and Topology Definition

1.1 Project Overview

In this project, we address two fundamental issues related to Quality of Service (QoS) in modern networks: Topology Slicing and Service Slicing.

- **Topology Slicing:** The first objective is to **restrict communication between hosts** according to a well-defined **topological scheme**, divided into slices, i.e., logical portions of the network. To this end, it is necessary to configure the SDN controller **Ryu** to implement appropriate routing rules (flow rules), binding traffic to follow predetermined paths. In this scenario, we assume the role of network administrators and determine which hosts can communicate with each other.

- **Service Slicing:** The second objective is traffic prioritization. In particular, **a higher priority is required for a specific type of traffic**, such as video traffic. Again, the configuration of the **Ryu** controller is central: it is necessary to identify this traffic and apply rules that allow the network to treat it with priority over other types of traffic.

In the following chapters, detailed implementations for both solutions will be provided along with the presentation of the extensions and their implementations.

1.2 Topology Definition

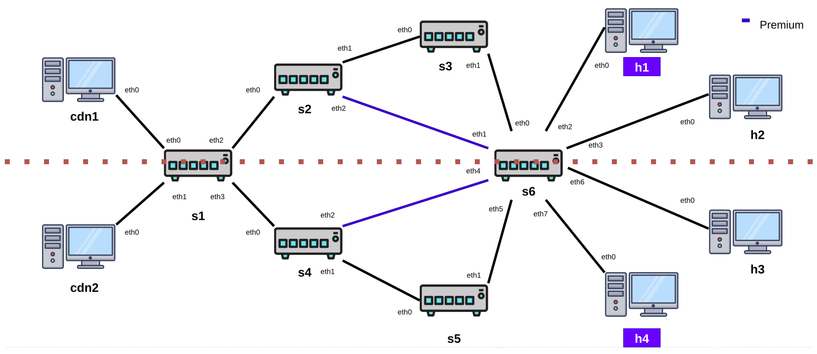
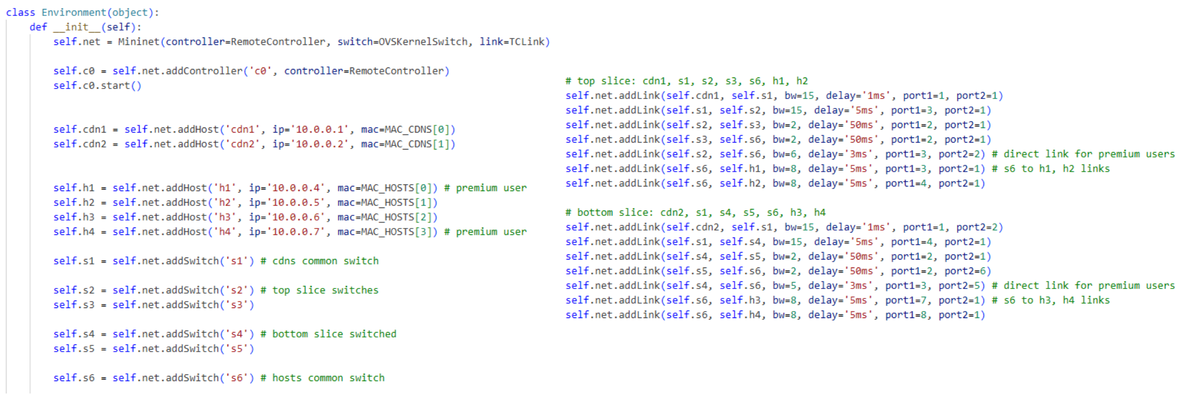


Figure 1.1: network topology

In this network, it has been supposed that there are two different Content Delivery Networks (cdn1, cdn2) that provide content to many hosts (h1, h2, h3, h4), splitting the served hosts with topology slicing, so that cdn1 can communicate only to h1 and h2, while cdn2 can communicate only to h3 and h4.
Furthermore, we imagine that in a network there could be both **premium users and default users**, who will receive different QOS. In our case, for the premium users, h1 and h4, we have added a dedicated premium link, from switch 2 / 4 to switch 6, which will be used only for video streaming.

1.2.1 Implementation

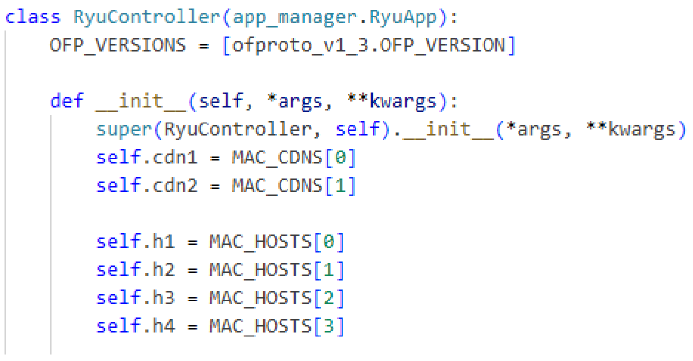
We started defining the Mininet configuration, proceeding to add the network's switches, links, and hosts with static MAC addresses and IPv4 configurations.
Notable are the **premium links** (from switch 2 / 4 to 6) that have a greater bandwidth and a much lower delay.



```
class Environment(object):
    def __init__(self):
        self.net = Mininet(controller=RemoteController, switch=OVSKernelSwitch, link=TCLink)

        self.c0 = self.net.addController('c0', controller=RemoteController)
        self.c0.start()


        self.cdn1 = self.net.addHost('cdn1', ip='10.0.0.1', mac=MAC_CDNS[0])
        self.cdn2 = self.net.addHost('cdn2', ip='10.0.0.2', mac=MAC_CDNS[1])


        self.h1 = self.net.addHost('h1', ip='10.0.0.4', mac=MAC_HOSTS[0]) # premium user
        self.h2 = self.net.addHost('h2', ip='10.0.0.5', mac=MAC_HOSTS[1])
        self.h3 = self.net.addHost('h3', ip='10.0.0.6', mac=MAC_HOSTS[2])
        self.h4 = self.net.addHost('h4', ip='10.0.0.7', mac=MAC_HOSTS[3]) # premium user

        self.s1 = self.net.addSwitch('s1') # cdns common switch

        self.s2 = self.net.addSwitch('s2') # top slice switches
        self.s3 = self.net.addSwitch('s3')

        self.s4 = self.net.addSwitch('s4') # bottom slice switched
        self.s5 = self.net.addSwitch('s5')

        self.s6 = self.net.addSwitch('s6') # hosts common switch
```

```
        # top slice: cdn1, s1, s2, s3, s6, h1, h2
        self.net.addLink(self.cdn1, self.s1, bw=15, delay='1ms', port1=1, port2=1)
        self.net.addLink(self.s1, self.s2, bw=15, delay='5ms', port1=3, port2=1)
        self.net.addLink(self.s2, self.s3, bw=2, delay='50ms', port1=2, port2=1)
        self.net.addLink(self.s3, self.s6, bw=2, delay='50ms', port1=2, port2=1)
        self.net.addLink(self.s2, self.s6, bw=6, delay='3ms', port1=3, port2=2) # direct link for premium users
        self.net.addLink(self.s6, self.h1, bw=8, delay='5ms', port1=3, port2=1) # s6 to h1, h2 links
        self.net.addLink(self.s6, self.h2, bw=8, delay='5ms', port1=4, port2=1)

        # bottom slice: cdn2, s1, s4, s5, s6, h3, h4
        self.net.addLink(self.cdn2, self.s1, bw=15, delay='1ms', port1=1, port2=2)
        self.net.addLink(self.s1, self.s4, bw=15, delay='5ms', port1=4, port2=1)
        self.net.addLink(self.s4, self.s5, bw=2, delay='50ms', port1=2, port2=1)
        self.net.addLink(self.s5, self.s6, bw=2, delay='50ms', port1=2, port2=6)
        self.net.addLink(self.s4, self.s6, bw=5, delay='3ms', port1=3, port2=5) # direct link for premium users
        self.net.addLink(self.s6, self.h3, bw=8, delay='5ms', port1=7, port2=1) # s6 to h3, h4 links
        self.net.addLink(self.s6, self.h4, bw=8, delay='5ms', port1=8, port2=1)
```

Figure 1.2: network topology definition



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

    def __init__(self, *args, **kwargs):
        super(RyuController, self).__init__(*args, **kwargs)
        self.cdn1 = MAC_CDNS[0]
        self.cdn2 = MAC_CDNS[1]

        self.h1 = MAC_HOSTS[0]
        self.h2 = MAC_HOSTS[1]
        self.h3 = MAC_HOSTS[2]
        self.h4 = MAC_HOSTS[3]
```

Figure 1.3: hosts definition for controller

Chapter 2

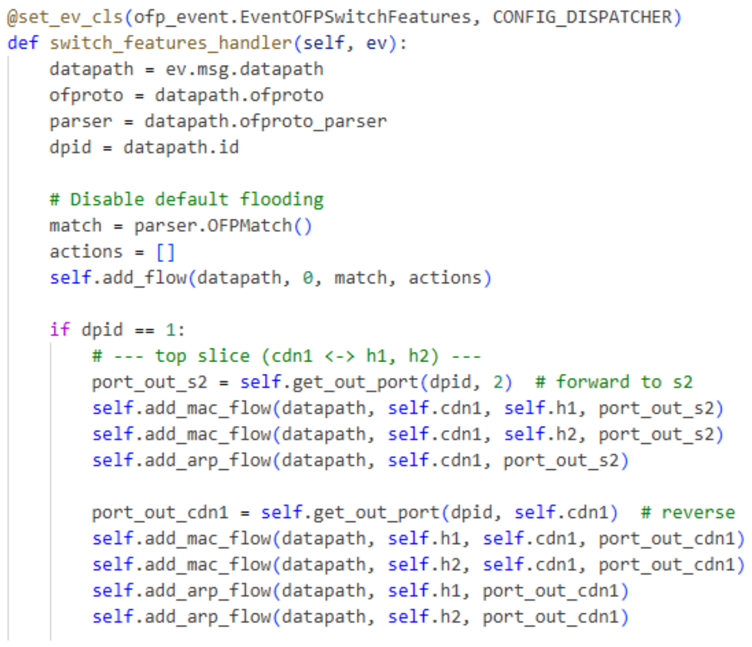
Topology Slicing

Nowadays, it is common to **share network resources across isolated traffic fluxes** from different source and destination hosts, and this is what we achieve with topology slicing, where we virtually slice our network into two different sections:

- **top slice**: communications between cdn1, h1 and h2 only trough switch 1, 2, 3, 6
- **bottom slice**: communications between cdn2, h3 and h4 only trough switch 1, 4, 5, 6

2.1 Implementation

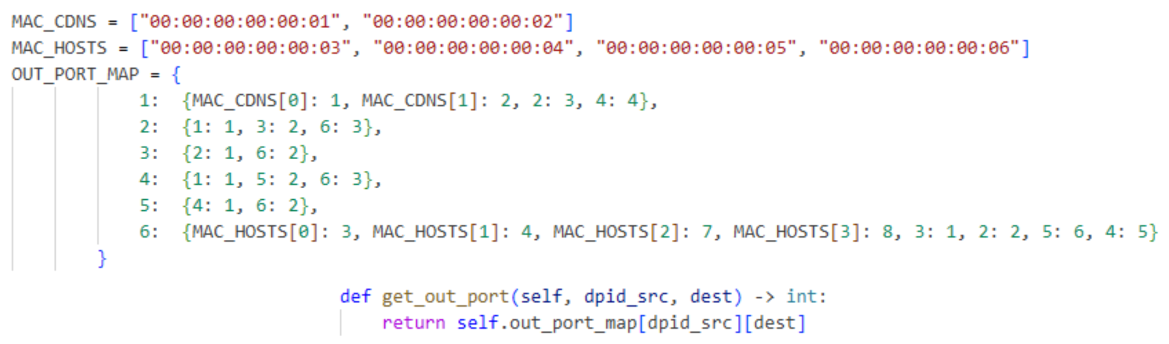
Slicing is managed by the Ryu controller, defining flow rules using a **proactive approach** to reduce network setup delay and controller work.
To define flow rules proactively, every **new switch activation event** is intercepted by a specific handler that will install the flow rules according to allowed slice paths, e. the switch one allows forwarding only between cdn1 and h1/h2 in both directions (and the same applies for cdn2 and h3/h4).



```
@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
    dpid = datapath.id

    # Disable default flooding
    match = parser.OFPMatch()
    actions = []
    self.add_flow(datapath, 0, match, actions)

    if dpid == 1:
        # --- top slice (cdn1 <-> h1, h2) ---
        port_out_s2 = self.get_out_port(dpid, 2)  # forward to s2
        self.add_mac_flow(datapath, self.cdn1, self.h1, port_out_s2)
        self.add_mac_flow(datapath, self.cdn1, self.h2, port_out_s2)
        self.add_arp_flow(datapath, self.cdn1, port_out_s2)

        port_out_cdn1 = self.get_out_port(dpid, self.cdn1)  # reverse
        self.add_mac_flow(datapath, self.h1, self.cdn1, port_out_cdn1)
        self.add_mac_flow(datapath, self.h2, self.cdn1, port_out_cdn1)
        self.add_arp_flow(datapath, self.h1, port_out_cdn1)
        self.add_arp_flow(datapath, self.h2, port_out_cdn1)
```

Figure 2.1: proactive flow rules example

As shown in fig.2.1, to improve readability, maintainability, etc., many **helper methods** have been made to better modularize the code.
First of all, due to static configuration, when adding a flow rule we must also tell the output port as action, all the output ports to use have been mapped in *OUT_PORT_MAP* variable defined in common.py (can be seen as a global constraints holder) that can be accessed trough the *get_out_port* method, so that is easier to change mapping approach to use more sophisticated structures in future implementations.

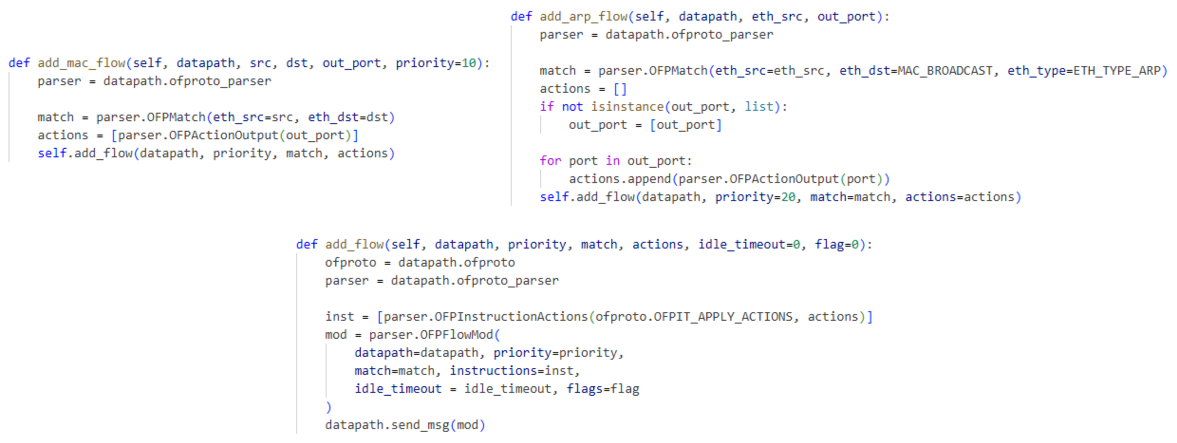


```
MAC_CDNS = ["00:00:00:00:00:01", "00:00:00:00:00:02"]
MAC_HOSTS = ["00:00:00:00:00:03", "00:00:00:00:00:04", "00:00:00:00:00:05", "00:00:00:00:00:06"]
OUT_PORT_MAP = {
            1:  {MAC_CDNS[0]: 1, MAC_CDNS[1]: 2, 2: 3, 4: 4},
            2:  {1: 1, 3: 2, 6: 3},
            3:  {2: 1, 6: 2},
            4:  {1: 1, 5: 2, 6: 3},
            5:  {4: 1, 6: 2},
            6:  {MAC_HOSTS[0]: 3, MAC_HOSTS[1]: 4, MAC_HOSTS[2]: 7, MAC_HOSTS[3]: 8, 3: 1, 2: 2, 5: 6, 4: 5}
        }
```

```
def get_out_port(self, dpid_src, dest) -> int:
    return self.out_port_map[dpid_src][dest]
```

Figure 2.2: out port map

Other helpful methods for topology slicing are *add_mac_flow* and *add_arp_flow*, which install flow rules regarding general packet forwarding. The first one regards every transmission matching on source and destination MAC address. In contrast, ARP rules match on the Ethernet type of the packet and MAC address broadcast destination, so that **ARP requests are not allowed across slices and avoid loops**.



```
def add_mac_flow(self, datapath, src, dst, out_port, priority=10):
    parser = datapath.ofproto_parser

    match = parser.OFPMatch(eth_src=src, eth_dst=dst)
    actions = [parser.OFPActionOutput(out_port)]
    self.add_flow(datapath, priority, match, actions)
```

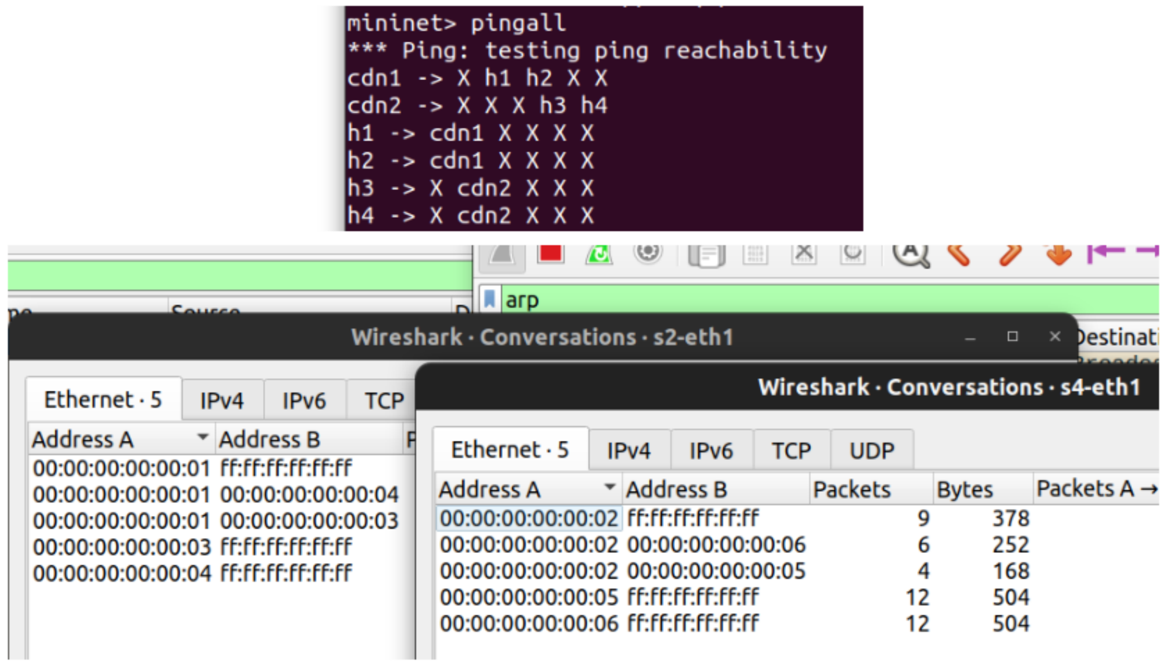
```
def add_arp_flow(self, datapath, eth_src, out_port):
    parser = datapath.ofproto_parser

    match = parser.OFPMatch(eth_src=eth_src, eth_dst=MAC_BROADCAST, eth_type=ETH_TYPE_ARP)
    actions = []
    if not isinstance(out_port, list):
        out_port = [out_port]

    for port in out_port:
        actions.append(parser.OFPActionOutput(port))
    self.add_flow(datapath, priority=20, match=match, actions=actions)
```

```
def add_flow(self, datapath, priority, match, actions, idle_timeout=0, flag=0):
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
    mod = parser.OFPFlowMod(
        datapath=datapath, priority=priority,
        match=match, instructions=inst,
        idle_timeout = idle_timeout, flags=flag
    )
    datapath.send_msg(mod)
```

Figure 2.3: add flow rules helper methods

2.2 Testing

Topology slicing can be easily tested using *pingall* on Mininet and checking **involved hosts in conversations** on switch 2 and switch 4 in Wireshark.



```
mininet> pingall
*** Ping: testing ping reachability
cdn1 -> X h1 h2 X X
cdn2 -> X X X h3 h4
h1 -> cdn1 X X X X
h2 -> cdn1 X X X X
h3 -> X cdn2 X X X
h4 -> X cdn2 X X X
```

arp

Wireshark · Conversations · s2-eth1

Ethernet · 5 | IPv4 | IPv6 | TCP

Address A	Address B
00:00:00:00:00:01	ff:ff:ff:ff:ff:ff
00:00:00:00:00:01	00:00:00:00:00:04
00:00:00:00:00:01	00:00:00:00:00:03
00:00:00:00:00:03	ff:ff:ff:ff:ff:ff
00:00:00:00:00:04	ff:ff:ff:ff:ff:ff

Wireshark · Conversations · s4-eth1

Ethernet · 5 | IPv4 | IPv6 | TCP | UDP

Address A	Address B	Packets	Bytes	Packets A →
00:00:00:00:00:02	ff:ff:ff:ff:ff:ff	9	378	
00:00:00:00:00:02	00:00:00:00:00:06	6	252	
00:00:00:00:00:02	00:00:00:00:00:05	4	168	
00:00:00:00:00:05	ff:ff:ff:ff:ff:ff	12	504	
00:00:00:00:00:06	ff:ff:ff:ff:ff:ff	12	504	

Figure 2.4: topology slicings results

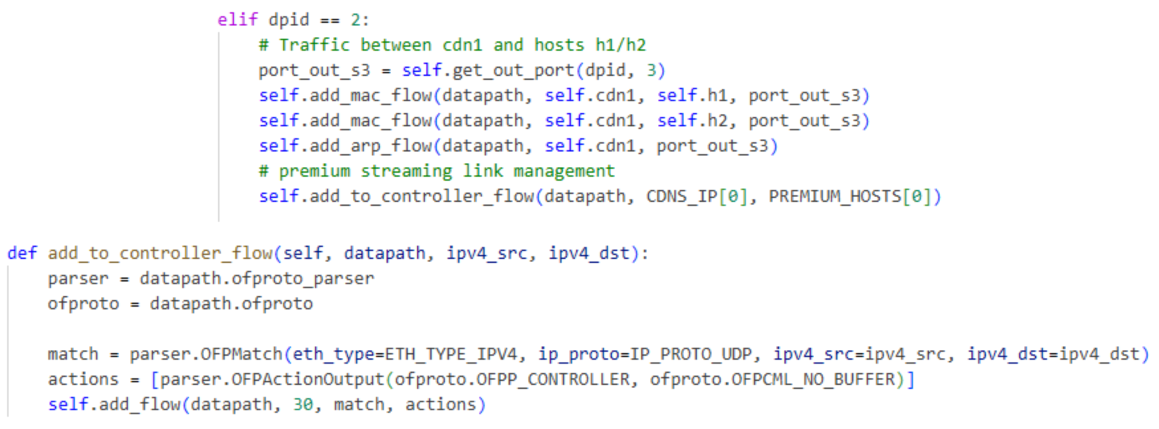
Chapter 3

Service Slicing

On the other hand, we have service slicing, where forwarding is instead **defined based on traffic flow specifications**. In this case, it is implemented by adding a **premium link** (from switch 2/4 to switch 6) **for video streaming fluxes directed to premium users** (h1 and h4)

3.1 Implementation

The biggest problem for service slicing is the **classification of traffic flows** to identify video streaming traffic.
The key operations are the **insertion of new flow rules** managed with an active approach, and **identification by heuristics** using FlowStats from OpenFlow 1.3, which allows collecting a wide variety of metrics on OpenFlow switches.
First of all, it's mandatory to **manage new flux packets by the controller**. To achieve this, a new flow rule, with higher priority than the past ones, has been added to switch 2 and 4 that match packets by UDP protocol, IPv4 source (must be a CDN), and destination (must be a premium user).



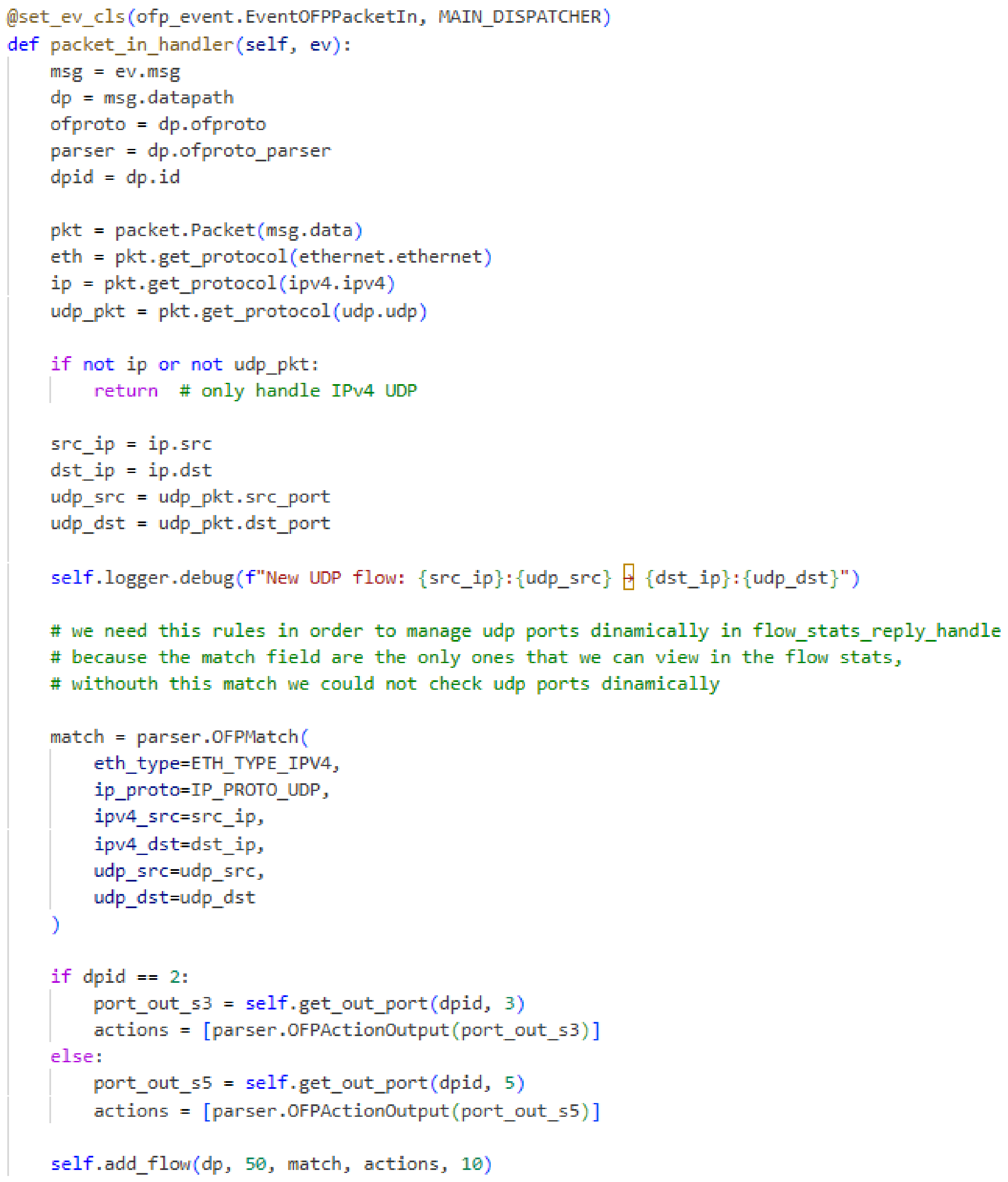
```
elif dpid == 2:
    # Traffic between cdn1 and hosts h1/h2
    port_out_s3 = self.get_out_port(dpid, 3)
    self.add_mac_flow(datapath, self.cdn1, self.h1, port_out_s3)
    self.add_mac_flow(datapath, self.cdn1, self.h2, port_out_s3)
    self.add_arp_flow(datapath, self.cdn1, port_out_s3)
    # premium streaming link management
    self.add_to_controller_flow(datapath, CDNS_IP[0], PREMIUM_HOSTS[0])
```

```
def add_to_controller_flow(self, datapath, ipv4_src, ipv4_dst):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto

    match = parser.OFPMatch(eth_type=ETH_TYPE_IPV4, ip_proto=IP_PROTO_UDP, ipv4_src=ipv4_src, ipv4_dst=ipv4_dst)
    actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER)]
    self.add_flow(datapath, 30, match, actions)
```

Figure 3.1: flow rule to redirect packages to controller

Once a packet matches this new flow rule, it will be managed by the controller's handler that will also install a new flow rule that still forwards to the default path, but matching the **5-tuple {UDP protocol, IP src, IP dst, UDP port src, UDP port dst}**.
This rule is needed due to how OpenFlow Stats works; it identifies flow statistics based **only on flow rule matching fields**, so to separate a premium flux from other UDP fluxes, **also**

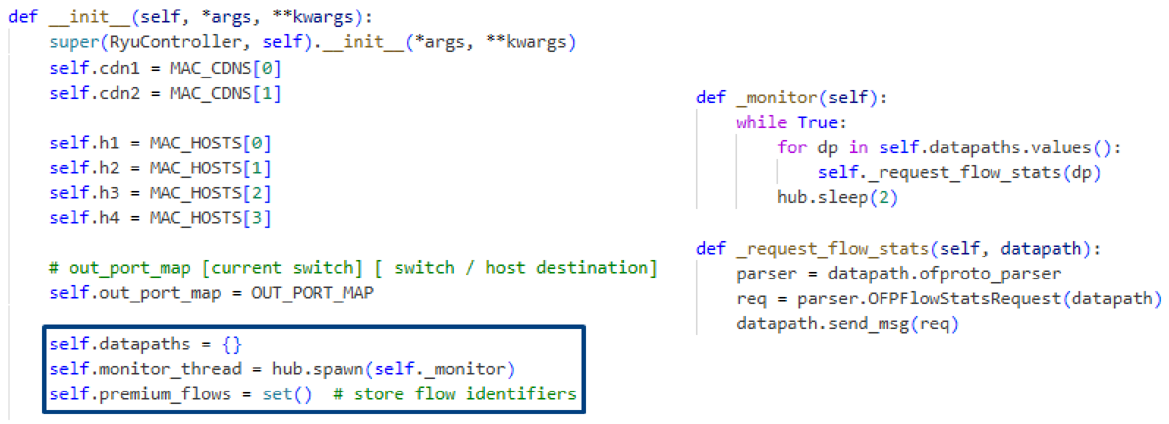
UDP ports must be checked, but they can't be known a priori, so it is needed to add them in the match fields of a flow rule dynamically.



```python
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def packet_in_handler(self, ev):
    msg = ev.msg
    dp = msg.datapath
    ofproto = dp.ofproto
    parser = dp.ofproto_parser
    dpid = dp.id

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocol(ethernet.ethernet)
    ip = pkt.get_protocol(ipv4.ipv4)
    udp_pkt = pkt.get_protocol(udp.udp)

    if not ip or not udp_pkt:
        return  # only handle IPv4 UDP

    src_ip = ip.src
    dst_ip = ip.dst
    udp_src = udp_pkt.src_port
    udp_dst = udp_pkt.dst_port

    self.logger.debug(f"New UDP flow: {src_ip}:{udp_src} → {dst_ip}:{udp_dst}")

    # we need this rules in order to manage udp ports dinamically in flow_stats_reply_handle
    # because the match field are the only ones that we can view in the flow stats,
    # withouth this match we could not check udp ports dinamically

    match = parser.OFPMatch(
        eth_type=ETH_TYPE_IPV4,
        ip_proto=IP_PROTO_UDP,
        ipv4_src=src_ip,
        ipv4_dst=dst_ip,
        udp_src=udp_src,
        udp_dst=udp_dst
    )

    if dpid == 2:
        port_out_s3 = self.get_out_port(dpid, 3)
        actions = [parser.OFPActionOutput(port_out_s3)]
    else:
        port_out_s5 = self.get_out_port(dpid, 5)
        actions = [parser.OFPActionOutput(port_out_s5)]

    self.add_flow(dp, 50, match, actions, 10)
```

Figure 3.2: controller reactive packet handler

Now, to capture flow statistics, the controller must request them. This can be achieved by activating a **thread monitor** on controller startup, which **periodically sends stats requests** to switches.

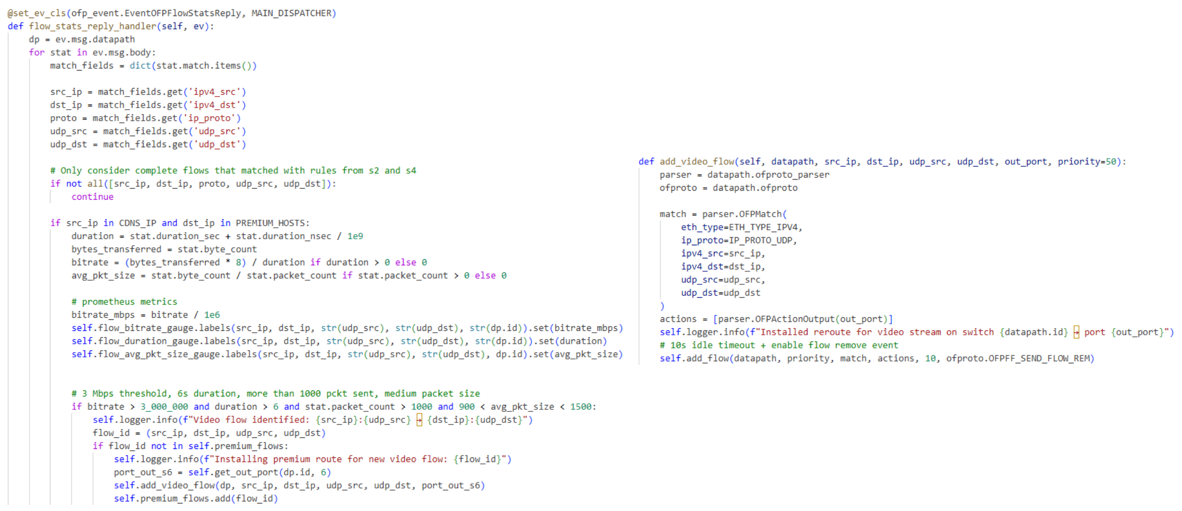


```python
def __init__(self, *args, **kwargs):
    super(RyuController, self).__init__(*args, **kwargs)
    self.cdn1 = MAC_CDNS[0]
    self.cdn2 = MAC_CDNS[1]

    self.h1 = MAC_HOSTS[0]
    self.h2 = MAC_HOSTS[1]
    self.h3 = MAC_HOSTS[2]
    self.h4 = MAC_HOSTS[3]

    # out_port_map [current switch] [ switch / host destination]
    self.out_port_map = OUT_PORT_MAP

    self.datapaths = {}
    self.monitor_thread = hub.spawn(self._monitor)
    self.premium_flows = set()  # store flow identifiers
```

```python
def _monitor(self):
    while True:
        for dp in self.datapaths.values():
            self._request_flow_stats(dp)
        hub.sleep(2)

def _request_flow_stats(self, datapath):
    parser = datapath.ofproto_parser
    req = parser.OFPFlowStatsRequest(datapath)
    datapath.send_msg(req)
```

Figure 3.3: network stats monitor

But the true heart of Flow Classification is in the **Flow Stats Reply event handler**, where every flux sent by the switches will be checked.
As told previously, to separate an UDP video streaming flux from all other fluxes, we use the 5-tuple {UDP protocol, IP src, IP dst, UDP port src, UDP port dst}, so we can directly **discard the fluxes that doesn't have all the fields**, because they're fluxes that haven't matched with one of the proactive flow rule inserted previously, **and that are not directed from one of the cdns, to one of the premium users**.
Finally, it's time to evaluate flow metrics and classify the flux as premium or not, by **heuristic threshold approach**, checking bitrate, duration, packet count, and packet size.
If a premium flow is identified, a new flow rule will be installed to use the direct premium link from switch 2/4 to switch 6, matching the 5-tuple with idle timeout of 10s, and save this 5-tuple in a local controller set to avoid duplicate flow rules.

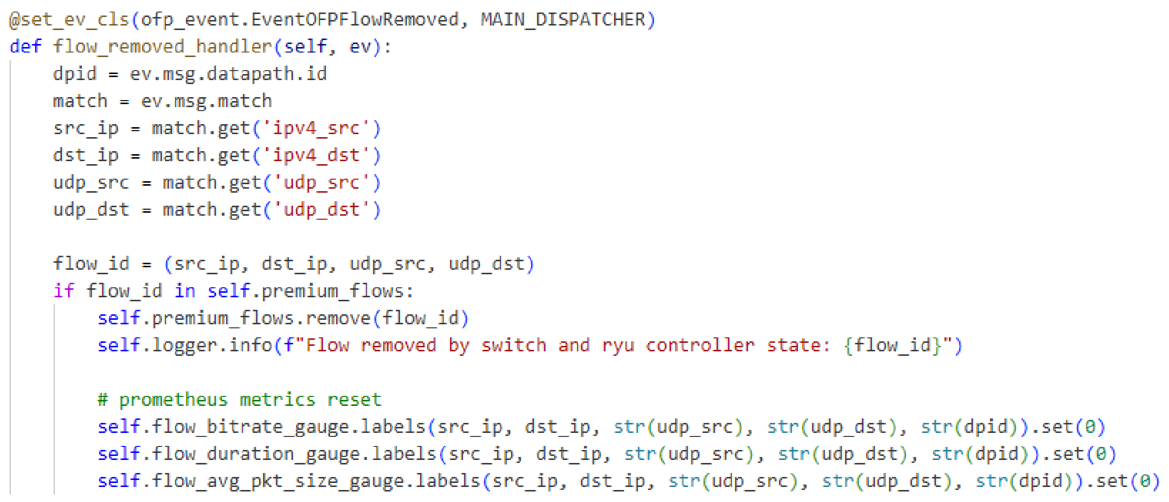


```python
@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def flow_stats_reply_handler(self, ev):
    dp = ev.msg.datapath
    for stat in ev.msg.body:
        match_fields = dict(stat.match.items())

        src_ip = match_fields.get('ipv4_src')
        dst_ip = match_fields.get('ipv4_dst')
        proto = match_fields.get('ip_proto')
        udp_src = match_fields.get('udp_src')
        udp_dst = match_fields.get('udp_dst')

        # Only consider complete flows that matched with rules from s2 and s4
        if not all([src_ip, dst_ip, proto, udp_src, udp_dst]):
            continue

        if src_ip in CDNS_IP and dst_ip in PREMIUM_HOSTS:
            duration = stat.duration_sec + stat.duration_nsec / 1e9
            bytes_transferred = stat.byte_count
            bitrate = (bytes_transferred * 8) / duration if duration > 0 else 0
            avg_pkt_size = stat.byte_count / stat.packet_count if stat.packet_count > 0 else 0

            # prometheus metrics
            bitrate_mbps = bitrate / 1e6
            self.flow_bitrate_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dp.id)).set(bitrate_mbps)
            self.flow_duration_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dp.id)).set(duration)
            self.flow_avg_pkt_size_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), dp.id).set(avg_pkt_size)


            # 3 Mbps threshold, 6s duration, more than 1000 pckt sent, medium packet size
            if bitrate > 3_000_000 and duration > 6 and stat.packet_count > 1000 and 900 < avg_pkt_size < 1500:
                self.logger.info(f"Video flow identified: {src_ip}:{udp_src} → {dst_ip}:{udp_dst}")
                flow_id = (src_ip, dst_ip, udp_src, udp_dst)
                if flow_id not in self.premium_flows:
                    self.logger.info(f"Installing premium route for new video flow: {flow_id}")
                    port_out_s6 = self.get_out_port(dp.id, 6)
                    self.add_video_flow(dp, src_ip, dst_ip, udp_src, udp_dst, port_out_s6)
                    self.premium_flows.add(flow_id)
```

```python
def add_video_flow(self, datapath, src_ip, dst_ip, udp_src, udp_dst, out_port, priority=50):
    parser = datapath.ofproto_parser
    ofproto = datapath.ofproto

    match = parser.OFPMatch(
        eth_type=ETH_TYPE_IPV4,
        ip_proto=IP_PROTO_UDP,
        ipv4_src=src_ip,
        ipv4_dst=dst_ip,
        udp_src=udp_src,
        udp_dst=udp_dst
    )
    actions = [parser.OFPActionOutput(out_port)]
    self.logger.info(f"Installed reroute for video stream on switch {datapath.id} → port {out_port}")
    # 10s idle timeout + enable flow remove event
    self.add_flow(datapath, priority, match, actions, 10, ofproto.OFPFF_SEND_FLOW_REM)
```

Figure 3.4: OpenFlow Stats reply handler and add premium video flow method

After a flow is closed, next flows could use the same 5-tuple, so to not let wrong flows use premium links, an **idle timeout** has been added to premium flow rules. The timeout also triggers the **Flow Removed event**, used to remove the flow from the controller's local set.

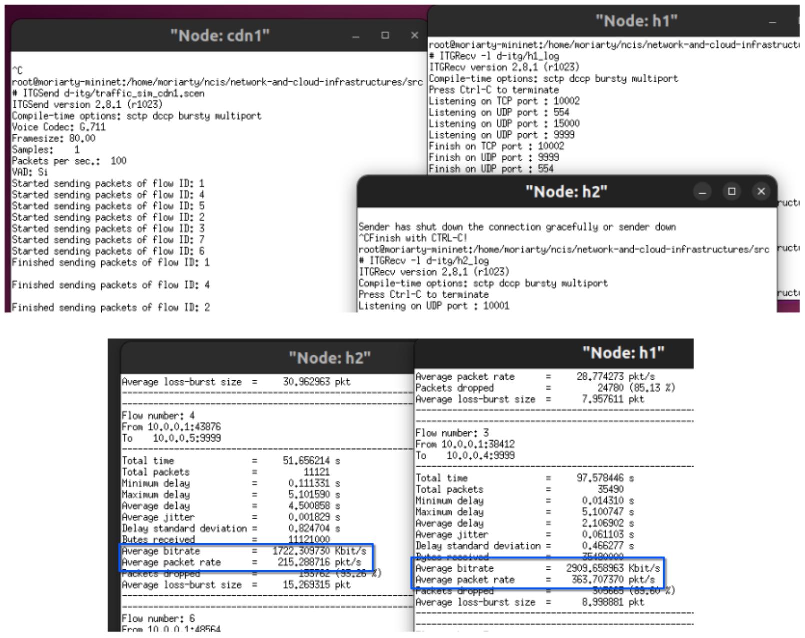


```
@set_ev_cls(ofp_event.EventOFPFlowRemoved, MAIN_DISPATCHER)
def flow_removed_handler(self, ev):
    dpid = ev.msg.datapath.id
    match = ev.msg.match
    src_ip = match.get('ipv4_src')
    dst_ip = match.get('ipv4_dst')
    udp_src = match.get('udp_src')
    udp_dst = match.get('udp_dst')

    flow_id = (src_ip, dst_ip, udp_src, udp_dst)
    if flow_id in self.premium_flows:
        self.premium_flows.remove(flow_id)
        self.logger.info(f"Flow removed by switch and ryu controller state: {flow_id}")

        # prometheus metrics reset
        self.flow_bitrate_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dpid)).set(0)
        self.flow_duration_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dpid)).set(0)
        self.flow_avg_pkt_size_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dpid)).set(0)
```

Figure 3.5: flow remove event handler

3.2 Testing

For service slicing, we need more advanced testing than what we've done for topology slicing to effectively show performance improvements of using the premium links, and we also need to simulate real network traffic, so here it comes, D-ITG[1]. **D-ITG allows advanced traffic generation** starting from file configuration, but also **generates flow statistics reports** helpful to compare premium traffic against normal traffic.



```
^C
root@moriarty-mininet:/home/moriarty/ncis/network-and-cloud-infrastructures/src
# ITGSend d-itg/traffic_sim_cdn1.scen
ITGSend version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Voice Codec: G.711
Framesize: 80.00
Samples:    1
Packets per sec.:  100
VAD: Si
Started sending packets of flow ID: 1
Started sending packets of flow ID: 4
Started sending packets of flow ID: 5
Started sending packets of flow ID: 2
Started sending packets of flow ID: 3
Started sending packets of flow ID: 7
Started sending packets of flow ID: 6
Finished sending packets of flow ID: 1

Finished sending packets of flow ID: 4

Finished sending packets of flow ID: 2
```

```
root@moriarty-mininet:/home/moriarty/ncis/network-and-cloud-infrastruct
# ITGRecv -l d-itg/h1_log
ITGRecv version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Press Ctrl-C to terminate
Listening on TCP port : 10002
Listening on UDP port : 554
Listening on UDP port : 15000
Listening on UDP port : 9999
Finish on TCP port : 10002
Finish on UDP port : 9999
Finish on UDP port : 554
```

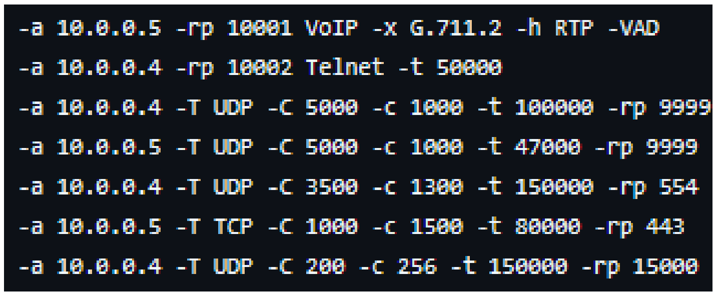
```
Sender has shut down the connection gracefully or sender down
^CFinish with CTRL-C!
root@moriarty-mininet:/home/moriarty/ncis/network-and-cloud-infrastructures/src
# ITGRecv -l d-itg/h2_log
ITGRecv version 2.8.1 (r1023)
Compile-time options: sctp dccp bursty multiport
Press Ctrl-C to terminate
Listening on UDP port : 10001
```

```
Average loss-burst size  =     30.962963 pkt
----------------------------------------------------------
----------------------------------------------------------
Flow number: 4
From 10.0.0.1:43876
To    10.0.0.5:9999
----------------------------------------------------------
Total time               =     51.656214 s
Total packets            =         11121
Minimum delay            =      0.111331 s
Maximum delay            =      5.101590 s
Average delay            =      4.500858 s
Average jitter           =      0.001829 s
Delay standard deviation =      0.824704 s
Bytes received           =      11121000
Average bitrate          =   1722.309730 Kbit/s
Average packet rate      =    215.288716 pkt/s
Packets dropped          =        155762 (93.26 %)
Average loss-burst size  =     15.269315 pkt
----------------------------------------------------------
----------------------------------------------------------
Flow number: 6
From 10.0.0.1:48564
```

```
Average packet rate      =     28.774273 pkt/s
Packets dropped          =         24780 (85.13 %)
Average loss-burst size  =      7.957611 pkt
----------------------------------------------------------
----------------------------------------------------------
Flow number: 3
From 10.0.0.1:38412
To    10.0.0.4:9999
----------------------------------------------------------
Total time               =     97.578446 s
Total packets            =         35490
Minimum delay            =      0.014310 s
Maximum delay            =      5.100747 s
Average delay            =      2.106902 s
Average jitter           =      0.061103 s
Delay standard deviation =      0.466277 s
Bytes received           =      35490000
Average bitrate          =   2909.658963 Kbit/s
Average packet rate      =    363.707370 pkt/s
Packets dropped          =        305665 (89.60 %)
Average loss-burst size  =      8.998881 pkt
----------------------------------------------------------
----------------------------------------------------------
```

Figure 3.6: D-ITG traffic generation on network and reports example (h1 premium, h2 normal)

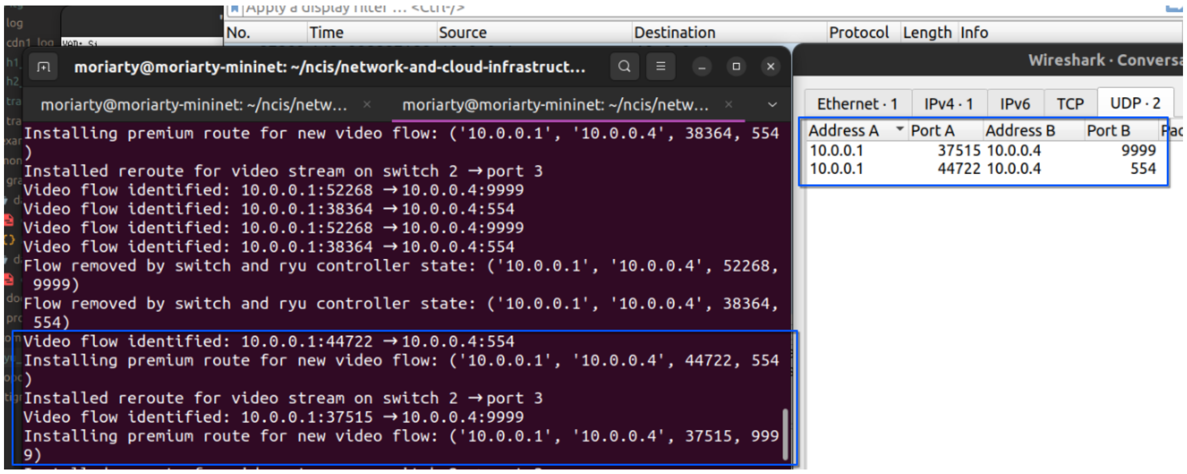
[1]for further references check [ITG]



```
-a 10.0.0.5 -rp 10001 VoIP -x G.711.2 -h RTP -VAD
-a 10.0.0.4 -rp 10002 Telnet -t 50000
-a 10.0.0.4 -T UDP -C 5000 -c 1000 -t 100000 -rp 9999
-a 10.0.0.5 -T UDP -C 5000 -c 1000 -t 47000 -rp 9999
-a 10.0.0.4 -T UDP -C 3500 -c 1300 -t 150000 -rp 554
-a 10.0.0.5 -T TCP -C 1000 -c 1500 -t 80000 -rp 443
-a 10.0.0.4 -T UDP -C 200 -c 256 -t 150000 -rp 15000
```

Figure 3.7: D-ITG traffic generation file

As shown in Fig. 3.8, we can also see that premium video streaming fluxes are correctly identified by the controller, while with Wireshark it is possible to check that **on the premium links** (like switch 2, interface 3) **only the premium fluxes can pass**.



moriarty@moriarty-mininet: ~/ncis/network-and-cloud-infrastruct...
moriarty@moriarty-mininet: ~/ncis/netw...
moriarty@moriarty-mininet: ~/ncis/netw...
Wireshark · Conversa
Ethernet · 1
IPv4 · 1
IPv6
TCP
UDP · 2

```
Installing premium route for new video flow: ('10.0.0.1', '10.0.0.4', 38364, 554
)
Installed reroute for video stream on switch 2 →port 3
Video flow identified: 10.0.0.1:52268 →10.0.0.4:9999
Video flow identified: 10.0.0.1:38364 →10.0.0.4:554
Video flow identified: 10.0.0.1:52268 →10.0.0.4:9999
Video flow identified: 10.0.0.1:38364 →10.0.0.4:554
Flow removed by switch and ryu controller state: ('10.0.0.1', '10.0.0.4', 52268,
 9999)
Flow removed by switch and ryu controller state: ('10.0.0.1', '10.0.0.4', 38364,
 554)
Video flow identified: 10.0.0.1:44722 →10.0.0.4:554
Installing premium route for new video flow: ('10.0.0.1', '10.0.0.4', 44722, 554
)
Installed reroute for video stream on switch 2 →port 3
Video flow identified: 10.0.0.1:37515 →10.0.0.4:9999
Installing premium route for new video flow: ('10.0.0.1', '10.0.0.4', 37515, 999
9)
```

Address A	Port A	Address B	Port B
10.0.0.1	37515	10.0.0.4	9999
10.0.0.1	44722	10.0.0.4	554

Figure 3.8: premium flows identification and rerouting

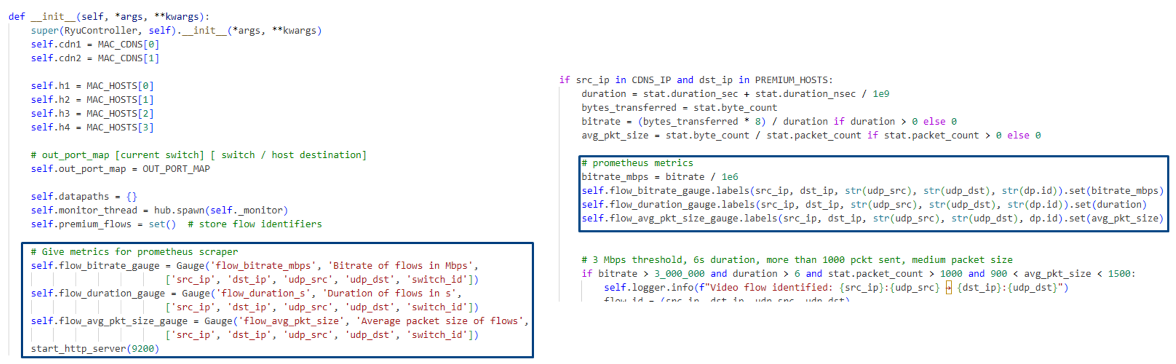
Chapter 4

Monitoring

Another crucial task in network management is **monitoring** to continuously evaluate network performance, check network state, etc.
In this network, it has been assumed that **only premium links can be monitored as part of its paid service**, but there is no reason to deny monitoring all the links of the network to always know the full network's state.
Thanks to OpenFlow Stats, most of the work is already done because the metrics are already available in the Flow Stats Reply event, so it is only needed to eventually **elaborate and export them to some dashboard**. The dashboard is a combination of:

- **Prometheus**: periodically scrape metrics from controller as time series, to aggregate and/or elaborate them (can also activate Alerts)
- **Grafana**: graphical tool to create custom dashboard, can use Prometheus as data source

To use these tools[1], **the network metrics must be exposed to let Prometheus scrape them**. To do this is mandatory to install the Python client library to open an HTTP server and specify exposed metrics on controller startup, to later collect those metrics like bitrate, duration and average packet size in the Flow Stats Reply handler.



```python
def __init__(self, *args, **kwargs):
    super(RyuController, self).__init__(*args, **kwargs)
    self.cdn1 = MAC_CDNS[0]
    self.cdn2 = MAC_CDNS[1]

    self.h1 = MAC_HOSTS[0]
    self.h2 = MAC_HOSTS[1]
    self.h3 = MAC_HOSTS[2]
    self.h4 = MAC_HOSTS[3]

    # out_port_map [current switch] [ switch / host destination]
    self.out_port_map = OUT_PORT_MAP

    self.datapaths = {}
    self.monitor_thread = hub.spawn(self._monitor)
    self.premium_flows = set()  # store flow identifiers

    # Give metrics for prometheus scraper
    self.flow_bitrate_gauge = Gauge('flow_bitrate_mbps', 'Bitrate of flows in Mbps',
                                    ['src_ip', 'dst_ip', 'udp_src', 'udp_dst', 'switch_id'])
    self.flow_duration_gauge = Gauge('flow_duration_s', 'Duration of flows in s',
                                    ['src_ip', 'dst_ip', 'udp_src', 'udp_dst', 'switch_id'])
    self.flow_avg_pkt_size_gauge = Gauge('flow_avg_pkt_size', 'Average packet size of flows',
                                    ['src_ip', 'dst_ip', 'udp_src', 'udp_dst', 'switch_id'])
    start_http_server(9200)
```

```python
if src_ip in CDNS_IP and dst_ip in PREMIUM_HOSTS:
    duration = stat.duration_sec + stat.duration_nsec / 1e9
    bytes_transferred = stat.byte_count
    bitrate = (bytes_transferred * 8) / duration if duration > 0 else 0
    avg_pkt_size = stat.byte_count / stat.packet_count if stat.packet_count > 0 else 0

    # prometheus metrics
    bitrate_mbps = bitrate / 1e6
    self.flow_bitrate_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dp.id)).set(bitrate_mbps)
    self.flow_duration_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), str(dp.id)).set(duration)
    self.flow_avg_pkt_size_gauge.labels(src_ip, dst_ip, str(udp_src), str(udp_dst), dp.id).set(avg_pkt_size)

    # 3 Mbps threshold, 6s duration, more than 1000 pckt sent, medium packet size
    if bitrate > 3_000_000 and duration > 6 and stat.packet_count > 1000 and 900 < avg_pkt_size < 1500:
        self.logger.info(f"Video flow identified: {src_ip}:{udp_src} → {dst_ip}:{udp_dst}")
```

Figure 4.1: Prometheus setup in Ryu controller and stats reply handler

Another crucial point is the **deployment** of these two applications. They could be installed as normal applications, but they're also Cloud Ready and can be directly **managed as Docker containers**, so to simplify the setup, the apps are managed by **Docker Compose** passing the configuration through YAML files.

[1]an advanced example can be viewed in [Th]

```yaml
services:
  prometheus:
    image: prom/prometheus:latest
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml
    ports:
      - "9090:9090"

  grafana:
    image: grafana/grafana:latest
    ports:
      - "3000:3000"
    volumes:
      - ./grafana/provisioning:/etc/grafana/provisioning
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=password
```

Figure 4.2: docker compose yaml file

Now the setup is complete and already working, so after scraping some network metrics from the controller, they will be available to Grafana, where it is possible to create a **custom real-time Dashboard accessible from a web UI** like the one in Figure 4.3.

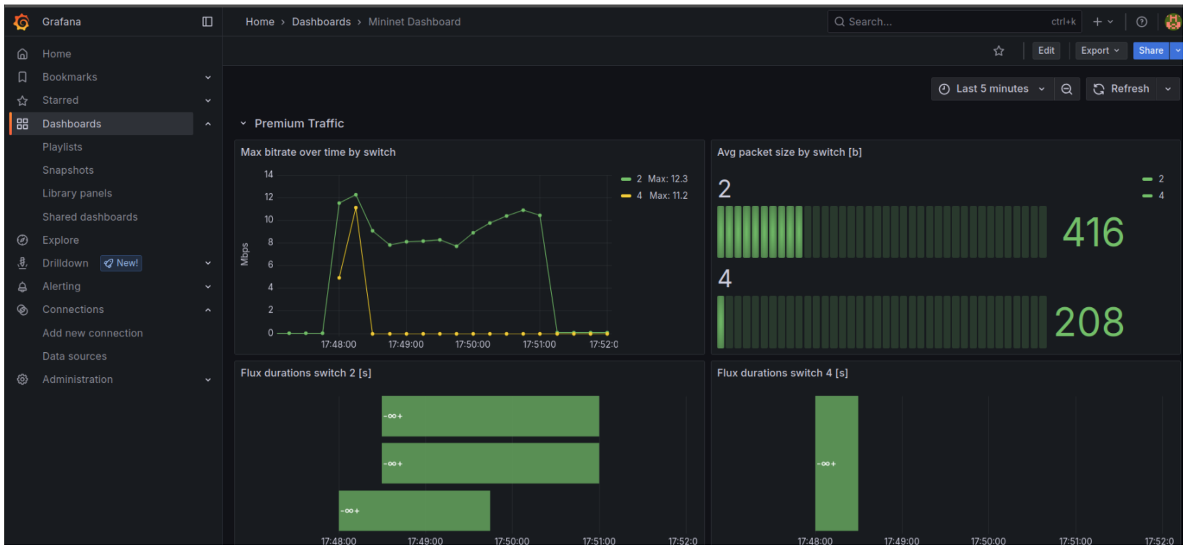


Figure 4.3: Grafana dashboard example for premium links

Bibliography

[ITG] Advanced Traffic Generation with D-ITG, https://shorturl.at/s8I31.

[Th] Marcello Esposito. Monitoraggio dell'orchestratore Kubernetes con Prometheus, https://www.overleaf.com/read/ghpzcqxrnhgf#3d481e, 2024.