Report on the Advanced Computer Network lab:

DASH through legacy and SDN network

by **Adam MIR-SADJADI**
Computer Science Master 1 Student
adam.mir-sadjadi@etu.univ-cotedazur.fr

Submitted by **Ramon APARICIO-PARDO**

Schoolyear 2024-2025, Semester 2
Université Côte d'Azur

May 5, 2025

Contents

Sorry this report is longer than asked but I couldn't fit it in the format asked as I didn't achieve the main goal of lab 3 and it is easier to read if I recall each question and put the plots in-between rather than in appendices.

1 Lab 2 : Legacy Networking

1.1 Question 1

Check the python code topo legacy.py used to generate the network. Why do you think that the RTTs obtained in the pings are close to 20 ms ? Is this consistent with the routing tables in the devices ?

As we can see, the delay for each link between the routers is set to 10ms :

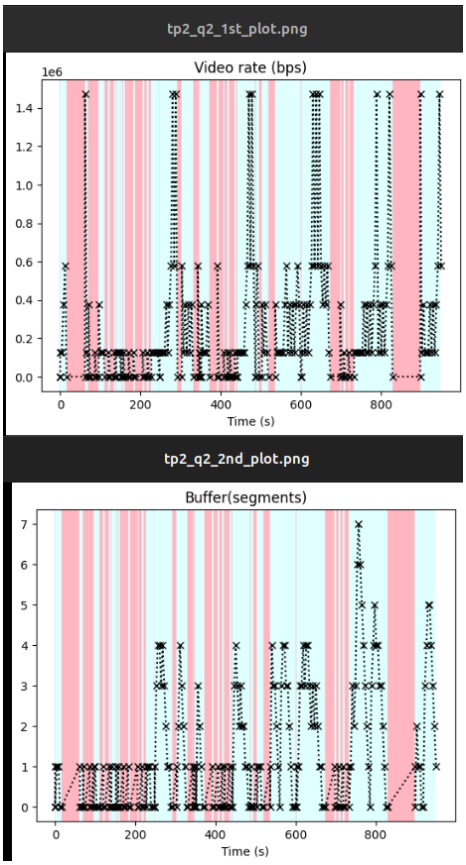
```
info( '*** Add links\n')
      r3r1 = {'bw':1,'delay':'10ms'}
      net.addLink(r3, r1, intfName1='r3-eth1',intfName2='r1-eth1',
          cls=TCLink , **r3r1)
      r1r2 = {'bw':5,'delay':'10ms'}
      net.addLink(r1, r2, intfName1='r1-eth0',intfName2='r2-eth0',
          cls=TCLink , **r1r2)
      r2r3 = {'bw':5,'delay':'10ms'}
      net.addLink(r2, r3, intfName1='r2-eth1',intfName2='r3-eth0',
          cls=TCLink , **r2r3)
```

Therefore, values close to 20ms returned by the ping indicate that a single link is followed twice (forward and backward, as these are RTT values).
We can say that the client reaches the server through the r1 - r3 1mbps link, otherwise values would not be able to reach under 40ms ((10ms + 10ms) * 2) in the case the connection used r1-r2 then r2-r3 links.
It is consistent with the routing tables as the r1-r3 link is the default link for communication of both r1 and r3, which are always the first and last component of the connection before or after client and server :

```
r1.cmd("route add default gw 10.0.3.3 r1-eth1")
r3.cmd("route add default gw 10.0.3.1 r3-eth1")
```

1.2 Question 2

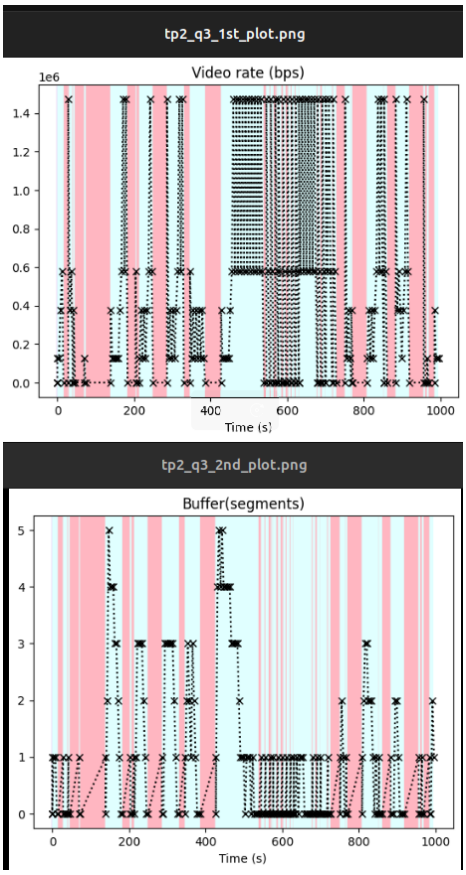
Question 2 : Do as indicated above and plot with the Python notebook : the curves showing the video rate and the buffer status. Which is the actual end-to-end bandwidth seen by the client ? Is this consistent with the routing configuration ? What are the consequences over the video coding rate selected by the DASH adaptation algorithm ?



As we can see, the bitrate is not stable and oscillates between 0 and almost 1.5 Mbps. If it was always sufficient, the buffer size would not change, but here the video is paused several times. We can estimate visually the actuel end-to-end bandwidth is in average 0.3 Mbps. Each time it reaches 1.5Mbps it instantly drops, meaning this is too high a bitrate for the connection.
Yes it is consistent with the routing configuration because as we have seen the link used between client and server is the r1 - r3 link which is limited to 1Mbps, therefore a bandwidth of 1.5Mbps cannot actually be achieved. We can see a "step" pattern corresponding to the DASH adaptation algorithm changing the selected segment quality (coding rate) for the next segment each time the bitrate is not adequate for the bandwidth

1.3 Question 3

How do the video quality and the buffered video time vary depending on the FTP transfer ? Does adaptive streaming allow attaining the maximal quality when the FTP transfer is active ? Why ?

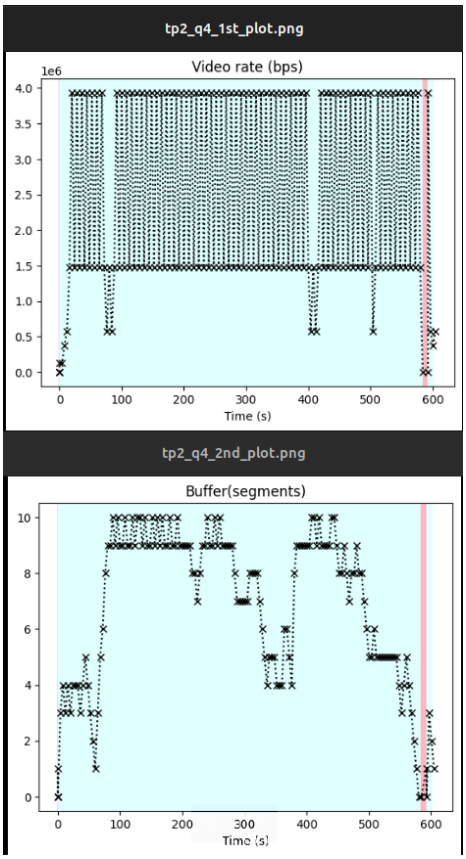


On the new plots, we can see clear red background roughly between 20s and 150s, corresponding to the behavior described for the FTP file transfer lauched from script_ftp.sh (knowing the transfer starts around 20s and ends around 110s). At this time, DASH video playback is frozen due to unsufficient available bandwidth. At the same time, we see the buffer struggles to build up after 20s before staying empty on the 60-150s time period. The FTP transfer using TCP congestion control maximizes its bandwidth used while DASH, in competition with it, lowers the video quality to maintain playback. The file transfer can then take even more space until the point where DASH doesn't even have enough available bandwidth to maintain playback at the lowest quality. Once the transfer is stopped, we observe the same DASH behavior as before, which confirms the file transfer was the cause. Therefore no, adaptive streaming doesn't allow attaining the maximal quality when FTP transfer is active, and it can even prevent attaining the minimal quality.

1.4 Question 4

How would you improve the video quality of the adaptive streaming in such a case ? Hint : as network engineer you cannot change the application protocols, that are standards, but you can introduce changes in your network configuration. Implement your solution and test it

I would want to differentiate between the two types of traffic (basic FTP and DASH) and direct the FTP traffic on the 1Mbps link while directing the DASH exchange on the 5Mbps link (going through r2). As legacy networking can't and we're exchanging between the same client and server, my solution will be to add a new secondary IP and server dedicated to DASH exchange and use it for a secondary routing overriding the default one, enabling to use both routes and attain maximal quality for the video playback.
See the new topo_legacy.py file in the zip archive.
I used static IP routing using the resources given in the lab. We now request the DASH playback from 172.16.1.2 (new secondary IP server address) and ensure traffic going to and coming from (using PBR) 172.16.1.2 goes through r2 while default traffic goes through r1r3 link as before. The syntax of the DASH script call is now the following :
python3 AStream-master/dist/client/dash_client.py -m
http://172.16.1.2/media/BigBuckBunny/4sec/BigBuckBunny_4s.mpd -p 'basic' -d
Redirection through the 5 Mbps link works, we obtained rtt values while pinging 172.16.1.2 from client above 40ms, ensuring both ways of traffic go through r2 with 4*10ms delay. Pinging 172.16.1.1 still returns values above 20m so default traffic takes the same route as before.



1.5 Question 5

What happens now with the video quality ? You are not anymore a networking engineer. You are completely free to propose a solution, even if it is not realistic with the current state of the art. Could you propose a better solution ?

On the new plots obtained at question 4, we see first the video quality requested is much higher, oscillating between 3.9Mbps and 1.4 Mbps which attests available bandwidth is higher than before (no more video pausing or very little, the background is continuously green), and second that there seems not to be any impact from the ftp transfer which was launched at the beginning of the DASH exchange, even though buffer build up seems to struggle a bit more during the first 100ms, but it might be my virtual machine struggling with the multiple processes too.
We can conclude the separation is effective and seems to work as intended. We might wonder why quality is not stable at 3.9Mbps but once again it could be my virtual machine struggling.

This solution works but is really specific and might not be the best approach as it requires the user to use a different IP address when requesting a DASH media. A better solution, ignoring current restrictions, would be to be able to treat differently DASH or high priority traffic (such as with QoS), and for instance reserve a bandwidth equal to the one necessary for a given quality (requested by the user or the highest one) given it is available while being "fair" as for network ressource sharing. This would also allow more flexibility as it might be more easily reassignable and scalable.

2 Lab 3 : SDN Networking

First of all, I didn't manage to achieve this lab. I think I have the right approach but there must be a problem either with my machine (less likely) or the syntax of my code (most likely, but I didn't find it after several hours). I will detail what I did and why I think it should work, and I'm giving you the code snippets I wrote, but the plots I obtained (which I don't put here because they are irrelevant) for the DASH traffic never went above 1.4Mbps as I managed to do in lab 2.

2.1 Question 1

Set up an SDN rule so that DASH video traffic is handled separately from other traffic types using OpenFlow commands. Once properly configured, verify that video streaming achieves maximum quality.

The rules corresponding to the 1Mbps per second are the following :

```
sudo ovs-ofctl add-flow s1 ,in_port=3,actions=output:1
sudo ovs-ofctl add-flow s3 ,in_port=1,actions=output:3
sudo ovs-ofctl add-flow s3 ,in_port=3,actions=output:1
sudo ovs-ofctl add-flow s1 ,in_port=1,actions=output:3
```

I want to target HTTP traffic. Usually, http uses port 80, 443 (https), and I saw there was a mapping between port 80 and 8080 in the topo_sdn.py server defintion. I used tcpdump on my client and server to monitor traffic and I saw it was indeed directed to the "http" port of the server (172.16.1.1). I started only with the flow for port 80 but eventually added the others as it still didn't work up to 3.9Mbps.
Therefore I added (following the exact syntax of the ovs-ofctd documentation you linked in the lab) :

```
sudo ovs-ofctl add-flow s1
    dl_type=0x86dd,in_port=3,nw_proto=6,tp_dst=80,actions=output:2
sudo ovs-ofctl add-flow s1
    dl_type=0x86dd,in_port=3,nw_proto=6,tp_dst=443,actions=output:2
sudo ovs-ofctl add-flow s1
    dl_type=0x86dd,in_port=3,nw_proto=6,tp_dst=8080,actions=output:2
sudo ovs-ofctl add-flow s1
    dl_type=0x0800,in_port=3,nw_proto=6,tp_dst=80,actions=output:2
sudo ovs-ofctl add-flow s1
    dl_type=0x0800,in_port=3,nw_proto=6,tp_dst=443,actions=output:2
sudo ovs-ofctl add-flow s1
    dl_type=0x0800,in_port=3,nw_proto=6,tp_dst=8080,actions=output:2
sudo ovs-ofctl add-flow s2 in_port=1,actions=output:2
sudo ovs-ofctl add-flow s3 in_port=2,actions=output:3
```

```
sudo ovs-ofctl add-flow s3
    dl_type=0x86dd,in_port=3,nw_proto=6,tp_src=80,actions=output:2
sudo ovs-ofctl add-flow s3
    dl_type=0x86dd,in_port=3,nw_proto=6,tp_src=443,actions=output:2
sudo ovs-ofctl add-flow s3
    dl_type=0x86dd,in_port=3,nw_proto=6,tp_src=8080,actions=output:2
sudo ovs-ofctl add-flow s3
    dl_type=0x0800,in_port=3,nw_proto=6,tp_src=80,actions=output:2
sudo ovs-ofctl add-flow s3
    dl_type=0x0800,in_port=3,nw_proto=6,tp_src=443,actions=output:2
sudo ovs-ofctl add-flow s3
    dl_type=0x0800,in_port=3,nw_proto=6,tp_src=8080,actions=output:2
sudo ovs-ofctl add-flow s2 in_port=2,actions=output:1
sudo ovs-ofctl add-flow s1 in_port=2,actions=output:3
```

According to the documentation, nw_proto = 6 corresponds to TCP traffic, dl_type **must** be precised when using tp_src or tp_dst and correspond to ipv4 or ipv6, the ports seem to be the right ones according to the lab, the topo_sdn.py file and the informations I obtained about my network using dump-flows and ovs-ofctl show s1 (or s2 or s3)...
I decided to move on to the next part, being unable to fulfill this task with any syntax after several hours (note that my virtual machine is struggling a lot with all the processes I launch which makes each try harder).

2.2 Question 2

modify the code of_ myPOX.py to get the wished behaviour

Once again, I didn't manage to make it work as I wanted. I added new methods for routing specifically HTTP packets using the http_packet method which was defined in the original code (I added ipv6 compatibility in case it was the issue, but it was not). I can see HTTP packets are recognized when I launch the DASH exchange through the printing the result of each http_packet check (1 or 0) but it isn't routed right. First, I could simulate a whole streaming session but the plots didn't show quality going above 1.4Mbps, and on my latest tries I cannot reach the server anymore (maybe the result of a modification of the controller that I didn't notice or just my machine needing rebooting). I shouldn't be too far from the solution in my opinion but I would need to spend more time on the subject. Being alone for this homework I didn't find enough time even with the delay to go through the whole debugging.
Here is the code snippet I added for http rules specifically (as well as minor modification of the instructions flow not listed here so http_packets identified go through the new method http_via_s2(self, packet, packet_in)) :

```
def http_via_s2(self, packet, packet_in):
  dpid = dpid_to_str(self.connection.dpid)

  if dpid == "00-00-00-00-00-01": #s1
    if packet_in.in_port == 3: #HTTP from client to server
      self.resend_packet(packet_in, 2)
      msg = of.ofp_flow_mod()
      msg.match.dl_type = packet.type
      msg.match.in_port = 3
      msg.match.nw_proto = 6
      msg.match.tp_dst = 80
      msg.match.dl_type = ethernet.IP_TYPE
      msg.actions.append(of.ofp_action_output(port=2))
      self.connection.send(msg)

    elif packet_in.in_port == 2:
      self.resend_packet(packet_in, 3)
      msg = of.ofp_flow_mod()
      msg.match.in_port = 2
      msg.actions.append(of.ofp_action_output(port=3))
      self.connection.send(msg)

    elif dpid == "00-00-00-00-00-02": #s2
      if packet_in.in_port == 1:
        self.resend_packet(packet_in, 2)
        msg = of.ofp_flow_mod()
        msg.match.in_port = 1
        msg.actions.append(of.ofp_action_output(port=2))
        self.connection.send(msg)

      elif packet_in.in_port == 2:
        self.resend_packet(packet_in, 1)
        msg = of.ofp_flow_mod()
        msg.match.in_port = 2
        msg.actions.append(of.ofp_action_output(port=1))
        self.connection.send(msg)

      elif dpid == "00-00-00-00-00-03": #s3
        if packet_in.in_port == 2:
          self.resend_packet(packet_in, 3)
          msg = of.ofp_flow_mod()
          msg.match.in_port = 2
          msg.actions.append(of.ofp_action_output(port=3))
          self.connection.send(msg)

        elif packet_in.in_port == 3: #HTTP from server to client
          self.resend_packet(packet_in, 2)
          msg = of.ofp_flow_mod()
          msg.match.dl_type = packet.type
```

```
msg.match.in_port = 3
msg.match.nw_proto = 6
msg.match.tp_src = 80
msg.match.dl_type = ethernet.IP_TYPE
msg.actions.append(of.ofp_action_output(port=2))
self.connection.send(msg)
```

The idea is just to redirect traffic coming from client or server (in_port 3 on s1 and s3) to s2 if they match the HAS characteristics (HTTP traffic so dst or src port 80, over TCP, ipv4 or 6) and to forward messages in and from s2 to the following port. It is simple but should be enough to emulate what we tried to do for the previous question and what we did with the secondary IP address in lab 2.