Lab 3: Video streaming & SDN routing lab

Advanced Computer Networks
Master 1 2024-25
Université Côte d'Azur
Instructor : Ramón APARICIO PARDO
(ramon.aparicio-pardo@univ-cotedazur.fr)

9th April 2025

UNIVERSITÉ
CÔTE D'AZUR

1 Objective

This group project aims that you understand the tradeoff in terms of Quality of Experience between video quality and rebuffering time by designing your own adaptive bitrate algorithm. To do that, a DASH server and a DASH client are provided.

2 Rules

You can work in groups of 1 or 2 students.

You have to deliver : (i) some source code, and (ii) a final report with the answers of the Lab 2&3 of the lab by April 27th 2025.

3 Before the lab

For the labs, we will use Containernet, a fork of the famous Mininet network emulator, which allows using Docker containers as hosts in emulated network topologies.

You will need to install it in your Ubuntu VM following these instructions. I recommend you the Option 1: Bare-metal installation. I didn't check the *Option 2 : Nested Docker deployment*.

I suggest you that you install `Containernet` in your `/home/USERNAME` folder.

4 Configuring the network

In this laboratory activity, we are going to work with the topology shown in Fig. 1 representing the SDN version of the legacy network studied in the previous lab.

Basically, we replace the Layer-3 routers with SDN switches and we add a SDN controller : all this implemented with `Containernet`. The client and server machines are implemented

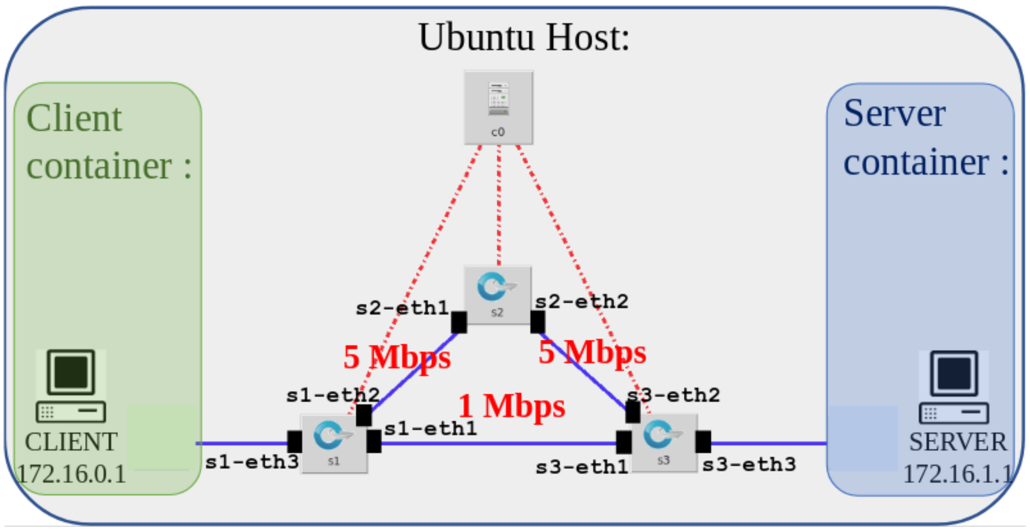

FIGURE 1 – SDN network topology.

again by the same containers as in the previous lab to use the same DASH client and the same FTP and HTTP servers.

In the `lms` course, you will find the python code `topo_sdn.py` implementing the SDN network in Fig. 1. This code should be in the folder `/containernet/custom/`. Then, you can run this network by typing in the terminal of your host system :

```
USERNAME@HOSTNAME:~$  sudo -E env PATH=$PATH python3 custom/topo_sdn.py
```

or

```
USERNAME@HOSTNAME:~$  sudo python3 custom/topo_sdn.py
```

5 Configuring SDN flows tables

In contrat to the previous part of the Lab, at this moment, if you try on testing the end-to-end connectivity client-to-server (e.g., `ping 172.16.1.1` from the client container), you will fail since the SDN switches are not "programmed." That is the main difference with respect to Layer-3 routers. A router has a by-default behaviour and it works properly if it's configured (IP adresses assigned to interfaces and routing tables populated.) Here, "someone" (a human operator or a SDN controller) needs to "program" the SDN switches using the OpenFlow protocol.

First, manually, you are going to program the switches to have a global behaviour equivalent to the by-default routing tables of the legacy network of the previous lab, that is, the direct route `s1`↔`s2`. You will type in the terminal of the linux hosting system :

```
USERNAME@HOSTNAME:~$ sudo ovs-ofctl add-flow s1 ,in_port=3,actions=output:1
USERNAME@HOSTNAME:~$ sudo ovs-ofctl add-flow s3 ,in_port=1,actions=output:3
USERNAME@HOSTNAME:~$ sudo ovs-ofctl add-flow s3 ,in_port=3,actions=output:1
USERNAME@HOSTNAME:~$ sudo ovs-ofctl add-flow s1 ,in_port=1,actions=output:3
```

These commands allows injecting SDN rules in the switches. For example, the first one injects in the switch s1 a rule that can be stated as *for any packet incoming via port 3 (matching condition), forward it via port 1 (action).* You will find more information about the OpenFlow commands in : (i) the ovs+ofctl+Commands site, (ii) the Open vSwitch Manual and, (iii) the Mininet wiki. *NOTE : bridge in the terminology of these websites means SDN switch.*

If you need to know how `containernet` has mapped links and network interfaces to SDN switch port numbers, type in the `containernet` command prompt the next commands

```
containernet> ports
containernet> links
```

To verify the status of the SDN switch tables, type in the Mininet VM console :

```
USERNAME@HOSTNAME:~$ sudo ovs-ofctl dump-flows s1
USERNAME@HOSTNAME:~$ sudo ovs-ofctl dump-flows s2
USERNAME@HOSTNAME:~$ sudo ovs-ofctl dump-flows s3
```

If you what erase the tables, type :

```
USERNAME@HOSTNAME:~$ sudo ovs-ofctl del-flows s1
USERNAME@HOSTNAME:~$ sudo ovs-ofctl del-flows s2
USERNAME@HOSTNAME:~$ sudo ovs-ofctl del-flows s3
```

Now, we come back to the issue detected at the end of previous lab. You have realised that all the traffic (FTP and HTTP-based Adaptive Streaming) was routed directly to the 1-Mbps bottleneck link between routers 1 and 2 and that was perturbing the video streaming. One solution would be separate the FTP from the DASH video traffic. Then, set

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.*

6 Using a SDN controller

On this section, we aim to introduce the same configuration as above but using a SDN controller. We will use the POX controller (a configurable OpenFlow controller). Then, you will need to download the code of the github POX project. I recommend you to do it in your `/home/USERNAME` folder, next to the `Containernet` folder, by typing :

```
USERNAME@HOSTNAME:~$ git clone https://github.com/noxrepo/pox.git
```

Once that you have the POX project code, you need to download from the `lms` course the python code `of_myPOX.py` implementing a POX controller (a configurable OpenFlow controller), and put in the right place (~`/pox/pox/misc.`) and, run it from the terminal of the linux hosting system :

```
USERNAME@HOSTNAME:~/pox$ ./pox.py log.level --DEBUG misc.of_myPOX
```

You will find documentation about how programming the POX controller here :

1. Mininet wiki
2. OpenFlow Tutorial - OpenFlow Wiki
3. Pox github site
4. Mininet site

You can use the `of_myPOX.py` file as template.

Question 2 *: Modify the code `of_myPOX.py` to get the wished behaviour*

7 Final assignment

You have to deliver a final zip folder by April 27th at 23 :59 at the moodle drop box. The zip folder must contain the elements proving that you have managed to implement a POX controller able to treat HAS (HTTP-based adaptive streaming) traffic in such a way that the maximal video quality (3.9 Mbps) is attained during the most part of the video session. The SDN controller will manage to do that, regardless the presence of other TCP flows, as a FTP download. Then, the zip folder must consist of :

1. A pdf report (max. 3 pages) explaining the solution that you found, with screenshots showing that your solution is working properly for two cases :
   — When the only existing traffic in the emulation framework is the video HAS traffic.
   — When the `script_ftp.sh` is running at the same time as the HAS video session. In this case, during ca. 90 seconds, a FTP flow and a HAS flow will be forced to share the framework.
2. The Python source code of the SDN POX controller.

You will be graded depending on : (1) mainly, the performance of the POX controller to handle the two cases ; and, (2) complementarily, the quality of the pdf report (clarity, writing, ...)

8 Annexe

```
packet   # This is the parsed packet data.
packet_in # The actual ofp_packet_in message.

def http_packet(self, packet): # to identify a HTTP packet
  http_pkt = 0

  # We check if the type field in the the data link layer (MAC) header
                                       corresponds to "IPv4" (for other
                                       protocols encapsulated by the  data
                                        link layer (MAC) header, see https
                                       ://github.com/att/pox/blob/master/
                                       pox/lib/packet/ethernet.py#L43)
  if packet.type == ethernet.IP_TYPE:
    #  Find the specified protocol layer based on its class type or name (a
                                         list of class types or names in
                                         https://github.com/att/pox/blob/
                                         master/pox/lib/packet/__init__.py
                                         #L56)
    ipv4_packet = packet.find("ipv4")
    # We check if the protocol field of the IPv4 header matches the value
                                         associated to TCP (see https://
                                         github.com/att/pox/blob/master/
                                         pox/lib/packet/ipv4.py)
    if ipv4_packet.protocol == ipv4.TCP_PROTOCOL:
      tcp_segment = ipv4_packet.find("tcp")
      # We check if the source or destination ports field of the TCP header
                                           match the port number
                                           associated to HTTP (see https
                                           ://github.com/att/pox/blob/
                                           master/pox/lib/packet/tcp.py)
      if tcp_segment.dstport == 80 or tcp_segment.srcport == 80 :
        http_pkt = 1
  return http_pkt


def proc_http (self, packet, packet_in):

  ipv4_packet = packet.find("ipv4")
  if ipv4_packet.protocol ==  ipv4.TCP_PROTOCOL:
     tcp_segment = packet.find("tcp")
     if tcp_segment.dstport == 80 or tcp_segment.srcport == 80 :
        log.debug("TCP segment targeted to %s:%d captured by first time by
                                        switch %s" % (ipv4_packet.
                                        dstip, tcp_segment.dstport,
                                        dpid_to_str(self.connection.
                                        dpid)))
        #if captured at s1 and targeted to h2, we send out via port 2
        if dpid_to_str(self.connection.dpid) == "00-00-00-00-00-01":
          if ipv4_packet.dstip == IPAddr(serverIP_str) :
            log.debug("We send out directly packet to s2 via port 2")
            self.resend_packet(packet_in, 2)
            log.debug("Installing flow ...")
            msg = of.ofp_flow_mod() # An  OpenFlow message is created. This
                                        message will encapsulate a
                                         flow table entry.
            #msg.match = of.ofp_match.from_packet(packet) #The matching rule
                                        of the flow table entry is
                                         defined. In this case,
                                        any packet matching the
                                        headers of packet.
            #Another way of defining the matching rule of the flow table
                                        entry. In this case, we
                                        set the target values of
                                        the headers (see https://
                                        link.infini.fr/-5HG6X12)
            # To see the list of matching fields and their equivalent in the
                                        parsed packet data "packet
                                        ," https://github.com/att/
                                        pox/blob/master/pox/
                                        openflow/libopenflow_01.py
                                        #L891
            msg.match.dl_type = packet.type  # The type field in the the data
                                         link layer (MAC) header.
            msg.match.nw_dst = ipv4_packet.dstip # The IPv4 destination
                                        address in the IPv4 header
                                        .
            msg.match.nw_proto = ipv4_packet.protocol # The transport
                                        protocol field in the IPv4
                                         header.
            msg.match.tp_dst = tcp_segment.dstport # The TCP destination port
                                         in the TCP header.
            msg.actions.append(of.ofp_action_output(port=2)) #The action of
                                        the flow table entry is
                                        defined: here forward the
                                        packet via the port 2 of
                                        the switch.
            self.connection.send(msg)
```