Project: Management of software-defined networks

Assembly & Tools

- Operating System: *Linux Ubuntu*
- Tools and software: Oracle VirtualBox, *Mininet, POX*

Notes

- This work should be done in groups of 2
  - The following documents are available:
    - *Mininet User Manual*
    - Documentation: https://noxrepo.github.io/pox-doc/html/ describing the POX basics.

Objectives

In our course sessions, we have noted the constant evolution of network requirements where the volume of traffic continues to increase. In this context, we emphasised that the almost irremovable nature of network architectures has led to an overall increase in complexity which has, in turn, hindered innovation within networks.

Nowadays, with the emergence of new needs, tools and applications (e.g. multimedia, robotics, etc.), network operators have become increasingly interested in new network paradigms such as SDN (software-defined networks).

We saw in lab 2, a classic case of traffic congestion induced mainly by a static routing mechanism. Among the potential solutions recommended, we noted that a flexible routing mechanism should be put in place to exploit the various available routes in order to minimize congestion in the network.

So, for this project, we are trying to take advantage of the SDN's centralized networking approach to develop a load balancing algorithm that adapts the route of each flow to the current state of the network in order to achieve a better allocation of resources, reducing overall cost and adapting its behaviour to traffic growth.

Formulation of the problem

The increase in the volume of data traffic on the networks has led to the establishment of complex networks unable to meet the needs of operators and users.

A new paradigmatic approach to the centralized control plan, with a comprehensive view and control of each element of the network, can significantly improve network management. It makes it possible to decide the route of each stream, taking into account its characteristics in terms of bandwidth usage or other parameters, in order to achieve a much better use of resources.

The aim of this project is to analyse the possibilities that software-defined networking brings us to develop an application capable of detecting the state of the network and adapting its behavior in order to obtain better performance and better use of network resources.

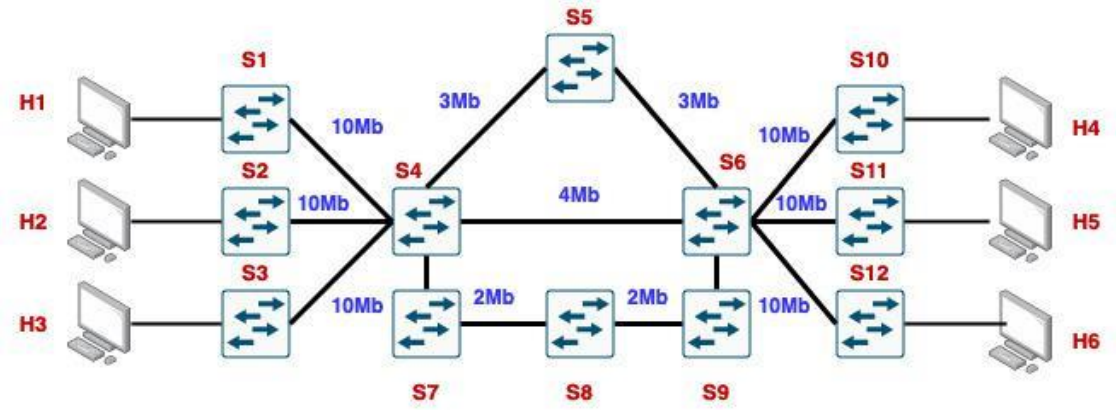
However, the use of resources is not the only important factor. In order to guarantee a certain quality to the traffic carried, it is crucial to maintain certain parameters, such as the time limit within well defined limits.

For example, this project is trying to assess the possibilities offered by the SDN to manage the route for each stream in order to make the most of the available network bandwidth, and is trying to answer the following questions:

- How can software-defined networking improve total network throughput when there is more than one path between source and destination?
- What is the best way to decide which flows should be redirected?
- What are the effects of redirecting a flow in terms of throughput, delay and packet loss?
- Is it possible to keep the quality parameters within a certain margin?

Topology Mininet

The topology considered is the following:



You are asked to write the Mininet python script to get the topology above.

Algorithms to be implemented [1]

Before elaborating on the algorithms to be implemented, we first define some syntax elements:

FlowID: to identify each stream with a unique identifier.
SrcIP: this field contains the IPv4 of the source host that initialized a stream.
DstIP: IPv4 of the destination host.
SrcPort: port number of the source host.
DstPort: port number of the destination host.

UsedBandwidth: transmission speed of a specific flow, in Mbps.
PathID: identify each possible path with a unique identifier.
Hops: contains an identifier for each of the path switches.
Links: list of all links involved in the path. Each link consists of a pair of Switch-Port identifiers.
Ingress: identifier of the switch to which the source host is connected.
Egress is the identifier of the switch to which the destination host is connected.
Capacity: Specifies the maximum capacity of the path. Which corresponds to the capacity of the link having the smallest capacity along the way.
Flows: list of flows that are routed through this path.
UsedBandwidth: sum of the traffic of all streams that use this path, in Mbps. **FreeCapacity**: capacity available in this path. It is the free minimum capacity of the links that form the path.

Data structures:

Flow = {<FlowID>, <SrcIP>, <DstIP>, <SrcPort>, <DstPort>, <UsedBandwidth>}

FlowsCollection = {< $Flow_1$ >, < $Flow_2$ > ... < $Flow_n$ >}

Path = {<PathID>, <Hops>, <Links>, <Ingress>, <Egress>, <Capacity>, <Flows>, <UsedBandwidth>, <FreeCapacity>}

PathsCollection = {< $Path_1$ >, < $Path_2$ > ... < $Path_n$ >}

As explained above, the objective of the algorithm is to dynamically balance loads according to traffic conditions in order to optimize the use of resources.
The first step is to detect a new flow event. This is done when a packet arrives at a switch in the network and the header of that packet does not match any of the switch rules, which triggers algorithm 1.

```
if new flow detected then
    OpenFlowMatch ← header fields that identify the flow;
    endPoints ← source and destination of the new flow;
    if endPoints is not in PathsCollection then
        search all the paths between endPoints;
        add each possible path to PathsCollection;
    end
    routeSelected ← path with highest capacity available;
    for each hop in routeSelected do
        add OpenFlowMatch to forward two-way the flow;
    end
end
```

Algorithm 1 Procedure for detecting a new flow [1]

The next step is to determine if all possible paths between the finish points have already been calculated, which avoids recalculating the possible paths in case they have already been calculated. This is useful because different streams may have the same input and output switches (for example, streams from the same source host to the same destination but with different ports, or hosts connected to the same switch). In case the paths between the input and output switches have not yet been calculated, a function to calculate them is triggered. One way to find all possible paths is to use a depth search algorithm (DFS) below [2].

```
cross-country journeys (graph G)
      for any top s of graph G
            if s is not marked then
                  explore(G, s)

explorer(graph G, top s)
      mark the top s
      display(s)
      for any summit t adjacent to the summit s
            if t is not marked then
                  explore (G, t);
```

For each new path discovered, a new path is stored in **PathsCollection**, having as a unique identifier a string with all nodes through which the path passes.

Once all possible paths are found, it is necessary, first, to select a route and, second, to write the rules in the flow tables of each switch within the selected route. To select the route, the algorithm searches for the path with the highest value of **FreeCapacity**. Since when the flow starts, there is no information about the traffic it will support, we guarantee that the flow will use the route where there is the most capacity available.

Once the route is selected, and using the jumps and links of the specific path, it is necessary to send the appropriate messages to the switches to forward the packets through that path. Thus, each switch within the selected path will have the necessary flow inputs to make the communication between the two end points.

Once the stream has initiated its communication between the two hosts, and begins to transit with the traffic, algorithm 2 begins to capture information about the state of the network in order to adapt to the conditions of each moment. Once congestion is detected on a link, it appears

necessary to redirect some of the current flows. In order to achieve the best possible adaptation of resources, this algorithm focuses on redirecting flows with the lowest traffic.

```
if congestion detected then
    CongestedPort ← switch port which reach capacity threshold;
    Flows ← HashMap of all the active flows;
    FlowToReroute ← null;
    for each flow in Flows do
        if flow is using CongestedPort & flow used bandwidth < FlowToReroute
        used bandwidth then
            FlowToReroute ← flow;
        end
    end
    PC ← all possible path for FlowToReroute;
    Path ← null;
    PendingFlow ← null;
    for each path in PC do
        if path free capacity > FlowToReroute used bandwidth then
            reroute FlowToReroute through this path;
            Algorithm ends;
        end
    end
    if path capacity > Path capacity then
        Path ← path;
        SFC ← flows using this path with used bandwidth < FlowToReroute used
        bandwidth;
        for each flow in SFC do
            if flow used bandwidth < PendingFlow used bandwidth then
                PendingFlow ← flow;
            end
            else
                remove flow from SFC
            end
        end
        if PendingFlow used bandwidth + Path free capacity > FlowToReroute
        used bandwidth then
            reroute FlowToReroute to Path;
            Algorithm ends;
        end
    end
    else
        remove path from PC
    end
end
```

Algorithm 2 Congestion detection procedure [1]

Implementation Notes:

You must define a module developed in Python, to be deployed at the POX controller level.

Data structures must be defined as Python dictionaries.

A *Packet-in listener* is required in your module to capture new packets and trigger algorithm 1.

You must match incoming packets and capture information such as source IP address and destination IP address. Each flow is saved according to the structure described above.
In order to get information about the capabilities used, the developed module must send a message to each of the switches (index: *ofp_flow_stats_request()* and *ofp_port_stats_request()*), to get the counters (see table below) which provide the data needed to calculate bandwidth usage, and update the capacity used of each stream and the free capacity of each path.

Per Table	Per Flow	Per Port	Per Queue
Active Entries	Received Packets	Received Packets	Transmit Packets
Packet Lookups	Received Bytes	Transmitted Packets	Transmit Bytes
Packet Matches	Duration (sec)	Received Bytes	Transmit Overrun Errors
	Duration (nano)	Transmitted Bytes	
		Receive Drops	
		Transmit Drops	
		Receive Errors	
		Transmit Errors	
		Receive Frame Alignment Errors	
		Receive Overrun Errors	
		Receive CRC Errors	
		Collisions	

Figure 1 Openflow meters [1]

Since the meters provide the total amount of bytes transmitted, it is necessary to calculate these data to obtain the bandwidth used for each of the streams and links.
To avoid a large amount of traffic between the switches and the controller, the function is called every second, which allows to update the use of each stream and each switch port in the network.
The available capacity of each path is calculated by taking the information from all the switch ports involved on it, and taking the minimum value (counting received and transmitted bytes).

When the available capacity value of one of the ports reaches 0, algorithm 2 is executed.

Scenario to be simulated:

Hosts H1, H2 and H3 generate three constant flow f1, f2 and f3 (respectively 4, 3, 2 Mbps) which are received by hosts H4, H5 and H6 respectively.
À through figures, show:

- Flow rate in relation to time
- Total flow over time
- Time per flow versus time
- Packet loss in relation to time

Deliverables:

You must submit a compressed file that contains the following files:

- Your Mininet Python script «topologie.py» and the script «parefeu.py»

References:

[1] Navarro, Martí Boada. "Dynamic Load Balancing in Software Defined Networks." *Aalborg University* (2014).

[2] https://en.wikipedia.org/wiki/Algorithme_de_parcours_en_profondeur