Chapter 3

Lecture 3: Quality of Service

Quality of Service (QoS) is the measurement of overall performance of a network service. SDN offers the opportunity to implement different Quality of Service (QoS) mechanisms in the network. The controller has a central view of the whole network, making it easier to react and adapt to changing network conditions. A real-time monitoring system is essential for this. In these exercises, you will create a controller app that monitors two important QoS metrics (throughput and delay) for flows in the network.

Download the files for these exercises here: `https://surfdrive.surf.nl/files/index.php/s/8op45oQrGvZB62F` Extract this archive in `~/HPDN_Exercises/`. This should create a new directory `week_3` in which you can do the exercises.

3.1 Preparation - Ring Topology

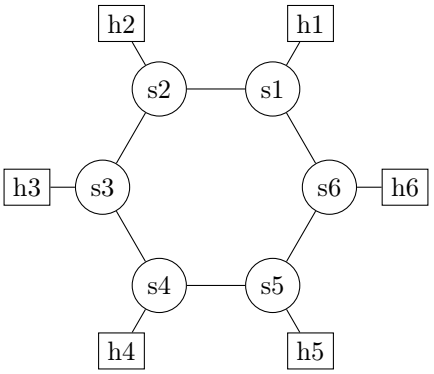

Figure 3.1: Ring topology ($n = 6$).

Exercise:
Write a Mininet Python script that can create ring topologies of any size n. In a ring topology, the switches form a single connected ring (all switches are connected and each switch is linked to exactly two other switches) and each switch connects to one host. See figure 3.1 for an example of a ring topology of $n = 6$ switches.

In addition, write a small automation script that starts an iperf server on host `h1` and an iperf client (connecting to the server) on host `h5`. The client should run for 20 seconds (this can be done using `iperf -t 20`), after which the server should be shut down again.

When running the script in the ring topology, iperf fails. Why?

3.2 Exercise 1. - Bandwidth monitoring

First, we will add an app that routes *all* Ethernet packets between Mininet hosts. Open the Ryu application in `Monitoring.py` (from the archive for this week that you downloaded).

This application forwards packets based on the source and destination MAC addresses. It is similar to the Firewall application from week 2; the main difference is the implementation of the `_add_flow_entry` and `packet_in_handler`.

```
    def _add_flow_entry(self, sid, src, dst, port, ethtype=""):
        """Adds flow entries on switch sid,
        outputting all (allowed) traffic with destination address dst to port.
```

```
        Arguments:
        sid: switch id
        src: src mac address
        dst: dst mac address
        port: output port
        ethtype: ethernet type
        """
        dp = self.network.nodes[sid]['switch'].dp
        ofp = dp.ofproto
        parser = dp.ofproto_parser
        # Add flow rule to match all other data packets
        match = parser.OFPMatch(eth_src=src, eth_dst=dst)
        actions = [parser.OFPActionOutput(port)]
        priority = 1

        instr = [parser.OFPInstructionActions(ofp.OFPIT_APPLY_ACTIONS, actions)]
        cmd = parser.OFPFlowMod(datapath=dp, priority=priority, match=match, instructions=instr)
        dp.send_msg(cmd)

        self.logger.info('ADDED FLOWS ON SWITCH ' + str(sid) + " TO DESTINATION " + str(dst))

    # Packet received
    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        pkt = packet.Packet(msg.data)
        eth = pkt[0]
        src = eth.src
        dst = eth.dst
        if eth.protocol_name != 'ethernet':
            # We should not receive non-ethernet packets
            self.logger.warning('Received unexpected packet:')
            self.logger.warning(str(pkt))
            return

        # Don't do anything with LLDP, not even logging
        if eth.ethertype == ether_types.ETH_TYPE_LLDP:
            return

        self.logger.info('Received ethernet packet')
        self.logger.info('From ' + src + ' to ' + dst)

        if dst not in self.network:
        # We have not yet received any packets from dst
        # So we do not know its location in the network
        # Simply broadcast the packet to all ports without known links
            self._broadcast_unk(msg.data)
            return

        dp = msg.datapath
        sid = dp.id
        if eth.ethertype == 0x0800:
        # Compute path to dst
            try:
                path = nx.shortest_path(self.network, source=sid, target=dst)
            except (nx.NetworkXNoPath, nx.NetworkXError):
                self.logger.warning('No path from switch ' + str(sid) + ' to ' + dst)
                return False

        self._install_path(src, path)

        # Send packet directly to dst
```

```
self._output_packet(path[−2], [self.network[path[−2]][path[−1]]['src_port']], msg.data)
```

While the Firewall application filtered packets based on the destination MAC address, in this application both source and destination MAC are used to identify separate flows (when `_add_flow_entry` triggers). Alternatively, you could use source and destination IP addresses or any other unique flow identifier. Additionally, the `packet_in_handler` was simplified to process all Ethernet packets. If you run into problems in this step, reread the **Routing packets** subsection from Chapter 2.

Exercise: As a simple test, create a ring topology of $n = 5$ switches and use the `pingall` command to verify connectivity (remember to start Mininet with `--arp` enabled). Additionally, run the automation script you previously created and compare the outputs with and without the controller.

Don't forget to run the Ryu controller with the `--observe-links` option to enable link discovery.

Next, you will extend the app to make it monitor the throughput of each flow by periodically querying switches.

3.2.1 Collecting monitoring statistics

The OpenFlow protocol defines several messages that allow the controller to query switches for statistics about their current state, such as flow stats, port stats, and table stats (Fig. 3.2).

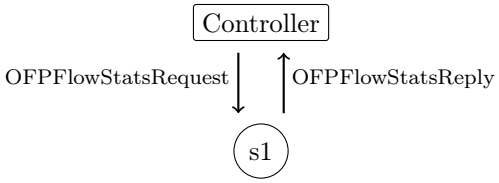

Figure 3.2: Polling statistics from an OpenFlow switch.

Information about the query messages supported by Ryu can be found at: `https://ryu.readthedocs.io/en/latest/ofproto_v1_3_ref.html#multipart-messages`. For our purposes within the scope of this exercise, the following message types suffice:

- `OFPFlowStatsRequest` and `OFPFlowStatsReply` messages to request individual flow statistics (e.g. byte_count, duration_sec etc.)
- `OFPAggregateStatsRequest` and `OFPAggregateStatsReply` messages to request aggregate flow statictics.
- `OFPPortStatsRequest` and `OFPPortStatsReply` messages to request ports statistics (e.g. rx_packets, tx_packets, rx_errors, tx_errors, etc.).
- `OFPPortDescStatsRequest` and `OFPPortDescStatsReply` messages to query port descriptions (e.g. max_speed, and curr_speed).

To add monitoring functionality to the app, our first step is to create two functions:
`request_stats` and `flow_stats_reply`.

The functions are included in `Monitoring.py`, but you need to uncomment them. You may want to add a small change to `request_stats` to avoid that it polls stats for every flow from the switch, and you will have to add a chunk of code to `flow_stats_reply` (as part of this exercise).

`request_stats` will be used to send FlowStats messages to switches. When a switch receives this message, it will send back a FlowStatsReplay message. We process this message in the `flow_stats_reply` function.

```
def request_stats(self, sid):
    """ Send statistics request to a switch sid
    Arguments:
    sid: switch id
    """
    # You will have to change this function a bit
    dp = self.network.nodes[sid]['switch'].dp
    ofp = dp.ofproto
    parser = dp.ofproto_parser

    req = parser.OFPFlowStatsRequest(dp)
    dp.send_msg(req)

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def flow_stats_reply(self, ev):
    """Process flow stats reply info.
```

```
        Calculate flow speed and save it.
        """

        sid = ev.msg.datapath.id
        self.logger.info('Receiving statistics from switch ' + str(sid))

        body = ev.msg.body
        # Fill in this part yourself
```

The `OFPFlowStatsReply` message contains both the total amount of bytes sent through each flow entry, as well as the duration each entry has been installed for.

1. Implement a function called `monitor` that periodically (e.g., every second) polls all switches in the network and computes the average throughput of each flow entry in each switch since the last time they were polled.

2. Finally, create a separate thread to run `monitor` on by adding the following line to your init function:

   ```
   self.monitor_thread = hub.spawn(self.monitor)
   ```

   The above line creates a new attribute `monitor_thread` and calls the function hub.spawn() to create a new thread. Do not forget to import the hub module from ryu.lib (`from ryu.lib import hub`).

Hint: It will be helpful to create a new class "Flow" to create objects to store values in.
After you've completed your implementation, test your app as follows:

1. Start Mininet with your ring topology. If you want to, you can limit the bandwidth of all links, e.g. to 20 Mbps.

2. Start your monitoring app.

3. Use the automation script you prepared before to start an iperf server and client.

4. Check the output from the monitoring app. Compare the values computed by your app to those of iperf, and to the pre-configured bandwidth limit if you had set one. Are there any differences?

What would be advantages or disadvantages of shorter or longer monitoring intervals?

3.3 Exercise 2. - Delay monitoring

Although the switches can provide the controller with a variety of statistics and information, some information can only be obtained by active measurement. For example, we cannot query OF switches for network delay. Thus, if we want to know how much delay the traffic in our network experiences, we need to actively send test packets to the network ourselves. In this final QoS exercise, you will learn how to use Ryu's capability to inject packets into the network.

Suppose we want to measure the delay of a flow travelling from switch `s1` to switch `s3`. To do so, your app should periodically send a probe to switch `s1`, which then should take the exact same path as the flow itself to switch `s3`. Now, by instructing `s3` to send these packets back to the controller, your app can determine the total latency of the path. This is demonstrated in Figure 3.3.

Theoretically, to determine the total latency over the path, the app needs to obtain three values:

- the delay of the probe packet t_{probe} (on the path from s1 to s3 in our example)
- the delay between the first switch and the controller t_1
- the delay between the last switch and the controller t_3

Then, the delay for the path can be calculated by

$$t_{probe} - t_1 - t_3$$

For us, all the switches and the controller run in the same VM. So, values t_1 and t_3 should not have a significant contribution. Moreover, the calculation of t_1 and t_3 can be inaccurate and depends on the scheduling of different processes by the Linux kernel or the resources assigned to the VM. Thus, in this example, we will only calculate the t_{probe} value by keeping track of when the probe packet was sent to the first switch and received by the controller again.

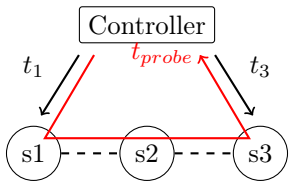

Figure 3.3: Calculating the delay of a path passing through switches s1 - s2 - s3.

Injecting packets into the network

To complete this exercise you will need to inject packets into the network. The following function can be used to send a probe packet out of a specific port on a switch:

```
def send_latency_probe_packet(self, sid, port):
    '''
    Injects latency probe packets in the network
    '''
    self.logger.info('Injecting latency probe packets')
    dp = self.network.nodes[sid]['switch'].dp
    actions = [dp.ofproto_parser.OFPActionOutput(port)]

    pkt = packet.Packet()
    pkt.add_protocol(ethernet.ethernet(ethertype=0xBBBB,
                                       dst=0xAAAAAAAAAAAA,
                                       src=0xBBBBBBBBBBBB))

    pkt.serialize()
    payload = '%d;%f' % (sid, time.time())
    # We must encode the payload to be able to append it to data
    # because data is a bytearray
    data = pkt.data + payload.encode()

    out = dp.ofproto_parser.OFPPacketOut(datapath=dp,
                                         buffer_id=dp.ofproto.OFP_NO_BUFFER,
                                         data=data,
                                         in_port=dp.ofproto.OFPP_CONTROLLER,
                                         actions=actions)
    dp.send_msg(out)
```

(Be careful with copy-pasting from this listing, as some characters (e.g. ') might not get recognized correctly.)

Do not forget to import all required modules:

```
from ryu.lib.packet import (packet, ethernet, ipv4, ipv6)
import time
```

Exercise:

Modify the function `send_latency_probe_packet` to fit your needs. Since you can customize all headers of the packet, make sure that your probe packet will match the same flow entries as the packets generated by the hosts.

Additionally, notice that you can embed the timestamp of the moment when the packet was sent in the payload of the probe packet. You can use this to simplify the calculation of t_{probe}. Modify your PacketIn handler (function `packet_in_handler`) to process the probe packets in addition to the useful data and calculate t_{probe}. Do not forget to install additional rules in the last switch that will forward the probe packet back to the controller.

Finally, make sure to call the function `send_latency_probe_packet` for every monitored path from your previously created monitoring thread.

Test your app as follows:

1. Extend the ring topology by adding bandwidth and delay properties to every link of the ring. There should be no delay between the hosts and the switches (only between switches).

2. Go through the same steps as in exercise 1.

Can you measure the delay?
How many additional flow entries did you need to install?