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.

Solution

The solution files are available here: `https://surfdrive.surf.nl/files/index.php/s/jESqVQuB2nNhT2B`

3.1 Preparation - Ring Topology

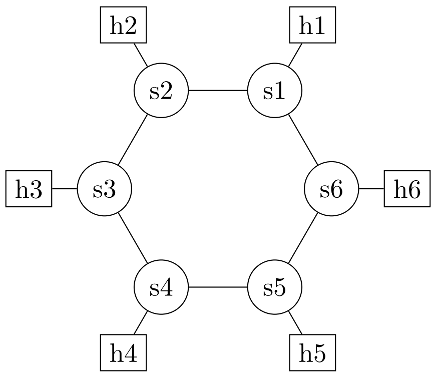

Figure 3.1: Ring topology ($n = 6$).

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?

Solution

The code for the ring topology:

```
from mininet.topo import Topo

class RingTopo(Topo):
    "Simple_topology_example."
    def __init__(self, n):
        "Create custom topo."
        # Initialize topology
        Topo.__init__(self)
        # Add hosts and switches
        switches = []
        hosts = []
        # connect switches to the hosts
        for i in range(n):
            switches.append(self.addSwitch('s%d' % (i+1)))
            hosts.append(self.addHost('h%d' % (i+1)))
            self.addLink(switches[i], hosts[i])

        for i in range(n):
            self.addLink(switches[i], switches[(i+1) % n])


topos = {'ring': RingTopo }
```

The code for the automation script:

```
h1 iperf -s &
h5 iperf -c 10.0.0.1 -t 20
h1 kill %iperf
```

To create the custom topology you should run the command:

```
sudo mn --custom ring.py --topo ring,5
```

After you run the custom script, you should get the following output:

```
connect failed: No route to host
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
```

As you can notice, a TCP connection cannot be established between hosts h1 and h5. The reason for this is the presence of loops in the topology (cf. chapter 1, exercise 5, Section 1.8.2). By default, Mininet switches act as learning switches and have no mechanisms to handle broadcast storms. Thus, in the next part of this exercise we will add an SDN controller to solve the issue.

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.

Solution

The output of the pingall command should look like this.

```
mininet> pingall
*** Ping: testing ping reachability
h0 -> h1 h2 h3 h4
h1 -> h0 h2 h3 h4
h2 -> h0 h1 h3 h4
h3 -> h0 h1 h2 h4
h4 -> h0 h1 h2 h3
*** Results: 0\% dropped (20/20 received)
```

The output of the automation script should look like:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.5 port 60732 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-20.0 sec  46.8 GBytes  20.1 Gbits/sec
[  4] local 10.0.0.1 port 5001 connected with 10.0.0.5 port 60732
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-20.0 sec  46.8 GBytes  20.1 Gbits/sec
```

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).

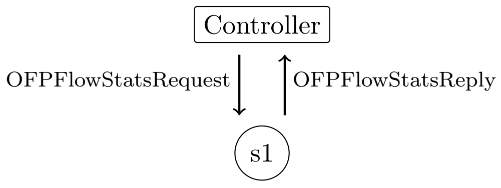

Figure 3.2: Polling statistics from an OpenFlow switch.

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?

Solution

To store the flows we wish to monitor we will use an array. Additionally, we will specify a variable to represent the monitoring interval.

```
    def __init__(self, *args, **kwargs):
        super(Monitoring, self).__init__(*args, **kwargs)
        self.network = nx.DiGraph()
        self.monitored_paths = [] #array to store information about flows we wish to monitor
        self.monitor_period = 1
        self.monitor_thread = hub.spawn(self._monitor)
```

Since we want to collect several matrices for each flow we can simply store them in a separate structure as shown below. We will extend this structure as we add more matrices (e.g. delay, loss, etc.).

```
class Flow():
        def __init__ (self, src, dst, did, port):
                #values needed to identify the flow (src mac, dst mac, first and last switch)
                self.destinationSwitch = did
                self.src_mac = src
                self.dst_mac =dst
                #port from the first switch specifying the port to which the probe packet will be sent out to
                self.sourcePort = port
                self.throughput = 0
                self.bytes_last = 0

        def __eq__(self, other):
        #override the default equal behaviour
                if isinstance(other, self.__class__):
                        return (self.src_mac == other.src_mac) and (self.dst_mac == other.dst_mac)
                else:
                        return False

        def __ne__(self, other):
        #override the default unequal behaviour
                if isinstance(other, self.__class__):
                        return (self.src_mac != other.src_mac) or (self.dst_mac != other.dst_mac)
                else:
                        return True

        def __str__ (self):
                return 'Sw: %d (%s - %s) - (%s kbps)' %
                    (self.destinationSwitch, self.src_mac, self.dst_mac, self.throughput)
```

For each of the monitored flows we need to remember the first and the last switch in the path as they carry the most information. For example, to calculate the throughput we only need to query the last switch in the path as all the packets that are received by the destination host are processed by this switch. Additionally, in the `packetIN` handler you need to add every new flow you wish to monitor to the `monitored_paths` array.

```
        flow = Flow(src, dst, path[len(path)-2], self.network[path[0]][path[1]]['src_port'])
        if flow not in self.monitored_paths:
            self.monitored_paths.append(flow)
```

The function to request the statistics from the switch is shown below. The only difference is to the function shown in the additional match argument. If the match field is not included, this function will poll all the flow stats from the switch, for every flow entry present. In our case, since the number of entries is small, this would not be a problem as we can simply filter out the stats we are interested in. However, in case of a switch that has hundreds of flow entries, the processing delay, as well as the received message would be too large and cause too much overhead. To avoid this, an additional match argument is specified and one request message is sent per each monitored flow.

```
        def request_stats(self, sid, src, dst):
                """ Send statistics request to a switch sid
                Arguments:
                sid: switch id
                src: src mac address
                dst: dst mac address
                """
                dp = self.network.nodes[sid]['switch'].dp
                ofp = dp.ofproto
                parser = dp.ofproto_parser
                #only ask for the stats for the flow we monitor
                #If this wasn't included we would request all the flows. This would
                #increase the processing delay at the switch
                match = parser.OFPMatch(eth_src = src, eth_dst = dst)
                req = parser.OFPFlowStatsRequest(dp, match=match)
                dp.send_msg(req)
```

The function to process the statistics received from the switch is shown below. The body of this message will contain the statistics for all the flows we requested in the previous message. As we only requested one (by specifying the match field) this message will contain only one stats message.

```
@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
        body = ev.msg.body
        #self.logger.info('Receiving statistics from switch ' + str(sid))
        #process the replay message

        for stat in body:
                #the switch will return the stats for all flow entries that match src and dst address
                #for the first switch that will be just one entry, but for the second switch it will be 2
                #(one entry to match packets, one entry to match probe packets)
                #do not process probe packets
                if "eth_type" not in stat.match or stat.match["eth_type"]==2048:
                        #find a flow from monitored_paths that corresponds to this replay
                        matches = [flow for flow in self.monitored_paths if
                            (flow.src_mac == stat.match["eth_src"]
                                    and flow.dst_mac == stat.match["eth_dst"])]
                        #there should be only one match as we sent the requests with the match field
                        matches[0].throughput =
                                8*(stat.byte_count-matches[0].bytes_last)/(1000*self.monitor_period)
                        matches[0].bytes_last = stat.byte_count
```

This function calculates the throughput in bits for the specified monitoring interval. The received *byte* count needs to be multiplied by 8 in order to receive the requested throughput in *bits* per second.

Finally, we need to create a monitoring function that is called by the previously defined monitoring thread:

```
def _monitor(self):
        """
        Main method for the monitoring actions.
        """
        while True:
                self.logger.info('Monitoring stats: num_flows:' + str(len(self.monitored_paths)))
                for m in self.monitored_paths:
                        print m
                        #save timestamps to measure delay from the controller to the switches
                        self.request_stats(m.destinationSwitch, m.src_mac, m.dst_mac)
                hub.sleep(self.monitor_period)
```

The output of this app is shown below (for a maximum bandwidth of $\approx 40Mbps$). Two flows are detected. The second flow is representing the useful traffic sent from the source to the destination, while the first flow represents the ACK messages sent in the opposite direction (used protocol is TCP thus the traffic is always bidirectional). If the monitoring interval is small, we can observe the slow-start phase of the TCP connection, i.e. the sender starts slowly and doubles its sending rate every RTT until a loss is detected.

```
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (0 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (3372 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (13297537 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (22466 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (34428398 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (27281 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (39330670 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (33629 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (41421782 kbps)
Monitoring stats: num_flows:2
```

```
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (35722 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (41271127 kbps)
```

By changing the value of the monitoring period to 3 seconds:

```
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (0 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (20409 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (28069949 kbps)
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (27639 kbps)
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (36958272 kbps)
```

With a longer interval, we get less information, but we cause less additional load in the network.

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

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

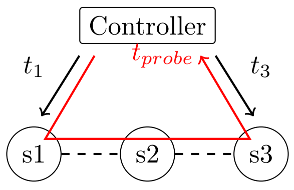

Figure 3.3: Calculating the delay of a path passing through switches s1 - s2 - s3.

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.

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?

Solution

To calculate the delay we need to first extend the previously defined Flow class as shown below, while the instantiation of the Flow objects needs to be updated accordingly:

```
class Flow():
    def __init__ (self, src, dst, sid, did, port):
        #values needed to identify the flow (src mac, dst mac, first and last switch)
        self.destinationSwitch = did
        self.sourceSwitch = sid
        self.src_mac = src
        self.dst_mac =dst
        #port from the first switch specifing the port to which the probe packet will be sent out to
        self.sourcePort = port
        self.delay = 0 #delay of the probe packet
        self.throughput = 0
        self.bytes_last = 0

    def __eq__(self, other):
    #override the default equal behaviour
        if isinstance(other, self.__class__):
                return (self.src_mac == other.src_mac) and (self.dst_mac == other.dst_mac)
            else:
                return False

    def __ne__(self, other):
    #override the default unequal behaviour
            if isinstance(other, self.__class__):
                return (self.src_mac != other.src_mac) or (self.dst_mac != other.dst_mac)
            else:
                return False

    def __str__ (self):
            return 's%d-->s%d (%s - %s) - (%s kbps %s ms)' % (self.destinationSwitch,
                self.sourceSwitch, self.src_mac, self.dst_mac, self.throughput, self.delay*1000)
```

We will first explain how to calculate the delay t_{probe}. We first need to modify the `send_latency_probe_packet` function to fit our needs. To do this, we will add additional arguments to this function such as the switch ID (of the first switch for the monitored path) to send the probe packet to, as well as the output port on the first switch to forward this packet to. Additionally, we will use the source and destination MAC addresses of the monitored flow. This way, the probe packet will be forwarded using the same processing rules as the packets belonging to the flow.

```
    def send_latency_probe_packet(self, sid, port, src, dst):
            '''
            Injects latency probe packets in the network
            Arguments:
            sid: switch id
            src: src mac address
            dst: dst mac address
            port: output port

            '''
            #self.logger.info('Injecting latency probe packets')
    dp = self.network.nodes[sid]['switch'].dp
    actions = [dp.ofproto_parser.OFPActionOutput(port)]

    pkt = packet.Packet()
    # probe packet should have the same source and destination mac as the packets belonging to the monitored flow
    # these packets should be matched by the rules of the original flow on all the switches except the
    # first one
    pkt.add_protocol(ethernet.ethernet(ethertype=self.PROBE_ETHERTYPE, dst=dst, src=src))
    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)
    self.logger.info("Probe sent!")
    dp.send_msg(out)
```

Additionally, in the `init` method of the app we need to add an argument self.PROBE_ETHERTYPE that will be used to differentiate the probe packets from regular packets belonging to the flow. To forward this packet back to the controller at the last switch we need to add one additional rule that will match on the ethtype.

```
    def _install_path(self, src, path):
            """Installs path in the network.
            path[-1] should be a host,
            all other elements of path are required to be switches

```

```
        Arguments:
        path: Sequence of network nodes
        """

        dst = path[-1]

        for i in range(0,len(path)-1):
            current = path[i]
            next = path[i+1]

            port = self.network[current][next]['src_port']
            self._add_flow_entry(current, src, dst, port)
            #add additonal rule for a probe packet at the last switch
            if i == len(path)-2:
                self._add_probe_flow_entry(current, src, dst, port)
```

The `_add_probe_flow_entry` will install a rule to forward the probe packets to the controller. To make sure that probe packets are processed by this rule at the last switch we need to configure a higher priority than the rule to process "normal" packets.

```
    def _add_probe_flow_entry(self, sid, src, dst, port):
        """Adds probe flow entry 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
        match = parser.OFPMatch(eth_src = src, eth_dst = dst, eth_type = self.PROBE_ETHERTYPE)
        actions = [parser.OFPActionOutput(ofp.OFPP_CONTROLLER, ofp.OFPCML_NO_BUFFER)]
        priority = 2

        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␣PROBE␣ENTRY␣ON␣SWITCH␣' + str(sid))
```

Finally, this packet is received in the PacketIN handler (method `packet_in_handler`) and needs to be parsed to extract the delay information as shown below.

```
    if eth.ethertype == self.PROBE_ETHERTYPE:
        #self.logger.info('Received probe packet')
        # data (ip header + timestamp (when the probe packet was sent)) is the payload
        # We must decode it to get a str (data is a bytearray)
        data = pkt[1].decode()
        split_data = data.split(';')
        timestamp = split_data[len(split_data)-1] # last field is the timestamp
        matches = [flow for flow in self.monitored_paths if (flow.src_mac == src and flow.dst_mac == dst)]
        matches[0].delay = time.time() - float(timestamp)
        return
```

If we look at the flow rules at the switches we can notice the following:

```
hpdn@hpdn-VirtualBox:~/ryu$ sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s2
 cookie=0x0, duration=12.587s, table=0, n_packets=27, n_bytes=1620, priority=65535,
    dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=5.739s, table=0, n_packets=2, n_bytes=158, priority=2,
     dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03,dl_type=0x07c7 actions=CONTROLLER:65535
 cookie=0x0, duration=5.756s, table=0, n_packets=199988, n_bytes=13199456, priority=1,
     dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02 actions=output:"s2-eth2"
 cookie=0x0, duration=5.740s, table=0, n_packets=416287, n_bytes=21120858142, priority=1,
     dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03 actions=output:"s2-eth1"
 cookie=0x0, duration=12.626s, table=0, n_packets=2, n_bytes=134,
        priority=0 actions=CONTROLLER:65535
hpdn@hpdn-VirtualBox:~/ryu$ sudo ovs-ofctl --protocols=OpenFlow13 dump-flows s1
 cookie=0x0, duration=14.414s, table=0, n_packets=34, n_bytes=2040, priority=65535,
     dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:65535
 cookie=0x0, duration=7.547s, table=0, n_packets=3, n_bytes=237, priority=2,
     dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02,dl_type=0x07c7 actions=CONTROLLER:65535
 cookie=0x0, duration=7.547s, table=0, n_packets=199988, n_bytes=13199456, priority=1,
     dl_src=00:00:00:00:00:03,dl_dst=00:00:00:00:00:02 actions=output:"s1-eth1"
 cookie=0x0, duration=7.533s, table=0, n_packets=416287, n_bytes=21120858142, priority=1,
     dl_src=00:00:00:00:00:02,dl_dst=00:00:00:00:00:03 actions=output:"s1-eth3"
 cookie=0x0, duration=14.423s, table=0, n_packets=4, n_bytes=280,
        priority=0 actions=CONTROLLER:65535
```

We can notice the additional rules (with the match field `dl_type=0x88cc`) for two flows (between h1 and h2 and between h2 and h1) that are installed only on the last switches and that have the action to send the packet back to the controller. Finally, we need to call the function `send_latency_probe_packet` from the `_monitor` method once for every monitored path.

The output of the app (in case the delay on all the links was configured to 100ms) is shown below. The last value represents t_{probe}.

```
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (52 kbps  107.933044434 ms )
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (9532 kbps  104.384899139 ms )
```

By increasing the delay on the links, we can observe that t_{probe} reflects this.

```
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (0 kbps  506.588935852 ms  )
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps  504.956007004 ms )
Monitoring stats: num_flows:2
(s2 - s1) (00:00:00:00:00:03 - 00:00:00:00:00:02) - (1 kbps  505.844831467 ms )
(s1 - s2) (00:00:00:00:00:02 - 00:00:00:00:00:03) - (0 kbps  507.808923721 ms )
```