Softwarized Network Documentation

Stefan Gore

July 2024

1 Creating Topology

Creating a class Topology that take as input mininet.topo

```
from mininet.topo import Topo

class Topology(Topo):
    def build(self):
        ...
```

Adding a switch:

```
s1 = self.addSwitch('s1')
```

Adding a host:

```
h1 = self.self.addHost('h1', mac='00:00:00:00:00:01')
```

Adding a link:

```
l1 = self.addLink(s1, h1, cls=TCLink, bw=40, delay='15ms')
```

Use a for to creating multiple structure.

To create a controller, fisrt we call from bash the ryu-manager controller and then...

```
system("ryu-manager controller.py &")
controller = RemoteController("c1", "127.0.0.1", 6633)
```

In the start metod we create the net onject as following:

```
    net = Mininet(
    topo=Topology(),
    switch=OVSKernelSwitch,
    controller=controller,
    build=False,
    autoSetMacs=True,
    autoStaticArp=True,
    link=TCLink,
)
```

To enable the Ryu controller you can do it from the terminal:

```
ryu-manager
```

1.1 Controller

1.2 Code analysis

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # install the table-miss flow entry.
    match = parser.OFPMatch()
    actions = [ parser.OFPActionOutput(ofproto.OFPP_CONTROLLER, ofproto.OFPCML_NO_BUFFER) ]
    self.add_flow(datapath, 0, match, actions)
```

The following code is trigger by an event, which is when a switch first connects to the controller and want's to receive configuration messages.
The actions fields is applies to the packet. It sends the packet to the controller 'OFPP-CONTROLLER' and indicates that entire packet should be sent 'OFPCML-NO-BUFFER'

1.3 A summary of the entire process

The process can be rappresented in different steps:

Initialization of the controller, all data structure will be initialize. At the end the controller will create a thread. This part will execute when following terminal command:

```
ryu-manager Controller.py
```

When we perform the following command, we create the switches and the hosts.

```
sudo python3 Topology.py
```

When the switches will connected, the controller will execute some code for each switch with the goal to send request of statistics about the flow and the ports.

When the hosts will connects and will send the **arp request** to the switch that it is connected. The switch will redirect the packet to the controller that will manage with 'packet_in_' message.

When the controller will receive a ARP REQUEST, it will send a flooding Overflow packets to all switches.
OBSERVATION: This process is must faster, indeed we don't need to wait to propagate the ARP REQUEST to all the topology, but only to the controller and then in parallel from controller to all switches.

The purpose of the ARP protocol is to resolve the MAC-IP association and to build the arp tables.

1.4 STATISTICS

The statistic's that the system retrieve are several and this is the table. Here i will describe data_map, bandwidth-portdict. bandwidth flow dict, rttToDpidPortStats, rtt to dpid

We need to retrive different types of statistics: latency, bandwidth and RTT. This because we can do a path optimization based on some or bandwidth or latency.

1.4.1 Latency

To calculate the latency, it is use a custom Ethernet packet. In the following code we are creating the custom packet and inside we are storing the time at the creation and sending. This code will run continuosly to retrive some information. **The packet is created by the controller and sent to the switch**.

```
    def monitor_latency(self, datapath, ofproto):
        hub.sleep(5)
        print("MONITORING LATENCY STARTED dpid: {}".format(datapath.id))
        self.latency_measurement_flag = True
        while True:
            #preparing the OverFlow message
            ofp = datapath.ofproto
            ofp_parser = datapath.ofproto_parser
            actions = [ofp_parser.OFPActionOutput(ofp.OFPP_FLOOD, 0)]
            #custom packet creation
            pkt = packet.Packet()
            pkt.add_protocol(ethernet.ethernet(ethertype=0x07c3,
                                               dst='ff:ff:ff:ff:ff:ff',
                                               src='00:00:00:00:00:09'))
            whole_data = str(time.time()) + '#' + str(datapath.id) + '#'
            pkt.add_protocol(bytes(whole_data, "utf-8"))
            pkt.serialize()
            data = pkt.data
            #building and sending the message
            req = ofp_parser.OFPPacketOut(datapath, ofproto.OFP_NO_BUFFER,
                                          ofproto.OFPP_CONTROLLER, actions, data)
            datapath.send_msg(req)
            hub.sleep(interval_update_latency)
```

The switch when will receive the packet will send back to the controller that will handle in the packet_in function.

```
if eth_pkt.ethertype == 0x07c3:
    ...
                self.data_map[dpid_rec][dpid_sent] = {}
                self.data_map[dpid_rec][dpid_sent]['in_port'] = in_port
                self.data_map[dpid_rec][dpid_sent]['bw'] = []
                self.data_map[dpid_rec][dpid_sent]['latencyRTT'] = []
            latency_link_echo_rtt = time_difference - (float(self.rtt_portStats_to_dpid[dpid_sent]) / 2) -
                    float(self.rtt_portStats_to_dpid[dpid_rec]) / 2)
            latency_obj_rtt = {'timestamp': timestamp_sent, 'value': latency_link_echo_rtt * 1000}
            self.data_map[dpid_rec][dpid_sent]['latencyRTT'].append(latency_obj_rtt)
            self.last_arrived_package[dpid_rec][dpid_sent] = time.time()
    ...
```

When the controller will receive the custom packet, it will calculate create and add values to data_map. This dictionary will present all the statistics that we need.

1.4.2 Bandwidth

The bandwidth can be retrieve from different source: the bandwidth of the single port, or the bandwidth of the flow.

To do this we spawn a functions for each switch that send a request for flow and port statistics.

```
def monitor_sw_controller_latency(self, datapath):
    hub.sleep(0.5 + self.waitTillStart)
    iterator = 0
    while True:
        if iterator % 2 == 0:
            self.send_port_stats_request(datapath)
            print("Sent")
        else:
            self.send_flow_stats_request(datapath)
        iterator += 1
        hub.sleep(interval_controller_switch_latency)
```

All the switches **receiving these request** will sends all the data that have collected to the controller.
The Controller handle the data packet send by the switch with the following functions:

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def port_stats_reply_handler(self, ev):
    ...

@set_ev_cls(ofp_event.EventOFPFlowStatsReply, MAIN_DISPATCHER)
def flow_stats_reply_handler(self, ev):
    ...
```

Both of these functions have same way, fill the rtt_portStats_to_dpid dictionary (explained in the following paragraph), and most important the **bandwidth_flow_dict** and **bandwidth_port_dict** which consists in dictionaries that store bandwidth of ports or flow.

The bandwidth is calculated in this way: rate of byte transmitted over a interval of time.

During the collection of the stats, using this type of packets, the controller will also calculate the **RTT** and stored in *rtt_portStats_to_dpid*
To calculate BW using flow-stats , you can use the formula (number of packets at t2 - number of packets at t1)/t2-t1. Please also note that icmp packets are tiny - hence it might not reflect correct bandwidth measurement. Rather, you can use traffic generation tools such as iperf to generate tcp/udp traffic, periodically measure flow tables and calculate BW using above formula.

1.5 ARP PROTOCOL

ARP (Address Resolution Protocol) is used in networks to map an IP address (Layer 3) to a MAC address (Layer 2). When a device wants to communicate with another device on the same network, it uses ARP to find out the MAC address associated with the target's IP address, allowing the data to be correctly delivered over Ethernet or Wi-Fi. (Adding more abstraction is useful when merging one or more local network, accessing internet).

When run the mini-net topology, the host will send arp request to solve MAC-IP association inside the network. The packets received by the switch will but will be sent straight to the controller and be handle by the following function:

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
    ...
        if arp_pkt:
```

In case of an **arp request**, the controller will check if the mac associate with the destionation ip is already present. Otherwise we must perform a flooding with special packet to all the switches/hosts to retrive the information.

```
elif arp_pkt.opcode == arp.ARP_REQUEST:
    if dst_ip in self.arp_table:
        dst_mac = self.arp_table[dst_ip]
        h1 = self.hosts[src_mac]
        h2 = self.hosts[dst_mac]
        if (h1, h2) not in self.already_routed:
            self.arp_table[src_ip] = src_mac
            dst_mac = self.arp_table[dst_ip]
            h1 = self.hosts[src_mac]
            h2 = self.hosts[dst_mac]
            self.routing_arp(h1, h2, src_ip, dst_ip)
            self.logger.info("Calc needed for DFS routing between h1: {} and h2: {}".format(src_ip, dst_ip
            self.already_routed.append((h1, h2))
        return
    else:
        # flooding ARP request
        actions = [parser.OFPActionOutput(out_port)]
        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data
        out = parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions, data=data)
        datapath.send_msg(out)
```

The following code is when a hosts sends a **arp reply** to the switch in which it is connected. This packet will be redirect to controller, will populate the **arp table**

```
    if arp_pkt.opcode == arp.ARP_REPLY:
        self.arp_table[src_ip] = src_mac
        h1 = self.hosts[src_mac]
        h2 = self.hosts[dst_mac]
        if (h1, h2) not in self.already_routed:
            self.routing_arp(h1, h2, src_ip, dst_ip)
        return
```

When possible the controller will try to actuate a routing which will be described in the proper paragraph.

1.6 TOPOLOGY DISCOVERY

The controller must know the topology to perform its goals. we use different data structure and API calls to retrieve the information. For example we will use data_map and hosts.

The study of the topology can be divided in two phases: In the first step we have a static topology, then when we will perform a host migration or a link failure simulation it must learn the new topology on the fly.

One of the most important data structure in topology discovery is the association between (host mac) with (switch id + port). To add entries to hosts we can do it in different ways: The first way is: to analyse a packet that entry in switch (and then forward to the controller). We store the source mac of the packet and the current switch and port that received the packet.

```
from ryu.controller import ofp_event

@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    ...
    if src_mac not in self.hosts:
        self.hosts[src_mac] = (dpid_rec, in_port)
    ...
```

Another way is to exploit the following decorator and event that will trigger the function in the controller

```
from ryu.topology import event

@set_ev_cls(event.EventHostAdd)
def _event_host_add_handler(self, ev):
```

Another data structure that will determine our topology is the data_map, which contain the association between a switch and another switch with the relative port. This data structure will be created and populated when custom packet will arrive.
In the dynamic topology phase, The data_map structure will change if a link or a switch will go down. This will be possible thanks to decorator:

```
@set_ev_cls(ofp_event.EventOFPPortStateChange, MAIN_DISPATCHER)
    def port_state_change_handler(self, ev):
```

The event that can activate the function is when a link goes down, for example from the mininet CLI with the command *link s1 s2 down*.

Another alternative for the previous event handler is the following:

```
from ryu.controller import dpset

@set_ev_cls(dpset.EventPortModify, MAIN_DISPATCHER)
    def port_modify_handler(self, ev):
```

1.6.1 Ryu topology API for topology discorvery

In alternative for event handler and data structure like dictionaries, we can use some build-in Ryu API function that can provide useful and more easily information.

```
from ryu.topology.api import get_switch, get_link, get_host

new_hosts = get_host(self, dpid=None)
```

new_hosts is a API function that can gives all the hosts of a network or the hosts that are connected to a particular switch if we give as parameter the dpid.
get_link will return all the links in the topology.

1.7 PACKET HANDLER

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    ...
```

The function is called when a packet arrives at the switch, it doesn't match any flow entry, and is sent to the controller.
The function tries to manage in different way different type of packages.

- 0x07c3: this is a custom packet used for latency mesuarment
- LLDP: stands for Link Layer Discovery Protocol. These packets generally are used for topology discovery (to discover switches, routers). I filter these packets because i manly use other strategy for topology discovery.
- IPV6: i filter these because add overcomplexity that doesn't mean nothing for the project.
- ARP: see dedicated paragraph.
- IPV4: see dedicated paragraph
- ICMP: these packets are used for debugging and diagnostic, indeed are used in ping and traceroute commands. The controller will not handles this type of packet.

1.8 DATA STRUCTURE

The main data dictionary is data_map which is structure in the following way:

- first-key represents the switch/datapath which receive the packet
- second-key represents the switch/datapath which sent the packet
- There are several attributes: the input port number, the bandwidth and the latency.

```
self.data_map = {
    1: {  # dpid_rec
        2212: {  # dpid_sent
            'in_port': 21,
            'bw': [],
            'latencyRTT': [
                {'timestamp': 1627356123.123, 'value': 15.6},  # Example latencyRTT data
                {'timestamp': 1627356187.456, 'value': 16.2}
            ]
        }
    }
}
```

self.hosts

Description

This dictionary maps MAC addresses of hosts in the network to their associated switch and port information. It is used to keep track of which switch (identified by a datapath ID) and which port a particular host is connected to.

Structure

- **Key (str)**: The MAC address of the host in the format `"XX:XX:XX:XX:XX:XX"`.
- **Value (tuple)**: A tuple containing two elements:
  - **Element 1 (int)**: The datapath ID (DPID) of the switch the host is connected to.
  - **Element 2 (int)**: The port number on the switch where the host is connected.

Example Content

```
{
    '02:67:fe:a5:72:15': (3, 3),
    '16:82:31:5f:d8:c7': (1, 4),
    '2a:b3:4d:7a:87:39': (1, 1)
}
```

Explanation of Example

- `'02:67:fe:a5:72:15': (3, 3)`
  - Host with MAC address `'02:67:fe:a5:72:15'` is connected to switch with DPID 3 on port 3.

self.arp_table

Description

A dictionary that maps IP addresses to their corresponding MAC addresses in the network.

Structure

- **Key (str)**: The IP address in the format `"X.X.X.X"`.
- **Value (str)**: The MAC address associated with the IP, in the format `"XX:XX:XX:XX:XX:XX"`.

Example Content

```
arp_table = {
    '10.0.0.24': '00:00:00:00:00:01',
    '11.0.0.24': '00:00:00:00:00:02'
}
```

Already Routed Dictionary

Description

`self.already_routed` is a list of tuples representing routing paths that have already been established in the network.

Structure

Each tuple in the list contains two elements:

- **Element 1 (tuple)**: A tuple containing the datapath ID (DPID) and port number of the source switch.
- **Element 2 (tuple)**: A tuple containing the datapath ID (DPID) and port number of the destination switch.

Example Content

```
self.already_routed = [
    ((1, 4), (3, 3)),  # Path from Host with MAC '16:82:31:5f:d8:c7' to Host with MAC '02:67:fe:a5:72:15'
    ((3, 3), (1, 1))   # Path from Host with MAC '02:67:fe:a5:72:15' to Host with MAC '2a:b3:4d:7a:87:39'
]
```

1.9 REAL TIME SYSTEM

1.10 DEBUGGING

When the switches in Mininet are not starting, one way to resolve the issue is to execute the following command:

```
$ sudo service openvswitch-switch restart
```

Additional command to see if any related mininet process is running:

```
$ ps aux | grep -i mininet
```

OBSEVATION: It's not adviced to put the controller on sleep, because will stop it's functionalities, even the packet in handling.

To show the Overflow table, we need in MININET CLI to insert this command

```
dpctl dump-flows
```

1.11 DICTIONARIES AND LISTS ANALYSIS

1.12 HOST MIGRATION

```
@set_ev_cls(dpset.EventPortModify, MAIN_DISPATCHER)
def port_modify_handler(self, ev):
    print("PORT MODIFY")
    port = ev.port
    dp = ev.dp
    pprint(self.hosts)
    switches = get_switch(self, None)
    for l in switches:
        print (" \t\t" + str(l))
    hosts = get_host(self, dpid=dp.id)
    for l in hosts:
        print (" \t\t" + str(l))
```

1.13 FIRST ROUTING

Given a topology made in mininet we start a communication between two hosts. In our case we have h1 as a client and h4 as a server. When h1 start communication with h4 it will begin the ARP PROTOCOL, see the ARP PARAGRAPH. During the ARP Protocol we will handle a routing process.

The Routing process, in the function routing, consists in different steps.

- The first step consists in finding the best path between the switches where the two hosts are connected.
- The second step consists in installing the best path inside all the Overflow switches.

To find the best path we use the functions:

- **get_optimal_path** : it consists in, given the source and destination switch and given the latency dictionary, it will calculate with a DFS the best path with respect to latency. The result path will have the following structure: [1, 2, 4] which represent the switch we must follow.
  Example to call the function:

  ```
  get_optimal_path(self.latency_dict, 1, 3, 'arp')
  ```

  Where 1, 3 represent the starting and the ending switch.
- get_path_cost: ...

The previous function of finding the best route depends on the latency_dict data structure, on his turn depends on the data_map structure.

1.13.1 Path installation

For path installation it means to insert flows in the OpenFlow switches. The path installation needs several parameters like the switches where to install the entry, the match which consists in destinations and source IP, and the action that will consists in output port. In the entry we will also add a **timer**, consequently if an entry will not be used for a certain amount of time, the entry will be deleted. When installing the path, we will install for the arp match.

```
flow_mod = parser.OFPFlowMod(
        datapath=datapath,
        priority=priority,
        command=ofproto.OFPFC_ADD,
        idle_timeout=idle_timeout,  # Set the idle timeout
        hard_timeout=hard_timeout,  # Set the hard timeout
        match=match,
        nstructions=instructions                                    )
```

1.14 HOST MIGRATION

The host migration consists in the movement the connection of the host from one switch to another. Due to this motion we need to change also the OpenFlow tables. One simple strategy is to take all the switch that have as a flow that ip in the match section and delete it. This can be done using two function **del_flow_specific_switch** and **_del_arp_flow_specific_switch**.

1. The process begin when **@set_ev_cls(ofp_event.EventOFPPortStateChange, MAIN_DISPATCHER)** will trigger the controller when a link is down.
2. The event will give us the port and switch was affected. If the port affected is a switch-host port and it was deleted, this represent the beginning of the host migration.
3. We need to change data structure like self.hosts and self.already_routed, because the topology was changed.
4. Consequently we will delete form any flow related with that ip and mac from every switch.

When the same host will reconnect to another switch, it will perform again the ARP PROTOCOL and therefore the routing process.

1.15 LINK FAILURE

The link failure process is intended when a link between two switches goes down and can be done in the MININET CLI with the command *link s1 s2 down*.

1. The process begin when **@set_ev_cls(ofp_event.EventOFPPortStateChange, MAIN_DISPATCHER)** is triggered.
2. We analyse the event, if the port and switch is related with a switch-switch connection and the link was deleted we are detection a link-failure.
3. The first thing to do is to change those data structure that collects and have topology data. So first we do some modification on the self.data_map.
4. We then analyse which flow was inside in the affected switches. Indeed we retrive the ip_scr and ip_dst info form these.
5. We apply the Rerouting function

1.16 SWITCH FAILURE???

1.17 LINK DELAY???

1.18 ANNOTATION

Provare i seguenti eventi: event.EventLinkDelete Inoltre andare a vedere come funziona network.EventMacAddress

1.19 SUPPORT FUNCTION

The following instruction will print the link

```
    for l in links:
        print (" \t\t" + str(l))
```

The output will be the following:

```
**** Printing LINKS *********
Link: Port<dpid=3, port_no=2, LIVE> to Port<dpid=4, port_no=2, LIVE>
Link: Port<dpid=1, port_no=2, LIVE> to Port<dpid=4, port_no=1, LIVE>
Link: Port<dpid=2, port_no=2, LIVE> to Port<dpid=3, port_no=1, LIVE>
Link: Port<dpid=3, port_no=1, LIVE> to Port<dpid=2, port_no=2, LIVE>
Link: Port<dpid=4, port_no=2, LIVE> to Port<dpid=3, port_no=2, LIVE>
Link: Port<dpid=4, port_no=1, LIVE> to Port<dpid=1, port_no=2, LIVE>
```

We can notice that this API functin

For the host we will have this structure

```
Host<mac=00:00:00:00:00:03, port=Port<dpid=1, port_no=5, LIVE>,10.0.0.3::>
Host<mac=00:00:00:00:00:05, port=Port<dpid=3, port_no=4, LIVE>,10.0.0.5::>
Host<mac=00:00:00:00:00:01, port=Port<dpid=1, port_no=3, LIVE>,10.0.0.1>
Host<mac=00:00:00:00:00:02, port=Port<dpid=1, port_no=4, LIVE>,10.0.0.2>
Host<mac=00:00:00:00:00:04, port=Port<dpid=3, port_no=3, LIVE>,10.0.0.4>
Host<mac=00:00:00:00:00:06, port=Port<dpid=3, port_no=5, LIVE>,10.0.0.6>
```

1.20 Annotation

Create a Class for ARP handling as following:

```
class ARPTable:
    def __init__(self):
        self.ip_to_mac = {}
        self.mac_to_ip = {}

    def add_entry(self, ip, mac):
        """Add or update an ARP table entry."""
        self.ip_to_mac[ip] = mac
        self.mac_to_ip[mac] = ip

    def get_mac(self, ip):
        """Retrieve the MAC address associated with the given IP address."""
        return self.ip_to_mac.get(ip)

    def get_ip(self, mac):
        """Retrieve the IP address associated with the given MAC address."""
        return self.mac_to_ip.get(mac)
    ...
```

We also store the source and destination ip, path and its cost in the following data structure called chosen_path_per_flow

```
    {
    '192.168.1.1': {
        '192.168.1.2': (
            ['A', 'B', 'C', 'D', 'E'],  # Optimal path
            50  # Total cost (latency)
        )
    }
}
```

The `PORT STATS` is represented as a nested dictionary:

```
{
    switch\_id: {
        port\_id: bandwidth\_value,
        ...
    },
    ...
}
```

- **switch_id**: Unique identifier of the switch.
- **port_id**: Unique identifier of the port on the switch.
- **bandwidth_value**: Bandwidth usage (e.g., Mbps) for the specified port.

2 Example

Consider the following example data:

```
{
    1: {1: 120.5, 2: 150.0},
    2: {1: 140.0, 2: 160.0}
}
```

This represents:

- **Switch 1**:
  - **Port 1**: 120.5 Mbps
  - **Port 2**: 150.0 Mbps
- **Switch 2**:
  - **Port 1**: 140.0 Mbps
  - **Port 2**: 160.0 Mbps

3 Fields Description

- **switch_id** (Integer): Identifier of the switch.
- **port_id** (Integer): Identifier of the port on the switch.
- **bandwidth_value** (Float): Bandwidth usage for the port, typically in Mbps.

The following instruction is how to manage the Ryu topology API

```
for l in switches:
    print (" \t\t" + str(l))
host = get_host(self, dpid=dp.id)
for l in host:
    print (" \t\t" + str(l))
    switch_port = l.mac
```

This is how we add flow in the OpenFlow Switch

```
def add_flow(self, datapath, priority, match, actions):
    """
    Add flow entry
    :param datapath:
    :param priority:
    :param match:
    :param actions:
    """
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # construct flow_mod message and send it.
    inst = [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                         actions)]
    mod = parser.OFPFlowMod(datapath=datapath,
                            flags=ofproto.OFPFC_ADD,
                            priority=priority,
                            match=match, instructions=inst)
    datapath.send_msg(mod)
```

h2 iperf -c h1