Softwarized Network Documentation

Francesco Pietri, Stefan Gore

September 7, 2024

1 PROJECT OVERVIEW

This project leverages Software-Defined Networking (SDN) to optimize network performance by exploiting the capabilities of a centralized controller. The controller enables efficient routing and real-time detection, analysis, and mitigation of network issues, such as switch or link failures and host migrations.

When hosts connect to the network and send an **ARP request** to the switch they are attached to, this triggers the ARP protocol. The ARP protocol resolves the association between MAC addresses and IP addresses, populating the ARP tables that facilitate network communication.

Simultaneously, we collect various statistics, such as latency and bandwidth, thanks to some threads to periodically send requests to gather statistics on both flow and port activities. This enables the controller to monitor the state of the network in real-time. These metrics are especially useful when determining the optimal path using the Depth-First Search (DFS) algorithm.

In addition to performance metrics, we also gather topology-related information. This data is crucial in cases of host migration, or when handling link or switch failures. All collected information, including performance stats and topology data, is stored locally in dictionaries for efficient retrieval and use.

To handle network failures, such as when a host migrates from one switch-port to another, we implement a mechanism that detects and manages these changes. In the case of a link failure, where a network link is no longer operational, we dynamically reroute traffic to the best available path based on the current topology. Similarly, switch failures are addressed by rerouting traffic to maintain network functionality.

The controller have one assumption: there is always a possible route from one host to another. In addition the switch and host must connect rapidly otherwise may arise different exception.

2 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.

2.1 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.

2.2 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.

3 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
```

3.1 ROUTING

Given a network topology created in Mininet, we initiate communication between two hosts. In this case, `h1` acts as the client, and `h4` as the server. When `h1` begins communication with `h4`, the ARP protocol is initiated (see the ARP section for details). During the ARP process, we also initiate the routing procedure.

The routing process, implemented in the `routing` function, consists of the following steps:

- The first step involves determining the optimal path between the switches to which the two hosts are connected.
- The second step installs the optimal path across all the relevant switches in the network.

`get_optimal_path`: This function calculates the optimal path between the source and destination switches based on the latency information. It uses a Depth-First Search (DFS) algorithm to find the path with the lowest latency. The resulting path is a list of switches, such as `[1, 2, 4]`, representing the switches that must be traversed.

`Example` function call: `get_optimal_path(self.latency_dict, 1, 3, 'arp')` where `1` and `3` represent the source and destination switches, respectively.

The function that finds the optimal route relies on the `latency_dict` data structure, which, in turn, depends on the `data_map` structure to gather the necessary latency information.

3.1.1 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                                    )
```

4 TOPOLOGY DISCOVERY

The controller requires knowledge of the network topology to fulfill its functions. To gather this information, we use various data structures and API calls, such as `data_map` and `hosts`. The process of studying the topology can be divided into two phases: initially, we work with a static topology, and later, the topology dynamically adapts when simulating host migration or link failures, allowing the controller to learn changes in real-time.

To populate the `hosts` dictionary,important for the topolgy, we employ different methods:

Packet-in Analysis: When a packet arrives at a switch and is forwarded to the controller, we store the source MAC address along with the switch ID and the port that received the packet. This can be done as follows:

```
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)
    ...
```

Host Addition Event: Another method is to leverage the following event and decorator, which trigger the corresponding function when a new host is detected:

```
from ryu.topology import event
@set_ev_cls(event.EventHostAdd)
def _event_host_add_handler(self, ev):
```

The `data_map` data structure is another crucial element for determining the network topology. It maintains the associations between switches and the corresponding ports used to connect them. This structure is populated when custom packets arrive at the controller.

In the dynamic topology phase, the `data_map` structure will update in response to link or switch failures. This is achieved through the use of event handlers that monitor state changes in the network. For example, we can detect port state changes as follows:

```
from ryu.controller import ofp_event
@set_ev_cls(ofp_event.EventOFPPortStateChange, MAIN_DISPATCHER)
def port_state_change_handler(self, ev):
```

This event is triggered when a link goes down, such as when using the Mininet CLI command `link s1 s2 down` to simulate a link failure.

Alternatively, the same event can be handled using the following method:

```
from ryu.controller import dpset
@set_ev_cls(dpset.EventPortModify, MAIN_DISPATCHER)
def port_modify_handler(self, ev):
```

Both approaches ensure that the `data_map` is updated dynamically when network changes occur, enabling the controller to maintain an accurate view of the current topology.

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_host 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.

5 PACKET HANDLER

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

The function is triggered when a packet arrives at a switch, fails to match any flow entry, and is forwarded to the controller for processing. The function handles different packet types as follows:

- **0x07c3**: This is a custom packet type used specifically for latency measurement.
- **LLDP (Link Layer Discovery Protocol)**: These packets are typically used for topology discovery, helping to identify switches and routers in the network. However, in this project, LLDP packets are filtered out as an alternative method for topology discovery is employed.
- **IPv6**: These packets are filtered out due to the added complexity they introduce, which is unnecessary for the scope of this project.
- **ARP**: See the dedicated section for handling ARP packets.
- **IPv4**: See the dedicated section for handling IPv4 packets.
- **ICMP (Internet Control Message Protocol)**: ICMP packets, often used for network diagnostics and troubleshooting (e.g., `ping` and `traceroute`), are not processed by the controller.

---

6 Host Migration

Host migration refers to the process where a host's connection is moved from one switch to another. During this migration, the OpenFlow tables must be updated accordingly to ensure accurate routing. A straightforward approach is to remove all flow entries from any switch that contains a match rule with the migrating host's IP address.

The host migration process follows these steps:

1. The process begins when the decorator `@set_ev_cls(ofp_event.EventOFPPortStateChange, MAIN_DISPATCHER)` triggers the controller due to a link failure.
2. The event provides the affected switch and port information. If the port affected is a host-facing port (i.e., a port connecting a host to the switch) and the port was deleted, it indicates the beginning of the host migration.
3. We must update relevant data structures, such as `self.hosts` and `self.already_routed`, to reflect the changed topology.
4. Following this, all flow entries related to the migrating host's IP and MAC address must be deleted from every switch that previously had routes for this host.
5. Once the host reconnects to a different switch, it will trigger the ARP protocol again, which initiates a new routing process.

---

7 LINK FAILURE

The link failure process is triggered when a connection between two switches goes down. In Mininet, this can be simulated using the command `link s1 s2 down`. A key assumption in this process is that an alternative path exists between the hosts when one link fails.

The link failure process consists of the following steps:

1. The process begins when the event handler `@set_ev_cls(ofp_event.EventOFPPortStateChange, MAIN_DISPATCHER)` is triggered, detecting changes in the state of a port or switch.
2. We analyze the event to determine if the affected port and switch are part of a switch-to-switch connection. If the link has been deleted, we classify this as a link failure.
3. The first task is to update the data structures that store topology information. Specifically, modifications are made to the `self.data_map`, which maintains switch-to-switch connectivity and port associations.
4. Next, we identify the flows in the affected switches. We retrieve information regarding the source IP (`ip_src`) and destination IP (`ip_dst`) associated with these flows.
5. Finally, we invoke the rerouting function to establish a new path between the affected hosts, ensuring uninterrupted communication despite the link failure.

This process ensures that the network remains resilient to link failures by dynamically updating the topology and re-establishing routes in response to changes.

---

8 SWITCH FAILURE

A switch failure is simulated using the Mininet CLI command `switch 'name_switch' stop`. The main assumption in this process is that alternative paths exist for host communication even if a switch goes down.

The switch failure process involves the following steps:

1. The process begins when the event handler `@set_ev_cls(event.EventSwitchLeave)` is triggered:

   ```
   @set_ev_cls(event.EventSwitchLeave)
   def switch_leave_handler(self, ev):
   ```

   From the event variable `ev`, we can retrieve crucial information about which switch has failed.
2. Upon switch failure, all flow entries in the switch's tables are erased or corrupted.
3. We then retrieve all communication flows between hosts that were affected by the switch failure from the `path_cost_flow` dictionary (see the documentation for details). With this information, we can proceed with rerouting the affected communications.
4. Before performing rerouting, we need to update all data structures that contain topology information, such as `data_map`, which the rerouting function relies on.
5. Finally, we perform the rerouting to establish new paths for the affected communications, ensuring that host communication continues uninterrupted despite the switch failure.

This process ensures that the network can handle switch failures effectively by updating the topology and rerouting traffic as needed.

9 DATA STRUCTURE

data_map

The `data_map` is a dictionary structure that stores network information such as input port number, bandwidth, and latency for packet transfers between switches/datapaths.

Structure:

- **First key:** The ID of the switch/datapath that received the packet (`dpid_rec`).
- **Second key:** The ID of the switch/datapath that sent the packet (`dpid_sent`).
- **Attributes:**
  - `in_port`: The input port number through which the packet was received.
  - `bw`: A list that can store bandwidth data.
  - `latencyRTT`: A list of dictionaries that records the timestamp and round-trip latency in milliseconds.

Example Data Map:

```
self.data_map = {
    1: {                        # dpid_rec (Receiver Switch ID)
        2212: {                 # dpid_sent (Sender Switch ID)
            'in_port': 21,      # Input port number
            'bw': [],           # Bandwidth data (currently empty)
            'latencyRTT': [     # Round-Trip Latency Data
                {
                    'timestamp': 1627356123.123,  # Time of measurement
                    'value': 15.6                 # RTT latency in milliseconds
                },
                {
                    'timestamp': 1627356187.456,  # Another timestamp
                    'value': 16.2                 # Another RTT latency value
                }
            ]
        }
    }
}
```

Explanation of Example:

- `1`: Represents the ID of the switch that received the packet (`dpid_rec`).
- `2212`: Represents the ID of the switch that sent the packet (`dpid_sent`).
- `in_port`: The packet arrived at input port 21.
- `latencyRTT`: Stores two round-trip latency measurements: 15.6 ms and 16.2 ms at different timestamps.

---

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.

---

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

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'
]
```

---

chosen_path_per_flow

The `chosen_path_per_flow` data structure stores the optimal path and its associated cost (e.g., latency) between a pair of source and destination IP addresses. This is useful for routing decisions in networks.

Structure

The structure is a nested dictionary:

- Outer dictionary key: **Source IP address**.
- Inner dictionary key: **Destination IP address**.
- Value: A tuple containing:
  - **Optimal path**: A list of nodes.
  - **Cost**: The total cost (latency) of the path.

Example

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

9.1 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
```