Report

COL334 - Assignment 3

JASKARAN SINGH BHALLA (2021TT11139) **BASIL LABIB (2021TT11175)**

Overview

We are using **OpenFlow version 1.0** for this assignment. We implemented the assignment on a standard machine that BaadalVM provided.

Baadal VM specifications

OS	Ubuntu 20.04 LTS
RAM	4 GB
vCPU	4

Software version:

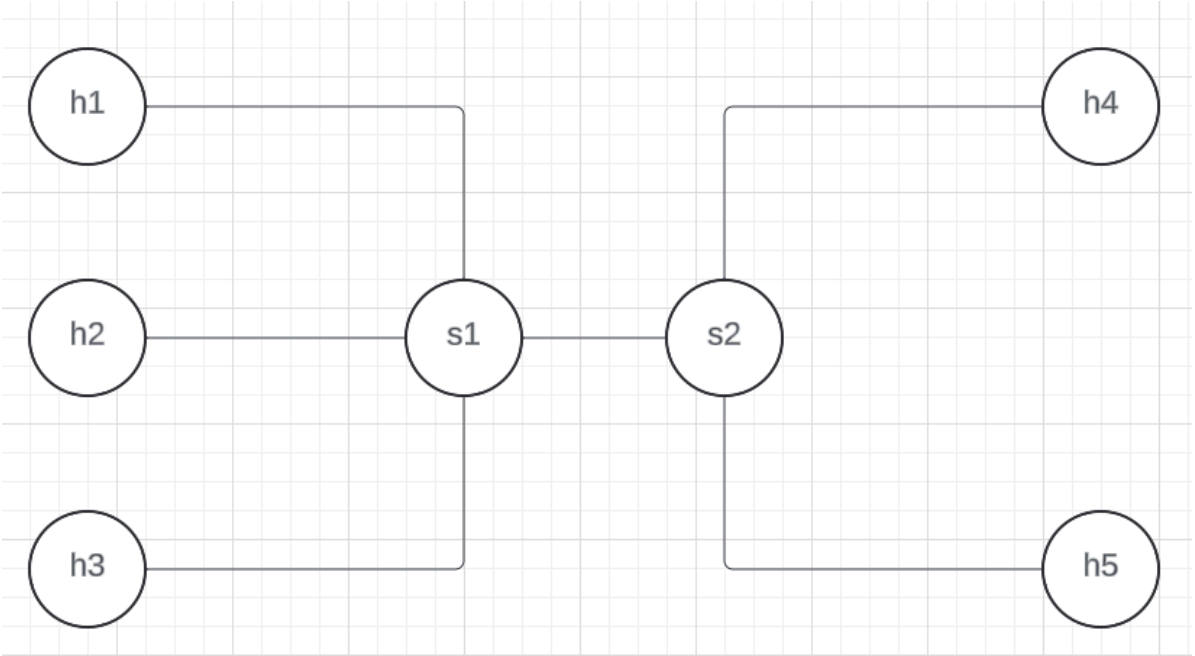
Python	3.9.2
Ryu controller	4.34
Mininet	2.3.4b
eventlet	0.30.2

In addition to this, we are using the pprintpp package for cleaner debugging.

Part 1: Controller Hub and Learning Switch

Network topology

We present the results of running our hub and learning switch on the given topology of 5 hosts and 2 switches as shown below.



Controller Hub

Overview

We have implemented a basic hub controller using the Ryu SDN (Software Defined Networking) framework. A hub forwards incoming packets to all ports except the one it was received on, which is mimicked in this code by the OFPP_FLOOD mechanism. This controller works with OpenFlow version 1.0 and listens for PacketIn events triggered by switches under the controller's domain.

Implementation

Controller Initialisation

The hub controller inherits from ryu.base.app_manager.RyuApp, and does not have any extra attributes. It simply calls the constructor of the base class.

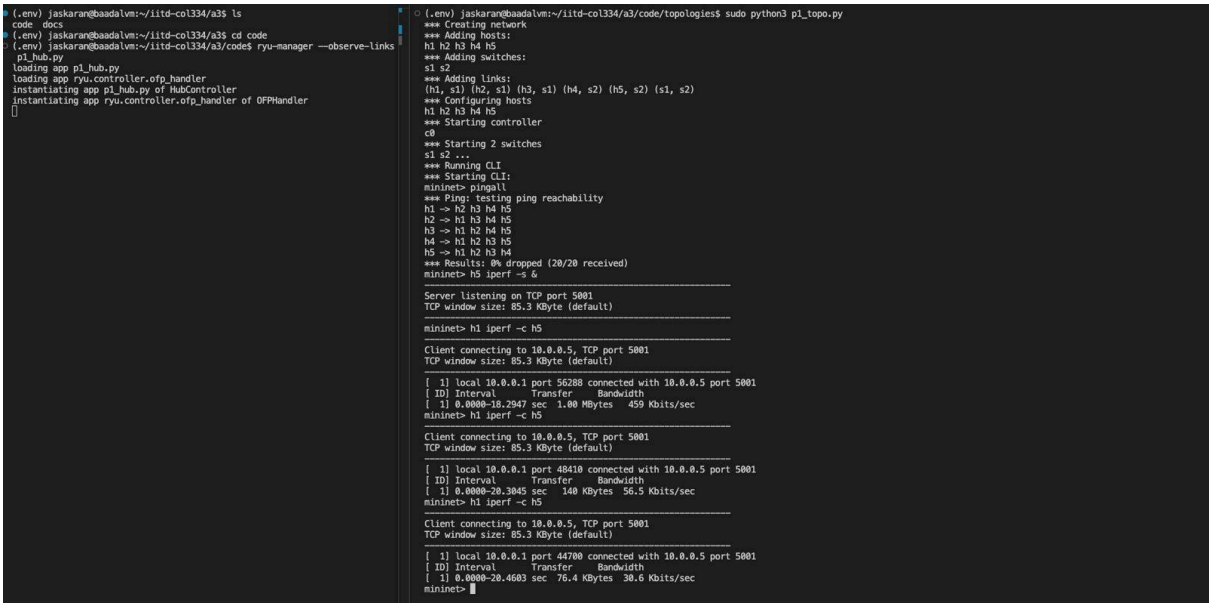
Packet Handling

As the mechanism for flooding is implemented using the OFPP_FLOOD mechanism, we simply extract the data packet, the datapath, and the OpenFlow protocol used for that packet. We then call

the send_packet function, which sets the action on the OFPPacketOut event and sends the msg (the data packet) on that datapath.

Results

- Pingall and Iperf



Discussion

The first command, h5 iperf -s & designates h5 to run in server mode in the background. The second command h1 iperf -c h5 designates h1 as the client (the -c flag) and h5 as the host for the iperf test.

Moreover, we can also see the screenshot for the pingall command and see no packet loss in the topology.

Learning Switch

Overview

The controller hub architecture is very inefficient as it always floods the entire network. The learning switch is a smarter alternative as it keeps a record of which port to send a packet on to reach a certain mac_address. The basic idea is to record the port from which a packet came and to send any packet designated for that host on that port in the future. If no record exists, then the packet is simply flooded.

Implementation

The major changes to the learning switch is essentially maintaining a lookup mapping from **mac_address** to the switch **port**. This is done using the **self.mac_to_port** dictionary of type **Dict [datapath.id -> Dict [mac_addr -> port]]**

The dictionary is updated in these lines for each incoming packet, which is not an LLDP packet.

```
    self.mac_to_port.setdefault(datapath.id, {})
    self.mac_to_port[datapath.id][src_mac] = msg.in_port
```

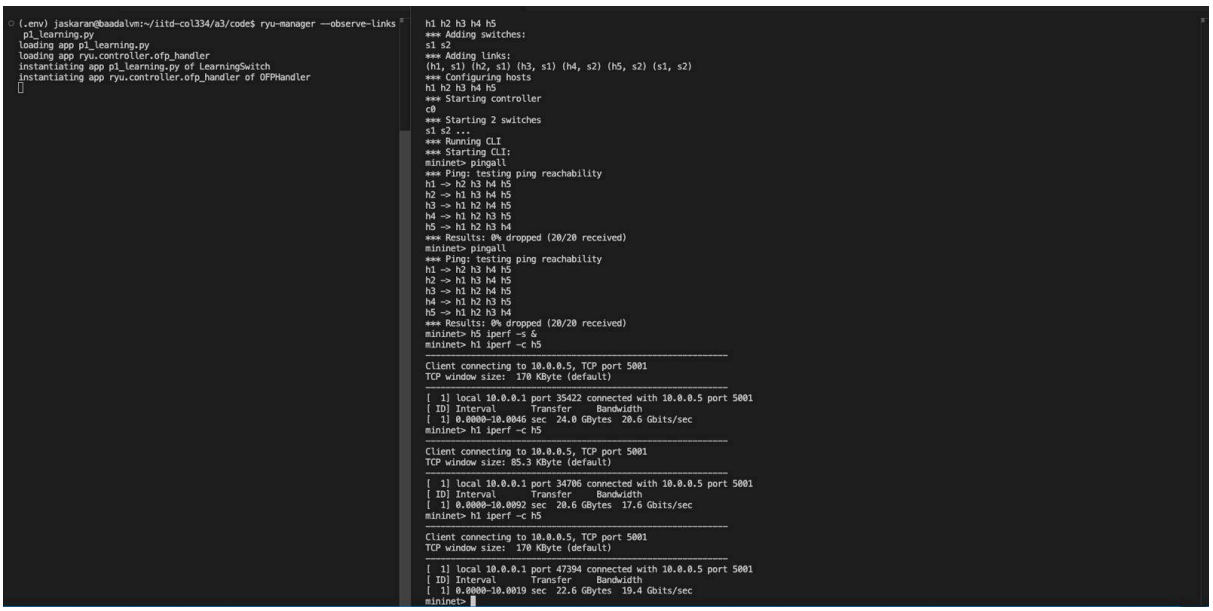
We also implement the **add_flow** function with the following signature:

```
def add_flow(self, datapath, in_port, out_port, src, dst):
```

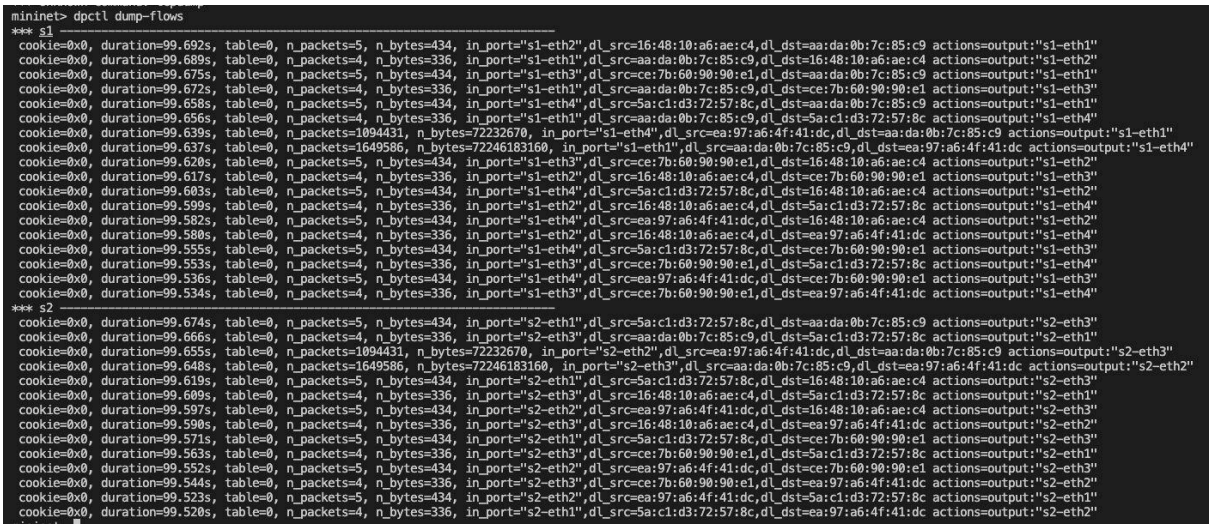
which matches for the **in_port**, **src**, and **dst** of a packet and then sends the packets through the **out_port**. This is implemented using the **datapath.ofproto_parser.OFPMatch** and **ofp_parser.OFPFlowMod** functions.

Results

- Pingall and Iperf



- Tcpdump flows



This image shows the learned rules for routing for both the switches in this topology.

Discussion

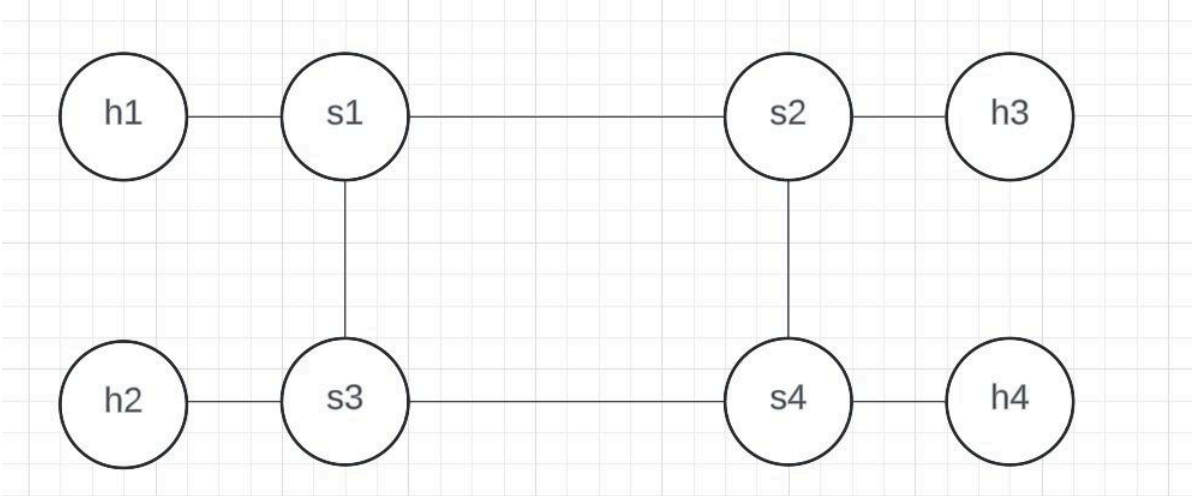
As we can see, there is a huge improvement in the network performance due to the removal of the redundant packet flooding.

Part 2: Spanning Tree

Overview

In any network topology, if there is a cycle, a packet may end up getting into a loop while being routed by the switches. One solution to this issue is to construct a spanning tree from the given graph (with potentially some number of cycles). In this part of the assignment, we construct such a spanning tree for the given topology once using Prim's algorithm to block certain ports on certain switches during the routing of packets.

We use the following network topology for preparing this report:



Assumptions

1. We are **not** constructing the minimum spanning tree since the graph is unweighted.
2. All the switches in the topology are connected. In other words, the graph is connected, and we do not have any isolated switches.
3. The graph is undirected.

Event Handling

In this part, we compute the spanning tree for every **event.EventSwitchEnter** event is fired. This event is fired when a new switch is added to the network. Moreover, the usual **packet-in** and **packet-out** events are handled as previously done.

Implementation

Controller Initialisation

In addition to the **mac_to_port** dict, we also maintain a **graph** dictionary (an adjacency list representation of the network topology), a **spanning_tree** list (list of edge tuples), and a set of all blocked ports in **blocked_ports**.

Topology Construction

We get the switches first using the **ryu.topology.api.get_switch** function. Then, we loop over all links in the topology and return the links.

Once we have the links, then we construct the graph.

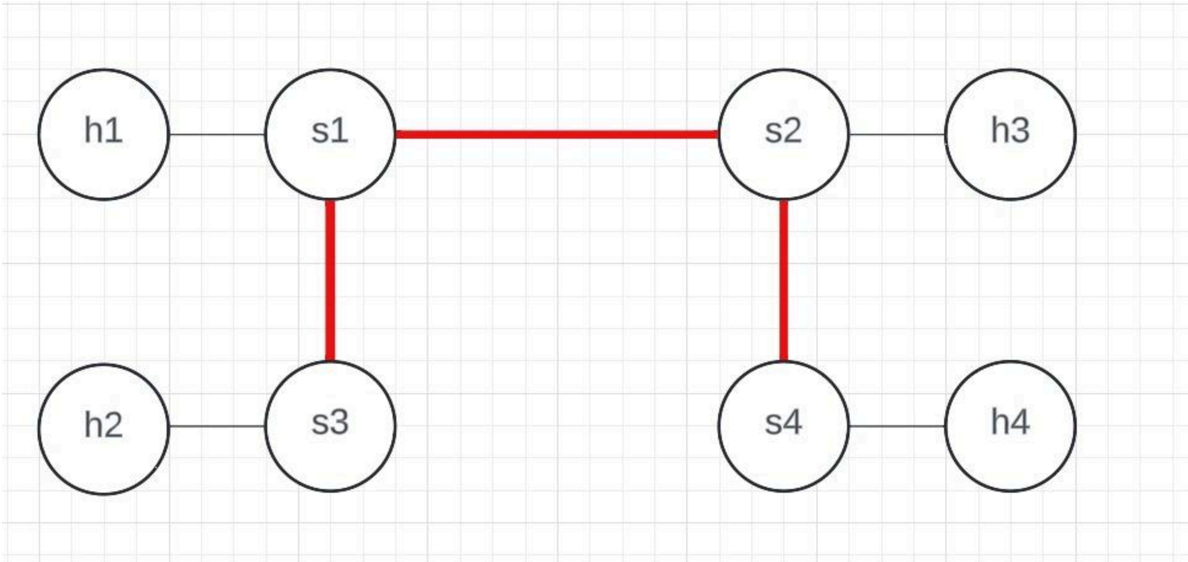
Spanning Tree Construction

We follow Prim's algorithm for constructing the spanning tree. The pseudocode is given below:

```
  Algorithm:
1. Initialise an empty set 'visited' to keep track of visited vertices.
2. Set'start' as the ID of the first switch in the 'switches' list.
3. Add 'start' to the 'visited' set.
4. Initialise an empty list 'edges' to store the edges of the minimum spanning tree.
5. For each neighbour of'  'start' in the 'graph' dictionary:
 a. If the neighbour is not in the 'visited' set, add the edge (start, neighbour) to 'edges'.
6. While 'edges' is not empty:
  a. Sort 'edges' based on the second element of each edge.
  b. Pop the first edge from 'edges' and assign it to 'next_edge'.
  c. Extract the source and destination vertices from 'next_edge'.
  d. If the destination vertex is not in the 'visited' set:
   i. Add the 'next_edge' to the 'spanning_tree' list.
   ii. Add the destination vertex to the 'visited' set.
   iii. For each neighbour of the destination vertex in the 'graph' dictionary:
    - If the neighbour is not in the 'visited' set, add the edge (destination, neighbour) to 'edges'.
7. Return None.
```

Finally, based on the spanning tree edge tuple list that we computed, we construct the **blocked_ports** set, which indicates which ports to block on a given switch during routing.

The following spanning tree was constructed:

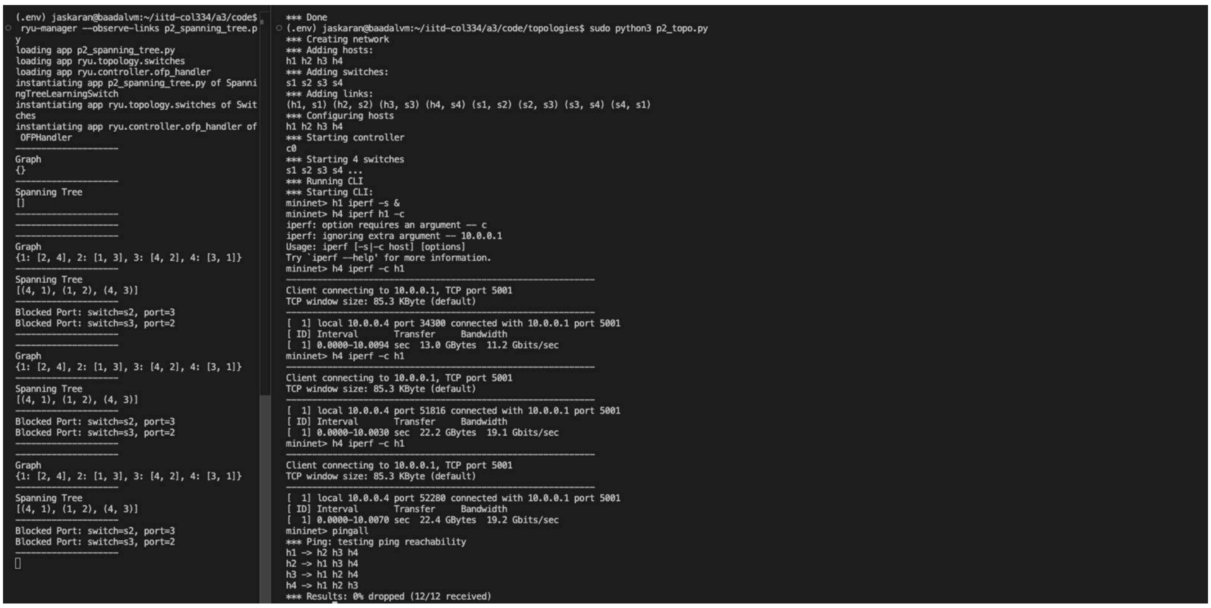


Packet Handling

The packet handling happens analogously to the learning switch with one difference. Whenever we send a packet out in flood mode, we exclude the blocked ports, the message in port, and the local port from the list of all ports of the current switch.

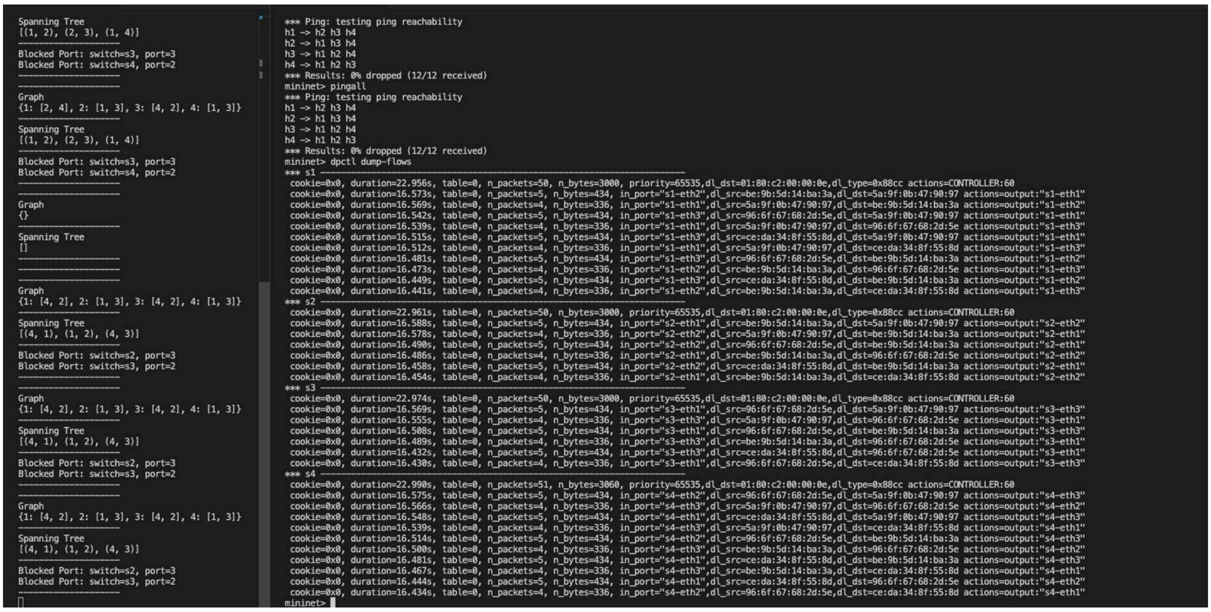
Results

- Pingall and Iperf



As the switch is a learning switch, we observe a similar level of performance as in the case of the learning switch. Moreover, the graph, the spanning tree, and the blocked ports are observable on the left-hand side.

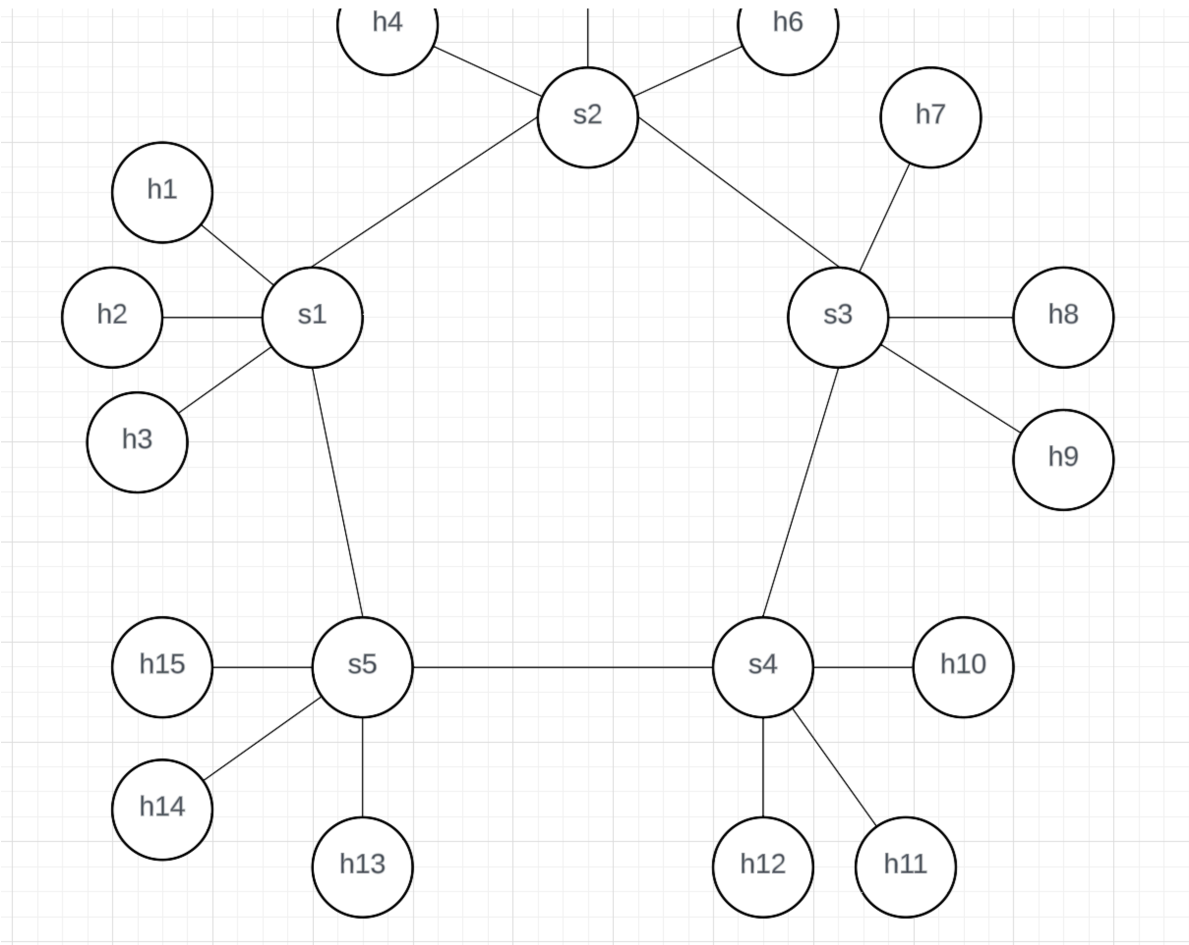
- Tcpdump flows

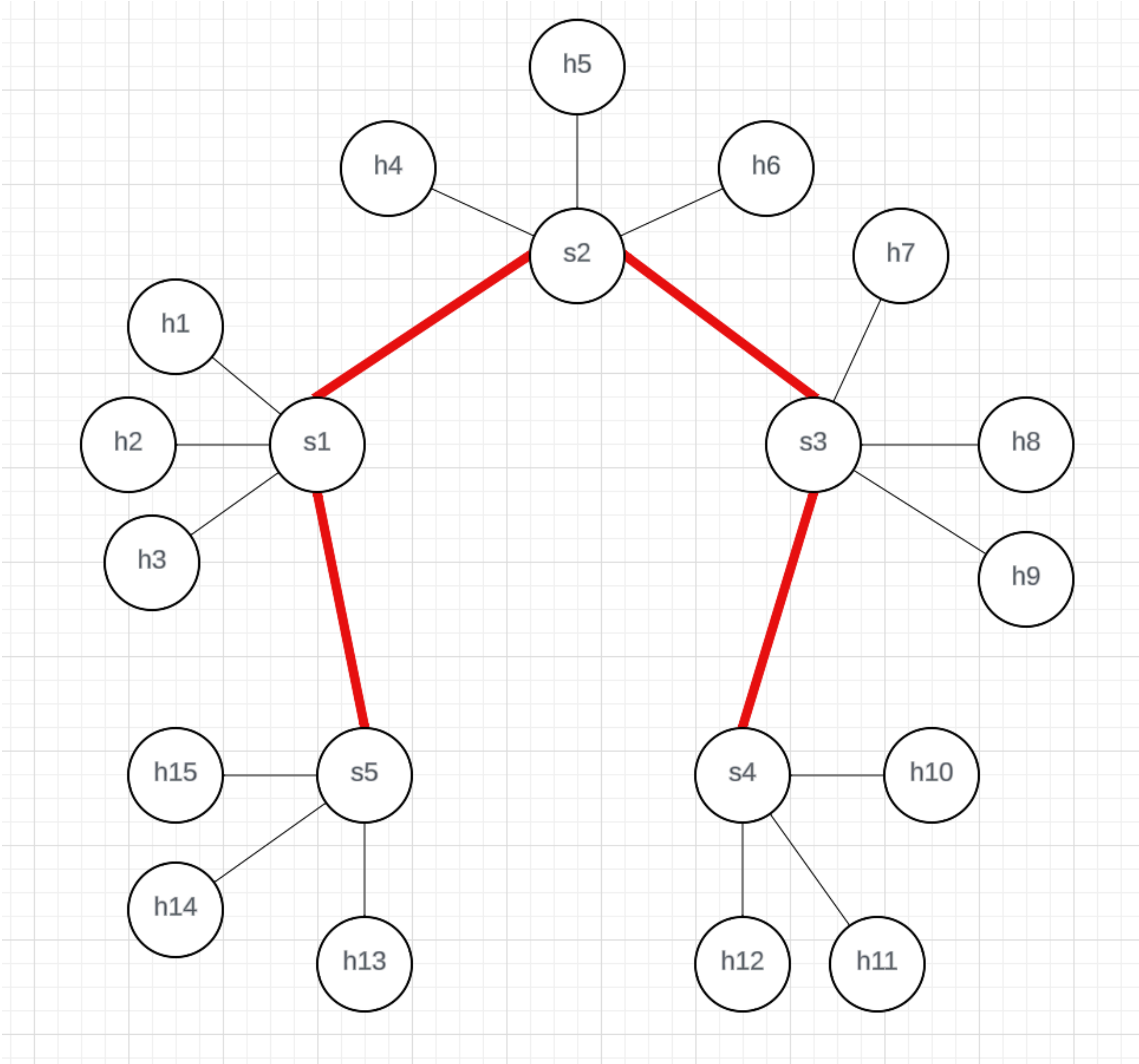


This screenshot shows the rules that all four switches have learnt in this topology.
We also ran the code for a couple of custom topologies

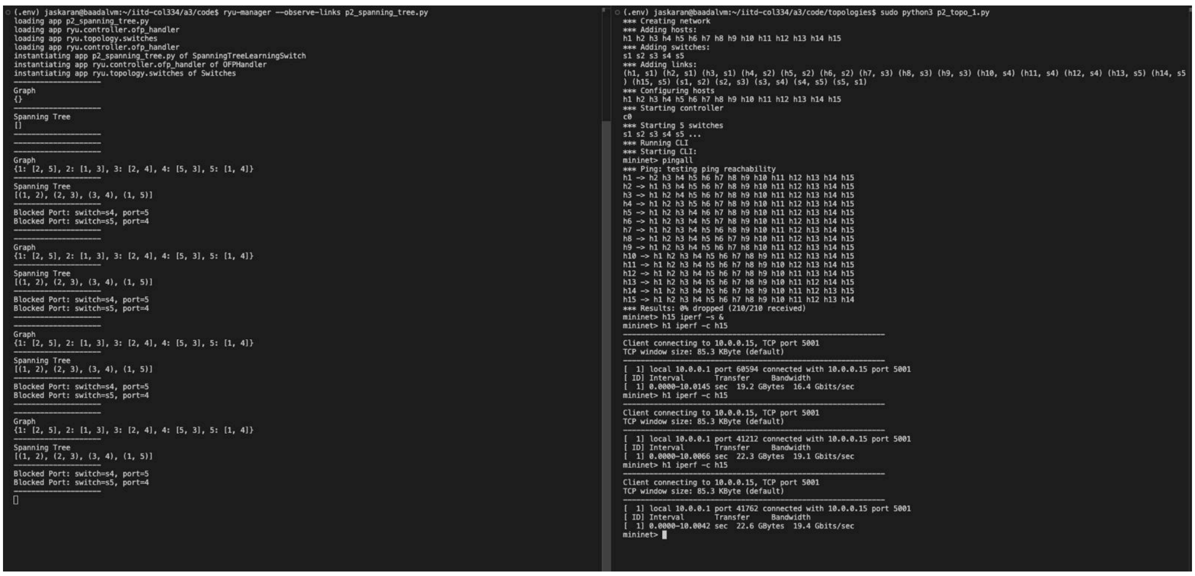
Additional Topologies

p2_topo_1.py

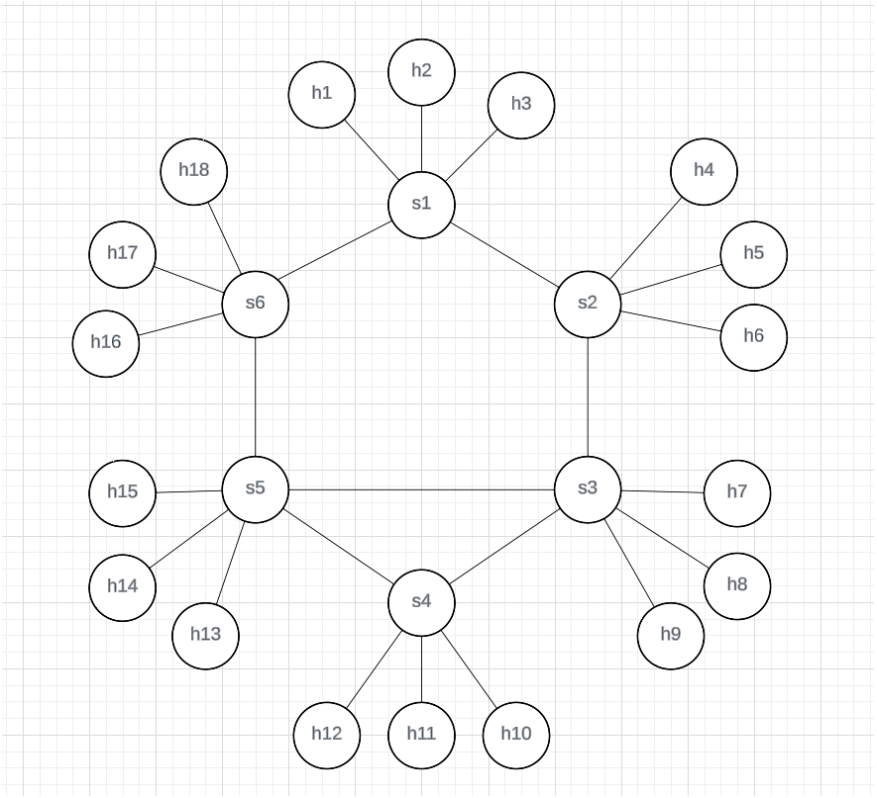


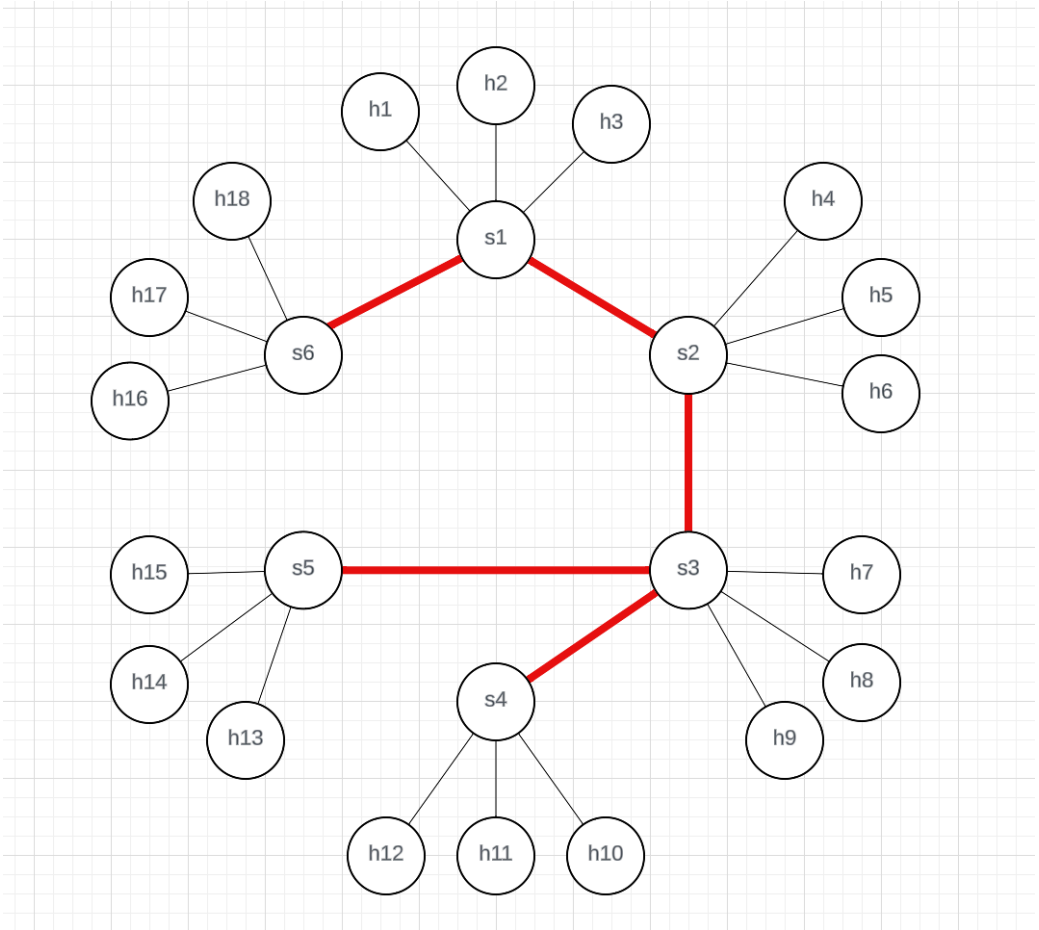


pingall and iperf

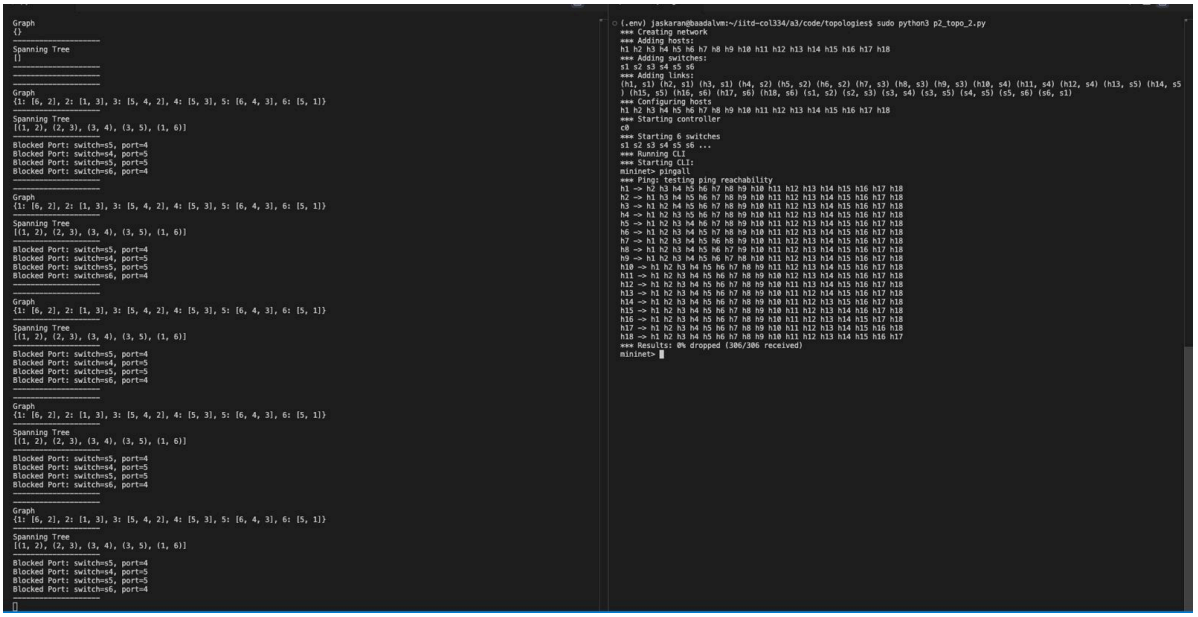


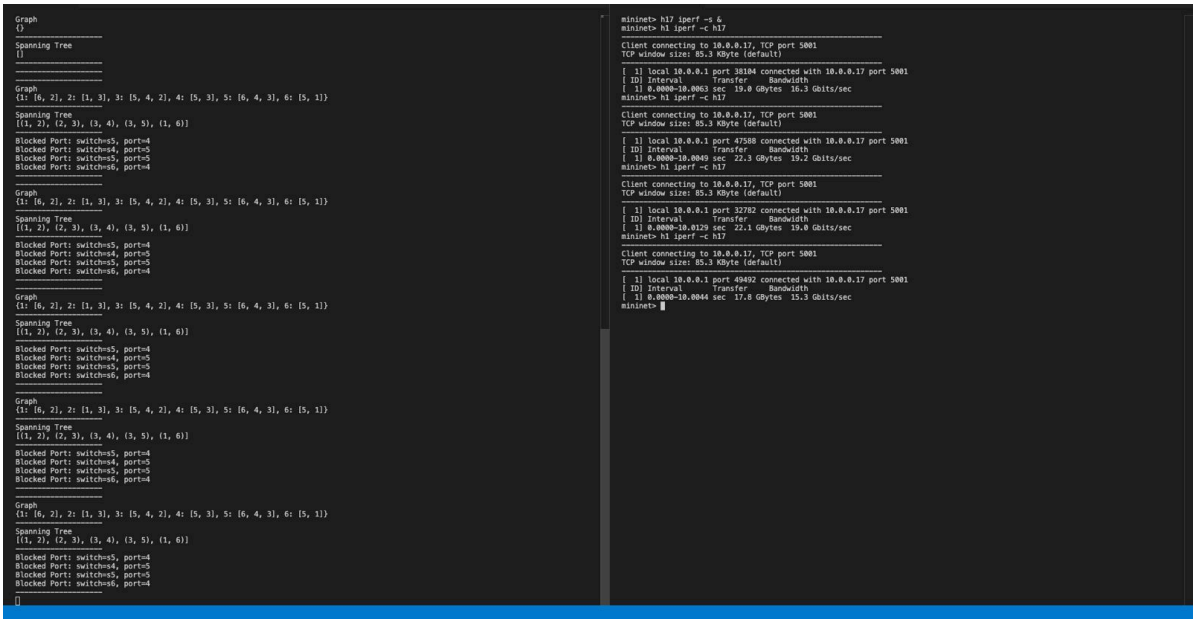
p2_topo_2.py


h1
h2
h3
h18
s1
h4
h17
h5
s6
s2
h16
h6
h15
s5
s3
h7
h14
h8
s4
h13
h9
h12
h11
h10

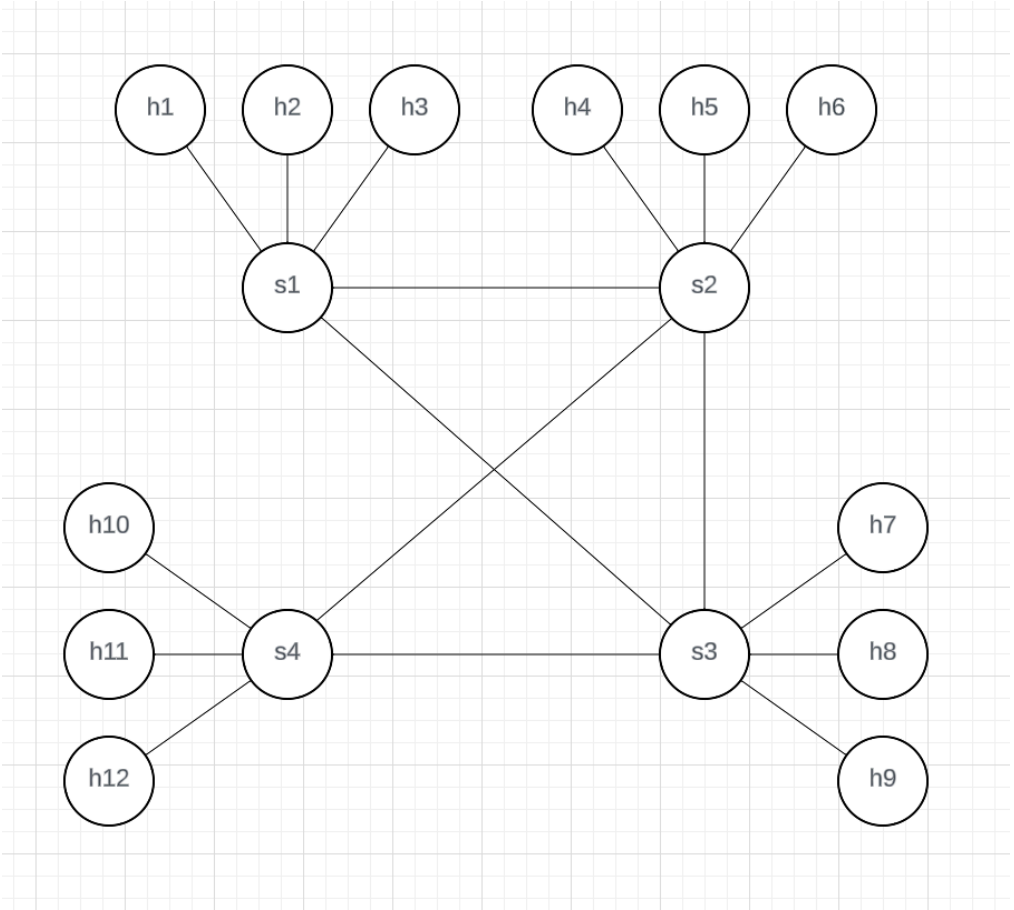

h1
h2
h3
h18
s1
h4
h17
h5
s6
s2
h16
h6
h15
s5
s3
h7
h14
h8
s4
h13
h9
h12
h11
h10

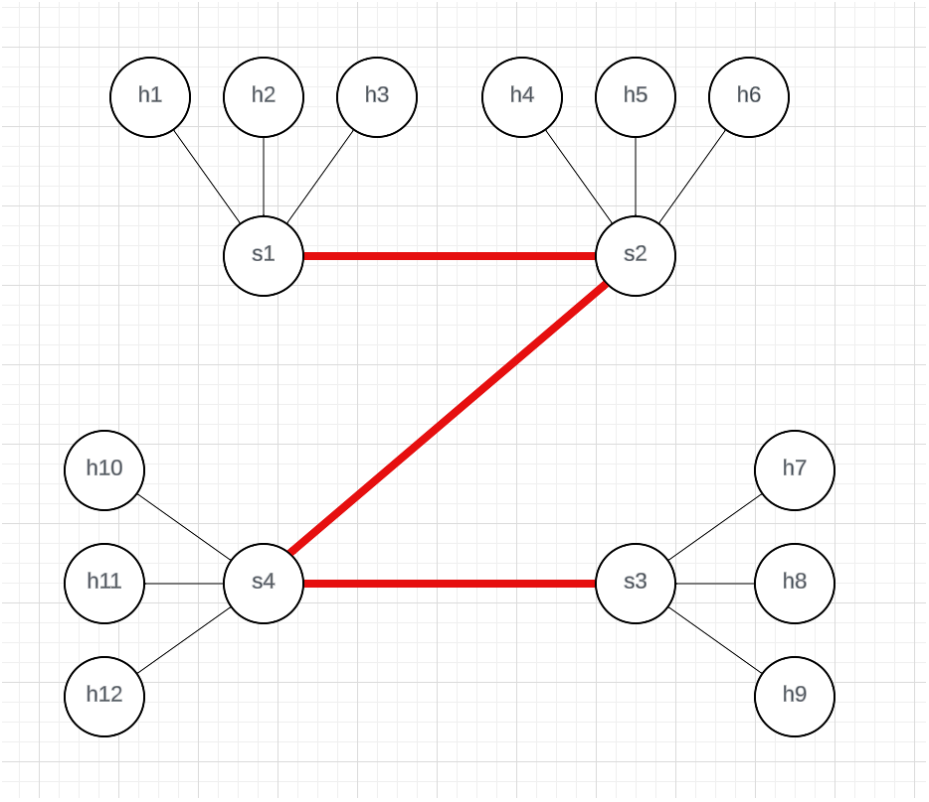
pingall and iperf



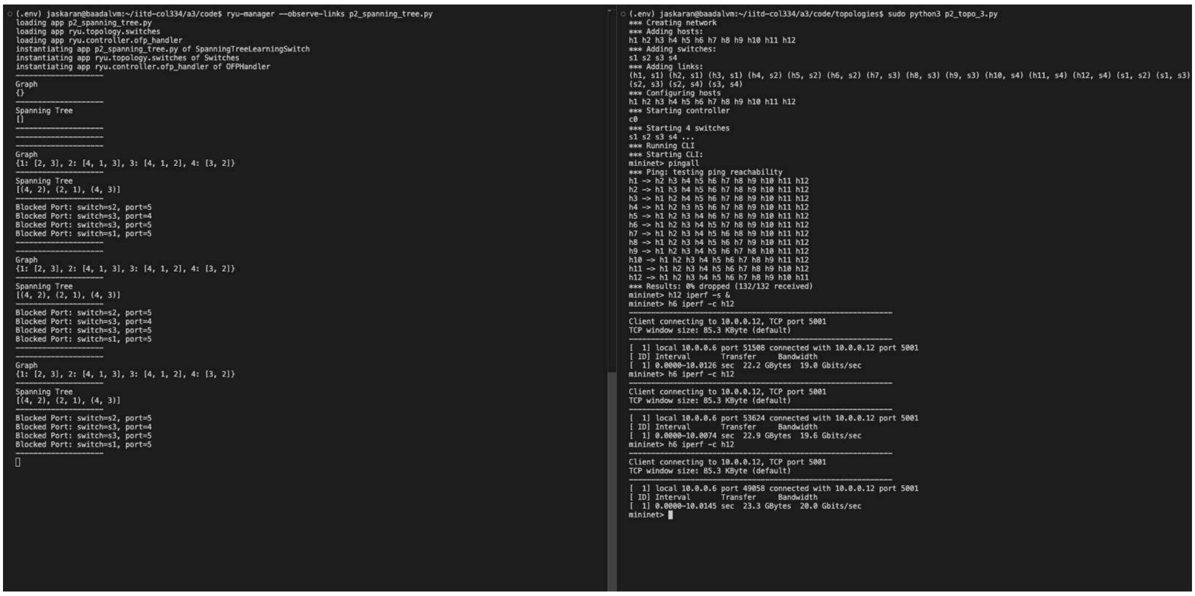


P2_topo_3.py

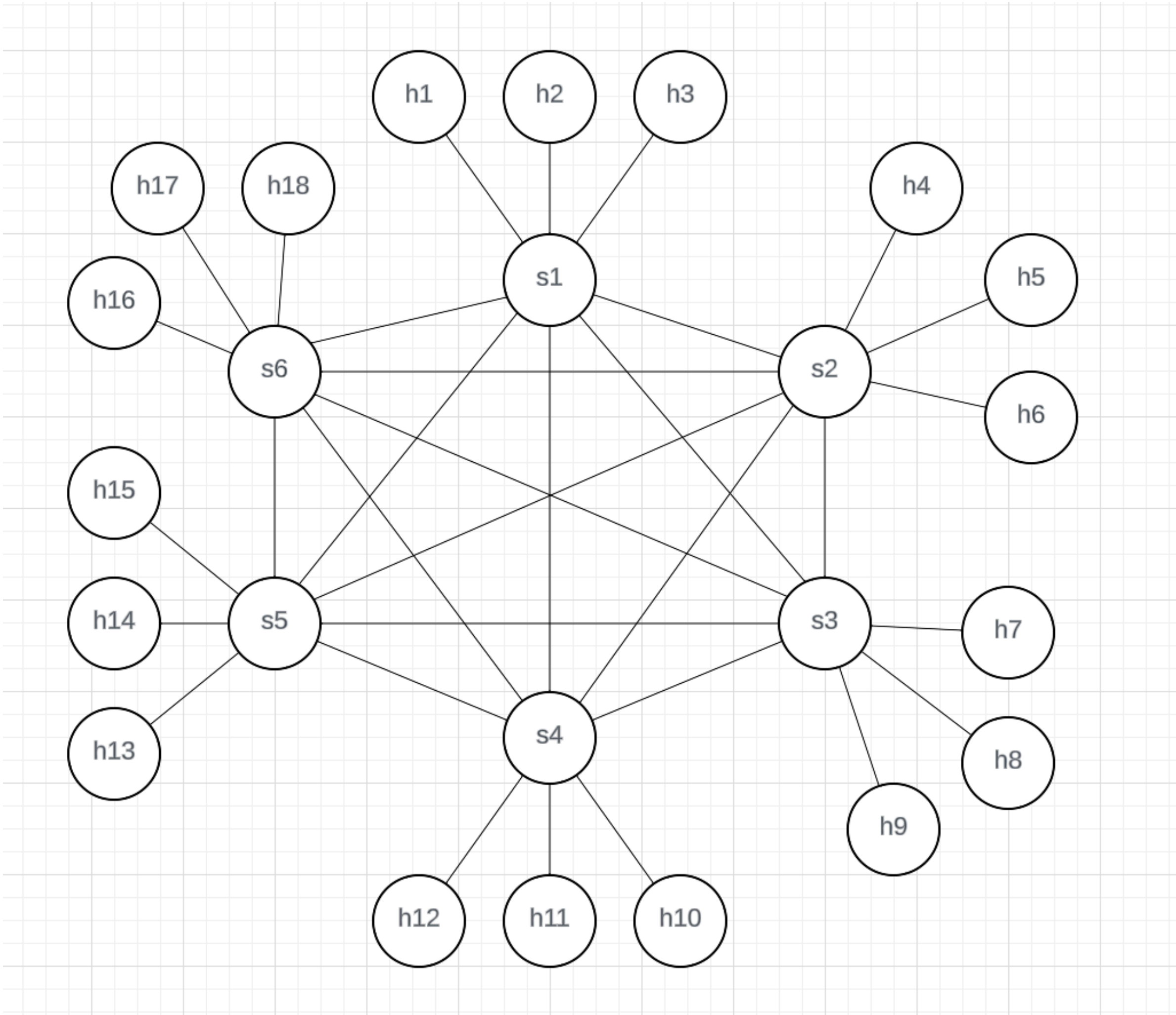

h1
h2
h3
h4
h5
h6
s1
s2
h10
h7
h11
s4
s3
h8
h12
h9

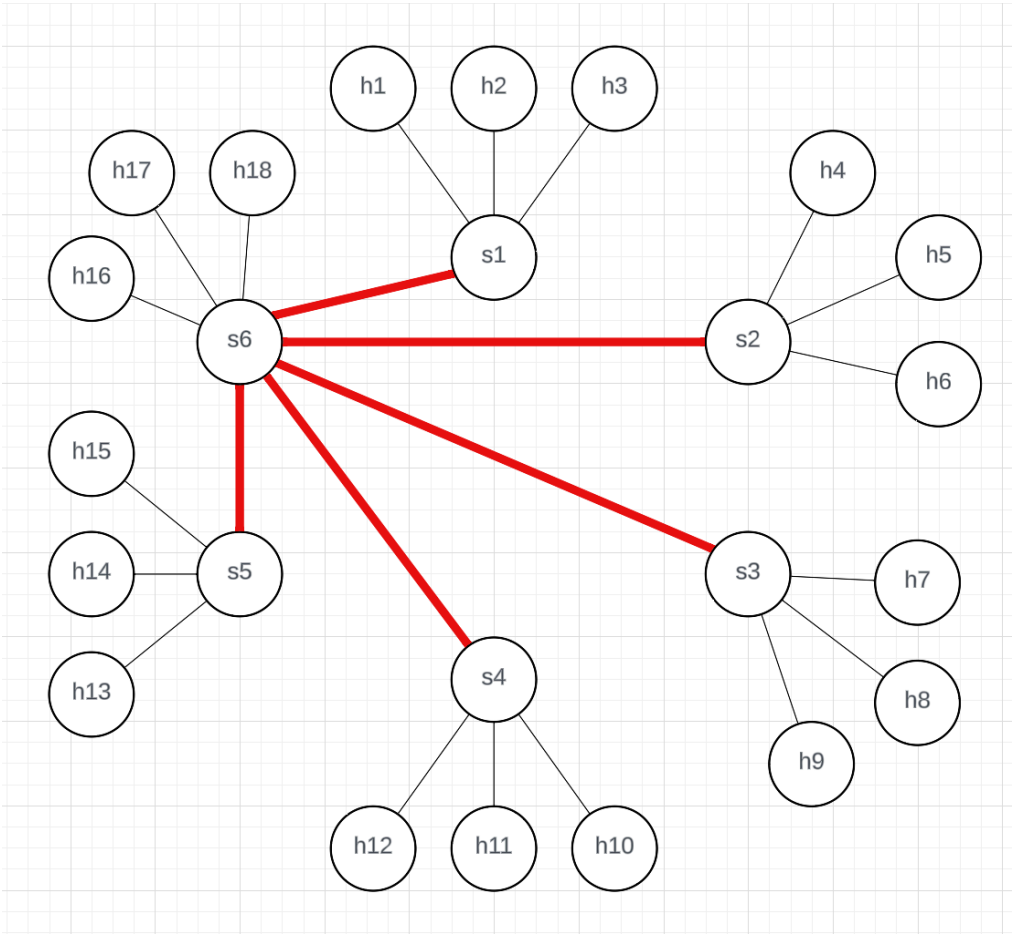

h1
h2
h3
h4
h5
h6
s1
s2
h10
h7
h11
s4
s3
h8
h12
h9

Pingall and iperf

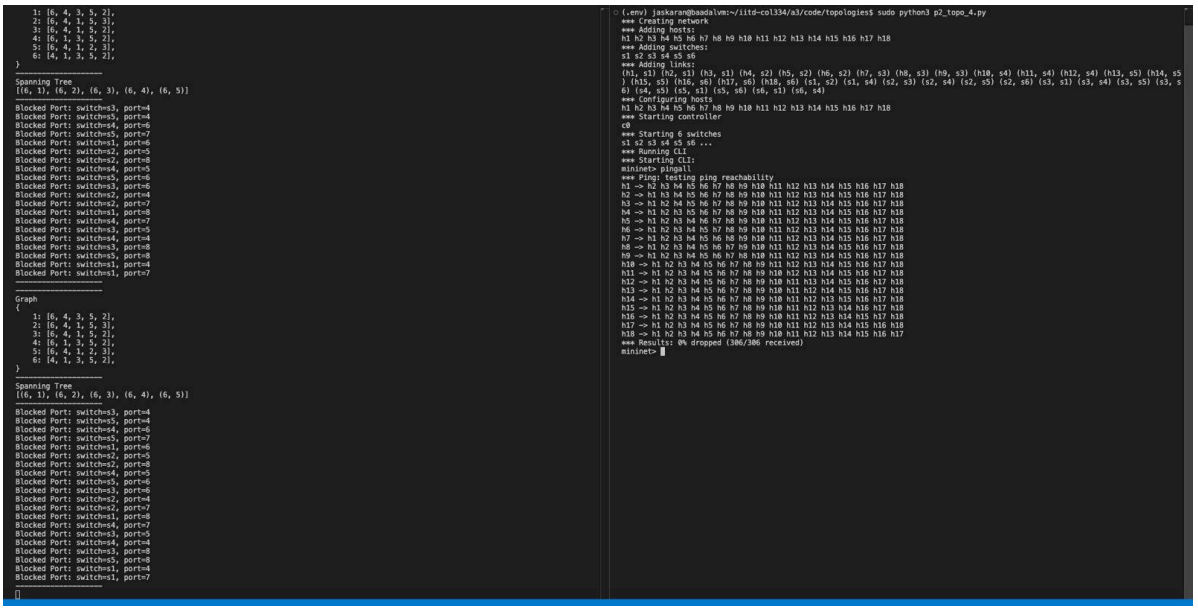


P2_topo_4.py





Pingall and iperf



The spanning tree was successfully constructed for all the network topologies given above.

Part 3: Shortest Path Routing

Overview

In this part of the assignment, we implement shortest path routing at the link layer. We are doing shortest path routing between the switches. We devised several strategies to execute this, such as establishing a shortest path tree, preserving the path, and identifying the port for data transmission at the controller level. We tried them all, but the last one works best. We also calculated the link delays between the switches using LLDP (Link Layer Discovery Protocol) packets. In addition, we have handled ARP, IPv4, and LLDP packets using separate handlers.

Event Handling

When we start the controller, we handle three different RYU events

1. Host Add [EventHostAdd]
2. Switch Enter [EventSwitchEnter]
3. Packet In [EventOFPacketIn]

EventHostAdd

The host add event is responsible for updating the entries in the host whenever a new host joins in. We haven't used this directly in our implementation but have tracked it for debugging purposes.

EventSwitchEnter

This is a key event that governs the program; we utilise it to regulate the implementation whenever a new switch joins the network. This event initiates a thread, which starts sending the LLDP packets to construct the network graph.

EventOFPacketIn

This is a standard event for RYU controllers, which handles the packets entering the controller from the switches

Assumption

For this assignment, we assumed we only wanted to route IPv4 traffic through the shortest path. Initially, when the shortest paths are not processed and we want to discover the host Mac-to-IP relationship using the ARP packets. So the ARP packets are handled using the spanning tree we developed in the previous part of the assignment.

Implementation

Controller Initialisation

Here we initialise a lot of controller class objects to store information to store, process and use data for various sub-implementations. The sub-implementations and their variables include the following:

LLDP (Link Layer Discovery Protocol)

The **lldp_active flag** controls the handling of LLDP packets; it tells the controller whether it should handle LLDP packets or not; if the flag is set, it will process LLDP packets and calculate the link-delay weighted graph, which are crucial for discovering network topology.

The **lldp_duration** defines the time interval for handling LLDP packets, ensuring regular updates on link status. As we have been told, LLDP packets should only be handled for the first few seconds. Currently, the default value of lldp_durations is set to 10.

src_lldp_timestamps stores timestamps of LLDP packets for each source, allowing the controller to track link updates. These are used whenever an LLDP packet is received by a switch to check the timestamp when the packet was published and sent to the switch that sent it.

Topology and MAC Mapping:

The controller maintains mappings of MAC addresses to switch ports in **mac_to_port** for the shortest path routing

The **graph** and **links** attributes are initialised for managing the network's topology, where graph likely represents a logical view of the network and links tracks connections between switches.

The controller also stores switches in the switches list and hosts in the hosts dictionary, enabling dynamic discovery and interaction with the network elements.

Shortest Path:

To support routing based on shortest paths, **w_graph** is initialised to hold the weighted graph of the network. Here it stores the weights in forms of link delays.

The **shortest_path, shortest_path_pred,** and **shortest_path_trees** attributes are used to compute and store shortest paths and their corresponding predecessor nodes.

The **blocked_ports_on_shortest_path_tree** tracks ports blocked along the shortest path to avoid loops or redundant forwarding.

Spanning Tree:

The controller uses a spanning tree algorithm to prevent network loops, with **blocked_ports** tracking ports disabled in the spanning tree.

spanning_tree holds the structure of the tree itself, while **spt_mac_to_port** maps MAC addresses to switch ports for spanning tree forwarding.

LLDP Timer:

A dedicated thread is spawned (**lldp_timer_thread**) to handle the LLDP packet, ensuring the network topology is updated continuously.

switch_threads and **switch_stop_flags** are prepared for thread management to handle the controller's tasks associated with individual switches.

Topology construction

This section of the code implements an LLDP (Link Layer Discovery Protocol) processing system within a Ryu OpenFlow controller. It is used to measure the link delays andconstruct a link delay weighed graph The switch_thread method sends LLDP packets on all non-local ports of the switches. The packets are built by build_lldp_packet, where an Ethernet frame is constructed, encapsulating LLDP information such as the switch's DPID (Data Path Identifier) and port number. The time stamp for creation of each LLDP packet is recorded and stored so that when the packet is received, it can be used to calculate the delay. The send_lldp_packets_on_all_ports method ensures LLDP packets are transmitted across all ports, and the corresponding timestamps are stored.

The lldp_timer method runs for lldp_duration, triggering LLDP processing at intervals defined by lldp_duration. Once an LLDP cycle is completed, **on_lldp_complete** processes the network topology by fetching switch links, constructing a graph of the network, and computing various routing strategies like spanning trees and shortest paths. The graph structure is built by **create_graph**, which maps all switches to their connected neighbors. After the LLDP cycle is over, LLDP packet handling is disabled by setting **lldp_active to false**. The **stop_all_threads** method is responsible for gracefully stopping all ongoing LLDP threads when necessary, ensuring that each thread is terminated properly. Here, in order to optimise time, we create a thread to send and process lldp packets whenever a new switch is added, but each thread waits for 0.5 ms and if a new switch is created, this thread is killed and new thread is created. So only the thread which is created when the last switch enters is the one which builds and sends the lldp packets and **on_lldp_complete** is executed once this thread is completed

Spanning Tree Construction

This part of the code is borrowed from the previous part of the assignment. This section of the code handles the creation of a spanning tree using **Prim's algorithm** to avoid network loops. The **create_spanning_tree** method calls **prims**, which constructs the spanning tree by visiting each node in the network graph and adding edges to form a tree. Once the spanning tree is built, the **get_ports_to_block** method identifies the ports that are not part of the spanning tree. These ports are marked for blocking to ensure only the necessary links are active, preventing loops in the network. Blocked ports are stored in a set for further action. This tree is used to send ARP packets in the network in order to prevent them from getting stuck in the loops before the shortest paths are calculated. This is a design choice that we have made to allow ARP packets to flow even before shortest paths are calculated; we could use shortest path routing here as well. Another thing to be noted here is that we don't add_flow for this section as we want to add flow for shortest path routing only. So ARP packets will also be routed using the shortest path only after the flows are added.

Shortest Path calculations

We have implemented multiple ways to store the shortest paths, which include shortest path trees and shortest path arrays with the src_port, which leads to the shortest path. The src_port and link delay of the shortest path to the end host are stored for each datapath. We are using a modified version of the dijsktra's algorithm to calculate the shortest paths.

Link delay weighted Graph Construction

This graph is constructed as the LLDP packets are received. The handle_lldp_packet_in function processes incoming LLDP packets to calculate the link delay between switches, which is then used

to construct a link-delay weighted graph. When an LLDP packet is received, the function extracts key information such as the source chassis ID (switch ID) and port ID using the packet's Type-Length-Value (TLV) fields. The function checks if the LLDP packet arrived within a valid time frame (less than 10 seconds) by comparing the timestamp of when the LLDP packet was sent to when it was received. If valid, the function calculates the link delay in milliseconds and updates the weighted graph (w_graph) with the delay and source port information, representing the connection between the switches. This helps build a network topology based on real-time link delays.

Packet Handling and Flow management

This section of the code defines the packet_in_handler function, which handles incoming packets based on their underlying protocol. When a packet is received, it is parsed into its respective protocols: ARP, IPv4, IPv6, or LLDP (Link Layer Discovery Protocol). The function first checks if the packet is an LLDP packet by examining its Ethernet type; if LLDP is active, the **handle_lldp_packet_in** function is invoked to process it; otherwise, the packet is ignored. For ARP packets, the code distinguishes between ARP requests and replies, logging relevant details like source and destination MAC and IP addresses, and then handles it with **handle_arp_packet_in**. For IPv4 packets, it filters out multicast and broadcast addresses and processes only unicast packets via **handle_common_packet_in**. Similarly, for IPv6 packets, multicast addresses are ignored, and valid packets are logged. The handler ensures that only relevant packets are processed based on their protocol type while ignoring unnecessary traffic such as multicast or LLDP when it is inactive.

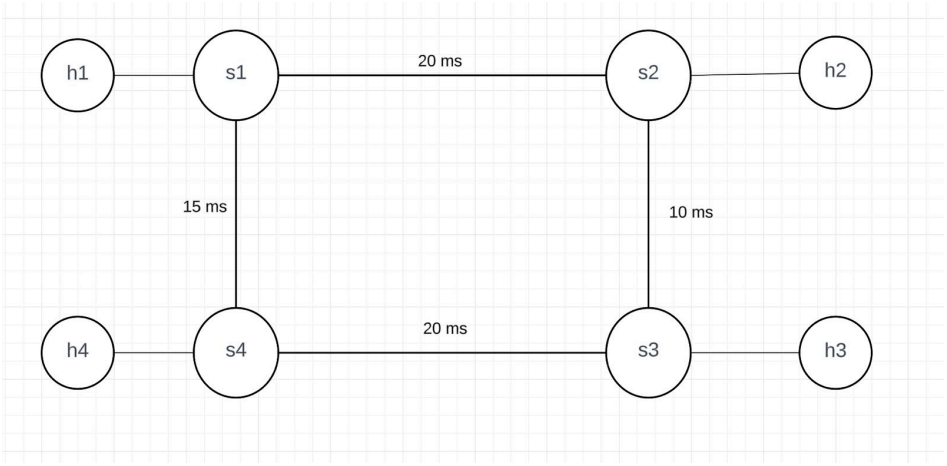
The **handle_arp_packet_in** function processes incoming ARP packets, distinguishing between ARP requests and replies. When an ARP request is received, the function checks if the source IP is already in the ARP table; if not, it adds the source MAC address to the table unless the IP address starts with "169.254." If the destination IP is found in the ARP table, an ARP reply is created and sent back to the requester with the corresponding source MAC address. If the destination IP is not in the table, the function floods the packet through the network using a spanning tree approach, ensuring that it does not send messages back out of the same port they arrived on and avoids blocked ports. For ARP replies, the function updates the ARP table and similarly identifies the correct output port for the response, either by checking the MAC-to-port mapping or by determining the shortest path to the destination.

The handle_common_packet_in function manages the processing of non-ARP packets, primarily focussing on the management of MAC addresses and forwarding packets based on their destination MAC address. It updates the MAC-to-port mapping for the incoming packet's source MAC address and checks whether the destination MAC address is already known. If the destination MAC is in the mapping, it retrieves the corresponding output port; otherwise, it looks up the shortest path to the destination's datapath and determines the output port accordingly. The commented-out section of the code suggests an alternative approach that would involve flooding packets when the destination MAC is unknown. However, this option was not implemented, likely to optimise the network's efficiency and reduce unnecessary traffic by utilising known paths instead of flooding the network with packets.

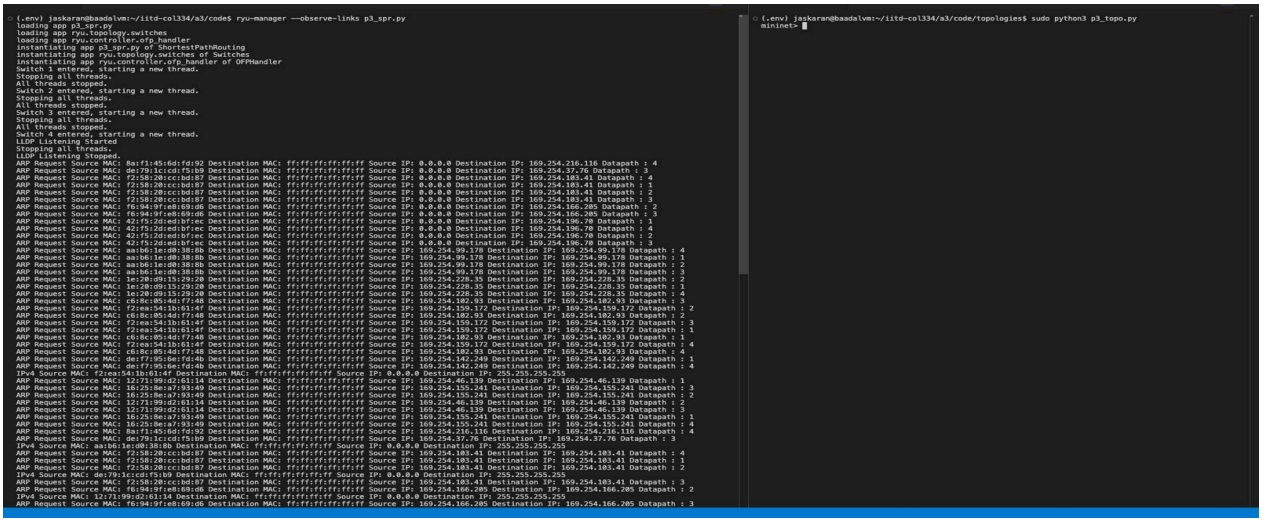
Results

p3_topo.py

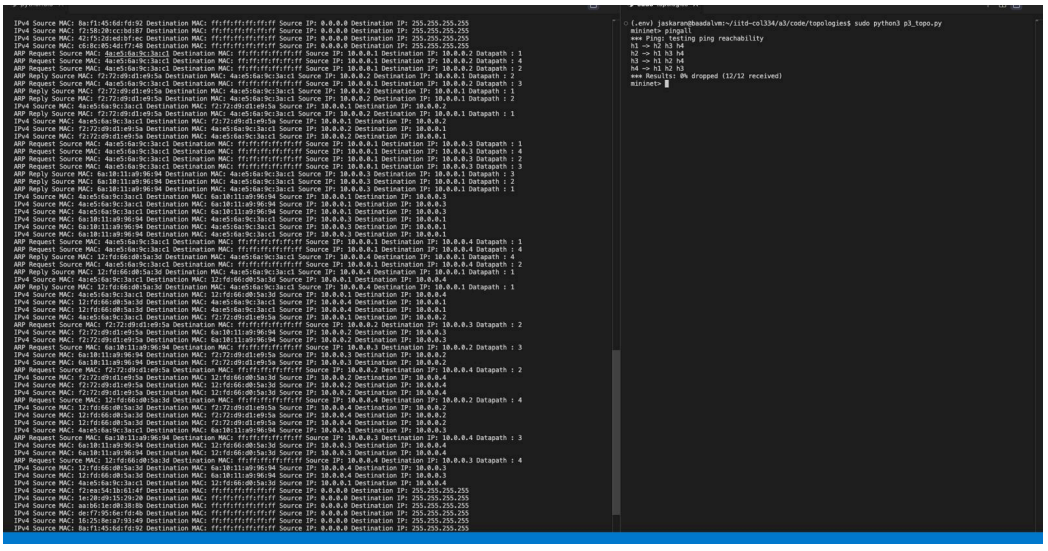
Topology



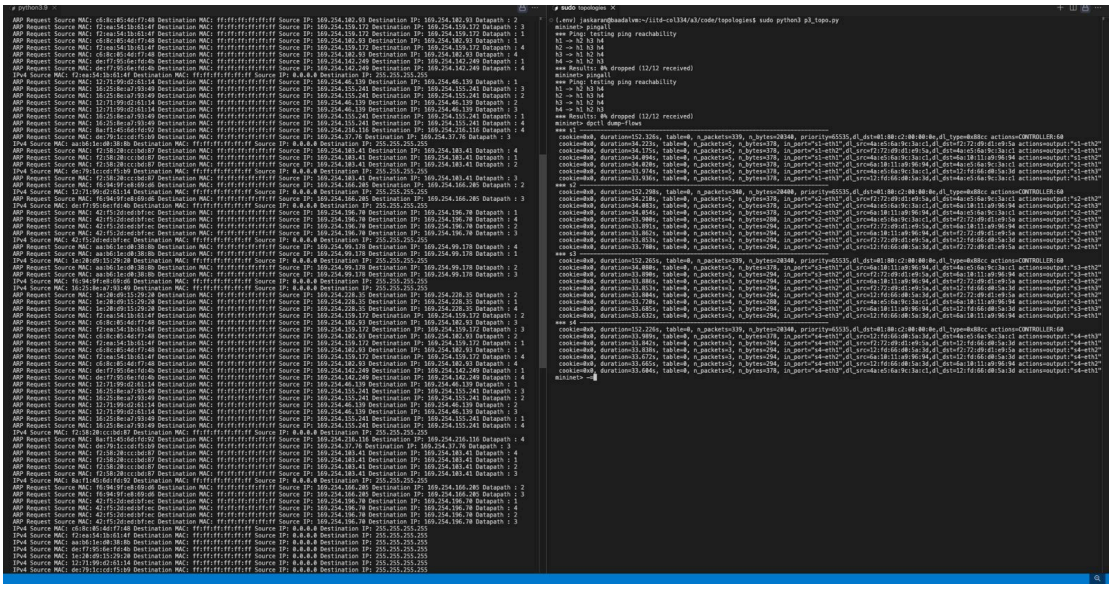
LLDP packets flow and construction of weighted graph



pingall

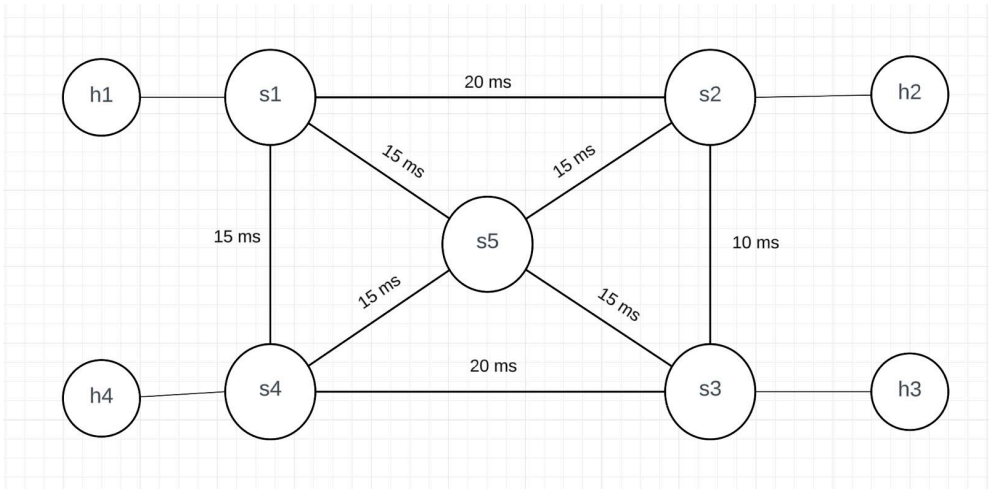


dptcl dump-flows

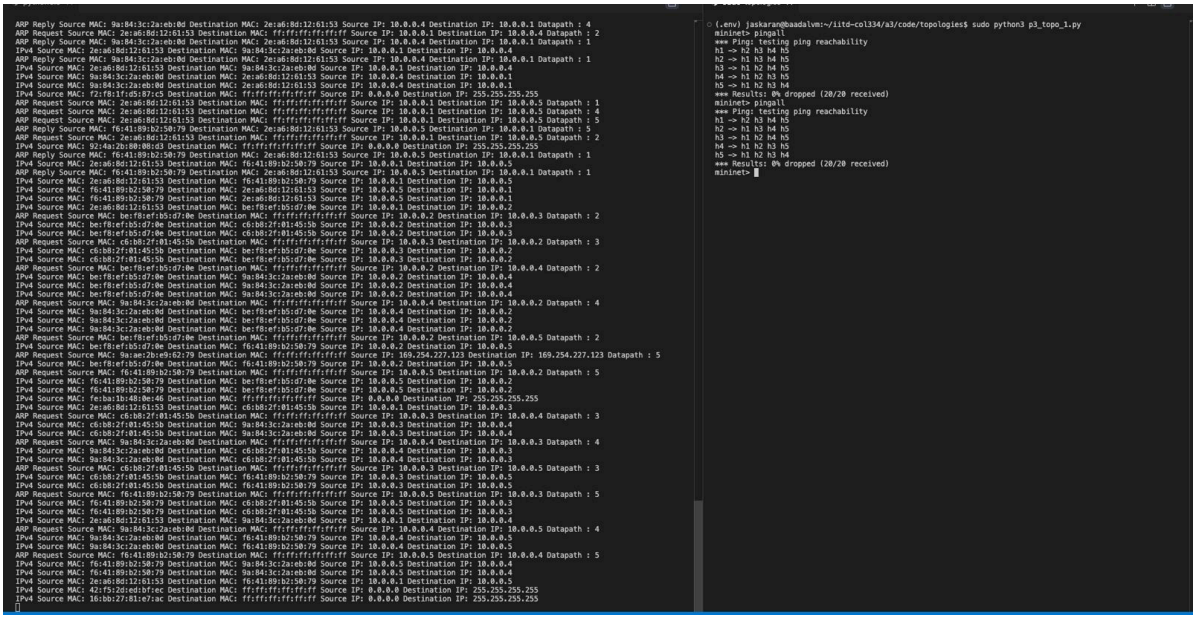


P3_topo_1.py

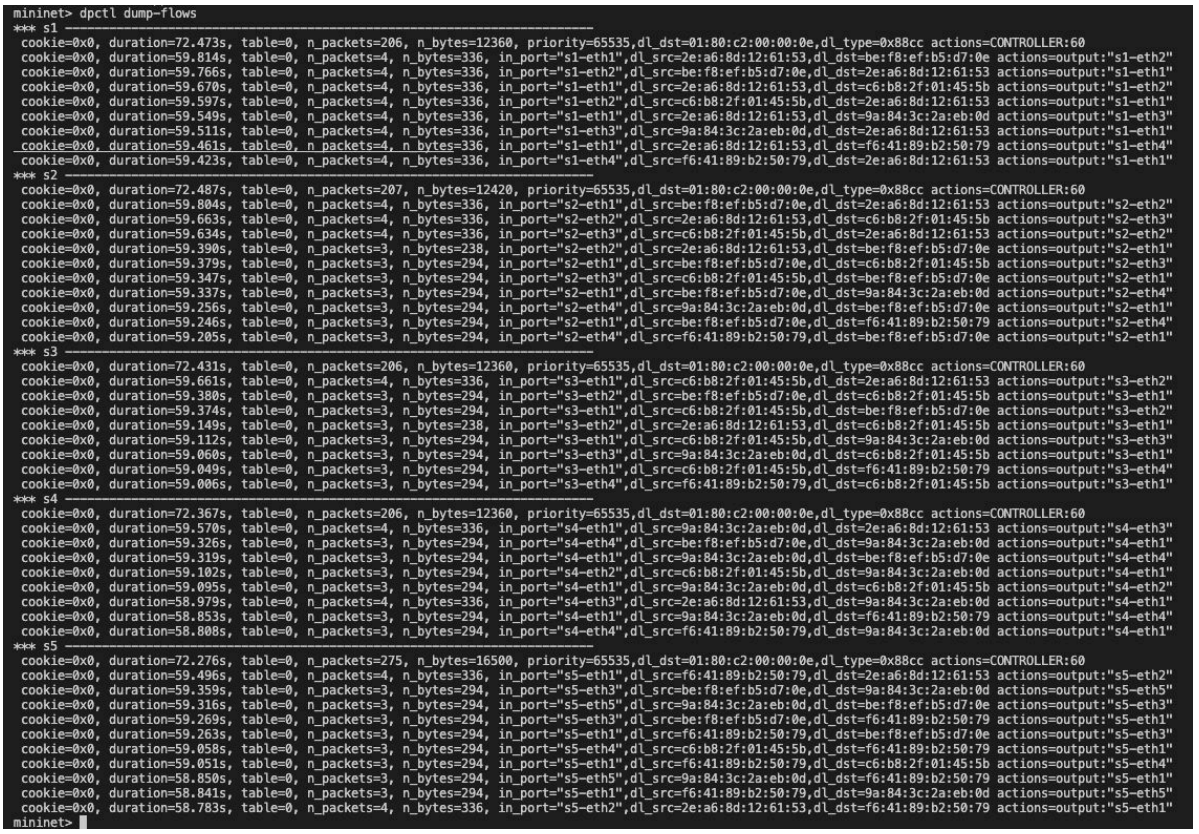
Topology



pingall



dpctl dump-flows



Discussion

The results above show that we were successfully able to implement the shortest path routing and we can analyse dptcl dump-flows to check that.

Part 4: Congestion-Aware Shortest Path Routing

Overview

In this part of the assignment, we implement a congestion-aware routing mechanism using the Link Layer Discovery Protocol (LLDP). By leveraging LLDP, we construct a weighted graph that captures link delays between switches in the network. This dynamic graph enables us to calculate the shortest paths based on real-time network conditions, thus optimising data flow and minimising congestion. The implementation involves sending LLDP packets periodically to update the link delay information, recalculating the shortest paths, and managing flow entries accordingly. Here, the difference from part 3 is that we periodically make the shortest path calculations, and the weighted graph is changing continuously. As the weighed graph is only needed to calculate the shortest path, once it is calculated, we allow for lldp packets to inflow and the shortest path and update the shorts_paths object accordingly accordingly used for routing the packets.

Implementation

The core of our implementation is centred around **the periodic_lldp_timer** function, which is executed as a background thread. This function periodically sends LLDP packets across all switch ports to gather up-to-date information about the network topology and link delays.

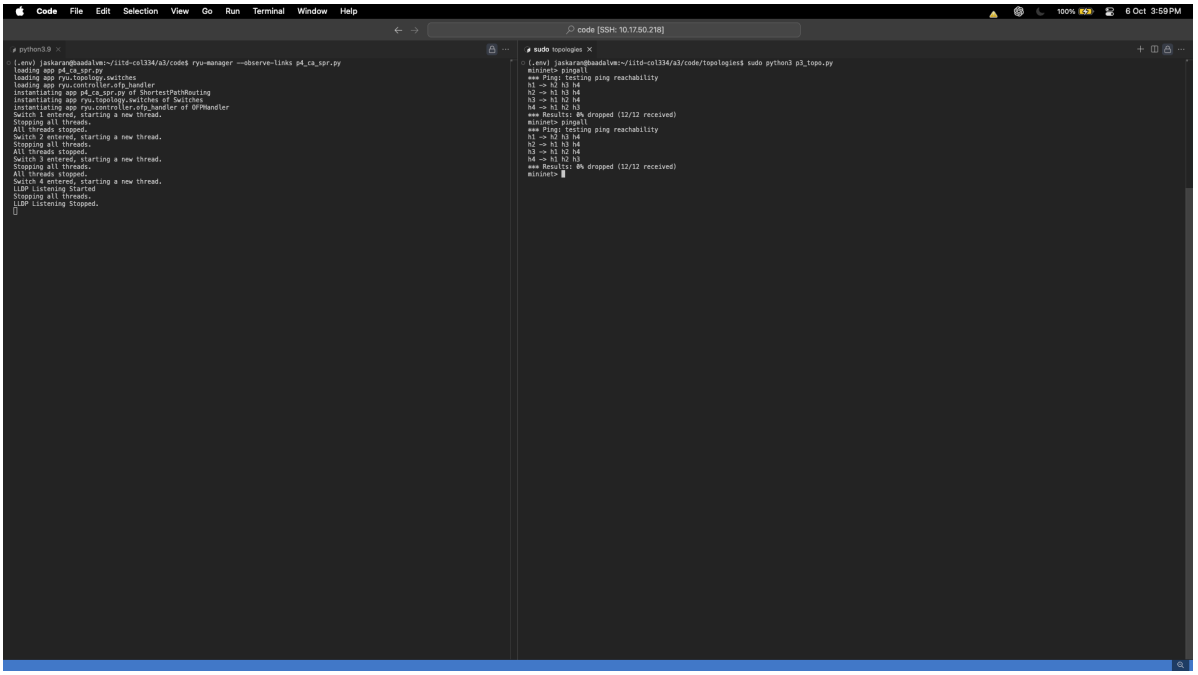
Link delay weighted Graph Construction

For this part of the assignment, we spawn the lldp_thread for the first time At the end of the LLDP thread, the periodic_lldp_timer function is initiated. Inside this function, LLDP packets are sent out every 20 seconds, allowing switches to discover their neighbours and record the link delay information. After collecting this data, the function updates the weighted graph (self.w_graph) used for shortest path calculations. It sends LLDP packets on all ports of each switch to collect information about neighbouring devices. It logs “Graph Updated” to indicate the completion of the update process. It recalculates the shortest paths based on the newly constructed weighted graph, ensuring that routing decisions are made with the latest data. The function iterates through all switches to clear existing flow entries using **delete_all_flows**, preparing the switches to apply new flow entries based on the updated shortest paths.

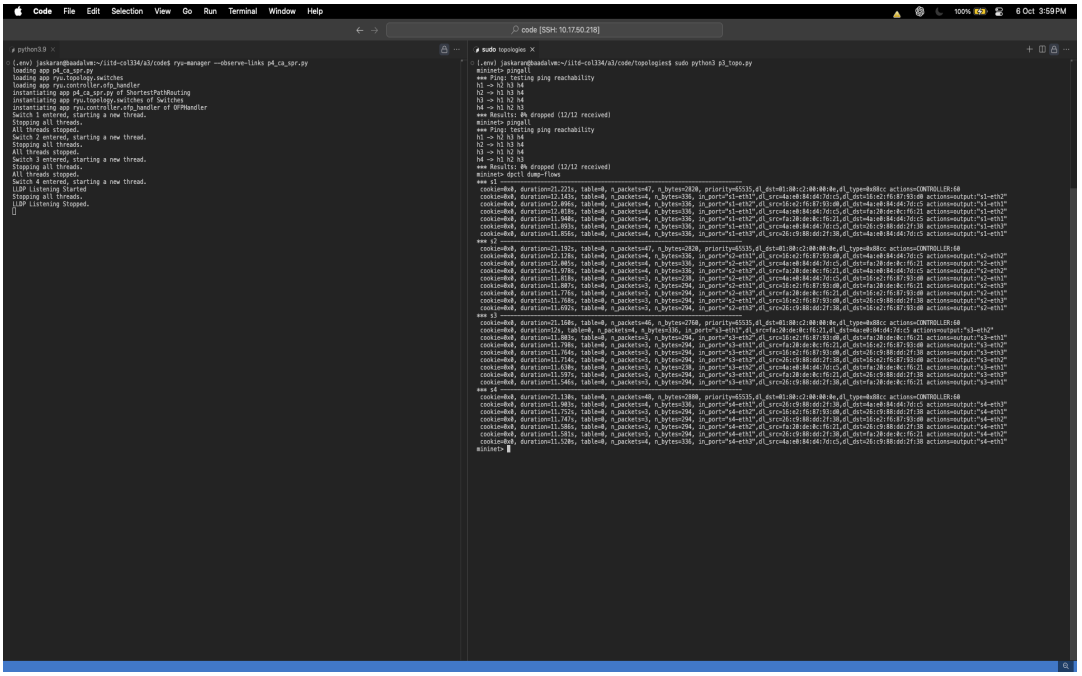
Results

p3_topo.py

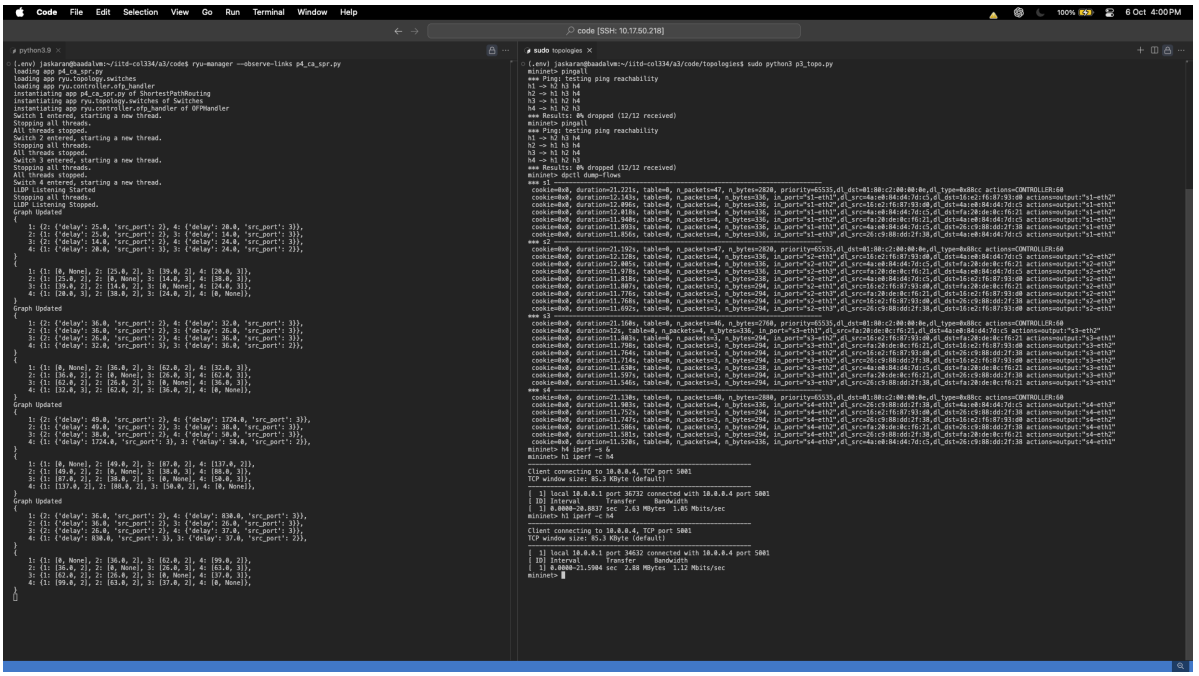
pingall when server starts



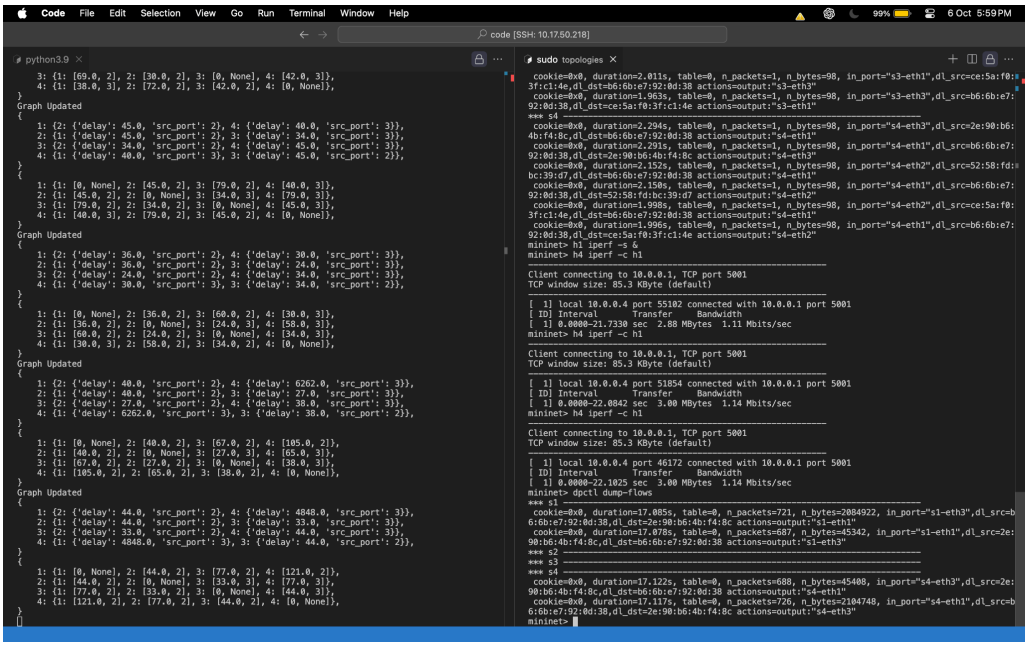
dpctl-flows



Running iperf to overload h1-h4 link, we can see the delay increases for it on the right side and the port is changed from 3 to 2 for sending data



As it deleted the flows, now the flow table is populated only between h1-h4



If we run pingall, then the the table gets populated using the new values