ECE361 Lab 5
Switching and Routing

Due Apr. 9, 2020 @ 11:59 PM

1 Overview

The countless hosts that comprise the Internet, from your laptop, to your phone, to security cameras, etc. are all inter-connected by frame forwarding devices. The vast majority of these devices function as bridges and/or routers.

- *Switching* is used when inter-connecting hosts within a single local area network (LAN).
- *Routing*, on the other hand, is used to inter-connect two or more LANs, enabling hosts belonging to different networks to communicate.

While these two functions both pass frames from one interface to another, they operate on different layers of the network stack. In particular, bridges work by looking only at the layer 2 addresses (e.g. Ethernet MAC addresses) in frames, while routers look at layer 3 addresses (e.g. IP addresses).

In this lab, you will explore these concepts in more detail. In the first part, you will implement the core learning logic of bridges to enable hosts in a network to communicate. In the second part, you will add virtual LANs (VLANs) as a set of logical ports to share the same flooding or broadcast characteristics. Finally, in the last part, you will implement the base functionality of routers to enable hosts across different networks to communicate.

Note that in this lab, we use the words 'bridge' and 'switch' interchangeably.

2 Lab Initialization

This lab includes starter code that will be executed by a network controller. A Python virtual environment will be created that will include the controller. We have provided an initialization script for you that will setup this virtual environment, along with the starter code.

It's highly recommended to update your VM before starting each lab. Open a terminal (in the VM), and type `ece361-update`

To run the initialization script for lab 5, open a terminal and type `ece361-lab-init 5`. You should see something similar to Listing 1.

```
1 ubuntu@ece361:~$ ece361-lab-init 5
2 Finding available UG EECG host...
3 Warning: Permanently added 'ug251.eecg.utoronto.ca,128.100.13.251' (ECDSA) to the list of known
      hosts.
```

```
Creating working directory for lab 5 at /home/ubuntu/lab5
Creating Python3 virtual environment...
Installing libraries...

...

Done. Now run source /home/ubuntu/lab5/sourceMe to activate the virtual environment.
```

Listing 1: Initializing lab 5.

The initialization script creates a working directory for this lab, located at `~/lab5`. To activate the virtual environment, run the source command as shown in Listing 1. Once the virtual environment is activated, you should see a `(.venv)` appear at the beginning of your prompt.

You will need to activate the virtual environment **in each terminal** that runs code belonging to this lab.

The `ece361` library used in this lab is different than the ones from the previous three labs. Your code from the previous labs will not work in this new virtual environment.

In your lab 5 working directory, you will work with only two files:

- `switch_router.py`

  – Starter code for switching and routing in this lab. You need to complete a few functions in this file to complete the lab.

- `configs/CONFIG`

  – Config file which contains only two variables. You can control whether you are working on the first, second or the third part of the lab using these variables which will be explained later.

The other files in your directory (which you do not need to touch) are:

- `network_topo_part1.py`

  – The network topology for the first part of the lab. Running this file creates a virtual network with the given topology using Mininet.

- `network_topo_part2.py`

  – The network topology for the second part of the lab. Running this file creates a virtual network with the given topology using Mininet.

- `lab5-controller`

  – An script to start, stop, or restart the network controller. The network controller enables you to manage the bridges in the network.

- `configs/IP_CONFIG`

  - Contains the IP information of the hosts and router interfaces within the network.

- `configs/VLAN_CONFIG`
  - Contains VLAN information for the second part of the lab.
- `configs/ROUTING`
  - The routing information used by the router in the third part of the lab.

Summary: You only need to complete a few functions in `switch_router.py`, and change the variables in `CONFIG` to control which part of the lab you are working on. You do not need to touch the other files unless you wish to test other topologies.

If you discover a bug in any of the files provided, please report it on Piazza.

3 Part 1: Switching

In lectures, you have seen that hosts in a LAN are connected to a shared medium and can communicate with each other. The shared medium may also be connected to a bridge. Thus, hosts connected to the shared medium can talk to each other as well as to hosts on the other LANs which are connected to the bridge (Fig. 1).

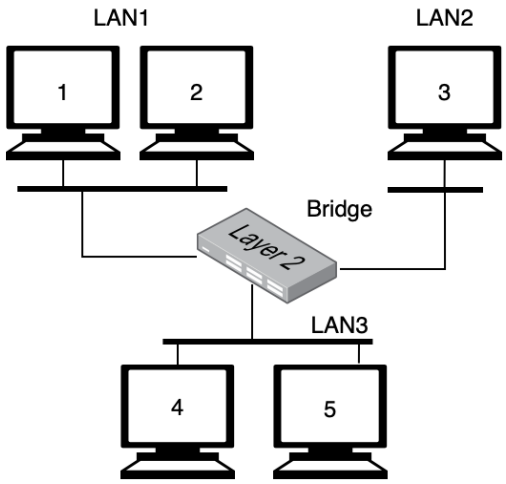


Figure 1: LAN using shared medium to connect hosts in each LAN. They all are connected by the bridges.

To avoid contention in shared mediums, modern wired networks are configured such that the hosts in a LAN are connected directly to bridges, as shown in Fig. 2. In this lab, we assume each host is directly connected to a bridge.

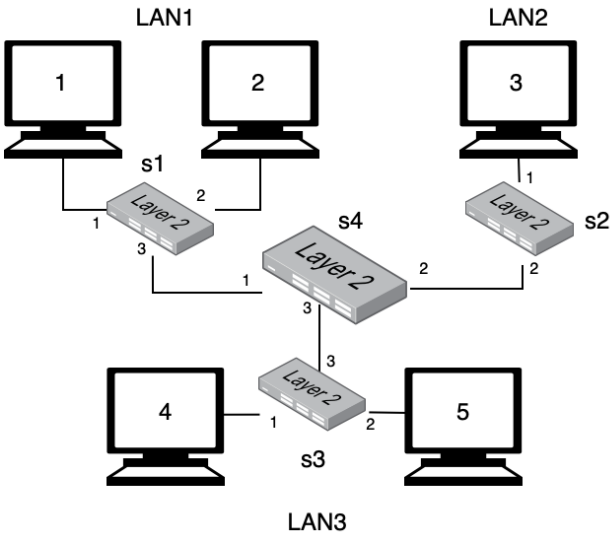


Figure 2: Topology for part 1, using bridge to connect hosts in a LAN

For this part of the lab, we have provided a Mininet topology file to build a network like that shown in Fig. 2. You can execute the `network_topo_part1.py` file with sudo privileges, as shown in Listing 2. After running this file, a mininet terminal shows up.

Make sure you use **sudo python**, and not python3, when running the Mininet topology files.

```
(.venv) ubuntu@361:~/lab5$ sudo python network_topo_part1.py
```

Listing 2: Execute the the network topology file to create the virtual network

3.1 Logic of Learning Bridge

Assume all hosts and bridges in the network have just come online for the first time. When a host first sends a frame out, and it arrives at a bridge, the bridge will not initially know which port the frame should be sent out on. As shown in Fig. 3, each frame has a source and destination MAC addresse which represent the MAC address of the frame's sender and its destination. When the frame arrives from a port in the bridge, the bridge learns that the sender of the frame is reachable from that port. For instance, in Fig. 3, the bridge learns that the source with `MAC: 00:00:00:00:00:01` is reachable using port 0. Thus, it associates the port with the source MAC address of the frame in its forwarding table.

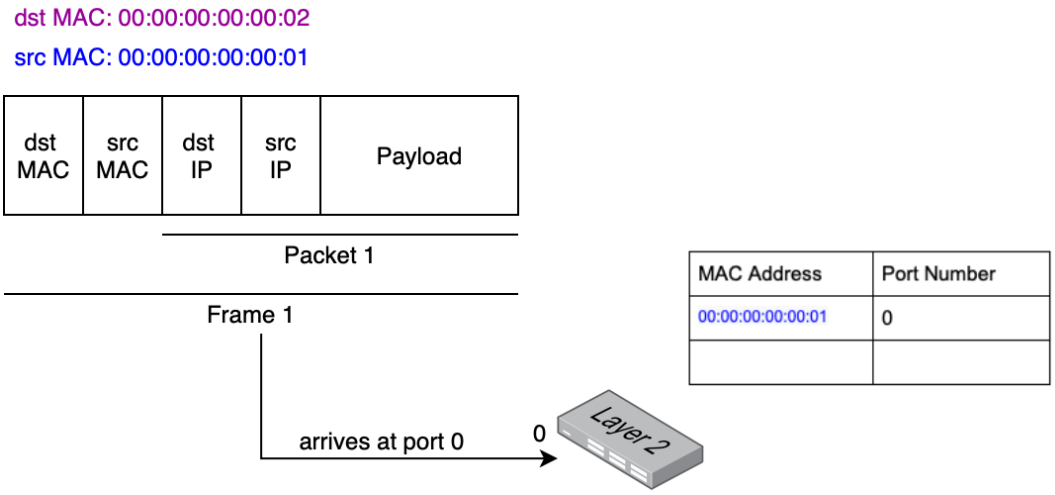


Figure 3: A frame arrives at port 0 of the bridge

To decide which port it should send the frame out on, it takes the destination MAC address and looks up its forwarding table. However, since the destination MAC address does not yet exist in the forwarding table, the bridge does not know which output port should be used. Hence, it floods the frame to all its port and every host connected to the bridge receives the frame. If the destination host is connected, it will receive the frame, and it may send a response frame. When the bridge first receives the reply frame from the destination host through a port, it associates that port with the source MAC address of the reply frame as shown in Fig. 4. The process of learning MAC addresses and associating them to appropriate ports is called *learning*.

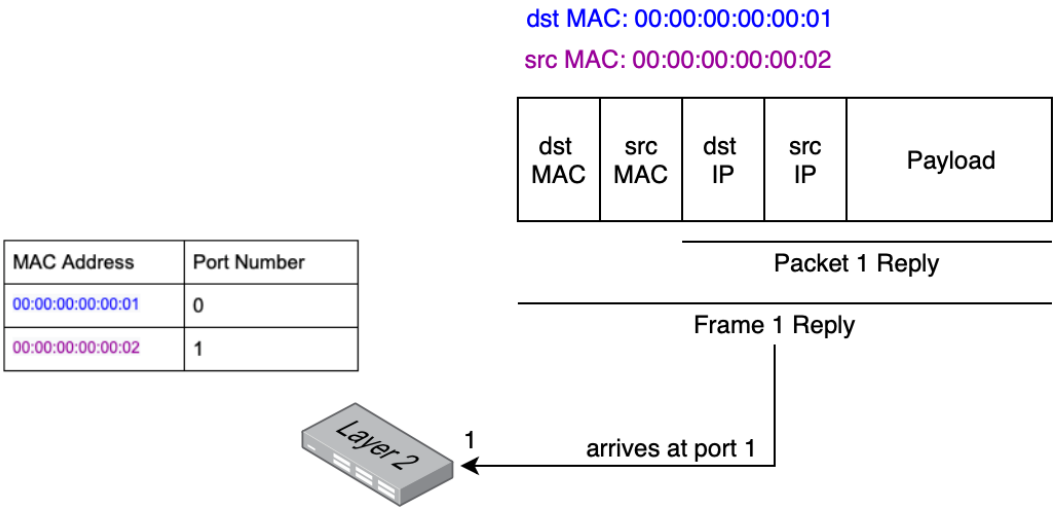


Figure 4: A reply frame arrives at port 1 of the bridge.

3.2 Task

Your job in this part of the lab is to handle the learning for the bridges in the network by completing two functions in the `switch_router.py` file:

- **learn_mac_to_port(dpid, eth_src, in_port)**: Perform learning when a frame with the given source MAC (Ethernet) address (`eth_src`) arrives at bridge (specified by `dpid`) on the given input port (`in_port`). This function is called whenever a frame arrives at a bridge, and it populates the forwarding table for the bridge. Please take a look at the comments in the starter code to familiar yourself with the format of the parameters and the output. This function return nothing.
- **get_out_port(dpid, eth_dst)**: Given a bridge (specified by `dpid`) and the destination MAC (Ethernet) address (`eth_dst`), this function returns the appropriate output port. This is done by looking up the forwarding table of the bridge with the given `dpid`. Please take a look at comments in the starter code to familiar yourself with the formats. This function should return a single number (int), specifying which output port to send the frame out on.

Each bridge has a unique identifier called `dpid`. We use the `dpid` in almost all functions to distinguish between different network devices in the network.

You should familiarize yourself with Python's dictionary data structure: https://docs.python.org/3/library/stdtypes.html#typesmapping.

Once you write the required functions, you can test the network. All bridges in the network are managed by a network controller. Your code will also be executed by the controller. You can start, restart and stop the controller using the `lab5-controller` script in the lab directory. For example, to start the controller, see Listing 3.

```
(.venv) ubuntu@361:~/lab5$ ./lab5-controller start
Started new Ryu instance with PID 30043
```

Listing 3: Sample output when starting the network controller.

Once you start the network controller using `lab5-controller`, and create the network by running `network_topo_part1.py`, you can test your code. In the terminal where you run the `network_topo_part1.py` file, a Mininet terminal is available. You can issue the commands within each virtual hosts to check their connectivity with other hosts. For an example of the basic Mininet commands you'll need, please study Listing 4 carefully.

```
(.venv) ubuntu@ece361:~/lab5$ ./lab5-controller start
Started new Ryu instance with PID 30043
(.venv) ubuntu@ece361:~/lab5$ sudo python network_topo_part1.py
*** Adding controller
*** Adding hosts
*** Adding Routers
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4 h5
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Running CLI
*** Starting CLI:
```

```
mininet> h1 ping -c1 h2
PING 10.0.1.12 (10.0.1.12) 56(84) bytes of data.
64 bytes from 10.0.1.12: icmp_seq=1 ttl=64 time=3.83 ms
...
mininet> h2 ping -c1 h4
PING 10.0.1.14 (10.0.1.14) 56(84) bytes of data.
64 bytes from 10.0.1.14: icmp_seq=1 ttl=64 time=6.64 ms
...
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5
h2 -> h1 h3 h4 h5
h3 -> h1 h2 h4 h5
h4 -> h1 h2 h3 h5
h5 -> h1 h2 h3 h4
*** Results: 0% dropped (20/20 received)
mininet> exit
*** Stopping network*** Stopping 1 controllers
c0
*** Stopping 8 links
........
*** Stopping 4 switches
s1 s2 s3 s4
*** Stopping 5 hosts
h1 h2 h3 h4 h5
*** Done
```

Listing 4: Sample workflow for creating the topology in part 1 and testing the code.

If you change your code, you will have to restart the controller. However, you do not have to re-create the topology.

To help you debug, the output of the controller is stored in a log file in the lab directory called `controller.log`. The log file also contains the output of any `print()` statements you choose to include in your code. You may check the log by looking at the end of the `controller.log` file using the `tail` command, as shown in Listing 5.

```
(.venv) ubuntu@361:~/lab5$ tail controller.log
```

Listing 5: Checking the last few lines of the controller's output log file.

Using `tail -n100` shows the last 100 lines in the file. Alternatively, `tail -f` will allow you to follow the output file as more text is appended to it.

4 Part 2: Virtual LANs (VLANs)

Continuing from the previous part, we want to have a set of logical ports to share the same flooding or broadcast characteristics. Consider the network topology for this part shown in Fig. 5. The purple hosts belong to `VLAN1` while the black ones belong to the `VLAN2`. Unlike the layer 2 network in Part 1, where all hosts can reach to each other, in a VLAN network, only hosts within the same

VLAN should be able to communicate with each other. For instance, all purple hosts should be able to reach each other, and all black hosts should be reachable from other black hosts. A purple host cannot communicate with a black host.

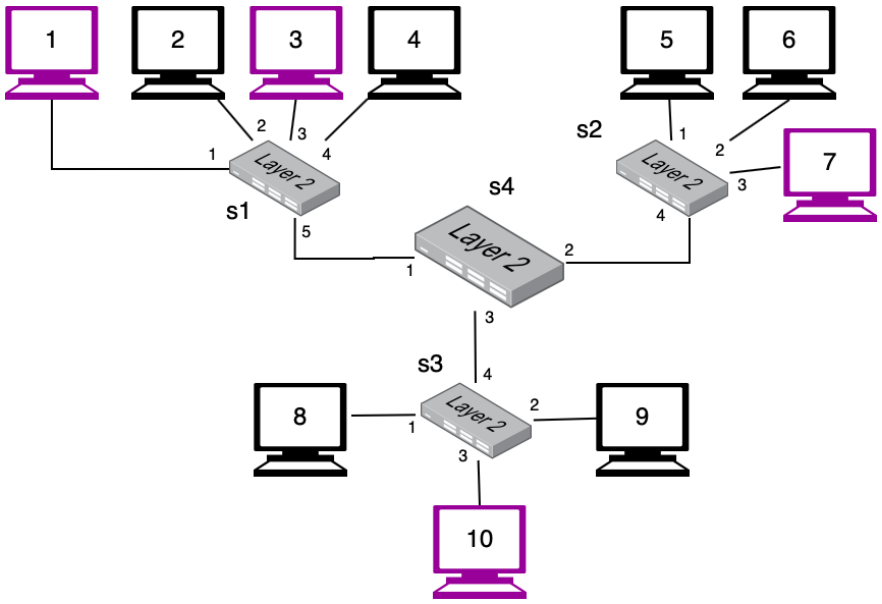


Figure 5: Topology for part 2, VLAN network

Note: It is also possible for a host to belong to multiple VLANs (not shown in the figure). In such a case, it would be able to send/receive frames to/from hosts belonging to all the VLANs which it is a member of.

4.1 Task

Your job in this part is to do the learning again while considering the VLAN assignments in the network. The VLAN configuration information is available in `VLAN_CONFIG` file, but you do not need to modify its content. You are to complete two functions in the `switch_router.py` file:

- **get_vlans_of_port(dpid, port_no)**: Returns a list of VLAN name(s) for the given `port_no` of the given bridge with `dpid`. To do so, you should use `vlan_to_port`, a class member variable which has been automatically populated for you.
- **get_out_port_vlan(dpid, source_host_vlans, mac_addr):** Returns the appropriate output port when VLANs are enabled. You have to consider two cases:

1. If the destination MAC address is in the bridge's forwarding table, then return the output port if it belongs to a VLAN which the source host is a member of. However, if the output port does not belong to a VLAN which the source is a member of, then discard the frame.
2. If the output port corresponding to the destination MAC is not found in the forwarding table, then return back all ports that belong to a VLAN which the source host is a member of (i.e. VLAN-constrained flooding).

To run and check your functions you have to perform three steps:

1. **Update CONFIG file**: Change the `"VLAN"` to `"enable"` in the `CONFIG` file.
2. **Run the controller**: Run (or restart) the controller.
3. **Run the network topology file**: Execute the `network_topo_part2.py` file.

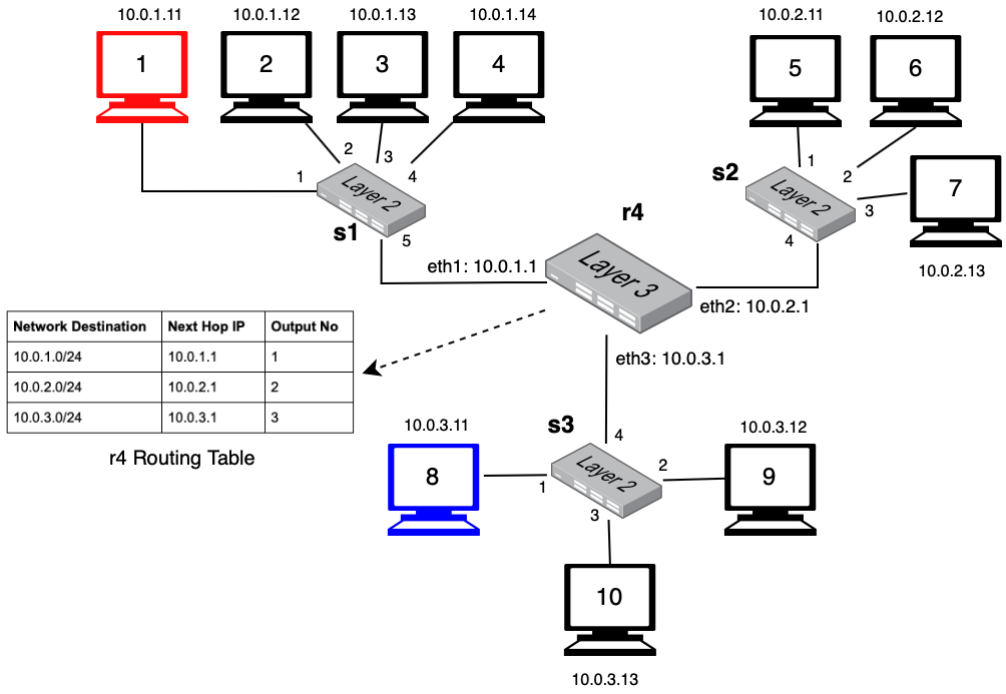
You may check the logs of the controller for possible errors or outputs. Then in the Mininet terminal you can test the connectivity between each hosts. For example, in Listing 6, note that host 1 is able to ping host 3, but it is unable to ping host 2.

```
(.venv) ubuntu@ece361:~/lab5$ ./lab5-controller restart
No current instance of Ryu found to be running
Started new Ryu instance with PID 31267
(.venv) ubuntu@ece361:~/lab5$ sudo python network_topo_part2.py
*** Adding controller
*** Adding hosts
*** Adding Routers
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...
*** Running CLI
*** Starting CLI:
mininet> h1 ping -c1 h3
PING 10.0.1.13 (10.0.1.13) 56(84) bytes of data.
64 bytes from 10.0.1.13: icmp_seq=1 ttl=64 time=7.66 ms

--- 10.0.1.13 ping statistics ---
1 frames transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.669/7.669/7.669/0.000 ms
mininet> h1 ping -c1 h2
PING 10.0.1.12 (10.0.1.12) 56(84) bytes of data.
From 10.0.1.11 icmp_seq=1 Destination Host Unreachable

--- 10.0.1.12 ping statistics ---
1 frames transmitted, 0 received, +1 errors, 100% packet loss, time 0ms

mininet>
```

Listing 6: Sample workflow for creating the topology in part 2 and testing the code.

5 Part 3: Routing

To connect multiple LANs and create an *inter-network* topology, we use routers. As shown in Fig. 6, we replace one of the bridges in part 2 with a router (named as `r4`), segmenting the hosts into three different LANs.



Network Destination	Next Hop IP	Output No
10.0.1.0/24	10.0.1.1	1
10.0.2.0/24	10.0.2.1	2
10.0.3.0/24	10.0.3.1	3

r4 Routing Table

Figure 6: Topology for Part 3, Routing

In this part, we will use the term *interface* to talk about the ports on a router.

Each host, as well as each one of the router's interfaces, has an IP address. Note the similarity between the router's IP address for interface `eth1`, and the IP addresses for hosts within the LAN interconnected by switch `s1` (i.e. hosts 1 to 4). Note the same pattern for the other router interfaces' IP address, and the addresses of the hosts in the LANs reachable via that interface. The router acts as a common host between all three LANs that it is connected to. Unlike a regular host, it performs the special task of `routing` frames between its various interfaces.

5.1 Operation of Hosts

When a host wishes to send a frame, it is able to determine if the destination host is within the same LAN by comparing the destination's IP with its own IP. If it determines that the destination host is within the same LAN, it creates a frame where the layer 3 (IP) source and destination fields are its own IP, and the destination host's IP. The layer 2 source and destination fields will be its own MAC address, and the destination's MAC address. On the other hand, if it determines the destination host is outside its LAN, it still creates a frame with its own source IP, the destination's IP, and its own source MAC address; however, the destination MAC address will be the router's interface which connects to the LAN.

Layer 2 addresses (Ethernet MACs) are always local within the LAN.

5.2 Operation of Routers

When a router receives a frame on one of its interfaces, must determine which of its interfaces should be used to send the frame out on to reach the target host. To accomplish this, it looks up a *routing table*. A sample routing table is shown in Fig. 6. The entries in the routing table can be inserted manually or dynamically using routing algorithms. If the router determines that the received frame is destined for a LAN on another interface (i.e. not the same interface upon which it arrived on), then the router creates a new frame to send out on the interface connected to the destination IPs LAN. In this new frame, it uses the same layer 3 source and destination as the received frame, but with its own MAC address and the destination host's MAC address as the layer 2 source and destination.

Layer 3 addresses can be used across networks -- they are inter-network addresses!

Under what circumstances would a router receive a frame on an interface, where the destination host is reachable upon the same interface it arrived on?

In Fig. 6, assume Host 1 wants to send a frame to Host 8. It determines that Host 8 is in another network. Thus, it constructs a frame with its source IP and MAC address, and IP address of Host 8, but it uses the MAC address of the router interface connected to its LAN (i.e. `r4-eth1`) as the destination MAC address. The bridge `s1`, being unaware of layer 3 addresses, simply forwards the frame to the router. When the frame arrives at the router, the router looks into its routing table and searches for the interface whose IP address is in the same network as the destination IP address of the frame, and sends the frame out on that interface.

5.3 Updating the Starter Code

This section is only relevant if you have already started the lab and previously ran the lab init script prior to the release of part 3. If you ran the init script after the release of lab 3, proceed to subsection 5.4.

Please update the starter code by running `ece361-lab-init 5` again. You should see something similar to Listing 7.

```
ubuntu@ece361:~$ ece361-lab-init 5
Finding available UG EECG host...
Warning: Permanently added 'ug251.eecg.utoronto.ca,128.100.13.251' (ECDSA) to the list of known
    hosts.
Updating code and config files for Part 3...
Saving copy of switch_router.py as switch_router.py.1585636916
Editing switch_router.py to add new variables and functions...
patching file switch_router.py
Done editing switch_router.py
```

Listing 7: Updating the lab 5 starter code.

The working directory (i.e. `~/lab5`) should now contain a new file: `network_topo_part3.py`. Additionally, the `switch_router.py` file should contain two new member variables at the top of

the class, and five new member functions at the bottom of the class. The new member variables are: `self.routing_table` and `self.ip_to_mac`. The new functions are: `get_bridgeName_by_dpid()`, `get_mac_by_ip()`, `is_ip_within_net()`, `get_out_iface_info()`, and `send_frame_by_router()`. The new member variables and the first three member functions are helpers which you will need to use in completing the last two functions.

5.4 Task

For this last part, your task is to find the correct source and destination IP and MAC addresses, as well as the interface on which a frame should be sent out, when it arrives at the router. In particular, you have to complete these functions:

- **send_frame_by_router(dpid, src_mac, dst_mac, src_ip, dst_ip)**: This function is called when a frame arrives at the router. Given the `dpid` of a router, and the layer 2 and 3 addresses of the received frame, this function returns a Python List object containing the source MAC, destination MAC, source IP, destination IP, and the output interface number. **Note that the order of these values should be exactly as described.**
- **get_out_iface_info(dpid, ip_addr):** Given the `dpid` of a router and an IP address `ip_addr`, this function finds the interface that should be used to send a frame to reach destination `ip_addr`. Returns the interface's IP address and output interface number.

Carefully read the comments for the two new member variables and the three helper functions to understand their usage before you begin your coding.

You may check your work by following these steps:

1. **Update CONFIG file**: Change the `"VLAN"` to `"disable"` and `L3` to `"enable"` in the `CONFIG` file. If you do not change the `"VLAN"` to `"disable"` the controller does not work for this part.
2. **Run the controller**: Run (or restart) the controller. Each time you change your code, you have to restart the controller.
3. **Run the network topology file**: Execute the `network_topo_part3.py` file.

When you have properly completed the code, you will see results similar to Listing 8:

```
(.venv) ubuntu@ece361:~/lab5$ ./lab5-controller restart
No current instance of Ryu found to be running
Started new Ryu instance with PID 31234
(.venv) ubuntu@ece361:~/lab5$ sudo python network_topo_part3.py
*** Adding controller
*** Adding hosts
*** Adding Routers
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8 h9 h10
*** Starting controller
c0
```

```
*** Starting 4 switches
s1 s2 s3 r4 ...
*** Running CLI
*** Starting CLI:
mininet> h1 ping -c1 h3
PING 10.0.1.13 (10.0.1.13) 56(84) bytes of data.
64 bytes from 10.0.1.13: icmp_seq=1 ttl=64 time=7.66 ms

--- 10.0.1.13 ping statistics ---
1 frames transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 7.669/7.669/7.669/0.000 ms
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8 h9 h10
h2 -> h1 h3 h4 h5 h6 h7 h8 h9 h10
h3 -> h1 h2 h4 h5 h6 h7 h8 h9 h10
h4 -> h1 h2 h3 h5 h6 h7 h8 h9 h10
h5 -> h1 h2 h3 h4 h6 h7 h8 h9 h10
h6 -> h1 h2 h3 h4 h5 h7 h8 h9 h10
h7 -> h1 h2 h3 h4 h5 h6 h8 h9 h10
h8 -> h1 h2 h3 h4 h5 h6 h7 h9 h10
h9 -> h1 h2 h3 h4 h5 h6 h7 h8 h10
h10 -> h1 h2 h3 h4 h5 h6 h7 h8 h9
*** Results: 0% dropped (90/90 received)
```

Listing 8: Sample workflow for creating the topology in part 3 and testing the code.

6 Exerciser

You can use the exerciser to help test the correctness of your implementation. The exerciser will run a set of public test cases against your code. However, note that the tests in the exerciser is not complete and it is your responsibility to test your code to make sure it conforms to the requirement of the lab. Additionally, we may test your code with custom topologies. Ensure you have the following file all within the same directory:

- **switch_router.py**: Your completed code for all three parts.

In terminal, browse to the directory containing the files and type `ece361-exercise 5`.

7 Submission

Once you are confident of your implementation, you can run the submission process (which will invoke the exerciser before submitting). Only one person in the group needs to submit.

From the same directory as where you ran the exerciser, type `ece361-submit submit 5`.

You can then verify the submission by typing `ece361-submit list 5`.

At some point after the lab's due date, private test cases will be run against your submission to calculate your final mark.