Lab 1: MAC Learning, Forwarding, and STP (10 Points)

1 Download Lab

Download the Lab 1 files from Brightspace. The source code will be inside the directory "`Lab1/`". You must use the environment from Lab 0 to run and test your code. Next, open a terminal and "`cd`" into the "`Lab1/`" directory. Now you are ready to run the lab!

2 Task

For this lab, your task is to implement MAC learning, MAC forwarding, and Spanning Tree Protocol (STP). You will create a *forwarding table* at each switch, and implement STP and MAC learning to populate those forwarding tables. Next, you will implement MAC forwarding at each switch to forward DATA packets using the information in the switch's forwarding table. For this lab, you can assume that switches and clients never fail. But the link status can change dynamically, i.e., during the runtime, existing links can be removed, or new links can be added, or the link cost can change. Your implementation must be resilient to changes in link status.

3 Source Code

For this lab, you will do your implementation inside a simulated network environment. The simulated network has switches, clients (end hosts), links, and packets just like a real network. Files "`switch.py`", "`client.py`", "`link.py`" and "`packet.py`" contain the implementations of a switch, a client, a link, and a packet respectively. You must **not** modify any of these files, however you can import the classes defined in these files and use their fields and methods for your implementation. You will do your implementation inside the file "`STPswitch.py`". "`STPswitch`" is a subclass of the "`Switch`" class and inherits all its fields and methods while overriding its "`handlePacket`", "`handleNewLink`", "`handleRemoveLink`", and "`handlePeriodicOps`" methods.

"`handlePacket`" method is called every time a switch receives a packet (DATA or CONTROL).

"`handleNewLink`" method is called every time a new link (including the initial set of links) is added to the switch or the cost of an existing link changes.

"`handleRemoveLink`" method is called every time an existing link is removed from the switch.

"`handlePeriodicOps`" method is called periodically. The period is set using the "`heartbeatTime`" value in the json file as described in the next section.

You must **not** override any other "`Switch`" class methods, but you can add new fields and methods to "`STPswitch`" as you see fit.

Tip 1: For each switch, all the links directly connected to that switch are stored in the "`links`" data structure in "`switch.py`" file. Also, whenever a new link is added or removed or the link cost changes, this information is updated automatically in the "`links`" data structure and the respective "`handle...`" method is called.

Tip 2: Go through "`packet.py`" to understand packet format and types. In particular, there are two kinds of packets – DATA packets that are generated by clients and forwarded by the switches, and CONTROL packets that are generated and exchanged between switches to implement the STP algorithm. Refer to "`sendDataPackets`" method in "`client.py`" to understand how new packets are created. Go through "`link.py`" to understand various link parameters, such as "`cost`" and "`status`".

4 Running the Code

To start the network simulator, run the following command in the terminal,

```
$ python3 network.py [networkConfigurationFile.json]
```

The json file argument specifies the configuration of the simulated network. This is explained in the next section. You are provided with three sample json files ("`01.json`", "`02.json`", "`03.json`").

To run the experiments with all the provided network configuration json files, run the following command,

```
$ ./run_all.sh
```

To clean temporary files from the previous run of the experiment, run the following command,

```
$ ./clean.sh
```

5 Network Configuration Json File

A sample network configuration json file can be found at "`01.json`". The "`switch`" and "`client`" lists in the file specify the address of all the switches and clients in the network. Switches are addressed using numbers and clients are addressed using letters. Letter "X" is reserved for broadcast address and cannot be used as a client address. Next, given n clients in the network, each client periodically sends out n DATA packets, one of which is a broadcast packet (with destination address "X") and the remaining $n-1$ packets are unicast packets destined to the $n-1$ clients (not including the source client). A link is represented as `[e1, e2, p1, p2, c]`, where a node (switch or client) `e1` is connected to node `e2` using port `p1` on `e1` and port `p2` on `e2`, and `c` is the cost of the link connecting `e1` and `e2`. Also, the links in the network can be added or removed dynamically as specified in the "`changes`" list. An existing link can be removed using the format `[t, [e1, e2], "down"]`, meaning the link between nodes `e1` and `e2` would be removed at time `t`. Similarly, a new link can be added using the format `[t, [e1, e2, p1, p2, c], "up"]`, meaning a link between port `p1` of node `e1` and port `p2` of node `e2` would be added at time `t`, and the cost of that link would be `c`. The above format for link addition can also be used to change the cost of an existing link. The "`handlePeriodicOps`" method is called every "`heartbeatTime`". You can change the value of "`heartbeatTime`" in the json file to decide the rate of periodic operations. The network simulator runs for a fixed duration of time stated in the "`endTime`" field. The provided json files have "`endTime`" set to 1000 simulation time units, which equates to ~100 seconds in real time. **So, you should wait for each experiment to run for ~2 minutes before it prints the final output.** You may change the "`endTime`" value for your own testing.

NOTE: The simulator will start adding the initial set of links specified in the json file at time t=0, but it may take a few simulation time units for all the links to be added. The "`handlePeriodicOps`" method will also be first triggered at time t=0, and then periodically every "`heartbeatTime`". But you cannot assume that all the initial links will already be added before the first trigger of "`handlePeriodicOps`" because of the reason mentioned above.

6 Output

At the "`endTime`", the simulator cleans up all the active/queued packets in the network, and generates a last batch of broadcast and unicast packets between each pair of clients. It also tracks the route taken by each packet in the last batch, and prints it as the final output on the stdout. For your output to match the correct output, your solution must converge to the correct spanning tree and forwarding table entries before the "`endTime`". If the output path between a pair of clients is empty, i.e., [], then it means the packet generated from the source client did not reach the destination client (probably because your solution converged to a wrong route).

7 Grading

We will test your submission against 10 test cases (network configuration json files), for a total of 10 test case runs. For each test case run, if your entire output matches the correct output, you will get 1 point, else a 0. **No partial credit.** We have provided you with 3 out of those 10 test cases for your testing. The remaining 7 test cases are private, and will **not** be released at any point. However, we will release your output for each test case run to let you know which test case runs you passed and which ones you failed. You are highly encouraged to create your own test cases to test the robustness of your implementation.

IMPORTANT: Your code must **not** print any custom / debug statements to the terminal (stdout) other than what the simulator already prints.
Violations of this guideline will result in a 10% grade penalty.

8 Debugging

The provided network configuration json files also contain the correct routes between each pair of clients, which you can use to debug your implementation. Besides, the network simulator also generates .dump files for each switch and client. These files contain information about each packet received by a switch or a client during the runtime of the simulator. A received DATA packet in the dump file will be tagged as "DUP PKT" if the packet is a duplicate of some previously received DATA packet, and tagged as "WRONG DST" if the destination address of the DATA packet does not match the address of the recipient client. You can use these dump files to debug your implementation.

9 A Note About Parallel and Distributed Computation

One of the hardest things to grasp about this lab (and this course in general) is the parallel and distributed nature of algorithms needed to make our networks work and scale. We are trained to think of computation as a series of serial logic execution. But in this lab, just like in a real network, the code at each switch will be running in parallel, and the order of events (such as packet receipt) will be **non-deterministic**, i.e., it can change with every new run of the experiment even with the exact same test case. Fortunately, the algorithms you are implementing for this lab (and this course in general) are designed to be resilient to any non-deterministic system behavior, and so if your implementation is bug free, it is guaranteed to pass every possible test case in every run of the experiment. **However, if you have a bug in your code, then due to the non-deterministic nature of the system, your code might sometimes pass and other times fail the same exact test case.** This takes some time to wrap your head around and can be very frustrating. Debugging a parallel and distributed code is one of the hardest things in computer science, and unfortunately there is no principled way to approach this problem. Hence, we provide you with the dump files so you can trace the history of events to see where things might have failed. Another suggestion would be to print your forwarding tables periodically during the code run, to see at what point the forwarding table entries do not match the expected values, and use that to debug your code.

10 Some Implementation Hints and Guidelines

1. Do not generate too many unnecessary CONTROL packets! Only generate them periodically (every "`heartbeatTime`" provided in the json file) and when your local state *changes*. Otherwise, it may overload the network, not allowing your solution to converge to the correct spanning tree and forwarding table entries before the experiment ends. This could result in either incorrect or empty paths in the final output! This is a very important consideration whenever you are implementing a network protocol in the real world — CONTROL packets must **not** overload the network!
2. Do not send DATA packets (broadcast or unicast) over the link/port packet arrived on or over an INACTIVE link/port. Ignore any DATA packets received on an INACTIVE link/port.
3. Send CONTROL packets over *all* available links/ports at a switch, including INACTIVE link/port.
4. When Case 2 in STP succeeds, i.e., you choose a new next hop link, make sure to make the new link ACTIVE.
5. After receiving each CONTROL packet, you should check whether the link on which the packet arrived needs to be ACTIVE or INACTIVE, and explicitly change the status of the link as one or the other each time.
6. Remove all entries from the forwarding table corresponding to a link that has become INACTIVE.

11 Submission

You are required to submit a single file "`STPswitch.py`" on Brightspace.