Lab 3: Distributed Routing (10 Points)

1 Download Lab

Download the Lab 3 files from Brightspace. The source code will be inside the directory "`Lab3/`". You must use the environment from Lab 0 to run and test your code. Next, open a terminal and "`cd`" into the "`Lab3/`" directory. Now you are ready to run the lab!

2 Task

For this lab, your task is to implement network routing (Distance Vector (DV)) and forwarding protocols. You will create a *routing table* at each router, and implement DV to populate those routing tables. Next, you will implement forwarding logic at each router to forward DATA packets using the information in the routing table. For this lab, you can assume that we **never add** to the initial set of routers and clients, and that routers and clients **never fail**. You can also assume **no packet drops**. However, 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 solution must be resilient to such changes.

3 Honors Milestone

For Honors milestone, you will implement the Link State (LS) routing protocol in addition to the Distance Vector (DV) protocol. **Note that you only need to do the Honors milestone if you have signed the Honors contract for this class.**

4 Source Code

For this lab, you will do your implementation inside a simulated network environment. The simulated network has routers, clients (end hosts), links, and packets just like a real network. Files "`router.py`", "`client.py`", "`link.py`" and "`packet.py`" contain the implementations of a router, a client, a link, and a packet respectively. You must **not** modify any of these files, however, you may import the classes defined in these files and use their fields and methods for your implementation. You will do your implementation inside the file "`DVrouter.py`" (and "`LSrouter.py`" for Honors milestone). "`DVrouter`" (and "`LSrouter`") are subclasses of the "`Router`" class and inherit all its fields and methods while overriding its "`handlePacket`", "`handleNewLink`", "`handleRemoveLink`", and "`handlePeriodicOps`" methods.

"`handlePacket`" method is called every time a router 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 router or the cost of an existing link changes.

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

"`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 "`Router`" class method except for ones mentioned above. However, you are free to add new fields and methods to "`DVrouter`" (and "`LSrouter`" for Honors milestone).

Tip 1: For each router, all the links directly connected to that router are stored in the "`links`" data structure in "`router.py`". Whenever a new link is added or an existing link is removed or the link cost changes, **this information is updated automatically in the "`links`" data structure**, and the "`handleNewLink`" method (on link addition or link cost change) or "`handleRemoveLink`" method (on link removal) is called inside "`DVrouter.py`" (and "`LSrouter.py`").

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 routers, and CONTROL packets, that are generated and exchanged between routers to implement the routing 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`".

Tip 3: To implement DV (and LS for Honors milestone), you will need to exchange routing tables and neighbor lists between routers using the CONTROL packets. This will typically be implemented using data structures such as a dictionary or a list. However, a packet's content can only be of type String (refer to "`content`" field in "`packet.py`"). To convert a data structure to String, use "`dumps(data structure)`", and to convert the String back to the data structure, use "`loads(String)`".

5 Running the Code

You must run and test your code on `eceprog` using the environment from Lab 0. If your code does not run in that environment, you will not get any credit!

Start the simulator using the command,

```
$ python3 network.py [networkConfigurationFile.json] [DV|LS]
```

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`"). Use "`DV`" to run the Distance Vector implementation and "`LS`" to run the Link State implementation.

For example, to run the DV implementation with file "`01.json`", use the command:

```
$ python3 network.py 01.json DV
```

And to run the LS implementation (for Honors milestone) with file "`01.json`", use the command:

```
$ python3 network.py 01.json LS
```

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

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

6 Json File for Network Configuration

A sample json file specifying the network configuration can be found at "`01.json`". The "`router`" and "`client`" lists in the file specify the address of all the routers and clients in the network. Routers are addressed using numbers and clients are addressed using letters. Next, given n clients in the network, each client periodically (once every "`clientSendRate`" time period) sends out $n-1$ unicast DATA packets, one to each of the other $n-1$ clients in the network (excluding itself). A link is represented as `[e1, e2, p1, p2, c]` where a node (router 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`. You can assume that there will

be **at most** one link between any two nodes at any given time. Further, 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 may change the value of "`heartbeatTime`" in the json file to control the rate of periodic operations. The value of "`infinity`" in the json file specifies the cost of infinity for Distance Vector. 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.**

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`". However, you must not assume that all the initial links will already be added before the first trigger of "`handlePeriodicOps`" because of the reason mentioned above.

A Note about Addressing – At the network layer, the addresses tend to be *hierarchical* (e.g., IP address) to save space in the routing tables. But for simplicity, in this lab all the addresses are *flat*. So each routing table will need to store the information about every other router and host in the network.

7 Output

At the "`endTime`", the simulator cleans up all the active/queued packets in the network, and generates a last batch of 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 terminal. For your output to match the correct output, your solution must converge to the correct routing 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 any of the destination clients (probably because the packet was incorrectly dropped at a router by your solution or the packet is in a loop).

Tip: 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 result in massive queuing at the routers, which may not allow the packets to reach their destination before the experiment ends, which, in turn, may not allow the routing tables to converge to the correct entries before the experiment end time. This will result in either incorrect or empty paths in the final output! Note that this is an important consideration whenever you are implementing a network protocol in the real world as well — CONTROL packets must **not** overwhelm the network!

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 inside the "`logs/`" directory that contain information about each packet received by a router 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 may use these dump files to debug your implementation.

9 Grading

We will test your DV implementation against 10 test cases (json files). We may run the same test case multiple times to test the robustness of your implementation. **A test case run will be considered successful only if your entire output matches the correct output.** Each test case will be worth 1 point, and to receive full points for a test case, your implementation must successfully pass every run of that test case. Otherwise, your grade for that test case will be the percentage of test case runs passed. Your final grade will be the sum total of the points obtained across all test cases.

If you are also doing the Honors milestone, we will test both your DV and LS implementations against the same 10 test cases (json files). We will follow the same rules for grading as mentioned above, except now each test case will be worth 0.5 point for DV and 0.5 point for LS.

We have provided you with 3 out of the 10 test cases (json files) 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 final code must **not** print any custom / debug statements to the terminal other than what the simulator already prints, as this may break the auto-grader parser.

Violation of this guideline will result in a 20% grade penalty.

10 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 router 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 routing tables periodically during the code run, to see at what point the routing table entries do not match the expected values, and use that to debug your code.

11 Submission

You are required to submit **one** file "`DVrouter.py`" on Brightspace. For Honors milestone, you will submit an additional file "`LSrouter.py`" on Brightspace. **Do not submit a .zip file.**