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 Distance Vector (DV) routing and forwarding protocol (with Split Horizon and Max Infinity Counter). 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 existing link's cost can change. Your solution must be resilient to such changes.

3 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`". "`DVrouter`" is a subclass of the "`Router`" class and inherits 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`".

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`".

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: In DV, you will need to exchange routing tables between routers using the CONTROL packets. Routing tables and neighbor lists will be implemented using data structures such as a dictionary or a list. However, a packet's content can only be of type String ("`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)`".

4 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
```

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.

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

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

To clean the temporary files from the previous run of the experiment, run `./clean.sh`.

5 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 between routers can be added, updated, or removed dynamically during the simulation as specified in the "`changes`" list. You can assume that the link between a client and a router is never changed after initialization. An existing router to router link can be removed using the format `[t, [e1, e2], "down"]`, meaning the link between routers `e1` and `e2` is removed at time `t`. Similarly, a new link can be added between routers using the format `[t, [e1, e2, p1, p2, c], "up"]`, meaning a link between port `p1` of router `e1` and port `p2` of router `e2` is added at time `t`, and the cost of that link is `c`. The above format for link addition can also be used to update the cost of an existing router to router link. The "`handlePeriodicOps`" method is called periodically every "`heartbeatTime`". 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 specified in the "`endTime`" field. The provided json files

have "`endTime`" set to 1000 simulation time units, which translates to around 100 seconds in real time. **Hence, 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.

6 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. If the output route for a packet is empty, i.e., [], it is probably because your code incorrectly dropped that packet at some router along the path. If no route exists between two clients A and B in the final network graph (because the graph is disconnected), no route should be printed on the terminal for packets $A \rightarrow B$ and $B \rightarrow A$. For the routes output by your solution to match the correct routes, your solution must converge to the correct routing table entries at each router before the simulation "`endTime`".

Tip: Do not generate unnecessary CONTROL packets! Only generate them every "`heartbeatTime`" specified in the json file **and** when your local state *changes* (e.g., change in routing table in DV). 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, resulting in incorrect routes in the 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!

7 Debugging

The provided network configuration json files contain the "`correctRoutes`" between each pair of clients, which you can use to debug your implementation. If no route exists between a pair of clients in the final network graph (because the graph is disconnected), there will be no corresponding entry in the "`correctRoutes`". 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 course of the simulation. 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.

8 Grading

We will test your solution against 10 test cases (json files). For each test case, if the **entire** set of routes output by your solution matches the correct set of routes, you will receive 1 point, else you will receive a 0 for that test case. Your final grade will be the sum total of points obtained across all test cases.

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 to let you know which test cases 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.

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 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.

10 Submission

You are required to submit **one** file "`DVrouter.py`" on Brightspace. **Do not submit a .zip file.**