Eric Galego - galegoer
Vipul Taneja - tanejav1
William Li - liwill13

Mininet Analysis and Report

Implementation

Host

1. Initialization:
    - Once given an IP address, the Host will broadcast a packet to the router stating some general information like IP address, port number other info just to tell the router it exist
    - The packet the host send is constructed using the *make_broadcast_packet* function explained below
    - The host does have functionality that it can detect if no router is active.
2. Host KEEP-ALIVE (thread)
    - The host will then continue to send broadcast packets to the router every few seconds so that the router knows that the host is still active.
3. Router KEEP-ALIVE (thread)
    - the host will actively be receiving broadcast packets from the router, in order to determine if the router is still active. The router sends out an advertisement packet which we choose to use as the keep-alive for the router. Our understanding is the if router is making advertisements then its active

Multi-Router

- The router starts off by getting all its interfaces and starting a thread on each interface. Each interface thread will record its data into the forwarding table. *NOTE since OSPF and RIP protocol can work using a flag, we use forwarding table for RIP and neighbors for OSPF*
- Broadcast receiving and sending threads are initialized to send data to neighboring routers and to receiving data from monitor and neighbor routers
- Monitor node will send the topology for each router
- neighbor advertisements will update the forwarding table depending on the number of hops
- Various commands can be used inside the terminal for debugging purposes and display purposes:

  Multi-Router Commands List
    - PRINT FORWARD TABLE
        - This command prints out the forward_table. forward_table is mainly used for RIP protocol as OSPF using small_topology

- PRINT NEIGHBORS
  - This print the nearby neighbors of the current router
- PRINT HOSTS
  - This will print out all the hosts with their address and port number
- PRINT TOPOLOGY
  - This print out the topology which only store in memory the fastest routes to get to other routers/hosts in the network and all the distances to get the that node
- SET DELAY <int>
  - This will set the delay from current router to all the other routers
  - this command also prints out the delay that's set to confirm if the delay was set properly

Monitor

- Our implementation uses a monitor node that broadcasts a monitor_request packet to every active router node and this basically requests the router to send monitor their neighbor topology.
- Router will get this request packet and send the monitor node a monitor_response packet with its nearest neighbors and hosts connected with the delay to each node.
- Once the Monitor has processed all the responses, it sends a monitor_topo packet to each router which is the shortest path graph for each router.

Packets

a. Regular Packet
   i. make_packet: populates a python dictionary with information like the packets destination IP, source IP, TTL, protocol, message, and delay
   ii. This would be the typical packet sent between 2 hosts and all the routers in between.

b. Broadcast packet
   i. make_broadcast_packet: populates a python dictionary with information like, type (initializing or keep alive), the address, message, ttl , and router interface
   ii. These packets are sent to the router from the host to both initialize the host and to tell the router that the host is still active. for KEEP-ALIVE the type is changed.

ForwardTable

a. Advertise Packets

i. Populates and returns a dictionary with TYPE: ADVERTISE. Additional keys include FROM, TABLE and NEIGHBORS, with information regarding the router's ip, its forwarding table and neighbors. This packet is sent from each router to its neighboring routers.
ii. This packet is used to update neighbors on any changes that might have happened like a new host joining or new router connecting.
iii. These packets are also used by the hosts connected to the router to check if the router is still active.

b. Monitor Packets
   i. monitor_request: populates and returns a dictionary with a single key, TYPE: MONITOR_REQUEST. Used by the monitor node when requesting information from routers in the network
   ii. monitor_response: populates and returns a dictionary with TYPE: MONITOR_RESPONSE. The returned dictionary also includes the ip of a router, stored in the ROUTER_INTERFACE key and information regarding its neighbors, stored in the NEIGHBORS key. This packet will be sent from neighboring routers of the monitor node after they receive a TYPE: MONITOR_REQUEST packet.
   iii. monitor_topo: populates and returns a dictionary with TYPE: MONITOR_TOPO. Includes an additional key with topology information with the NEIGHBORS key. After this is created the monitor will calculate the shortest path topology for each router and add the information to the packet. The router then can load the new topology.

Design Decisions
What data structures and algorithms you used, and how they affect the efficiency and usability of your code

We decided to use dictionaries for our HOST_LIST, FORWARD_TABLE, and NEIGHBORS tables containing information on neighbors. Although dictionaries in python use more memory than lists, there are multiple instances where we need to access information or check whether a key (which in this case is an address of a router) is already inside our forwarding table so that we can, for example, update the forwarding table accordingly. Since python dictionaries are basically hash tables, this allows O(1) lookups when we try to access specific information in our forwarding tables, NAT or neighbors table.

Our forwarding table specifically contains key: dictionary pairs, the key being a router's address and the dictionary value containing a list of various *HOSTS (their addresses)* that connect to the router, the number of *HOPS* to reach the router and *SEND_TO* which determines the next hop ip in order to reach the router. Since we decided to use dictionaries for our key:value pairs, this will also allow O(1) lookups when we want specific information for a specific router address in our forwarding table. In general, dictionaries also make it easier for us to access specific info compared to another data structure like lists, since we can simply do something like *forwarding_table[<router_address>]['HOSTS']* to access the list of hosts connected to the *router_address* key.

To determine the shortest path our packet must go to reach its destination, we used dijkstra's algorithm with routing delay.

Packets follow a similar design choice with our host_list, forward_table and neighbors tables. Packet information is stored in a dictionary for ease of access and readability in our code compared to other data structures. Also we can easily send data by converting a dictionary into a JSON and sending that through the socket.

We have decided to use UDP over TCP as we were having trouble with TCP initially. One of the downsides of using UDP is that we are unable to know whether a router is not active anymore. We try to solve this by using keep_alive times and we check if the router is still active before we send off our packets. One of the smaller benefits of using UDP in our code is that there is no overhead due to forming connections before we send our packets, so UDP in a way is faster than TCP but comes at the cost of reliability.

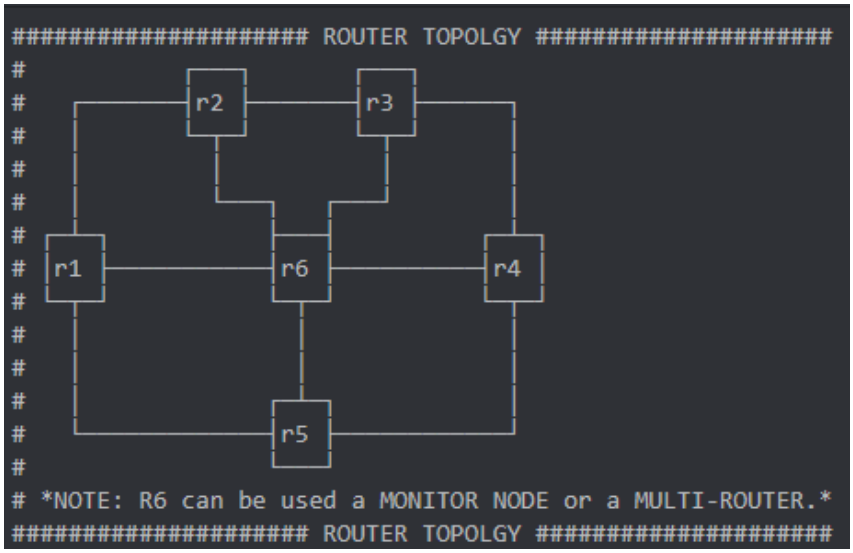
We are considering the delay between hosts and routers in the same network to be close to 0 and therefore we ignore delay in these scenarios.

Packets being sent from monitor node to router are all dictionaries that are converted to JSON and then converted back to a dictionary when it arrives at destination. Again dictionaries were used due to their fast lookup time and simplicity when trying to send them through a socket.

Testing

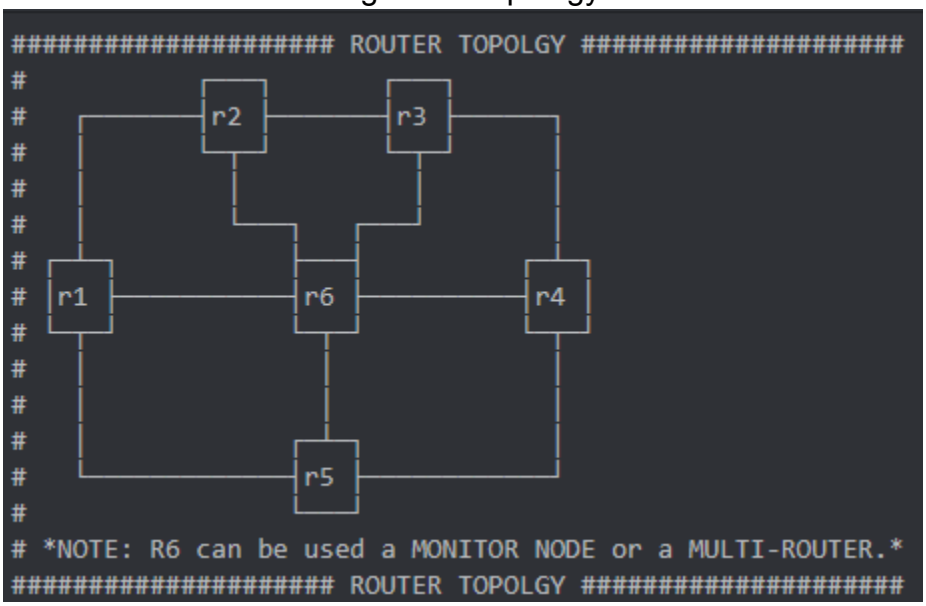
How you tested your code, the various topologies used and how you decided whether your code was working or not

- We first tested our multi-router with a simple topology called Linux-Router provided by mininet. We tested if we could send messages to Hosts within the network.
- Once the simple topology was successful we made a complex topology with 5 routers and 1 monitor node connected to all the routers.



- We manually opened each node in xterm then start routers, hosts and a monitor node on r6 specifically
- We started setting the delays for each router. So if r1 has a delay of 20 that means a packet going from r1 to any other connected router will have 20 delay added to it.
- Now we made a diagram mapping each delay to an edge and sending messages from one host to another.
- The first indication our code was working was that the delay output was correct. As each router had a different delay only one route could have a specified delay.
- The second indication that our router was sending packets to the right router was that we print every packet that arrives in the router. When we try to send different packets to different hosts, only routers who were supposed to get the packet, got the packet.
- The third indication was we did not see any packet go to other routers in wireshark.
- With these three indications we could definitely say that our multi router was working accordingly.
- We also use some router commands while testing to see everything was functioning well throughout our testing

- Test Cases:
  - We tested via the following router topology



  - TEST CASES
    - Delays:
      - r1 -> neighbors: delay 10
      - r2 -> neighbors: delay 10
      - r3 -> neighbors: delay 10
      - r4 -> neighbors: delay 5
      - r5 -> neighbors: delay 50
      - *clearly using r5 would not be a good choice*
  - LINK FROM R1 - R4
    - RIP: Delay 60
    - OSPF: Delay 20
  - LINK FROM R4 - R1
    - RIP: Delay 55
    - OSPF: Delay 25
  - LINK FROM R1 - R4 (REMOVED / KILLED R5)
    - RIP: Delay 30
    - OSPF: Delay 30

Our first test case traveling from R1 to R4 proved that our RIP algorithm decided on the path with the shortest numbers of hops which in turn resulted in a larger delay time (60) as it went through r5. OSPF on the other hand decided on the path with the shortest delay going through r2 and r3 to reach r4. This resulted in a delay time of 20 but 2 hops instead.

Our next test case was traveling backwards from R4 to R1. Similarly we noticed the delay for RIP was based on the shorter hop count with a delay of 55 while OSPF was 25. This delay is different then our first test case as R4's delay is set to 5.
Lastly we tested if removing our R5 node allows for both algorithms to decide on the path left and if it took into account that R5 is removed. We noticed both took R5 into account as being removed and both resulted in a delay of 30 as the only path left from R1 to R4 was traveling through R2 and R3.

Extra Credit Testing

- Implement sub-nets, including variable length masks and subnet-level broadcasting
    - Tested <BROADCAST> [message] on our hosts that were in the same subnet and those that were not. The ones that were in the same subnet received the message while those that weren't did not receive any messages.
- Allow nodes to disconnect from the network and reconnect elsewhere with the same IP
    - We tested this by connecting some hosts then disconnecting them. We then reconnected them somewhere else on the network or on a different interface of the same router. We validated that our router handles the disconnects and reconnects by printing the forward tables and host list and seeing if appropriate changes were made.
- Have non-uniform link costs based on traffic intensity
    - We tested this by setting random delays to each router then printing the neighbors to see if the table has been updated. We also tested this by sending packets throughout the network and tracing what the delays should be and if they matched our calculations.
- Simulate changes in bandwidth of links and broken/dropped connections
    - We changed the delay after having our router run for a while and seeing how our forwarding tables and neighbors data has changed by printing it. Also for dropped/broken connections we tested this by killing a router and seeing if neighboring routers can detect if the router is not active then we print forwarding tables and neighbors to check if they have been updated which they were.

Performance

An evaluation of the two routing algorithms in various networks, which works better and why and under what conditions each is optimized. This section should have a conclusion with details about which routing algorithm makes the most sense for this network, and under what circumstances your decision would be altered to the other algorithm.

Algorithms

- **OSPF**
  - **Pros:**
    - OSPF has knowledge of entire network topology. Routers only store the shortest paths to neighboring nodes
    - OSPF has instant updates
  - **Cons:**
    - OSPF has a big flaw where every small update would require every router topology to be recalculated. So if new host was added then all router topologies in the network would have to be updated
- **RIP**
  - **Pros:**
    - RIP does not require an update every time the network topology changes. As it is periodically updated.
    - RIP does not require the topology to be recalculated every time there is a new update; instead some tables would just need a small update which is nowhere as expensive as calculating topologies using Dijkstra's algorithm.
  - **Cons:**
    - RIP can create a traffic bottleneck as it broadcasts its updates periodically.
    - If a delay increases then RIP faces issues finding the optimal router as Bad News Travels Slow throughout the network.

Although OSPF calculates the optimal route, it comes at a very big price which has to do with the computation of finding all the shortest paths for a router. Dijkstra's runtime is O(|E|log|V|) and to find each routers shortest path it will take V*O(|E|log|V|) so O(V*|E|log|V|). Now this is fine if the topology is not updated but suppose you add a host then because of this small change every routers' table needs to be updated.

Although RIP does not have a big problem with heavy computation the issue with RIP is having bad news travel slow and the fact that a new update will take some time to reach all the routers in the network. As updates are done in variable length time.

Overall for our current network the ideal algorithm would be OSPF because it is a relatively small network that does not make constant changes and even if there is a change it is not that bad as again the network is very small. However if this network got bigger and had many changes going on then we would prefer RIP as there is less overhead recalculating the topology each time there is an update.

Extra Credits (All Completed)

- Implement sub-nets, including variable length masks and subnet-level

broadcasting [5%]
    - Subnet-level broadcasting is achieved using <BROADCAST> [message] inside our host terminals. This will send a message to nodes on the same subnet by checking their netmasks
- Allow nodes to disconnect from the network and reconnect elsewhere with the same IP [5%]
    - Router can detect that a host/another router has disconnected/inactive, and that IP can be reused by another host elsewhere on the network
- Have non-uniform link costs based on traffic intensity [5%]
    - Setting delays of routers to different values. Will only affect OSPF. RIP uses hop count values and not delay values. Can be set by using command “SET DELAY ##”. Once updated, this update will be sent to Monitor node and neighbors via advertisements
- Simulate changes in bandwidth of links and broken/dropped connections [5%]
    - We can set delay anytime on any router and the router's connection can drop and the neighboring router will detect this and send a message to the console. The router updates it tables

References:

https://www.codegrepper.com/code-examples/python/python+send+data+to+ip+address
https://code.activestate.com/recipes/578802-send-messages-between-computers/
https://stackoverflow.com/questions/50867435/get-subnet-from-ip-address
https://stackoverflow.com/questions/819355/how-can-i-check-if-an-ip-is-in-a-network-in-python
https://community.fs.com/blog/rip-vs-ospf-what-is-the-difference.html
https://github.com/mburst/dijkstras-algorithm
https://github.com/joyrexus/dijkstra