Michael Lemmler

Assignment 4 - Question 4 - SDN Design Document - 5/9/25

Architecture

The core script only requires the ‘matplotlib’ library in order to visualize the network. All network and topology functions are implemented by hand without the use of any libraries.

Topology Graph

- The main component of my topology graph is the NetworkNode class (line 59).
- Network nodes contain:
  - a graph position (for visualization purposes),
  - a debug name,
  - a list of link interfaces (represented by indices to other nodes in the global ‘all_nodes’ array),
  - A forwarding table, which holds a reference to the link interface needed to hop along the shortest path to carry a packet to its destination,
  - And a second ‘cost’ table, which maps directly to the forwarding table and provides cost information for each destination node.
- Internally, network nodes are all spaced at a distance of ‘1’ from each other. Their positions on the graph are just for visualization purposes and have no impact on the actual routing.
  - The implementation could be extended to support different distances for nodes, however this SDN controller is just a prototype.
- There is also a global ‘link_stats’ array, which is used to track how many times a link between two nodes has been used in order to allow for load balancing.
  - The current ‘times_used’ counter of each link is displayed on top of the links in the GUI

Path Selection

- In order to simulate a packet in the ‘simulate_packet’ function (line 323),
  - For the current node, I choose an ‘exit interface’ that the packet should travel through.
  - This is done by calculating link weights for each interface.
  - For high priority packets, the only weighted link is the path that will reach the destination in the least number of hops.

    - For low priority packets, I offset the link weights by 0.25 * the ‘times_used’ counter along each link, added to the total distance for each interface.
        - As a result, low priority packets will initially take the shortest path, but as this path becomes congested packets will begin to take alternate routes through the network. This creates a load balancing effect.

<u>Forwarding Tables</u>

- The forwarding table of a node contains an entry for each node in the network (including us).
    - For unreachable nodes, and the node itself, the forwarding table entry contains a value of ‘None’.
    - For nodes that can be reached, the forwarding table contains the index of the link interface which will reach that node in the shortest number of hops (fwd_table[dest_node]).
    - Therefore, links[fwd_table[dest_node]] represents the index of the next node in the path. This is used in the ‘simulate’ command to display the path that a packet takes.

<u>Route Reconfiguration</u>

- When a node is added, a node is removed, a link is added, or a link is removed, the update_network_topology() function is called, which reconfigures the entire network based on the new topology.
    - This consists of recalculating the forwarding table for each node (calc_shortest_paths) and then some helpers to redraw links in the UI.
- Effectively, this reconfigures all potential routes through the network. In the simulator, this happens instantaneously, however in a real OpenFlow scenario this behavior would obviously be somewhat delayed, as the controller propagates the new forwarding tables to each router in the network.
- Link failure can be simulated through the “unlink” and “rem” commands in order to drop links and nodes respectively.

<u>CLI interface</u>

- The command-line interface supports the following commands with the specified syntax (all node names are case-sensitive):
    - **add** <x> <y> <name>
        - Creates a node named “name” at the specified (x, y) position.

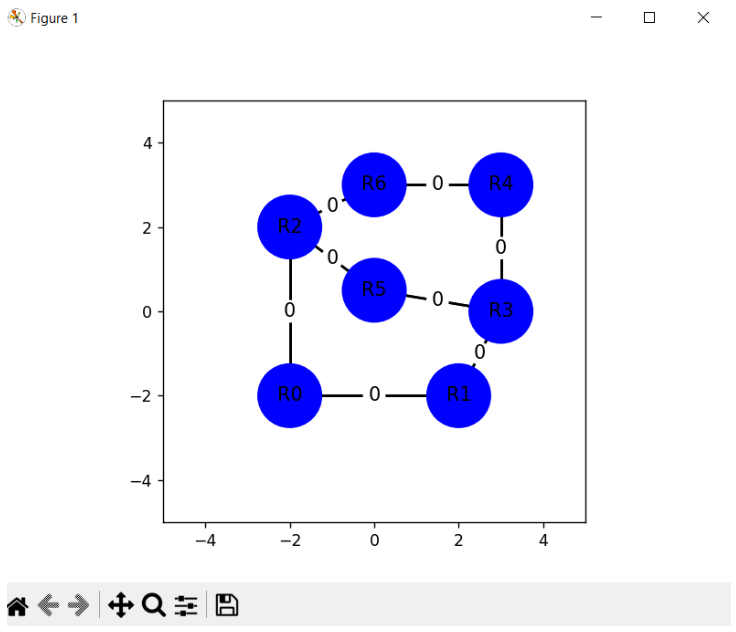
    - Ex: add 4 4 R6
- **rem** <name>
    - Remove the node named "name" if it exists.
    - Ex: rem R6
- **link** <name1> <name2>
    - Create a link between node "name1" and "name2".
    - The command will fail if a link already exists or one of the nodes cannot be found.
    - Ex: link R2 R3
- **unlink** <name1> <name2>
    - Removes the link between "name1" and "name2".
    - The command will fail if there is no link between the nodes, or one of the nodes cannot be found.
    - Ex: unlink R2 R3
- **print** <name>
    - Print the local topology of node "name".
    - The command will output the list of link interfaces, followed by the forwarding table of the node.
    - Ex: print R2
- **simulate** <src> <dst>
    - Simulate a packet traveling from "src" to "dst".
    - At each node along the route, the command will print the link weights that contributed to the routing decision, and the interface that was chosen as a result of those weights.
    - The command will fail if no path can be found between the two nodes.
    - After the command has run, the GUI will update to show the new 'times_used' statistic for each link (this will contribute to future routing decisions)
    - Ex: simulate R2 R3
- **simulate_high_priority** <src> <dst>
    - Simulate a high priority packet traveling from "src" to "dst".
    - This command has the same behavior as the simulate command, with the exception that the packet will always take the shortest path

between the nodes. It will not evaluate alternate routes or attempt to load-balance.

  - Ex: simulate_high_priority R2 R3

Visualization

- The visualization component uses the ‘matplotlib’ Python library.
- Each node is represented by a blue circle with the node name over it.
- Links between nodes are represented by black lines between the nodes.
- Each link is annotated with the number of times that link has been used (this indicates which routing decisions will be made)



Algorithms

Shortest Path

- For the shortest path selection, I wrote a simple implementation of Dijkstra’s algorithm. There are other algorithms which can calculate a number of paths based on their cost metrics etc., which would be better suited for a real OpenFlow implementation.
- Since this is only a prototype, I went with Dijkstra’s as it’s relatively simple and you can manage to do simple load balancing with a few small tweaks.

- To calculate the forwarding table for a node, I maintain the shortest distance path to each node (shortest_dist, initialized to a very large number, 1000, at the beginning of the algorithm).
- As the algorithm runs, I mark nodes in the table as visited, and enumerate through their neighbors to see if a shorter path is found. If one is found, I update shortest_dist to reflect this and also keep track of each node's predecessor within the "prev" array.
- Once the algorithm has completed, I calculate the full path to each node in the table (get_path_to_node), and use the first hop in each path to build the final forwarding table.

<u>Load Balancing</u>

- Load balancing is done in the "choose_exit_interface" function.
- First, I evaluate the "ideal" interface. This is the link interface that will result in the shortest path across the network if the packet exits along that link.
- Then, I look at each neighbor node. If the node has a different route to the destination that the packet could also travel along (with the added cost of an extra hop), I add this neighbor's interface to the candidates list.
- When all neighbors have been explored, I build a link_weights array for every outgoing link on this node.
    - Links that do not produce a valid path to the destination are given the special weight '-1' to indicate that they must be skipped
    - Other links are given the weight of (in pseudocode): times_used * 0.25 + interface_cost
- In the "simulate" command, the interface with the smallest weight is chosen as the next hop for the packet.
    - For example, on the default network provided when the application starts, if you run "simulate R2 R3" four times, the packet will take the path R2 → R5 → R3, as that is the shortest path.
    - However, once this path has been congested (used four times), the cost of this interface will have increased, and so the network will start load balancing the packet to the R0 and R6 interfaces (taking the paths R2 → R6 → R4 → R3, or R2 → R0 → R1 → R3).
    - Although these paths require an extra hop, they will reduce congestion on the network.

Uniqueness Verification

- One challenge I faced when implementing this was getting load-balancing to work. Dijkstra's algorithm only computes the shortest path to a node, and doesn't provide you with any alternate routes based on cost etc. There are some other algorithms besides Dijkstra's which can provide this, but I figured they would be too complex for this simple prototype.
- Instead I rolled my own solution (detailed in the "load balancing" section above) which just glances at potential alternate routes for the packet during simulation rather than during network configuration. A real SDN implementation would probably have a more involved solution to this problem.
- Initially I just built the forwarding table in the calc_shortest_paths() function. To support load balancing, I added the 'fwd_distances' member to my node class which is used by routing to evaluate the potential path costs.
- In my first implementation of packet simulation, I was just routing packets based off of the forwarding table. In order to do load balancing, I replaced this with the choose_exit_interface() function which uses the link weights I discussed earlier to determine a good route for the packet to take.
- The hash of my student ID (I hope I calculated it right?): 859f931871d5fc4dc6b44361c8574947071714045d301d703ffd3b13af59b770
- I also put this as a comment in my controller script on line 7.