Project Final Report

CS2520: Wide Area Networks

Kadie Clancy and Brian Falkenstein

Please refer to the README.md file as the user manual for our project.

Programming Language

We implemented our project using python as the programming language. We utilized the python socket API for low-level network interfacing. Neither one of us had experience with socket programming prior to this project. We felt that we learned a lot about programming with sockets and how networks route data in general by implementing the sLSRP via this project.

User Interface

We proposed to implement a command line interface to be utilized at the client level. We implemented the client as proposed. A command line interface allowed us to focus on the functionality of the sLSRP and the file transfer system as first priority.

Bootstrapping and Initial Setup

Our network depends on a single central connection router in `conn_router.py`. This connection router reads in the network configuration file. When network routers come online, they first connect to this connection router to obtain neighbor information. Note that this connection router is only used for neighbor information distribution and is *not* a part of the overall network topology (i.e. this router will not be used to forward packets through the network). To run the network, first run `conn_router.py`. Then other routers can come online by running `router.py` with arguments `router_id, IP, port` that refer to the router ID in the config_file, the IP address of that router and the assigned port number respectively. Each router will receive its neighbor information from the connection router and subsequently start the neighbor acquisition process. The neighbor acquisition process is run via a thread that listens for its neighbors to come online. Once the router has successfully connected with all its neighbors, the router's local link state database is updated and the initial routing table is constructed. This completes the initial setup.

At any time after the initial setup is complete, a client can connect to a router. This is how the user interacts with the network via a command line interface. The client commands include:

1. Viewing control information for the corresponding router
2. Sending a file to a router in the network
3. View incoming messages

4. View the routing table of the corresponding router
5. View the shortest paths of the corresponding router
6. Exit the client
7. Help

Flow Control

In our design report, we proposed to implement Stop-and-Wait flow control to ensure that all packets are received in proper order. We implemented Stop-and-Wait in our implementation by sending one packet at a time and waiting for an ack reply before allowing any further packets to be sent. This form of flow control is simple, efficient, and ensures that all packets are delivered to the receiving router for reassembly which is why we originally proposed it for our design.

Network Activity

In our implementation, we use threads for packets that need to be sent periodically. Every `alive_interval` number of seconds, each router pings all of its neighboring routers to determine the delay. Each neighbor will reply with an ack packet. Each time a new delay is calculated, it is also updated in the local link state database. We chose to use the round trip time as our delay metric so that we are able to account for both time for the packet to travel the path and queueing delays. Instead of sending a separate alive message to all neighbors, we relied on our pinging of neighbors to calculate delay to tell us whether a link is alive or dead via a timeout on waiting for an ack.

Using a seperate `update_interval` (12 sec in our implementation) each router sends its local link state (which contains information on the delays to its neighbors) and the mapping of IP/port to link state indices to all of its neighboring routers. When a router receives such a link state advertisement, it updates its routing table using this link state and mapping and then forwards the message to all neighbors aside from the one in which it received the link state advertisement.

End users will occasionally send files to routers through the client. Thus, routers will occasionally receive files. For a user to send a file to a router in the network, the user must know the port number associated with that router. The user will interact with the command menu in `client.py` to input the message and receiver. The message will be transmitted through the network to the receiver and then that receiver will send an ack packet back to the sender. The receiving client can view incoming messages via the command line interface.

Error Checking

As part of the project requirements, we introduced error to packet contents, specifically 1/10 packets will contain an error. We implemented an error by randomly changing one of the characters in the packet. To protect against this, we had to implement error checking.

To check for packet transmission errors, the packet layer of the sending router will compute the checksum of the packet contents and append it to the end of the packet. The packet layer of the receiving router will compute the checksum of the packet and check this value against the checksum provided. If the values do not match, the function will alert the router that something is wrong with the packet and the receiver will send an error message requesting that the corrupt packet be present. Error generation and checking is implemented in the packet class of `packet.py`.

Routing Algorithm Implementation

We proposed to use a matrix-based implementation of Dijkstra’s and this is what we implemented. While we are aware of the shortcomings of a matrix-based compared to an adjacency-list representation, it was more intuitive to use a matrix when adding new link state databases to the routing table. We also believe that since we are operating this simulation on relatively small networks, performance degradation will be minimal.

Data Structures

In our design proposal we indicated that our LSDB will be implemented as a hash table for efficiency. In our actual implementation, we used a python dictionary within a link state object. This choice came down to time constraints and ease of implementation. The link state object is responsible for creating only local link states that can be added to the routing table.

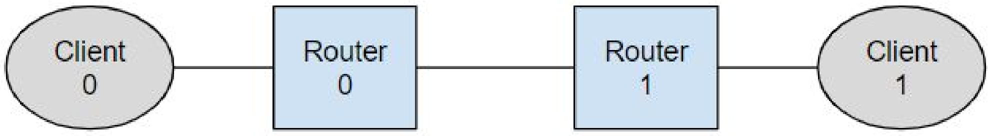
We also proposed to implement our routing table as a hash table. For the same reasons as above, we actually implemented it as a dictionary within a routing table object. When a new link state object is received from another router, it is given to the routing table object where the internal adjacency matrix is updated, dijkstra's is computed and then the shortest paths and corresponding path weight are stored in a dictionary indexed by router number. This router number is assigned internally to a router and is kept track of by a mapping dictionary. If we had more time to work on this project we could improve our implementation by using more efficient data structures.

Packet Fragmentation

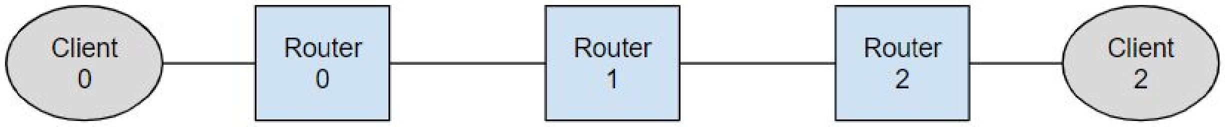
In our implementation, we impose a max packet size of 10 characters. If a message to be sent from one client to another is greater than 10 characters, the message is fragmented and then sent (in a stop-and-wait fashion) to the receiving client. The receiving router reassembles the message, which arrives in order due to the flow control used, before passing to the associated client. We assume all control packets (link state advertisements, neighbor acquisition and alive messages) are smaller than the max size and are thus not fragmented.

Test Cases

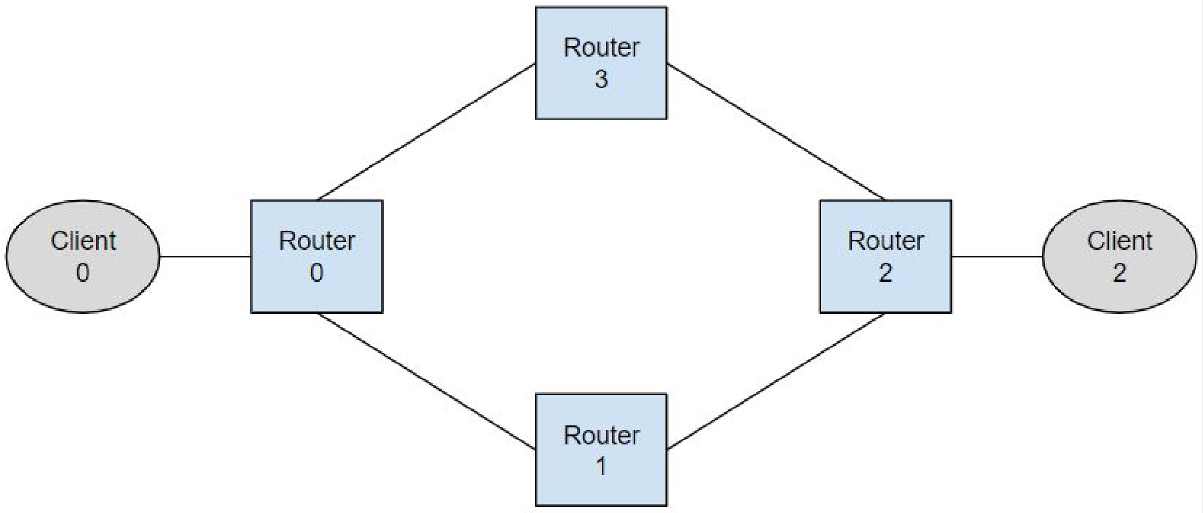
We implemented our network initially using the simplest case of a network, just two routers connected with one link shown below**.** This allowed us to implement the network functionality in a controlled way before testing more complex networks. This network is in `config_file_simple.txt`



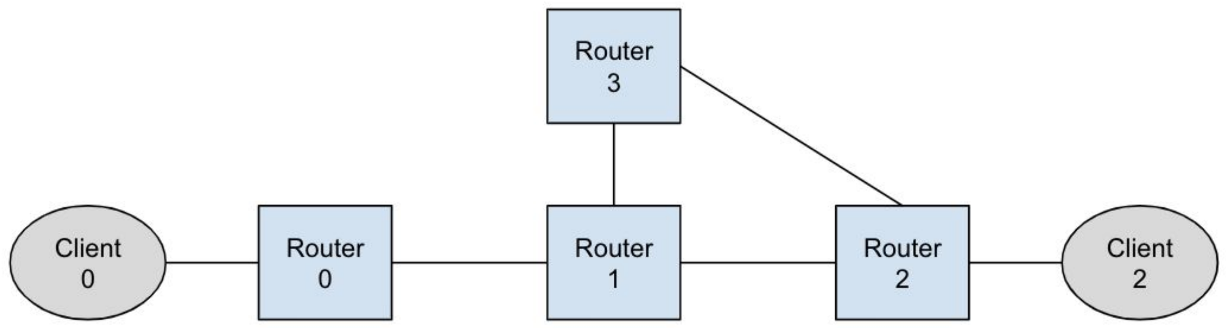
We also included more complex networks to demonstrate our implementation on different network structures. Instructions on how to run each test case is found in the README file. First, we have a line network shown below. This network is found in `config_file_line.txt`.



Next, we have included a circular network to test loops shown below. This network is found in `config_file_circle.txt`.



Finally, we have a more complex combination network shown below. This network is found in `config_file_combo.txt`.



* Note that we provided these premade test cases for demonstration purposes and our implementation should work with any network structure.