Lab 7
Link State Routing + Dijkstra

COEN 146L Spring 2021

Reminders

- All the labs are due at 11:59 PM on the next Monday after the lab has been assigned
- TA office hours:
  - Rachael (rfreitag@scu.edu)
    - Monday: 12 - 1 pm
    - Tuesday: 2:30 - 4 pm
    - Thursday: 11 am - 12:30 pm
  - Jesse (jschen@scu.edu)
    - Monday: 1 - 2 pm
    - Tuesday, Friday: 5 - 6 pm
  - Or by appointment

Lab 7 Objectives

- Concept: Implement Link State Routing Protocol
- Concept: Implement Dijkstra's Shortest Path Algorithm
- Programming: Write a program that simulates the behavior of a router/node in a network.
    - network nodes and network topology are given to you in files.

Concepts

- UDP Socket Programming: Lab 5, 6
- Multithreading: not new
- Link State Routing Protocol: new!
- Dijkstra's Shortest Path Algorithm: new!
- Mutexes: new!

Link State Routing Protocol

- Each node in the network maintains a connectivity map of all nodes in the network
  - node A knows that it takes X time-interval to get from node B to node C
- To accomplish this, each node informs the network ***when the state of its links change***
  - if the time to get from node A to node B changes, they let *everyone* in the network know of this change
  - everyone in the network can now update their connectivity map

Dijkstra's Shortest Path Algorithm (pseudocode)

- Data:
  - array of visited nodes N' (should be empty at first)
  - array of all nodes in network N (should have all nodes at first)
  - source node u
  - define C(x,y) as the cost of the link between nodes x and y (should already have this information)
  - define D(v) as cost of path from source u to dest v
  - define P(v) as predecessor node along path from u to v
- Initialization:
  - remove u from N and add to N'
  - D(u) = 0
  - for each node v in N:
    - D(v) = C(u,v)
    - P(v) = u
- Loop:
  - while N isn't empty:
    - let n be the node in N with shortest distance from u
    - remove n from N and add to N'
    - for all nodes v adjacent to n and not in N':
      - D(v) = min(D(v), D(n) + C(n,v))
      - if D(v) == D(n) + C(n,v):
        - P(v) = n
- **Result: for each node v in the network, D(v) and P(v) will now tell you what the shortest path to that node is**

Mutexes - `pthread_mutex_t`

- marks off a sections of code that can only be accessed by one thread at a time
  - if thread 1 calls mutex_lock(), other threads' call to mutex_lock() will block/hang until thread 1 calls mutex_unlock()
  - is used to synchronize reads/write to data structures that are shared between threads
    - don't want a data structure to change halfway while reading it

`pthread_mutex_t` - how to use

1. declare variable
   - `pthread_mutex_t lock;`
2. initialize lock
   - `pthread_mutex_init(&lock, NULL);`
3. acquire/wait for lock
   - `pthread_mutex_lock(&lock);`
4. modify shared variables
5. release lock
   - `pthread_mutex_unlock(&lock);`

*** example.c on the camino is an example of how to use locks ***

`costs` file

- file is formatted as an N x N matrix
- position (i,j) is the cost of the link between node i and node j
  - is a symmetrical table because the cost of (i,j) should be the same as the cost of (j,i)

```
0       1       1       1000
1       0       1000    1
1       1000    0       1
1000    1       1       0
```

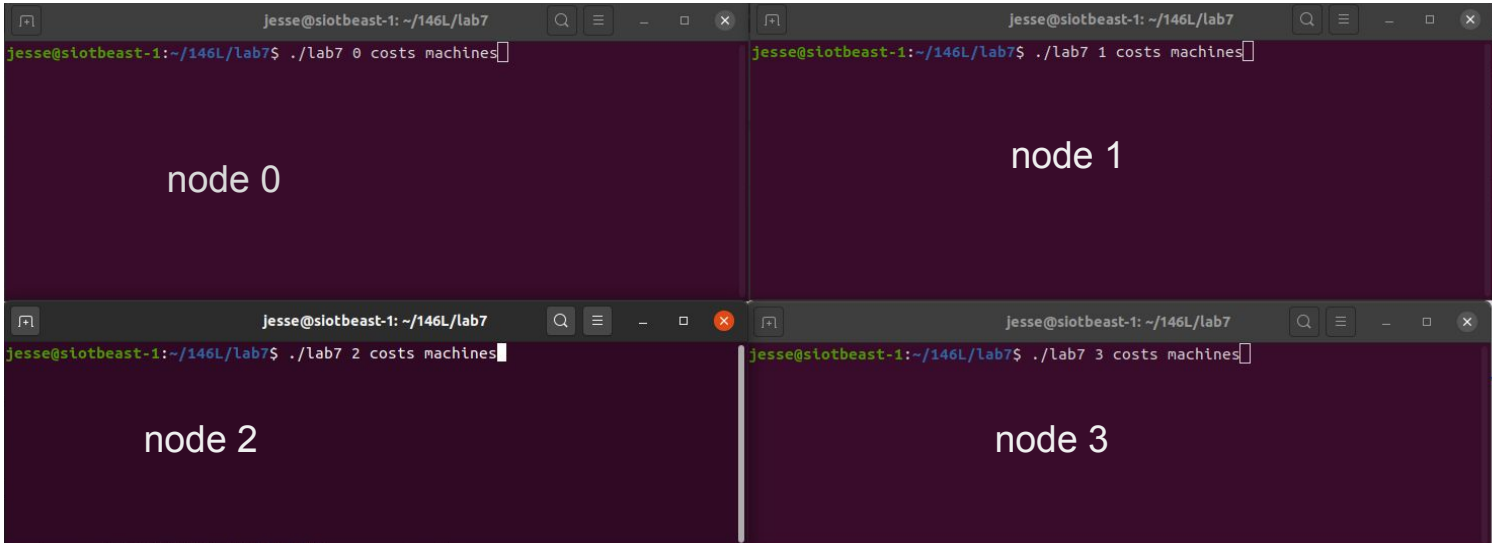
`machines` file

- list of information about all nodes in the the network
- each node is formatted as "<machine_name> <ip_addr> <port_num>"

```
linux60808 129.210.16.78 5000
linux60808 129.210.16.78 6000
linux60808 129.210.16.78 7000
linux60808 129.210.16.78 8000
```

Program Structure

- Each node in the network is a separate process/terminal on your machine
  1. node id is specified as a command line parameter
  2. each node id corresponds to a node in the `machines` file
     - each process should use the same `costs` and `machines` files



Program Structure - threads

```
received data, updating cost table...
local cost table updated: (1, 3):5
0 1 1 1000
1 0 1000 5
1 1000 0 1
1000 5 1 0
```

receive_info

- Each process will have 3 threads:
  1. receive_info: listens for link state updates from its neighbors
     - when an update is received, update the cost table
       - update message format is: "<node_id><neighbor_id><new_cost>"

run_link_state

```
**new shortest distances**
distance to node 0: 0
distance to node 1: 1
distance to node 2: 1
distance to node 3: 2
```

  2. run_link_state: loops forever
     - re-runs Dijkstra each loop to keep distances array updated
     - wait for a random amount of time between 10-20 seconds after each loop iteration

  3. user_update: gets input from stdin(the user) about changes to the cost table
     - users can manually update the costs to each node
       - input format is: "<neighbor_id> <new_cost>"
     - will also update other nodes about these changes
       - update message format is: "<node_id><neighbor_id><new_cost>"

```
enter path cost changes: 3 5
local cost table updated: (1, 3):5
0 1 1 1000
1 0 1000 5
1 1000 0 1
1000 5 1 0
sent update to linux0
sent update to linux2
sent update to linux3
enter path cost changes:
```

user_update

*** your outputs/print statements do not have to match these, but they need to make sense ***

Steps

1. finish the lab
    a. set up your code to read contents of the files and start the threads
    b. recommendation: do **receive_info** and **user_update** threads first
        i. those two threads work in conjunction with each other so you can't really test one without the other
    c. **run_link_state** thread can run by itself
2. demo lab
    a. for demo, I will ask to see one terminal for each node