1 Introduction

Goal: Using NetSim, a network simulator written in Python, develop and demonstrate the key elements of a simple link-state routing protocol. The pre-lab and lab teach the following ideas:

1. Understanding how HELLO packets allow the nodes at the ends of a link to maintain accurate neighbor and link status information.
2. Understanding how to flood link state advertisements (LSAs) between nodes.
3. Developing the code for the shortest path computation at a node by first processing the LSA information to form a graph, and then running a reasonably efficient implementation of Dijkstra’s shortest path algorithm on the graph.
4. Understanding and demonstrating how the implemented protocol responds to link failure and recovery.

Instructions:

1. Complete the pre-lab activities in Section 5.
2. Complete the lab activities described in Section 6 in lab on Wednesday.
3. Prepare the requested material and think about the questions posed in the Checkoff Sheet, then find a staff member to complete your post-lab interview.

2 Lab Setup

In this lab you will develop a simple link-state routing protocol in NetSim, a network simulator written in Python. We’ll use NetSim in the next lab as well (in which you’ll develop a reliable data transport protocol). Like the broadcast network simulator in Lab 9 (WSim), NetSim uses wxPython. If you’d like to try out the simulator on your own machine you’ll need to install:

Python from http://www.python.org/download/

There are pre-built binaries for Windows, Mac OS X and Linux.

As in Lab 9, we’ll use the IDLE integrated programming environment to develop and run our Python programs.¹ You can start IDLE by running setup 6.02 and then idle-602 at the Athena prompt.² Use File→Open to open a source file for editing; hit the F5 key to execute the current editor buffer.

Let’s first set up our Lab 10 environment:

¹IDLE isn’t required; you can also use python-602 from your shell.
²Our version of IDLE is currently only available under Linux; you will probably need to come into lab to run it.
In what follows, we will refer to your version of the code as lab10.py. After running setup 6.02, idle-602 and python2.5 should both be in your path.

### 3 Overview of the Network Simulator

NetSim is a packet-level network simulator. You will be modifying the code that defines the behavior of the nodes—what packets they send, how they respond to incoming packets, and what computations they perform. NetSim is implemented by the class definitions found in network.py and includes a graphical front-end. If you read lab10.py into IDLE and hit F5, you’ll get a window that looks like:

The top part of the window shows the network topology (i.e., the graph), which contains:

- **Nodes (black squares):** These are instances of the Node class and represent named network nodes.

- **Links (black lines between nodes):** These are instances of the Link class and represent bidirectional communication links between the nodes at either end of the link. Each link has two queues to hold the packets flowing in each direction.
• **Packets (colored circles):** These are instances of the Packet class and represent the messages that will be sent from a source node to a destination node.

Underneath the topology is a row of clickable buttons that control the simulation:

• **Reset:** return the network to its initial state.

• **Step N:** advance the simulation by 1, 10 or 100 time steps.

• **Exit:** terminates program.

A status bar appears at the bottom of the window, showing:

• **Time:** Current simulated time.

• **Pending:** A count of packets in the link queues and the transmit queues of the nodes. The simulation ends when this number goes to 0.

• **Total:** The total number of packets that have been created during the simulation.

• **Status:** When you move the mouse near a node or packet on the map the appropriate status message appears here. You can provide a status method for these objects that will display any information that you deem useful.

### 3.1 How the Simulation Works

NetSim simulates how packets flow through the network, traveling from node to node via the interconnecting links. The network being simulated is actually an instance of the `MyCostNetwork` class, which is also defined in `lab10.py`. At the beginning of the simulation, the `reset` method of `MyCostNetwork` calls the `reset` method for the parent class to perform the usual initializations. It then inserts a single packet into the network, choosing a random source and destination.

NetSim simulates the behavior of the nodes and links of the network one time-step at a time. Here’s what happens in each time step (this information is provided so you understand the overall structure of the simulator; you won’t need to change any of this structure):

1. The simulator calls the `phase1` method of each node, which in turn calls the `receive` method of each link connected to the node, which returns the first packet that was queued on the link for delivery to the node.

2. The simulator then calls the `phase2` method of each node to process the packets collected from each link during phase 1. Processing proceeds in two sub-steps:
   (a) Call the node’s `process` method for each packet, which in turn calls either the `receive` method if the node is the packet’s destination, or the `forward` method if the packet should be passed on to a neighbor for eventual delivery to its final destination. The `forward` method chooses a link to use by consulting the routing table, `self.routes`, then calls that link’s `send` method to actually add the packet to the appropriate queue within the link.
(b) After all the incoming packets have been processed, call the node’s `transmit` method, which can produce new packets to be delivered by the network. The default `transmit` method maintains a queue of packets that are introduced into the network when the packet’s declared start time is reached. The transmit queue is filled by calls to the node’s `add_packet` method during network initialization.

As in real-world networks, the only information a node has about other nodes in the network is information that has arrived in packets. Each node must use this information to construct its routing table.

The following features of NetSim are worth understanding for this lab:

- The source and destination fields of each packet contain addresses. An address is a string that uniquely identifies each node in the network.

- `Link.receive` returns a 2-tuple `(link, packet)`, so each incoming packet can be associated with the link it arrived on.

- The `Packet` class has the following:

  ```
  source -- node where this packet originated
  destination -- node where this packet should be delivered
  start -- time at which packet left source node
  properties -- a dictionary mapping property names to property values
  arrived_from() -- node that this packet just arrived from
  ```

- `Node` has some useful instance variables:

  1. self.neighbors: a dictionary that maps a link to a 3-tuple:
  2. self.LSA: a dictionary that maps an address to an N-tuple.
  3. self.LSA_seqnum: an integer used to uniquely identify LSA broadcasts.
  4. self.routes: a dictionary that maps a (destination) address to a link, which is the link that will be used to forward packets to that destination.
  5. self.Hello_INTERVAL: the number of time steps between `HELLO` packets.
  6. self.LSA_INTERVAL: the number of time steps between LSA broadcasts.

- `Node.process(self, p, link, time)` is called with the incoming packet, the current time, and the incoming link.

- `Node.forward(self, p)` uses the packet’s destination address to consult the `self.routes` dictionary to determine which outgoing link to use when forwarding the packet.

- **Note:** You can click on a node to invoke that node’s `OnClick` method. *MyNode’s* implementation for that method prints out the contents of the node’s neighbors, LSA, and routes dictionaries. *This feature will be useful in debugging your protocol and in demonstrating that it works properly.*

- You can click on a link to change its state from OKAY to BROKEN. Broken links are displayed with a big red “X” and drop any packets sent along the link.

- To send a packet along a link, use the following function in the `Link` class:
send(n, p) -- send packet to other end of link

• The Network class has a useful function:

make_packet(src, dst, start, **props) -- make a new packet

The above list summarizes the various functions you need for the lab, but there’s a small chance it isn’t complete. Please look through network10.py to see what other functions are defined there in the Node, Link, and Network classes (some of these functions are overridden in the MyNode and MyCostNetwork classes).

4 Overview of the Pre-lab and Lab Tasks

Our goal is to build each node’s routing table using link state information received from other nodes in the network. We’ll rebuild the routing tables periodically using updated link state information. There are three steps involved in this process:

1. (Pre-lab) Have each node periodically figure out which of its links are currently operational and thus who its current neighbors are: we’ll do this task by having each node periodically send special HELLO packets on its links. The source addresses of incoming HELLO packets can be saved to make list active links and the neighbors’ addresses. The HELLO packet is distinguished by its destination field being set to “HELLO”.

   Code in Node.process checks for the presence of the special destination (“HELLO”) and processes the packets accordingly.

2. (In lab) Have each node periodically broadcast an up-to-date list of its neighbors’ addresses and link costs. This list is called a link state advertisement (LSA). Every node saves the incoming broadcasts and thus learns about the current neighbors for all the other nodes in the network. Each node also rebroadcasts any received LSA to its own neighbors. An LSA packet has destination field “LSA”.

   Code in Node.process checks for the presence of the special destination (“LSA”) and processes the packets accordingly.

3. (In lab) Have each node periodically rebuild its routing table by running Dijkstra’s algorithm on the neighbors data collected in step 2, using the current node as the root of the search.

Link states can change (in this lab we’ll do that by clicking on the links in the graph to have them toggle between “OKAY” and “Broken”), and the routing tables will be need to be updated to reflect those changes. We’ll do the updates by periodically rebuilding the routing tables from scratch.

The next two sections detail the pre-lab and lab activities, respectively.
5 Pre-lab Exercises

For the pre-lab, you should:

1. Get familiar with the structure of the simulator, its classes and methods.
2. Complete the first step, which involves a small amount of coding.

Step 1: Maintain an up-to-date list of neighbors’ addresses

In this step our goal is to have each node keep an up-to-date list of its neighbors’ addresses. To help with this task, we will use `self.neighbors`, a dictionary that takes a link as a key and returns a 2-tuple (timestamp, address) containing the address of the neighbor at the other end of the link. The timestamp is used below to identify entries that have not been updated recently and should hence be removed from `self.neighbors`.

**Step 1(a).** First add some code to `MyNode.transmit` that periodically (every `self.HELLO_INTERVAL` time steps) sends out `HELLO` packets along each link. A `HELLO` packet is distinguished from other packets by having a destination of “HELLO”. It doesn’t have any content, since the neighbors will only be interested in the source address field of the packet that contains this node’s address. Set the color property of the packet to be ‘green’ so you can distinguish `HELLO` packets from other packets during simulation. You can make a packet by calling `self.network.make_packet` with the appropriate arguments: the address of the sending node, the type of the packet, the current time, and the color property string.

`MyNode.process` detects arriving `HELLO` packets and uses their source address to update the `self.neighbors` dictionary using the arrival link as the key and setting the value to the tuple `(time, p.source, link.cost)`.

**Step 1(b).** Next, add some code to `MyNode.transmit` that periodically checks each entry in `self.neighbors`, discarding entries that haven’t been updated in a while. We will define “a while” to be twice the `self.HELLO_INTERVAL`, so we need to delete all `self.neighbor[link]` entries that were sent at a time older than that long ago. These are entries that aren’t being updated by incoming `HELLO` packets, which means some link (or node) has gone down and we’re no longer receiving packets from that neighboring node.

Note: Please write the code for steps 1(a) and 1(b) to your version of `lab10.py` before coming to the lab.

Test out your modifications to `lab10.py` by running it and clicking on “Step 100”. You can now click on a node in the network diagram and another window will pop up and show the information stored in `self.neighbors`. Now click on a link—a big red ‘X’ should appear showing that the link has been broken. Click on “Step 100” again and check the neighbors listing for nodes at either end of the broken link to see that those nodes are no longer neighbors. Click on the broken link to repair it, click “Step 100”, and check that the neighbors lists have returned to their original state.

Show and run your code to a 6.02 staff person at the beginning of lab.

6 In the Lab (Wednesday, May 7, 2008)

The lab will involve Steps 2 and 3 mentioned in Section 4.
**Step 2: Broadcast a list of current neighbors, process broadcasts from other nodes**

Step 2 is for a node to tell every node in the network about who its neighbors are by broadcasting link-state announcement (LSA) packets. We will use another instance variable, `self.LSA`, to track the processing of incoming LSA packets. It’s a dictionary accessed using the LSA packet’s source address as a key, with values that are lists of the form `(sequence_number neighbor1 cost1 neighbor2 cost2 ...)`. Here, `cost1` is the cost from the node to `neighbor1`, `cost2` the cost to `neighbor2`, and so on.

**Step 2(a).** Add some code to `MyNode.transmit` that periodically (every `self.LSA_INTERVAL` time steps) sends out LSA packets along each link. We have set `LSA_INTERVAL` to 50 seconds, so LSA messages are sent by each node at time 0, 50, 100, 150, ... seconds.

LSA packets are distinguished by having a destination of “LSA”. Each LSA packet should contain:

1. A sequence number that gets incremented each time a new set of LSA packets is sent.
2. A list containing the address of each neighboring node immediately followed by the link cost to reach that neighbor. If the neighbors are `n1`, `n2`, ..., with costs `c1`, `c2`, ... respectively, then this part of the LSA has the form `n1,c1,n2,c2,...`.

The sequence number will be used by the other nodes to determine if they’ve already processed this LSA broadcast (for this lab, we will assume that the sequence number field doesn’t wrap around). Set the color property of LSA packets to ‘red’ so that they can be distinguished during simulation.

The code to make an LSA packet with a sequence number will look like:

```python
p = self.network.make_packet(self.address, 'LSA', time,
                           color='red',
                           seqnum=self.LSA_seqnum,
                           neighbors=lsa_info)
```

`MyNode.transmit` includes code to clear out older entries from `self.LSA`, which runs after each round of LSA broadcasts. In this case “old” refers to entries that have a sequence number smaller than `self.LSA_seqnum-1`. The idea is to ensure that old neighbor information is discarded; the dictionary will be updated as new LSA packets arrive from other nodes in the network. You don’t have to touch this part of the code, but just remember that an older LSA from a neighboring node that has a sequence number smaller than the current LSA sent out by the node will be discarded.\(^3\)

**Step 2(b).** Add some code to `MyNode.process` in the part that recognizes incoming LSA packets and checks the incoming sequence number against the stored sequence number for the LSA packet’s source address. If the incoming sequence number is less than or equal to the stored sequence number, the incoming LSA packet can be discarded. Otherwise update the stored sequence number and neighbor list from the incoming packet, and rebroadcast the packet. To rebroadcast the packet, you should send duplicates of the packet along each of the node’s links, using `self.network.duplicate_packet()`.

Once all the LSA broadcasts have made their way through the network, `self.LSA` for each node should have a list of neighbors’ addresses for every reachable node in the network. Test out your new code by

---

\(^3\) As an aside, this code can only work when all nodes start in the network at the same time and have update their LSA sequence numbers in “lock-step”; in real networks, because nodes can come up at any time, discarding old LSA entries is usually done by timing them out. In our simulation, the sequence number strategy is roughly, but not exactly, equivalent to timing out entries older than 50 seconds.
running `lab10.py`, stepping 100 time steps and clicking on a couple of nodes to see the listing of their LSA dictionaries. Break a few links, step another 100 time steps, and check the LSA entries have been updated correctly.

Then, try breaking a set of links that partitions the network into two unconnected pieces: do the updated LSA tables reflect the new network topology?

*Demonstrate your working code to a staff member and get checked-off.*

**Step 3: Build routing table from LSA dictionary using Dijkstra’s shortest paths algorithm**

We now turn to implementing the shortest-path computation. Your code should produce `self.routes`, a dictionary of routes at the node. This dictionary is keyed by destination node address, with the value being the link that must be used at the node to reach the corresponding destination.

This step runs every `LSA_INTERVAL` seconds in the middle of the `LSA_INTERVAL` epoch (i.e., if `LSA_INTERVAL` is 50 seconds, then the code runs at times 25, 75, 125, 175, ...)

This code periodically rebuilds the routing table using the neighbor information collected from incoming LSA packets after a new round of LSA transmissions has updated the information. Look for the comment lines in `MyNode.transmit` marked “STEP 3” to see where your code should go. Please write your code below this location.

This code, which each node in the simulator runs, uses a dictionary, `self.spcost`. `self.spcost` maintains the current shortest path cost to each node in the network that this node knows about.

Before Step 3 runs, we need to perform some initializations. First, we scan of the LSA advertisements to obtain `nodeset`, the set of nodes in the network. We only want to include those nodes currently reachable from the current node at which the computation is being done. The code starting with `for u in nodeset:` does that.\(^4\) You don’t have to modify this code; just note that `nodeset` is the set of reachable nodes to which you need to find routes.

We have also updated the `spcost` to each node in `nodeset` to INFINITY, and have set the route and `spcost` for `self.address` correctly.

Please read the comments in the source code file below “STEP 3”, where we describe the steps you should implement in the main loop, repeating until `nodeset` becomes empty.

1. Obtain the LSA entry for `last_node`, the last node removed from `nodeset` and update the `spcost` and tentative best route to each neighbor of that node *if necessary*. The LSA information for node `u` is given by `self.LSA[u][1:]`, which is the list with the first element (the sequence number) removed. The elements of this list alternate between neighbor and cost, so you need to traverse this list in steps of 2. You can do that in Python using

   ```python
   for i in range(0, len(mylist), 2):
   ```

   The “2” above is the step size.

\(^4\)We use this BFS traversal from round to round rather than just counting the number of entries in `self.LSA`. That’s because the failure of links or nodes, the recovery of links or nodes, or the arrival of new links and nodes all affect the number of nodes that are actually reachable from us.
2. Pick the node with minimum \texttt{spcost} and remember to remove it from nodeset.

3. Use \texttt{self.getlink} to update the route to \texttt{last\_node} if it is directly connected.

Test your code by running \texttt{lab10.py}, clicking on “Step 100” and then clicking nodes to examine their routing tables. Break a few links, click on “Step 100” again, and see if the routing tables have been recomputed correctly. Recover those links and see if the routing tables are recomputed correctly.

\textit{Demonstrate your working code to a staff member and get it checked off.}

Congratulations—you’ve now completed Lab 10 and are now a link-state routing guru (well, at least you’re well on your way to getting there)!
Check-off sheet for Lab #10 (May 7, 2008)

Names: ________________________________________________________________

1. Step 1 checkoff (before lab, for each group member by initials):

2. Step 2 checkoff (both 2(a) and 2(b)):

3. Step 3 checkoff (shortest-path computation):

Interview questions

1. If a link fails, how long does it take for all the other nodes in the network to discover that failure in the protocol you implemented in the simulator?

2. What happens if, instead of a link failing, a node crashes and later restarts, forgetting everything it had learned previously about the network? How well will your routing protocol handle the failure? Does it converge to the correct answer eventually? Even if it converges eventually, can you think of any transient problems that might arise?

3. Suppose we replaced each link’s cost with a value of 1, but left the network topology unchanged otherwise. Under what conditions would the shortest path tree computed in this new network (all costs 1) from some source node also produce a valid minimum-cost shortest path tree in the original network?