Lab 5 - Routing

Sean Miller

April 13, 2017

1 Brief Explanation of Variables

self.node = node self.timeout = 30 self.distance$_vectors = -> dictionaryoftuples, distanceandwhichneighboritispoint$
$(0, node.hostname) -> firstentryisself, whichisalwayszeroself.timer = Sim.scheduler.add(delay =$
$self.timeout, event =' broadcast', handler = self.broadcast) -> whenthistimerexpires, distancevectorsaresent$

self.neighbor$_timers = -> thisisadictionaryoftimersthatarestoredforeachneighbor.iftheytimeout, thenthatlink$

2 Experiment 1 - Simple Test of Five Nodes in a Row

This first test is very simple, consisting of five nodes in a row. No links are dropped. The Network configuration is as follows:

```
#
#
#   n1 -- n2 --- n3 -- n4 --- n5
#
#
n1  n2
n2  n1  n3
n3  n2  n4
n4  n3  n5
n5  n4
```

The top portion is a representation of the network. The bottom shows the connection each node has.

Each node begins by not knowing anything about its neighbors, but it can transmit packets with a TTL of 1, meaning to one neighbor away.

Upon receiving a neighbor's distance vector list (which I store as a dictionary, the key being the node and the value being the hops away), it checks if it currently has all that the neighbor has. If not, it adds the missing vectors, with a distance of +1 to the neighbor's distance.

If it does have it, it then checks to see if the path is better. This code snippet will explain that, where vector$_i ististhereceiveddistancevectorsfromtheneighbor$:

```
for k, v in vector_list.items():
  if k in self.distance_vectors and int(v[0]) <= (int(self.distance_vectors[k][0])-2):
    self.distance_vectors[k] = (int(v[0])+1,sending_node)
  elif k not in self.distance_vectors:
    #add the distance vector
    self.distance_vectors[k] = (int(v[0])+1,sending_node)
```

2.1 Experiment 1 Results

Trace of protocol I will show you how the protcol now works. This test is short enough that I will include it in its entirety:

```
n1: {'n1': (0, 'n1')}
n2: {'n2': (0, 'n2')}
n3: {'n3': (0, 'n3')}
n4: {'n4': (0, 'n4')}
n5: {'n5': (0, 'n5')}
n2: {'n1': (1, 'n1'), 'n2': (0, 'n2')}
n1: {'n1': (0, 'n1'), 'n2': (1, 'n2')}
n3: {'n2': (1, 'n2'), 'n3': (0, 'n3')}
n2: {'n1': (1, 'n1'), 'n2': (0, 'n2'), 'n3': (1, 'n3')}
n4: {'n3': (1, 'n3'), 'n4': (0, 'n4')}
n3: {'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4')}
n5: {'n4': (1, 'n4'), 'n5': (0, 'n5')}
n4: {'n3': (1, 'n3'), 'n4': (0, 'n4'), 'n5': (1, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (1, 'n2'), 'n3': (2, 'n2')}
n3: {'n1': (2, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4')}
n2: {'n1': (1, 'n1'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3')}
n4: {'n2': (2, 'n3'), 'n3': (1, 'n3'), 'n4': (0, 'n4'), 'n5': (1, 'n5')}
n3: {'n1': (2, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4'), 'n5': (2, 'n4')}
n5: {'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (1, 'n2'), 'n3': (2, 'n2'), 'n4': (3, 'n2')}
n5: {'n2': (3, 'n4'), 'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')}
n2: {'n1': (1, 'n1'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3'), 'n5': (3, 'n3')}
n4: {'n1': (3, 'n3'), 'n2': (2, 'n3'), 'n3': (1, 'n3'), 'n4': (0, 'n4'), 'n5': (1, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (1, 'n2'), 'n3': (2, 'n2'), 'n4': (3, 'n2'), 'n5': (4, 'n2')}
n5: {'n1': (4, 'n4'), 'n2': (3, 'n4'), 'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')}
```

As we can see, each node only knows about itself at the start. It then learns about its neighbors. Since n3 is in the middle of the row, it makes sense that it finishes first, since it can receive both halves of the path to itself. We can see that neighbors increment on each step, to learning about 1 away, 2 away, and so on.

I will also show the trace of an example node, say here n3: n3: 'n3': (0, 'n3') n3: 'n2': (1, 'n2'), 'n3': (0, 'n3') n3: 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4') n3: 'n1': (2, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4') n3: 'n1': (2, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4'), 'n5': (2, 'n4')

There are no other possible routes (as it is only a straight line of nodes), so it can't get any better.

Final Results : We see here each node and their distance to each other node n1: 'n1': (0, 'n1'), 'n2': (1, 'n2'), 'n3': (2, 'n2'), 'n4': (3, 'n2'), 'n5': (4, 'n2') n2: 'n1': (1, 'n1'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3'), 'n5': (3, 'n3') n3: 'n1': (2, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4'), 'n5': (2, 'n4') n4: 'n1': (3, 'n3'), 'n2': (2, 'n3'), 'n3': (1, 'n3'), 'n4': (0, 'n4'), 'n5': (1, 'n5') n5: 'n1': (4, 'n4'), 'n2': (3, 'n4'), 'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')

3 Experiment 2 - Five Node Ring and Taking Down a Link

In this second test, the network is five nodes, all connected in a ring. The network appears as follows:

```
#
#   n1 --- n2
#   |       |
#   |       |
#   |       n3
#   |       |
#   |       |
#   n5 --- n4
#
n1  n2  n5
```

```
n2 n1 n3
n3 n2 n4
n4 n3 n5
n5 n4 n1
```

The situation is the same as stated in the previous example, with each node only knowing about itself to begin.

The difference about this experiment is that we are beginning to play around with taking down links. In this example, I will take down one link, n1 ¡–¿ n2.

As we will see, routes will change. I will show both the paths before and after and go through the steps of what happens after the link is dropped.

Note: Links are dropped as follows

```
    Sim.scheduler.add(delay=120, event=None, handler=n1.get_link('n2').down)
    Sim.scheduler.add(delay=120, event=None, handler=n2.get_link('n1').down)
```

It's necessary to end the link both ways or one continues sending.

3.1 Experiment 2 Results

Before Dropping the Link:

```
n2: {'n1': (1, 'n1'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3'), 'n5': (2, 'n1')}
n3: {'n1': (2, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4'), 'n5': (2, 'n4')}
n5: {'n1': (1, 'n1'), 'n2': (2, 'n1'), 'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')}
n4: {'n1': (2, 'n5'), 'n2': (2, 'n3'), 'n3': (1, 'n3'), 'n4': (0, 'n4'), 'n5': (1, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (1, 'n2'), 'n3': (2, 'n2'), 'n4': (2, 'n5'), 'n5': (1, 'n5')}
```

This is the order in which they are received. I won't go through the route up until this point, because it is trivial, as it has already been shown in experiment 1. These are the expected results.

The link is now dropped A link is noticed to be dropped when three timeouts have occurred. I measure this by setting a timer whenever a packet is received. The code is as follows:

```
if(int(v[0]) == 0):
    if(k in self.neighbor_timers):
        if(self.neighbor_timers[k]):
            Sim.scheduler.cancel(self.neighbor_timers[k])
        new_timer = Sim.scheduler.add(delay=3*self.timeout, event=k, handler=self.handle_drop
        self.neighbor_timers[k] = new_timer
    else:
        new_timer = Sim.scheduler.add(delay=3*self.timeout, event=k, handler=self.handle_drop
        self.neighbor_timers[k] = new_timer
```

This code is included in the same for loop as the code in the previous experiment, just prior to calling the code from Experiment 1. It checks if a timer exists for that neighbor. If so, it cancels that timer. If not, then it adds the timer. If the node is receiving consistent packets (every 30 seconds), then it won't time out. However, if three of those cycles pass without receiving a packet (and therefore not renewing the timer), the handle$_{d}ropped_{l}inkfunctioniscalled.$

```
def handle_dropped_link(self,event):
  #set the timer to none
   self.neighbor_timers[event] = None
  #I think I should then zero out the distance_vectors for this node
   self.distance_vectors = {}
   self.distance_vectors[self.node.hostname] = (0,self.node.hostname)
```

What follows then is this now-empty distance vector is passed to other neighbors. They check to see if their paths were through this node, and if they were, it also evaluates to see if that path has changed.

Before a node responds to the received distance$_v ector, itcheckstoseeifithasanypathsthatusedtheneighborithasrece$

```
for k in self.distance_vectors.keys():
  #if it went through this sending_node before, check for a change
  if(self.distance_vectors[k][1] == sending_node):
      #if this previously held distance is still in the sending node, check for a change.
      # if there's no change, don't worry about it.
      if(k in vector_list and vector_list[k][0] != self.distance_vectors[k][0]):
          #then set the value to the new value... it'll update later if need be
          updated_distance = int(vector_list[k][0]+1)
          dist_tuple = (updated_distance,sending_node)
          self.distance_vectors[k] = dist_tuple
      #if it's no longer in the vector list, delete it, because the path is compromised
      elif(k not in vector_list):
          del self.distance_vectors[k]
```

See comments in code snippet for explanation

The changes then echo throughout the network. Here are the steps following the dropped link:

```
n2: {'n1': (3, 'n3'), 'n2': (0, 'n2')}
n2: {'n1': (3, 'n3'), 'n2': (0, 'n2'), 'n3': (1, 'n3')}
n2: {'n1': (3, 'n3'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3')}
n2: {'n1': (3, 'n3'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3'), 'n5': (3, 'n3')}
n1: {'n1': (0, 'n1'), 'n2': (3, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (3, 'n5'), 'n3': (3, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (3, 'n5'), 'n3': (3, 'n5'), 'n4': (2, 'n5')}
n1: {'n1': (0, 'n1'), 'n2': (3, 'n5'), 'n3': (3, 'n5'), 'n4': (2, 'n5'), 'n5': (1, 'n5')}
n5: {'n1': (1, 'n1'), 'n2': (4, 'n1'), 'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')}
n3: {'n1': (4, 'n2'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4'), 'n5': (2, 'n4')}
n3: {'n1': (3, 'n4'), 'n2': (1, 'n2'), 'n3': (0, 'n3'), 'n4': (1, 'n4'), 'n5': (2, 'n4')}
n5: {'n1': (1, 'n1'), 'n2': (3, 'n4'), 'n3': (2, 'n4'), 'n4': (1, 'n4'), 'n5': (0, 'n5')}
n2: {'n1': (4, 'n3'), 'n2': (0, 'n2'), 'n3': (1, 'n3'), 'n4': (2, 'n3'), 'n5': (3, 'n3')}
n1: {'n1': (0, 'n1'), 'n2': (4, 'n5'), 'n3': (3, 'n5'), 'n4': (2, 'n5'), 'n5': (1, 'n5')}
```

You may note that n4 is not listed here. That is because I only print out when a change has been made. n4 did not change. Its paths would not have been affected by the link between n1 and n2 dropping.

Setting	Result
1	1.0
2	3.45
3	7.85
4	15.89

Nam sed lacus sit amet nisl bibendum rutrum vel id nisl. Etiam sit amet ipsum vulputate tellus fringilla tristique a et augue. Etiam suscipit ante id est lobortis hendrerit. Vivamus vel nisl sit amet metus volutpat faucibus. Praesent nunc urna, luctus vel convallis eget, luctus et odio. Nunc et nisl felis. Fusce quis libero sit amet libero cursus pretium. Vivamus dictum risus non tellus commodo non bibendum tortor convallis. Cras tempor orci eu leo auctor sed euismod arcu consectetur. In scelerisque felis et erat commodo bibendum. Pellentesque hendrerit enim vitae neque sollicitudin bibendum. In ligula lorem, blandit sit amet aliquet eget, accumsan ut sem. Maecenas in velit justo. Morbi tellus sem, ultricies in tristique non, aliquam a lacus. Sed rhoncus blandit ligula, ut eleifend magna lacinia quis.

1. Item.

2. Another item.

4 Section Name

Donec luctus, libero et egestas tincidunt, arcu ante commodo nunc, quis sodales leo risus non libero. Mauris ac blandit ligula. Praesent in dolor non nibh congue blandit. Curabitur in sodales neque. Curabitur tincidunt nisl nec mauris bibendum molestie. Suspendisse non justo erat. Ut quis odio elit, sit amet ullamcorper dui. Nam massa urna, tempus non hendrerit porttitor, feugiat quis quam. Quisque lacinia cursus nulla, id placerat enim accumsan et. Donec porta pharetra tincidunt.

5 Section Name

d_{trans} is the transmission delay. d_{prop} is the propagation delay.

$$\begin{aligned} d &= d_{trans} + d_{prop} \\ &= (1000 * 8)/1000000 + 0.05 \\ &= 0.058 \end{aligned}$$