Routing

Nathan Davis

Apr 13, 2016

1 Setup

1.1 General Setup

The beginning source for these experiments was taken from the bene lab. The main additions were the dvrouting.py file and the experiment (1-3) python files.

1.2 DVPacket

We created a custom packet (DVPacket) that inherits from the base packet. The main purpose of this was to plug in default values and add a structure for passing the distance vectors in a packet.

```
class DVPacket(Packet):
    def __init__(self,source_address=1,source_port=0,
                 destination_address=0,destination_port=0,
                 ident=0,ttl=1,protocol="dvrouting",body="",length=0,
                 source_hostname="",dvs={}):
        Packet.__init__(...)
        self.source_hostname = source_hostname
        self.dvs = dvs
```

Having the distance vectors within the packet made it easier to transfer them and parse them out then they would have been in the body of a packet.

1.3 DistanceVector

We also created a structure for distance vectors. We packaged up the distance and the next hop for easier referencing.

```
class DistanceVector(object):
    def __init__(self,distance,next_hop):
        self.distance = distance
        self.next_hop = next_hop
```

These distance vectors were used in a structure within the DVRoutingApp to keep track of the smallest distance to each address within reach.

```
# self.dvs format:
#   address: distance_vector
# {
```

```
#    2: DistanceVector(10, 'n3')
# }
```

1.4 HandleDataApp

We created a simple app that would log when data was received.

```
class HandleDataApp(object):
    def __init__(self,node):
        self.node = node

    def receive_packet(self,packet):
        print Sim.scheduler.current_time(),self.node.hostname,"received data"
```

This was used when sending data packets between nodes and used the "data" protocol.

```
n1.add_protocol(protocol="data",handler=HandleDataApp(n1))
```

We also altered the tracing for packets to enable us to trace packets based on protocol as well. We created our own debugging statements for the "dvrouting" protocol but we wanted to use the built-in tracing for "data" packets. By including the protocol name in the trace type name we were able to narrow it down to only tracing "data" packets within the nodes.

```
def trace(self,message,protocol=""):
    Sim.trace("Node"+protocol,message + " ("+protocol+")")

... In ex.py ...
Sim.set_debug(True)
Sim.set_debug('Nodedata')
```

1.5 DVRoutingApp __init__

We created the DVRoutingApp in dvrouting.py to do the heavy lifting for the routing protocol. When the App is initialized we create a new distance vector and we begin broadcasting the distance vector every 30 seconds. We also put a hard stop time at 400 seconds to avoid letting the simulation run forever.

```
def __init__(self,node,stop_time=400):
    self.node = node
    self.stop_time = stop_time

    # Timestamp of last dvs update from each node
    self.neighbor_timestamps = {}

    # Neighbor distance vectors
    self.neighbor_dvs = {}

    # Initialize distance vectors
```

```
12      self.dvs = self.new_dvs()
13
14      # Broadcast every 30 seconds
15      self.recurring_broadcast_dvs()
```

1.6 Broadcasting Distance Vectors

Using the DVPacket outlined above, we were able to broadcast a node's distance vectors. The recurring broadcast would send the broadcast immediately and then schedule another broadcast in 30 seconds.

```
1  def broadcast_dvs(self, delay=0):
2      # send a broadcast packet to all neighbors
3      dvp = DVPacket(source_hostname=self.node.hostname, dvs=self.dvs)
4      Sim.scheduler.add(delay=delay, event=dvp, handler=self.node.send_packet)
5
6  def recurring_broadcast_dvs(self, delay=30):
7      self.broadcast_dvs()
8      # Key to kill simulation after self.stop_time seconds
9      if Sim.scheduler.current_time() < self.stop_time:
10         Sim.scheduler.add(delay=delay, event=30, handler=self.recurring_broadcast_dvs)
```

1.7 DVRoutingApp receive_packet

The receive packet method was the template for when the app would receive a packet from one of its neighbors. It adds the neighbor's dvs to the key-value store of dvs and sets the timestamp for the last time the node has heard from that neighbor. Then it checks for broken links. We'll review that later in this paper. Then it calculates the new dvs and compares it to the old dvs. If there are any changes, the node broadcasts the new dvs out to its neighbors.

```
1  def receive_packet(self,packet):
2
3      print Sim.scheduler.current_time(),self.node.hostname,
4          "received distance vectors from",packet.source_hostname
5
6      # Keep the neighbor distance vectors updated
7      self.neighbor_dvs[packet.source_hostname] = packet.dvs
8      self.neighbor_timestamps[packet.source_hostname] = Sim.scheduler.current_time()
9
10     # Check for broken links
11     self.check_links()
12
13     # Add the dvs to the current set of dvs
14     pristine = self.calculate_dvs()
15
16     # If anything changed, rebroadcast the current dvs
17     if not pristine:
18         self.broadcast_dvs()
```

1.8 DVRoutingApp check_links

The check_links logic goes through each neighbor and checks that the last time the node received communication from it was within 90 seconds. If the timestamp is further back then 90 seconds it deletes that neighbor from the key-value store of neighbors. The next time the node's distance vectors are calculated from its neighbors, that neighbor will not be included.

```
    # If the neighbor has gone down
    if not timestamp or current_time - timestamp > max_time:
        print Sim.scheduler.current_time(), self.node.hostname,
            "detected that the link to", neighbor, "is down"
        dead_neighbors.append(neighbor)

    ...

    # For all the neighbors that have gone down
    for neighbor in dead_neighbors:
        del self.neighbor_dvs[neighbor]
```

1.9 DVRoutingApp new_dvs

This method is used to create a new set of distance vectors from the known distance vectors received from neighbors. First the node sets its own links to a distance of zero. Then the node iterates through each of its neighbors and iterates through each of their links looking for shorter distances to each address. If one of its neighbors next hops is the original node it skips that distance comparison because going to a neighbor and then back to itself is never the shortest route.

```
def new_dvs(self):
    """Calculate the dvs from the neighbor dvs"""

    # Initialize distance vectors to myself
    ret_dvs = {}
    for l in self.node.links:
        ret_dvs[l.address] = DistanceVector(0, self.node.hostname)

    # Go through all my neighbors distance vectors
    for neighbor, dvs in self.neighbor_dvs.iteritems():

        # Match it against ret dvs to "learn" new or quicker paths
        for a, v in dvs.iteritems():

            # Skip routes that point back to myself
            if v.next_hop == self.node.hostname:
                continue

            new_distance = v.distance + 1
            current_vector = ret_dvs.get(a)
            if current_vector == None:
                current_distance = None
            else:
                current_distance = current_vector.distance

            # If it's not there, put it in
```

```
27              # If the new one is smaller, replace it
28              if current_distance == None or current_distance > new_distance:
29                  ret_dvs[a] = DistanceVector(new_distance, neighbor)
30
31      return ret_dvs
```

In the end it returns the new distance vectors for comparison to the old ones.

1.10 DVRoutingApp calculate_dvs

We designed this function to request the new distance vectors, update the old distance vectors and return a boolean stating whether it had changed anything in the old copy. First it created the new distance vectors using new_dvs. Then it checked for items in the new distance vectors (new_dvs) that didn't exist in the old one (self.dvs). These items were added to the old one. Then it checked for items which had been removed in the new version and removed those from the old one. All the while it recorded any time something was added or removed from the original set of distance vectors.

```
1  def calculate_dvs(self):
2      """Take a dvs and add it to self.dvs
3      Return True if anything changed"""
4
5      # Mark if any changes are made
6      pristine = True
7
8      # Create a new dvs from the neighbors dvs
9      new_dvs = self.new_dvs()
10
11     # Check for things in the new one to update in the old one
12     for a, v in new_dvs.iteritems():
13         new_vector = v
14         current_vector = self.dvs.get(a)
15         if current_vector == None or current_vector.distance != new_vector.distance or curren
16             self.dvs[a] = new_vector
17             pristine = False
18             self.add_forwarding_entry_to_host(a, new_vector.next_hop)
19
20     # Check for things that have disappeared from the new one to delete from the old one
21     dead_addresses = []
22     for a, v in self.dvs.iteritems():
23         if a not in new_dvs:
24             self.node.delete_forwarding_entry(a, None)
25             dead_addresses.append(a)
26             pristine = False
27             print Sim.scheduler.current_time(),self.node.hostname,"removed route to address",a
28     for a in dead_addresses:
29         del self.dvs[a]
30
31     return pristine
```

2 Five Nodes in a Row

2.1 Setup

We created a network with five nodes in a string. Each node was connected to one or two other nodes and was instructed to use the DVRoutingApp for the "dvrouting" protocol and the HandleDataApp for the "data" protocol.

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

We also set up three packets to send after 10, 12, and 14 seconds. These packets would ensure that the routing was working correctly.

```
# send one packet from n1 to n2
p = packet.Packet(destination_address=n2.get_address('n1'),protocol='data')
Sim.scheduler.add(delay=10, event=p, handler=n1.send_packet)

# send one packet from n1 to n5
p = packet.Packet(destination_address=n5.get_address('n4'),protocol='data')
Sim.scheduler.add(delay=12, event=p, handler=n1.send_packet)

# send one packet from n5 to n1
p = packet.Packet(destination_address=n1.get_address('n2'),protocol='data')
Sim.scheduler.add(delay=14, event=p, handler=n5.send_packet)
```

2.2 Results

When the simulation starts all of the nodes begin transimitting their distance vectors to their neighbors. After 0.001 seconds the neighbors receive the distance vectors and begin adding to their known routes. Below we can se the beginning of the network setup. Node 2 received distance vectors from node 1 and added a route to node 1. Nodes 1 and 3 received distance vectors from node 2 and were able to then set routes to node 2's addresses.

```
0.001 n2 received distance vectors from n1
0.001 n2 set route to address 1 though n1
...
0.001 n1 received distance vectors from n2
0.001 n1 set route to address 2 though n2
0.001 n1 set route to address 3 though n2
...
0.001 n3 received distance vectors from n2
```

```
0.001 n3 set route to address 2 though n2
0.001 n3 set route to address 3 though n2
```

Towards the end of the setup phase we can see that the water is reaching the end of the row. As each node receives distance vectors from its neighbors, the node updates its distance vectors and sends them on. The last few phases of the setup happen when node 1 learns about address 8 (on node 5) and node 5 learns about address 1 (on node 1). These are the final steps. Also of note is that many times when a node receives distance vectors then nothing changes for that node. When that happens, no new broadcasts are sent out and the chain of updates dies there. We can see this in the final update of node 4 from node 5. Node 5 has just updated its tables to show a route to node 1 but node 4 already knows a shorter route to node 1 so it doesn't update anything.

```
0.004 n1 received distance vectors from n2
0.004 n1 set route to address 8 though n2
...
0.004 n5 received distance vectors from n4
0.004 n5 set route to address 1 though n4
...
0.005 n2 received distance vectors from n1
0.005 n4 received distance vectors from n5
```

The packets sent at 10, 12 and 14 seconds are traced through the system. The first packet is sent from n1 to n2 using n1's forwarding table. The second one is forwarded from n1 to n2, n3, n4 and finally to n5 who receives the data. The third packet does just the opposite and travels from n5 to n1 through the chain

```
10.0 n1 forwarding packet to 2 (data)
10.001 n2 received packet (data)
10.001 n2 received data
...
12.0 n1 forwarding packet to 8 (data)
12.001 n2 forwarding packet to 8 (data)
12.002 n3 forwarding packet to 8 (data)
12.003 n4 forwarding packet to 8 (data)
12.004 n5 received packet (data)
12.004 n5 received data
...
14.0 n5 forwarding packet to 1 (data)
14.001 n4 forwarding packet to 1 (data)
14.002 n3 forwarding packet to 1 (data)
14.003 n2 forwarding packet to 1 (data)
14.004 n1 received packet (data)
14.004 n1 received data
```

3 Five Nodes in a Loop

3.1 Setup

The network for this experiment involved a ring of nodes where each node could send to two other nodes in that ring. Each node was equipped with the HandleDataApp and DVRoutingApp as before.

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

For the packets, we sent one packet from n1 to n5 at 5 seconds and another from n4 to n1 at 7 seconds.

```
# send one packet from n1 to n5
p = packet.Packet(destination_address=n5.get_address('n1'),protocol='data')
Sim.scheduler.add(delay=5, event=p, handler=n1.send_packet)

# send one packet from n4 to n1
p = packet.Packet(destination_address=n1.get_address('n5'),protocol='data')
Sim.scheduler.add(delay=7, event=p, handler=n4.send_packet)
```

At 10 seconds we removed the links between n1 and n5. We gave the network 130 seconds to figure out that the link was down and make adjustments. Then we sent the same packets again at 130 and 132 seconds respectively.

```
# take the link down between n1 and n5
Sim.scheduler.add(delay=10, event=None, handler=n1.get_link('n5').down)
Sim.scheduler.add(delay=10, event=None, handler=n5.get_link('n1').down)
```

3.2 Results

During the network setup phase node 1 assigned packets addressed to node 5 to go straight there. Node 4 assigned packets addressed to node 1 to go through node 5.

```
0.001 n1 received distance vectors from n5
0.001 n1 set route to address 9 though n5
0.001 n1 set route to address 10 though n5
...
0.002 n4 received distance vectors from n5
0.002 n4 set route to address 1 though n5
0.002 n4 set route to address 2 though n5
```

When the packets are sent, we see this pattern holding true. n1 sends a packet straight to n5 and n4 sends a packet through n5 to n1. Both use the link between n1 and n5.

```
5.0 n1 forwarding packet to 10 (data)
```

```
5.001 n5 received packet (data)
5.001 n5 received data
...
7.0 n4 forwarding packet to 2 (data)
7.001 n5 forwarding packet to 2 (data)
7.002 n1 received packet (data)
7.002 n1 received data
```

When the link is dropped, n1 and n5 stop getting updates from each other. After 90 seconds they drop each other from their forwarding tables. First, n1 sets routes to n4 through n2 since it can no longer use n5. Then it sends out the distance vectors letting n2 know it no longer has a route to n5. n2 updates its routes to go through n3 to get to n5. Then, when it sends the distance vectors back to n1, n1 sets routes to n5 via n2. The same pattern happens for n5 in the reverse direction.

```
120.001 n1 detected that the link to n5 is down
120.001 n1 set route to address 7 though n2
120.001 n1 set route to address 8 though n2
120.001 n1 removed route to address 9
120.001 n1 removed route to address 10
...
120.001 n5 detected that the link to n1 is down
120.001 n5 set route to address 3 though n4
120.001 n5 set route to address 4 though n4
120.001 n5 removed route to address 1
120.001 n5 removed route to address 2
...
120.002 n2 received distance vectors from n1
120.002 n2 set route to address 9 though n3
120.002 n2 set route to address 10 though n3
...
120.002 n4 received distance vectors from n5
120.002 n4 set route to address 1 though n3
120.002 n4 set route to address 2 though n3
...
120.003 n1 received distance vectors from n2
120.003 n1 set route to address 9 though n2
120.003 n1 set route to address 10 though n2
...
120.003 n5 received distance vectors from n4
120.003 n5 set route to address 1 though n4
120.003 n5 set route to address 2 though n4
...
```

When the second pair of packets is sent, they both use the new pathways with n1's packet going through n2, n3, n4, n5 and n4's packet going through n3, n2 and n1.

```
130.0 n1 forwarding packet to 10 (data)
130.001 n2 forwarding packet to 10 (data)
130.002 n3 forwarding packet to 10 (data)
130.003 n4 forwarding packet to 10 (data)
130.004 n5 received packet (data)
130.004 n5 received data
...
```

```
132.0 n4 forwarding packet to 2 (data)
132.001 n3 forwarding packet to 2 (data)
132.002 n2 forwarding packet to 2 (data)
132.003 n1 received packet (data)
132.003 n1 received data
```

4 Fifteen Node Mesh

4.1 Setup

The network for this experiment involved 15 nodes interconnected into a mesh.

```
#
#   n1 --- n2 --- n3
#   |      |      |
#   |      |      |
#   n4 --- n5 --- n6
#   |      |      |
#   |      |      |
#   n7 --- n8 --- n9
#   |      |      |
#   |      |      |
#   n10 -- n11 -- n12
#   |      |      |
#   |      |      |
#   n13 -- n14 -- n15
#
#
n1 n2 n4
n2 n1 n5 n3
n3 n2 n6
n4 n1 n5 n7
n5 n4 n2 n6 n8
n6 n3 n5 n9
n7 n4 n8 n10
n8 n5 n7 n9 n11
n9 n6 n8 n12
n10 n7 n11 n13
n11 n8 n10 n12 n14
n12 n9 n11 n15
n13 n10 n14
n14 n11 n13 n15
n15 n12 n14
```

For the experiment portion we tested sending packets from n1 to n7. We took down various links to ensure that the network would reroute properly. First we sent a packet from n1 to n7. Then we took down the link between n4 and n7.

```
#   n1 --- n2 --- n3
#   |      |      |
#   |      |      |
#   n4 --- n5 --- n6
```

```
#           |       |
#           |       |
#   n7 --- n8 --- n9
#   |       |       |
#   |       |       |
#   n10 -- n11 -- n12
#   |       |       |
#   |       |       |
#   n13 -- n14 -- n15
```

We sent another packet from n1 to n7 and then took down the link between n5 and n8.

```
#   n1 --- n2 --- n3
#   |       |       |
#   |       |       |
#   n4 --- n5 --- n6
#                   |
#                   |
#   n7 --- n8 --- n9
#   |       |       |
#   |       |       |
#   n10 -- n11 -- n12
#   |       |       |
#   |       |       |
#   n13 -- n14 -- n15
```

We sent another packet from n1 to n7 and then put the link back up between n4 and n7.

```
#   n1 --- n2 --- n3
#   |       |       |
#   |       |       |
#   n4 --- n5 --- n6
#   |               |
#   |               |
#   n7 --- n8 --- n9
#   |       |       |
#   |       |       |
#   n10 -- n11 -- n12
#   |       |       |
#   |       |       |
#   n13 -- n14 -- n15
```

Then we sent a final packet from n1 to n7.

4.2 Results

After the initial setup, the packet took the route n1, n4, n7 as expected.

```
5.0 n1 forwarding packet to 18 (data)
5.001 n4 forwarding packet to 18 (data)
5.002 n7 received packet (data)
5.002 n7 received data
```

After the link between n4 and n7 went down the routing tables were updated and the packet took the route n1, n2, n5, n8, n7.

```
140.0 n1 forwarding packet to 18 (data)
140.001 n2 forwarding packet to 18 (data)
140.002 n5 forwarding packet to 18 (data)
140.003 n8 forwarding packet to 18 (data)
140.004 n7 received packet (data)
140.004 n7 received data
```

After the link between n5 and n8 went down and routing tables were updated, the packet took the route n1, n2, n3, n6, n9, n8, n7.

```
280.0 n1 forwarding packet to 18 (data)
280.001 n2 forwarding packet to 18 (data)
280.002 n3 forwarding packet to 18 (data)
280.003 n6 forwarding packet to 18 (data)
280.004 n9 forwarding packet to 18 (data)
280.005 n8 forwarding packet to 18 (data)
280.006 n7 received packet (data)
280.006 n7 received data
```

When the link between n4 and n7 is set back up, the next broadcast alerts n4 that it may now forward directly to n7. n4 broadcasts the new route and n1 picks it up as well as all of n4's neighbors. The new routes propogate out until the entire mesh is aware of them.

```
300.001 n4 received distance vectors from n7
300.001 n4 set route to address 18 though n7
```

The next packet is once again forwarded directly along the route n1, n4, n7.

```
332.0 n1 forwarding packet to 18 (data)
332.001 n4 forwarding packet to 18 (data)
332.002 n7 received packet (data)
332.002 n7 received data
```