Lab 4 Report - Distance Vector Routing

Dallin Christensen

April 12, 2014

1 Routing Protocol

To implement distance vector routing protocol, I made changes to the node class so that it creates a distance vector table. I also implemented a DVRoutingApp that takes care of broadcasting the distance vector tables from each node to their neighbors, as well as setting up the forwarding tables for each node.

1.1 Changes to the Node class

I modified the Node class so that each node holds a dictionary that maps their neighboring nodes to the distance vector tables for each node. No node stores their own distance vector table; instead, the node computes their distance vector table whenever necessary by iterating through all of their neighbors tables. Thus, there are no issues of erroneous caches of distances that have been changed elsewhere, but not in any given node.

Distance Vector Routing in the Node class

```
class Node(object):
    def __init__(self,hostname):
        self.hostname = hostname
        self.links = []
        self.protocols = {}
        self.forwarding_table = {}
        self.neighbors_dv_tables = {}

    def distance_vector_table(self):
        dv_table = {self.hostname: 0, }
        if not self.neighbors_dv_tables:
            return dv_table
        for neighbor in self.neighbors_dv_tables:
            neighbor_table = self.neighbors_dv_tables[neighbor]
            if not neighbor_table:
                continue
            link = self.get_link(neighbor.hostname)
            for dest_node_name in neighbor_table:
                if dest_node_name == self.hostname:
                    continue
                new_distance = neighbor_table[dest_node_name]
                if dest_node_name not in dv_table:
                    dv_table[dest_node_name] = new_distance + 1
                else:  # already in the dv_table
                    if dv_table[dest_node_name] > new_distance + 1:
                        dv_table[dest_node_name] = new_distance + 1
        return dv_table
```

1.2 DVRoutingApp

Determining distances from one node to another is part of what needs to be done in order for routing to work properly. To take care of the necessary functions, I created a DVRoutingApp that handled any packets labeled with the protocol “dvrouting”. This app served three main purposes: to broadcast the current nodes distance vector table, to receive broadcasts from other nodes, and to set up the forwarding table for each node based on all of the distance vector tables.

1.2.1 Send Broadcast

Each node starts off knowing only about itself. It sends a broadcast packet to all of its neighbors (which it discovers by iterating over its links) with a TTL of 1. Thus, the broadcast packet does not get sent to any nodes besides the current nodes neighbors. It also resets a broadcast timer.

Broadcasting DV Table to Neighboring Nodes

```python
def broadcast(self, event):
    body = {"node": self.node, "dv_table": self.prev_dv_table}
    p = packet.Packet(
        source_address=self.node.get_address(self.node.hostname),
        destination_address=0, ident='bcast', ttl=1,
        protocol='dvrouting', length=100, body=body)
    Sim.scheduler.add(delay=0, event=p, handler=self.node.send_packet)
    self.timer = Sim.scheduler.add(
                delay=30, event='broadcast', handler=self.broadcast)
    Sim.trace("Broadcasting %s's distance vector table" % (self.node.hostname))
```

1.2.2 Receive Broadcast Information

Once the nodes receive information from their neighbors, they add those distance vectors to their dictionary and are able to calculate a new distance vector table for themselves, which they can then send to their neighbors. Of note, it only sends out a broadcast if it detects that the distance vector table has changed. Otherwise, nothing happens.

Receiving DV Tables from Neighboring Nodes

```python
def add_dv_table(self, node, dv_table):
    self.node.neighbors_dv_tables[node] = dv_table

def receive_packet(self, p):
    self.add_dv_table(p.body["node"], p.body["dv_table"])
    cur_table = self.node.distance_vector_table()
    if cur_table != self.prev_dv_table:
        Sim.trace("Update distance vector table for %s to %s"
                  % (self.node.hostname, cur_table))
        self.prev_dv_table = cur_table
        self.setup_forwarding_entries()
        self.broadcast('broadcast')
```

1.2.3 Set Up Forwarding Tables

Using the information in the distance vector table, the DVRoutingApp adds links to the forwarding table for each node. For any node that is not a neighbor of the current node, the DRRoutingApp recursively follows each link from the current node until it finds the destination node, adding the appropriate link and destination address to the node's forwarding table.

```
def setup_forwarding_entries(self):
    dv_table = self.node.distance_vector_table()
    ...
    for neighbor in self.node.neighbors_dv_tables:
        neighbor_table = self.node.neighbors_dv_tables[neighbor]
        ...
        link = self.node.get_link(neighbor.hostname)
        for dest_node_name in neighbor_table:
            ...
            try:
                distance_from_self_node = dv_table[dest_node_name]
            except: # node is unreachable from current node
                ...
                continue
            dest_addr = self.get_dest_addr(self.node, dest_node_name,
                                            distance_from_self_node)
            if dest_addr in self.node.forwarding_table:
                # Already have a forwarding table entry to destination
                ...
                continue
            self.node.add_forwarding_entry(dest_addr, link)
            ...
    return dv_table

def get_dest_addr(self, start_node, dest_node_name, cur_distance):
    dest_addr = start_node.get_address(dest_node_name)
    if dest_addr:
        ...
        return dest_addr
    else:
        for neighbor in start_node.neighbors_dv_tables:
            n_dv_table = start_node.neighbors_dv_tables[neighbor]
            try:
                new_distance = n_dv_table[dest_node_name]
            except:
                new_distance = float("inf")

            if new_distance < cur_distance:
                ...
                return self.get_dest_addr(neighbor, dest_node_name, new_distance)
```

1.2.4 Timeouts

Each DVRoutingApp object has a broadcast timer that causes the node to broadcast it's distance vector table periodically. This timer has been set to go off every 30 seconds. There is also a timeout for receiving data from neighbors. This timeout is set to 90 seconds. If a node doesn't receive data from any given neighbor for 90 seconds (the equivalent of 3 broadcast periods), the node removes that neighbor's distance vector table from its dictionary and recalculates it's own distance vector (which it then broadcasts). The node forgets about that neighbor, but can easily add the neighbor back, should the neighbor start broadcasting again.

2 Experiments

For each of these examples, show a trace of the routing protocol so that you can demonstrate how it works, plus a trace of the nodes as they forward a packet correctly.

2.1 Five Nodes in a Row

I set up a network of five nodes in a row (n1 ↔ n2 ↔ n3 ↔ n4 ↔ n5). I tested the network using these routes:

- n1 to n4 (n1 → n2 → n3 → n4)
- n2 to n5 (n2 → n3 → n4 → n5)
- n3 to n1 (n3 → n2 → n1)
- n4 to n2 (n4 → n3 → n2)

The routing protocol correctly set up the forwarding table for each node, and each packet was successfully sent to the intended destination. To see the trace of the routing protocol and the packet forwarding, see "fiveinarow.txt" in the attached "trace" directory.

2.2 Five Nodes in a Ring

I set up a network of five nodes in a ring (n1 ↔ n2 ↔ n3 ↔ n4 ↔ n5 ↔ n1 . . .). I tested the network using these routes:

- n1 to n4 (n1 → n5 → n4)
- n2 to n5 (n2 → n1 → n5)
- n3 to n1 (n3 → n2 → n1)
- n2 to n4 (n2 → n3 → n4)

The routing protocol correctly set up the forwarding table for each node, and each packet was successfully sent to the intended destination. To see the trace of the routing protocol and the packet forwarding, see "fiveinarow.txt" in the attached "trace" directory.

Unfortunately, I was unable to correctly take a link down to show that the routes adjusted accordingly.

2.3 Fifteen Nodes

Using the provided fifteen node configuration, I sent packets from various nodes to make sure that the routing protocol worked correctly. A few of the routes that I tested were:

- n1 to n12 (n1 → n4 → n5 → n12)
- n11 to n12 (n15 → n4 → n5 → n3 → n14 → n15)
- n9 to n12 (n13 → n6 → n1 → n4 → n5 → n13)
- n3 to n10 (n3 → n2 → n1 → n10)

The routing protocol correctly set up the forwarding table for each node, and each packet was successfully sent to the intended destination. To see the trace of the routing protocol and the packet forwarding, see “fifteen.txt” in the attached “trace” directory.

As with the previous experiment, I was unable to correctly tear down a link in the mesh, so I was unable to thoroughly test the recalculation of distance vector tables when a link went down or when a link was restored. Even though I was unable to figure out what needed to happen in order for the nodes to detect a broken or removed link, I have full confidence that the algorithm would have been able to properly determine shortest paths.