6.02 Fall 2012 Lecture #19

- Addressing, Forwarding, Routing
- Distance-vector routing
- Link-state routing

[Today: no failures]

The Problem: Distributed Methods for Finding Paths in Networks

- Addressing (how to name nodes?)
  - Unique identifier for global addressing
  - Link name for neighbors
- Forwarding (how does a switch process a packet?)
- Routing (building and updating data structures to ensure that forwarding works)
- Functions of the network layer

Forwarding

- Core function is conceptually simple
- `lookup(dst_addr)` in routing table returns route (i.e., outgoing link) for packet
- `enqueue(packet, link_queue)`
- `send(packet)` along outgoing link
- And do some bookkeeping before enqueue
  - Decrement hop limit (TTL); if 0, discard packet
  - Recalculate checksum (in IP, header checksum)

Shortest Path Routing

- Each node wants to find the path with minimum total cost to other nodes
  - We use the term “shortest path” even though we’re interested in min cost (and not min #hops)
- Several possible distributed approaches
  - Vector protocols, e.g. distance vector (DV)
  - Link-state protocols (LS)
Routing Table Structure

<table>
<thead>
<tr>
<th>Destination</th>
<th>Link (next-hop)</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ROUTE L1</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>B Self</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>C L1</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>D L2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>E ROUTE L1</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Routing table @ node B

Distance-Vector Routing

- **DV advertisement**
  - Send info from routing table entries: (dest, cost)
  - Initially just (self,0)
- **DV integration step [Bellman-Ford]**
  - For each (dest, cost) entry in neighbor’s advertisement
    - Account for cost to reach neighbor: (dest, my_cost)
    - my_cost = cost_in_advertisement + link_cost
  - Are we currently sending packets for this neighbor?
    - See if link matches what we have in routing table
    - If so, update cost in routing table to be my_cost
  - Otherwise, is my_cost smaller than existing route?
    - If so, neighbor is offering a better deal! Use it…
    - Update routing table so that packets for dest are sent to this neighbor

Distributed Routing: A Common Plan

- **Determining live neighbors**
  - Common to both DV and LS protocols
  - HELLO protocol (periodic)
    - Send HELLO packet to each neighbor to let them know who’s at the end of their outgoing links
    - Use received HELLO packets to build a list of neighbors containing an information tuple for each link: (timestamp, neighbor addr, link)
    - Repeat periodically. Don’t hear anything for a while → link is down, so remove from neighbor list.
- **Advertisement step (periodic)**
  - Send some information to all neighbors
  - Used to determine connectivity & costs to reachable nodes
- **Integration step**
  - Compute routing table using info from advertisements
  - Dealing with stale data

Distance-Vector Routing

- **DV advertisement**
  - Send info from routing table entries: (dest, cost)
  - Initially just (self,0)
- **DV integration step [Bellman-Ford]**
  - For each (dest, cost) entry in neighbor’s advertisement
    - Account for cost to reach neighbor: (dest, my_cost)
    - my_cost = cost_in_advertisement + link_cost
  - Are we currently sending packets for dest to this neighbor?
    - See if link matches what we have in routing table
    - If so, update cost in routing table to be my_cost
  - Otherwise, is my_cost smaller than existing route?
    - If so, neighbor is offering a better deal! Use it…
    - Update routing table so that packets for dest are sent to this neighbor

DV Example: round 2

Node A: update routes to B, D, E
Node B: update routes to A, E
Node C: no updates
Node D: update routes to A, B, E
Node E: update routes to A, B, C

Node A: update routes to B, D, E
Node B: update routes to A, E
Node C: no updates
Node D: update routes to A, B, E
Node E: update routes to A, B, C
DV Example: round 3

Node A: no updates
Node B: no updates
Node C: no updates
Node D: no updates
Node E: no updates

When link breaks: eliminate routes that use that link.

DV Example: round 4

Node A: update cost to B
Node B: update routes to A, C, D, E
Node C: update routes to B
Node D: no updates
Node E: update routes to B

DV Example: round 5

Node A: update route to B
Node B: no updates
Node C: no updates
Node D: no updates
Node E: no updates
Optimal substructure property fundamental to correctness of both Bellman-Ford and Dijkstra’s shortest path algorithms.

- Suppose shortest path from X to Y goes through Z. Then, the sub-path from X to Z must be a shortest path.

Proof of Bellman-Ford via induction on number of walks on shortest (min-cost) paths:
- Easy when all costs > 0 and synchronous model (see notes)
- Harder with distributed async model (not in 6.02)

How long does it take for distance-vector routing protocol to converge?
- Time proportional to largest number of hops considering all the min-cost paths

Link-State Routing
- Advertisement step
  - Send information about its links to its neighbors (aka link state advertisement or LSA):
    
    `[seq#, [(nbhr1, linkcost1), (nbhr2, linkcost2), ...]]`
  - Do it periodically (liveness, recover from lost LSAs)
- Integration
  - If `seq#` in incoming LSA > `seq#` in saved LSA for source node:
    - Update LSA for node with new `seq#`, neighbor list rebroadcast LSA to neighbors (~ flooding)
    - Remove saved LSAs if `seq#` is too far out-of-date
  - Result: Each node discovers current map of the network
- Build routing table
  - Periodically each node runs the same shortest path algorithm over its map (e.g., Dijkstra’s alg)
  - If each node implements computation correctly and each node has the same map, then routing tables will be correct

Correctness & Performance

- Periodically originate LSA
- LSA travels each link in each direction
  - Don’t bother with figuring out which link LSA came from
- Termination: each node rebroadcasts LSA exactly once
  - Use sequence number to determine if new, save latest seq
- Multiple opportunities for each node to hear any given LSA
  - Time required: number of links to cross network
**Integration Step: Dijkstra’s Algorithm (Example)**

Suppose we want to find paths from A to other nodes

**Dijkstra’s Shortest Path Algorithm**

- Initially
  - nodeset = [all nodes] = set of nodes we haven’t processed
  - spcost = {me: 0, all other nodes: ∞} # shortest path cost
  - routes = {me: --, all other nodes: ?} # routing table
- while nodeset isn’t empty:
  - find u, the node in nodeset with smallest spcost
  - remove u from nodeset
  - for v in [u’s neighbors]:
    - d = spcost(u) + cost(u,v) # distance to v via u
    - if d < spcost(v): # we found a shorter path!
      - spcost[v] = d
      - routes[v] = routes[u] (or if u == me, enter link from me to v)
- Complexity: N = number of nodes, L = number of links
  - Finding u (N times): linear search=O(N), using heapq=O(log N)
  - Updating spcost: O(L) since each link appears twice in neighbors

---

**Another Example**

Finding shortest paths from A:

<table>
<thead>
<tr>
<th>LSAa</th>
<th>A: [(B, 19), (C, 7)]</th>
<th>B: [(A, 19), (C, 11), (D, 4)]</th>
<th>C: [(A, 7), (B, 11), (D, 15), (E, 5)]</th>
<th>D: [(B, 4), (C, 15), (E, 13)]</th>
<th>E: [(C, 5), (D, 13)]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td>a</td>
<td>Nodeset</td>
<td>spcost</td>
<td>route</td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>---</td>
<td>---------</td>
<td>--------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>[A,B,C,D,E]</td>
<td>0 6 7 7 7</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>[B,C,D,E]</td>
<td>0 19 7 6 6</td>
<td>19</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>[B,D,E]</td>
<td>0 16 7 22 12</td>
<td>7</td>
<td>L1</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>[B,D]</td>
<td>0 18 7 22 12</td>
<td>11</td>
<td>L1</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>[D]</td>
<td>0 18 7 22 12</td>
<td>15</td>
<td>L1</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>[]</td>
<td>0 18 7 22 12</td>
<td>13</td>
<td>L1</td>
</tr>
</tbody>
</table>