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, esp. *distance vector* (DV)
  - Link-state protocols (LS)

(Assume all costs $\geq 0$)
Routing Table Structure

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

Routing table @ node B

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 1

Node A: update routes to B, C
Node B: update routes to A, C, D
Node C: update routes to A, B, D, E
Node D: update routes to B, C, E
Node E: update routes to C, D

Subscript indicates node that gave better route
DV Example: round 2

\[
\begin{align*}
\{A': (L0,19), &\quad B': (L0,4), \\
\quad C': (L1,11), &\quad D': (None,0), \\
\quad E': (L2,13) &\}\ \\
\{A': (None,0), &\quad B': (L0,19), \\
\quad C': (L1,7), &\quad D': (L1,18), \\
\quad E': (L3,5) &\}\ \\
\{A': (L0,7), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (L3,5) &\}\ \\
\{A': (L0,7), &\quad B': (L1,18), \\
\quad C': (L1,7), &\quad D': (L1,22), \\
\quad E': (L1,12) &\}\ \\
\{A': (None,0), &\quad B': (L1,11), \\
\quad C': (None,0), &\quad D': (L2,15), \\
\quad E': (L3,5) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (L2,13) &\}\ \\
\{A': (None,0), &\quad B': (L1,11), \\
\quad C': (None,0), &\quad D': (L2,15), \\
\quad E': (None,0) &\}\ \\
\{A': (L0,7), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (L0,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}.
\end{align*}
\]

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

DV Example: Break a Link

\[
\begin{align*}
\{A': (L1,18), &\quad B': (None,0), \\
\quad C': (L1,15), &\quad D': (None,0), \\
\quad E': (L2,13) &\}\ \\
\{A': (None,0), &\quad B': (L1,18), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (L0,7), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}\ \\
\{A': (None,0), &\quad B': (L0,4), \\
\quad C': (None,0), &\quad D': (None,0), \\
\quad E': (None,0) &\}.
\end{align*}
\]

Node A: update cost to B.
Node B: update routes to A, C, D, E.
Node C: update routes to B, D, E.
Node D: no updates.
Node E: update routes to B.
Correctness & Performance

• 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):
    \[ \text{seq#, } [(\text{nbhr1, linkcost1}), (\text{nbhr2, linkcost2}), \ldots] \]
  – 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
    – Flooding LSA to neighbors
  – 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
### LSA Flooding

- 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

Suppose we want to find paths from A to other nodes.

#### LSA: \([F, \text{seq}, (G, 8), (C, 2)]\)

- Periodically originate LSA
- LSA travels each link in each direction
- 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

### Dijkstra’s Shortest Path Algorithm

- Initially
  - \(\text{nodeset} = \{\text{all nodes}\} = \text{set of nodes we haven’t processed}\)
  - \(\text{spcost} = \{\text{me}=0, \text{all other nodes} = \infty\}\) # shortest path cost
  - \(\text{routes} = \{\text{me}=-, \text{all other nodes} = ?\}\) # 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:
    - \(d = \text{spcost}(u) + \text{cost}(u,v)\) # distance to \(v\) via \(u\)
    - if \(d < \text{spcost}(v)\): # we found a shorter path!
      - \(\text{spcost}(v) = d\)
      - \(\text{routes}(v) = \text{routes}(u)\) (or if \(u = \text{me}\), enter link from me to \(v\))
- Complexity: \(N = \text{number of nodes}, L = \text{number of links}\)
  - 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>Step</th>
<th>(u)</th>
<th>(\text{Nodeset})</th>
<th>(\text{spcost})</th>
<th>(\text{route})</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>[A, B, C, D, E]</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>[B, C, D, E]</td>
<td>7</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>[B, D, E]</td>
<td>12</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>[B, D]</td>
<td>15</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>[D]</td>
<td>23</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>[]</td>
<td>27</td>
<td></td>
</tr>
</tbody>
</table>