Shortest-Path Routing
Reading: Sections P&D 4.2

EE122: Intro to Communication Networks
Fall 2006 (MW 4:00-5:30 in Donner 155)
Vern Paxson
TAs: Dilip Antony Joseph and Sukun Kim
http://inst.eecs.berkeley.edu/~ee122/
Materials with thanks to Jennifer Rexford, Ion Stoica and colleagues at Princeton and UC Berkeley

What we have learnt so far ...

• Sending IP packets from source to destination through a series of routers
  – Router looks up destination IP address in a table
  – Forwards the packet to the corresponding next hop

• Sending Ethernet frames from source to destination through a series of switches
  – Switch looks up destination MAC address in a table
  – Forwards the frame on the corresponding port

• What important task did we NOT talk about?
  – How are these forwarding tables filled with information?
  – Routing

• What about self-learning in switches?
What is Routing?

• A famous quotation from RFC 791
  “A name indicates what we seek. An address indicates where it is. A route indicates how we get there.”
  -- Jon Postel

Why Does Routing Matter?

• We need good end-to-end performance
  – Find the shortest/best path
    • Propagation delay, throughput, packet loss

• Ensure efficient use of network resources
  – Balance traffic over the routers and links
  – Avoid congestion by directing traffic to lightly-loaded links

• Withstand disruptions
  – Failures, maintenance, and load balancing
  – Limit packet loss and delay during changes
Know Thy Network

- Routing requires knowledge of the network structure
- Centralized global state
  - Single entity knows the complete network structure
  - Can calculate all routes centrally
  - Problems with this approach?
- Distributed global state
  - Every router knows the complete network structure
  - Independently calculates routes
  - Problems with this approach?
- Distributed only local state
  - Every router knows only about its neighboring routers
  - Independently calculates routes
  - Problems with this approach?

Link State Routing
E.g. Algorithm: Dijkstra
E.g. Protocol: OSPF

Distance Vector Routing
E.g. Algorithm: Bellman Ford
E.g. Protocol: RIP

Modeling a Network

- Modeled as a graph
  - Routers \rightarrow nodes
  - Link \rightarrow edges
    - Edge cost
      - delay
      - congestion level
- Goal of Routing
  - Determine a “good” path through the network from source to destination
  - Good means usually the shortest path
Link State Routing

- Each router has a complete picture of the network

- How does each router get the global state?
  - Each router reliably floods information about its neighbors to every other router (more later)

- Each router independently calculates the shortest path from itself to every other router
  - Dijkstra’s Shortest Path Algorithm

Dijkstra’s Shortest Path Algorithm

- Named after Edsger W. Dijkstra (1930-2002)

- INPUT
  - Net topology, link costs known to all nodes

- OUTPUT
  - Least cost paths from one node (“source”) to all other nodes
Notation

- \( c(i,j) \): link cost from node \( i \) to \( j \); cost infinite if not direct neighbors
- \( D(v) \): current value of cost of path from source to destination \( v \)
- \( p(v) \): predecessor node along path from source to \( v \), that is next to \( v \)
- \( S \): set of nodes whose least cost path definitively known

Dijsktra’s Algorithm

1. **Initialization:**
   2. \( S = \{A\} \);
   3. for all nodes \( v \)
   4. if \( v \) adjacent to \( A \)
   5. then \( D(v) = c(A,v) \);
   6. else \( D(v) = \infty \);
   7.

8. **Loop**
   9. find \( w \) not in \( S \) such that \( D(w) \) is a minimum;
   10. add \( w \) to \( S \);
   11. update \( D(v) \) for all \( v \) adjacent to \( w \) and not in \( S \):
   12. \( D(v) = \min( D(v), D(w) + c(w,v) ) \);
      // new cost to \( v \) is either old cost to \( v \) or known
      // shortest-path cost to \( w \) plus cost from \( w \) to \( v \)
   13. until all nodes in \( S \);
Example: Dijkstra’s Algorithm

**Initialization:**
1. \( S = \{A\}; \)
2. for all nodes \( v \)
3. if \( v \) adjacent to \( A \)
4. then \( D(v) = c(A,v) \);
5. else \( D(v) = \infty \);

... 

**Loop**
8. find \( w \) not in \( S \) s.t. \( D(w) \) is a minimum;
9. add \( w \) to \( S \);
10. update \( D(v) \) for all \( v \) adjacent to \( w \) and not in \( S \):
11. \( D(v) = \min( D(v), D(w) + c(w,v) ) \);
12. until all nodes in \( S \);
Example: Dijkstra’s Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>start S</th>
<th>D(B),p(B)</th>
<th>D(C),p(C)</th>
<th>D(D),p(D)</th>
<th>D(E),p(E)</th>
<th>D(F),p(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2,A</td>
<td>5,A</td>
<td>1,A</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>AD</td>
<td>4,D</td>
<td>2,D</td>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADE</td>
<td>3,E</td>
<td>4,E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{Loop} \]
- \( \text{find } w \text{ not in } S \text{ s.t. } D(w) \text{ is a minimum; } \)
- \( \text{add } w \text{ to } S; \)
- \( \text{update } D(v) \text{ for all } v \text{ adjacent to } w \text{ and not in } S; \)
- \( D(v) = \min( D(v), D(w) + c(w,v) ); \)
- \( \text{until all nodes in } S; \)
### Example: Dijkstra’s Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>start S</th>
<th>D(B),p(B)</th>
<th>D(C),p(C)</th>
<th>D(D),p(D)</th>
<th>D(E),p(E)</th>
<th>D(F),p(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2,A</td>
<td>5,A</td>
<td>1,A</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>AD</td>
<td>4,D</td>
<td>2,D</td>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADE</td>
<td>3,E</td>
<td>4,E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ADEB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADEBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... **Loop**

9. find w not in S s.t. D(w) is a minimum;
10. add w to S;
11. update D(v) for all v adjacent to w and not in S:
12. \( D(v) = \min( D(v), D(w) + c(w,v) ) \);
13. until all nodes in S;

---

### Example: Dijkstra’s Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>start S</th>
<th>D(B),p(B)</th>
<th>D(C),p(C)</th>
<th>D(D),p(D)</th>
<th>D(E),p(E)</th>
<th>D(F),p(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>2,A</td>
<td>5,A</td>
<td>1,A</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>AD</td>
<td>4,D</td>
<td>2,D</td>
<td>∞</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ADE</td>
<td>3,E</td>
<td>4,E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>ADEB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ADEBC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>ADEBCF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... **Loop**

9. find w not in S s.t. D(w) is a minimum;
10. add w to S;
11. update D(v) for all v adjacent to w and not in S:
12. \( D(v) = \min( D(v), D(w) + c(w,v) ) \);
13. until all nodes in S;
The Forwarding Table

- Running Dijkstra at node A gives the shortest path from A to all destinations
- We then construct the forwarding table

<table>
<thead>
<tr>
<th>Destination</th>
<th>Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>(A,B)</td>
</tr>
<tr>
<td>C</td>
<td>(A,D)</td>
</tr>
<tr>
<td>D</td>
<td>(A,D)</td>
</tr>
<tr>
<td>E</td>
<td>(A,D)</td>
</tr>
<tr>
<td>F</td>
<td>(A,D)</td>
</tr>
</tbody>
</table>

- Why have separate routing and forwarding tables?

Complexity

- How much processing does running the Dijkstra algorithm take?
- Assume a network consisting of n nodes
  - Each iteration: need to check all nodes, w, not in S
  - n*(n+1)/2 comparisons: O(n^2)
  - More efficient implementations possible: O(n*log(n))
Oscillations

- Assume link cost = amount of carried traffic

- How can you avoid oscillations?

Obtaining Global State

- Flooding
  - Each router sends link-state information out through its links
  - The next node sends it out through all of its links
    - except the one where the information arrived
Flooding the Link State

- **Reliable flooding**
  - Ensure all nodes receive link-state information
  - Ensure all nodes use the latest version

- **Challenges**
  - Packet loss
  - Out-of-order arrival

- **Solutions**
  - Acknowledgments and retransmissions
  - Sequence numbers

Transient Disruptions

- **Detection delay**
  - A node does not detect a failed link immediately
  - … and forwards data packets into a “blackhole”
  - Depends on timeout for detecting lost hellos
### Transient Disruptions

- **Inconsistent link-state database**
  - Some routers know about failure before others
  - The shortest paths are no longer consistent
  - Can cause transient forwarding loops

![Diagram of transient disruptions](image.png)

A and D think that this is the path to C  
E thinks that this is the path to C

### Scaling Link-State Routing

- **Overhead of link-state routing**
  - Flooding link-state packets throughout the network
  - Running Dijkstra’s shortest-path algorithm
  - Becomes unscalable when 100s of routers

- **Introducing hierarchy through “areas”**

![Diagram of area hierarchy](image.png)

area border router

Area 0
Area 1
Area 2
Area 3
Area 4
5 Minute Break

• After the break: Distance Vector Routing

Distance Vector Routing

• Each router knows the links to its immediate neighbors
  – Does not flood this information to the whole network

• Each router has some idea about the shortest path to each destination
  – E.g.: Router A: I can get to router B with cost 11 via next hop router D
  – Routers exchange this information with their neighboring routers
    • Again, no flooding the whole network
  – Routers update their idea of the best path using info from neighbors
Bellman-Ford Algorithm

- Named after Richard Bellman and Ford
- INPUT
  - Link costs to each neighbor
- OUTPUT
  - Next hop to each destination and the corresponding cost
  - Does not give the complete path to the destination

Bellman-Ford - Overview

- Each router maintains a table
  - Row for each possible destination
  - Column for each directly-attached neighbor to node
  - Entry in row Y and column Z of node X ➔ best known distance from X to Y, via Z as next hop
- Each local iteration caused by:
  - Local link cost change
  - Message from neighbor
- Notify neighbors only if least cost path to any destination changes
  - Neighbors then notify their neighbors if necessary

Each node:

wait for (change in local link cost or msg from neighbor)

recompute distance table

if least cost path to any dest has changed, notify neighbors
Distance Vector Algorithm (cont'd)

1 Initialization:
2 for all neighbors V do
3 if V adjacent to A
4 D(A, V) = c(A, V);
5 else
6 D(A, V) = ∞;
7 loop:
8 wait (until A sees a link cost change to neighbor V
9 or until A receives update from neighbor V)
10 if (D(A, V) changes by d)
11 for all destinations Y through V do
12 D(A, Y) = D(A, Y) + d
13 else if (update D(V, Y) received from V)
14 /* shortest path from V to some Y has changed */
15 D(A, Y) = D(A, V) + D(V, Y);
16 if (there is a new minimum for destination Y)
17 send D(A, Y) to all neighbors
18 forever

Example: Distance Vector Algorithm

<table>
<thead>
<tr>
<th>Node</th>
<th>D(A,B)</th>
<th>D(A,C)</th>
<th>D(B,A)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dest</td>
<td>Cost</td>
<td>NextHop</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>∞</td>
<td>-</td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: for simplicity in this lecture examples we show only the shortest distances to each destination
Example: 1st Iteration (C → A)

\[ D(A, D) = D(A, C) + D(C, D) = 7 + 1 = 8 \]

Characteristics:
- Node A:
  - Destination: \(B\), Cost: 2, Next Hop: B
  - Destination: \(C\), Cost: 7, Next Hop: C
  - Destination: \(D\), Cost: 8, Next Hop: C

- Node B:
  - Destination: \(A\), Cost: 2, Next Hop: A
  - Destination: \(C\), Cost: 1, Next Hop: C
  - Destination: \(D\), Cost: 3, Next Hop: D

Example: 1st Iteration (B → A, C → A)

Characteristics:
- Node A:
  - Destination: \(B\), Cost: 2, Next Hop: B
  - Destination: \(C\), Cost: 3, Next Hop: B
  - Destination: \(D\), Cost: 5, Next Hop: B

- Node B:
  - Destination: \(A\), Cost: 2, Next Hop: A
  - Destination: \(C\), Cost: 1, Next Hop: C
  - Destination: \(D\), Cost: 3, Next Hop: D

- Node C:
  - Destination: \(A\), Cost: 7, Next Hop: A
  - Destination: \(B\), Cost: 1, Next Hop: B
  - Destination: \(D\), Cost: 1, Next Hop: D

- Node D:
  - Destination: \(A\), Cost: ∞, Next Hop: -
  - Destination: \(B\), Cost: 3, Next Hop: B
  - Destination: \(C\), Cost: 1, Next Hop: C
Example: End of 1\textsuperscript{st} Iteration

![Diagram showing network nodes and message flow between nodes A, B, C, and D.]

Node A
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
B & 2 & B \\
C & 3 & B \\
D & 5 & B \\
\hline
\end{array}\]

Node B
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
A & 2 & A \\
C & 1 & C \\
D & 2 & C \\
\hline
\end{array}\]

Node C
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
A & 3 & B \\
B & 1 & B \\
D & 1 & D \\
\hline
\end{array}\]

Node D
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
A & 5 & B \\
B & 2 & C \\
C & 1 & C \\
\hline
\end{array}\]


---

Example: End of 2\textsuperscript{nd} Iteration

![Diagram showing network nodes and message flow between nodes A, B, C, and D.]

Node A
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
B & 2 & B \\
C & 3 & B \\
D & 4 & B \\
\hline
\end{array}\]

Node B
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
A & 2 & A \\
C & 1 & C \\
D & 2 & C \\
\hline
\end{array}\]

Node C
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
A & 3 & B \\
B & 1 & B \\
D & 1 & D \\
\hline
\end{array}\]

Node D
\[\begin{array}{|c|c|c|}
\hline
\text{Dest.} & \text{Cost} & \text{NextHop} \\
\hline
A & 4 & C \\
B & 2 & C \\
C & 1 & C \\
\hline
\end{array}\]

**Example: End of 3rd Iteration**

<table>
<thead>
<tr>
<th>Node A</th>
<th>Dest.</th>
<th>Cost</th>
<th>NextHop</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>3</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>4</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node B</th>
<th>Dest.</th>
<th>Cost</th>
<th>NextHop</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>2</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

**Node C**

<table>
<thead>
<tr>
<th>Dest.</th>
<th>Cost</th>
<th>NextHop</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

**Node D**

<table>
<thead>
<tr>
<th>Dest.</th>
<th>Cost</th>
<th>NextHop</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
</tr>
</tbody>
</table>

Loop:

- else if (update D(V, Y) received from V)
- D(A, Y) = D(A, V) + D(V, Y);
- if (there is a new minimum for destination Y)
- send D(A, Y) to all neighbors
- forever

Nothing changes → algorithm terminates

**Distance Vector: Link Cost Changes**

```
7 loop:
8 wait (until A sees a link cost change to neighbor V
9     or until A receives update from neighbor V)
10 if (D(A, V) changes by d)
11     for all destinations Y through V do
12         D(A, Y) = D(A, Y) + d
13 else if (update D(V, Y) received from V)
14     D(A, Y) = D(A, V) + D(V, Y);
15     if (there is a new minimum for destination Y)
16     send D(A, Y) to all neighbors
17 forever
```

**Node B**

<table>
<thead>
<tr>
<th>A</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

**Node C**

<table>
<thead>
<tr>
<th>A</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>

**Node D**

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
</tbody>
</table>

“good news travels fast”
Distance Vector: Count to Infinity Problem

7 loop:
8 wait (until $A$ sees a link cost change to neighbor $V$
9 or until $A$ receives update from neighbor $V$)
10 if ($D(A, V)$ changes by $d$)
11 for all destinations $Y$ through $V$ do
12 $D(A, Y) = D(A, Y) + d$;
13 else if (update $D(V, Y)$ received from $V$)
14
B also maintains shortest distance to $A$ through $C$, which is 6.
Thus $D(B, A)$ becomes 6!

Distance Vector: Poisoned Reverse

- If $C$ routes through $B$ to get to $A$:
  - $C$ tells $B$ its ($C$'s) distance to $A$ is infinite
    (so $B$ won't route to $A$ via $C$)
  - Will this completely solve count to infinity problem?

Node B

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>4</td>
<td>A</td>
<td>6</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>C</td>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>N</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>B</td>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

Node C

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>6</td>
<td>A</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>N</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>A</td>
<td>5</td>
<td>A</td>
<td>A</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>B</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>

Link cost changes here; $B$ updates $D(B, A) = 60$ as $C$ has advertised $D(C, A) = \infty$

Algorithm terminates
Link State vs. Distance Vector

Per-node message complexity
- LS: \( O(e) \) messages
  - \( e \): number of edges
- DV: \( O(d) \) messages, many times
  - \( d \): node’s degree

Complexity/Convergence
- LS: \( O(n^2) \) computation
- DV: convergence time varies
  - may be routing loops
  - count-to-infinity problem

Robustness: what happens if router malfunctions?
- LS:
  - node can advertise incorrect link cost
  - each node computes only its own table
- DV:
  - node can advertise incorrect path cost
  - each node’s table used by others; error propagate through network

Are we done?
- No. We still have to take care of:
  - Scaling to Internet size
  - Routing policy issues
- Next Lecture
  - Internet scale routing by Prof. Scott Shenker
Conclusions

• Routing is a distributed algorithm
  – Different from forwarding
  – React to changes in the topology
  – Compute the shortest paths

• Two main shortest-path algorithms
  – Dijkstra $\rightarrow$ link-state routing (e.g., OSPF)
  – Bellman-Ford $\rightarrow$ distance vector routing (e.g., RIP)

• Convergence process
  – Changing from one topology to another
  – Transient periods of inconsistency across routers

• Next time: BGP and policy
  – Reading: Section 4.3.3, 4.3.4

Backup Slides

• To be covered if time permits

• Refer to textbook for more information about these topics
When to Initiate Flooding

- **Topology change**
  - Link or node failure
  - Link or node recovery

- **Configuration change**
  - Link cost change

- **Periodically**
  - Refresh the link-state information
  - Typically (say) 30 minutes
  - Corrects for possible corruption of the data

Convergence

- **Getting consistent routing information to all nodes**
  - E.g., all nodes having the same link-state database

- **Consistent forwarding after convergence**
  - All nodes have the same link-state database
  - All nodes forward packets on shortest paths
  - The next router on the path forwards to the next hop
Convergence Delay

• Time elapsed before every router has a consistent picture of the network

• Sources of convergence delay
  – Detection latency
  – Flooding of link-state information
  – Shortest-path computation
  – Creating the forwarding table

• Performance during convergence period
  – Lost packets due to blackholes and TTL expiry
  – Looping packets consuming resources
  – Out-of-order packets reaching the destination

• Very bad for VoIP, online gaming, and video

Reducing Convergence Delay

• Faster detection
  – Smaller hello timers
  – Link-layer technologies that can detect failures

• Faster flooding
  – Flooding immediately
  – Sending link-state packets with high-priority

• Faster computation
  – Faster processors on the routers
  – Incremental Dijkstra algorithm

• Faster forwarding-table update
  – Data structures supporting incremental updates
Detecting Topology Changes

• Beaconing
  – Periodic “hello” messages in both directions
  – Detect a failure after a few missed “hellos”

  “hello”

• Performance trade-offs
  – Detection speed
  – Overhead on link bandwidth and CPU
  – Likelihood of false detection