Routing and Routers

Router architecture
Logical architecture of a router

Switch Fabric ("Backplane")

Connects inputs to outputs based on addresses

Routing Processor

Looks up address in routing table to find "best" output port for each packet

Addresses

Control Signals For Switch

Input Ports

Output Ports

A typical router design

L3 Controller

Backplane

L2 Controller

switch

port

port
Cisco 7400

- 320 Gbps, 270Mpps (OC-192)
- Crossbar switch
- 16 slots
  - 15x10Gbe or
  - 15 x OC192 or
  - 120xFastEthernet or
  - 15120 x T1s
- 6’ x 1.5’ x 2’
- 390# fully populated, 4.7KW
- $100K+ (2005)

Cisco 12416 Internet Router

- 320 Gbps, 270Mpps (OC-192)
- Crossbar switch
- 16 slots
  - 15x10Gbe or
  - 15 x OC192 or
  - 120xFastEthernet or
  - 15120 x T1s
- 6’ x 1.5’ x 2’
- 390# fully populated, 4.7KW
- $100K+ (2005)
Cisco 7613 Edge Router

- 30 Mpps
- 256 Gbps
- 33" x 17" x 18"
- 13 slots
  • Up to OC-48, 10GbE
  • 240# fully populated, 4KW
  • $50K+ (2005)

Procket PRO/8812 Internet Router

- 960Gbps, 1.2Gpps
- 12 linecard slots
- Shared-memory-based switch fabric
- Up to OC-768
- 37.5" x 17.4" x 25.5"
- 571# fully loaded, 5.4 KW
  • ~$250K (2005)
Switch Fabrics

• Many types of designs

• Considerations:
  - Speed
  - Flexibility (Blocking behavior)
  - Scalability (cost/size/power/…)

A simple way to build a switch fabric

• Advantages
  - Not custom technology
  - Highly Flexible
  - Smart/Programmable

• Disadvantages
  - Single bus limits scalability
  - Hard to build general-purpose design that’s as fast as a custom design
Non-blocking switch (Crossbar)

- Advantages
  - Any combination of inputs can be connected to any combination of outputs ("non-blocking")
  - Fast – just one switch between input and output

- Disadvantages
  - Not scalable: $O(n^2)$

Multi-stage switch (Banyan)

- Advantages
  - Scalable: $O(n \log_2 n)$
  - Modest number of switches input to output ($\log_2 n$)

- Disadvantages
  - Blocking
A Banyan example

We want to connect these three inputs to the indicated output ports.

A combination of inputs that blocks

One of the inputs will have to remain queued until the next clock cycle.
Routing algorithms

- How do we build the routing tables?
- How do we keep the routing tables current?
- Some dead ends:
  - Static tables -- networks are too dynamic
  - Human-built tables -- networks are too large
- Ideally, we’d like the routers to build and maintain their own routing tables
“Cost” of a route

- There are often multiple paths from any router to a destination. We need to be able to pick the “best” one of them.

- We don’t always simply want to determine the shortest route to the destination:
  - A path with more hops may be faster
  - Some paths have a higher monetary cost than others
  - We may have to take a longer path to meet a user performance requirement (e.g., bandwidth)

- In general, we want to select the path that minimizes whatever “cost” we are interested in. The cost may be a single factor (delay, $ cost, etc) or a weighted combination of factors.

- Cost factors are usually associated with links

Path cost examples

Costs from router y to destination D:

<table>
<thead>
<tr>
<th>Path</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>y z D</td>
<td>2</td>
</tr>
<tr>
<td>y x z D</td>
<td>5</td>
</tr>
<tr>
<td>y w x z D</td>
<td>8</td>
</tr>
</tbody>
</table>
Distance Vector Routing

The idea:
- Each router knows the cost to each of its immediate neighbors
- Each router builds a “distance vector” that contains the total cost of the best-known route to every destination (initial costs = ∞)
- At intervals, each router sends its DV to all neighbors
- When a router R receives a DV from a neighbor N, R scans the table to see if there are any cases where, for Destination D:
  
  \[ N\text{'s cost to get to } D + \text{R's cost to get to } N < \text{R's current cost to get to } D \]

If there are any such cases, R updates its table so that future traffic for D is sent to N.

Example:
- Assume router R knows a path to D with a cost of 25.
- R’s neighbor N knows a path to D with a cost of 20.
- If the cost from R to N is 3, then R can get to D through N with a total cost of 23.
- Since this is less than the current cost, R will update its routing table so that all traffic for D goes to N.

Note about the following DV example

Although the final state is not meaningfully affected by the ordering of actions between the different routers, interim results can vary.

For clarity in the following example, I am assuming that the algorithm runs in network-wide cycles and that in each cycle, each router broadcasts its DV BEFORE it processes incoming DV’s.
Initial DV's

Note: 99 is used here to represent "infinity" – that is, it is meant to be a number that is larger than any cost that could legitimately arise in the algorithm. For some networks and cost strategies, it may be necessary to use a larger number.

First Cycle – focusing on w:
1. neighbors send DV to w

From y
A - 99
B - 99
C - 99
D - 99

From x
A - 99
B - 99
C - 99
D - 99
First Cycle – focusing on \( w \):
2. \( w \) adds cost to get to the neighbor

From \( y \):
- \( A \) 99 + 102
- \( B \) 99 + 102
- \( C \) 0 + 3
- \( D \) 99 + 3

From \( x \):
- \( A \) 99 + 1100
- \( B \) 0 + 1
- \( C \) 99 + 1100
- \( D \) 99 + 1100

First Cycle – focusing on \( w \):
3. \( w \) updates its RT with lowest costs

From \( y \):
- \( A \) 99 + 102
- \( B \) 99 + 102
- \( C \) 0 + 3
- \( D \) 99 + 3

From \( x \):
- \( A \) 99 + 1100
- \( B \) 0 + 1
- \( C \) 99 + 1100
- \( D \) 99 + 1100
State after First Cycle

Underlining indicates changes that occurred this cycle.

Second cycle - focusing on x

A w 1
B x 0
C y 1
D z 4

A w 0
B x 1
C y 3
D - 99

A w 1
B x 0
C y 1
D z 4

A w 0
B x 1
C y 3
D - 99

A w 1
B x 0
C y 1
D z 4

A w 0
B x 1
C y 3
D - 99

A w 0
B x 1
C y 3
D - 99

A w 3
B x 1
C y 0
D z 2

A w 3
B x 1
C y 0
D z 2

A w 0
B x 1
C y 3
D - 99

A w 0
B x 1
C y 3
D - 99
Second cycle - focusing on z

State after second cycle
State after third cycle

A problem with the basic DV algorithm

The costs will continue to slowly rise without stopping. This is called the "Count to Infinity" problem.
Fixes for the Count to Infinity problem

- Define a small number as “infinity” so that problem becomes apparent sooner
- Don’t send cost to the neighbor you received your current cost from (“split horizon”) 
- Send infinity to the neighbor you received your current cost from (“split horizon with poison reverse”)

The Link-State routing algorithm

- Used for Open Shortest-Path First (OSPF) routing in Internet
- The idea:
  - Each router discovers cost to immediate neighbors (by pinging, etc)
  - At intervals, this info is flooded to all other routers in a “Link State Packet”. This gives all routers a map of the network and link costs.
  - Each router runs a part-finding algorithm (e.g, Dijkstra’s Shortest Path Algorithm) to calculate least-cost paths.
Dijkstra’s Shortest Path Algorithm: Terminology and Notation

- Goal is to determine the minimum-cost path from Source node “S” to Destination node “D”.
- “W” = the ID of the “working node”
- “Cn” is the sum of the link costs for every link in the least-cost presently known path from S to node n.
- “L(a,b)” = cost associated with the link from node a to node b. L(a,b) = infinity if a and b are not directly connected.
- “Label” of a node = n(Cn,y)
  - “n” = the node’s ID
  - “y” = ID of the preceding node on the least-cost presently known path from S to node n.
- At various stages of the algorithm, a node label can be “tentative” or “permanent.”
  - A tentative label is one that might be changed at a later stage of the algorithm.
  - A permanent label will not be changed. We indicate permanent labels with bold text.

Dijkstra’s Shortest Path Algorithm

0. W=S. Assign S the permanent label S(-,0) For every node r, r<>S, assign the label r(-, 99)

While D is not permanently labeled:
1. For each neighbor N of W that is not permanently labeled, calculate x=CW+L(W,N). If x < CN, assign the tentative label N(W, x).
2. Examine the entire graph and find the tentatively-labeled node, w, with the minimum cost in its label. Set W = w.
3. Change W’s tentative label to a permanent label.

End While

5. Record the name of D. The “copying node”, C = D.
6. Record the name P, where C(P,x) is the label of the copying node.
7. P is the new copying node. If P <> S, repeat 5-6.
8. The least-cost path is the reverse order of the recorded node names.

* 99 is used here to represent “infinity” – that is, it is intended to be a number that is larger than any cost that could legitimately arise in the Dijkstra’s calculation. For some networks and cost strategies, it may be necessary to use a larger number.
Dijkstra’s Shortest Path Example

Find the least-cost path from A (g) to D (j)

Working node

0. W=S. Assign S the permanent label S(-,0).
   For every node r, r<>S, assign the label r(-, 99)
Dijkstra’s Shortest Path Example

1. For each neighbor N of W that is not permanently labeled, assign the tentative label N (W, CW+L(W,N))

2. Examine the entire graph and find the tentatively-labeled node, w, with the minimum cost in its label. Set W = w.
Dijkstra’s Shortest Path Example

1. For each neighbor N of W that is not permanently labeled, assign the tentative label N (W, CW+L(W,N))

2. Change W’s tentative label to a permanent label.

Dijkstra’s Shortest Path Example

3. Change W’s tentative label to a permanent label.
Dijkstra’s Shortest Path Example

2. Examine the entire graph and find the tentatively-labeled node, \( w \), with the minimum cost in its label. Set \( W = w \).

Dijkstra’s Shortest Path Example

3. Change \( W \)’s tentative label to a permanent label.
Dijkstra’s Shortest Path Example

1. For each neighbor \( N \) of \( W \) that is not permanently labeled, assign the tentative label \( N (W, CW+L(W,N)) \)

2. Examine the entire graph and find the tentatively-labeled node, \( w \), with the minimum cost in its label. Set \( W = w \).
Dijkstra’s Shortest Path Example

3. Change W’s tentative label to a permanent label.
Dijkstra’s Shortest Path Example

Destination is permanently labeled – copy path back to S and read in reverse

Reverse path = j – h – k – g
Path = g – k – h – j

Reverse path = j – h – k – g
Path = g – k – h – j