

Internetworking I

Chapter 3 in the textbook

With slides from Mike Katchabaw

Assignment 2

- Don't know how to start?
- Let's look at a simple RLE encoder...



```
#include <stdio.h>

int
main(void)
{
    int c = getchar();
    while (c != EOF) {
        int d, counter = 1;
        while ((d = getchar()) == c && counter < 255)
            counter++;
        putchar(c);
        putchar(counter);
        c = d;
    }
    return 0;
}
```

What to do about it?

- Use Lempel-Ziv instead ;)
- Or...
 - Shorten the “length” codes
 - Change the symbol size
 - Put special markers in the stream to turn RLE on and off

Lempel-Ziv implementation

- I think you'll get the best results if you compress bit-wise instead of byte-wise
 - Make an freadbit() function
 - Start with basic codes of 0 and 1

Hamming implementation

- You don't need to do matrices!

0

Hamming implementation

- You don't need to do matrices!

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \pmod{2}$$

Hamming implementation

- $\text{int } p1 = d1 \wedge d2 \wedge d4;$
- $\text{int } p2 = d1 \wedge d3 \wedge d4;$

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \pmod{2}$$

Hamming implementation

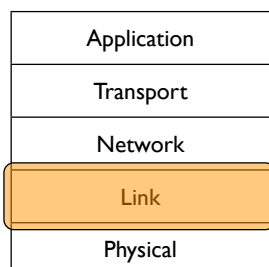
- $\text{int } \text{check1} = p1 \wedge d1 \wedge d2 \wedge d4;$
- $\text{int } \text{check2} = p2 \wedge d1 \wedge d3 \wedge d4;$

$$\begin{matrix} p1 & p2 & d1 & p3 & d2 & d3 & d4 \\ \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 3 \\ 2 \\ 3 \end{pmatrix} \equiv \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} \pmod{2} \end{matrix}$$

Basic Hamming implementation

- Encoding:
 - while not end of file:
 - read in 4 bits, write 7 bits
- Decoding:
 - while not end of file:
 - read in 7 bits, correct errors, write 4 bits

From link layer to network layer



- We can form frames ("packets") at the link layer
- We can get information from one node to another
 - Reliably!
- We can form networks

Key Network-Layer Functions

- forwarding*: move packets from router's input to appropriate router output
 - analogy:
 - routing: process of planning trip from source to destination
 - forwarding: process of getting through single intersection
- routing*: determine route taken by packets from source to destination
 - Routing algorithms*

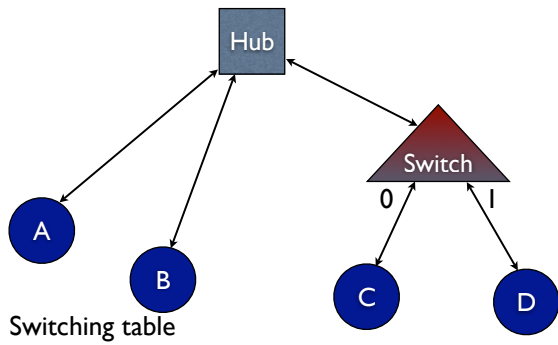
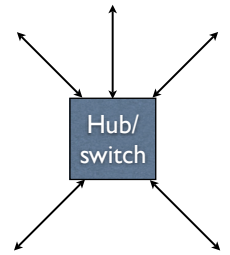
Forwarding

- Repeating
 - One input port and one output port, used usually for degrading signal strength

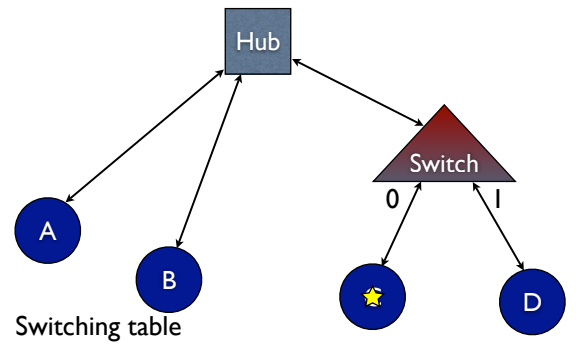


Forwarding

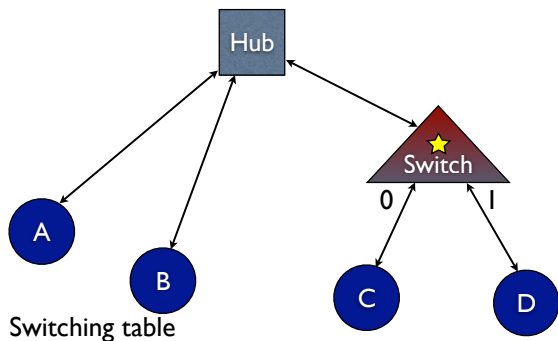
- Flooding
 - Used by hubs (and switches sometimes)
 - Broadcasts input to every output port
- Switching
 - Choosing only the appropriate port



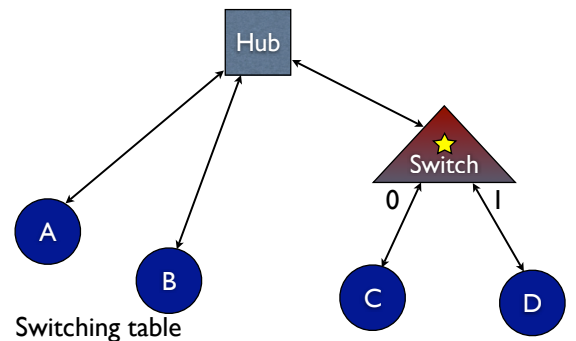
Switching table



Switching table

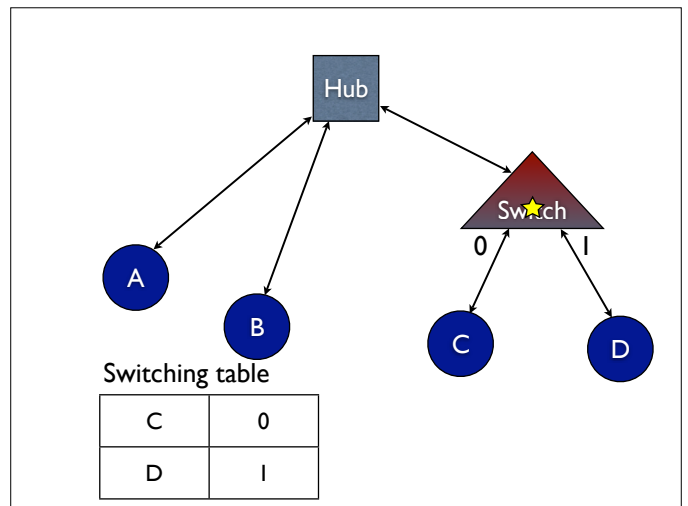
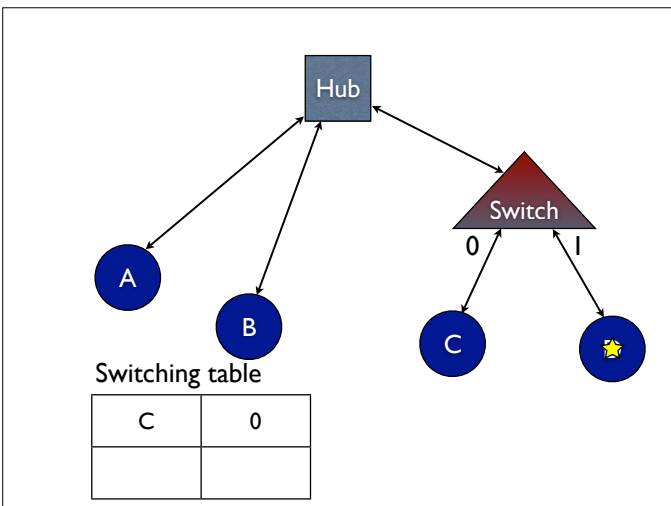
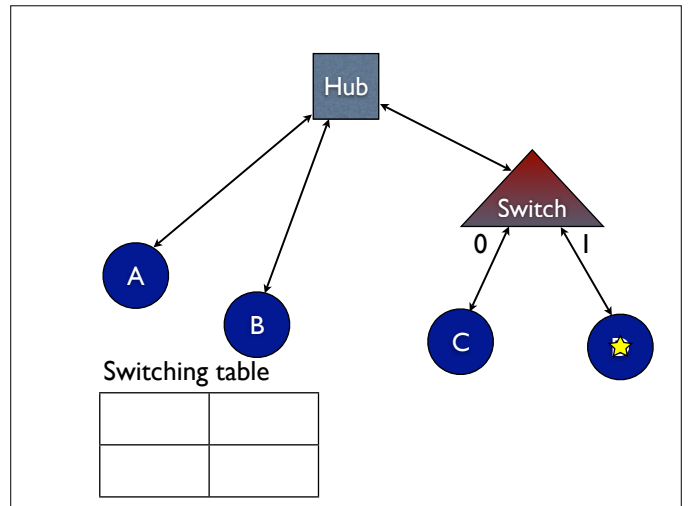
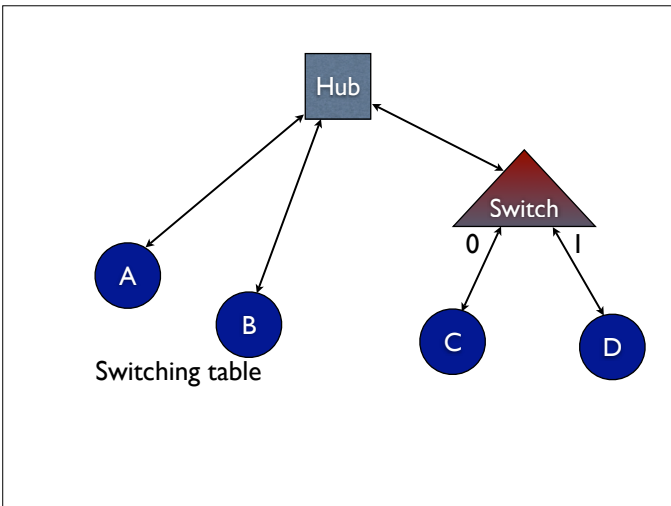
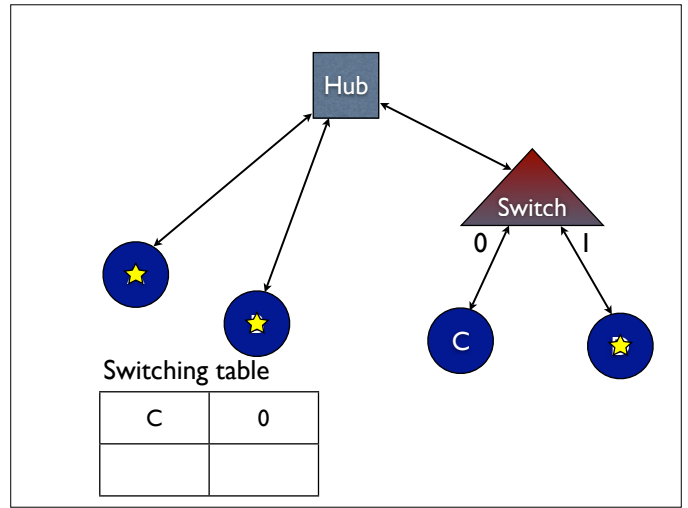
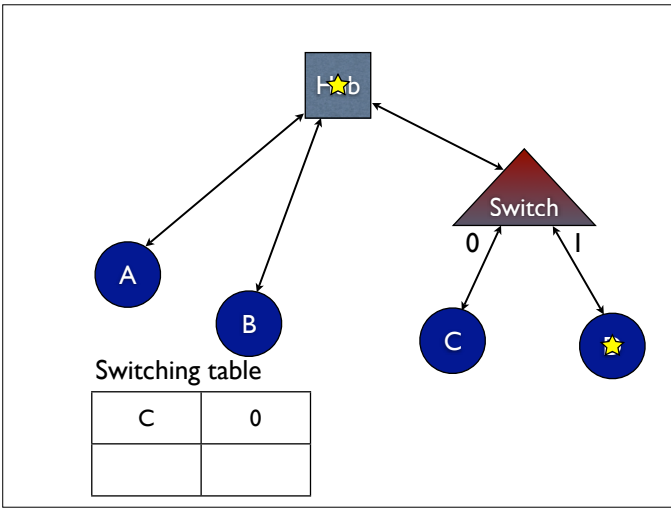


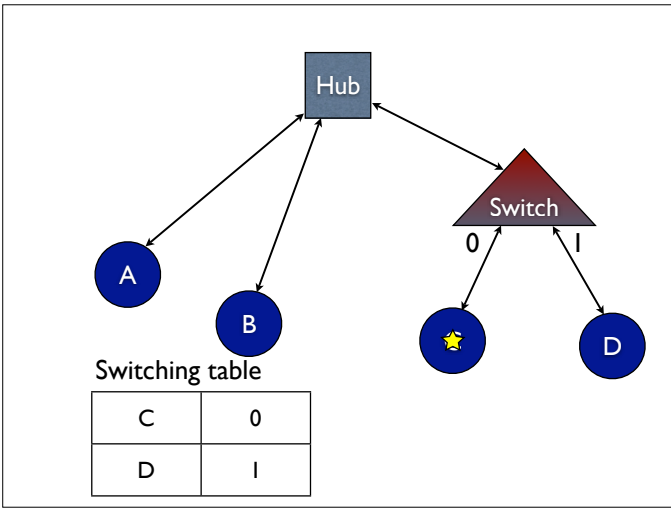
Switching table



Switching table

C	0



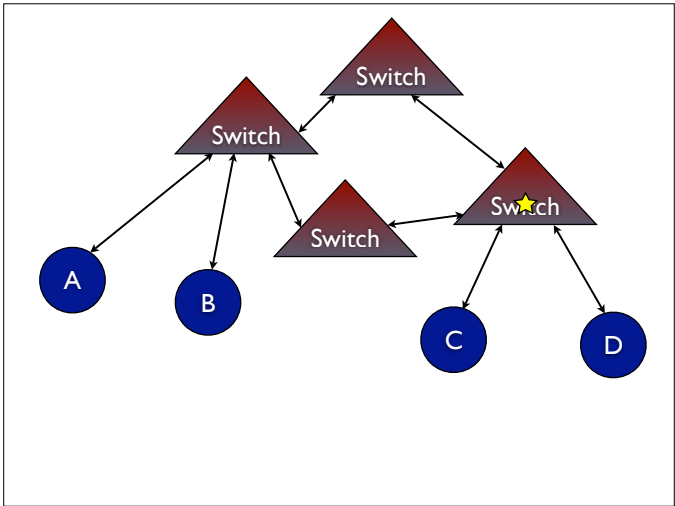
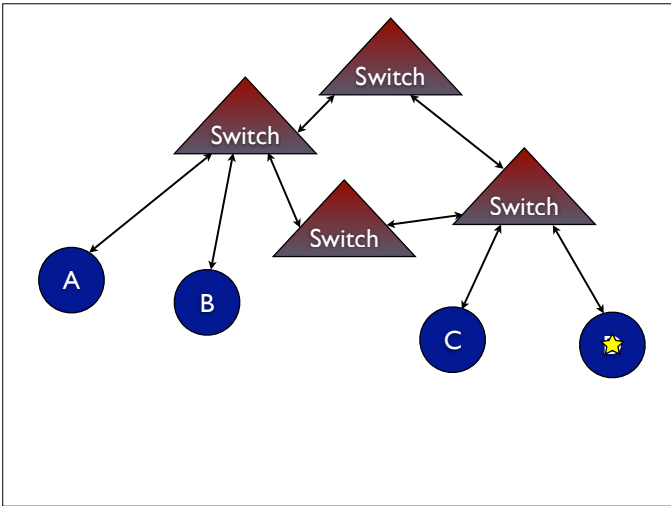
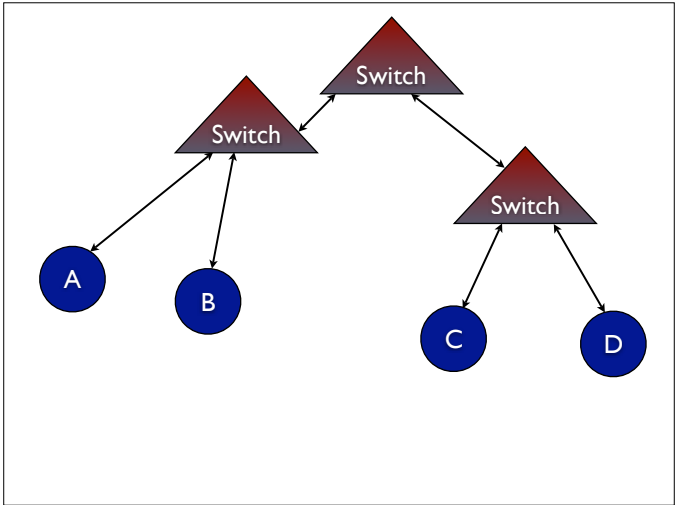


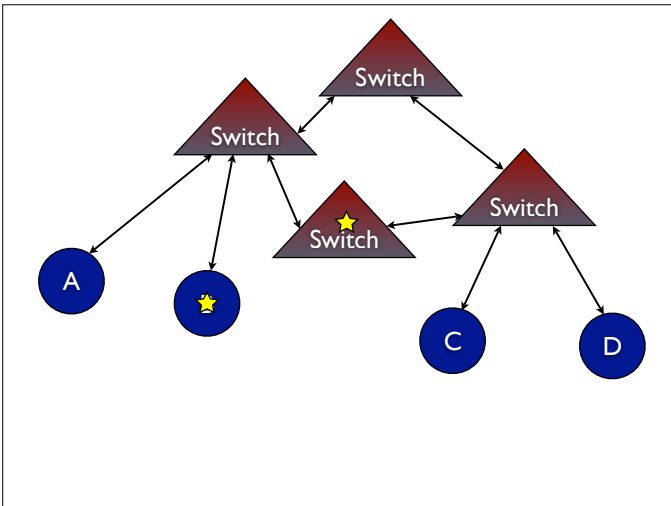
Other configurations

- What we've learned so far about self-learning switches is what the textbook calls "source routing"
- "Datagram routing" is essentially the same thing
- Difference: the forwarding table is manually set up instead of learnt

Source/datagram routing

- There are problems with source/datagram routing
- The forwarding table doesn't scale
 - One entry in the table per node in the network
 - If only there were some "network layer" to deal with this for us....
- And cycles!





A Link-State Routing Algorithm

Dijkstra's algorithm

- net topology, link costs known to all nodes
 - accomplished via "link state broadcast"
 - all nodes have same info
- computes least cost paths from one node ("source") to all other nodes
 - gives routing table for that node
- iterative: after k iterations, know least cost path to k destinations

Notation:

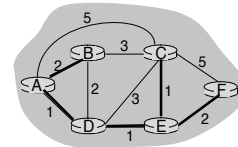
- $c(i,j)$: link cost from node i to j . cost is infinite if not direct neighbors
- $D(v)$: cost of path from source to destination v that currently has the least cost
- $p(v)$: previous node along path from source to v (a neighbor to v)
- N : set of nodes whose least cost path from the source is definitively known

Dijkstra's Algorithm for Source Node A

- 1 Initialization:
- 2 $N = \{A\}$
- 3 for all nodes v
- 4 if v adjacent to A
- 5 then $D(v) = c(A,v)$
- 6 else $D(v) = \text{infinity}$
- 7
- 8 Loop:
- 9 find node w not in N such that $D(w)$ is a minimum
- 10 add w to N
- 11 update $D(v)$ for all v adjacent to w and not in N :
- 12 $D(v) = \min(D(v), D(w) + c(w,v))$
- 13 l^* new cost to v is either old cost to v or known
- 14 shortest path cost to w plus cost from w to v *
- 15 until all nodes in N

Dijkstra's algorithm: example

Step	start	N	$D(B), p(B)$	$D(C), p(C)$	$D(D), p(D)$	$D(E), p(E)$	$D(F), p(F)$
→ 0	A		2,A	5,A	1,A	infinity	infinity
→ 1	AD		2,A	4,D		2,D	infinity
→ 2	ADE		2,A	3,E			4,E
→ 3	ADEB			3,E			4,E
→ 4	ADEBC						4,E
→ 5	ADEBCF						



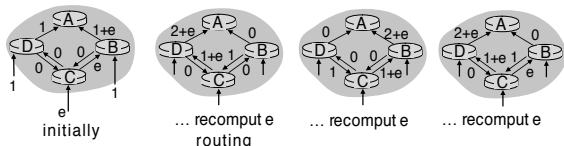
Dijkstra's algorithm, discussion

Algorithm complexity: n nodes

- each iteration: need to check all nodes, w , not in N
- $n^2(n+1)/2$ comparisons: $O(n^2)$
- more efficient implementations possible: $O(n \log n)$

Oscillations possible:

- e.g., link cost = amount of carried traffic



Implementing it in the network

- The root node sends a packet to its neighbours with time-to-live of 1
- Neighbours report cost (perhaps latency? bandwidth?)
- Root node distributes cost information to neighbours and sends out packets with time-to-live of 2, etc.

Implementing it in the network

- In the end, every node should know which links to use and which links cause loops
- As a bonus, it's a minimum spanning tree, so each we get a tree utilizing maximum bandwidth (or minimum latency)

Distance Vector Routing Algorithm

iterative:

- continues until no nodes exchange info
- *self-terminating*: no "signal" to stop

asynchronous:

- nodes need *not* exchange info/iterate in lock step!

distributed:

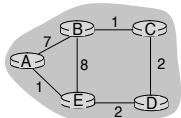
- each node communicates *only* with directly-attached neighbors

Distance Table data structure

- each node has its own table
- row for each possible destination
- column for each directly-attached neighbor to node
- example: in node X, for destination Y via neighbor Z:

$$D^X(Y,Z) = \text{distance from X to Y, via Z as next hop} \\ = c(X,Z) + \min_w \{D^Z(Y,w)\}$$

Distance Table: example



$$D^E(C,D) = c(E,D) + \min_w \{D^D(C,w)\} \\ = 2+2 = 4$$

$$D^E(A,D) = c(E,D) + \min_w \{D^D(A,w)\} \\ = 2+3 = 5 \text{ loop!}$$

$$D^E(A,B) = c(E,B) + \min_w \{D^B(A,w)\} \\ = 8+6 = 14 \text{ loop!}$$

		cost to destination via		
$D^E()$		A	B	D
destination	A	1	14	5
	B	7	8	5
	C	6	9	4
	D	4	11	2

Distance table gives routing table

		cost to destination via			Outgoing link to use, cost	
$D^E()$		A	B	D		
destination	A	1	14	5	A	A,1
	B	7	8	5	B	D,5
	C	6	9	4	C	D,4
	D	4	11	2	D	D,2

Distance table → Routing table

Distance Vector Routing: overview

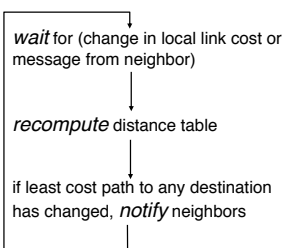
Iterative, asynchronous:
each local iteration caused by:

- local link cost change
- message from neighbor: a least cost path to a destination has changed

Distributed:

- each node notifies neighbors *only* when its least cost path to any destination changes
 - neighbors then notify their neighbors if necessary

Each node:



Distance Vector Algorithm:

At all nodes, X:

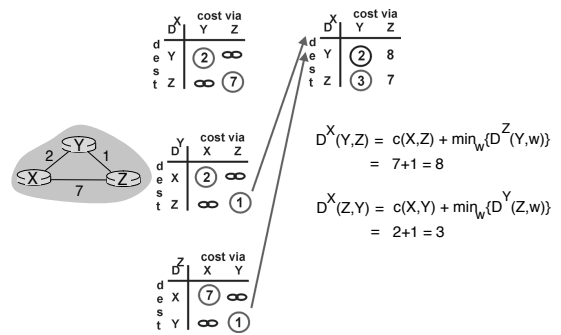
- 1 Initialization:
- 2 for all adjacent nodes v:
- 3 $D^X(*,v) = \text{infinity}$ /* the * operator means "for all rows" */
- 4 $D^X(v,v) = c(X,v)$
- 5 for all destinations, y
- 6 send $\min_w D^X(y,w)$ to each neighbor /* w over all X's neighbors */

Distance Vector Algorithm (continued):

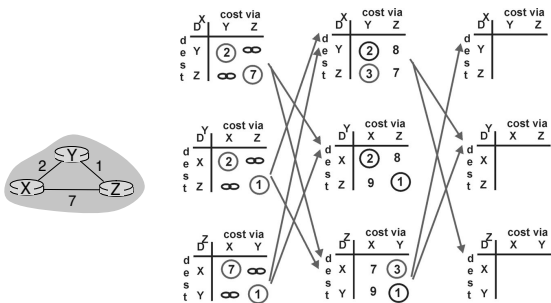
```

8 loop
9 wait (until I see a link cost change to neighbor V
10 or until I receive update from neighbor V)
11
12 if (c(X,V) changes by d)
13 /* change cost to all destinations via neighbor V by d */
14 /* note: d could be positive or negative */
15 for all destinations y:  $D^X(y,V) = D^X(y,V) + d$ 
16
17 else if (update received from V wrt destination Y)
18 /* shortest path from V to some Y has changed */
19 /* V has sent a new value for its  $\min_w D^V(Y,w)$  */
20 /* call this received new value "newval" */
21 for the single destination y:  $D^X(Y,V) = c(X,V) + \text{newval}$ 
22
23 if we have a new  $\min_w D^X(Y,w)$  for any destination Y
24 send new value of  $\min_w D^X(Y,w)$  to all neighbors
25
26 forever
    
```

Distance Vector Algorithm: example



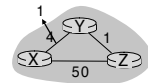
Distance Vector Algorithm: example



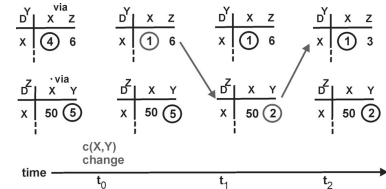
Distance Vector: link cost changes

Link cost changes:

- node detects local link cost change
- updates distance table (line 15)
- if cost changes in a least cost path, notify neighbors (lines 23,24)



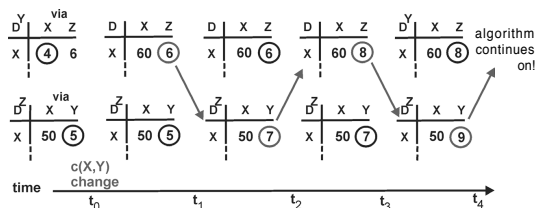
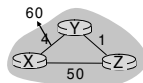
"good news travels fast"



Distance Vector: link cost changes

Link cost changes:

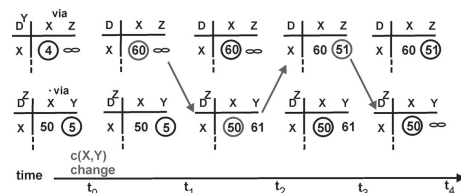
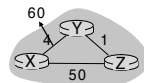
- good news travels fast
- bad news travels slow - "count to infinity" problem!



Distance Vector: poisoned reverse

If Z routes through Y to get to X :

- Z tells Y its (Z's) distance to X is infinite (so Y won't route to X via Z)
- will this completely solve count to infinity problem?



Comparison of LS and DV algorithms

Message complexity

- **LS:** with n nodes, E links, $O(nE)$ msgs sent each
- **DV:** exchange between neighbors only
 - convergence time varies

Speed of Convergence

- **LS:** $O(n^2)$ algorithm requires $O(nE)$ msgs
 - may have oscillations
- **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:

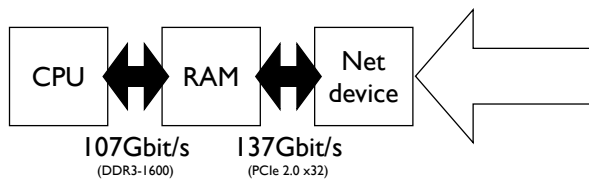
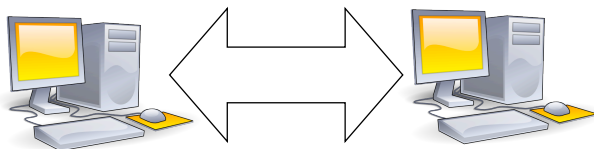
- DV node can advertise incorrect *path* cost
- each node's table used by others
 - error propagate through network

CS357b 89

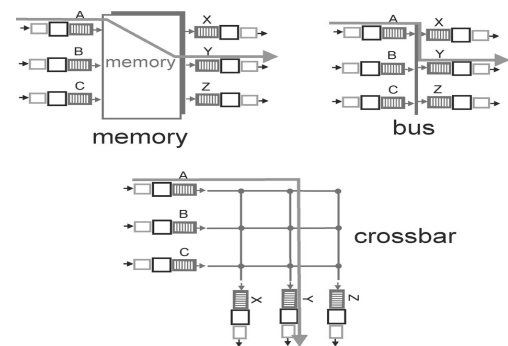
Switching fabrics

- Not such a concern for consumer-grade switches and routers
- For backbones, cutting down latency across a switch is a huge concern

Side note: performance



Three types of switching fabrics

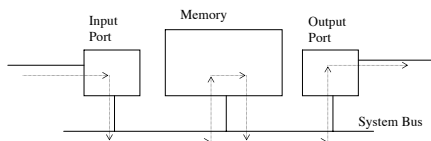


CS357b 24

Switching Via Memory

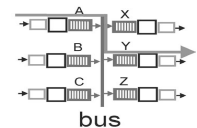
First generation routers:

- traditional computers with switching under direct control of CPU
- packet copied to system's memory
- speed limited by memory bandwidth (2 bus crossings per datagram)



CS357b 25

Switching Via a Bus



- datagram from input port memory to output port memory via a shared bus
- bus contention: switching speed limited by bus bandwidth
- 1 Gbps bus, Cisco 1900: sufficient speed for access and enterprise routers (not regional or backbone)

CS357b 26

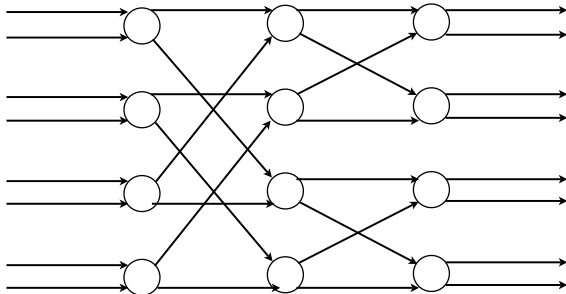
Switching Via An Interconnection Network (crossbar)

- ❑ Overcome bus bandwidth limitations
- ❑ Banyan networks, other interconnection nets initially developed to connect processors in multiprocessor computer architectures
- ❑ Advanced design: fragmenting datagram into fixed length cells, switch cells through the fabric.
- ❑ Cisco 12000: switches Gbps through the interconnection network

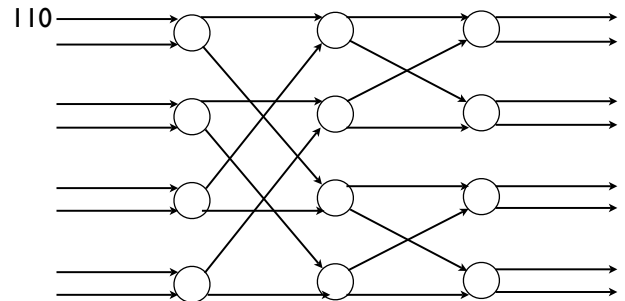
Self-routing fabrics (banyan)

- As a packet comes out of the input port, tag it with an address indicating which output port it should go to
- Send it through a network
- The address is stripped off as it goes through

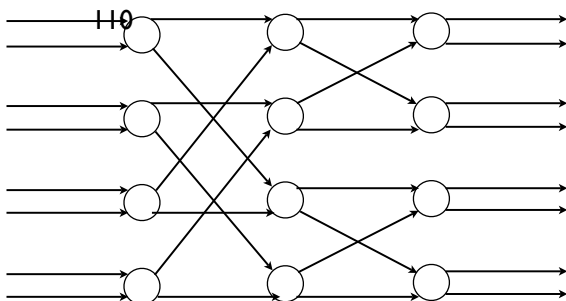
Banyan network



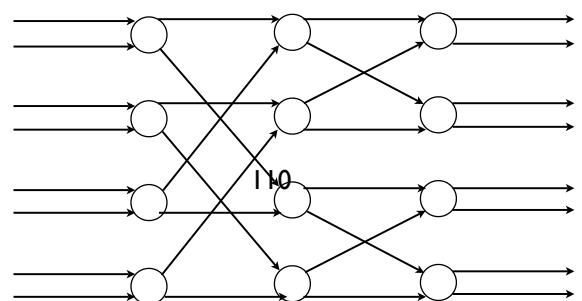
Banyan network



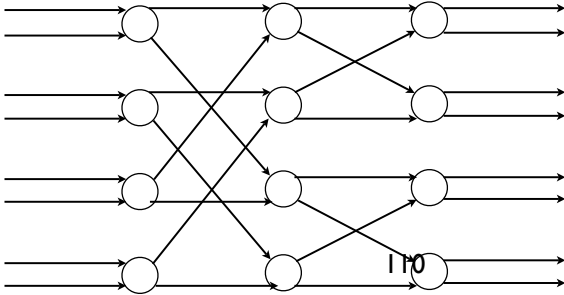
Banyan network



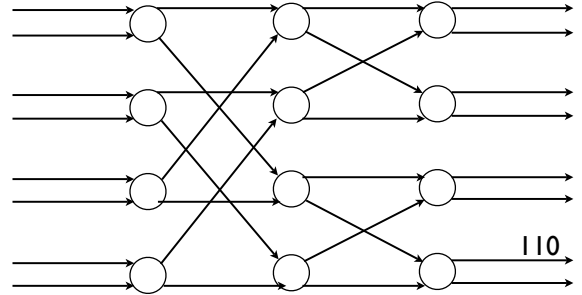
Banyan network



Banyan network



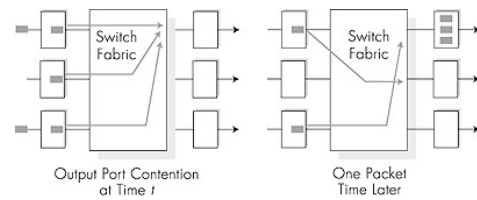
Banyan network



Banyan networks

- Piece together 2x2 switches to make them any (power of 2) size we like
- Very fast!
- Collisions still happen :(

Output port queuing



- buffering when arrival rate via switch exceeds output line speed
- *queueing (delay) and loss due to output port buffer overflow!*