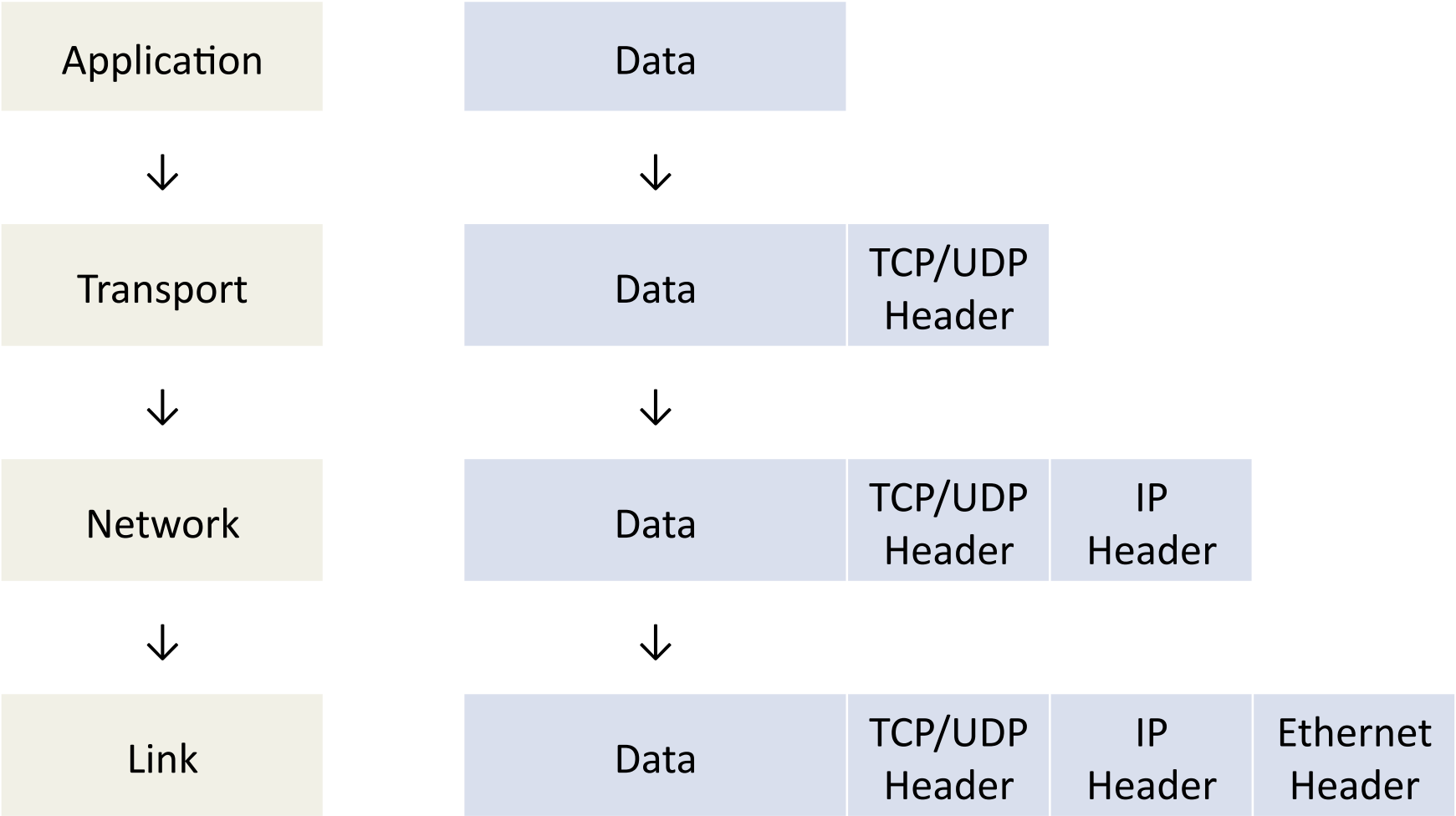
# IP Review

CS144 Review Session 3

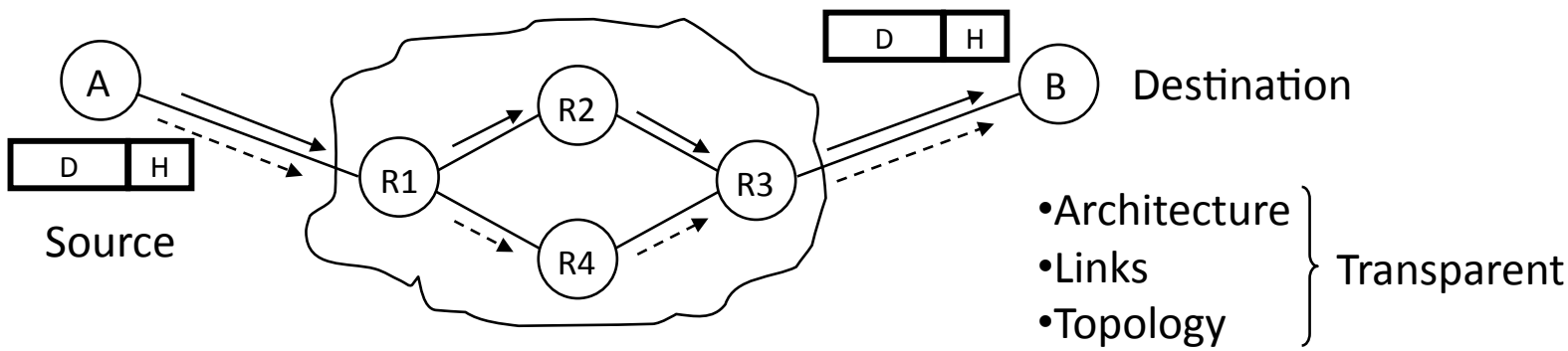
April 18, 2008

Ben Nham

# Layering Review



# Characteristics of IP




- Connectionless datagram service
  - No per-flow state in routers
  - Packets may appear out of sequence (successive packets are individually routed)
- Unreliable: packets may be dropped
- Best-effort: …but only if necessary

Credit: CS244A Handout 4

# Sending IP Packets

- Addressing
  - Gives endpoints unique names
  - Sets of addresses represented by subnets
- Forwarding
  - Determine the correct egress port for an incoming packet based on a routing table
- Routing
  - Algorithms to build routing tables in router
  - Purpose: choose best path to take from source to dest

# IP Addresses

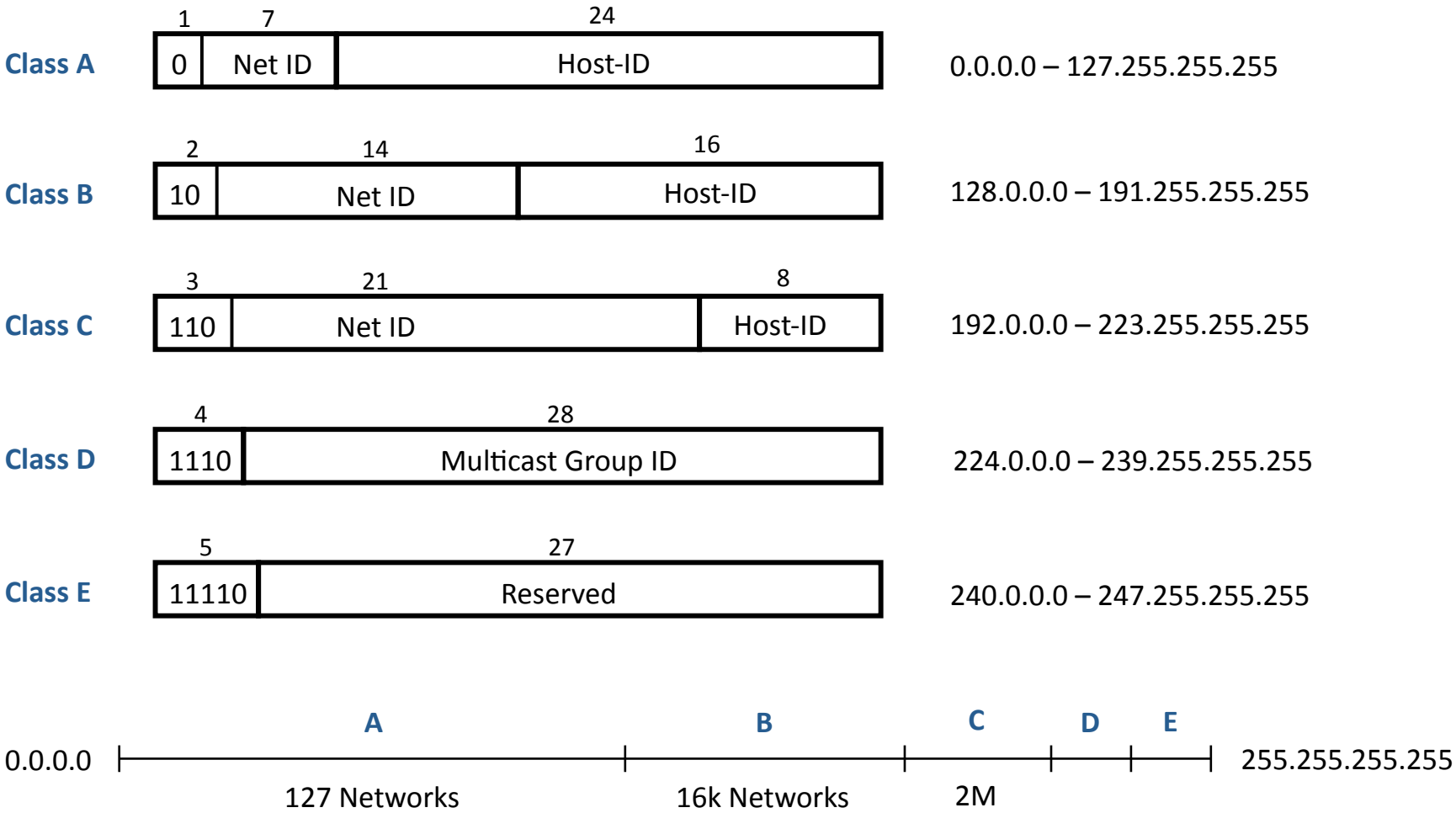
- Every network interface on a host has a globally unique IP address*
- 32-bit number, usually split into 4 octets:
  - 128.12.92.1

| | | | |
|---|---|---|---|
| 10000000 | 00001100 | 01011100 | 00000001 |

- Grouping IP addresses together is advantageous
  - Provides hierarchy
  - Consolidates routing table entries

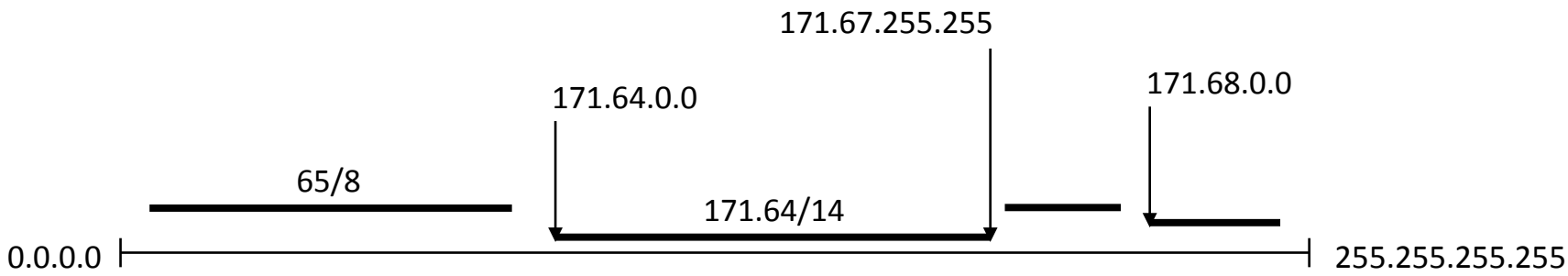
*Ignoring NAT and a few other details we get into later

# Classful Addressing (Deprecated)




Credit: CS244A Handout 4

# CIDR – Classless Inter-Domain Routing

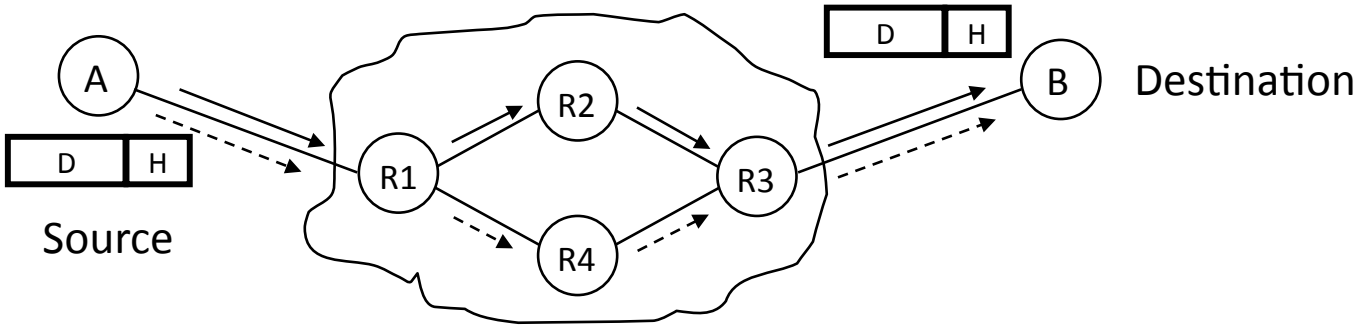



- Lets us use variable-length prefixes
  - 171.64.0.0/14 has room for 256k hosts ($2^{(32-14)}$)
    - IP address: 171.64.0.0
    - Netmask: 255.252.0.0

| | | | |
|---|---|---|---|
| 11111111 | 11111100 | 00000000 | 00000000 |

  - Maps to range: 171.64.0.0 – 171.67.255.255
- Try some lookups yourself
  - whois -h whois.arin.net 171.64.0.1
  - whois -h whois.arin.net 171.68.0.1

# Forwarding

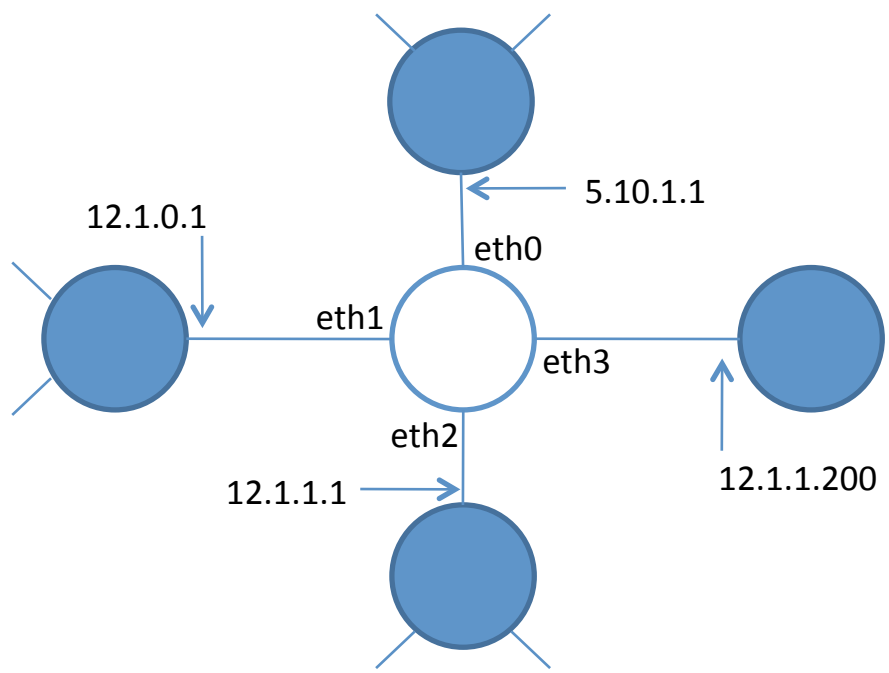



- Suppose R1 gets A's packet
  - Does it go out the top or bottom interface?
- Resolve by inspecting a routing table

# Routing Tables

- Matches if: ($IP_{dst}$ & netmask) == ($IP_{table}$ & netmask)
- Can have multiple matches
  - 12.1.1.200 matches every entry!
  - 0.0.0.0 entry matches everything (default route)
- Each interface has its own IP address

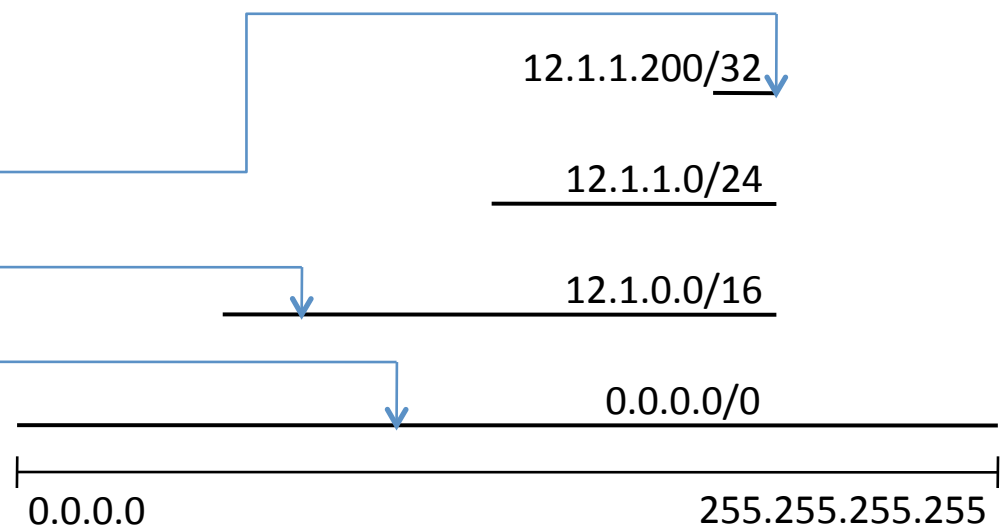
| IP Address | Netmask | Gateway | Interface |
|---|---|---|---|
| 0.0.0.0 | 0.0.0.0 | 5.10.1.1 | eth0 |
| 12.1.0.0 | 255.255.0.0 | 12.1.0.1 | eth1 |
| 12.1.1.0 | 255.255.255.0 | 12.1.1.1 | eth2 |
| 12.1.1.200 | 255.255.255.255 | 12.1.1.200 | eth3 |



# Longest Prefix Match

- If we have multiple matches, choose entry with longest prefix (largest netmask)
- What is the LPM for:
  - 12.1.1.200
  - 12.1.0.20
  - 128.12.92.53

| IP Address | Netmask | Gateway | Interface |
|---|---|---|---|
| 0.0.0.0 | 0.0.0.0 | 5.10.1.1 | eth0 |
| 12.1.0.0 | 255.255.0.0 | 12.1.0.1 | eth1 |
| 12.1.1.0 | 255.255.255.0 | 12.1.1.1 | eth2 |
| 12.1.1.200 | 255.255.255.255 | 12.1.1.200 | eth3 |



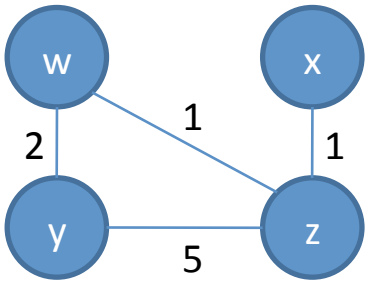
# Distance Vectors

- Distance vector contains cost to *every* destination node in the network
- For routing, also need next hop to every destination node
- Add a cost column to the routing table
  - Now routing table is DV: maps Dest IP → (Cost, Next Hop)

# Distributed DV Algorithm (Bellman-Ford)

- Initially: only know cost to ourselves and neighbors; everyone else has cost of infinity
- Every time we get a distance vector from a neighbor:
  - Store neighbor's distance vector
  - For each destination node t in DV:
    - Cost to get to t through a neighbor: cost to go to neighbor (in our DV) + cost to go from neighbor to t (in neighbor's DV)
    - Calculate cost to get to t through every neighbor as next hop. The cost to get to t is then the minimum of all these costs
- Periodically push our DV to neighbors
  - Local routing algorithm: don't know topology of entire network

# Distance Vector Example




| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **0** | W | **∞** | - | **2** | Y | **1** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **∞** | - | **0** | X | **∞** | - | **1** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **2** | W | **∞** | - | **0** | Y | **5** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **1** | W | **1** | X | **5** | Y | **0** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **0** | W | **2** | Z | **2** | Y | **1** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **2** | Z | **0** | X | **6** | Z | **1** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **2** | W | **6** | Z | **0** | Y | **3** | W |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **1** | W | **1** | Z | **3** | W | **0** | Z |

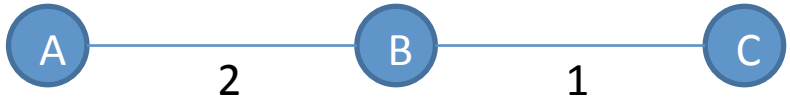
| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **0** | W | **2** | Z | **2** | Y | **1** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **2** | Z | **0** | X | **4** | Z | **1** | Z |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **2** | W | **4** | W | **0** | Y | **3** | W |

| W | | X | | Y | | Z | |
|---|---|---|---|---|---|---|---|
| **1** | W | **1** | Z | **3** | W | **0** | Z |

# Problems with Distance Vector

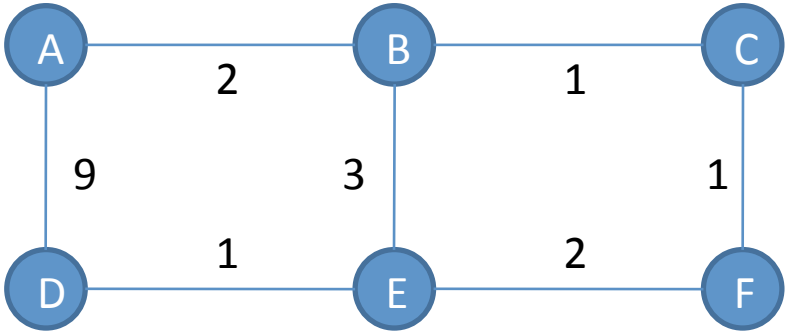



- Increase in link cost is propagated slowly
- Can "count to infinity"
  - What happens if we delete (B, C)?
  - B now tries to get to C through A, and increase its cost to C
  - A will see that B's cost of getting to C increased, and will increase its cost
  - Shortest path to C from B and A will keep increasing to infinity
- Partial solutions
  - Set infinity: set infinity to some finite value
  - Split horizon: don't advertise cost to next hop
  - Poison reverse: advertise infinite cost to next hop

# Dijkstra's Algorithm

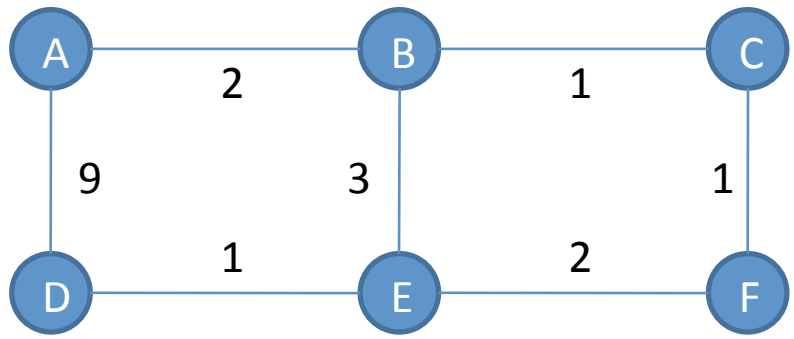
- Given a graph G and a starting vertex s, find shortest path from s to any other vertex in G
- Use greedy algorithm:
  - Maintain a set S of nodes for which we know the shortest path
  - On each iteration, grow S by one vertex, choosing shortest path through S to any other node not in S
  - If the cost from S to any other node has decreased, update it

# Example




| Explored Set S | Unexplored Set Q = V - S |
|---|---|
| A(0, -) | **B(0+2, A)**, C(∞, -), D(0+9, A), E(∞, -), F(∞, -) |
| | |
| | |
| | |
| | |
| | |

# Solution




| Explored Set S | Unexplored Set Q = V - S |
|---|---|
| A(0, -) | **B(0+2, A)**, C(∞, -), D(0+9, A), E(∞, -), F(∞, -) |
| A(0, -), B(2,A) | **C(2+1, B)**, D(9, A), E(2+3, B), F(∞, -) |
| A(0, -), B(2, A), C(3, B) | D(9, A), E(5, B), **F(3+1, B)** |
| A(0, -), B(2, A), C(3, B), F(4, B) | D(9, A), **E(5, B)** |
| A(0, -), B(2, A), C(3, B), F(4, B), E(5, B) | **D(5+1, B)** |
| A(0, -), B(2, A), C(3, B), F(4, B), E(5, B), D(6, B) | |

# Link-State (Using Dijkstra's)

- Algorithm must know the cost of every link in the network
  - Each node broadcasts LS packets to all other nodes
  - Contains source node id, costs to all neighbor nodes, TTL, sequence #
  - If a link cost changes, must rebroadcast
- Calculation for entire network is done locally

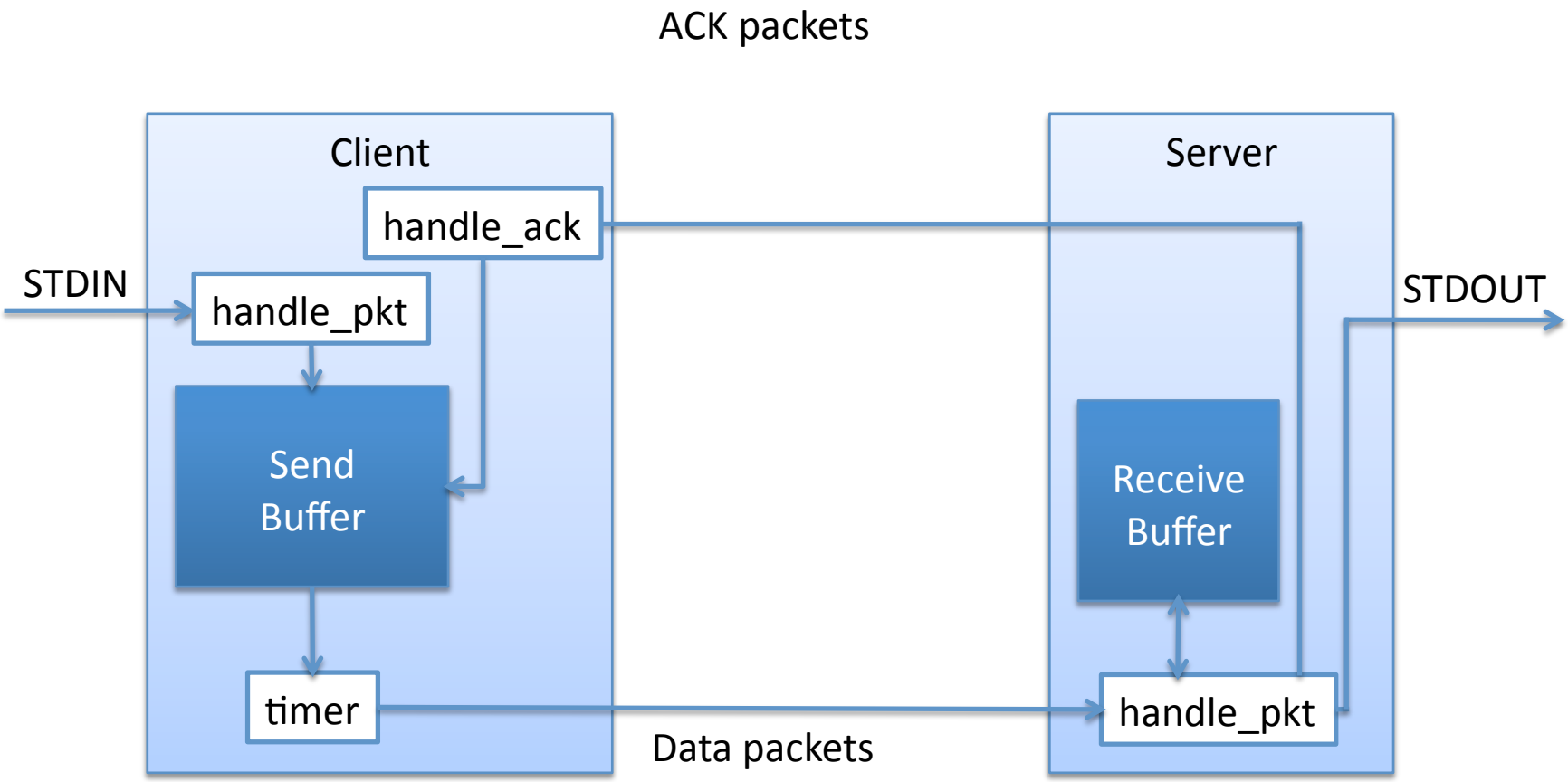
# Comparison between LS and DV

- Messages
  - In distance vector: Each node shares a distance vector (distance to every node in network) with its neighbor
  - In link state: Each node broadcasts a link state advertisement to the whole network
- How long does it take to converge?
  - DV: Varies, can be infinite
  - LS: $O(|E|^2)$; can be even less if we use a min-heap
- Robustness
  - DV: An incorrect distance vector can propagate through the whole network, since we are advertising path costs
  - LS: we can only incorrectly broadcast cost to neighbors

# Lab 3: Reliable Transport

- Half-duplex reliable stream of bytes from client to server
- Client reads data from stdin, packetizes data, sends data over network
- Server reads data from stdout
- Must handle lossy network, corrupted packets, out of order packets w/ buffering (i.e. linked list of packets)
  - Sender can only have up to 4 unacked packets outstanding: sender needs state
  - Receiver has to handle out of order and missing packets: receiver needs state

# Lab 3 Operation



# Implementation Notes

- Takes much more time than Labs 1 and 2, so start early
- Define some packet header format to put in front of every packet
  - Seq #, Ack #, type (SYN, ACK, …), etc.
- For the buffers, probably best to use built-in linked list/stack/queue library
  - #include <sys/queue.h>
  - Google for manpage
  - Tail-queue or doubly-linked list might be useful
- Endianness
  - Remember htonl/htons and ntohl/ntohs on ints and shorts
  - Otherwise your transfer won't work across different endian machines
- Can be selective repeat or Go-Back-N, up to you
  - GBN w/ cumulative ACKs is used by TCP and requires less state, so it's probably easier