IP Review

CS144 Review Session 3
April 18, 2008
Ben Nham
Layering Review

Application ➞ Data ➞ Transport ➞ Data ➞ Network ➞ Data ➞ Link ➞ Data

- TCP/UDP Header
- IP Header
- Ethernet Header
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
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
    
    \[
    \begin{array}{cccc}
    10000000 & 00001100 & 01011100 & 00000001 \\
    \end{array}
    \]

• 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 (Depreciated)

Class A
- Net ID: 0
- Host-ID: 7 to 24
- Range: 0.0.0.0 – 127.255.255.255

Class B
- Net ID: 10
- Host-ID: 14 to 16
- Range: 128.0.0.0 – 191.255.255.255

Class C
- Net ID: 110
- Host-ID: 21 to 8
- Range: 192.0.0.0 – 223.255.255.255

Class D
- Multicast Group ID
- Range: 224.0.0.0 – 239.255.255.255

Class E
- Reserved
- Range: 240.0.0.0 – 247.255.255.255

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
      - 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
• 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: \( (IP_{\text{dst}} \& \text{netmask}) == (IP_{\text{table}} \& \text{netmask}) \)
- 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

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Netmask</th>
<th>Gateway</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>0.0.0.0</td>
<td>5.10.1.1</td>
<td>eth0</td>
</tr>
<tr>
<td>12.1.0.0</td>
<td>255.255.0.0</td>
<td>12.1.0.1</td>
<td>eth1</td>
</tr>
<tr>
<td>12.1.1.0</td>
<td>255.255.255.0</td>
<td>12.1.1.1</td>
<td>eth2</td>
</tr>
<tr>
<td>12.1.1.200</td>
<td>255.255.255.255</td>
<td>12.1.1.200</td>
<td>eth3</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>IP Address</th>
<th>Netmask</th>
<th>Gateway</th>
<th>Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0.0.0</td>
<td>0.0.0.0</td>
<td>5.10.1.1</td>
<td>eth0</td>
</tr>
<tr>
<td>12.1.0.0</td>
<td>255.255.0.0</td>
<td>12.1.0.1</td>
<td>eth1</td>
</tr>
<tr>
<td>12.1.1.0</td>
<td>255.255.255</td>
<td>12.1.1.1</td>
<td>eth2</td>
</tr>
<tr>
<td>12.1.1.200</td>
<td>255.255.255.255</td>
<td>12.1.1.200</td>
<td>eth3</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>∞</td>
<td>-</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>-</td>
<td>0</td>
<td>X</td>
<td>∞</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>W</td>
<td>∞</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>1</td>
<td>X</td>
<td>5</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>2</td>
<td>Z</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>W</td>
<td>6</td>
<td>Z</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>1</td>
<td>Z</td>
<td>3</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>W</td>
<td>2</td>
<td>Z</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Z</td>
<td>0</td>
<td>X</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>W</td>
<td>4</td>
<td>W</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>W</td>
<td>1</td>
<td>Z</td>
<td>3</td>
</tr>
</tbody>
</table>
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** $S$

- $A(0, -)$

**Unexplored Set** $Q = V - S$

- $B(0+2, A)$
- $C(\infty, -)$
- $D(0+9, A)$
- $E(\infty, -)$
- $F(\infty, -)$

![Graph Diagram]
**Solution**

![Graph](image)

<table>
<thead>
<tr>
<th>Explored Set S</th>
<th>Unexplored Set Q = V - S</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(0, -)</td>
<td>B(0+2, A), C(\infty, -), D(0+9, A), E(\infty, -), F(\infty, -)</td>
</tr>
<tr>
<td>A(0, -), B(2,A)</td>
<td>C(2+1, B), D(9, A), E(2+3, B), F(\infty, -)</td>
</tr>
<tr>
<td>A(0, -), B(2, A), C(3, B)</td>
<td>D(9, A), E(5, B), F(3+1, B)</td>
</tr>
<tr>
<td>A(0, -), B(2, A), C(3, B), F(4, B)</td>
<td>D(9, A), E(5, B)</td>
</tr>
<tr>
<td>A(0, -), B(2, A), C(3, B), F(4, B), E(5, B)</td>
<td>D(5+1, B)</td>
</tr>
<tr>
<td>A(0, -), B(2, A), C(3, B), F(4, B), E(5, B), D(6, B)</td>
<td></td>
</tr>
</tbody>
</table>

---

**Diagram Description**

- **Nodes**: A, B, C, D, E, F
- **Edges**:
  - A to B: weight 2
  - B to C: weight 1
  - B to D: weight 3
  - C to D: weight 1
  - B to E: weight 2
  - E to F: weight 1

**Graph Configuration**

- **Set S**:
  - exploring set
- **Set Q**:
  - unexplored set

**Table Legend**

- **A**: Node A
- **B**: Node B
- **C**: Node C
- **D**: Node D
- **E**: Node E
- **F**: Node F
- **0**: Source node
- **∞**: End node
- **-, +**: Weights
- **V**: Total vertices
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

ACK packets

Client
- handle_ack
- handle_pkt
- Send Buffer
- timer

Server
- Receive Buffer
- handle_pkt

STDIN

Data packets

STDOUT
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