SWITCHING, FORWARDING, AND ROUTING

George Porter
Oct 4 and 9, 2018

ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Alex C. Snoeren, UC San Diego
Outline

1. Switching
2. Routing
3. Forwarding
4. UDP/TCP

SELECTIVE FORWARDING

- Only rebroadcast a frame to the LAN where its destination resides
  - If A sends packet to X, then bridge must forward frame
  - If A sends packet to B, then bridge shouldn’t
FORWARDING

- Need to know “destination” of frame
  - Destination address in frame header (48bit in Ethernet)
- Need know which destinations are on which LANs
  - One approach: statically configured by hand
    - Table, mapping address to output port (i.e. LAN)
  - But we’d prefer something automatic and dynamic
- Simple algorithm
  
  Receive frame $f$ on port $q$
  Lookup $f$.dest for output port /* know where to send it? */
  If $f$.dest found
    then if output port is $q$ then drop /* already delivered */
    else forward $f$ on output port;
  else flood $f$;
/* forward on all ports but the one where frame arrived*/

LEARNING BRIDGES

- Eliminate manual configuration by learning which addresses are on which LANs

- Basic approach
  - If a frame arrives on a port, then associate its source address with that port
  - As each host transmits, the table becomes accurate

- What if a node moves? Table aging
  - Associate a timestamp with each table entry
  - Refresh timestamp for each new packet with same source
  - If entry gets too stale, remove it

<table>
<thead>
<tr>
<th>Host</th>
<th>Port</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>W</td>
<td>2</td>
</tr>
<tr>
<td>X</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>3</td>
</tr>
<tr>
<td>Z</td>
<td>2</td>
</tr>
</tbody>
</table>
LEARNING EXAMPLE

Suppose $C$ sends frame to $D$ and $D$ replies back with frame to $C$

- $C$ sends frame, bridge has no info about $D$, so floods to both LANs
  - bridge notes that $C$ is on port 1
  - frame ignored on upper LAN
  - frame received by $D$

LEARNING EXAMPLE

Suppose $C$ sends frame to $D$ and $D$ replies back with frame to $C$

- $D$ generates reply to $C$, sends
  - bridge sees frame from $D$
  - bridge notes that $D$ is on port 2
  - bridge knows $C$ on port 1, so **selectively** forwards frame via port 1
WHAT TO DO ABOUT CYCLES?

- Learning works well in tree topologies
- But trees are fragile
  - Net admins like redundant/backup paths
- How to handle Cycles?
  - Where should B1 forward packets destined for LAN A?

SPANNING TREES

- Spanning tree uses subset of bridges so there are no cycles
  - Prune some ports
  - Only one tree
- Q: How do we find a spanning tree?
  - Automatically!
  - Elect root, find paths
SPANNING TREE PROTOCOL

- Each bridge sends periodic configuration messages
  - (RootID, Distance to Root, BridgeID)
  - All nodes think they are root initially
- Each bridge updates route/Root upon receipt
  - Smaller root address is better
  - Select port with lowest cost to root as “root port”
  - To break ties, bridge with smaller address is better
- Rebroadcast new config to ports for which we’re “best”
  - Don’t bother sending config to LANs with better options
  - Add 1 to distance, send new configs on ports that haven’t told us about a shorter path to the root
  - Only forward packets on ports for which we’re on the shortest path to root (prunes edges to form tree)

SPANNING TREE EXAMPLE

- Sample messages to and from B3:
  1. B3 sends (B3, 0, B3) to B2 and B5
  2. B3 receives (B2, 0, B2) and (B5, 0, B5) and accepts B2 as root
  3. B3 sends (B2, 1, B3) to B5
  4. B3 receives (B1, 1, B2) and (B1, 1, B5) and accepts B1 as root
  5. B3 wants to send (B1, 2, B3) but doesn’t as it’s nowhere “best”
  6. B3 receives (B1, 1, B2) and (B1, 1, B5) again and again...

  Data forwarding is turned off for LAN A
SPANNING TREE DETAILS

• What if root bridge fails?
  • Age configuration info
    • If not refreshed for MaxAge seconds then delete root and recalculate spanning tree
    • If config message is received with a more recent age, then recalculate spanning tree
  • Applies to all bridges (not just root)

• Temporary loops
  • When topology changes, takes a bit for new configuration messages to spread through the system
  • Don’t start forwarding packets immediately -> wait some time for convergence

Outline

1. Switching
2. Routing
3. Forwarding
4. UDP/TCP
ROUTE TASKS

- Forwarding
  - Move packet from input link to the appropriate output link
  - Purely local computation
  - Must go be very fast (executed for every packet)

- Routing
  - Make sure that the next hop actually leads to the destination
  - Global decisions; distributed computation and communication
  - Can go slower (only important when topology changes)

ROUTE TABLES

- The routing table at A, lists – at a minimum – the next hops for the different destinations

<table>
<thead>
<tr>
<th>Dest</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>G</td>
<td>F</td>
</tr>
</tbody>
</table>
BUILDING ROUTING TABLES

• Static
  • Input the right answers and hope it is always true

• Link state
  • Tell everyone what you know about your neighbors

• Distance vector
  • Tell your neighbors what you know about everyone

LINK-STATE ROUTING

• Two phases
  • Reliable flooding
    • Tell all routers what you know about your local topology
  • Path calculation (Dijkstra’s algorithm)
    • Each router computes best path over complete network

• Motivation
  • Global information allows optimal route computation
  • Straightforward to implement and verify
BROADCASTING LINK STATE INFORMATION

- Reliable flooding
  - Each router transmits a Link State Packet (LSP) on all links
  - A neighboring router forwards out all links except incoming
    - Keep a copy locally; don’t forward previously-seen LSPs

DIJKSTRA’S SHORTEST PATH TREE

- So you have all of these LSPs. Now what?
- Graph algorithm for single-source shortest path tree (find best route to all nodes)

```
S ← {}
Q ← <remaining nodes keyed by distance>
While Q !={}  
  u ← extract-min(Q)  \( u = \text{node with lowest cost} \)
  S ← S plus \{u\}  \( \text{is it cheaper to go through } u? \)
  Within Q:
    for each node v adjacent to u
      “relax” the cost of v
```

T=0 T=1

\[ X \rightarrow A \]
\[ C \rightarrow B \rightarrow D \]

T=2 T=3

\[ X \rightarrow A \]
\[ C \rightarrow B \rightarrow D \]
DIJKSTRA STEP 1

- **Green nodes** are “confirmed”
- **Yellow nodes** are “tentative”
- We can add ourselves to “confirmed”

DIJKSTRA STEP 2

- **Green nodes** are “confirmed”
- **Yellow nodes** are “tentative”
- First look at neighbors
- “5” is cheaper than “10”
- We can confirm path with cost “5”
DIJKSTRA STEP 3

- **Green** nodes are “confirmed”
- **Yellow** nodes are “tentative”
- Update costs
- Look at 5’s neighbors
- 7 is cheapest
- We can confirm path with cost 7

DIJKSTRA STEP 4

- **Green** nodes are “confirmed”
- **Yellow** nodes are “tentative”
- Update costs
- 7 has no new neighbors
- 8 is cheapest
- We can confirm 8
DIJKSTRA STEP 5

- Green nodes are “confirmed”
- Yellow nodes are “tentative”
- Update costs
- No new neighbors
- 9 is cheapest
- We can confirm path with cost 9

DIJKSTRA EXAMPLE DONE
BUILDING ROUTING TABLES

- Static
  - Input the right answers and hope it is always true

- Link state
  - Tell everyone what you know about your neighbors

- Distance vector
  - Tell your neighbors what you know about everyone

DISTANCE VECTOR ALGORITHM

- Base assumption
  - Each router knows its own address and the cost to reach each of its directly connected neighbors

- Bellman-Ford algorithm
  - Distributed route computation using only neighbor’s info
BELLMAN-FORD ALGORITHM

- Define distances at each node $X$
  - $d_x(y) = \text{cost of least-cost path from } X \text{ to } Y$
- Update distances based on neighbors
  - $d_x(y) = \min \{c(x,v) + d_v(y)\}$ over all neighbors $V$

DISTANCE VECTOR ALGORITHM

- Iterative, asynchronous: each local iteration caused by:
  - Local link cost change
  - Distance vector update message from neighbor
- Distributed:
  - Each node notifies neighbors when its DV changes
  - Neighbors then notify their neighbors if necessary

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

recompute estimates

if distance to any destination has changed, notify neighbors
STEP-BY-STEP

- \( c(x,v) = \) cost for direct link from \( x \) to \( v \)
  - Node \( x \) maintains costs of direct links \( c(x,v) \)

- \( D_x(y) = \) estimate of least cost from \( x \) to \( y \)
  - Node \( x \) maintains distance vector \( D_x = [D_x(y): y \in N] \)

- Node \( x \) maintains its neighbors’ distance vectors
  - For each neighbor \( v \), \( x \) maintains \( D_v = [D_v(y): y \in N] \)

- Each node \( v \) periodically sends \( D_v \) to its neighbors
  - And neighbors update their own distance vectors
  - \( D_x(y) \leftarrow \min_v [c(x,v) + D_v(y)] \) for each node \( y \in N \)

EXAMPLE: INITIAL STATE

<table>
<thead>
<tr>
<th>Info at node</th>
<th>Distance to Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>
**D SENDS VECTOR TO E**

I'm 2 from C, 0 from D and 2 from E

<table>
<thead>
<tr>
<th>Info at node</th>
<th>Distance to Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0 7 ∞ ∞ 1</td>
</tr>
<tr>
<td>B</td>
<td>7 0 1 ∞ 8</td>
</tr>
<tr>
<td>C</td>
<td>∞ 1 0 2 ∞</td>
</tr>
<tr>
<td>D</td>
<td>∞ ∞ 2 0 2</td>
</tr>
<tr>
<td>E</td>
<td>1 8 4 2 0</td>
</tr>
</tbody>
</table>

D is 2 away, 2+2< ∞, so best path to C is 4

**B SENDS VECTOR TO A**

I'm 7 from A, 0 from B, 1 from C & 8 from E

<table>
<thead>
<tr>
<th>Info at node</th>
<th>Distance to Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0 7 8 ∞ 1</td>
</tr>
<tr>
<td>B</td>
<td>7 0 1 ∞ 8</td>
</tr>
<tr>
<td>C</td>
<td>∞ 1 0 2 ∞</td>
</tr>
<tr>
<td>D</td>
<td>∞ ∞ 2 0 2</td>
</tr>
<tr>
<td>E</td>
<td>1 8 4 2 0</td>
</tr>
</tbody>
</table>

B is 7 away, 1+7< ∞ so best path to C is 8
E SENDS VECTOR TO A

E is 1 away, 4+1<8 so C is 5 away, 1+2<∞ so D is 3 away

I'm 1 from A, 8 from B, 4 from C, 2 from D & 0 from E

...UNTIL CONVERGENCE

<table>
<thead>
<tr>
<th>Info at node</th>
<th>Distance to Node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>C</td>
<td>∞</td>
</tr>
<tr>
<td>D</td>
<td>∞</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Info at node</th>
<th>Distance to Node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>6</td>
</tr>
<tr>
<td>C</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>
ADDRESSING CONSIDERATIONS

- Fixed length or variable length addresses?
- Issues:
  - Flexibility
  - Processing costs
  - Header size
- Engineering choice: IP uses fixed length addresses
- 32-bits in an IPv4 address
  - Dotted decimal format a.b.c.d
  - Each represent 8 bits of address
- Hierarchical: Network part and host part
  - E.g. IP address 128.54.70.238
  - 128.54 refers to the UCSD campus network
  - 70.238 refers to the host ieng6.ucsd.edu
- Which part is network vs. host?
CLASS-BASED ADDRESSING

- Most significant bits determines “class” of address
  
<table>
<thead>
<tr>
<th>Class</th>
<th>Network</th>
<th>Host</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td>127 nets, 16M hosts</td>
</tr>
<tr>
<td>B</td>
<td>1 14</td>
<td>16</td>
<td>16K nets, 64K hosts</td>
</tr>
<tr>
<td>C</td>
<td>1 1 21</td>
<td>8</td>
<td>2M nets, 254 hosts</td>
</tr>
</tbody>
</table>

- Special addresses
  - Class D (1110) for multicast, Class E (1111) experimental
  - 127.0.0.1: local host (a.k.a. the loopback address)
  - Host bits all set to 0: network address
  - Host bits all set to 1: broadcast address

IP FORWARDING TABLES

- Router needs to know where to forward a packet
- Forwarding table contains:
  - List of network names and next hop routers
  - Local networks have entries specifying which interface
    - Link-local hosts can be delivered with Layer-2 forwarding
  - E.g. www.ucsd.edu address is 132.239.180.101
  - Class B address – class + network is 132.239
  - Lookup 132.239 in forwarding table
  - Prefix – part of address that really matters for routing
**SUBNETTING**

- Individual networks may be composed of several LANs
  - Only want traffic destined to local hosts on physical network
  - Routers need a way to know which hosts on which LAN

- Networks can be arbitrarily decomposed into **subnets**
  - Each subnet is simply a prefix of the host address portion
  - Subnet prefix can be of any length, specified with **netmask**

**SUBNET ADDRESSES**

- Every (sub)network has an address and a netmask
  - Netmask tells which bits of the network address is important
  - Convention suggests it be a proper prefix

- Netmask written as an all-ones IP address
  - E.g., Class B netmask is 255.255.0.0
  - Sometimes expressed in terms of number of 1s, e.g., /16

- Need to size subnet appropriately for each LAN
  - Only have remaining bits to specify host addresses
IP ADDRESS PROBLEM (1991)

• Address space depletion
  • In danger of running out of classes A and B

• Why?
  • Class C too small for most organizations (only ~250 addresses)
  • Very few class A – very careful about giving them out (who has 16M hosts anyway?)
  • Class B – greatest problem
CIDR

• Classless Inter-Domain Routing (1993)
  • Networks described by variable-length prefix and length
  • Allows arbitrary allocation between network and host address

<table>
<thead>
<tr>
<th>Network</th>
<th>Host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>Mask=# significant bits representing prefix</td>
</tr>
</tbody>
</table>

• e.g. 10.95.1.2 contained within 10.0.0.0/8:
  • 10.0.0.0 is network and remainder (95.1.2) is host

• Pro: Finer grained allocation; aggregation
• Con: More expensive lookup: longest prefix match

FORWARDING TABLE EXAMPLE

• Packet to 10.1.1.6
• Matches 10.1.0.0/23

<table>
<thead>
<tr>
<th>Destination</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td>loopback</td>
</tr>
<tr>
<td>Default or 0/0</td>
<td>10.1.16.1</td>
</tr>
<tr>
<td>10.1.8.0/24</td>
<td>interface1</td>
</tr>
<tr>
<td>10.1.2.0/23</td>
<td>interface2</td>
</tr>
<tr>
<td><strong>10.1.0.0/23</strong></td>
<td>10.1.2.2</td>
</tr>
<tr>
<td>10.1.16.0/24</td>
<td>interface3</td>
</tr>
</tbody>
</table>
FORWARDING TABLE EXAMPLE 2

- Packet to 10.1.1.6
- Matches 10.1.1.4/30
  - Longest prefix match

Routing table at R1

<table>
<thead>
<tr>
<th>Destination</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td>loopback</td>
</tr>
<tr>
<td>Default</td>
<td>10.1.2.1</td>
</tr>
<tr>
<td>10.1.0.0/24</td>
<td>interface1</td>
</tr>
<tr>
<td>10.1.1.0/24</td>
<td>interface2</td>
</tr>
<tr>
<td>10.1.2.0/23</td>
<td>interface3</td>
</tr>
<tr>
<td>10.1.1.4/30</td>
<td>10.1.1.101</td>
</tr>
</tbody>
</table>

FORWARDING TABLE EXAMPLE 3

- Packet to 10.1.1.6
- Direct route
  - Longest prefix match

Routing table at H1

<table>
<thead>
<tr>
<th>Destination</th>
<th>Next Hop</th>
</tr>
</thead>
<tbody>
<tr>
<td>127.0.0.1</td>
<td>loopback</td>
</tr>
<tr>
<td>Default</td>
<td>10.1.1.1</td>
</tr>
<tr>
<td>10.1.1.0/24</td>
<td>interface1</td>
</tr>
<tr>
<td>10.1.1.4/30</td>
<td>interface2</td>
</tr>
</tbody>
</table>
Outline

1. Switching
2. Routing
3. Forwarding
4. UDP/TCP

NAMING PROCESSES/SERVICES

- Process here is an abstract term for your Web browser (HTTP), Email servers (SMTP), hostname translation (DNS)

- How do we identify for remote communication?
  - Process id or memory address are OS-specific and transient

- So TCP and UDP use **ports**
  - 16-bit integers representing mailboxes that processes “rent”
  - Identify process uniquely as (IP address, protocol, port)
PICKING PORT NUMBERS

- We still have the problem of allocating port numbers
  - What port should a Web server use on host X?
  - To what port should you send to contact that Web server?

- Servers typically bind to well-known port numbers
  - e.g., HTTP 80, SMTP 25, DNS 53, ... look in /etc/services
  - Ports below 1024 traditionally reserved for well-known services

- Clients use OS-assigned temporary (ephemeral) ports
  - Above 1024, recycled by OS when client finished

USER DATAGRAM PROTOCOL (UDP)

- Provides unreliable message delivery between processes
  - Source port filled in by OS as message is sent
  - Destination port identifies UDP delivery queue at endpoint
- Connectionless (no state about who talks to whom)
UDP DELIVERY

UDP CHECKSUM

- UDP includes optional protection against errors
  - Checksum intended as an end-to-end check on delivery
  - So it covers data, UDP header, and IP pseudoheader (history)
TRANSMISSION CONTROL PROTOCOL

- Reliable bi-directional byte stream between processes
  - Uses a sliding window protocol for efficient transfer
- Connection-oriented
  - Conversation between two endpoints with beginning and end
- Flow control (last lecture)
  - Prevents sender from over-running receiver buffers
  - (tell sender how much buffer is left at receiver)
- Congestion control (later in term)
  - Prevents sender from over-running network capacity

TCP DELIVERY

![TCP Delivery Diagram]

Application process

TCP Send buffer

Write bytes

Transmit segments

Segment | Segment | ... | Segment

TCP Receive buffer

Application process

TCP

Read bytes
TCP HEADER FORMAT

• Ports plus IP addresses identify a connection
  • 4-tuple

TCP HEADER FORMAT

• Sequence, Ack numbers used for the sliding window
• How big a window? Flow control/congestion control determine
TCP HEADER FORMAT

- Flags may be ACK, SYN, FIN, URG, PSH, RST

THREE-WAY HANDSHAKE

- Opens both directions for transfer
TCP STATE TRANSITIONS

AGAIN, WITH STATES