Announcements

- **Homework 2:**
  - Posted today
  - Due next Wednesday, Oct 1
- **Project 1:**
  - Basic test scripts will be posted on-line today
- **Class news group?**
  - Replaced by mailing list
  - We’ll change the sender address to ee122 so you can easily filter
  - Can use gmail (or other e-mail clients) to search for relevant info
Names & Addresses

- Names
  - Human readable
  - No location semantics
  - E.g., “yahoo.com”, “sky.cs.berkeley.edu”

- Addresses
  - Easy to manipulate in software (usually fixed length)
  - Location semantics
  - E.g., 206.190.60.37, 128.32.37.169.229

- But sometimes not clear cut…

Names & Addresses

- What is “Leonardo da Vinci”?

- What is “Seven-of-Nine”?

- Depends on the context
  - An address in one context can become a name in another context
Hop-by-Hop Packet Forwarding

- Each router has a *forwarding table*
  - Maps destination addresses…
  - … to outgoing *interfaces* (= links)
- Upon receiving a packet
  - Inspect the destination IP address in the header
  - Index into the table
  - Find the *longest* prefix match
  - Forward packet out interface associated with match
- Where does forwarding table come from?
  - *Routing* algorithms (or static configs)

Longest-Prefix-Match Forwarding

<table>
<thead>
<tr>
<th>destination</th>
<th>prefix</th>
<th>outgoing link</th>
</tr>
</thead>
<tbody>
<tr>
<td>201.10.7.17</td>
<td>192.0.0.0/4</td>
<td>2</td>
</tr>
<tr>
<td>201 10 7 17</td>
<td>4.83.128.0/17</td>
<td>1</td>
</tr>
<tr>
<td>201.10.0.0/21</td>
<td>201.10.0.0/21</td>
<td>3</td>
</tr>
<tr>
<td>201.10.6.0/23</td>
<td>201.10.6.0/23</td>
<td>2</td>
</tr>
<tr>
<td>201.10.0.0/24</td>
<td>126.255.103.0/24</td>
<td>3</td>
</tr>
</tbody>
</table>
### Longest-Prefix-Match Forwarding

**Forwarding Table**

<table>
<thead>
<tr>
<th>prefix</th>
<th>outgoing link</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.0.0.0/4</td>
<td>2</td>
</tr>
<tr>
<td>4.8.128.0/17</td>
<td>1</td>
</tr>
<tr>
<td>201.10.0.0/21</td>
<td>3</td>
</tr>
<tr>
<td>201.10.6.0/23</td>
<td>2</td>
</tr>
<tr>
<td>126.255.103.0/24</td>
<td>3</td>
</tr>
</tbody>
</table>

### Destination: 201.10.7.17

- **201**: 11001001
- **10**: 00001010
- **7**: 00000111
- **17**: 00010001

**Prefixes:**

- **192**: 11000000

### Longest-Prefix-Match Forwarding

**Forwarding Table**

<table>
<thead>
<tr>
<th>prefix</th>
<th>outgoing link</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.0.0.0/4</td>
<td>2</td>
</tr>
<tr>
<td>4.83.128.0/17</td>
<td>1</td>
</tr>
<tr>
<td>201.10.0.0/21</td>
<td>3</td>
</tr>
<tr>
<td>201.10.6.0/23</td>
<td>2</td>
</tr>
<tr>
<td>126.255.103.0/24</td>
<td>3</td>
</tr>
</tbody>
</table>

### Destination: 201.10.7.17

- **201**: 11001001
- **10**: 00001010
- **7**: 00000111
- **17**: 00010001

**Prefixes:**

- **4**: 00000100
- **83**: 01010011
- **128**: 10000000
### Longest-Prefix-Match Forwarding

**Forwarding Table**

<table>
<thead>
<tr>
<th>destination</th>
<th>prefix</th>
<th>outgoing link</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>201.10.7.17</strong></td>
<td>192.0.0.0/4</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>4.83.128.0/17</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>201.10.0.0/21</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>201.10.6.0/23</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>126.255.103.0/24</td>
<td>3</td>
</tr>
</tbody>
</table>

**Prefixes:**

- **201**: 11001001
- **10**: 00001010
- **7**: 00000111
- **17**: 00010001
Longest-Prefix-Match Forwarding

Algorithmic problem: how do we do this fast?

Simple Algorithms Are Too Slow

- Scan the forwarding table one entry at a time
  - See if the destination matches the entry
  - If so, check the size of the mask for the prefix
  - Keep track of the entry with longest-matching prefix

- Overhead is linear in size of the forwarding table
  - Today, that means 200,000-250,000 entries!
  - And, the router may have just a few nanoseconds
  - ... before the next packet arrives

- Need greater efficiency to keep up with line speed
  - Better algorithms
  - Hardware implementations
Patricia Tree

- Store the prefixes as a tree
  - One bit for each level of the tree
  - Some nodes correspond to valid prefixes (w/ next-hop interfaces)
- When a packet arrives
  - Traverse the tree based on the destination address
  - Stop upon reaching the longest matching prefix
  - Running time: scales with # bits in address (but takes more memory)
  - Lot of work on still-faster algorithms

```
0 1
0 0 1
00*
0*
11*
0
1
1
```

How Does Sending End Host Forward?

- No need to run a routing protocol
  - Packets to the host itself (e.g., 1.2.3.4/32)
    - Delivered locally
  - Packets to other hosts on the LAN (e.g., 1.2.3.0/25)
    - Sent out the interface with LAN address
    - Can tell they’re local using subnet mask (e.g., 255.255.255.128)
  - Packets to external hosts (any others)
    - Sent out interface to local gateway
    - I.e., IP router on the LAN
- How this information is learned
  - Static setting of address, subnet mask, and gateway
  - Or: Dynamic Host Configuration Protocol (DHCP)
What About Reaching the End Hosts?

- How does the last router reach the destination?

- Each interface has a persistent, global identifier:
  - **MAC address** (Media Access Control) - **Layer 2**
  - Programmed into Network Interface Card (NIC)
  - Usually **flat** address structure (i.e., no hierarchy)

- Constructing an **address resolution** table:
  - Mapping MAC address to/from IP address
  - Address Resolution Protocol (ARP)

Transport Layer
Transport Protocols

- Provide *logical communication* between application processes running on different hosts
- Run on end hosts
  - Sender: breaks application messages into *segments*, and passes to network layer
  - Receiver: reassembles segments into messages, passes to application layer
- Multiple transport protocol available to applications
  - Internet: TCP and UDP (mainly)

Internet Transport Protocols

- Datagram messaging service (UDP)
  - No-frills extension of "best-effort" IP
  - Multiplexing/demultiplexing among processes
- Reliable, in-order delivery (TCP)
  - Connection set-up & tear-down
  - Discarding of corrupted packets
  - Retransmission of lost packets
  - Flow control
  - Congestion control
- Services not available
  - Delay guarantees
  - Bandwidth guarantees
  - Sessions that survive change-of-IP-address
<table>
<thead>
<tr>
<th>4-bit Version</th>
<th>4-bit Header Length</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
<td>13-bit Fragment Offset</td>
<td>8-bit Time to Live (TTL)</td>
</tr>
<tr>
<td>8-bit Protocol</td>
<td>16-bit Header Checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td>Options (if any)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Payload
<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
</tr>
<tr>
<td>5</td>
<td>16-bit Total Length (Bytes)</td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
<td></td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Payload

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
</tr>
<tr>
<td>5</td>
<td>16-bit Total Length (Bytes)</td>
<td></td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>3-bit Flags</td>
<td></td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
<td></td>
</tr>
<tr>
<td>6 = TCP 17 = UDP</td>
<td>16-bit Header Checksum</td>
<td></td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Payload
Multiplexing and Demultiplexing

- Host receives IP datagrams
  - Each datagram has source and destination IP address,
  - Each datagram carries one transport-layer segment
  - Each segment has source and destination port number

TCP/UDP segment format

32 bits

source port # | dest port #
other header fields
application data (message)

Payload
Unreliable Message Delivery Service

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket
- User Datagram Protocol (UDP; RFC 768 - 1980!)
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents
    - (checksum field = 0 means “don’t verify checksum”)

```
<table>
<thead>
<tr>
<th>SRC port</th>
<th>DST port</th>
</tr>
</thead>
<tbody>
<tr>
<td>checksum</td>
<td>length</td>
</tr>
</tbody>
</table>
```

Ports

- Need to decide which application gets which packets
- Solution: map each socket to a port
- Client must know server’s port
- Separate 16-bit port address space for UDP and TCP
  - (src_IP, src_port, dst_IP, dst_port) uniquely identifies TCP connection
- **Well known ports** (0-1023): everyone agrees which services run on these ports
  - e.g., ssh:22, http:80
  - On UNIX, must be root to gain access to these ports (why?)
- Ephemeral ports (most 1024-65535): given to clients
  - e.g. chat clients, p2p networks
Why Would Anyone Use UDP?

- Finer control over what data is sent and when
  - As soon as an application process writes into the socket
  - … UDP will package the data and send the packet
- No delay for connection establishment
  - UDP just blasts away without any formal preliminaries
  - … which avoids introducing any unnecessary delays
- No connection state
  - No allocation of buffers, sequence #s, timers …
  - … making it easier to handle many active clients at once
- Small packet header overhead
  - UDP header is only 8 bytes

Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting lost/corrupted packets often pointless by the time the packet is retransmitted, it’s too late
  - E.g., telephone calls, video conferencing, gaming
- Simple query protocols like Domain Name System
  - Connection establishment overhead would double cost
  - Easier to have application retransmit if needed

“Address for bbc.co.uk?”

“212.58.224.131”
5 Minute Break

Questions Before We Proceed?

Transmission Control Protocol (TCP)

- Connection oriented
  - Explicit set-up and tear-down of TCP session
- Stream-of-bytes service
  - Sends and receives a stream of bytes, not messages
- Congestion control
  - Dynamic adaptation to network path’s capacity
- Reliable, in-order delivery
  - TCP tries very hard to ensure byte stream (eventually) arrives intact
    - In the presence of corruption and loss
- Flow control
  - Ensure that sender doesn’t overwhelm receiver
Reliable Delivery

- How do we design for reliable delivery?
  - One possible model: how does it work talking on your cell phone?
- **Positive** acknowledgment (“Ack”)
  - Explicit confirmation by receiver
  - TCP acknowledgments are cumulative (“I’ve received everything up through sequence #N”)
- **Negative** acknowledgment (“Nack”)
  - “I’m missing the following: …”
  - How might the receiver tell something’s missing? Can they always do this?
  - (Only used by TCP in implicit fashion - “fast retransmit”)

Reliable Delivery (con’t)

- **Timeout**
  - If haven’t heard anything from receiver, send again
  - Problem: for how long do you wait?
    - TCP uses function of estimated RTT
  - Problem: what if no Ack for retransmission?
    - TCP (and other schemes) employs exponential backoff
    - Double timer up to maximum - tapers off load during congestion
- A very different approach to reliability: send redundant data
  - Cell phone analogy: “Meet me at 3PM - repeat 3PM”
  - **Forward error correction**
  - Recovers from lost data nearly immediately!
  - But: only can cope with a limited degree of loss
  - And: adds load to the network
TCP Support for Reliable Delivery

- Sequence numbers
  - Used to detect *missing* data
  - ... and for putting the data back in order
- Checksum
  - Used to detect *corrupted* data at the receiver
  - ...leading the receiver to drop the packet
  - No error signal sent - recovery via normal retransmission
- Retransmission
  - Sender retransmits lost or corrupted data
  - Timeout based on estimates of round-trip time (RTT)

Efficient Transport Reliability
Automatic Repeat reQuest (ARQ)

- Automatic Repeat Request
  - Receiver sends acknowledgment (ACK) when it receives packet
  - Sender waits for ACK and *times out* if does not arrive within some time period
- Simplest ARQ protocol
  - Stop and Wait
    - Send a packet, stop and wait until ACK arrives

How Fast Can Stop-and-Wait Go?

- Suppose we're sending from UCB to New York:
  - Bandwidth = 1 Mbps (megabits/sec)
  - RTT = 100 msec
  - Maximum Transmission Unit (MTU) = 1500 B = 12,000 b
  - No other load on the path and no packet loss
- What (approximately) is the fastest we can transmit using Stop-and-Wait?
- How about if Bandwidth = 1 Gbps?
Allowing Multiple Packets in Flight

- “In Flight” = “Unacknowledged”
- Sender-side issue: how many packets (bytes)?
- Receiver-side issue: how much buffer for data that’s “above a sequence hole”?
  - I.e., data that can’t be delivered since previous data is missing
  - Assumes service model is in-order delivery (like TCP)

Sliding Window

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver
  - … though not too far ahead
**Sliding Window (con’t)**

- Both sender & receiver maintain a window that governs amount of data in flight (sender) or not-yet-delivered (receiver)
- **Left edge** of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of undelivered data
- For the sender:
  - Window size = maximum amount of data in flight
    - Determines rate
    - Sender must have at least this much buffer (maybe more)
- For the receiver:
  - Window size = maximum amount of undelivered data
    - Receiver has this much buffer

---

**Sliding Window**

- For the sender, when receives an acknowledgment for new data, window advances (*slides* forward)

![Diagram](https://via.placeholder.com/150)

- **Sending process**
- **TCP**
- **Last byte written**
- **Sender Window**
- **Last byte ACKed**
- **Last byte can send**
**Sliding Window**

- For the sender, when receives an acknowledgment for new data, window advances (*slides* forward)

  ![Sending process](image)

  *Sender Window*

  TCP
  Last byte written

  Last byte ACKed

- For the receiver, as the receiving process consumes data, the window slides forward

  ![Receiving process](image)

  *Receiver Window*

  TCP
  Last byte read

  Next byte needed

  Last byte received
**Sliding Window**

- For the receiver, as the receiving process consumes data, the window slides forward.

![Sliding Window Diagram]

**Sliding Window (con’t)**

- Sender: window **advances** when new data ack’d.
- Receiver: window advances as receiving process consumes data.
- What happens if sender’s window size exceeds the receiver’s window size?
- Receiver **advertises** to the sender where the receiver window currently ends (“righthand edge”).
  - Sender agrees not to exceed this amount.
  - It makes sure by setting its own window size to a value that can’t send beyond the receiver’s righthand edge.
Performance with Sliding Window

- Given previous UCB ↔ New York 1 Mbps path with 100 msec RTT
  and Sender (and Receiver) window = 100 Kb = 12.5 KB
  - How fast can we transmit?
  - What about with 12.5 KB window & 1 Gbps path?
  - Window required to fully utilize path:
    - Bandwidth-delay product (or "delay-bandwidth product")
    - 1 Gbps * 100 msec = 100 Mb = 12.5 MB
    - Note: large window = many packets in flight

Summary

- IP packet forwarding
  - Based on longest-prefix match
  - End systems use subnet mask to determine if traffic destined for their LAN …
    - In which case they send directly, using ARP to find MAC address
  - … or for some other network
    - In which case they send to their local gateway (router)
  - This info either statically config'd or learned via DHCP
- Transport protocols
  - Multiplexing and demultiplexing via port numbers
  - UDP gives simple datagram service
  - TCP gives reliable byte-stream service
  - Reliability immediately raises performance issues
    - Stop-and-Wait vs. Sliding Window
Next Lecture

- DNS = Domain Name System (Brighten)
- Reading: K&R 2.5
- Homework #2 out: due October 1
- Project 1, 1st part: test scripts will be available today