Socket programming

Two socket types, depending on transport services
- UDP: unreliable datagram
- TCP: reliable, byte-stream oriented

Application at end host distinguished by binding socket to a port number
- 16 bit unsigned number; 0-1023 are bound to well-known applications
  - web server = 80; mail = 25; telnet = 23
Socket Programming with UDP

- No connection between client and server
  - no handshaking before sending data
  - Sender: explicitly attaches destination IP address and port number to each packet
  - Receiver: extracts sender IP address and port number from received packet

- **Best effort:** Data may be lost or received out-of-order

- UDP provides applications with unreliable transfer of a group of bytes ("datagram") between client and server
Connectionless Demux

- Distinct UDP segments with same dest IP address and port, go to the same socket
- even if they come from different source IP!
- The application must sort things out!
UDP: Perspective

- **Speed**
  - no connection establishment (takes time)
  - no congestion control: UDP can blast away!

- **Simplicity**
  - no connection state at sender/receiver

- **Extra work for applications**
  - reordering, duplicate suppression, missing packets...
  - but some applications may not care!
    - streaming multimedia: loss tolerant, rate sensitive (want constant, fast speeds)
Transmission Control Protocol (TCP)

- Reliable, ordered communication
- Adaptive protocol that delivers good-enough performance and handles congestion well
- All Web traffic travels over TCP/IP
  - Enough applications demand reliable ordered delivery that they should not have to implement their own protocol
  - ...but not really end-to-end (just socket to socket)
Socket Programming with TCP

Client
- Creates TCP socket with server’s IP address and port number
- Client TCP establishes connection to server TCP

Server
- Contacted by client
- Already running
- Already created a “welcoming socket”
- When contacted by client, creates a new TCP socket to communicate just with that client
  - Socket identified by 4-tuple
    - source IP; source port no; dest. IP; dest port no.
- Server can concurrently serve multiple clients
Host receives three TCP segments
- all destined to IP address B, port 80
- demuxed to different sockets through socket's 4-tuple
TCP Connections

- Initiated by a three-way handshake
- 1.5 RTTs
- Create shared state on both sides of connection
  - Both sides know first sequence number to be used
  - Both sides know other side is ready to receive
Typical TCP Usage

- Three round trips to
  - set up a connection
  - send a data packet
  - receive a response
  - tear down connection

- FINs tear down connection
  - Can be piggybacked on Ack
TCP Segments

Each segment carries SEQ, a unique sequence number

- initial value of SEQ chosen randomly
  - SEQ incremented by the data length
    - for simplicity, 4410 slides assume payloads of size 1

Each segment carries an acknowledgement

- acknowledge a set of packets by acking latest received SEQ
- the acknowledgment is the sequence number of the next expected packet!
Reliable Transport

- TCP at sender keeps a copy of all sent, but unacknowledged, packets
- Packet resent if ACK does not arrive within a timeout
- Timeout interval adjusts to round-trip delay

\[
\text{AverageRTT} = (1 - \alpha) \text{OldAverageRTT} + \alpha \text{LatestRTT}
\]
\[
\text{AverageVar} = (1 - \beta) \text{OldAverageVar} + \beta \text{LatestVar}
\]

where \( \text{LatestRTT} = (\text{ack\_receive\_time} - \text{send\_time}) \), \( \text{LatestVar} = |\text{LatestRTT} - \text{AverageRTT}| \), \( \alpha = 1/8 \), \( \beta = 1/4 \) typically.

\[
\text{Timeout} = \text{AverageRTT} + 4 \times \text{AverageVar}
\]

Here is joke about TCP. Did you get it? Did you get it? Did you get it?
How long does it take to send a segment?

Let \( L \): one-way latency (sec)
\( b \): bandwidth (bytes/sec)
\( S \): Size of segment (bytes)

- Time between start sending and end receiving
  \( L + \frac{S}{b} \) sec. (ignoring headers)
- Time before ack is received by sender: \( L \) sec
  - assuming acks are small
- End-to-end throughput
  \( \frac{S}{2L + \frac{S}{b}} \) bytes/sec [goes to 0 as \( L \) grows]
Pipelining

- Sender allows multiple, “in flight”, yet-to-be-acknowledged packets (a “window”)
  - Increases throughput
  - Needs buffering at sender and receiver

How large should the window be?

What if a packet in the middle is missing?
How Much Data “Fits” in a Pipe?

Suppose

- bandwidth is $b$ bytes/sec
- RTT is $r$ seconds
- ACK is a small message

Then, can send $b \cdot r$ bytes before receiving ack for first byte...

- of course, $b$ and $r$ can change over time...
TCP Window, Size 4

When first item in window is acknowledged, sender can send the 5th item.
TCP Fast Retransmit

DATA, seq=17
DATA, seq=18
DATA, seq=19
DATA, seq=20
ack=18

DATA, seq=18
DATA, seq=18
ack=21

ack=18
ack=18
ACK=18

ack=21
TCP Congestion Control

- **Additive Increase/Multiplicative Decrease (AIMD)**
  - window_size++ every RTT if no packet dropped
  - window_size/2 if packet is dropped
    - drop detected by acknowledgments

- Slowly builds to max bandwidth, and hovers there
  - Does not achieve maximum bandwidth
  + Shares bandwidth well with other TCP connections

- Policy of linear increase, exponential backoff under congestion known as TCP friendliness
TCP Window Size

Linear Increase
Exponential Backoff

Assuming losses in the network only due to bandwidth

Window Size:
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
5, 6, 7, 8, 9, 10
5, 6, 7, 8, 9, 10
...

Time
Max Bandwidth
Bandwidth
TCP Slow Start

- Linear Increase
  - Most file transactions end before that happens...
  - It takes long to reach window size that matches $b \cdot r$

- Exponential Increase
  - TCP builds large window quickly by doubling window size for each ack received until first loss
TCP Window Size with Exponential Start
TCP Fairness

- If $k$ TCP sessions share the same bottleneck link of bandwidth $R$, each should have rate $R/k$.

- Is AIMD fair?

Diagram:
- Equal Bandwidth Share
- Start at an arbitrary point
- Bandwidth grows equally for both connections: 45° line
- 1. Additive Increase
TCP Fairness

If $k$ TCP sessions share the same bottleneck link of bandwidth $R$, each should have rate $R/k$.

Is AIMD fair?

- Start at an arbitrary point.
- Bandwidth halves: half the distance between this point and origin.
- Multiplicative Decrease.
TCP Fairness

If $k$ TCP sessions share the same bottleneck link of bandwidth $R$, each should have rate $R/k$

IS AIMD fair?

Equal Bandwidth Share

Bandwidth grows equally for both connections: $45^\circ$ line

Start at an arbitrary point

1. Additive Increase
TCP Fairness

- If $k$ TCP sessions share the same bottleneck link of bandwidth $R$, each should have rate $R/k$.
- IS AIMD fair? Converges around equal bandwidth.

Diagram:
- Equal Bandwidth Share
- Start at an arbitrary point.
- Bandwidth halves: half the distance between this point and origin.
- 2. Multiplicative Decrease
TCP Summary

- Reliable ordered message delivery
  - Connection oriented, 3-way handshake
- Transmission window for better throughput
  - Timeouts based on link parameters
- Congestion control
  - Linear increase, exponential backoff
- Fast adaptation
  - Exponential increase in the initial phase