Where we are in the Course

• Starting the Transport Layer!
  – Builds on the network layer to deliver data across networks for applications with the desired reliability or quality

```
Application
Transport
Network
Link
Physical
```
Recall

- Transport layer provides end-to-end connectivity across the network
Recall (2)

- Segments carry application data across the network
- Segments are carried within packets within frames
Transport Layer Services

- Provide different kinds of data delivery across the network to applications

<table>
<thead>
<tr>
<th></th>
<th>Unreliable</th>
<th>Reliable</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Messages</strong></td>
<td>Datagrams (UDP)</td>
<td>Streams (TCP)</td>
</tr>
<tr>
<td><strong>Bytestream</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Comparison of Internet Transports

- TCP is full-featured, UDP is a glorified packet

<table>
<thead>
<tr>
<th>TCP (Streams)</th>
<th>UDP (Datagrams)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connections</td>
<td>Datagrams</td>
</tr>
<tr>
<td>Bytes are delivered once,</td>
<td>Messages may be lost, reordered, duplicated</td>
</tr>
<tr>
<td>reliably, and in order</td>
<td></td>
</tr>
<tr>
<td>Arbitrary length content</td>
<td>Limited message size</td>
</tr>
<tr>
<td>Flow control matches sender to</td>
<td>Can send regardless of receiver state</td>
</tr>
<tr>
<td>receiver</td>
<td></td>
</tr>
<tr>
<td>Congestion control matches</td>
<td>Can send regardless of network state</td>
</tr>
<tr>
<td>sender to network</td>
<td></td>
</tr>
</tbody>
</table>
Ports

• Application process is identified by the tuple IP address, protocol, and port
  – Ports are 16-bit integers representing local “mailboxes” that a process leases

• Servers often bind to “well-known ports”
  – <1024, require administrative privileges

• Clients often assigned “ephemeral” ports
  – Chosen by OS, used temporarily
## Some Well-Known Ports

<table>
<thead>
<tr>
<th>Port</th>
<th>Protocol</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>20, 21</td>
<td>FTP</td>
<td>File transfer</td>
</tr>
<tr>
<td>22</td>
<td>SSH</td>
<td>Remote login, replacement for Telnet</td>
</tr>
<tr>
<td>25</td>
<td>SMTP</td>
<td>Email</td>
</tr>
<tr>
<td>80</td>
<td>HTTP</td>
<td>World Wide Web</td>
</tr>
<tr>
<td>110</td>
<td>POP-3</td>
<td>Remote email access</td>
</tr>
<tr>
<td>143</td>
<td>IMAP</td>
<td>Remote email access</td>
</tr>
<tr>
<td>443</td>
<td>HTTPS</td>
<td>Secure Web (HTTP over SSL/TLS)</td>
</tr>
<tr>
<td>543</td>
<td>RTSP</td>
<td>Media player control</td>
</tr>
<tr>
<td>631</td>
<td>IPP</td>
<td>Printer sharing</td>
</tr>
</tbody>
</table>
Topic

- Sending messages with UDP
  - A shim layer on packets

I just want to send a packet!
User Datagram Protocol (UDP)

- Used by apps that don’t want reliability or bytestreams
  - Voice-over-IP (unreliable)
  - DNS, RPC (message-oriented)
  - DHCP (bootstrapping)

(If application wants reliability and messages then it has work to do!)
UDP Buffering

Application

Ports

Transport (TCP)

Message queues

Port Mux/Demux

Network (IP)

packet
Topic

- How to set up connections
  - We’ll see how TCP does it
Connection Establishment

- Both sender and receiver must be ready before we start the transfer of data
  - Need to agree on a set of parameters
  - e.g., the Maximum Segment Size (MSS)

- This is signaling
  - It sets up state at the endpoints
  - Like “dialing” for a telephone call
Three-Way Handshake

- Used in TCP; opens connection for data in both directions
- Each side probes the other with a fresh Initial Sequence Number (ISN)
  - Sends on a SYNchronize segment
  - Echo on an ACKnowledge segment
- Chosen to be robust even against delayed duplicates
Three-Way Handshake (2)

- Three steps:
  - Client sends SYN(x)
  - Server replies with SYN(y)ACK(x+1)
  - Client replies with ACK(y+1)
  - SYNs are retransmitted if lost

- Sequence and ack numbers carried on further segments
Connection Release

• Orderly release by both parties when done
  – Delivers all pending data and “hangs up”
  – Cleans up state in sender and receiver

• Key problem is to provide reliability while releasing
  – TCP uses a “symmetric” close in which both sides shutdown independently
TCP Connection Release

• Two steps:
  – Active sends FIN(x), passive ACKs
  – Passive sends FIN(y), active ACKs
  – FINs are retransmitted if lost

• Each FIN/ACK closes one direction of data transfer
TCP Connection Release (2)

- **Two steps:**
  - Active sends FIN(x), passive ACKs
  - Passive sends FIN(y), active ACKs
  - FINs are retransmitted if lost

- **Each FIN/ACK closes one direction of data transfer**
Topic

• The sliding window algorithm
  – Pipelining and reliability
  – Building on Stop-and-Wait

Yeah!
Recall

- ARQ with one message at a time is Stop-and-Wait (normal case below)
Limitation of Stop-and-Wait

- It allows only a single message to be outstanding from the sender:
  - Fine for LAN (only one frame fits)
  - Not efficient for network paths with $BD >> 1$ packet
Limitation of Stop-and-Wait (2)

- Example: R=1 Mbps, D = 50 ms
  - RTT (Round Trip Time) = 2D = 100 ms
  - How many packets/sec?

- What if R=10 Mbps?
Sliding Window

- Generalization of stop-and-wait
  - Allows $W$ packets to be outstanding
  - Can send $W$ packets per RTT ($=2D$)
  - Pipelining improves performance
  - Need $W=2BD$ to fill network path
Sliding Window (2)

• What $W$ will use the network capacity?
• Ex: $R=1$ Mbps, $D = 50$ ms

• Ex: What if $R=10$ Mbps?
Sliding Window (3)

- Ex: R=1 Mbps, D = 50 ms
  - $2BD = 10^6 \text{ b/sec} \times 100 \times 10^{-3} \text{ sec} = 100 \text{ kbit}$
  - $W = 2BD = 10$ packets of 1250 bytes

- Ex: What if R=10 Mbps?
  - $2BD = 1000 \text{ kbit}$
  - $W = 2BD = 100$ packets of 1250 bytes
Sliding Window Protocol

- Many variations, depending on how buffers, acknowledgements, and retransmissions are handled

- Go-Back-N
  - Simplest version, can be inefficient

- Selective Repeat
  - More complex, better performance
Sliding Window – Sender

- Sender buffers up to $W$ segments until they are acknowledged
  - $\text{LFS} = \text{LAST FRAME SENT}$, $\text{LAR} = \text{LAST ACK REC'D}$
  - Sends while $\text{LFS} - \text{LAR} \leq W$

```
.. A cked  Unacked  Unavailable ..
```

- Sliding Window
- $W=5$
- Available
- LAR
- LFS
- seq. number
Sliding Window – Sender (2)

- Transport accepts another segment of data from the Application ...
  - Transport sends it (as LFS–LAR \( \rightarrow \) 5)

```
.. A cked  Unacked  Unavailable  .
L AR
L FS
seq. number
```

\( W = 5 \)
Sliding Window – Sender (3)

- Next higher ACK arrives from peer...
  - Window advances, buffer is freed
  - LFS–LAR → 4 (can send one more)
Sliding Window – Go-Back-N

• Receiver keeps only a single packet buffer for the next segment
  – State variable, LAS = LAST ACK SENT

• On receive:
  – If seq. number is LAS+1, accept and pass it to app, update LAS, send ACK
  – Otherwise discard (as out of order)
Sliding Window – Selective Repeat

• Receiver passes data to app in order, and buffers out-of-order segments to reduce retransmissions

• ACK conveys highest in-order segment, plus hints about out-of-order segments

• TCP uses a selective repeat design; we’ll see the details later
Sliding Window – Selective Repeat (2)

- Buffers W segments, keeps state variable, LAS = LAST ACK SENT

- On receive:
  - Buffer segments [LAS+1, LAS+W]
  - Pass up to app in-order segments from LAS+1, and update LAS
  - Send ACK for LAS regardless
Sliding Window – Retransmissions

- Go-Back-N sender uses a single timer to detect losses
  - On timeout, resends buffered packets starting at LAR+1

- Selective Repeat sender uses a timer per unacked segment to detect losses
  - On timeout for segment, resend it
  - Hope to resend fewer segments
Sequence Numbers

- Need more than 0/1 for Stop-and-Wait ...
  - But how many?

- For Selective Repeat, need W numbers for packets, plus W for acks of earlier packets
  - 2W seq. numbers
  - Fewer for Go-Back-N (W+1)

- Typically implement seq. number with an N-bit counter that wraps around at $2^N - 1$
  - E.g., N=8: ..., 253, 254, 255, 0, 1, 2, 3, ...
Sequence Time Plot

Transmissions (at Sender)

Acks (at Receiver)

Delay (=RTT/2)
Sequence Time Plot (2)

Go-Back-N scenario
Sequence Time Plot (3)
• Adding flow control to the sliding window algorithm
  – To slow the over-enthusiastic sender
Problem

- Sliding window uses pipelining to keep the network busy
  - What if the receiver is overloaded?

Big Iron  Streaming video  Wee Mobile
Sliding Window – Receiver

- Consider receiver with W buffers
  - LAS = LAST ACK SENT, app pulls in-order data from buffer with recv() call

```
..  Finished  Acceptable  Too high  ..
```

- LAS = LAST ACK SENT
- seq. number

W=5

Sliding Window
Sliding Window – Receiver (2)

- Suppose the next two segments arrive but app does not call `recv()`
Sliding Window – Receiver (3)

- Suppose the next two segments arrive but app does not call `recv()`
  - LAS rises, but we can’t slide window!

![Diagram](image)

- LAS: `W=5`
- Finished: `Acked`
- Too high
- seq. number

Finished | Aced | Too high | ..
Sliding Window – Receiver (4)

- If further segments arrive (even in order) we can fill the buffer
  - Must drop segments until app recvs!
Sliding Window – Receiver (5)

- App `recv()` takes two segments
  - Window slides (phew)

![Diagram showing sliding window with sequence numbers and acknowledgments.]

- `W=5` Acceptable
- `LAS` seq. number
- `Finished` Acker
- `Too high`
Flow Control

- Avoid loss at receiver by telling sender the available buffer space
  - $\text{WIN} =$ #Acceptable, not $W$ (from LAS)

![Diagram of flow control with sequence numbers and buffer states]

- LAS
- W=5
- Acceptable
- Too high
- Finished
- Acked
- Seq. number

.. Finished | Acked | Too high | ..
Flow Control (2)

- Sender uses the lower of the sliding window and flow control window (WIN) as the effective window size.
Flow Control (3)

- TCP-style example
  - SEQ/ACK sliding window
  - Flow control with WIN
  - SEQ + length < ACK+WIN
  - 4KB buffer at receiver
  - Circular buffer of bytes
Topic

- How to set the timeout for sending a retransmission
  - Adapting to the network path
Retransmissions

• With sliding window, the strategy for detecting loss is the **timeout**
  – Set timer when a segment is sent
  – Cancel timer when ack is received
  – If timer fires, **retransmit** data as lost
Timeout Problem

- Timeout should be “just right”
  - Too long wastes network capacity
  - Too short leads to spurious resends
  - But what is “just right”?

- Easy to set on a LAN (Link)
  - Short, fixed, predictable RTT

- Hard on the Internet (Transport)
  - Wide range, variable RTT
Example of RTTs

BCN → SEA → BCN
Example of RTTs (2)

BCN → SEA → BCN

Variation due to queuing at routers, changes in network paths, etc.

Propagation (+transmission) delay ≈ 2D
Example of RTTs (3)

- Timer too high!
- Need to adapt to the network conditions
- Timer too low!
Adaptive Timeout

- Keep smoothed estimates of the RTT (1) and variance in RTT (2)
  - Update estimates with a moving average
    1. $\text{SRTT}_{N+1} = 0.9*\text{SRTT}_N + 0.1*\text{RTT}_{N+1}$
    2. $\text{Svar}_{N+1} = 0.9*\text{Svar}_N + 0.1*|\text{RTT}_{N+1} - \text{SRTT}_{N+1}|$

- Set timeout to a multiple of estimates
  - To estimate the upper RTT in practice
    - TCP Timeout$_N = \text{SRTT}_N + 4*\text{Svar}_N$
Example of Adaptive Timeout

![Graph showing RTT (ms) vs Seconds]

- **RTT (ms)**
- **Seconds**

**SRTT**

**Svar**
Example of Adaptive Timeout (2)

![Graph showing RTT (ms) vs. Seconds with two timeout markers: Early timeout and Timeout (SRTT + 4*Svar).]
Adaptive Timeout (2)

- Simple to compute, does a good job of tracking actual RTT
  - Little “headroom” to lower
  - Yet very few early timeouts

- Turns out to be important for good performance and robustness