Topics

• Principles underlying transport-layer services
  – Demultiplexing
  – Detecting corruption
  – Reliable delivery
  – Flow control

• Transport-layer protocols
  – User Datagram Protocol (UDP)
  – Transmission Control Protocol (TCP)
Role of Transport Layer

• Application layer
  – Communication between networked applications
  – Protocols: HTTP, FTP, NNTP, and many others

• Transport layer
  – Communication between processes (e.g., socket)
  – Relies on network layer and serves the application layer
  – Protocols: TCP and UDP

• Network layer
  – Communication between nodes
  – Protocols: IP
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
Internet Transport Protocols

• Datagram *messaging* service (UDP)
  – No-frills extension of “best-effort” IP
  – Just send the data – **each send is a message**

• Reliable, *streaming*, in-order delivery (TCP)
  – Connection set-up
  – Discarding of corrupted packets
  – Retransmission of lost packets
  – Flow control
  – Congestion control (next lecture)

• **Services not** available from UDP or TCP
  – Delay guarantees
  – Bandwidth guarantees
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
- Host uses IP addresses and port numbers to direct the segment to appropriate socket

TCP/UDP segment format:

- 32 bits
- source port #  |  dest port #
- other header fields
- application data (message)
UDP
User Datagram Protocol (UDP)

- Lightweight communication between processes
  - Avoid overhead and delays of ordered, reliable delivery
  - Send messages to and receive them from a socket

- Lightweight delivery service
  - IP plus port numbers to support (de)multiplexing
  - Optional error checking on the packet contents

```
+--------+-------+
| SRC port | DST port |
+--------+-------+
| length  | checksum |
+--------+-------+
| DATA    |
```
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

• Low delay
  – UDP just blasts away without any formal preliminaries
  – … which avoids introducing delays such as setup

• No connection state
  – No allocation of buffers, parameters, sequence #s, etc.
  – … making it easier to handle many active clients

• Small packet header overhead
  – UDP header is only eight-bytes long
Popular Applications That Use UDP

- Multimedia streaming
  - Retransmitting lost/corrupted packets is not worthwhile
  - 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
  - Overhead of connection establishment is overkill
  - Easier to have application retransmit if needed

“Address for www.cnn.com?”

“12.3.4.15”
TCP
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
  – Similar to file I/O

• Reliable, in-order delivery
  – Checksums to detect corrupted data
  – Acknowledgments & retransmissions for reliable delivery
  – Sequence numbers to detect losses and reorder data

• Flow control
  – Prevent overflow of the receiver’s buffer space

• Congestion control
  – Adapt to network congestion for the greater good
TCP Segments
TCP “Stream of Bytes” Service
...Emulated Using TCP “Segments”

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out, or
3. “Pushed” by application.
TCP Segment

- **IP packet**
  - No bigger than **Maximum Transmission Unit (MTU)**
  - E.g., up to 1500 bytes on an Ethernet

- **TCP packet**
  - IP packet with a TCP header and data inside
  - TCP header is typically 20 bytes long

- **TCP segment**
  - No more than **Maximum Segment Size (MSS)** bytes
  - E.g., up to 1460 consecutive bytes from the stream
Sequence Numbers

Host A

ISN (initial sequence number)

Sequence number = 1st byte

TCP Data

ACK sequence number = next expected byte

Host B
Initial Sequence Number (ISN)

- Sequence number for the very first byte
  - Why not a de facto ISN of 0?

- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … and there is a chance an old packet is still in flight
  - … and might be associated with the new connection

- Security issue
  - An adversary can guess ISNs and hijack a connection

- So, TCP requires changing the ISN over time
  - Set from a 32-bit clock that ticks every 4 microseconds
  - … which only wraps around once every 4.55 hours!

- But, this means the hosts need to exchange ISNs
TCP Three-Way Handshake
Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open) to the host B
  - Host B returns a SYN acknowledgment (**SYN ACK**)
  - Host A sends an **ACK** to acknowledge the SYN ACK
  - Each host tells its ISN to the other host.
<table>
<thead>
<tr>
<th>Flags:</th>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RST</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PSH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>URG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flags:</th>
<th>Sequence number</th>
<th>Acknowledgment</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>0</th>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
<th>Data</th>
</tr>
</thead>
</table>

### Step 1: A’s Initial SYN Packet

#### Flags:
- SYN
- FIN
- RST
- PSH
- URG
- ACK

<table>
<thead>
<tr>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td>A’s Initial Sequence Number</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Acknowledgment</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Flags</th>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Options (variable)</th>
</tr>
</thead>
</table>

A tells B it wants to open a connection…
Step 2: B’s SYN-ACK Packet

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>B’s port</td>
<td>A’s port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B’s Initial Sequence Number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A’s ISN plus 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>Flags</td>
<td>Advertised window</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B tells A it accepts, and is ready to hear the next byte...

... upon receiving this packet, A can start sending data
Step 3: A’s ACK of the SYN-ACK

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>A’s port</th>
<th>B’s port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sequence number
- B’s ISN plus 1
- \[20\] 0 Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)

A tells B it’s okay to start sending!

... upon receiving this packet, B can start sending data
What if the SYN Packet Gets Lost?

• Suppose the SYN packet gets lost
  – Packet is lost inside the network, or
  – Server rejects the packet (e.g., listen queue is full)

• Eventually, no SYN-ACK arrives
  – Sender sets a timer and waits for the SYN-ACK
  – … and retransmits the SYN if needed

• How should the TCP sender set the timer?
  – Sender has no idea how far away the receiver is
  – Hard to guess a reasonable length of time to wait
  – Some TCPs use a very conservative default of 3 or 6 seconds
SYN Loss and Web Downloads

• User clicks on a hypertext link
  – Browser creates a socket and does a “connect”
  – The “connect” triggers the OS to transmit a SYN

• If the SYN is lost…
  – The 3-6 seconds of delay may be very long
  – The user may get impatient
  – … and click the hyperlink again, or click “reload”

• User triggers an “abort” of the “connect”
  – Browser creates a new socket and does a “connect”
  – Essentially, forces a faster send of a new SYN packet!
  – Sometimes very effective, and the page comes fast
TCP Retransmissions
Automatic Repeat reQuest (ARQ)

• Automatic Repeat Request
  – Receiver sends acknowledgment (ACK) when it receives packet
  – Sender waits for ACK and timeouts if it does not arrive within some time period

• Simplest ARQ protocol
  – Stop and wait
  – Send a packet, stop and wait until ACK arrives
Reasons for Retransmission

Packet lost

ACK lost
DUPLICATE PACKET

Early timeout
DUPLICATE PACKETS
How Long Should Sender Wait?

• Sender sets a timeout to wait for an ACK
  – Too short: wasted retransmissions
  – Too long: excessive delays when packet lost

• TCP sets timeout as a function of the RTT
  – Expect ACK to arrive after an RTT
  – … plus a fudge factor to account for queuing

• But, how does the sender know the RTT?
  – Can estimate the RTT by watching the ACKs
  – Smooth estimate: keep a running average of the RTT
    • $\text{EstimatedRTT} = a \times \text{EstimatedRTT} + (1 - a) \times \text{SampleRTT}$
  – Compute timeout: $\text{TimeOut} = 2 \times \text{EstimatedRTT}$
Example RTT Estimation

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

- Time (seconds): 1 to 106
- RTT (milliseconds): 100 to 350

Graph showing fluctuations in RTT over time, with two lines indicating SampleRTT and Estimated RTT.
A Flaw in This Approach

• An ACK doesn’t really acknowledge a transmission
  – Rather, it acknowledges receipt of the data

• Consider a retransmission of a lost packet
  – If you assume the ACK goes with the 1st transmission
  – … the SampleRTT comes out way too large

• Consider a duplicate packet
  – If you assume the ACK goes with the 2nd transmission
  – … the Sample RTT comes out way too small
Retransmission Ambiguity

Simple solution in the Karn/Partridge algorithm:
Only collect samples for segments sent one single time
Yet Another Limitation…

• Doesn’t consider variance in the RTT
  – If variance is small, the EstimatedRTT is pretty accurate
  – … but, if variance is large, the estimate isn’t all that good

• Key observation:
  – Using 2*RTT for timeout doesn’t work
  – At high loads round trip variance is high

• Solution:
  – If D denotes mean variance
  – Timeout = RTT + 4D

• Jacobson/Karels algorithm
  – See Section 5.2 of the Peterson/Davie book for details
TCP Sliding Window
Motivation for Sliding Window

• Stop-and-wait is inefficient
  – Only one TCP segment is “in flight” at a time
  – Especially bad when delay-bandwidth product is high

• Numerical example
  – 1.5 Mbps link with a 45 msec round-trip time (RTT)
    • Delay-bandwidth product is 67.5 Kbits (or 8 KBytes)
  – But, sender can send at most one packet per RTT
    • Assuming a segment size of 1 KB (8 Kbits)
    • … leads to 8 Kbits/segment / 45 msec/segment = 182 Kbps
    • That’s just one-eighth of the 1.5 Mbps link capacity!
Sliding Window

- Allow a larger amount of data “in flight”
  - Allow sender to get ahead of the receiver
  - … though not too far ahead
Receiver Buffering

- **Window size**
  - Amount that can be sent without acknowledgment
  - Receiver needs to be able to store this amount of data

- **Receiver advertises the window to the sender**
  - Receiver tells the sender the amount of free space left
  - … and the sender agrees not to exceed this amount
### TCP Header for Receiver Buffering

<table>
<thead>
<tr>
<th>Flags:</th>
<th>SYN</th>
<th>FIN</th>
<th>RST</th>
<th>PSH</th>
<th>URG</th>
<th>ACK</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence number</td>
<td></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

- **Data**
Fast Retransmission
Timeout is Inefficient

- Timeout-based retransmission
  - Sender transmits a packet and waits until timer expires
  - ... and then retransmits from the lost packet onward
Fast Retransmission

• Better solution possible under sliding window
  – Although packet n might have been lost
  – … packets n+1, n+2, and so on might get through

• Idea: have the receiver send ACK packets
  – ACK says that receiver is still awaiting n\textsuperscript{th} packet
    • And \textit{repeated} ACKs suggest later packets have arrived
  – Sender can view the “duplicate ACKs” as an early hint
    • … that the n\textsuperscript{th} packet must have been lost
    • … and perform the retransmission early

• Fast retransmission
  – Sender retransmits data after the triple duplicate ACK
Effectiveness of Fast Retransmit

When does Fast Retransmit work best?
- Long data transfers
  - High likelihood of many packets in flight
- Large window size
  - High likelihood of many packets in flight
- Low burstiness in packet losses
  - Higher likelihood that later packets arrive successfully

Implications for Web traffic
- Most Web transfers are short (e.g., 10 packets)
  - Short HTML files or small images
- So, often there aren’t many packets in flight
- … making fast retransmit less likely to “kick in”
- Forcing users to like “reload” more often…
Tearing Down the Connection
Tearing Down the Connection

Closing the connection

– Finish (FIN) to close and receive remaining bytes
  • And other host sends a FIN ACK to acknowledge
– Reset (RST) to close and not receive remaining bytes
Sending/Receiving the FIN Packet

- **Sending a FIN: close()**
  - Process is done sending data via the socket
  - Process invokes “close()” to close the socket
  - Once TCP has sent all of the outstanding bytes…
  - … then TCP sends a FIN

- **Receiving a FIN: EOF**
  - Process is reading data from the socket
  - Eventually, the attempt to read returns an EOF
Conclusions

• Transport protocols
  – Multiplexing and demultiplexing
  – Sequence numbers
  – Window-based flow control
  – Timer-based retransmission
  – Checksum-based error detection

• Reading from the book
  – Sections 2.5, 5.1-5.2, and 6.1-6.4