What Did We Learn Last Time?

- Transport has two purposes:
  - Mux/Demux: done using *Ports* and *Sockets*
  - Optional: Reliable delivery

- Reliable delivery involves many mechanisms
  - Requires delicate design to get them to work together
  - TCP is the standard example of a reliable transport
Reminder: The TCP Abstraction

- TCP delivers a **reliable, in-order, bytestream**

- **Reliable**: TCP resends lost packets (recursively)
  - Until it gives up and shuts down connection

- **In-order**: TCP only hands consecutive chunks of data to application

- **Bytestream**: TCP assumes there is an incoming stream of data, and attempts to deliver it to app
What Will We Cover Today?

- How TCP supports reliability

- TCP is not a perfect design
  - Probably wouldn’t make exactly same choices today

- But it is good enough
  - And is a great example of sweating the details...
  - ..and just happens to carry most of your traffic
TCP Header

<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></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data

Used to mux and demux
Last time: Components of a solution for reliable transport

- Checksums (for error detection)
- Timers (for loss detection)
- Acknowledgments
  - cumulative
  - selective
- Sequence numbers (duplicates, windows)
- Sliding Windows (for efficiency)
- Retransmissions
  - Go-Back-N (GBN)
  - Selective Replay (SR)
What does TCP do?

Many of our previous ideas, but some key differences

- Checksum
TCP Header

<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></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

Computed over pseudoheader and data
What does TCP do?

Many of our previous ideas, but some key differences

- Checksum

- **Sequence numbers are byte offsets**
  - And also includes notion of a “segment” and ISNs
  - Proof that networking is boring: 7 slides on sequence numbers!
TCP: Segments and Sequence Numbers
TCP “Stream of Bytes” Service…

Application @ Host A

Application @ Host B
... Provided Using TCP “Segments”

Segment sent when:
1. Segment full (Max Segment Size),
2. Not full, but times out
TCP Segment

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

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

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

Host A

ISN (initial sequence number)

$k$ bytes

Sequence number = 1\textsuperscript{st} byte in segment = ISN + $k$
Sequence Numbers

ISN (initial sequence number)

Host A

Sequence number = 1st byte in segment = ISN + k

TCP Data

TCP HDR

Host B

ACK sequence number = next expected byte = seqno + length(data)
## TCP Header

### Starting byte offset of data carried in this segment

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgment</td>
</tr>
</tbody>
</table>

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

### Data
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
**ACKing and Sequence Numbers**

- **Sender sends packet**
  - Data starts with sequence number $X$
  - Packet contains $B$ bytes $[X, X+1, X+2, \ldots X+B-1]$

- **Upon receipt of packet, receiver sends an ACK**
  - If all data prior to $X$ already received:
    - ACK acknowledges $X+B$ (because that is next expected byte)
  - If highest in-order byte received is $Y$ s.t. $(Y+1) < X$
    - ACK acknowledges $Y+1$
    - Even if this has been ACKed before
Normal Pattern

- Sender: seqno=X, length=B
- Receiver: ACK=X+B
- Sender: seqno=X+B, length=B
- Receiver: ACK=X+2B
- Sender: seqno=X+2B, length=B

- Seqno of next packet is same as last ACK field
TCP Header

Acknowledgment gives seqno just beyond highest seqno received **in order** ("What Byte is Next")
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers can buffer out-of-sequence packets (like SR)
Loss with cumulative ACKs

- Sender sends packets with 100B and seqnos:
  - 100, 200, 300, 400, 500, 600, 700, 800, 900, ...

- Assume the fifth packet (seqno 500) is lost, but no others

- Stream of ACKs will be:
  - 200, 300, 400, 500, 500, 500, 500, 500, ...
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers may not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
“Duplicate ACKs” are a sign of an isolated loss
- The lack of ACK progress means 500 hasn’t been delivered
- Stream of ACKs means some packets are being delivered

Therefore, could trigger resend upon receiving k duplicate ACKs
- TCP uses k=3

But response to loss is trickier....
Loss with cumulative ACKs

- Two choices:
  - Send missing packet and move sliding window by the number of dup ACKs
    - Speeds up transmission, but might be wrong
  - Send missing packet, and wait for ACK to move sliding window
    - Is slowed down by single dropped packets

- Which should TCP do?
What does TCP do?

Most of our previous tricks, but a few differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout
Retransmission Timeout

- If the sender hasn’t received an ACK by timeout, retransmit the first packet in the window

- How do we pick a timeout value?
Timing Illustration

Timeout too long → inefficient

Timeout too short → duplicate packets
Retransmission Timeout

- If haven’t received ack by timeout, retransmit the first packet in the window

- How to set timeout?
  - Too long: connection has low throughput
  - Too short: retransmit packet that was just delayed

- Solution: make timeout proportional to RTT

- But how do we measure RTT?
RTT Estimation

- Use exponential averaging of RTT samples

\[ \text{SampleRTT} = \text{AckRcvdTime} - \text{SendPacketTime} \]
\[ \text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT} \]
\[ 0 < \alpha \leq 1 \]
Exponential Averaging Example

\[ \text{EstimatedRTT} = \alpha \times \text{EstimatedRTT} + (1 - \alpha) \times \text{SampleRTT} \]

Assume RTT is constant \(\rightarrow\) \(\text{SampleRTT} = \text{RTT}\)
Problem: Ambiguous Measurements

- How do we differentiate between the real ACK, and ACK of the retransmitted packet?
Karn/Partridge Algorithm

- Measure $SampleRTT$ only for original transmissions
  - Once a segment has been retransmitted, do not use it for any further measurements
- Computes $EstimatedRTT$ using $\alpha = 0.875$

- Timeout value (RTO) = $2 \timesEstimatedRTT$
- Employs **exponential backoff**
  - Every time RTO timer expires, set $RTO \leftarrow 2 \cdot RTO$
  - (Up to maximum $\geq 60$ sec)
  - Every time new measurement comes in (= successful original transmission), collapse RTO back to $2 \times EstimatedRTT$
Karn/Partridge in action

Figure 5: Performance of an RFC793 retransmit timer

from Jacobson and Karels, SIGCOMM 1988
Jacobson/Karels Algorithm

- Problem: need to better capture variability in RTT
  - Directly measure deviation

- Deviation = | SampleRTT – EstimatedRTT |
- EstimatedDeviation: exponential average of Deviation

- RTO = EstimatedRTT + 4 x EstimatedDeviation
With Jacobson/Karels

Figure 6: Performance of a Mean+Variance retransmit timer
What does TCP do?

Most of our previous ideas, but some key differences

- Checksum
- Sequence numbers are byte offsets
- Receiver sends cumulative acknowledgements (like GBN)
- Receivers do not drop out-of-sequence packets (like SR)
- Introduces fast retransmit: optimization that uses duplicate ACKs to trigger early retransmission
- Sender maintains a single retransmission timer (like GBN) and retransmits on timeout
TCP Header: What’s left?

<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>Advertised window</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

“Must Be Zero”
6 bits reserved

Number of 4-byte words in TCP header;
5 = no options
TCP Header: What’s left?

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sequence number

Acknowledgment

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

Checksum

Options (variable)

Data

Used with **URG** flag to indicate urgent data (not discussed further)

- **URG** flag to indicate urgent data (not discussed further)
TCP Header: What’s left?

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

Data
TCP Connection Establishment and Initial Sequence Numbers
Initial Sequence Number (ISN)

- Sequence number for the very first byte
- Why not just use ISN = 0?
- Practical issue
  - IP addresses and port #s uniquely identify a connection
  - Eventually, though, these port #s do get used again
  - … small chance an old packet is still in flight
  - Also, others might try to spoof your connection
- Why does using ISN help?
- TCP therefore requires changing ISN
- Hosts exchange ISNs when they establish a connection
Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a **SYN** (open; “synchronize sequence numbers”) to 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.
## TCP Header

<table>
<thead>
<tr>
<th>Source port</th>
<th>Destination port</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Advertised window</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

**Flags:**
- SYN
- ACK
- FIN
- RST
- PSH
- URG
Step 1: A’s Initial SYN Packet

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

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

- A’s port
- B’s port
- A’s Initial Sequence Number
- B’s ISN plus 1
- 20B
- Flags
- Advertised window
- Checksum
- Urgent pointer
- Options (variable)

Flags: SYN, ACK, FIN, RST, PSH, URG

A tells B it’s likewise okay to start sending

... upon receiving this packet, B can start sending data
Timing Diagram: 3-Way Handshaking

Client (initiator)

- `connect()`

Active
- Open

Server
- Passive
- Open

SYN, SeqNum = x

SYN + ACK, SeqNum = y, Ack = x + 1

ACK, Ack = y + 1

listen()
What if the SYN Packet Gets Lost?

- Suppose the SYN packet gets lost
  - Packet is lost inside the network, or:
  - Server discards the packet (e.g., it’s too busy)

- 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
  - SHOULD (RFCs 1122 & 2988) use default of 3 seconds
    - Some implementations instead use 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…
  - 3-6 seconds of delay: can be very long
  - User may become impatient
  - … and click the hyperlink again, or click “reload”
- User triggers an “abort” of the “connect”
  - Browser creates a new socket and another “connect”
  - Essentially, forces a faster send of a new SYN packet!
  - Sometimes very effective, and the page comes quickly
Tearing Down the Connection
Normal Termination, One Side At A Time

- Finish (**FIN**) to close and receive remaining bytes
  - **FIN** occupies one byte in the sequence space
- Other host acks the byte to confirm
- Closes A’s side of the connection, but **not** B’s
  - Until B likewise sends a **FIN**
  - Which A then acks

**TIME_WAIT:**
- Avoid reincarnation
- B will retransmit **FIN** if ACK is lost
Normal Termination, Both Together

- Same as before, but B sets **FIN** with their ack of A’s **FIN**
A sends a RESET (RST) to B
  - E.g., because application process on A crashed

That’s it
  - B does not ack the RST
  - Thus, RST is not delivered reliably
  - And: any data in flight is lost
  - But: if B sends anything more, will elicit another RST
### TCP Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Source port of the packet</td>
</tr>
<tr>
<td>Destination port</td>
<td>Destination port of the packet</td>
</tr>
<tr>
<td>Sequence number</td>
<td>Sequence number of the packet</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>Acknowledgment number of the packet</td>
</tr>
<tr>
<td>Advertised window</td>
<td>Advertised window of the receiver</td>
</tr>
<tr>
<td>Flags</td>
<td>Flags of the packet</td>
</tr>
<tr>
<td>HdrLen</td>
<td>Header length</td>
</tr>
<tr>
<td>Checksum</td>
<td>Checksum of the packet</td>
</tr>
<tr>
<td>Urgent pointer</td>
<td>Urgent pointer of the sender</td>
</tr>
<tr>
<td>Options (variable)</td>
<td>Variable options of the packet</td>
</tr>
<tr>
<td>Data</td>
<td>Data of the packet</td>
</tr>
</tbody>
</table>

**Flags:**
- SYN
- ACK
- FIN
- RST
- PSH
- URG
TCP State Transitions

- **CLOSED**
  - Passive open
  - Close
- **LISTEN**
  - Passive open
  - Close
- **SYN_RCVD**
  - SYN/SYN + ACK
  - SYN + ACK/ACK
  - ACK
  - Close/FIN
- **SYN_SENT**
  - Send/SYN
  - ACK
- **ESTABLISHED**
  - FIN/ACK
  - FIN/ACK
  - FIN/FIN
- **FIN_WAIT_1**
  - ACK
  - FIN/ACK
  - Close/FIN
- **FIN_WAIT_2**
  - ACK + FIN/ACK
  - FIN/ACK
- **CLOSING**
  - ACK
  - Timeout after two segment lifetimes
  - FIN/ACK
  - TIME_WAIT
  - ACK
- **CLOSE_WAIT**
  - Close/FIN
- **LAST_ACK**
  - ACK
  - CLOSED

Data, ACK exchanges are in here
An Simpler View of the Client Side

- **CLOSED**
  - SYN (Send)
  - Rcv. SYN+ACK, Send ACK

- **SYN_SENT**
  - Send ACK
  - Rcv. SYN+ACK, Send ACK

- **FIN_WAIT1**
  - Send FIN
  - Rcv. ACK, Send Nothing

- **FIN_WAIT2**
  - Rcv. FIN, Send ACK

- **TIME_WAIT**
  - Rcv. FIN, Send ACK
# TCP Header

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
</tr>
<tr>
<td>Sequence number</td>
<td>Acknowledgment</td>
</tr>
<tr>
<td>Advertised window</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
</tbody>
</table>

Used to negotiate use of additional features *(details in section)*
# TCP Header

<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></td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
</tbody>
</table>

Data
Last time: Sliding Window

- Both sender & receiver maintain a window

- Left edge of window:
  - Sender: beginning of unacknowledged data
  - Receiver: beginning of expected data
    - First “hole” in received data
    - When sender gets ack, knows that receiver’s window has moved

- Right edge: Left edge + constant
  - constant only limited by buffer size in the transport layer
TCP: Sliding Window (so far)

- Both sender & receiver maintain a window

- **Left edge** of window:
  - Sender: beginning of *unacknowledged* data
  - Receiver: beginning of *undelivered* data

- **Right edge:** Left edge + *constant*
  - constant only limited by buffer size in the transport layer
Sliding Window at Sender (so far)

Sending process

TCP

Buffer size (B)

Previously ACKed bytes

First unACKed byte

Last byte written

Last byte can send

Previously ACKed bytes

First unACKed byte
Sliding Window at Receiver (so far)

- Receiving process
- Last byte read
- Buffer size (B)
- Received and ACKed
- Next byte needed (1st byte not received)
- Last byte received

Sender might overrun the receiver’s buffer
Solution: Advertised Window (Flow Control)

- Receiver uses an “Advertised Window” (W) to prevent sender from overflowing its window
  - Receiver indicates value of W in ACKs
  - Sender limits number of bytes it can have in flight <= W
Sliding Window at Receiver

\[ W = B - (\text{LastByteReceived} - \text{LastByteRead}) \]
Sliding Window at Sender (so far)

Sending process

TCP

W

First unACKed byte

Last byte written

Last byte can send
Sliding Window w/ Flow Control

- Sender: window **advances** when new data ack’d
- Receiver: window advances as receiving process **consumes** data
- Receiver **advertises** to the sender where the receiver window currently ends (‘’righthand edge’’)
  - Sender agrees not to exceed this amount
Advertised Window Limits Rate

- Sender can send no faster than $W/RTT$ bytes/sec
- Receiver only advertises more space when it has consumed old arriving data
- In original TCP design, that was the **sole** protocol mechanism controlling sender’s rate
- What’s missing?
Taking Stock (1)

The concepts underlying TCP are simple

- acknowledgments (feedback)
- timers
- sliding windows
- buffer management
- sequence numbers
Taking Stock (1)

- The concepts underlying TCP are simple
- But tricky in the details
  - How do we set timers?
  - What is the seqno for an ACK-only packet?
  - What happens if advertised window = 0?
  - What if the advertised window is $\frac{1}{2}$ an MSS?
  - Should receiver acknowledge packets right away?
  - What if the application generates data in units of 0.1 MSS?
  - What happens if I get a duplicate SYN? Or a RST while I’m in FIN_WAIT, etc., etc., etc.
Taking Stock (1)

- The concepts underlying TCP are simple
- But tricky in the details
- Do the details matter?
Sizing Windows for Congestion Control

- What are the problems?
- How might we address them?