CS 168
Transport and TCP

Fall 2022
Sylvia Ratnasamy
CS168.io
Agenda

- What does the Transport layer do?
- The design of TCP
Transport Layer
Transport in our layered architecture

- Application
- Transport
- Network
- Datalink
- Physical

- OS (networking stack)
- NIC (Network Interface Card)

Applications connect to the OS through the networking stack, which resides on the NIC.
Role of Transport Layer

- Bridging the gap between
  - The abstractions application designers want
  - The abstractions networks can easily support

- Could have been built into apps, but a common implementation makes app development easier
Role of Transport Layer?

- **Application layer**
  - Communication for specific applications
  - E.g., File Transfer Protocol (FTP), Network Time Protocol (NTP), HyperText Transfer Protocol (HTTP), Internet Message Access Protocol (IMAP)

- **Transport layer**
  - *What do we need here?*

- **Network layer**
  - Logical communication between nodes
  - E.g., IP
Role of Transport Layer?

● Application layer
  ● Communication for specific applications
  ● E.g., File Transfer Protocol (FTP), Network Time Protocol (NTP), HyperText Transfer Protocol (HTTP), Internet Message Access Protocol (IMAP)

● Transport layer
  ● What do we need here?

● Network layer
  ● Best-effort global packet delivery
  ● IP
Role of Transport Layer?

- **Application layer**
  - Communication for specific applications
  - E.g., File Transfer Protocol (FTP), Network Time Protocol (NTP), HyperText Transfer Protocol (HTTP), Internet Message Access Protocol (IMAP)

- **Transport layer**
  - *What do we need here?*

- **Network layer**
  - Best-effort global packet delivery
  - IP
What does the transport layer address?

- Demultiplexing
  - Identify app this data belongs to (lecture#3)
- Reliability
  - Last lecture
- Translate from packets to app-level abstractions
  - E.g., between bytestreams and packets (this lecture)
- Avoid overloading the receiver (this lecture)
- Avoid overloading the network (future lectures)
Let’s first talk about these issues in general

...and then how TCP addresses them
Demultiplexing?
Recall: **logical ports**

Place where app connects to the OS network stack
Hence, demultiplexing

- Achieved by defining a field ("port") that identifies the application
- Field is carried in a packet’s L4 protocol header
Reliable Delivery

- Last lecture

- We’ve identified our design building blocks
  - Checksums
  - ACK/NACKs
  - Timeouts
  - Retransmissions
  - Sequence numbers
  - Windows

- And discussed tradeoffs in how to apply them
  - Individual vs. Full vs. Cumulative ACKs
  - Timeout vs. ACK-driven loss detection
Application-layer abstractions

- Ideally, app doesn’t see the gory details of the network
  - packets, ACKs, duplicates, reordering, corruption, ...

- Want a higher-level abstraction that meets app needs
Application Abstractions

- **Reliable in-order bytestream** delivery (TCP)
  - Logical “pipe” between sender and receiver
  - Bytes inserted into pipe by sender-side app
  - They emerge, in order, at the receiving app

- **Individual message** delivery (UDP)
  - Unreliable (application responsible for resending)
  - Messages limited to single packet
What does the transport layer address?

- Demultiplexing
  - Identify app this data belongs to (logical ports, lecture#3)
- Reliability
  - Last lecture (though TCP does things a little differently)
- Translate from packets to app-level abstractions
  - E.g., between bytestreams and packets
- Avoid overloading the receiver
- Avoid overloading the network
How big should the window be?

● Lecture#12: Pick window size $W$ to balance three goals
  ● Take advantage of network capacity ("fill the pipe")
  ● But don’t overload the receiver (flow control)
  ● And don’t overload links (congestion control)

● Lecture#12: For the first goal: $W \times \text{pkt}_\text{size} \sim \text{RTT} \times B$
  ● RTT is round-trip time and $B$ is the bottleneck BW
  ● This is an upper bound on the desired size of $W$

● Now consider the other two goals...
Don’t overload the receiver

- Consider the transport layer at the receiver side

- May receive packets out-of-order but can only deliver them to the application in order

- Hence, the receiver must buffer incoming packets that are out of order
  - Must continue to do so until all “missing” packets arrive!

- Must ensure the receiver doesn’t run out of buffers
Hence: Flow Control

The basic idea is very simple...

- Receiver tells sender how much space it has left
  - TCP calls this the “advertised” window
- Advertisement is carried in ACKs
- Sender adjusts its window accordingly
  - Packets in flight cannot exceed the receiver’s advertised window
Don’t overload the network

- Previously: sender sets $W$ to fully consume the bottleneck link bandwidth
  - I.e., sender is sending data at the rate of $B$
- In practice, bottleneck is shared with other flows
- Hence, sender should only consume *its share* of $B$
- But what is this share?

![Diagram showing network}
Congestion Control

- The transport layer at the sender implements a congestion control algorithm that dynamically computes the sender’s share of the bottleneck link BW

- TCP calls this the sender’s congestion window (cwnd)

- Computed to balance multiple goals
  - Maximize my performance
  - Without overloading any link (avoid dropped packets)
  - While sharing bandwidth ”fairly” with other senders

- Topic for (multiple) future lectures
How big should the window be?

• Pick window size $W$ to balance three goals
  • Take advantage of network capacity ("fill the pipe")
  • But don’t overload the receiver (flow control)
  • And don’t overload links (congestion control)

• First goal: $W \sim RTT \times B$
• Second: $W \sim$ receiver’s advertised window
• Third: $W \sim$ sender’s congestion window ($cwnd$)

• Window size is set to the minimum of the above
In practice

- A sender’s cwnd should be $\leq RTT \times B$
- And it’s difficult for the sender to discover $B$
- Hence, window size is the minimum of:
  - The congestion window computed at the sender
  - The receiver’s advertised window
Recap: what the transport layer tackles

- Demultiplexing
  - logical ports
- Reliability
  - acks, timeouts, windows, etc.
- Translation between abstractions
  - between packets and bytestreams (coming up)
- Avoid overloading the receiver
  - receiver’s advertised window
- Avoid overloading the network
  - sender computes a congestion window
What if your app doesn’t want all these features?

- E.g., an application that doesn’t need reliability
- E.g., an app that exchanges very short messages

- UDP: User Datagram Protocol
  - A no-frills, minimalist protocol
  - Only implements mux/demux
TCP

Vint Cerf

Bob Kahn
The TCP Abstraction

- TCP delivers a **reliable, in-order, bytestream**
TCP “Stream of Bytes” Service...

Application @ Host A

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

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

- TCP packet
  - IP packet with a TCP header and data inside

- IP packet
  - No bigger than Maximum Transmission Unit (MTU)

- TCP segment
  - No more than Maximum Segment Size (MSS) bytes
  - MSS = MTU – (IP header) – (TCP header)
... Implemented Using TCP “Segments”

Application @ Host A

Application @ Host B
… Described by TCP headers

Header carries a "sequence number" that indicates where in the bytestream this segment fits.
Major Notation Change

- Previously we focused on packets:
  - Packets had numbers
  - ACKs referred to those numbers
  - Window sizes expressed in terms of # of packets

- TCP focuses on bytes. Thus,
  - Packets identified by the bytes they carry
  - ACKs refer to the bytes received
  - Window size expressed in terms of # of bytes

- You should be prepared to reason in terms of either
TCP Sequence Numbers

Numbering starts with an **ISN (Initial Sequence Number)**

1st byte is ISN+1

Host A

Host B
TCP Sequence Numbers

Numbering starts with an ISN (Initial Sequence Number)

1st byte is ISN+1

Sequence number = 1st byte in segment (e.g., ISN+k)
TCP Sequence Numbers

Numbering starts with an **ISN (Initial Sequence Number)**

1st byte is ISN+1

ACK sequence number = next expected byte (e.g., ISN+k + length(data))

Host A

Host B
The TCP Abstraction

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

- Reliability requires keeping state
  - Sender: packets sent but not ACKed, related timers
  - Receiver: out-of-order packets

- Each bytestream is called a **connection** or session
  - Each with their own connection state
  - State is in hosts, not network!
Note#1: TCP is “connection oriented”

- TCP includes a connection setup and tear-down step
  - Used to initialize connection state at both endpoints
  - Details coming up ...
#2: TCP connections are full-duplex

- So far, we’ve talked about a connection as having a sender side and a receiver side

- But connections in TCP are full-duplex
  - Each side of the connection can be sender and receiver
  - I.e., A can send data to B, while B sends data to A
  - Simultaneously, over the same connection
  - Packets carry both data and ACK info

- We can usually ignore this point (for this class)
- Except when it comes to connection establishment
- Will return to this later ...
The TCP Abstraction

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

- TCP is connection-oriented
  - Per-connection state is maintained at sender & receiver
Functionality

- Mux/demux among processes
- Reliability
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down
Ports

- 16-bit port address space for TCP and UDP

- Some ports are “well known” (0-1023)
  - e.g., ssh:22, http:80
  - Services can listen on well-known port
  - Client (app) knows appropriate port on server

- Other ports are “ephemeral” (most 1024-65535):
  - Given to clients (at random)
<table>
<thead>
<tr>
<th>4</th>
<th>5</th>
<th>8-bit Type of Service (TOS)</th>
<th>16-bit Total Length (Bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>16-bit Identification</td>
<td>3-bit Flags 13-bit Fragment Offset</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8-bit Protocol</td>
<td>16-bit Header Checksum</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>32-bit Source IP Address</td>
<td>32-bit Destination IP Address</td>
</tr>
</tbody>
</table>

Payload
<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Service (TOS)</td>
<td>8</td>
</tr>
<tr>
<td>16-bit Total Length (Bytes)</td>
<td>16</td>
</tr>
<tr>
<td>16-bit Identification</td>
<td>16</td>
</tr>
<tr>
<td>3-bit Flags</td>
<td>3</td>
</tr>
<tr>
<td>13-bit Fragment Offset</td>
<td>13</td>
</tr>
<tr>
<td>8-bit Time to Live (TTL)</td>
<td>8</td>
</tr>
<tr>
<td>6 = TCP</td>
<td></td>
</tr>
<tr>
<td>17 = UDP</td>
<td></td>
</tr>
<tr>
<td>16-bit Header Checksum</td>
<td>16</td>
</tr>
<tr>
<td>32-bit Source IP Address</td>
<td>32</td>
</tr>
<tr>
<td>32-bit Destination IP Address</td>
<td>32</td>
</tr>
</tbody>
</table>

**Payload**
<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of Service (TOS)</td>
<td>8-bit</td>
<td>4</td>
</tr>
<tr>
<td>Total Length (Bytes)</td>
<td>16-bit</td>
<td>5</td>
</tr>
<tr>
<td>Identification</td>
<td>16-bit</td>
<td>8</td>
</tr>
<tr>
<td>Flags</td>
<td>3-bit</td>
<td>13</td>
</tr>
<tr>
<td>Fragment Offset</td>
<td>13-bit</td>
<td>3</td>
</tr>
<tr>
<td>Time to Live (TTL)</td>
<td>8-bit</td>
<td>5</td>
</tr>
<tr>
<td>Protocol</td>
<td>6</td>
<td>6 = TCP</td>
</tr>
<tr>
<td></td>
<td></td>
<td>17 = UDP</td>
</tr>
<tr>
<td>Header Checksum</td>
<td>16-bit</td>
<td>14</td>
</tr>
<tr>
<td>Source IP Address</td>
<td>32-bit</td>
<td>18</td>
</tr>
<tr>
<td>Destination IP Address</td>
<td>32-bit</td>
<td>34</td>
</tr>
</tbody>
</table>

TCP header

Payload
# TCP Header

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

Data

**Note:** Used to mux and demux
Functionality

- Mux/demux among processes
- Reliability
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down
How does TCP handle reliability?

Many of our previous ideas, with some key differences:

- Sequence numbers are **byte offsets**
- Uses **cumulative** ACKs; with “next expected byte” semantics
- Uses **sliding window**: up to W contiguous bytes in flight

Next expected byte, as ACKed by receiver
How does TCP handle reliability?

Many of our previous ideas, with some key differences

- Sequence numbers are **byte offsets**
- Uses **cumulative** ACKs; with “next expected byte” semantics
- Uses **sliding window**: up to W contiguous bytes in flight
- Retransmissions triggered by **timeouts** and **duplicate ACKs**
- **Single timer**, for left hand side (1st byte) of the window
- Window size is a function of cwnd and advertised window
  - With special accounting for duplicate ACKs (future lecture)
- **Timeouts are computed from RTT measurements**
  - Covered in section
**ACKing and Sequence Numbers**

- **Sender sends packet**
  - Data starts with sequence number $X$
  - Packet contains $B$ bytes
    - $X$, $X+1$, $X+2$, …. $X+B-1$

- **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 contiguous byte received is a smaller value $Y$
    - ACK acknowledges $Y+1$ (*because TCP uses cumulative ACKs*)
**Pattern (w/ only one packet in flight)**

- **Sender**: seq number =X, length=B
- **Receiver**: ACK=X+B
- **Sender**: seq number =X+B, length=B
- **Receiver**: ACK=X+2B
- **Sender**: seq number =X+2B, length=B
- Seq number of next packet is same as last 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></td>
</tr>
<tr>
<td>Acknowledgment</td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>Advertised window</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

Starting byte offset of data carried in this segment.
TCP Header

Acknowledgment gives sequence number just beyond the highest sequence number received in order (i.e., next expected byte)
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>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>

Data
Functionality

- Mux/demux among processes
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (to not overload network)
- “Connection” set-up & tear-down
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>Advertised window</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Flags</th>
<th>Checksum</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Data
Implementing Sliding Window

- Sender maintains a window
  - Data that has been sent but not yet ACK’ed
  - Window size = maximum amount of data in flight

- **Left edge** of window:
  - Beginning of unacknowledged data

- **Right edge** of window (ignoring congestion control)
  - Depends on the window advertised by receiver
  - Which depends on receiver’s buffer space

Next expected byte, as ACKed by receiver

![Diagram of sliding window](image)
**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>

<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>

*“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>

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

<table>
<thead>
<tr>
<th>HdrLen</th>
<th>Urgent pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<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>

Used with **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>
</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>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>

Options (we’ll ignore)
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></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Data</th>
</tr>
</thead>
</table>
Functionality

- Mux/demux among processes
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (future lecture)
- “Connection” set-up & tear-down
Functionality

- Mux/demux among processes
- Retransmission of lost and corrupted packets
- Flow control (to not overflow receiver)
- Congestion control (future lecture)
- “Connection” set-up & tear-down
TCP Connection Establishment and Initial Sequence Numbers
Establishing a TCP Connection

- Three-way handshake to establish connection
  - Host A sends a **SYN** 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>Fields</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Source port number</td>
</tr>
<tr>
<td>Destination port</td>
<td>Destination port number</td>
</tr>
<tr>
<td>Sequence number</td>
<td>Sequence number sent by sender</td>
</tr>
<tr>
<td>Acknowledgment</td>
<td>Acknowledgment of received packets</td>
</tr>
<tr>
<td>Advertised window</td>
<td>Advertised window size</td>
</tr>
<tr>
<td>HdrLen</td>
<td>Header length</td>
</tr>
<tr>
<td>Flags</td>
<td>Control flags: SYN, ACK, FIN, RST, PSH, URG</td>
</tr>
<tr>
<td>Checksum</td>
<td>Checksum for data integrity</td>
</tr>
<tr>
<td>Urgent pointer</td>
<td>Pointer for urgent data</td>
</tr>
<tr>
<td>Options</td>
<td>Options: variable</td>
</tr>
<tr>
<td>Data</td>
<td>Data area</td>
</tr>
</tbody>
</table>

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

A’s port | B’s port
---|---
| A’s Initial Sequence Number
(Irrelevant since ACK not set)

Flags: SYN
ACK
FIN
RST
PSH
URG

5=20B | 0 | Flags | Advertised window

Checksum | Urgent pointer

Options (variable)

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 tells B it’s likewise okay to start sending

... upon receiving this packet, B can start sending data
Tearing Down the Connection
## 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>Flags</td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>Options (variable)</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
</tr>
</tbody>
</table>

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

See `/usr/include/netinet/tcp.h` on Unix Systems
Normal Termination, One Side At A Time

- Finish (FIN) to close connections
- Other host ack’s
- Closes A’s side of the connection, but not B’s
  - Until B likewise sends a FIN
  - Which A then acks

Connection now closed
Connection now half-closed
Wait a while
A sends a RESET (RST) to B
  • E.g., because A restarted

That’s it
  • B does not ack the RST
  • Thus, RST is not delivered reliably
  • And: any data in flight is lost
  • If B sends anything more, will elicit another RST
TCP State Transitions

- **CLOSED**
  - Passive open
  - Close
  - Active open / SYN

- **LISTEN**
  - SYN/SYN + ACK

- **SYN_RCVD**
  - Close/FIN
  - SYN/SYN + ACK

- **SYN_SENT**
  - Send/SYN
  - SYN + ACK/ACK

- **ESTABLISHED**
  - ACK
  - SYN/SYN + ACK
  - Close/FIN
  - FIN/ACK

- **FIN_WAIT_1**
  - ACK
  - FIN/ACK
  - Close/FIN

- **FIN_WAIT_2**
  - FIN/ACK

- **CLOSING**
  - ACK
  - Timeout after two segment lifetimes

- **CLOSE_WAIT**
  - Close/FIN

- **LAST_ACK**
  - ACK

- **TIME_WAIT**
  - ACK

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

CLOSED

SYN (Send)

TIME_WAIT

Rcv. FIN, Send ACK

FIN_WAIT2

FIN_WAIT1

Rcv. ACK, Send Nothing

FIN_WAIT1

Send FIN

SYN_SENT

ESTABLISHED

Rcv. SYN+ACK, Send ACK
In Summary

- **TCP**
  - An elegant (though not perfect) piece of engineering that has stood the test of time
    - Thought experiment: will TCP continue to be a good solution?
  - Plenty of evolution in individual pieces
    - **Congestion control**
      - Better acknowledgements, ISN selection, timer estimation, *etc.*
  - But core architectural decisions/abstractions remain
    - Bytestreams, connection oriented, windows etc.

- Next time: start on congestion control!
Any Questions?