TCP Usage Model

- Connection setup
  - 3-way handshake

- Data transport
  - Sender writes data
  - TCP
    - Breaks data into segments
    - Sends each segment over IP
    - Retransmits, reorders and removes duplicates as necessary
  - Receiver reads some data

- Teardown
  - 4 step exchange
TCP Connection Establishment

- 3-Way Handshake
  - Sequence Numbers
    - J, K
  - Message Types
    - Synchronize (SYN)
    - Acknowledge (ACK)
  - Passive Open
    - Server listens for connection from client
  - Active Open
    - Client initiates connection to server

Client

Server

listen

Synchronize (SYN) J

SYN K, acknowledge (ACK) J+1

ACK K+1

Time flows down
Purpose of the handshake

- Why use a handshake before sending / processing data?
- Suppose we don’t wait for the handshake
  - send data (e.g., HTTP request) along with SYN
  - deliver to application
  - send some results (e.g., index.html) along with SYN ACK
- What could go wrong?
  - Hint: remember packets can be delayed, dropped, duplicated, …
Purpose of the handshake

- Why use a handshake before sending / processing data?
- Duplicated packet causes data to be sent to application twice
- Why does handshake fix this?
Purpose of the handshake

- If server receives request a second time, it responds with SYN ACK a second time
- But sender will not subsequently respond with ACK ("what is this garbage I just received??")
Another purpose of the handshake

- No handshake == security hole
  - Attacker sends request
  - ...but spoofs source address, using address of a victim (C)
  - Server happily sends massive amounts of data to victim
  - Attacker repeats for 10,000 web servers
  - Massive denial of service attack, almost free and anonymous for the attacker!

- Used in the largest distributed denial of service (DDoS) attacks in 2008, 2009, and 2010
  - Use services that lack handshake (e.g., DNS over UDP)
  - Amplification factor 1:76 in 2008!
Another purpose of the handshake

- Handshake lets server verify source address is real

- SYN

- SYN ACK

- Doesn’t match a connection initiated by C: ignore (or reply with reset)

- No ACK received after timeout: drop connection without sending data

Q: does this prevent reflection attack?  
A: No, but at least it prevents amplification
Handshaking

- Internet was not designed for accountability
  - Hard to tell where a packet came from
  - ISPs filter suspicious packets: sometimes easy, sometimes hard, and sometimes not done
    - And the Internet is not secure until everyone filters

- More generally, Internet was not designed for security
  - Vulnerabilities in most of the core protocols
  - Even with handshake, early designs are vulnerable
    - Had predictable Initial Sequence Number (why’s that bad?)
    - Because security was not initial goal of the handshake
TCP Data Transport

- Data broken into segments
  - Limited by maximum segment size (MSS)
  - Defaults to 352 bytes
  - Negotiable during connection setup
  - Typically set to
    - MTU of directly connected network – size of TCP and IP headers

- Three events cause a segment to be sent
  - ≥ MSS bytes of data ready to be sent
  - Explicit PUSH operation by application
  - Periodic timeout
TCP Byte Stream

- Application process
  - Write bytes
  - Send buffer
- TCP Segment
- TCP Segment
- TCP Segment

- Application process
  - Read bytes
  - Recv buffer
TCP Connection Termination

- Two generals problem
  - Enemy camped in valley
  - Two generals’ hills separated by enemy
  - Communication by unreliable messengers
  - Generals need to agree whether to attack or retreat
Two generals problem

- Can messages over an unreliable network be used to guarantee two entities do something simultaneously?
  - No, even if all messages get through

- No way to be sure last message gets through!
TCP Connection Termination

- **Message Types**
  - Finished (FIN)
  - Acknowledge (ACK)

- **Active Close**
  - Sends no more data

- **Passive Close**
  - Accepts no more data
## TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Start</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>Header Length</td>
<td>16</td>
<td>24</td>
</tr>
<tr>
<td>Flags</td>
<td>24</td>
<td>32</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>32</td>
<td>40</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>40</td>
<td>48</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>48</td>
<td>56</td>
</tr>
<tr>
<td>Options</td>
<td>56</td>
<td>64</td>
</tr>
</tbody>
</table>
### TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-31</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td></td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>8</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>16</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>24</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>32</td>
</tr>
<tr>
<td>Options</td>
<td></td>
</tr>
</tbody>
</table>

- **16-bit source and destination ports**
TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-31</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>32-39</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>40-47</td>
</tr>
<tr>
<td>Header Length</td>
<td>48-49</td>
</tr>
<tr>
<td>Flags</td>
<td>50-51</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>52-59</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>60-63</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>64-65</td>
</tr>
<tr>
<td>Options</td>
<td>66-67</td>
</tr>
</tbody>
</table>

- 32-bit send and ACK sequence numbers
ACKing and Sequence Numbers

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

![Diagram showing sequence of bytes](image-url)
ACKing and Sequence Numbers

- 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)
ACKing and Sequence Numbers

- Upon receipt of packet, receiver sends an ACK
  - If highest byte already received is some smaller value Y
    - ACK acknowledges Y+1
    - Even if this has been ACKed before

![Diagram showing byte Y and byte Y+1](image-url)
## TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-31</td>
</tr>
<tr>
<td>Header Length</td>
<td>32-35</td>
</tr>
<tr>
<td>Flags</td>
<td>36-39</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>40-47</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>48-55</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>56-59</td>
</tr>
<tr>
<td>Options</td>
<td>60-63</td>
</tr>
</tbody>
</table>

- **4-bit header length in 4-byte words**
  - Minimum 5 bytes
  - Offset to first data byte
## TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-31</td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
</tr>
<tr>
<td>Flags</td>
<td>3-4</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>5-15</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>16-23</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>24-27</td>
</tr>
<tr>
<td>Options</td>
<td>28-31</td>
</tr>
</tbody>
</table>

### Reserved
- **Must be 0**
TCP Segment Header Format

<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>ACK Sequence Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Header Length</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>TCP Checksum</th>
<th>Urgent Pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

6 1-bit flags

- **URG**: Contains urgent data
- **ACK**: Valid ACK seq. number
- **PSH**: Do not delay data delivery
- **RST**: Reset connection
- **SYN**: Synchronize for setup
- **FIN**: Final segment for teardown
## TCP Segment Header Format

<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>Destination Port</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence Number</td>
<td>ACK Sequence Number</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Length</td>
<td>Flags</td>
<td>Advertised Window</td>
<td></td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>Urgent Pointer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **16-bit advertised window**
  - Space remaining in receive window
TCP Segment Header Format

16-bit checksum
- Uses IP checksum algorithm
- Computed on header, data and pseudo header
## TCP Segment Header Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24</td>
</tr>
<tr>
<td>Header Length</td>
<td>32</td>
</tr>
<tr>
<td>Flags</td>
<td>33</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>34</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>35</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>36</td>
</tr>
<tr>
<td>Options</td>
<td>37</td>
</tr>
</tbody>
</table>

- **16-bit urgent data pointer**
  - If URG = 1
  - Index of last byte of urgent data in segment
TCP Options

- Negotiate maximum segment size (MSS)
  - Each host suggests a value
  - Minimum of two values is chosen
  - Prevents IP fragmentation over first and last hops

- Packet timestamp
  - Allows RTT calculation for retransmitted packets
  - Extends sequence number space for identification of stray packets

- Negotiate advertised window granularity
  - Allows larger windows
  - Good for routes with large bandwidth-delay products
# TCP State Descriptions

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>Disconnected</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Waiting for incoming connection</td>
</tr>
<tr>
<td>SYN_RCVD</td>
<td>Connection request received</td>
</tr>
<tr>
<td>SYN_SENT</td>
<td>Connection request sent</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>Connection ready for data transport</td>
</tr>
<tr>
<td>CLOSE_WAIT</td>
<td>Connection closed by peer</td>
</tr>
<tr>
<td>LAST_ACK</td>
<td>Connection closed by peer, closed locally, await ACK</td>
</tr>
<tr>
<td>FIN_WAIT_1</td>
<td>Connection closed locally</td>
</tr>
<tr>
<td>FIN_WAIT_2</td>
<td>Connection closed locally and ACK’ d</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Connection closed by both sides simultaneously</td>
</tr>
<tr>
<td>TIME_WAIT</td>
<td>Wait for network to discard related packets</td>
</tr>
</tbody>
</table>
TCP State Transition Diagram
TCP State Transition Diagram

- **CLOSED**
  - Passive open
  - Close

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

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

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

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

- **FIN_WAIT_2**
  - FIN/ACK
  - FIN + ACK/ACK

- **CLOSING**
  - ACK

- **TIME_WAIT**
  - FIN/ACK

- **CLOSE_WAIT**
  - Close/FIN
  - LAST_ACK

- **LAST_ACK**
  - ACK

- **CLOSED**
  - Timeout

- **Active Close**

- **Passive Close**

- **Active open/SYN**

- **Timeout**
TCP State Transition Diagram

- Passive open
- Active open
- SYN open
- SYN + ACK
- Send/ACK
- FIN/ACK
- Close
- Close/FIN
- FIN/ACK
- ACK
- ACK
- Time out
- FIN/ACK
- ACK
- FIN/ACK
- SYN/ACK
- SYN/SYN + ACK
- SYN/ACK
- ACK
- FIN/ACK
- FIN/ACK
- FIN/SYN + ACK
- FIN/SYN + ACK
- SYN/SYN + ACK
TCP State Transition Diagram

- **CLOSED**
  - Passive open
  - Close
  - Reset after SYN/ACK was sent

- **LISTEN**
  - SYN_RCVD
    - Passive open
    - Send/ SYN

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

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

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

- **FIN_WAIT_2**
  - FIN/ ACK
  - FIN + ACK/ ACK

- **CLOSING**
  - ACK
  - FIN/ ACK
  - Timeout

- **TIME_WAIT**
  - FIN/ ACK

- **CLOSE_WAIT**
  - Close/ FIN

- **LAST_ACK**
  - ACK

- **CLOSED**
  - ACK

- Active open/ SYN
TCP State Transition Diagram

Questions

State transitions

- Describe the path taken by a server under normal conditions
- Describe the path taken by a client under normal conditions
- Describe the path taken assuming the client closes the connection first
TCP State Transition Diagram

Establishment under normal conditions

- Passive open
- SYN/SYN + ACK
- ACK
- Close/FIN
- SYN_RCVD

- Close
- SYN/SYN + ACK
- LISTEN

- Send/SYN
- SYN_SENT

- SYN + ACK/ACK
- Active open/SYN

- FIN/ACK
- ESTABLISHED

- FIN/ACK
- CLOSING

- FIN + ACK/ACK
- TIME_WAIT

- FIN/ACK
- CLOSE_WAIT

- Close/FIN
- LAST_ACK

- Close/FIN
- CLOSED

- Timeout
- ACK

- FIN/ACK
- FIN_WAIT_2

- FIN/ACK
- FIN_WAIT_1

- Close/FIN
- CLOSED
TCP State Transition Diagram

Lost ACK from receiver?

- Passive open
- SYN/SYN + ACK
- Active open/SYN
- SYN_SENT
- SYN/SYN + ACK
- Send/SYN
- SYN_RCVD
- Close/FIN
- FIN_WAIT_1
- FIN_WAIT_2
- FIN/ACK
- Close/FIN
- FIN/ACK
- CLOSING
- FIN/ACK
- ACK
- TIME_WAIT
- FIN/ACK
- Close/FIN
- LAST_ACK
- ACK
- CLOSED
- Timeout
- Close/FIN

Lost ACK from receiver?
TCP State Transition Diagram
TCP State Transition Diagram

Timeouts?

CLOSED

LISTEN

SYN_SENT

SYN_RCVD

FIN_WAIT_1

FIN_WAIT_2

CLOSING

TIME_WAIT

CLOSE_WAIT

LAST_ACK

CLOSED

Passive open

Send/SYN

Close

Close

SYN/SYN + ACK

ACK

SYN/SYN + ACK

SYN/SYN + ACK

If no response after multiple tries, return to CLOSED

Timeout
TCP State Transition Diagram

One side closes first
TCP TIME_WAIT State

- What purpose does the TIME_WAIT state serve?
- Problem
  - What happens if a segment from an old connection arrives at a new connection?
- Maximum Segment Lifetime
  - Max time an old segment can live in the Internet
- TIME_WAIT State
  - Connection remains in this state from two times the maximum segment lifetime
TCP State Transition Diagram

Both sides close at the same time
TCP State Transition Diagram

FIN_ACK received (rare)

- Passive open
- SYN/SYN + ACK
- SYN_RCVD
- Close/FIN
- SYN_RCVD
- SYN_SENT
- Send/SYN
- LISTEN
- SYN/SYN + ACK
- ESTABLISHED
- SYN/SYN + ACK
- FIN_WAIT_1
- ACK
- FIN_WAIT_2
- FIN + ACK/ACK
- FIN_WAIT_2
- FIN_WAIT_1
- FIN_ACK received
- FIN_ACK received
- Close/FIN
- FIN_WAIT_2
- FIN_WAIT_1
- CLOSING
- FIn/ACK
- ESTABLISHED
- FIN/ACK
- CLOSE_WAIT
- Close/FIN
- LAST_ACK
- ACK
- TIME_WAIT
- Close/FIN
- FIN/ACK
- CLOSE_WAIT
- ACK
- TIME_WAIT
- Timeout
- CLOSED
- Close/FIN
- FIN + ACK/ACK
- FIN ACK received
TCP Sliding Window Protocol

- Sequence numbers
  - Indices into byte stream

- ACK sequence number
  - Actually next byte expected as opposed to last byte received
TCP Sliding Window Protocol

- Initial Sequence Number
  - Why not just use 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
  - … and might be associated with new connection

- TCP requires (RFC793) changing ISN
  - Set from 32-bit clock that ticks every 4 microseconds
  - … only wraps around once every 4.55 hours

- To establish a connection, hosts exchange ISNs
**TCP Sliding Window Protocol**

- **Advertised window**
  - Enables dynamic receive window size

- **Receive buffers**
  - Data ready for delivery to application until requested
  - Out-of-order data to maximum buffer capacity

- **Sender buffers**
  - Unacknowledged data
  - Unsent data out to maximum buffer capacity
TCP Sliding Window Protocol – Sender Side

- \( \text{LastByteAcked} \leq \text{LastByteSent} \)
- \( \text{LastByteSent} \leq \text{LastByteWritten} \)
- Buffer bytes between \( \text{LastByteAcked} \) and \( \text{LastByteWritten} \)

Maximum buffer size

Advertised window

First unacknowledged byte

Data available, but outside window

Last byte sent
TCP Sliding Window Protocol – Receiver Side

- \( \text{LastByteRead} < \text{NextByteExpected} \)
- \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \)
- Buffer bytes between \( \text{NextByteRead} \) and \( \text{LastByteRcvd} \)

Maximum buffer size

Advertised window

Buffered, out-of-order data

Next byte expected (ACK value)

Next byte to be read by application
Flow Control vs. Congestion Control

- Flow control
  - Preventing senders from overrunning the capacity of the receivers

- Congestion control
  - Preventing too much data from being injected into the network, causing switches or links to become overloaded

Which one does TCP provide?

TCP provides both
  - Flow control based on advertised window
  - Congestion control discussed later in class
Advertised Window Limits

Rate

- $W = \text{window size}$
  - Sender can send no faster than $W/\text{RTT}$ bytes/sec
  - Receiver implicitly limits sender to rate that receiver can sustain
  - If sender is going too fast, window advertisements get smaller & smaller
TCP Flow Control: Receiver

- **Receive buffer size**
  - \[ = \text{MaxRcvBuffer} \]
  - \[ \text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuf} \]

- **Advertised window**
  - \[ = \text{MaxRcvBuf} - (\text{NextByteExp} - \text{NextByteRead}) \]
  - Shrinks as data arrives and
  - Grows as the application consumes data
TCP Flow Control: Sender

- Send buffer size
  - $\text{Send buffer size} = \text{MaxSendBuffer}$
  - $\text{LastByteSent - LastByteAcked} \leq \text{AdvertWindow}$

- Effective buffer
  - $\text{Effective buffer} = \text{AdvertWindow} - (\text{LastByteSent} - \text{LastByteAck})$
  - $\text{EffectiveWindow} > 0$ to send data

- Relationship between sender and receiver
  - $\text{LastByteWritten} - \text{LastByteAcked} \leq \text{MaxSendBuffer}$
  - block sender if ($\text{LastByteWritten} - \text{LastByteAcked} + y > \text{MaxSenderBuffer}$)
TCP Flow Control

- **Problem: Slow receiver application**
  - Advertised window goes to 0
  - Sender cannot send more data
  - Non-data packets used to update window
  - Receiver may not spontaneously generate update or update may be lost

- **Solution**
  - Sender periodically sends 1-byte segment, ignoring advertised window of 0
  - Eventually window opens
  - Sender learns of opening from next ACK of 1-byte segment
TCP Flow Control

Problem: Application delivers tiny pieces of data to TCP
- Example: telnet in character mode
- Each piece sent as a segment, returned as ACK
- Very inefficient

Solution
- Delay transmission to accumulate more data
- Nagle’s algorithm
  - Send first piece of data
  - Accumulate data until first piece ACK’d
  - Send accumulated data and restart accumulation
- Not ideal for some traffic (e.g., mouse motion)
TCP Flow Control

Problem: Slow application reads data in tiny pieces
- Receiver advertises tiny window
- Sender fills tiny window
- Known as silly window syndrome

Solution
- Advertise window opening only when MSS or \( \frac{1}{2} \) of buffer is available
- Sender delays sending until window is MSS or \( \frac{1}{2} \) of receiver’s buffer (estimated)
TCP Bit Allocation Limitations

- Sequence numbers vs. packet lifetime
  - Assumed that IP packets live less than 60 seconds
  - Can we send $2^{32}$ bytes in 60 seconds?
  - Less than an STS-12 line

- Advertised window vs. delay-bandwidth
  - Only 16 bits for advertised window
  - Cross-country RTT = 100 ms
  - Adequate for only 5.24 Mbps!
## TCP Sequence Numbers – 32-bit

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Speed</th>
<th>Time until wrap around</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1.5 Mbps</td>
<td>6.4 hours</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10 Mbps</td>
<td>57 minutes</td>
</tr>
<tr>
<td>T3</td>
<td>45 Mbps</td>
<td>13 minutes</td>
</tr>
<tr>
<td>FDDI</td>
<td>100 Mbps</td>
<td>6 minutes</td>
</tr>
<tr>
<td>STS-3</td>
<td>155 Mbps</td>
<td>4 minutes</td>
</tr>
<tr>
<td>STS-12</td>
<td>622 Mbps</td>
<td>55 seconds</td>
</tr>
<tr>
<td>STS-24</td>
<td>1.2 Gbps</td>
<td>28 seconds</td>
</tr>
</tbody>
</table>
### TCP Advertised Window – 16-bit

<table>
<thead>
<tr>
<th>Bandwidth</th>
<th>Speed</th>
<th>Delay x Bandwidth Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>1.5 Mbps</td>
<td>18 KB</td>
</tr>
<tr>
<td>Ethernet</td>
<td>10 Mbps</td>
<td>122 KB</td>
</tr>
<tr>
<td>T3</td>
<td>45 Mbps</td>
<td>549 KB</td>
</tr>
<tr>
<td>FDDI</td>
<td>100 Mbps</td>
<td>1.2 MB</td>
</tr>
<tr>
<td>STS-3</td>
<td>155 Mbps</td>
<td>1.8 MB</td>
</tr>
<tr>
<td>STS-12</td>
<td>622 Mbps</td>
<td>7.4 MB</td>
</tr>
<tr>
<td>STS-24</td>
<td>1.2 Gbps</td>
<td>14.8 MB</td>
</tr>
</tbody>
</table>
Reasons for Retransmission

- Packet Timeout
- Packet ACK Timeout
- Packet ACK Lost
- Packet 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 Round Trip Time and Timeout

How should TCP set its timeout value?
- Longer than RTT
  - But RTT varies
- Too short
  - Premature timeout
  - Unnecessary retransmissions
- Too long
  - Slow reaction to segment loss

Estimating RTT
- SampleRTT
  - Measured time from segment transmission until ACK receipt
  - Will vary
  - Want smoother estimated RTT
- Average several recent measurements
  - Not just current SampleRTT
TCP Adaptive Retransmission Algorithm - Original

- **Theory**
  - Estimate RTT
  - Multiply by 2 to allow for variations

- **Practice**
  - Use exponential moving average ($\alpha = 0.1$ to 0.2)
  - Estimate = ($\alpha$) * measurement + (1- $\alpha$) * estimate
  - Influence of past sample decreases exponentially fast
Problem: What does an ACK really ACK?

- Was ACK in response to first, second, etc transmission?
TCP Adaptive Retransmission Algorithm – Karn-Partridge

- Algorithm
  - Exclude retransmitted packets from RTT estimate
  - For each retransmission
    - Double RTT estimate
    - Exponential backoff from congestion
TCP Adaptive Retransmission Algorithm – Karn-Partridge

Problem

- Still did not handle variations well
- Did not solve network congestion problems as well as desired
  - At high loads round trip variance is high
Example RTT Estimation

- SampleRTT
- Estimated RTT

RTT (milliseconds)

time (seconds)
TCP Adaptive Retransmission Algorithm – Jacobson

**Algorithm**
- **Estimate variance of RTT**
  - Calculate mean interpacket RTT deviation to approximate variance
  - Use second exponential moving average
  - \[ \text{Dev} = (\beta) \times |\text{RTT} - \text{Sample}| + (1-\beta) \times \text{Dev} \]
  - \( \beta = 0.25, A = 0.125 \) for RTT_est
- **Use variance estimate as component of RTT estimate**
  - \( \text{Next}_\text{RTT} = \text{RTT} + 4 \times \text{Dev} \)
- **Protects against high jitter**
TCP Adaptive Retransmission Algorithm – Jacobson

Notes
- Algorithm is only as good as the granularity of the clock
- Accurate timeout mechanism is important for congestion control
Evolution of TCP

- **1975**
  - Three-way handshake
  - Raymond Tomlinson
  - In SIGCOMM 75

- **1974**
  - TCP described by Vint Cerf and Bob Kahn
  - In IEEE Trans Comm

- **1975**
  - TCP

- **1982**
  - TCP & IP
  - RFC 793 & 791

- **1983**
  - BSD Unix 4.2
  - supports TCP/IP

- **1984**
  - Nagel’s algorithm
  - to reduce overhead of small packets; predicts congestion collapse

- **1985**
  - Congestion collapse observed

- **1986**
  - Karn’s algorithm
  - to better estimate round-trip time

- **1987**
  - Karn’s algorithm
  - congestion avoidance and congestion control
  - (most implemented in 4.3BSD Tahoe)

- **1988**
  - Van Jacobson’s algorithms
  - fast retransmit delayed ACK’s

- **1990**
  - 4.3BSD Reno
  - fast retransmit delayed ACK’s
TCP Through the 1990s

1993
TCP Vegas (Brakmo et al)
delay-based congestion avoidance

1994
ECN (Floyd)
Explicit Congestion Notification

1996
SACK TCP (Floyd et al)
Selective Acknowledgement

1996
Hoe
NewReno startup and loss recovery

And beyond:
TCP in challenged (e.g. wireless) conditions;
faster flow completion; lower latency; “incast” problem; …