TCP Internals
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

```plaintext
SYN K, acknowledge (ACK) J+1
ACK K+1
```

Time flows down

listen
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?
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

 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

TCP

Send buffer

TCP Segment

TCP Segment

TCP Segment

Application process

Read bytes

TCP

Recv buffer

TCP Segment

TCP Segment

TCP Segment
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

Time flows down
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td></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>
## 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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Header Length</td>
<td>0</td>
<td>Flags</td>
<td>Advertised Window</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Options</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Position</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>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>8-15</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>16-23</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>32-39</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>32-39</td>
</tr>
<tr>
<td>Options</td>
<td>32-39</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
    - $X$, $X+1$, $X+2$, …, $X+B-1$
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)

---

B bytes

byte X+B
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
### 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>

- **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>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-8</td>
</tr>
<tr>
<td>Destination Port</td>
<td>9-16</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>17-23</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-30</td>
</tr>
<tr>
<td>Header Length</td>
<td>31</td>
</tr>
<tr>
<td>Flags</td>
<td>32</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>33</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>34</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>35</td>
</tr>
<tr>
<td>Options</td>
<td>36-31</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Length</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>16</td>
<td>Source port of the process sending the data</td>
</tr>
<tr>
<td>Destination Port</td>
<td>16</td>
<td>Destination port of the process receiving the data</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>32</td>
<td>Sequence number of the data being sent</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>32</td>
<td>Sequence number of the expected data to receive</td>
</tr>
<tr>
<td>Header Length</td>
<td>4</td>
<td>Length of the TCP header in 32-bit words</td>
</tr>
<tr>
<td>Flags</td>
<td>4</td>
<td>6 1-bit flags: URG, ACK, PSH, RST, SYN, FIN</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>16</td>
<td>Advertised window size for the receiver</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>4</td>
<td>Pointer to urgent data in the data segment</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>4</td>
<td>Checksum of the TCP header and data</td>
</tr>
<tr>
<td>Options</td>
<td>Variable</td>
<td>Additional options for the connection</td>
</tr>
</tbody>
</table>

### 6 1-bit flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>URG</td>
<td>Contains urgent data</td>
</tr>
<tr>
<td>ACK</td>
<td>Valid ACK seq. number</td>
</tr>
<tr>
<td>PSH</td>
<td>Do not delay data delivery</td>
</tr>
<tr>
<td>RST</td>
<td>Reset connection</td>
</tr>
<tr>
<td>SYN</td>
<td>Synchronize for setup</td>
</tr>
<tr>
<td>FIN</td>
<td>Final segment for teardown</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>16-23</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>8</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>16</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>31</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>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source Port</td>
<td>0-7</td>
<td>Port number of the source application</td>
</tr>
<tr>
<td>Destination Port</td>
<td>8-15</td>
<td>Port number of the destination application</td>
</tr>
<tr>
<td>Sequence Number</td>
<td>16-23</td>
<td>Sequence number of the segment</td>
</tr>
<tr>
<td>ACK Sequence Number</td>
<td>24-31</td>
<td>Acknowledgment number of the last byte sent</td>
</tr>
<tr>
<td>Header Length</td>
<td>0-4</td>
<td>Length of the header in 32-bit units</td>
</tr>
<tr>
<td>Flags</td>
<td>5-15</td>
<td>Flags indicating various TCP options</td>
</tr>
<tr>
<td>Advertised Window</td>
<td>16-20</td>
<td>Window advertised by the receiver</td>
</tr>
<tr>
<td>TCP Checksum</td>
<td>21-31</td>
<td>Checksum of the TCP header</td>
</tr>
<tr>
<td>Urgent Pointer</td>
<td>32-39</td>
<td>16-bit urgent data pointer</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

- **CLOSED**: Passive open, Close
- **LISTEN**: SYN/SYN + ACK, Send/SYN
- **SYN_RCVD**: SYN/SYN + ACK, ACK
- **ESTABLISHED**: SYN + ACK/ACK, FIN/ACK
- **FIN_WAIT_1**: FIN/ACK, ACK
- **FIN_WAIT_2**: FIN + ACK/ACK
- **CLOSING**: ACK
- **CLOSE_WAIT**: Close/FIN
- **LAST_ACK**: ACK
- **TIME_WAIT**: Timeout
- **CLOSED**: FIN/ACK, Close/FIN
TCP State Transition Diagram
TCP State Transition Diagram

- Passive open
  - SYN/SYN + ACK
  - ACK

- Active open/SYN

- Messages from local application:
  - Passive open/SYN + ACK
  - Close/FIN
  - Close/ACK
  - FIN/ACK

- Messages from receiver:
  - Event from local application:
    - SYN/SYN + ACK
    - SYN + ACK/ACK
    - FIN/ACK
    - FIN/ACK
    - CLOSE_WAIT
    - LAST_ACK
    - CLOSED

- Event from local application:
  - Message sent
  - Response sent

- Timeouts:
  - Timeout
  - ACK
TCP State Transition Diagram

Reset after SYN/ACK was sent

- Passive open
- Close
- Send/SYN
- SYN/SYN + ACK
- SYN/SYN + ACK
- SYN + ACK/ACK
- FIN/ACK
- CLOSE_WAIT
- LAST_ACK
- CLOSED
- FIN/ACK
- FIN + ACK/ACK
- FIN_WAIT_1
- FIN_WAIT_2
- CLOSING
- TIME_WAIT
- TIMEOUT
- ACK
- FIN/ACK
- RST
- Close/FIN
- Close/ACK
- SYN/SYN + ACK
- Close/FIN
- SYN_SENT
- ESTABLISHED
- CLOSED
- STOP
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

- Closed
- LISTEN
- SYN_SENT
- ESTABLISHED
- FIN_WAIT_1
- FIN_WAIT_2
- CLOSING
- TIME_WAIT
- CLOSED
- CLOSE_WAIT
- LAST_ACK
- LAST_ACK

States:
- Passive open
- SYN/SYN + ACK
- ACK
- FIN/ACK
- FIN/ACK
- FIN/ACK
- FIN/ACK
- SYN/SYN + ACK
- SYN + ACK/ACK
- Active open/SYN
- Close
- Close/FIN
- Close/FIN
- Close/FIN
- Close/FIN
- Close/FIN
- Close/FIN
- Close/FIN

Events:
- Send/SYN
- FIN/ACK
- FIN + ACK/ACK
- Timeout
- ACK
- ACK
- ACK
- ACK

Establishment under normal conditions
TCP State Transition Diagram

Lost ACK from receiver?

CLOSED
- Passive open
- Close
- Close

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

SYN_SENT
- Active open/SYN
- SYN/ACK/ACK

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

FIN_WAIT_1
- ACK
- FIN/Ack

FIN_WAIT_2
- FIN/ACK

ESTABLISHED
- ACK
- FIN/ACK

CLOSING
- FIN/ACK
- FIN + ACK/ACK

TIME_WAIT
- ACK
- FIN/ACK

CLOSE_WAIT
- Close/FIN

LAST_ACK
- ACK
- Timeout

CLOSED
TCP State Transition Diagram

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

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

- **SYN_RCVD**
  - Close/FIN
  - ACK
  - Never used

- **FIN_WAIT_1**
  - FIN/ACK

- **FIN_WAIT_2**
  - FIN/ACK

- **CLOSING**
  - FIN + ACK/ACK
  - ACK

- **TIME_WAIT**
  - FIN/ACK

- **CLOSE_WAIT**
  - Close/FIN

- **LAST_ACK**
  - ACK

- **CLOSED**
  - Timeout

Local send when in LISTEN
TCP State Transition Diagram

Timeouts?

**CLOSED**
- Passive open
- Close

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

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

**FIN_WAIT_1**
- ACK

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

**CLOSING**
- ACK

**TIME_WAIT**
- Timeout

**CLOSE_WAIT**
- Close/FIN

**LAST_ACK**
- ACK

**CLOSED**
- Active open/SYN

If no response after multiple tries, return to CLOSED
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

- **CLOSED**
  - Passive open
  - Close

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

- **SYN_RCVD**
  - SYN/SYN + ACK
  - ACK

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

- **FIN_WAIT_1**
  - FIN/ACK
  - FIN_WAIT_2
  - ACK

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

- **CLOSING**
  - ACK

- **ESTABLISHED**
  - FIN/ACK

- **CLOSE_WAIT**
  - Close/FIN

- **LAST_ACK**
  - ACK

- **TIME_WAIT**
  - Timeout

- **CLOSED**
  - FIN_ACK received (rare)

Active open/SYN

- Close
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

- $\text{LastByteAacked} \leq \text{LastByteSent}$
- $\text{LastByteSent} \leq \text{LastByteWritten}$
- Buffer bytes between $\text{LastByteAacked}$ 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 $\text{LastByteRcvd}$
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
  - $\text{Receive buffer size} = \text{MaxRcvBuffer}$
  - $\text{LastByteRcvd} - \text{LastByteRead} \leq \text{MaxRcvBuf}$

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

- **Send buffer size**
  - $\text{Send buffer size} = \text{MaxSendBuffer}$
  - $\text{LastErrorSent} - \text{LastErrorAcked} \leq \text{AdvertWindow}$

- **Effective buffer**
  - $\text{Effective buffer} = \text{AdvertWindow} - (\text{LastErrorSent} - \text{LastErrorAck})$
  - $\text{EffectiveWindow} > 0$ to send data

- **Relationship between sender and receiver**
  - $\text{LastErrorWritten} - \text{LastErrorAcked} \leq \text{MaxSendBuffer}$
  - block sender if $(\text{LastErrorWritten} - \text{LastErrorAcked}) + y > \text{MaxSenderBuffer}$
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 ½ of buffer is available
  - Sender delays sending until window is MSS or ½ 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

1. Packet lost
2. ACK lost
3. Early timeout
4. Duplicate packets

Packet
ACK
Timeout
Packet lost
ACK
Timeout
ACK lost
Duplicate packet
Timeout
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 $= (\alpha) \times$ measurement + $(1-\alpha) \times$ estimate
  - Influence of past sample decreases exponentially fast
TCP Adaptive Retransmission Algorithm - Original

Problem: What does an ACK really ACK?
- Was ACK in response to first, second, etc transmission?

Sample RTT

Original transmission

retransmission

ACK

Original transmission

retransmission

ACK

Sample RTT
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
      - \( \text{Dev} = (\beta) \times |\text{RTT}_{\text{Est}} - \text{Sample}| + (1-\beta) \times \text{Dev} \)
      - \( \beta = 0.25, A = 0.125 \) for \( \text{RTT}_{\text{est}} \)
  - Use variance estimate as component of RTT estimate
    - \( \text{Next}_\text{RTT} = \text{RTT}_{\text{Est}} + 4 \times \text{Dev} \)
  - 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

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
1986
Van Jacobson’s algorithms congestion avoidance and congestion control (most implemented in 4.3BSD Tahoe)

1987
Karn’s algorithm to better estimate round-trip time

1988
1990
4.3BSD Reno fast retransmit delayed ACK’s

1980
1985
Congestion collapse observed

1975
1980
1985
1990
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; …