Chapter 3
Transport Layer

Computer Networking: A Top Down Approach
5th edition.
Jim Kurose, Keith Ross
Addison-Wesley, April 2009.

Some of the slides taken from the course text book
Transport services and protocols

- provide *logical communication* between app processes running on *different hosts*

- transport protocols run in *hosts*
  - send side: breaks app messages into *segments*, passes to network layer
  - rcv side: reassembles segments into *messages*, passes to app layer

- more than one *transport protocol* available to apps
  - Internet: TCP and UDP
Transport vs. network layer

- **network layer**: logical communication between hosts

- **transport layer**: logical communication between processes at different hosts
  - relies on network layer services

---

**Household analogy:**

- 12 kids sending letters to 12 kids
  - processes = kids
  - app messages = letters in envelopes
  - hosts = houses
  - transport protocol = Alice and Bob who demux to in-house siblings
  - network-layer protocol = postal service
Internet transport-layer protocols

- **reliable, in-order delivery**
  - Transmission Control Protocol (TCP)
    - congestion control
    - flow control
    - connection setup

- **unreliable, unordered delivery**: User Datagram Protocol (UDP)
  - no-frills extension of “best-effort” IP

- **services not available**: Delay Guarantees, bandwidth guarantees, Cannot Control Network Cores
Multiplexing/demultiplexing

Demultiplexing at rcv host:
- delivering received segments to correct socket

Multiplexing at send host:
- gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

= socket  = process

```
+-----------------+          +-----------------+          +-----------------+
| application |       | application |       | application |
|   P3        |       |   P1         |       |   P4         |
| transport   |       | transport    |       | transport    |
| network     |       | network      |       | network      |
| link        |       | link         |       | link         |
| physical    |       | physical     |       | physical     |

host 1          host 2          host 3
```
How demultiplexing works

- **host receives IP datagrams**
  - each datagram has
    - source IP address,
    - destination IP address
  - each datagram carries one transport-layer segment
  - each segment has
    - source port number
    - destination port number

- **host uses IP addresses & port numbers to direct segment to appropriate socket**

TCP/UDP segment format:
- 32 bits
- source port #
- dest port #
- other header fields
- application data
- (message)
Connectionless demultiplexing

- **recall**: create sockets with host-local port numbers:
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  DatagramSocket mySocket2 = new DatagramSocket(12535);
  ```

- **recall**: when creating datagram to send into UDP socket, must specify 
  
  (dest IP address, dest port number)

- when host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
DatagramSocket serverSocket = new DatagramSocket(6428);

SP provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

- recv host uses all four values to direct segment to appropriate socket

- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request
Connection-oriented demux:

- Client IP: A
  - Destination IP: C
  - Source Port: 9157
  - Destination Port: 80

- Server IP: C
  - Source IP: B
  - Source Port: 5775
  - Destination Port: 80

- Client IP: B
  - Destination IP: C
  - Source Port: 9157
  - Destination Port: 80
UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender & receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender & receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired

Any Applications?
UDP: more

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive
- Also used for
  - DNS
  - SNMP
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

UDP segment format:

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Application data (message)

Length, in bytes of UDP segment, including header
**UDP checksum**

**Goal:** detect “errors” (e.g., flipped bits) in transmitted segment

**Sender:**
- treat segment contents as sequence of 16-bit integers
- checksum: addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

**Receiver:**
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected.
  *But maybe errors nonetheless? More later....*
Internet Checksum Example

- Note: when adding numbers, a carryout from the most significant bit needs to be added to the result

- Example: add two 16-bit integers

```
1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
________________________
wraparound 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1
sum 1 0 1 1 1 0 1 1 1 0 1 1 1 1 0 0
checksum 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```
Internet Checksum Example

- Complement of the sum is stored in the checksum field
- At the receiver, all the byte fields are added along with the checksum
- Sum + checksum must be all 1s
- No error else discard packet
- UDP checksum is optional in IPv4
- UDP checksum is mandatory in IPv6
Reliable Data Transfer

- Problem: Reliability
  - Want an abstraction of a reliable link even though packets can be corrupted or get lost

- Where can packets be corrupted or lost?
  - In the network
  - At the receiver

- Solution: keep track of the packets
  - Not as simple as one would expect
Reliable transmission & Flow Control

- What to do when there is a packet loss?
  - On the link (in the network)
  - At the receiver (buffer overflow)

- Need to recoup losses

- What happens if the packet is lost in the network?
  - A random event, retransmit

- What happens if the sender tries to transmit faster than the receiver can accept?
  - Data will be lost unless flow control is implemented
Reliability support

- Sender needs to know if a packet was lost
- How?
  - Acknowledgement
  - Positive ack and negative ack
- Sender needs to retransmit
- How?
  - Timeouts
  - Acks can also get lost
Some Flow Control Algorithms

- Flow control for the ideal network
- Stop and Wait for noiseless channels
- Stop and Wait for noisy channels
- Sliding window protocols
- Sliding window with error control
  - Go Back N
  - Selective Repeat
Flow control in the ideal network

Assumptions:
(1) Error free transmission link,
(2) Infinite buffer at the receiver

No acknowledgement of frames necessary
Since the data link is error-free and the receiver can buffer as many frames as it likes, no frame will ever be lost
Flow control in the ideal network (cont’d)
stop-and-wait Normal Operation

Packet Length = L; Bandwidth = R; RTT = 2 * Prop Delay

- first packet bit transmitted, $t = 0$
- last packet bit transmitted, $t = L / R$
- first packet bit arrives
- last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = RTT + L / R$
- Timeout

sender

receiver
stop-and-wait Packet Lost

sender

receiver

first packet bit arrives

last packet bit arrives, send ACK

Timeout

ACK
stop-and-wait Ack Lost

sender

receiver

Timeout

Packet retransmitted

ACK
stop-and-wait Delayed Ack

Timeout too short
Duplicate Transmission
stop-and-wait Detecting duplicates

sender

receiver

Timeout too short
Duplicate Transmission

Ack0

Ack1

0

1

Time Out
Performance of stop and wait

- example: 1 Gbps link, 1.5 ms e-e prop. delay, 1KB packet:

\[
T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8000 \text{ bits/pkt}}{10^{-9} \text{ b/sec}} = 8 \text{ microsec}
\]

\[
U_{\text{sender}} = \frac{L / R}{\text{RTT} + L / R} = \frac{0.008}{3.008} = 0.0027
\]

- \(U_{\text{sender}}\): utilization - fraction of time sender busy sending
- 1KB pkt every 3 msec \(\rightarrow\) 330kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
Is Stop and Wait the best we can do?

Stop and Wait is an effective form of flow control, but...

It’s not very efficient.

1. Only one data frame can be in transit on the link at a time
2. When waiting for an acknowledgement, the sender cannot transmit any frames

Better solution? Sliding Window
Bandwidth delay product

- Continuously send data until first Ack
- How much? BW*RTT
- Known as Bandwidth delay product
- Number of packets \( N = \frac{BW \times RTT}{\text{Packet size}} \)
Pipelined protocols

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
Pipelining Example: increased utilization

Utilization \( U_{\text{sender}} \) – fraction of time sender busy sending

\[
U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{3.008} = 0.008
\]

Increase utilization by a factor of 3!
Simple Sliding Window with Window Size of 1

A sliding window with a maximum window size of 1 frame

Window for a 3-bit sequence number
### Sliding Window example

**Sender window**

(a) Initial state, no frames transmitted, receiver expects frame 0
(b) Sender transmits frame 0, receiver buffers frame 0
(c) Receiver ACKS frame 0
(d) Sender receives ACK, removes frame 0
Simple Sliding Window with Window size 1 (cont’d)

This protocol behaves identically to stop and wait for a noisy channel
Sliding Window Protocols

Sender Definitions

**Sequence Number**: Each frame is assigned a sequence number that is incremented as each frame is transmitted.

**Sender Window**: Keeps track of sequence numbers for frames that have been sent but not yet acknowledged.

**Sender Window size**: The maximum number of frames the sender may transmit without receiving any acknowledgements.

**Last Frame sent**: The highest sequence number frames sent.

**Last Ack received**: sequence number of last ACK.
Sliding Window with Maximum Sender Window Size $SWS$

With a maximum window size of $SWS$, the sender can transmit up to $SWS$ frames before "being blocked. This allows the sender to transmit several frames before waiting for an acknowledgement.

$LFS - LAR \leq SWS$
Sender-Side Window with $W_S=2$

(a) Initial window state
(b) Send frame 0
(c) Send frame 1
(d) ACK for frame 0 arrives
(e) Send frame 2
(f) ACK for frame 1 arrives
(g) ACK for frame 2 arrives, send frame 3
(h) ACK for frame 3 arrives
Receiver-Side Window with $W_R = 2$

(a) Initial window state
(b) Nothing happens
(c) Frame 0 arrives, ACK frame 0
(d) Nothing happens
(e) Frame 1 arrives, ACK frame 1
(f) Frame 2 arrives, ACK frame 2
(g) Nothing happens
(h) Frame 3 arrives, ACK frame 3
Sliding Window Protocols

Receiver Definitions

Receiver Window: Keeps track of sequence numbers for frames the receiver is allowed to accept.

Receiver Window size: The maximum number of frames the receiver may receive before returning an acknowledgement to the sender.

Largest Frame received: The highest in-sequence numbered frame received so far.

Largest Acceptable Frame: Highest sequence number acceptable.
Sliding Window with Maximum Receiver Window Size $\text{RWS}$

With a maximum window size of $\text{RWS}$, the receiver rejects packets if $\text{SeqNum} \leq \text{LFR}$ or $\text{SeqNum} > \text{LAF}$

Why? Outside the window

$LFR$ (Last Frame Received) $\text{LAF}$ (Last ACK Sent)

$LAF - LFR \leq \text{RWS}$

Receiver acknowledges seqnumber s.t all earlier frames have been received
What about Errors?

What if a data or acknowledgement frame is lost when using a sliding window protocol?

Two Solutions:

- Go Back N
- Selective Repeat
Pipelined protocols - how?

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

❖ range of sequence numbers must be increased
❖ buffering at sender and/or receiver

❖ Two generic forms of pipelined protocols: go-Back-N, selective repeat
Sliding Window with Go Back N

- When the receiver notices a missing or erroneous frame, it simply discards all frames with greater sequence numbers and sends no ACK.

- The sender will eventually time out and retransmit all the frames in its sending window.
Go Back N

Sender
Maximum window size = 8

Receiver
Maximum window size = 8

Frame with error

Timeout interval

Discarded by receiver

Time
Go-Back-N

Sender:
- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

ACK(n): ACKs all pkts up to seq # n - “cumulative ACK”

timeout(n): retransmit pkt n and all higher seq # pkts in window

One timer for all in-flight pkts
Go Back N (cont’d)

Go Back N can recover from erroneous or missing frames

But...

It is wasteful. If there are errors, the sender will spend time retransmitting frames the receiver has already seen
Sliding Window with Selective Repeat

The sender retransmits only the frame with errors

- The receiver stores all the correct frames that arrive following the bad one. (Note that the receiver requires a frame buffer for each sequence number in its receiver window.)

- When the receiver notices a skipped sequence number, it keeps acknowledging the last good sequence number.

- When the sender times out waiting for an acknowledgement, it just retransmits the one unacknowledged frame, not all its successors.
Selective Repeat

Sender
Maximum window size = 8

Receiver
Maximum window size = 8

Timeout interval

Frame with error

Buffered by receiver

Time
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
TCP

- TCP provides the end-to-end reliable connection that IP alone cannot support

- The protocol
  - Connection management
  - Retransmission
  - Flow control
  - Congestion control
  - Frame format
Connection Management
TCP Connection Establishment

- Three-way Handshake
TCP Connection Tear-down

- Two double handshakes:
Retransmission
TCP Retransmission

- When a packet remains unacknowledged for a period of time, TCP assumes it is lost and retransmits it.
- TCP tries to calculate the round trip time (RTT) for a packet and its acknowledgement.
- From the RTT, TCP can guess how long it should wait before timing out.
Round Trip Time (RTT)

RTT = Time for packet to arrive at destination
+ Time for ACK to return from destination
RTT Calculation

\[
\text{RTT} = 2.2 \text{ sec} - 0.9 \text{ sec.} = 1.3 \text{ sec}
\]
Smoothing the RTT measurement

First, we must smooth the round trip time due to variations in delay within the network:

\[ SRTT = \alpha SRTT + (1-\alpha) RTT_{\text{arriving ACK}} \]

- The smoothed round trip time (SRTT) weights previously received RTTs by the \( \alpha \) parameter
- \( \alpha \) is typically equal to 0.875
Retransmission Timeout Interval (RTO)

- The timeout value is then calculated by multiplying the smoothed RTT by some factor (greater than 1) called $\beta$

\[
\text{Timeout} = \beta \times \text{SRTT}
\]

- This coefficient of $\beta$ is included to allow for some variation in the round trip times.
Example

Initial SRTT = 1.50
\( \alpha = 0.875, \beta = 4.0 \)

<table>
<thead>
<tr>
<th>RTT Meas.</th>
<th>SRTT</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5 s</td>
<td>= 1.50</td>
<td>= ( \beta \times 1.50 = 6.00 )</td>
</tr>
<tr>
<td>1.0 s</td>
<td>= 1.50(\times \alpha + 1.0\times(1-\alpha) = 1.44 )</td>
<td>= ( \beta \times 1.44 = 5.76 )</td>
</tr>
<tr>
<td>2.2 s</td>
<td>= 1.44(\times \alpha + 2.2\times(1-\alpha) = 1.54 )</td>
<td>= ( \beta \times 1.54 = 6.16 )</td>
</tr>
<tr>
<td>1.0 s</td>
<td>= 1.54(\times \alpha + 1.0\times(1-\alpha) = 1.47 )</td>
<td>= ( \beta \times 1.47 = 5.88 )</td>
</tr>
<tr>
<td>0.8 s</td>
<td>= 1.47(\times \alpha + 0.8\times(1-\alpha) = 1.39 )</td>
<td>= ( \beta \times 1.39 = 5.56 )</td>
</tr>
<tr>
<td>3.1 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0 s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem with RTT Calculation

Sender Timeout

RTT?  

RTT?  

RTT?
Karn’s Algorithm

- Retransmission ambiguity
  - Measure RTT from original data segment
  - Measure RTT from most recent segment
- Either way there is a problem in RTT estimate
- One solution
  - Never update RTT measurements based on acknowledgements from retransmitted packets
- Problem: Sudden change in RTT can cause system never to update RTT
  - Primary path failure leads to a slower secondary path
Karn’s algorithm

- Use back-off as part of RTT computation
- Whenever packet loss, RTO is increased by a factor
- Use this increased RTO as RTO estimate for the next segment (not from SRTT)
- Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from SRTT
Another Problem with RTT Calculation

- RTT measurements can sometimes fluctuate severely
  - smoothed RTT (SRTT) is not a good reflection of round-trip time in these cases
- Solution: Use Jacobson/Karels algorithm:

\[
\text{Error} = \text{RTT} - \text{SRTT}
\]
\[
\text{SRTT} = \text{SRTT} + (\alpha \times \text{Error})
\]
\[
\text{Dev} = \text{Dev} + h(|\text{Error}| - \text{Dev})
\]
\[
\text{Timeout} = \text{SRTT} + (\beta \times \text{Dev})
\]
Jacobson/Karels Algorithm

Example

Initial SRTT = 1.50, Dev = 0
\(\alpha = 0.125, \delta = 0.25, \beta = 4.0\)

Error = RTT - SRTT
SRTT = SRTT + (\(\alpha \times \) Error)
Dev = Dev + [\(\delta \times (|\text{Error}| - \text{Dev})\)]
Timeout = SRTT + (\(\beta \times \text{Dev}\))

<table>
<thead>
<tr>
<th>RTT Meas.</th>
<th>Error</th>
<th>SRTT</th>
<th>Dev.</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.5 s</td>
<td>= 0.0</td>
<td>= 1.50</td>
<td>= 0.00</td>
<td>= 1.50</td>
</tr>
<tr>
<td>1.0 s</td>
<td>= -0.50</td>
<td>= 1.44</td>
<td>= 0.13</td>
<td>= 1.94</td>
</tr>
<tr>
<td>2.2 s</td>
<td>= +0.76</td>
<td>= 1.54</td>
<td>= 0.28</td>
<td>= 2.67</td>
</tr>
<tr>
<td>1.0 s</td>
<td>= -0.54</td>
<td>= 1.47</td>
<td>= 0.35</td>
<td>= 2.85</td>
</tr>
<tr>
<td>0.8 s</td>
<td>= -0.67</td>
<td>= 1.39</td>
<td>= 0.43</td>
<td>= 3.09</td>
</tr>
<tr>
<td>3.1 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.0 s</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Flow Control
TCP Flow Control

- TCP uses a modified version of the sliding window
- In acknowledgements, TCP uses the “Window size” field to tell the sender how many bytes it may transmit
- TCP uses bytes, not packets, as sequence numbers
TCP Flow Control (cont’d)

Important information in TCP/IP packet headers

Send

- Number of bytes in packet (N)
- Sequence number of first data byte in packet (SEQ)

Recv

- ACK bit set
- Sequence number of next expected byte (ACK)
- Window size at the receiver (WIN)

Contained in IP header

- Contained in TCP header

ACK bit

Window size

Sequence number of first data byte in packet (SEQ)
Example TCP session

(1)remus:$ tcpdump -S host scully
Kernel filter, protocol ALL, datagram packet socket
tcpdump: listening on all devices

15:15:22.152339 eth0 > remus.4706 > scully.echo: S 1264296504:1264296504(0) win 32120 <mss 1460,sack OK,timestamp 71253512 0,nop,wscale 0>
15:15:22.153865 eth0 < scully.echo > remus.4706: S 875676030:875676030(0) ack 1264296505 win 8760 <mss 1460>
15:15:22.153912 eth0 > remus.4706 > scully.echo: . 1264296505:1264296505(0) ack 875676031 win 32120

remus: telnet scully 7
A <return>
A
Example TCP session

Packet 1: 15:15:22.152339 eth0 > remus.4706 > scully.echo: S
1264296504:1264296504(0) win 32120 <mss 1460,sackOK,timestamp
71253512 0,nop,wscale 0> (DF)

Packet 2: 15:15:22.153865 eth0 < scully.echo > remus.4706: S
875676030:875676030(0) ack 1264296505 win 8760 <mss 1460)

1264296505:1264296505(0) ack 875676031 win 32120
TCP data transfer

Packet 4: 15:15:28.591716 eth0 > remus.4706 > scully.echo: P 1264296505:1264296508(3) ack 875676031 win 32120

data

Packet 5: 15:15:28.593255 eth0 < scully.echo > remus.4706: P 875676031:875676034(3) ack 1264296508 win 8760

# bytes
TCP Flow Control (cont’d)

Application does a 2K write

Sender

Sender is blocked

Sender may send up to 2K

Application does a 3K write

Receiver

Receiver’s buffer

0 4K

Empty

Full

Application reads 2K

2K

1K 2K
TCP Flow Control (cont’d)

Piggybacking: Allows more efficient bidirectional communication
Congestion Control
TCP Header Format

- Source Port
- Destination Port
- Sequence Number
- Acknowledgement number
- Window Size
- Checksum
- Urgent Pointer
- Options (0 or more 32-bit words)
- Data
TCP Header Fields

- **Source & Destination Ports**
  - 16 bit port identifiers for each packet

- **Sequence number**
  - The packet’s unique sequence ID

- **Acknowledgement number**
  - The sequence number of the next packet expected by the receiver
TCP Header Fields (cont’d)

- **Window size**
  - Specifies how many bytes may be sent after the first acknowledged byte

- **Checksum**
  - Checksums the TCP header and IP address fields

- **Urgent Pointer**
  - Points to urgent data in the TCP data field
TCP Header Fields (cont’d)

- Header bits
  - **URG** = Urgent pointer field in use
  - **ACK** = Indicates whether frame contains acknowledgement
  - **PSH** = Data has been “pushed”. It should be delivered to higher layers right away.
  - **RST** = Indicates that the connection should be reset
  - **SYN** = Used to establish connections
  - **FIN** = Used to release a connection
Principles of Congestion Control

Congestion:

- informally: “too many sources sending too much data too fast for network to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!
TCP Congestion Control

- Recall: Network layer is responsible for congestion control
- However, TCP/IP blurs the distinction
- In TCP/IP:
  - the network layer (IP) simply handles routing and packet forwarding
  - congestion control is done end-to-end by TCP
TCP Congestion Control

- **Goal:** fully (fairly) utilize the resource (bandwidth)
  - Don’t over use - congestion
  - Don’t under use - waste

- **Goal:** achieve self-clocking state
  - Even if don’t know bandwidth of bottleneck
  - Bottleneck may change over time
Self-Clocking Model

1. Send Burst
5. Send a data packet
4. Receive Acknowledgement
3. Send Acknowledgement
2. Receive data packet

Given: \( P_b = P_r = A_r = A_b = A_r \) (in units of time)
Sending a packet on each ACK keeps the bottleneck link busy
TCP Congestion Window

- TCP introduces a second window, called the “congestion window”

- This window maintains TCP’s best estimate of amount of outstanding data to allow in the network to achieve self-clocking

- Sending size = min(congestion control window, flow control window)
TCP Congestion Control

- Two phases to keep bottleneck busy (fully utilize the resource):
  - Increase the usage (window size) to keep probing the network
  - Decrease the usage when congestion is detected
TCP Slow Start

- When connection begins, $\text{CongWin} = 1 \text{ MSS}$
  - Example: MSS = 500 bytes

- available bandwidth may be $>> \text{MSS/RTT}$
  - desirable to quickly ramp up to respectable rate
  - Increase exponentially until first loss

MSS - “maximum segment size”, the maximum size a TCP packet can be (including header)
TCP Slow Start (more)

- incrementing CongWin for every ACK received
- double CongWin every RTT

initial rate is slow but ramps up exponentially fast
TCP Slow Start (cont’d)

- Congestion detection
  - Packet losses
  - Sender side Timeout

- Timeout
  - the congestion window is reduced to 1 MSS
  - everything starts over
TCP Slow Start (cont’d)

- Congestion window
- Timed out Transmissions
- Inefficient, Low throughput
TCP Linear Increase

- Don’t push the network too fast

- Slow start (exponential)
  - Threshold -> linear increase
TCP Linear Increase Algorithm

- **Algorithm:**
  - Start the threshold at 64K
  - Slow start
  - Once the threshold is passed
    - For each ack received, \( \text{cwnd} = \text{cwnd} + \frac{(\text{mss} \times \text{mss})}{\text{cwnd}} \)
      - 1 MSS for each congestion window of data transmitted
  - **Timeout**
    - reset the congestion window size to 1 MSS
    - Set threshold to \( \max(2 \times \text{mss}, \frac{1}{2} \times \text{MIN(} \text{sliding window, congestion window)}) \)
TCP Linear Increase Threshold Phase

Example: Maximum segment size = 1K
Assume thresh=32K

Timeout occurs when \( \text{MIN}(\text{sliding window, congestion window}) = 40K \)
TCP Congestion Control

- Can we do better at detecting congestion than using timeout?

- Receiver send duplicate ack for out-of-order packets
  - Possible loss?
TCP Fast Retransmit

- **Idea:** When sender sees 3 duplicate ACKs, it assumes something went wrong.

- The packet is immediately retransmitted instead of waiting for it to timeout.
TCP Fast Retransmit

Example

MSS = 1K

Fast Retransmit occurs (2nd packet is now retransmitted w/o waiting for it to timeout)
TCP Recap

- **Timeout Computation**
  - Timeout is a function of 2 values
    - the weighted average of sampled RTTs
    - The sampled variance of each RTT

- **Congestion control:**
  - **Goal:** Keep the self-clocking pipe full in spite of changing network conditions
  - **3 key Variables:**
    - Sliding window (Receiver flow control)
    - Congestion window (Sender flow control)
    - Threshold (Sender’s slow start vs. linear mode line)
TCP Recap (cont)

- **Slow start**
  - Add 1 segment for each ACK to the congestion window
    - Double’s the congestion window’s volume each RTT

- **Linear mode (Congestion Avoidance)**
  - Add 1 segment’s worth of data to each congestion window
  - Adds 1 segment per RTT
Algorithm Summary: TCP Congestion Control

- When CongWin is below Threshold, sender in slow-start phase, window grows exponentially.

- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.

- When a triple duplicate ACK occurs, Threshold set to \( \max(\text{FlightSize}/2, 2\times\text{mss}) \) and CongWin set to Threshold+3*mss. (Fast retransmit, Fast recovery)

- When timeout occurs, Threshold set to \( \max(\text{FlightSize}/2, 2\times\text{mss}) \) and CongWin is set to 1 MSS.

**FlightSize**: The amount of data that has been sent but not yet acknowledged.
# TCP sender congestion control

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK receipt for previously unacked data</td>
<td>Slow Start (SS)</td>
<td>CongWin = CongWin + MSS, If (CongWin &gt; Threshold) set state to “Congestion Avoidance”</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Congestion Avoidance (CA)</td>
<td>CongWin = CongWin+MSS * (MSS/CongWin)</td>
<td>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loss event detected by triple duplicate ACK</td>
<td>SS or CA</td>
<td>Threshold = max(FlightSize/2,2<em>mss), CongWin = Threshold+3</em>mss, Set state to “Congestion Avoidance”</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timeout</td>
<td>SS or CA</td>
<td>Threshold = max(FlightSize/2,2*mss), CongWin = 1 MSS, Set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Duplicate ACK</td>
<td>SS or CA</td>
<td>Increment duplicate ACK count for segment being acked</td>
<td>CongWin and Threshold not changed</td>
</tr>
</tbody>
</table>