Recap: Reliable Data Transfer

- rdt3.0
  - stop-and-wait
  - checksum
  - seq. # (one bit, 0 and 1)
  - ACKs
  - timeouts
  - retransmissions
  - data can be corrupted or lost
Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 30 ms RTT, 8000 bit packet:

\[
D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}
\]

- \(U_{\text{sender}}\): utilization – fraction of time sender busy sending

\[
U_{\text{sender}} = \frac{L}{RTT + L/R} = \frac{.008}{30.008} = 0.00027
\]

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!
**rdt3.0: stop-and-wait operation**

- **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,** $t = RTT + L / R$

\[
U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{0.008}{30.008} = 0.00027
\]
Pipelined protocols

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
Pipelining: increased utilization

$U_{\text{sender}} = \frac{3L}{R} \frac{30.008}{RTT + L/R} = 0.00081$

3-packet pipelining increases utilization by a factor of 3!
Pipelined protocols: overview

**Go-back-N:**
- sender can have up to $N$ unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

**Selective Repeat:**
- sender can have up to $N$ unack’ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet
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, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window
GBN: sender extended FSM

- rdt_send(data)
  - if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum, data, checksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
      start_timer
      nextseqnum++
  } else
    refuse_data(data)
- timeout
- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  - base = getacknum(rcvpkt) + 1
  - If (base == nextseqnum)
    stop_timer
  else
    start_timer
- rdt_send(rcvpkt) && corrupt(rcvpkt)
GBN: receiver extended FSM

**ACK-only**: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember *expectedseqnum*

### out-of-order pkt:

- discard (don’t buffer): *no receiver buffering!*
- re-ACK pkt with highest in-order seq #

```plaintext
default
udt_send(sndpkt)

Λ
expectedseqnum=1
sndpkt = make_pkt(0,ACK,chksum)
```

```plaintext
rdt_rcv(rcvpkt)
&& notcurrrupt(rcvpkt)
&& hasseqnum(rcvpkt,expectedseqnum)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(expectedseqnum,ACK,chksum)
udt_send(sndpkt)
expectedseqnum++
```
GBN in action

**sender window (N=4)**

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

**sender**

- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)

**receiver**

- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, discard, (re)send ack1
- receive pkt4, discard, (re)send ack1
- receive pkt5, discard, (re)send ack1

- rcv ack0, send pkt4
- rcv ack1, send pkt5

- ignore duplicate ACK

- pkt 2 timeout

- send pkt2
- send pkt3
- send pkt4
- send pkt5

- rcv pkt2, deliver, send ack2
- rcv pkt3, deliver, send ack3
- rcv pkt4, deliver, send ack4
- rcv pkt5, deliver, send ack5
Selective repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer

- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

- sender window
  - $N$ consecutive seq #’s
  - limits seq #s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

**sender**

- **data from above:**
  - if next available seq # in window, send pkt

- **timeout(n):**
  - resend pkt n, restart timer

- **ACK(n) in [sendbase,sendbase+N]:**
  - mark pkt n as received
  - if n smallest unACKed pkt, advance window base to next unACKed seq #

**receiver**

- **pkt n in [rcvbase, rcvbase+N-1]:**
  - send ACK(n)
  - out-of-order: buffer
  - in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

- **pkt n in [rcvbase-N,rcvbase-1]:**
  - ACK(n)

- **otherwise:**
  - ignore
Selective repeat in action

sender window \((N=4)\)

<table>
<thead>
<tr>
<th>0 1 2 3</th>
<th>4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8</td>
</tr>
</tbody>
</table>

sender

send pkt0
send pkt1
send pkt2
send pkt3
(ed)wait

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer, send ack3
receive pkt4, buffer, send ack4
receive pkt5, buffer, send ack5

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived

 pkt 2 timeout
send pkt2

record ack4 arrived
record ack5 arrived

rcv pkt2; deliver pkt2, pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?
Selective repeat: dilemma

example:
- seq #’s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?
Outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
3.6 principles of congestion control
3.7 TCP congestion control
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order byte stream:**
  - no “message boundaries”

- **pipelined:**
  - TCP congestion and flow control set window size

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **connection-oriented:**
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
TCP segment structure

- Source port #: 32 bits
- Dest port #: 32 bits
- Sequence number
- Acknowledgement number
- Head len: used
- Urg: urgent data (generally not used)
- Ack: ACK # valid
- PSH: push data now (generally not used)
- Options (variable length)
- Application data (variable length)
- Checksum
- Receive window
- Urg data pointer
- Internet checksum (as in UDP)
- RST, SYN, FIN: connection estab (setup, teardown commands)
- Counting by bytes of data (not segments!)
TCP seq. numbers, ACKs

**sequence numbers:**
- byte stream “number” of first byte in segment’s data

**acknowledgements:**
- seq # of next byte expected from other side
- cumulative ACK
TCP seq. numbers, ACKs

**Simple telnet scenario**

- **Host A**
  - User types 'C'
  - Seq=42, ACK=79, data = ‘C’
  - Seq=79, ACK=43, data = ‘C’
  - Seq=43, ACK=80

- **Host B**
  - host ACKs receipt of ‘C’, echoes back ‘C’

**Diagram:**

- User types 'C'
- Seq=42, ACK=79, data = ‘C’
- Seq=79, ACK=43, data = ‘C’
- Seq=43, ACK=80
- host ACKs receipt of ‘C’, echoes back ‘C’
TCP round trip time, timeout

Q: how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

- \textbf{SampleRTT}: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- \textbf{SampleRTT} will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current \textbf{SampleRTT}
TCP round trip time, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \(\alpha = 0.125\)
TCP round trip time, timeout

- **timeout interval**: `EstimatedRTT` plus “safety margin”
  - large variation in `EstimatedRTT` → larger safety margin

- estimate `SampleRTT` deviation from `EstimatedRTT`:
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \\
  \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]
  (typically, $\beta = 0.25$)

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
Outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
   • segment structure
   • **reliable data transfer**
   • flow control
   • connection management
3.6 principles of congestion control
3.7 TCP congestion control
TCP reliable data transfer

- TCP creates rdt service on top of IP’s unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
- retransmissions triggered by:
  - timeout events
  - duplicate acks

Let’s initially consider simplified TCP sender:
- ignore duplicate acks
- ignore flow control, congestion control
TCP sender events:

**data rcvd from app:**
- create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running
  - think of timer as for oldest unacked segment
  - expiration interval: TimeOutInterval

**timeout:**
- retransmit segment that caused timeout
- restart timer

**ack rcvd:**
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments
TCP sender (simplified)

data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
  start timer

timeout
retransmit not-yet-acked segment
  with smallest seq. #
  start timer

ACK received, with ACK field value y
if (y > SendBase) {
  SendBase = y
  /* SendBase–1: last cumulatively ACKed byte */
  if (there are currently not-yet-acked segments)
    start timer
  else stop timer
}

\[ \Lambda \]

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

wait for event
TCP: retransmission scenarios

**lost ACK scenario**
- Host A: Seq=92, 8 bytes of data, ACK=100
- Host B: Seq=92, 8 bytes of data
- Timeout

**premature timeout**
- Host A: Seq=100, 20 bytes of data, ACK=120
- Host B: Seq=92, 8 bytes of data
- Timeout
- Host A: SendBase=120
- Host B: SendBase=120
- ACK=120
TCP: retransmission scenarios

Host A

- Seq=92, 8 bytes of data
- Seq=100, 20 bytes of data
- timeout

Host B

- ACK=100
- ACK=120
- Seq=120, 15 bytes of data
- cumulative ACK

Transport Layer 3-30
## TCP ACK generation

[RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send <em>duplicate ACK</em>, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP fast retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data (“triple duplicate ACKs”), resend unacked segment with smallest seq #

- likely that unacked segment lost, so don’t wait for timeout
TCP fast retransmit

Host A

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data
ACK=100
ACK=100
ACK=100
ACK=100
Seq=100, 20 bytes of data

Host B

fast retransmit after sender receipt of triple duplicate ACK
Outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
   • segment structure
   • reliable data transfer
   • flow control
   • connection management

3.6 principles of congestion control
3.7 TCP congestion control
TCP flow control

Flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast

TCP flow control

application may remove data from TCP socket buffers ....

... slower than TCP receiver is delivering (sender is sending)

flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP flow control

- receiver “advertises” free buffer space by including `rwnd` value in TCP header of receiver-to-sender segments
  - `RcvBuffer` size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust `RcvBuffer`
- sender limits amount of unacked ("in-flight") data to receiver’s `rwnd` value
- guarantees receive buffer will not overflow

`rwnd` value in TCP header of receiver advertises free buffer space to sender.

**Diagram:**
- `buffered data` is the data that has been received and not yet acknowledged.
- `free buffer space` is the portion of the receive buffer that is available for new data.
- `rwnd` value is used to limit the amount of data that can be sent without being acknowledged.

`TCP segment payloads` are sent to the application process via the receiver-side buffering mechanism.
Outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
   • segment structure
   • reliable data transfer
   • flow control
   • connection management
3.6 principles of congestion control
3.7 TCP congestion control
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

Socket clientSocket = newSocket("hostname","port number");

Socket connectionSocket = welcomeSocket.accept();
Agreeing to establish a connection

2-way handshake:

Let’s talk  
OK
ESTAB

choose x
req_conn(x)
acc_conn(x)
ESTAB

Q: will 2-way handshake always work in network?
- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side
Agreeing to establish a connection

2-way handshake failure scenarios:

- Choose x
- Retransmit req_conn(x)
- ESTAB
- req_conn(x)
- acc_conn(x)
- Client terminates
- Connection x completes
- Half open connection! (no client!)
- Server forgets x
- ESTAB

- Choose x
- Retransmit req_conn(x)
- ESTAB
- req_conn(x)
- acc_conn(x)
- Client terminates
- Connection x completes
- Server forgets x
- ESTAB

- Choose x
- Retransmit req_conn(x)
- ESTAB
- req_conn(x)
- data(x+1)
- Client terminates
- Connection x completes
- Server forgets x
- ESTAB

- Choose x
- Retransmit req_conn(x)
- ESTAB
- req_conn(x)
- data(x+1)
- Accept data(x+1)

Transport Layer 3-40
### TCP 3-way handshake

**Client state**

- **LISTEN**
- **SYNSENT**
  - choose init seq num, x
  - send TCP SYN msg
- **ESTAB**
  - received SYNACK(x) indicates server is live;
  - send ACK for SYNACK;
  - this segment may contain client-to-server data

**Server state**

- **LISTEN**
- **SYN_RCVD**
- **ESTAB**
  - choose init seq num, y
  - send TCP SYNACK msg, acking SYN
  - SYNbit=1, Seq=x
  - SYNbit=1, Seq=y
  - ACKbit=1; ACKnum=x+1
  - ACKbit=1, ACKnum=y+1
  - received ACK(y) indicates client is live
TCP 3-way handshake: FSM

Socket connectionSocket = welcomeSocket.accept();

SYN(x)
SYNACK(seq=y,ACKnum=x+1)
create new socket for communication back to client

Socket clientSocket = newSocket("hostname","port number");

SYN(seq=x)

SYN rcvd

ESTAB

ACK(ACKnum=y+1)
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled
TCP: closing a connection

**client state**
- ESTAB
- FIN_WAIT_1
  - clientSocket.close()
  - can no longer send but can receive data
- FIN_WAIT_2
  - wait for server close
- TIMED_WAIT
  - timed wait for 2*max segment lifetime
- CLOSED

**server state**
- ESTAB
- CLOSE_WAIT
  - can still send data
- LAST_ACK
  - can no longer send data
- CLOSED

- FINbit=1, seq=x
  - ACKbit=1; ACKnum=x+1
- FINbit=1, seq=y
  - ACKbit=1; ACKnum=y+1
Outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  • segment structure
  • reliable data transfer
  • flow control
  • connection management

3.6 principles of congestion control
3.7 TCP congestion control
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!
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: $R$
- no retransmission

- maximum per-connection throughput: $R/2$

- large delays as arrival rate, $\lambda_{\text{in}}$, approaches capacity
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{\text{in}} = \lambda_{\text{out}}$
  - transport-layer input includes retransmissions: $\lambda'_{\text{in}} \geq \lambda_{\text{in}}$

![Diagram showing network traffic and buffer management](image-url)
Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

\[
\lambda_{\text{in}} : \text{original data} \\
\lambda_{\text{in}}' : \text{original data, plus retransmitted data}
\]

\[
\lambda_{\text{out}}
\]

free buffer space!

finite shared output link buffers

Host B

A
Causes/costs of congestion: scenario 2

**Idealization:** *known loss*
packets can be lost, dropped at router due to full buffers

- sender only resends if packet *known* to be lost
Causes/costs of congestion: scenario 2

**Idealization:** known loss
packets can be lost, dropped at router due to full buffers

- sender only resends if packet known to be lost

![Diagram showing network traffic and buffer space]

- $\lambda_{in}$: original data
- $\lambda'_{in}$: original data, plus retransmitted data
- $\lambda_{out}$: output data
- $\lambda_{out}$ when sending at $R/2$, some packets are retransmissions but asymptotic goodput is still $R/2$ (why?)

Free buffer space!
Causes/costs of congestion: scenario 2

Realistic: *duplicates*

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

![Diagram of network flow](image)

- When sending at R/2, some packets are retransmissions including duplicated that are delivered!
Causes/costs of congestion: scenario 2

Realistic: duplicates

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered

“costs” of congestion:

- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput

when sending at R/2, some packets are retransmissions including duplicated that are delivered!
Causes/costs of congestion: scenario 3

- four senders doing $\lambda_{in}$
- multihop paths
- timeout/retransmit

**Q:** what happens as $\lambda_{in}$ and $\lambda_{in}'$ increase?

**A:** as red $\lambda_{in}'$ increases, all arriving blue pkts at upper queue are dropped, blue throughput $\rightarrow 0$
Causes/costs of congestion: scenario 3

another “cost” of congestion:
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!
Outline

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
  • segment structure
  • reliable data transfer
  • flow control
  • connection management

3.6 principles of congestion control

3.7 TCP congestion control
TCP congestion control: additive increase multiplicative decrease

- **approach**: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase**: increase $cwnd$ by 1 MSS every RTT until loss detected
  - **multiplicative decrease**: cut $cwnd$ in half after loss

 AIMD saw tooth behavior: probing for bandwidth

cwnd: congestion window, number of unACKed bytes allowed at sender.

MSS: maximum segment size
TCP Congestion Control: details

**TCP sending rate:**
- *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

**sender limits transmission:**
- `cwnd` is dynamic, function of perceived network congestion

\[
\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}
\]

\[
\text{LastByteSent} - \text{LastByteAcked} \leq \min(\text{cwnd}, \text{rwnd})
\]
TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially $cwnd = 1$ MSS
  - double $cwnd$ every RTT
  - done by incrementing $cwnd$ for every ACK received

- **summary:** initial rate is slow but ramps up exponentially fast
TCP: detecting, reacting to loss

- loss indicated by timeout:
  - cwnd set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly

- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - cwnd is cut in half window then grows linearly

- TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?
A: when $cwnd$ gets to 1/2 of its value before timeout.

Implementation:
- variable $ssthresh$
- on loss event, $ssthresh$ is set to 1/2 of $cwnd$ just before loss event
Summary: TCP Congestion Control

**Slow Start**
- cwnd = 1 MSS
- ssthresh = 64 KB
- dupACKcount = 0
- duplicate ACK
- cwnd = cwnd + MSS
- transmit new segment(s), as allowed

**Fast Recovery**
- dupACKcount == 3
- ssthresh = cwnd/2
- cwnd = ssthresh + 3
- retransmit missing segment

**Congestion Avoidance**
- cwnd ≥ ssthresh
- ssthresh = cwnd/2
- cwnd = 1 MSS
- dupACKcount = 0
- transmit new segment(s), as allowed
- cwnd = ssthresh
- dupACKcount = 0
- New ACK

**Fast Recovery**
- cwnd = cwnd + MSS
- transmit new segment(s), as allowed
TCP throughput

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is $\frac{3}{4} W$
  - avg. throughput is $\frac{3}{4}W$ per RTT

$$\text{avg TCP throughput} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$
TCP Futures: TCP over “long, fat pipes”

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires $W = 111,111$ in-flight segments
- throughput in terms of segment loss probability, $L$
  [Mathis 1997]:

  \[
  \text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}
  \]

  $\Rightarrow$ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ — a very small loss rate!

- new versions of TCP for high-speed
TCP Fairness

**fairness goal:** if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K
Why is TCP fair?

two competing sessions:

- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally

Connection 1 throughput vs. Connection 2 throughput diagram:
- Additive increase: connection increases its window proportionally.
- Multiplicative decrease: connection decreases its window by a factor of 2 upon loss.

Equal bandwidth share:
- The diagonal line represents equal bandwidth share.

Congestion avoidance:
- Additive increase: connection increases its window proportionally after loss.
- Multiplicative decrease: connection decreases its window by a factor of 2.
Fairness (unfair situations)

*Fairness and UDP*
- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

*Fairness, parallel TCP connections*
- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate $R$ with 9 existing connections:
  - new app asks for 1 TCP, gets rate $R/10$
  - new app asks for 11 TCPs, gets $R/2$
Explicit Congestion Notification (ECN)

**network-assisted congestion control:**
- two bits in IP header (ToS field) marked *by network router* to indicate congestion
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram) sets ECN bit on receiver-to-sender ACK segment to notify sender of congestion
Transport Layer: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- instantiation, implementation in the Internet
  - UDP
  - TCP

**next:**

- leaving the network “edge” (application, transport layers)
- into the network “core”
- two network layer chapters:
  - data plane
  - control plane