Chapter 3: Transport Layer

Our goals:
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control & flow control

Chapter 3 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

Transport services and protocols
- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, then passes them to network layer
  - rcv side: reassembles segments into messages, then passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP

Transport vs. network layer
- transport layer: logical communication between processes
  - relies on and enhances network layer services
- network layer: logical communication between hosts

Household analogy:
12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service
Internet transport-layer protocols

- Reliable, in-order delivery: TCP
  - Congestion control
  - Flow control
  - Connection setup

- Unreliable, unordered delivery: UDP
  - No-frills extension of "best-effort" IP

- Services not available:
  - Delay guarantees
  - Bandwidth guarantees

Chapter 3 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

Multiplexing / Demultiplexing

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

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

Recall: Segment - unit of data exchanged between transport layer entities
  - aka TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct application layer processes
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:
  - DatagramSocket mySocket1 = new DatagramSocket(9911);
  - DatagramSocket mySocket2 = new DatagramSocket(9922);
- UDP socket identified by two-tuple:
  - (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

Connectionless demux (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

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
  
SP provides “return address”
Connection-oriented demux (cont.)

Client
IP: A
P1

server
IP: C

P4
P5
P6

P2
P3

SP: 9157
DP: 80
S-IP: B
D-IP: C

SP: 9157
DP: 80
S-IP: B
D-IP: C

Connection-oriented demux: Threaded Web Server

Client
IP: A
P1

server
IP: C

P4

P2
P3

SP: 9157
DP: 80
S-IP: A
D-IP: C

SP: 9157
DP: 80
S-IP: A
D-IP: C

Multiplexing/demultiplexing: examples

host A

source port: x

dest. port: 23

host C

source port: 23

dest. port: x

port use: simple telnet app

Web client
host C

Source IP: C
Dest IP: B
source port: y
dest. port: 80

Web client
host A

Source IP: A
Dest IP: B
source port: x
dest. port: 80

Chapter 3 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

Transport Layer 3-13

Transport Layer 3-14

Transport Layer 3-15

Transport Layer 3-16
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

UDP: more

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

UDP checksum

Goal: detect "errors" in transmitted segment
  e.g., flipped bits

Sender:
- treat segment contents as sequence of 16-bit integers
- checksum: addition of segment contents (using 1's complement sum)
- 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 may be 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 0 0 1 0 0 0
1 1 0 1 0 1 0 1 0 1 1 0 1 1 0 1

1 1 0 1 1 1 0 1 1 1 0 1 1 0 1 1
wrapping
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 0 0 0
checksum
1 0 0 1 0 1 0 0 0 1 0 0 0 0 1 1
```
**Chapter 3 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 Reliable data transfer**

- important in app., transport, link layers
- top-10 list of important networking topics!

**Reliable data transfer: getting started**

We'll:
- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

**Reliable data transfer: getting started**

- rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- udt_send(): called by rdt, to transfer packet over unreliable channel to receiver
- deliver_data(): called by rdt to deliver data to upper
- rdt_rcv(): called when packet arrives on rcv-side of channel
Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

```
wait for call from above
send

wait for call from below
recv
```

Rdt 2.0: channel with bit errors

- underlying channel may flip bits in packet
  → introduce a checksum to detect bit errors
- the question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt received OK
  - negative acknowledgements (NAKs): receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback:
    -- control msgs (ACK,NAK) rcvr->sender

```
wait for call from above
send

wait for call from below
recv
```

Rdt2.0: FSM specification

```
wait for call from above
send

wait for call from below
recv
```

Rdt2.0: operation with no errors

```
wait for call from above
send

wait for call from below
recv
```
**rtdt2.0: error scenario**

- `rtd_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- Wait for call from above
- `rtd_rcv(rcvpkt)` && `isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rtd_rcv(rcvpkt)` && `isACK(rcvpkt)`
- `udt_send(ACK)`
- Wait for call from above

**rtdt2.0 has a fatal flaw!**

**What happens if ACK/NAK corrupted?**
- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

**Handling duplicates:**
- sender adds sequence number to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

**Sender and wait**
- Sender sends one packet, then waits for receiver response

---

**rtdt2.1: sender, handles garbled ACK/NAKs**

- `rtd_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `rtd_rcv(rcvpkt)` && `isNAK(rcvpkt)`
- `udt_send(sndpkt)`
- `rtd_rcv(rcvpkt)` && `isACK(rcvpkt)`
- `udt_send(ACK)`
- `rtd_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

---

**rtdt2.1: receiver, handles garbled ACK/NAKs**

- `rtd_rcv(rcvpkt)` && `notcorrupt(rcvpkt)` && `has_seq0(rcvpkt)`
- `extract(rcvpkt.data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK0, checksum)`
- `udt_send(sndpkt)`
- `rtd_rcv(rcvpkt)` && `notcorrupt(rcvpkt)` && `has_seq1(rcvpkt)`
- `extract(rcvpkt.data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK1, checksum)`
- `udt_send(sndpkt)`

- `rtd_rcv(rcvpkt)` && `notcorrupt(rcvpkt)` && `has_seq0(rcvpkt)`
- `extract(rcvpkt.data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK0, checksum)`
- `udt_send(sndpkt)`
- `rtd_rcv(rcvpkt)` && `notcorrupt(rcvpkt)` && `has_seq1(rcvpkt)`
- `extract(rcvpkt.data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK1, checksum)`
- `udt_send(sndpkt)`
**rdt2.1: discussion**

**Sender:**
- seq # added to pkt
- two seq. #’s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “current” pkt has 0 or 1 seq. #

**Receiver:**
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is the expected pkt seq #
- note: receiver cannot know if its last ACK/NAK received OK at sender

**rdt2.2: a NAK-free protocol**

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: retransmit current pkt
  - WHY?

**rdt2.2: sender, receiver fragments**

**rdt3.0: channels with errors and loss**

**New assumption:** underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

**Approach:** sender waits “reasonable” amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be a duplicate
  - use of seq. #’s already handles this
  - receiver must specify seq. # of pkt being ACKed
- requires countdown timer
**rdt3.0 sender**

```
rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
```

```
rdt_rcv(rcvpkt)
& & notcorrupt(rcvpkt)
& & isACK(rcvpkt,1)
stop_timer & deliver data
```

```
udt_send(sndpkt)
```

```
stop_timer
& & deliver data
```

```
udt_send(sndpkt)
start_timer
```

```
timeout
```

```
r retval(rcvpkt)
& & notcorrupt(rcvpkt)
& & isACK(rcvpkt,0)
```

```
stop_timer & deliver data
```

```
udt_send(sndpkt)
start_timer
```

```
timeout
```

```
r retval(rcvpkt)
```

**Performance of rdt3.0**

- rdt3.0 works, but performance stinks
  - example: 1 Gbps link, 15 ms end-end prop. delay, 1KB packet:
    - \[ T_{\text{transmit}} = \frac{L \text{ (packet length in bits)}}{R \text{ (transmission rate, bps)}} = \frac{8\text{kb/pkt}}{10^{10} \text{ b/sec}} = 8 \text{ microsec} \]
    - \[ U_{\text{sender}} = \frac{L}{RTT + L} = \frac{0.008}{30.008} = 0.00027 \]
  - \( U_{\text{sender}} \): utilization - fraction of time sender busy sending
  - 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
  - network protocol limits use of physical resources!
**Pipelining: 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, \( t = RTT + L / R \)

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

**Pipelining: increased utilization**

First packet bit transmitted, \( t = 0 \)

Last bit transmitted, \( t = L / R \)

First packet bit arrives

Last bit of 2nd packet arrives, send ACK

Last bit of 3rd packet arrives, send ACK

ACK arrives, send next

packet, \( t = RTT + L / R \)

Increase utilization by a factor of 3!

\[ U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008 \]

**Pipelined protocols**

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged packets

- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

- Two generic forms of pipelined protocols:
  - **go-Back-N**
  - **selective repeat**

**Go-Back-N**

Sender:

- \( k \)-bit seq # in pkt header
- "window" of up to \( N \), consecutive unack'ed pkts allowed

<table>
<thead>
<tr>
<th>[send_base]</th>
<th>[nextseqnum]</th>
</tr>
</thead>
<tbody>
<tr>
<td>already ack'ed</td>
<td>sent, not yet ack'ed</td>
</tr>
<tr>
<td>usable, not yet sent</td>
<td></td>
</tr>
</tbody>
</table>

- ACK(\( n \)): Acks all pkts up to, including seq \# \( n \) - "cumulative ACK"
  - May receive duplicate ACKs (see receiver)
- Timer for each in-flight pkt
- \( \text{timeout}(N) \): Retransmit pkt \( N \) and all higher seq \# pkts in window
**GBN: sender extended FSM**

```
shadowed

if (nextseqnum < base+N) {
  sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
  udt_send(sndpkt[nextseqnum])
  if (base == nextseqnum)
    start_timer
    nextseqnum++
  } else
  refuse_data(data)
}
```

**GBN in action**

```
sender
send pkt0
send pkt1
send pkt2
send pkt3 (wait)
rcv pkt0
rcv pkt1
rcv pkt2
rcv pkt3, discard
rcv ACK0
send pkt4
rcv ACK1
send pkt5
pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

receiver
rcv pkt0
send ACK0
rcv pkt1
send ACK1
rcv pkt2, deliver
rcv pkt3, deliver
rcv pkt4, discard
rcv pkt5, discard
```

**Selective Repeat**

- **receiver individually acknowledges all correctly received packets**
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- **sender only resends packets for which ACK not received**
  - sender timer for each unACKed pkt
- **sender window**
  - N consecutive seq #s
  - again limits seq #s of sent, unACKed pkts
Selective repeat: sender, receiver windows

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

Example:
- seq #'s: 0, 1, 2, 3
- window size: 3
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?
Chapter 3 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

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control
  - set window size
- send & receive buffers
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs)
    - initialize sender & receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP Segment structure

<table>
<thead>
<tr>
<th></th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>dest port #</td>
</tr>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>head len</td>
<td>not used</td>
</tr>
<tr>
<td>checksum</td>
<td></td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td>(variable length)</td>
</tr>
</tbody>
</table>

TCP segment structure

<table>
<thead>
<tr>
<th></th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>dest port #</td>
</tr>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>head len</td>
<td>not used</td>
</tr>
<tr>
<td>checksum</td>
<td></td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td>(variable length)</td>
</tr>
</tbody>
</table>
TCP seq. #’s and ACKs

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

Q: how receiver handles out-of-order segments
- A: TCP spec doesn’t say
- up to implementor

User types ‘C’
Seq=42, ACK=79, data = ‘C’
Seq=79, ACK=43, data = ‘C’
Seq=43, ACK=80

Host A
Host B

**Q:** how receiver handles out-of-order segments

**TCP Flow Control**

**flow control**

**Role:** Assure sender won’t overrun receiver’s buffers by transmitting too much, too fast

Receiver:
- explicitly informs sender of amount of free buffer space (dynamically changing)
  - **RcvWindow** field in TCP segment

Sender:
- keeps the amount of transmitted, unACKed data less than most recently received **RcvWindow**

**RcvBuffer** = size of TCP Receive Buffer

**RcvWindow** = amount of spare room in Buffer

**TCP Round Trip Time and 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?
- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT

**EstimatedRTT = (1-α)*EstimatedRTT + α*SampleRTT**
- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$
Example RTT estimation:

TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus "safety margin"
  - large variation in EstimatedRTT → larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

\[
\text{DevRTT} = (1-\beta)\times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
\]

(typically, \(\beta = 0.25\))

Then set timeout interval:

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

Chapter 3 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
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks
- Initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control
TCP: reliable data transfer

**TCP sender events:**

- **data rcvd from app:**
  - Create segment with seq #
  - seq # is byte-stream number of first data byte in segment
- **timeout:**
  - retransmit segment that caused timeout
  - restart timer
- **Ack rcvd:**
  - If acknowledges previously unacked segments
    - update what is known to be acked
    - start timer if there are outstanding segments

**Simplified TCP sender**

```plaintext
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
    switch(event)
    event: data received from application above
        create TCP segment with sequence number NextSeqNum
        if (timer currently not running)
            start timer
            NextSeqNum = NextSeqNum + length(data)
            pass segment to IP
    event: timer timeout for segment with sequence number y
        retransmit segment with sequence number y
        compute new timeout interval for segment y
        restart timer for segment y
    event: ACK received, with ACK field value of y
        if (y > SendBase) {
            // cumulative ACK of all data up to y
            SendBase = y
            if (there are currently not-yet-acknowledged segments)
                start timer
        } else if (a duplicate ACK for already ACKed segment y)
            increment number of duplicate ACKs received for y
            if (number of duplicate ACKs received for y == 3) {
                // TCP fast retransmit
                resend segment with sequence number y
                restart timer for segment y
            }
    } /* end of loop forever */
```

Comment:
- SendBase-1: last cumulatively acked byte
- Example:
  - SendBase-1 = 71;
  - y = 73, so the rcvr wants 73+;
  - y > SendBase, so new data is acked
**TCP ACK generation** \[RFC 1122, RFC 2581\]

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>in-order segment arrival, no gaps, everything else already ACKed</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>in-order segment arrival, no gaps, one delayed ACK pending</td>
<td>immediately send single cumulative ACK</td>
</tr>
<tr>
<td>out-of-order segment arrival, higher-than-expect seq. #, gap detected</td>
<td>send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>arrival of segment that partially or completely fills gap</td>
<td>immediate ACK if segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

**TCP retransmission scenarios**

**TCP retransmission scenarios (more)**

**TCP ACK generation** \[RFC 1122, RFC 2581\]

**Event at Receiver**
- Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed
- Arrival of in-order segment with expected seq #. One other segment has ACK pending
- Arrival of out-of-order segment higher-than-expect seq. #. Gap detected
- Arrival of segment that partially or completely fills gap

**TCP Receiver action**
- Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
- Immediately send single cumulative ACK, ACKing both in-order segments
- Immediately send duplicate ACK, indicating seq. # of next expected byte
- Immediate send ACK, provided that segment starts at lower end of gap
**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.
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires

**Fast retransmit algorithm:**

```
if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
        start timer
} else {
    increment count of dup ACKs received for y
    if (count of dup ACKs received for y = 3) {
        resend segment with sequence number y
    }
```

**Chapter 3 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**

- receive side of TCP connection has a receive buffer:
  - flow control
  - sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - speed-matching service: matching the send rate to the receiving app’s drain rate
  - app process may be slow at reading from buffer
**TCP Flow control: how it works**

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
  - guarantees receive buffer doesn't overflow

(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer
  - $\text{RcvWindow} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead})$

**Chapter 3 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 Connection Management**

**Recall:** TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)
- client: connection initiator
  - Socket clientSocket = new Socket("hostname","port number");
- server: contacted by client
  - Socket connectionSocket = welcomeSocket.accept();

**Three way handshake:**

**Step 1:** client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data

**Step 2:** server host receives SYN, replies with SYNA CK segment
  - server allocates buffers
  - specifies server initial seq. #

**Step 3:** client receives SYNA CK, replies with ACK segment, which may contain data
Closing a Connection: Reaching Agreement

Q: can I decide to close, knowing the other entity has also agreed to close and knows that I will close?

A similar scenario: can two armies coordinate their attacks if communication is unreliable?

TCP Connection Management (cont.)

Closing a connection:

client closes socket:
    `clientSocket.close();`

Step 1: client end system sends TCP FIN control segment to server

Step 2: server receives FIN, replies with ACK. Closes connection, sends FIN.

Step 3: client receives FIN, replies with ACK.

- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.

TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle
**Chapter 3 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
- no retransmission

**Causes/costs of congestion: scenario 3**

Another “cost” of congestion:
- when packet dropped, any "upstream transmission capacity used for that packet was wasted!"
Approaches to congestion control

Two broad approaches to congestion control:

- **End-end congestion control:**
  - no explicit feedback from network
  - congestion inferred from end-system observed loss, delay
  - approach taken by TCP

- **Network-assisted congestion control:**
  - routers provide feedback to end systems
    - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
    - explicit rate at which sender should send

---

Case study: ATM ABR congestion control

**ABR: available bit rate:**
- "elastic service"
- if sender's path "underloaded": sender should use available bandwidth
- if sender's path congested: sender is throttled back to minimum guaranteed rate

**RM (resource management) cells:**
- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("network-assisted")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

---

Case study: ATM ABR congestion control

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender's send rate thus minimum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell

---

Chapter 3 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

- end-end control
  - no network assistance
- sender limits transmission:
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]
- Roughly, CongWin is a dynamic, function of perceived network congestion

\[ \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ bytes/sec} \]

How does sender perceive congestion?
- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (CongWin) after loss event

three mechanisms:
- AIMD
- slow start
- conservative after timeout events

TCP congestion control:
- "probing" for usable bandwidth:
  - ideally: transmit as fast as possible (CongWin as large as possible) without loss
  - increase CongWin until loss (congestion)
  - loss: decrease CongWin, then begin probing (increasing) again
- two "phases"
  1. slow start
  2. congestion avoidance
- important variables:
  - CongWin
  - threshold: defines threshold between:
    - slow start phase and
    - congestion control phase

TCP AIMD

- multiplicative decrease: cut CongWin in half after loss event
- additive increase: increase CongWin by 1 MSS every RTT in the absence of loss events: probing

Long-lived TCP connection
TCP Congestion Avoidance

Congestion avoidance
/* slowstart is over */
/* Congwin > threshold */
Until (loss event) {
   every w segments ACKed:
      Congwin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart

TCP Slow Start

- When connection begins,
  CongWin = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps

- available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate

TCP Slow Start (more)

- When connection begins,
  increase rate exponentially until first loss event:
    - double CongWin every RTT
    - done by incrementing CongWin for every ACK received

- Summary: initial rate is slow but ramps up exponentially fast

Refinement

- After 3 dup ACKs:
  - CongWin is cut in half
  - window then grows linearly

- But after timeout event:
  - CongWin instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Philosophy:

- 3 dup ACKs indicates network capable of delivering some segments
- While, a timeout before 3 dup ACKs is "more alarming"
**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 &quot;Congestion Avoidance&quot;</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>ACK receipt for previously unacked data</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>Loss event detected by triple duplicate ACK</td>
<td>SS or CA</td>
<td>Threshold = CongWin/2, CongWin = Threshold, Set state to &quot;Congestion Avoidance&quot;</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>Timeout</td>
<td>SS or CA</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to &quot;Slow Start&quot;</td>
<td>Enter slow start</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>

**TCP throughput**

- What’s the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughput: .75 W/RTT
**TCP Futures**

- Example: 1500 byte segments, 100ms RTT, want 10Gbps throughput
  - Requires window size $W = 83,333$ in-flight segments
  - Throughput in terms of loss rate:
    \[
    L = 2 \cdot 10^{-10} \frac{1.22 \cdot MSS}{RTT \sqrt{L}}
    \]
- New versions of TCP for high-speed needed!

**TCP Fairness**

**Fairness goal:** if $K$ TCP sessions share the 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 throughput increases
- Multiplicative decrease decreases throughput proportionally

**Fairness and UDP**

- Multimedia apps often do not use TCP
  - Do not want rate throttled by congestion control
- Instead use UDP:
  - Pump audio/video at constant rate, tolerate packet loss

**Fairness and parallel TCP connections**

- Nothing prevents app from opening parallel connections between 2 hosts.
- Web browsers do this
- Example: link of rate $R$ supporting 9 connections:
  - New app asks for 1 TCP, gets rate $R/10$
  - New app asks for 11 TCPs, gets $R/2$!
**Delay modeling**

Q: How long does it take to receive an object from a Web server after sending a request?

Ignoring congestion, delay is influenced by:
- TCP connection establishment
- data transmission delay
- slow start

Notation, assumptions:
- Assume one link between client and server of rate R
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

Window size:
- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start

**Fixed congestion window (1)**

First case:
WS/R > RTT + S/R: ACK for first segment in window returns before window's worth of data sent

delay = 2RTT + O/R

**Fixed congestion window (2)**

Second case:
WS/R < RTT + S/R: wait for ACK after sending window's worth of data sent

delay = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]

**TCP Delay Modeling: Slow Start (1)**

Now suppose window grows according to slow start

Will show that the delay for one object is:

$$\text{Latency} = 2RTT + \frac{O}{R} + P \left( RTT + \frac{S}{R} - (2^P - 1) \frac{S}{R} \right)$$

where $P$ is the number of times TCP idles at server:

$$P = \min(Q, K - 1)$$

- where Q is the number of times the server idles if the object were of infinite size.
- and K is the number of windows that cover the object.
TCP Delay Modeling: Slow Start (2)

Delay components:
- 2 RTT for connection estab and request
- O/R to transmit object
- time server idles due to slow start

Server idles:
\[ P = \min(K-1,Q) \] times

Example:
- O/S = 15 segments
- K = 4 windows
- Q = 2
- \( P = \min(K-1,Q) = 2 \)

Server idles \( P=2 \) times

TCP Delay Modeling (3)

\[ \frac{S}{R} + RTT = \text{time from when server starts to send segment until server receives acknowledgement} \]

\[ 2^{k-1} \frac{S}{R} = \text{time to transmit the } k\text{th window} \]

\[ \left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] = \text{idle time after the } k\text{th window} \]

\[ \text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} \text{idleTime}_p \]

\[ = \frac{O}{R} + 2RTT + \left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] \]

\[ = \frac{O}{R} + 2RTT + P(\text{RTT} + \frac{S}{R}) - (2^P - 1) \frac{S}{R} \]

TCP Delay Modeling (4)

Recall \( K \) = number of windows that cover object

How do we calculate \( K \)?

\[ K = \min\{ k : 2^0S + 2^1S + \cdots + 2^{k-1}S \geq O \} \]

\[ = \min\{ k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S \} \]

\[ = \min\{ k : k \geq \log_2\left( \frac{O}{S} + 1 \right) \} \]

\[ = \left[ \log_2\left( \frac{O}{S} + 1 \right) \right] \]

Calculation of \( Q \), number of idles for infinite-size object, is similar (see HW).

HTTP Modeling

- Assume Web page consists of:
  - 1 base HTML page (of size \( O \) bits)
  - \( M \) images (each of size \( O \) bits)

- Non-persistent HTTP:
  - \( M+1 \) TCP connections in series
  - Response time = \( (M+1)O/R + (M+1)2RTT + \text{sum of idle times} \)

- Persistent HTTP:
  - 2 RTT to request and receive base HTML file
  - 1 RTT to request and receive M images
  - Response time = \( (M+1)O/R + 3RTT + \text{sum of idle times} \)

- Non-persistent HTTP with \( X \) parallel connections:
  - Suppose \( M/X \) integer.
  - 1 TCP connection for base file
  - \( M/X \) sets of parallel connections for images.
  - Response time = \( (M+1)O/R + (M/X + 1)2RTT + \text{sum of idle times} \)
HTTP Response time (in seconds)

For low bandwidth, connection & response time dominated by transmission time. Persistent connections only give minor improvement over parallel connections.

Chapter 3: Summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet
  - UDP
  - TCP

Next:
- leaving the network "edge" (application, transport layers)
- into the network "core"