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

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, 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
  - relies on, enhances, network layer services

**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

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)

Transport Layer 3-5

Transport Layer 3-6

Transport Layer 3-7

Transport Layer 3-8
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
- host uses IP addresses & port numbers to direct segment to appropriate socket

Connectionless demultiplexing

- Create sockets with port numbers:
  - DatagramSocket mySocket1 = new DatagramSocket(12534);
  - DatagramSocket mySocket2 = new DatagramSocket(12535);
- UDP socket identified by two-tuple:
  - (dest IP address, dest port number)

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
- 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 (cont)

Client IP: A
P1

server IP: C
P2

SP: 9157
DP: 80
S-IP: A
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

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

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

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

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
- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

**UDP segment format**

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

**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 1 0 0 1 1 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

sum checksum: 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 0 0

**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

**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...
Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!
- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

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

(a) provided service
(b) service implementation
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

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

Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - 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

Rdt2.0: FSM specification
rdt2.0: operation with no errors

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `Wait for call from above`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `Wait for call from below`
- `rdt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(NAK)`
- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`

rdt2.0: error scenario

- `rdt_send(data)`
- `snkpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`
- `Extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`
- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)`
- `Wait for call from above`
- `rdt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
- `udt_send(NAK)`
- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`

rdt2.0 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 retransmits current pkt if ACK/NAK garbled
- sender adds sequence number to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt

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

rdt2.1: sender, handles garbled ACK/NAKs

- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `Wait for call 0 from above`
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))`
- `udt_send(sndpkt)`
- `rdt_send(data)`
- `sndpkt = make_pkt(1, data, checksum)`
- `udt_send(sndpkt)`
- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))`
**rdt2.1: receiver, handles garbled ACK/NAKs**

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 expected pkt seq #
- note: receiver can **not** 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

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

Sender FSM fragment:
- rdt_send(data)
  - sndpkt = make_pkt(0, data, checksum)
  - udt_send(sndpkt)

Receiver FSM fragment:
- rdt_recv(rcvpkt)
  - rdt_send(data)
    - rdt_recv(rcvpkt) && notcorrupt(rcvpkt) && 
      has_seq1(rcvpkt)
      udt_send(sndpkt)
    - extract(rcvpkt, data)
      deliver_data(data)
      sndpkt = make_pkt(ACK1, checksum)
      udt_send(sndpkt)
    - rdt_recv(rcvpkt) && notcorrupt(rcvpkt) ||
      has_seq1(rcvpkt)
      udt_send(sndpkt)

Receiver:
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can **not** know if its last ACK/NAK received OK at sender
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 duplicate, but use of seq. #'s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer

rdt3.0 in action

rdt3.0 in action
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- example: 1 Gbps link, 15 ms e-e prop. delay, 1KB packet:

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

- Utilization - fraction of time sender busy sending:

\[ U_{sender} = \frac{L}{RTT + L/R} \]

- 1KB pkt every 30 msec \( \rightarrow \) 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, \( t = RTT + L/R \)

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

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

- First packet bit transmitted, \( t = 0 \)
- Last bit transmitted, \( t = L/R \)
- First packet bit arrives
- Last packet bit arrives, send ACK
- Last bit of 2nd packet arrives, send ACK
- Last bit of 3rd packet arrives, send ACK

\[ U_{sender} = \frac{3L}{RTT + L/R} = \frac{0.024}{30.008} = 0.0008 \]

Increase utilization by a factor of 3!
**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 each in-flight pkt
- timeout(n): retransmit pkt 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, chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
        nextseqnum++
    } else
        refuse_data(data)
```

```
udt_send(sndpkt[base])
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: 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 #

**GBN in action**
**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
  - Again limits seq #'s of sent, unACKed pkts

---

**Selective repeat: sender, receiver windows**

---

**Selective repeat in action**
Selective repeat: dilemma

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
- 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) init's sender, receiver state before data exchange
- flow controlled: sender will not overwhelm receiver

TCP segment structure
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

Host A: Seq=42, ACK=79, data = 'C'
Host B: Seq=79, ACK=43, data = 'C'
User types 'C' host ACKs receipt of echoed 'C', echoes back 'C'

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

Example RTT estimation:

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
**TCP Round Trip Time 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:
\[ \text{Timeout Interval} = \text{EstimatedRTT} + 4 \times \text{DevRTT} \]

---

**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

---

**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 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 acknowledges previously unacked segments
      - update what is known to beacked
      - start timer if there are outstanding segments
TCP sender (simplified)

```plaintext
loop (forever) {
  switch(event) {
    case data received from application above:
      create TCP segment with sequence number NextSeqNum
      if (timer currently not running)
        start timer
      pass segment to IP
      NextSeqNum = NextSeqNum + length(data)
    case timer timeout:
      retransmit not-yet-acknowledged segment with
      smallest sequence number
      start timer
    case ACK received, with ACK field value of y:
      if (y > SendBase) {
        SendBase = y
        if (there are currently not-yet-acknowledged segments)
          start timer
      }
  }
}
```

Comment:
- SendBase-1: last cumulatively ack'ed byte
- Example:
  SendBase-1 = 71; y = 73, so the rcvr wants 73+; y > SendBase, so that new data is acked

TCP: retransmission scenarios

- Host A: Seq=100, 20 bytes data
  - ACK=100
  - Timeout

- Host B: Seq=92, 8 bytes data
  - ACK=120
  - Session timeout

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 duplicate ACK, 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 startssize lower end of gap</td>
</tr>
</tbody>
</table>
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:

```java
event: ACK received, with ACK field value of y
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:
  - data from IP
  - segment structure
  - TCP data to buffer
  - flow control
  - application process
  - ReceiveWindow

- send 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
  - \( Rcv\text{Window} = RcvBuffer - \text{[LastByteRcvd - LastByteRead]} \)

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 SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management (cont.)

Closing a connection:

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

  \[ \text{closed} \]

  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.

  \[ \text{closed} \]

  close

  timed wait

  FIN

  ACK
**TCP Connection Management (cont.)**

**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.

---

**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
- large delays when congested
- maximum achievable throughput

Causes/costs of congestion: scenario 2

- one router, finite buffers
- sender retransmission of lost packet
- "perfect" retransmission only when loss: \( \lambda'_\text{in} > \lambda'_\text{out} \)
- retransmission of delayed (not lost) packet makes \( \lambda'_\text{in} \) larger (than perfect case) for same \( \lambda'_\text{out} \)

Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as \( \lambda'_\text{in} \) and \( \lambda'_\text{out} \) increase?
**Causes/costs of congestion: scenario 3**

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

**Approaches towards congestion control**

Two broad approaches towards 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 sender should send at

**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 throttled 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

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender send rate thus maximum 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: additive increase, multiplicative decrease

- **Approach**: increase transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase**: increase $\text{CongWin}$ by 1 MSS every RTT until loss detected
  - **multiplicative decrease**: cut $\text{CongWin}$ in half after loss

TCP Slow Start

- When connection begins, $\text{CongWin} = 1 \text{ MSS}$
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- When connection begins, increase rate exponentially fast until first loss event
- available bandwidth may be $>> \text{MSS/RTT}$
  - desirable to quickly ramp up to respectable rate

TCP Congestion Control: details

- sender limits transmission: $\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$
- Roughly, $\text{rate} = \frac{\text{CongWin}}{\text{RTT}}$ bytes/sec
- $\text{CongWin}$ is dynamic, function of perceived network congestion

*How does sender perceive congestion?*

- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate ($\text{CongWin}$) after loss event

*three mechanisms:*

- AIMD
- slow start
- conservative after timeout events
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

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

**Implementation:**
- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

Refinement: inferring loss

- 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
- timeout indicates a "more alarming" congestion scenario

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 CongWin/2 and CongWin set to Threshold.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.
**TCP sender congestion control**

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Start (SS)</td>
<td>ACK receipt for previously unacked data</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>Congestion Avoidance (CA)</td>
<td>ACK receipt for previously unacked data</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>SS or CA</td>
<td>Loss event detected by triple duplicate ACK</td>
<td>Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Timeout</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Duplicate ACK</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 throughput 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: TCP over “long, fat pipes”**

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size W = 83,333 in-flight segments
- Throughput in terms of loss rate:
  \[
  \frac{1.22 \cdot MSS}{RTT \sqrt{L}}
  \]
  \[
  \Rightarrow L = 2 \cdot 10^{-10} \text{ Wow}
  \]
- 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

\[ \begin{align*}
    \text{Connection 1 throughput} & = \frac{R}{10} \\
    \text{Connection 2 throughput} & = \frac{R}{2}
\end{align*} \]

Equal bandwidth share

Fairness (more)

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
- Research area: TCP friendly

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!

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”