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 vs. network layer
- network layer: logical communication between hosts
  - processes = kids
  - app messages = letters in envelopes
  - hosts = houses
  - transport protocol = Ann and Bill
  - network-layer protocol = postal service
- transport layer: logical communication between processes
  - relies on, enhances, network layer services
  - Household analogy: 12 kids sending letters to 12 kids

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
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

**Transport Layer**

**Multiplexing/demultiplexing**

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

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

**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 a source, destination port number
  - host uses IP addresses & port numbers to direct segment to appropriate socket

**TCP/UDP segment format**

- 32 bits
  - source port #
  - dest port #
  - other header fields
  - application data (message)

**Connectionless demultiplexing**

- Create sockets with port numbers:
  - DatagramSocket mySocket1 = new DatagramSocket(12534);
  - DatagramSocket mySocket2 = new DatagramSocket(12535);

- 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

**Connection-oriented demultiplexing**

- 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
### 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

### 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 checksum

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

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

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

- **Note**
  - When adding numbers, a carryout from the most significant bit needs to be added to the result
  - Example: add two 16-bit integers

```
  1 1 1 0 0 1 1 0 0 1 1 0 1 0 1 0 1
  1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
  --------------------------------
  1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 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!

```
(a) provided service
```

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Principles of Reliable data transfer: getting started

```
send side
```

- **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.

```
receive side
```

- **deliver_data()**: called by rdt to deliver data to upper layer.
- **rdt_rcv()**: called when packet arrives on rcv-side of channel.

Reliable data transfer: get started

```
(a) provided service
```

```
(b) service implementation
```

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
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 in both directions!
- use finite state machines (FSM) to specify sender, receiver

Transport Layer 3-25

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 read data from underlying channel

Transport Layer 3-26

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

Transport Layer 3-27

Rdt2.0: FSM specification

sender

receiver

Transport Layer 3-28

Rdt2.0: operation with no errors

Transport Layer 3-29

Rdt2.0: error scenario

Transport Layer 3-30
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

Sender:
- seq # added to pkt
- two seq #’s (0,1) will suffice. Why?
- must check if received ACK/NAK garbled
- 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

Transport Layer 3-35

Transport Layer 3-36

Transport Layer 3-33

Transport Layer 3-32

Transport Layer 3-34
**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

**Example:**
```
rdt_sender

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

receiver: rdt_rcv(rcvpkt)
if notcorrupt(rcvpkt)
  if isACK(rcvpkt,1)
    udt_send(sndpkt)
    start_timer
  else
    rdt_rcv(rcvpkt)
    rdt_send(data)

sender: timeout
```

**Transport Layer**

**Performance of rdt3.0**

- **rdt3.0 works, but performance stinks**
- Ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
  \[
  d_{max} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^{9} \text{ bps}} = 8 \text{ microseconds}
  \]
  - \( U_{sender} = \frac{L}{R} \) = 0.008
  - \( L / R \) = 30.008 = 0.00027
  - 1 KB pkt every 30 msec → 33 kbps/sec throughput over 1 Gbps link
  - Network protocol limits use of physical resources!

```
sender

rate = \( \frac{L}{RTT + L} \) = 0.008
```

**Transport Layer**

**rdt3.0 sender**

**rdt3.0 in action**

(a) operation with no loss

(b) lost packet

(c) lost ACK

(d) premature timeout

**rdt3.0: stop-and-wait operation**

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

**Transport Layer**
### 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

![Diagram of pipelined protocol](image)

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

### Pipelining: increased utilization

![Diagram of pipelining utilization](image)

- first packet bit transmitted, $t = 0$
- last packet bit transmitted, $t = L / R$
- ACK arrives, send next packet, $t = RTT + L / R$
- last bit of 3rd packet arrives, send ACK

### Pipeline Protocols

**Go-Back-N: big picture:**
- Sender can have up to $N$ unacked packets in pipeline
- Rcvr only sends cumulative acks
  - 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: big picture:**
- Sender can have up to $N$ unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacknowledged packet
  - When timer expires, retransmit only unacknowledged 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 each in-flight pkt
- timeout(n): retransmit pkt $n$ and all higher seq # pkts in window

### GBN: sender extended FSM

```
# rtt_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)

# timeout
if (base == nextseqnum)
    stop_timer
else
    start_timer
```

**GBN:** sender extended FSM

![Diagram of GBN FSM](image)
**GBN: receiver extended FSM**

- **Default:**
  - udt_send(sndpkt)
- **Wait:**
  - nd_rcv(hdr,rcvpkt)
  - &~ack_seqnum(rcvpkt)
  - &~chksum(rcvpkt)
  - &~expected_seqnum(rcvpkt)
  - &~delivered(rcvpkt)
  - &~expected_seqnum(rcvpkt)
  - udt_send(sndpkt)
  - make_pkt(expected_seqnum, ACK, chksum)

**ACK-only:** always send ACK for correctly-received pkt with highest in-order seq #
- may generate duplicate ACKs
- need only remember expected_seqnum

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

**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 in action**

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

**Selective repeat in action**

- Send pkt1
- Send pkt2
- Send pkt3
- Send pkt4
- Send pkt5
- Send pkt6
- Send pkt7

**Selective repeat: sender, receiver windows**

- **send_base**
  - **next_seqnum**
  - **window size**
  - **window full**
  - **already ack ad**
  - **not yet acked**
  - **usable, not yet sent**
  - **not usable**

**Transport Layer**

- 3-49
- 3-50
- 3-51
- 3-52
- 3-53
- 3-54
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?

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
- 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 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

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

TCP segment structure

- source port #
- dest port #
- sequence number
- acknowledgment number
- options (variable length)
- data (variable length)
- checksum
- max receive window
-urg data marker
-rcvr willing to accept
- # bytes
- flow control
- connection management
- reliable data transfer
- segment structure

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

RFCs: 793, 1122, 1323, 2018, 2581

Transport Layer 3-05

Transport Layer 3-06

Transport Layer 3-07

Transport Layer 3-08

Transport Layer 3-09

Transport Layer 3-10
TCP Round Trip Time and Timeout

**EstimatedRTT = (1-\(\alpha\))*EstimatedRTT + \(\alpha\)*SampleRTT**

- Exponential weighted moving average
- Influence of past sample decreases exponentially fast
- Typical value: \(\alpha = 0.125\)

**Example RTT estimation:**

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>RTT (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>150</td>
</tr>
<tr>
<td>3</td>
<td>200</td>
</tr>
<tr>
<td>4</td>
<td>250</td>
</tr>
<tr>
<td>5</td>
<td>300</td>
</tr>
<tr>
<td>6</td>
<td>350</td>
</tr>
</tbody>
</table>

**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-\(\beta\))*\text{DevRTT} + \(\beta\)*|\text{SampleRTT}-\text{EstimatedRTT}|\]
  - Typically, \(\beta = 0.25\)

Then set timeout interval:

**TimeoutInterval = EstimatedRTT + 4*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
- 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 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 be acked
    - Start timer if there are outstanding segments
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

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

TCP sender (simplified)

TCP retransmission scenarios

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
    }
}
```

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

(Suppose TCP receiver discards out-of-order segments)
- spare room in buffer
  - RcvWindow
  - RcvBuffer - (LastByteRcvd - LastByteRead)

TCP Connection Management

Recall: TCP sender, receiver establish 'connection' before exchanging data segments
- Three way handshake:
  - Step 1: client host sends TCP SYN segment to server
  - Step 2: server host receives SYN, replies with SYNACK segment
  - Step 3: client receives SYNACK, replies with ACK segment, which may contain data

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

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

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();
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.

Transport Layer

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

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!

Transport Layer

Causes/costs of congestion: scenario 1
- two senders, two receivers
- one router, infinite buffers
- no retransmission
- large delays when congested
- maximum achievable throughput

Transport Layer
Causes/costs of congestion: scenario 2
- one router, finite buffers
- sender retransmission of lost packet

senders retransmission of lost packet

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")
  - NE 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 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 CongWin by 1 MSS every RTT until loss detected
  - *multiplicative decrease:* cut CongWin in half after loss

TCP Congestion Control: details

- *sender limits transmission:* CongWin ≤ LastByteSent - LastByteAcked
- *Roughly,*
  \[
  \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}
  \]
- *CongWin is dynamic, function of perceived network congestion*
- *three mechanisms:*
  - AIMD
  - slow start
  - conservative after timeout events

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
- *When connection begins,* increase rate exponentially fast until first loss event

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: 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

When should the exponential increase switch to linear?
- 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

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 &quot;Congestion Avoidance&quot;</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 &quot;Congestion Avoidance&quot;</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 &quot;Slow Start&quot;</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:
  \[
  \frac{1.22 \cdot MSS}{RTT \cdot L}
  \]
  - \( L = 2 \cdot 10^{-10} \) 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

![Image of TCP connections](Transport Layer 3-103)

**Why is TCP fair?**

Two competing sessions:
- Additive increase gives slope of 1, as throughout increases.
- Multiplicative decrease decreases throughput proportionally.

![Graph showing TCP throughput](Transport Layer 3-104)

**Fairness (more)**

- **Fairness and UDP**
  - Multimedia apps often do not use TCP.
  - Instead use UDP to 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".