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
- gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

Multiplexing at send host:
- sending data from multiple sockets, enveloping data with header

Diagram showing the interaction between application, transport, network, link, and physical layers.
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.

Connectionless demultiplexing
- Create sockets with port numbers:
  - `DatagramSocket mySocket1 = new DatagramSocket(99111);`
  - `DatagramSocket mySocket2 = new DatagramSocket(99222);`
- 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);`
- IP provides "return address".

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.
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**

**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

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

- Sum
  - 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 1 1 1 0 1
  - Checksum 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1

**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...
**TCP: Overview**

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order byte stream:**
  - no "message boundaries"
- **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
- **send & receive buffers**

**TCP segment structure**

- URG: urgent data (generally not used)
- ACK: ACK # valid
- PSH: push data now (generally not used)
- RST, SYN, FIN: connection estab (setup, teardown commands)

**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
  - 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 Round Trip Time and Timeout

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

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

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:

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

Example RTT estimation:

![Graph showing RTT estimation over time]

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: retransmission scenarios

Host A
 Seq = 100, 20 bytes data
ACK = 100

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

Host A
Seq = 92, 8 bytes data
ACK = 100

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

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:
- if (y > SendBase) {
  SendBase = y
  if (there are currently not-yet-acknowledged segments) {
    start timer
  }
}

TCP sender
(simplified)

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
    pass segment to IP
    NextSeqNum = NextSeqNum + length(data)
  event: timer timeout
    retransmit not-yet-acknowledged segment with smallest sequence number
    start timer
  event: ACK received, with ACK field value of y
    if (y > SendBase) {
      SendBase = y
      if (there are currently not-yet-acknowledged segments) {
        start timer
      }
    }
} /* end of loop forever */
TCP retransmission scenarios (more)

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver

<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, ACK'ing 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 starts at lower end of gap</td>
</tr>
</tbody>
</table>

TCP Receiver action

- Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
- Immediately send single cumulative ACK, ACK'ing 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:
  - 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.
- Fast retransmit:
  - 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:

```
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:
  - speed-matching service: matching the send rate to the receiving app's drain rate

**TCP Flow control: how it works**

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

- spare room in buffer
  - RcvrBuffer - (LastByteRcvd - LastByteHead)
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

Closing a connection:

Step 1: client closes socket: clientSocket.close();
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.
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

Transport Layer 3-45
Transport Layer 3-46
Causes/costs of congestion: scenario 2
- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when loss: $\lambda_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes $\lambda_{out}$ larger (than perfect case) for same $\lambda_{in}$

"costs" of congestion:
- more work (retrans) for given "goodput" 
- unneeded retransmissions: link carries multiple copies of pkt

Causes/costs of congestion: scenario 3
- four senders
- multihop paths
- timeout/retransmit

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

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

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: \[ \text{LastByteSent - LastByteAcked} \leq \text{CongWin} \]
- Roughly, \[ \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec} \]
- CongWin is dynamic, function of perceived network congestion

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 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

---

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
- timeout before 3 dup ACKs is "more alarming"
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

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>Event</th>
<th>State</th>
<th>TCP Sender Action</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK receipt for previously acknowledged data</td>
<td>Slow Start (SS)</td>
<td>CongWin = CongWin - MSS, if CongWin &gt; Threshold set state to “Congestion Avoidance”</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>ACK receipt for previously acknowledged 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>Data loss detected by triple duplicate ACK</td>
<td>SS or CA</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”</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</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 asked</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

- 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:
  \[ \frac{1.22 \cdot \text{MSS}}{\text{RTT} \cdot L} \]
- $L = 230^{-10}$ Wow
- 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 throughout increases
- Multiplicative decrease decreases throughput proportionally

Fairness (more)

- 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 UDP:
- 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

\[
\text{delay} = 2RTT + O/R
\]

---

Fixed congestion window (2)

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

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

---

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

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

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 $P \times 2$ times

TCP Delay Modeling (3)

$\frac{S}{R}$ + RTT = time from when server starts to send segment until server receives acknowledgment

$\frac{S}{R} + \frac{S}{R}$ = time to transmit the kth window

$\frac{S}{R} + \frac{S}{R}$ = idle time after the kth window

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

HTTP Modeling

Recall $K =$ number of windows that cover object

How do we calculate $K$?

$$K = \min \left( k : 2^k \geq O \right)$$

Response time 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 + 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 + 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)2RTT + sum of idle times
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"