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)

---

**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

---

**TCP/UDP segment format**

- **32 bits**
  - source port #
  - dest port #
  - other header fields
  - application data (message)
Transport Layer 3-9

Connectionless demultiplexing

- Create sockets with port numbers:
  
  ```java
  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

Transport Layer 3-10

Connectionless demux (cont)

```java
DatagramSocket serverSocket = new DatagramSocket(6428);
```

- Client IP: B
- P2
- Server IP: C
- P3
- Client IP: B

**SP provides “return address”**
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

Connection-oriented demux (cont)
Connection-oriented demux: Threaded Web Server

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!

source port # | dest port # |
-------------|------------|
length       | checksum   |

UDP segment format

Application data (message)

32 bits
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

\[
\begin{array}{cccccccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\hline
1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
\]

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

transport layer 3-17

Transport Layer 3-18
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!

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

---

**Transport Layer 3-21**

---

**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)

---

**Transport Layer 3-22**
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

![State Transition Diagram]

- **state**: when in this “state” next state uniquely determined by next event
- **event causing state transition**
- **actions taken on state transition**
**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

```
Wait for call from above
| rdt_send(data) |
| packet = make_pkt(data) |
| udt_send(packet) |
```

```
Wait for call from below
| rdt_rcv(packet) |
| extract (packet, data) |
| deliver_data(data) |
```

**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

\[
\begin{align*}
\text{sender} & \quad \text{Halt} \\
\text{rdt_send(data)} \\
\text{snkpkt} = \text{make_pkt(data, checksum)} \\
\text{udt_send(sndpkt)} & \quad \text{rdt_rcv(rcvpkt)} & \quad \text{isNAK(rcvpkt)} \\
\text{udt_send(sndpkt)} & \quad \text{udt_send(ACK)} \\
\text{Wait for ACK or NAK} & \quad \text{Wait for call from above} \\
\text{rdt_rcv(rcvpkt)} & \quad \text{isACK(rcvpkt)}
\end{align*}
\]

receiver

\[
\begin{align*}
\text{rdt_rcv(rcvpkt)} & \quad \text{corrupt(rcvpkt)} \\
\text{udt_send(NAK)} & \quad \text{Wait for call from below} \\
\text{rdt_rcv(rcvpkt)} & \quad \text{notcorrupt(rcvpkt)} \\
\text{extract(rcvpkt, data)} & \quad \text{deliver_data(data)} \\
\text{udt_send(ACK)} & \quad \text{Wait for ACK or NAK} \\
\end{align*}
\]

rdt2.0: operation with no errors

\[
\begin{align*}
\text{sender} & \quad \text{Halt} \\
\text{rdt_send(data)} \\
\text{snkpkt} = \text{make_pkt(data, checksum)} \\
\text{udt_send(sndpkt)} & \quad \text{rdt_rcv(rcvpkt)} & \quad \text{isNAK(rcvpkt)} \\
\text{udt_send(sndpkt)} & \quad \text{udt_send(ACK)} \\
\text{Wait for ACK or NAK} & \quad \text{Wait for call from above} \\
\text{rdt_rcv(rcvpkt)} & \quad \text{isACK(rcvpkt)}
\end{align*}
\]

receiver

\[
\begin{align*}
\text{rdt_rcv(rcvpkt)} & \quad \text{corrupt(rcvpkt)} \\
\text{udt_send(NAK)} & \quad \text{Wait for call from below} \\
\text{rdt_rcv(rcvpkt)} & \quad \text{notcorrupt(rcvpkt)} \\
\text{extract(rcvpkt, data)} & \quad \text{deliver_data(data)} \\
\text{udt_send(ACK)} & \quad \text{Wait for ACK or NAK} \\
\end{align*}
\]
**rdt2.0: error scenario**

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

**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

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land \text{notcorrupt(rcvpkt)} \\
&\land \text{isACK(rcvpkt)} \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\text{rdt} \_\text{rcv(rcvpkt)} &\land (\text{corrupt(rcvpkt)} \land \text{isNAK(rcvpkt)}) \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

Wait for call 0 from above

\[
\begin{align*}
\text{sndpkt} &\equiv \text{make_pkt}(0, \text{data}, \text{checksum}) \\
\text{udt} \_\text{send} &\equiv \text{sndpkt} \\
\end{align*}
\]

rdt Send(data)

Wait for ACK or NAK 0

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land (\text{corrupt(rcvpkt)} \land \text{isNAK(rcvpkt)}) \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land \text{notcorrupt(rcvpkt)}  \\
&\land \text{isACK(rcvpkt)} \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

Wait for call 1 from above

\[
\begin{align*}
\text{sndpkt} &\equiv \text{make_pkt}(1, \text{data}, \text{checksum}) \\
\text{udt} \_\text{send} &\equiv \text{sndpkt} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land (\text{corrupt(rcvpkt)} \land \text{isNAK(rcvpkt)}) \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

Transport Layer 3-31

rdt2.1: receiver, handles garbled ACK/NAKs

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land \text{notcorrupt(rcvpkt)} \\
&\land \text{has_seq0(rcvpkt)} \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

Wait for 0 from below

\[
\begin{align*}
\text{sndpkt} &\equiv \text{make_pkt}(\text{NAK}, \text{checksum}) \\
\text{udt} \_\text{send} &\equiv \text{sndpkt} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land (\text{corrupt(rcvpkt)} \land \text{has_seq1(rcvpkt)}) \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

Wait for 1 from below

\[
\begin{align*}
\text{sndpkt} &\equiv \text{make_pkt}(\text{ACK}, \text{checksum}) \\
\text{udt} \_\text{send} &\equiv \text{sndpkt} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land (\text{corrupt(rcvpkt)} \land \text{has_seq0(rcvpkt)}) \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt} \_\text{rcv(rcvpkt)} &\land (\text{corrupt(rcvpkt)} \land \text{has_seq1(rcvpkt)}) \\
&\land \text{udt} \_\text{send(sndpkt)} \\
\end{align*}
\]

\[
\begin{align*}
\text{extract} &\equiv \text{rcvpkt}.\text{data} \\
\text{deliver} &\equiv \text{data} \\
\text{sndpkt} &\equiv \text{make_pkt}(\text{ACK}, \text{checksum}) \\
\text{udt} \_\text{send} &\equiv \text{sndpkt} \\
\end{align*}
\]

Transport Layer 3-32
rdt2.1: discussion

Sender:
- seq # added to pkt
- two seq. #’s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
  - 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:
  
  \[ \text{retransmit current pkt} \]
**rdt2.2: sender, receiver fragments**

sender FSM fragment

\[
\begin{align*}
\text{rdt}_\text{send}(\text{data}) \\
\text{sndpkt} = \text{make}_\text{pkt}(0, \text{data}, \text{checksum}) \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\text{Wait for call 0 from above} \\
\text{Wait for ACK 0} \\
\end{align*}
\]

receiver FSM fragment

\[
\begin{align*}
\text{rdt}_\text{rcv}(\text{rcvpkt}) \\
\&\& \text{(corrupt(\text{rcvpkt}) || has_seq1(\text{rcvpkt}))} \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\text{Wait for 0 from below} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt}_\text{rcv}(\text{rcvpkt}) \\
\&\& \text{notcorrupt(\text{rcvpkt})} \\
\&\& \text{isACK(\text{rcvpkt},1)} \\
\text{udt}_\text{send}(\text{sndpkt}) \\
\end{align*}
\]

**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 sender

- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_rcv(rcvpkt)`
- `time_out`
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_send(data)`
- `sndpkt = make_pkt(1, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `stop_timer`
- `timeout`
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_rcv(rcvpkt)`
- `time_out`
- `udt_send(sndpkt)`
- `start_timer`
- `rdt_send(data)`
- `sndpkt = make_pkt(1, data, checksum)`
- `udt_send(sndpkt)`
- `start_timer`
- `stop_timer`
- `timeout`
- `udt_send(sndpkt)`
- `start_timer`

### rdt3.0 in action

(a) Operation with no loss

- `rdt_rcv(rcvpkt)`
- `& & notcorrupt(rcvpkt)`
- `& & isACK(rcvpkt, 0)`
- `stop_timer`
- `timeout`
- `udt_send(sndpkt)`
- `start_timer`

(b) Lost packet

- `rdt_rcv(rcvpkt)`
- `& & notcorrupt(rcvpkt)`
- `& & isACK(rcvpkt, 0)`
- `stop_timer`
- `timeout`
- `udt_send(sndpkt)`
- `start_timer`
**rdt3.0 in action**

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

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

\[
U_{\text{sender}} = \frac{L}{R} (\text{transmission rate, bps}) = \frac{0.008}{30.008} = 0.00027
\]

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

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

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, $t = L / R$
- ACK arrives, send next packet, $t = RTT + L / R$

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

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

- If $(\text{nextseqnum} < \text{base} + N)$:
  - $\text{sndpkt}[$nextseqnum$] = \text{make_pkt}($nextseqnum$, \text{data}, \text{chksum})$
  - $\text{udt_send}($sndpkt$[$nextseqnum$])$
- If $(\text{base} == \text{nextseqnum})$
  - $\text{start_timer}$
  - nextseqnum++
- Else $\text{refuse_data(data)}$

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

If $\text{base} == \text{nextseqnum}$
- $\text{stop_timer}$
- Else $\text{start_timer}$

- $\text{rdt_rcv(rcvpkt)}$ && $\text{notcorrupt(rcvpkt)}$
- $\text{base} = \text{getacknum(rcvpkt)} + 1$
- If $(\text{base} == \text{nextseqnum})$
  - $\text{stop_timer}$
  - Else $\text{start_timer}$
- $\text{udt_send(sndpkt)}$

 Ack-only: always send ACK for correctly-received pkt with highest in-order seq #
- May generate duplicate ACKs
- Need only remember $\text{expectedseqnum}$

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

Transport Layer 3-45

GBN: receiver extended FSM

- $\text{default}$
  - $\text{udt_send(sndpkt)}$

- $\text{rdt_rcv(rcvpkt)}$ && $\text{notcorrupt(rcvpkt)}$
  - $\text{extract(rcvpkt,data)}$
  - $\text{deliver_data(data)}$
  - $\text{sndpkt} = \text{make_pkt}($expectedseqnum$, \text{ACK}, \text{chksum})$
  - $\text{udt_send(sndpkt)}$
  - $\text{expectedseqnum}++$

- $\text{expectedseqnum} = 1$

Transport Layer 3-46
GBN in action

Transport Layer 3-47

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

Transport Layer 3-48
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat

**sender**
- data from above:
  - if next available seq # in window, send pkt
- timeout(n):
  - resend pkt n, restart timer
- ACK(n) in [sendbase, sendbase+N]:
  - mark pkt n as received
  - if n smallest unACKed pkt, advance window base to next unACKed seq #

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

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) init’s sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver
### TCP segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number of the first byte in the segment's data</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number of the next byte expected from the other side</td>
</tr>
<tr>
<td>Urg data pnter</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>Additional options such as URG, ACK, PSH, RST, SYN, FIN</td>
</tr>
<tr>
<td>application data</td>
<td>Variable length data that is the actual application data</td>
</tr>
<tr>
<td>receive window</td>
<td>Receiver's window size for data acceptance</td>
</tr>
<tr>
<td>checksum</td>
<td>Internet checksum, as in UDP</td>
</tr>
</tbody>
</table>

**URG**: urgent data (generally not used)

**ACK**: ACK # valid

**PSH**: push data now (generally not used)

**RST, SYN, FIN**: connection establishment (setup, teardown commands)

**Internet checksum**: (as in UDP)

### 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

**Simple telnet scenario**

- Host A sends Seq=42, ACK=79, data = ‘C’
- Host B sends Seq=43, ACK=80
- Host B receives Seq=42, Ack=79, data = ‘C’
- Host B sends Seq=43, ACK=80
- Host A receives Seq=43, Ack=80

User types ‘C’

- Host A sends Seq=79, ACK=43, data = ‘C’
- Host B sends Seq=43, ACK=80
- Host B receives Seq=79, Ack=43, data = ‘C’
- Host B sends Seq=43, ACK=80
- Host A receives Seq=43, Ack=80

User types ‘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**

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr

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 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

---

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 */
```

Comment:
- SendBase-1: last cumulatively acked 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=92, 8 bytes data
ACK=100

SendBase = 100

time

lost ACK scenario

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

Seq=100, 20 bytes data
ACK=120

SendBase = 120

premature timeout

TCP retransmission scenarios (more)

Host A
Seq=92, 8 bytes data
ACK=120

Seq=100, 20 bytes data
ACK=120

SendBase = 120

Cumulative ACK scenario
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 starts at 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:

**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:
  - sender won’t overflow receiver’s buffer by transmitting too much, too fast
  - app process may be slow at reading from buffer

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

TCP Flow control: how it works

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

- spare room in buffer
  - \[ \text{spare room} = \text{RcvBuffer} - (\text{LastByteRcvd} - \text{LastByteRead}) \]

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
  - guarantees receive buffer doesn’t overflow
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
  - SOCKERC\function = 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();

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

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

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

![Diagram](Transport Layer 3-81)

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 $\lambda_{in}$

```
$\lambda_{out} > \lambda_{in}$
```

“costs” of congestion:

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

![Graphs](Transport Layer 3-82)
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

$\lambda_{\text{in}}$: what happens as $\lambda_{\text{in}}$ increase?

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
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: 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

![Graph showing saw tooth behavior]

TCP Congestion Control: details

- sender limits transmission: $\text{LastByteSent-LastByteAcked} \leq \text{CongWin}$
- Roughly, $\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/second}$
- $\text{CongWin}$ 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

- When connection begins, \( \text{CongWin} = 1 \) MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
- available bandwidth may be \( \gg \) 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 \( \text{CongWin} \) every RTT
  - done by incrementing \( \text{CongWin} \) for every ACK received
- **Summary:** initial rate is slow but ramps up exponentially fast
Q: When should the exponential increase switch to linear?

A: When $\text{CongWin}$ 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:
  - $\text{CongWin}$ is cut in half
  - window then grows linearly
- But after timeout event:
  - $\text{CongWin}$ 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 \texttt{CongWin} is below \texttt{Threshold}, sender is in slow-start phase, window grows exponentially.
- When \texttt{CongWin} is above \texttt{Threshold}, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, \texttt{Threshold} set to \texttt{CongWin}/2 and \texttt{CongWin} set to \texttt{Threshold}.
- When timeout occurs, \texttt{Threshold} set to \texttt{CongWin}/2 and \texttt{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>\texttt{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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Congestion Avoidance (CA)</td>
<td>ACK receipt for previously unacked data</td>
<td>\texttt{CongWin = CongWin + MSS * (MSS/CongWin)}</td>
<td>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS or CA</td>
<td>Loss event detected by triple duplicate ACK</td>
<td>\texttt{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></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS or CA</td>
<td>Timeout</td>
<td>\texttt{Threshold = CongWin/2, CongWin = 1 MSS, Set state to &quot;Slow Start&quot;}</td>
<td>Enter slow start</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SS or CA</td>
<td>Duplicate ACK</td>
<td>\texttt{CongWin = CongWin + MSS, if (CongWin &gt; Threshold) set state to &quot;Congestion Avoidance&quot;}</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 throughout: .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:
  \[
  \frac{1.22 \cdot MSS}{RTT \sqrt{L}}
  \]
- New versions of TCP for high-speed needed!
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
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 $R/2$ !

### 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

\[
\text{delay} = 2\text{RTT} + \frac{O}{R} + (K-1)[S/R + RTT - WS/R]
\]
TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

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

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)

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

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

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

\[
\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} \text{idleTime}_p \\
= \frac{O}{R} + 2RTT + \sum_{k=1}^{P} \left( \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right) \\
= \frac{O}{R} + 2RTT + \left[ RTT + \frac{S}{R} \right] - \left( 2^{P} - 1 \right) \frac{S}{R}
\]

TCP Delay Modeling (4)

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

How do we calculate \( K \)?

\[
K = \min \left\{ k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O \right\} \\
= \min \left\{ k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq \frac{O}{S} \right\} \\
= \min \left\{ k : k \geq \log_2 \left( \frac{O}{S} + 1 \right) \right\} \\
= \left\lceil \log_2 \left( \frac{O}{S} + 1 \right) \right\rceil
\]

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)

RTT = 100 msec, $O$ = 5 Kbytes, $M$=10 and $X$=5

For low bandwidth, connection & response time dominated by transmission time.
Persistent connections only give minor improvement over parallel connections.
For larger RTT, response time dominated by TCP establishment & slow start delays. Persistent connections now give important improvement: particularly in high delay-bandwidth networks.

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”