Set 3
Chapter 3: Transport Layer

our goals:

- understand principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- learn about Internet transport layer protocols:
  - UDP: connectionless transport
  - TCP: connection-oriented reliable 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
  - recv 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 in Ann’s house sending letters to 12 kids in Bill’s house:
- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service
Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
   ▪ segment structure
   ▪ reliable data transfer
   ▪ flow control
   ▪ connection management
3.6 principles of congestion control
3.7 TCP congestion control
**Multiplexing/demultiplexing**

**Multiplexing at sender:** handle data from multiple sockets, add transport header (later used for demultiplexing)

**Demultiplexing at receiver:** use header info to deliver received segments to correct socket
How demultiplexing works

- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries one 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

- Source port #
- Destination port #
- Other header fields
- Application data (payload)
- 32 bits
Connectionless demultiplexing

- **recall**: created socket has host-local port #:
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  ```

- **recall**: when creating datagram to send into UDP socket, must specify:
  - destination IP address
  - destination port #

- when host receives UDP segment:
  - checks destination port # in segment
  - directs UDP segment to socket with that port #

- IP datagrams with *same dest. port #*, but different source IP addresses and/or source port numbers will be directed to *same socket* at dest
Connectionless demux: example

DatagramSocket
mySocket2 = new DatagramSocket (9157);

DatagramSocket
serverSocket = new DatagramSocket (6428);

DatagramSocket
mySocket1 = new DatagramSocket (5775);

source port: 9157
dest port: 6428

source port: 6428
dest port: 9157

source port: ?
dest port: ?

source port: ?
dest port: ?

Transport Layer 3-11
Connection-oriented demux

- **TCP socket identified by 4-tuple:**
  - source IP address
  - source port number
  - dest IP address
  - dest port number

- **demux:** receiver 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: example

three segments, all destined to IP address: B, dest port: 80 are demultiplexed to different sockets
Connection-oriented demux: example

Transport Layer 3-14

Source IP, port: A, 9157
Dest IP, port: B, 80

Source IP, port: B, 80
Dest IP, port: A, 9157

Source IP, port: C, 5775
Dest IP, port: B, 80

Source IP, port: C, 9157
Dest IP, port: B, 80

Threaded server

Host: IP address A

Host: IP address C

Server: IP address B
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 use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
- reliable transfer over UDP:
  - add reliability at application layer
  - application-specific error recovery!
UDP: segment header

UDP segment format

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

- 32 bits
- length, in bytes of UDP segment, including header

why is there a UDP?

- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired
UDP checksum

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

**sender:**
- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one’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.

...
Internet checksum: example

device: add two 16-bit integers

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

wraparound

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

Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
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 application, transport, link layers
  - Top-10 list of important networking topics!

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

(a) provided service
Principles of reliable data transfer

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

Characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: getting started

**rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

**udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**deliver_data()**: called by rdt to deliver data to upper

**rdt_rcv()**: called when packet arrives on rcv-side of channel
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:** when in this “state” next state uniquely determined by next event

**event causing state transition**

**actions taken on state transition**

**event**

**actions**
rdt 1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver reads data from underlying channel

```
Wait for call from above

packet = make_pkt(data)
udt_send(packet)

rdt_send(data)

sender
```

```
Wait for call from below

extract (packet, data)
delver_data(data)

rdt_rcv(packet)

receiver
```
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:

How do humans recover from “errors” during conversation?
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
  - feedback: control msgs (ACK, NAK) from receiver to sender
**rdt2.0: FSM specification**

**sender**

- `rdt_send(data)`
- `sndpkt = make_pkt(data, checksum)`
- `udt_send(sndpkt)`

**receiver**

- `rdt_rcv(rcvpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `udt_send(ACK)`
- `udt_send(NAK)`

**Finite State Machine Diagram**

- `rdt_send(data)`
  - `sndpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt)`
  - `&& isACK(rcvpkt)`
  - `&& isNAK(rcvpkt)`

- `udt_send(sndpkt)`

- `wait for call from above`

- `wait for ACK or NAK`

- `wait for call from below`
rdt2.0: operation with no errors

\[ \text{rdt_send(data)} \]
\[ \text{snkpkt} = \text{make_pkt(data, checksum)} \]
\[ \text{udt_send(sndpkt)} \]

- Wait for call from above

- rdt_rcv(rcvpkt) && isNAK(rcvpkt) 
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && isACK(rcvpkt)

- Wait for ACK or NAK

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - udt_send(ACK)

- Wait for call from below

- rdt_rcv(rcvpkt) && corrupt(rcvpkt)
  - udt_send(NAK)
**rdt2.0: error scenario**

```plaintext
rdt_send(data)

snkpkt = make_pkt(data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
isNAK(rcvpkt)
udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
corrupt(rcvpkt)
udt_send(NAK)

Wait for call from above

Wait for ACK or NAK

Wait for call from below

rdt_rcv(rcvpkt) &&
isACK(rcvpkt)
udt_send(ACK)

extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

Transport Layer 3-31
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 corrupted
- sender adds *sequence number* to each pkt
- receiver discards (doesn’t deliver up) duplicate pkt

---

**stop and wait**
sender sends one packet, then waits for receiver response
**rdt2.1: sender, handles garbled ACK/NAKs**

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

- **Wait for call 0 from above**
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`

- **Wait for ACK or NAK 0**
  - `rdt_send(data)`
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`

- **Wait for call 1 from above**
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`

- **Wait for ACK or NAK 1**
  - `rdt_send(data) && (corrupt(rcvpkt) || isNAK(rcvpkt))`
  - `udt_send(sndpkt)`
rdt2.1: receiver, handles garbled ACK/NAKs

- `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
  - `sndpkt = make_pkt(NAK, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt)`
  - `sndpkt = make_pkt(NAK, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)`
  - `sndpkt = make_pkt(ACK, checksum)`
  - `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) && (corrupt(rcvpkt)`
  - `sndpkt = make_pkt(NAK, checksum)`
  - `udt_send(sndpkt)`
rdt2.1: discussion

sender:
- seq # added to pkt
- two seq. #'s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “expected” pkt should have seq # of 0 or 1

receiver:
- must check if received packet is duplicate
  - state indicates whether 0 or 1 is expected pkt seq #
- note: receiver can not know if its last ACK/NAK received OK at sender
rdt2.2: a NAK-free protocol

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must explicitly include seq # of pkt being ACKed
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
**rdt2.2: sender, receiver fragments**

**Sender FSM fragment**

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

- **Wait for call 0 from above**

- **rdt_rcv(rcvpkt) &&**
  - (corrupt(rcvpkt) || has_seq1(rcvpkt))
  - `udt_send(sndpkt)`

**Receiver FSM fragment**

- **Wait for 0 from below**

- **rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)**

- **extract(rcvpkt, data)**
  - **deliver_data(data)**

- **sndpkt = make_pkt(ACK1, checksum)**
  - `udt_send(sndpkt)`

- **Wait for ACK 0**

- **rdt_send(sndpkt)**

- **rdt_rcv(rcvpkt) && isACK(rcvpkt, 1)**
  - `udt_send(sndpkt)`

- **rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)**

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

**new assumption:** underlying channel can also lose packets (data, 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 seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed
- requires countdown timer
rdt3.0 sender

- **Start**: rdt3.0 sender
- **Function**: rdt_send(data)
  - **Action**: sndpkt = make_pkt(0, data, checksum)
  - **Action**: udt_send(sndpkt)
  - **Action**: start_timer

- **Next State**: Wait for call 0 from above
  - **Condition**: rdt_rcv(rcvpkt)
  - **Action**: Wait for call 0 from above

- **Next State**: rdt_send(data)
  - **Condition**: (corrupt(rcvpkt) || isACK(rcvpkt, 1))

- **Next State**: Wait for ACK0
  - **Action**: rdt_send(data)
  - **Condition**: notcorrupt(rcvpkt) && isACK(rcvpkt, 1)
  - **Action**: stop_timer

- **Next State**: Wait for call 1 from above
  - **Condition**: rdt_rcv(rcvpkt)
  - **Action**: Wait for call 1 from above

- **Next State**: rdt_send(data)
  - **Condition**: (corrupt(rcvpkt) || isACK(rcvpkt, 0))

- **Next State**: Wait for ACK1
  - **Action**: rdt_send(data)
  - **Condition**: notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
  - **Action**: stop_timer

- **Next State**: Wait for call 0 from above
  - **Condition**: rdt_rcv(rcvpkt)
  - **Action**: Wait for call 0 from above

- **Next State**: rdt_send(data)
  - **Condition**: (corrupt(rcvpkt) || isACK(rcvpkt, 0))

- **Next State**: Wait for call 1 from above
  - **Condition**: rdt_rcv(rcvpkt)
  - **Action**: Wait for call 1 from above

- **Next State**: rdt_send(data)
  - **Condition**: (corrupt(rcvpkt) || isACK(rcvpkt, 1))

- **Next State**: Wait for ACK0
  - **Action**: rdt_send(data)
  - **Condition**: notcorrupt(rcvpkt) && isACK(rcvpkt, 1)
  - **Action**: stop_timer
**rdt3.0 in action**

(a) no loss

- **sender**
  - send pkt0
  - rcv ack0
  - send pkt1
  - rcv ack1
  - send pkt0

- **receiver**
  - pkt0
  - rcv pkt0
  - send ack0
  - rcv pkt1
  - send ack1
  - rcv pkt0
  - send ack0

(b) packet loss

- **sender**
  - send pkt0
  - rcv ack0
  - send pkt1
  - timeout
  - resend pkt1

- **receiver**
  - pkt0
  - rcv pkt0
  - send ack0
  - rcv ack0
  - send ack1
  - rcv pkt0
  - send ack0

Transport Layer 3-40
rdt3.0 in action

(sender)
send pkt0
rcv pkt0
send ack0
send pkt1
rcv pkt1
send ack1
rcv ack1
send pkt0
rcv pkt0
send ack0

(receiver)
send pkt0
rcv pkt0
send ack0
rcv pkt1
send ack1
rcv ack1
send pkt0
rcv pkt0
send ack0

(c) ACK loss

(d) premature timeout/ delayed ACK
Performance of rdt3.0

- rdt3.0 is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[ D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \]

- **U_{\text{sender}}**: utilization – fraction of time sender busy sending

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

- if RTT=30 msec, 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**

- First packet bit transmitted, $t = 0$
- Last packet bit transmitted, $t = \frac{L}{R}$
- First packet bit arrives
- Last packet bit arrives, send ACK
- ACK arrives, send next packet, $t = \text{RTT} + \frac{L}{R}$

**Equation:**

$$U_{\text{sender}} = \frac{L}{R} \quad \frac{\text{RTT}}{\frac{L}{R}} = \frac{.008}{30.008} = 0.00027$$
Pipelined protocols

pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver

- two generic forms of pipelined protocols: go-Back-N, selective repeat
Pipelining: increased utilization

first packet bit transmitted, $t = 0$
last bit transmitted, $t = L / R$

first packet bit arrives
last packet bit arrives, send ACK
last bit of 2$^{nd}$ packet arrives, send ACK
last bit of 3$^{rd}$ packet arrives, send ACK

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

$U_{sender} = \frac{3L / R}{RTT + L / R} = \frac{0.0024}{30.008} = 0.00081$

3-packet pipelining increases utilization by a factor of 3!
Pipelined protocols: overview

Go-back-N:
- sender can have up to N unacknowledged packets in pipeline
- receiver only sends cumulative ack
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacknowledged packet
  - when timer expires, retransmit all unacknowledged packets

Selective Repeat:
- sender can have up to N unacknowledged packets in pipeline
- rcvr sends individual ack for each packet
- sender maintains timer for each unacknowledged packet
  - when timer expires, retransmit only that unacknowledged packet
Go-Back-N: sender

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed

![](image)

- `send_base`
- `nextseqnum`
- `already ack’ed`
- `sent, not yet ack’ed`
- `usable, not yet sent`
- `not usable`

- `window size N`

- `ACK(n)`: ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- `timeout(n)`: retransmit packet n and all higher seq # pkts in window
GBN: sender extended FSM

```
rdt_send(data)
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
    nextseqnum++
} else
    refuse_data(data)
else
    timeout
    start_timer
    udt_send(sndpkt[base])
    udt_send(sndpkt[base+1])
    ...
    udt_send(sndpkt[nextseqnum-1])
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
base = getacknum(rcvpkt) + 1
if (base == nextseqnum)
    stop_timer
else
    start_timer
```
ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #

- may generate duplicate ACKs
- need only remember `expectedseqnum`

* out-of-order pkt:
  - discard (don’t buffer): *no receiver buffering!*
  - re-ACK pkt with highest in-order seq #
**GBN in action**

**sender window (N=4)**

<table>
<thead>
<tr>
<th>0 1 2 3</th>
<th>4 5 6 7 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8</td>
</tr>
<tr>
<td>0 1 2 3</td>
<td>4 5 6 7 8</td>
</tr>
</tbody>
</table>

**sender**

- send pkt0
- send pkt1
- send pkt2
- send pkt3
  - (wait)

**receiver**

- receive pkt0, send ack0
- receive pkt1, send ack1
- receive pkt3, discard, (re)send ack1
- receive pkt4, discard, (re)send ack1
- receive pkt5, discard, (re)send ack1
- receive pkt2, deliver, send ack2
- receive pkt3, deliver, send ack3
- receive pkt4, deliver, send ack4
- receive pkt5, deliver, send ack5

**pkt 2 timeout**

- ignore duplicate ACK
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
  - limits seq #s of sent, unACKed pkts
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**

Packet 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

Packet n in [rcvbase-N, rcvbase-1]:
- ACK(n)

Otherwise:
- ignore
Selective repeat in action

sender window (N=4)

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1
receive pkt3, buffer,

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived

record ack4 arrived

pkt 2 timeout

send pkt2

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?
Selective repeat: dilemma

example:
- seq #’s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?
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 steam:**
  - 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) inits sender, receiver state before data exchange

- **flow controlled:**
  - sender will not overwhelm receiver
TCP segment structure

- **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)

<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 segment</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number of the</td>
</tr>
<tr>
<td></td>
<td>previous segment received</td>
</tr>
<tr>
<td>head len</td>
<td>Header length (not used)</td>
</tr>
<tr>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>UAPR</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>SF</td>
<td></td>
</tr>
<tr>
<td>receive window</td>
<td>Receive window size</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>options (variable length)</td>
<td></td>
</tr>
<tr>
<td>application data</td>
<td>Application data (variable length)</td>
</tr>
<tr>
<td>(variable length)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- Counting by bytes of data (not segments!)
- # bytes rcvr willing to accept
- Transport Layer 3-58
TCP seq. numbers, ACKs

**sequence numbers:**
- byte stream “number” of first byte in segment’s data

**acknowledgements:**
- 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 seq. numbers, ACKs

simple telnet scenario
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, timeout

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

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$
TCP round trip time, timeout

- **timeout interval**: EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT → larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]
  (typically, $\beta = 0.25$)

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
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
- single retransmission timer

Retransmissions triggered by:
- timeout events
- duplicate acks

Let’s 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 ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - start timer if there are still unacked segments
TCP sender (simplified)

wait for event

\[ \text{NextSeqNum} = \text{InitialSeqNum} \]
\[ \text{SendBase} = \text{InitialSeqNum} \]

- create segment, seq. #: NextSeqNum
- pass segment to IP (i.e., “send”)
- NextSeqNum = NextSeqNum + length(data)
- if (timer currently not running)
  - start timer

data received from application above

- if (y > SendBase) {
  - SendBase = y
  /* SendBase–1: last cumulatively ACKed byte */
- if (there are currently not-yet-acked segments)
  - start timer
- else stop timer
}

- timeout
- retransmit not-yet-acked segment with smallest seq. #
- start timer

ACK received, with ACK field value y
TCP: retransmission scenarios

lost ACK scenario

Host A

<table>
<thead>
<tr>
<th>Seq=92, 8 bytes of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK=100</td>
</tr>
<tr>
<td>timeout</td>
</tr>
<tr>
<td>Seq=92, 8 bytes of data</td>
</tr>
<tr>
<td>ACK=100</td>
</tr>
</tbody>
</table>

Host B

premature timeout

Host A

| SendBase=92            |
| Seq=92, 8 bytes of data |
| ACK=100                |
| SendBase=100           |
| Seq=100, 20 bytes of data |
| ACK=100                |
| SendBase=120           |
| Seq=92, 8 bytes of data |
| ACK=120                |

Host B
TCP: retransmission scenarios

Host A

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data

Host B

ACK=100

ACK=120

Seq=120, 15 bytes of data

cumulative ACK
<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 <em>duplicate ACK</em>, 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 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.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #
  - likely that unacked segment lost, so don’t wait for timeout
TCP fast retransmit

Host A
Seq=92, 8 bytes of data
Seq=100, 20 bytes of data
ACK=100
ACK=100
ACK=100
ACK=100

Host B

fast retransmit after sender receipt of triple duplicate ACK
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

flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP flow control

- receiver “advertises” free buffer space by including rwnd value in TCP header of receiver-to-sender segments
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust RcvBuffer
- sender limits amount of unacked (“in-flight”) data to receiver’s rwnd value
- guarantees receive buffer will not 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
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

```
Socket clientSocket = newSocket("hostname","port number");
```

```
Socket connectionSocket = welcomeSocket.accept();
```
Agreeing to establish a connection

2-way handshake:

Q: Will 2-way handshake always work in network?

- variable delays
- retransmitted messages (e.g. req_conn(x)) due to message loss
- message reordering
- can’t “see” other side
Agreeing to establish a connection

2-way handshake failure scenarios:

- Choose x
- req_conn(x)
- ESTAB
- acc_conn(x)
- connection x completes
- client terminates
- half open connection! (no client!)

- req_conn(x)
- ESTAB
- data(x+1)
- accept data(x+1)
- client terminates
- req_conn(x)
- ESTAB
- data(x+1)
- accept data(x+1)
- server forgets x
TCP 3-way handshake

**Client state**
- **LISTEN**
  - choose init seq num, $x$
  - send TCP SYN msg

- **SYNSENT**
  - received SYNACK($x$)
  - indicates server is live
  - send ACK for SYNACK;
  - this segment may contain client-to-server data

- **ESTAB**
  - SYNbit=1, Seq=$x$
  - SYNbit=1, Seq=$y$
  - ACKbit=1; ACKnum=$x+1$
  - ACKbit=1, ACKnum=$y+1$

**Server state**
- **LISTEN**
- **SYN RCVD**
  - choose init seq num, $y$
  - send TCP SYNACK msg, acking SYN
  - SYNbit=1, Seq=$y$
  - ACKbit=1; ACKnum=$x+1$

- **ESTAB**
  - received ACK($y$)
  - indicates client is live
TCP 3-way handshake: FSM

Socket connectionSocket = welcomeSocket.accept();

SYN(x)
SYNACK(seq=y, ACKnum=x+1)
create new socket for communication back to client

ACK(ACKnum=y+1)
Λ

Socket clientSocket = newSocket("hostname","port number");

SYN(seq=x)

SYN(rcvd)

SYN(sent)

SYNACK(seq=y, ACKnum=x+1)
ACK(ACKnum=y+1)
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled
TCP: closing a connection

**client state**
- ESTAB
  - FIN_WAIT_1
    - FINbit=1, seq=x can no longer send but can receive data
  - FIN_WAIT_2
    - wait for server close
  - TIMED_WAIT
    - timed wait for 2*max segment lifetime
  - CLOSED

**server state**
- ESTAB
  - FINbit=1, seq=x
  - ACKbit=1; ACKnum=x+1
  - can still send data
- CLOSE_WAIT
  - FINbit=1, seq=y
  - ACKbit=1; ACKnum=y+1
  - can no longer send data
- LAST_ACK
  - can no longer send data
- CLOSED

**Actions**
- clientSocket.close()
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
- output link capacity: R
- no retransmission

Original data: $\lambda_{\text{in}}$

Throughput: $\lambda_{\text{out}}$

Unlimited shared output link buffers

Host A

Host B

$\frac{R}{2}$

$\lambda_{\text{out}}$

$\lambda_{\text{in}}$

$\frac{R}{2}$

- maximum per-connection throughput: $\frac{R}{2}$
- large delays as arrival rate, $\lambda_{\text{in}}$, approaches capacity
Causes/costs of congestion: scenario 2

- one router, \textit{finite} buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
  - transport-layer input includes retransmissions: $\lambda_{in}' \geq \lambda_{in}$
Causes/costs of congestion: scenario 2

idealization: perfect knowledge

- sender sends only when router buffers available

\[ \lambda_{\text{in}} : \text{original data} \]
\[ \lambda'_{\text{in}} : \text{original data, plus retransmitted data} \]

\[ \lambda_{\text{out}} \]

\[ \frac{R}{2} \]

Host B

finite shared output link buffers

free buffer space!
**Causes/costs of congestion: scenario 2**

**Idealization: known loss**
packets can be lost, dropped at router due to full buffers

- sender only resends if packet known to be lost

---

\( \lambda_{\text{in}} : \) original data
\( \lambda'_{\text{in}} : \) original data, plus retransmitted data

Host B

no buffer space!
Causes/costs of congestion: scenario 2

**Idealization: known loss**
packets can be lost, dropped at router due to full buffers

- sender only resends if packet known to be lost

![Diagram showing network traffic and buffer spaces with annotations for original and retransmitted data, and the impact on congestion when sending at R/2, with asymptotic goodput still R/2.](transport_layer_3-90_diagram.png)
Causes/costs of congestion: scenario 2

**Realistic: duplicates**

- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

\[ \lambda_{\text{in}} \]
\[ \lambda'_{\text{in}} \]
\[ \lambda_{\text{out}} \]

when sending at R/2, some packets are retransmissions including duplicated that are delivered!

Timeout

Free buffer space!
Causes/costs of congestion: scenario 2

Realistic: *duplicates*
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending *two* copies, both of which are delivered

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

Diagram:
- $\lambda_{out}$ vs $\lambda_{in}$
- when sending at R/2, some packets are retransmissions including duplicated that are delivered!
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{in}$ and $\lambda_{in}'$ increase?
A: as red $\lambda_{in}'$ increases, all arriving blue pkts at upper queue are dropped, blue throughput $\to 0$
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 for sender to 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
  - senders’ send rate thus max supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, receiver 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**: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - **additive increase**: increase $cwnd$ by 1 MSS every RTT until loss detected
  - **multiplicative decrease**: cut $cwnd$ in half after loss

AIMD saw tooth behavior: probing for bandwidth
TCP Congestion Control: details

- **sender sequence number space**
  - `cwnd`

- **TCP sending rate:**
  - *roughly:* send `cwnd` bytes, wait RTT for ACKS, then send more bytes

- **sender limits transmission:**
  \[
  \text{LastByteSent} - \text{LastByteAcked} \leq cwnd
  \]

- **`cwnd`** is dynamic, function of perceived network congestion

- **rate**
  \[
  \text{rate} \approx \frac{cwnd}{RTT} \text{ bytes/sec}
  \]
TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially \( cwnd = 1 \) MSS
  - double \( cwnd \) every RTT
  - done by incrementing \( cwnd \) for every ACK received

- **summary**: initial rate is slow but ramps up exponentially fast
TCP: detecting, reacting to loss

- loss indicated by timeout:
  - cwnd set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - cwnd is cut in half window then grows linearly
- TCP Tahoe always sets cwnd to 1 (timeout or 3 duplicate acks)
TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when \texttt{cwnd} gets to 1/2 of its value before timeout.

Implementation:
- variable \texttt{ssthresh}
- on loss event, \texttt{ssthresh} is set to 1/2 of \texttt{cwnd} just before loss event
Summary: TCP Congestion Control

**Slow Start**
- $cwnd = 1 \text{ MSS}$
- $ssthresh = 64 \text{ KB}$
- $\text{dupACKcount} = 0$
- $\text{timeout}$
- $ssthresh = cwnd/2$
- $cwnd = 1 \text{ MSS}$
- $\text{dupACKcount} = 0$
- $\text{retransmit missing segment}$
- $\text{dupACKcount} == 3$
- $ssthresh = cwnd/2$
- $cwnd = ssthresh + 3$
- $\text{retransmit missing segment}$

**Congestion Avoidance**
- $cwnd \geq ssthresh$
- $\text{timeout}$
- $ssthresh = cwnd/2$
- $cwnd = 1 \text{ MSS}$
- $\text{dupACKcount} = 0$
- $\text{retransmit missing segment}$
- $\text{dupACKcount} == 3$
- $ssthresh = cwnd/2$
- $cwnd = ssthresh + 3$
- $\text{retransmit missing segment}$

**Fast Recovery**
- $\text{duplicate ACK}$
- $cwnd = cwnd + \text{MSS}$
- $\text{transmit new segment(s), as allowed}$

**New ACK**
- $cwnd = cwnd + \text{MSS} \cdot (\text{MSS}/cwnd)$
- $\text{transmit new segment(s), as allowed}$
- $\text{dupACKcount} = 0$
- $\text{new ACK}$
TCP throughput

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send

- $W$: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is $\frac{3}{4} W$
  - avg. throughput is $\frac{3}{4}W$ per RTT

$$\text{avg TCP throughput} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$
TCP Futures: TCP over “long, fat pipes”

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires $W = 83,333$ in-flight segments
- throughput in terms of segment loss probability, $L$ [Mathis 1997]:

$$\text{TCP throughput} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ to achieve 10 Gbps throughput, need a loss rate of $L = 2 \cdot 10^{-10}$ — a very small loss rate!

- new versions of TCP for high-speed
TCP Fairness

**fairness goal:** if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K
Why is TCP fair?

two competing sessions:
- additive increase gives slope of 1, as throughout increases
- multiplicative decrease decreases throughput proportionally

Diagram:
- Connection 1 throughput vs. Connection 2 throughput
- Equal bandwidth share
- Congestion avoidance: additive increase
- Loss: decrease window by factor of 2

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate $R$ with 9 existing 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, implementation in the Internet
  - UDP
  - TCP

next:
- leaving the network “edge” (application, transport layers)
- into the network “core”