CMPT 371: Data Communications and Networking

Chapter 3 – Transport Layer

Instructor: Joseph Peters

Slides based on Kurose & Ross, 7th Edition
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
  - 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 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

Diagram showing the layers of the network with process P1 and P2 communicating with the transport layer, which uses header info for demultiplexing.
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 #
- dest 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
serverSocket = new DatagramSocket (6428);

DatagramSocket
mySocket1 = new DatagramSocket (5775);

DatagramSocket
mySocket2 = new DatagramSocket (9157);

Source port: 9157
dest port: 6428

Source port: 6428
dest port: 9157

Source port: 9157
dest port: 9157

Source port: ?
dest port: ?

Source port: 9157
dest port: 6428

Source port: ?
dest port: ?
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
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**

<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**

UDP segment format

**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.

But maybe errors nonetheless? More later….
example: add two 16-bit integers

\[
\begin{array}{cccccccccccc}
1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1
\end{array}
\]

\[
\begin{array}{cccccccccccc}
1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
\end{array}
\]

*Note: when adding numbers, a carryout from the most significant bit needs to be added to the result*

* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)
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!

(a) provided service

- 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 (rtd)
**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

**send side**

- **rdt_send()**: called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- **udt_send()**: called by rdt, to transfer packet over unreliable channel to receiver

**receive side**

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

sender

receiver

\[ \text{Wait for call from above} \]
\[ \text{rdt\_send(data)} \]
\[ \text{packet = make\_pkt(data)} \]
\[ \text{udt\_send(packet)} \]

\[ \text{Wait for call from below} \]
\[ \text{rdt\_recv(packet)} \]
\[ \text{extract (packet, data)} \]
\[ \text{deliver\_data(data)} \]
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:

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)`

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

- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`
  - `Lambda` (denoted by Λ)

**receiver**

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

- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
  - `udt_send(NAK)`

**States and Transitions**

- Wait for call from above
- Wait for ACK or NAK

Transport Layer 3-29
rdt2.0: operation with no errors

```
r2dt_send(data)
  snkpkt = make_pkt(data, checksum)
  udt_send(sndpkt)

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

  rdt_rcv(rcvpkt) &&
  isACK(rcvpkt)

  Wait for
  call from
  above

  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)
  rdt_send(data)
```

Wait for
ACK or
NAK

Transport Layer 3-30
**rdt2.0: error scenario**

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

**Wait for call from above**

**Wait for ACK or NAK**

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

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

**Wait for call from below**

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

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

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ

Wait for call 0 from above

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)

Wait for ACK or NAK 0

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ

Wait for call 1 from above

Wait for ACK or NAK 1

rdt_send(data)

sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && ( 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, chksum)
        udt_send(sndpkt)
```

```
rdt2.1: receiver, handles garbled ACK/NAKs

```
```
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:

- \( \text{rdt}\_\text{send}(\text{data}) \)
- \( \text{sndpkt} = \text{make}\_\text{pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt}\_\text{send}(\text{sndpkt}) \)
- Wait for call 0 from above

receiver FSM fragment:

- \( \text{rdt}\_\text{rcv}(\text{rcvpkt}) \) \&\&
  - \( \text{has Seq1}(\text{rcvpkt})) \)
  - \( \text{udt}\_\text{send}(\text{sndpkt}) \)

- \( \text{extract}(\text{rcvpkt}, \text{data}) \)
- \( \text{deliver}\_\text{data}(\text{data}) \)

- \( \text{sndpkt} = \text{make}\_\text{pkt}(\text{ACK1}, \text{checksum}) \)
- \( \text{udt}\_\text{send}(\text{sndpkt}) \)
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**

![Diagram of rdt3.0 sender](image)

- **Wait for call 0 from above**
  - `rdt_rcv(rcvpkt)`
  - `Lambda`
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)`
  - `stop_timer`

- **Wait for ACK0**
  - `Wait for call 0 from above`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(0, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`
  - `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)`
  - `stop_timer`

- **Wait for call 1 from above**
  - `timeout`
  - `udt_send(sndpkt)`
  - `start_timer`

- **Wait for ACK1**
  - `Wait for call 1 from above`
  - `rdt_send(data)`
  - `sndpkt = make_pkt(1, data, checksum)`
  - `udt_send(sndpkt)`
  - `start_timer`

- **Error Handling**
  - `rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))`
  - `Lambda`
rdt3.0 in action

(a) no loss

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

(receiver)
pkt0
pkt1
ack0
ack1
ack0

(b) packet loss

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

(receiver)
pkt0
pkt1
ack0
ack1
ack0

Transport Layer 3-40
rdt3.0 in action

sender

send pkt0

rcv ack0

send pkt1

rcv ack1

send pkt0

rcv ack1

send pkt1

rcv pkt0

send ack0

rcv pkt1

send ack1

rcv pkt0

send ack0

rcv pkt1

send ack1

(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

\[ U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027 \]

- if RTT=30 msec, 1KB pkt every 30 msec: 33kB/sec thruput over 1 Gbps link

- network protocol limits use of physical resources!
**rdt3.0: stop-and-wait operation**

First packet bit transmitted, $t = 0$ 

Last packet bit transmitted, $t = L / R$ 

First packet bit arrives 

Last packet bit arrives, send ACK 

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

\[
U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027
\]
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, \( t = \frac{L}{R} \)
- 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, \( t = RTT + \frac{L}{R} \)

3-packet pipelining increases utilization by a factor of 3!

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

\( U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = 0.00081 \)
Pipelined protocols: overview

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

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

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

ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for 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) && corrupt(rcvpkt)

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

**sender**

- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)
- rcv ack0, send pkt4
- rcv ack1, send pkt5
- ignore duplicate ACK
- pkt 2 timeout

**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
- rcv pkt2, deliver, send ack2
- rcv pkt3, deliver, send ack3
- rcv pkt4, deliver, send ack4
- rcv pkt5, deliver, send ack5
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

send_base  nextseqnum

window size $N$

already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable

out of order (buffered) but already ack'ed
acceptable (within window)
Expected, not yet received
not usable
**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

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)?

Receiver can’t see sender side. Receiver behavior identical in both cases! Something’s (very) wrong!

sender window (after receipt)

receiver window (after receipt)

(a) no problem

(b) oops!
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

RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order *byte stream:**
  - no “message boundaries”

- **pipelined:**
  - TCP congestion and flow control set window size

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **connection-oriented:**
  - handshaking (exchange of control msgs) inits 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 of the sending host</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number of the receiving host</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number of the TCP segment</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number of the receiver</td>
</tr>
<tr>
<td>receive window</td>
<td>Receive window of the receiver</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum of the TCP segment</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Pointer to the end of the urgent data portion</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>Additional options that can be included in the segment</td>
</tr>
<tr>
<td>application data</td>
<td>Data portion that is application-specific</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) counting by bytes of data (not segments!)
- **rcvr willing to accept**: # bytes rcvr willing to accept
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

<table>
<thead>
<tr>
<th>sequence number</th>
<th>acknowledgement number</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>dest port #</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>checksum</th>
<th>urg pointer</th>
</tr>
</thead>
</table>

outgoing segment from sender

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledgement number</td>
</tr>
<tr>
<td>rwnd</td>
</tr>
</tbody>
</table>

window size $N$

sender sequence number space

sent | sent, not-yet ACKed (“in-flight”) | usable but not yet sent | not usable

incoming segment to sender

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>sequence number</th>
</tr>
</thead>
<tbody>
<tr>
<td>acknowledgement number</td>
</tr>
<tr>
<td>A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>checksum</th>
<th>urg pointer</th>
</tr>
</thead>
</table>
TCP seq. numbers, ACKs

User types ‘C’

Seq=42, ACK=79, data = ‘C’

Seq=79, ACK=43, data = ‘C’

Seq=43, ACK=80

host ACKs receipt of echoed ‘C’

host ACKs receipt of ‘C’, echoes back ‘C’

simple telnet scenario
TCP round trip time, 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, 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, $\beta = 0.25$)

`TimeoutInterval = EstimatedRTT + 4 \times \text{DevRTT}`

* Check out the online interactive exercises for more examples: [http://gaia.cs.umass.edu/kurose_ross/interactive/](http://gaia.cs.umass.edu/kurose_ross/interactive/)
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)

- NextSeqNum = InitialSeqNum
- SendBase = InitialSeqNum

wait for event

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

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

ACK received, with ACK field value y

timeout
retransmit not-yet-acked segment with smallest seq. #
start timer
TCP: retransmission scenarios

lost ACK scenario:
- Seq=92, 8 bytes of data
- ACK=100
- timeout
- Seq=92, 8 bytes of data
- ACK=100

premature timeout:
- SendBase=92
- Seq=92, 8 bytes of data
- ACK=100
- SendBase=120
- ACK=120
- Seq=100, 20 bytes of data
- ACK=100
- SendBase=120
- ACK=120
- Seq=92, 8 bytes of data
- ACK=120
TCP: retransmission scenarios

Host A

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

X

ACK=100

ACK=120

Seq=120, 15 bytes of data

Host B

Cumulative ACK
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</td>
<td>delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>expected seq # already ACKed</td>
<td></td>
</tr>
<tr>
<td>arrival of in-order segment with expected seq #. One other segment</td>
<td>immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>has ACK pending</td>
<td></td>
</tr>
<tr>
<td>arrival of out-of-order segment higher-than-expect seq. # . Gap</td>
<td>immediately send <em>duplicate ACK</em>, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>detected</td>
<td></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
Seq=100, 20 bytes of data

Host B

timeout

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.

Application process

TCP socket receiver buffers

TCP code

IP code

from sender

receiver protocol stack

Application may remove data from TCP socket buffers...

... slower than TCP receiver is delivering (sender is sending)

Transport Layer 3-74
TCP flow control

- receiver “advertises” free buffer space by including \texttt{rwnd} value in TCP header of receiver-to-sender segments
  - \texttt{RcvBuffer} size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust \texttt{RcvBuffer}
- sender limits amount of unacked (“in-flight”) data to receiver’s \texttt{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**
- **client terminates**
- **half open connection!** (no client!)

- **retransmit req_conn(x)**
- **ESTAB**
- **server forgets x**

- **req_conn(x)**
- **acc_conn(x)**
- **ESTAB**
- **client terminates**

- **req_conn(x)**
- **ESTAB**
- **server forgets x**

- **retransmit req_conn(x)**
- **acc_conn(x)**
- **ESTAB**
- **client terminates**

- **req_conn(x)**
- **acc_conn(x)**
- **ESTAB**
- **server forgets x**

- **data(x+1)**
- **accept data(x+1)**

- **data(x+1)**
- **accept data(x+1)**
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**
  - choose init seq num, y
  - send TCP SYNACK msg, acking SYN

- **SYN RCVD**
  - SYNbit=1, Seq=y
  - ACKbit=1; ACKnum=x+1

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

1. **Closed**
   - Socket connectionSocket = welcomeSocket.accept();
   - SYN(x)

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

3. **SYN Rcvd**
   - ACK(ACKnum=y+1)

4. **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
  - can no longer send but can receive data
  - clientSocket.close()
- FIN_WAIT_2
  - wait for server close
- TIMED_WAIT
  - timed wait for 2*max segment lifetime
- CLOSED

**server state**

- ESTAB
- CLOSE_WAIT
  - can still send data
- LAST_ACK
  - can no longer send data
- CLOSED

**FIN_WAIT_2**

- FINbit=1, seq=y
  - ACKbit=1; ACKnum=y+1
  - wait for server close

**FIN_WAIT_1**

- FINbit=1, seq=x
  - ACKbit=1; ACKnum=x+1
  - can no longer send data

**TIMED_WAIT**

- FINbit=1, seq=y
  - ACKbit=1; ACKnum=y+1
  - can no longer send data
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
  • segment structure
  • reliable data transfer
  • flow control
  • connection management
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

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

- one router, *finite* buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: \( \lambda_{\text{in}} = \lambda_{\text{out}} \)
  - transport-layer input includes *retransmissions*: \( \lambda'_{\text{in}} \geq \lambda_{\text{in}} \)

![Diagram showing Host A and Host B with finite shared output link buffers, and the flow of data with original and retransmitted data.](image)
idealization: perfect knowledge

- sender sends only when router buffers available

Causes/costs of congestion: scenario 2

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

\[ \lambda_{out} \]

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

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

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

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

when sending at R/2, some packets are retransmissions but asymptotic goodput is still R/2 (why?)

**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

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

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!
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

![Graph showing AIMD behavior]

- Additively increase window size...
- ... until loss occurs (then cut window in half)
TCP Congestion Control: details

- **sender limits transmission:**
  - \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd} \)

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

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

\[
\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \quad \text{bytes/sec}
\]
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

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/
Summary: TCP Congestion Control

- **Slow Start**
  - $cwnd = 1 \text{ MSS}$
  - $ssthresh = 64 \text{ KB}$
  - $\text{dupACKcount} = 0$
  - Retransmit missing segment

- **Fast Recovery**
  - $\text{dupACKcount} = 3$
  - $ssthresh = cwnd/2$
  - $cwnd = ssthresh + 3$
  - Retransmit missing segment

- **Congestion Avoidance**
  - $cwnd = cwnd + \text{MSS} \cdot (\text{MSS}/cwnd)$
  - Transmit new segment(s), as allowed

- **New ACK**
  - $cwnd = cwnd + \text{MSS}$
  - $\text{dupACKcount} = 0$
  - Transmit new segment(s), as allowed

- **Timeout**
  - $ssthresh = cwnd/2$
  - $cwnd = 1 \text{ MSS}$
  - $\text{dupACKcount} = 0$
  - Retransmit missing segment

- **Duplicate ACK**
  - $\text{dupACKcount}++$

- **New ACK!**
  - New ACK
  - $cwnd = ssthresh$
  - $\text{dupACKcount} = 0$
  - Transmit new segment(s), as allowed
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
Fairness (more)

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 $R/2$
Explicit Congestion Notification (ECN)

network-assisted congestion control:

- two bits in IP header (ToS field) marked by network router to indicate congestion
- congestion indication carried to receiving host
- receiver (seeing congestion indication in IP datagram) sets ECE bit on receiver-to-sender ACK segment to notify sender of congestion

TCP ACK segment

ECN=00

ECN=11

IP datagram

Transport Layer 3-107
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”
- two network layer chapters:
  - data plane
  - control plane