CDA 4506
Design and Implementation of Data Communication Networks

Lecture Set 4
Dr. R. Lent
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
1. reliable, in-order delivery (TCP)
   1. congestion control
   2. flow control
   3. connection setup
2. unreliable, unordered delivery: UDP
   1. 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)

= socket  = process

```
+---+---+---+---+
| P1 | P2 | P3 | P4 |
| application  | P1 | application  | application  |
| transport    |    | transport    |    |
| network      |    | network      |    |
| link         |    | link         |    |
| physical     |    | physical     |    |
```

host 1  host 2  host 3
How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket
Connectionless demultiplexing

- Create sockets with port numbers:
  - `DatagramSocket mySocket1 = new DatagramSocket(99111);`
  - `DatagramSocket mySocket2 = new DatagramSocket(99222);`

- UDP socket identified by two-tuple:
  - `(dest IP address, dest port number)`

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number

- IP datagrams with different source IP addresses and/or source port numbers directed to same socket
Connectionless demux (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

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)
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!

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

UDP segment format

Length, in bytes of UDP segment, including header

Application data (message)
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 may be errors nonetheless? More later...*
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)
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

state 1 → state 2

event causing state transition

actions taken on state transition

event

actions
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)
exttract (packet, data)
deliver_data(data)
```

sender

receiver
Rdt2.0: channel with bit errors

- Underlying channel may flip bits in packet
  - recall: UDP checksum detects bit errors

- The question: how to recover from errors:
  - acknowledgements (ACKs): receiver explicitly tells sender that pkt was 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

sender

\[
\begin{align*}
\text{rdt\_send(data)} & \\
\text{snkpkt} = \text{make\_pkt(data, checksum)} & \\
\text{udt\_send(sndpkt)} & \\
\text{Wait for call from above} & \\
\text{Wait for ACK or NAK} & \\
\text{rdt\_rcv(rcvpkt) && isNAK(rcvpkt)} & \\
\text{udt\_send(sndpkt)} & \\
\text{rdt\_rcv(rcvpkt) && isACK(rcvpkt)} & \\
\wedge & \\
\end{align*}
\]

receiver

\[
\begin{align*}
\text{rdt\_rcv(rcvpkt) && corrupt(rcvpkt)} & \\
\text{udt\_send(NAK)} & \\
\text{Wait for call from below} & \\
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)} & \\
\text{extract(rcvpkt, data)} & \\
\text{deliver\_data(data)} & \\
\text{udt\_send(ACK)} & \\
\end{align*}
\]
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

What to do?
- sender ACKs/NAKs receiver’s ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of a correctly received pkt!

Handling duplicates:
- sender adds sequence number to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- 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)

rtl_send(data)

rtl_rcv(rcvpkt) && ( corrupt(rcvpkt) || isNAK(rcvpkt) )
udt_send(sndpkt)

rtl_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Lambda

rtl_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Lambda

rtl_rcv(rcvpkt) && ( corrupt(rcvpkt) || isNAK(rcvpkt) )
udt_send(sndpkt)
```

```
rtl_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```
rdt2.1: receiver, handles garbled ACK/NAKs

\begin{itemize}
  \item rdt\_rcv(rcvpkt) && (corrupt(rcvpkt)
  \item sndpkt = make\_pkt(NAK, chksum)
  \item udt\_send(sndpkt)
  \item rdt\_rcv(rcvpkt) && not corrupt(rcvpkt) && has\_seq1(rcvpkt)
  \item sndpkt = make\_pkt(ACK, chksum)
  \item udt\_send(sndpkt)
  \item rdt\_rcv(rcvpkt) && not corrupt(rcvpkt) && has\_seq0(rcvpkt)
  \item extract(rcvpkt, data)
  \item deliver\_data(data)
  \item sndpkt = make\_pkt(ACK, chksum)
  \item udt\_send(sndpkt)
  \item rdt\_rcv(rcvpkt) && (corrupt(rcvpkt)
  \item sndpkt = make\_pkt(NAK, chksum)
  \item udt\_send(sndpkt)
  \item rdt\_rcv(rcvpkt) && not corrupt(rcvpkt) && has\_seq1(rcvpkt)
  \item sndpkt = make\_pkt(ACK, chksum)
  \item udt\_send(sndpkt)
\end{itemize}
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 “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 NAKs 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

receiver FSM fragment

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)

Wait for call 0 from above

Wait for ACK 0

udt_send(sndpkt)

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) || isACK(rcvpkt, 1))

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

udt_send(sndpkt)

L

rdt_rcv(rcvpkt) &&
(corrupt(rcvpkt) || has_seq1(rcvpkt))

udt_send(sndpkt)

Wait for 0 from below

extract(rcvpkt, data)
deliver_data(data)

sndpkt = make_pkt(ACK1, checksum)
udt_send(sndpkt)
New assumption: underlying channel can also lose packets (data or ACKs)
- checksum, seq. #, ACKs, retransmissions will be of help, but not enough

Q: how to deal with loss?
- sender waits until certain data or ACK lost, then retransmits
- yuck: drawbacks?

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

Wait for call 0 from above

rdt_rcv(rcvpkt)
Λ

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,1)
stop_timer

Wait for ACK1

timeout
udt_send(sndpkt)
start_timer

rdt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,0) )
Λ

Wait for ACK0

start_timer

timeout
udt_send(sndpkt)

rdt_send(data)

sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer

Wait for call 1 from above

rdt_rcv(rcvpkt)
Λ

rdt_rcv(rcvpkt)
&& notcorrupt(rcvpkt)
&& isACK(rcvpkt,0)
stop_timer

Wait for ACK1

rdt_send(data)

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

Wait for call 1 from above

rdt_rcv(rcvpkt)
Λ
rdt3.0 in action

(a) operation with no loss

(b) lost packet
rdt3.0 in action

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

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

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

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

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

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

\[ U_{sender} = \frac{3 \times L/R}{RTT + L/R} \]

\[ U_{sender} = \frac{30.008}{30.008} = 0.0008 \]

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 deceive duplicate ACKs (see receiver)
- timer for in-flight pkts
- \textit{timeout(n)}: retransmit pkt n and all higher seq # pkts in window
GBN: sender extended FSM

\[
\begin{align*}
\text{rdt\_send(data)} \\
\text{if (nextseqnum < base+N) \{} \\
\text{sndpkt[nextseqnum] = make\_pkt(nextseqnum, data, chksum)} \\
\text{udt\_send(sndpkt[nextseqnum])} \\
\text{if (base == nextseqnum)} \\
\text{start\_timer} \\
\text{nextseqnum++} \\
\text{\}} \\
\text{else} \\
\text{refuse\_data(data)} \\
\end{align*}
\]

\[
\begin{align*}
\Lambda \\
\text{base=1} \\
\text{nextseqnum=1} \\
\end{align*}
\]

\[
\begin{align*}
\text{rdt\_rcv(rcvpkt)} \\
\text{&& corrupt(rcvpkt)} \\
\end{align*}
\]

\[
\begin{align*}
\text{timeout} \\
\text{start\_timer} \\
\text{udt\_send(sndpkt[base])} \\
\text{udt\_send(sndpkt[base+1])} \\
\text{...} \\
\text{udt\_send(sndpkt[nextseqnum-1])} \\
\end{align*}
\]

\[
\begin{align*}
\text{base = getacknum(rcvpkt)+1} \\
\text{If (base == nextseqnum)} \\
\text{stop\_timer} \\
\text{else} \\
\text{start\_timer} \\
\end{align*}
\]
GBN: receiver extended FSM

ACK-only: always send ACK for correctly-received pkt with highest \textit{in-order} seq #
- may generate duplicate ACKs
- need only remember \texttt{expectedseqnum}

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

sender
- send pkt0
- send pkt1
- send pkt2
- send pkt3 (wait)
- rcv ACK0
- send pkt4
- rcv ACK1
- send pkt5
- pkt2 timeout
- send pkt2
- send pkt3
- send pkt4
- send pkt5

receiver
- rcv pkt0
- send ACK0
- rcv pkt1
- send ACK1
- rcv pkt3, discard
- send ACK1
- rcv pkt4, discard
- send ACK1
- rcv pkt5, discard
- send ACK1
- rcv pkt2, deliver
- send ACK2
- rcv pkt3, deliver
- send ACK3

(loss)
Selective Repeat

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt
- sender window
  - N consecutive seq #’s
  - again limits seq #’s of sent, unACKed pkts
Selective repeat: 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

pkt0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 sent
0 1 2 3 4 5 6 7 8 9

pkt2 sent
0 1 2 3 4 5 6 7 8 9

pkt3 sent, window full
0 1 2 3 4 5 6 7 8 9

pkt0 rcvd, delivered, ACK0 sent
0 1 2 3 4 5 6 7 8 9

pkt1 rcvd, delivered, ACK1 sent
0 1 2 3 4 5 6 7 8 9

 pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

ACK0 rcvd, pkt4 sent
0 1 2 3 4 5 6 7 8 9

ACK1 rcvd, pkt5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 TIMEOUT, pkt2 resent
0 1 2 3 4 5 6 7 8 9

ACK3 rcvd, nothing sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9
Selective repeat: dilemma

Example:
- seq #'s: 0, 1, 2, 3
- window size=3
- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?
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
- **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

- **Source port #**
- **Destination port #**
- **Sequence number**
- **Acknowledgement number**
- **Receive window**
- **Checksum**
- **Urg data pointer**
- **Options (variable length)**
- **Application data** (variable length)

**Fields and Flags:**

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

**Notes:**

- Counting by bytes of data (not segments!)
- # bytes rcvr willing to accept
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

---

**Q:** how receiver handles out-of-order segments

**A:** TCP spec doesn’t say, - up to implementor

---

**Diagram:**
- **Host A**:
  - User types ‘C’
  - Seq=42, ACK=79, data = ‘C’
  - Seq=79, ACK=43, data = ‘C’
  - Seq=43, ACK=80

- **Host B**:
  - host ACKs receipt of ‘C’, echoes back ‘C’

---

**Simple telnet scenario**
TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - but RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT
TCP Round Trip Time and Timeout

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

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

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

RTT (milliseconds)

SampleRTT  Estimated RTT

time (seconds)
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 overview

- 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 (simplified)

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 seq# NextSeqNum
        if (timer currently not running)
            start timer
        pass segment to IP
        NextSeqNum = NextSeqNum + length(data)

    event: timer timeout
        retransmit not-yet-acknowledged segment with
        smallest sequence number
        start timer

    event: ACK received, with ACK field value of y
        if (y > SendBase) {
            SendBase = y
            if (there are currently not-yet-acknowledged
                segments)
                start timer
        }
}

} /* end of loop forever */
TCP: retransmission scenarios

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

Host B
Seq = 92, 8 bytes data

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

Host B
Seq = 92, 8 bytes data

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

Host B
Seq = 92, 8 bytes data

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

Host B
Seq = 92, 8 bytes data

SendBase = 100

SendBase = 120

SendBase = 120

SendBase = 100

lost ACK scenario

premature timeout
TCP retransmission scenarios (more)

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

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

Loss
SendBase = 120
ACK = 120

Cumulative ACK scenario
Modifications

1. Doubling the timeout interval
   - After a timeout event:
     - TCP retransmits the not yet ack’d segment with the smallest sequence number
     - TCP sets the next timeout interval to twice the previous value (rather than deriving it from $\text{EstimatedRTT}$ and $\text{DevRTT}$)
   - Provides a limited form of congestion control

2. Fast retransmit
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
Packet Loss

Packet loss detected by
- Retransmission timeouts
- Duplicate ACKs (at least 3)

Packets

Acknowledgements
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
    }
  }

- A duplicate ACK for already ACKed segment
- Fast retransmit
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
Flow Control

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

**flow control**

sender won’t overflow receiver’s buffer by transmitting too much, too fast

- How can it be implemented?
Window Flow Control

At most $W$ packets per RTT
packet size $\leq$ MSS bytes
Window Flow control

- Limit the number of packets in the network to window $W$

- Source rate = $\frac{W \times \text{MSS}}{\text{RTT}}$ bps

- Notes:
  - If $W$ too small then rate « capacity (low utilization)
  - If $W$ too big then rate > capacity => congestion
  - Solution: Adapt $W$ to network (and conditions)

\[ W \times \text{MSS} = \text{BW} \times \text{RTT} \]
TCP Flow Control

- The receive side of TCP connection has a receive buffer:

  - flow control: sender won’t overflow receiver’s buffer by transmitting too much, too fast

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

- App process may be slow at reading from buffer
TCP Flow control: how it works

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

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

- During this phase, TCP initializes variables:
  - Sequence numbers
  - Buffers: flow control information (**RcvWindow**)

- TCP also provides mechanisms to close connections
TCP 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 sequence number
- No data

**Step 3:** client receives SYNACK, replies with ACK segment, which may contain data

```java
server: Socket connectionSocket = welcomeSocket.accept();
client: Socket clientSocket = new Socket("hostname","port number");
```

Connection request
- SYN, seq=client_isn
- SYN+ACK, seq=server_isn, ack=client_isn+1
- ACK, Ack=server_isn+1

Connection granted
Connection established
Connection established
Closing a TCP Connection

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

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

**Step 3:** client receives FIN, replies with ACK.
- Enters “timed wait” - will respond with ACK to received FINs.

**Step 4:** server, receives ACK. Connection closed.

client closes socket:
```
clientSocket.close();
```
Typical TCP Server Lifecycle

- **CLOSED**: server application creates a listen socket
- **LISTEN**: receive SYN send SYN & ACK
- **SYN_RCVD**: receive ACK send nothing
- **ESTABLISHED**: receive FIN send ACK
- **CLOSE_WAIT**: send FIN
- **LAST_ACK**: receive ACK send nothing
Typical TCP Client Lifecycle

- **CLOSED**: Client application initiates a TCP connection. Send SYN.
- **SYN_SENT**: Receive SYN & ACK, send ACK.
- **ESTABLISHED**: Client application initiates close connection.
- **FIN_WAIT_1**: Send FIN.
- **FIN_WAIT_2**: Receive ACK, send nothing.
- **TIME_WAIT**: Wait 30 seconds.
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
Network Congestion

Informally: “too many sources sending too much data too fast for network to handle”

Different from flow control! (why?)

Effects of congestion:
- Packet loss (buffer overflow at routers)
- Retransmissions
- Reduced throughput
- Long delays (queueing in router buffers)

Network collapse:
- Unnecessarily retransmitted packets
- Undelivered or unusable packets
Causes of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission

- large delays when congested
- maximum achievable throughput
Causes of congestion: scenario 2

- one router, *finite* buffers (packet loss!)
- sender retransmission of lost packets
Causes of congestion: scenario 2 (cont’d)

- \( \lambda_{\text{in}} = \lambda_{\text{out}} \)
- Retransmission when packet loss: \( \lambda'_{\text{in}} > \lambda_{\text{out}} \)
- Retransmission of delayed (not lost) packet makes \( \lambda'_{\text{in}} \) even larger

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

\[
\text{Retransmission when packet loss: } \lambda'_{\text{in}} > \lambda_{\text{out}}
\]

\[
\text{Retransmission of delayed (not lost) packet makes } \lambda'_{\text{in}} \text{ even larger}
\]
Causes of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmission

Q: what happens as $\lambda_{in}$ and $\lambda'_{in}$ increase?
Causes/cost of congestion: scenario 3

when a packet is dropped, any “upstream transmission capacity used for that packet is wasted!
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:
  - To sender (Choke packet), or
  - To receiver and from there to sender. E.g. single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]
- Roughly,

\[
\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}
\]

- \text{CongWin} is dynamic, function of perceived network congestion

How does sender perceive congestion?
- loss event = timeout or 3 duplicate acks

Two mechanisms:
- Slow start
- Congestion Avoidance:
  - AIMD
  - Reaction to timeout events
TCP Tahoe (Jacobson 1988)

SS: Slow Start
CA: Congestion Avoidance
TCP Slow Start

- When connection begins, \( \text{CongWin} = 1 \text{ MSS} \)
  - Example: MSS = 500 bytes & RTT = 200 msec
  - initial rate = 20 kbps
  - available bandwidth may be MSS/RTT
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast until first loss event
TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double $\text{CongWin}$ every RTT
  - done by incrementing $\text{CongWin}$ for every ACK received
- **Summary:** initial rate is slow but ramps up exponentially fast
TCP AIMD

**additive increase:** increase CongWin by 1 MSS every RTT in the absence of loss events: *probing*

**multiplicative decrease:** cut CongWin in half after loss event

Long-lived TCP connection
Congestion Avoidance

cwnd ← cwnd + 1 (for each cwnd ACKs)
Reaction to Timeout Events

- After 3 dup ACKs:
  - **CongWin** is cut in half
  - window then grows linearly

- **But** after timeout event:
  - **CongWin** instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

---

**Philosophy:**

- 3 dup ACKs indicates network capable of delivering some segments
- timeout is “more alarming”
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
Summary: TCP Congestion Control

- When $\text{CongWin}$ is below $\text{Threshold}$, sender in slow-start phase, window grows exponentially.

- When $\text{CongWin}$ is above $\text{Threshold}$, sender is in congestion-avoidance phase, window grows linearly.

- When a triple duplicate ACK occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ set to $\text{Threshold}$.

- When timeout occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ is set to 1 MSS.
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 showing equal bandwidth share and congestion avoidance with additive increase and multiplicative decrease.]
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

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 10 connections;
  - new app asks for 1 TCP, gets rate R/11
  - new app asks for 10 TCPs, gets R/2!