Chapter 3
Transport Layer

A note on the use of these ppt slides:
We’re making these slides freely available to all (faculty, students, readers). They’re in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a lot of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we’d like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Computer Networking: A Top Down Approach
7th edition.
Jim Kurose, Keith Ross
Pearson/Addison-Wesley, April 2016.

All material copyright 1996-2010
J.F Kurose and K.W. Ross, All Rights Reserved
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*, for end-to-end transport
  - 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 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

Potential Confusion: Layer 3 PDUs = Datagram (hence IP Datagram)
Layer 4 UDP PDUs: also traditionally called Datagram
⇒ to de-confuse, consistently use TCP/UDP _Segments_
**Multiplexing/demultiplexing**

Sockets: APIs between the Application Layer Process & Network Layer end-point

Transport Layer: delivers network layer data to/form sockets

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

<table>
<thead>
<tr>
<th>application</th>
<th>P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td></td>
</tr>
<tr>
<td>network</td>
<td></td>
</tr>
<tr>
<td>link</td>
<td></td>
</tr>
<tr>
<td>physical</td>
<td></td>
</tr>
</tbody>
</table>

host 1

| P1 |
|    |
| application |
| transport |

| P2 |
|    |
| application |
| transport |
| network |
| link |
| physical |

host 2

| P4 |
|    |
| application |
| transport |

|        |
| network |
| link |
| physical |

host 3

Transport Layer
How demultiplexing works

- host receives transport layer segment
  - each segment has source, destination port number
  - each IP datagram has source, destination IP address in **network header**

- host uses **IP addresses & port numbers** to direct segment to appropriate socket
**Connectionless demultiplexing**

- **recall**: create sockets with host-local port numbers:
  
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  DatagramSocket mySocket2 = new DatagramSocket(12535);
  ```

- **recall**: when creating segment to send into UDP socket, must specify 2-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

  e.g. multiple responses to same DNS query
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: example

three segments, all destined to IP address: B, dest port: 80 are demultiplexed to different sockets
Chapter 3 outline

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer
3.5 Connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management
3.6 Principles of congestion control
3.7 TCP congestion control
UDP: User Datagram Protocol [RFC 768]

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out of order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?
- no connection establishment (which can add delay)
- simple: no connection state at sender, receiver
- small segment header
- no congestion control: UDP can blast away as fast as desired
**UDP: more**

- **often used for streaming multimedia apps**
  - loss tolerant
  - rate sensitive
- **other UDP uses**
  - DNS
  - SNMP
- **reliable transfer over UDP:** *add reliability at application layer*
  - application-specific error recovery!

---

**UDP segment format**

- **source port #**
- **dest port #**
- **length**
- **checksum**

Length, in bytes of UDP segment, including header

**Application data**

*(message)*

**UDP segment format**
**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 (entire) received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. *But maybe errors nonetheless? More later...*
Internet Checksum Example

- Note: when adding numbers, a carryout from the most significant bit needs to be added to the result
- Example: add two 16-bit integers

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

wraparound: 1

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

(a) provided service

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of Reliable data transfer

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

(a) provided service

(b) service implementation

- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Principles of Reliable data transfer

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

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

**rdt_rcv()**: called when packet arrives on rcv-side of channel

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

**send side**

**receive side**
Impacts of Unreliable Channel

- **Bi-directional Channel**: Both hosts have data to send to each other!

* Suffices to understand protocol operation by considering uni-directional data (e.g. A → B)

- **Channels cause** -
  - Errors in data (due to noise in channel)
  - Loss in data (data is irretrievably delayed)
  - Data is re-sequenced (i.e. arrives at receiver out-of-order)
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**
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

sender

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
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

Automatic Repeat Request (ARQ)
**rdt2.0: FSM specification**

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

**Wait for call from above**

**Wait for ACK or NAK**
- `rdt_rcv(rcvpkt) && isACK(rcvpkt)`

**Stop and wait**
- `Sender sends one packet, then waits for receiver response`

**Receiver**
- `rdt_rcv(rcvpkt) && corrupt(rcvpkt)`
  - `udt_send(NAK)`

**Wait for call from below**

- `udt_send(NAK)`
- `Wait for call from above`
- `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
  - `udt_send(sndpkt)`

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

- `rdr_send(sndpkt)`

- `Wait for ACK or NAK`
  - `rdt_rcv(rcvpkt) && isACK(rcvpkt)`

- `\Lambda`
**rdt2.0: operation with no errors**

- `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)**

- `udt_send(ACK)`

**Wait for ACK or NAK**

**Wait for call from below**

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

- `udt_send(NAK)`

- `Wait for ACK or NAK`
**rdt2.0: error scenario**

```
rdt_send(data)

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

Wait for call from above

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

Wait for ACK or NAK

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

Wait for call from below

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

Wait for call from above

```
Lambda
```

```
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 garbled
- sender adds *sequence number* to each pkt
- receiver discards (doesn't deliver up) duplicate pkt

**STOP & WAIT**

new packet NOT sent until previous packet successfully sent/acked
- Sender inserts Seq # (0,1) in pkt
- “Wait for ACK or NAK 0” → ACK/NAK for the imm. (previous) transmission with seq. # 0 (ACK/NAK does not carry a seq. # in rdt 2.1, yet there is no confusion at sender as to which packet is being ACK/NAKed)
**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)

- rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
  - sndpkt = make_pkt(ACK, chksum)
  - udt_send(sndpkt)

- rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
  - extract(rcvpkt, data)
  - deliver_data(data)
  - sndpkt = make_pkt(ACK, chksum)
  - 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 “current” pkt has 0 or 1 seq. #

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

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

- **ACK reflects the seq # of last correct packet**

  - **Sender FSM fragment**
    - \( \text{rdt}_\text{send}(\text{data}) \)
    - \( \text{sndpkt} = \text{make_pkt}(0, \text{data}, \text{checksum}) \)
    - \( \text{udt}_\text{send}(\text{sndpkt}) \)
    - \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \&\& (\text{notcorrupt}(\text{rcvpkt}) \| \text{has}\_\text{seq0}(\text{rcvpkt})) \)
    - \( \text{extract}(\text{rcvpkt}, \text{data}) \)
    - \( \text{deliver}\_\text{data}(\text{data}) \)
    - \( \text{sndpkt} = \text{make_pkt}(\text{ACK},0, \text{chksum}) \)
    - \( \text{udt}_\text{send}(\text{sndpkt}) \)

  - **Receiver FSM fragment**
    - \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \&\& (\text{corrupt}(\text{rcvpkt}) \| \text{has}\_\text{seq1}(\text{rcvpkt})) \)
    - \( \text{sndpkt} = \text{make_pkt}(\text{ACK},1, \text{chksum}) \)
    - \( \text{udt}_\text{send}(\text{sndpkt}) \)

- Wait for call 0 from above
- Wait for 0 from below
- Wait for 1 from below
- Wait for ACK 0

- \( \Lambda \)
**rdt3.0: channels with errors and loss**

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

**Approach:** sender waits “reasonable” amount of time for ACK
- retransmits if no ACK received in this time
- if pkt (or ACK) just delayed (not lost):
  - retransmission will be duplicate, but use of seq. #’s already handles this
  - receiver must specify seq # of pkt being ACKed

**requires countdown timer**
**rdt3.0 sender**

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

**ALTERNATING BIT PROTOCOL**
**rdt3.0:receiver** (same as rdt2.2)!

- **ACK reflects the seq # of last correct packet**

```
rdt_rcv(rcvpkt) &&
  (notcorrupt(rcvpkt) ||
   has_seq0(rcvpkt))
__________________________
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, 0, chksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) &&
  (notcorrupt(rcvpkt) ||
   has_seq1(rcvpkt))
__________________________
extract(rcvpkt, data)
deliver_data(data)
sndpkt = make_pkt(ACK, 1, chksum)
udt_send(sndpkt)
```
rdt3.0 in action

(a) operation with no loss

(b) lost packet
rdt3.0 in action

(c) lost ACK

(d) premature timeout

ALTERNATING BIT PROTOCOL
Performance of rdt3.0 (ideal channel)

- rdt3.0 works, but performance stinks (even in best case)
- ex: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:

\[
d_{trans} = \frac{L}{R} = \frac{8000\text{bits}}{10^9\text{bps}} = 8\text{ microseconds}
\]

- \(U_{SW}\) sender: utilization = fraction of time sender busy = fraction of time channel is utilized

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

\(\rightarrow\) 270 kB/sec throughput over 1 Gbps link!
rdt3.0: stop-and-wait operation

\[ U_{SW} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027 \]
Let $1 - P_f = \text{probability frame arrives w/o errors}$

- **Avg.** # of transmissions to first correct arrival is then $\frac{1}{1-P_f}$

- “If 1-in-10 get through without error, then **avg.** 10 tries to success”

- Avg. Total Time per frame is then $(RTT+L/R)/(1 - P_f)$

- $U_{SW} = \frac{L/R}{RTT + L/R} (1 - P_f)$

**Effect of frame loss**

*Review:* Geom. Prob. distribution!
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

\[
\begin{align*}
\text{(a) a stop-and-wait protocol in operation} & \quad \text{(b) a pipelined protocol in operation}
\end{align*}
\]

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

\[
U_{\text{sender}} = \frac{3 \times L/R}{\text{RTT} + L/R} = \frac{.024}{30.008} = 0.0008
\]

Increase utilization by a factor of 3!
Pipelined Protocols

Go-back-N: big picture
- sender can have up to N unacked packets in pipeline
- rcvr only sends cumulative acks
  - doesn’t ack packet if there’s a gap
- sender has timer for oldest unacked packet
  - if timer expires, retransmit all unack’ed packets

Selective Repeat: big pic
- sender can have up to N unack’ed packets in pipeline
- rcvr sends an ack with seq# for each rcvd. packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only unack’ed packet
Go-Back-N (sliding window protocol)

Sender:
- k-bit seq # in pkt header → seq. # space is \{0, 1, ..., 2^k -1\}
- "window" of up to N, consecutive unack’ed pkts allowed (N ≤ 2^k is OK; typically N = 2^k)

$\begin{align*}
N &= 2^k \\
\text{window size} &= N \\
\text{send_base} &\rightarrow \text{nextseqnum}
\end{align*}$

- ACK(n): ACKs all pkts up to & including seq # n - "cumulative ACK"
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window
Go Back N

- k-bit seq # in pkt header → seq. #’s {0, 1, ..., 2^k - 1}
- Sender’s window of (up to) N consecutive Unacked pkts allowed
  \[ N \leq 2^k - 1 \]
- Receiver’s Window Size: 1 (for expected in-order pkt, all others discar...

\[ \text{send\_base} \quad \text{nextseqnum} \]

\[ \text{already} \quad \text{ack’ed} \quad \text{usable, not} \quad \text{sent} \quad \text{not usable} \]

\[ \text{window size} \quad N \]

- ACK(n): ACKs all pkts up to & including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window

Transport Layer 3-50
GBN: sender extended FSM

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

Wait

\[
\begin{align*}
\text{rdt\_rcv(rcvpkt) \&\& notcorrupt(rcvpkt)} & \\
\text{base = getacknum(rcvpkt)+1} & \\
\text{If (base == nextseqnum) } & \\
\text{stop\_timer} & \\
\text{else} & \\
\text{start\_timer} & \\
\end{align*}
\]

\text{timeout}

\text{start\_timer}

\text{udt\_send(sndpkt[base])}

\text{udt\_send(sndpkt[base+1])}

\text{...}

\text{udt\_send(sndpkt[nextseqnum-1])}

Suffices for sender only keep ONE Timer: OLDEST un'acked pkt
**GBN: receiver extended FSM**

**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 (N=4)**

Upon timeout:
- resend ALL outstanding frames 2,3,4,5 in send buffer

---

**sender**
- send pkt0
- send pkt1
- send pkt2 (wait)
- send pkt3
- 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
- equiv. Send ACK5
Frame with errors and subsequent out-of-sequence frames are ignored

Error recovery via timer time-out & go-back-4 (normal recovery mechanism for packet error)
Efficiency of Go-Back-N

- GBN is completely efficient, if $N$ is large enough to keep channel busy, and if channel is error-free.

- Assume $P_f$ = frame loss probability, then time to deliver a frame is:
  - $t_f = \frac{L}{R}$ if first transmission succeeds
    - prob. of event $(1 - P_f)$
  - $t_f + N \left( \frac{t_f}{1-P_f} \right)$ if the first transmission does not succeed
    - prob. of event $P_f$

**av.** time to send a frame successfully -

$$t_{GBN} = t_f (1 - P_f) + P_f \left( t_f + \frac{N t_f}{1-P_f} \right) = t_f + P_f \frac{N t_f}{1-P_f}$$

$$U_{GBN} = \frac{t_f}{t_{GBN}} = \frac{1-P_f}{1+(N-1)P_f}$$
Selective Repeat

- Receiver *individually* acks all correctly recd. packets
  - Buffers correctly rec’d but out-of-order packets for eventual in-order delivery to upper layer
- Sender re-sends only packets for which timer expires (i.e. ack not rec’d)
  - Each sent & unacked packet has a timer running
- Sender window size $N_s$
  - seq. # of sent & unacked consecutive packets
- Receiver window size $N_r$
  - seq. # of consecutive packets that receiver will accept

Typically: $N_r = N_s = N = 2^{k-1}$

Sequence # field: k-bit $\rightarrow \{0, 1, \ldots, 2^k-1\}$, i.e. the window size is (at most) $\frac{1}{2} \times 2^k$
Selective Repeat ARQ

Sender & Receiver Buffer (N)

Transmitter

Send Window

Frames transmitted and ACKed

$S_{\text{last}}$ $S_{\text{recent}}$ $S_{\text{last}} + N - 1$

Receiver

Receive Window

Frames received

$R_{\text{next}}$ $R_{\text{next}} + N - 1$

Buffers

$S_{\text{last}}$

Timer

$S_{\text{last}} + 1$

... 

$S_{\text{recent}}$

... 

$S_{\text{last}} + N - 1$

Buffers

$R_{\text{next}}$

... 

$R_{\text{next}} + N - 1$

max Seq # accepted
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 \([sendbase, sendbase+N-1]\):
  - 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
  - Send ACK\( (n) \)
- otherwise:
  - ignore
Example: SR with Out-of-Order Frames Storec @ Rx.

Receiver Buffers (correctly rec'd) Out-of-Order frames !!

For comparison: CUM acking →

DupACK @ sender: Knows that Frame 2 is missing, and that packets with seq# > 2 are being rx. correctly
Selective repeat: Sliding Window (N=4)
Efficiency of Selective Repeat

- Assume $P_f =$frame loss probability, then number of transmissions required to deliver a frame is:
  - $t_f / (1-P_f)$ where $t_f = L/R$

\[
U_{SR} = \frac{t_f}{t_f/(1-P_f)} = (1 - P_f)
\]
Example: Efficiency with Channel Errors

$L = 1250$ bytes = 10000 bits, $R = 1$ Mbps, RTT = 1 s.

Compare S&W, GBN & SR efficiency for channel frame error rate with $P_f = 0, 10^{-3}, 10^{-1}$

<table>
<thead>
<tr>
<th>Efficiency</th>
<th>$P_f = 0$</th>
<th>$10^{-3}$</th>
<th>$10^{-1}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S&amp;W</td>
<td>1%</td>
<td>1%</td>
<td>0.9%</td>
</tr>
<tr>
<td>GBN (N=100 )</td>
<td>100%</td>
<td>9.1%</td>
<td>8.2%</td>
</tr>
<tr>
<td>SR</td>
<td>100%</td>
<td>99.9%</td>
<td>90%</td>
</tr>
</tbody>
</table>

- Selective Repeat outperforms GBN and S&W, but efficiency drops as error rate increases
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
   - Connection management
   - reliable data transfer
   - flow control

3.6 Principles of congestion control
3.7 TCP congestion control
TCP: Overview

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

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

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

- **send & receive buffers**

**RFCs:** 793, 1122, 1323, 2018, 2581

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

- **connection-oriented:**
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange

- **flow control**
  - sender will not overwhelm receiver

- **congestion control**
  - sender rate adapted to mitigate congestion at intermediate nodes.
TCP - 3 Phases

- **Connection Establishment Phase**
  - Sets up a connection between the sending and receiving application processes by creating and initializing variables that are used in the protocol.

- **Data Transfer Phase**
  - Data is delivered to the application processes in each direction, correctly and in sequence. TCP does not assume that the underlying IP network service is reliable. To implement reliability, TCP uses a form of Selective Repeat ARQ.

- **Connection Termination Phase**
  - Each direction of the connection is terminated independently, allowing data to continue flowing in one direction after the other direction has stopped sending data.
TCP Connection Establishment

“Three-way Handshake” (Why doesn't 2-way handshake suffice?)

Host A

- SYN, Seq_no = x
- SYN, Seq_no = y, ACK, Ack_no = x+1
- Seq_no = x+1, ACK, Ack_no = y+1

Host B

Initial Sequence No: randomly chosen anew for each connection!

e.g. use local clock to select ISN

Note: Ack x+1 in response to Seq_no x!
TCP Connection Establishment

- **Step 1**: Host A sends a connection request to host B by setting the SYN bit. Host A also registers its initial sequence no (Seq. No = x).

- **Step 2**: Host B acknowledges the request by setting the ACK bit and indicating the next data byte to receive (ACK No. = x+1). The “plus one” is needed because the SYN bit consumes one sequence #. At the same time, host B also sends a request by setting the SYN bit and registering its initial sequence no to use (Seq. No = y).

- **Step 3**: Host A acknowledges the request from B by setting the ACK bit and confirming the next data byte to receive (ACK No. = y+1). The Sequence No is set to x+1. When B receives this, the connection is established.
Data Transfer: Reliable Byte-Stream Service

- Stream Data Transfer
  - transfers a contiguous stream of bytes across the network, with no indication of boundaries
  - groups bytes into segments
  - transmits segments (layer 4 PDU)

- Reliability
  - error control mechanism to deal with IP transfer impairments
TCP segment structure

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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>destination port #</td>
</tr>
<tr>
<td>sequence number</td>
<td>acknowledgement number</td>
</tr>
<tr>
<td>available window</td>
<td>receive window</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urg data pointer</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>options (variable length)</td>
</tr>
<tr>
<td>max. segment size (MSS)</td>
<td>max. segment size (MSS)</td>
</tr>
</tbody>
</table>

**Max. Segment Size (MSS) [payload]**
- Upto 64 kBytes; default 576 (IP PDU)
- -20 (IP header) -20 (TCP header) = 536 bytes
TCP Header Fields

Sequence Number (32 bits)

- Byte-wise count
  - \textit{Identifies 1}\textsuperscript{st} byte in segment

Example: If the Sequence No is 100 and the data contains 5 bytes, the next time this TCP module sends a segment, the Sequence Number will be 105.

- \( 0 \leq SN \leq 2^{32}-1 \)
- \textit{Initial sequence number} selected randomly during connection setup

Acknowledgement Number (32 bits)

- SN of \textit{next byte expected} by receiver
- \textit{Cum ACK}:
  - Acknowledges that all prior bytes in stream have been received correctly
  - Valid if ACK flag set

Header length

- Minimum header length - 20 bytes
- Maximum header length - 60 bytes
TCP: A “state-full” protocol

- Each TCP connection preserves some *state* variables -
  - **Current (Sender’s) Congestion Window (CWnd), SendBase** [refer to TCP Congestion Control Algorithm]
  - **Current Seq, Ack #**

- TCP state is lost when the connection is lost (host crashes/restarts)

- TCP Connection Establishment (3-way handshake): ensures a *consistent initial* state between TCP sender and receiver → the ISN and ACK # between the sender and receiver must be synched!
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

![Diagram of TCP segment fields]

<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</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>rwnd</td>
<td>Window size</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>urg pointer</td>
<td>Urgent pointer</td>
</tr>
<tr>
<td>sent ACKed</td>
<td>Sent and acknowledged</td>
</tr>
<tr>
<td>not-yet ACKed</td>
<td>Not-yet acknowledged (&quot;in-flight&quot;)</td>
</tr>
<tr>
<td>usable</td>
<td>Usable</td>
</tr>
<tr>
<td>not usable</td>
<td>Not usable</td>
</tr>
</tbody>
</table>

Transport Layer 3-73
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

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

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

Transport Layer 3-74
TCP Connection Termination

- Each end of the data flow must be shut down independently ("half-close")
- If one end is done it sends a FIN segment. This means that no more data will be sent
- **Four steps involved:**
  1. X sends a FIN to Y (**active close**)
  2. Y ACKs the FIN,
     (at this time: Y can still send data to X)
  3. and Y sends a FIN to X (**passive close**)
  4. X ACKs the FIN.
**TCP Connection Management (cont.)**

**Closing a connection:**

client closes socket:
```
clientSocket.close();
```

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

**Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.
**TCP Connection Management (cont.)**

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

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

**Note:** with small modification, can handle simultaneous FINs.

Timed wait: duration to prevent delayed packets from one TCP connection being accepted by a later connection (identical TCP sockets); value is implementation dep. (30 sec, 1 min, 2 min)
# TCP States

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSED</td>
<td>No connection is active or pending</td>
</tr>
<tr>
<td>LISTEN</td>
<td>The server is waiting for an incoming call</td>
</tr>
<tr>
<td>SYN RCVD</td>
<td>A connection request has arrived; wait for Ack</td>
</tr>
<tr>
<td>SYN SENT</td>
<td>The client has started to open a connection</td>
</tr>
<tr>
<td>ESTABLISHED</td>
<td>Normal data transfer state</td>
</tr>
<tr>
<td>FIN WAIT 1</td>
<td>Client has said it is finished</td>
</tr>
<tr>
<td>FIN WAIT 2</td>
<td>Server has agreed to release</td>
</tr>
<tr>
<td>TIMED WAIT</td>
<td>Wait for pending packets (&quot;2MSL wait state&quot;)</td>
</tr>
<tr>
<td>CLOSING</td>
<td>Both Sides have tried to close simultaneously</td>
</tr>
<tr>
<td>CLOSE WAIT</td>
<td>Server has initiated a release</td>
</tr>
<tr>
<td>LAST ACK</td>
<td>Wait for pending packets</td>
</tr>
</tbody>
</table>

**MSL (Maximum Segment Lifetime):** max. lifetime of TCP segment
(= 2 min. in RFC793); estimate not a hard limit
TCP Connection Management (cont)

**TCP client lifecycle**
- **CLOSED**: wait 30 seconds
- **TIME_WAIT**: receive FIN; send ACK
- **FIN_WAIT_2**: receive ACK; send nothing
- **FIN_WAIT_1**: receive ACK; send nothing
- **LAST_ACK**: send FIN
- **CLOSE_WAIT**: receive FIN; send ACK
- **ESTABLISHED**: client application initiates a TCP connection; send SYN
- **SYN_SENT**: receive SYN & ACK; send ACK
- **ESTABLISHED**: client application initiates close connection; send FIN

**TCP server lifecycle**
- **LISTEN**: receive SYN; send SYN & ACK
- **SYN_RCVD**: receive ACK; send nothing
- **CLOSE_WAIT**: send FIN
- **ESTABLISHED**: client application creates a listen socket
TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?
- longer than RTT
  - end-2-end RTT varies due to congestion at intermediate nodes!
- 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 (done via time-stamping)
  - TCP uses ONLY 1st time successful transmissions to measure sample RTT (i.e. ignores retransmissions) [why? Hint: ambiguity relating the ACK to initial or retransmitted segment; see Karn's Algorithm]
- 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

\[
\text{EstimatedRTT}(n) = \alpha \left[ \text{SampleRTT}(n) + (1- \alpha) \text{SampleRTT}(n-1) + (1-\alpha)^2 \text{SampleRTT}(n-2) + \ldots \right]
\]

- Influence of past sample decreases exponentially fast
- Typical value: \( \alpha = 0.125 \).

- Use above update only for 1st time successful Segment-Ack pairs (ignore any retransmission - Karn’s Algorithm) to get SampleRTT
Example RTT estimation:

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

- RTT (milliseconds)
- Time (seconds)

SampleRTT
Estimated RTT
TCP Round Trip Time and Timeout

Setting the timeout timer (RTO)

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- 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 as below (this calculation is done every time a timer is started)

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

Higher margin when there is a lot of variability in SampleRTT

- every time a segment is retransmitted resulting from timeout event, the timeout interval is set to twice the previous value, instead of calculating it using EstimatedRTT and DevRTT.

→ this timeout doubling provides some 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
  - Connection management
  - reliable data transfer
  - flow control
3.6 Principles of congestion control
3.7 TCP congestion control
TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- pipelined segments
- cumulative acks
- TCP uses per-segment retransmission timer

- retransmissions are triggered by:
  - timeout events
  - duplicate acks

- initially consider simplified TCP sender:
  - ignore duplicate acks
  - ignore flow control, congestion control
TCP sender events:

data rcvd from app:
- Create segment with seq #
- seq # is byte-stream number of first data byte in segment
- start timer if not already running (for all unacked segment)
- expiration interval: $\text{TimeOutInterval}$ or $\text{RTO}$

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 Data Transfer

- Sliding Window mechanism
  → window slides on a byte basis instead of a packet basis.

- TCP Sender resends individual segments not delivered to receiver, receiver stores out-of-order packets (hence Selective Repeat) but uses Cumulative ACKs (Go Back N feature)

![Diagram of TCP data transfer](image)
TCP: retransmission scenarios

- **Host A**
  - **SendBase**: 100
  - **Seq**: 92
  - **Data**: 8 bytes

- **Host B**
  - **ACK**: 100

**Loss Scenario**
- **Host A**: **Seq**: 92
- **Host B**: **X**
- **Timeout**

**Lost ACK Scenario**
- **Host A**: **SendBase**: 100, **Seq**: 92
- **Host B**: **ACK**: 100

**Premature Timeout**
- **Host A**: **Seq**: 92, **Data**: 8 bytes
- **Host B**: **ACK**: 100

**TCP rx. detects duplicate due to repeated Seq. #92**

Transport Layer 3-88
TCP retransmission scenarios (more)

Host A

SendBase = 120

Host B
time

timeout

Seq=92, 8 bytes data

ACK=100

Seq=100, 20 bytes data

ACK=120

Cumulative ACK scenario

X loss
## TCP ACK generation [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send <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>Immediately send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
Fast Retransmit

- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

- if sender receives 3 DupACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires
Resending a segment after *triple duplicate ACK* = 4 ACKs with identical Seq #
**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
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
   - Connection management
   - reliable data transfer
   - flow control

3.6 Principles of congestion control
3.7 TCP congestion control
TCP Flow Control

- receive side of TCP connection has a receive buffer
- app process may be slow at reading from buffer
- 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

from sender
receiver protocol stack
TCP Flow control: how it works

(suppose TCP receiver has only in-order segments)

- **spare room in buffer**
  
  \[ \text{RcvWindow} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}] \]

- **rcvr advertises spare room by including value of RcvWindow in TCP segments**
  - default 4096 bytes

- **sender limits unACKed data to current RcvWindow value**
  - guarantees receiver buffer doesn't overflow

\[ \text{RcvWindow} = \text{Rwnd} \]
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
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 (routers) to handle”
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!
- Different from flow control!!
Causes/costs of congestion

- two senders, two receivers
- one router, \textit{infinite} buffer
- no retransmission

- large delays when congested
- maximum achievable throughput

\[ \lambda_{\text{out}} \]
\[ \lambda_{\text{in}} \]
\[ C/2 \]
\[ C/2 \]
Causes/costs of congestion:

- one router, finite buffers
- sender retransmission of timed-out packet
  - application-layer input = application-layer output: \( \lambda_{in} = \lambda_{out} \)
  - transport-layer input includes retransmissions: \( \lambda'_{in} \geq \lambda_{in} \)
Congestion scenario 2a: ideal case

- sender sends only when router buffers available
Congestion scenario 2b: known loss

- packets may get dropped at router due to full buffers
  - sometimes not lost
- sender only resends if packet known to be lost (admittedly idealized)

When sending at R/2, some packets are retransmissions but asymptotic goodput is still R/2 (why?)
Congestion scenario 2c: duplicates

- packets may get 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!
**Congestion scenario 2c: duplicates**

- packets may get 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
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

Q: what happens as $\lambda_{\text{in}}$ and $\lambda'_{\text{in}}$ increase?

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

$\lambda'_{\text{in}}$: original data, plus retransmitted data

finite shared output link buffers

A-C and D-B traffic competes at R2!
Causes/costs of congestion: scenario 3

For very high $\lambda'_\text{in}$, router R1 gets congested, and hence the flow rate from R1 → R2 drops. As a result, B-D traffic fills the R2 buffer increasingly and ultimately the A-C throughput via R2 drops to (near) ze

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: Principles

- **End-to-end**: network congestion is inferred by end systems using observed packet loss/delay at the end user!

- **Strategy**: TCP source sends packets into the network and reacts to observable events (ACK, Loss ..); each source probes the network in a distributed, greedy way to determine how much capacity is available for its own flow.
  - Implicit congestion signal (packet loss)

- Maintains **three** variables:
  - **Cwnd** - congestion window
  - **Receive wind (Rwnd)** - receiver advertised window
  - **ssthresh** - congestion threshold size (for Cwnd adjustment)
  - **ACKs 'pace'** the transmission of packets \(\rightarrow\) TCP is “self-clock
**Congestion Control: Send Side**

CongWin/Cwnd = amount of unacknowledged data (TCP segments) at current time - measured in units of Max Segment Size or MSS

At any given time, a TCP sender CANNOT send data with a sequence number higher than the sum of the highest acknowledged sequence number and Cwnd.
**Congestion Control: Receive Side**

- At host B, \( \text{LastByteRcvd} - \text{LastByteRead} \leq \text{RcvBuffer} \).
- \( \text{Rwnd} = \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}] \)

The amount of unacknowledged data segments @ TCP sender is also limited by current \( \text{Rwnd} \) →

\[
\text{Last Byte sent} - \text{Last Byte Acked} \leq \min \{\text{Cwnd}, \text{Rwnd}\}
\]
TCP Congestion Window (steady state)

- For **long TCP flows**, Cwnd achieves a (dynamic) steady-state as below quickly:
  - **Cwnd adaptation**: increase 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

![Graph showing saw tooth behavior for Cwnd adaptation](image-url)

- **saw tooth behavior**: probing for bandwidth
TCP Congestion Control: details

Assume RcvWindow $\gg$ Cwnd

- sender limits transmission
  $\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}$

- cwnd is dynamic, varies from round-to-round
  Round = 1 RTT duration

- Note: If Cwnd is in TCP segments $\rightarrow$ Cwnd $\times$ MSS to convert to Bytes.

How does sender perceive congestion?

- loss event = timeout or 3 duplicate acks

TCP sender reduces rate (cwnd) after loss event

- slow start (transient)
- Additive Increase, Multiplicativce Decrease (AIMD) (steady state)
- conservative reaction after timeout events
I. TCP Slow Start

- when connection begins (or after time-out), increase rate *exponentially* until first loss event:
  - initially $cwnd = 1 \text{ MSS}$
  - double $cwnd$ every RTT by incrementing $cwnd$ by 1 MSS for every ACK received

- **summary:** initial rate is slow but ramps up exponentially fast
Transition: Slow Start $\rightarrow$ Cong. Avoid (AIMD)

Throttle Back – exponential to *linear increase* upon Cwnd exceeding the (initial) *ssthresh*!

LONG FLOWS REACH AI-MD/CONG. AVOID STEADY-STATE!
TCP in Congestion Avoid (AI-MD) Phase

Additive Increase
Add one segment to CWnd each RTT

Multiplicative Decrease
Halve CongWin
TCP Slow Start

- When connection begins, $Cwnd = 1 \text{ MSS}$
  - Example: $MSS = 500 \text{ bytes} \& RTT = 200 \text{ msec}$
  - initial rate = 20 kbps

- Initial available bandwidth for the connection may be $>> 20 \text{ Kbps}$
  - desirable to quickly ramp up to respectable rate

- When connection begins, increase rate exponentially fast (doubling $Cwnd$ every $RTT$) until it crosses the $ssthresh$, it then enters the AI-MD (Cong. Avoid) phase.

- If it ramps up exponentially fast, why “slow start”?
  - even if the ramp up is fast, the amount of data sent (relative to the available bandwidth) is small!
Inferring loss (TCP Tahoe)

- after 3 dup ACKs (TD’s)
  - Initiate Fast Re-transmit without waiting for TO
    - but is treated the same as timeout event
    - Cwnd set to 1 MSS;
    - new ssthresh is set to 1/2 of cwnd just before loss event
    - TCP sender returns to Slow Start, i.e. window again grows exponentially to a threshold, then grows linearly
Refinement in Loss Handling: Fast Retransmit & Recovery (TCP Reno)

TCP Reno (FRR):
- Treats loss from TD and Time-Outs **differently**!
  - For loss due to time-out, same as TCP Tahoe!
  - but for TD's, does **NOT** reset CWnd=1 and force Slow Start, instead keeps sender in Cong. Avoid!

Philosophy:
- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a "more alarming" congestion scenario
TCP Reno with T-D Acks

Adjust $SS$ threshold = $\frac{1}{2} \times current\ cwnd$

Set new $Cwnd = SSThreshold + 3$

(+3: for the 3 successfully rec’d segments that triggered triple DUP ACKs).

Example: If current value of $Cwnd = 20$ MSS and a loss is detected, new $SS$threshold = 10 and new $Cwnd = 13$ MSS for the next AI-MD cycle.

Note: OK to ignore +3 and use $Cwnd = ssthresh$ after loss via TD ACKs in Reno!

→ Over several AI-MD cycles in steady-state - expect $SS$threshold to equal $\frac{1}{2}$ the available bandwidth for the connection!
Evolution of Congestion Window
TCP-Reno

Transmission Round

Congestion Window (in segments)

timeout

W

W/2

drops to 1

halved

fast re-tx

fast re-tx
Summary: TCP Congestion Control

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

**Congestion Avoidance**
- \( cwnd = \text{cwnd} + \text{MSS} \cdot (\text{MSS}/\text{cwnd}) \)
- \( \text{dupACKcount} = 0 \)
- \( \text{transmit new segment(s), as allowed} \)
- \( \text{new ACK} \)
- \( \text{cwnd} = \text{ssthresh} + 3 \)
- \( \text{retransmit missing segment} \)
- \( \text{dupACKcount} = 3 \)

**Fast Recovery**
- \( cwnd = \text{cwnd} + \text{MSS} \)
- \( \text{transmit new segment(s), as allowed} \)
- \( \text{new ACK} \)
- \( \text{dupACKcount} = 0 \)
- \( \text{new ACK} \)
- \( \text{retransmit missing segment} \)
- \( \text{dupACKcount} = 3 \)
- \( \text{new ACK} \)
- \( \text{retransmit missing segment} \)
TCP throughput (steady state Congestion Avoid)

- what’s the average throughput of TCP as a function of window size and RTT?
  - *ignore slow start*

- let $W$ be the window size when loss occurs.
  - when window is $W$, throughput is $W/RTT$
  - just after loss, window drops to $W/2$, throughput to $W/2$
  - (long-term) average throughput = $0.75 \frac{W}{RTT}$

\[
(0.75 \frac{W\cdot MSS}{RTT})
\]
**TCP Link Utilization (Single Flow)**

- So TCP window size oscillates between $W/2$ and $W$ in steady-state → link utilization is between 50 and 100% (av. 75%)??
  - only if the router @ link is unbuffered!

Minimum window for full utilization → Router cannot utilize the link fully and achieves 75%!
TCP Utilization Single Flow (Buffered Link)

- With suff. large router queue → can get 100% utilization
  - Price – additional delays due to router queue
- How big must the router queue be?
  - Windows vary from $W \to W/2$
    - Must make sure that link is always full, hence
      - $W/2 > RTT \times BW$
      - $W = RTT \times BW + Qsize$
    → $Qsize = RTT \times BW$ (ensures 100% utilization)
- Delay?
  - Varies between RTT and 2 * RTT
TCP Fairness

- BW proportional to 1/RTT, hence TCP is RTT fair
  - Flows sharing a bottleneck link with same RTTs get same bandwidth
  - Otherwise, in inverse proportion to the RTT

**fairness goal:** if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K (if they have same RTT)
TCP: Fair, Efficient Bandwidth Allocation

- Too slow
  - Fail to take advantage of available bandwidth $\rightarrow$ underload
    $$x_1 + x_2 < R$$

- Too fast
  - Overshoot knee $\rightarrow$ overload, high delay, loss
    $$x_1 + x_2 > R$$

- Everyone's doing it
  - May all under/over shoot $\rightarrow$ large oscillations

- Efficiency line (2 user case)
  $$x_1 + x_2 = R$$
Fair Allocation

Max-min fairness
- Flows which share the same bottleneck link get the same amount of bandwidth

\[ F(x) = \frac{\left( \sum x_i \right)^2}{n \left( \sum x_i^2 \right)} \]

Fairness Line (2 user)

\[ x_1 = x_2 \]
Fairness: Additive Increase, Multiplicative Decrease

- Converges to fairness
- Converges to efficiency
- Increments smaller as fairness increases
  - effect on metrics?
**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 9 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 and implementation in the Internet
  - UDP
  - TCP

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