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

All material copyright 1996-2010
J.F Kurose and K.W. Ross, All Rights Reserved

Jim Kurose, Keith Ross
Pearson/Addison-Wesley, April 2016.
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/UDPSegments
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

host 1

<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 2

<table>
<thead>
<tr>
<th>P1</th>
<th>application</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td>network</td>
<td>link</td>
</tr>
<tr>
<td>network</td>
<td>link</td>
<td>physical</td>
</tr>
</tbody>
</table>

host 3

<table>
<thead>
<tr>
<th>P4</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport</td>
<td>network</td>
</tr>
<tr>
<td>network</td>
<td>link</td>
</tr>
<tr>
<td>physical</td>
<td></td>
</tr>
</tbody>
</table>
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

TCP/UDP segment format

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>other header fields</td>
<td></td>
</tr>
<tr>
<td>application data (message)</td>
<td></td>
</tr>
</tbody>
</table>

Transport Layer Segment

IP Datagram

Hn   Ht   M
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 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

```
1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

wraparound: 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

sum: 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 0
checksum: 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 1
```
Chapter 3 outline

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management

3.6 Principles of congestion control
3.7 TCP congestion control
Principles of Reliable data transfer

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

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

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

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

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

**receive side**
- **deliver_data()**: called by rdt to deliver data to upper
- **rdt_rcv()**: called when packet arrives on rcv-side of channel

**Diagram: Shown below the text**
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

![Finite State Machine Diagram]

**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)`
- `uedt_send(sndpkt)`

receiver

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

Wait for call from below

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

**Stop and wait**
Sender sends one packet, then waits for receiver response
rdt2.0: operation with no errors

\[
\begin{align*}
\text{rdt\_send(data)} \\
\text{snkpkt = make\_pkt(data, checksum)} \\
\text{udt\_send(sndpkt)} \\
\text{rdt\_rcv(rcvpkt) && isNAK(rcvpkt)} \\
\text{udt\_send(sndpkt)} \\
\text{Wait for call from above} \\
\text{Wait for ACK or NAK} \\
\text{\Lambda} \\
\text{rdt\_rcv(rcvpkt) && isACK(rcvpkt)} \\
\text{extract(rcvpkt, data)} \\
\text{deliver\_data(data)} \\
\text{udt\_send(ACK)} \\
\text{corrupt(rcvpkt)} \\
\text{udt\_send(NAK)} \\
\text{Wait for call from below} \\
\text{notcorrupt(rcvpkt)}
\end{align*}
\]
**rdt2.0: error scenario**

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

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

- `Wait for call from above`

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

- `Lambda` (Λ)

- `Wait for call from below`

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

- `Wait for ACK or NAK`
**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**

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \text{ && notcorrupt}(\text{rcvpkt}) \text{ && has_seq0}(\text{rcvpkt}) \)
  - \( \text{extract}(\text{rcvpkt}, \text{data}) \)
  - \( \text{deliver_data}(\text{data}) \)
  - \( \text{sndpkt} = \text{make_pkt}(\text{ACK}, \text{chksum}) \)
  - \( \text{udt}_\text{send}(\text{sndpkt}) \)

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \text{ && notcorrupt}(\text{rcvpkt}) \text{ && has_seq1}(\text{rcvpkt}) \)
  - \( \text{extract}(\text{rcvpkt}, \text{data}) \)
  - \( \text{deliver_data}(\text{data}) \)
  - \( \text{sndpkt} = \text{make_pkt}(\text{ACK}, \text{chksum}) \)
  - \( \text{udt}_\text{send}(\text{sndpkt}) \)

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \text{ && corrupt}(\text{rcvpkt}) \)
  - \( \text{sndpkt} = \text{make_pkt}(\text{ACK}, \text{chksum}) \)
  - \( \text{udt}_\text{send}(\text{sndpkt}) \)

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \text{ && notcorrupt}(\text{rcvpkt}) \text{ && has_seq1}(\text{rcvpkt}) \)
  - \( \text{extract}(\text{rcvpkt}, \text{data}) \)
  - \( \text{deliver_data}(\text{data}) \)
  - \( \text{sndpkt} = \text{make_pkt}(\text{ACK}, \text{chksum}) \)
  - \( \text{udt}_\text{send}(\text{sndpkt}) \)

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \text{ && notcorrupt}(\text{rcvpkt}) \text{ && has_seq0}(\text{rcvpkt}) \)
  - \( \text{sndpkt} = \text{make_pkt}(\text{NAK}, \text{chksum}) \)
  - \( \text{udt}_\text{send}(\text{sndpkt}) \)

- \( \text{rdt}_\text{rcv}(\text{rcvpkt}) \text{ && corrupt}(\text{rcvpkt}) \)
  - \( \text{sndpkt} = \text{make_pkt}(\text{NAK}, \text{chksum}) \)
  - \( \text{udt}_\text{send}(\text{sndpkt}) \)
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

- \( \text{sndpkt} = \text{make_pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt_send}(\text{sndpkt}) \)
- \( \text{rdt_send}(\text{data}) \)
- \( \text{rdt_rcv}(\text{rcvpkt}) \) 
  - \&\& (notcorrupt(\text{rcvpkt}) || \
  - has_seq0(\text{rcvpkt})) 
- extract(\text{rcvpkt}, \text{data})
- deliver_data(\text{data})
- \( \text{sndpkt} = \text{make_pkt}(\text{ACK},0, \text{chksum}) \)
- \( \text{udt_send}(\text{sndpkt}) \)

receiver FSM fragment

- \( \text{rdt_rcv}(\text{rcvpkt}) \) 
  - \&\& (corrupt(\text{rcvpkt}) || has_seq1(\text{rcvpkt}))
- \( \text{sndpkt} = \text{make_pkt}(\text{ACK},1, \text{chksum}) \)
- \( \text{udt_send}(\text{sndpkt}) \)
- Wait for 0 from below
- Wait for 1 from below
**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**

```plaintext
rdt_send(data)

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

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

wait for call 0 from above

wait for ACK0

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

wait for call 1 from above

wait for ACK1

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

timeout
udt_send(sndpkt)
start_timer

timeout
udt_send(sndpkt)
start_timer

rdt_send(data)

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

wait for rdt_rcv(rcvpkt) && ( corrupt(rcvpkt) || isACK(rcvpkt,0) )

wait for call 0 from above

wait for ACK1

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

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)
```

Diagram:
- **Wait for 0 from below**
  - rdt_rcv(rcvpkt) &&
    (notcorrupt(rcvpkt) || has_seq0(rcvpkt))
  - sndpkt = make_pkt(ACK,0, chksum)
  - udt_send(sndpkt)
- **Wait for 1 from below**
  - rdt_rcv(rcvpkt) &&
    (notcorrupt(rcvpkt) || has_seq1(rcvpkt))
  - 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:

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

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

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

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

\[
U_{SW} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027
\]
S&W Efficiency in Channel with Errors

- Let $1 - P_f$ = probability frame arrives w/o errors
- \textbf{Avg.} \# of transmissions to first correct arrival is then $1/ (1 - P_f)$
- “If 1-in-10 get through without error, then \textbf{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: \textit{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

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

- First packet bit transmitted, \( t = 0 \)
- Last bit transmitted, \( t = L / R \)
- First packet bit arrives
- Last bit of 2nd packet arrives, send ACK
- Last bit of 3rd packet arrives, send ACK
- ACK arrives, send next packet, \( t = RTT + L / R \)

\[
U_{\text{sender}} = \frac{3 \times L / R}{RTT + L / R} = \frac{0.024}{30.008} = 0.0008
\]

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)

![Diagram of Go-Back-N protocol](image)

- **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 $\rightarrow$ seq. #’s $\{0, 1, \ldots, 2^k-1\}$
- Sender’s window of (up to) N consecutive Unacked pkts allowed $N \leq 2^k - 1$

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

\[ \begin{align*}
\text{rdt\_send(data)} & \\
\text{if (nextseqnum < base+N) } & \\
\text{sndpkt[nextseqnum] =} & \\
\text{make\_pkt(nextseqnum, data, checksum)} & \\
\text{udt\_send(sndpkt[nextseqnum])} & \\
\text{if (base == nextseqnum)} & \\
\text{start\_timer} & \\
\text{nextseqnum++} & \\
\text{else} & \\
\text{refuse\_data(data)} & \\
\text{nextseqnum++} & \\
\text{else} & \\
\text{timeout} & \\
\text{start\_timer} & \\
\text{udt\_send(sndpkt[base])} & \\
\text{udt\_send(sndpkt[base+1])} & \\
\text{...} & \\
\text{udt\_send(sndpkt[nextseqnum-1])} & \\
\end{align*} \]

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 \textit{in-order} seq #

- may generate duplicate ACKs
- need only remember expectedseqnum

- out-of-order pkt:
  - \textit{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

pkt2 timeout
send pkt2
send pkt3
send pkt4
send pkt5

equiv. Send ACK5
Frame with errors and subsequent out-of-sequence frames are ignored
Transmitter is forced to go back when window of 4 is exhausted (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
  - \( t_f + N \frac{t_f}{1-P_f} \) if the first transmission does not succeed

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

\[ U_{GBN} = \frac{t_f}{t_{GBN}} = \frac{1 - P_f}{1 + (N - 1) P_f} \]
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

\( R_{\text{next}} \) \( R_{\text{next}+ N} \)

Buffers

Sender

\( S_{\text{last}} \) \( S_{\text{last}+1} \) \( S_{\text{recent}} \) \( S_{\text{last}+ N-1} \)

Receiver

\( R_{\text{next}} \) \( R_{\text{next}+1} \) \( R_{\text{next}+ N-1} \)

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)

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

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

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

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
Efficiency of Selective Repeat

- Assume $P_f =$frame loss probability, then number of transmissions required to deliver a frame is:
  - $\frac{t_f}{1-P_f}$ where $t_f = \frac{L}{R}$

$$U_{SR} = \frac{t_f}{t_f/(1-P_f)} = (1-P_f)$$
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  Host B

SYN, Seq_no = x

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

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

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

![Diagram showing data transfer process](image-url)
**TCP segment structure**

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>Receive window</td>
<td>Receive window</td>
</tr>
<tr>
<td>Urg data pnter</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Application data</td>
</tr>
</tbody>
</table>

- **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)

**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
  - Identifies 1<sup>st</sup> 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.
- \( 0 \leq SN \leq 2^{32}-1 \)
- Initial sequence number selected randomly during connection setup

Acknowledgement Number (32 bits)
- SN of next byte expected by receiver
- 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) CongestionWindow (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
TCP sender (simplified)

- **wait for event**
  - NextSeqNum = InitialSeqNum
  - SendBase = InitialSeqNum

- 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

- 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
  }
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 (“2MSL wait state”)</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 ACK</td>
</tr>
</tbody>
</table>

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

TCP client lifecycle

TCP server lifecycle
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]
- SampleRTT will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT
TCP Round Trip Time and Timeout

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

- 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: \( \alpha = 0.125 \).
Example RTT estimation:

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

- SampleRTT
- Estimated RTT
TCP Round Trip Time and Timeout

Setting the timeout timer (RTO)

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT $\rightarrow$ larger safety margin
- estimate of how much SampleRTT deviates from EstimatedRTT:

\[
\text{DevRTT} = (1-\beta)\text{DevRTT} + \\
\beta|\text{SampleRTT} - \text{EstimatedRTT}|
\]

(typically, $\beta = 0.25$)

Then set timeout interval as below (this calculation is done every time a timer is started)

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4\text{DevRTT}
\]

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 segments)
- expiration interval: $TimeOutInterval$ or $RTO$

**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)
TCP: retransmission scenarios

Host A

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seq=92, 8 bytes data</td>
</tr>
<tr>
<td></td>
<td>ACK=100</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>loss</td>
</tr>
</tbody>
</table>

Host B

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seq=92 timeout</td>
</tr>
</tbody>
</table>

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

- Lost ACK scenario
- Premature timeout
- TCP: retransmission scenarios
- Host A
- Host B
TCP retransmission scenarios (more)

Cumulative ACK scenario

Host A

- Seq=92, 8 bytes data
- Seq=100, 20 bytes data
- ACK=100
- ACK=120

Host B

- SendBase = 120
- timeout
- loss

time

Transport Layer 3-89
## TCP ACK generation [RFC 1122, RFC 2581]

<table>
<thead>
<tr>
<th>Event at Receiver</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed</td>
<td>Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK</td>
</tr>
<tr>
<td>Arrival of in-order segment with expected seq #. One other segment has ACK pending</td>
<td>Immediately send single cumulative ACK, ACKing both in-order segments</td>
</tr>
<tr>
<td>Arrival of out-of-order segment higher-than-expect seq. #. Gap detected</td>
<td>Immediately send duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>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
Figure 3.37 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
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 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
**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 receive buffer doesn't overflow

RcvWindow = 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
- no retransmission

- large delays when congested
- maximum achievable throughput
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

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

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

R/2

finite shared output link buffers

\( \lambda_{\text{in}} \): original data

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

free buffer space!
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?)

\[ \lambda_{out} \]

\[ \lambda_{in} \]

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

\[ \lambda_{out} \]

free buffer space!
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.

when sending at R/2, some packets are retransmissions including duplicated that are delivered!
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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

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

another “cost” of congestion:

- when packet dropped, any “upstream transmission capacity used for that packet was wasted!
Chapter 3 outline

3.1 Transport-layer services
3.2 Multiplexing and demultiplexing
3.3 Connectionless transport: UDP
3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP
   ▪ segment structure
   ▪ reliable data transfer
   ▪ flow control
   ▪ connection management

3.6 Principles of congestion control
3.7 TCP congestion control
TCP Congestion Control: 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 it’s 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 → TCP is “self-clock
Congestion Control: Send Side

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

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 Rwnd →

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

- For **long TCP flows**, Cwind achieves a (dynamic) steady-state as below quickly -
  - *Cwind 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

![Sawtooth behavior: probing for bandwidth](image.png)
TCP Congestion Control: details

Assume RcvWindow >> Cwnd

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

Inst. send rate = \( \frac{\text{cwnd}}{\text{RTT}} \) Bytes/sec

- cwnd is dynamic, varies from round-to-round
  Round (cycle): 1 RTT duration

- Note: If Cwnd is in TCP segments \( \rightarrow \) Cwnd X MSS to convert to Bytes.

How does sender perceive congestion?

- loss event = \text{timeout or 3 duplicate acks}
- TCP sender reduces rate (cwnd) after loss event

three phases
  - slow start (transient)
  - Additive Increase, Multiplicativve 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 $cwnd = 1$ 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 $\rightarrow$ II. Cong. Avoid (AIM)

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

Multiplicative Decrease

Add one packet each RTT
TCP Slow Start

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

- Initial available bandwidth for the connection may be >> 20 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 \[ \text{SS threshold} = \frac{1}{2} \times \text{current cwnd} \]

Set new \[ \text{Cwnd} = \text{SSThreshold} + 3 \]

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

Example: If current value of \( \text{Cwnd} = 20 \) MSS and a loss is detected, new SSThreshold = 10 and new Cwnd = 13 MSS for the next AI-MD cycle.

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

Transmission Round

<table>
<thead>
<tr>
<th>Congestion Window (in segments)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast re-tx</td>
</tr>
<tr>
<td>timeout</td>
</tr>
<tr>
<td>drops to 1</td>
</tr>
<tr>
<td>W/2</td>
</tr>
<tr>
<td>fast re-tx</td>
</tr>
</tbody>
</table>

Evolution Graph:
- Congestion Window increases with each transmission round.
- Fast retransmission (fast re-tx) is triggered when a segment is lost.
- A timeout occurs, halving the congestion window to W/2.
- The process repeats with each new transmission round.
Summary: TCP Congestion Control

- **Slow Start**
  - cwnd = 1 MSS
  - ssthresh = 64 KB
  - dupACKcount = 0
  - retransmit missing segment

- **Congestion Avoidance**
  - cwnd > ssthresh
  - ssthresh = cwnd/2
  - cwnd = 1 MSS
  - dupACKcount = 0
  - retransmit missing segment

- **Fast Recovery**
  - dupACKcount == 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
  - retransmit missing segment

- **New ACK**
  - cwnd = cwnd + MSS (MSS/cwnd)
  - dupACKcount = 0
  - transmit new segment(s), as allowed

- **Timeout**
  - ssthresh = cwnd/2
  - cwnd = 1 MSS
  - dupACKcount = 0
  - retransmit missing segment

- **Duplicate ACK**
  - dupACKcount++

- **New Duplicate ACK**
  - cwnd = ssthresh
  - dupACKcount = 0
  - transmit new segment(s), as allowed

- **Fast Recovery**
  - dupACKcount = 0
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
  - retransmit missing segment

- **Slow Start**
  - cwnd = cwnd + MSS
  - dupACKcount = 0
  - transmit new segment(s), as allowed

- **Slow Start**
  - cwnd = 1 MSS
  - ssthresh = 64 KB
  - dupACKcount = 0
  - retransmit missing segment

- **Slow Start**
  - dupACKcount == 3
  - ssthresh = cwnd/2
  - cwnd = ssthresh + 3
  - retransmit missing segment

- **New ACK**
  - cwnd = cwnd + MSS
  - dupACKcount = 0
  - transmit new segment(s), as allowed

- **Slow Start**
  - cwnd = 1 MSS
  - ssthresh = 64 KB
  - dupACKcount = 0
  - retransmit missing segment

- **Slow Start**
  - cwnd = cwnd + MSS
  - dupACKcount = 0
  - transmit new segment(s), as allowed

Transport Layer 3-122
TCP throughput

- What’s the average throughput of TCP as a function of window size and RTT?
  - Ignore slow start
- Let $W$ be the window size when loss occurs.
  - When window is $W$, throughput is $W/RTT$
  - Just after loss, window drops to $W/2$, throughput to $W/2RTT$.
  - (Long-term) average throughput: $0.75 \frac{W}{RTT}$
TCP Fairness

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

- **Too slow**
  - Fail to take advantage of available bandwidth → underload

- **Too fast**
  - Overshoot knee → overload, high delay, loss

- **Everyone’s doing it**
  - May all under/over shoot → large oscillations

- **Optimal:**
  - \( \sum x_i = X_{goal} \)

- Efficiency = 1 - distance from efficiency line
Fair Allocation

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

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

- Assumes no knowledge of priorities
- Fairness = 1 - distance from fairness line
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”