Chapter 3: Transport Layer

Our goals:
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

Transport services and protocols
- provide logical communication between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP
**Transport vs. network layer**

- **network layer**: logical communication between hosts
- **transport layer**: logical communication between processes
  - relies on, enhances, network layer services

  **Household analogy:**
  - 12 members of household sending letters to 12 members of another household
  - app messages = letters in envelopes
  - processes = family members
  - hosts = houses
  - transport protocol = family mail handlers Ann and Bill
  - 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

**E2E argument** [Saltzer, Reed, Clark - 1984]

- Philosophy behind Internet design: Move complex operations to the edges of the network. Why?
  - Not all apps may require complex operations, e.g., reliability (audio, video), security
  - Also, some functionality hard to implement in network core: duplicates suppression, FIFO ordering
  - Ops often repeated at edge anyways as safety check
  - Keeps authority over more complex operations local

- Implications of E2E argument to the Internet: most complex ops should be performed toward the top of the protocol stack.

**E2E argument pros and cons**

- **Pros**:
  - reduces network complexity - eases deployment, network recovery
  - reduces redundant checks since app often provides checks anyways
  - network bugs easier to fix
- **Cons**:
  - network less efficient (e.g., hop-to-hop reliability would reduce BW requirements and delivery delays)
  - more responsibility lies with the application: longer development cycle, frequent bugs


**Chapter 3 outline**

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

**Multiplexing/demultiplexing**

Demultiplexing at rcv host: delivering received segments (4-PDUs) to correct socket

Multiplexing at send host: gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)

**How demultiplexing works**

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

**Connectionless demultiplexing**

- Create sockets with port numbers:
  - DatagramSocket mySocket1 = new DatagramSocket(9111);
  - DatagramSocket mySocket2 = new DatagramSocket(9222);
- UDP socket identified by two-tuple:
  - (dest IP address, dest port number)
- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- Possible for IP datagrams with different source IP addresses and/or source port numbers to be directed to same socket
Connectionless demux (cont)

DatagramSocket serverSocket = new DatagramSocket(6428);

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

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's main service seems to be its "lack of service"! Why do we not send datagrams directly?

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

Sender:
- treat segment contents as sequence of 16-bit integers
- checksum addition (1's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

Receiver:
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected, But maybe errors nonetheless? More later...

UDP: more

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

UDP segment format

<table>
<thead>
<tr>
<th>32 bits</th>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length, in bytes of UDP segment, including header</td>
<td>Length</td>
<td>checksum</td>
</tr>
</tbody>
</table>

Application data (message)

Chapter 3 outline

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

- Important in app., transport, link layers
- Top-10 list of important networking topics

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

Reliable data transfer: getting started

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

Reliable data transfer over a reliable channel

- Assumption: underlying channel perfectly reliable
  - No bit errors
  - No loss of packets
- Separate FSMs for sender, receiver:
  - Sender sends data into underlying channel
  - Receiver reads data from underlying channel
Rdt2.0: channel with bit errors

- Assumption: underlying channel may flip bits in packet
  - recall: UDP 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

rtdt2.0: FSM specification

```
<table>
<thead>
<tr>
<th>Event</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>rdt_send(data)</td>
<td>Make pkt(data, checksum) udt_send(sndpkt)</td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;</td>
<td>udt_send(ACK)</td>
</tr>
<tr>
<td>notcorrupt(rcvpkt)</td>
<td></td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;</td>
<td>udt_send(NAK)</td>
</tr>
<tr>
<td>isACK(rcvpkt)</td>
<td></td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt)</td>
<td>rdt_send(sndpkt)</td>
</tr>
<tr>
<td>rdt_send(data)</td>
<td></td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;</td>
<td>udt_send(NAK)</td>
</tr>
<tr>
<td>isNAK(rcvpkt)</td>
<td></td>
</tr>
<tr>
<td>rdt_rcv(rcvpkt) &amp;</td>
<td>rdt_send(sndpkt)</td>
</tr>
<tr>
<td>corrupt(rcvpkt)</td>
<td></td>
</tr>
<tr>
<td>rdt_send(data)</td>
<td></td>
</tr>
</tbody>
</table>
```

sender

receiver
**rdt2.0 has a fatal flaw!**

What happens if ACK/NAK corrupted?
- sender doesn’t know what happened at receiver
- can’t just retransmit: possible duplicate

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

Handling duplicates:
- sender adds sequence number to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn’t deliver up) duplicate pkt

Rdt2.0 is an example of a stop and wait protocol. Sender sends one packet, and then waits for receiver response.

**rdt2.1: sender, handles garbled ACK/NAKs**

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**

- sender FSM
- receiver FSM

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

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

**rdt3.0 sender**

- sender FSM
- receiver FSM
Performance of rdt3.0

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

\[
T_{\text{trans}} = \frac{L}{R} = \frac{8 \times 10^5 \text{ b/pkt}}{10^9 \text{ b/sec}} = 8 \times 10^{-6} \text{ sec}
\]

\[
U_{\text{sender}} = \frac{T_{\text{trans}}}{RTT + T_{\text{trans}}} = \frac{8 \times 10^{-6}}{30 \times 10^{-3} + 8 \times 10^{-6}} = 0.00027
\]

- \(U_{\text{sender}}\): utilization - fraction of time sender busy sending
- 1KB pkt every 30 msec → 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!
**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**

---

**Go-Back-N**

**Sender:**
- k-bit seq # in pkt header—Seq. #s: 0, 1, …, 2^k-1
- "window" of up to N, consecutive unack'd pkts allowed (N<2k)
- 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**

- All arithmetic operations are modulo 2^k, where k = # of bits in the sequence numbers.
- N must be strictly less than 2^k
**Transport Layer**

**Selective Repeat**
- Receiver individually acknowledges all correctly received packets.
  - Buffers packets, as needed, for eventual in-order delivery to upper layer.
- Sender only retransmits packets for which ACK not received.
  - Sender timer for each unACKed pkt.
- Sender window:
  - $N$ consecutive seq #'s.
  - Again limits seq #'s of sent, unACKed pkts.

**Selective Repeat: sender, receiver windows**

---

**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 expected seqnum.
- Out-of-order pkt:
  - Discard (Don't buffer) -> No receiver buffering.
  - Re-ACK pkt with highest in-order seq #.

**GBN in action**

---
Selective repeat (SR)

Sender:
- Data from above:
  - If next available seq # in window, send pkt
- Timeout(n):
  - Resend pkt n, restart timer
- ACK(n) in [sendbase, sendbase+N]
  - Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

Receiver:
- Pkt n in [rcvbase, rcvbase-N-1]
  - Send ACK
- Out-of-order: Buffer
  - In-order: Deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt
  - Pkt n in [rcvbase-N, rcvbase-1]
    - ACK(n)
  - Otherwise:
    - Ignore

Selective repeat in action

Sequence #s and SR window size

- $2^m = \text{# seq. #s}$
  - $W_s = \text{Tx window size}$
  - $W_r = \text{Rx window size}$

- To avoid "wraparound", seq. #s of timedout packets in Tx window must never lie in the Rx window
  - $\text{Tx and Rx #s must be disjoint}$
  - $2^m \geq W_r \cdot W_s$

Special cases
- Stop-and-wait (rdt3.0) is SR with $W_r = W_s = 1$
- Go-Back-N is SR with $W_r = 2^{m-1}$ and $W_s = 1$

In general we want:
- $W_r$ large enough to "fill the pipe", but no larger, to keep buffer overhead low, e.g., set $W_r = [R \cdot RTT / L + 1]$
- $W_s$ as large as possible to delay onset of GBN behavior, but $W_r$, to keep buffer overhead low, e.g., set $W_r = W_s$. 
Transport Layer

Stop and wait and selective repeat performance in the presence of error

- **Model**
  - Packets lost/corrupted with probability $p$.
  - ACKs never lost/corrupted.
  - $U_{\text{stop and wait}} = \frac{T_{\text{R}}}{\text{RTT} + T_{\text{u}}}$
  - $U_{\text{SR}} = \frac{T_{\text{R}}}{\text{RTT} + T_{\text{u}}}$

- For stop-and-wait and SR with $W_t = W_r$
  - $U_{\text{SR}} = (1-p) \frac{W_{\text{t}}}{\text{RTT} + T_{\text{u}}}$
  - $E[\text{time between successful XMTs}] = \frac{1}{1-p} \frac{(1-p)\text{RTT} + T_{\text{u}}}{\text{RTT} + T_{\text{u}}}$ (from HW1-9)

Go-back-$N$ performance in the presence of error

- For GBN with $N=W_t$, when a lost/corrupted packet is detected the $K$ packets "in flight" at the time of detection must be resent
  - $K=W_t$ must be resent when $W_t < \left\lceil \frac{R \cdot \text{RTT}}{L} + 1 \right\rceil$
  - $K=\left\lceil \frac{R \cdot \text{RTT}}{L} + 1 \right\rceil$ must be resent when $W_t \geq \left\lceil \frac{R \cdot \text{RTT}}{L} + 1 \right\rceil$

ARQ protocol performance for packet loss prob=$10^{-3}$, ACK loss prob=0

- Plot of utilization (fraction of time the channel is in use) versus normalized one-way delay ($\text{packets"in the pipe"}/\text{packet length}$)

- **Comments:**
  - Wrt utilization, GBN and SR outperform stop-and-wait.
  - Advantage of SR over GBN becomes apparent as $N$ and normalized one-way delay increase.
  - Plot highlights fact that bandwidth-delay product increases destroy utilization unless packet length is also increased.

Chapter 3 outline

- **3.1 Transport-layer services**
  - Segment structure
  - Reliable data transfer
  - Flow control
  - Connection management

- **3.3 Connectionless transport: UDP**

- **3.4 Principles of reliable data transfer**

- **3.5 Connection-oriented transport: TCP**
  - Reliable data transfer
  - Flow control
  - Connection management
  - Principles of congestion control
  - TCP congestion control
TCP: Overview

- **Point-to-point:** one sender, one receiver
- **Reliable, in-order byte stream:** no "message boundaries"
- **Full duplex data:** bi-directional data flow in same connection
- **MSS:** maximum segment size
- **Connection-oriented:**
  - Handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- **Flow controlled:** sender will not overwhelm receiver
- **Send & receive buffers**

TCP segment structure

- **Source port #**
- **Destination port #**
- **Sequence number**
- **Acknowledgement number**
- **Data** (variable length)
- **Options** (variable length)
  - **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)

TCP seq. #’s and ACKs

- **Seq. #’s**:
  - Byte stream "number" of first byte in segment’s data
- **ACKs**:
  - Seq # of next byte expected from other side
  - Cumulative ACK
  - How receiver handles out-of-order segments
- **Q:** TCP spec doesn’t say - up to implementer

TCP Round Trip Time and Timeout

- **Q:** how to set TCP timeout value?
  - Longer than RTT
  - But RTT varies
  - Too short: premature timeout
  - Unnecessary retransmissions
  - Too long: slow reaction to segment loss

- **Q:** how to estimate RTT?
  - **SampleRTT:** measured time from segment transmission until ACK receipt
  - Ignore retransmissions
  - **SampleRTT** will vary, want estimated RTT "smoother"
  - Average several recent measurements, not just current SampleRTT
**Exponentially Weighted Moving Average**

- Useful when average is time-varying
- Let $A_t$ be the average computed for time $t = 0, 1, 2, \ldots$
- Let $S_t$ be the sample taken at time $t$
- Let $x \in [0, 1]$ be the weight
- $A_0 = S_0$
- $A_t = (1-x) A_{t-1} + x S_t$ for $t > 0$

$A_t$ has "Desirable" average features:
- If $S_i = C$ for all $i$, then $A_t = C$
- If $\lim S_i = C$, then $\lim A_t = C$
- If $C_1 \leq S_i \leq C_2$ for all $i$, then $C_1 \leq A_t \leq C_2$
- A larger $x$ means more emphasis on recent measurements, less on history (e.g., $x = 1$ gives $A_t = S_t$)

**TCP Round Trip Time and Timeout**

- EstimatedRTT = $(1-\alpha) * $EstimatedRTT + $\alpha * $SampleRTT
- exponential weighted moving average
- Influence of given sample decreases exponentially fast
- typical value of $\alpha$: 0.125

**Setting the timeout**

- EstimatedRTT plus "safety margin"
- large variation in EstimatedRTT $\Rightarrow$ larger safety margin
- first estimate how much SampleRTT deviates from EstimatedRTT:
  - Deviation = $(1-\beta) * $Deviation + $\beta * |SampleRTT - EstimatedRTT|
  - typically, $\beta = 0.25$

Then set timeout interval:
- Timeout = EstimatedRTT + 4 * Deviation

**Example RTT estimation:**

<table>
<thead>
<tr>
<th>Time (seconds)</th>
<th>SampleRTT</th>
<th>Estimated RTT</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>150</td>
<td>150</td>
</tr>
<tr>
<td>20</td>
<td>200</td>
<td>200</td>
</tr>
<tr>
<td>30</td>
<td>250</td>
<td>250</td>
</tr>
<tr>
<td>40</td>
<td>300</td>
<td>300</td>
</tr>
</tbody>
</table>

**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
- 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 single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate ACKs
- Initially consider simplified TCP sender:
  - ignore duplicate ACKs
  - ignore flow control, congestion control

TCP sender events:

- data rcvd from app:
  - Create segment with seq #
  - seq # is byte-stream number of first data byte in segment
  - start timer if not already running (think of timer as for oldest unACKed segment)
- ACK rcvd:
  - If acknowledges previously unACKed segments
    - update what is known to be ACKed
    - start timer if there are outstanding segments

TCP: retransmission scenarios

Host A
Seq=100
SendBase=100

Host B
Seq=92
SendBase=92

Timeout event:
- premature timeout
- 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

Example:
- SendBase-1: last cumulatively ACKed byte
- Example:
  - SendBase-1 = 71; y = 73, so the rnr wants 73+; y > SendBase, so that new data is ACKed

SendBase = 100
SendBase = 120
SendBase = 120
SendBase = 120
TCP retransmission scenarios (more)

### Fast Retransmit
- **Time-out period often relatively long:**
  - Long delay before resending lost packet
- **Detect lost segments via duplicate ACKs:**
  - Sender often sends many segments back-to-back
  - If segment is lost, there will likely be many duplicate ACKs.
- **If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:**
  - Fast retransmit: resend segment before timer expires

### 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-expected 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>Immediate send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

**Fast retransmit algorithm:**

```c
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
    }
}
debug("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

TCP Flow Control

- Receive side of TCP connection has a receive buffer:
  - spare room in buffer
    - RcVWindow
    - RcVBuffer - [LastByteRcvd - LastByteRead]
  - app process may be slow at reading from buffer

TCP Flow control: how it works

- RcVr advertises spare room by including value of RcVWindow in segments
- Sender limits unACKed data to RcVWindow
  - guarantees receive buffer doesn't overflow
- speed-matching service: matching the send rate to the receiving app's drain rate

Chapter 3 outline

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

Recall: TCP sender, receiver establish "connection" before exchanging data segments
- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. RcvWindow)
- client: connection initiator
  - Socket clientSocket = new Socket("hostname","port number");
- server: contacted by client
  - Socket connectionSocket = welcomeSocket.accept();

Three way handshake:

Step 1: client host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment
- server allocates buffers
- specifies server initial seq. #

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

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.

The state of active TCP connections on UNIX/WINDOWS machines can be viewed using the "netstat" command.
**Chapter 3 outline**

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

**Principles of Congestion Control**

**Congestion:**
- informally: “too many sources sending too much data too fast for network to handle”
- different from flow control
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!
**Transport Layer**

### Causes/costs of congestion: scenario 2

- **always**: $\lambda_{in} = \lambda_{out}$ (goodput)
- "perfect" retransmission only when no loss, otherwise: $\lambda_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes $\lambda_{in}$ larger (than perfect case) for same $\lambda_{out}$

**"costs" of congestion:**

- more work (retrans) for given "goodput"
- unneeded retransmissions: link carries multiple copies of pkt

### Approaches towards congestion control

Two broad approaches towards congestion control:

- **End-end congestion control:**
  - no explicit feedback from network
  - congestion inferred from end-system observed loss, delay
  - approach taken by TCP

- **Network-assisted congestion control:**
  - routers provide feedback to end systems
  - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
  - explicit rate sender should send at
**Case study: ATM ABR congestion control**

**ABR: available bit rate:**
- "elastic service"
- if sender's path "underloaded": sender should use available bandwidth
- if sender's path congested: sender throttled to minimum guaranteed rate

**RM (resource management) cells:**
- sent by sender, interspersed with data cells
- bits in RM cell set by switches ("network-assisted")
  - NI bit: no increase in rate (mild congestion)
  - CI bit: congestion indication
- RM cells returned to sender by receiver, with bits intact

**Transport Layer**

**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**

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

- Roughly, \( w \) segments, each with \( \text{MSS} \) bytes, sent every RTT, as allowed by \( \text{CongWin} \)

- rate = \( \frac{\text{MSS} \times \text{CongWin}}{\text{RTT} \times \text{RTT}} \)

- \( \text{CongWin} \) is dynamic, function of perceived network congestion
  - \( \text{MSS} = \) "maximum segment size"

- How does sender perceive congestion?
  - loss event = timeout or 3 duplicate acks
  - TCP sender reduces rate (\( \text{CongWin} \)) after loss event

- three mechanisms:
  - AIMD
  - slow start
  - conservative after timeout events
TCP AIMD

- **Multiplicative decrease:**
  - Cut CongWin in half after loss event

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

TCP Slow Start

- When connection begins, CongWin = 1 MSS
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate = 20 kbps
- Available bandwidth may be >> MSS/RTT
  - Desirable to quickly ramp up to respectable rate

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - Double CongWin every RTT
  - Done by incrementing CongWin for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast

Refinement

- After 3 dup ACKs:
  - CongWin is cut in half
  - Window then grows linearly
  - But after timeout event:
    - CongWin instead set to 1 MSS
    - Window then grows exponentially
    - To a threshold, then grows linearly

Philosophy:

- "Fast recovery" only implemented in current TCP version (Reno).
  - "3 dup ACKS indicates network capable of delivering some segments"
Refinement (more)

Q: When should the exponential increase switch to linear?
A: When CongWin gets to 1/2 of its value before timeout.

Implementation:
- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event

Summary: TCP Congestion Control

- When CongWin is below Threshold, sender in slow-start phase, window grows exponentially.
- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.

TCP Fairness

Fairness goal: if N TCP sessions share same bottleneck link, each should get 1/N of link capacity

AIMD

- TCP avoids congestion through use of AIMD: additive increase, multiplicative decrease
  - increase window by 1 per RTT when CongWin is above threshold.
  - decrease window by factor of 2 on loss event

Why is AIMD fair and congestion-avoiding?

Pictorial View: Two sessions compete for a link's bandwidth, R

A good CC protocol will always converge toward the desired region.
**Chiu/Jain model assumptions**

- Sessions can sense whether link is overused or underused (e.g., via lost pkts).
- Sessions cannot compare relative rates (i.e., don’t know of each other’s existence).
- Sessions adapt rates round-by-round
  - adapt simultaneously
  - in same direction (both increase or both decrease).

**AIMD Convergence (Chiu/Jain)**

Additive Increase - up at 45º angle

Multiplicative Decrease - down toward the origin

C/J also show other combos (e.g., AIAD) don’t converge!

**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
- Research area: TCP friendly

**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 !

**Delay modeling**

**Q:** How long does it take to receive an object from a Web server after sending a request?

**Ignoring congestion, delay is influenced by:**

- TCP connection establishment
- data transmission delay
- slow start

**Notation, assumptions:**

- Assume one link between client and server of rate R
- S: MSS (bits)
- O: object size (bits)
- no retransmissions (no loss, no corruption)

**Window size:**

- First assume: fixed congestion window, W segments
- Then dynamic window, modeling slow start
Transport Layer

Fixed congestion window (1)

First case:
\( \text{WS}/R > \text{RTT} + S/R \): ACK for first segment in window returns before window's worth of data sent

\[ \text{delay} = 2 \text{RTT} + O/R \]

Transport Layer 3-105

Fixed congestion window (2)

Second case:
\( \hat{\text{WS}}/R < \text{RTT} + S/R \): wait for ACK after sending window's worth of data sent

\[ \text{delay} = 2 \text{RTT} + O/R + (K-1)[S/R + \text{RTT} - \text{WS}/R] \]

where
\[ K := O/ WS \]

the number of windows of data that cover the object.

Transport Layer 3-106

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

We will show that the delay for one object is:

\[ \text{Latency} = 2 \text{RTT} + \frac{O}{R} + P \left[ \text{RTT} + \frac{S}{R} - (2^P - 1) \frac{S}{R} \right] \]

where \( P \) is the number of times TCP idles at server:

\[ P = \min(Q, K - 1) \]

\( Q \) is the number of times the server idles if the object were of infinite size, and

\( K \) is the number of windows that cover the object.

TCP Delay Modeling: Slow Start (2)

Delay components:

- 2 RTT for connection estab and request
- \( O/R \) to transmit object
- \( Q \# \) times server would idle due to slow start if the object were of infinite size

Server idles:

\[ P = \min(K, Q) \]

Example:

- \( O/S = 15 \) segments
- \( K = 4 \) windows
- \( Q = 2 \)

Server idles \( P = 2 \) times

Transport Layer 3-107
TCP Delay Modeling (3)

$$\tau = RTT = \text{time from when server starts to send segment until server receives acknowledgement}$$

$$2^{-k} \frac{S}{R} = \text{time to transmit the } k\text{th window}$$

$$\frac{S}{R} + RTT - 2^{-k} \frac{S}{R} = \text{idle time after the } k\text{th window}$$

$$\text{delay} = \frac{O}{R} + 2RTT + \sum_{j=1}^{\min\{2^{k}, O/S\}} \text{idles}$$

TCP Delay Modeling (4)

Recall $$K = \text{number of windows that cover object}$$

How do we calculate $$K$$?

$$K = \min\{k : 2^k S + 2^{k-1} S + \cdots + 2^{1-k} S \geq O\}$$

$$= \min\{k : 2^k - 1 \geq \frac{O}{S}\}$$

$$= \left\lfloor \log_2 \left(\frac{O}{S} + 1\right)\right\rfloor$$

Calculation of $$O$$, number of idles for infinite-size object, is similar (see HW).

HTTP Modeling

- Assume Web page consists of:
  - 1 base HTML page (of size $$O$$ bits)
  - $$M$$ images (each of size $$O$$ bits)
- Non-persistent HTTP:
  - $$M+1$$ TCP connections in series
  - Response time = $$(M+1)O/R + (M+1)2RTT + \text{sum of idle times}$$
- Persistent HTTP:
  - 2 $$RTT$$ to request and receive base HTML file
  - 1 $$RTT$$ to request and receive $$M$$ images
  - Response time = $$(M+1)O/R + 3RTT + \text{sum of idle times}$$
- Non-persistent HTTP with $$X$$ parallel connections
  - Suppose $$M/X$$ integer.
  - 1 TCP connection for base file.
  - $$M/X$$ sets of parallel connections for images.
  - Response time = $$(M+1)O/R + (M/X + 1)2RTT + \text{sum of idle times}$$

HTTP Response time (in seconds)

$$RTT = 100 \text{ msec, } O = 5 \text{ Kbytes, } M=10 \text{ and } X=5$$

For low bandwidth, connection and response time dominated by transmission time so persistent connections only give minor improvement over parallel connections. The advantage over non-persistent connections increases with bandwidth.
HTTP Response time (in seconds)

RTT = 1 sec, O = 5 Kbytes, M=10 and X=5

For larger RTT, response time dominated by TCP establishment & slow start delays. Persistent connections now give important improvement, particularly in high delay-bandwidth networks.

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"