Chapter 3: Transport Layer

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

3.1: Transport services and protocols

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

Transport vs. network layer

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

household analogy:
12 kids in Ann’s house sending letters to 12 kids in Bill’s house:
- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of “best-effort” IP
- services not available:
  - delay guarantees
  - bandwidth guarantees
3.2: Multiplexing/demultiplexing

**Multiplexing at sender:**
- Handle data from multiple sockets, add transport header (later used for demultiplexing)

**Demultiplexing at receiver:**
- Use header info to deliver received segments to correct socket

How demultiplexing works

- Host receives IP datagrams
  - Each datagram has source IP address, destination IP address
  - Each datagram carries one transport-layer segment
  - Each segment has source, destination port number
- Host uses IP addresses & port numbers to direct segment to appropriate socket

![Diagram showing network and socket relationships](image)

Connectionless demultiplexing

- **Recall:** created socket has host-local port #:
  - `DatagramSocket mySocket1 = new DatagramSocket(12534);`

- **Recall:** when creating datagram to send into UDP socket, must specify
  - Destination IP address
  - Destination port #

- When host receives UDP segment:
  - Checks destination port # in segment
  - Directs UDP segment to socket with that port #

IP datagrams with same dest. port #, but different source IP addresses and/or source port numbers will be directed to same socket at dest

Connectionless demux: example

- `DatagramSocket serverSocket = new DatagramSocket(6428);`
- `DatagramSocket mySocket1 = new DatagramSocket(5775);`
- `DatagramSocket mySocket2 = new DatagramSocket(9157);`

![Diagram showing datagram socket interactions](image)
**Connection-oriented demux**

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- demux: receiver uses all four values to direct segment to appropriate socket
- server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

---

**Connection-oriented demux: example**

**3.3: UDP: User Datagram Protocol [RFC 768]**

- “no frills,” “bare bones” Internet transport protocol
- “best effort” service, UDP segments may be:
  - lost
  - delivered out-of-order to app
- connectionless:
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- UDP use:
  - streaming multimedia apps (loss tolerant, rate sensitive)
  - DNS
  - SNMP
  - reliable transfer over UDP:
    - add reliability at application layer
    - application-specific error recovery!
UDP: segment header

- 32 bits
- source port #
- dest port #
- length
- checksum
- application data (payload)

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

Internet checksum: example

element: add two 16-bit integers

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

+------------+----------+
| sum        | checksum |
| 1 0 1 1 1 0| 1 1 1 1 0|
| 1 1 1 0 0 0| 1 0 0 0 1|

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

UDP checksum

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

sender:
- treat segment contents, including header fields, as sequence of 16-bit integers
- simple: no connection state at sender, receiver
- small header size
- no congestion control: UDP can blast away as fast as desired

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

Internet checksum: example

example: add two 16-bit integers

<p>| | | | | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

+------------+----------+
| sum        | checksum |
| 1 0 1 1 1 0| 1 1 1 1 0|
| 1 1 1 0 0 0| 1 0 0 0 1|

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

Principles of reliable data transfer

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

characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)
Reliable data transfer: getting started

- rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer
- deliver_data(): called by rdt to deliver data to upper

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
  - error detection
  - feedback: control msgs (ACK, NAK) from receiver to sender

Problems!

- what happens if ACK/NAK corrupted?
  - sender doesn’t know what happened at receiver!
  - can’t just retransmit: possible duplicate
  - handling duplicates:
    - sender retransmits current pkt if ACK/NAK corrupted
    - sender adds sequence number to each pkt
    - receiver discards (doesn’t deliver up) duplicate pkt
  - stop and wait
    - sender sends one packet, then waits for receiver response

Discussion

- sender:
  - seq # added to pkt
  - two seq. #’s (0,1) will suffice. Why?
  - must check if received ACK/NAK corrupted
- receiver:
  - must check if received packet is duplicate
    - state indicates whether 0 or 1 is expected pkt seq #
Channels with errors and loss

**new assumption:** underlying channel can also lose packets (data, ACKs)

- checksum, seq. #, ACKs, retransmissions will be of help ... but not enough

**approach:** sender waits “reasonable” amount of time for ACK

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

---

**Stop and wait**

- **sender**
  - send pkt0
  - rcv pkt0
  - send pkt1
  - rcv pkt1
  - send pkt0

- **receiver**
  - pkt0
  - ack0
  - send ack0
  - pkt1
  - ack1
  - send ack1
  - pkt0
  - ack0
  - send ack0

(a) no loss

(b) packet loss

---

**Performance**

- Stop & wait is correct, but performance stinks
- e.g.: 1 Gbps link, 15 ms prop. delay, 8000 bit packet:
  
  \[ D_{\text{trans}} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs} \]

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

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

  network protocol limits use of physical resources!
stop-and-wait operation

Pipelined protocols

Pipelining: increased utilization

Pipelined protocols: overview
**Go-Back-N: sender**

- k-bit seq # in pkt header
- “window” of up to N, consecutive unack’ed pkts allowed
- ACK(n): ACKs all pkts up to, including seq # n - “cumulative ACK”
  - may receive duplicate ACKs (see receiver)
- timer for oldest in-flight pkt
- timeout(n): retransmit packet n and all higher seq # pkts in window

**GBN**

ACK-only: always send ACK for correctly-received pkt with highest in-order seq #
- may generate duplicate ACKs
- need only remember expected seq num
- out-of-order pkt:
  - discard (don’t buffer): no receiver buffering!
  - re-ACK pkt with highest in-order seq #

**GBN in action**

![Diagram showing Go-Back-N protocol in action]

**Selective repeat**

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

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Selective repeat in action

Selective repeat: dilemma

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

Q: what relationship between seq # size and window size to avoid problem in (b)?
3.5: TCP: Overview

- **point-to-point:**
  - one sender, one receiver
- **reliable, in-order byte steam:**
  - no “message boundaries”
- **pipelined:**
  - TCP congestion and flow control set window size

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

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

- **flow controlled:**
  - sender will not overwhelm receiver

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

TCP seq. numbers, ACKs

simple telnet scenario
TCP round trip time, timeout

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

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

\[
\text{EstimatedRTT} = (1- \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}
\]
- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)

TCP round trip time, timeout

**timeout interval:** EstimatedRTT plus “safety margin”
- large variation in EstimatedRTT → larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

\[
\text{DevRTT} = (1- \beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT}-\text{EstimatedRTT}|
\]
- typically, \( \beta = 0.25 \)

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

TCP reliable data transfer

- TCP creates rdt service on top of IP’s unreliable service
  - pipelined segments
  - cumulative acks
  - single retransmission timer
  - retransmissions triggered by:
    - timeout events
    - duplicate acks

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

- **data rcvd from app:**
  - create segment with seq #
  - seq # is byte-stream number of first data byte in segment
  - start timer if not already running
    - think of timer as for oldest unacked segment
    - expiration interval: TimeOutInterval

- **timeout:**
  - retransmit segment that caused timeout
  - restart timer

- **ack rcvd:**
  - if ack acknowledges previously unacked segments
    - update what is known to be ACKed
    - start timer if there are still unacked segments

TCP sender (simplified)

```
// 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

// data received from application above
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: retransmission scenarios

- **lost ACK scenario**
- **premature timeout**
- **cumulative ACK**
TCP ACK generation [RFC 1122, RFC 2581]

**event at receiver**
- arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed
- arrival of in-order segment with expected seq #. One other segment has ACK pending
- arrival of out-of-order segment higher-than-expect seq #. Gap detected
- arrival of segment that partially or completely fills gap

**TCP receiver action**
- delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
- immediately send single cumulative ACK, ACKing both in-order segments
- immediately send duplicate ACK, indicating seq. # of next expected byte
- immediate send ACK, provided that segment starts at lower end of gap

TCP fast retransmit

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

- if sender receives 3 ACKs for same data (“triple duplicate ACKs”), resend unacked segment with smallest seq #
  - likely that unacked segment lost, so don’t wait for timeout

TCP fast retransmit after sender receipt of triple duplicate ACK

TCP flow control

flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast
TCP flow control

- receiver “advertises” free buffer space by including \texttt{rwnd} value in TCP header of receiver-to-sender segments
  - \texttt{RcvBuffer} size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust \texttt{RcvBuffer}
- sender limits amount of unacked (“in-flight”) data to receiver’s \texttt{rwnd} value
- guarantees receive buffer will not overflow

Connection Management

before exchanging data, sender/receiver “handshake”:
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

Agreeing to establish a connection

2-way handshake:

\textbf{Q:} will 2-way handshake always work in network?
- variable delays
- retransmitted messages (e.g. \texttt{req_conn(x)}) due to message loss
- message reordering
- can’t “see” other side
TCP 3-way handshake

client state

LISTEN
choose init seq num, x
send TCP SYN msg

SYN SENT
SYNbit=1, Seq=x
ACKbit=1, ACKnum=x+1
received ACK(y)
indicates client is live

ESTAB
received SYNACK(x)
indicates server is live;
send ACK for SYNACK;
this segment may contain
client-to-server data

server state

LISTEN
choose init seq num, y
send TCP SYN msg, asking SYN

SYN RCVD
SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1
create new socket for
communication back to client

ESTAB
SYNACK(seq=y, ACKnum=x+1)
create new socket for
communication back to client

TCP 3-way handshake: FSM

closed

LISTEN

SYN(seq=x)

SYN(x)

SYNRCVD

SYNACK(seq=y, ACKnum=x+1)

ESTAB

TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled

TCP: closing a connection

client state

ESTAB
can no longer send but can receive data

FIN_WAIT_1
FINbit=1, seq=x

FIN_WAIT_2
ACKbit=1; ACKnum=x+1
wait for server close

TIMED_WAIT
timed wait for 2*max segment lifetime

CLOSED
can no longer send data

server state

ESTAB

CLOSE_WAIT
can still send data

LAST_ACK
can no longer send data

CLOSED
### 3.6: Principles of congestion control

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

### Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- output link capacity: $R$
- no retransmission

**maximum per-connection throughput:** $R/2$

- large delays as arrival rate, $\lambda_{in}$, approaches capacity

### Causes/costs of congestion: scenario 2

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

**idealization: perfect knowledge**
- sender sends only when router buffers available

**free buffer space!**
Causes/costs of congestion: scenario 2

**Idealization: known loss**
- packets can be lost, dropped at router due to full buffers
  - sender only resends if packet known to be lost

![Diagram showing Idealization: known loss](image)

- \( \lambda_{in} \): original data
- \( \lambda_{in}^{'} \): original data, plus retransmitted data
- \( \lambda_{out} \)
- Host B
- no buffer space!

![Diagram showing Idealization: known loss](image)

- \( \lambda_{in}^{'} \): original data, plus retransmitted data
- \( \lambda_{out} \)
- Host B
- free buffer space!

Causes/costs of congestion: scenario 2

**Realistic: duplicates**
- packets can be lost, dropped at router due to full buffers
- sender times out prematurely, sending two copies, both of which are delivered

![Diagram showing Realistic: duplicates](image)

- \( \lambda_{in} \)
- \( \lambda_{in}^{'} \)
- \( \lambda_{out} \)
- Host B
- free buffer space!

![Diagram showing Realistic: duplicates](image)

- \( \lambda_{in}^{'} \)
- \( \lambda_{out} \)
- Host B
- free buffer space!

“costs” of congestion:
- more work (retrans) for given “goodput”
- unneeded retransmissions: link carries multiple copies of pkt
  - decreasing goodput
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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

A: as red $\lambda_{in}'$ increases, all arriving blue pkts at upper queue are dropped, blue throughput $\to 0$

another “cost” of congestion:
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!

3.7: TCP congestion control: additive increase multiplicative decrease

- approach: sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
  - additive increase: increase $\text{cwnd}$ by 1 MSS every RTT until loss detected
  - multiplicative decrease: cut $\text{cwnd}$ in half after loss

TCP Congestion Control: details

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

TCP sending rate:
- roughly: send $\text{cwnd}$ bytes, wait RTT for ACKS, then send more bytes
  
  rate $\approx \frac{\text{cwnd}}{\text{RTT}}$ bytes/sec

sender limits transmission:

\[
\text{LastByteSent} - \text{LastByteAcked} \leq \text{cwnd}
\]
TCP Slow Start

- when connection begins, increase rate exponentially until first loss event:
  - initially $cwnd = 1$ MSS
  - double $cwnd$ every RTT
  - done by incrementing $cwnd$ for every ACK received
- summary: initial rate is slow but ramps up exponentially fast

TCP: detecting, reacting to loss

- loss indicated by timeout:
  - $cwnd$ set to 1 MSS;
  - window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
  - dup ACKs indicate network capable of delivering some segments
  - $cwnd$ is cut in half window then grows linearly
- TCP Tahoe always sets $cwnd$ to 1 (timeout or 3 duplicate acks)

TCP: switching from slow start to CA

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

Implementation:
- variable $ssthresh$
- on loss event, $ssthresh$ is set to 1/2 of $cwnd$ just before loss event

Summary: TCP Congestion Control
TCP throughput

- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- $W$: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is $\frac{3}{4} W$
  - avg. throughput is $\frac{3}{4}W$ per RTT

$$\text{avg TCP throughput} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$

TCP Futures: TCP over “long, fat pipes”

- example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- requires $W = 83,333$ in-flight segments
- throughput in terms of segment loss probability, $L$ [Mathis 1997]:
  $$\text{TCP throughput} = 1.22 \cdot \frac{\text{MSS}}{\text{RTT}} \sqrt{L}$$
  $$\Rightarrow \text{to achieve 10 Gbps throughput, need a loss rate of } L = 2 \cdot 10^{-10} \text{ – a very small loss rate!}$$
- new versions of TCP for high-speed

TCP Fairness

*fairness goal:* if $K$ TCP sessions share same bottleneck link of bandwidth $R$, each should have average rate of $\frac{R}{K}$

Why is TCP fair?

two competing sessions:
- additive increase gives slope of 1, as throughput increases
- multiplicative decrease decreases throughput proportionally
**Fairness (more)**

*Fairness and UDP*
- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

*Fairness, parallel TCP connections*
- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate $R$ with 9 existing connections:
  - new app asks for 1 TCP, gets rate $R/10$
  - new app asks for 11 TCPs, gets $R/2$

**Explicit Congestion Notification (ECN)**

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

**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

**Fixed congestion window (1)**

*First case:*

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

$$\text{delay} = 2RTT + O/R$$
Fixed congestion window (2)

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

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

TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

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:

\[ P = \min\{Q,K-1\} \]

- where \( Q \) 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:
- \( 2\text{RTT} \) for connection
  estab and request
- \( \frac{O}{R} \) to transmit
  object
- time server idles due to slow start

Server idles:
\( P = \min(K-1,Q) \) times

Example:
- \( O/S = 15 \) segments
- \( K = 4 \) windows
- \( Q = 2 \)
- \( P = \min(K-1,Q) = 2 \)

Server idles \( P=2 \) times

TCP Delay Modeling (3)

\( \frac{S}{R} + \text{RTT} = \text{time from when server starts to send segment until server receives acknowledgement} \)

\[ 2^{k-1} \frac{S}{R} = \text{time to transmit the} \ k \text{th window} \]

\[ \left[ \frac{S}{R} + \text{RTT} - 2^{k-1} \frac{S}{R} \right] = \text{idle time after the} \ k \text{th window} \]

delay = \( \frac{O}{R} + 2\text{RTT} + \sum_{p=1}^{P} \text{idleTime}_p \)

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

\[ = \frac{O}{R} + 2\text{RTT} + P \left( \text{RTT} + \frac{S}{R} - (2^p - 1) \frac{S}{R} \right) \]
TCP Delay Modeling (4)

Recall K = number of windows that cover object

How do we calculate K?

\[ K = \min \{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O \} \]
\[ = \min \{k : 2^k - 1 \geq \frac{O}{S} \} \]
\[ = \min \{k : k \geq \log_2 \left( \frac{O}{S} + 1 \right) \} \]
\[ = \left\lceil \log_2 \left( \frac{O}{S} + 1 \right) \right\rceil \]

Calculation of Q, 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 msec, O = 5 Kbytes, M=10 and X=5

For low bandwidth, connection & response time dominated by transmission time.

Persistent connections only give minor improvement over parallel connections.

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

- Establishing a connection sounds easy, but tricky.
  - A sender sends a connection request and waits for a reply
  - It may not work since the network can lose, store, and duplicate packets.
- Three-way handshake to solve the problem
- Releasing a connection
Chapter 3: summary

- principles behind transport layer services:
  - multiplexing, demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation, implementation in the Internet
  - UDP
  - TCP

next:
- leaving the network "edge" (application, transport layers)
- into the network "core"
- two network layer chapters:
  - data plane
  - control plane