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 1996-2007
J.F Kurose and K.W. Ross, All Rights Reserved
Chapter 3: Transport Layer

**Our goals:**
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control
Chapter 3 outline

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

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

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

**Household analogy:**

12 kids sending letters to 12 kids
- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service
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
Multiplexing/demultiplexing

Demultiplexing at rcv host:
delivering received segments to correct socket

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

= socket    = process
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
- host uses IP addresses & port numbers to direct segment to appropriate socket

TCP/UDP segment format

- 32 bits
- | source port # | dest port # |
- other header fields
- application data (message)
Connectionless demultiplexing

- Create sockets with port numbers:
  
  ```java
  DatagramSocket mySocket1 = new DatagramSocket(12534);
  DatagramSocket mySocket2 = new DatagramSocket(12535);
  ```

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

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

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

DatagramSocket serverSocket = new DatagramSocket(6428);

SP provides “return address”
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number

- recv host uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple

- Web servers have different sockets for each connecting client:
  - non-persistent HTTP will have different socket for each request
Connection-oriented demux (cont)
Connection-oriented demux: Threaded Web Server

Client IP: A
- SP: 9157
- DP: 80
- S-IP: A
- D-IP: C

Server IP: C
- SP: 5775
- DP: 80
- S-IP: B
- D-IP: C

Client IP: B
- SP: 9157
- DP: 80
- S-IP: B
- D-IP: C
Chapter 3 outline

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

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

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

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive

- other UDP uses
  - DNS
  - SNMP

- reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

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

Application data (message)

UDP segment format
UDP checksum

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

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

**Receiver:**
- compute checksum of 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 0 1 1 0 0 1 1 0
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
```

```plaintext
wraparound  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 1 0 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!

(a) provided service

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

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

- 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.

**Network Stack**

- **Reliable data transfer protocol (sending side)**
- **Reliable data transfer protocol (receiving side)**

**Packet Flow**

- **Packet**: transferred between layers.
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
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
Wait for call from above
rdt_send(data)
  packet = make_pkt(data)
  udt_send(packet)

# receiver
Wait for call from below
rdt_rcv(packet)
  extract (packet, data)
  deliver_data(data)
```
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
rdt2.0: FSM specification

sender

Wait for call from above

\[
\text{rdt\_send(data)}
\]
\[
\text{sndpkt} = \text{make\_pkt(data, checksum)}
\]
\[
\text{udt\_send(sndpkt)}
\]

Wait for ACK or NAK

\[
\text{rdt\_rcv(rcvpkt) && isNAK(rcvpkt)}
\]
\[
\text{udt\_send(sndpkt)}
\]

receiver

Wait for call from below

\[
\text{rdt\_rcv(rcvpkt) && notcorrupt(rcvpkt)}
\]
\[
\text{extract(rcvpkt, data)}
\]
\[
\text{deliver\_data(data)}
\]
\[
\text{udt\_send(ACK)}
\]
**rdt2.0: operation with no errors**

- **Wait for call from above**
  - `rtt_send(data)`
  - `snkpkt = make_pkt(data, checksum)`
  - `udt_send(sndpkt)`

- **Wait for ACK or NAK**
  - `rdt_rcv(rcvpkt) && isNAK(rcvpkt)`
  - `udt_send(sndpkt)`

- **Wait for call from below**
  - `rdt_rcv(rcvpkt) && isACK(rcvpkt)`

- **Not corrupt**
  - `notcorrupt(rcvpkt)`
  - `extract(rcvpkt, data)`
  - `deliver_data(data)`
  - `udt_send(ACK)`

- **Corrupt**
  - `corrupt(rcvpkt)`

- **ACK or NAK**
  - `udt_send(NAK)`

- **Operation continues**
  - `Lambda`
rdt2.0: error scenario

```plaintext
rdt_send(data)

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

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

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

rdt_rcv(rcvpkt) &&
isACK(rcvpkt)

Wait for call from above

Wait for ACK or NAK

Wait for call from below

extract(rcvpkt, data)
deliver_data(data)
udt_send(ACK)
```

Transport Layer 3-30
rt2.0 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 and wait
Sender sends one packet, then waits for receiver response
**rdt2.1: sender, handles garbled ACK/NAKs**

- **rdt_send(data)**
  
  `sndpkt = make_pkt(0, data, checksum)`
  `udt_send(sndpkt)`

- **Wait for call 0 from above**
  
  `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`

- **Wait for ACK or NAK 0**
  
  `udt_send(sndpkt)`

- **Wait for call 1 from above**
  
  `rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)`

- **Wait for ACK or NAK 1**
  
  `rdt_send(data)`
  `sndpkt = make_pkt(1, data, checksum)`
  `udt_send(sndpkt)`
rdt2.1: receiver, handles garbled ACK/NAKs

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

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) &&
| has_seq1(rcvpkt)
| sndpkt = make_pkt(NAK, checksum)
| udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(ACK, checksum)
udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
| sndpkt = make_pkt(NAK, checksum)
| udt_send(sndpkt)
```

```
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) &&
| has_seq0(rcvpkt)
| sndpkt = make_pkt(ACK, checksum)
| udt_send(sndpkt)
```
rdr2.1: 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**

**sender FSM fragment**
- `rdt_send(data)`
- `sndpkt = make_pkt(0, data, checksum)`
- `udt_send(sndpkt)`

**receiver FSM fragment**
- `rdt_rcv(rcvpkt) & & (corrupt(rcvpkt) || has_seq1(rcvpkt))`
- `udt_send(sndpkt)`
- `extract(rcvpkt, data)`
- `deliver_data(data)`
- `sndpkt = make_pkt(ACK1, checksum)`
- `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) & & notcorrupt(rcvpkt) & & isACK(rcvpkt, 0)`
- `udt_send(sndpkt)`

- `rdt_rcv(rcvpkt) & & notcorrupt(rcvpkt) & & isACK(rcvpkt, 1)`
- `udt_send(sndpkt)`

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

- `corrupt(rcvpkt)`
- `isACK(rcvpkt, 1)`
- `notcorrupt(rcvpkt)`
- `isACK(rcvpkt, 0)`
- `has_seq1(rcvpkt)`
rdt3.0: channels with errors and loss

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

Approach: sender waits “reasonable” amount of time for ACK
  - retransmits if no ACK received in this time
  - if pkt (or ACK) just delayed (not lost):
    - retransmission will be duplicate, but use of seq. #’s already handles this
    - receiver must specify seq # of pkt being ACKed
  - requires countdown timer
**rdt3.0 sender**

```
rdt_send(data)

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

Wait for call 0 from above

rdt_rcv(rcvpkt)

∀

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

stop_timer

Wait for call 1 from above

rdt_send(data)

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

Wait for ACK0

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

∀

Wait for call 0 from above

rdt_send(data)

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

Wait for ACK1

rdt_rcv(rcvpkt)

∀

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

stop_timer

timeout
udt_send(sndpkt)
start_timer

Wait for call 1 from above

rdt_rcv(rcvpkt)

∀
```
**rdt3.0 in action**

(a) operation with no loss

(b) lost packet
rdt3.0 in action

![Diagram of rdt3.0 in action](image)

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

- rdt3.0 works, but performance stinks
- 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 - fraction of time sender busy sending

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

- 1KB pkt every 30 msec -> 33kB/sec thruput over 1 Gbps link
- network protocol limits use of physical resources!
**rdt3.0: stop-and-wait operation**

First packet bit transmitted, \( t = 0 \)

Last packet bit transmitted, \( t = \frac{L}{R} \)

First packet bit arrives

Last packet bit arrives, send ACK

ACK arrives, send next packet, \( t = RTT + \frac{L}{R} \)

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

Pipelining: sender allows multiple, “in-flight”, yet-to-be-acknowledged pkts
  - range of sequence numbers must be increased
  - buffering at sender and/or receiver

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

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

Increase utilization by a factor of 3!
Pipelining 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 unacked packets

Selective Repeat: big pic
- Sender can have up to $N$ unacked packets in pipeline
- Rcvr acks individual packets
- Sender maintains timer for each unacked packet
  - When timer expires, retransmit only unack packet
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 each in-flight pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window
GBN: sender extended FSM

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

\[
\text{base} = 1 \\
\text{nextseqnum} = 1
\]

\[
\text{rdt\_rcv}(\text{rcvpkt}) \ \&\& \ \text{corrupt}(\text{rcvpkt})
\]

\[
\text{timeout} \\
\text{start\_timer} \\
\text{udt\_send}(\text{sndpkt}[\text{base}]) \\
\text{udt\_send}(\text{sndpkt}[\text{base+1}]) \\
\ldots \\
\text{udt\_send}(\text{sndpkt}[\text{nextseqnum-1}])
\]

\[
\text{rdt\_rcv}(\text{rcvpkt}) \ \&\& \ \text{notcorrupt}(\text{rcvpkt})
\]

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

- **default**: `udt_send(sndpkt)`
- **Wait**:
  - `rdt_rcv(rcvpkt) && notcurrupt(rcvpkt) && hasseqnum(rcvpkt,expectedseqnum)`
  - `extract(rcvpkt,data)`
  - `deliver_data(data)`
  - `sndpkt = make_pkt(expectedseqnum,ACK,chksum)`
  - `udt_send(sndpkt)`
  - `expectedseqnum++`

**ACK-only**: always send ACK for correctly-received pkt with highest *in-order* seq #
- may generate duplicate ACKs
- need only remember `expectedseqnum`

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

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

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

- receiver *individually* acknowledges all correctly received pkts
  - buffers pkts, as needed, for eventual in-order delivery to upper layer

- sender only resends pkts for which ACK not received
  - sender timer for each unACKed pkt

- sender window
  - N consecutive seq #’s
  - again limits seq #’s of sent, unACKed pkts
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers
Selective repeat

Sender

Data from above:
- If next available seq # in window, send pkt

Timeout(n):
- Resend pkt n, restart timer

ACK(n) in [sendbase, sendbase+N]:
- Mark pkt n as received
- If n smallest unACKed pkt, advance window base to next unACKed seq #

Receiver

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

Pkt n in [rcvbase-N, rcvbase-1]:
- ACK(n)

Otherwise:
- Ignore
Selective repeat in action

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

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

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

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

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

pkt2 rcvd, pkt2, pkt3, pkt4, pkt5 delivered, ACK2 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
Selective repeat: dilemma

Example:
- seq #’s: 0, 1, 2, 3
- window size=3

- receiver sees no difference in two scenarios!
- incorrectly passes duplicate data as new in (a)

Q: what relationship between seq # size and window size?
Chapter 3 outline

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

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

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

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

- **send & receive buffers**

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

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

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

RFCs: 793, 1122, 1323, 2018, 2581
### TCP segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number for the segment.</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number for the segment.</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number identifying the order of the data within the stream.</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number indicating the sequence number of the last byte acknowledged by the receiver.</td>
</tr>
<tr>
<td>head len</td>
<td>Length of the header in bytes.</td>
</tr>
<tr>
<td>not used</td>
<td></td>
</tr>
<tr>
<td>UAP</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>Receive window</td>
<td>Number of bytes the receiver is willing to accept.</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum of the header and data.</td>
</tr>
<tr>
<td>Urg data pnter</td>
<td>Urgent data pointer indicating the position of the urgent data within the segment.</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td>Options field containing additional flags and control information.</td>
</tr>
<tr>
<td>application data</td>
<td>Application data contained in the segment.</td>
</tr>
<tr>
<td>(variable length)</td>
<td></td>
</tr>
</tbody>
</table>
TCP seq. #’s and ACKs

**Seq. #’s:**
- byte stream “number” of first byte in segment’s data

**ACKs:**
- seq # of next byte expected from other side
- cumulative ACK

**Q:** how receiver handles out-of-order segments
- **A:** TCP spec doesn’t say, - up to implementor

---

**Simple telnet scenario**

**Host A**
- User types ‘C’
- Seq=42, ACK=79, data = ‘C’
- host ACKs receipt of ‘C’, echoes back ‘C’

**Host B**
- Seq=79, ACK=43, data = ‘C’
- Seq=43, ACK=80

---

Transport Layer  3-58
**TCP Round Trip Time and Timeout**

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

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

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

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

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

<table>
<thead>
<tr>
<th>time (seconds)</th>
<th>RTT (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SampleRTT</td>
</tr>
<tr>
<td>8</td>
<td>Estimated RTT</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>57</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td>71</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td></td>
</tr>
<tr>
<td>85</td>
<td></td>
</tr>
<tr>
<td>92</td>
<td></td>
</tr>
<tr>
<td>99</td>
<td></td>
</tr>
<tr>
<td>106</td>
<td></td>
</tr>
</tbody>
</table>

Graph showing fluctuations in RTT over time.
TCP Round Trip Time and Timeout

Setting the timeout

- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT → larger safety margin
- first estimate of how much SampleRTT deviates from EstimatedRTT:

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

(typically, \(\beta = 0.25\))

Then set timeout interval:

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
Chapter 3 outline

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

- 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)
- expiration interval: TimeOutInterval

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

**Ack rcvd:**
- If acknowledges previously unacked segments
  - update what is known to be acked
  - start timer if there are outstanding segments
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: data received from application above
  create TCP segment with sequence number NextSeqNum
  if (timer currently not running)
    start timer
  pass segment to IP
  NextSeqNum = NextSeqNum + length(data)

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

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

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

Host A

SendBase = 100

SendBase = 120

Seq=92, 8 bytes data

ACK=100

ACK=120

timeout

X

loss

Host B

Seq=100, 20 bytes data

ACK=100

ACK=120

Seq=92, 8 bytes data

lost ACK scenario

timeout

premature timeout

Host A

SendBase = 100

SendBase = 120

Seq=92, 8 bytes data

ACK=100

ACK=120

timeout

X

loss

Host B
TCP retransmission scenarios (more)

Cumulative ACK scenario

Host A

Host B

SendBase = 120

time

timeout

Seq=92, 8 bytes data

Seq=100, 20 bytes data

ACK=100

ACK=120

X

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

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

- 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:
  - \textit{fast retransmit:} resend segment before timer expires
Figure 3.37 Resending a segment after triple duplicate ACK.
Fast retransmit algorithm:

**event:** ACK received, with ACK field value of \( y \)

if \( y > \text{SendBase} \) {
    SendBase = \( y \)
    if (there are currently not-yet-acknowledged segments)
        start timer
else {
    increment count of dup ACKs received for \( y \)
    if (count of dup ACKs received for \( y = 3 \) ) {
        resend segment with sequence number \( y \)
    }

A duplicate ACK for already ACKed segment

Fast retransmit
Chapter 3 outline

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

- receive side of TCP connection has a receive buffer:

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

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

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

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
  - guarantees receive buffer doesn’t overflow

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

- spare room in buffer
  - = RcvWindow
  - = RcvBuffer - [LastByteRcvd - LastByteRead]
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
  
  ```java
  Socket clientSocket = new Socket("hostname","port number");
  ```

- **server**: contacted by client
  
  ```java
  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:
```java
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.
TCP Connection Management (cont)

TCP client lifecycle

TCP server lifecycle
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!
Causes/costs of congestion: scenario 1

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

- large delays when congested
- maximum achievable throughput

![Diagram](image-url)
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet
Causes/costs of congestion: scenario 2

- always: $\lambda_{in} = \lambda_{out}$ (goodput)
- “perfect” retransmission only when loss: $\lambda'_{in} < \lambda_{out}$
- 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
Causes/costs of congestion: scenario 3

- four senders
- multihop paths
- timeout/retransmit

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

Another “cost” of congestion:
- when packet dropped, any “upstream transmission capacity used for that packet was wasted!
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
Case study: ATM ABR congestion control

- two-byte ER (explicit rate) field in RM cell
  - congested switch may lower ER value in cell
  - sender's send rate thus maximum supportable rate on path
- EFCI bit in data cells: set to 1 in congested switch
  - if data cell preceding RM cell has EFCI set, sender sets CI bit in returned RM cell
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: additive increase, multiplicative decrease**

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

Saw tooth behavior: probing for bandwidth
TCP Congestion Control: details

- **sender limits transmission:**
  \[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]

- **Roughly,**
  \[ \text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec} \]

- **CongWin** is dynamic, function of perceived network congestion

**How does sender perceive congestion?**

- **loss event** = timeout or 3 duplicate acks
- **TCP sender reduces rate** (CongWin) after loss event

**three mechanisms:**
- AIMD
- slow start
- conservative after timeout events
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
- When connection begins, increase rate exponentially fast until first loss event
TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
  - double CongWin every RTT
  - done by incrementing CongWin for every ACK received

- **Summary**: initial rate is slow but ramps up exponentially fast
Refinement: inferring loss

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

Philosophy:
- 3 dup ACKs indicates network capable of delivering some segments
- timeout indicates a “more alarming” congestion scenario
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 $\text{CongWin}$ is below $\text{Threshold}$, sender is in slow-start phase, window grows exponentially.

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

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

- When timeout occurs, $\text{Threshold}$ set to $\text{CongWin}/2$ and $\text{CongWin}$ is set to 1 MSS.
## TCP sender congestion control

<table>
<thead>
<tr>
<th>State</th>
<th>Event</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slow Start (SS)</td>
<td>ACK receipt for previously unacked data</td>
<td>CongWin = CongWin + MSS, If (CongWin &gt; Threshold) set state to “Congestion Avoidance”</td>
<td>Resulting in a doubling of CongWin every RTT</td>
</tr>
<tr>
<td>Congestion Avoidance (CA)</td>
<td>ACK receipt for previously unacked data</td>
<td>CongWin = CongWin + MSS * (MSS/CongWin)</td>
<td>Additive increase, resulting in increase of CongWin by 1 MSS every RTT</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Loss event detected by triple duplicate ACK</td>
<td>Threshold = CongWin/2, CongWin = Threshold, Set state to “Congestion Avoidance”</td>
<td>Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Timeout</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>SS or CA</td>
<td>Duplicate ACK</td>
<td>Increment duplicate ACK count for segment being acked</td>
<td>CongWin and Threshold not changed</td>
</tr>
</tbody>
</table>
TCP throughput

- What’s the average throughout 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 \( \frac{W}{RTT} \)
- Just after loss, window drops to \( \frac{W}{2} \), throughput to \( \frac{W}{2RTT} \).
- Average throughout: \( 0.75 \frac{W}{RTT} \)
TCP Futures: TCP over “long, fat pipes”

- Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput
- Requires window size $W = 83,333$ in-flight segments
- Throughput in terms of loss rate:

$$\frac{1.22 \cdot MSS}{RTT \sqrt{L}}$$

- $L = 2 \cdot 10^{-10}$ Wow
- 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 $R/K$
Why is TCP fair?

Two competing sessions:
- Additive increase gives slope of 1, as throughout 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:
  - 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
Fixed congestion window (1)

First case:
WS/R > RTT + S/R: ACK for first segment in window returns before window’s worth of data sent

delay = 2RTT + O/R
Fixed congestion window (2)

**Second case:**
- $WS/R < RTT + S/R$: wait for ACK after sending window’s worth of data sent

\[
delay = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]
\]
TCP Delay Modeling: Slow Start (1)

Now suppose window grows according to slow start

Will show that the delay for one object is:

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

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 RTT for connection estab and request
- 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)**

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

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

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

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

\[
= \frac{O}{R} + 2RTT + \sum_{k=1}^{P} \left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right]
\]

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

- First window = \(S/R\)
- Second window = \(2S/R\)
- Third window = \(4S/R\)
- Fourth window = \(8S/R\)

**Diagram:**
- Initiate TCP connection
- Request object
- First window
- Second window
- Third window
- Fourth window
- Complete transmission
- Time at server
- Time at client
- Object delivered
TCP Delay Modeling (4)

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

How do we calculate $K$?

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

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 = \((M+1)\frac{O}{R} + (M+1)2RTT + \text{sum of idle times}\)

- Persistent HTTP:
  - 2 RTT to request and receive base HTML file
  - 1 RTT to request and receive \(M\) images
  - Response time = \((M+1)\frac{O}{R} + 3RTT + \text{sum of idle times}\)

- 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 = \((M+1)\frac{O}{R} + (M/X + 1)2RTT + \text{sum of idle times}\)
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.
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”