Chapter 3: Transport Layer

Chapter goals:
- understand principles behind transport layer services:
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- instantiation and implementation in the Internet

Chapter Overview:
- transport layer services
- multiplexing/demultiplexing
- connectionless transport: UDP
- principles of reliable data transfer
- connection-oriented transport: TCP
  - reliable transfer
  - flow control
  - connection management
- principles of congestion control
- 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
  - relies on, enhances, network layer services

- **transport layer**: logical communication between processes

**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
Transport-layer protocols

Internet transport services:
- reliable, in-order unicast delivery (TCP)
  - congestion
  - flow control
  - connection setup
- unreliable ("best-effort"), unordered unicast or multicast delivery: UDP
- services not available:
  - real-time
  - bandwidth guarantees
  - reliable multicast
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

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

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

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

host 1

host 2

host 3

Comp 361, Spring 2005
Multiplexing/demultiplexing

**segment** - unit of data exchanged between transport layer entities

- aka TPDU: transport protocol data unit

Demultiplexing: delivering received segments to correct app layer processes

- application-layer data
- segment header
- segment

P1
- application
- transport
- network
- M

P3
- application
- transport
- network
- M

P4
- application
- transport
- network
- M

P2
- application
- transport
- network
- M
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

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(99111);
  DatagramSocket mySocket2 = new DatagramSocket(99222);
  ```

- 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: 9157
- DP: 80

Client IP: B
- SP: 5775
- 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 (8 Bytes)
- 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 (why?):
  - DNS: small delay
  - SNMP: stressful cond.
- reliable transfer over UDP:
  add reliability at application layer
  - application-specific error recover!

UDP segment format:

<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)

Length, in bytes of UDP segment, including header

Comp 361, Spring 2005
**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..
- Receiver may choose to discard segment or send a warning to app in case error
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

**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

- `udt_send()`: called by `rdt`, to transfer packet over unreliable channel to receiver
Reliable data transfer: getting started

We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

*state:* when in this "state" next state uniquely determined by next event
Incremental Improvements

- rdt1.0: assumes every packet sent arrives, and no errors introduced in transmission

- rdt2.0: assumes every packet sent arrives, but some errors (bit flips) can occur within a packet. Introduces concept of ACK and NAK

- rdt2.1: deals with corrupted ACKS/NAKS

- rdt2.2: like rdt2.1 but does not need NAKs

- Rdt3.0: Allows packets to be lost
Rdt1.0: **reliable transfer over a reliable channel**

- **underlying channel perfectly reliable**
  - no bit errors
  - no loss of packets

- **separate FSMs for sender, receiver:**
  - sender sends data into underlying channel
  - receiver read data from underlying channel

sender

receiver
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - 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
  - human scenarios using ACKs, NAKs?
- new mechanisms in rdt2.0 (beyond rdt1.0):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender
**rdt2.0: FSM specification**

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

**sender**

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

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

```
Lambda
```

**receiver**

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

```
Wait for call from below
```

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)
    extract(rcvpkt, data)
    deliver_data(data)
    udt_send(ACK)
```
**rtdt2.0: operation with no errors**

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

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

- `wait for call from above`

- `wait for ACK or NAK`

- `wait for call from below`

- `corrupt(rcvpkt)`
  - `udt_send(NAK)`
  - `wait for call from below`

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

- `Λ`
rdt2.0: error scenario

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

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

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

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

- `Lambda`
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. But receiver waiting!

What to do?
- sender ACKs/NAKs receiver’s ACK/NAK? What if sender ACK/NAK corrupted?
- retransmit, but this might cause retransmission of correctly received pkt!
- Receiver won’t know about duplication!

Handling duplicates:
- sender adds sequence number (0/1) to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn’t deliver up) duplicate pkt
- Duplicate packet is one with same sequence # as previous packet

stop and wait
Sender sends one packet, then waits for receiver response
- **Sender:** whenever sender receives control message it sends a packet to receiver.
  - A valid **ACK:** Sends next packet (if exists) with new sequence #
  - A NAK or corrupt response: resends old packet

- **Receiver:** sends **ACK/NAK** to sender
  - If received packet is corrupt: send NAK
  - If received packet is valid and has different sequence # as prev packet: send ACK and deliver new data up.
  - If received packet is valid and has same sequence # as prev packet, i.e., is a retransmission of duplicate: send ACK

- **Note:** **ACK/NAK** do not contain sequence #.
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)
Λ
```

Wait for ACK or NAK 0

```
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
```

Wait for call 1 from above

```
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
```

Wait for ACK or NAK 1

```
rdt_send(data)
udt_send(sndpkt)
```

Wait for call 0 from above

```
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isNAK(rcvpkt))
```

udt_send(sndpkt)
**rdt2.1: receiver, handles garbled ACK/NAKs**

- \( rdt\_rcv(rcvpkt) \&\& \text{notcorrupt(rcvpkt)} \)
  - \( \text{has_seq0(rcvpkt)} \)
  - \( \text{extract(rcvpkt, data)} \)
  - \( \text{deliver_data(data)} \)
  - \( \text{sndpkt = make_PKT(ACK, checksum)} \)
  - \( \text{udt\_send(sndpkt)} \)

- \( rdt\_rcv(rcvpkt) \&\& \text{not corrupt(rcvpkt)} \&\& \text{has_seq1(rcvpkt)} \)
  - \( \text{sndpkt = make_PKT(NAK, checksum)} \)
  - \( \text{udt\_send(sndpkt)} \)

- \( rdt\_rcv(rcvpkt) \&\& \text{corrupt(rcvpkt)} \)
  - \( \text{sndpkt = make_PKT(ACK, checksum)} \)
  - \( \text{udt\_send(sndpkt)} \)

- \( rdt\_rcv(rcvpkt) \&\& \text{not corrupt(rcvpkt)} \&\& \text{has_seq1(rcvpkt)} \)
  - \( \text{extract(rcvpkt, data)} \)
  - \( \text{deliver_data(data)} \)
  - \( \text{sndpkt = make_PKT(ACK, checksum)} \)
  - \( \text{udt\_send(sndpkt)} \)
rdt2.1: discussion

**Sender:**
- seq # added to pkt
- two seq. #’s (0,1) will suffice. Why?
- must check if received ACK/NAK corrupted
- twice as many states
  - state must “remember” whether “current” pkt has 0 or 1 seq. #

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

- same functionality as rdt2.1, using ACKs only
- instead of NAK, receiver sends ACK for last pkt received OK
  - receiver must *explicitly* include seq # of pkt being ACKed
    (in 2.1 seq #s included in data packets but not in ACKs/NAKs)
- duplicate ACK at sender results in same action as NAK: *retransmit current pkt*
rdt2.2: sender, receiver fragments

sender FSM fragment:
- \( \text{rdt\_send}(\text{data}) \)
- \( \text{sndpkt} = \text{make\_pkt}(0, \text{data}, \text{checksum}) \)
- \( \text{udt\_send}(<\text{sndpkt}>) \)
- Wait for call 0 from above
- Wait for ACK 0
- \( \text{rdt\_rcv}(\text{rcvpkt}) \land \text{isACK}(<\text{rcvpkt},1>) \)
- \( \text{udt\_send}(<\text{sndpkt}>) \)

receiver FSM fragment:
- \( \text{rdt\_rcv}(\text{rcvpkt}) \land \text{notcorrupt}(\text{rcvpkt}) \land \text{isACK}(<\text{rcvpkt},0>) \)
- \( \text{udt\_send}(<\text{sndpkt}>) \)
- Wait for 0 from below
- Wait for call 0 from above
- \( \text{extract}(\text{rcvpkt, data}) \)
- \( \text{deliver\_data}(\text{data}) \)
- \( \text{sndpkt} = \text{make\_pkt}(\text{ACK,1, checksum}) \)
- \( \text{udt\_send}(<\text{sndpkt}>) \)

Comp 361, Spring 2005
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
  (Retransmissions only triggered by timeouts)
- 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

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

Wait for call 0 from above

```
rtt_rcv(rcvpkt) &&
_timeout
udt_send(sndpkt)
start_timer
```

Wait for ACK0

```
rtt_rcv(rcvpkt) &&
( corrupt(rcvpkt) ||
isACK(rcvpkt,1) )
```

```
Lambda
```

Wait for call 1 from above

```
rtt_rcv(rcvpkt) &&
&notcorrupt(rcvpkt)
&isACK(rcvpkt,1)
stop_timer
```

```
Lambda
```

Wait for call 0 from above

```
rtt_rcv(rcvpkt) &&
&notcorrupt(rcvpkt)
&isACK(rcvpkt,0)
```

```
stop_timer
```

```
Lambda
```

```
Lambda
```

```
Lambda
```

```
Lambda
```

Wait for ACK1

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

```
Lambda
```

Wait for call 1 from above

```
rtt_rcv(rcvpkt) &&
&notcorrupt(rcvpkt)
&isACK(rcvpkt,0)
```

```
stop_timer
```

```
Lambda
```

```
Lambda
```

```
Lambda
```

```
Lambda
```

Comp 361, Spring 2005
**rdt3.0 in action**

(a) operation with no loss

(b) lost packet

Comp 361, Spring 2005
rdt3.0 in action

(c) lost ACK

(d) premature timeout
Performance of rdt3.0

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

\[ T_{\text{transmit}} = \frac{L}{R} = \frac{8\text{kb/pkt}}{10^{9}\text{ b/sec}} = 8 \text{ microsec} \]

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

- \( U_{\text{sender}} \): utilization - fraction of time sender busy sending
- 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**

- **sender**
  - first packet bit transmitted, \( t = 0 \)
  - last packet bit transmitted, \( t = \frac{L}{R} \)

- **receiver**
  - first packet bit arrives
  - last packet bit arrives, send ACK

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

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

(a) a stop-and-wait protocol in operation
(b) a pipelined protocol in operation
Pipelined protocols

- Advantage: much better bandwidth utilization than stop-and-wait

- Disadvantage: More complicated to deal with reliability issues, e.g., corrupted, lost, out of order data.
  - Two generic approaches to solving this
    - *go-Back-N protocols*
    - *selective repeat protocols*

- Note: *TCP is not exactly either*
Pipelining: increased utilization

- First packet bit transmitted, $t = 0$
- Last bit transmitted, $t = L / R$
- First packet bit arrives
- Last packet bit arrives, send ACK
- Last bit of 2nd packet arrives, send ACK
- Last bit of 3rd packet arrives, send ACK

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

Increase utilization by a factor of 3!
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)
- Only one timer: for oldest unacknowledged pkt
- timeout(n): retransmit pkt n and all higher seq # pkts in window
- Called a sliding-window protocol
GBN: Sender

- rdt_Send() called: checks to see if window is full.
  - No: send out packet
  - Yes: return data to application level

- Receipt of ACK(n): cumulative acknowledgement that all packets up to and including n have been received. Updates window accordingly and restarts timer.

- Timeout: resends ALL packets that have been sent but not yet acknowledged.
  - This is only event that triggers resend.
**GBN: sender extended FSM**

```
rdt_send(data)
if (nextseqnum < base+N) {
    sndpkt[nextseqnum] = make_pkt(nextseqnum, data, chksum)
    udt_send(sndpkt[nextseqnum])
    if (base == nextseqnum)
        start_timer
        nextseqnum++
} else
    refuse_data(data)

base = getacknum(rcvpkt)+1
If (base == nextseqnum)
    stop_timer
else
    start_timer
```

```
Lambda
base=1
nextseqnum=1
```
GBN: receiver extended FSM

- **default**: udt_send(sndpkt)
- **expectedseqnum=1**: sndpkt = make_pkt(0,ACK,chksum)
- **rdt_rcv(rcvpkt)**
  - && notcorrupt(rcvpkt)
  - && hasseqnum(rcvpkt,expectedseqnum)
- **extract(rcvpkt,data)**
- **deliver_data(data)**
- sndpkt = make_pkt(expectedseqnum,ACK,chksum)
- udt_send(sndpkt)
- expectedseqnum++

- **If expected packet received**:
  - Send ACK and deliver packet upstairs

- **If out-of-order packet received**:
  - discard (don't buffer) -> no receiver buffering!
  - Re-ACK pkt with highest in-order seq #
  - may generate duplicate ACKs
More on receiver

- The receiver always sends ACK for last correctly received packet with highest \emph{in-order} seq #
- Receiver only sends ACKS (no NAKs)
- Can generate duplicate ACKs
- need only remember \texttt{expectedseqnum}
GBN in action

sender

send pkt0
send pkt1
send pkt2
send pkt3 (wait)

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

pkt2 timeout

Comp 361, Spring 2005
GBN is easy to code but might have performance problems.

In particular, if many packets are in pipeline at one time (bandwidth-delay product large) then one error can force retransmission of huge amounts of data!

Selective Repeat protocol allows receiver to buffer data and only forces retransmission of required packets.
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
  - Compare to GBN which only had timer for base packet
- sender window
  - N consecutive seq #'s
  - again limits seq #'s of sent, unACKed pkts
  - Important: Window size < seq # range
Selective repeat: sender, receiver windows

(a) sender view of sequence numbers

(b) receiver view of sequence numbers

Comp 361, Spring 2005
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) (note this is a reACK)

- 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

pkt3 rcvd, buffered, ACK3 sent
0 1 2 3 4 5 6 7 8 9

pkt4 rcvd, buffered, ACK4 sent
0 1 2 3 4 5 6 7 8 9

pkt5 rcvd, buffered, ACK5 sent
0 1 2 3 4 5 6 7 8 9

pkt2 rcvd, pkt2,pkt3,pkt4,pkt5 delivered, ACK2 sent
0 1 2 3 4 5 6 7 8 9
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 is 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
More TCP Details

- Maximum Segment Size (MSS)
  - Depends upon implementation (can often be set)
  - The Max amount of application-layer data in segment
- Application Data + TCP Header = TCP Segment

- Three way Handshake
  - Client sends special TCP segment to server requesting connection. No payload (Application data) in this segment.
  - Server responds with second special TCP segment (again no payload)
  - Client responds with third special segment
    - This can contain payload
Even More TCP Details

- A TCP connection between client and server creates, in both client and server
  - (i) buffers
  - (ii) variables and
  - (iii) a socket connection to process.

- TCP only exists in the two end machines.
  No buffers and variables allocated to the connection in any of the network elements between the host and server.
TCP segment structure

- **source port #**
- **dest port #**
- **sequence number**
- **acknowledgement number**
- **Receive window**
- **checksum**
- **Urg data pnter**
- **Options (variable length)**
- **application data** (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)

- **32 bits**
- **counting by bytes of data (not segments!)**
- **# bytes rcvr willing to accept**
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 implementer

User types 'C' host ACKs receipt of 'C', echoes back 'C'
Seq=42, ACK=79, data = 'C' host ACKs receipt of echoed 'C'
Seq=79, ACK=43, data = 'C'
Seq=43, ACK=80

simple telnet scenario
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

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

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

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

SampleRTT  Estimated RTT
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
TCP sender (simplified)

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 */

Comment:
• SendBase-1: last cumulatively ack'ed byte
Example:
• SendBase-1 = 71;
y= 73, so the rcvr wants 73+; y > SendBase, so that new data is acked
TCP: retransmission scenarios

Host A
Seq=92, 8 bytes data
ACK=100
SendBase = 100
time
lost ACK scenario

Host B
Seq=92 timeout
Acc=100
time
lost ACK scenario

Host A
Seq=92 timeout
Acc=100
SendBase = 100

Host B
Seq=92 timeout
Acc=120
SendBase = 120

Host A
Seq=100, 20 bytes data
ACK=100
SendBase = 100

Host B
Seq=92, 8 bytes data
ACK=120
SendBase = 120

Comp 361, Spring 2005
TCP retransmission scenarios (more)

Cumulative ACK scenario

Host A
Seq=92, 8 bytes data
ACK=100

Host B
Seq=100, 20 bytes data
ACK=120

SendBase = 120

X loss

time

Comp 361, Spring 2005
## 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>
More on Sender Policies

- Doubling the Timeout Interval
  - Used by most TCP implementations
  - If timeout occurs then, after retransmission, Timeout Interval is doubled
  - Intervals grow exponentially with each consecutive timeout
  - When Timer restarted because of (i) new data from above or (ii) ACK received, then Timeout Interval is reset as described previously using Estimated RTT and DevRTT.
  - Limited form of Congestion Control
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
Fast retransmit algorithm:

**event:** ACK received, with ACK field value of y

  if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acknowledged segments)
      start timer
  }

else {
  increment count of dup ACKs received for y
  if (count of dup ACKs received for y = 3) {
    resend segment with sequence number y
  }
}

a duplicate ACK for already ACKed segment  fast retransmit
TCP: GBN or Selective Repeat?

- Basic TCP looks a lot like GBN

- Many TCP implementations will buffer received out-of-order segments and then ACK them all after filling in the range
  - This looks a lot like Selective Repeat

- TCP is a hybrid
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

- Sender should not overwhelm receiver’s capacity to receive data
- If necessary, sender should slow down transmission rate to accommodate receiver’s rate.
- Different from Congestion Control whose purpose was to handle congestion in network. (But both congestion control and flow control work by slowing down data transmission)
TCP Flow Control

- receive side of TCP connection has a receive buffer:

  - flow control: 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 segment structure

- **Source port #**
- **Destination port #**
- **Sequence number**
- **Acknowledgement number**
- **Receive window**
- **Checksum**
- **Urgent data pointer**
- **Options (variable length)**
- **Application data (variable length)**

**URG:** urgent data (generally not used)

**ACK:** ACK # valid

**PSH:** push data now (generally not used)

**RST, SYN, FIN:** connection establishment (setup, teardown commands)

- **Internet checksum** (as in UDP)
- **Counting by bytes of data (not segments!)**
- **# bytes receiver willing to accept**

---

Comp 361, Spring 2005

3: Transport Layer
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
  - \( RcvWindow \)
  - \( RcvBuffer - \text{[LastByteRcvd} - \text{LastByteRead]} \)
Technical Issue

- Suppose $RcvWindow=0$ and that receiver has already ACK'ed ALL packets in buffer.
- Sender does not transmit new packets until it hears $RcvWindow>0$.
- Receiver never sends $RcvWindow>0$ since it has no new ACKS to send to Sender.
- DEADLOCK

- Solution: TCP specs require sender to continue sending packets with one data byte while $RcvWindow=0$, just to keep receiving ACKS from B. At some point the receiver's buffer will empty and $RcvWindow>0$ will be transmitted back to sender.
Note on UDP

UDP has no flow control!

UDP appends packets to receiving socket’s buffer. If buffer is full then packets are lost!
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 end system sends TCP SYN control segment to server
  - specifies client_isn, the initial seq #
  - No application data

Step 2: server end system receives SYN, replies with SYNACK control segment
  - ACKs received SYN
  - allocates buffers
  - Replies with client_isn+1 in ACK field to signal synchronization
  - Specifies server_isn
  - No application data
TCP Connection Management (cont.)

**Step 3:** client end system receives SYNACK, replies with SYN=0 and server_isn+1

- Allocate buffers
- Allocates buffers
- Can include application data

SYN=0 signals that connection established
server_isn+1 signals that # is synchronized
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” – during which will respond with ACK to received FINs (that might arrive if ACK gets lost).
- Closes down after timed-wait

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

**Note:** with small modification, can handle simultaneous FINs.
TCP Connection Management (cont)

Example TCP server lifecycle

Example TCP client lifecycle
A few special cases

- Have not discussed what happens if both client and server decide to close down connection at same time.

- It is possible that first ACK (from server) and second FIN (also from server) are sent in same segment.
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 (queuing in router buffers)
- a top-10 problem!
Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission
- Send rate $0 - C/2$

- large delays when congested
- maximum achievable throughput
Causes/costs of congestion: scenario 2

- one router, \textit{finite} buffers
- sender retransmission of lost packet
(a) (b) & (c): always $\lambda_{in} = \lambda_{out}$ (goodput)

(a) Magic transmission; only send when there's space in buffer

(b) “perfect” retransmission only when loss: $\lambda_{in} > \lambda_{out}$

(c) retransmission of delayed (not lost) packet makes $\lambda_{in}'$ larger (than perfect case) for same $\lambda_{out}$

“costs” of congestion:

(b) and (c) more work (retrans) for given “goodput”

(c) 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 $\lambda_{\text{out}}$ 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: severe congestion indicator
- RM cells returned to sender by receiver, with bits intact
  - small exception - see next page
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 minimum supportable rate on path
- EFCI bit in data cells: set to 1 by congested switch
  - Signals congestion
  - if data cell preceding RM cell has EFCI=1, destination sets CI bit=1 before returning RM cell to source.
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

- end-end control (no network assistance)
- transmission rate limited by congestion window size, $\text{Congwin}$, over segments. Congwin dynamically modified to reflect perceived congestion.

- $w$ segments, each with MSS bytes sent in one RTT:
  
  $$\text{throughput} = \frac{w \times \text{MSS}}{\text{RTT}} \text{ Bytes/sec}$$
To simplify presentation we assume that RcvBuffer is large enough that it will not overflow.

Tools are “similar” to flow control. sender limits transmission using:

\[ \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \]

How does sender perceive congestion?

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

three mechanisms:

- AIMD = Additive Increase Multiplicative Decrease
- slow start = \( \text{CongWin} \) set to 1 and then grows exponentially
- 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* also known as *congestion avoidance*

Long-lived TCP connection
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
So Far
- Slow-Start: ramps up exponentially
- Followed by AIMD: sawtooth pattern

Reality (TCP Reno)
- Introduce new variable \textit{threshold}
- \textit{threshold} initially very large
- Slow-Start exponential growth stops when reaches \textit{threshold} and then switches to AIMD
- Two different types of loss events
  - 3 dup ACKS: cut \textit{CongWin} in half and set \textit{threshold}=\textit{CongWin} (now in standard AIMD)
  - Timeout: set \textit{threshold}=\textit{CongWin}/2, \textit{CongWin}=1 and switch to Slow-Start
Reason for treating 3 dup ACKS differently than timeout is that 3 dup ACKs indicates network capable of delivering some segments while timeout before 3 dup ACKs is “more alarming”.

Note that older protocol, TCP Tahoe, treated both types of loss events the same and always goes to slowstart with Congwin=1 after a loss event.

TCP Reno’s skipping of the slow start for a 3-DUP-ACK loss event is known as fast-recovery.
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. (only in TCP Reno)
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS. (TCP Tahoe does this for 3 Dup Ack as well)
The Big Picture

![Graph showing congestion window (in segments) versus transmission round for TCP Series 1 Tahoe and TCP Series 2 Reno. The graph has a horizontal axis labeled 'Transmission round' ranging from 0 to 15 and a vertical axis labeled 'Congestion window (in segments)' ranging from 0 to 14. The graph shows two lines: one for TCP Series 1 Tahoe and another for TCP Series 2 Reno. The lines are marked with thresholds.](Image)
## TCP sender congestion control

<table>
<thead>
<tr>
<th>Event</th>
<th>State</th>
<th>TCP Sender Action</th>
<th>Commentary</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACK receipt for previously unacked data</td>
<td>Slow Start (SS)</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>ACK receipt for previously unacked data</td>
<td>Congestion Avoidance (CA)</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>Loss event detected by triple duplicate ACK</td>
<td>SS or CA</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>Timeout</td>
<td>SS or CA</td>
<td>Threshold = CongWin/2, CongWin = 1 MSS, Set state to “Slow Start”</td>
<td>Enter slow start</td>
</tr>
<tr>
<td>Duplicate ACK</td>
<td>SS or CA</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 throughput of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT
TCP Futures

- 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:
  \[ \frac{1.22 \cdot MSS}{RTT \sqrt{L}} \]
- $\Rightarrow L = 2 \cdot 10^{-10}$ Wow
- New versions of TCP for high-speed needed!
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

![Graph showing equal bandwidth share and additive increase and multiplicative decrease in TCP congestion avoidance.](image)
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
- Current Research area:
  - How to keep UDP from congesting the internet.

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!
TCP Latency Modeling

Q: How long does it take to completely receive an object from a Web server after sending a request? This is known as the latency of the (request for the) object.

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 (W)**

Two cases

1. \( WS/R > RTT + S/R: \)
   - ACK for first segment in window returns before window’s worth of data sent
   - Latency = \( 2RTT + O/R \)

2. \( WS/R < RTT + S/R: \)
   - ACK for first segment in window returns after window’s worth of data sent
   - Latency = \( 2RTT + O/R + (K-1)[S/R + RTT - WS/R] \)
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

 latency = 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

\[
\text{latency} = 2RTT + O/R + (K-1)(S/R + RTT - WS/R)
\]
**TCP Latency Modeling: Slow Start (1)**

Now suppose window grows according to slow start (with no threshold and no loss events).
Will show that the delay for one object is:

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

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 Latency 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 Latency Modeling (3)

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

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

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

\[
\text{delay} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} \text{idleTime}_p \\
= \frac{O}{R} + 2RTT + \sum_{k=1}^{P} \left[ \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \right] \\
= \frac{O}{R} + 2RTT + P \left[ RTT + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R}
\]
TCP Latency Modeling (4)

Recall $K =$ number of windows that cover object

How do we calculate $K$?

$$K = \min \{ k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O \}$$

$$= \min \{ k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S \}$$

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

$$= \min \{ k : k \geq \log_2 \left( \frac{O}{S} + 1 \right) \}$$

$$= \left\lceil \log_2 \left( \frac{O}{S} + 1 \right) \right\rceil$$

Calculation of $Q$, number of idles for infinite-size object, is similar.
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
  - $Response\ time = (M+1)O/R + (M+1)2RTT + sum\ of\ idle\ times$

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

- 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.
  - $Response\ time = (M+1)O/R + (M/X + 1)2RTT + sum\ of\ idle\ times$
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”