Performance of rdt3.0

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

\[ T_{\text{transmit}} = \frac{L}{R} \text{ (packet length in bits)} = \frac{8\text{kb/pkt}}{10^{9}\text{ b/sec}} = 8 \text{ microsec} \]

\[ U_{\text{sender}}: \text{utilization} \ - \text{fraction of time sender busy sending} \]

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

- 1KB pkt every 30 msec -> 33kB/sec throughput over 1 Gbps link
- network protocol limits use of physical resources!

Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts
- range of sequence numbers must be increased
- buffering at sender and/or receiver
- Two generic forms of pipelined protocols: go-Back-N, selective repeat

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

GBN: receiver extended FSM

GBN in action
Selective Repeat

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

Selective repeat: sender, receiver windows

(sender view of sequence numbers)

- window size N
- send_base
- nextseqnum
- already ack'd
- sent, not yet ack'd
- usable, not yet sent
- not usable
- out of order (buffered) pkts
- already ack'd
- acceptable (within window)
- not usable
- expected, not yet received

(receiver view of sequence numbers)

Selective repeat in action

(sender)

- 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

(receiver)

- pkt n in [rcvbase-N,rcvbase-1]
  - ACK(n)
- otherwise:
  - ignore

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 pkt

Packet n in [rcvbase,rcvbase+N-1]
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?

TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable, in-order byte stream:
  - no “message boundaries”
- pipelined:
  - TCP congestion and flow control set window size
- send & receive buffers

- 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

TCP segment structure

- source port #: dest port #
- sequence number
- acknowledgement number
- header:
- URG: urgent data (generally not used)
- ACK: ACK # valid
- PSH: push data now (generally not used)
- RST, SYN, FIN: connection estab (setup, teardown commands)
- checksum
- 32-bit header
- Options (variable length)
- Internet checksum (as in UDP)
- application data (variable length)
- 3 bytes rcvr willing to accept
- counting by bytes of data (not segments)
**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

---

**TCP Round Trip Time and Timeout**

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

**Q:** how to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt
- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

**Example RTT estimation:**

![Example RTT graph]

**Transport Layer 3-17**

**Transport Layer 3-18**

**Transport Layer 3-19**

**Transport Layer 3-20**
**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:
  \[ \text{DevRTT} = (1-\beta)\text{DevRTT} + \beta|\text{SampleRTT}-\text{EstimatedRTT}| \]
  (typically, \( \beta = 0.25 \))
- Then set timeout interval:
  \[ \text{TimeoutInterval} = \text{EstimatedRTT} + 4\text{DevRTT} \]

**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
- **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

**Transport Layer 3-21**

**Transport Layer 3-22**

**Transport Layer 3-23**

**Transport Layer 3-24**
TCP sender (simplified)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)
  event: data received from application above
    create TCP segment with sequence number NextSeqNum
    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
    }
}
```

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

TCP retransmission scenarios (more)

TCP ACK generation [RFC 1122, RFC 2581]

Event at Receiver | TCP Receiver action
--- | ---
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending | Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. #. Gap detected | Immediately send duplicate ACK, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap | Immediate send ACK, provided that segment starts at lower end of gap
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:

```java
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
  }
}
```

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:
  - 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 Flow control: how it works**

- Rcrr 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
  - Socket clientSocket = new Socket("hostname","port number");
- **server:** contacted by client
  - Socket connectionSocket = welcomeSocket.accept();

**Three way handshake:**

- **Step 1:** client host sends TCP SYN segment to server
  - specifies initial seq #
  - no data
- **Step 2:** server host receives SYN, replies with SYNACK segment
  - server allocates buffers
  - specifies server initial seq. #
- **Step 3:** client receives SYNACK, replies with ACK segment, which may contain data

**Closing a connection:**

- client closes socket:
  - clientSocket.close();

  **Step 1:** client end system sends TCP FIN control segment to server

  **Step 2:** server receives FIN, replies with ACK. Closes connection, sends FIN.
**TCP Connection Management (cont.)**

**Step 3:** client receives FIN, replies with ACK.
- Enters "timed wait" - will respond with ACK to received FINs.

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

**Note:** with small modification, can handle simultaneous FINs.

---

**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

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 \( \lambda_{out} \)

- 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 \( \lambda_{in} \) and \( \lambda'_{in} \) 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' 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: details

- sender limits transmission: \( \text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin} \)
- Roughly, rate = \( \frac{\text{CongWin}}{\text{RTT}} \) 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 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

TCP Slow Start

- When connection begins, \( \text{CongWin} = 1 \text{ MSS} \)
  - 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

- **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

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:**

- **Summary: TCP Congestion Control**
  - When CongWin is below Threshold, sender in **slow-start** phase, window grows exponentially.
  - When CongWin is above Threshold, sender is in **congestion-avoidance** phase, window grows linearly.
  - When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
  - When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.
**TCP 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></td>
<td></td>
<td>Supplementary:  CongWin = 1 MSS, Set state to ‘Slow Start’</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Congestion Avoidance (CA)</strong></td>
<td></td>
</tr>
<tr>
<td></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:
  \[
  \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}
  \]
- \( \Rightarrow L = 2 \cdot 10^{-10} \) **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 \( \frac{R}{K} \)

![TCP connection diagram](image)
Why is TCP fair?

Two competing sessions:
- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally

Fairness (more)

- **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**

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”