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)

**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
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

Seq=92, 8 bytes data

ACK=100

SendBase = 100

lost ACK scenario

time

Host B

Seq=92, 8 bytes data

ACK=100

Host A

Seq=92, 8 bytes data

SendBase = 100

SendBase = 120

SendBase = 100

SendBase = 120

Seq=92, 8 bytes data

ACK=100

ACK=120

ACK=120

Seq=92, 8 bytes data

ACK=100

ACK=120

Seq=100, 20 bytes data

premature timeout

time

Host B

Seq=92, 8 bytes data

ACK=100

ACK=120

Seq=92, 8 bytes data

ACK=100

ACK=120

Seq=100, 20 bytes data

premature timeout

time
TCP retransmission scenarios (more)

Cumulative ACK scenario

Host A

Seq=92, 8 bytes data

Host B

Seq=100, 20 bytes data

ACK=100

ACK=120

SendBase = 120

time

timeout

loss
<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 <em>duplicate ACK</em>, 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>
• 1982: TCP/IP “Version 3” officially split TCP and IP into two protocol layers
• 1988: the “slow-start” scheme was added as TCP-Tahoe Version
• 19??: the “fast recovery” schemes was added as TCP-Reno Version
• 1995: TCP-Vegas Version
Fast Recovery

• 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
Figure 3.37 Resending a segment after triple duplicate ACK
event: ACK received, with ACK field value of $y$
  if ($y > \text{SendBase}$) {
    $\text{SendBase} = y$
    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 $y$
  }
}
event: ACK received, with ACK field value of \( y \)

  if (\( y > \text{SendBase} \)) {
    \text{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 \( y \)
  }
}
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

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

- spare room in buffer
  - \( \text{RcvWindow} \)
  - \( \text{RcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}] \)

Rcvr advertises spare room by including value of \( \text{RcvWindow} \) in segments

Sender limits unACKed data to \( \text{RcvWindow} \)

guarantees receive buffer doesn’t overflow