**The Transport Layer**
Reliable data delivery & flow control in TCP

*Jasleen Kaur*

March 31, 2016

---

**Transport Layer Protocols & Services**

**Outline**

- Fundamental transport layer services
  - Multiplexing/Demultiplexing
  - Error detection
  - Reliable data delivery
  - Pipelining
  - Flow control
  - Congestion control

- Internet transport protocols
  - UDP
  - TCP
TCP Overview

TCP is:

- Point-to-point, full-duplex
  - Bi-directional data flow with in a connection
- Reliable, in-order byte stream
  - No “message boundaries”
- Connection-oriented
  - Handshaking initializes sender and receiver state before data exchange
- Pipelined
  - Congestion and flow control determine window size
  - Each endpoint has **two** buffers: a send and receive buffer

- Congestion controlled
  - Internet would cease to function without this!
- Flow controlled
  - Sender and receiver have synchronized windows to ensure receiver is not overwhelmed

TCP Segment Structure

**Header and payload format**

- 32 bits
- "Urgent data" ("URG")
- ACK number is valid
- "Push data now" ("PSH")
- Options (variable length)
- Sequence number
- Acknowledgement number
- Source port #
- Destination port #
- HEADN (not used)
- RCVR window size
- Checksum
- PTR urgent data
- Payload ≤ MSS (Maximum Segment Size)
TCP Sequence Numbers and ACKs

Telnet example

- Sequence numbers:
  - Byte stream "index" of the first byte in the segment's payload
- ACKs:
  - Sequence number of next byte expected from the other side
  - ACKs are cumulative
- How does receiver handle out-of-order segments?
  - TCP spec doesn't say, it's up to the implementor

TCP Reliable Data Transfer (RDT)

- 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 flow control, congestion control (they determine the window size dynamically)
Reliable Data Transfer in TCP

**Sender’s state machine**

- TCP_send(data)
  - create segment
  - utl_send(segment nextseqnum)
  - start timer

- receive ACK for segment y
  - update timers
  - advance the window to y
  - retransmit y if third duplicate ACK

- timeout for segment y
  - utl_send(segment y)
  - start timer

- **TCP retransmits segments if:**
  - An expected ACK times out
  - 3 duplicate ACKs for a segment are received

---

**Reliable Data Transfer in TCP**

**Simplified sender**

- **Comment:**
  - SendBase = 1 = last cumulatively ACKed byte

- **Example:**
  - SendBase-1 = 71; ACK = 73, so the receiver wants 73+
  - ACK > SendBase, so new data (SeqNum = 72) is ACKed
Reliable Data Transfer in TCP
Simplified sender’s state machine

sendbase = initial_sequence number
nextseqnum = initial_sequence number

loop (forever) {
  switch(event) {
    event: data received from application above
      nextseqnum = nextseqnum + length(data)
    event: ACK received with value y
      send ACK of all data up to y
      sendbase = y
    event: timer timeout for segment with sequence number y
      retransmit segment with sequence number y
      compute new timeout interval for segment y
      restart timer for segment y
  }
}

if (y > sendbase) {
  /* Cumulative ACK of all data up to y */
  cancel timers for segments with sequence numbers y
  sendbase = y
}
else if (y == sendbase) {
  /* A duplicate ACK */
  increment number of duplicate ACKs received
  if (number of duplicate ACKs received == 3) {
    /* Fast retransmit */
    resend segment with sequence number sendbase
    restart timer for segment sendbase
  }
}
### Reliable Data Transfer in TCP

#### Setting the ACK timer

- How large should the ACK timeout value be?
  - Too short: Premature timeouts result in unnecessary retransmissions
  - Too long: Slow reaction to loss results in poor performance because the sender’s window stops advancing

- Timer should be longer than the RTT, but how do we estimate RTT?
  - Measure the time from segment transmission until receipt of ACK
  - Ignore retransmissions
  - Measure only one segment’s RTT at a time

  *SampleRTT* will vary, so we compute an average RTT based on several recent RTT samples

#### Estimating round-trip-time

- The estimated RTT is an exponential weighted moving average (EWMA)
  - Computes a “smooth” average
  - Influence of a given sample decreases exponentially fast
    \[
    E_n = x^*S_n + (1-x)^*S_{n-1} + (1-x)^2*S_{n-2} + \ldots + x(1-x)^{n-1}*S_1 + \ldots
    \]
  - Typical value of \(x\) is 0.125

- Timeout is *EstimatedRTT* plus “safety margin”

- Large variation in *EstimatedRTT* results in a larger safety margin
**Example RTT Estimate (smoothed)**

![Example RTT Estimate Graph]

**Fast Retransmit**

- Time-out period often relatively long:
  - long delay before resending lost packet
  - timer granularity ≈ 10-200 ms
  - minimum RTO = 200-1200 ms

- Detect lost segments via duplicate ACKs:
  - Sender often sends many segments back-to-back
  - If a segment is lost, there will likely be many duplicate ACKs (one for each segment that arrives safely)

- If sender receives 3 ACKs for the same data, it assumes that the segment after ACKed data was lost:
  - fast retransmit: resend segment before timer expires
  - Why wait for 3?
**Reliable Data Transfer in TCP**

**ACK generation rules [RFC 1122, RFC 2581]**

<table>
<thead>
<tr>
<th>Event</th>
<th>TCP Receiver action</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-order segment arrival, no gaps, all previous data already ACKed</td>
<td>Delayed ACK. Wait 200 ms (up to 500 ms allowed) for next segment. If no segment received, send ACK</td>
</tr>
<tr>
<td>In-order segment arrival, no gaps, one delayed ACK pending</td>
<td>Immediately send single cumulative ACK</td>
</tr>
<tr>
<td>Out-of-order segment arrival (higher than expected sequence number) — Gap detected</td>
<td>Send duplicate ACK, indicating sequence number of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediate ACK if segment starts at lower end of gap</td>
</tr>
</tbody>
</table>

**TCP Flow Control**

**Window control**

- Flow control is the problem of ensuring the receiver is not overwhelmed
  - The receiver can become overwhelmed if the application reads too slow or the sender transmits too fast
- The receiver’s window represents its remaining buffer capacity
- The window advances as the application reads received data
TCP Flow Control

Window control

- The receiver explicitly informs the sender of the amount of free buffer space in RcvBuffer
  - RcvWindow field in TCP segment
- The sender keeps the amount of transmitted, unACKed data less than most recently received RcvWindow

The goal is to ensure:

\[
\text{LastByteRecvd} - \text{LastByteRead} \leq \text{RcvBuffer}
\]

Sender is sent:

\[
\text{RcvWindow} = \text{RcvBuffer} - (\text{LastByteRecvd} - \text{LastByteRead})
\]
TCP Flow Control

Window control

- The sender ensures:

\[ \text{LastByteSent} - \text{LastByteACKed} \leq \text{RcvWindow} \]
TCP Connection Management

The three-way handshake

- Client sends SYN segment to server
  - The SYN specifies the client’s initial sequence number
- Server receives SYN, replies with SYN + ACK segment
  - ACKs received SYN
  - Allocates buffers
  - Specifies server’s initial sequence number
- Third segment may be an ACK only or an ACK+data

TCP Connection Management

Closing a connection

- Client sends FIN segment to server
- Server receives FIN, replies with ACK
  - Server closes connection, sends FIN
- Client receives FIN, replies with ACK
- Client enters “timed wait” state
  - Client will ACK any received FIN
TCP Connection Management
Client/Server lifecycles

- TCP client lifecycle
- TCP server lifecycle