The Transport Layer
Reliable data delivery & flow control in TCP

Jasleen Kaur
March 28, 2019
TCP Overview

TCP is...

- **Point-to-point, full-duplex**
  - Bi-directional data flow within 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**

<table>
<thead>
<tr>
<th>Field</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>16 bits</td>
<td>Source port of the sending application</td>
</tr>
<tr>
<td>dest port #</td>
<td>16 bits</td>
<td>Destination port of the receiving application</td>
</tr>
<tr>
<td>sequence number</td>
<td>32 bits</td>
<td>The sequence number of the data segment</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>32 bits</td>
<td>The acknowledgement number of the data segment</td>
</tr>
<tr>
<td>rcvr window size</td>
<td>16 bits</td>
<td>The size of the receiver's window</td>
</tr>
<tr>
<td>checksum</td>
<td></td>
<td>The checksum of the segment</td>
</tr>
<tr>
<td>ptr urgent data</td>
<td></td>
<td>The pointer to urgent data</td>
</tr>
<tr>
<td>Options</td>
<td>variable</td>
<td>Additional options may be present</td>
</tr>
<tr>
<td>Application data</td>
<td>variable</td>
<td>The actual data being transmitted</td>
</tr>
<tr>
<td>payload</td>
<td>≤ MSS</td>
<td>The amount of data that can be transmitted</td>
</tr>
</tbody>
</table>

*Note: 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 (simplified!)

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

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

wait for event

TCP_send(data)
create segment
udt_send(segment nextseqnum)
start timer

timeout for segment y
udt_send(segment y)
start timer

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

```plaintext
sendbase = initial_sequence_number
nextseqnum = initial_sequence_number

loop (forever){
    switch(event)
    case data received from application above:
    case timer timeout for segment with sequence number y:
    case ACK received with value y:
        break;
    default:
        create TCP segment with sequence number nextseqnum
        start timer for segment with nextseqnum
        pass segment to IP
        nextseqnum = nextseqnum + length(data)
        break;
}

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
        }
```

```plaintext
1st Byte
Last Byte
```
Reliable Data Transfer in TCP
Retransmission examples

- Lost ACK scenario
- Premature timeout

- Cumulative ACKs potentially avoid retransmissions
- Premature timeout
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
    (“SampleRTT”)
  - 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

Reliable Data Transfer in TCP

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
    \[ E_n = x \cdot S_n + (1-x) \cdot E_{n-1} + \ldots \]
    - Typical value of \( x \) is 0.125
  - Timeout is \( \text{EstimatedRTT} \) plus “safety margin”
  - Large variation in \( \text{EstimatedRTT} \) results in a larger safety margin

\[
\text{EstimatedRTT} = (1-x) \cdot \text{EstimatedRTT} + x \cdot \text{SampleRTT}
\]
\[
\text{Timeout} = \text{EstimatedRTT} + 4 \cdot \text{Deviation}
\]
\[
\text{Deviation} = (1-x) \cdot \text{Deviation} + x \cdot |\text{SampleRTT} - \text{EstimatedRTT}|
\]
Example RTT Estimate (smoothed)

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

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)</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

**Receiver 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

Receiver's window

source port  dest port
sequence number  ack number
data offset  window size
checksum  urgent data
TCP options
Application data

TCP Flow Control

Window control

◆ The goal is to ensure:
  LastByteRead - LastByteRcvd ≤ RcvBuffer
◆ Sender is sent:
  RcvWindow = RcvBuffer - (LastByteRcvd - LastByteRead)
TCP Flow Control
Sender Window control

- The sender ensures:
  \[\text{LastByteSent} - \text{LastByteACKed} \leq \text{RcvWindow}\]

TCP Connection Management
The three-way handshake

- TCP endpoints establish a "connection" before exchanging data segments
  - client: connection initiator
    ```
    clientSocket = socket(AF_INET, SOCK_STREAM)
    clientSocket.connect((serverName, serverPort))
    ```
  - server: contacted by client
    ```
    connectionSocket, addr = serverSocket.accept()
    ```
TCP Connection Management
The three-way handshake

- Client sends SYN segment to server
  - The SYN specifies the client's initial sequence number
  - The ACK number in the SYN will be 0

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence number</td>
<td></td>
</tr>
<tr>
<td>acknowledgement number</td>
<td></td>
</tr>
<tr>
<td>seq</td>
<td>not used</td>
</tr>
<tr>
<td>Options (variable length)</td>
<td></td>
</tr>
</tbody>
</table>

Application data (variable length)

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