TCP Handshake & Flow Control

CS640, 2015-03-10

Outline
- Basic reliability
- Connection setup & teardown
- TCP header
- Flow control

Basic reliability
- Goal: end-to-end communication channel that is robust to packet loss and reordering
- Basic mechanism: acknowledgements (ACKs) and timeouts
- Timeline diagram

[Timeline diagram showing data transmission and acknowledgement]

- Lost data

[Diagram showing TCP handshake and flow control]
- Lost ACK

![Diagram of lost ACK](image1)

- Overly aggressive timeout

![Diagram of overly aggressive timeout](image2)

- Stop-and-wait
  - Send one data packet; wait for ACK before sending next data packet
  - Inefficient -- requires at least one RTT for each data packet
  - Ideal: send multiple data packets in parallel
    - Packets may be reordered -- need way to ensure order
    - Some packets may be lost -- ideally only lost packets are retransmitted

- Challenges
  - Unique identifier for data -- to determine ordering and know which data was lost
  - Determining how many packets to send in parallel -- to avoid overloading receiver
  - Setting timeout -- timeout depends on RTT
  - Determining which data to retransmit -- hard when multiple packets are “in-flight”
Sequence #s
- Used for ordering data and knowing when data is missing
- Assign sequence # to each packet? — no
  - Requires retransmitted packets to contain the exact same payload
- Instead, assign sequence # to each byte of data
  - Data packet contains seq # for 1st byte in the packet payload
  - ACK packet contains seq # for next expected byte of data
  - No need for separate ACK packet for each received packet — cumulative ACKs

Sender

\[
\begin{align*}
\text{Data seq}=1, \text{len}=100 \\
\text{Data seq}=101, \text{len}=100 \\
\text{ACK, ack}=201
\end{align*}
\]

- Starting seq # is randomly chosen — need connection setup phase to share this info
- Challenge: defining the sequence # space

Connection Establishment
- Initializes control information (e.g., initial sequence #) on sender and receiver
- Three-way handshake

Active
Client

Passive
Server

- Slightly different ordering of SYNs and ACKs is possible
  - TCP state machine determines current state of connection — Figure 5.7, pg 405
  - Transition between states when:
    - Application calls socket function
    - Host receives TCP packet with specific flags set
Connection Termination

- Tell opposite side of connection that you are done sending data
- Four-way handshake

<table>
<thead>
<tr>
<th>Active</th>
<th>Passive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Client</td>
<td>Server</td>
</tr>
<tr>
<td><img src="image" alt="Diagram of four-way handshake" /></td>
<td></td>
</tr>
</tbody>
</table>

- Other orderings are also possible -- see TCP state transition diagram
- Wait maximum segment lifetime (MSL) before destroying control information
  - Necessary in case ACK is lost or other data packets still need to be retransmitted
  - MSL assumed to be 120 seconds

TCP header

<table>
<thead>
<tr>
<th>0</th>
<th>4</th>
<th>10</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source port</td>
<td>Destination port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sequence #</td>
<td>Acknowledgement #</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HdrLen</td>
<td>Unused</td>
<td>Flags</td>
<td>Window size</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Urgent pointer</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Sequence # -- seq # for the 1st byte in the payload of this TCP packet
- Acknowledgement # -- seq # for the next byte of data the receiver expects; only relevant if ACK flag is set
- Header length -- length of the TCP header in bytes; TCP header may contain options
- Flags -- SYN, ACK, FIN, RST, PSH, URG
  - PSH -- sender requested buffered data to be sent
  - URG -- packet contains urgent data at the start of the payload
- Window size -- how much buffer space the receiver has available
- Urgent pointer -- which byte of the payload is the start of the non-urgent data; only relevant if URG flag is set
Flow control
- Goal: specify what & when to send
- Necessary for reliability and flow control
- Sender buffer

- Inequalities that must hold
  - \( \text{LastByteAcked} \leq \text{LastByteSent} \) -- data that has not been sent cannot be acknowledged
  - \( \text{LastByteSent} \leq \text{LastByteWritten} \) -- data that has not been generated by the sending app cannot be sent

- Receiver buffer

- Inequalities that must hold
  - \( \text{LastByteRead} < \text{NextByteExpected} \) -- receiving app cannot consume data that has not been received; cannot consume later data when earlier data is missing
  - \( \text{NextByteExpected} \leq \text{LastByteRcvd} + 1 \) -- cannot expect data that is beyond the latest data that was received

- Buffers have finite size -- MaxSendBuffer and MaxRecvBuffer
  - Need to limit the amount of data a sender sends to the receiver -- otherwise the receiver won’t be able to buffer it and the sender will need to retransmit it
  - Need to limit the amount of data the sending app generates -- otherwise the sender won’t be able to buffer it while waiting for other data to be sent to/received by the receiver