Connection-oriented Transport: TCP

Outline:
- segment structure
- reliable data transfer
- flow control
- connection management

TCP: Overview

- point-to-point:
  - one sender, one receiver
- reliable data transfer:
  - guaranteed arrival, no error, in order
- pipelined:
  - multiple in-flight segments
- full duplex data:
  - bi-directional data flow in same connection
- connection-oriented:
  - handshaking (exchange of control msgs) to initialize sender, receiver state before data exchange
- flow controlled:
  - sender does not overwhelm receiver
- congestion controlled:
  - sender does not overwhelm the network
- no delay or bandwidth guarantee.

TCP Segment Structure

TCP Reliable Data Transfer

- TCP provides reliable data transfer service on top of IP's unreliable service
- Pipelined transmissions & cumulative ACKs
- Looks like Go-back-N, however the receiver could buffer out-of-order segments
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate ACKs

TCP Timeout

Q: how to set TCP timeout value?

- longer than RTT (round trip time)
  - but RTT varies
- too short: premature timeout and unnecessary retransmissions
- too long: slow reaction to segment loss
Estimating TCP Round Trip Time

Q: how to estimate RTT?
- SampleRTT: measured time from segment transmission until ACK receipt
- SampleRTT fluctuates, we want estimated RTT “smoother” to avoid short-term spikes
  - average several recent measurements, not just current SampleRTT
- we also want to give more recent measurements higher weight in case things do change

EWMA - Exponentially Weighted Moving Average

EstimatedRTT = ($\alpha$SampleRTT, $\alpha(1-\alpha)$SampleRTT, $\alpha(1-\alpha)^2$SampleRTT, ...

$\alpha$ is the most recent sample TTL.
SampleRTT is the next recent sample TTL, etc.

Then set timeout interval:
TimeoutInterval = EstimatedRTT + 4*DevRTT

TCP Timeout

Setting the timeout:
- EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT -> larger safety margin
- we need to estimate of how much SampleRTT deviates from EstimatedRTT (EWMA):

DevRTT = ($1-\beta$)DevRTT_last + $\beta$|SampleRTT-EstimatedRTT|

(typically, $\beta = 0.25$)

TCP Sender Events and Processing

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 un-ACKed segment)
- timeout vale: we just decided it!!

TCP byte-oriented 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

simple telnet scenario
TCP in Action: Retransmissions

Host A
ACK=100
time
Host B
Seq=92, 8 bytes data
Seq=92, 8 bytes data
Seq=92 timeout
ACK=100

Host A
Seq=92, 8 bytes data
ACK=100
loss
time

Host B
X
Seq=92, 8 bytes data
ACK=100
time
Seq=92 timeout
Slide send window base to 100

TCP in Action: Cumulative ACK

Host A
Seq=92, 8 bytes data
ACK=100
loss
time

Host B
X
Seq=100, 20 bytes data
ACK=120
Slide send window base to 120

TCP in Action: Duplicate ACKs and Fast Retransmission

Host A
Seq=92, 8 bytes data
ACK=92
loss
time

Host B
X
Seq=100, 8 bytes data
ACK=92
3 duplicate ACKs
resend 92
Seq=108, 8 bytes data
Seq=116, 8 bytes data
ACK=92
ACK=92

Fast Retransmission

- 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.
- When receiver receives out-of-order segments, it re-ACK the last in-order byte
- If sender receives 3 ACKs for the same data, it supposes that segment after ACKed data was lost:
  - fast retransmission: resend segment before timer expires, restart timer

TCP Flow Control

- receive side of TCP connection has a receive buffer:
  - flow control: sender does not overflow receiver's buffer by transmitting too much, too fast
  - speed-matching service: matching the send rate to the receiving app's drain rate

Outline

- segment structure
- reliable data transfer
- flow control
- connection management
TCP Flow Control: how it works

(Suppose TCP receiver discards out-of-order segments)
- spare room in buffer = RcvWindow
- RcvBuffer = [LastByteRcvd - LastByteRead]

- Rcvr advertises spare room by including value of RcvWindow in segments
- Sender limits unACKed data to RcvWindow
- guarantees receive buffer doesn't overflow

Where is RcvWindow in each segment?

<table>
<thead>
<tr>
<th>source port #</th>
<th>dest port #</th>
<th>sequence number</th>
<th>acknowledgement number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Options (variable length)

TCP Connection Management

- Establishment:
  - TCP sender, receiver establish "connection" before exchanging data segments
  - initialize TCP variables: starting seq. #s, MSS, buffers, flow control info (e.g. RcvWindow)
  - MSS is the maximum TCP segment size each side is willing to accept
    - typically the largest segment size fit into a link-layer frame
    - What is MSS for a IPv4 host connected with Ethernet?
- Teardown:
  - freeing up resources after mutually close

TCP Connection Establishment

Three way handshake:

Step 1: client (active open) host sends TCP SYN segment to server
- specifies initial seq #
- no data

Step 2: server (passive open) 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

TCP Connection Teardown

Closing a connection:
close socket: close(socket);

Step 1: A (active closing host) sends TCP FIN control segment to server

Step 2: B (passive closing host) receives FIN, replies with ACK. Closes connection, sends FIN.

Step 3: A receives FIN, replies with ACK.
- Enters "time out" - resend ACK in case it is lost

Step 4: B receives ACK. Connection closed.
TCP State Transition Diagram

- **CLOSED**
- **LISTEN**
- **SYN_RCVD**
- **SYN_SENT**
- **ESTABLISHED**
- **CLOSE_WAIT**
- **LAST_ACK**
- **CLOSING**
- **TIME_WAIT**
- **FIN_WAIT_2**
- **FIN_WAIT_1**

### Passive open
- Send/SYN
- SYN/SYN + ACK
- SYN + ACK/ACK

### Active open
- SYN/SYN + ACK
- Send/SYN
- SYN + ACK/ACK

### Close
- Close/FIN
- FIN/ACK
- ACK + FIN/ACK

### Timeout after two segment lifetimes
- Close/FIN
- FIN/ACK
- ACK
- ACK

### Disclaimer

- Parts of the lecture slides contain original work of James Kurose, Larry Peterson, and Keith Ross. The slides are intended for the sole purpose of instruction of computer networks at the University of Rochester. All copyrighted materials belong to their original owner(s).