Transmission Control Protocol (TCP)

CS 352, Lecture 8
http://www.cs.rutgers.edu/~sn624/352-S19

Srinivas Narayana
(slides heavily adapted from text authors’ material)
Transmission Control Protocol: Overview

- **point-to-point:**
  - one sender, one receiver

- **reliable, in-order *byte stream***:
  - no “message boundaries”

- **pipelined**:
  - TCP congestion and flow control set window size

- **full duplex data:**
  - bi-directional data flow in same connection
  - MSS: maximum segment size

- **connection-oriented**:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange

- **flow controlled**:
  - sender will not overwhelm receiver

RFCs: 793, 1122, 1323, 2018, 2581
## TCP segment structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port #</td>
<td>Source port number</td>
</tr>
<tr>
<td>dest port #</td>
<td>Destination port number</td>
</tr>
<tr>
<td>sequence number</td>
<td>Sequence number</td>
</tr>
<tr>
<td>acknowledgement number</td>
<td>Acknowledgement number</td>
</tr>
<tr>
<td>head len</td>
<td>Head length</td>
</tr>
<tr>
<td>not used</td>
<td>Not used</td>
</tr>
<tr>
<td>UAPRSF</td>
<td>Options (URG, ACK, PSH, RST, SYN, FIN)</td>
</tr>
<tr>
<td>receive window</td>
<td>Receive window</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>options (variable length)</td>
<td>Options (variable length)</td>
</tr>
<tr>
<td>application data</td>
<td>Application data</td>
</tr>
<tr>
<td>(variable length)</td>
<td>(variable length)</td>
</tr>
</tbody>
</table>

- **URG**: urgent data (generally not used)
- **ACK**: ACK # valid
- **PSH**: push data now (generally not used)
- **RST, SYN, FIN**: connection estab (setup, teardown commands)
- **Internet checksum**: (as in UDP)
TCP seq. numbers, ACKs

sequence numbers:
- byte stream “number” of first byte in segment’s data

acknowledgements:
- seq # of next byte expected from other side
- cumulative ACK

How does receiver handle out-of-order segments?
- A: TCP spec doesn’t say, up to implementor
TCP seq. numbers, ACKs

User types ‘C’

Seq=42, ACK=79, data = ‘C’

Seq=79, ACK=43, data = ‘C’

Seq=43, ACK=80

host ACKs receipt of echoed ‘C’

host ACKs receipt of echoed ‘C’, echoes back ‘C’

simple telnet scenario
Reliable transfer in TCP
TCP reliable data transfer

• TCP creates reliable service on top of IP’s unreliable service
  • pipelined segments
  • cumulative acks
  • single retransmission timer

• retransmissions triggered by:
  • timeout events
  • duplicate acks

Let’s initially consider simplified TCP sender:
• ignore duplicate acks
• ignore flow control, congestion control
TCP round trip time, timeout

**Q:** how to set TCP timeout value?

- longer than RTT
  - but RTT varies
- *too short:* premature timeout, unnecessary retransmissions
- *too long:* slow reaction to segment loss

**Q:** how to estimate RTT?

- **SampleRTT**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **SampleRTT** will vary, want estimated RTT “smoother”
  - average several recent measurements, not just current SampleRTT
TCP round trip time, timeout

\[
\text{EstimatedRTT} = (1 - \alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}
\]

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: \( \alpha = 0.125 \)
TCP round trip time, timeout

- **timeout interval**: $\text{EstimatedRTT}$ plus “safety margin”
  - large variation in $\text{EstimatedRTT}$ -> larger safety margin
- estimate $\text{SampleRTT}$ deviation from $\text{EstimatedRTT}$:
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]
  (typically, $\beta = 0.25$)

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}$$
TCP sender events: Managing a single timer

*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
  - expiration interval: `TimeOutInterval`

*timeout:*
- retransmit segment that caused timeout
- restart timer

*ack rcvd:*
- if ack acknowledges previously unacked segments
  - update what is known to be ACKed
  - restart timer if there are still unacked segments
TCP: retransmission scenarios

**Lost ACK scenario**

Host A

Seq=92, 8 bytes of data

ACK=100

X

Timeout

Host B

Seq=92, 8 bytes of data

ACK=100

**Premature timeout**

Host A

SendBase=92

Seq=92, 8 bytes of data

ACK=100

Timeout

Host B

SendBase=92

Seq=100, 20 bytes of data

ACK=100

ACK=120

SendBase=120

Seq=92, 8 bytes of data

SendBase=120

ACK=120

But segment 120 not transmitted
TCP: retransmission scenarios

Cumulative ACK avoids retransmission altogether
# TCP receiver events: ACKing

<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 <code>duplicate ACK</code>, 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>
TCP fast retransmit

- timeout period often relatively long:
  - long delay before resending lost packet

- Instead: 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 same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #
  - likely that unacked segment lost, so don’t wait for timeout
TCP fast retransmit

Host A
Seq=92, 8 bytes of data
Seq=100, 20 bytes of data

Host B
ACK=100
ACK=100
ACK=100
ACK=100

timeout

fast retransmit after sender receipt of triple duplicate ACK
Problem with RTT Calculation

Sender Timeout

RTT?

RTT?

Sender

Receiver

Sender Timeout

RTT?

RTT?

ACK = 2048
Karn’s algorithm

• Retransmission *ambiguity*
  • Measure RTT from original data segment
  • Measure RTT from most recent segment

• Either way there is a problem in RTT estimate

• One solution
  • Never update RTT measurements based on acknowledgements from retransmitted packets

• Problem: Sudden change in RTT can cause system never to update RTT
  • Primary path failure leads to a slower secondary path
Karn’s algorithm

• Use back-off as part of RTT computation
• Whenever packet loss, RTO is increased by a factor
• Use this increased RTO as RTO estimate for the next segment (not from SRTT)
• Only after an acknowledgment received for a successful transmission is the timer set to new RTT obtained from SRTT
Flow Control
TCP flow control

flow control
receiver controls sender, so sender won’t overflow receiver’s buffer by transmitting too much, too fast

TCP flow control
application may remove data from TCP socket buffers ....
... slower than TCP receiver is delivering (sender is sending)
receiver protocol stack

TCP socket receiver buffers
IP code
TCP code
application process
OS
from sender
TCP flow control

- receiver “advertises” free buffer space by including rwnd value in TCP header of receiver-to-sender segments
  - RcvBuffer size set via socket options (typical default is 4096 bytes)
  - many operating systems autoadjust RcvBuffer
- sender limits amount of unacked (“in-flight”) data to receiver’s rwnd value
- guarantees receive buffer will not overflow
Connection Management
Connection Management

before exchanging data, sender/receiver “handshake”:

• agree to establish connection (each knowing the other willing to establish connection)

• agree on connection parameters

Socket clientSocket =
newSocket("hostname","port number");

Socket connectionSocket =
welcomeSocket.accept();
Agreeing to establish a connection

2-way handshake:

Q: will 2-way handshake always work in network?
- variable delays
- retransmitted messages (e.g. `req_conn(x)`) due to message loss
- message reordering
- can’t “see” other side
Agreeing to establish a connection

2-way handshake failure scenarios:

- **choose x**
- **req_conn(x)**
- **ESTAB**
- **acc_conn(x)**
- **ESTAB**
- **client terminates**
- **half open connection!**
  (no client!)

- **retransmit req_conn(x)**
- **req_conn(x)**
- **ESTAB**
- **client terminates**

- **choose x**
- **req_conn(x)**
- **ESTAB**
- **data(x+1)**
- **accept data(x+1)**
- **server forgets x**
TCP 3-way handshake

**client state**

- **LISTEN**
  - choose init seq num, x
  - send TCP SYN msg

- **SYNSENT**
  - SYNbit=1, Seq=x
  - SYNbit=1, Seq=y
  - received SYNACK(x)
  - indicates server is live;
  - send ACK for SYNACK;
  - this segment may contain client-to-server data

- **ESTAB**
  - ACKbit=1, ACKnum=x+1
  - received ACK(y)
  - indicates client is live

**server state**

- **LISTEN**
  - SYN RCVD
  - choose init seq num, y
  - send TCP SYNACK msg, acking SYN

- **ESTAB**
  - SYNbit=1, Seq=y
  - ACKbit=1; ACKnum=x+1
  - received ACK(y)
  - indicates client is live

Transport Layer
TCP 3-way handshake: FSM

```
Socket connectionSocket = welcomeSocket.accept();

SYN(x)
SYNACK(seq=y,ACKnum=x+1)
create new socket for communication back to client

ACK(ACKnum=y+1)

Socket clientSocket = newSocket("hostname","port number");

SYN(seq=x)

SYN/rcvd
SYN/sent
ESTAB

Λ
Λ
Λ
```

Closed
Listen
SYN/rcvd
SYN/sent
ESTAB

Λ
Λ
Λ
TCP: closing a connection

- client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FIN
- simultaneous FIN exchanges can be handled
TCP: closing a connection

**client state**
- **ESTAB**
  - clientSocket.close()
  - FIN_WAIT_1
  - FIN_WAIT_2
  - TIMED_WAIT
- CLOSED

**server state**
- **ESTAB**
- CLOSE_WAIT
- LAST_ACK
- CLOSED

WORKFLOW:
- **FIN_WAIT_1**:
  - FINbit=1, seq=x
  - ACKbit=1, ACKnum=x+1
  - can no longer send but can receive data

- **FIN_WAIT_2**:
  - wait for server close
  - FINbit=1, seq=y
  - ACKbit=1; ACKnum=y+1

- **TIMED_WAIT**:
  - timed wait for 2*max segment lifetime

- **CLOSED**:
  - can no longer send data

- **LAST_ACK**:
  - can no longer send data