Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management

3.6 principles of congestion control
3.7 TCP congestion control
TCP: 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
### 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>Urg data pointer</td>
<td>Urgent data pointer</td>
</tr>
<tr>
<td>receive window</td>
<td>Receive window</td>
</tr>
<tr>
<td>checksum</td>
<td>Checksum</td>
</tr>
<tr>
<td>options</td>
<td>Variable length options</td>
</tr>
<tr>
<td>application data</td>
<td>Application data (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

Q: how receiver handles out-of-order segments
- A: TCP spec doesn’t say, - up to implementor
TCP seq. numbers, ACKs

simple telnet scenario

Host A

User types 'C'

host ACKs receipt of echoed 'C'

Seq=42, ACK=79, data = 'C'

Seq=79, ACK=43, data = 'C'

Seq=43, ACK=80

Host B

host ACKs receipt of 'C', echoes back 'C'
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**: EstimatedRTT plus “safety margin”
  - large variation in EstimatedRTT \(\rightarrow\) larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]
  (typically, \(\beta = 0.25\))

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
  • segment structure
  • reliable data transfer
  • flow control
  • connection management

3.6 principles of congestion control
3.7 TCP congestion control
TCP reliable data transfer

- TCP creates rdt 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 sender events:

**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
  - start timer if there are still unacked segments
**TCP sender** (simplified)

- \[ \text{NextSeqNum} = \text{InitialSeqNum} \]
- \[ \text{SendBase} = \text{InitialSeqNum} \]

**Wait for event**

- Create segment, seq. #: NextSeqNum
- Pass segment to IP (i.e., "send")
- \[ \text{NextSeqNum} = \text{NextSeqNum} + \text{length(data)} \]
- If (timer currently not running)
  - Start timer

**Data received from application above**

- \[ \text{ACK received, with ACK field value } y \]

- If \( y > \text{SendBase} \) {
  - \[ \text{SendBase} = y \]
  - /* SendBase–1: last cumulatively ACKed byte */
  - If (there are currently not-yet-acked segments)
    - Start timer
  - Else stop timer
}

- Timeout
  - Retransmit not-yet-acked segment
    - With smallest seq. #
  - Start timer
TCP: retransmission scenarios

lost ACK scenario

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

Host B
Seq=92, 8 bytes of data
ACK=100

premature timeout

Host A
SendBase=92
Seq=92, 8 bytes of data
ACK=100
SendBase=100
Seq=100, 20 bytes of data
ACK=100
SendBase=120
Seq=92, 8 bytes of data
ACK=120
SendBase=120

Host B
ACK=120
TCP: retransmission scenarios

Host A

Seq=92, 8 bytes of data
Seq=100, 20 bytes of data
timeout

Host B

X
ACK=100

ACK=120

Seq=120, 15 bytes of data

cumulative ACK
# TCP ACK generation

[RFC 1122, RFC 2581]

<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 duplicate ACK, indicating seq. # of next expected byte</td>
</tr>
<tr>
<td>Arrival of segment that partially or completely fills gap</td>
<td>Immediately send ACK, provided that segment starts at lower end of gap</td>
</tr>
</tbody>
</table>
TCP fast retransmit

- 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.

TCP fast retransmit
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
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
   - segment structure
   - reliable data transfer
   - flow control
   - connection management

3.6 principles of congestion control
3.7 TCP congestion control
TCP flow control

Application may remove data from TCP socket buffers …

... slower than TCP receiver is delivering (sender is sending)

Flow control

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

Receiver protocol stack
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
Chapter 3 outline

3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer

3.5 connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management

3.6 principles of congestion control
3.7 TCP congestion control
Connection Management

before exchanging data, sender/receiver “handshake”:

- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters

application

connection state: ESTAB
connection variables:
- seq # client-to-server
- server-to-client
- rcvBuffer size
  at server, client

network

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

application

connection state: ESTAB
connection Variables:
- seq # client-to-server
- server-to-client
- rcvBuffer size
  at server, client

network

Socket connectionSocket =
welcomeSocket.accept();

"Transport Layer 3-22"
TCP 3-way handshake

**client state**

- CLOSED
  - choose init seq num, x
  - send TCP SYN msg

- SYNSENT
  - received SYNACK(x)
  - SYNbit=1, Seq=x
  - ACKbit=1; ACKnum=x+1

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

**server state**

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

- SYNSENT
  - received ACK(x)
  - indicates server is live
  - send ACK for SYNACK;
  - this segment may contain client-to-server data

- ESTAB
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

ESTAB

SYN sent

Transport Layer 3-24
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()
  - FINbit=1, seq=x
  - can no longer send but can receive data

- **FIN_WAIT_1**
  - wait for server close
  - ACKbit=1; ACKnum=x+1
  - can still send data

- **FIN_WAIT_2**
  - timed wait for 2*max segment lifetime
  - FINbit=1, seq=y
  - ACKbit=1; ACKnum=y+1
  - can no longer send data

- **TIMED_WAIT**

- **CLOSED**

**Server state**

- **ESTAB**

- **CLOSE_WAIT**
  - FINbit=1, seq=x

- **LAST_ACK**
  - ACKbit=1; ACKnum=x+1
  - can still send data

- **CLOSED**

- **FIN_WAIT_1**
  - FINbit=1, seq=x

- **FIN_WAIT_2**

- **TIMED_WAIT**

- **CLOSED**