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

RFCs: 793, 1122, 1323, 2018, 2581

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

- reliable, in-order **byte steam**:
  - 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

- **source port #**
- **dest port #**
- **sequence number**
- **acknowledgement number**
- **receive window**
- **checksum**
- **Urg data pointer**
- **options (variable length)**
- **application data** (variable length)

**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)

**counting by bytes of data (not segments!)**

**# bytes rcvr willing to accept**
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

Host A

User types ‘C’

Seq=42, ACK=79, data = ‘C’

host ACKs receipt of echoed ‘C’

Seq=79, ACK=43, data = ‘C’

host ACKs

Seq=43, ACK=80

Host B

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

simple telnet scenario
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 $\text{EstimatedRTT} \rightarrow$ larger safety margin
- estimate SampleRTT deviation from $\text{EstimatedRTT}$:
  \[
  \text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|
  \]
  (typically, $\beta = 0.25$)

\[
\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}
\]
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)

wait for event

- NextSeqNum = InitialSeqNum
- SendBase = InitialSeqNum

create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
    start timer

data received from application above

ACK received, with ACK field value y
if (y > SendBase) {
    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

Transport Layer 3-12
TCP: retransmission scenarios

lost ACK scenario

Host A
Seq=92, 8 bytes of data
SendBase=92

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

timeout

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

Host B
Seq=100, 20 bytes of data
ACK=100

premature timeout

Host A
SendBase=120

Host B
Seq=92, 8 bytes of data
ACK=120

SendBase=120

SendBase=120

SendBase=92
TCP: retransmission scenarios

Host A

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

Host B

ACK=100

X

ACK=120

Seq=120, 15 bytes of data

Cumulative ACK
### TCP ACK generation

**event at receiver** | **TCP receiver action**
---|---
arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed | delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
arrival of in-order segment with expected seq #. One other segment has ACK pending | immediately send single cumulative ACK, ACKing both in-order segments
arrival of out-of-order segment higher-than-expect seq. #. Gap detected | immediately send *duplicate ACK*, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap | immediate send ACK, provided that segment starts at lower end of gap
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
ACK=100
ACK=100
ACK=100
timeout

Host B

X

fast retransmit after sender receipt of triple duplicate ACK

Transport Layer 3-17
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

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

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
- Retransmit req_conn(x)
- ESTAB
- req_conn(x)
- acc_conn(x)
- ESTAB
- req_conn(x)
- connection x completes
- server forgets x
- client terminates
- half open connection! (no client!)

- Choose x
- Retransmit req_conn(x)
- ESTAB
- req_conn(x)
- acc_conn(x)
- ESTAB
- data(x+1)
- client terminates
- connection x completes
- server forgets x
- accept data(x+1)
TCP 3-way handshake

**client state**

- **LISTEN**
  - Choose init seq num, \( x \)
  - Send TCP SYN msg

- **SYNSENT**
  - Choose init seq num, \( x \)
  - Send TCP SYN msg

- **ESTAB**
  - Received SYNACK(\( x \))
  - ACKbit=1; ACKnum=x+1

**server state**

- **LISTEN**
  - Choose init seq num, \( y \)
  - Send TCP SYNACK msg, acking SYN

- **SYN RCVD**
  - SYNbit=1, Seq=x

- **ESTAB**
  - Received ACK(\( y \))
  - Indicates client is live

**Transport Layer 3-25**
TCP 3-way handshake: FSM

Socket connectionSocket = welcomeSocket.accept();

SYN(seq=x)

create new socket for communication back to client

SYNACK(seq=y, ACKnum=x+1)

ACK(ACKnum=y+1)

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

SYN(seq=x)

SYN(sent)

SYNACK(seq=y, ACKnum=x+1)

ACK(ACKnum=y+1)

ESTAB

closed

listen

SYN(rcvd)

closed

SYN(rcvd)

Transport Layer 3-26
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**
  - can no longer send but can receive data
- **FIN_WAIT_2**
  - wait for server close
- **TIMED_WAIT**
  - timed wait for 2*max segment lifetime
- **CLOSED**

**Server state**
- **ESTAB**
- **CLOSE_WAIT**
  - can still send data
- **LAST_ACK**
  - can no longer send data
- **CLOSED**

FINbit=1, seq=x
ACKbit=1; ACKnum=x+1

FINbit=1, seq=y
ACKbit=1; ACKnum=y+1

Transport Layer 3-28