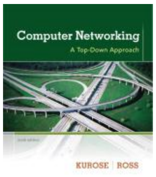
Chapter 3 Transport Layer

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you see the animations; and can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:

- If you use these slides (e.g., in a class) that you mention their source (after all, we'd like people to use our book!)
- If you post any slides on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR



Computer Networking: A Top Down Approach
6th edition
Jim Kurose, Keith Ross
Addison-Wesley
March 2012

The course notes are adapted for Bucknell's CSCI 363
Xiannong Meng
Spring 2016

Chapter 3 outline

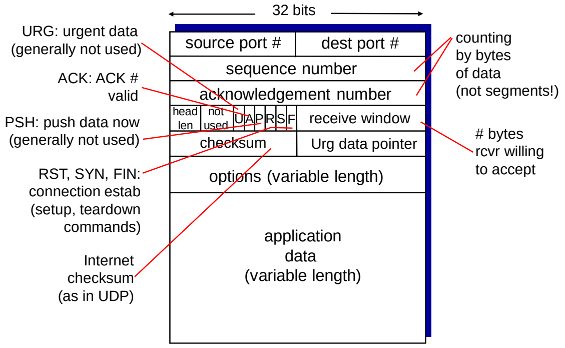
3.1 transport-layer services
3.2 multiplexing and demultiplexing
3.3 connectionless transport: UDP
3.4 principles of reliable data transfer
3.5 connection-oriented transport: TCP
- segment structure
- reliable data transfer
- flow control
- connection management

3.6 principles of congestion control
3.7 TCP congestion control

TCP: Overview RFCs: 793,1122,1323, 2018, 2581, 5681

- 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



32 bits

source port #	dest port #
sequence number	
acknowledgement number	
head len, not used, U A P R S F	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 header file

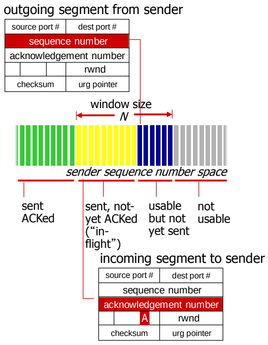
- The TCP header is defined in tcp.h in the directory of /usr/include/netinet/
- View it from Linux file system

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



outgoing segment from sender

source port #	dest port #
sequence number	
acknowledgement number	
	rwnd
checksum	urg pointer

window size
N

sender sequence number space

sent ACKed

sent, not-yet ACKed ("in-flight")

usable but not yet sent

not usable

incoming segment to sender

source port #	dest port #
sequence number	
acknowledgement number	
A	rwnd
checksum	urg pointer

TCP seq. numbers, ACKs (1)

- Assume A sends B a 500,000 bytes file, and Maximum Segment Size (MSS) is 1,000 bytes, the first byte is numbered 0, B only sends ack, no other information
- The file is segmented into 500 segments,
  - 0-999, 1000-1999, … 499,000-499,999
- (seq, ack) from A to B would be (0, n/a), (1000, n/a) …

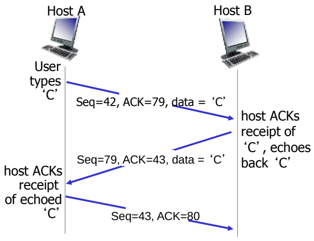
TCP seq. numbers, ACKs (2)

- If B also sends something data to A, the acks can be "piggy-backed" in data segments
- We may see the (seq, ack) between A and B as
  - A(0, n/a), B(0, 1000), A(1000, 5), B(5, 2000), …
  - Where B(0, 1000) means B is sending packet starting from 0, and B has received packets up to 999 from A, expecting packet 1000 from A
  - A(0, n/a) means A is sending packets starting from 0, the ack field is not used because nothing has received from B yet

TCP seq. numbers, ACKs (3)

- Packets could arrive out of order, for example A has received all the bytes from 0 through 535, and from 900 through 999, but missing packets between 536 and 899. How to handle? Two options
  - Ack through 535, discard 900 through 999
  - Ack through 535, buffer 900 through 999 for later reassemble
- TCP standards didn't specify what to do.
- The application layer always sees ordered data, nothing out-of-order is available to application.

TCP seq. numbers, ACKs


Host A
Host B
User types 'C'
Seq=42, ACK=79, data = 'C'
host ACKs receipt of 'C', echoes back 'C'
Seq=79, ACK=43, data = 'C'
host ACKs receipt of echoed 'C'
Seq=43, ACK=80

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

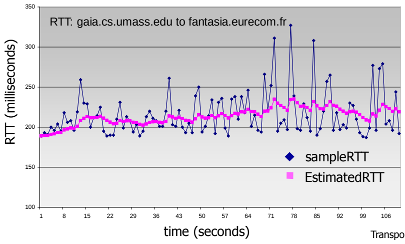
Q: how to estimate RTT?

- **`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

`EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT`

- exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: α = 0.125



TCP round trip time, timeout example

$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$

$$\alpha = 0.125$$

Packet #	Estimate RTT (ms)	Sample RTT (ms)
20	10	40
21	14	42
22	18	35
23	20	38
24	23	…

TCP round trip time, timeout

- timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin
- estimate SampleRTT deviation from EstimatedRTT:

$$\texttt{DevRTT} = (1-\beta)*\texttt{DevRTT} + \beta*|\texttt{SampleRTT}-\texttt{EstimatedRTT}|$$

(typically, $\beta = 0.25$)

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4*\texttt{DevRTT}$$

estimated RTT

"safety margin"

TCP round trip time, timeout example

$$\texttt{DevRTT} = (1-0.25)*5 + 0.25*|38-20| = 9 \text{ ms}$$

$$\texttt{TimeoutInterval} = \texttt{EstimatedRTT} + 4*\texttt{DevRTT}$$

estimated RTT

"safety margin"

In our example:
TimeoutInterval = 20 + 4 * 9 = 56 ms

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

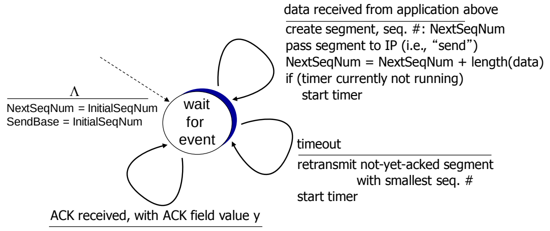
TCP sender events:

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 Fig. 3.33, p. 243)



Λ
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

wait for event

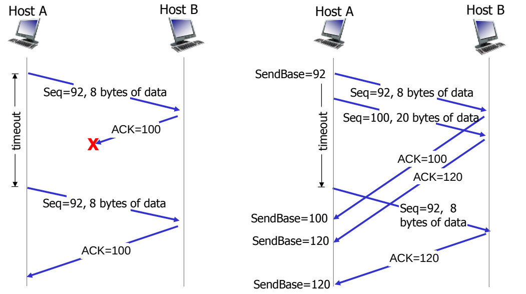
data received from application above
create segment, seq. #: NextSeqNum
pass segment to IP (i.e., “send”)
NextSeqNum = NextSeqNum + length(data)
if (timer currently not running)
start timer

timeout
retransmit not-yet-acked segment with smallest seq. #
start timer

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
}
```

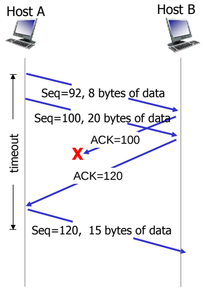
TCP: retransmission scenarios



lost ACK scenario

premature timeout

TCP: retransmission scenarios

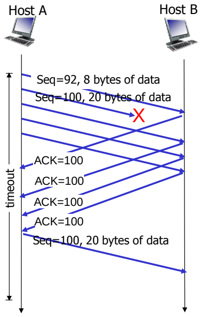


cumulative ACK

TCP ACK generation [RFC 1122, RFC 2581]

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 retransmit scenario



What does it mean if the sender receives multiple ACKs for the same packet?

An earlier packet is lost!

Fast retransmit after sender receipt of triple duplicate ACK

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