AMRITA
VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

COMPUTER NETWORKS
3-0-0 3

TRANSPORT LAYER

Chapter 3: Transport Layer

TCP segment structure

Chapter 3 outline

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 send and Receive Buffers

- Once connection TCP established through **3 way hand shaking**, the two application processes can send data to each other.
- TCP directs this data to the connection's **send buffer**, which is one of the buffers that is set aside during the initial three-way handshake
- The maximum amount of data that can be grabbed and placed in a segment is limited by the **maximum segment size (MSS)**.
- When TCP receives a segment at the other end, the segment's data is placed in the TCP connection's **receive buffer**, The application reads the stream of data from this buffer

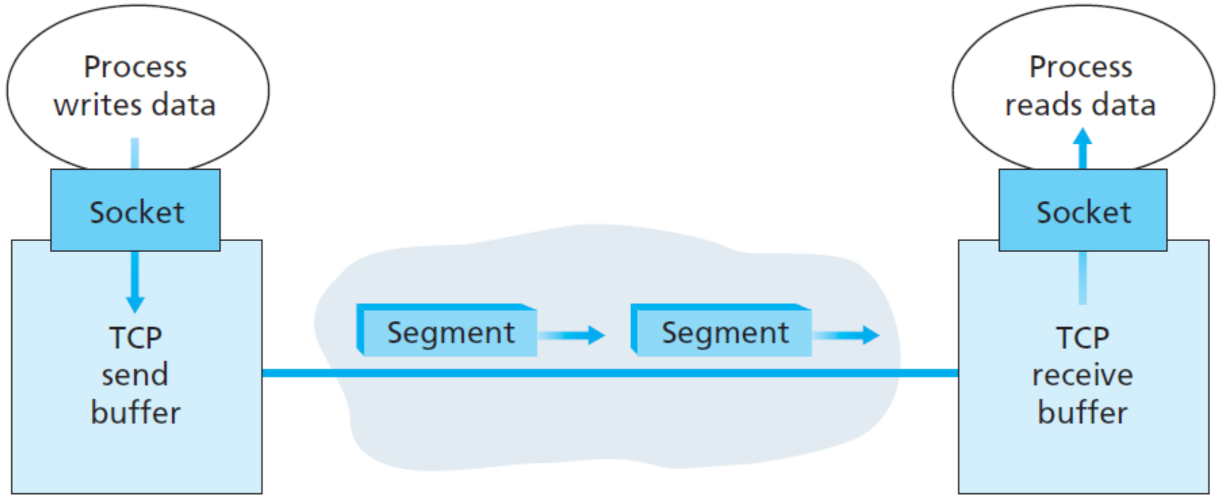


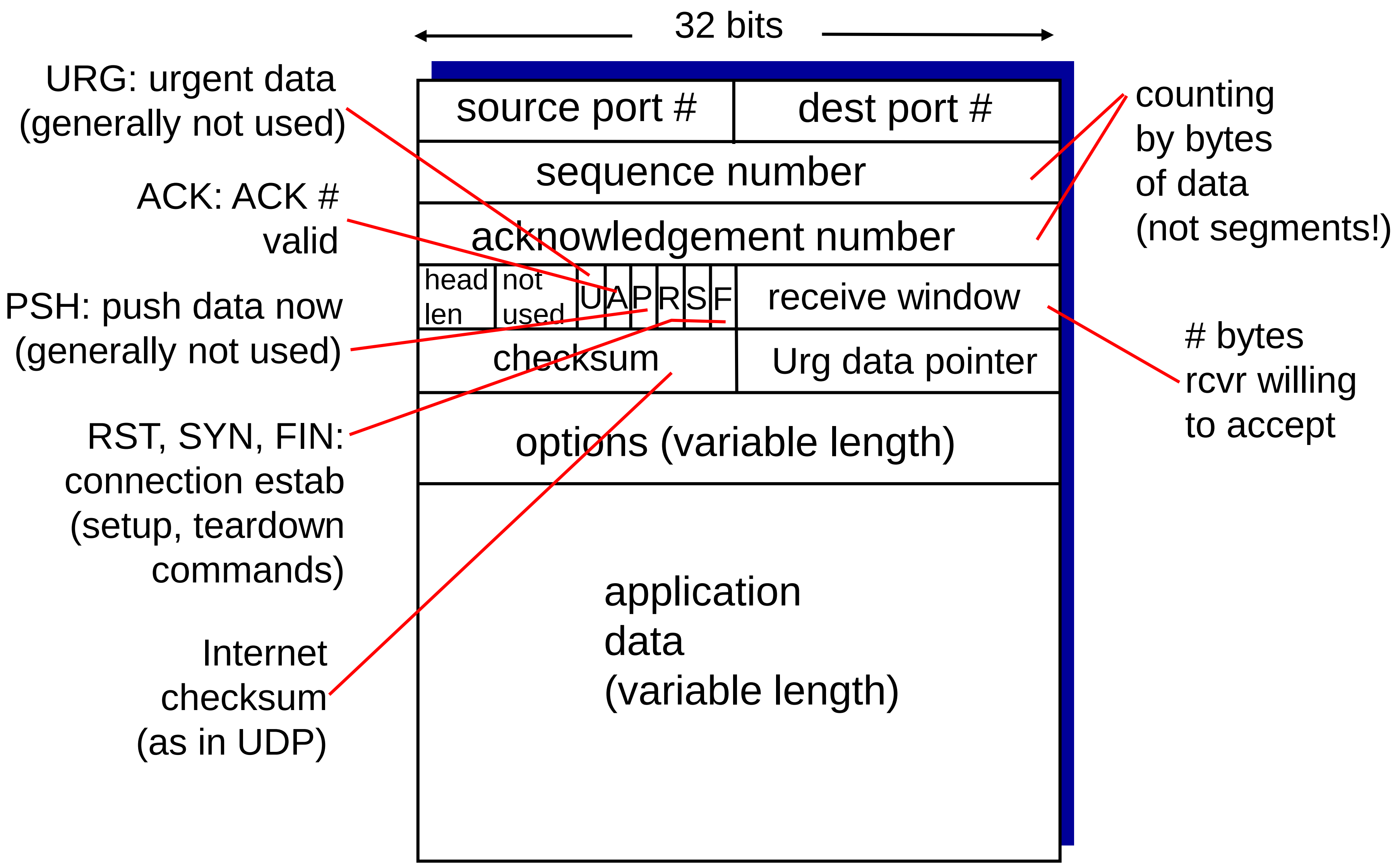
Figure 3.28 ♦ TCP send and receive buffers

TCP segment structure

The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown,

Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately.

, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as "urgent."


32 bits
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)
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)
counting by bytes of data (not segments!)
# bytes rcvr willing to accept

TCP Header

- The TCP header includes **source and destination port numbers**, which are used for multiplexing/demultiplexing data from/to upper-layer applications. Also, as with
- The header includes a **checksum field**.
- The 32-bit **sequence number field** and the 32-bit **acknowledgment number field** are used by the TCP sender and receiver in implementing a reliable data transfer service, as discussed below.
- • The 16-bit **receive window** field is used for flow control. The 4-bit **header length field** specifies the length of the TCP header in 32-bit words. The TCP header can be of variable length due to the TCP options field.
- The optional and variable-length **options field** is used when a sender and receiver negotiate the maximum segment size (MSS) or as a window scaling factor for use in high-speed networks.
- The **flag field** contains 6 bits.
- The **ACK bit** is used to indicate that the value carried in the acknowledgment field is valid; that is, the segment contains anacknowledgment for a segment that has been successfully received.
- The **RST**, **SYN**, and **FIN** bits are used for connection setup and teardown, as we will discuss at the end of this section.
- Setting the **PSH** bit indicates that the receiver should pass the data to the upper layer immediately.
- Finally, the **URG** bit is used to indicate that there is data in this segment that the sending-side upper-layer entity has marked as “urgent.” The location of the last byte of this urgent data is indicated by the 16-bit **urgent data pointer field**.

TCP seq. numbers, ACKs

sequence numbers:

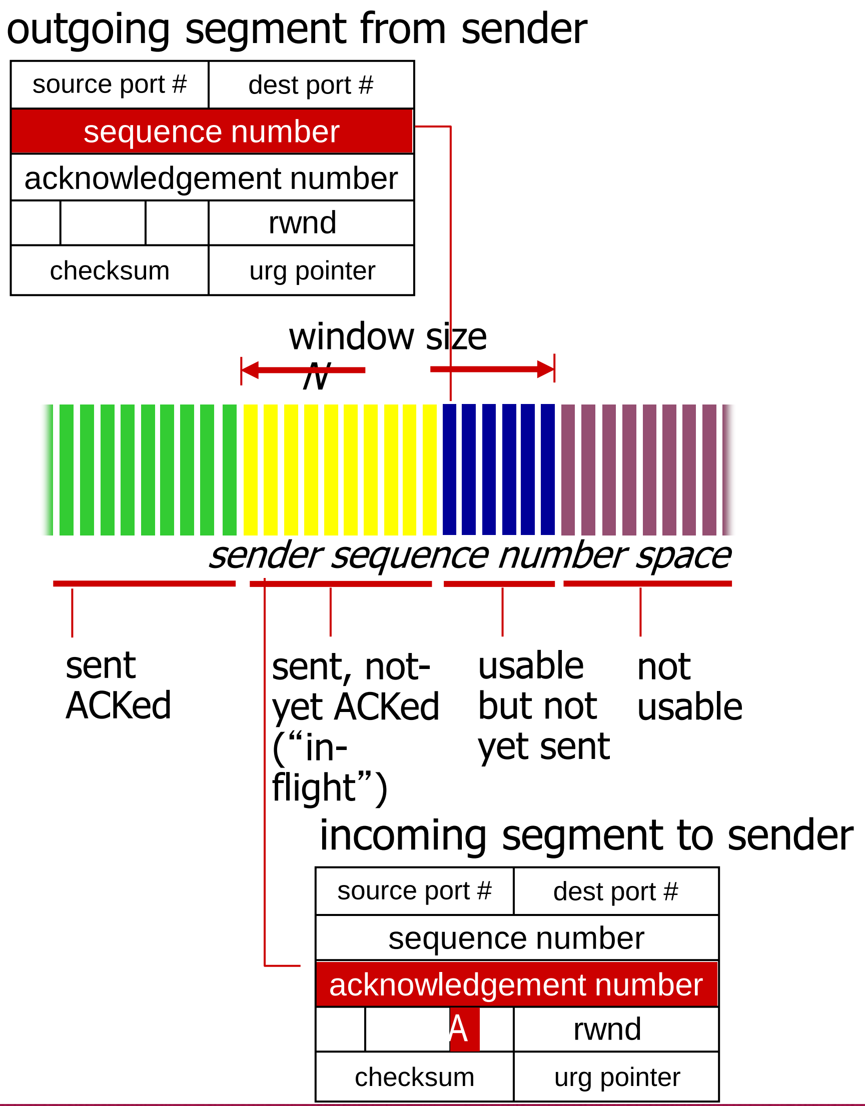
- byte stream "number" of first byte in segment's data

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

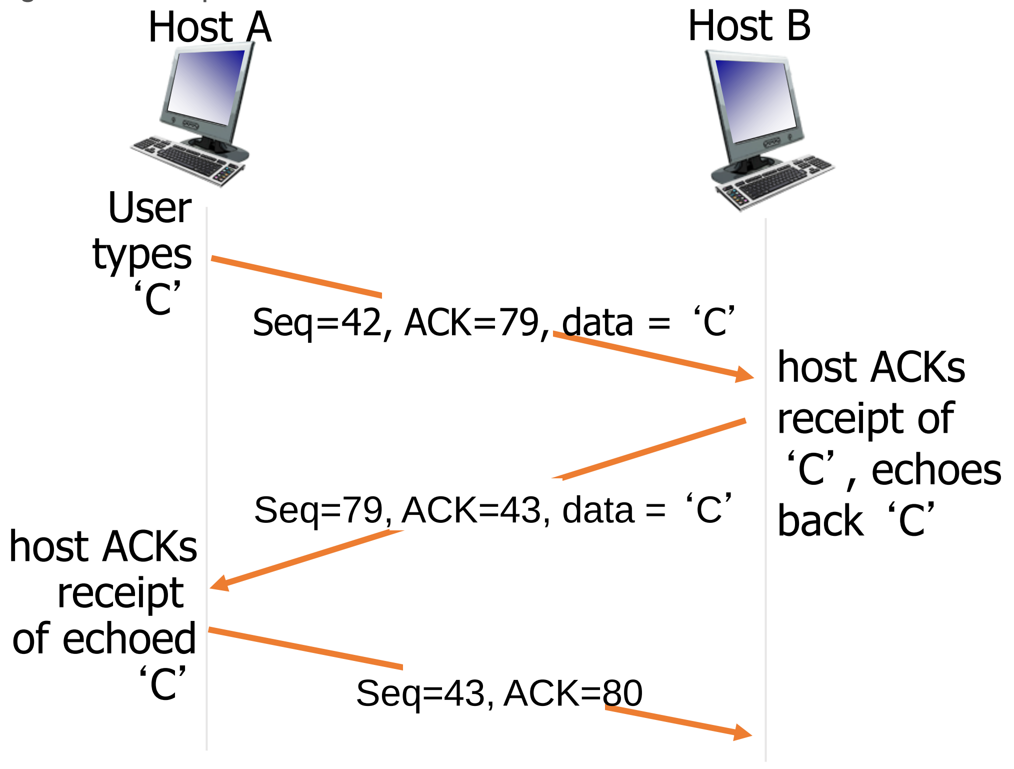


TCP seq. numbers, ACKs

Piggybacking data transfer. If station A wants to send both data and an acknowledgment, it keeps both fields there.

Sequence Number: The sequence number for a segment the byte-stream number of the first byte in the segment
Suppose that the data stream consists of a file consisting of 500,000 bytes, that the MSS is 1,000 bytes, and that the first byte of the data stream is numbered 0.
, TCP constructs 500 segments out of the data stream. The first segment gets assigned sequence number 0, the second segment gets assigned sequence number 1,000, the third segment gets assigned sequence number 2,000, and so on. Each sequence number is inserted in the sequence number field in the header of the appropriate TCP segment.

The acknowledgment number *that Host A puts in its segment is the sequence number of the next byte Host A is expecting from Host B.*
Suppose that Host A has received all bytes numbered 0 through 535 from B and suppose that it is about to send a segment to Host B.
Host A is waiting for byte 536 and all the subsequent bytes in Host B's data stream.
So Host A puts 536 in the acknowledgment number field of the segment it sends to B.


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

Suppose Host A initiates a Telnet session with Host B. Because Host A initiates the session, it is labeled the client and Host B is labeled the server. Each character typed by the user (at the client) will be sent to the remote host; the remote host will send back a copy of each character, which will be displayed on the Telnet user's screen

cumulative acknowledgments

Suppose that Host A has received one segment from Host B containing bytes 0 through 535 and another segment containing bytes 900 through 1,000. For some reason Host A has not yet received bytes 536 through 899.
In this example, Host A is still waiting for byte 536 (and beyond) in order to re-create B's data stream.
Thus, A's next segment to B will contain 536 in the acknowledgment number field. Because TCP only acknowledges bytes up to the first missing byte in the stream, TCP is said to provide **cumulative acknowledgments**

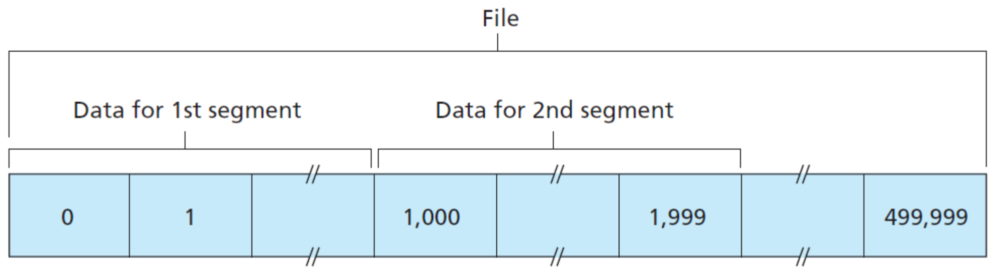


Figure 3.30 ♦ Dividing file data into TCP segments

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

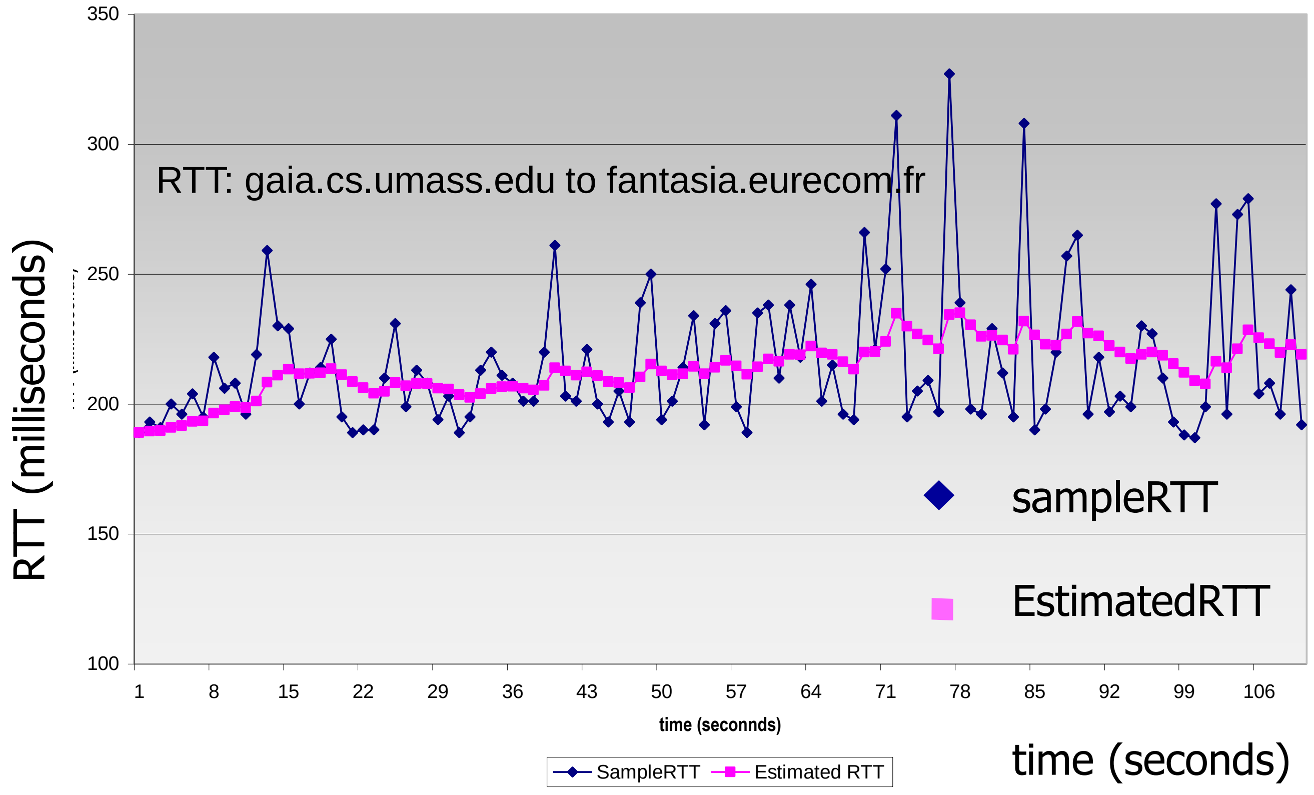
Q: how to estimate RTT?

- **`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: $\alpha = 0.125$



time (seconds)

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, $\beta = 0.25$)

$$\texttt{TimeoutInterval} = \underbrace{\texttt{EstimatedRTT}}_{\text{estimated RTT}} + \underbrace{4*\texttt{DevRTT}}_{\text{"safety margin"}}$$

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Chapter 3 outline

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 reliable data transfer

TCP creates a **reliable data transfer service** on top of IP's unreliable besteffort service. TCP's reliable data transfer service ensures that the data stream that a process reads out of its TCP receive buffer is uncorrupted, without gaps, without duplication, and in sequence; that is, the byte stream is exactly the same byte stream that was sent by the end system on the other side of the connection

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`

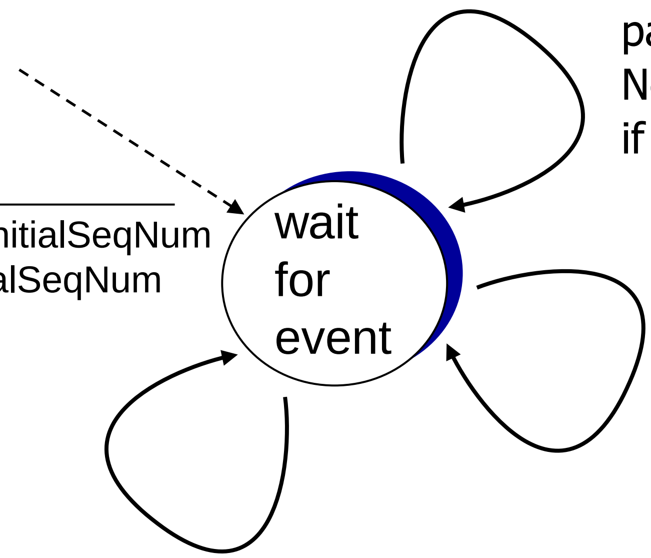
timeout:

- 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)



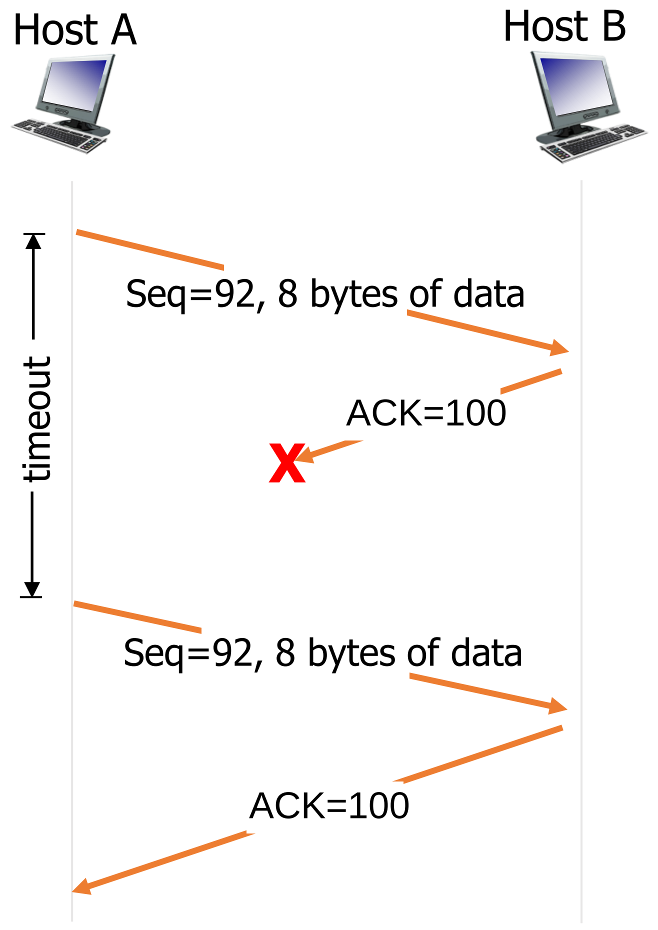
```
Λ
―――――――――――――――――――――
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
```

```
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
```

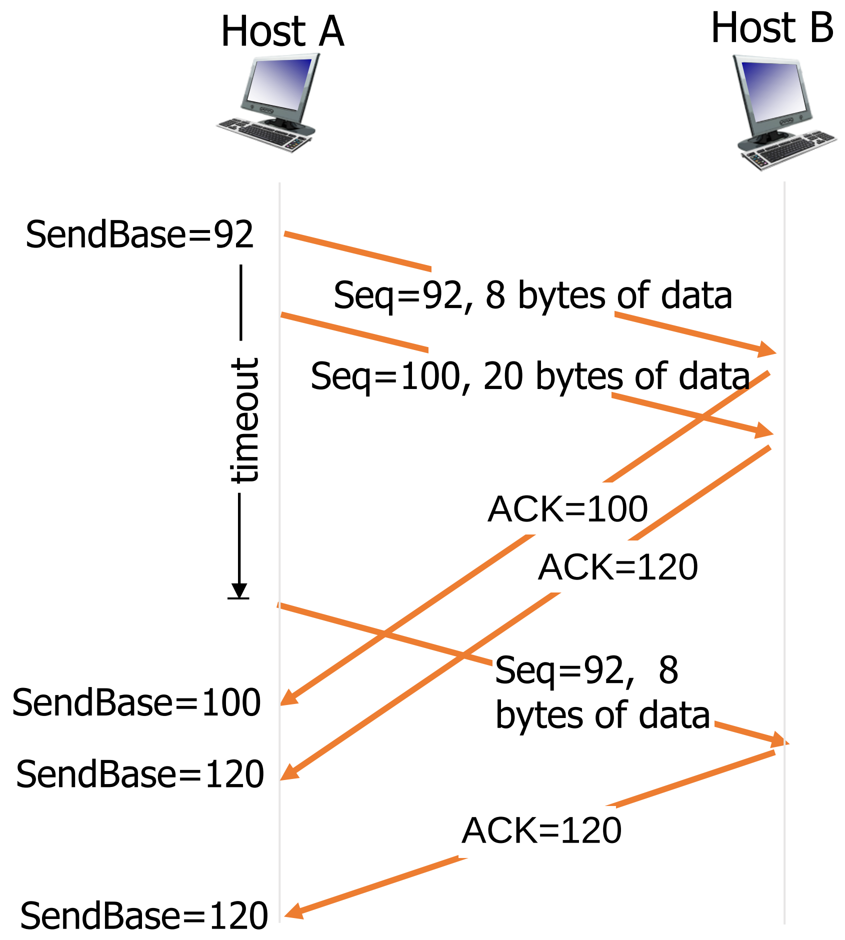
```
timeout
―――――――――――――――――――――――――――――――――
retransmit not-yet-acked segment
                with smallest seq. #
start timer
```

```
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



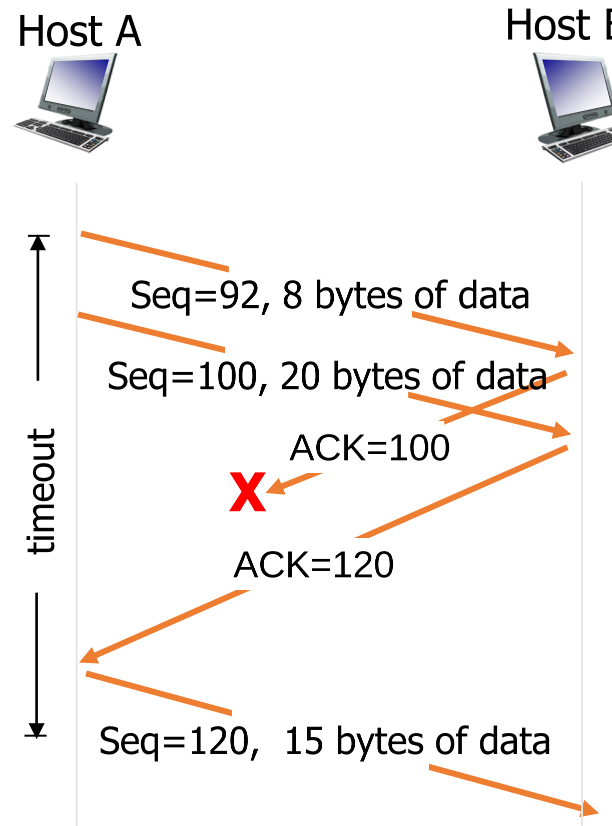
lost ACK scenario



premature timeout

No need to re transmit Seq 100. ACK 120 is cumulative

TCP: retransmission scenarios



cumulative ACK

Doubling the Timeout Interval

- TCP retransmits the not-yet acknowledged segment with the smallest sequence number, But each time TCP retransmits, it sets the next timeout interval to twice the previous value, rather than deriving it from the last `EstimatedRTT` and `DevRTT`
- Suppose `TimeoutInterval` associated with the oldest not yet acknowledged segment is .75 sec when the timer first expires. TCP will then retransmit this segment and set the new expiration time to 1.5 sec. If the timer expires again 1.5 sec later, TCP will again retransmit this segment, now setting the expiration time to 3.0 sec. Thus the intervals grow exponentially after each retransmission.
- However, whenever the timer is started after either of the two other events (that is, data received from application above, and ACK received), the `TimeoutInterval` is derived from the most recent values of `EstimatedRTT` and `DevRTT`

This modification provides a **limited form of congestion control**. The timer expiration is most likely caused by congestion in the network, that is, too many packets arriving at one (or more) router queues in the path between the source and destination, causing packets to be dropped and/or long queuing delays. In times of congestion, if the sources continue to retransmit packets persistently, the congestion may get worse. Instead, TCP acts more politely, with each sender retransmitting after longer and longer intervals.

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 fast retransmit

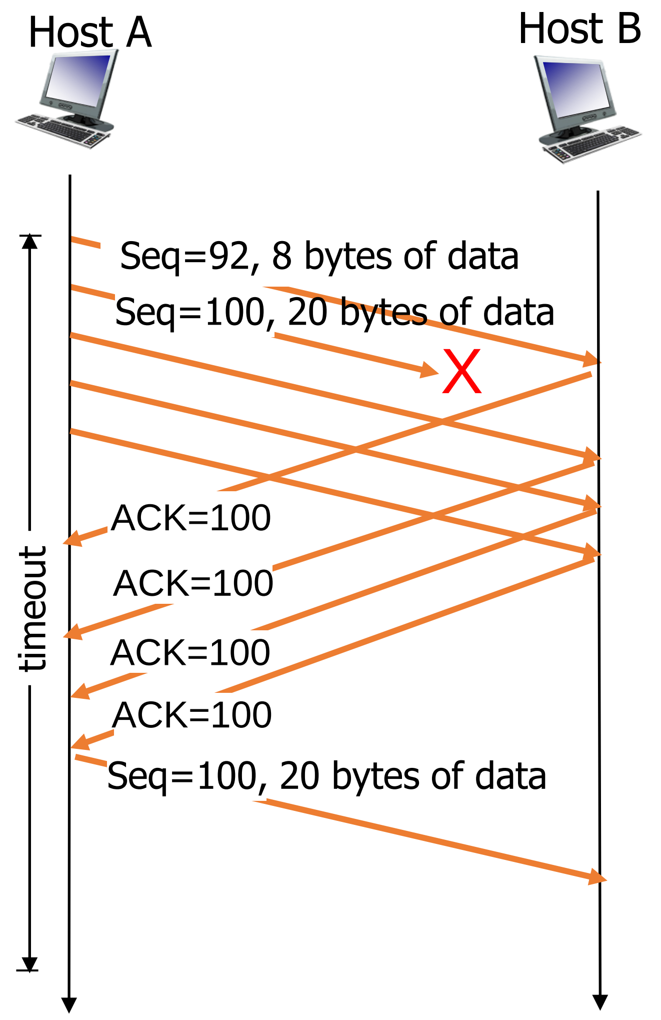
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

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



fast retransmit after sender receipt of triple duplicate ACK

Chapter 3 outline

Namah Shivaya