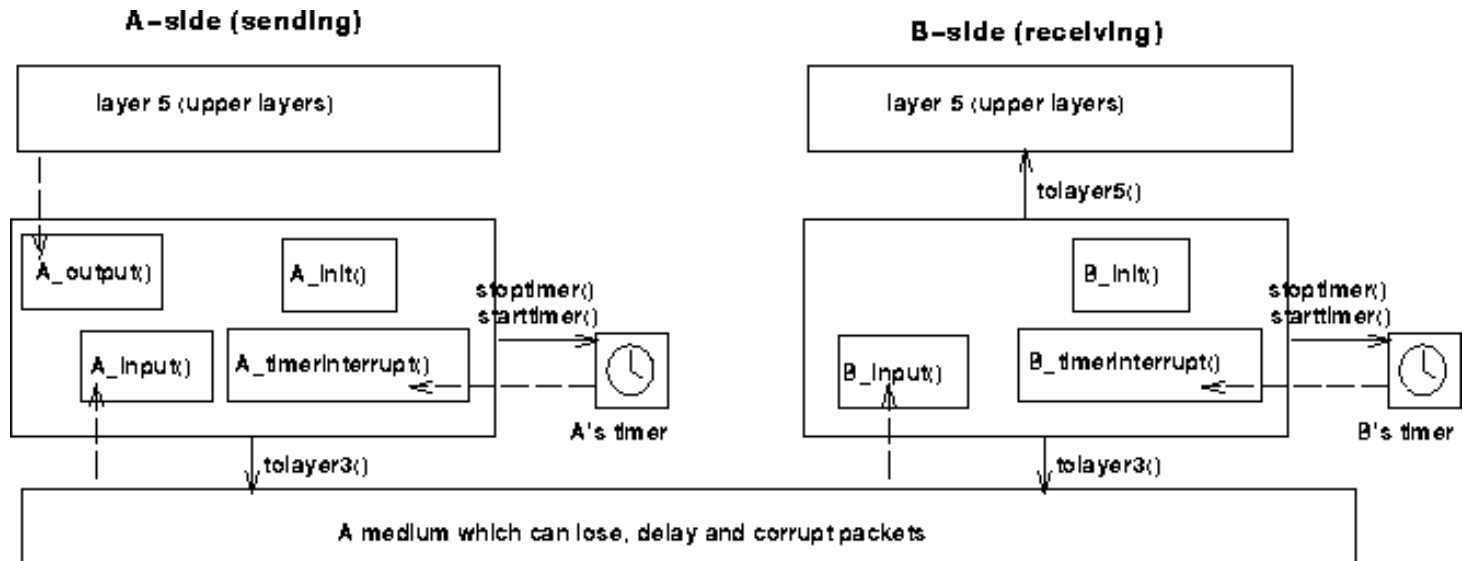


Lecture 5

Transport Layer: part II

Programming Assignment

- Implementing a Reliable Transport Protocol (Bidirectional Go-Back-N version)
 - Problem description: ProgAssn.pdf
 - Due 6/5 23:59:59 (late submission allowed up to 3 days with 10 pts reduction per day)
 - Upload your source file (w/ proper comments) and a design document to KLAS



Implementing Bidirectional GBN

- To write an application-level code simulating Bidirectional Go-Back-N (GBN) protocol
- Your code will have to execute in a simulated hardware/software environment
 - Event-driven simulation
 - Events occur at times determined stochastically by sim. env.
 - e.g. Programming interface from above and from below
 - e.g. Stopping/starting of timers
 - timer interrupts will cause your timer handling routine to be activated
- 8-bit or 16-bit 1's complement checksum operation with bit-wise addition operation is optional (for Extra Credit of 10 pts)
 - Decimal addition ('+') will not receive E.C.
- Piggybacking is required

Chapter 3: Transport Layer

Our goals:

- understand principles behind transport layer services:
 - multiplexing/demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- learn about transport layer protocols in the Internet:
 - UDP: connectionless transport
 - TCP: connection-oriented transport
 - TCP congestion control

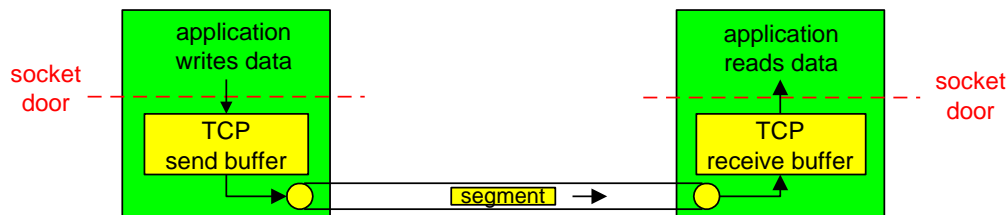
Chapter 3 outline

- 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

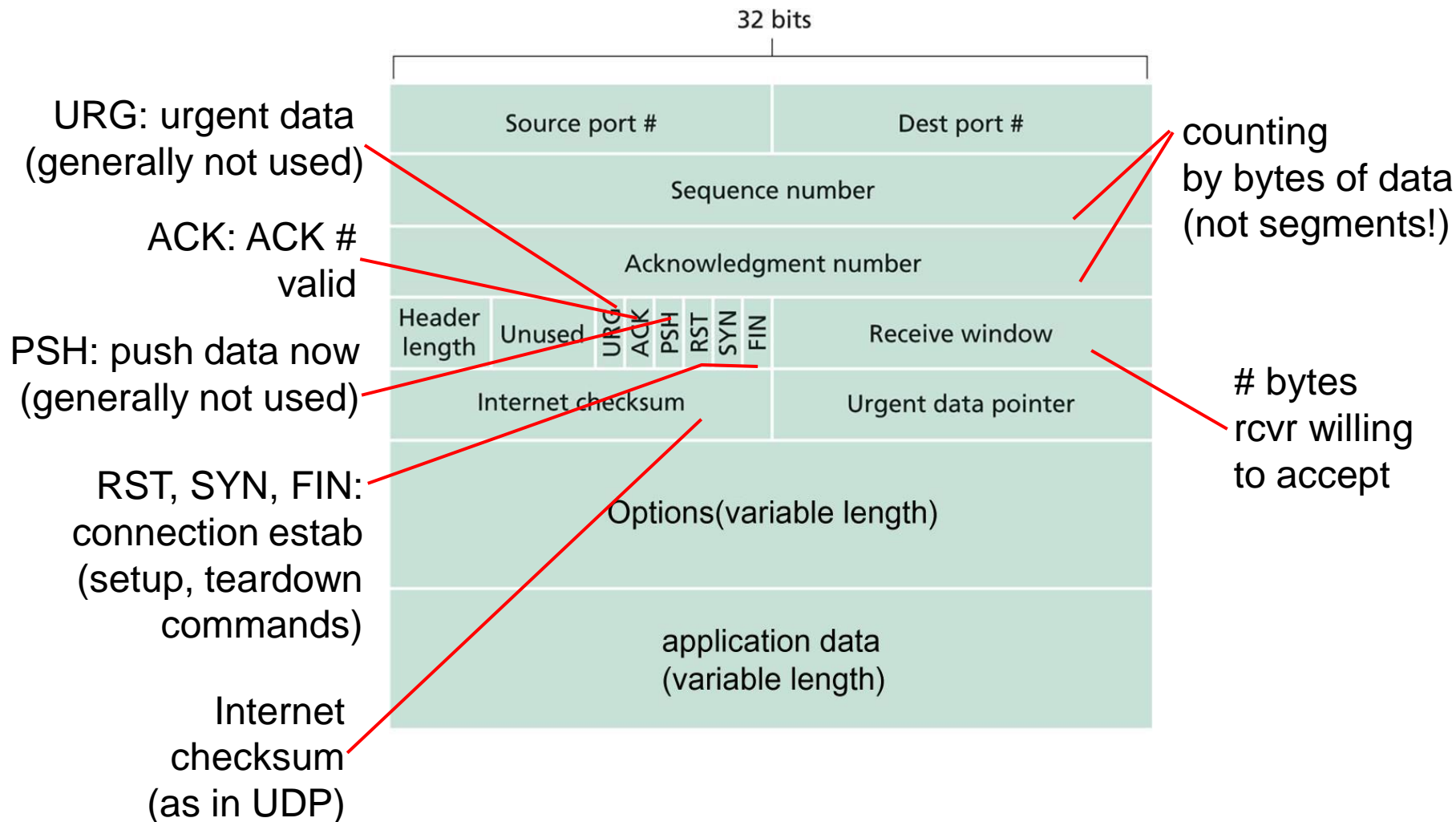
TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- **point-to-point:**
 - one sender, one receiver
- **reliable, in-order *byte stream*:**
 - no “message boundaries”
- **pipelined:**
 - TCP congestion and flow control set window size
- ***send & receive buffers***
- **full duplex data:**
 - bi-directional data flow in same connection
 - MSS: maximum segment size
- **connection-oriented:**
 - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- **flow controlled:**
 - sender will not overwhelm receiver



TCP segment structure



TCP seq. #'s and ACKs

Seq. #'s:

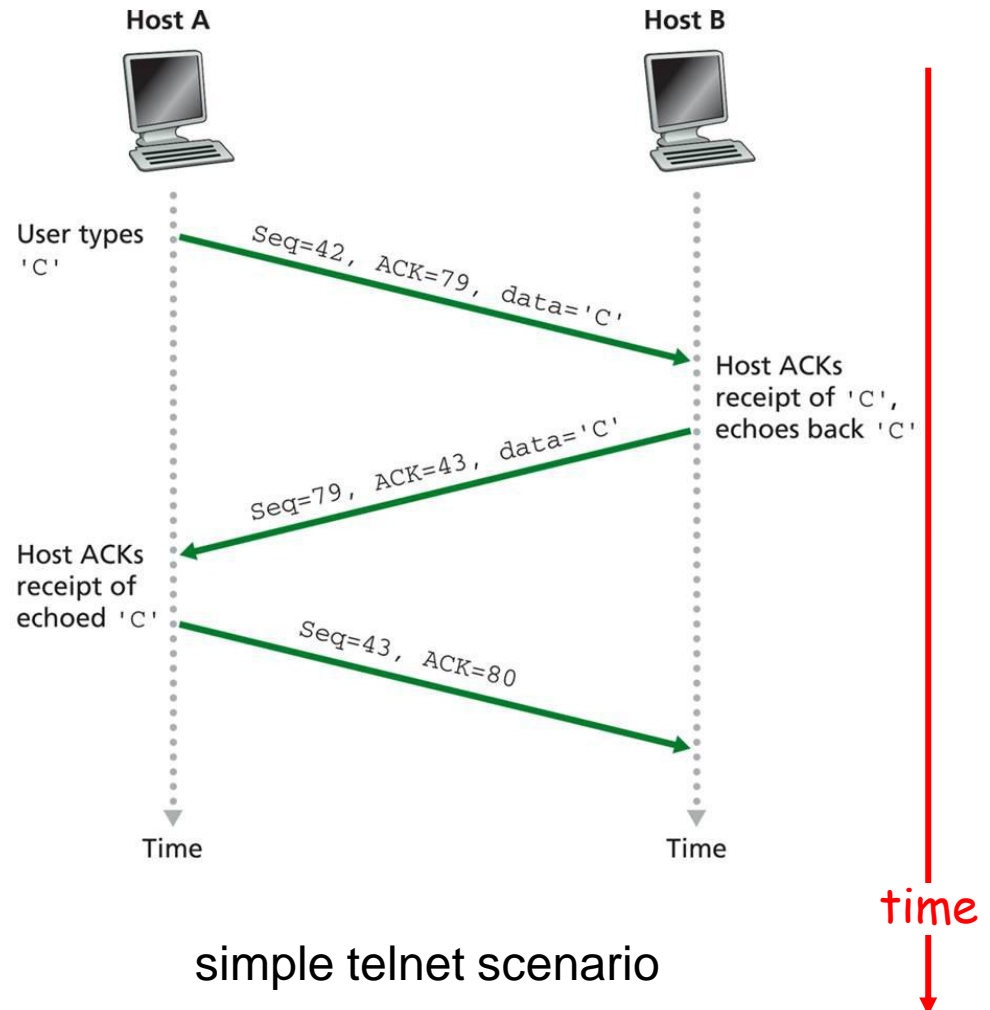
- byte stream “number” of first byte in segment’s data

ACKs:

- 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 Round Trip Time and 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**

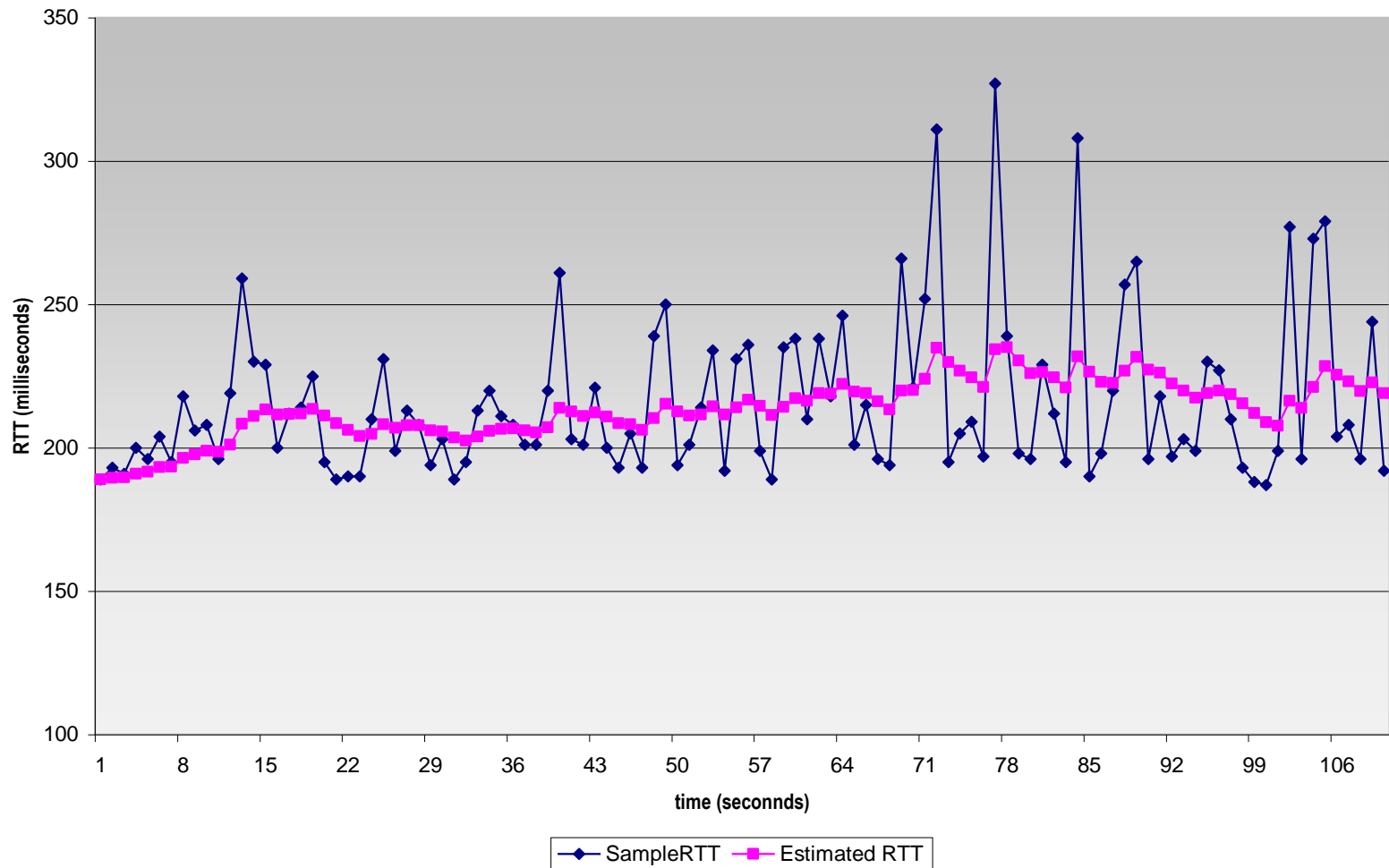
TCP Round Trip Time and Timeout

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

- Exponential weighted moving average
- influence of past sample decreases exponentially fast
- typical value: $\alpha = 0.125$

Example RTT estimation:

RTT: gaia.cs.umass.edu to fantasia.eurecom.fr



TCP Round Trip Time and Timeout

Setting the timeout

- **EstimatedRTT** plus “safety margin”
 - large variation in **EstimatedRTT** -> larger safety margin
- first estimate of how much **SampleRTT** deviates from **EstimatedRTT**:

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

(typically, $\beta = 0.25$)

Then set timeout interval:

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

TCP reliable data transfer

- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
 - timeout events
 - duplicate acks
- 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`

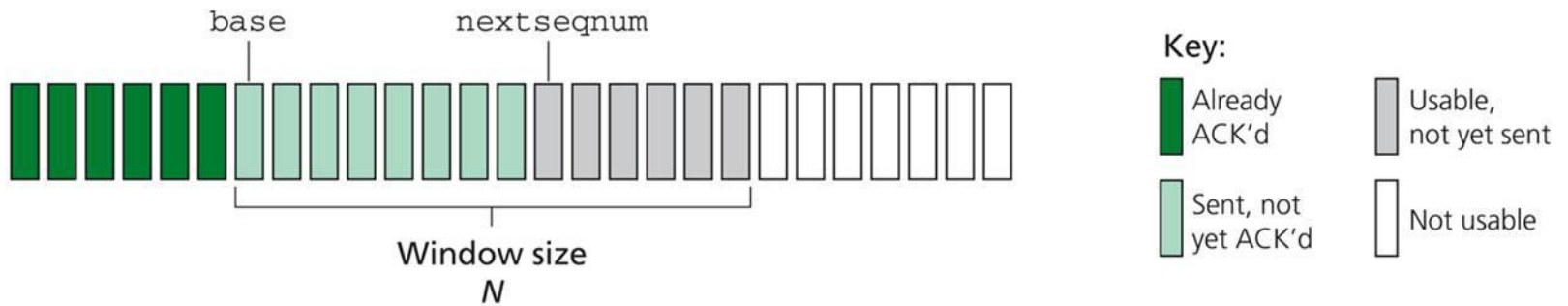
timeout:

- retransmit segment that caused timeout
- restart timer

Ack rcvd:

- If acknowledges previously unacked segments
 - update what is known to be acked
 - start timer if there are outstanding segments

TCP sender window



NextSeqNum = InitialSeqNum

SendBase = InitialSeqNum

```
loop (forever) {  
    switch(event)
```

```
    event: data received from application above  
        create TCP segment with sequence number NextSeqNum  
        if (timer currently not running)  
            start timer  
        pass segment to IP  
        NextSeqNum = NextSeqNum + length(data)
```

```
    event: timer timeout  
        retransmit not-yet-acknowledged segment with  
            smallest sequence number  
        start timer
```

```
    event: ACK received, with ACK field value of y  
        if (y > SendBase) {  
            SendBase = y  
            if (there are currently not-yet-acknowledged segments)  
                start timer  
        }
```

```
} /* end of loop forever */
```

TCP sender (simplified)

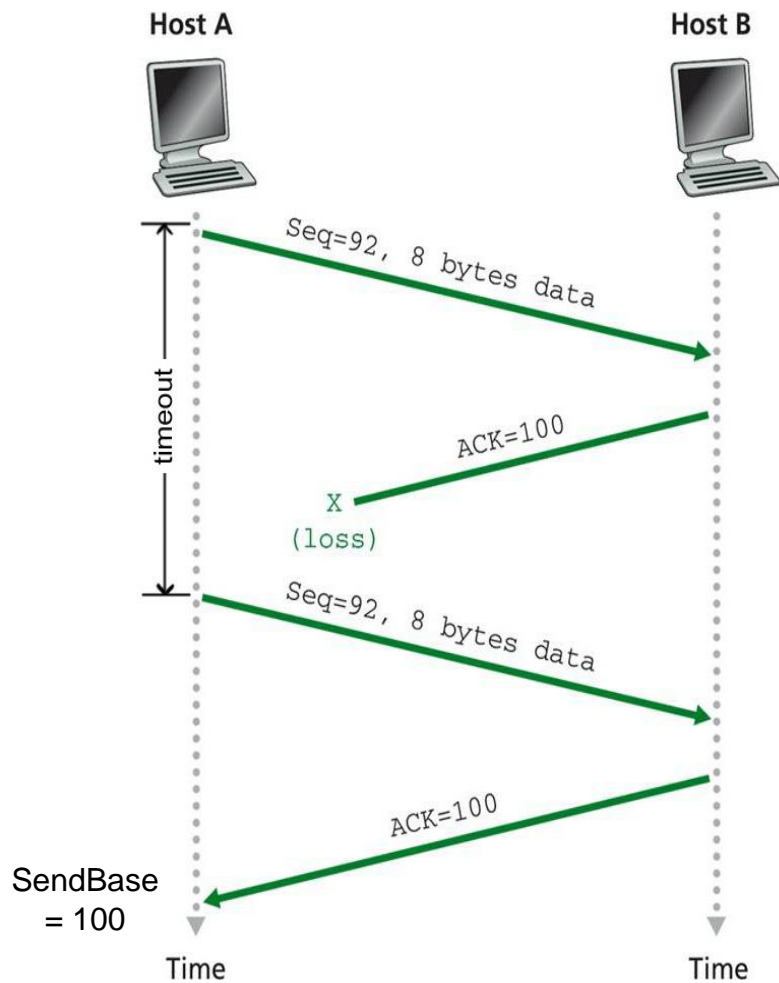
Comment:

- SendBase-1: last cumulatively ack'ed byte

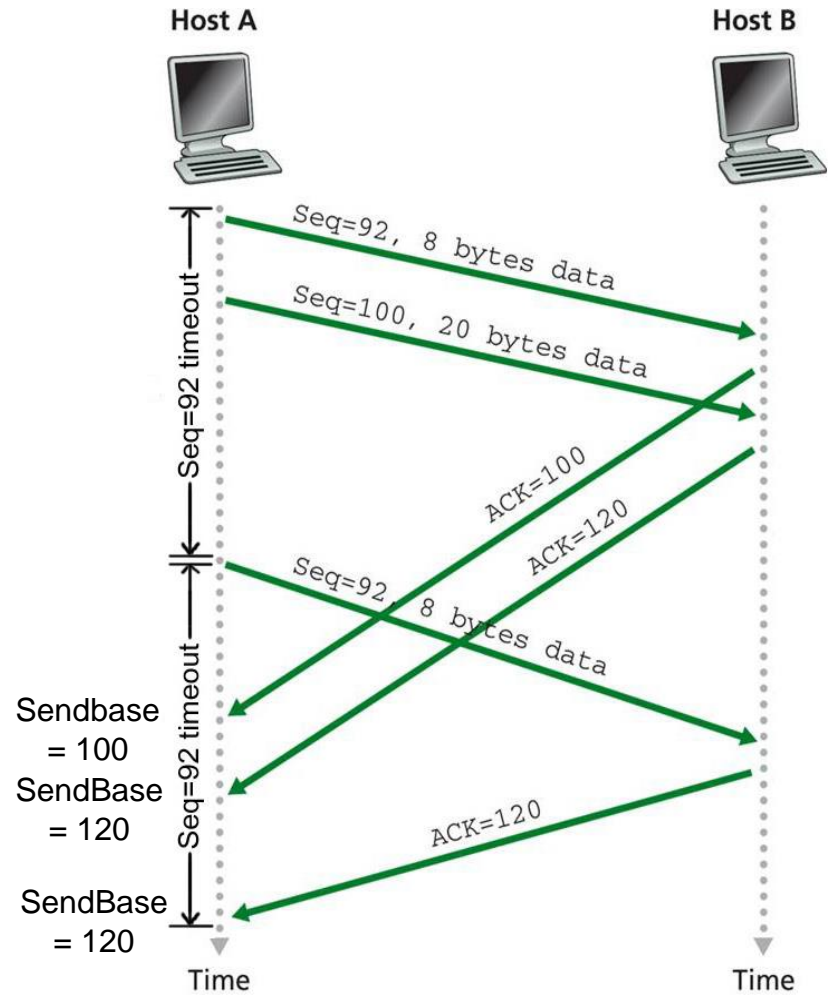
Example:

- SendBase-1 = 71;
y = 73, so the rcvr wants 73+ ;
y > SendBase, so that new data is acked

TCP: retransmission scenarios

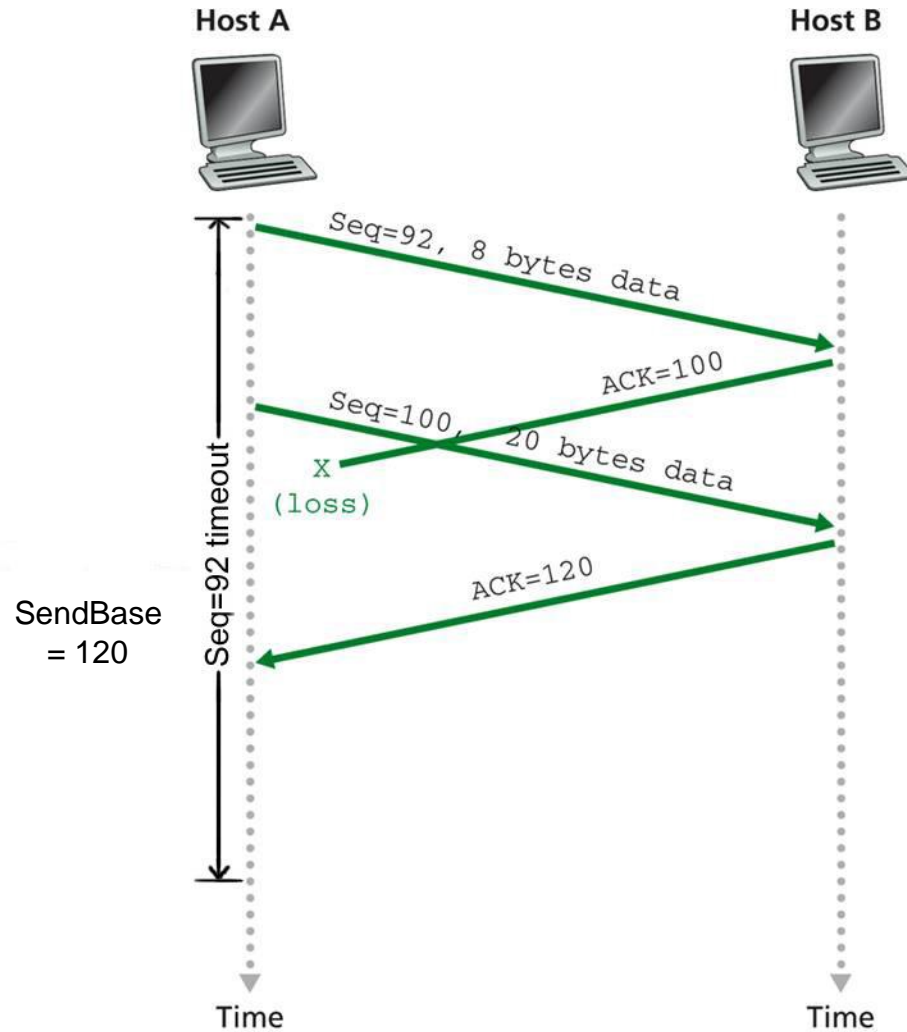


Lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



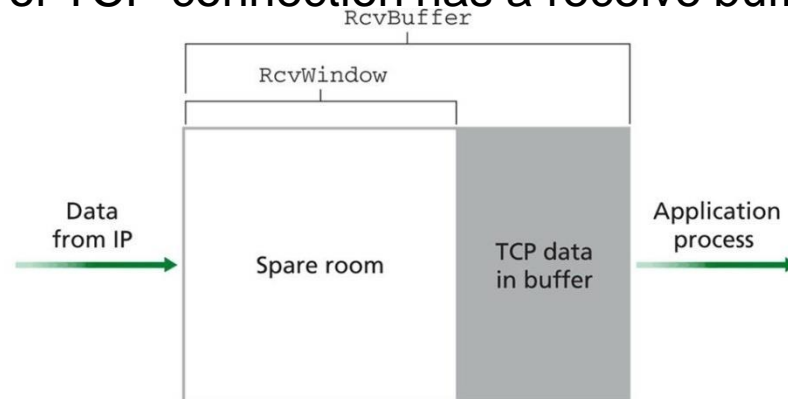
Cumulative ACK scenario

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 <i>duplicate ACK</i> , 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 Flow Control

- receive side of TCP connection has a receive buffer:

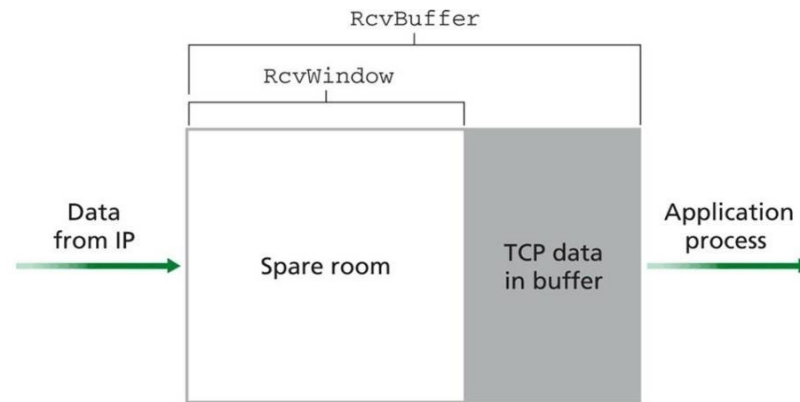


- app process may be slow at reading from buffer
- speed-matching service: matching the send rate to the receiving app's drain rate

flow control

sender won't overflow receiver's buffer by transmitting too much, too fast

TCP Flow control: how it works



(Suppose TCP receiver discards out-of-order segments)

- spare room in buffer

= **RcvWindow**

= **RcvBuffer - [LastByteRcvd - LastByteRead]**

- Rcvr advertises spare room by including value of **RcvWindow** in segments
- Sender limits unACKed data to **RcvWindow**
 - guarantees receive buffer doesn't overflow

TCP Connection Management

Recall: TCP sender, receiver establish "connection" before exchanging data segments

- initialize TCP variables:
 - seq. #s
 - buffers, flow control info (e.g. **RcvWindow**)

- *client*: connection initiator

```
Socket clientSocket = new  
    Socket("hostname", "port number");
```

- *server*: contacted by client

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

Three way handshake:

Step 1: client host sends TCP SYN segment to server

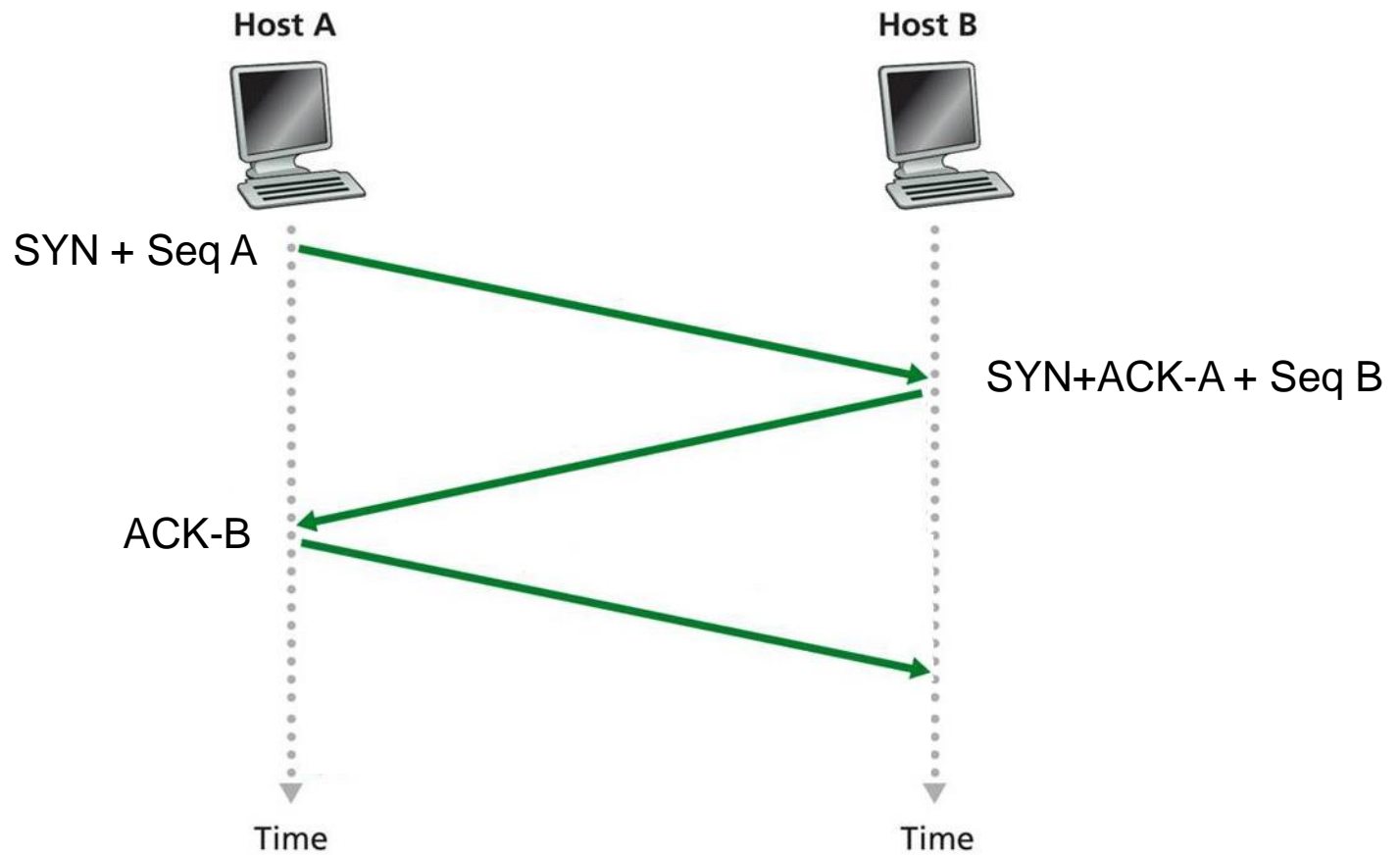
- specifies initial seq #
- no data

Step 2: server host receives SYN, replies with SYNACK segment

- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, replies with ACK segment, which may contain data

TCP Connection Management



TCP Connection Management (cont.)

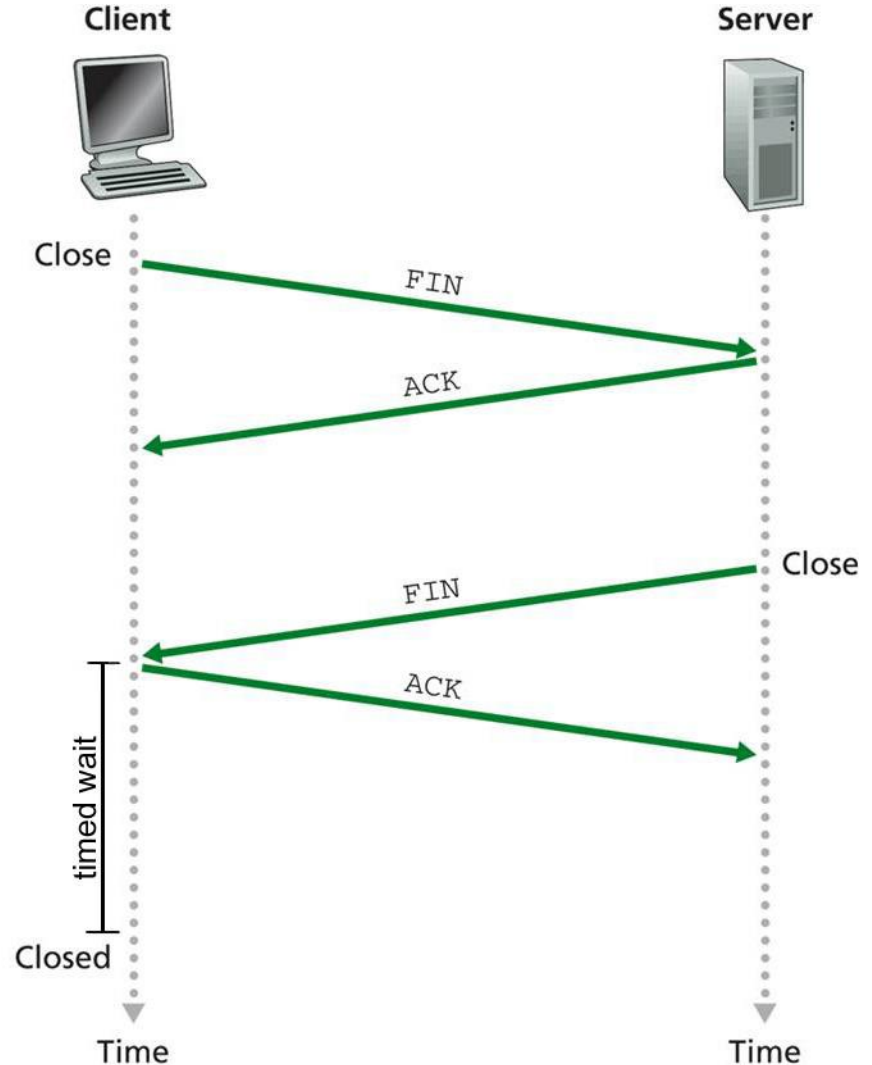
Closing a connection:

client closes socket:

```
clientSocket.close();
```

Step 1: client end system sends TCP
FIN control segment to server

Step 2: server receives FIN, replies
with ACK. Closes connection,
sends FIN.



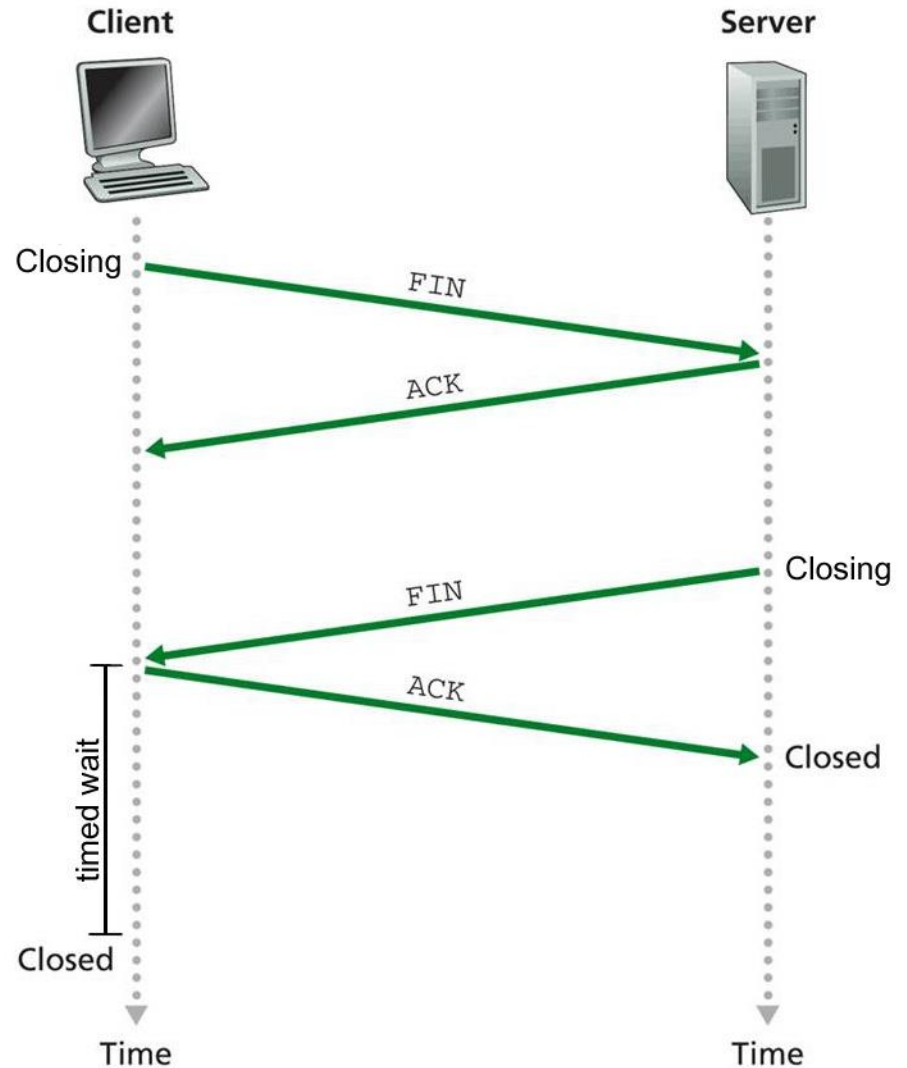
TCP Connection Management (cont.)

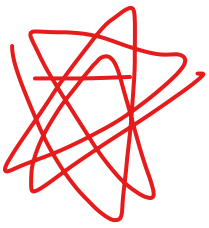
Step 3: client receives FIN, replies with ACK.

- Enters “timed wait” - will respond with ACK to received FINs

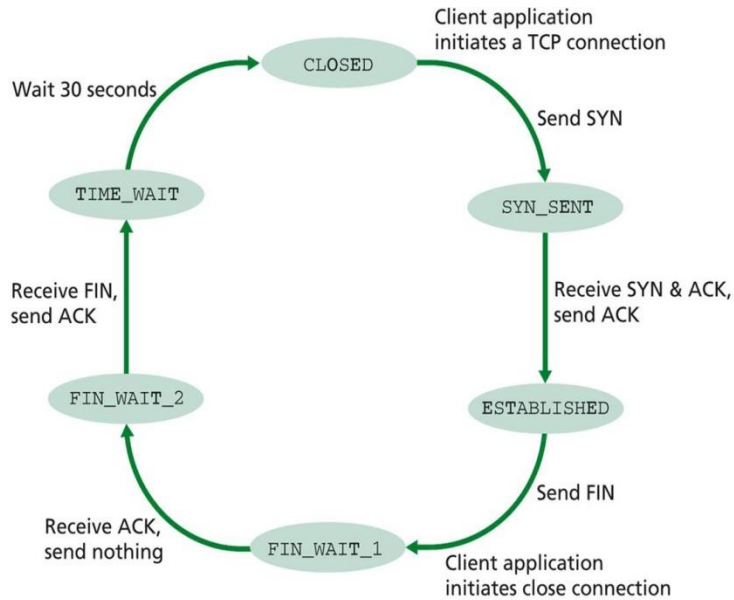
Step 4: server, receives ACK.
Connection closed.

Note: with small modification, can handle simultaneous FINs.



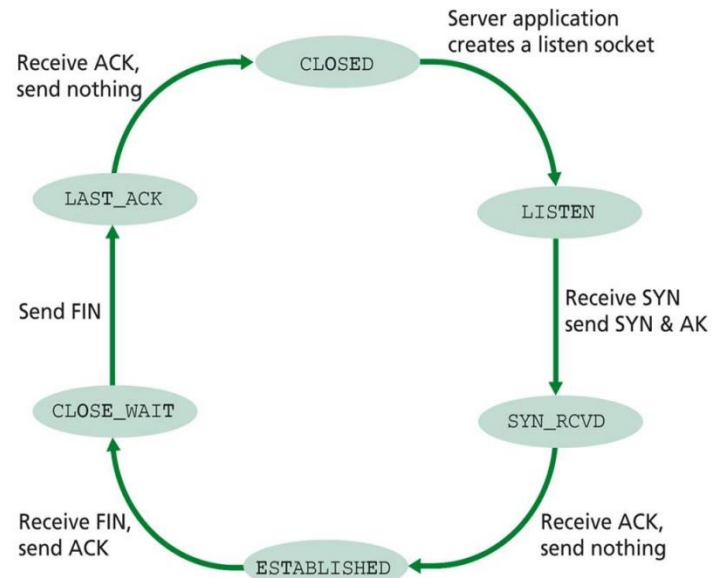


TCP Connection Management (cont)



TCP client lifecycle

TCP server lifecycle



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

Principles of Congestion Control

Congestion:

- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
 - lost packets (buffer overflow at routers)
 - long delays (queueing in router buffers)
- a top-10 problem!

Approaches towards congestion control

Two broad approaches towards congestion control:

End-end congestion control:

- no explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP

Network-assisted congestion control:

- routers provide feedback to end systems
 - single bit indicating congestion (SNA, DECbit, TCP/IP ECN, ATM)
 - explicit rate sender should send at

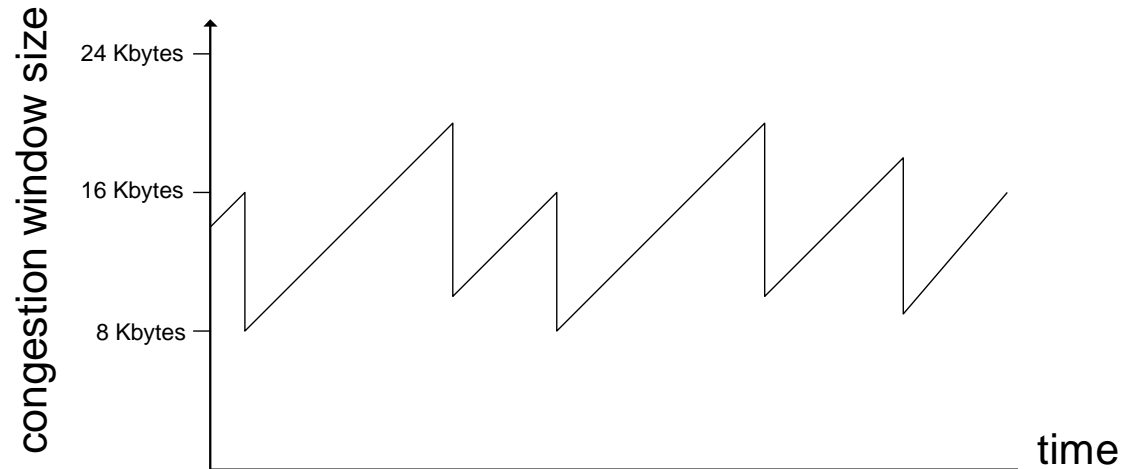
Chapter 3 outline

- 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 congestion control: additive increase, multiplicative decrease

- *Approach:* increase transmission rate (window size), probing for usable bandwidth, until loss occurs
 - *additive increase:* increase **CongWin** by 1 MSS every RTT until loss detected
 - *multiplicative decrease:* cut **CongWin** in half after loss

Saw tooth
behavior: probing
for bandwidth



TCP Congestion Control: details

- sender limits transmission:

$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{CongWin}$$

- Roughly,

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin** is dynamic, function of perceived network congestion

How does sender perceive congestion?

- loss event = timeout or 3 duplicate acks
- TCP sender reduces rate (**CongWin**) after loss event

three mechanisms:

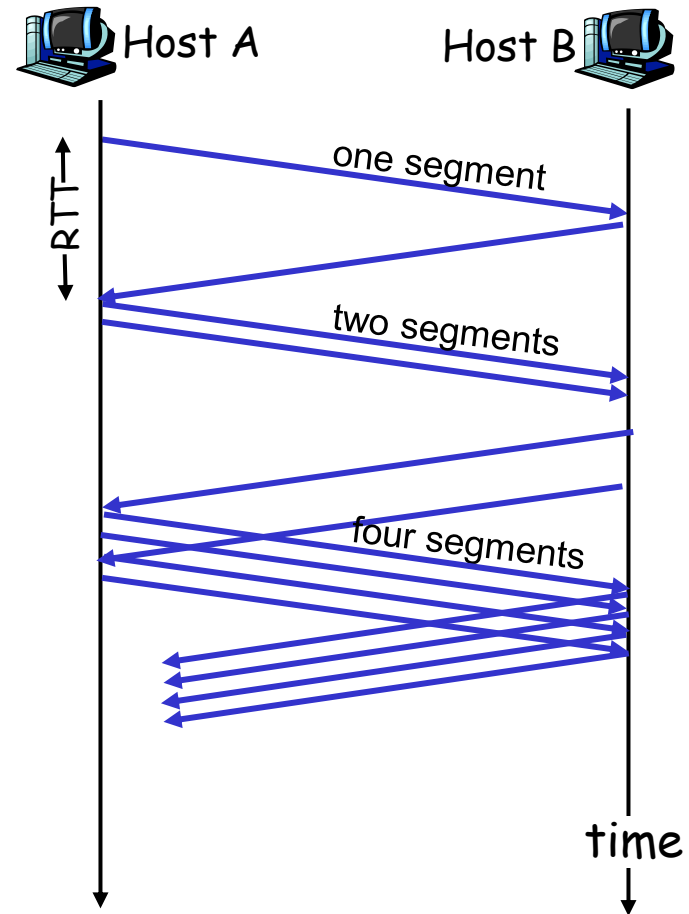
- AIMD
- slow start
- conservative after timeout events

TCP Slow Start

- When connection begins, **CongWin** = 1 MSS
 - Example: MSS = 500 bytes & RTT = 200 msec
 - initial rate = 20 kbps
- available bandwidth may be \gg MSS/RTT
 - desirable to quickly ramp up to respectable rate
- When connection begins, increase rate exponentially fast until first loss event

TCP Slow Start (more)

- When connection begins, increase rate exponentially until first loss event:
 - double **CongWin** every RTT
 - done by incrementing **CongWin** for every ACK received
- Summary: initial rate is slow but ramps up exponentially fast



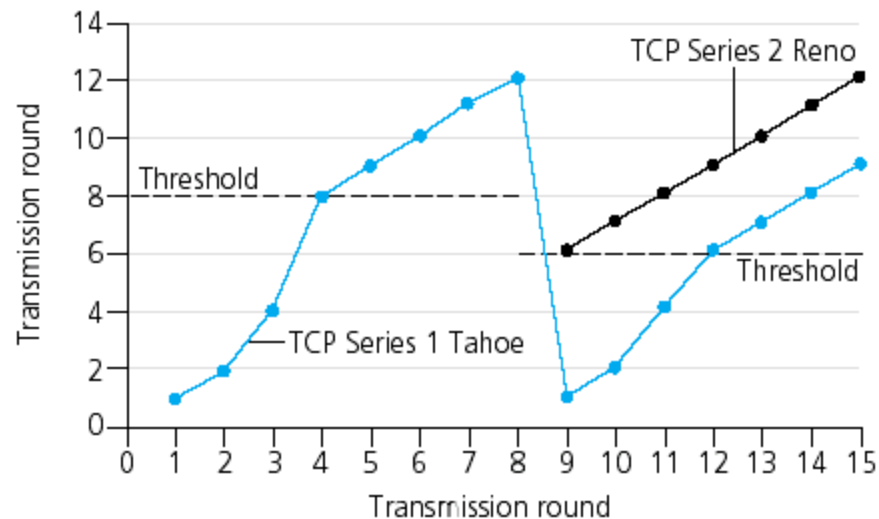
Refinement

Q: When should the exponential increase switch to linear?

A: When **CongWin** gets to 1/2 of its value before timeout.

Implementation:

- Variable Threshold
- At loss event, Threshold is set to 1/2 of CongWin just before loss event



Refinement: inferring loss

- After 3 dup ACKs:
 - **CongWin** is cut in half
 - window then grows linearly
- But after timeout event:
 - **CongWin** instead set to 1 MSS;
 - window then grows exponentially
 - to a threshold, then grows linearly

Philosophy:

- ❑ 3 dup ACKs indicates network capable of delivering some segments
- ❑ timeout indicates a "more alarming" congestion scenario

Summary: TCP Congestion Control

- When **CongWin** is below **Threshold**, sender in **slow-start** phase, window grows exponentially.
- When **CongWin** is above **Threshold**, sender is in **congestion-avoidance** phase, window grows linearly.
- When a **triple duplicate ACK** occurs, **Threshold** set to $\text{CongWin}/2$ and **CongWin** set to **Threshold**.
- When **timeout** occurs, **Threshold** set to $\text{CongWin}/2$ and **CongWin** is set to 1 MSS.

Chapter 3: Summary

- principles behind transport layer services:
 - multiplexing, demultiplexing
 - reliable data transfer
 - flow control
 - congestion control
- instantiation and implementation in the Internet
 - UDP
 - TCP