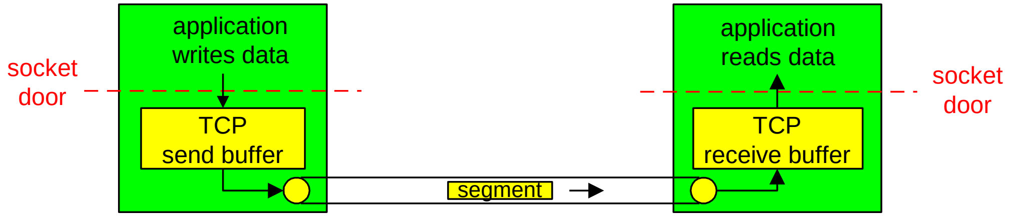
TCP: Overview

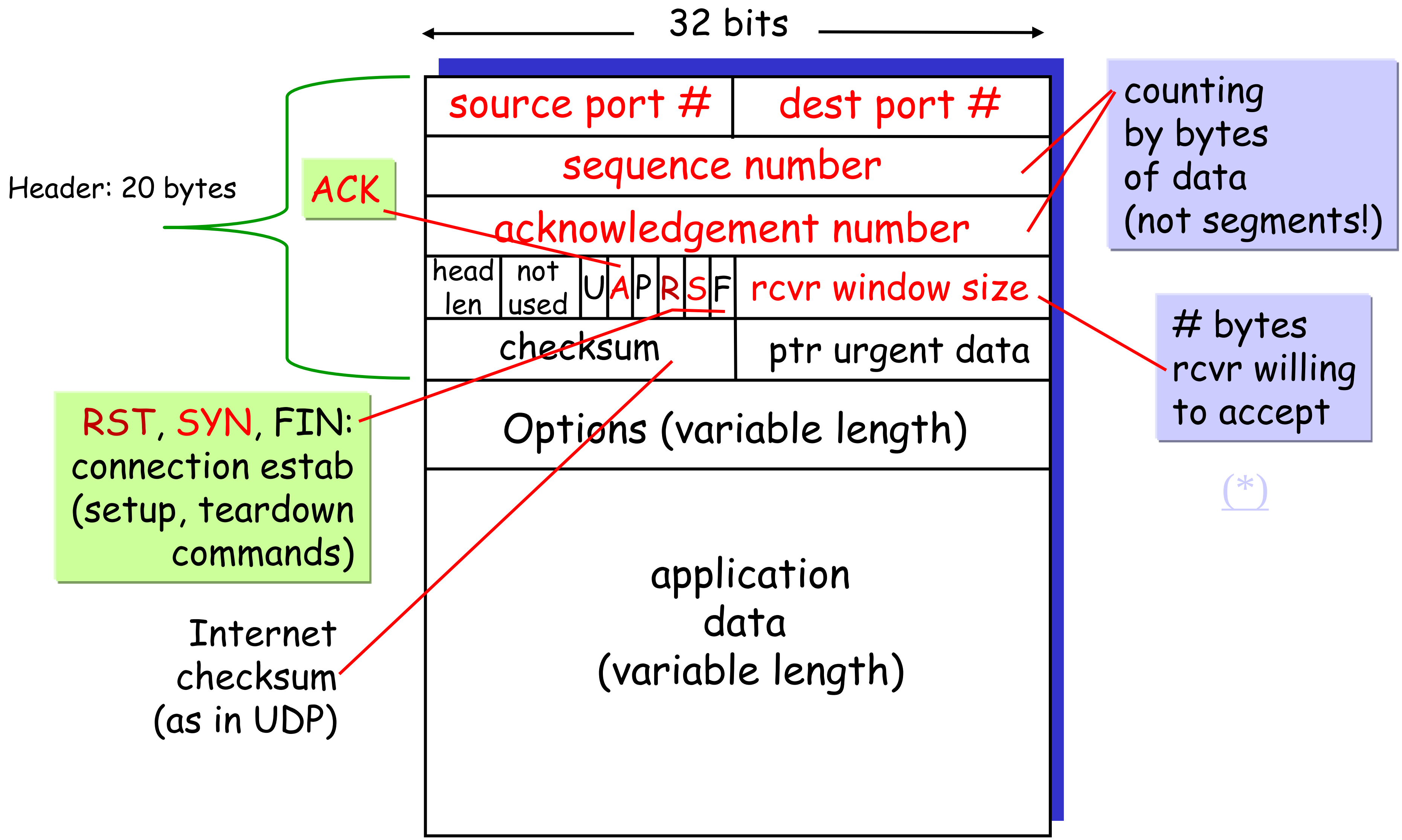
RFCs: 793, 1122, 1323, 2018, 2581

- 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*

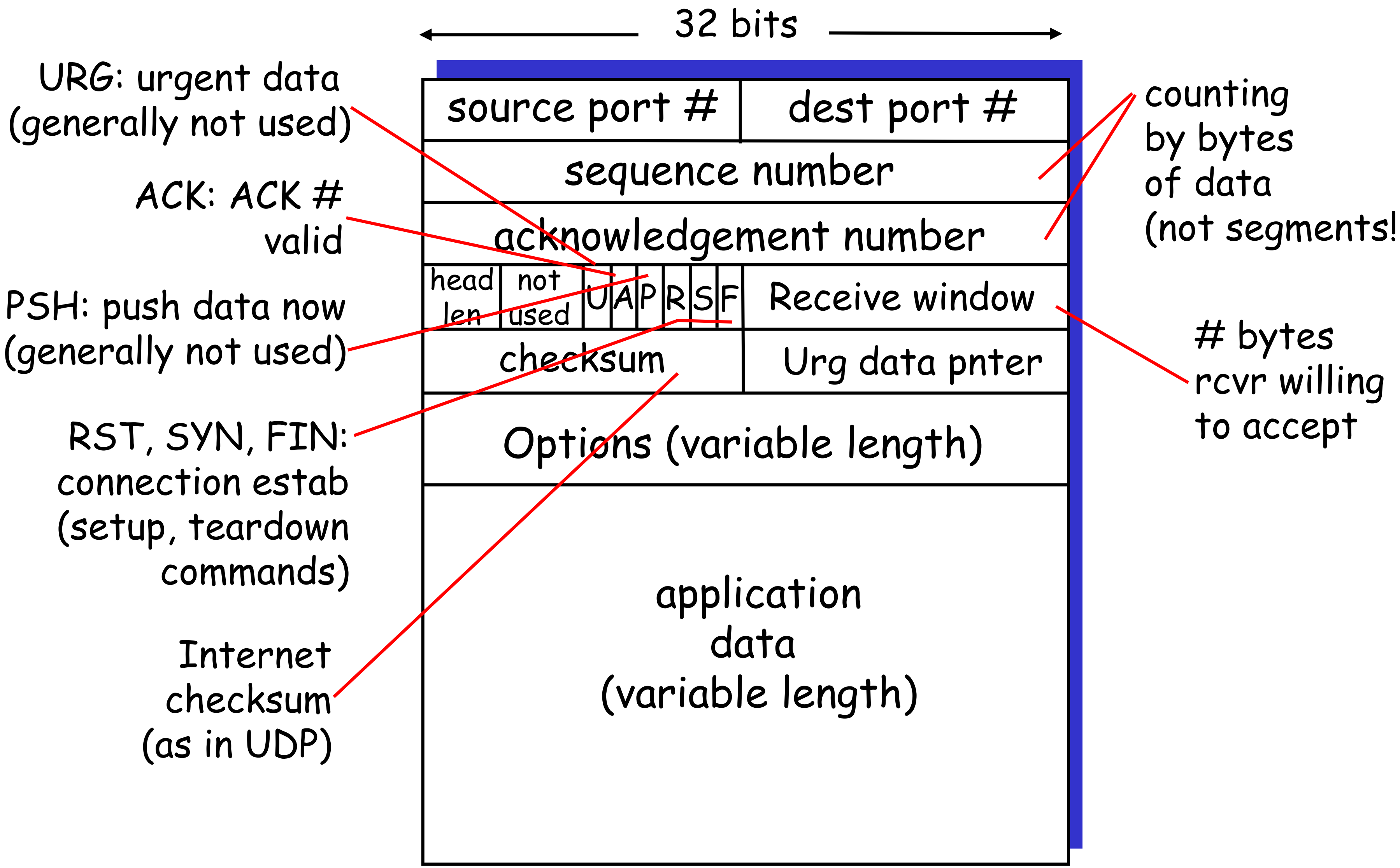


- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP segment structure



TCP Connection Management

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

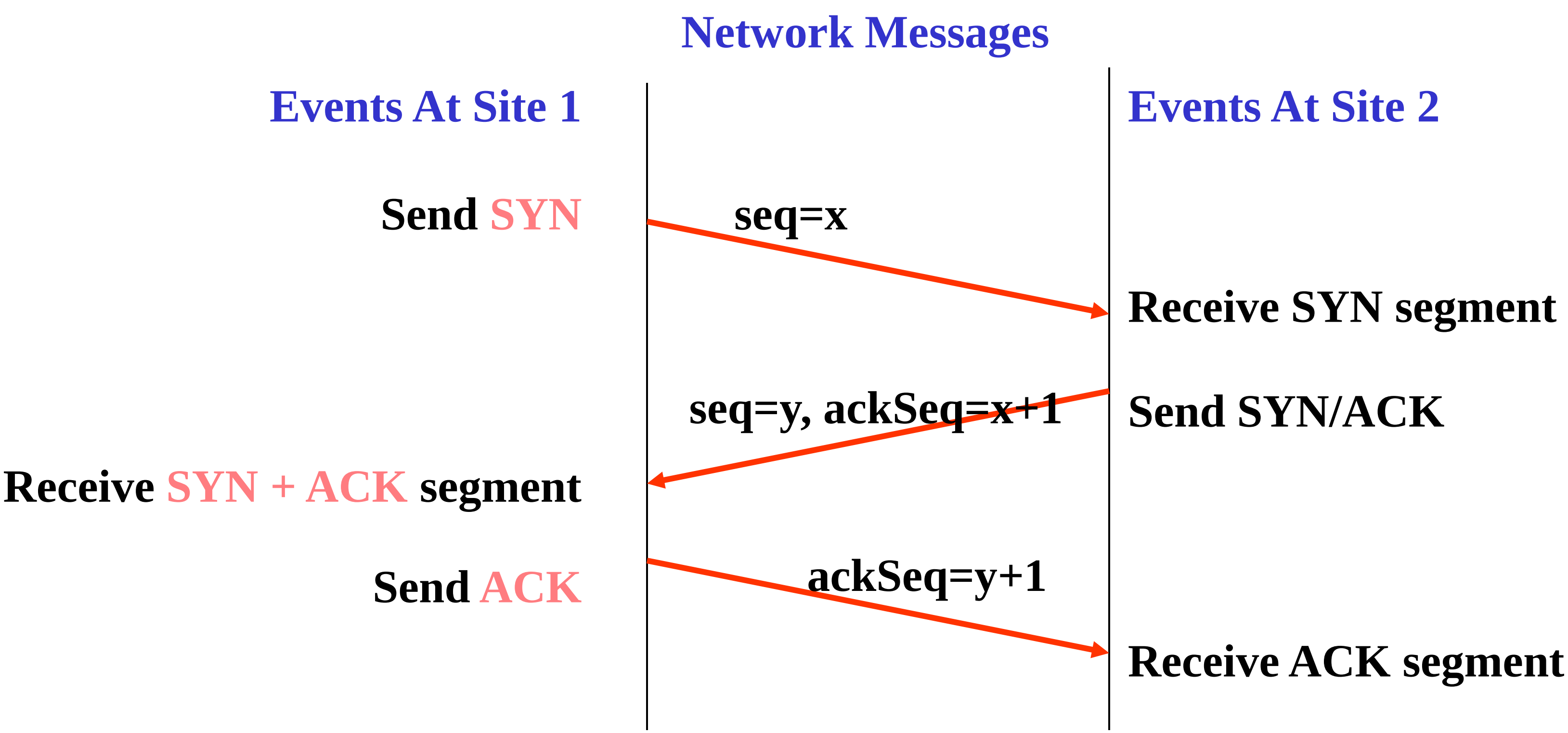
- initialize TCP variables:
  - seq. #s
  - buffers, flow control info (e.g. `RcvWindow`)

Three-way handshake:

Step 1: client end system sends TCP SYN control segment to server
- specifies initial seq #

Step 2: server end system receives SYN, replies with SYN/ACK control segment
- ACKs received SYN
- allocates buffers
- specifies server's receiving buffer initial seq. #

Connection Establishment using Three-Way Handshake

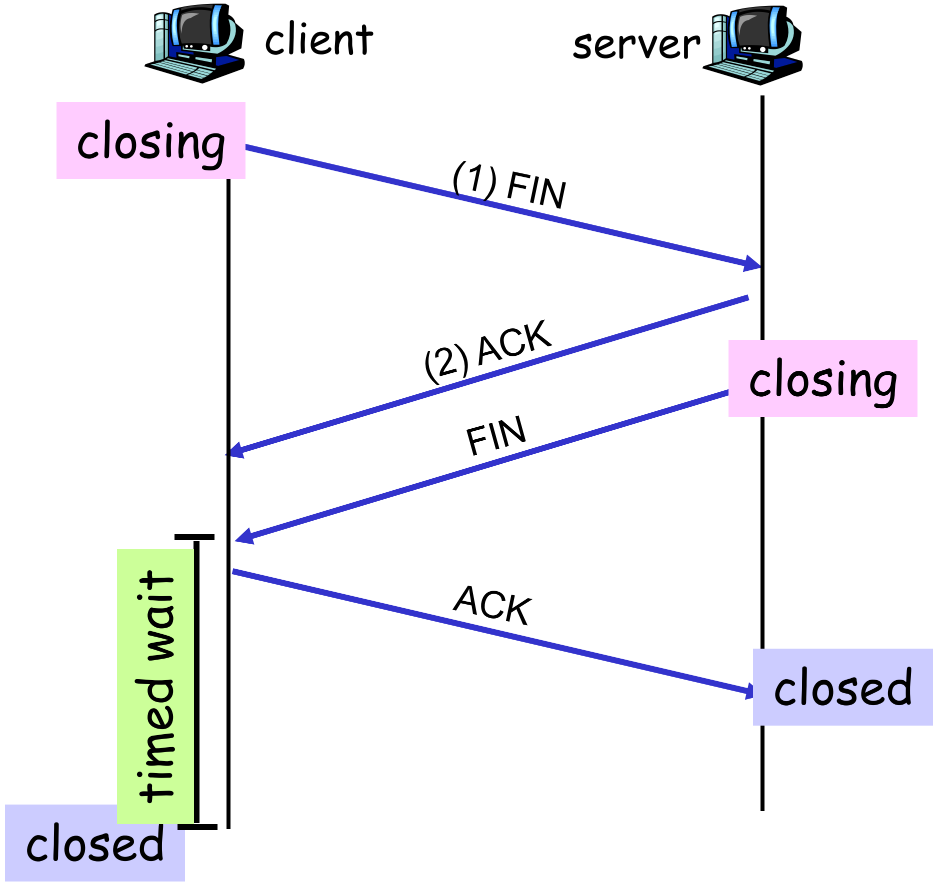


TCP Connection Management (cont.)

Closing a connection:

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

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



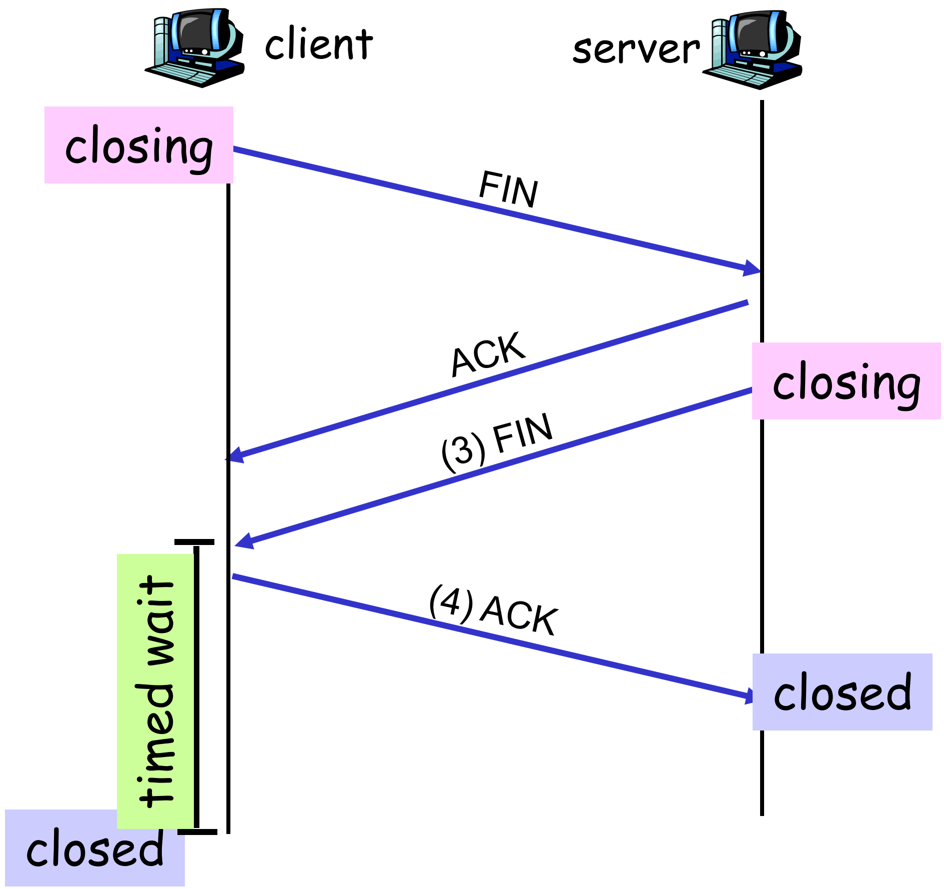
TCP Connection Management (cont.)

Step 3: client receives FIN, replies with ACK.

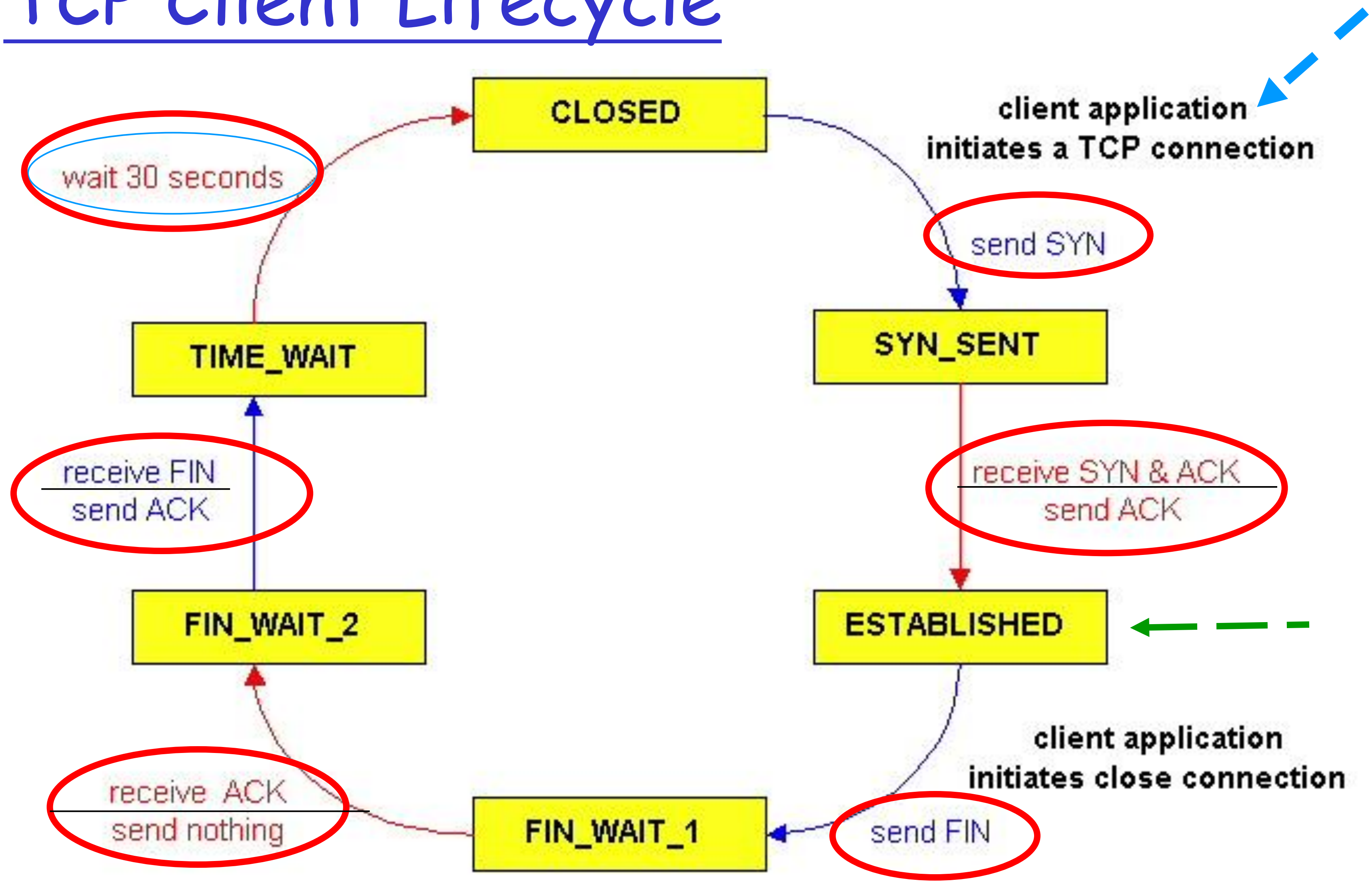
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

Note: with small modification, can handle simultaneous FINs.



TCP Client Lifecycle

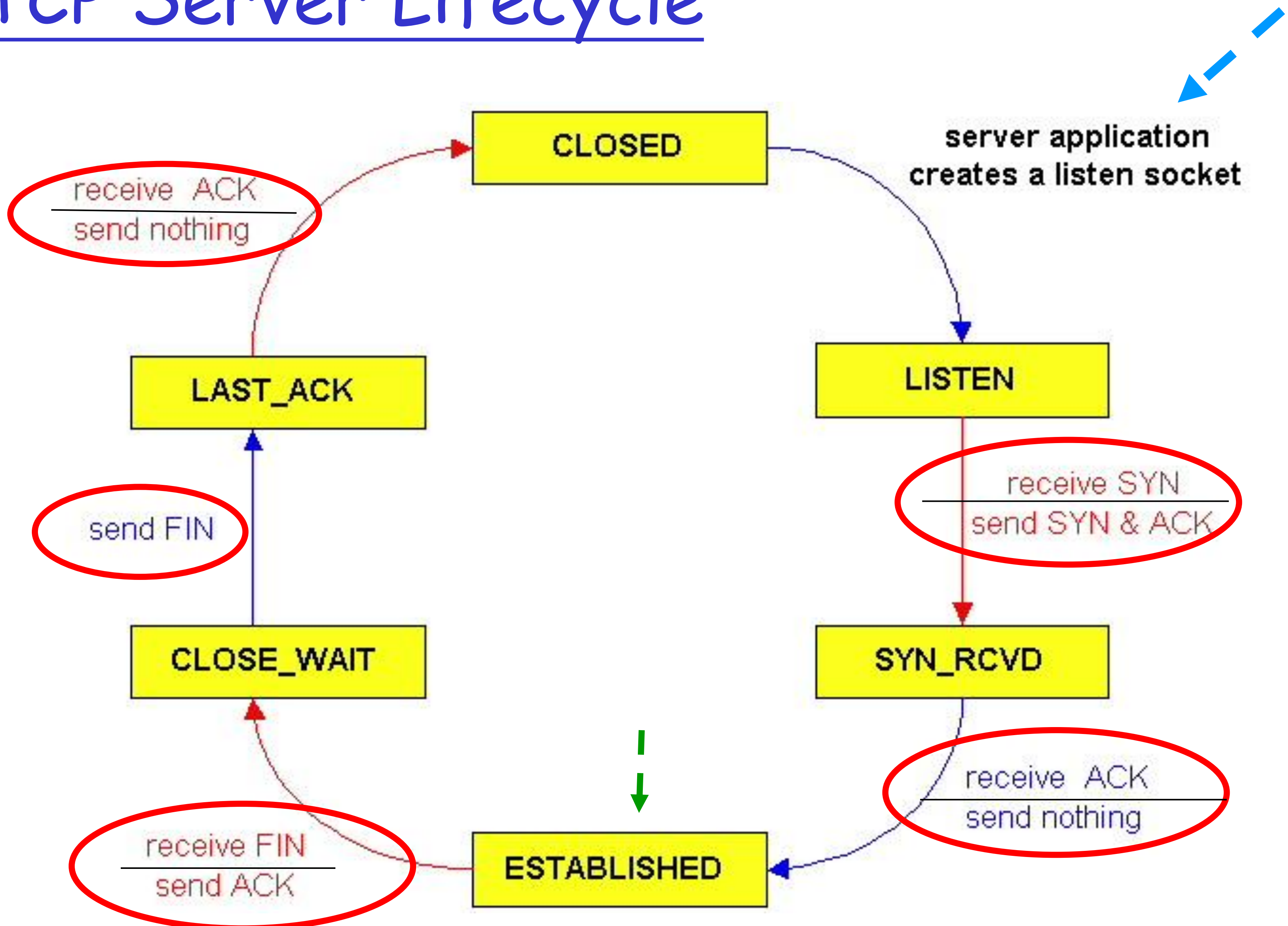


TCP Client Lifecycle (cont'd)

- Why need to wait for 30 seconds more?

- After received a FIN, client acknowledges and enters the TIME_WAIT state.
- In the state, client resend the final ACK in case the ACK is lost.
- The time spent in the state is implementation-dependent.
- The typical value is 30 seconds

TCP Server Lifecycle



TCP seq. #'s and ACKs

Seq. #'s:

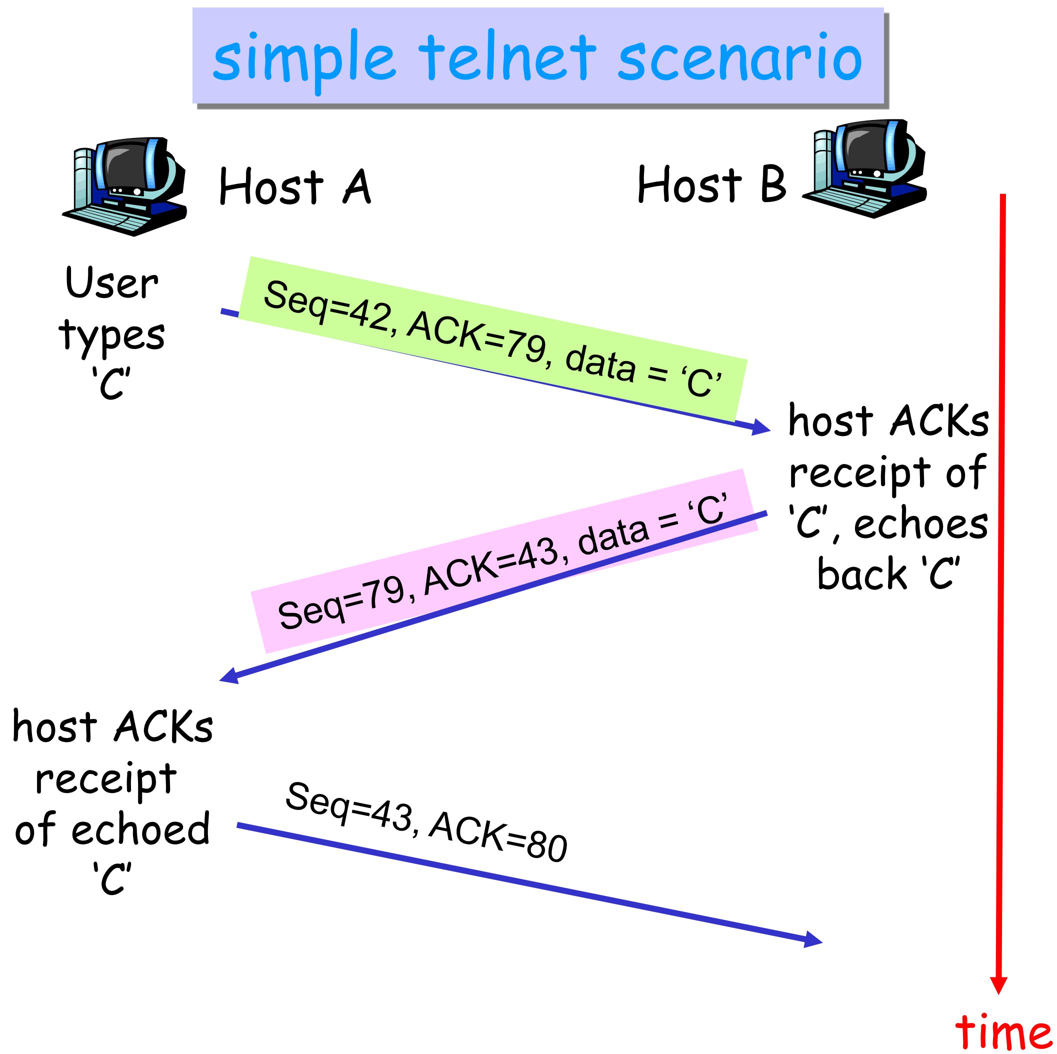
- byte stream "number" of first byte in segment's data

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

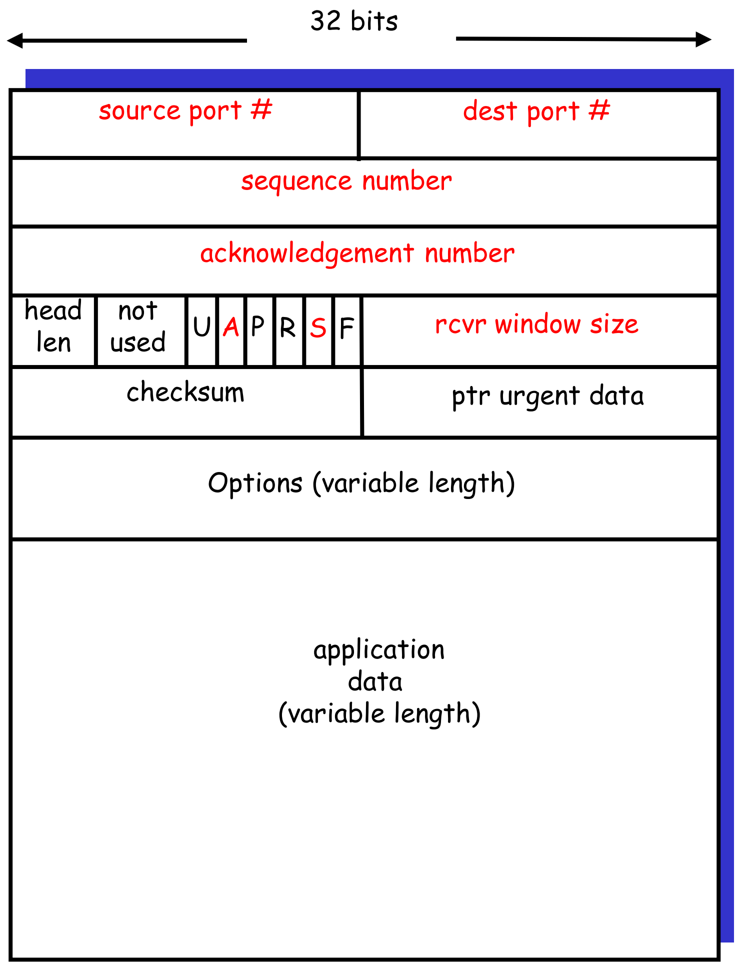


Initial Sequence Numbers

- At connection establishment phase, two sites agree on <u>initial</u> sequence numbers.
- Initial sequence number is chosen at random.

"Window Advertisement" by the Receiver

- Specify how many additional bytes of data the receiver is prepared to accept.
- Reflect receiver's current buffer size
- Sender adjusts its sliding windows size accordingly



Ports, Connections and Endpoints

- TCP uses ***protocol port*** numbers to identify the ultimate destination (processes) within a machine.
  - A port number is an integer number.

- TCP uses the ***connection*** (not the protocol port) as the fundamental abstraction.

Ports, Connections and Endpoints (cont'd)

- A TCP connection is identified by a pair of ***endpoints***.
- An endpoint is a pair of integers (host_IP address, TCP_port#)
  - e.g., endpoint (128.10.2.3, 21) specifies TCP port 21 on the machine 128.10.2.3 for "ftp" service
  - connections:
    - **(140.112.181.69, 1504) and (128.10.2.3, 21);**
    - **(192.56.132.8, 1184) and (128.10.2.3, 21),...**

Decimal	Keyword	UNIX Keyword	Description
0			Reserved
1	TCPMUX	-	TCP Multiplexor
5	RJE	-	Remote Job Entry
7	ECHO	echo	Echo
9	DISCARD	discard	Discard
11	USERS	systat	Active Users
13	DAYTIME	daytime	Daytime
15	-	netstat	Network status program
17	QUOTE	qotd	Quote of the Day
19	CHARGEN	chargen	Character Generator
20	FTP-DATA	ftp-data	File Transfer Protocol (data)
21	FTP	ftp	File Transfer Protocol
23	TELNET	telnet	Terminal Connection
25	SMTP	smtp	Simple Mail Transport Protocol
37	TIME	time	Time
42	NAMESERVER	name	Host Name Server
43	NICNAME	whois	Who Is
53	DOMAIN	nameserver	Domain Name Server
77	-	rje	any private RJE service
79	FINGER	finger	Finger
93	DCP	-	Device Control Protocol
95	SUPDUP	supdup	SUPDUP Protocol
101	HOSTNAME	hostnames	NIC Host Name Server
102	ISO-TSAP	iso-tsap	ISO-TSAP
103	X400	x400	X.400 Mail Service
104	X400-SND	x400-snd	X.400 Mail Sending
111	SUNRPC	sunrpc	SUN Remote Procedure Call
113	AUTH	auth	Authentication Service
117	UUCP-PATH	uucp-path	UUCP Path Service
119	NNTP	nntp	USENET News Transfer Protocol
129	PWDGEN	-	Password Generator Protocol
139	NETBIOS-SSN	-	NETBIOS Session Service
160-223	Reserved		

Figure 12.14 Examples of currently assigned TCP port numbers. To the extent possible, protocols like UDP use the same numbers.

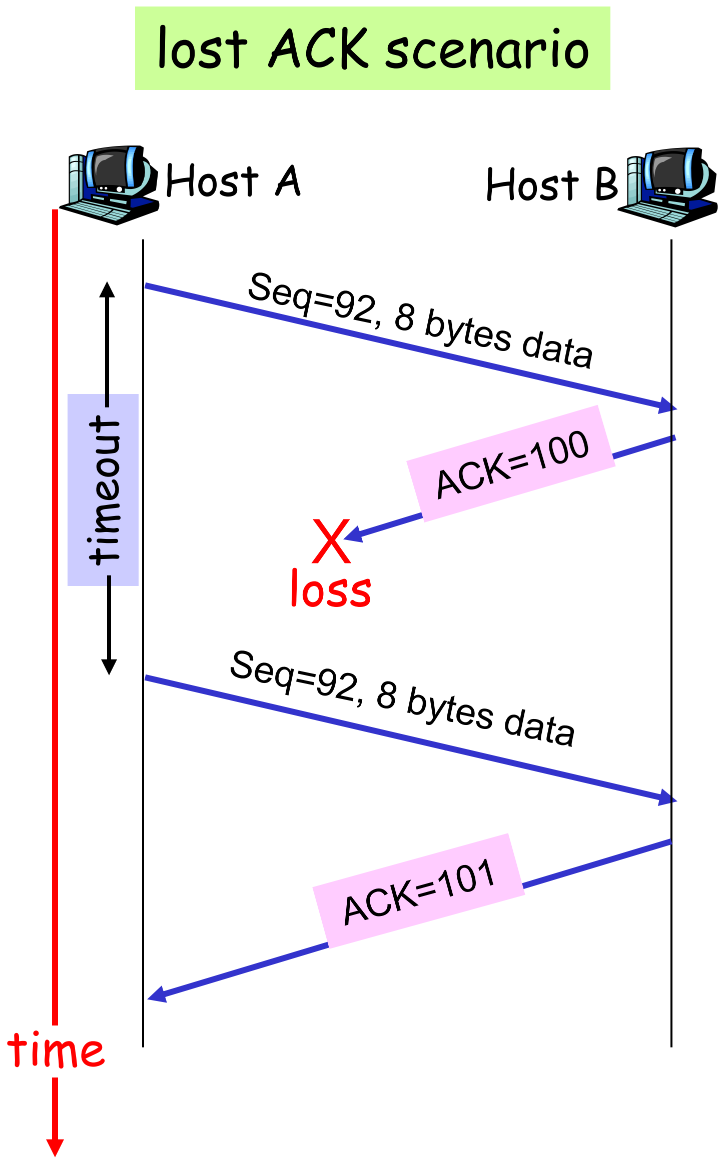
TCP reliable data transfer

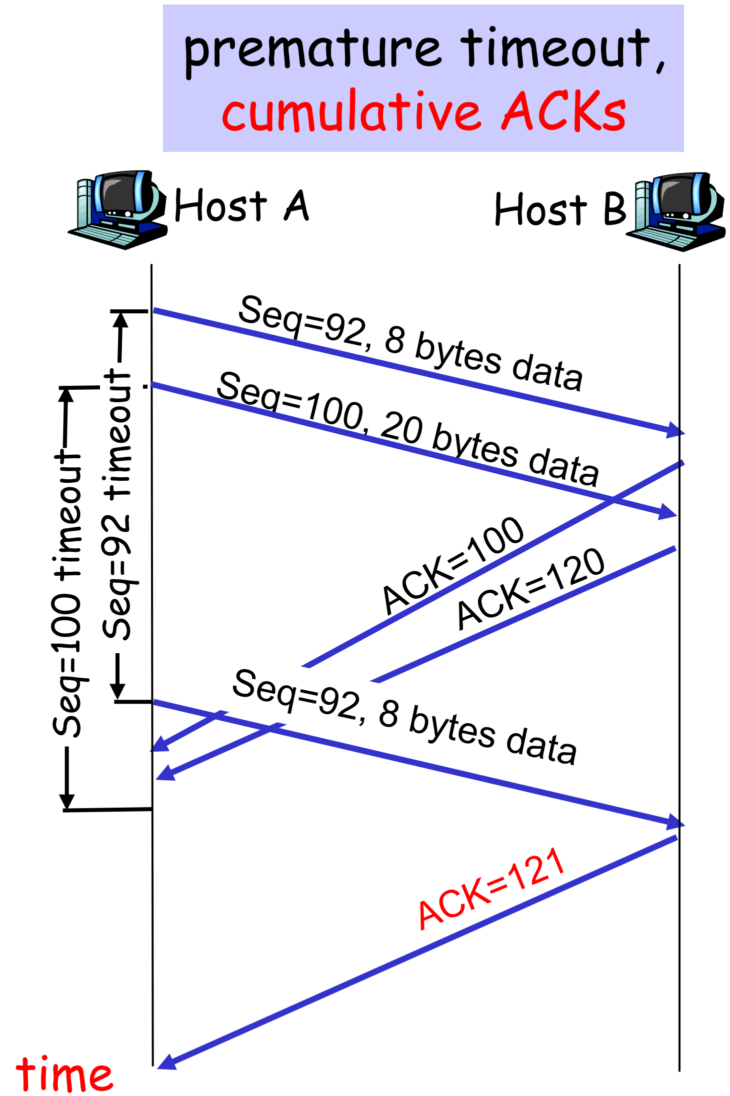
- TCP creates rdt service on top of IP's unreliable service
- Pipelined segments
- Cumulative acks
- TCP uses single retransmission timer
- Retransmissions are triggered by:
  - timeout events
  - duplicate acks

TCP ACK generation [RFC 1122, RFC 2581]

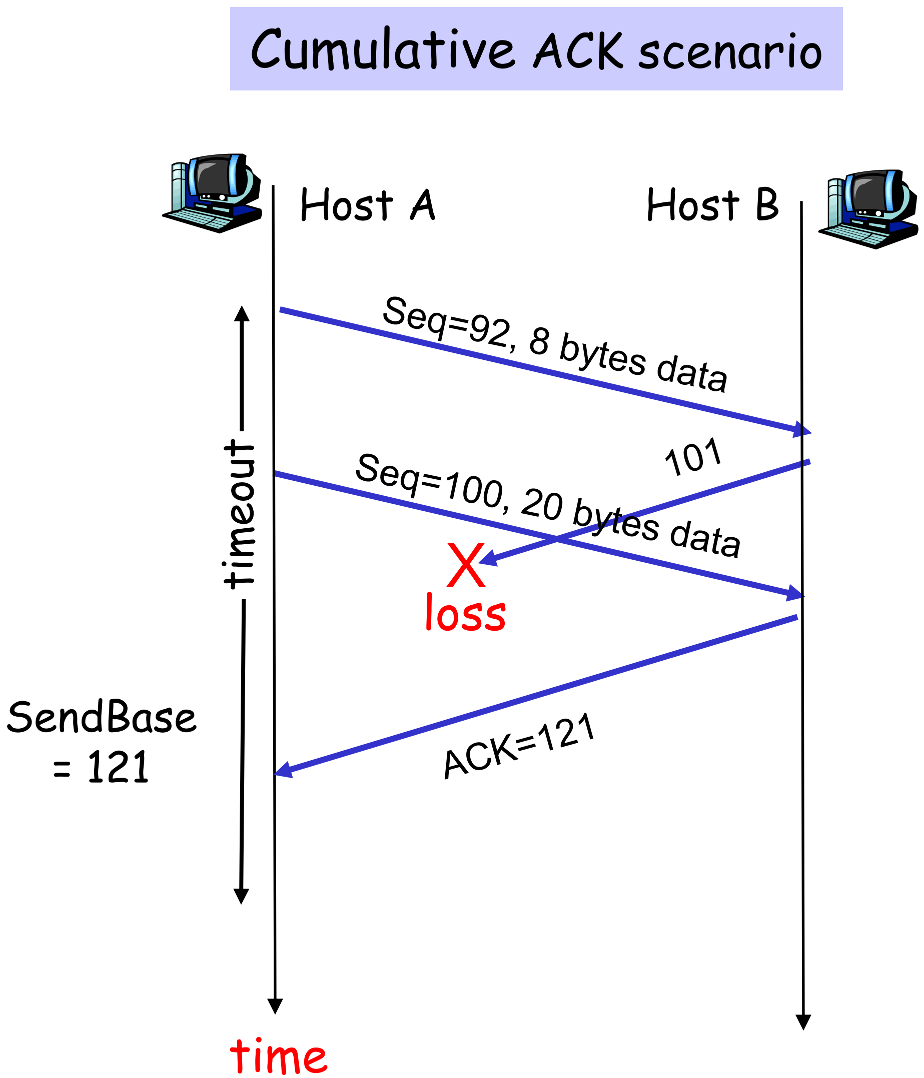
Event	TCP Receiver action
in-order segment arrival, no gaps, everything else already ACKed	delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
in-order segment arrival, no gaps, one delayed ACK pending	immediately send single cumulative ACK
out-of-order segment arrival higher-than-expect seq. # gap detected	send duplicate ACK, indicating seq. # of next expected byte
arrival of segment that partially or completely fills gap	immediate ACK if segment starts at lower end of gap

TCP: retransmission scenarios





TCP retransmission scenarios (more)



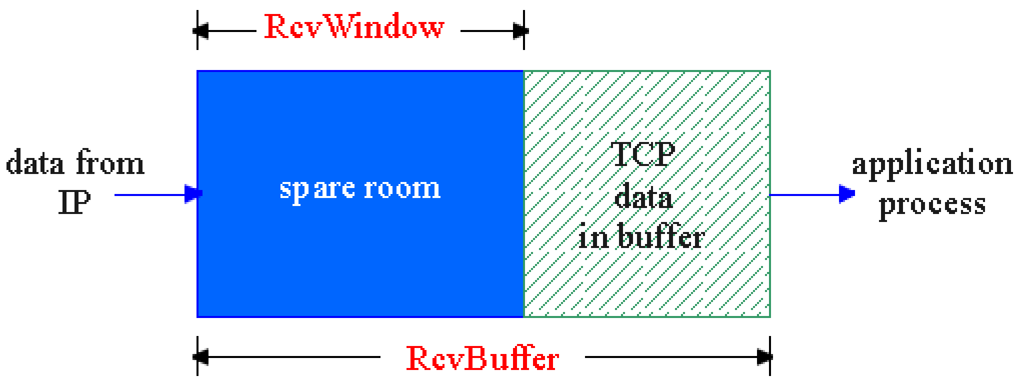
TCP Flow Control

flow control

sender won't overrun receiver's buffers by transmitting too much, too fast

`RcvBuffer` = size or TCP Receive Buffer

`RcvWindow` = amount of spare room in Buffer



receiver buffering

receiver: explicitly informs sender of (dynamically changing) amount of free buffer space

- `RcvWindow` field in TCP segment (*)

sender: keeps the amount of transmitted, unACKed data less than most recently received `RcvWindow`

TCP Round Trip Time and Timeout

Q: how to set TCP timeout value?

- better longer than RTT
  - RTT varies
- too short: premature timeout
  - unnecessary retransmissions
- too long: slow reaction to segment loss

Q: how to estimate RTT?

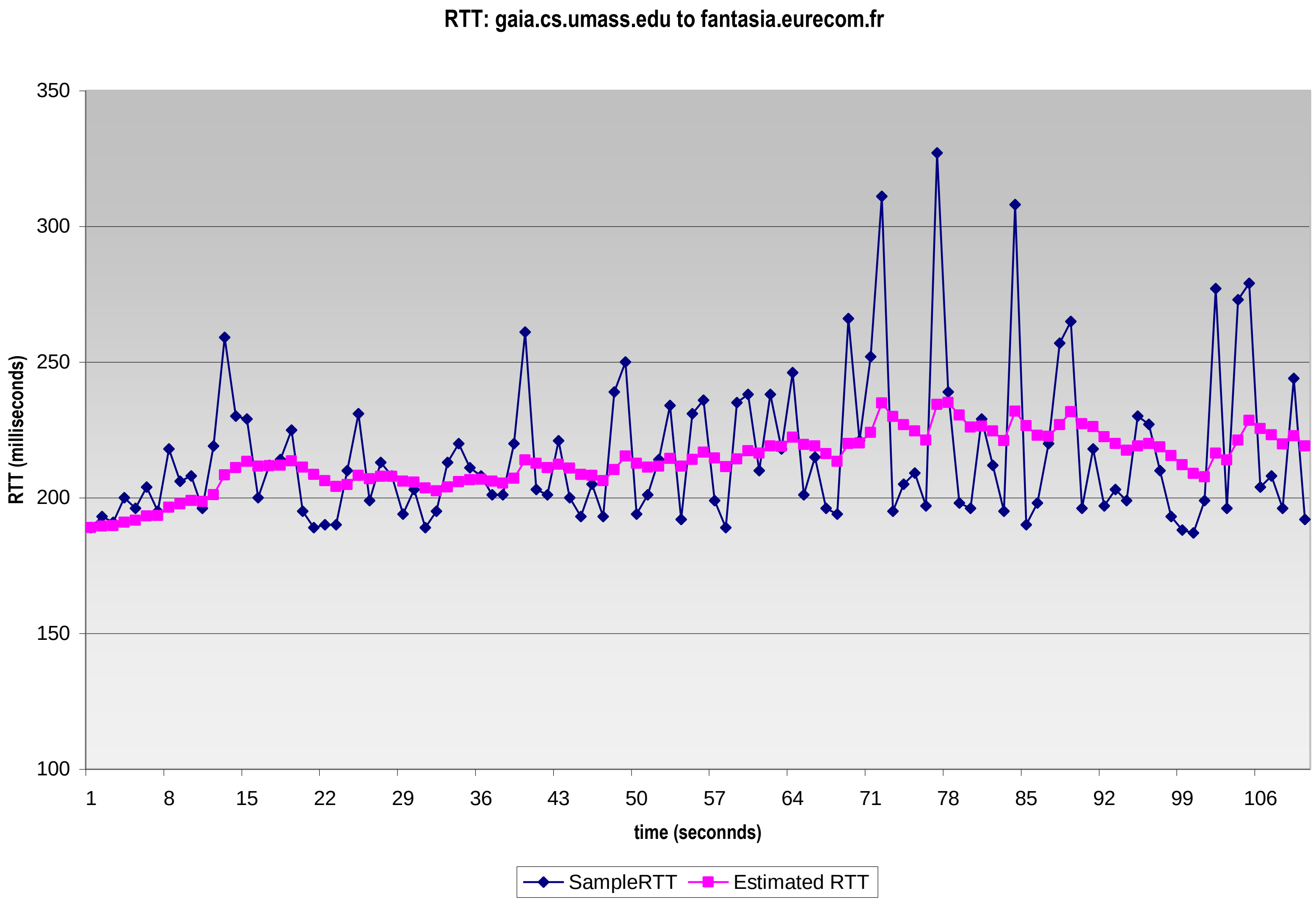
- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions, cumulatively ACKed segments
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - use several recent measurements, not just current **`SampleRTT`**

TCP Round Trip Time and Timeout

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

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

Example RTT estimation:



TCP Round Trip Time and Timeout

Setting the timeout

- **`EstimtedRTT`** plus "safety margin"
  - large variation in **`EstimatedRTT`** -> larger safety margin

- first estimate of how much SampleRTT deviates from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
               β*|SampleRTT-EstimatedRTT|

(typically, β = 0.25)
```

Then set timeout interval:

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

TCP Congestion Control

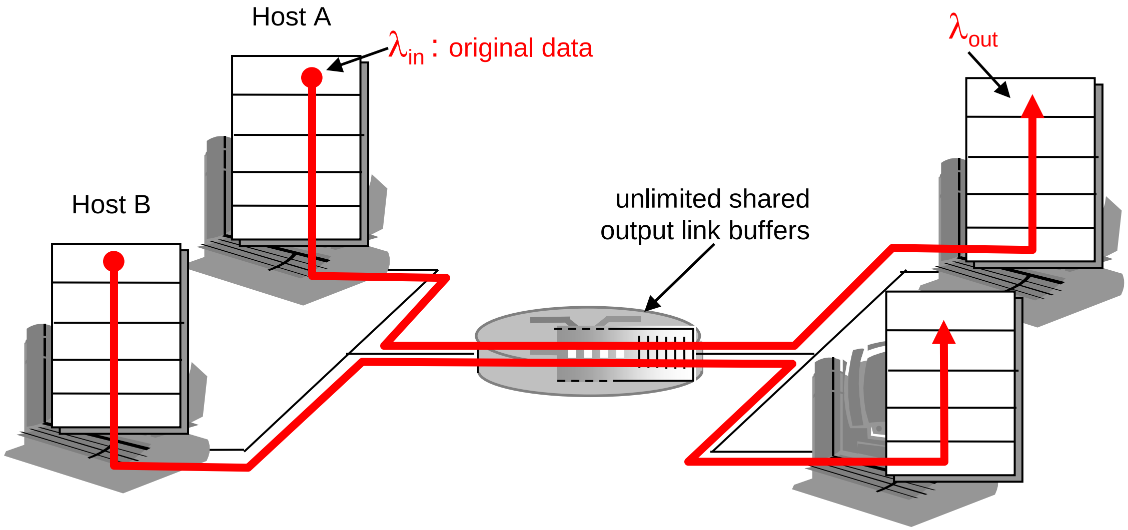
Principles of Congestion Control

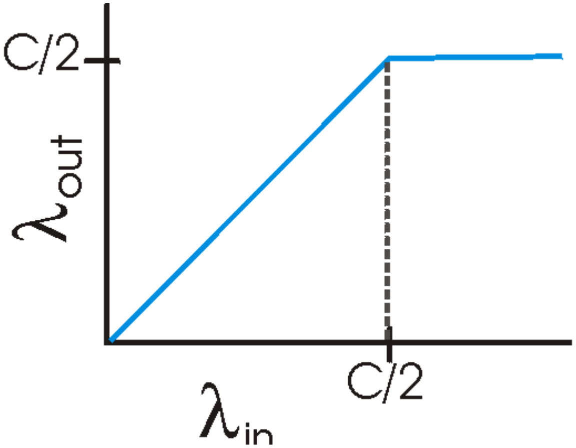
Congestion:

- informally: "too many sources sending too much data too fast for *network* to handle"
  - Demand exceeds capacity which lasts for a certain period of time
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

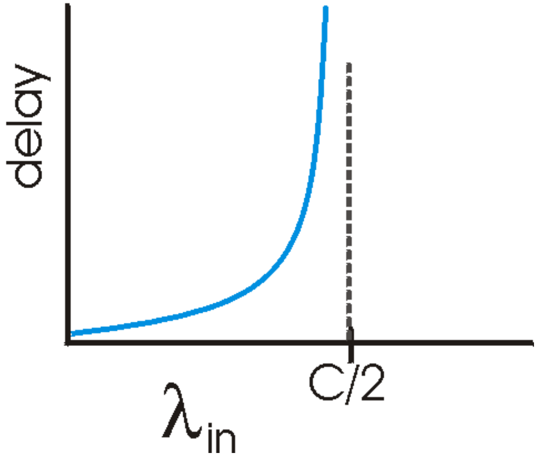
Causes/costs of congestion: scenario 1

- two senders, two receivers (two connections sharing a link)
- one router, infinite buffers
- no retransmission





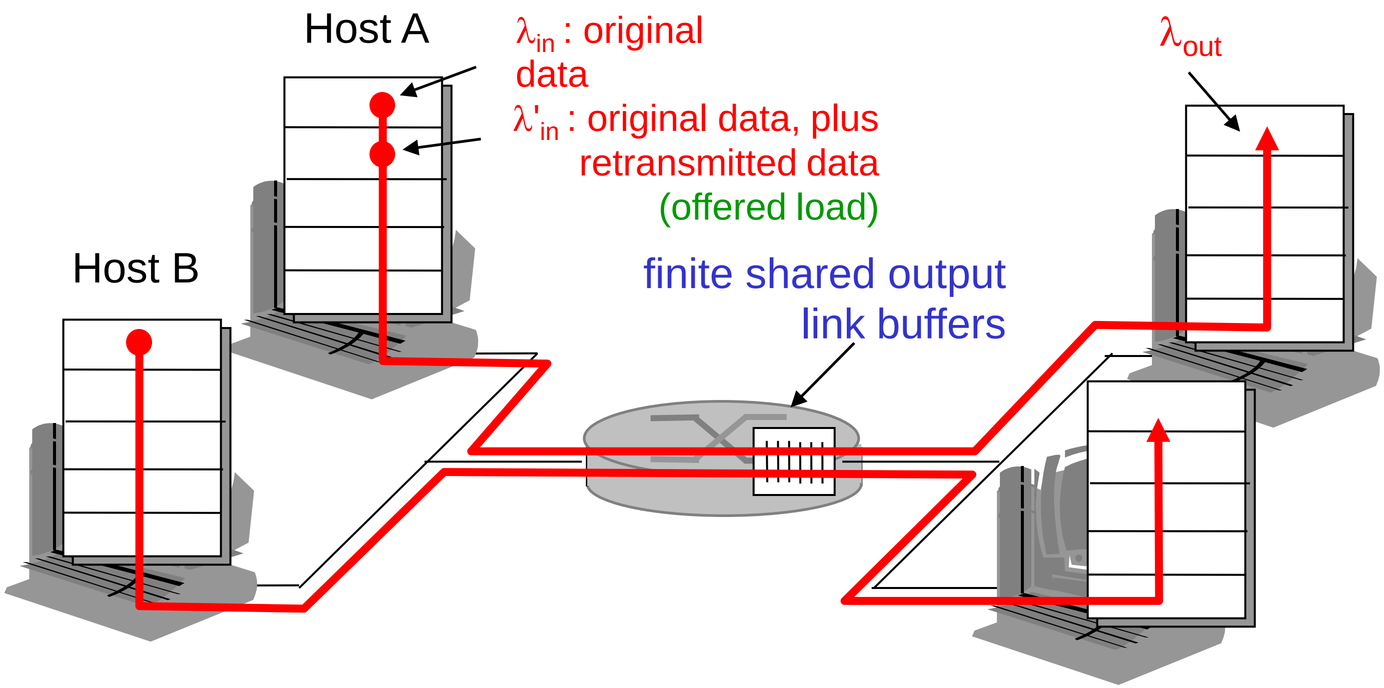
Per-connection throughput



- large delays when congested
- maximum achievable throughput

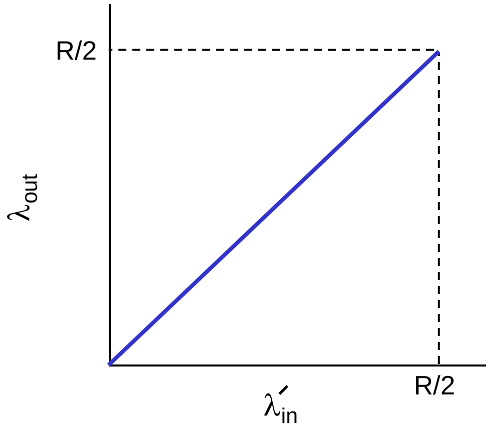
Causes/costs of congestion: scenario 2

- one router, *finite* buffers
- sender retransmission of lost packet

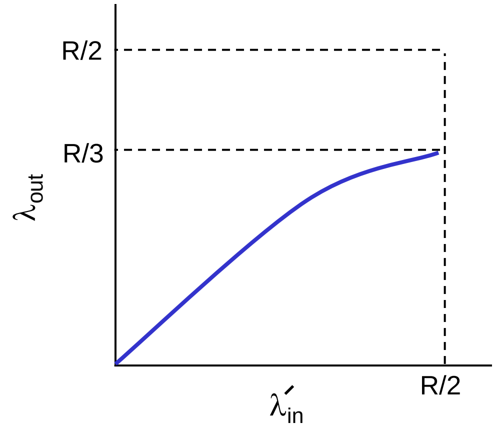


Causes/costs of congestion: scenario 2

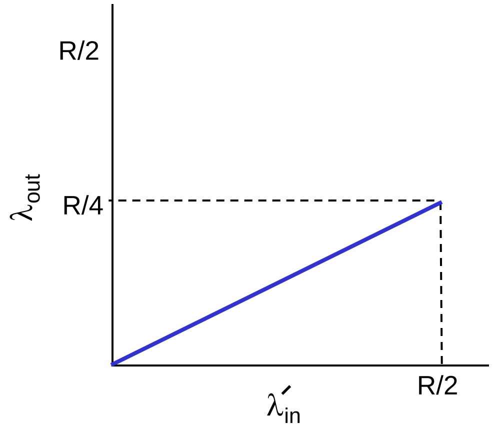
- always: $\lambda_{in} = \lambda_{out}$ (goodput, no loss) (vs. throughput)
- "perfect" retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- retransmission of delayed (not lost) packet makes λ'_{in} larger (than perfect case) for same λ_{out}



a.



b. Retx (1/3 of offered load)



c. Premature timeout & unnecessary retx (one retx per pkt)

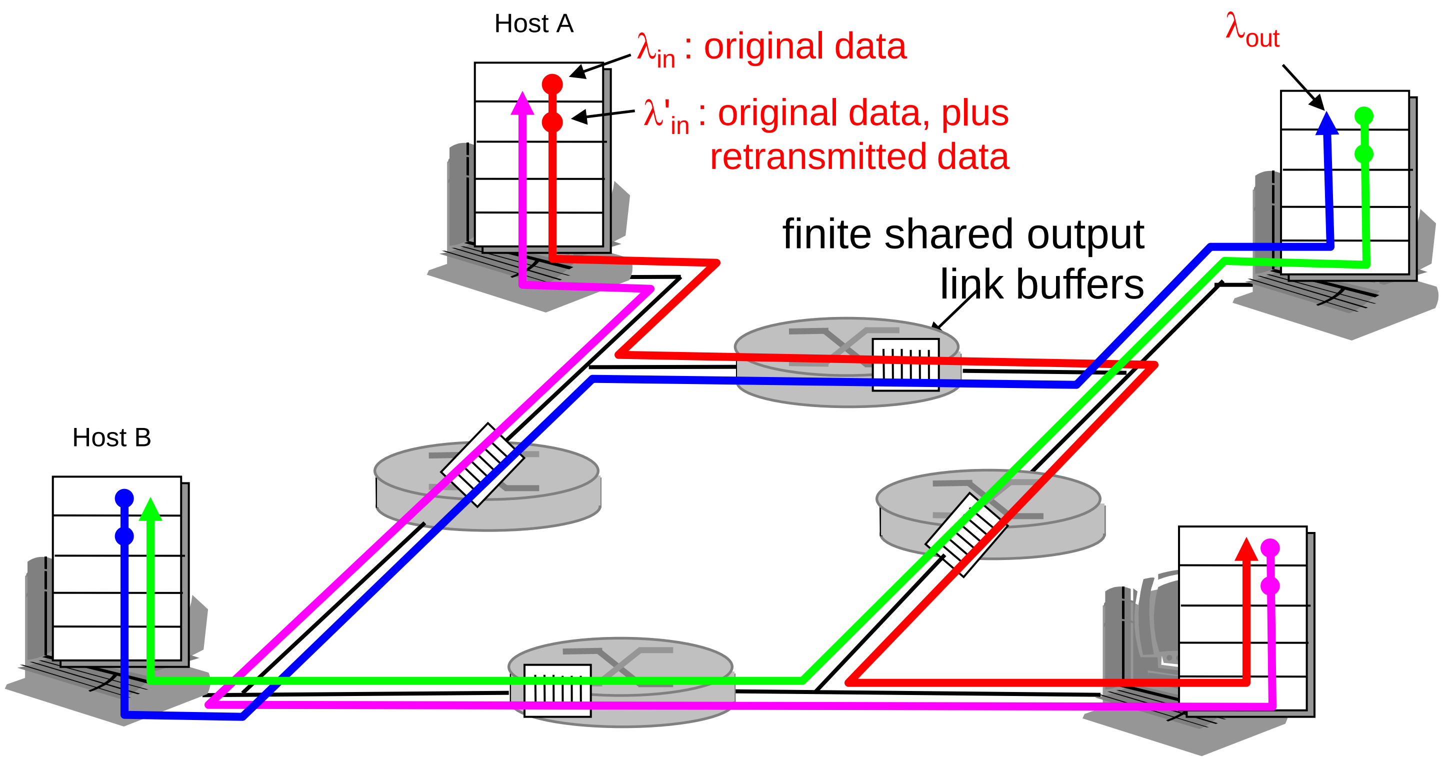
"costs" of congestion:

- more work (retrans) for given "goodput"
- *unneeded* retransmissions: link carries multiple copies of pkt

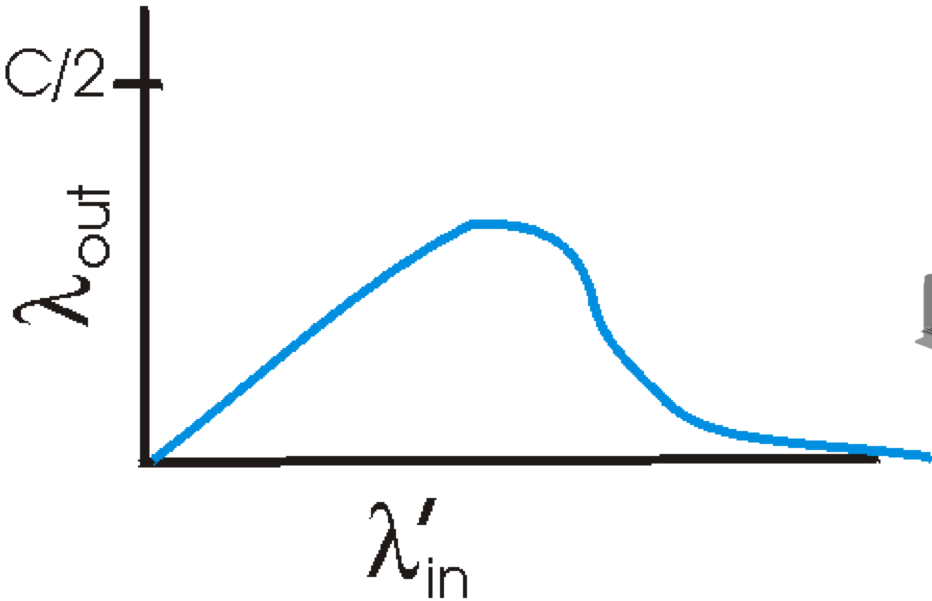
Causes/costs of congestion: scenario 3

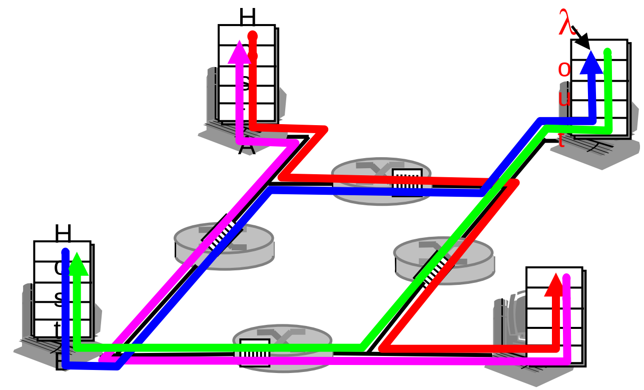
- four senders
- multihop paths
- timeout/retransmit

Q: what happens as λ_{in} and λ'_{in} increase ?



Causes/costs of congestion: scenario 3





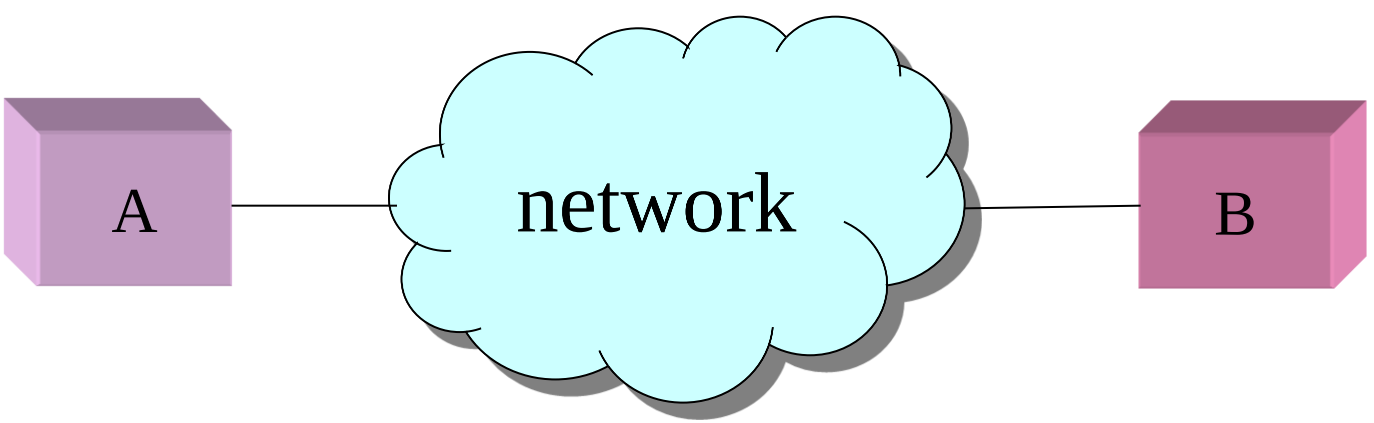
Another "cost" of congestion:

- when packet dropped, any upstream transmission capacity used for that packet was wasted!

Two Common Approaches towards congestion control

#1: End-to-end congestion control:

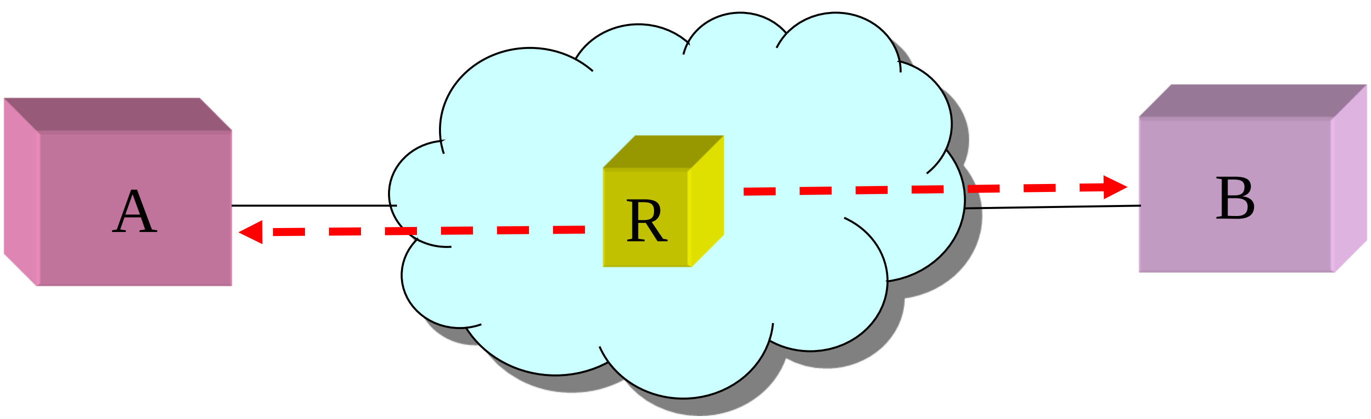
- No explicit feedback from network
- congestion inferred from end-system observed loss, delay
- approach taken by TCP



Two Common Approaches towards congestion control

#:2: 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



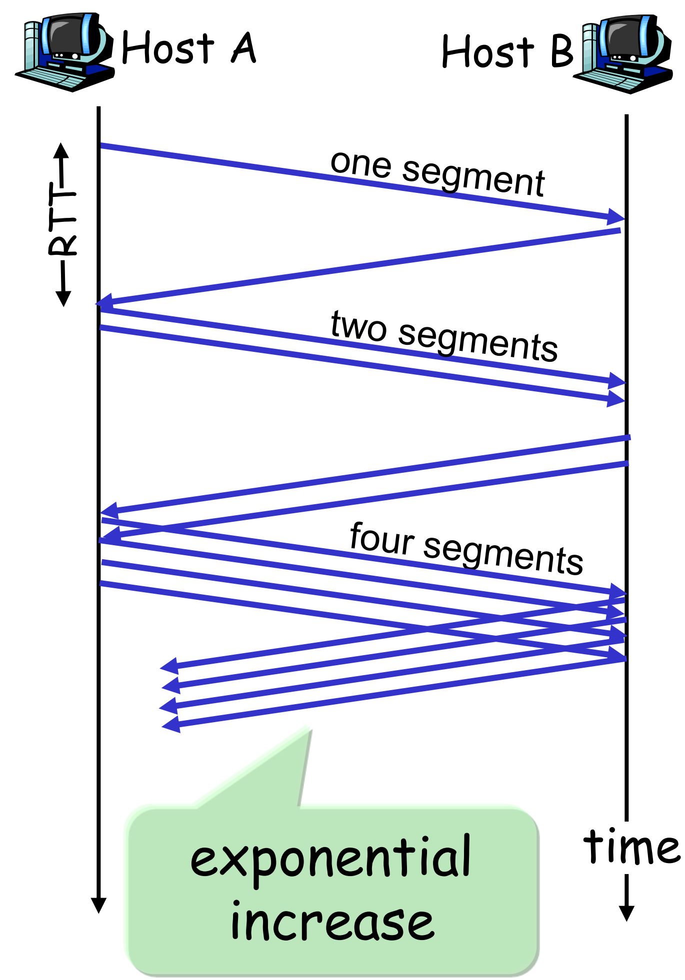
TCP Congestion Control

- Slow start
- Congestion avoidance

TCP Slow Start (1/3)

-> PROBE network's maximum "throughput"!

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



TCP Slow-Start (2/3)

- To get data flowing there must be acks to clock out packets; but to get acks there must be data flowing.
- Maintain a per connection state variable in the sender – "congestion window" *cwnd*
- "When to enter Slow-Start Phase?"
  - When a connection begins
  - After a timeout

TCP Slow Start (3/3)

- Algorithm –
  - When starting or restarting after a loss, set cwnd=1 packet.
  - Each time an ACK is received, *cwnd* is incremented by one segment size, i.e. one ack for each new data,

    cwnd =cwnd+1.
  - When sending, send the min(receiver's_advtiseWin, cwnd)
- *cwnd* is maintained in bytes.
  - The *segment size* is announced by the *receiver*.

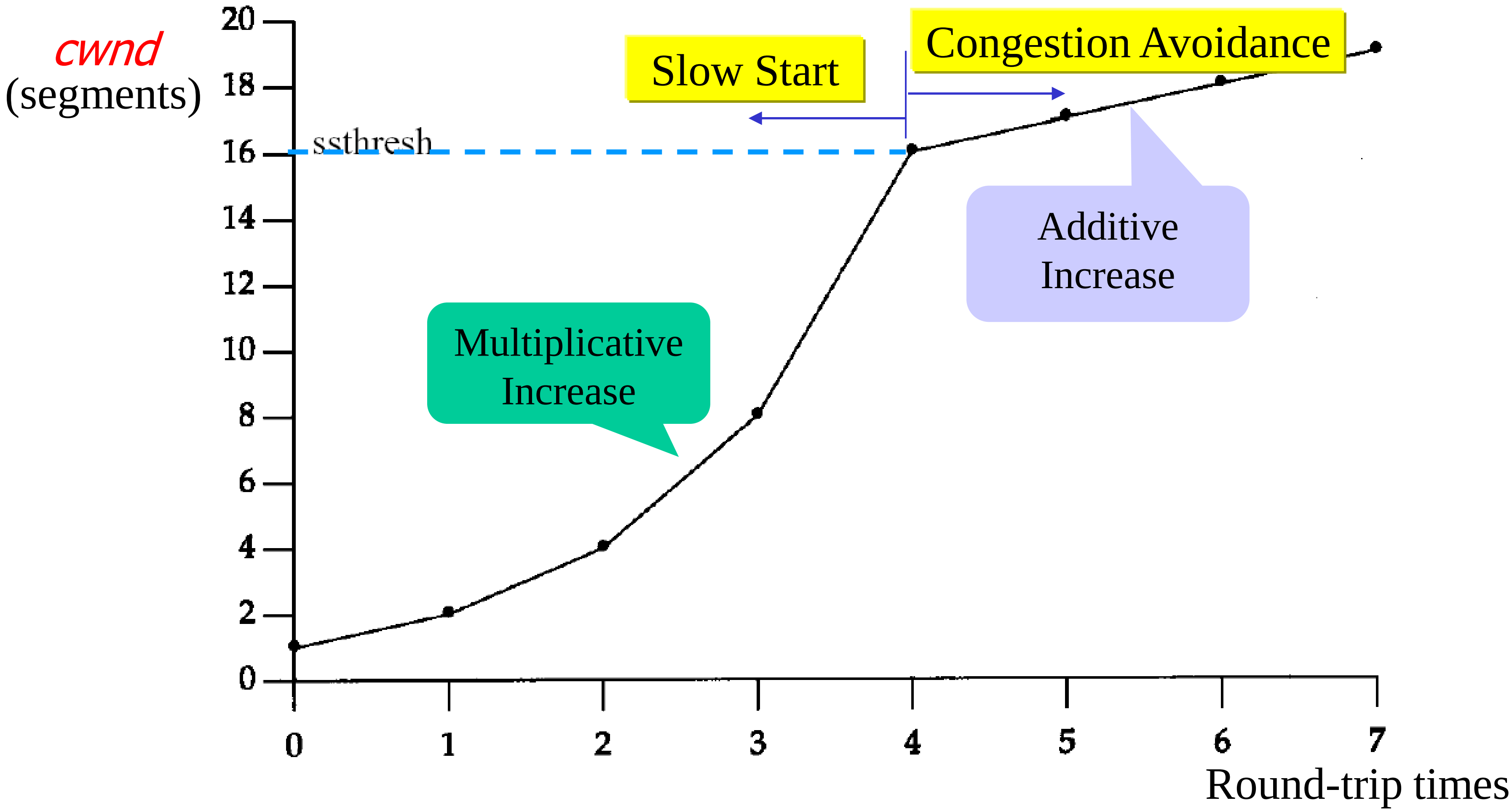
Congestion Avoidance

- Congestion is indicated by a *timeout* or the reception of *three* duplicate ACKs.
- The goal is to avoid increasing the window size too quickly and causing additional congestion.

Congestion Avoidance Algorithm

- Slow start phase
  - When a connection begins: *cwnd* is one segment and *ssthresh* (slow start threshold) is 65,535 bytes.
  - When congestion occurs, *ssthresh=cwnd/2*, cwnd=1
- Once *cwnd=ssthresh*, the connection enters the congestion avoidance phase.
  - On each ack for new data, cwnd=cwnd+1/cwnd (additive increase)
  - When sending, send the min(receiver's AdvertiseWinow, cwnd)

Additive increase



※Visualization of slow start and congestion avoidance.

Duplicate ACKs

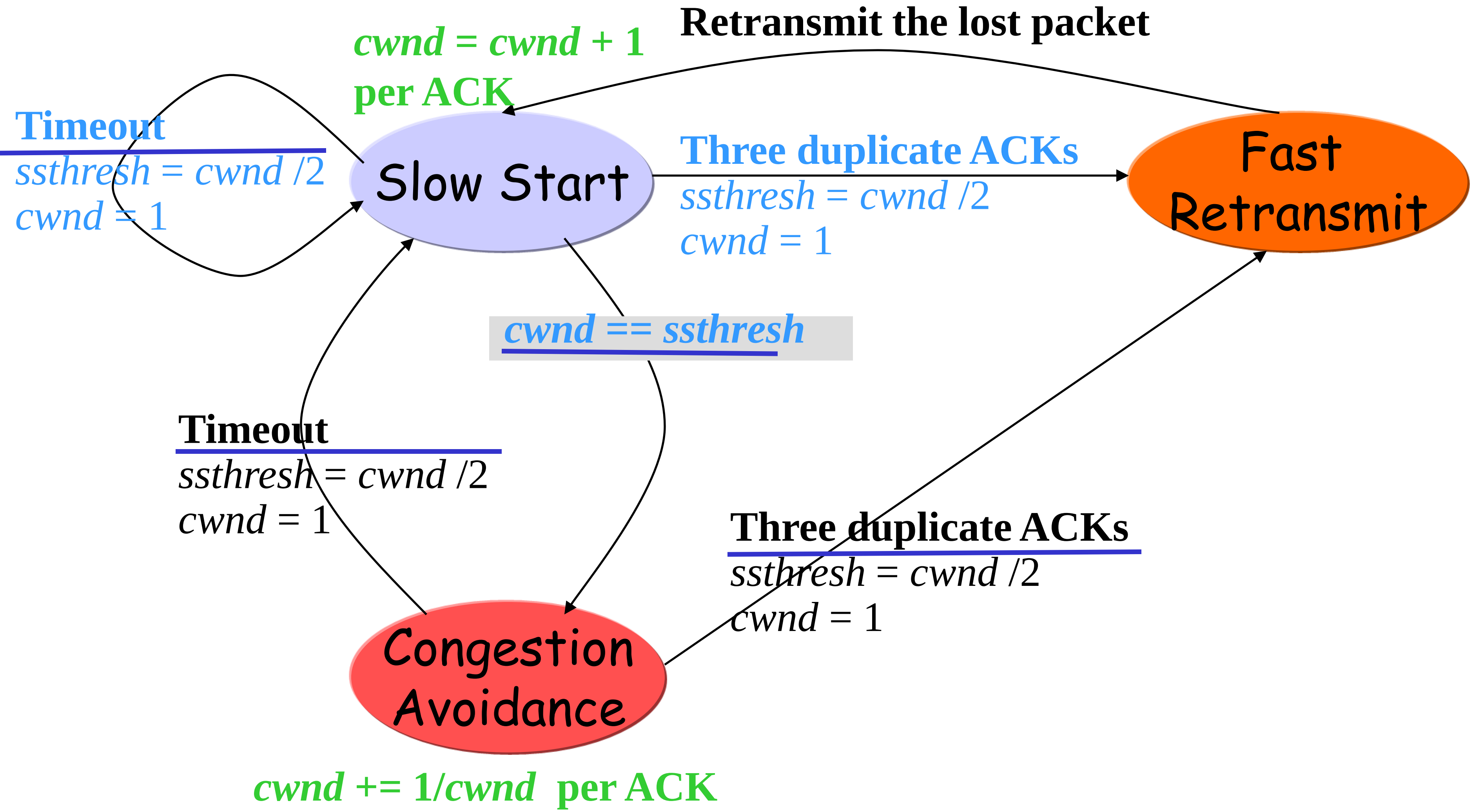
- If there are less than 3 duplicate ACKs, it is assumed that there is just a *reordering* of the segments.
- If 3 or more duplicate ACKs are received in a row, it is a *strong* indication that a segment has been lost.
- Fast Retransmit and Fast Recovery

  -> TCP-tahoe and TCP-reno

Fast Retransmit

- When 3 duplicate ACKs are received, a retransmission is performed *without* waiting for a retransmission timer to expire.
- ssthresh=cwnd/2 and cwnd = 1; (entering Slow Start phase)
- Retransmit the missing segment.

TCP Tahoe



Retransmit the lost packet
cwnd = cwnd + 1
per ACK
Timeout
ssthresh = cwnd /2
cwnd = 1
Slow Start
Three duplicate ACKs
ssthresh = cwnd /2
cwnd = 1
Fast
Retransmit
cwnd == ssthresh
Timeout
ssthresh = cwnd /2
cwnd = 1
Three duplicate ACKs
ssthresh = cwnd /2
cwnd = 1
Congestion
Avoidance
cwnd += 1/cwnd per ACK

TCP Tahoe

- After fast retransmit, goes to "slow-start" phase to probe the network again.
- To avoid congest the network.

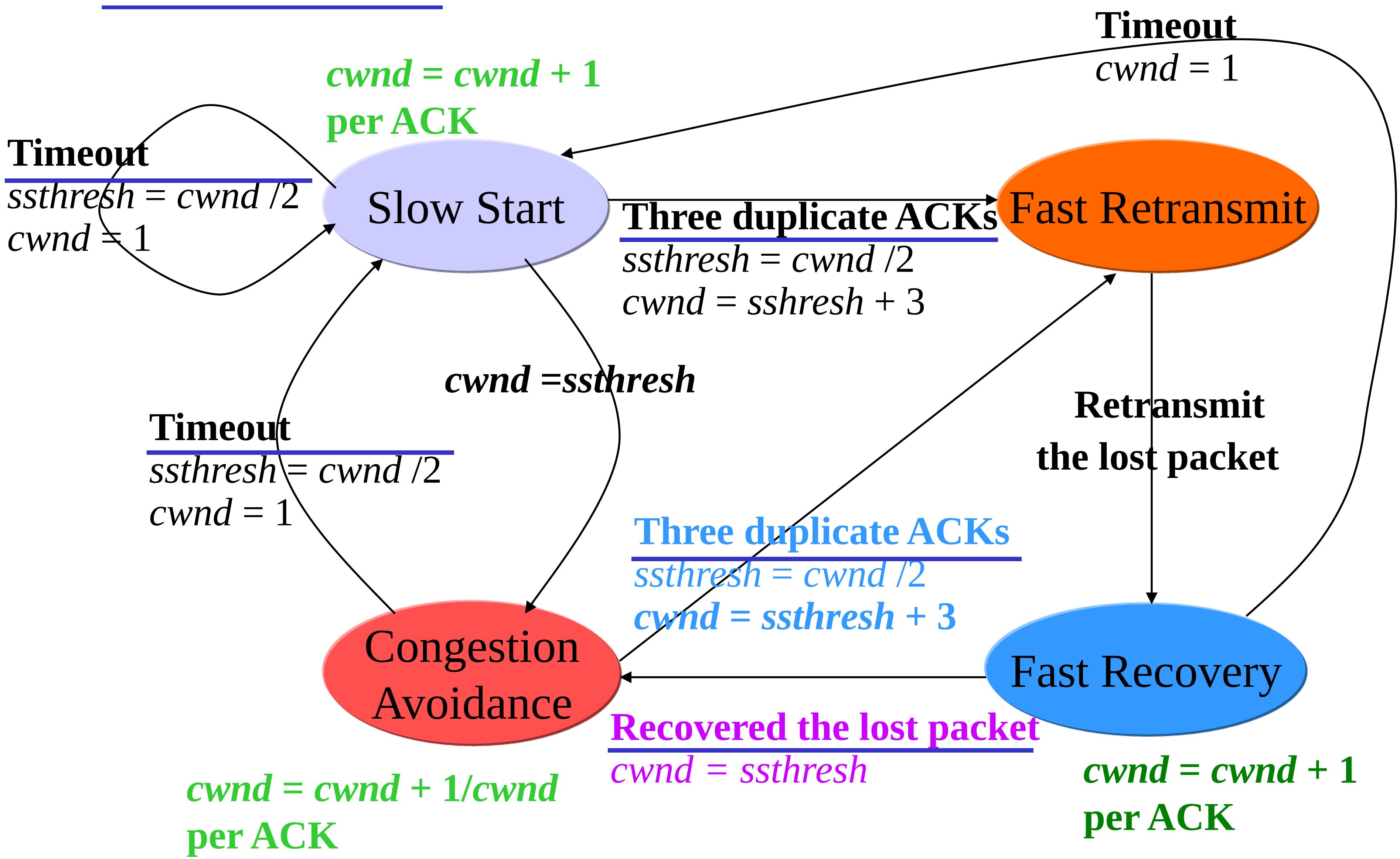
Fast Recovery

- Immediately after fast retransmit, instead of entering slow start, *congestion avoidance* is performed.
- To boot up throughput
- *ssthresh=cwnd/2; cwnd* = *ssthresh* + 3 segments
- Each time an ACK or a duplicate ACK arrives, increment *cwnd* by the segment size *cwnd++;*
- Allow to transmit new packet

Fast Recovery (cont'd)

- When the next ACK arrives that acknowledges the lost data,
  - set *cwnd* to *ssthresh*
  - enter congestion avoidance phase

TCP Reno



Timeout
cwnd = 1
cwnd = cwnd + 1
per ACK
Timeout
ssthresh = cwnd /2
cwnd = 1
Slow Start
Three duplicate ACKs
ssthresh = cwnd /2
cwnd = sshresh + 3
Fast Retransmit
cwnd =ssthresh
Retransmit
the lost packet
Timeout
ssthresh = cwnd /2
cwnd = 1
Three duplicate ACKs
ssthresh = cwnd /2
cwnd = ssthresh + 3
Congestion
Avoidance
Fast Recovery
Recovered the lost packet
cwnd = ssthresh
cwnd = cwnd + 1/cwnd
per ACK
cwnd = cwnd + 1
per ACK

TCP-Reno: Congestion Window Size

Slow start

(1) After every new acknowledgment

if (CWND < SSTHRESH)

CWND $\leftarrow$ CWND + 1

Congestion avoidance

else

CWND $\leftarrow$ CWND + 1/CWND

(2) Upon RTO (retransmission timeout)

Begin of slow start

SSTHRESH $\leftarrow$ CWND/2

CWND $\leftarrow$ 1

(3) When NDUP (# of duplicate ACKs) exceeds 3

Begin of fast recovery

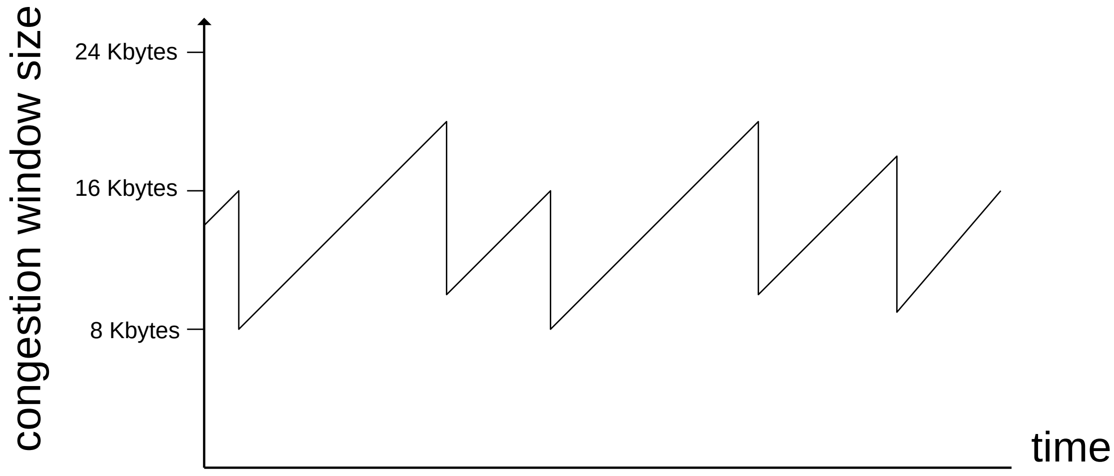
SSTHRESH $\leftarrow$ CWND/2

CWND $\leftarrow$ CWND/2 + 3

Summary of 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



Summary of TCP Congestion Control: details

- sender limits transmission:
  `LastByteSent-LastByteAcked` $\leq$ `CongWin`
- Roughly,

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

- `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

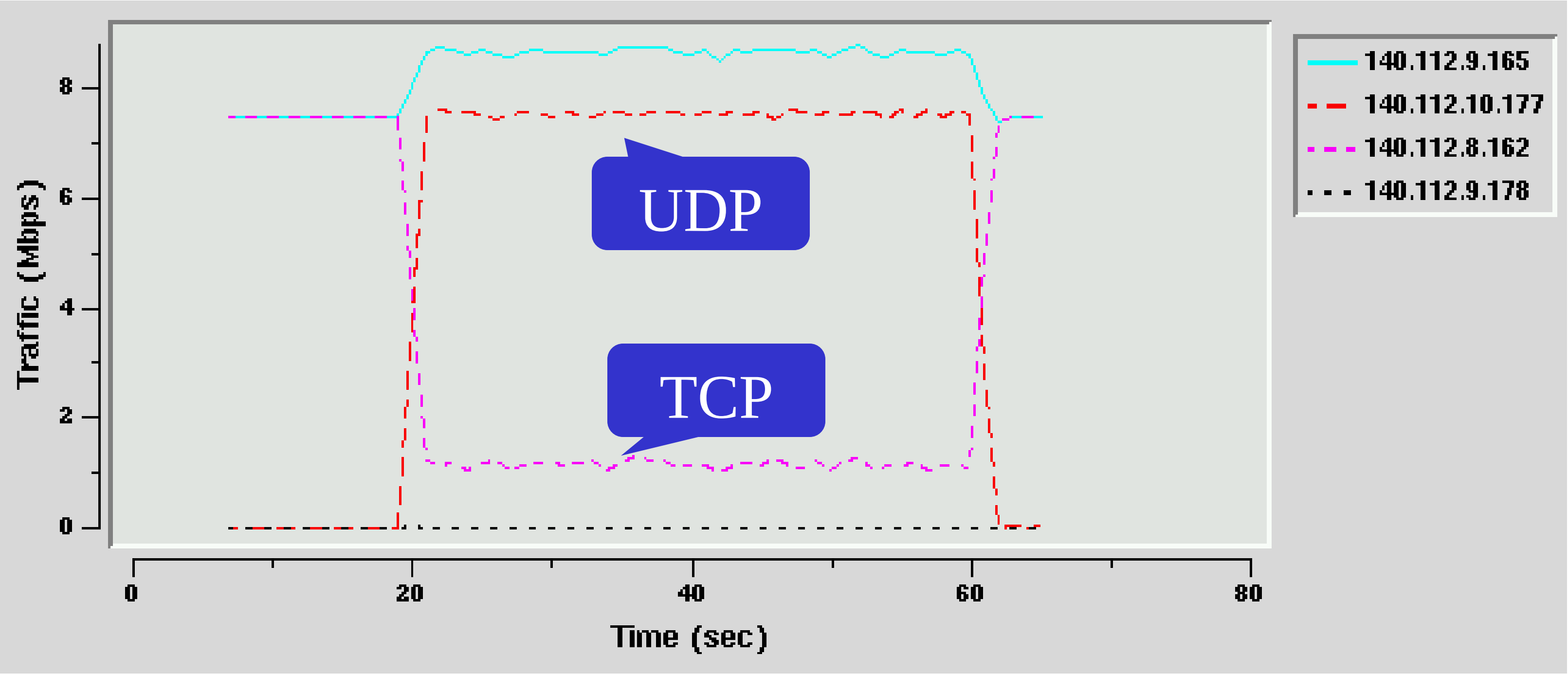
TCP throughput

- What's the average throughout of TCP as a function of window size and RTT?
  - Ignore slow start
- Let W be the window size when loss occurs.
- When window is W, throughput is W/RTT
- Just after loss, window drops to W/2, throughput to W/2RTT.
- Average throughout: .75 W/RTT

Competition of TCP connection with UDP flow

- Sender 1 (140.112.8.162)先以8 Mbps的速度送出TCP traffic
- 20秒後Sender 2 (140.112.10.177)再以8 Mbps的速度送出UDP traffic
- The buffer space is 100KB for both queues. There is no packet drop.
- After UDP traffic starts, TCP throughput drops to less than 2Mb，UDP has the rest。
- Possible cause: Receiver (140.112.9.165) fails to send ACKs to Sender 1，causing Sender 1以為發生packet loss，因此把window size調降，而使得傳送的速率下降。

Competition of TCP connection with UDP flow (cont'd)



The end. ☺

Homework #5

Chapter 3

- R5, R6, R10, R11, R14, P2, P5, P9, P12.

Homework #6

Chapter 3

- P15, P16, P24, P32, D1