רשתות תקשורת מחשבים
פרק 3 – שכבת התעבורה

אליאב מנשה
eliav.menachi@gmail.com

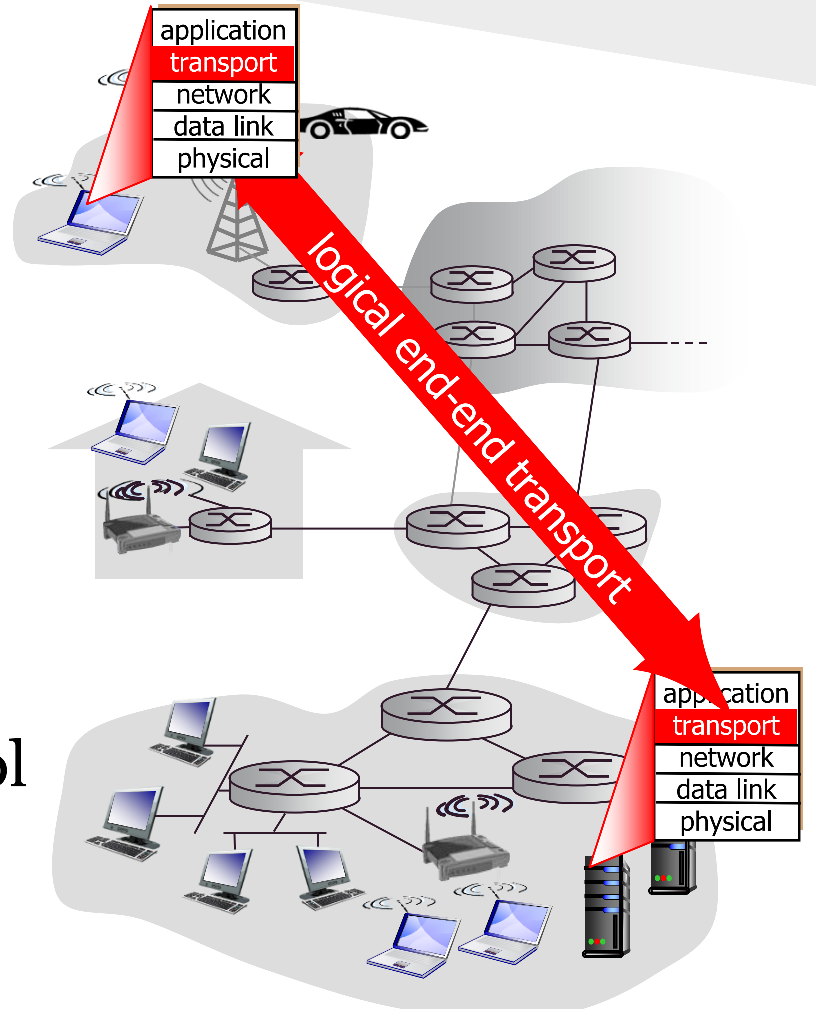
Transport-Layer Services

Transport services and protocols

- Provide *logical communication* between app processes running on different hosts

- Transport protocols run in end systems
  - send side: breaks app messages into ***segments***, passes to network layer
  - rcv side: reassembles ***segments*** into messages, passes to app layer

- more than one transport protocol available to apps
  - TCP and UDP



Transport vs. network layer

- *Transport layer:* logical communication between processes
- *Transport layer*, relies on, enhances, *network layer* services
- *Network layer:* logical communication between hosts

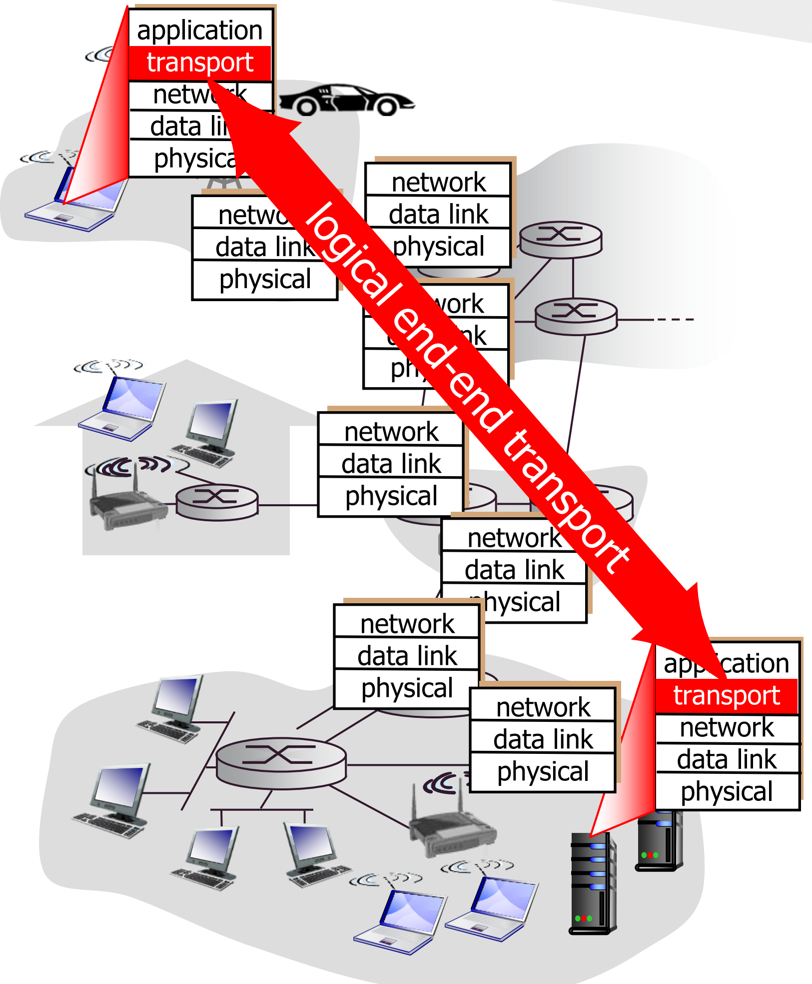
household analogy:

12 kids in Ann's house sending letters to 12 kids in Bill's house:

- hosts = houses
- processes = kids
- app messages = letters in envelopes
- transport protocol = Ann and Bill who demux to in-house siblings
- network-layer protocol = postal service

Internet transport-layer protocols

- Reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- Unreliable, unordered delivery (UDP)
  - no-frills extension of "best-effort" IP
- Services not available:
  - delay guarantees
  - bandwidth guarantees



connectionless transport: UDP

UDP: User Datagram Protocol

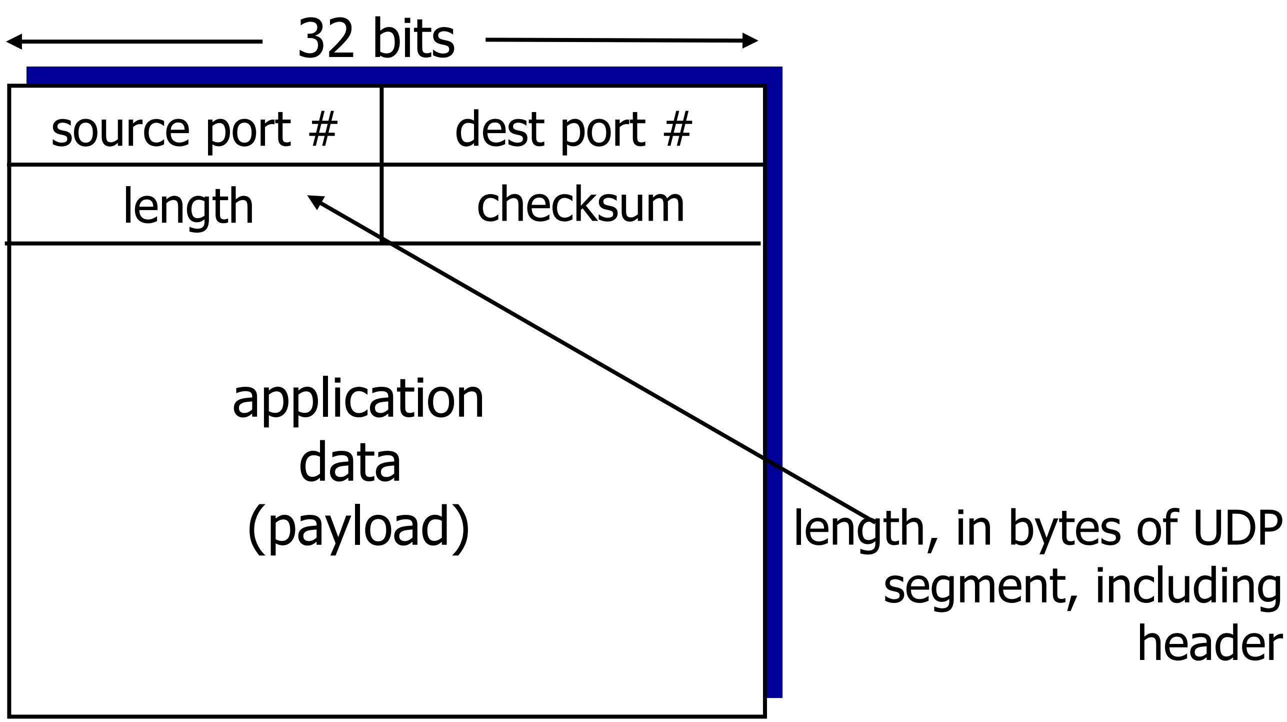
- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - Lost
  - delivered out-of-order to app
- *Connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others
- UDP use:
  - DNS
  - SNMP

why is there a UDP?

- **no connection establishment** (which can add delay)
- **simple**: no connection state at sender, receiver
- **small header size**
- **no congestion control**: UDP can blast away as fast as desired

UDP: segment header

Q: So we know what UDP is doing, lets think how is header look like?



UDP segment format

UDP checksum

Goal: detect "errors" (e.g., flipped bits) in transmitted segment

sender:
- treat segment contents, including header fields, as sequence of 16-bit integers
- checksum: addition (one's complement sum) of segment contents
- sender puts checksum value into UDP checksum field

receiver:
- compute checksum of received segment
- check if computed checksum equals checksum field value:
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless?

UDP checksum – on what?

bits	0 – 7	8 – 15	16 – 23	24 – 31
0	Source address			
32	Destination address			
64	Zeros	Protocol	UDP length	
96	Source Port		Destination Port	
128	Length		Checksum	
160+	Data			

Internet Checksum Example

Note: When adding numbers, a carryout from the most significant bit needs to be added to the result

Example: add two 16-bit integers

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
		1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
wraparound	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
sum		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Internet Checksum Example

שאלה ממבחן

- ב. ההודעה הבאה התקבלה ביישום מסויים המתקשר באמצעות UDP :
  - האם ההודעה שהתקבלה תקינה או לא? אם לא מה צריך להיות ה- checksum כדי שההודעה תהיה תקינה?
  - מהוא הפורט אליו נשלחה ההודעה?
  - מהוא פורט היישום ששלח את ההודעה?
  - מהוא אורך המידע שההודעה מכילה?
  - מהוא אורכה הכללי של הודעה זו?
  - לאיזה שירות\פרוטוקול נשלחת ההודעה והאם היא מתאימה לאותו שירות, הסבר בקצרה?

Internet Checksum Example

Src port	0000	0000	1010	0000
Dest port	0000	0000	0001	0101
length	0000	0000	0001	0100
Checksum	0001	0001	1101	1010

Src port	0000	0000	1010	0000
Dest port	0000	0000	0001	0101
lengh	0000	0000	0001	0100
	0000	0000	1100	1001
checksum	1111	1111	0011	0110

Internet Checksum Example

Src port	0000	0000	1010	0000
Dest port	0000	0000	0001	0101
lengh	0000	0000	0001	0100
	0000	0000	1100	1001
checksum	1111	1111	0011	0110

Bad Checksum

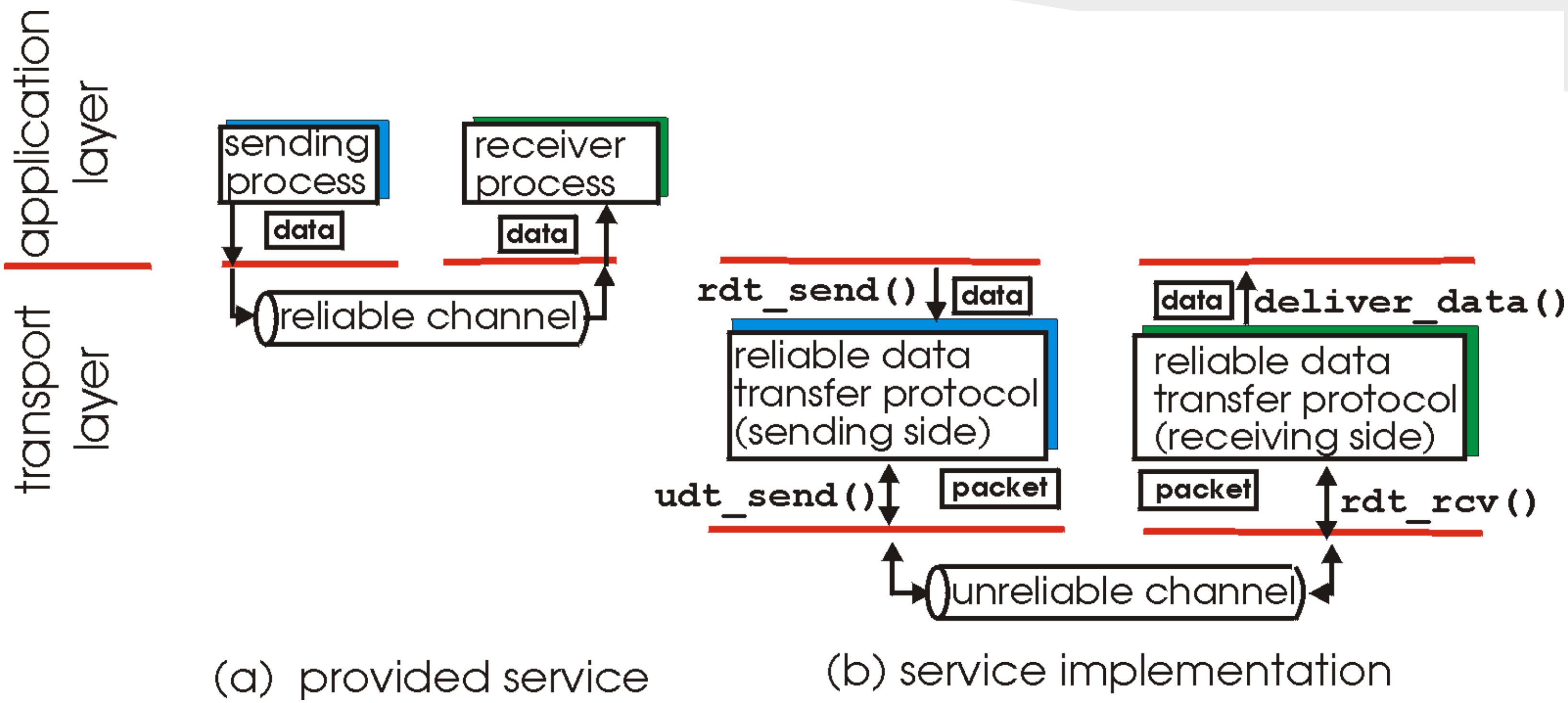
Source port 160

Destination Port 21, FTP….???

Segment Length 20

principles of reliable data transfer

Principles of Reliable data transfer

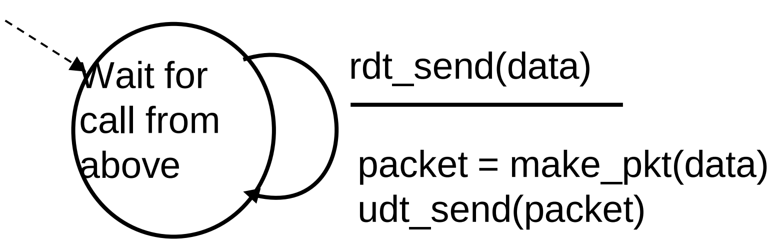


characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

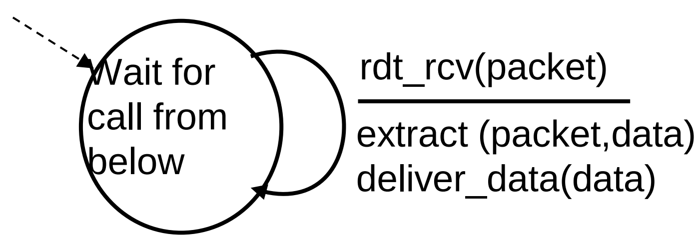
Reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets

- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



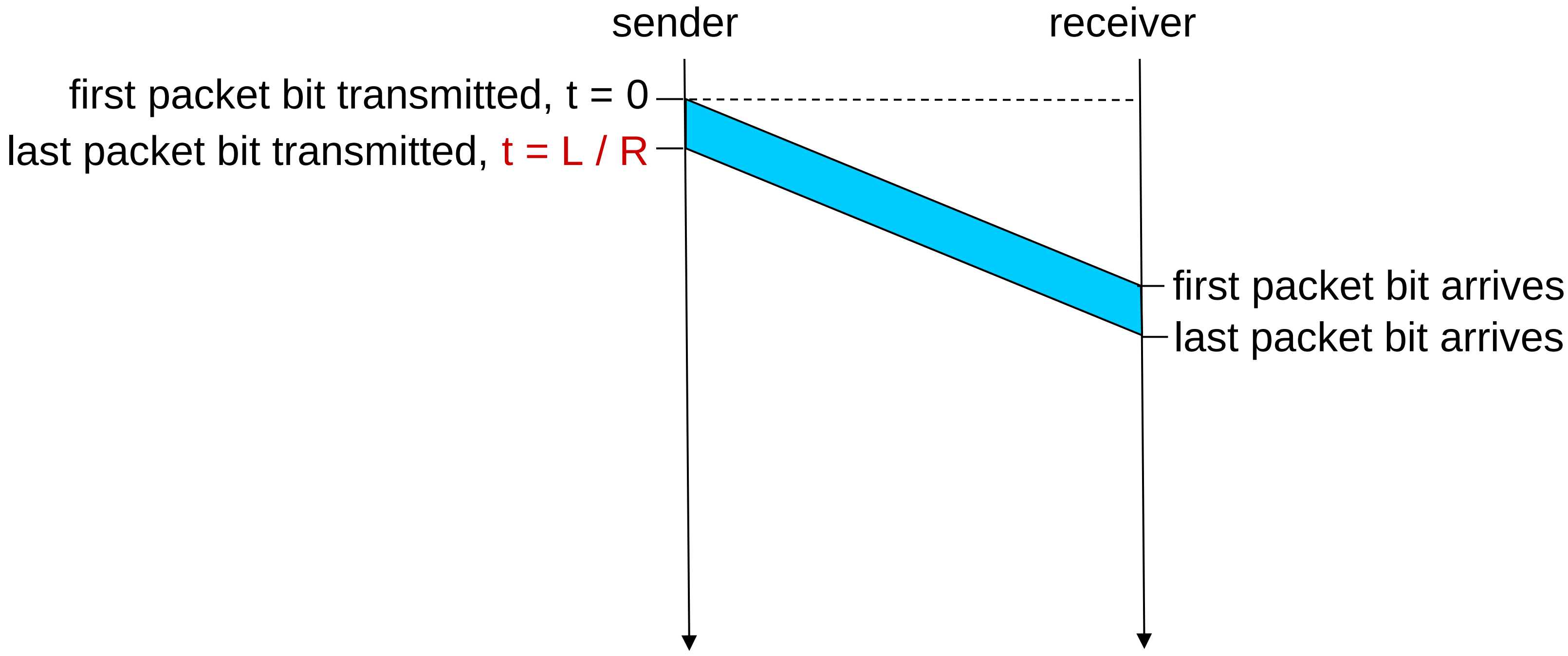
sender



receiver

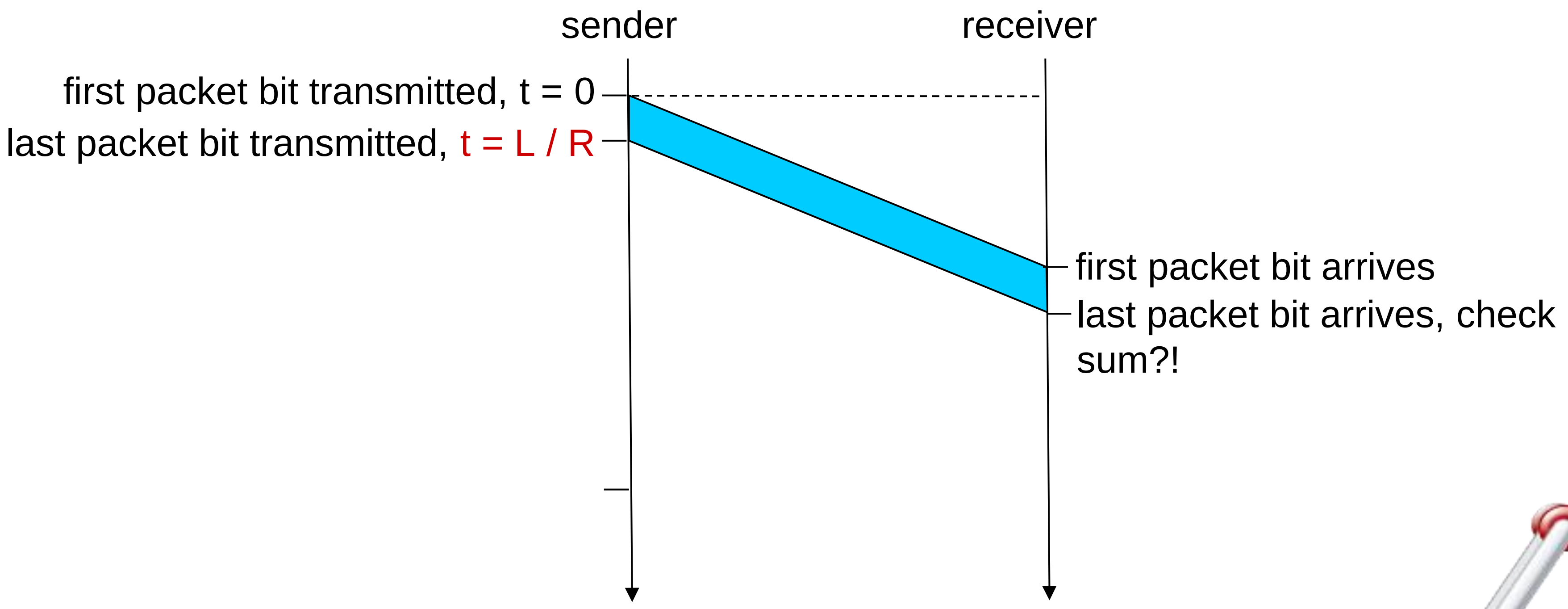
Level 1 Q: Build the reliable data Transfer, channel with bit errors

Think simple – how the protocol will look like



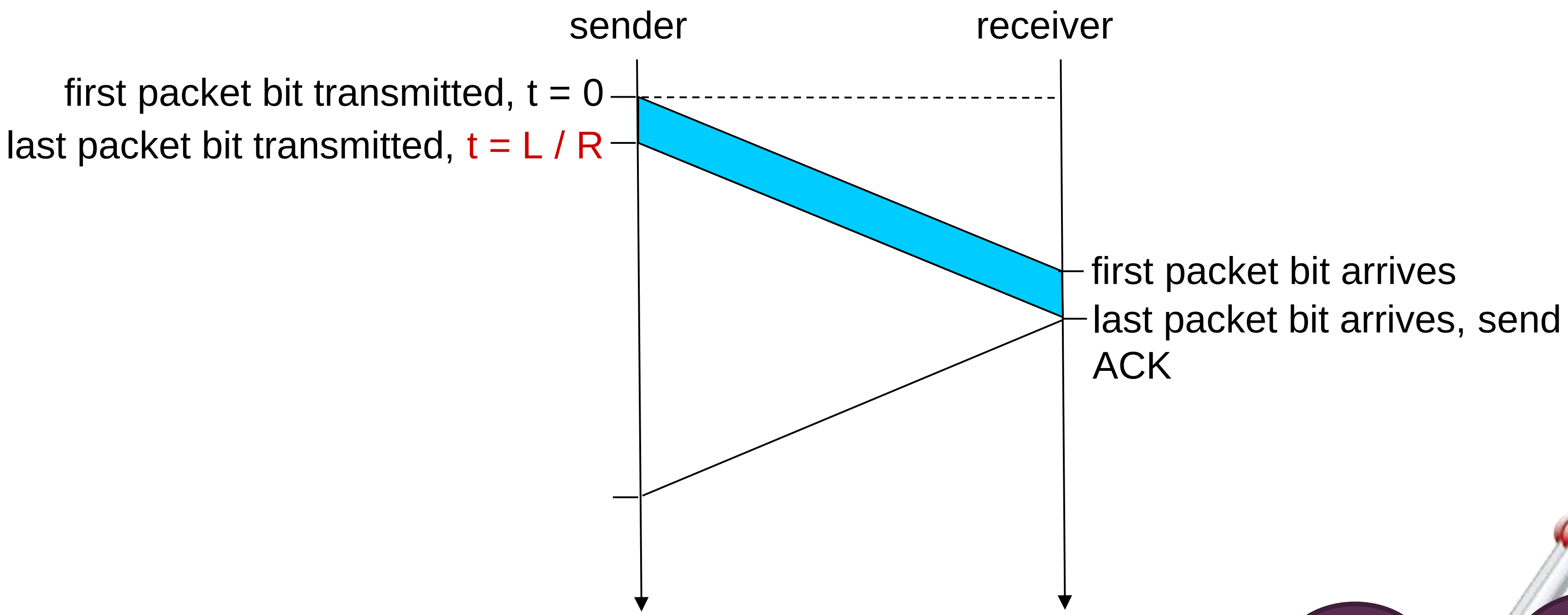
Level 1 Q: Build the reliable data Transfer, channel with bit errors

Think simple - How do we know if the packet arrive without any errors



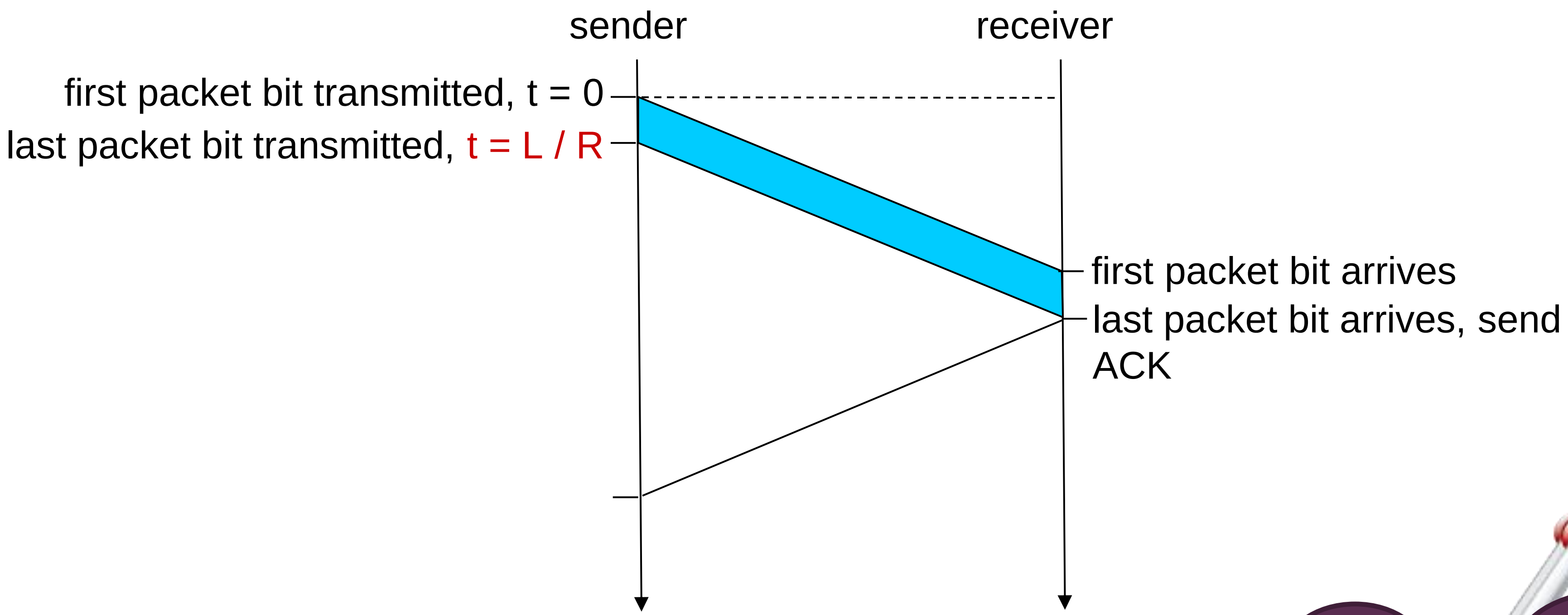
Level 1 Q: Build the reliable data Transfer, channel with bit errors

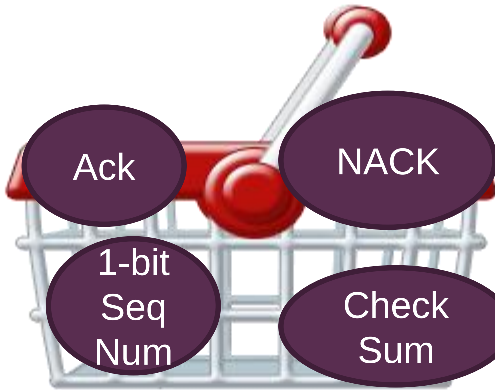
Think simple – Checksum is fine, what next?



Level 1 Q: Build the reliable data Transfer, channel with bit errors

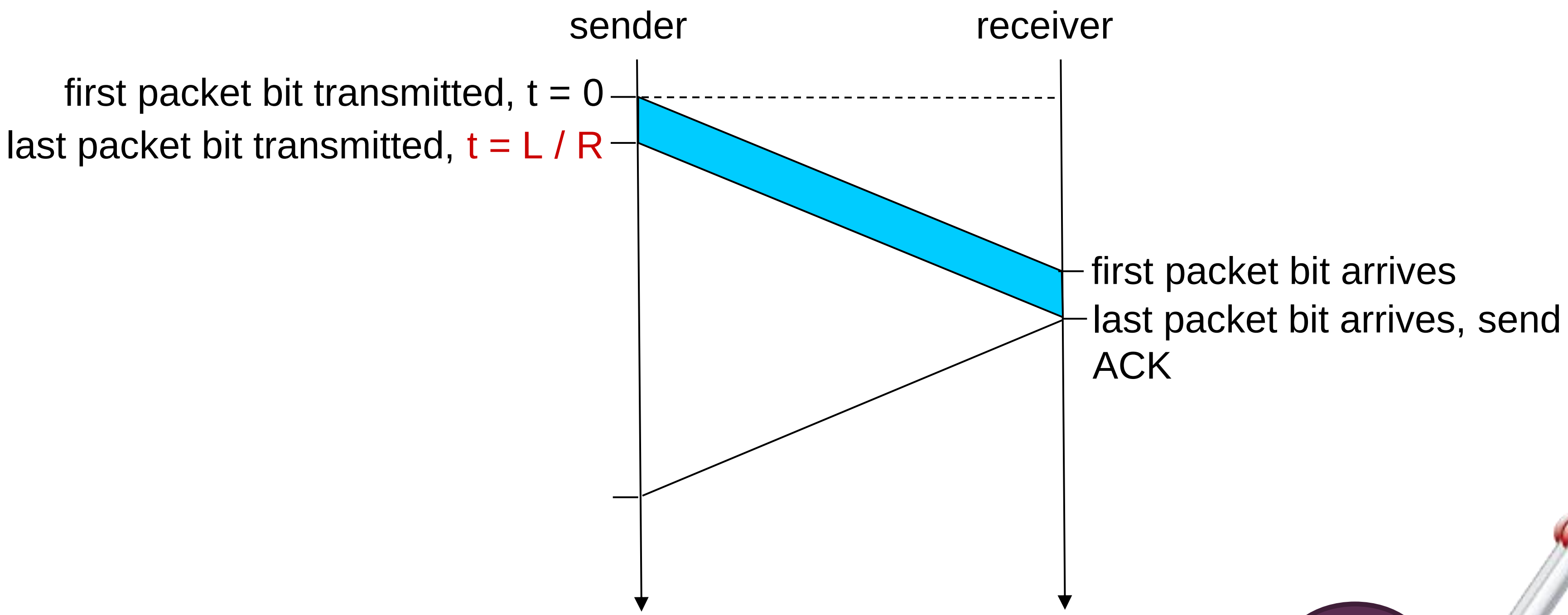
Think simple – received ACK with error?

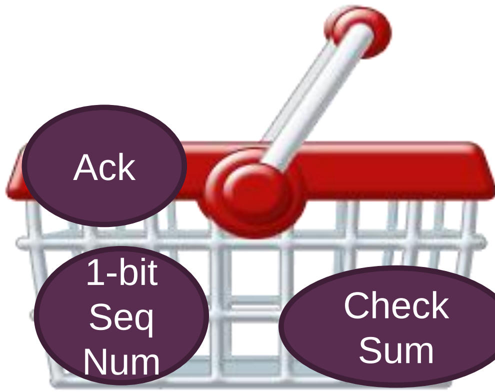




Level 1 Q: Build the reliable data Transfer, channel with bit errors

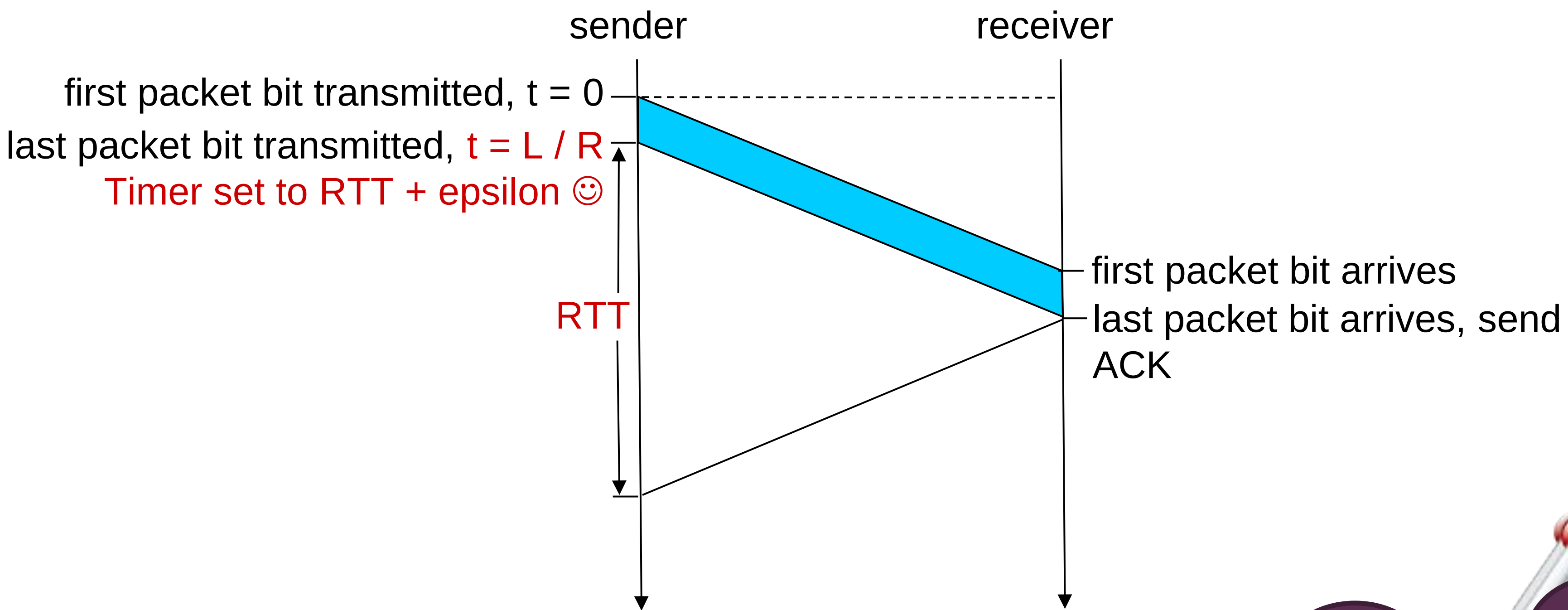
Think simple – What we can get by using the ACK + Seq num?

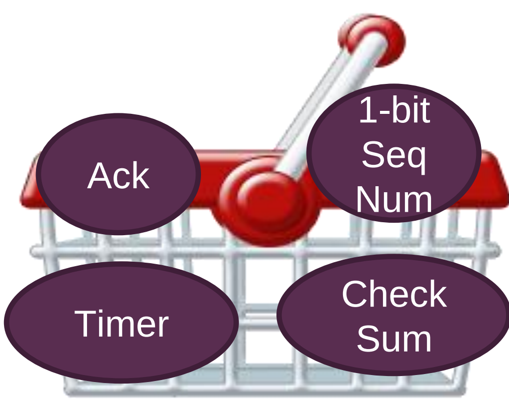




Level 2 Q: Build the reliable data Transfer, channels with errors + loss

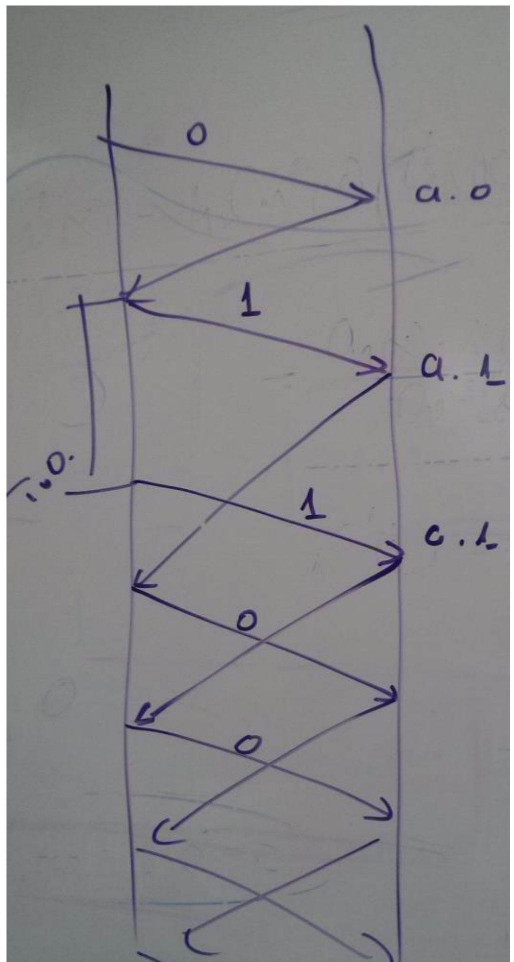
Think simple – Ack is coming?

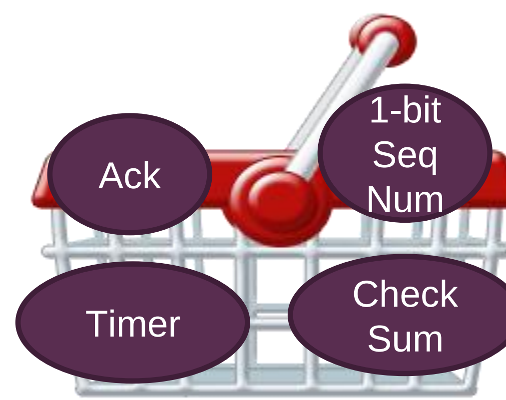




Level 2 Q: Build the reliable data Transfer, channels with errors + loss

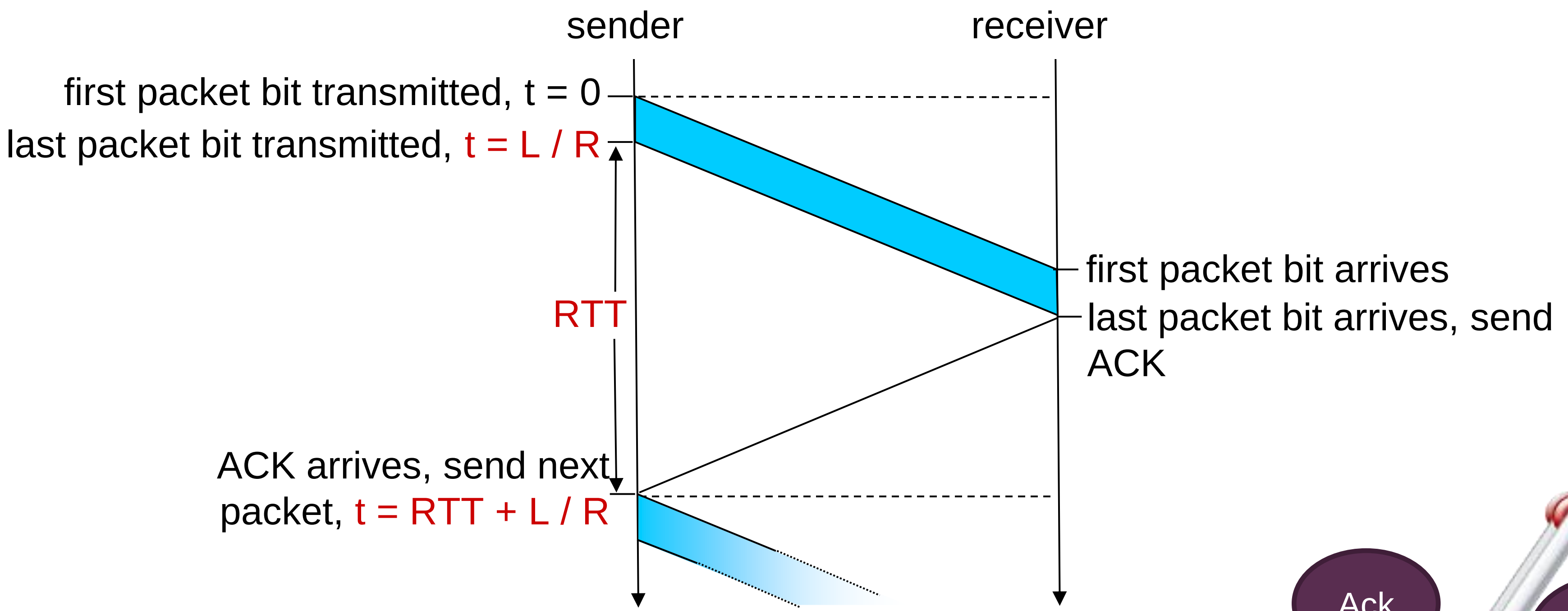
Think simple – retransmit via Timeout on receiving ACK?





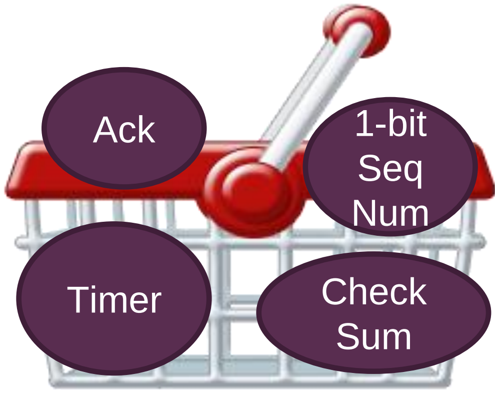
Level 2 Q: Build the reliable data Transfer, channels with errors + loss

Think simple – Lets see what is the efficiency of our new algorithm



$$U_{sender} = \frac{L / R}{RTT + L / R}$$

So our new algorithm is good or not

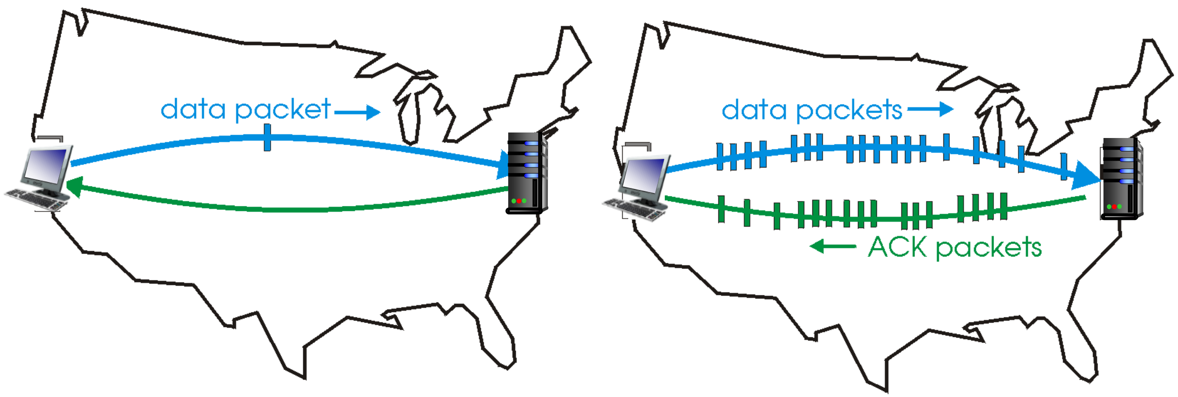


Level 2 Q: Build the next reliable data Transfer

How can we improve it

Pipelining:

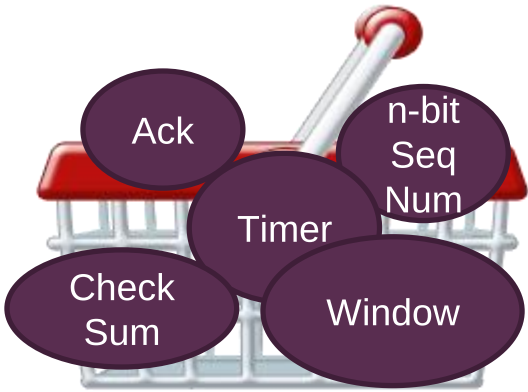
- sender allows multiple, "in-flight", yet to be acknowledged pkts
- range of sequence numbers must be increased buffering at sender and/or receiver



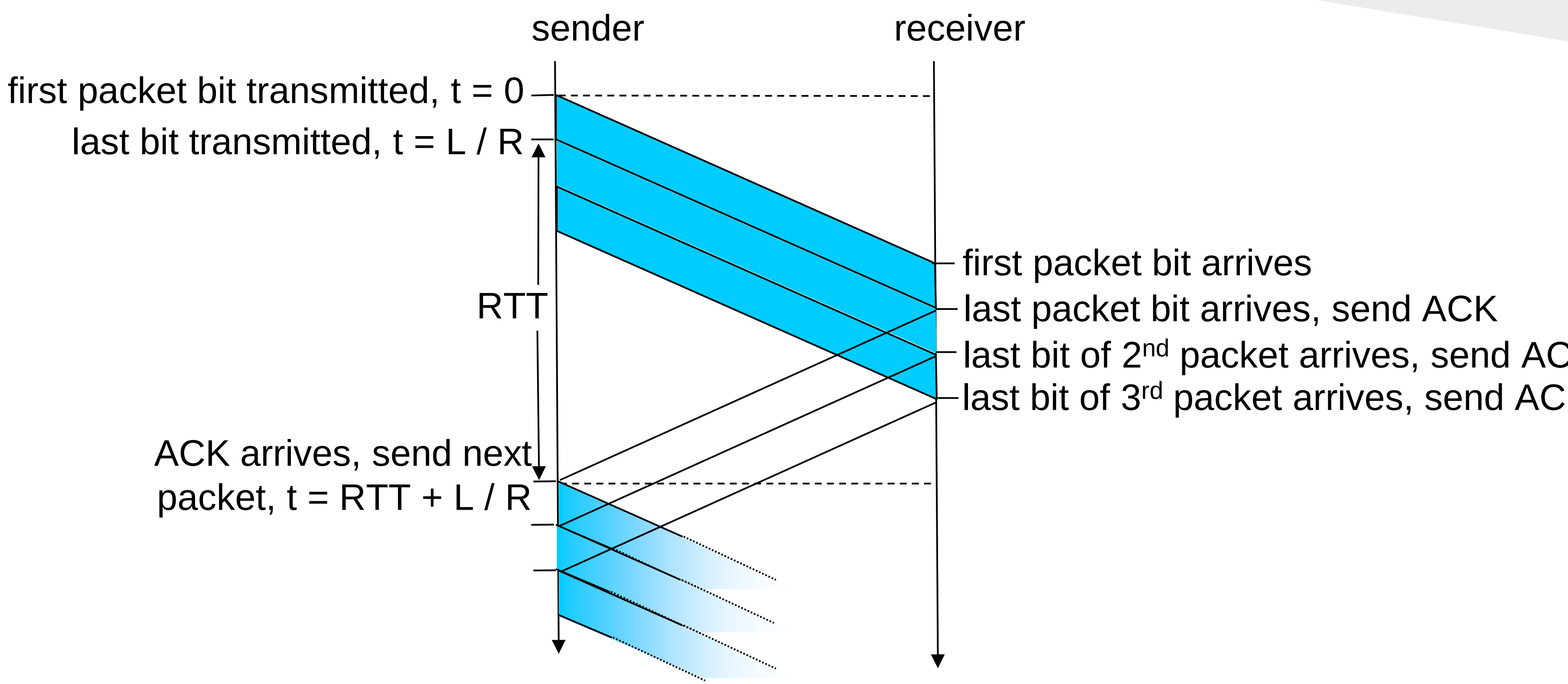
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

two generic forms of pipelined protocols:

go-Back-N, selective repeat



Pipelining: increased utilization



3-packet pipelining increases utilization by a factor of 3!

Pipelined protocols: overview

<u>Go-back-N:</u>

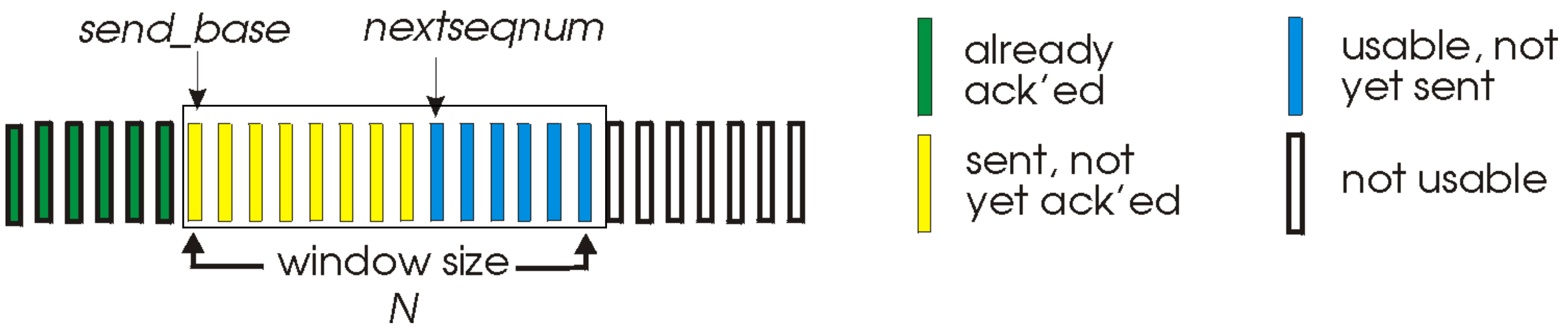
- sender can have up to N unacked packets in pipeline
- receiver only sends *cumulative ack*
  - doesn't ack packet if there's a gap
- sender has timer for oldest unacked packet
  - when timer expires, retransmit *all* unacked packets

<u>Selective Repeat:</u>

- sender can have up to N unack'ed packets in pipeline
- rcvr sends *individual ack* for each packet
- sender maintains timer for each unacked packet
  - when timer expires, retransmit only that unacked packet

Go-Back-N: sender

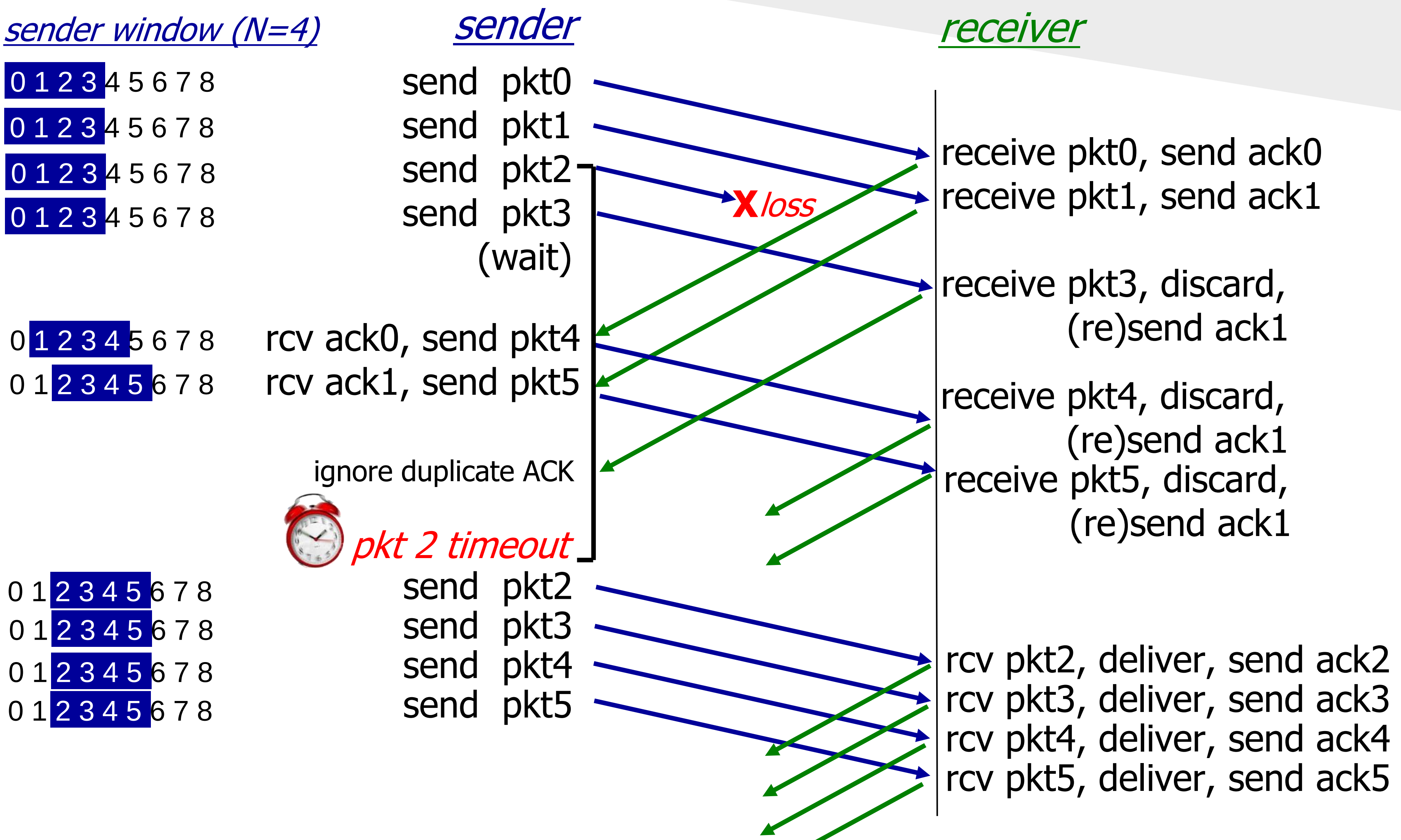
k-bit seq # in pkt header
"window" of up to N, consecutive unack'ed pkts allowed



- ACK(n): ACKs all pkts up to, including seq # n - *"cumulative ACK"*
- may receive duplicate ACKs (see receiver)

- timer for oldest in-flight pkt
- *timeout(n):* retransmit packet n and all higher seq # pkts in window

GBN in action



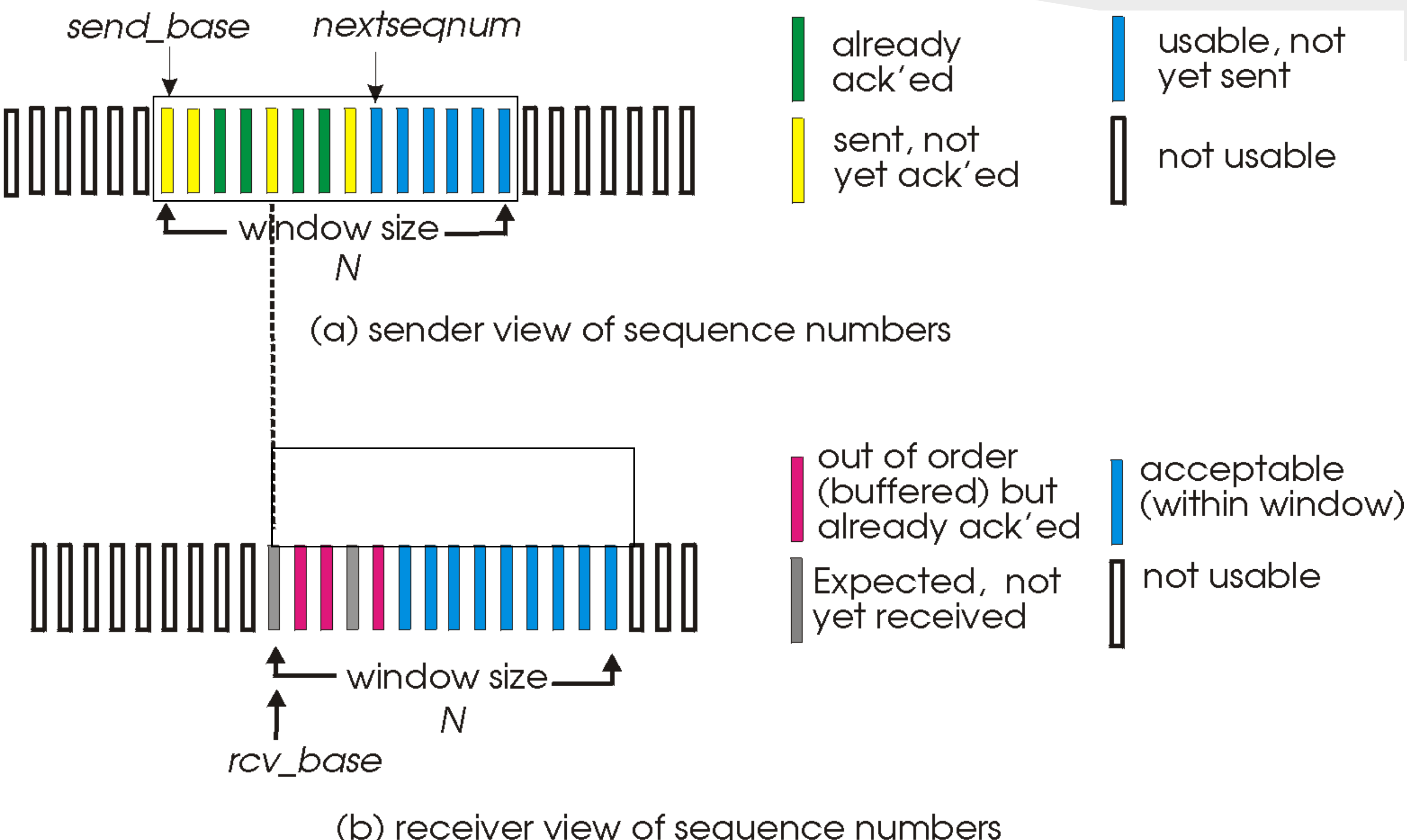
Selective repeat

- receiver *individually* acknowledges all correctly received pkts
- buffers pkts, as needed, for eventual in-order delivery to upper layer

- sender only resends pkts for which ACK not received
- sender timer for each unACKed pkt

- sender window
- *N* consecutive seq #'s
- limits seq #s of sent, unACKed pkts

Selective repeat: sender, receiver windows



Selective repeat

sender

data from above:

- if next available seq # in window, send pkt

timeout(n):

- resend pkt n, restart timer

ACK(n) in [sendbase,sendbase+N]:

- mark pkt n as received
- if n smallest unACKed pkt, advance window base to next unACKed seq #

receiver

pkt n in [rcvbase, rcvbase+N-1]

- send ACK(n)
- out-of-order: buffer
- in-order: deliver (also deliver buffered, in-order pkts), advance window to next not-yet-received pkt

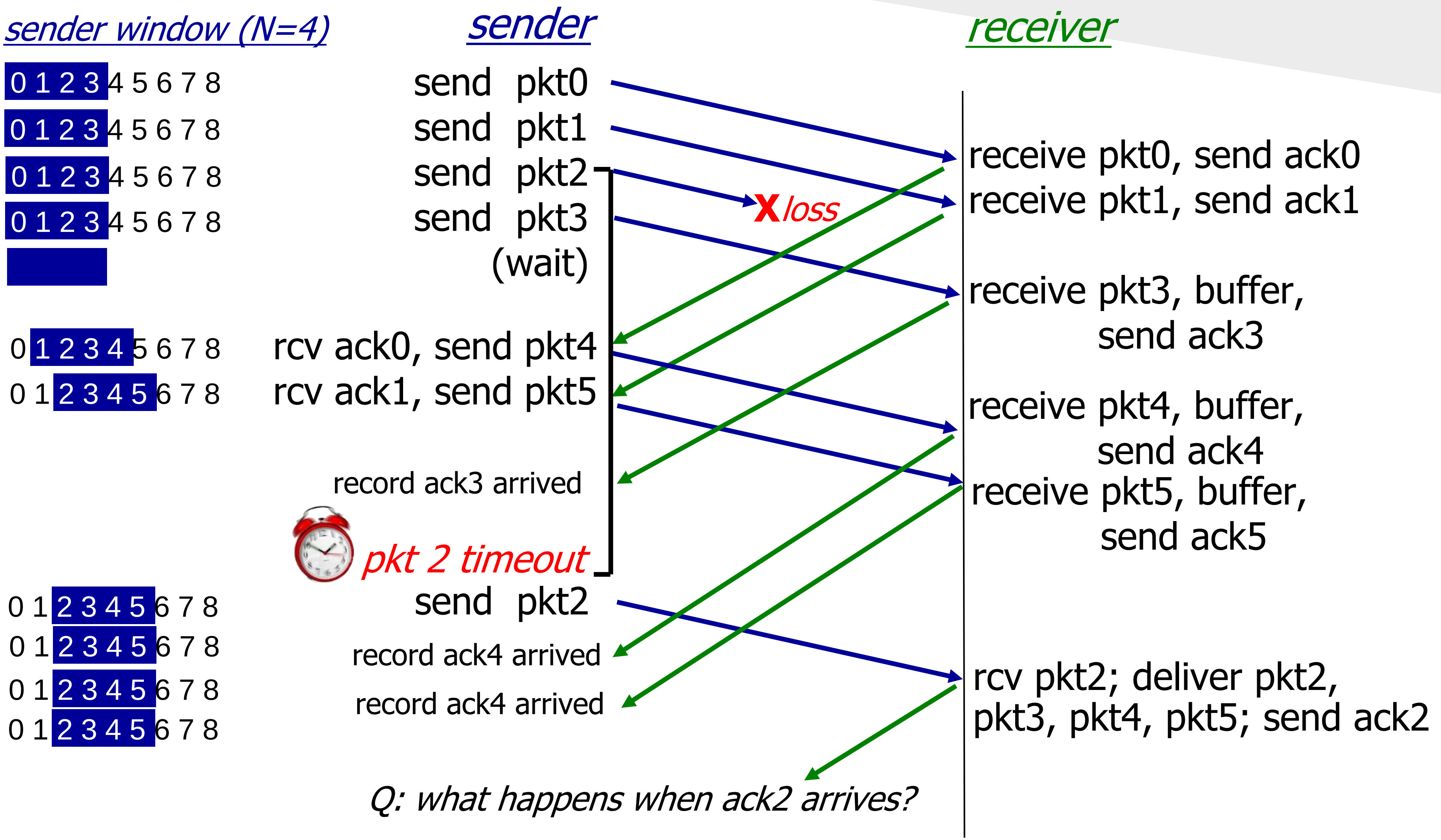
pkt n in [rcvbase-N,rcvbase-1]

- ACK(n)

otherwise:

- ignore

Selective repeat in action



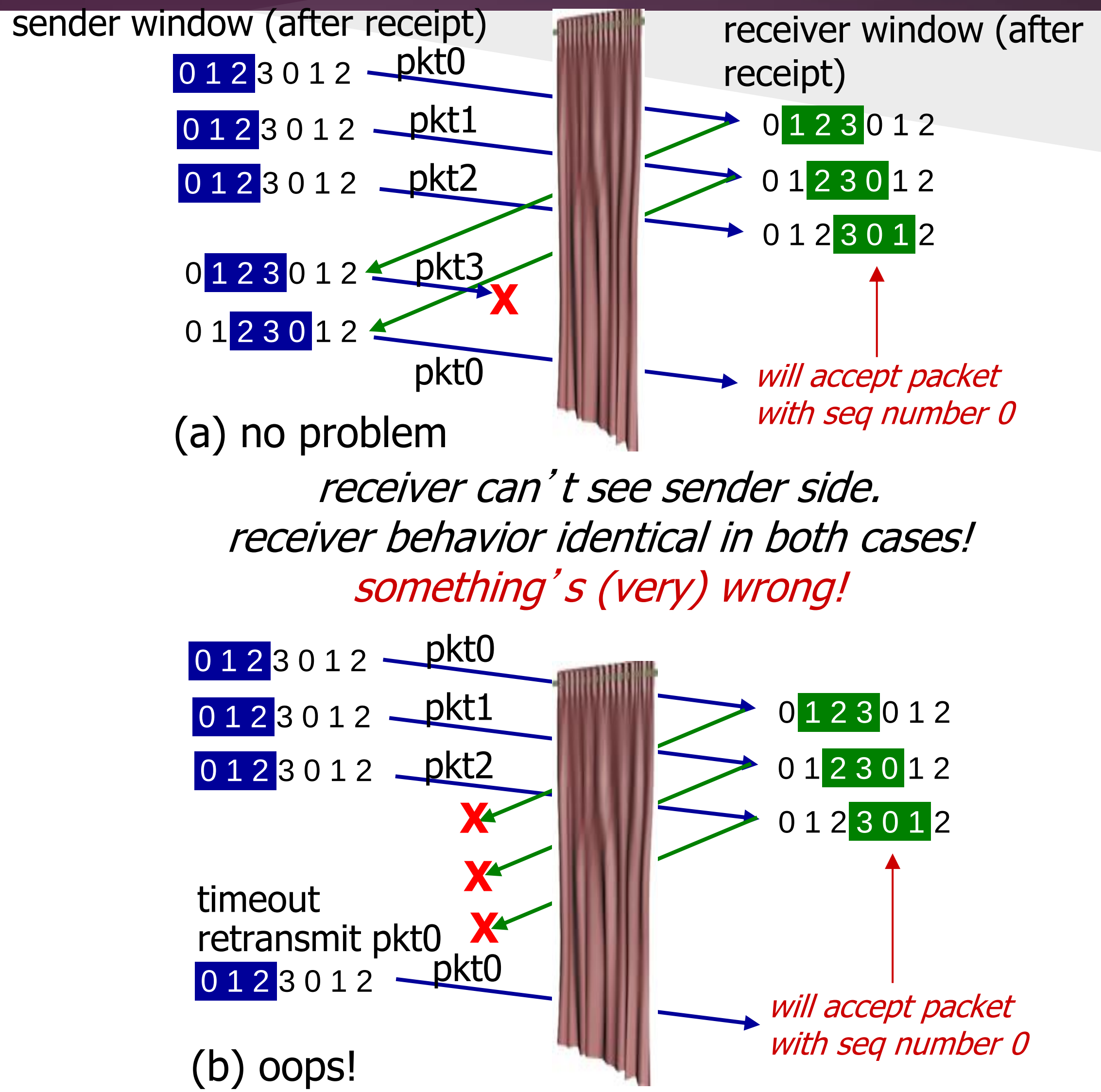
Q: what happens when ack2 arrives?

Selective repeat: dilemma

example:
seq #'s: 0, 1, 2, 3
window size=3

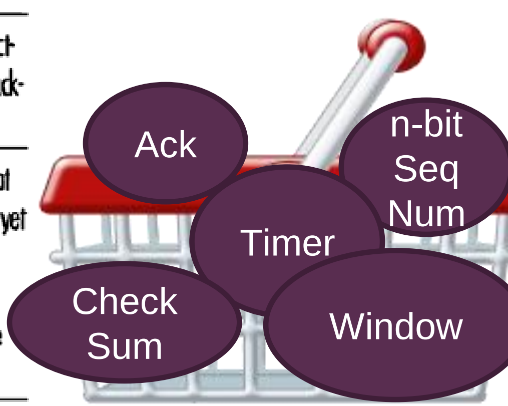
- receiver sees no difference in two scenarios!
- duplicate data accepted as new in (b)

Q: what relationship between seq # size and window size to avoid problem in (b)?



So, in any good transport protocol we need:

Mechanism	Use, Comments
Checksum	Used to detect bit errors in a transmitted packet.
Timer	Used to timeout/retransmit a packet, possibly because the packet (or its ACK) was lost within the channel. Because timeouts can occur when a packet is delayed but not lost (premature timeout), or when a packet has been received by the receiver but the receiver-to-sender ACK has been lost, duplicate copies of a packet may be received by a receiver.
Sequence number	Used for sequential numbering of packets of data flowing from sender to receiver. Gaps in the sequence numbers of received packets allow the receiver to detect a lost packet. Packets with duplicate sequence numbers allow the receiver to detect duplicate copies of a packet.
Acknowledgment	Used by the receiver to tell the sender that a packet or set of packets has been received correctly. Acknowledgments will typically carry the sequence number of the packet or packets being acknowledged. Acknowledgments may be individual or cumulative, depending on the protocol.
Negative acknowledgment	Used by the receiver to tell the sender that a packet has not been received correctly. Negative acknowledgments will typically carry the sequence number of the packet that was not received correctly.
Window, pipelining	The sender may be restricted to sending only packets with sequence numbers that fall within a given range. By allowing multiple packets to be transmitted but not yet acknowledged, sender utilization can be increased over a stop-and-wait mode of operation. We'll see shortly that the window size may be set on the basis of the receiver's ability to receive and buffer messages, or the level of congestion in the network, or both.



שאלות

מה תפקידו של ה-checksum?
תשובה: לאתר חבילות לא תקינות (הן מידע והן ack)

מה תפקידו של ה-ack?
תשובה: אישור על קבלת חבילות

מה זה cumulative ack?
תשובה: אישור על החבילות שקיבלתי עד כה

מדוע ישנה אפשרות לוותר על NACK?
תשובה: ack + seqnum נותן את אותו פתרון

שאלות

מה תפקידו של ה-timeout?
תשובה: להתגבר על מצב של אובדן חבילות

מה הקשר הרצוי בין seqnum לגודל החלון?
תשובה: פי 2, על מנת להמנע ממצב של קבלת חבילה שלא נשלחה.

מהו קצב השידור האפקטיבי של פרוטוקול?
תשובה: כמה מידע שהוא שולח ביחידת זמן.

כיצד מחשבים נצילות של פרוטוקול?
תשובה: פרק זמן שליחת המידע חלקי פרק הזמן הכולל או קצב השידור האפקטיבי חלקי קצב השידור המקסימאלי.

connection-oriented transport: TCP

TCP: Overview

- 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 reliable data transfer

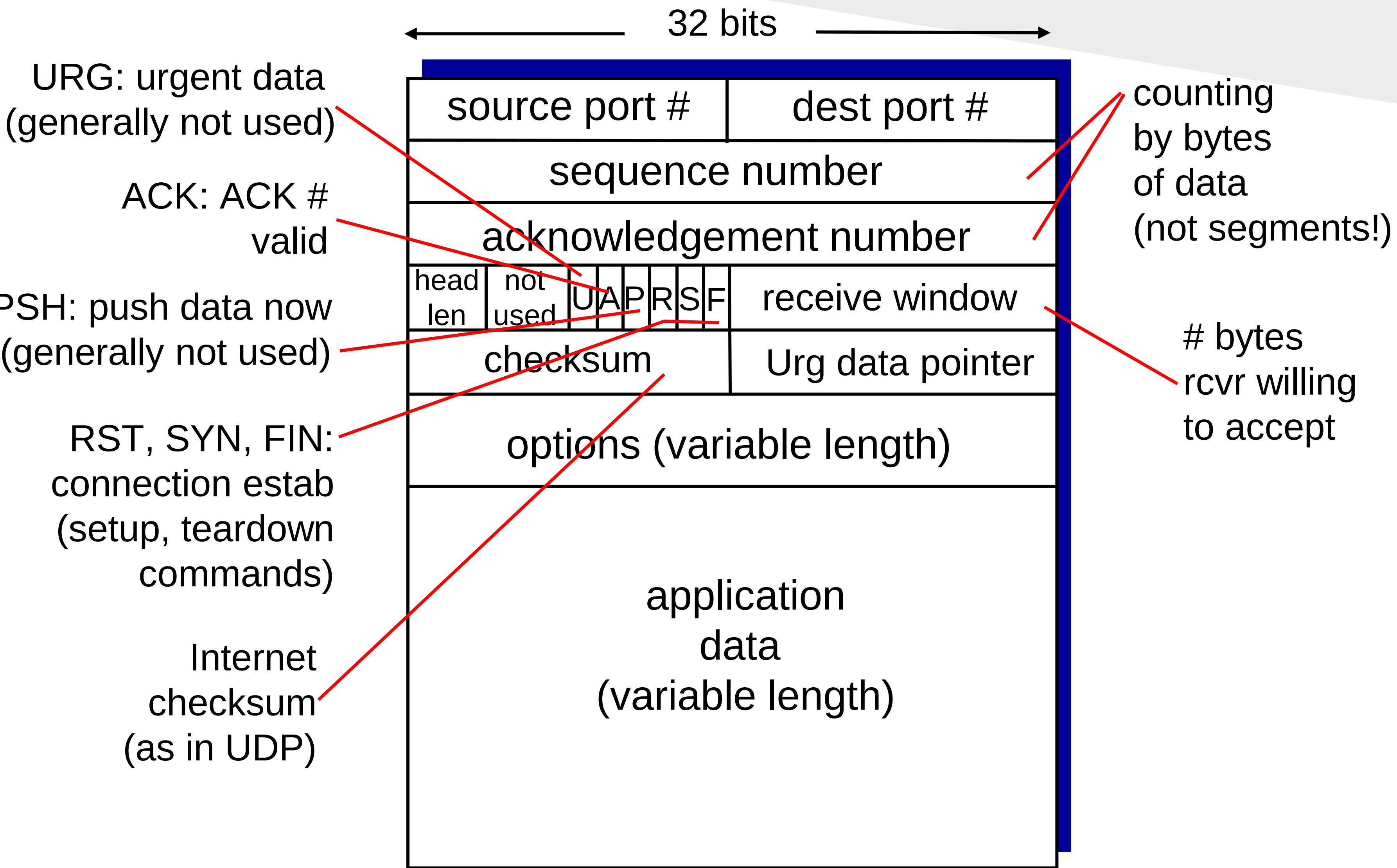
- **TCP creates reliable service on top of IP's unreliable service**
- **Pipelined segments**
- **Cumulative acks**
- **TCP uses single retransmission timer**
- **TCP spec doesn't say, how receiver handles out-of-order segments… up to implementer**

TCP - Header

TCP segment - How??

Ports
Seq
Ack
congestion and flow control

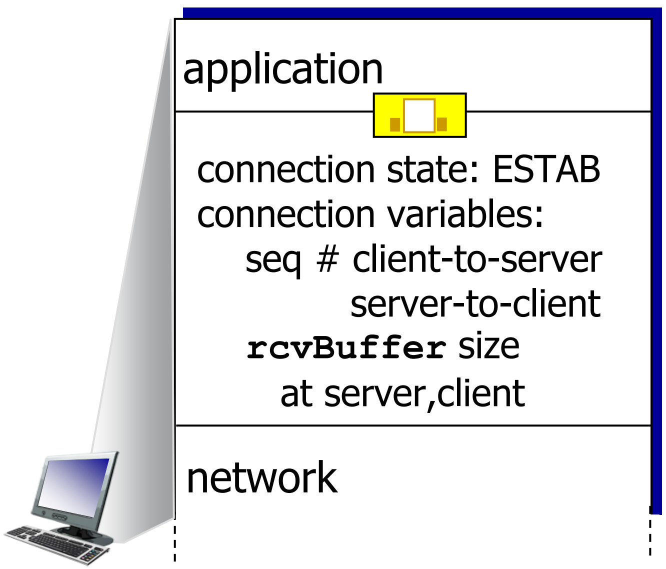
TCP segment structure



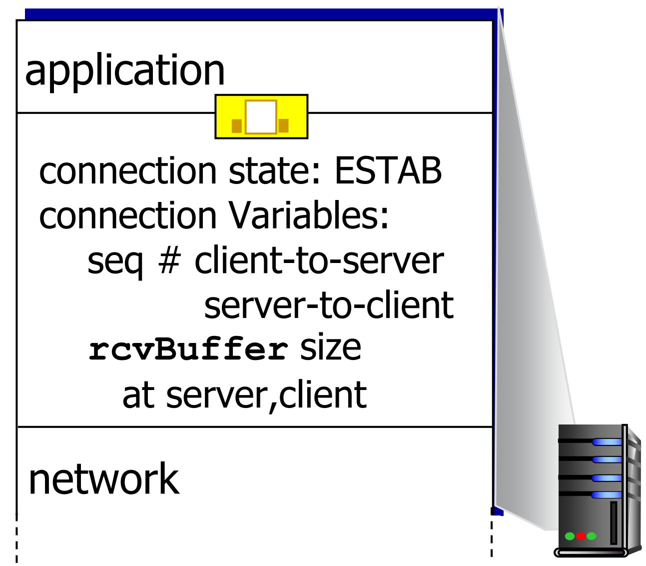
TCP - connection management

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");
```



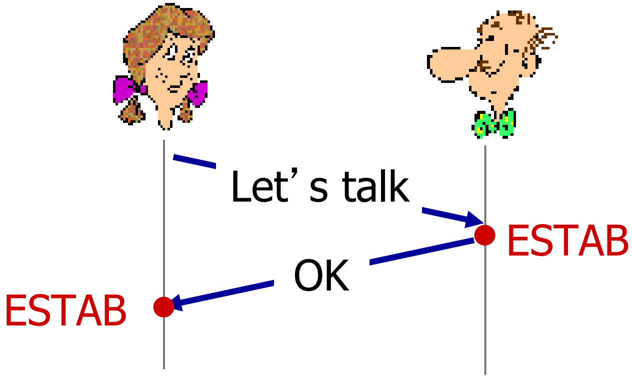

```
Socket connectionSocket =
  welcomeSocket.accept();
```

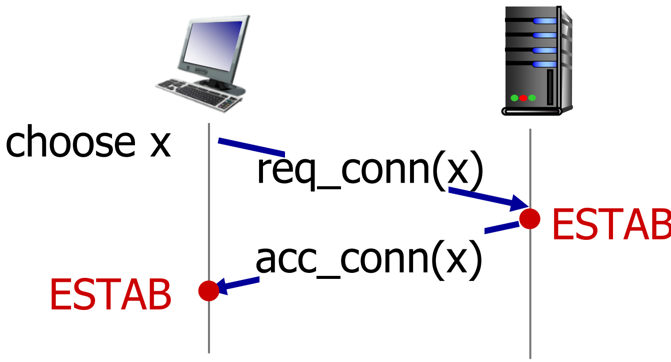
Agreeing to establish a connection

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

2-way handshake:



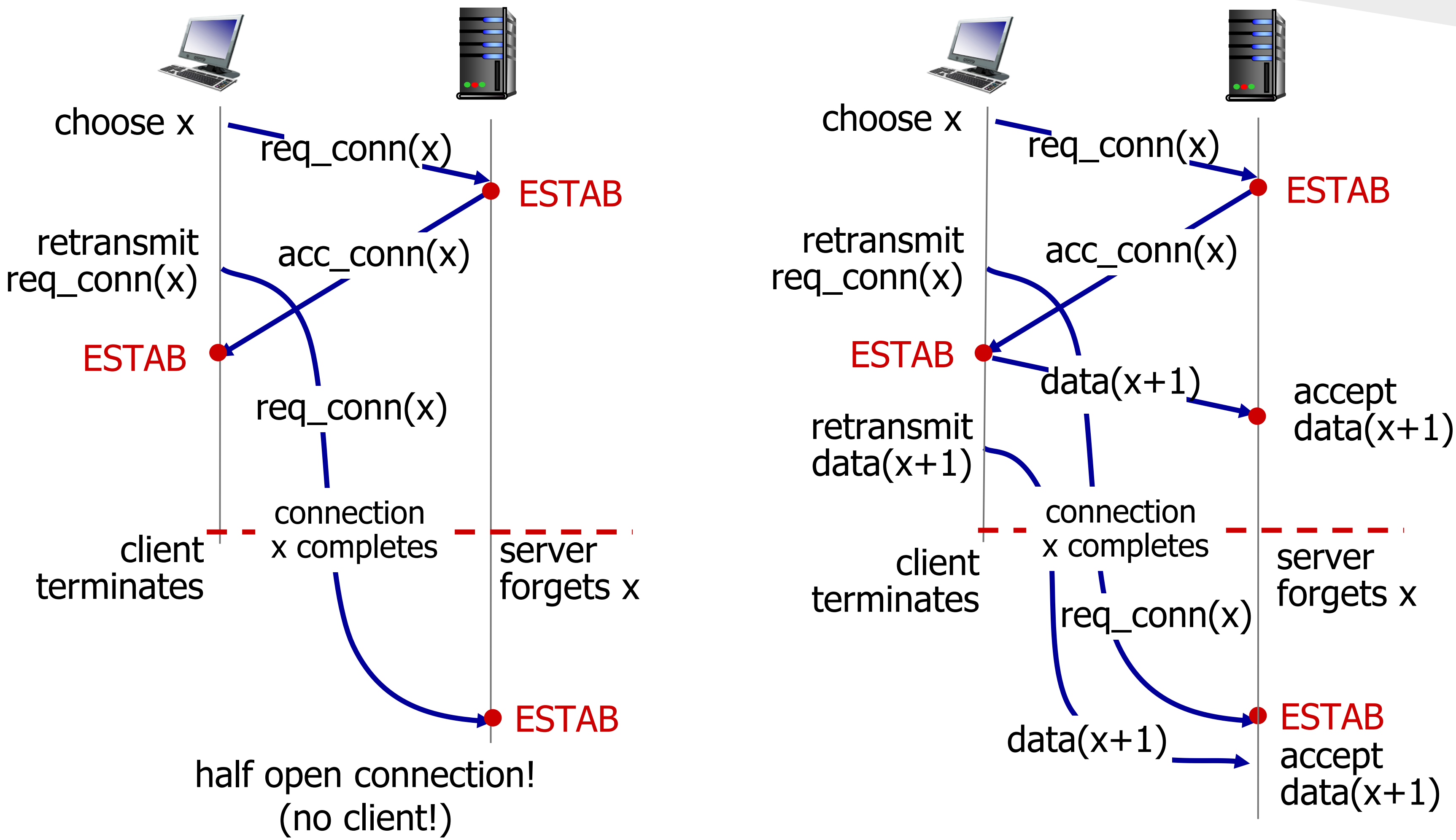


Delayed Duplicates Problem

- A user asks for a connection
- Due to congestion the packet is caught in a *traffic jam*
- The user asks again for the connection
- Destination accepts 2^{nd} connection request
- User sends info to dest.
- Info gets caught in a traffic jam
- User sends info again
- Dest receives the info
- Connection is closed by both parties
- The original connection request and user info find their way to the destination.

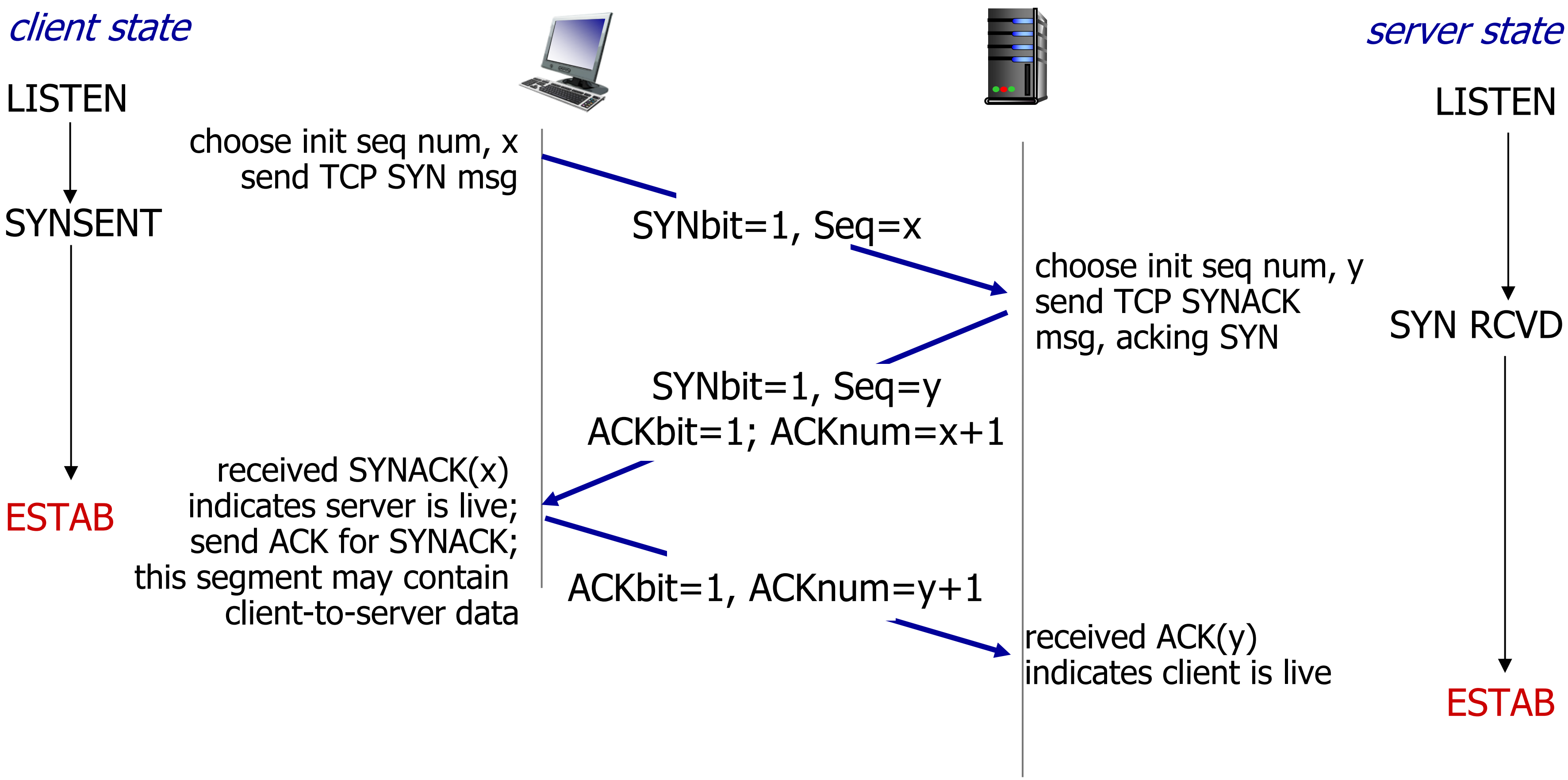
Agreeing to establish a connection

2-way handshake failure scenarios:



TCP 3-way handshake

Therefore, what we need?



TCP Connection Setup Example

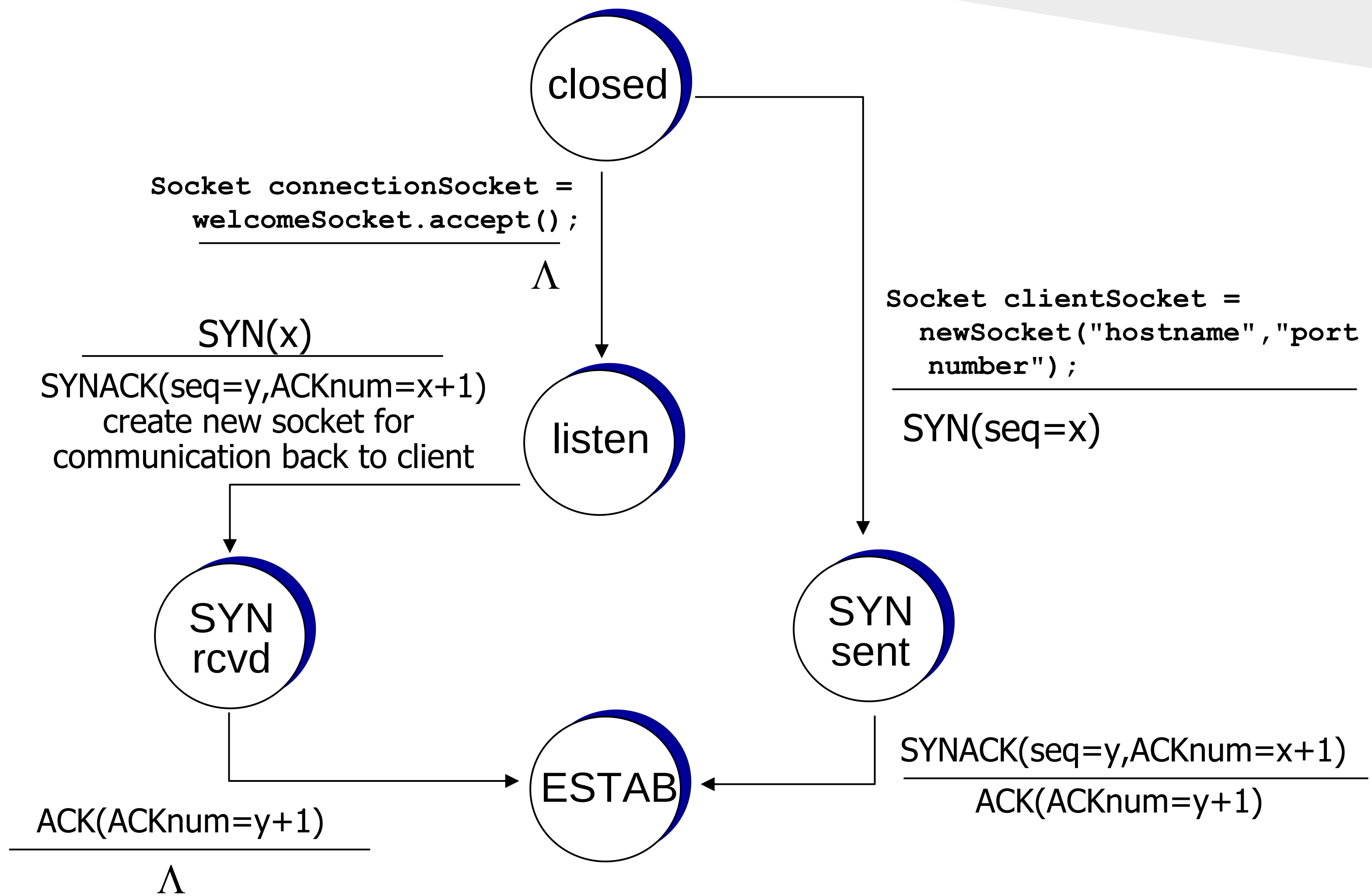
```
09:23:33.042318 IP 128.2.222.198.3123 > 192.216.219.96.80:
S 4019802004:4019802004(0) win 65535
<mss 1260,nop,nop,sackOK> (DF)

09:23:33.118329 IP 192.216.219.96.80 > 128.2.222.198.3123:
S 3428951569:3428951569(0) ack 4019802005 win 5840
<mss 1460,nop,nop,sackOK> (DF)

09:23:33.118405 IP 128.2.222.198.3123 > 192.216.219.96.80:
. ack 3428951570 win 65535 (DF)
```

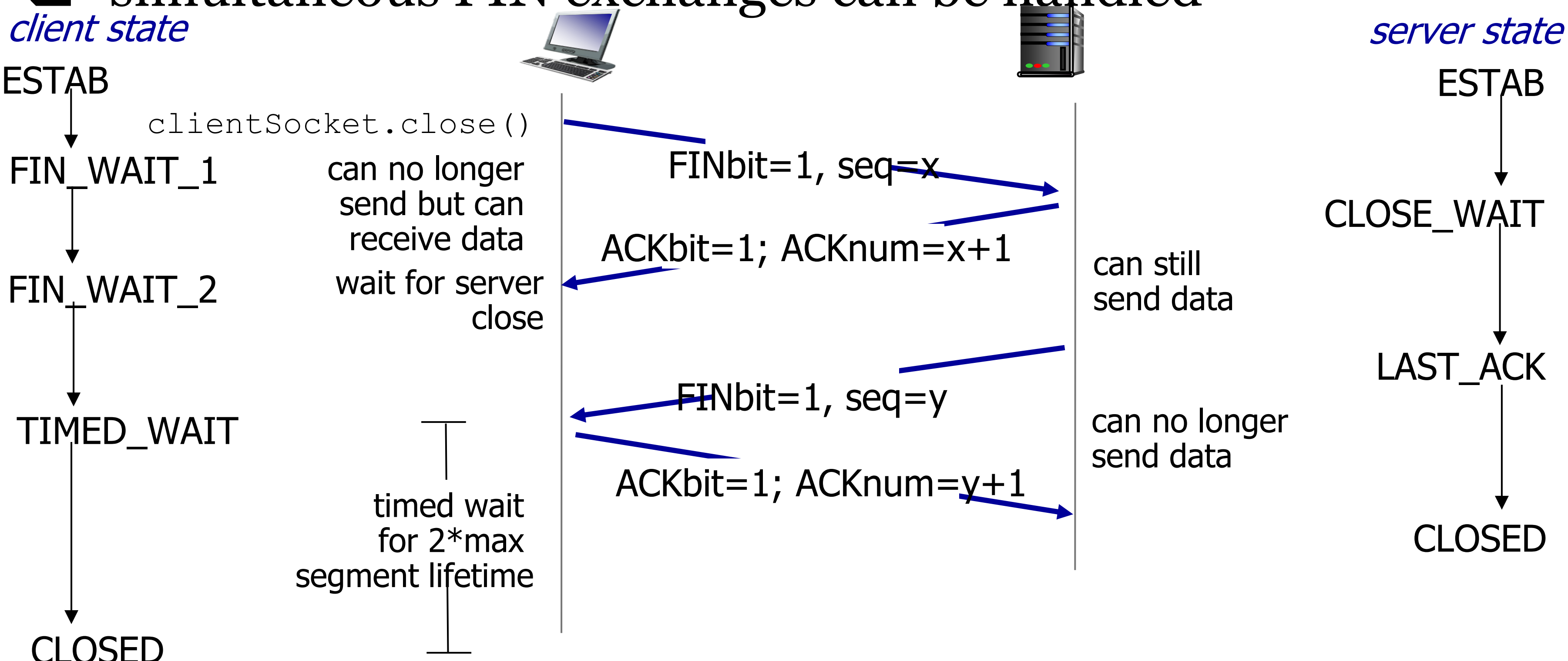
- Client SYN
  - SeqC: Seq. #4019802004, window 65535, max. seg. 1260
- Server SYN-ACK+SYN
  - Receive: #4019802005 (= SeqC+1)
  - SeqS: Seq. #3428951569, window 5840, max. seg. 1460
- Client SYN-ACK
  - Receive: #3428951570 (= SeqS+1)

TCP 3-way handshake: FSM

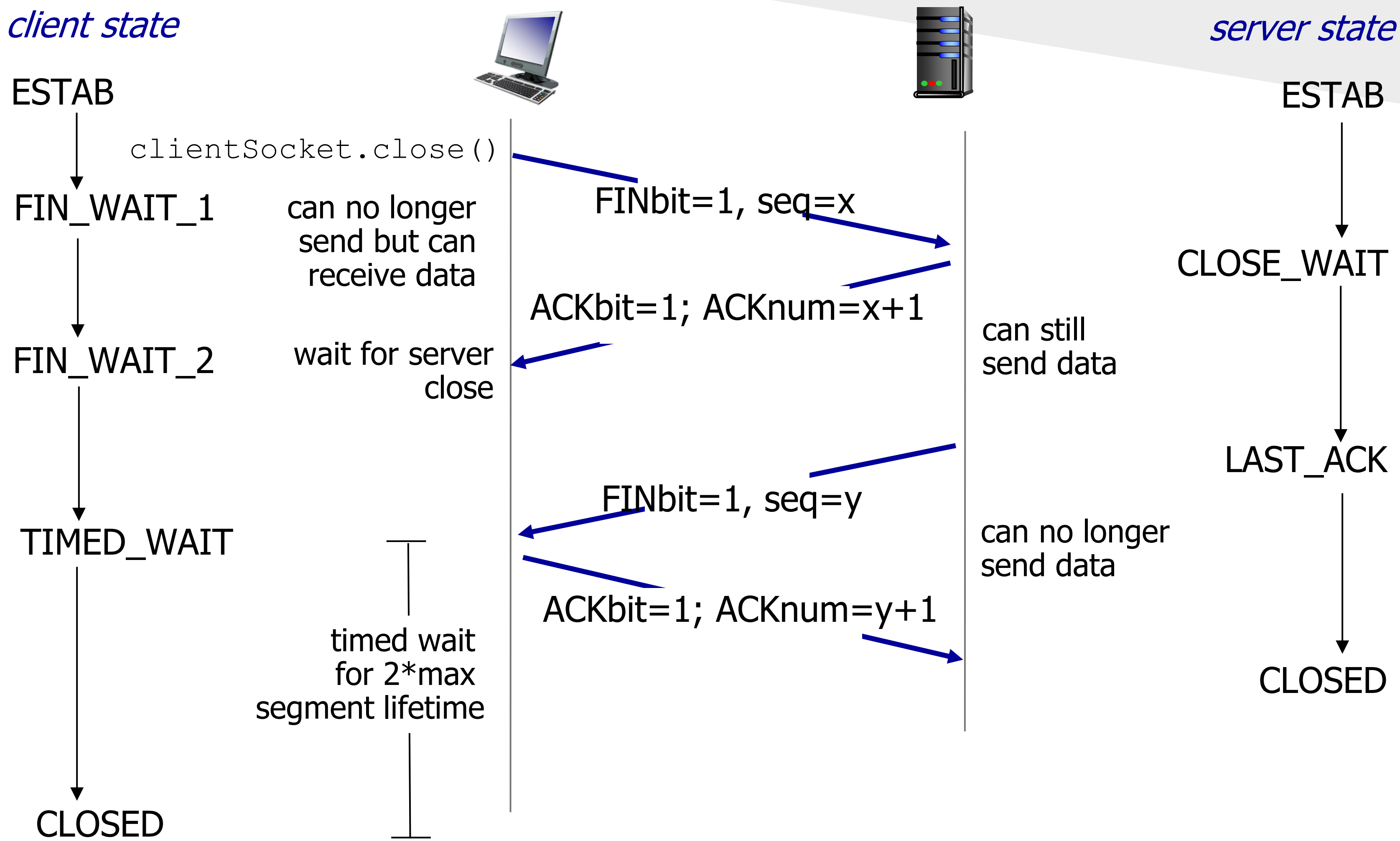


TCP: closing a connection

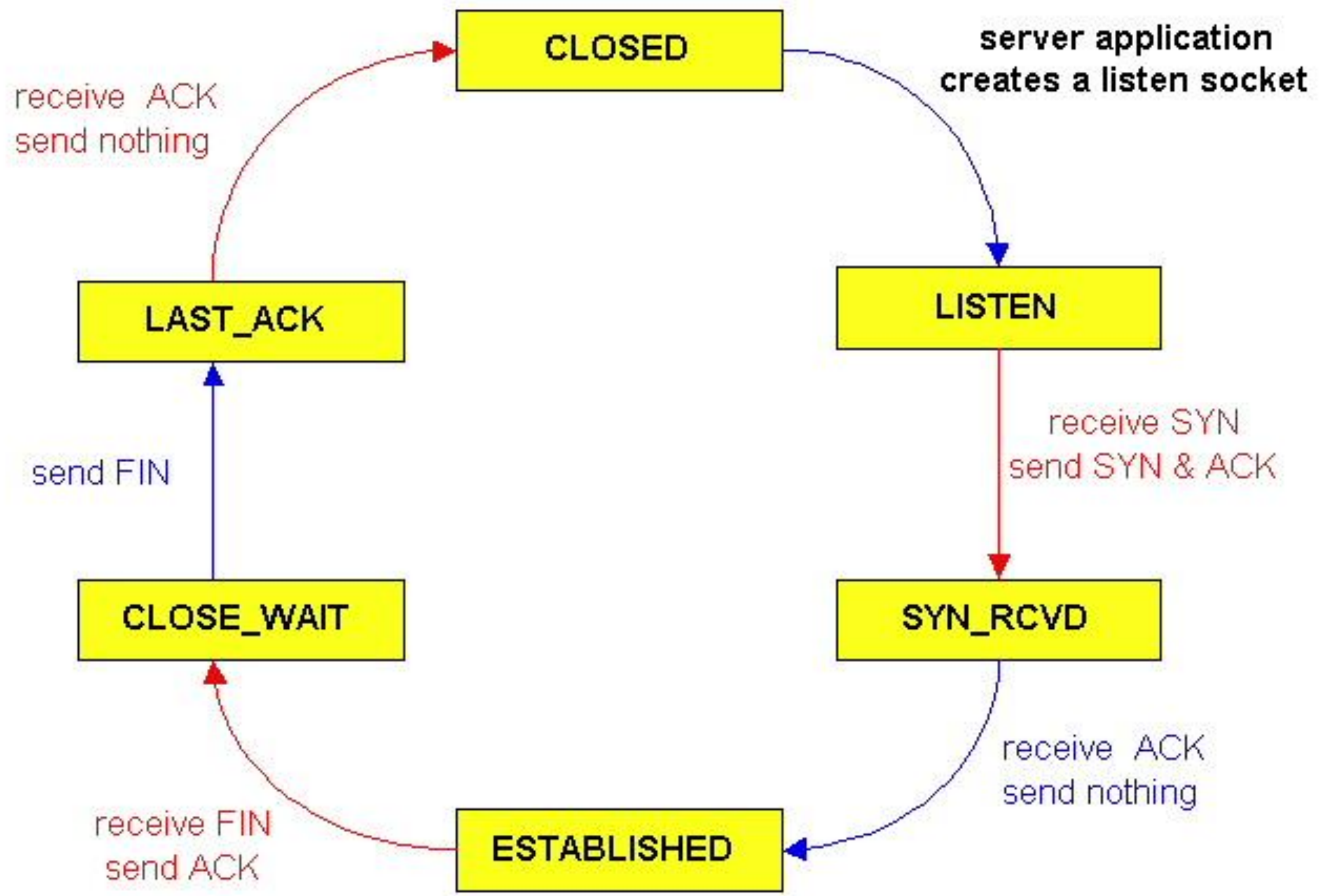
- Client, server each close their side of connection
  - send TCP segment with FIN bit = 1
- Respond to received FIN with ACK
  - on receiving FIN, ACK can be combined with own FI
- simultaneous FIN exchanges can be handled

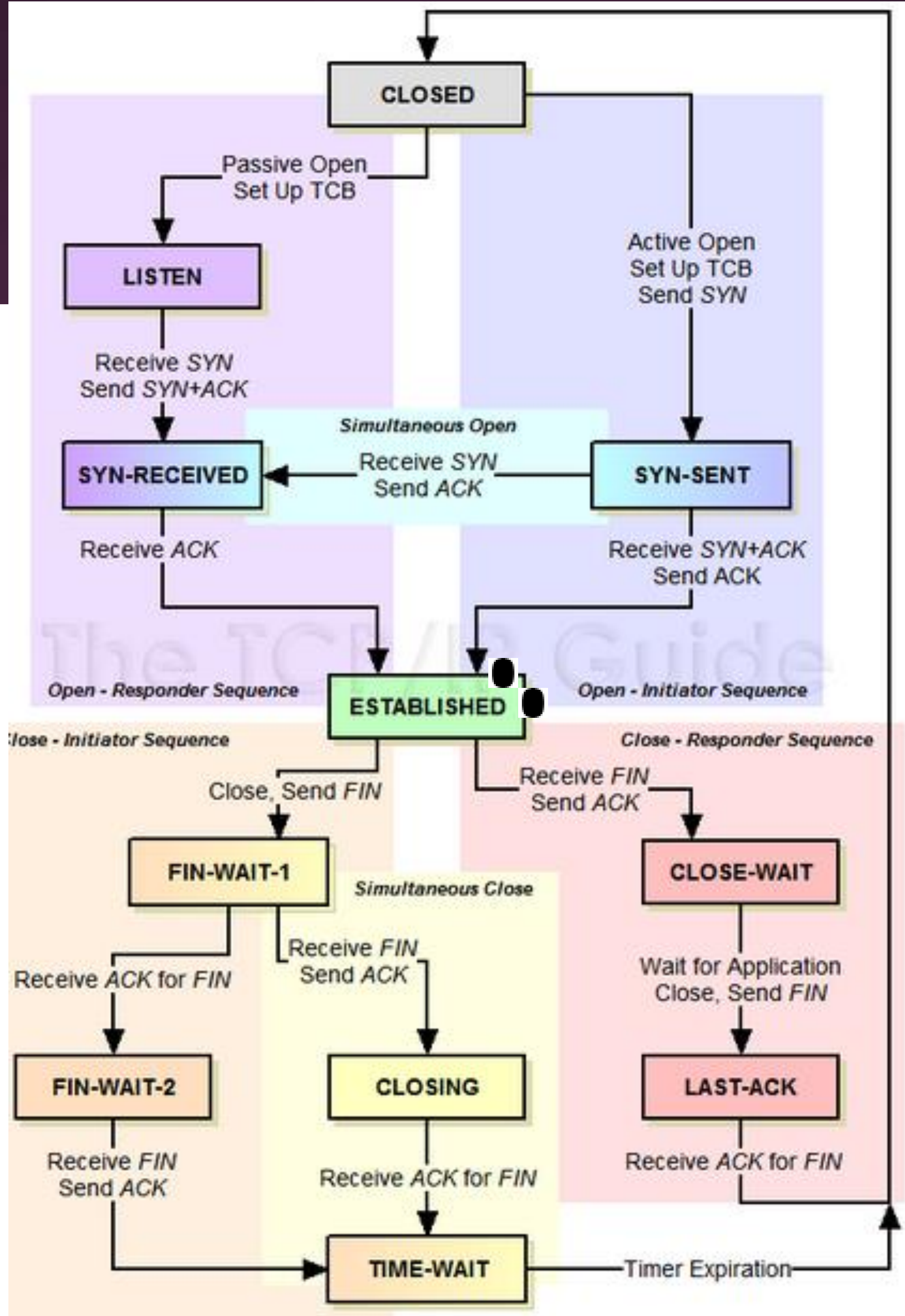


TCP: closing a connection



TCP – Life Cycle




CLOSED
Passive Open
Set Up TCB
LISTEN
Active Open
Set Up TCB
Send SYN
Receive SYN
Send SYN+ACK
Simultaneous Open
SYN-RECEIVED
Receive SYN
Send ACK
SYN-SENT
Receive ACK
Receive SYN+ACK
Send ACK
Open - Responder Sequence
ESTABLISHED
Open - Initiator Sequence
Close - Initiator Sequence
Close - Responder Sequence
Close, Send FIN
Receive FIN
Send ACK
FIN-WAIT-1
CLOSE-WAIT
Simultaneous Close
Receive FIN
Send ACK
Receive ACK for FIN
Wait for Application
Close, Send FIN
FIN-WAIT-2
CLOSING
LAST-ACK
Receive FIN
Send ACK
Receive ACK for FIN
Receive ACK for FIN
TIME-WAIT
Timer Expiration

TCP seq. numbers, ACKs

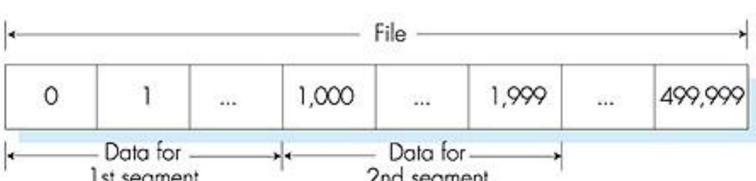
sequence numbers:

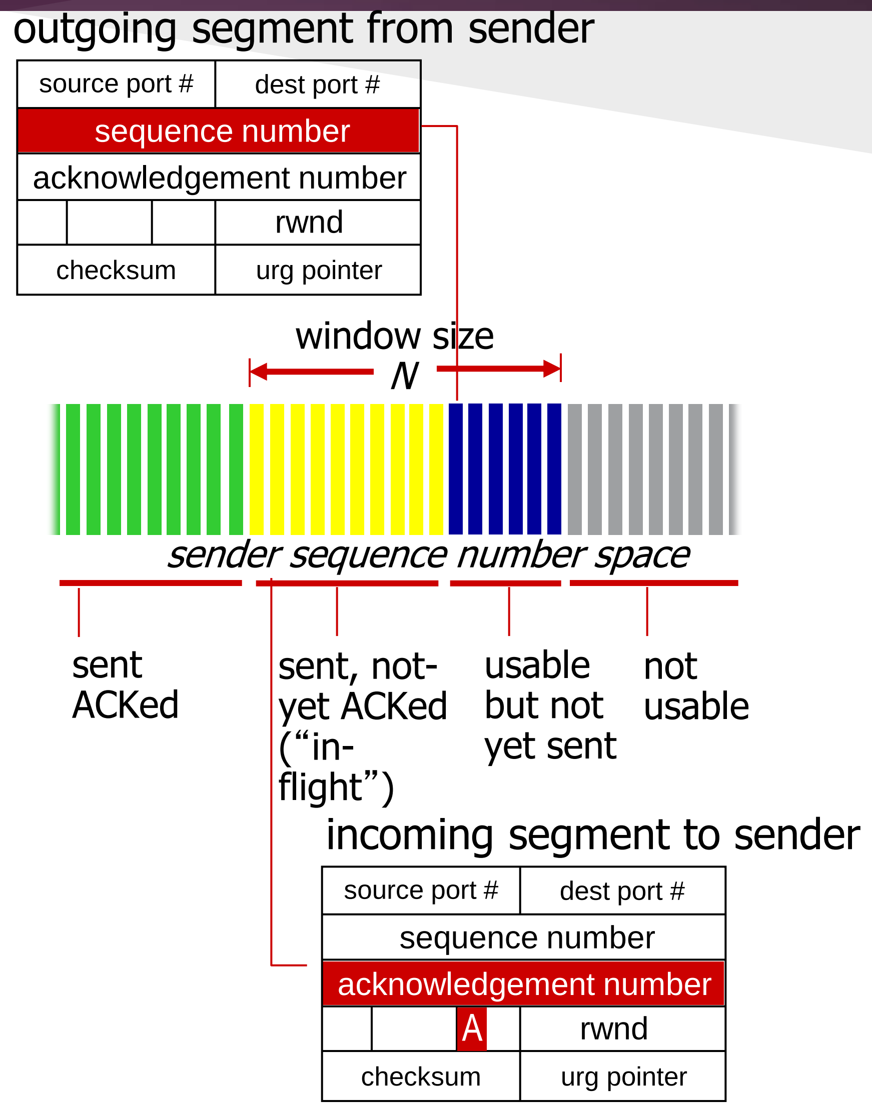
- byte stream "number" of first byte in segment's data

- The sequence number for a segment is the first byte-stream # of the first byte in the segment.

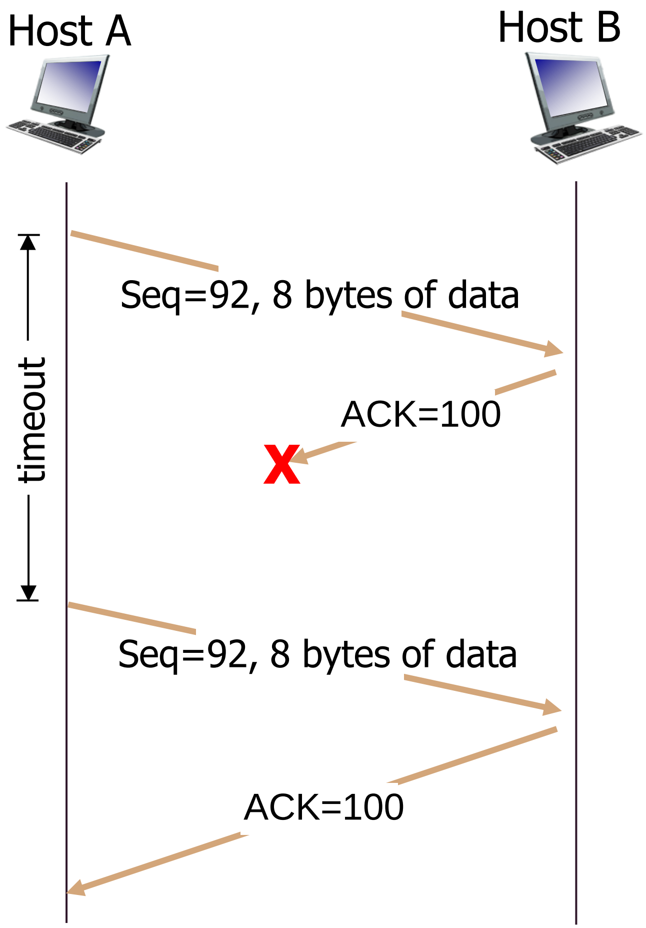
acknowledgements:

- seq # of next byte expected from other side
- cumulative ACK

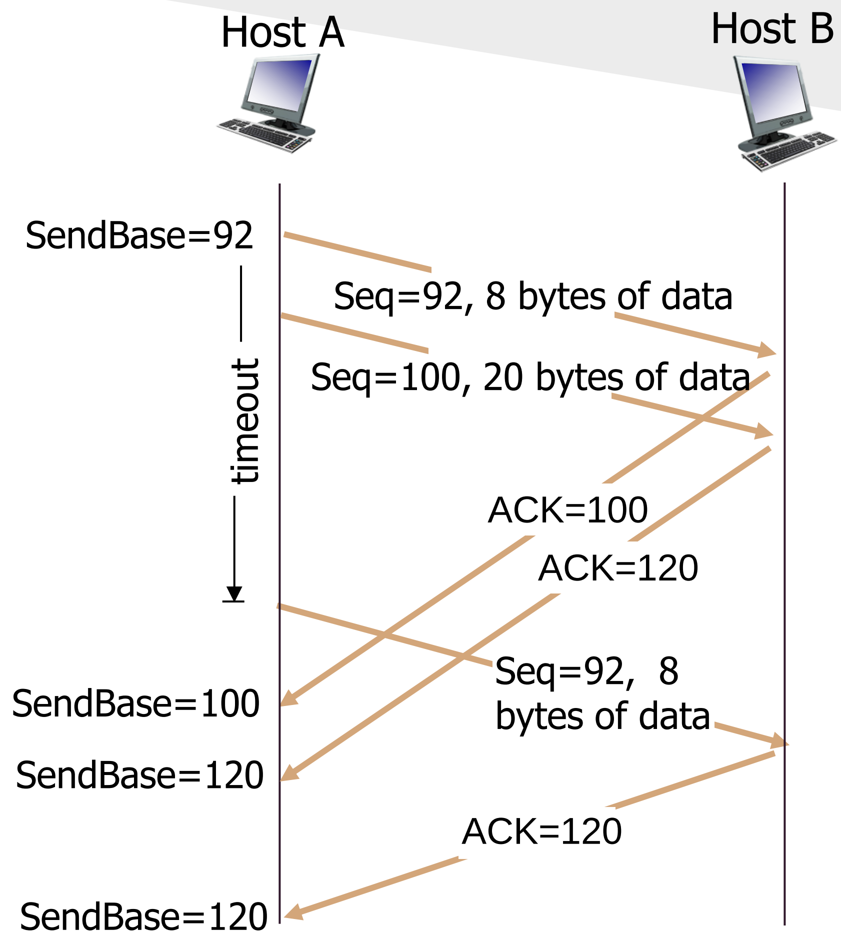




TCP: retransmission scenarios

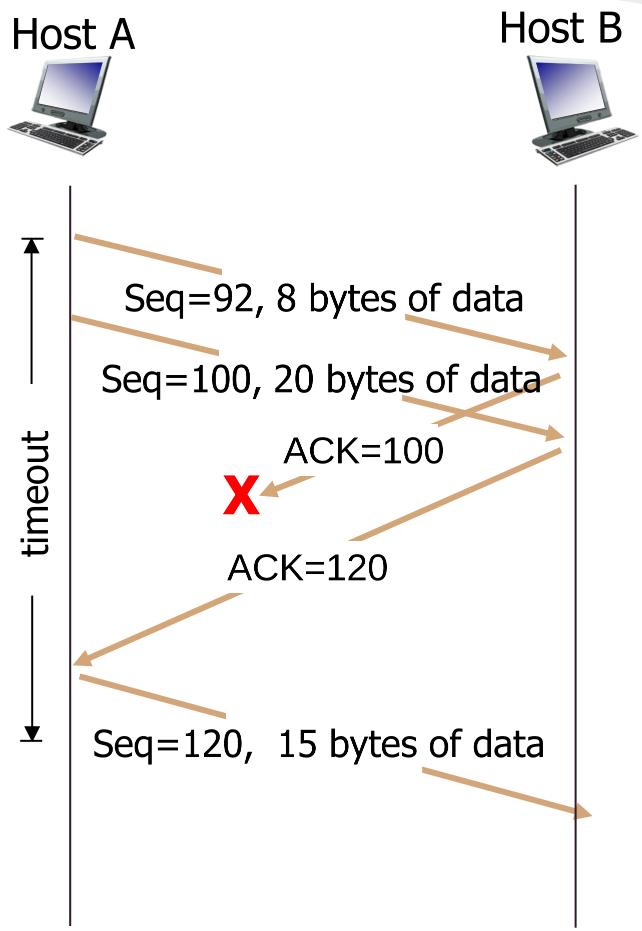


lost ACK scenario



premature timeout

TCP: retransmission scenarios



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

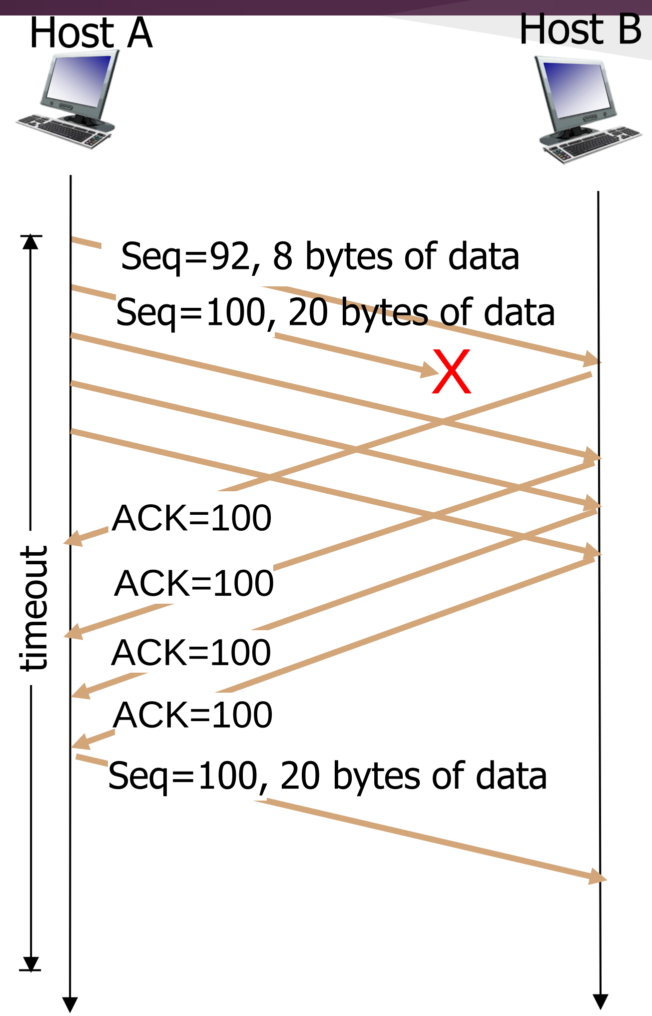
- time-out period often relatively long:
  - long delay before resending lost packet
- WHAT TO DO….
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

TCP fast retransmit

if sender receives 3 ACKs for same data ("triple duplicate ACKs"), resend unacked segment with smallest seq #

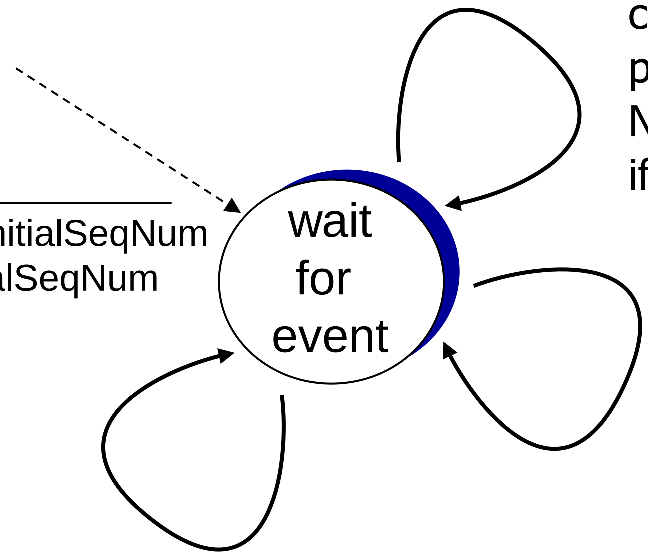
- likely that unacked segment lost, so don't wait for timeout

TCP fast retransmit



fast retransmit after sender receipt of triple duplicate ACK

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 - Timeout

TCP timeout

Q: How to set TCP timeout value?

A: longer than RTT

but RTT varies.....?

- too short: premature timeout, unnecessary retransmissions
- too long: slow reaction to segment loss

RTT... I know how to calculate but estimate RTT?

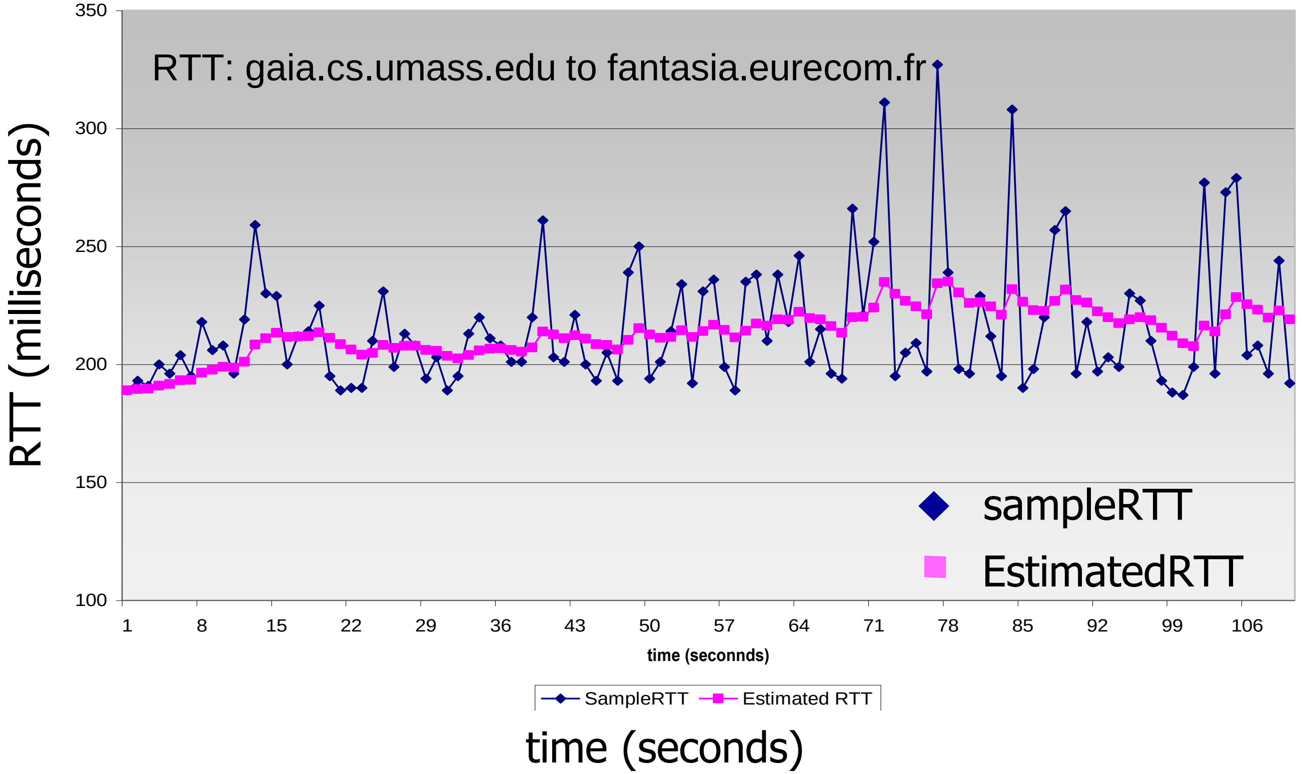
`SampleRTT`: measured time from segment transmission until ACK receipt ignore retransmissions.
Good...?

SampleRTT will vary, want estimated RTT "smoother" average several recent measurements, not just current SampleRTT

TCP - estimate timeout phase 1

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

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



TCP - estimate timeout phase 2

timeout interval: **`EstimatedRTT`** plus "safety margin"

large variation in **`EstimatedRTT`** -> larger safety margin

estimate SampleRTT deviation from EstimatedRTT:

```
DevRTT = (1-β)*DevRTT +
              β*|SampleRTT-EstimatedRTT|
              (typically, β = 0.25)
```

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

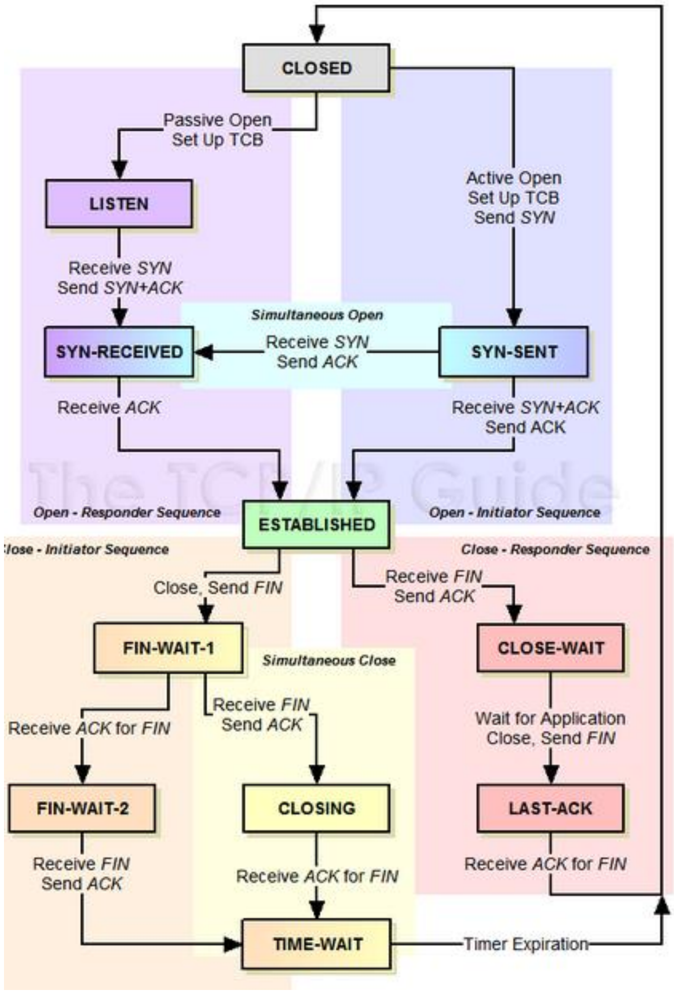
estimated RTT

"safety margin"

What else do we need to know?

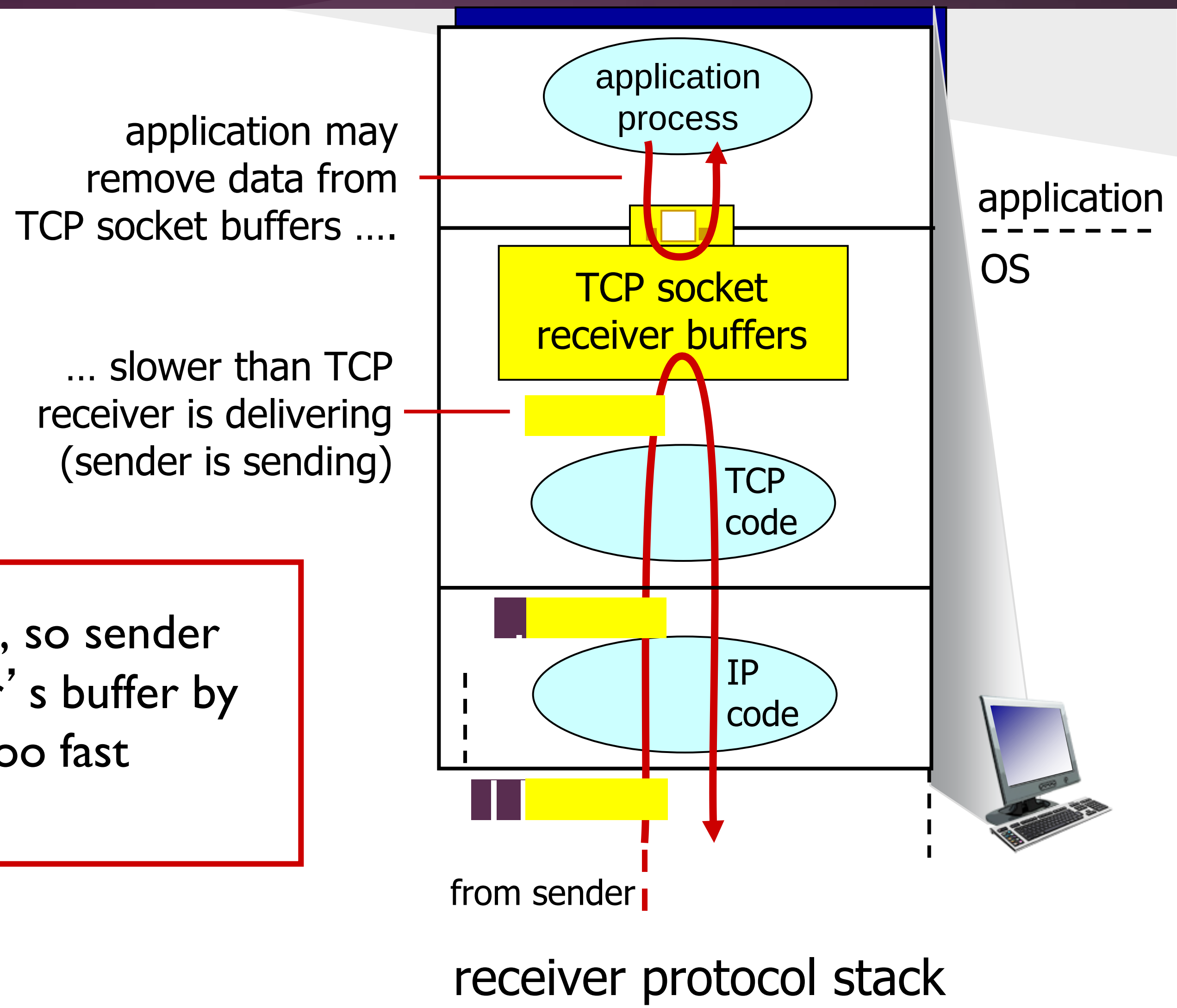
- We know
  - How to open session
  - How to send segments and receive ACKs
  - How to close session
  - How to estimate the timer`
- What to do when there is no problems in the network and the receiver gets segments … can you talk without breath?
- What to do when the network has problems…. Can you talk in the class with others

- Flow Control
- Congestion Control



TCP - flow control

TCP flow control

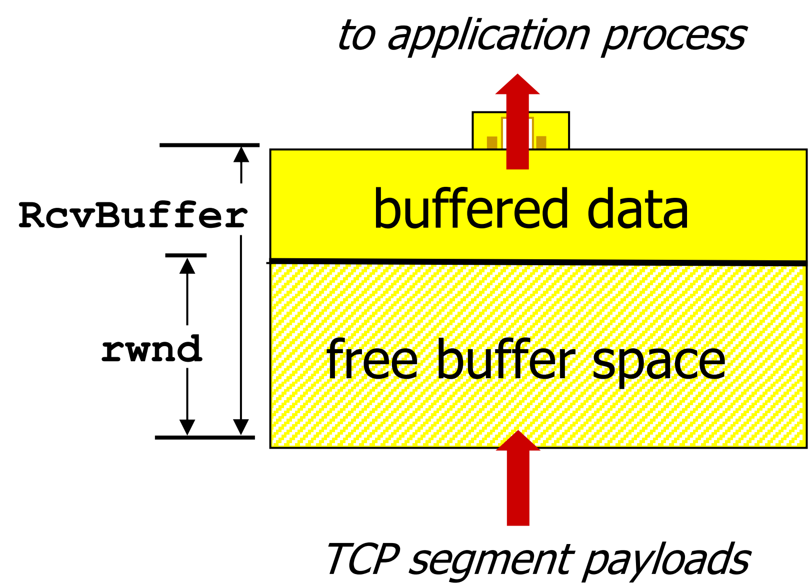


flow control

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

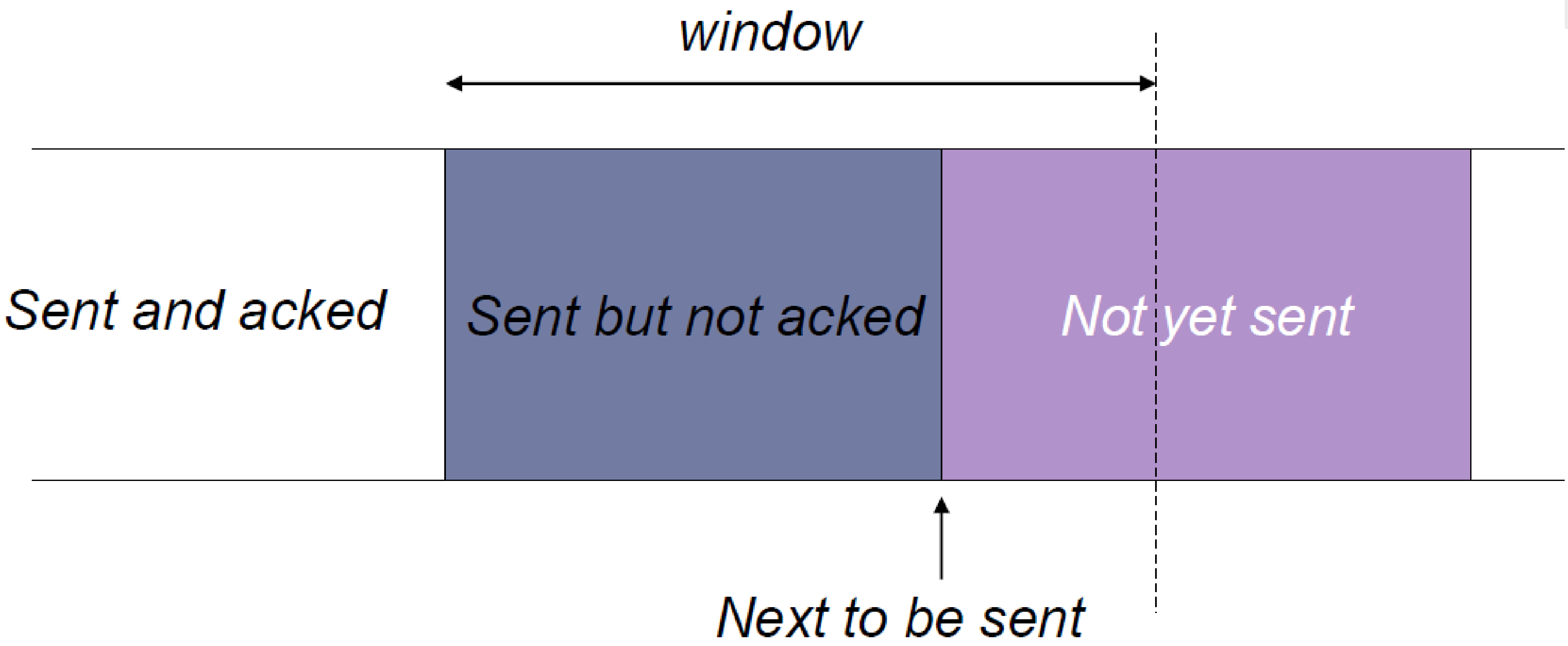
TCP flow control – What to Do

- 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

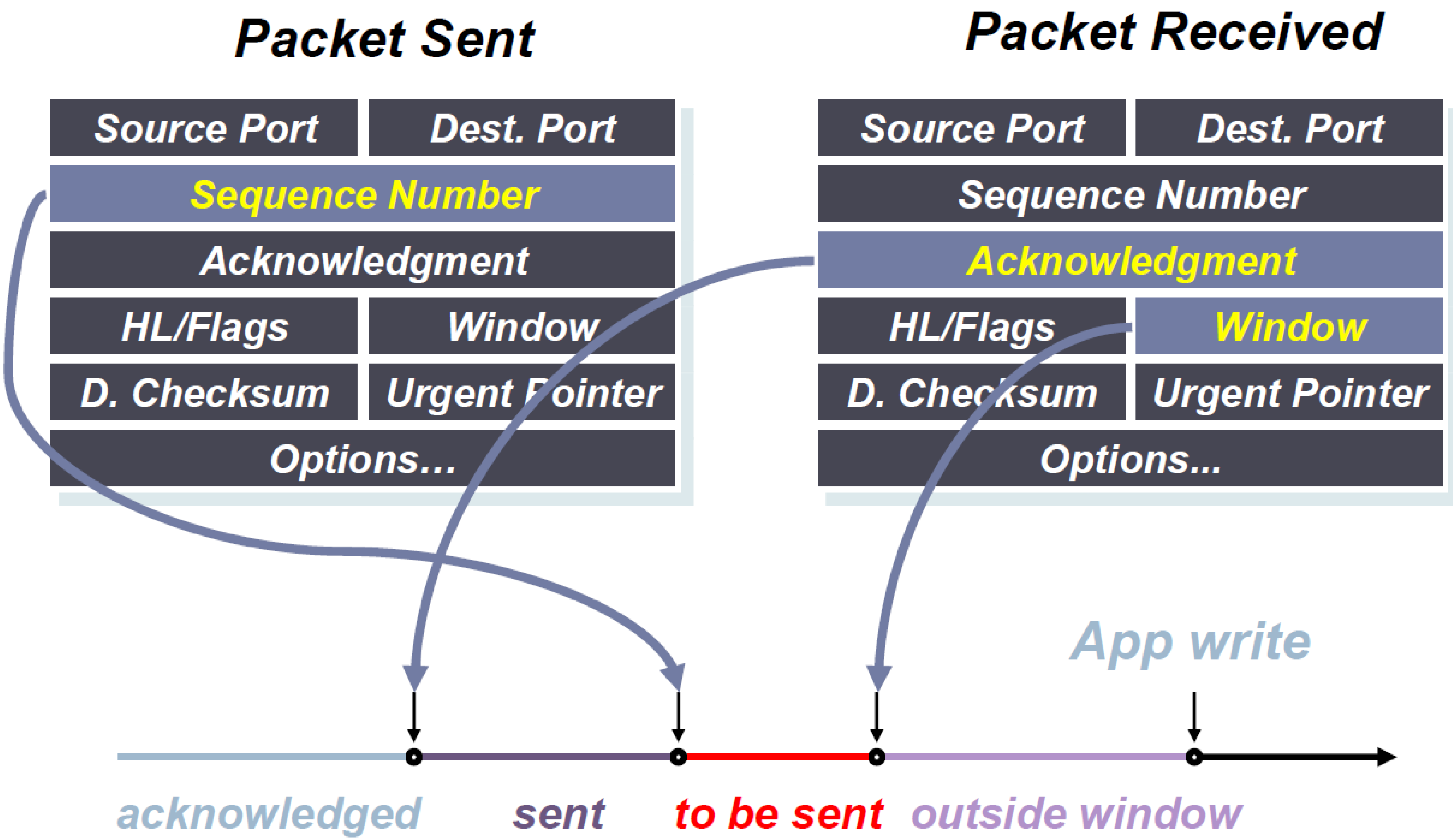


receiver-side buffering

TCP flow control – Sender Side



TCP flow control – Sender Side



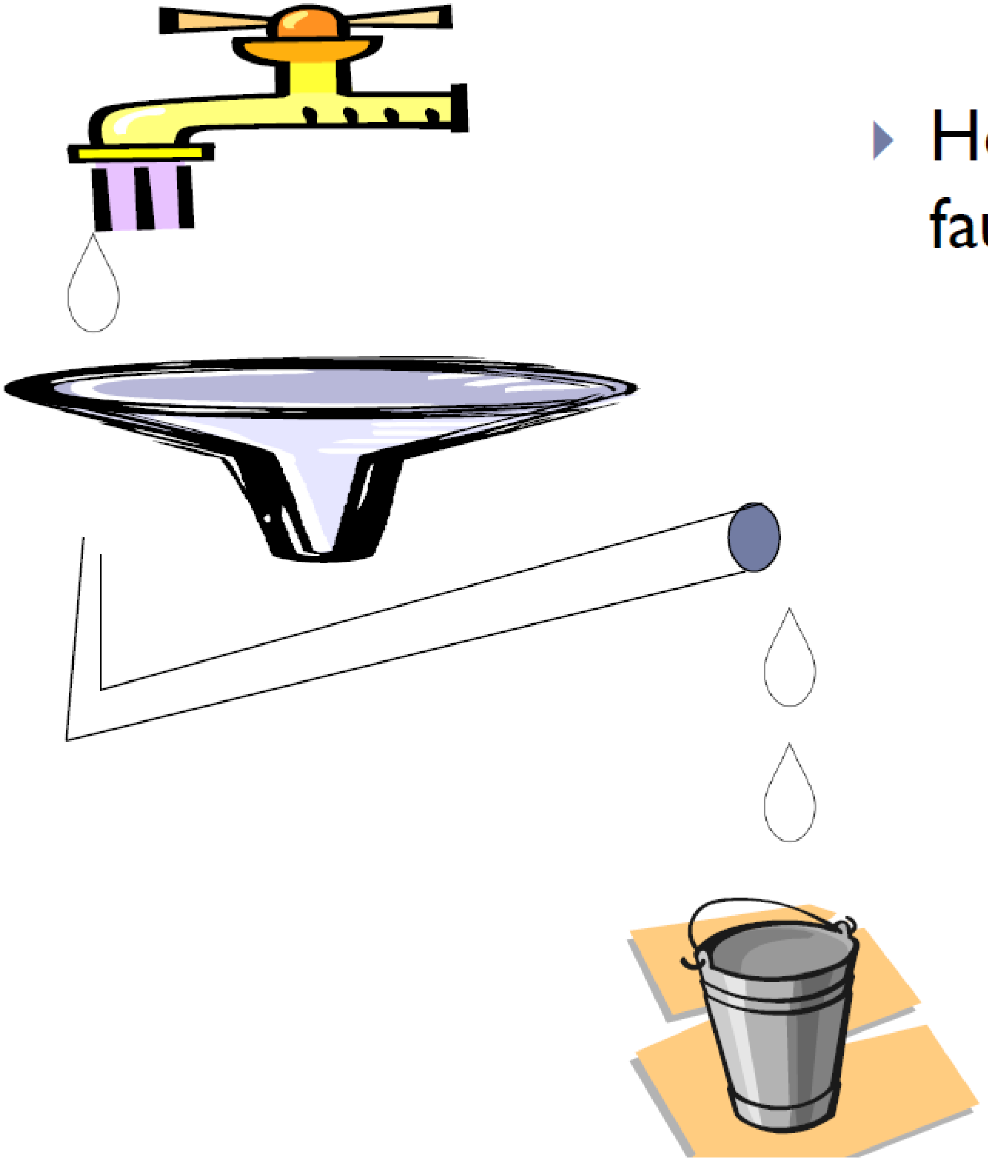
Data received to fast, to fast.... what to do....

TCP Persist

- What happens if window is 0?
  - Receiver updates window when application reads data
  - What if this update is lost?
- TCP Persist state
  - Sender periodically sends 1 byte packets
  - Receiver responds with ACK which contains the receive window

TCP – congestion control

Internet Pipes

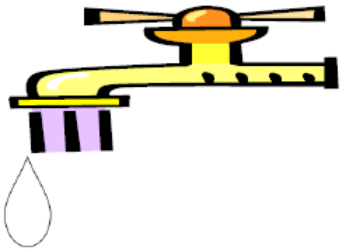


- How should you control the faucet?

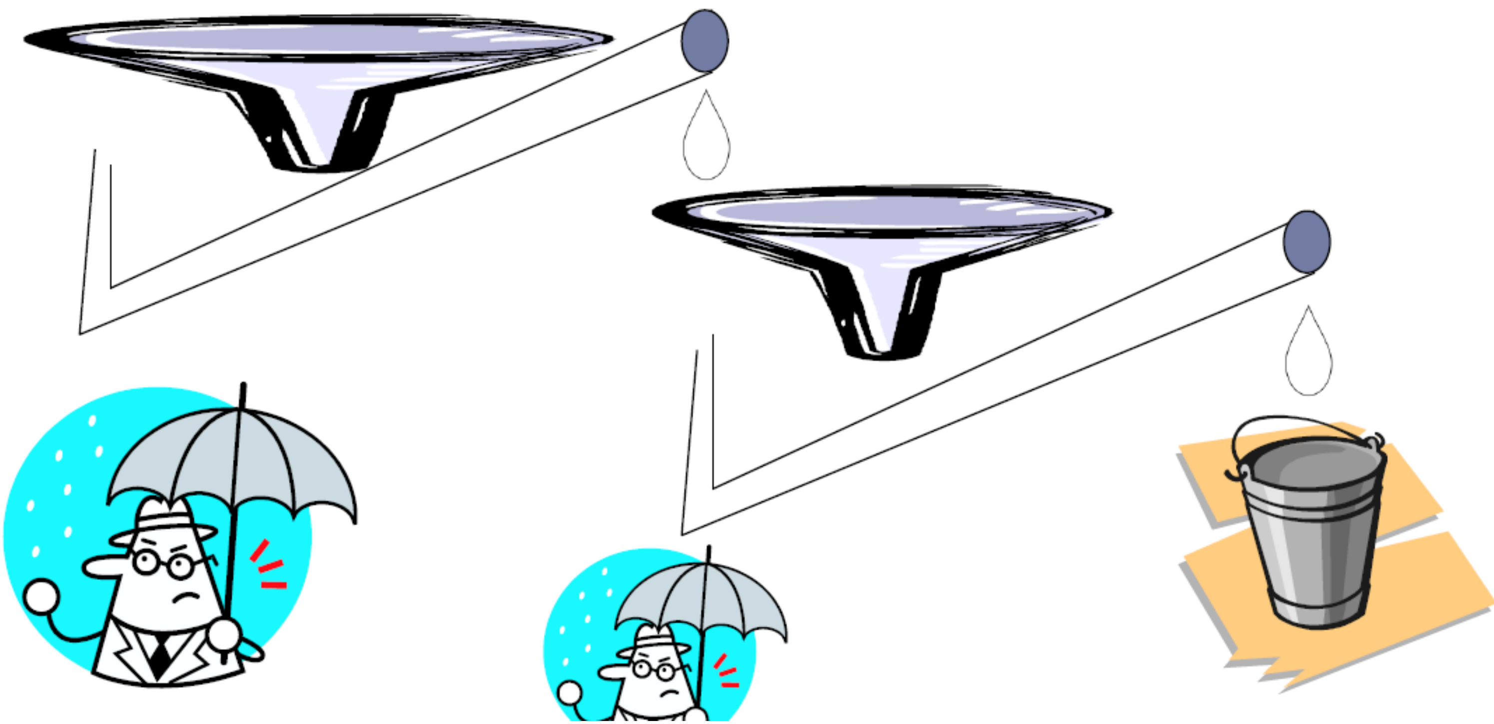
Internet Pipes

- How should you control the faucet?
  - Too fast – sink overflows
  - Too slow – what happens?
- Goals
  - Fill the bucket as quickly as possible
  - Avoid overflowing the sink
- Solution – watch the sink

Plumbers Gone Wild!

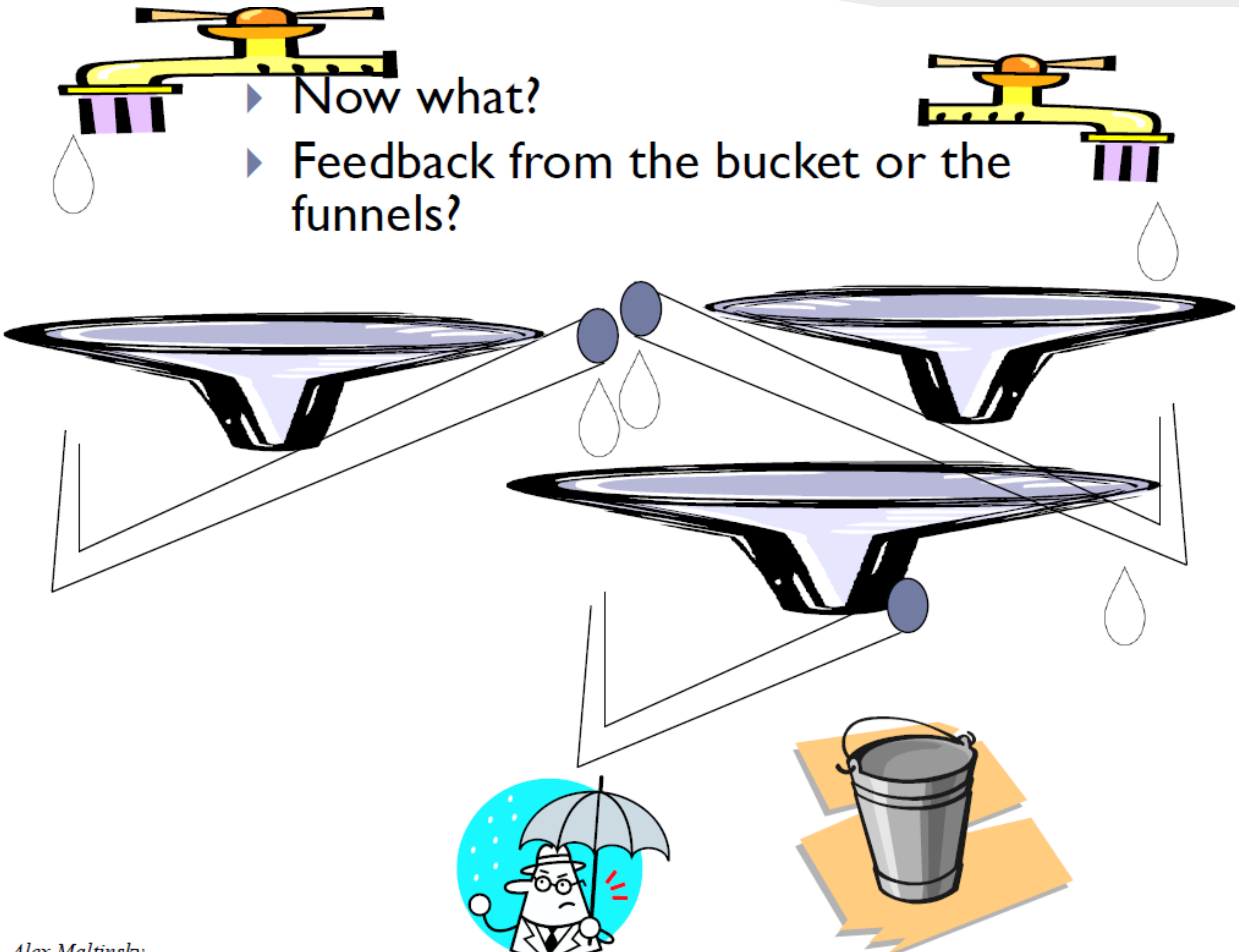


- How do we prevent water loss?
- Know the size of the pipes?



Plumbers Gone Wild -2

- Now what?
- Feedback from the bucket or the funnels?



er

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

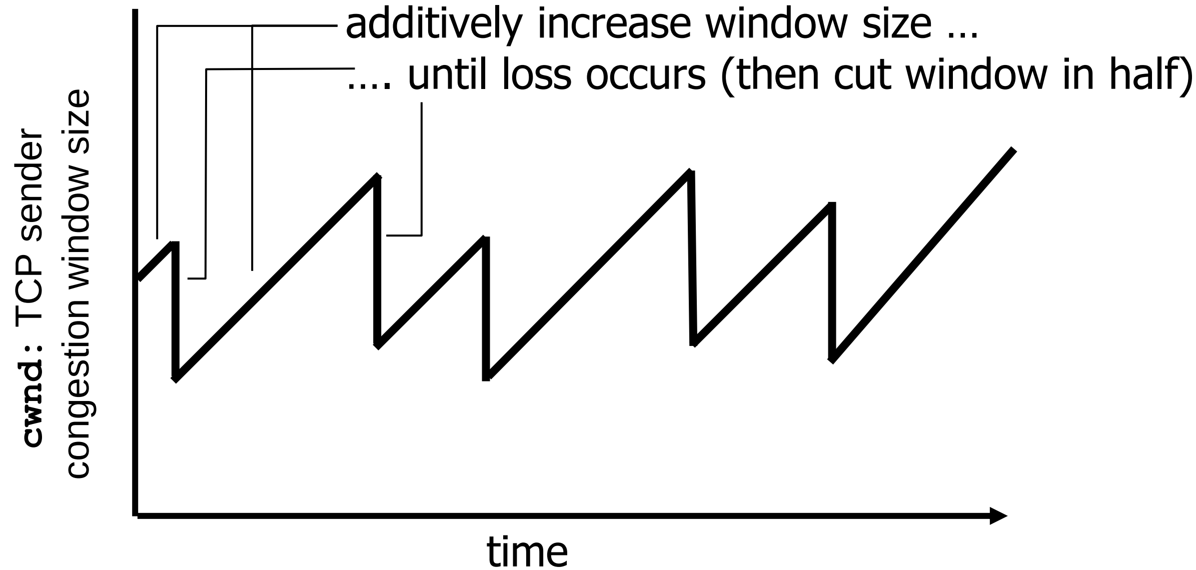
network-assisted congestion control:

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

TCP congestion control: AIMD

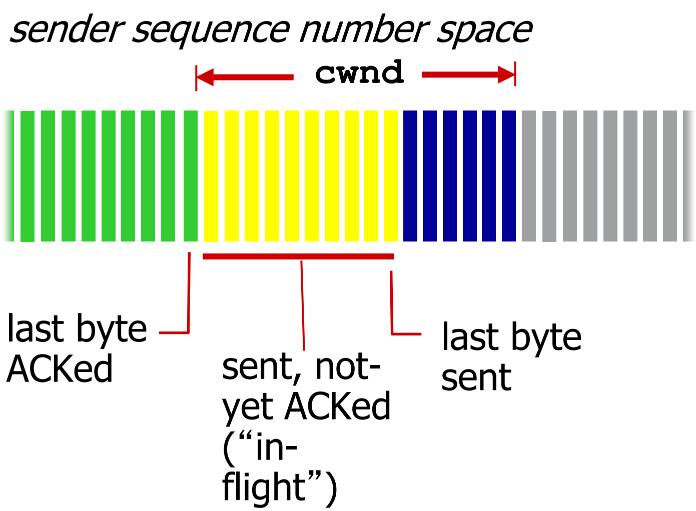
- *approach:* sender increases transmission rate (window size), probing for usable bandwidth, until loss occurs
- *additive increase:* increase **cwnd** by 1 MSS every RTT until loss detected
- *multiplicative decrease*: cut **cwnd** in half after loss

AIMD saw tooth behavior: probing for bandwidth



AIMD = additive increase multiplicative decrease

TCP Congestion Control: details



- sender limits transmission:

$$\boxed{\texttt{LastByteSent-LastByteAcked} \leq \texttt{cwnd}}$$

- **`cwnd`** is dynamic, function of perceived network congestion

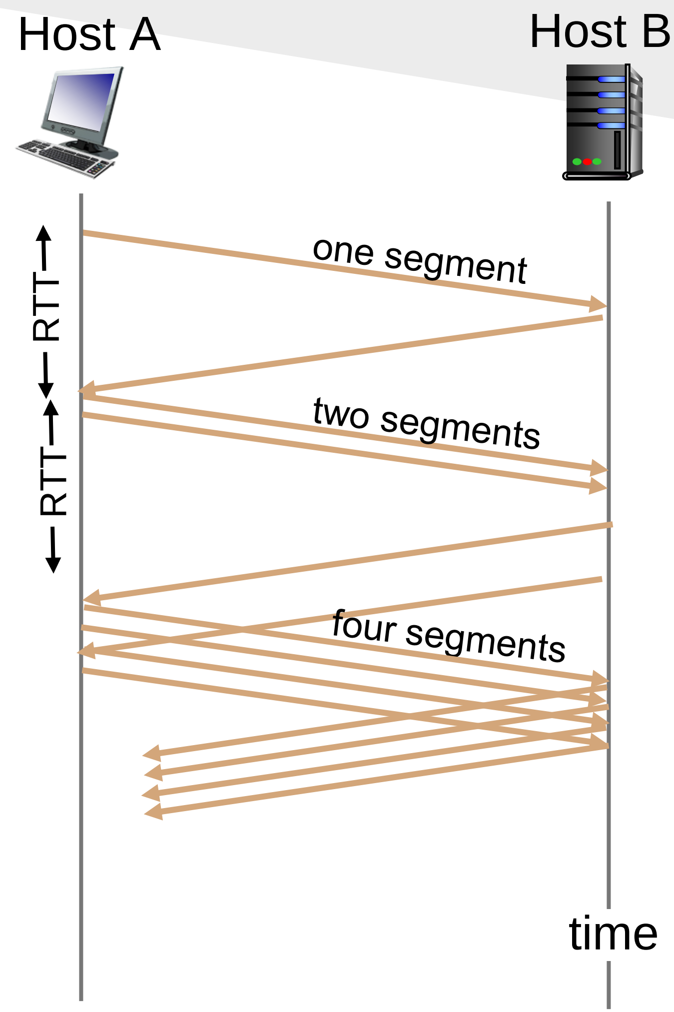
TCP sending rate:

- *roughly:* send cwnd bytes, wait RTT for ACKS, then send more bytes

$$\boxed{\text{rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ bytes/sec}}$$

TCP Slow Start

- When connection begins, increase rate exponentially until first loss event:
  - initially **cwnd** = 1 MSS
  - double **cwnd** every RTT
  - done by incrementing **cwnd** for every ACK received
- *Summary:* initial rate is slow but ramps up exponentially fast



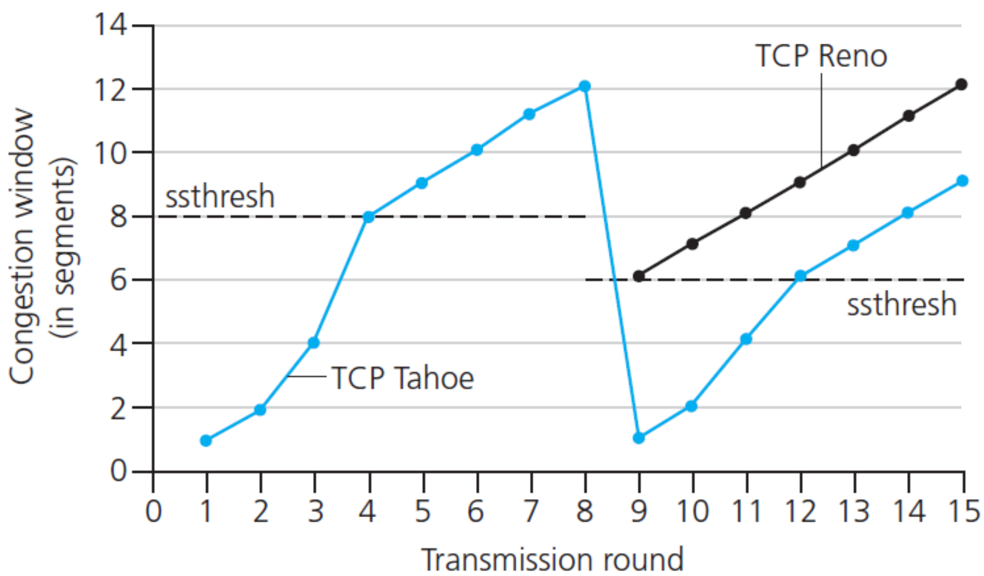
TCP: detecting, reacting to loss

- loss indicated by timeout:
- **`cwnd`** set to 1 MSS;
- window then grows exponentially (as in slow start) to threshold, then grows linearly
- loss indicated by 3 duplicate ACKs: TCP RENO
- dup ACKs indicate network capable of delivering some segments
- **`cwnd`** is cut in half window then grows linearly
- TCP Tahoe always sets **`cwnd`** to 1 (timeout or 3 duplicate acks)

TCP: switching from slow start to CA

Q: when should the exponential increase switch to linear?

A: when cwnd gets to 1/2 of its value before timeout.



Implementation:

- variable **`ssthresh`**
- on loss event, **`ssthresh`** is set to 1/2 of **`cwnd`** just before loss event

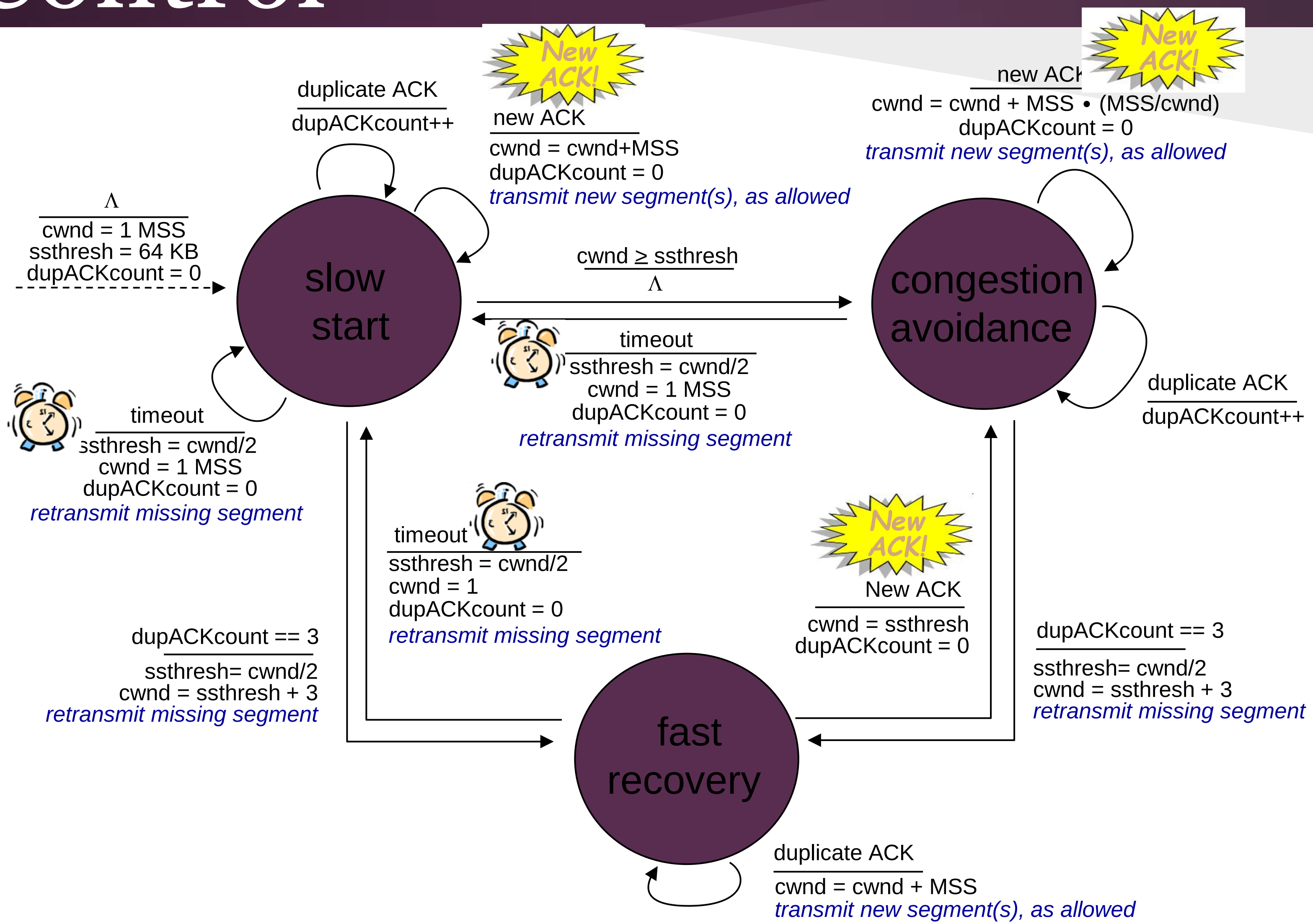
Refinement: inferring loss

- After 3 dup ACKs:
  - `CongWin` is cut in half
  - window then grows linearly
  - Part of Fast Recovery
- **<u>But</u> after timeout event:**
  - `CongWin` instead set to 1 MSS;
  - window then grows exponentially (This is SS)
  - to a threshold (ssthresh), 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

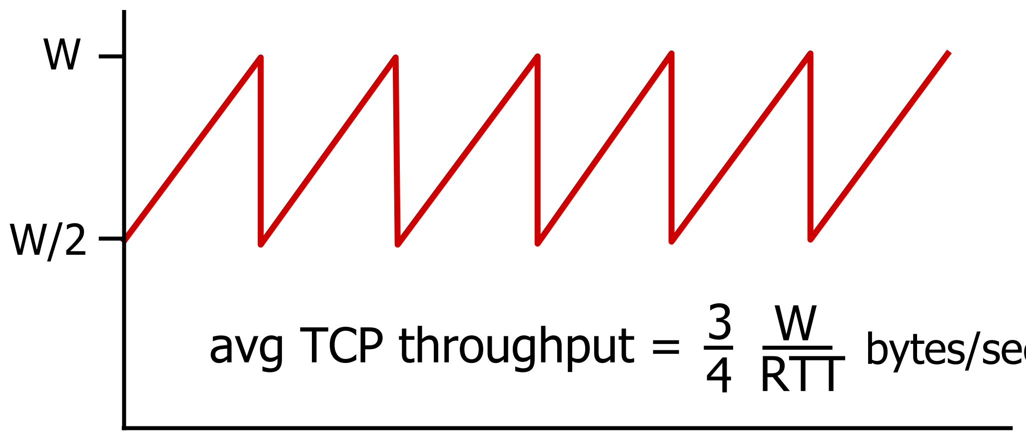


Summary: TCP Congestion Control

- when **cwnd < ssthresh**, sender in slow-start phase, window grows exponentially.
- when **cwnd >= ssthresh**, sender is in congestion-avoidance phase, window grows linearly.
- when triple duplicate ACK occurs, **ssthresh** set to **cwnd/2**, **cwnd** set to ~ **ssthresh**
- when timeout occurs, **ssthresh** set to **cwnd/2**, **cwnd** set to 1 MSS.

TCP throughput

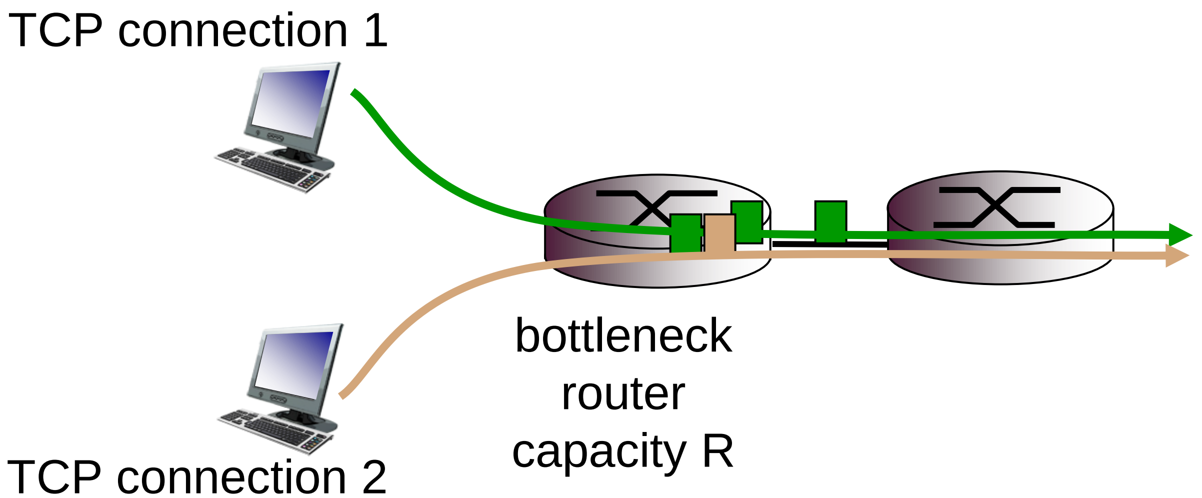
- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. throughput is 3/4W per RTT



$$\text{avg TCP throughput} = \frac{3}{4}\frac{W}{RTT} \text{ bytes/sec}$$

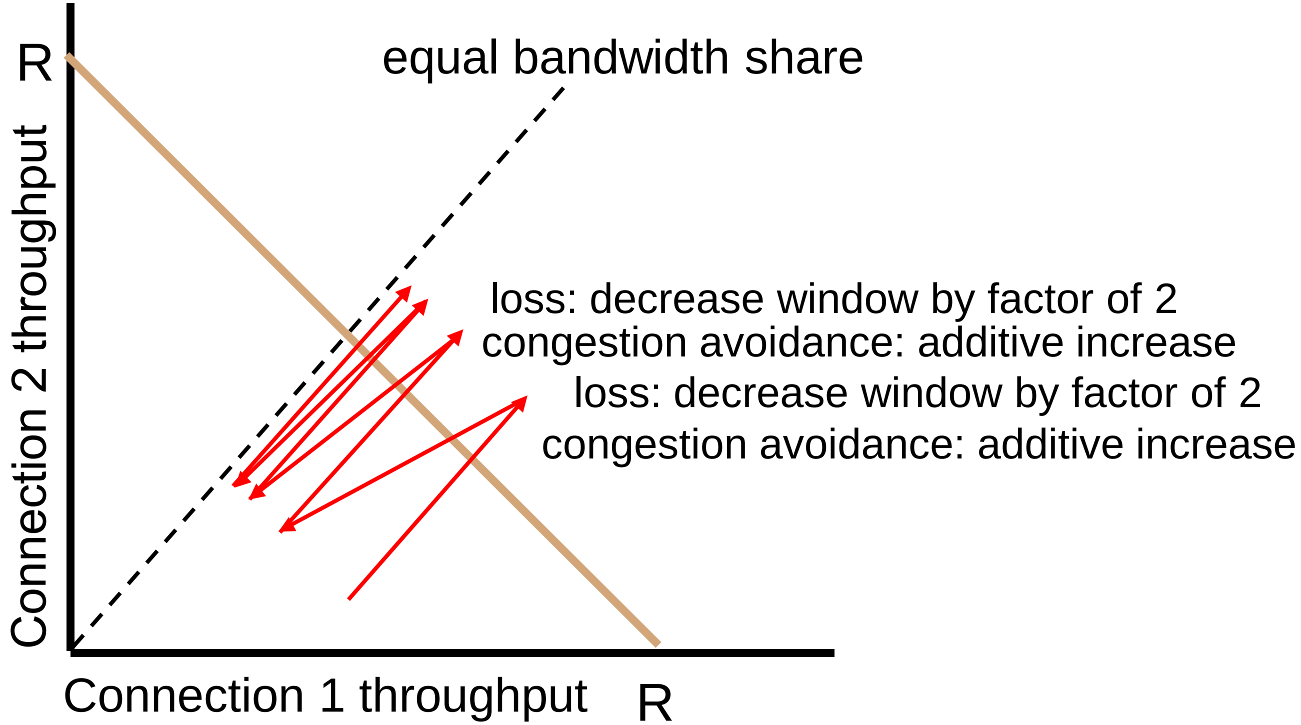
TCP Fairness

fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



Why is TCP fair?

- two competing sessions:
  - additive increase gives slope of 1, as throughout increases
  - multiplicative decrease decreases throughput proportionally



Fairness (more)

Fairness and UDP

- multimedia apps often do not use TCP
  - do not want rate throttled by congestion control
- instead use UDP:
  - send audio/video at constant rate, tolerate packet loss

Fairness, parallel TCP connections

- application can open multiple parallel connections between two hosts
- web browsers do this
- e.g., link of rate R with 9 existing connections:
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets R/2

שאלות

מה הקשר בין GBN, SR ו-TCP?
תשובה: שלושה פרוטוקולים שונים לאותה מטרה עם מנגנונים דומים.

על בסיס מה מחושב ה-timeout ב-TCP?
תשובה: דגימות של RTT.

לשם מה יש צורך במנגנון congestion control?
תשובה: עקב כך שאין אפשרות לדעת מה קצבי שידור הרשת בין המקור ליעד (קצב משתנה כל הזמן).

לשם מה יש צורך במנגנון flow control?
תשובה: עקב כך שאנו רוצים להמנע ממצב שהיעד זורק חבילות עקב תור עמוס.

שאלות

כמה פעמים אני צריך לקבל הודעה על מנת לשדר סגמנט שוב ואיזה הודעה זו?
תשובה: ack, אותו acknum 4 פעמים (מקור + 3 כפולים)

מדוע יש הבדל בטיפול בין timeout ל-fast retransmission?
תשובה: המקרה הראשון חמור יותר עקב כך שלא היה אפשרי להגיע ל- 3 duplicate ולכן מצב הרשת קשה לעומת השני שמצב הרשת בעייתי אך לא קשה.

מדוע לאחר סגירת קשר יש להמתין timeout ארוך למרות שאנו לא מצפים לקבל חבילה?
תשובה: על מנת להיות בטוחים ככל האפשר שהקשר נסגר

Useful Links

TCP

http://www.youtube.com/watch?v=KSJu5FqwEMM&feature=related

Slow Start

http://www.youtube.com/watch?feature=endscreen&v=_sxeFJRVSXw&NR=1

RTT

http://www.youtube.com/watch?feature=endscreen&v=Wcjxpmh7C4U&NR=1

SR ,GBN

http://www.youtube.com/watch?v=yT8SkFyRRrI

TCP VS UDP

http://www.youtube.com/watch?v=Vdc8TCESIg8

TCP – מקרים ותגובות

מקרים ותגובות

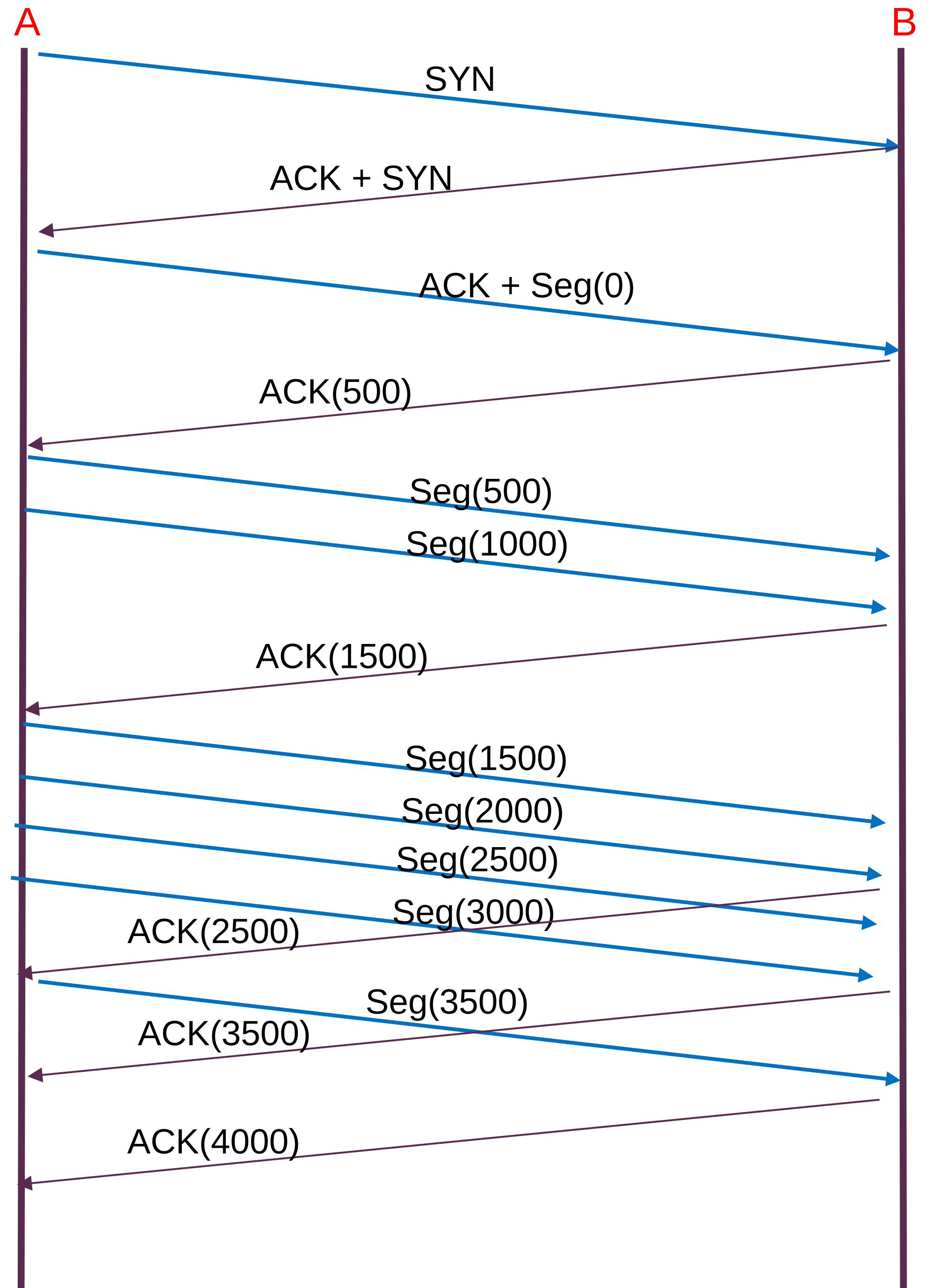
נתון:

- נתון מחשב A המתחבר לשרת B
- מחשב A שולח ל- B קובץ בגודל 4000 בתים.
- MSS הוא 500 בתים.
- הנח שזמן שידור הודעות בקרה זניח ack, syn, fin...

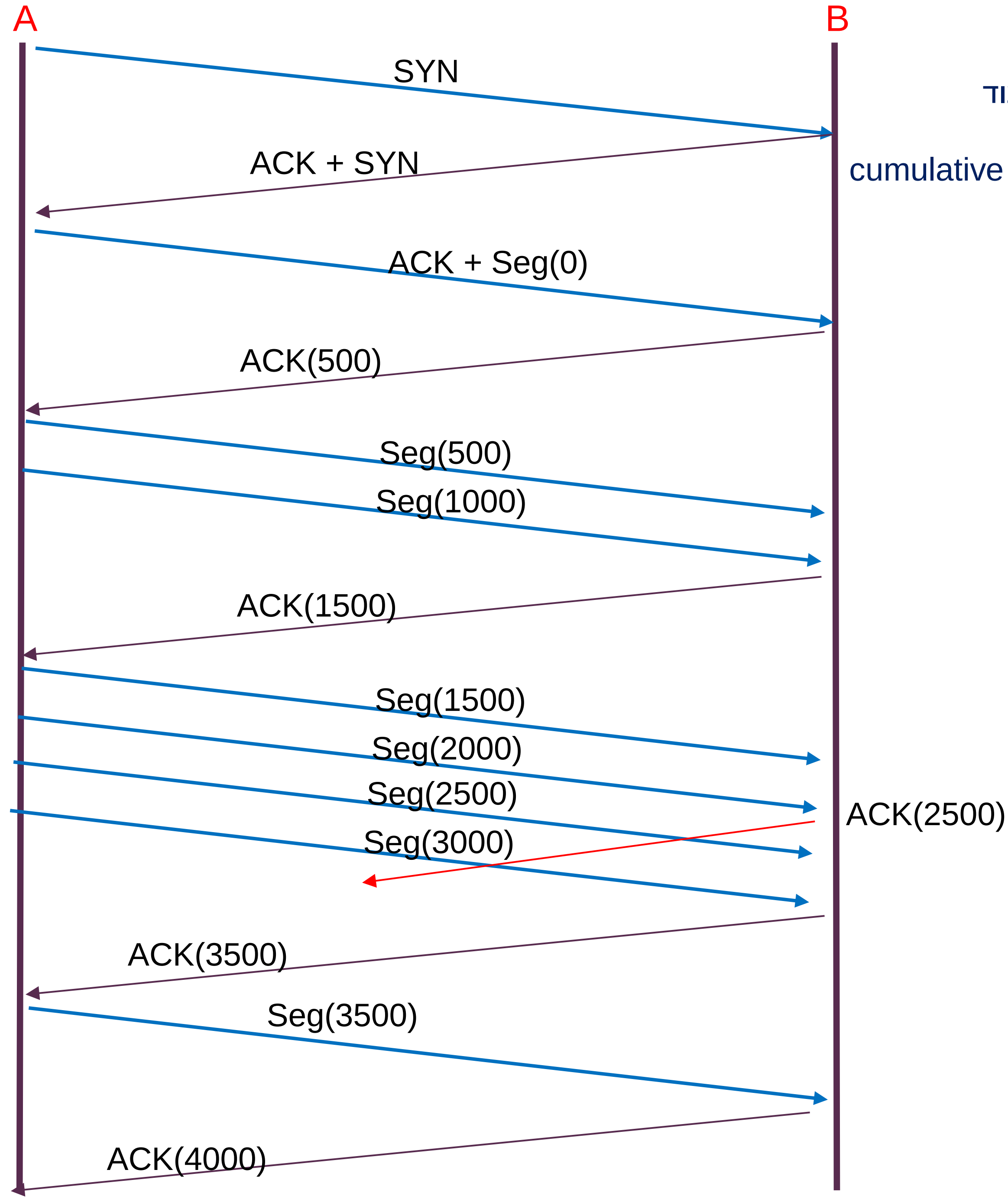
מקרים:

1) תהליך תקין.
2) Ack הולך לאיבוד וחבילות אחרות מגיעות <- cumulative ack
3) מספר הודעות מגיעות ליעד אך ה-acks נופלים <- timeout
4) Timeout קצר יחסית לRTT – premature time out
5) הודעה הולכת לאיבוד אך ההודעות אחרי מגיעות <- duplicate acks

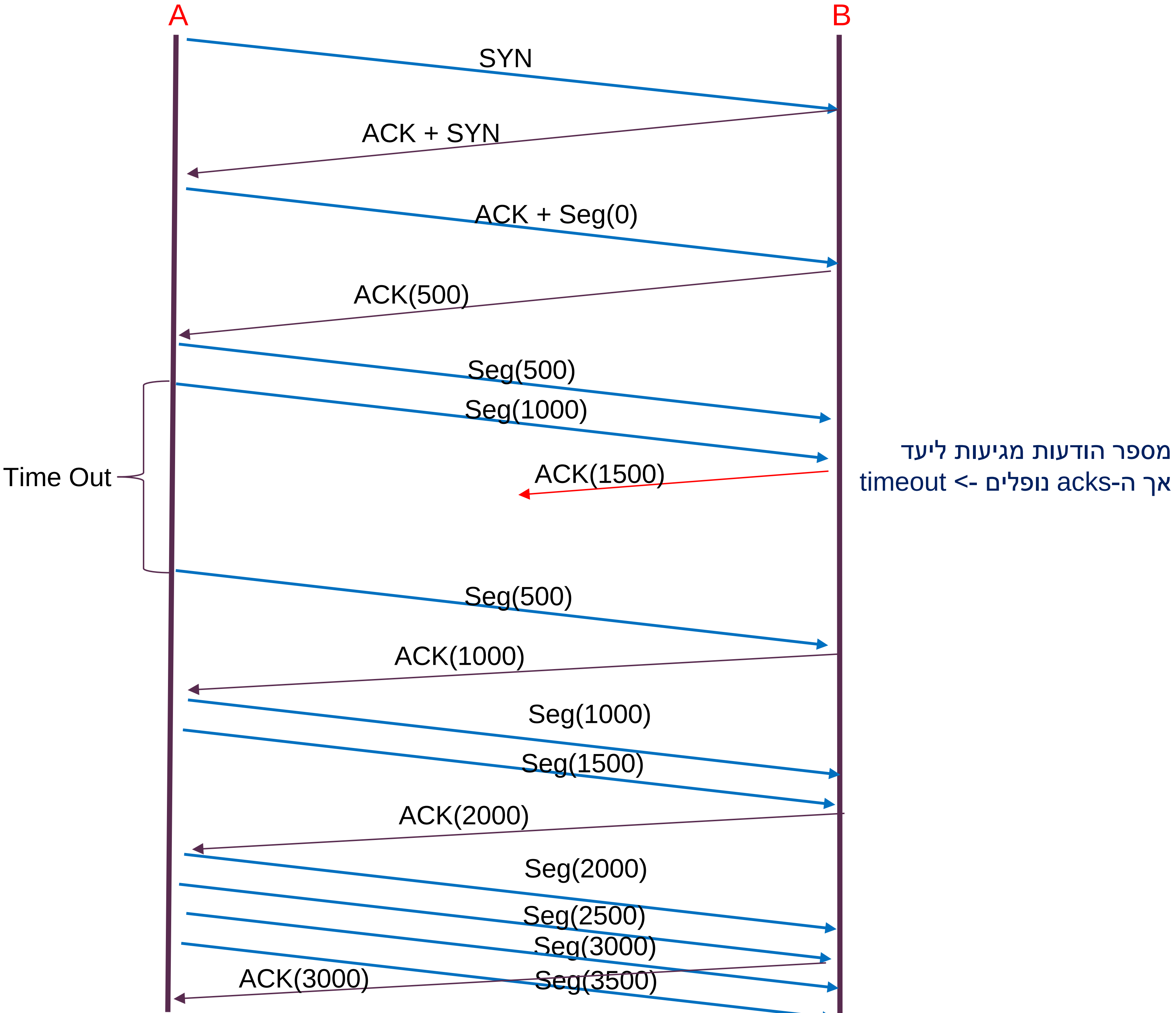
(1 תהליך תקין.



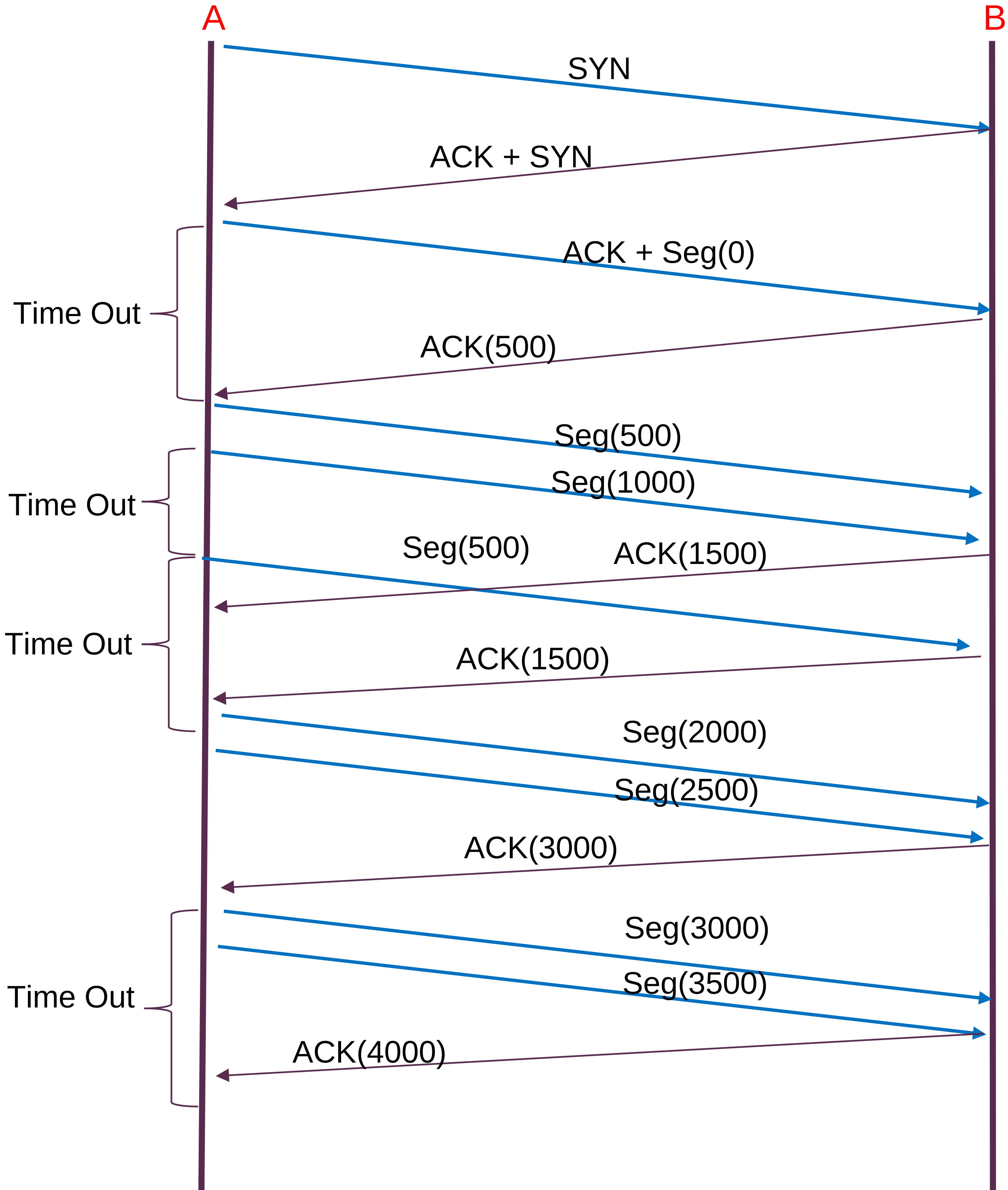
A
B
SYN
ACK + SYN
ACK + Seg(0)
ACK(500)
Seg(500)
Seg(1000)
ACK(1500)
Seg(1500)
Seg(2000)
Seg(2500)
Seg(3000)
ACK(2500)
Seg(3500)
ACK(3500)
ACK(4000)



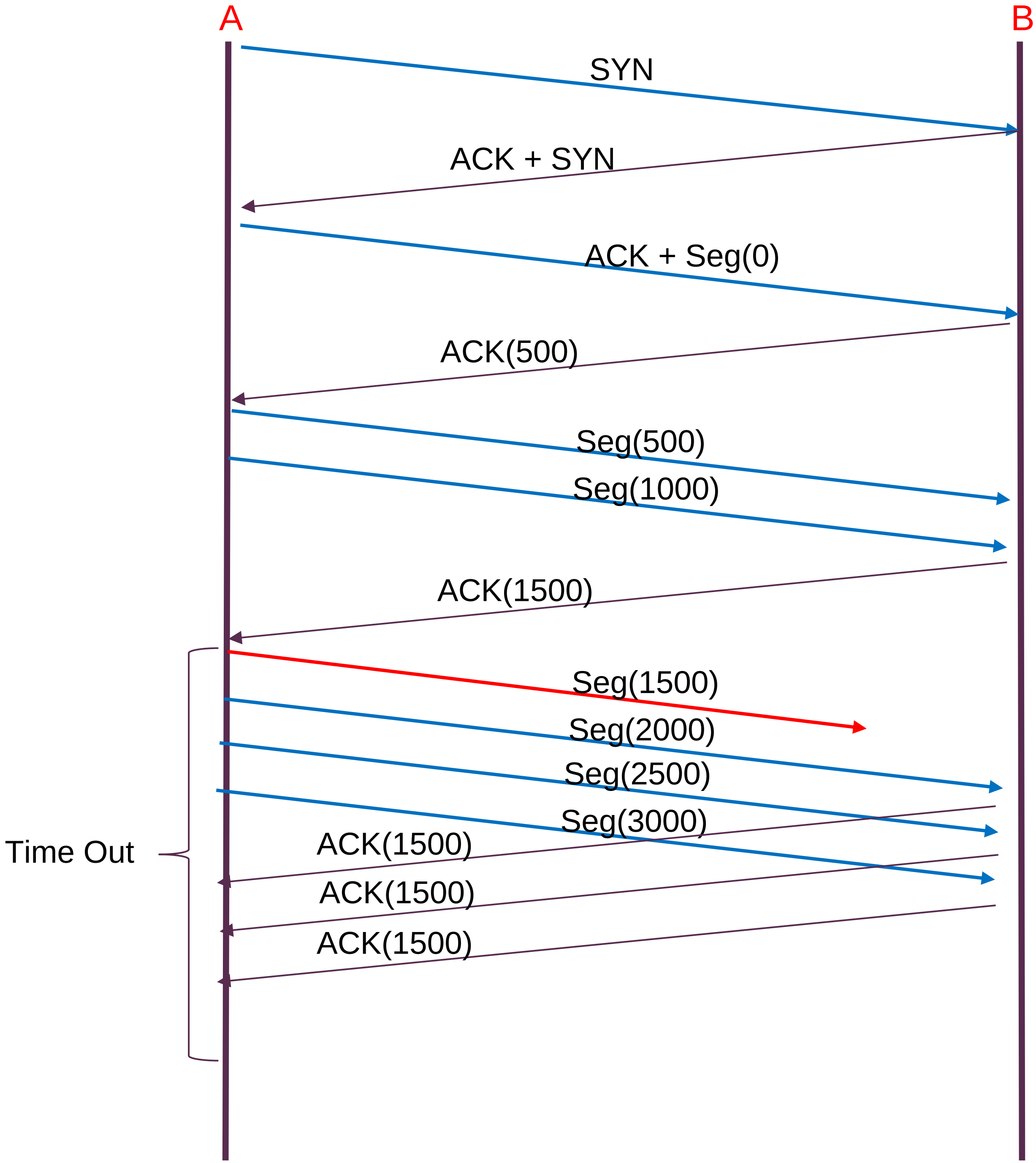
Ack הולך לאיבוד
וחבילות אחרות
מגיעות -> cumulative ack



A
B
SYN
ACK + SYN
ACK + Seg(0)
ACK(500)
Seg(500)
Seg(1000)
ACK(1500)
מספר הודעות מגיעות ליעד
אך ה-acks נופלים -> timeout
Time Out
Seg(500)
ACK(1000)
Seg(1000)
Seg(1500)
ACK(2000)
Seg(2000)
Seg(2500)
Seg(3000)
ACK(3000)
Seg(3500)



1) Timeout קצר יחסית
לRTT – premature time out



1) הודעה הולכת לאיבוד
אך ההודעות אחרי
מגיעות -> duplicate acks

Assume Duplicate ack is after 2 dup

Fast Retransmit