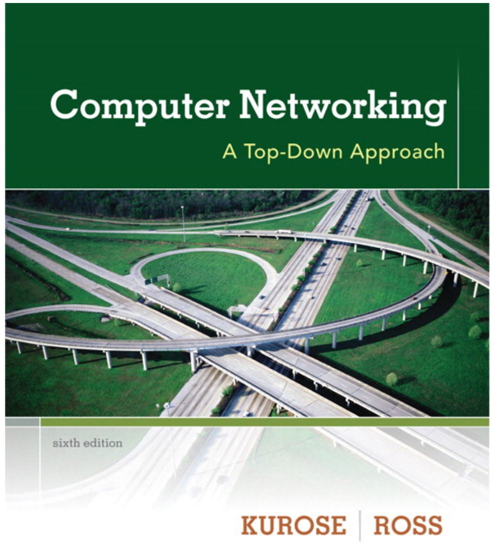
Chapter 3
Transport Layer



Reti degli Elaboratori
Canale AL
Prof.ssa Chiara Petrioli
a.a. 2016/2017

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

We thank for the support material Prof. Kurose-Ross

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

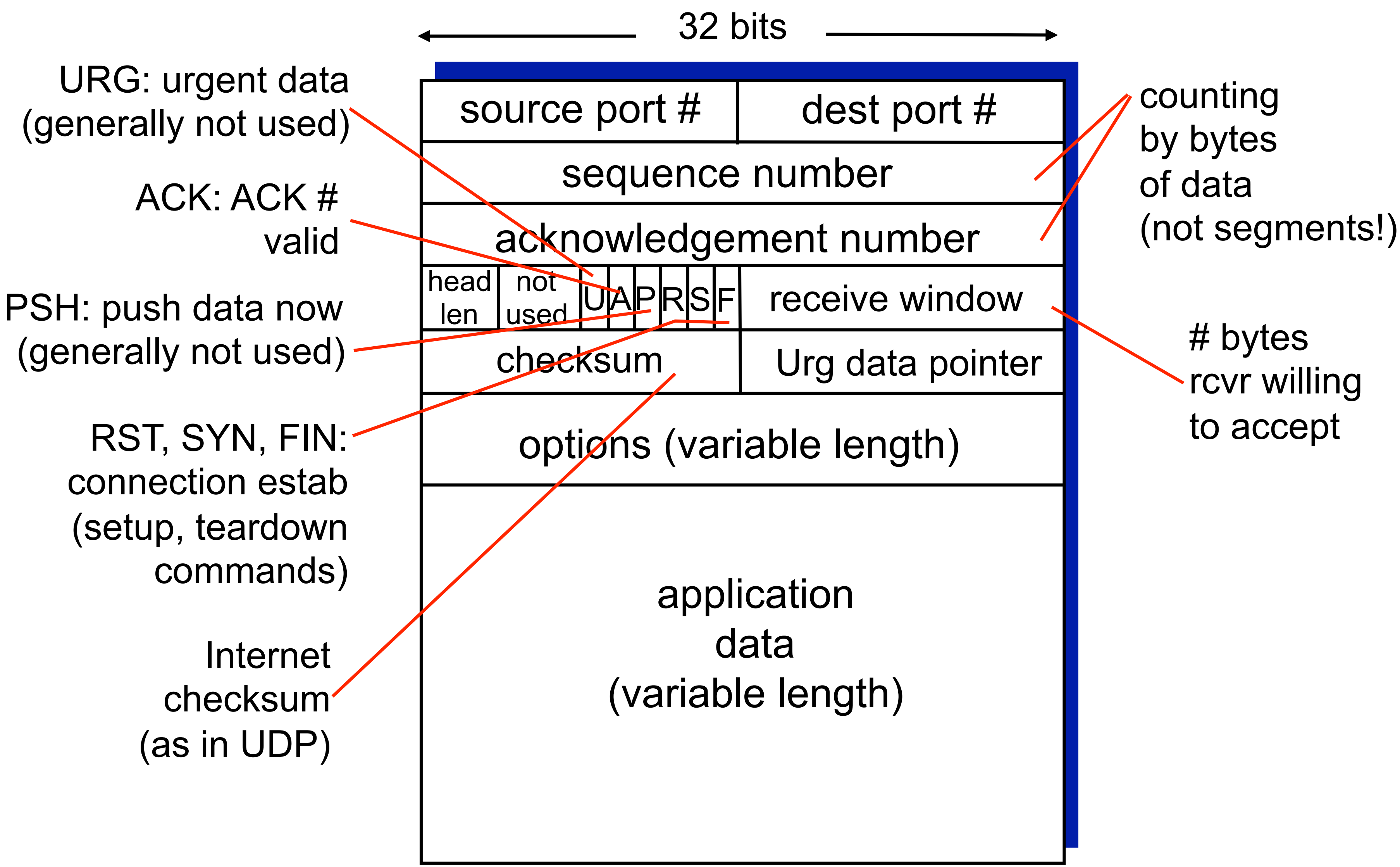
3.6 principles of congestion control

3.7 TCP congestion control

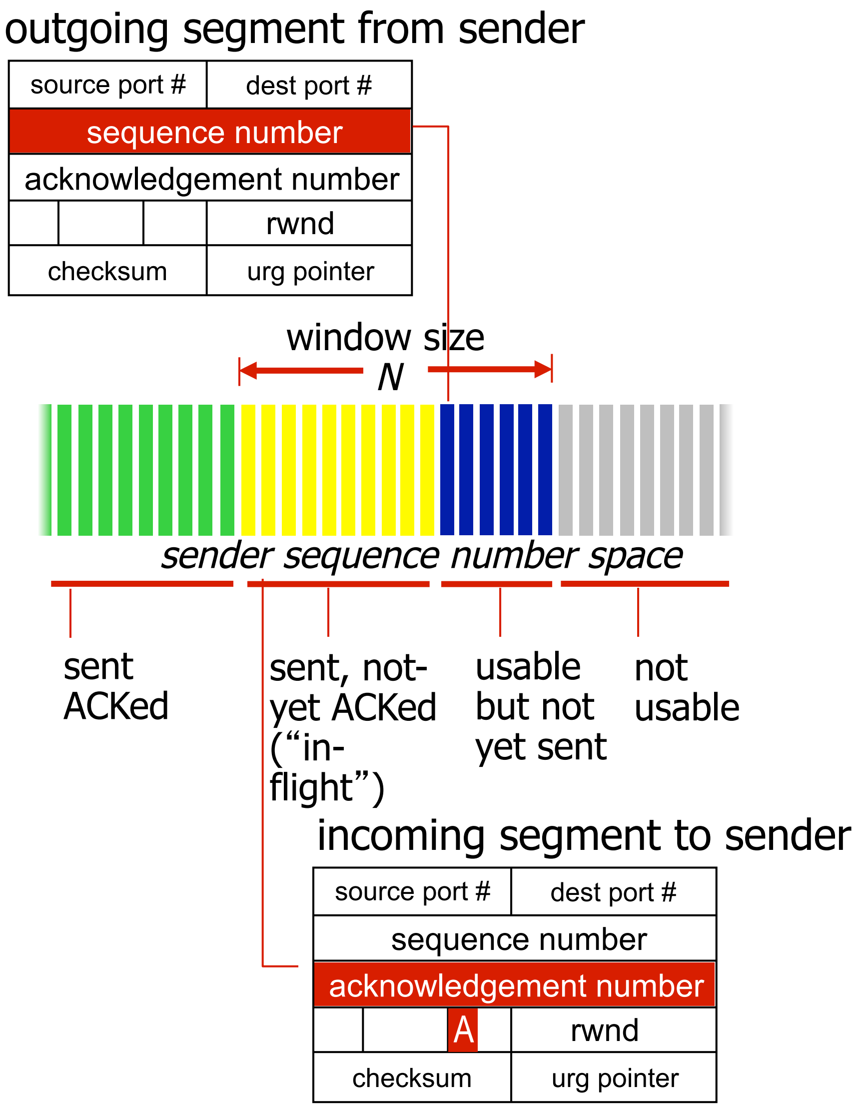
TCP: Overview RFCs: 793,1122,1323, 2018, 2581

- point-to-point:
  - one sender, one receiver
- reliable, in-order *byte steam:*
  - no "message boundaries"
- pipelined:
  - TCP congestion and flow control set window size
- full duplex data:
  - bi-directional data flow in same connection
  - MSS: maximum segment size
- connection-oriented:
  - handshaking (exchange of control msgs) inits sender, receiver state before data exchange
- flow controlled:
  - sender will not overwhelm receiver

TCP segment structure



TCP seq. numbers, ACKs



sequence numbers:

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

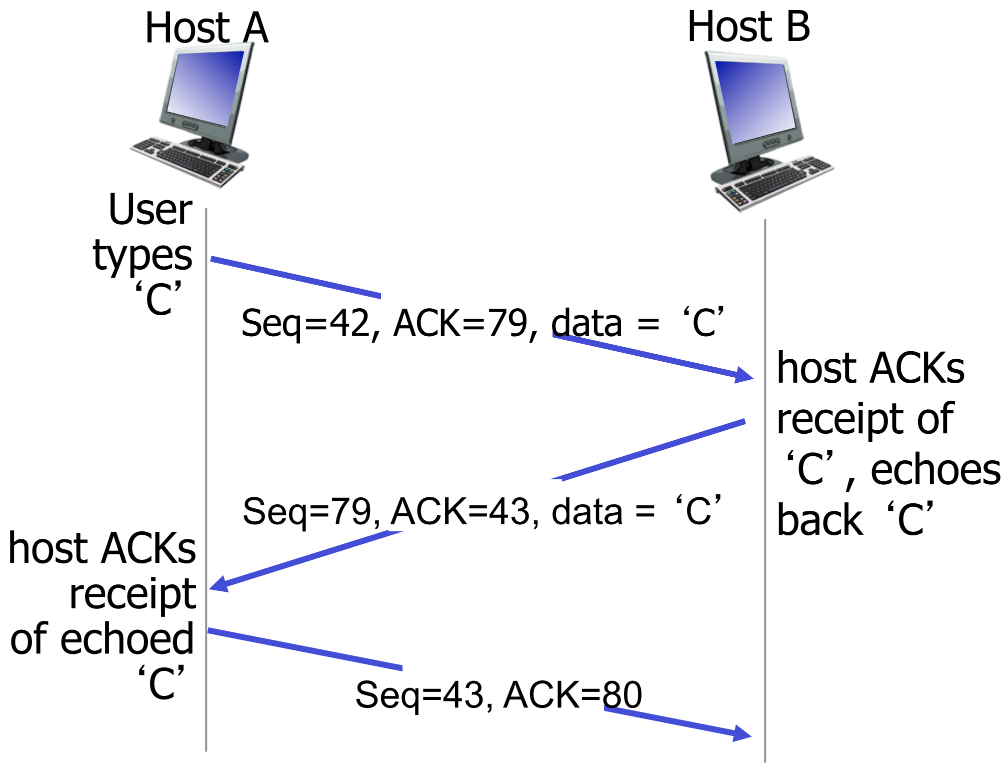
acknowledgements:

- 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



simple telnet scenario

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 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)

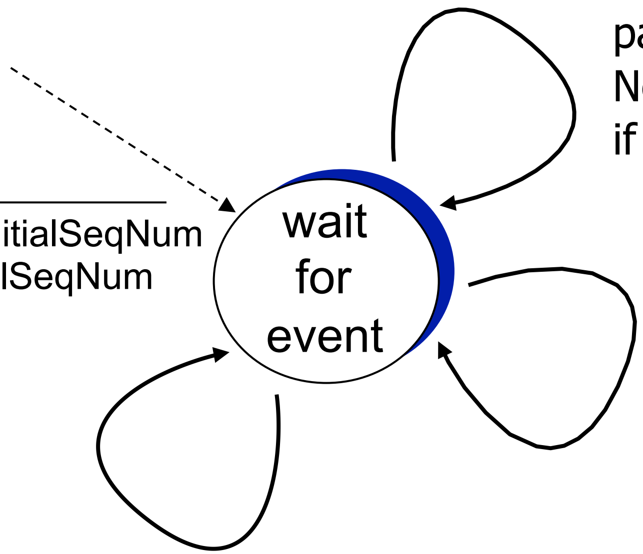
data received from application above

---

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

Λ

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum



timeout

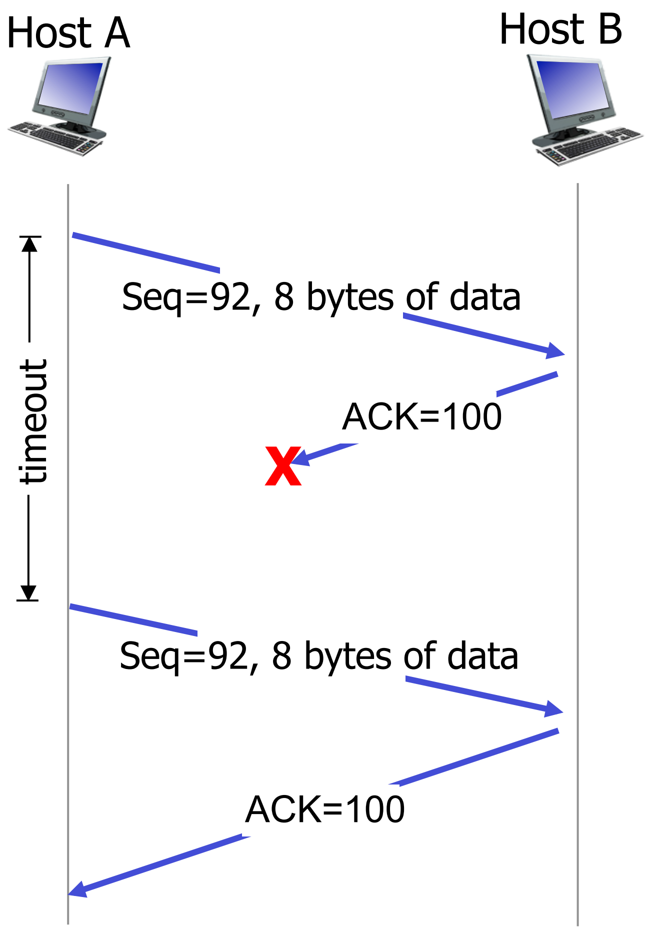
---

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

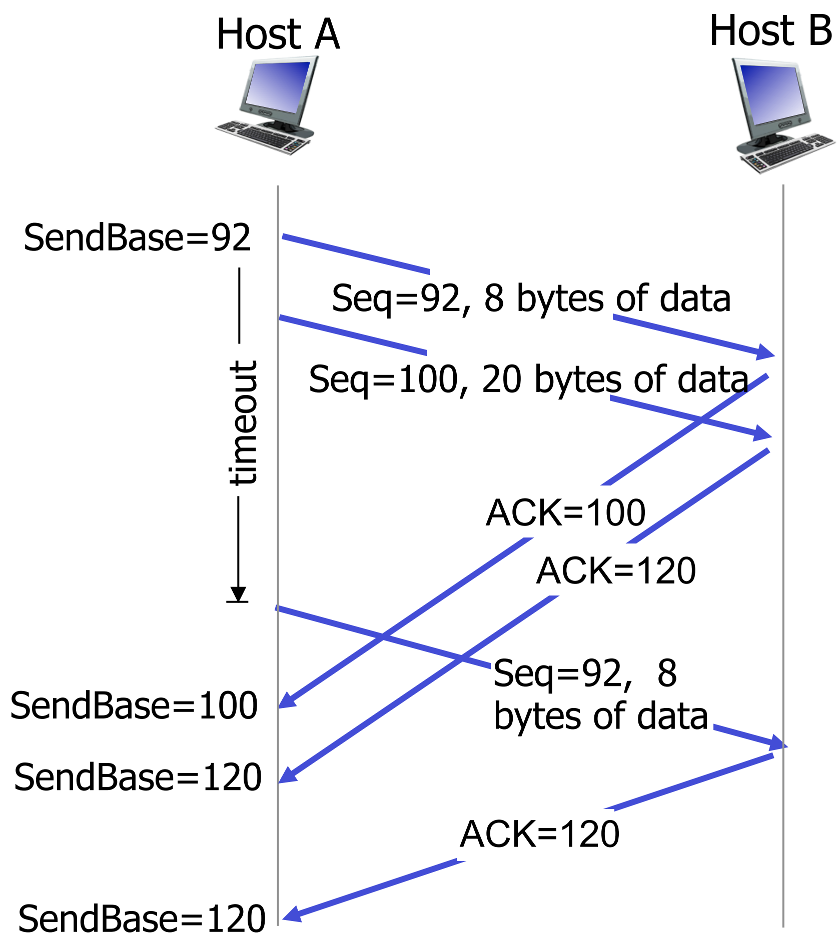
ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
       start timer
     else stop timer
  }
```

TCP: retransmission scenarios

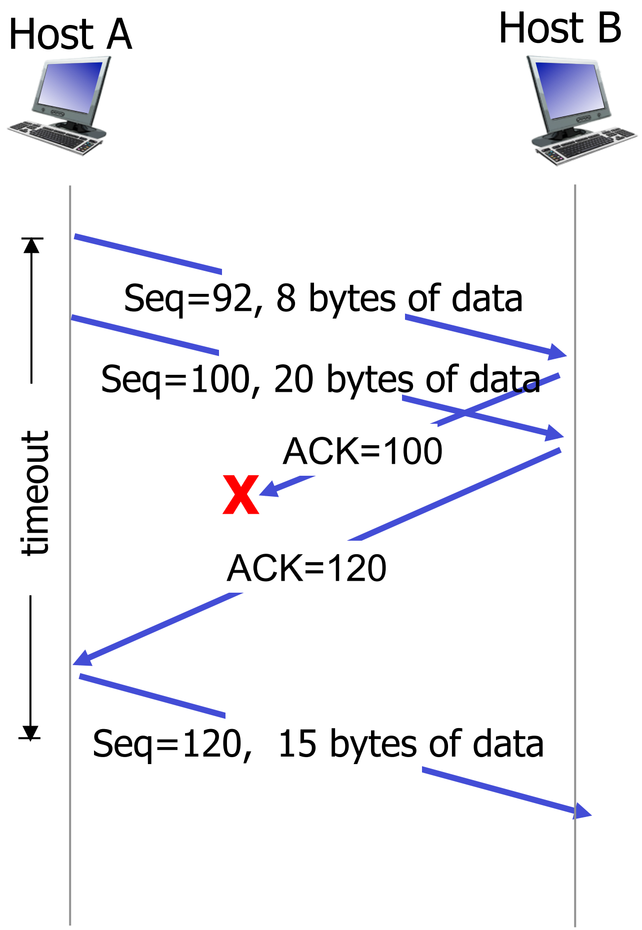


lost ACK scenario



premature timeout

TCP: retransmission scenarios



cumulative ACK

TCP 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

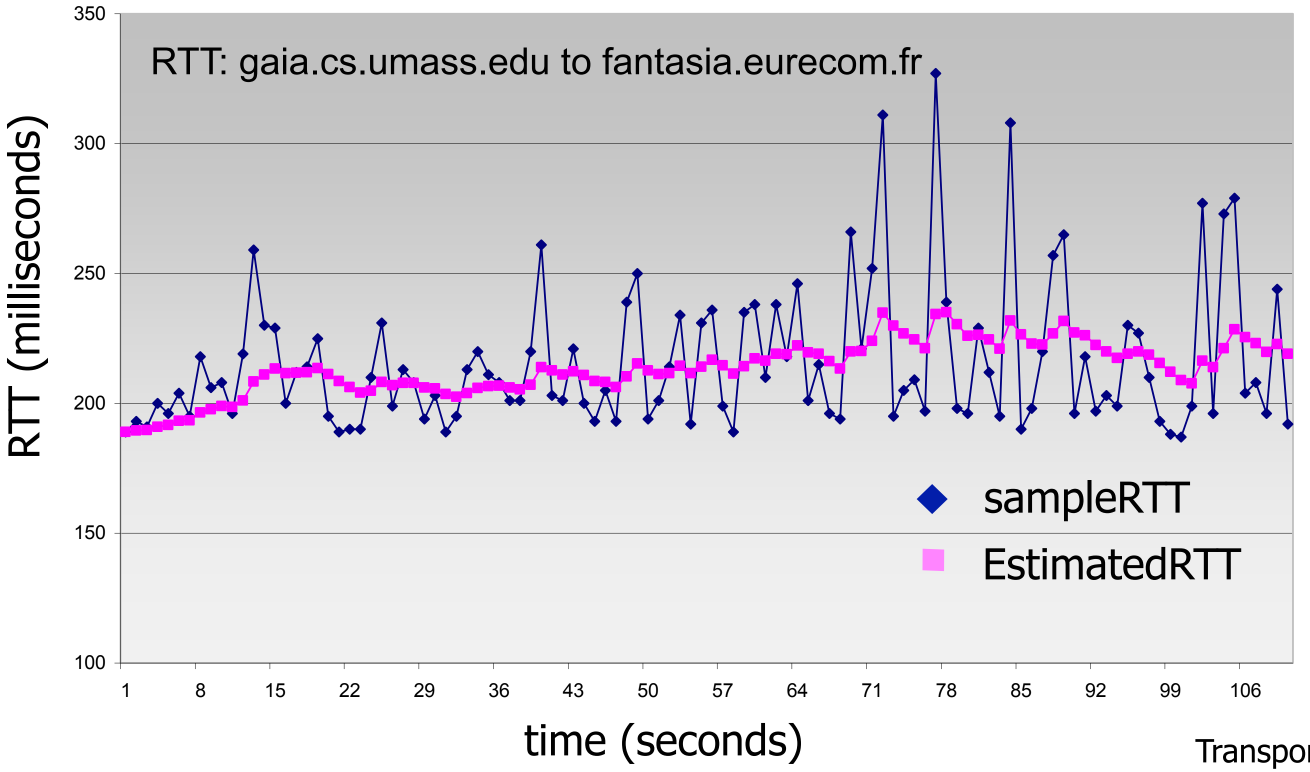
Q: how to estimate RTT?

- **`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: $\alpha = 0.125$



TCP round trip time, timeout

- 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"

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

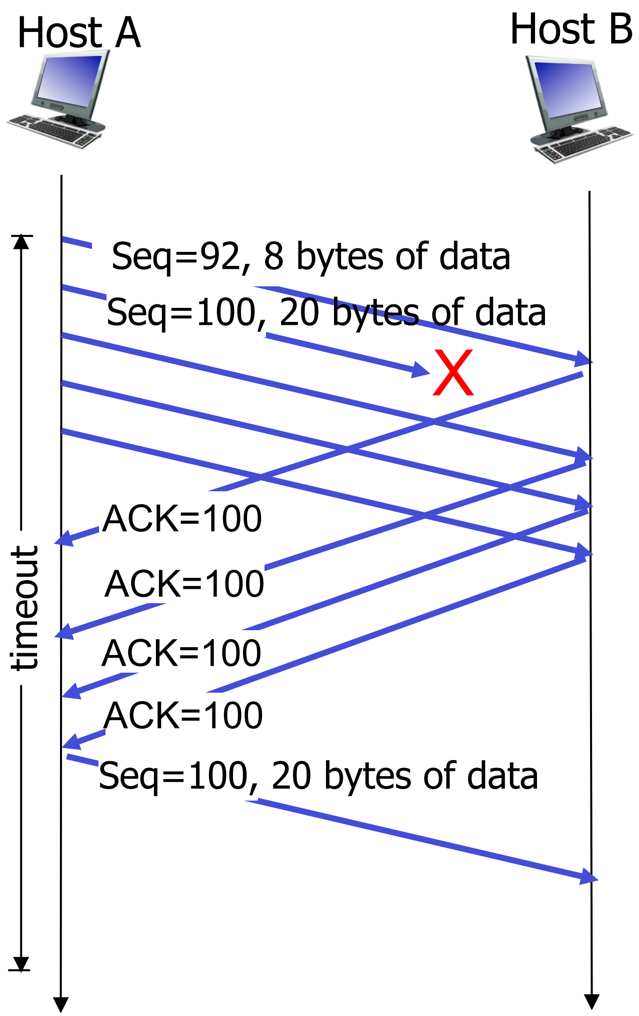
- time-out period often relatively long:
  - long delay before resending lost packet
- detect lost segments via duplicate ACKs.
  - sender often sends many segments back-to-back
  - if segment is lost, there will likely be many duplicate ACKs.

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

3.1 transport-layer services

3.2 multiplexing and demultiplexing

3.3 connectionless transport: UDP

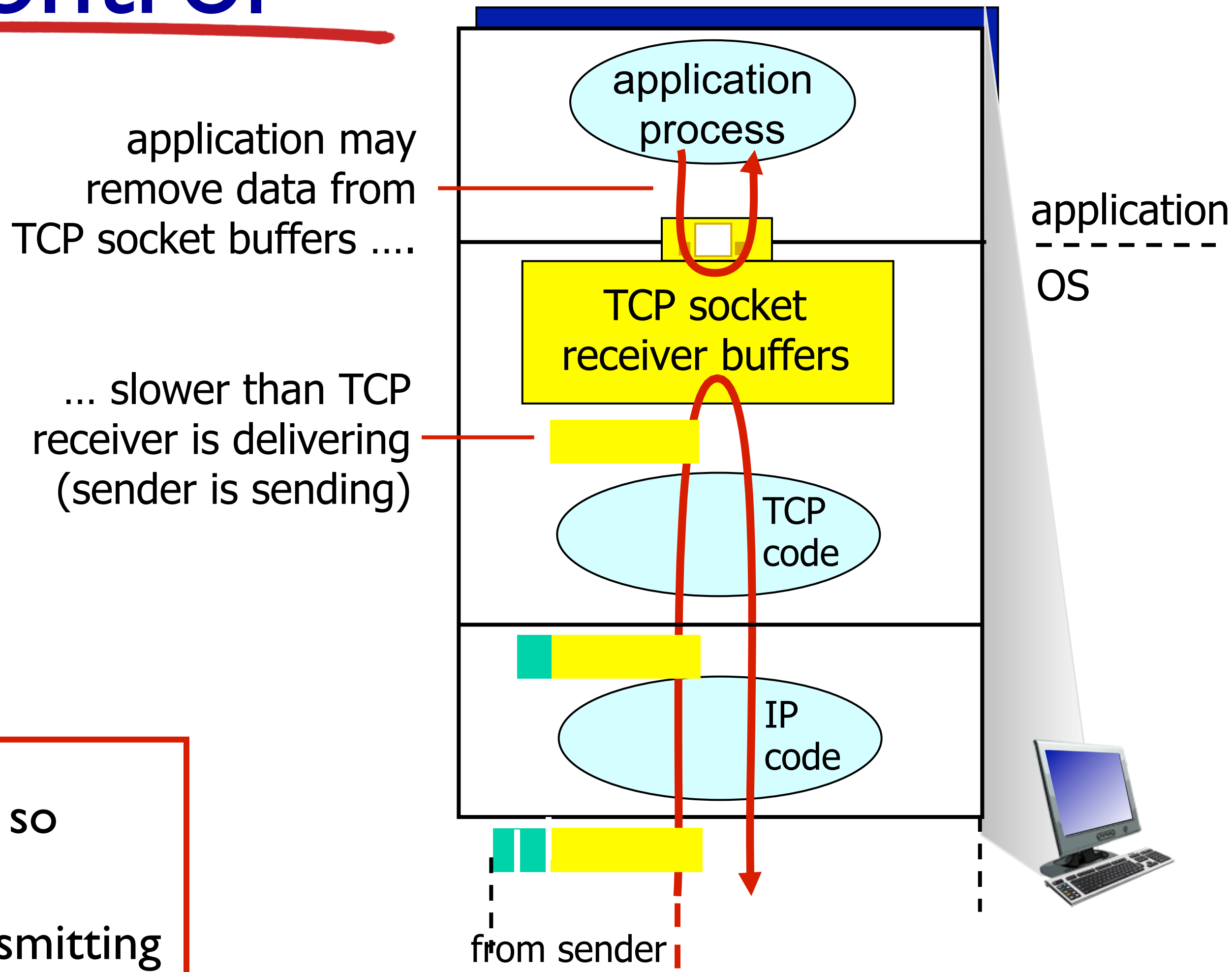
3.4 principles of reliable data transfer

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 flow control



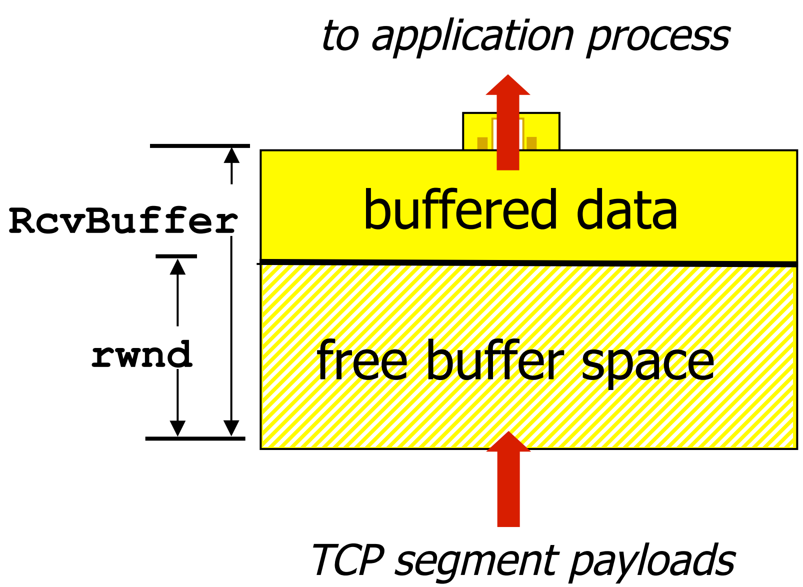
receiver protocol stack

flow control

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

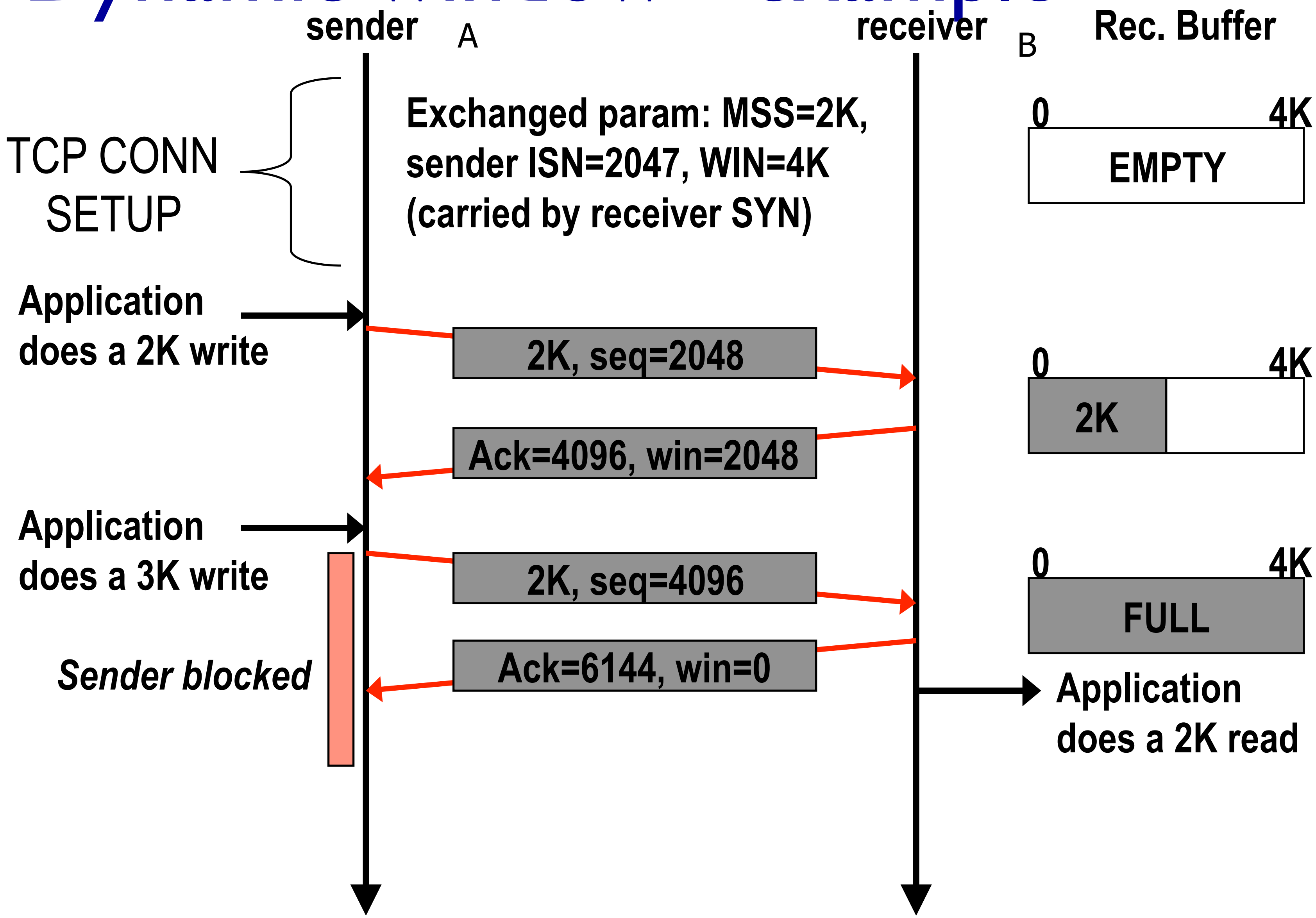
TCP flow control

- 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

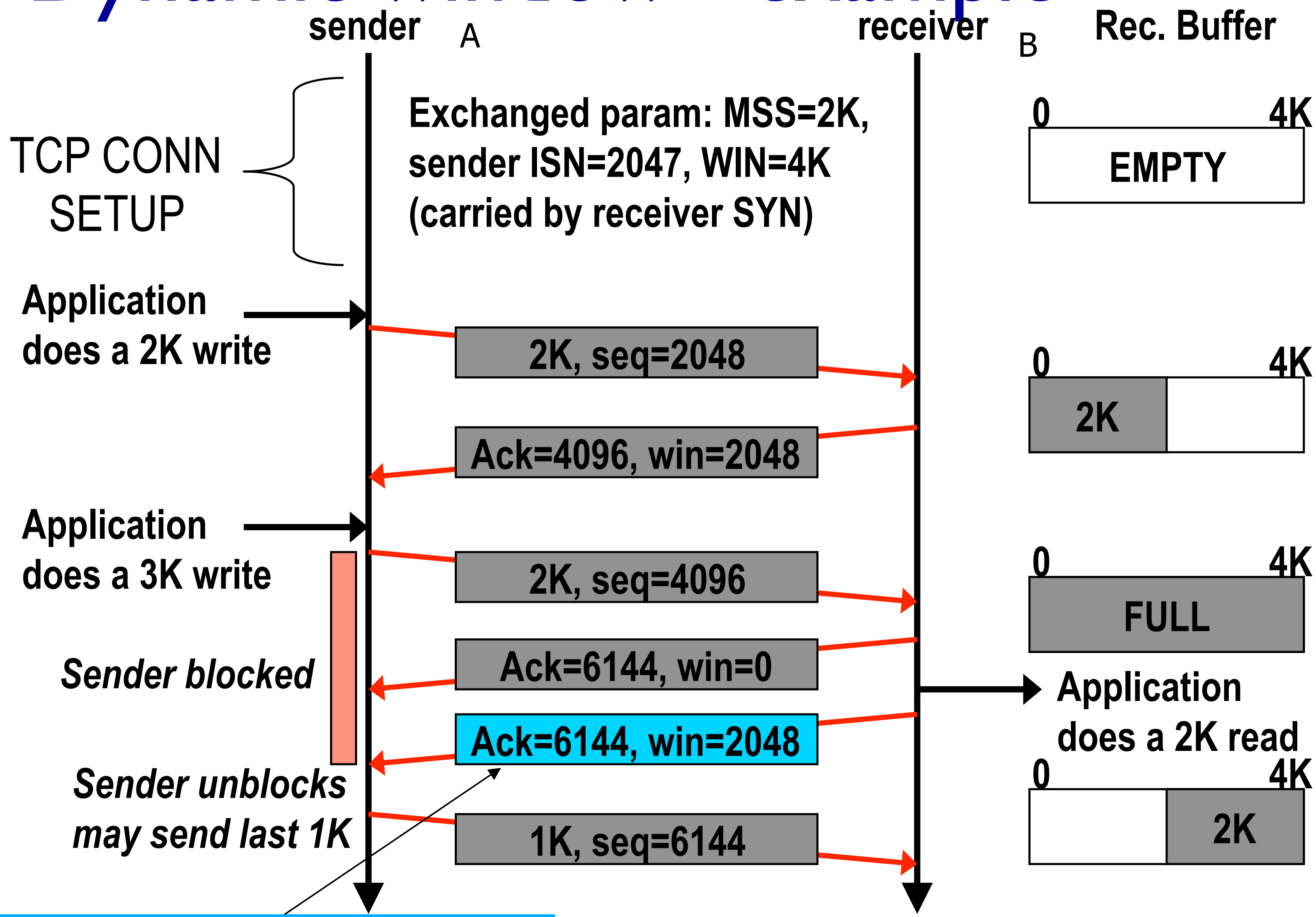


receiver-side buffering

Dynamic window - example



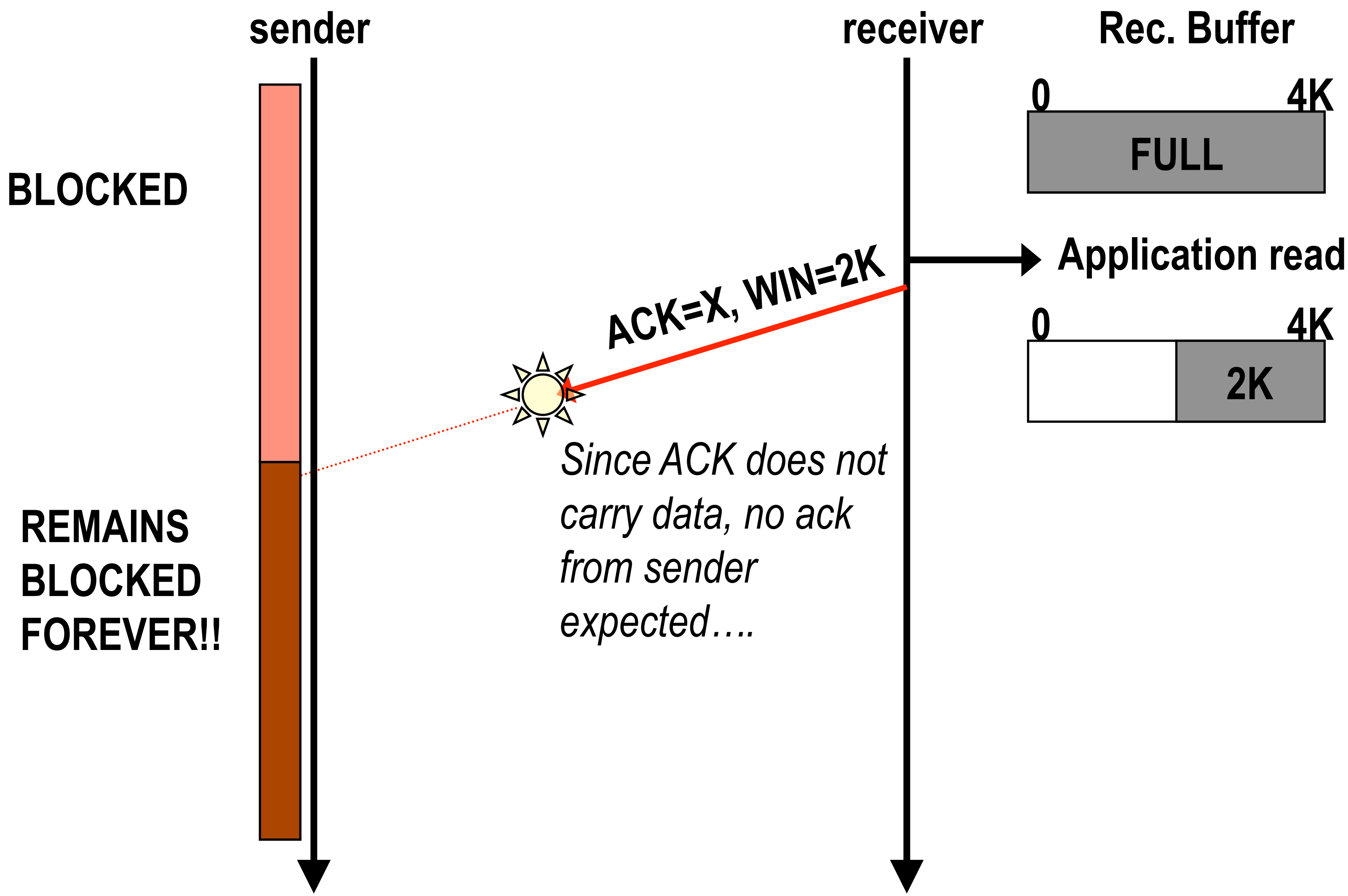
Dynamic window - example



Piggybacked in a packet sent from B to A

Window thus source rate limited by reading speed and buffer size at the receiver

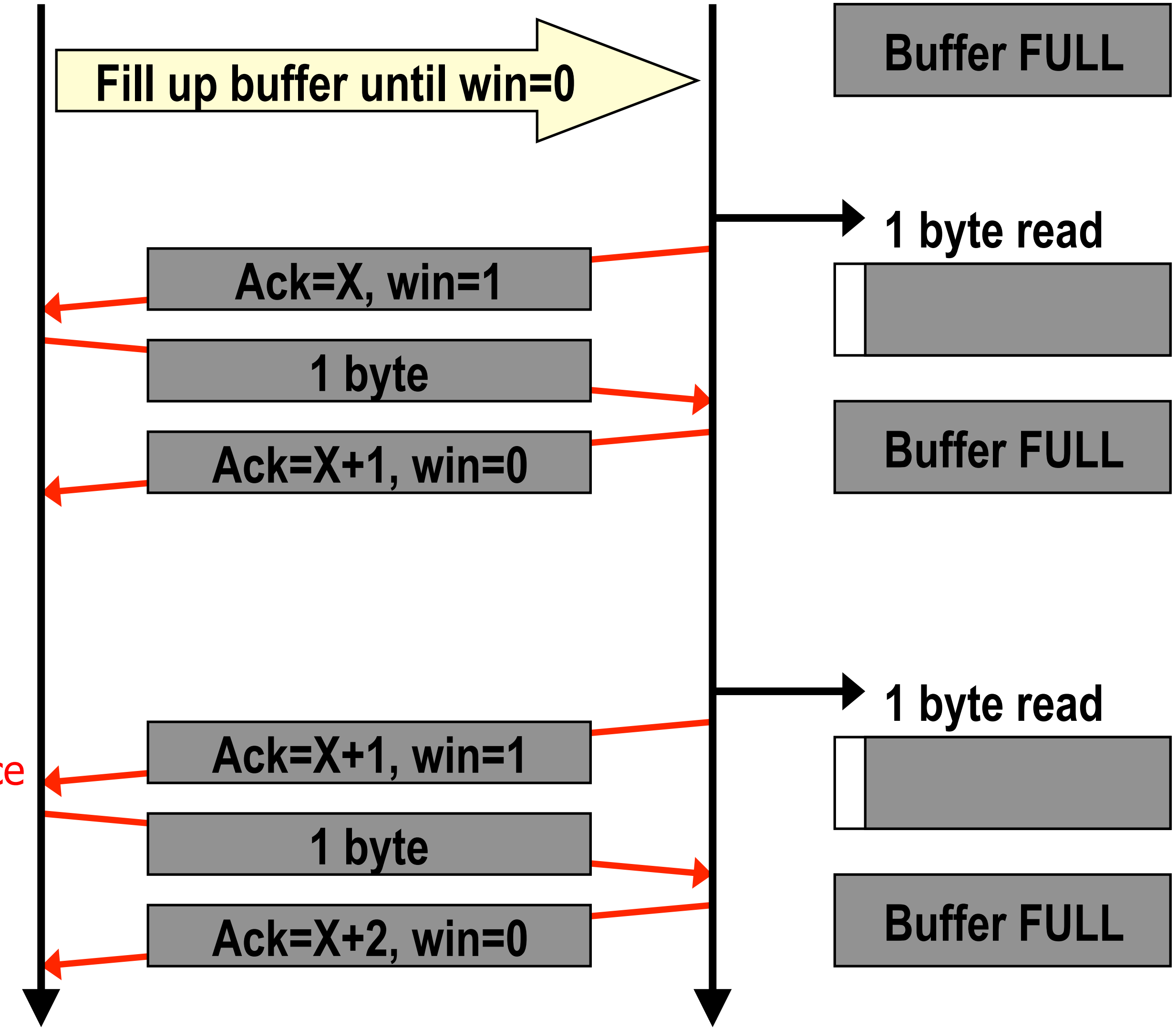
Blocked sender deadlock problem



Solution: Persist timer

- When win=0 (blocked sender), sender starts a "persist" timer
  - Initially 500ms (but depends on implementation)
- When persist timer elapses AND no segment received during this time, sender transmits "probe"
  - Probe = 1byte segment; makes receiver reannounce next byte expected and window size
    - this feature necessary to break deadlock
    - if receiver was still full, rejects byte
    - otherwise acks byte and sends back actual win
- Persist time management (exponential backoff):
  - Doubles every time no response is received
  - Maximum = 60s

The silly window syndrome



Network loaded with tinygrams (40bytes header + 1 payload!!)

Forever!

Anche se il ricevitore e' veloce
A passare i dati al livello
applicativo inviare segmenti
piccoli in un bulk di dati
ha questo effetto

Silly window solution

- Problem discovered by David Clark (MIT), 1982
- easily solved, by preventing receiver to send a window update for 1 byte
- rule: send window update when:
  - receiver buffer can handle a whole MSS

    or
  - half received buffer has emptied (if smaller than MSS)
- sender also may apply rule
  - by waiting for sending data when win low

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

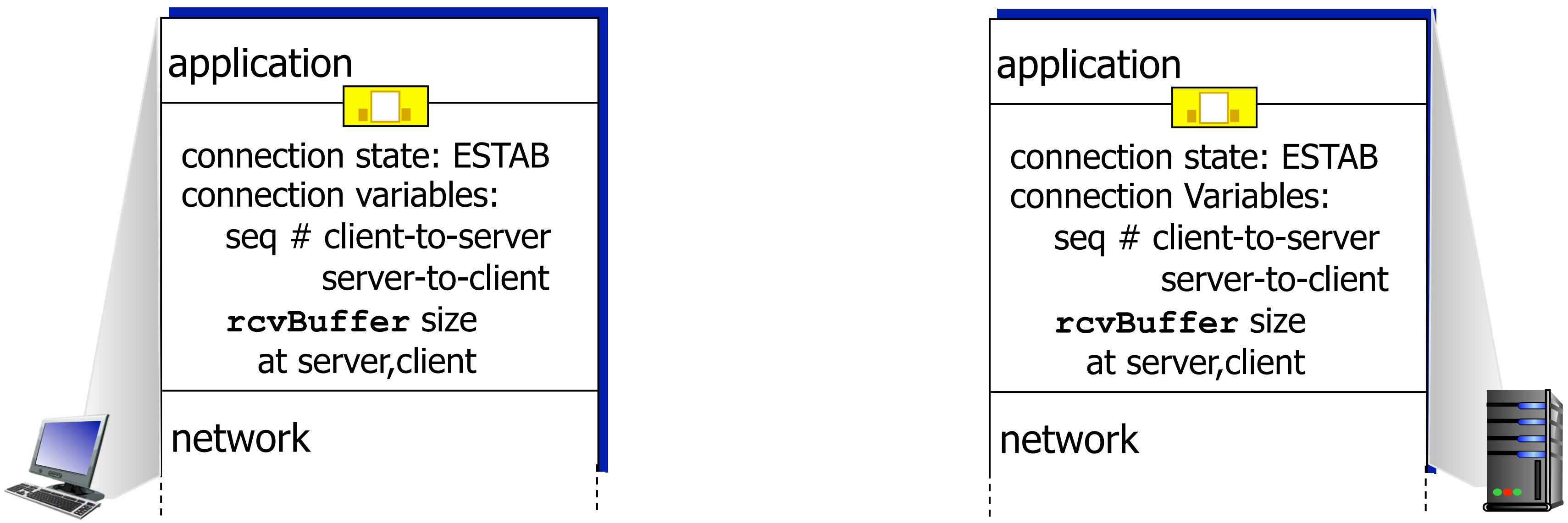
3.6 principles of congestion control

3.7 TCP congestion control

Connection Management

before exchanging data, sender/receiver "handshake":

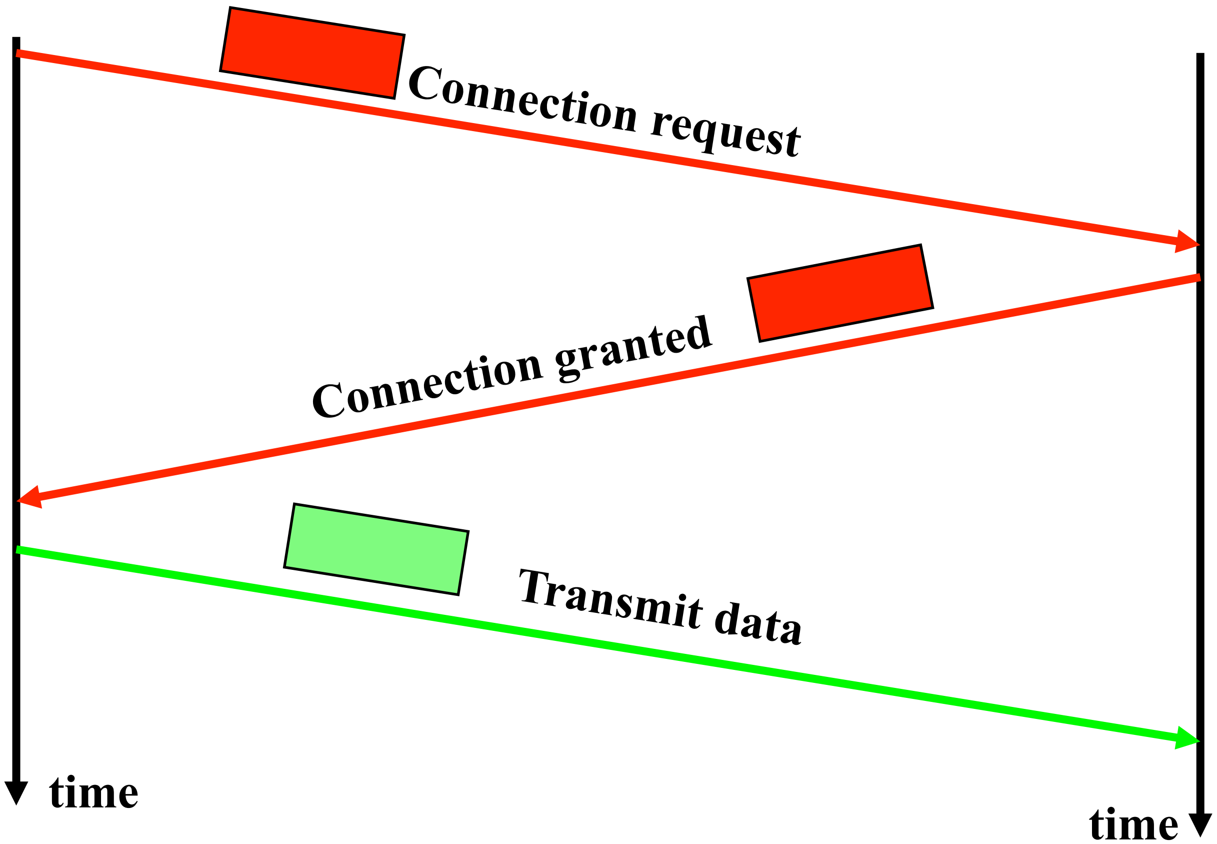
- agree to establish connection (each knowing the other willing to establish connection)
- agree on connection parameters



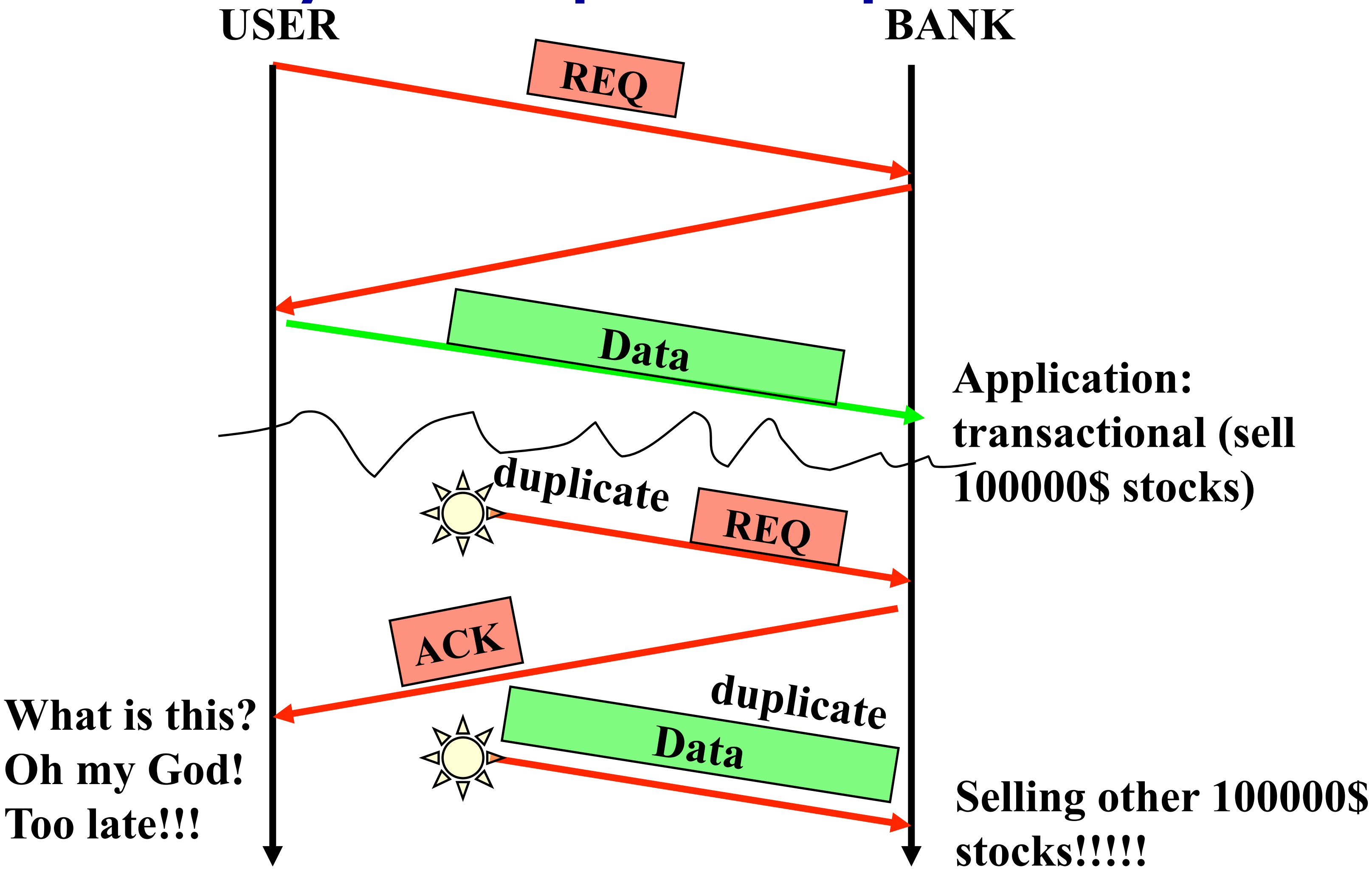

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```

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

Connection establishment: simplest approach (non TCP)

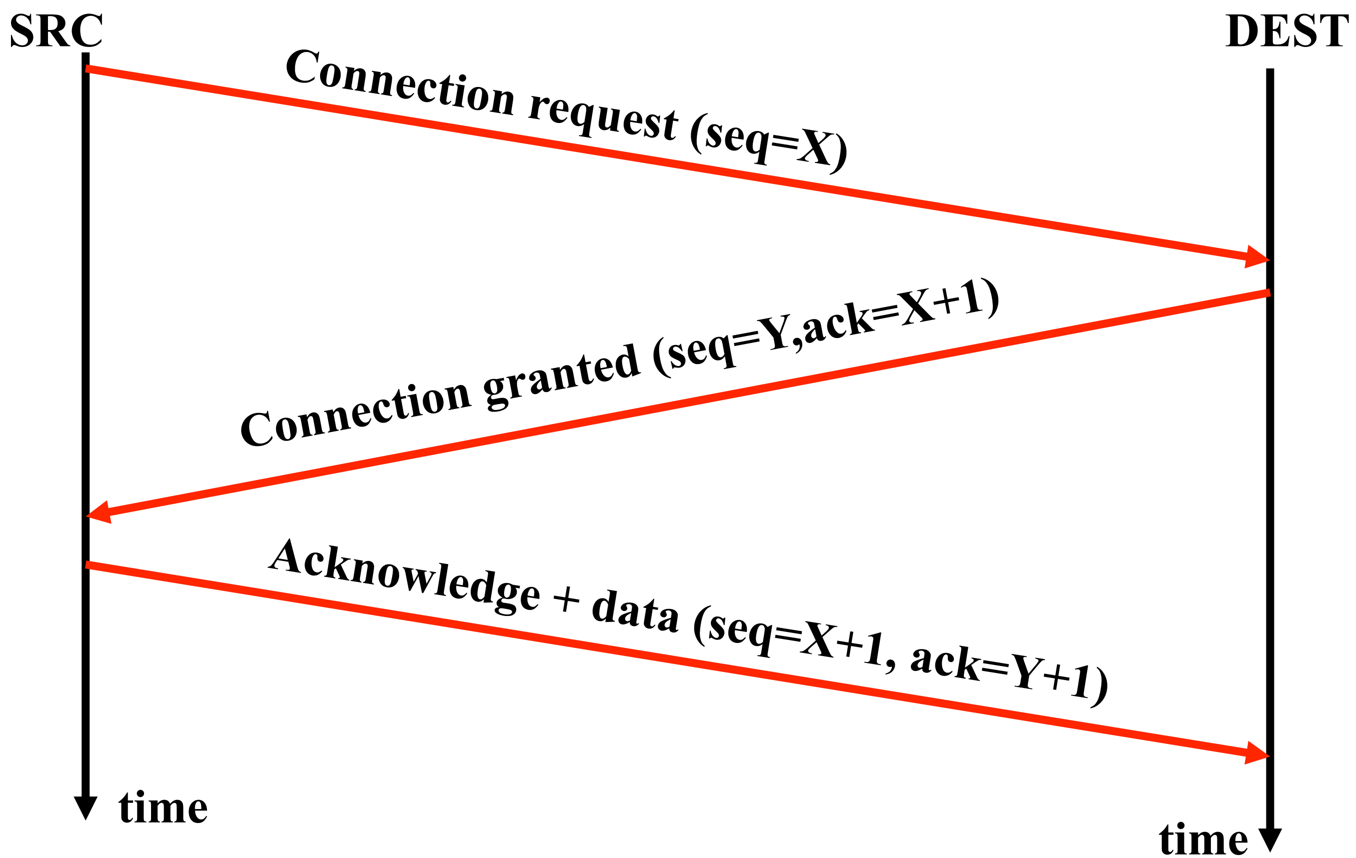


Delayed duplicate problem

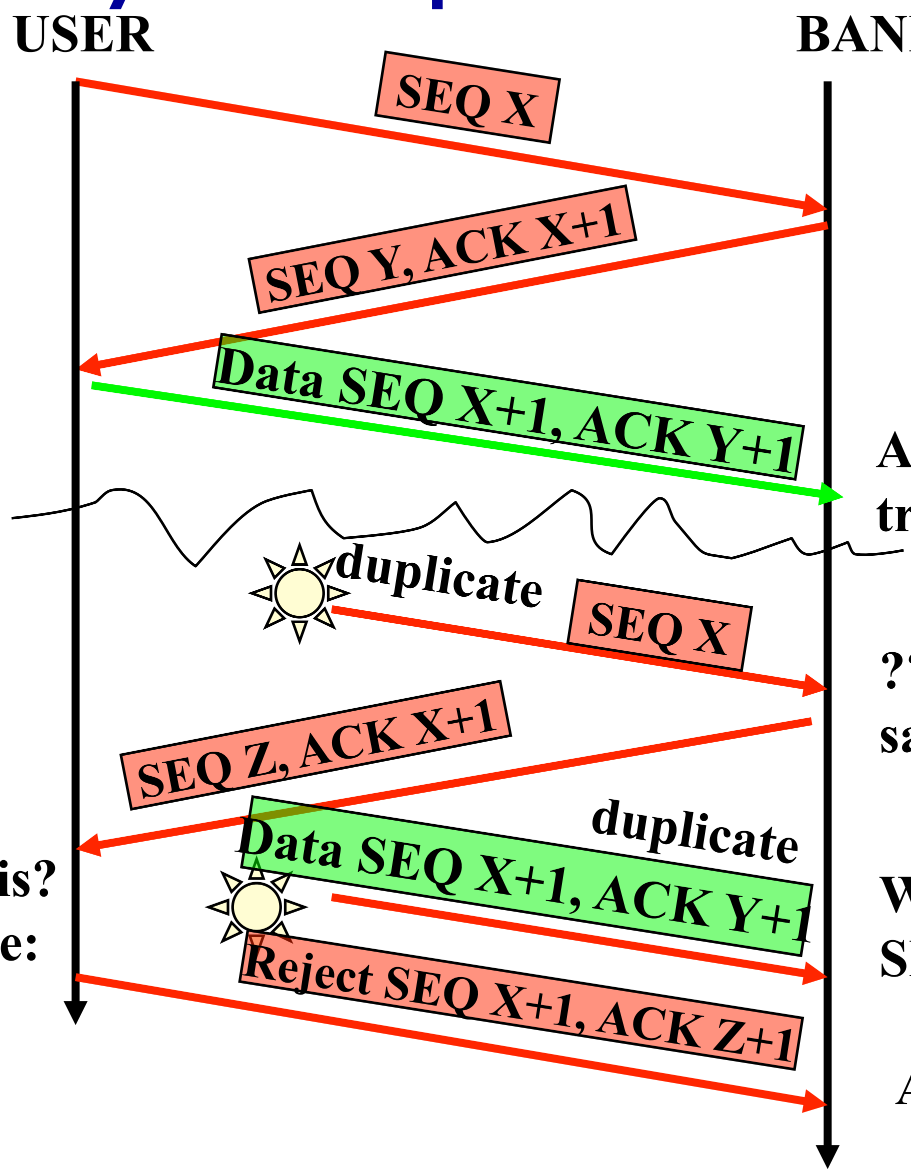


Solution: three way handshake

Tomlinson 1975



Delayed duplicate detection



Application: transactional (selling stocks)

??? What a case: request with same indicator X? anyway...

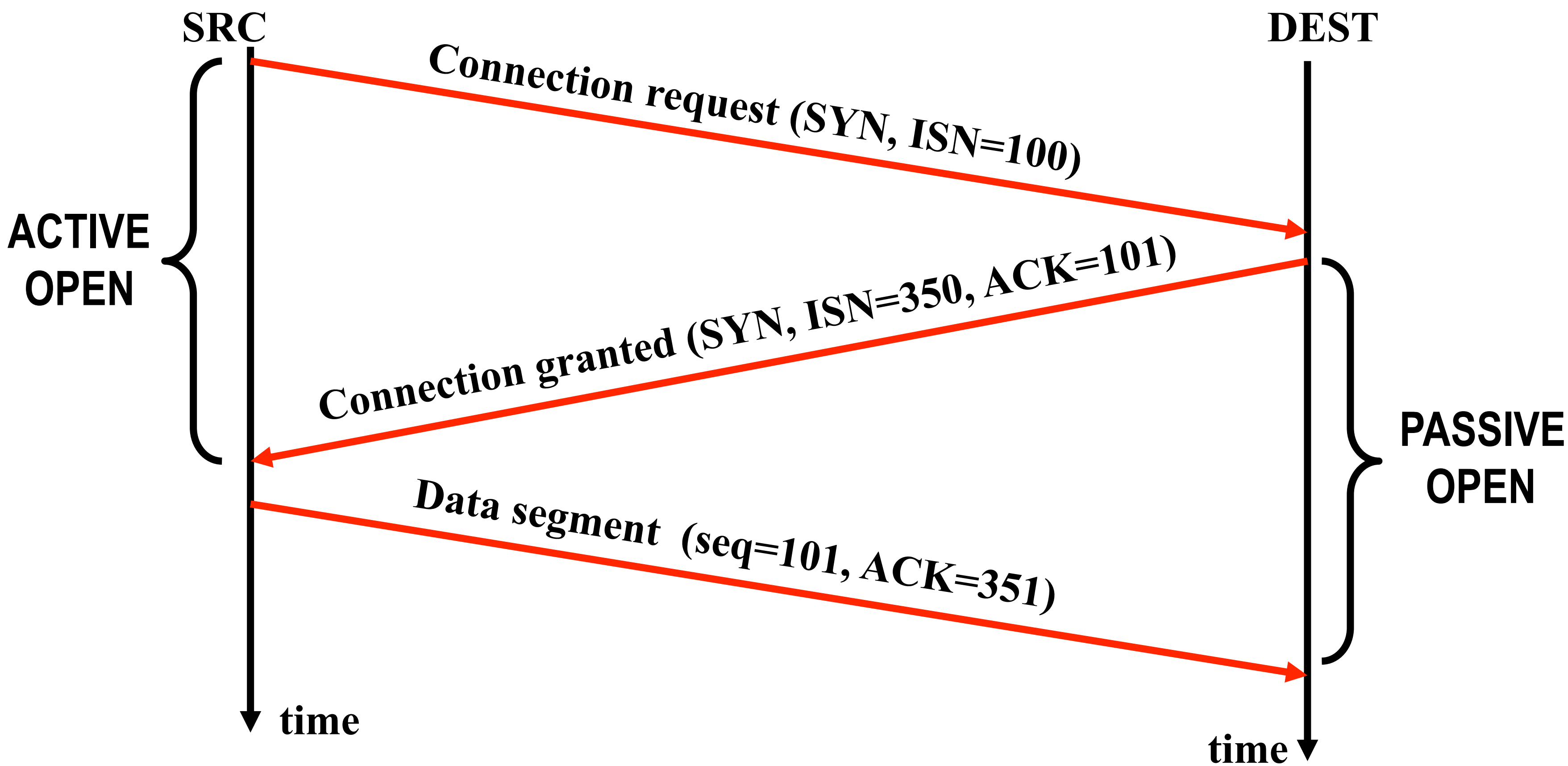
What is this? Not too late:

What is this??? Should be SEQ X, ACK Z!!!! STOP...

Ah ah! Got the problem!

Disaster could not be avoided with a two-way handshake

Three way handshake in TCP



Full *duplex connection: opened in both ways*

SRC: performs ACTIVE OPEN

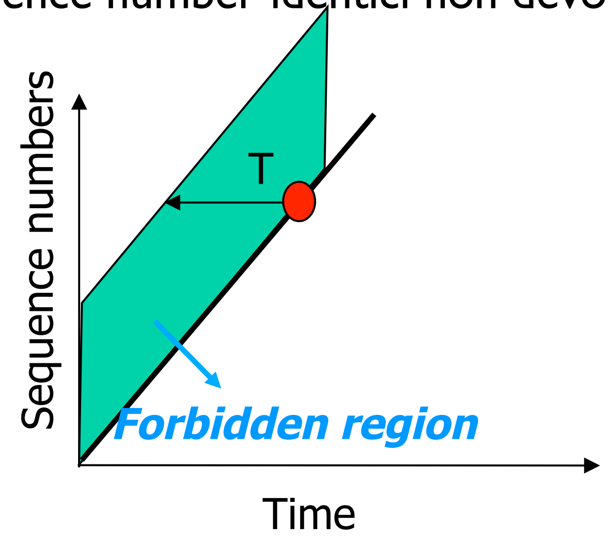
DEST: Performs PASSIVE OPEN

Initial Sequence Number

- Should change in time
  - RFC 793 (but not all implementations are conforming) suggests to generate ISN as a sample of a 32 bit counter incrementing at 4μs rate (4.55 hour to wrap around—Maximum Segment Lifetime much shorter)
- transmitted whenever SYN (Synchronize sequence numbers) flag active
  - note that both src and dest transmit THEIR initial sequence number (remember: full duplex)
- Data Bytes numbered from ISN+1
  - necessary to allow SYN segment ack

Forbidden Region

- Obiettivo: due sequence number identici non devono trovarsi in rete allo stesso tempo



- Aging dei pacchetti→ dopo un certo tempo MSL (Maximum Segment Lifetime) i pacchetti eliminati dalla rete
- Initial sequence numbers basati sul clock
- Un ciclo del clock circa 4 ore; MSL circa 2 minuti.
- → Se non ci sono crash che fanno perdere il valore dell'ultimo initial sequence number usato NON ci sono problemi (si riusa lo stesso initial sequence number ogni 4 ore circa, quando il segmento precedentemente trasmesso con quel sequence number non è più in rete) e non si esauriscono in tempo <MSL i sequence number
- → Cosa succede nel caso di crash? RFC suggerisce l'uso di un 'periodo di silenzio' in cui non vengono inviati segmenti dopo il riavvio pari all'MSL (per evitare che pacchetti precedenti connessioni siano in giro).

TCP Connection Management:Summary

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

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

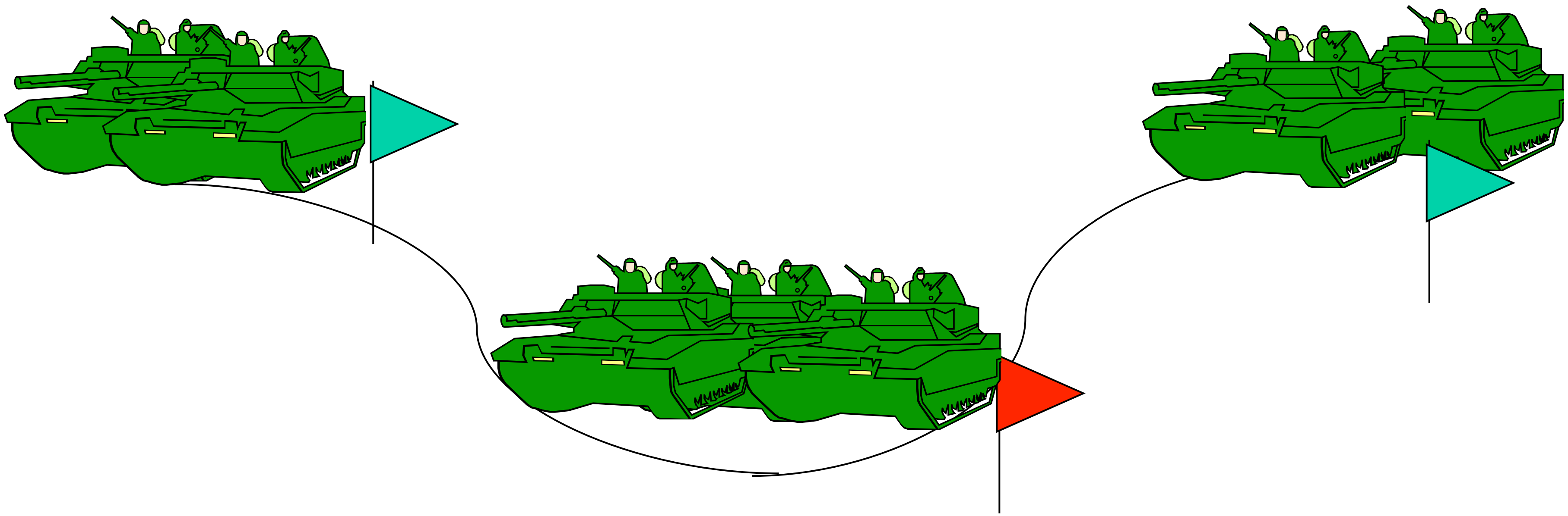
- server allocates buffers
- specifies server initial seq. #

Step 3: client receives SYNACK, allocates buffer and variables,replies with ACK segment, which may contain data

Per chiudere la connessione uno dei due estremi invia un messaggio con FIN flag a 1 a cui l'altro estremo della connessione risponde con ACK

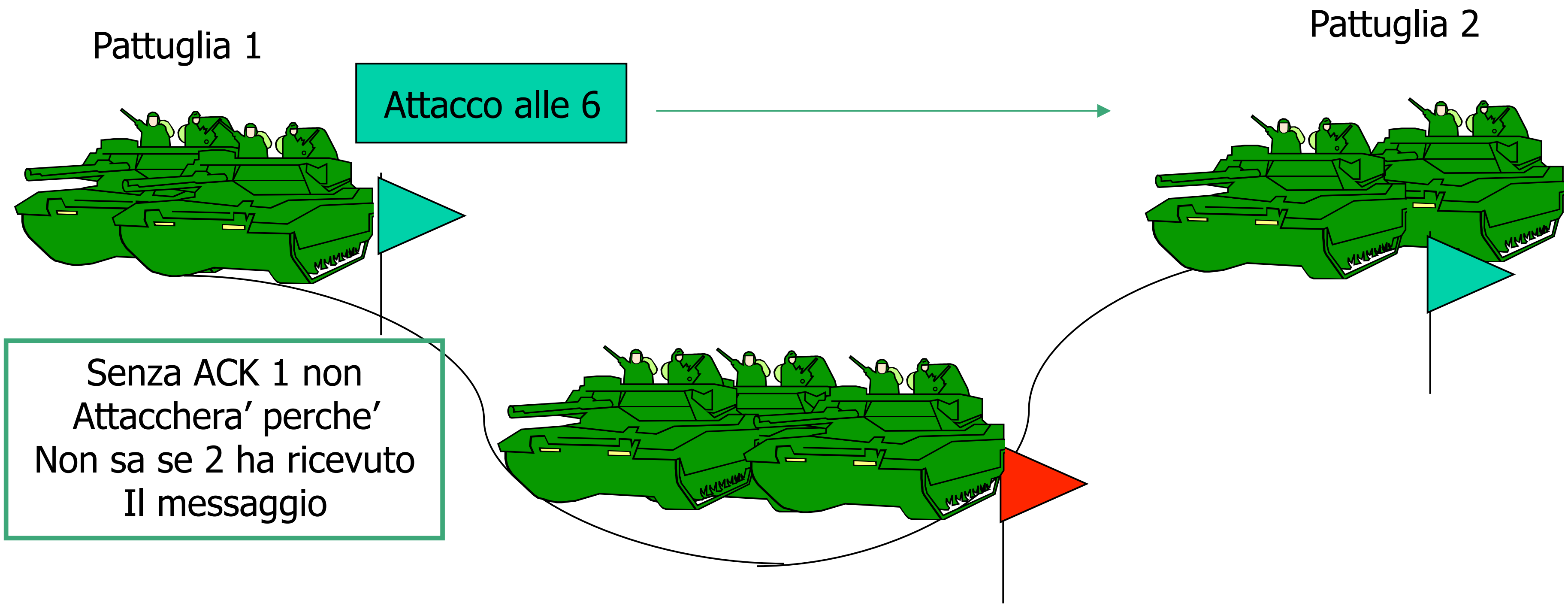
Problema dei due eserciti

- L'esercito rosso e' globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all'orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero' essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



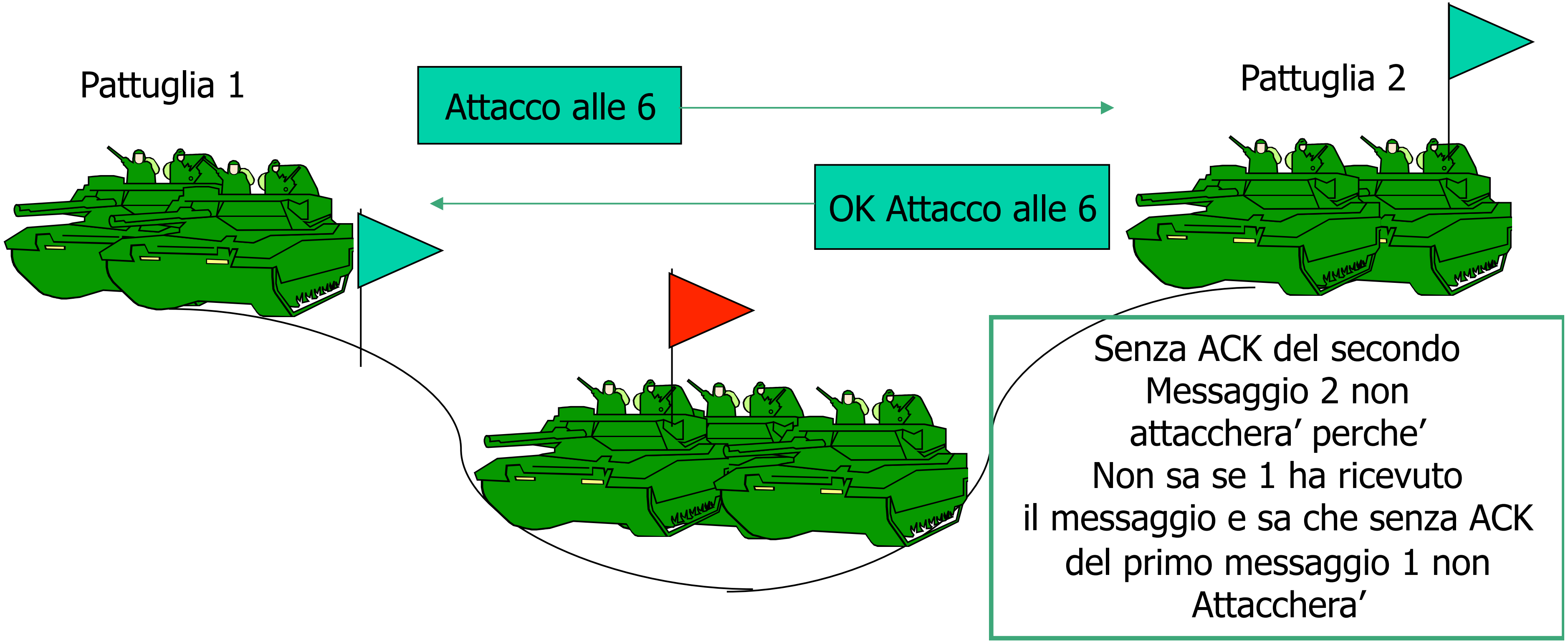
Problema dei due eserciti

- L'esercito rosso e' globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all'orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero' essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



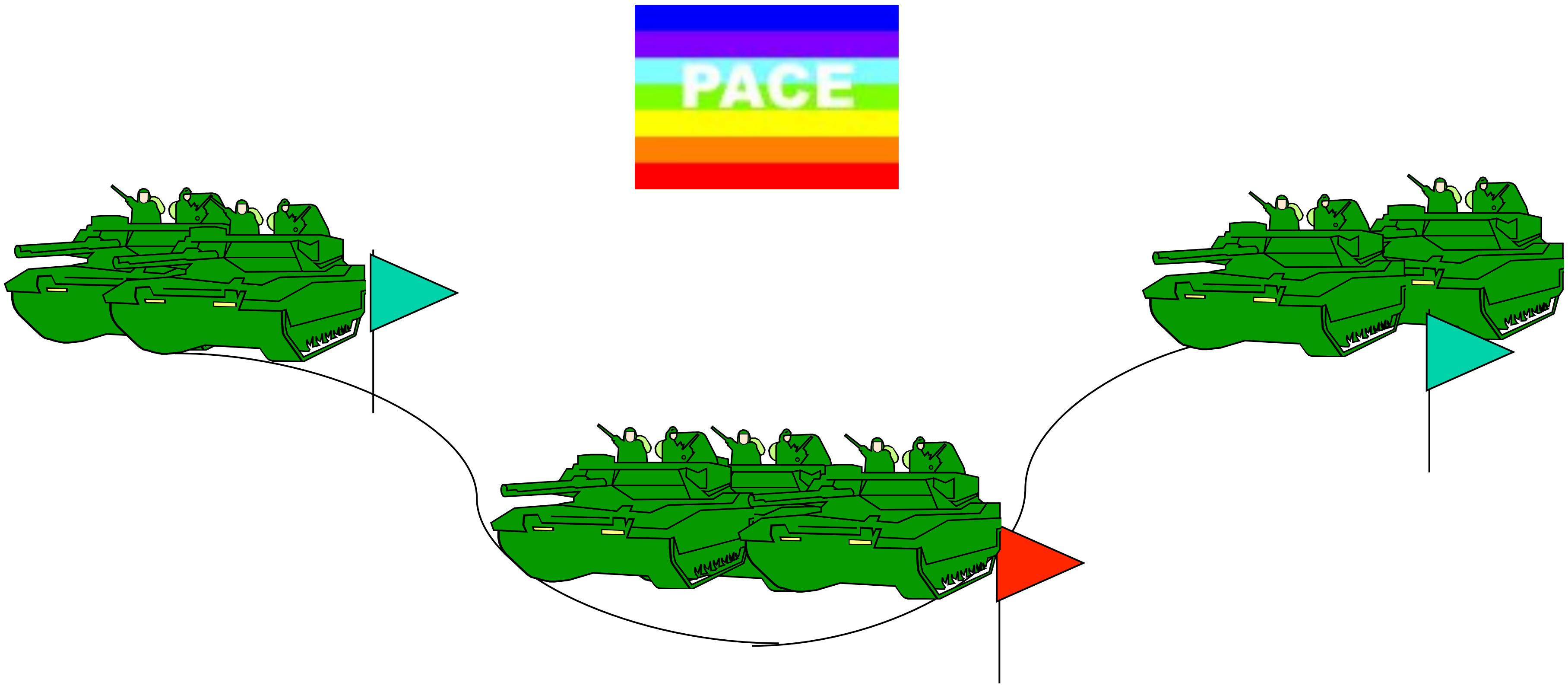
Problema dei due eserciti

- L'esercito rosso e' globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all'orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero' essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



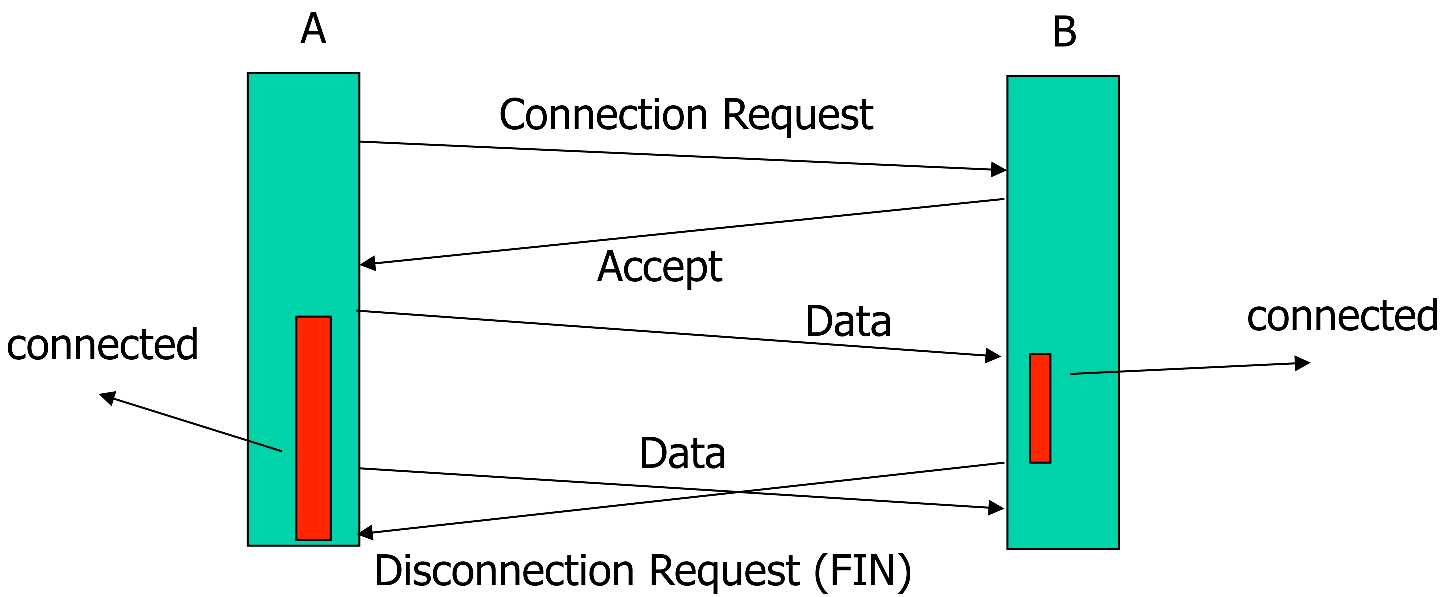
Problema dei due eserciti

- In generale: se N scambi di messaggi /Ack etc. necessari a raggiungere la certezza dell'accordo per attaccare allora cosa succede se l'ultimo messaggio 'necessario' va perso?
- →E' impossibile raggiungere questa certezza. Le due pattuglie non attaccheranno mai!!



Problema dei due eserciti: cosa ha a che fare con le reti e TCP??

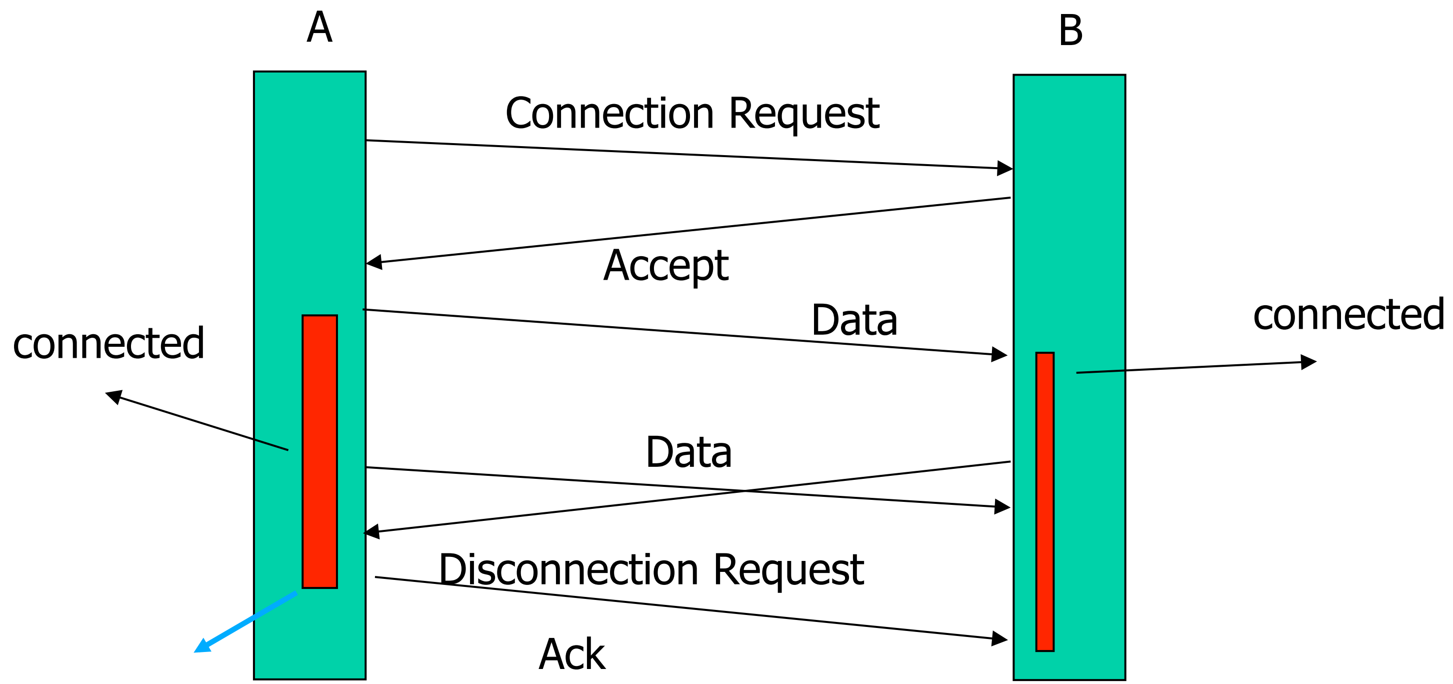
- Chiusura di una connessione. Vorremmo un accordo tra le due peer entity o rischiamo di perdere dati.



A pensa che il secondo pacchetto sia stato ricevuto. La connessione e' Stata chiusa da B prima che ciò avvenisse→ secondo pacchetto perso!!!

Quando si può dire che le due peer entity abbiano raggiunto un accordo???

- Problema dei due eserciti!!!



Ma se l'ACK va perso????

Soluzione: si e' disposti a correre piu' rischi quando si butta giu' una connessione d quando si attacca un esercito nemico. Possibili malfunzionamenti. Soluzioni 'di recovery' in questi casi

TCP Connection Management (cont.)

Since it is impossible to solve the proble use simple solution: two way handshake

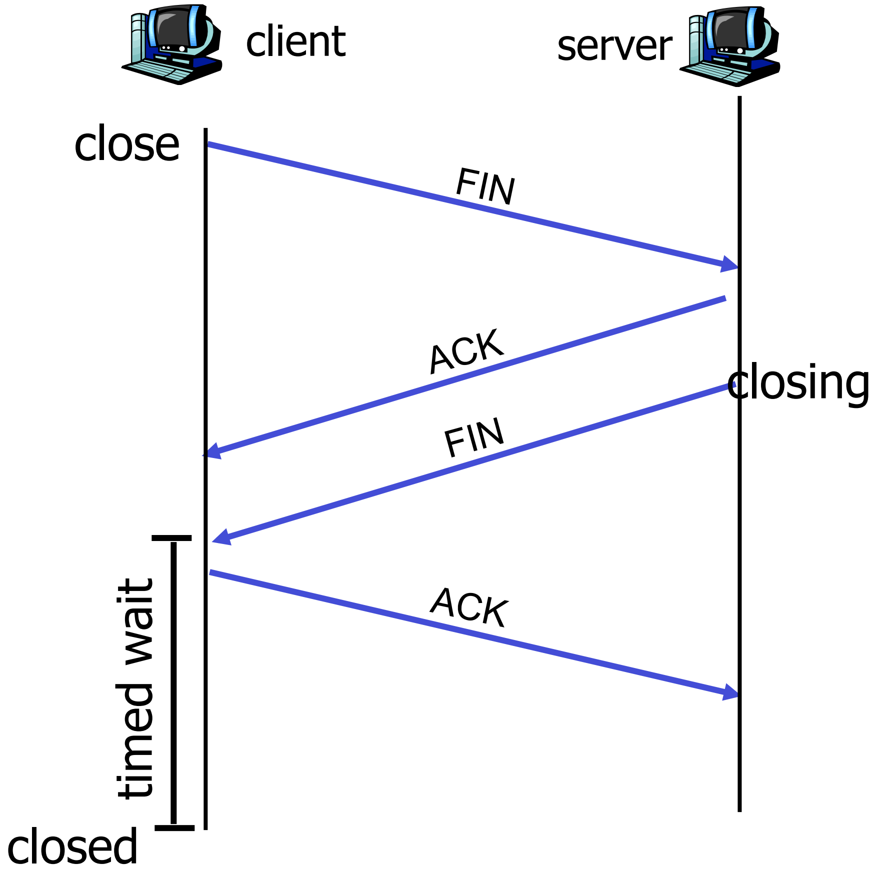
Closing a connection:

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.

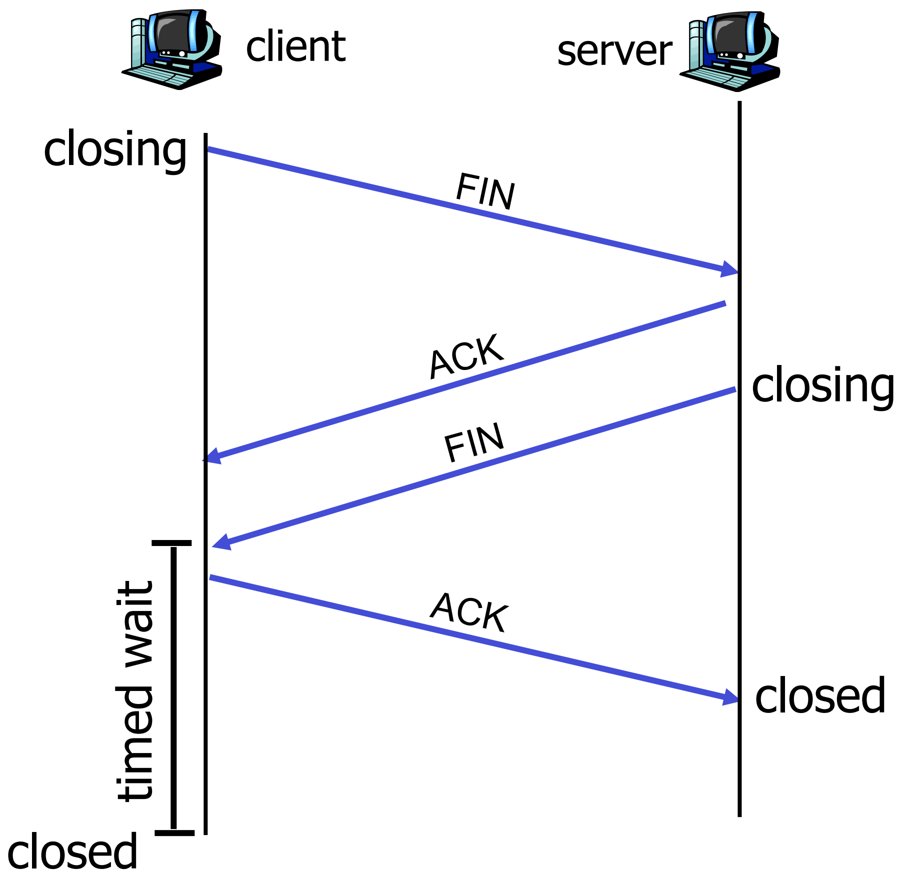


TCP Connection Management (cont.)

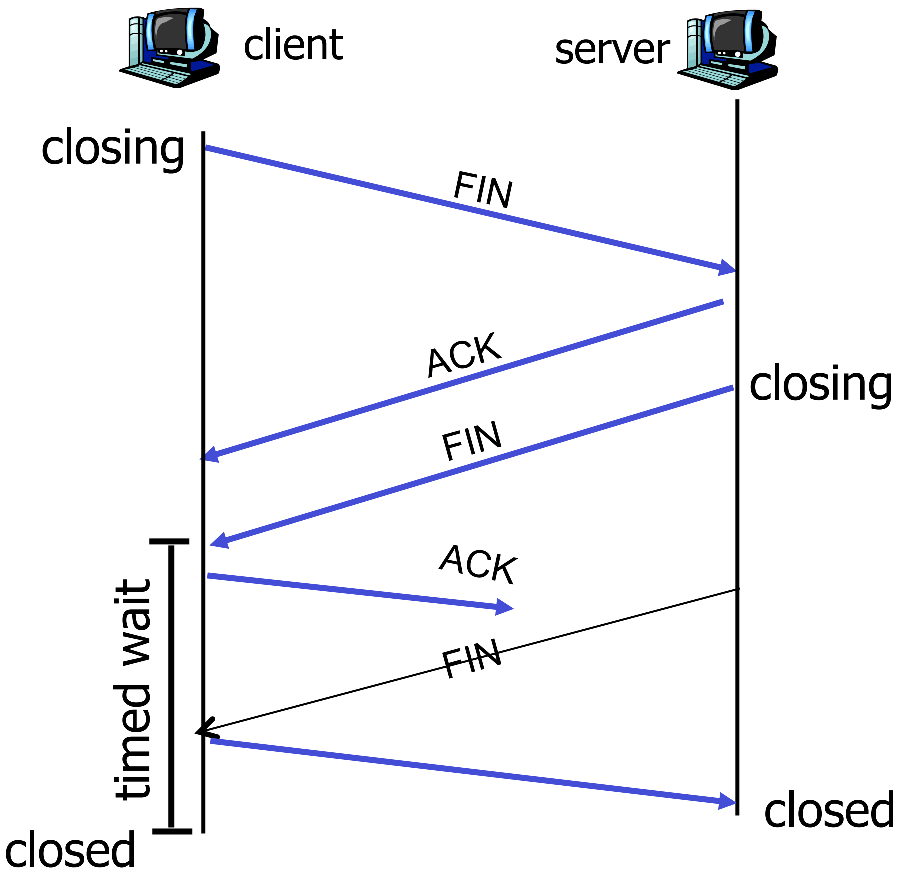
Step 3: client receives FIN, replies with ACK.

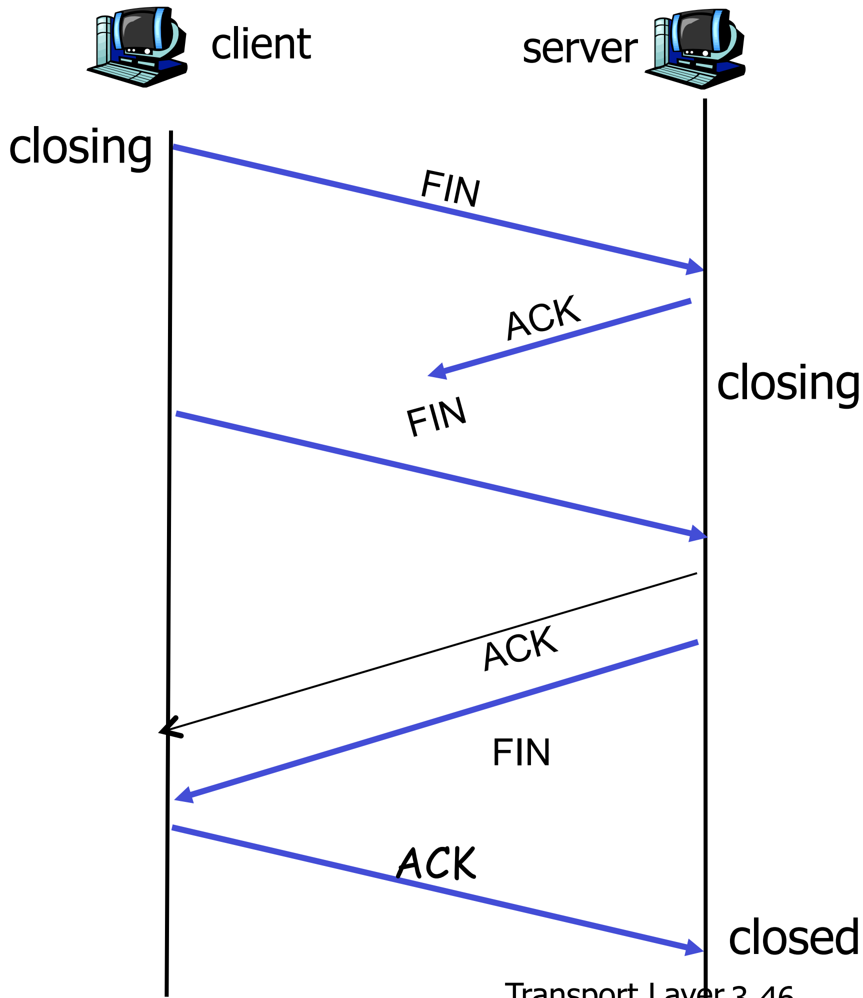
- Enters "timed wait" - will respond with ACK to received FINs

Step 4: server, receives ACK. Connection closed.

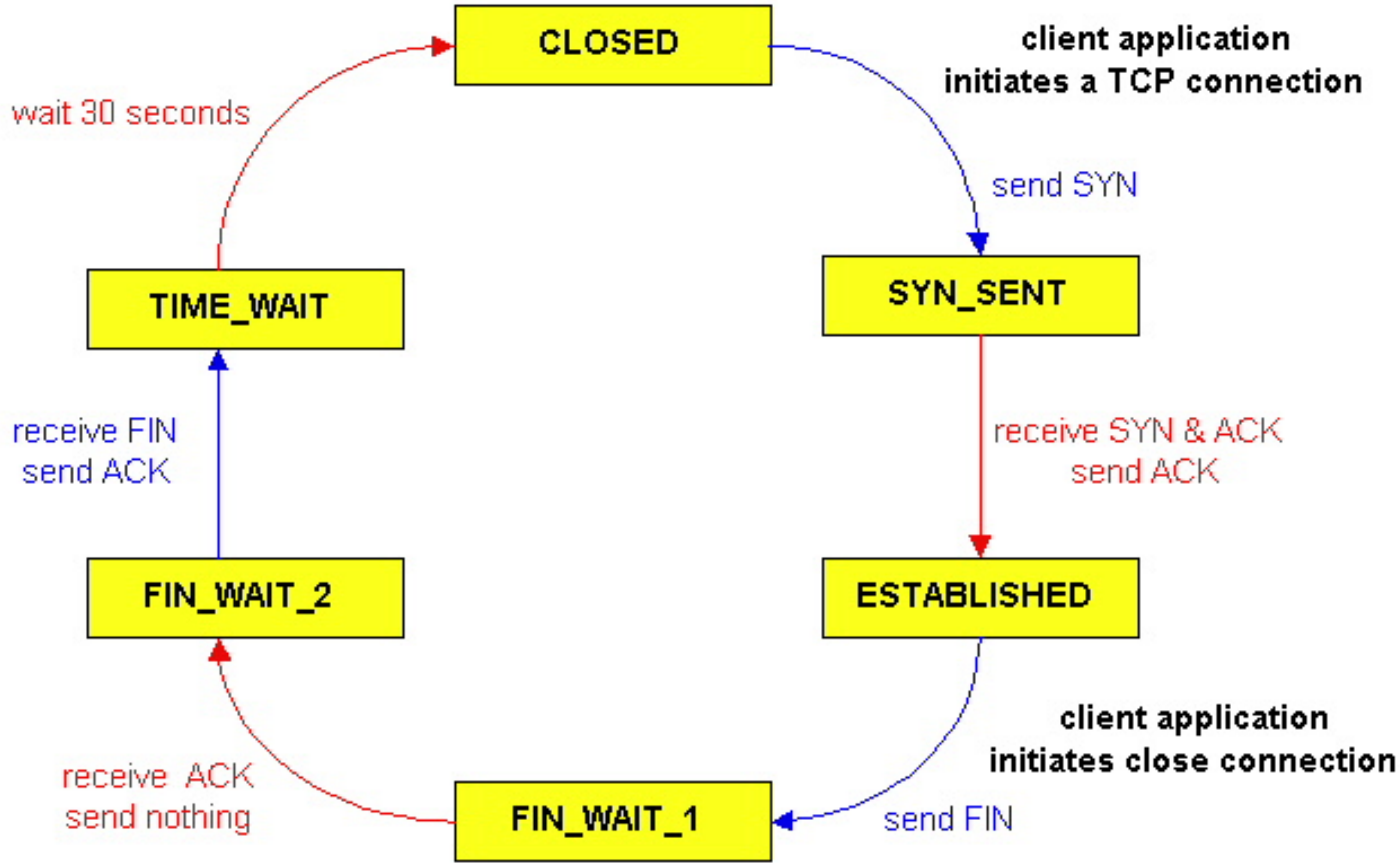


TCP Connection Management (examples)

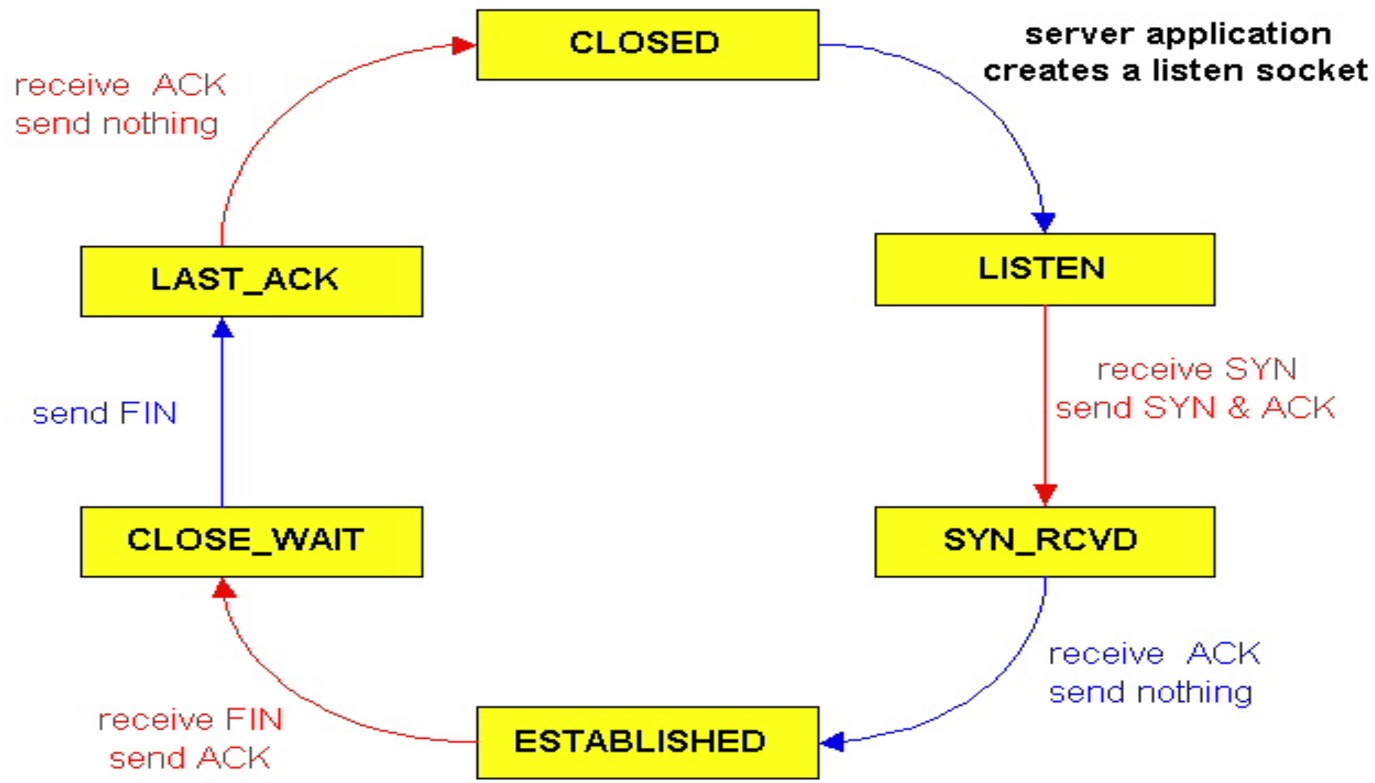




Connection states - Client



Connection States - Server



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

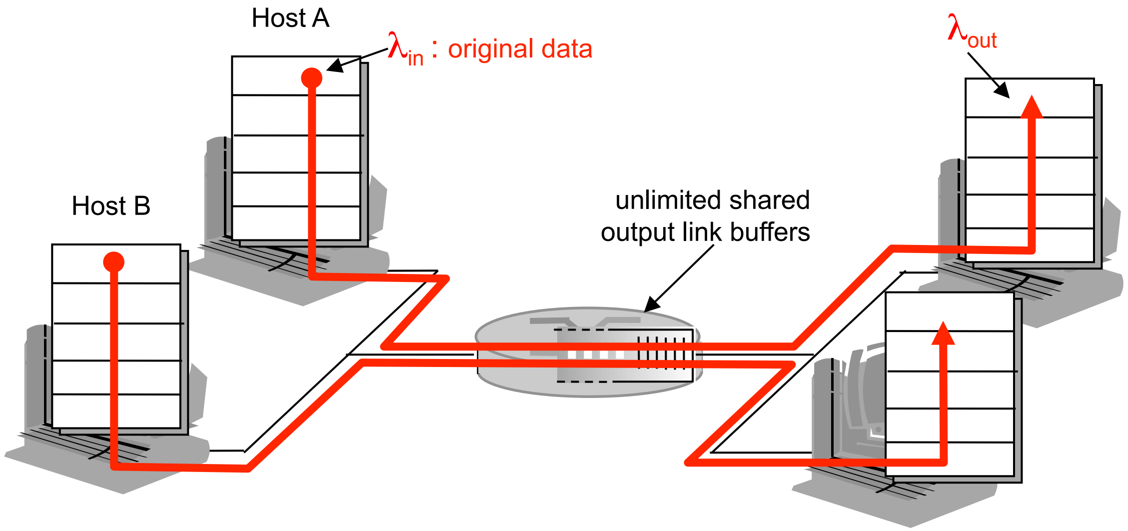
Principles of congestion control

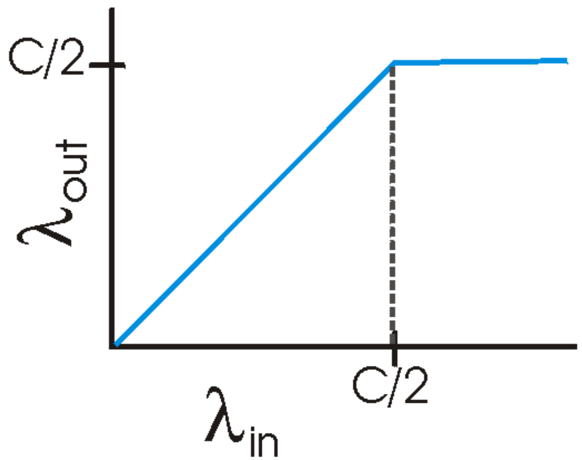
congestion:

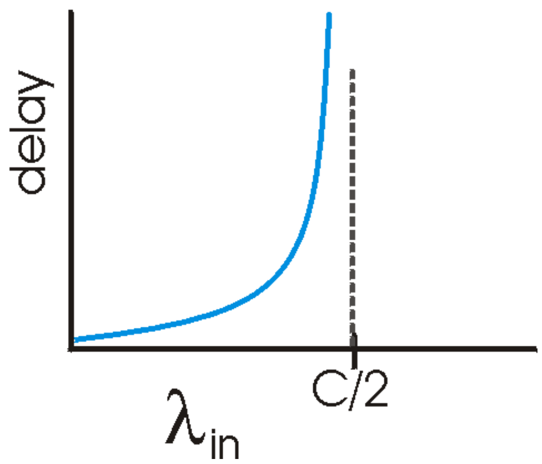
- informally: "too many sources sending too much data too fast for *network* to handle"
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

- two senders, two receivers
- one router, infinite buffers
- no retransmission



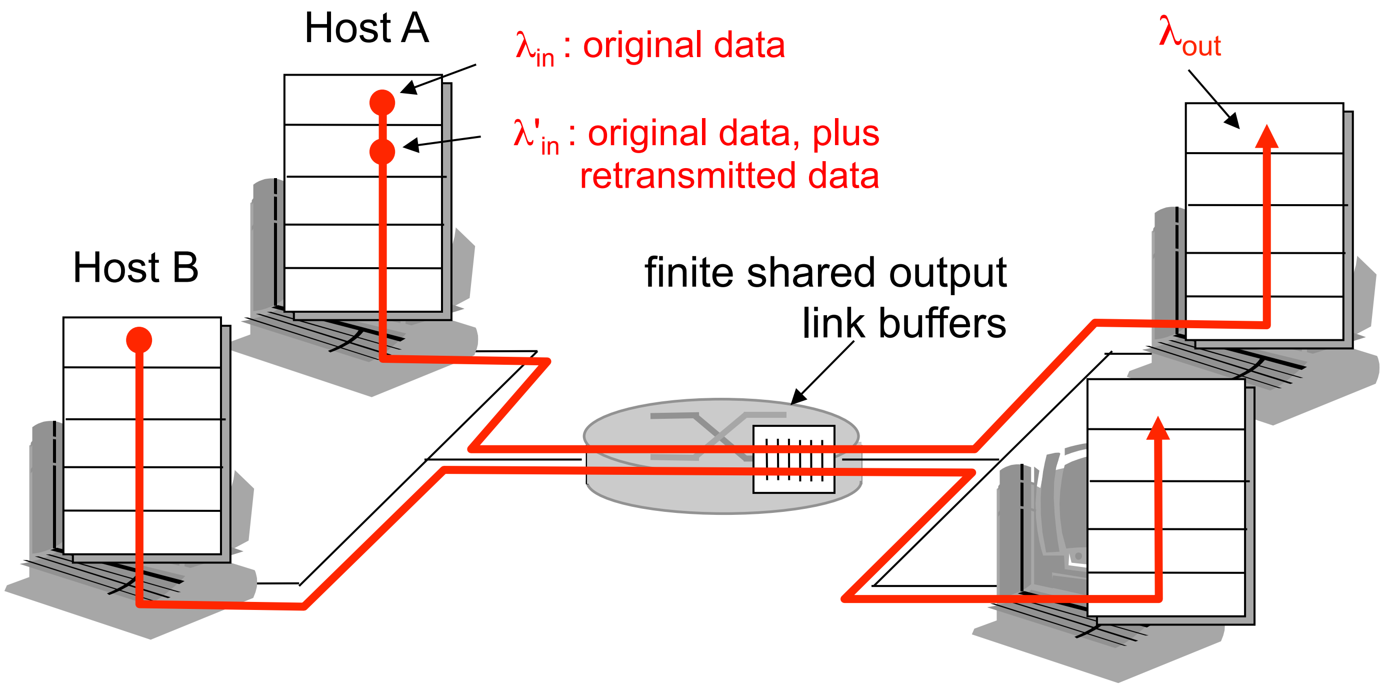




- large delays when congested
- maximum achievable throughput

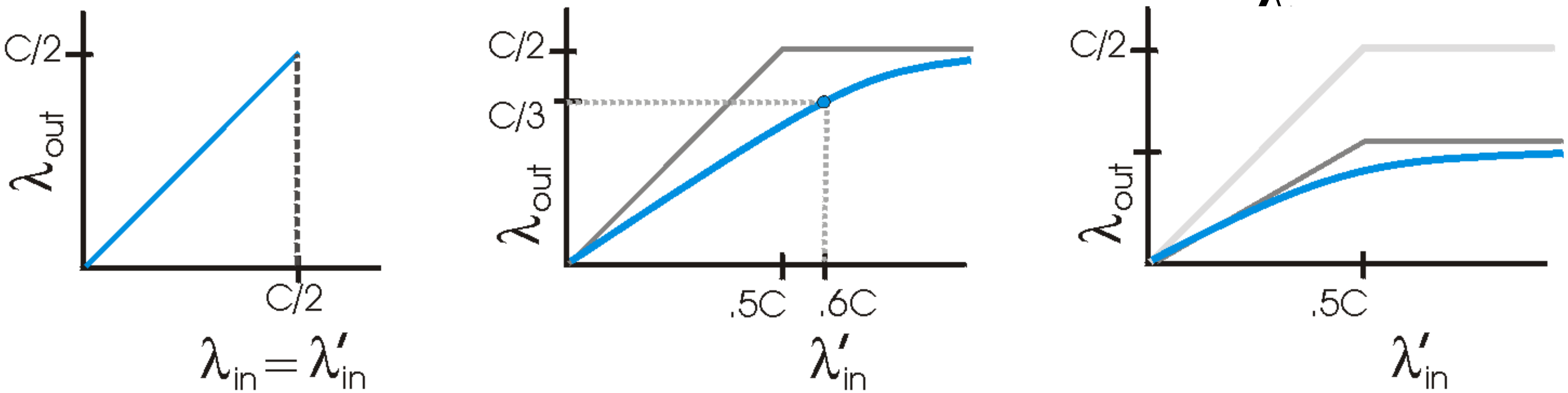
Causes/costs of congestion: scenario 2

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



Causes/costs of congestion: scenario 2

- always we want: $\lambda_{in} = \lambda_{out}$ (goodput)
- Second step …retransmission only when loss: $\lambda'_{in} > \lambda_{out}$
- **retransmission of delayed (not lost) packet** makes λ' larger (than second case) for same λ_{out}



Caso in cui ciascun pacchetto instradato
Sia trasmesso mediamente due volte dal router

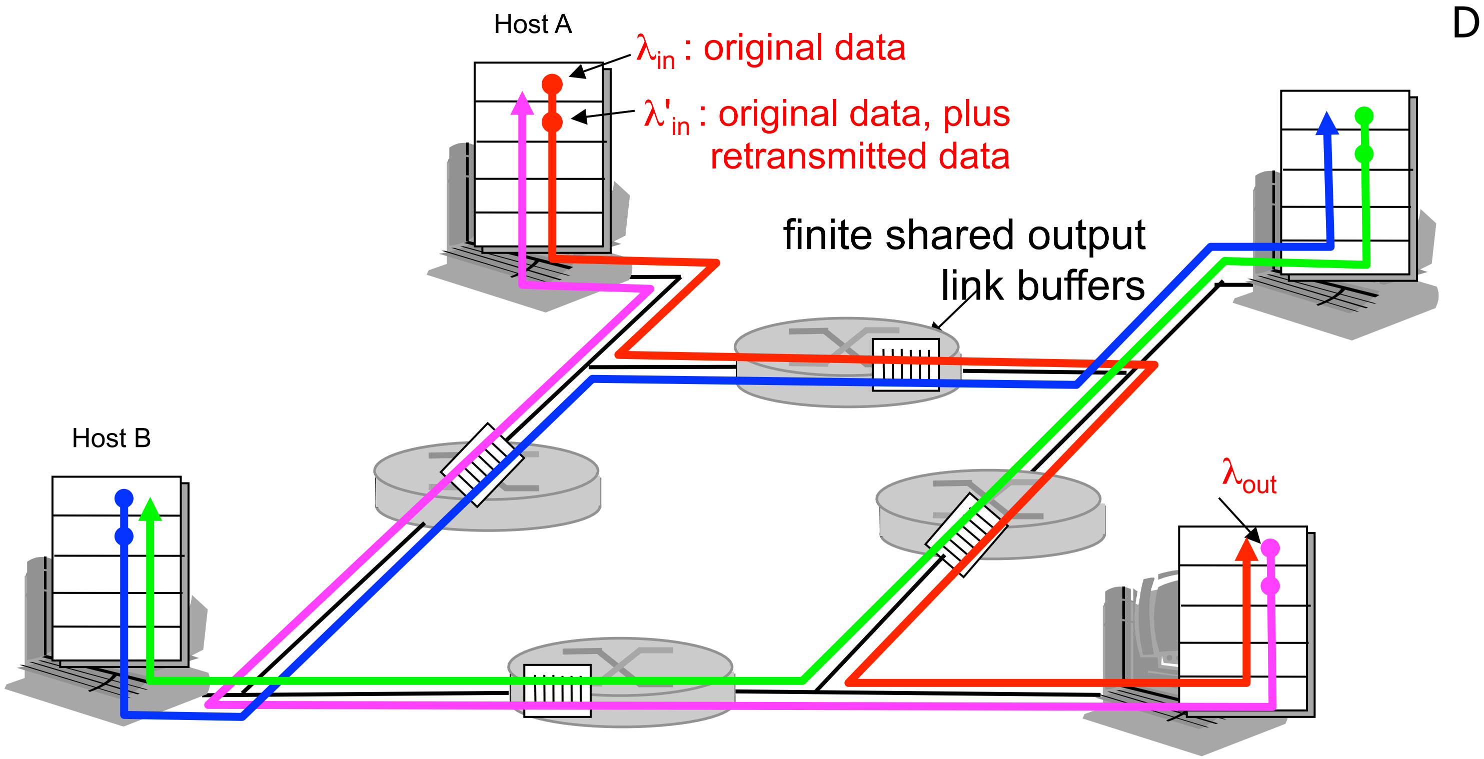
"costs" of congestion:

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

Causes/costs of congestion: scenario 3

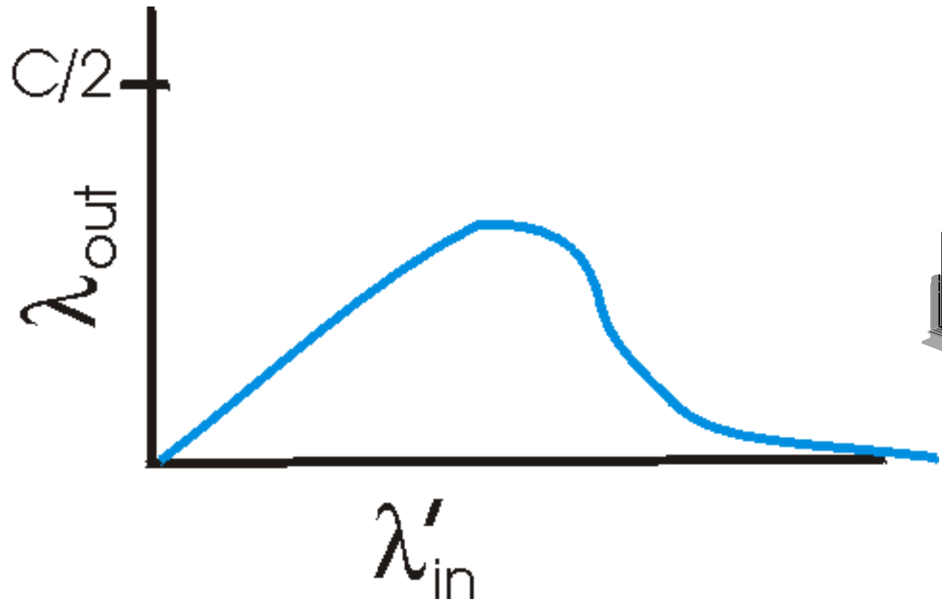
- four senders
- multihop paths
- timeout/retransmit

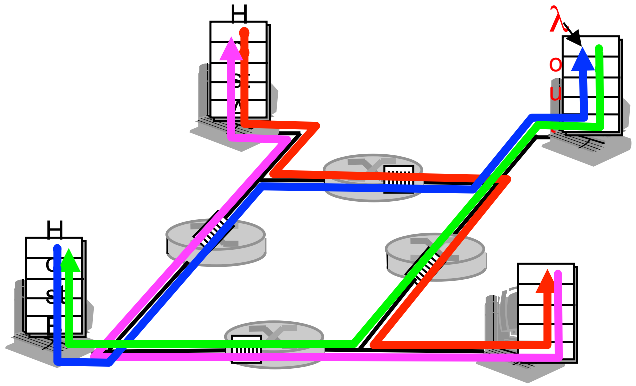
Q: what happens as λ_{in} and λ'_{in} increase ?



D-B traffic high

Causes/costs of congestion: scenario 3





Another "cost" of congestion:

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

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

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

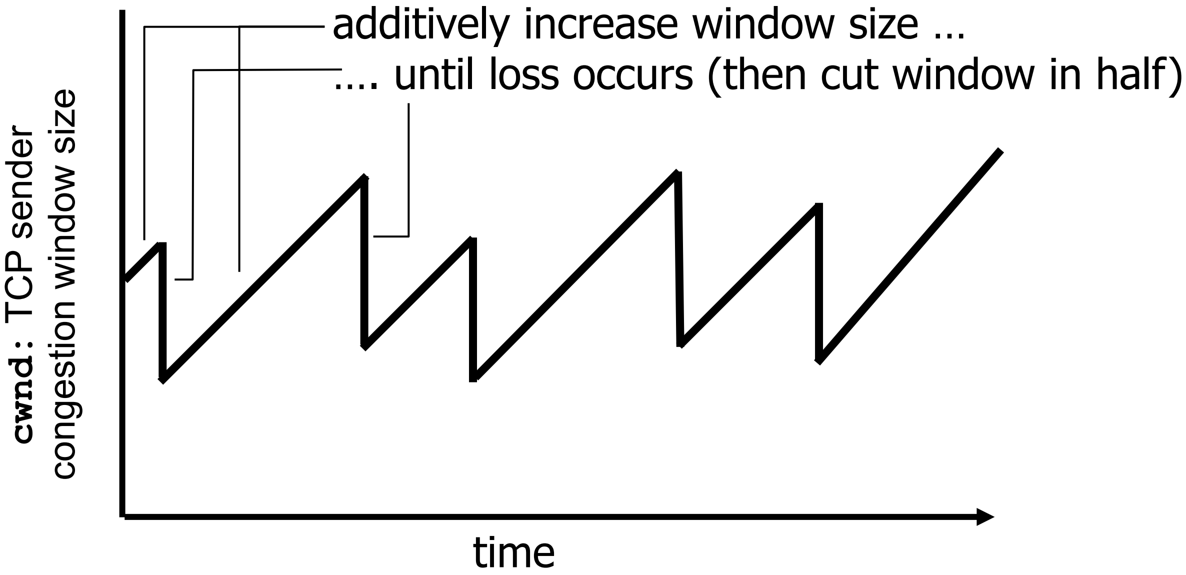
3.6 principles of congestion control

3.7 TCP congestion control

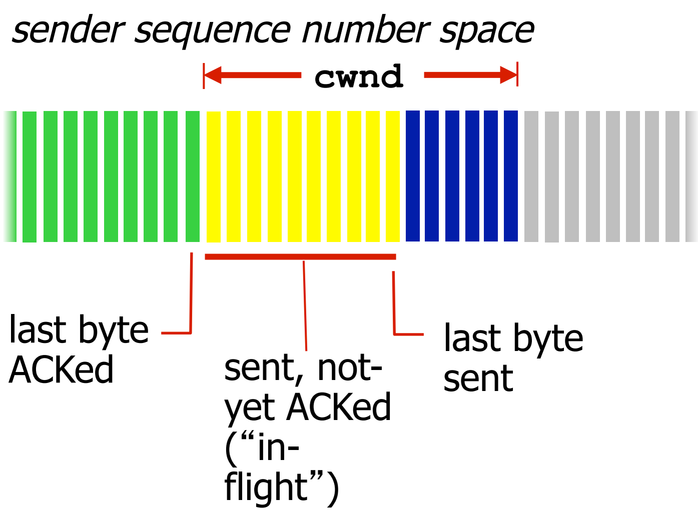
TCP congestion control: additive increase multiplicative decrease

- *approach:* 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



TCP Congestion Control: details



- sender limits transmission:

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

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

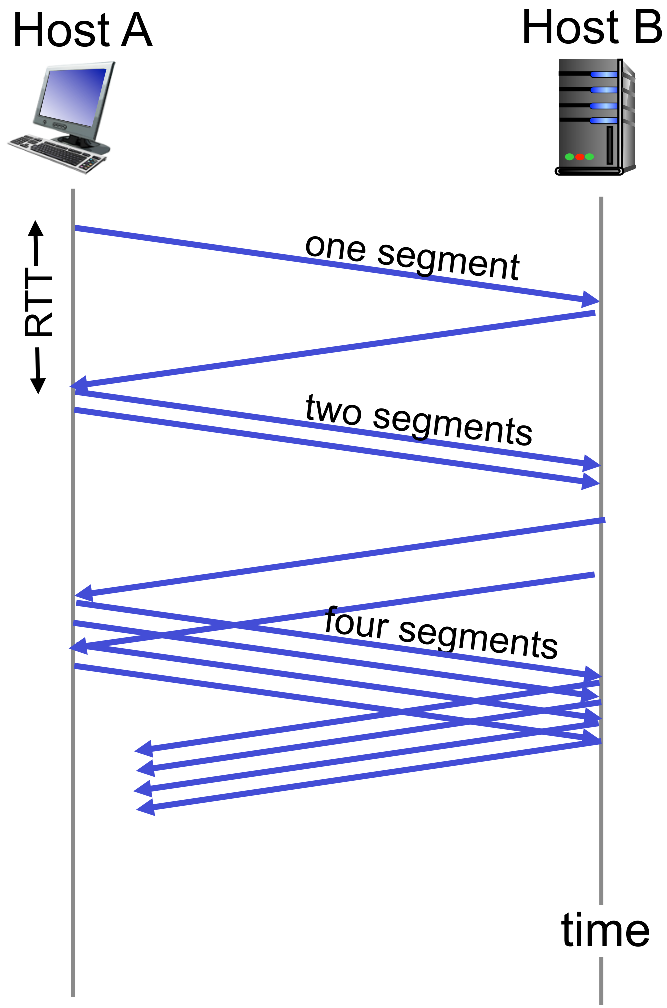
TCP sending rate:

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

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

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



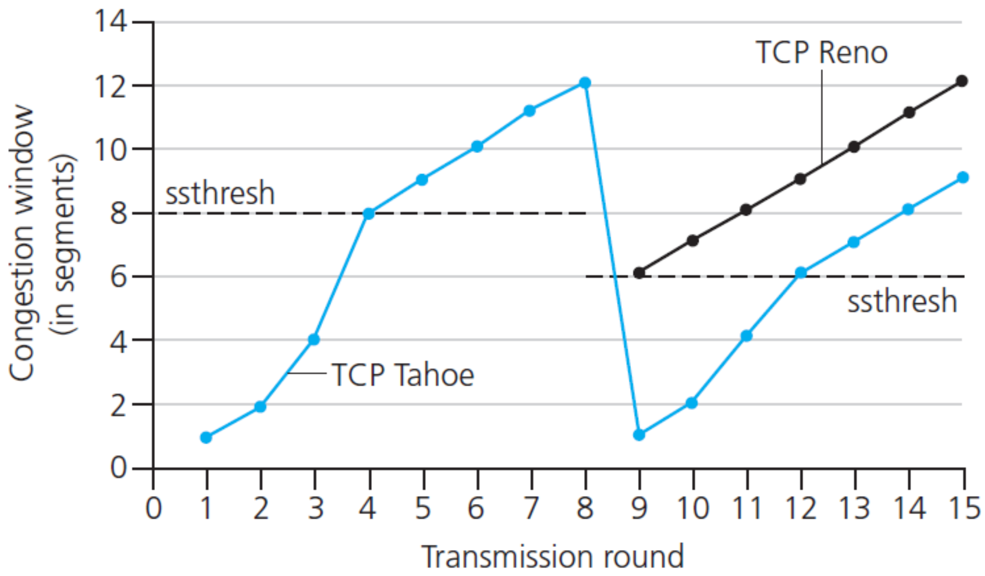
TCP: detecting, reacting to loss

- 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?

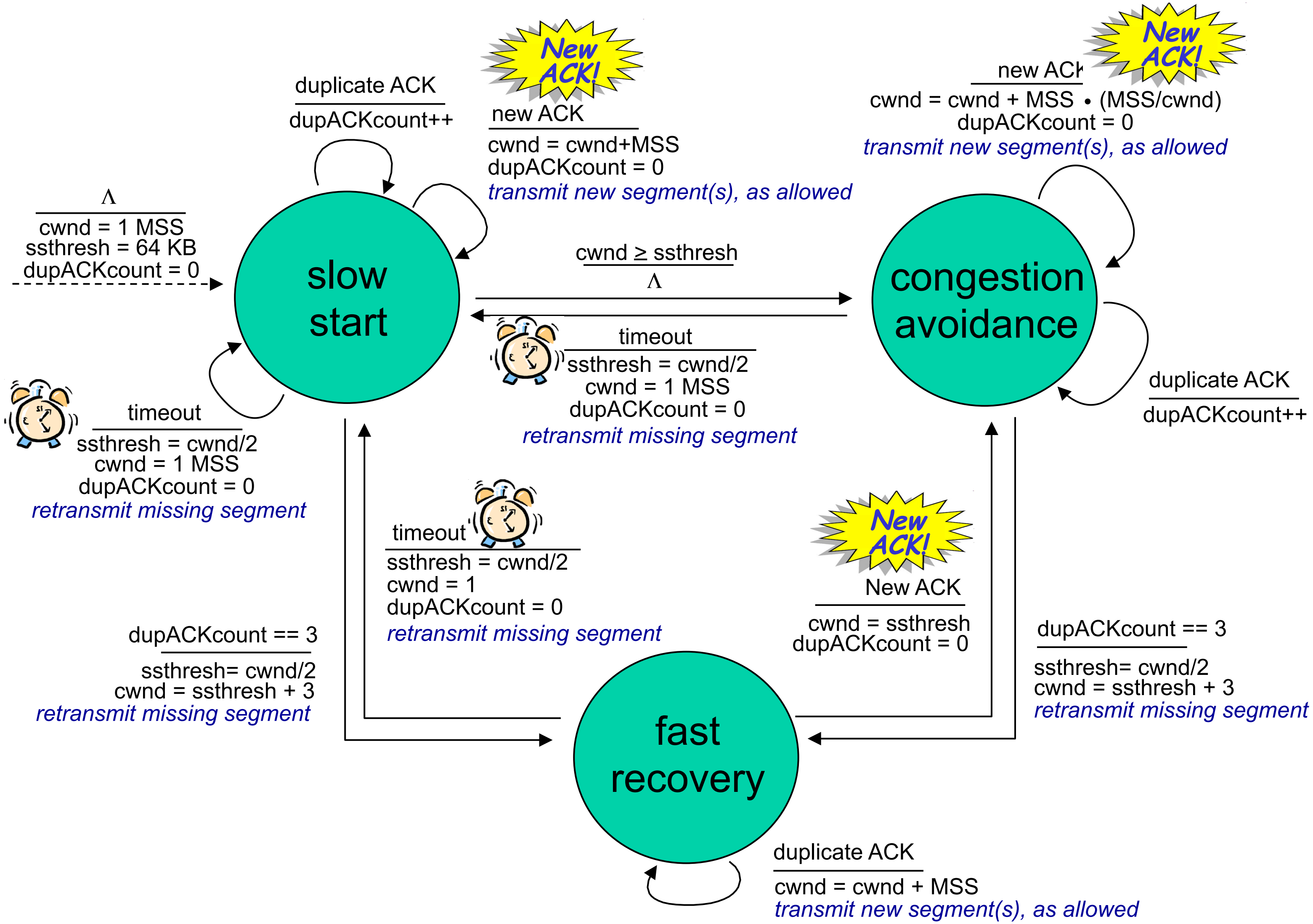
A: when `cwnd` gets to 1/2 of its value before timeout.



Implementation:

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

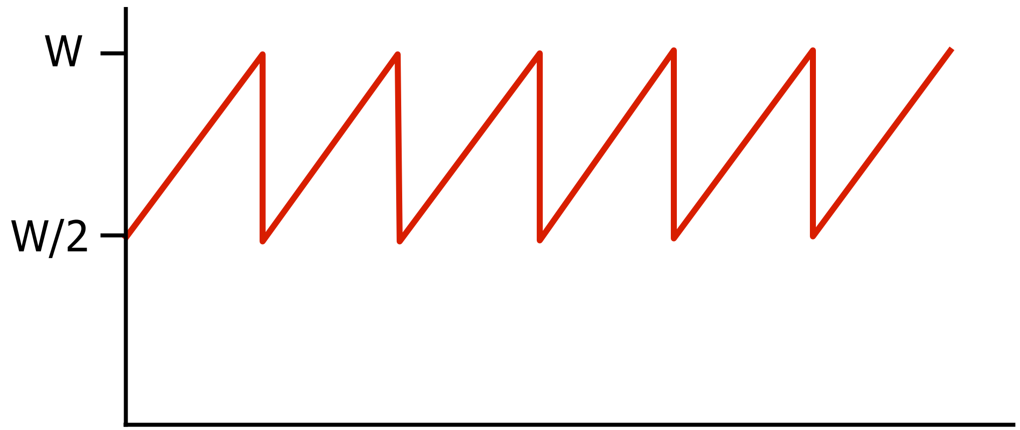
Summary: TCP Congestion Control



TCP throughput

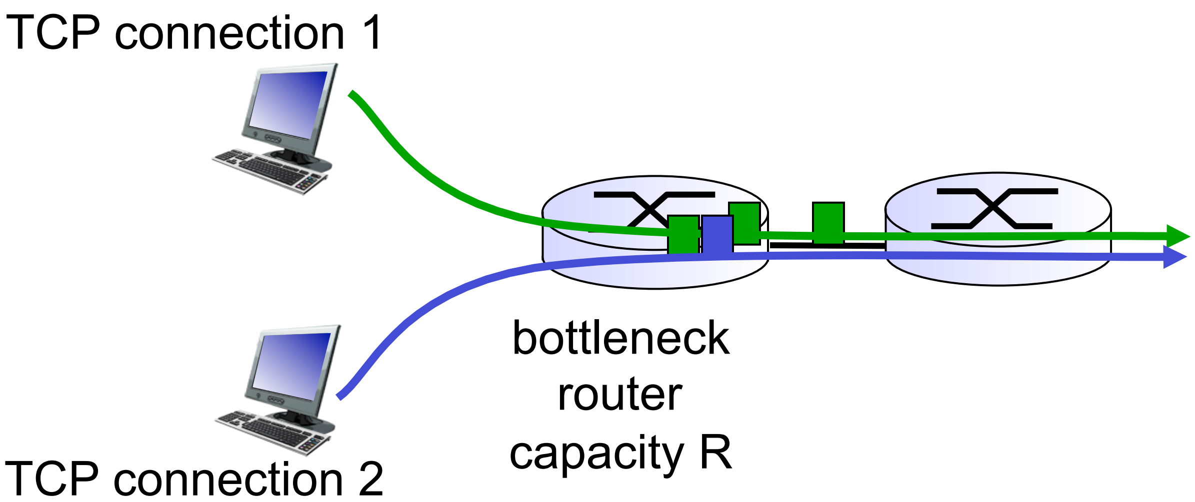
- avg. TCP throughput as function of window size, RTT?
  - ignore slow start, assume always data to send
- W: window size (measured in bytes) where loss occurs
  - avg. window size (# in-flight bytes) is ¾ W
  - avg. thruput is 3/4W per RTT

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



TCP Fairness

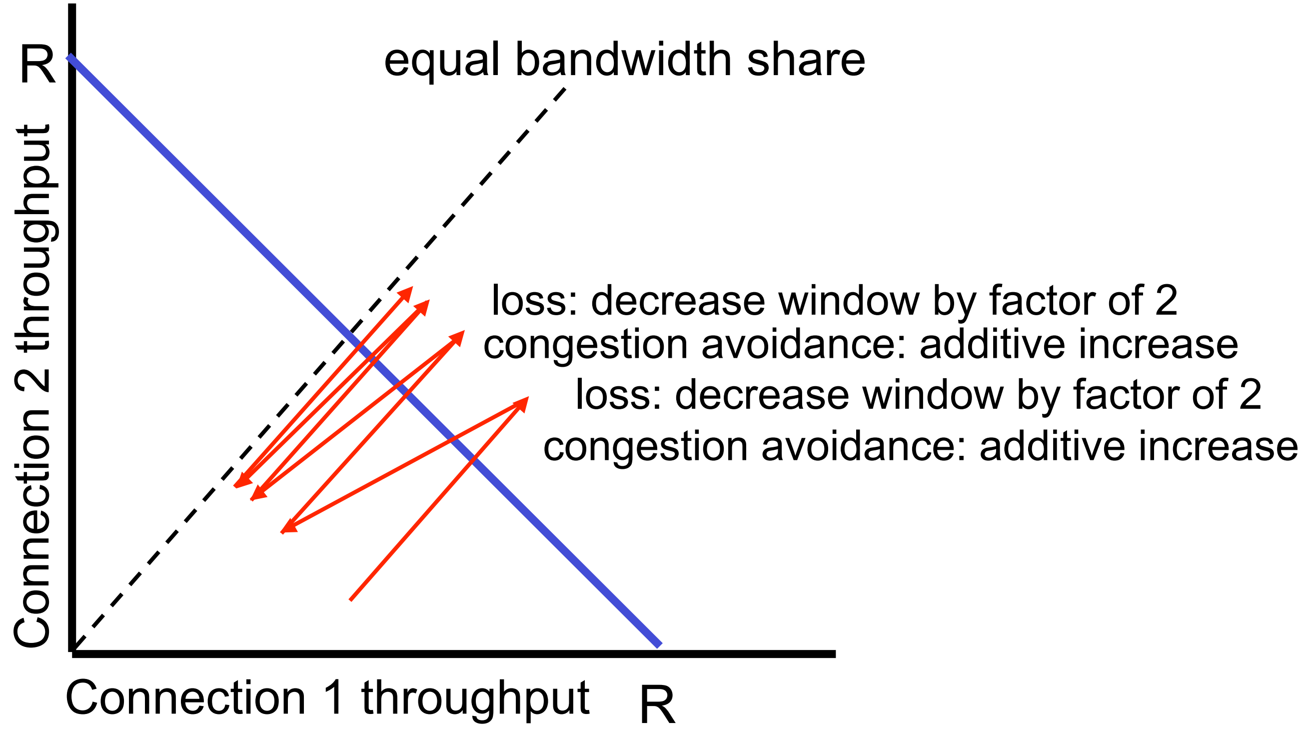
fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K



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

Chapter 3: summary

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

<u>next:</u>

- leaving the network "edge" (application , transport layers)
- into the network "core"