DIPARTIMENTO
DI INFORMATICA
Via Salaria 113 - 00198, Roma

SAPIENZA
UNIVERSITÀ DI ROMA

Chapter 3
Transport Layer

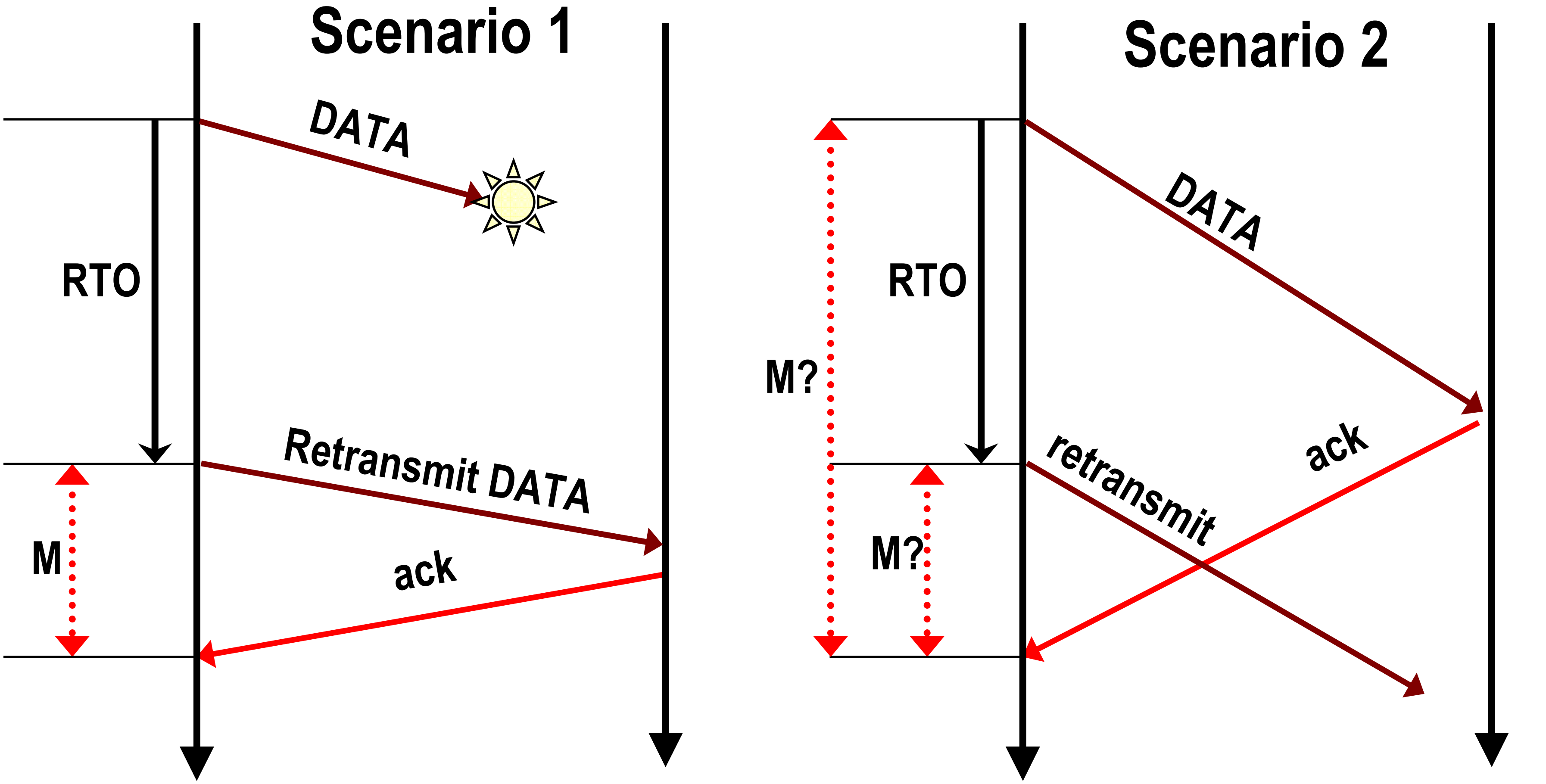
Reti di Elaboratori
Corso di Laurea in Informatica
Università degli Studi di Roma "La Sapienza"
Canale A-L
Prof.ssa Chiara Petrioli

Parte di queste slide sono state prese dal materiale associato al libro *Computer Networking: A Top Down Approach* , 5th edition.

Thanks also to Antonio Capone, Politecnico di Milano, Giuseppe Bianchi and Francesco LoPresti, Un. di Roma Tor Vergata

Guessing right?

Karn's problem



How can we distinguish among an ACK to the original segment and to a duplicate?

Solution to Karn's problem

- **Very simple: DO NOT update RTT when a segment has been retransmitted because of RTO expiration!**
- **Instead, use Exponential backoff**
  - *double RTO for every subsequent expiration of same segment*
    - When at 64 secs, stay
    - persist up to 9 minutes, then reset

TCP reliable data transfer (more in detail)

- 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 (simplified)

Purche' non si ecceda la finestra

```
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 */
```

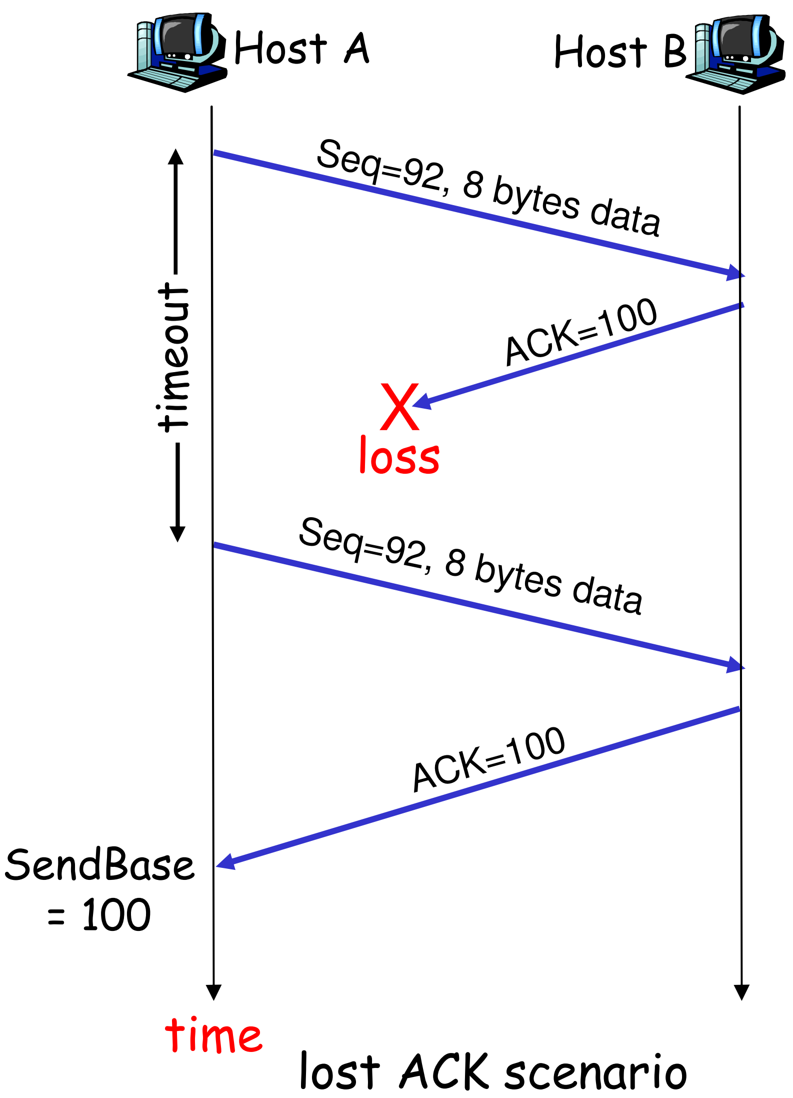
Comment:

- SendBase-1: last cumulatively ack'ed byte

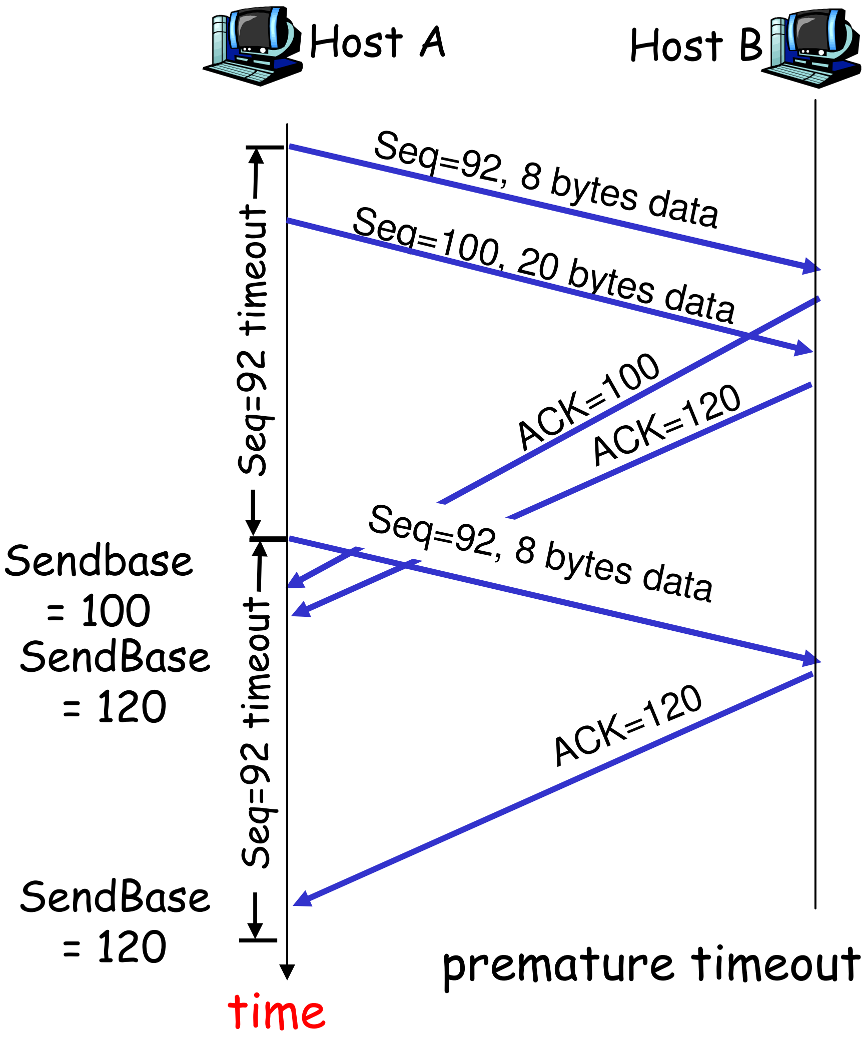
Example:

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

TCP: retransmission scenarios

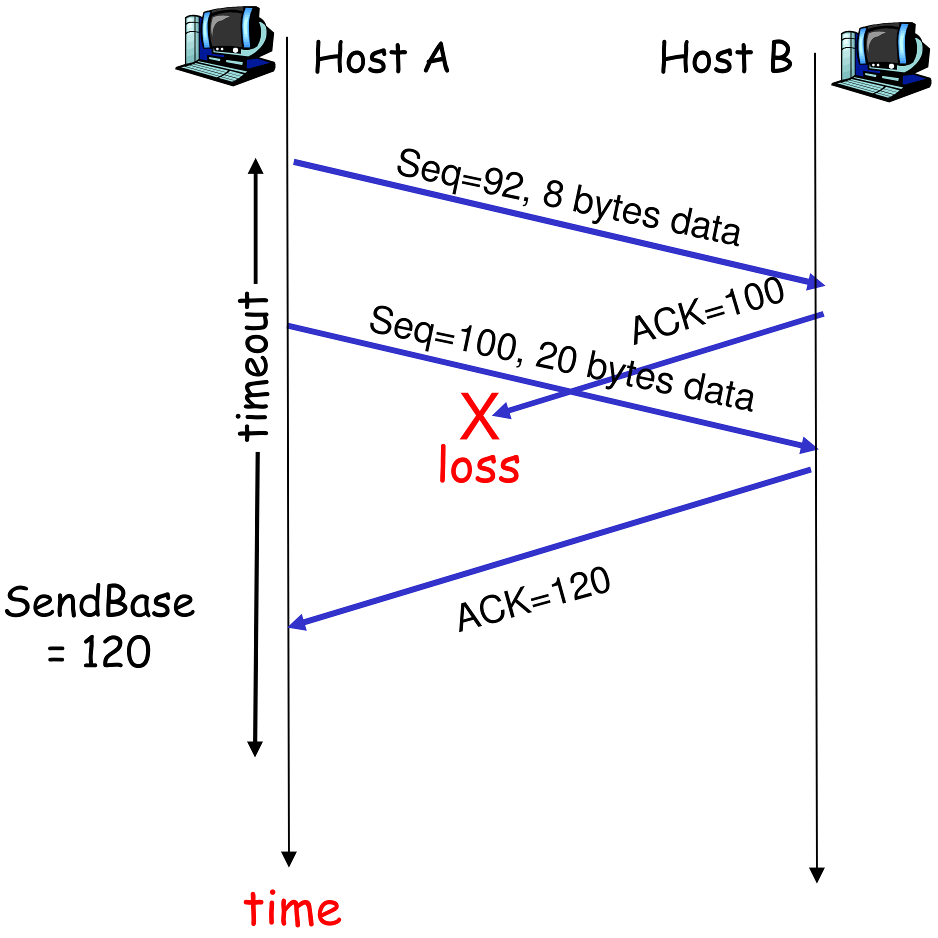


lost ACK scenario



premature timeout

TCP retransmission scenarios (more)



Cumulative ACK scenario

TCP ACK generation [RFC 1122, RFC 2581]

Main motivation: performance

Favor piggybacking

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**

Can advance source window

Duplicate ACK important feedback—more later

So what is the TCP solution

- Go-Back-N??
- Selective Repeat?
- A: An Hybrid solution.
  - Possibility of buffering correctly received packets AND selective retransmission of packets, BUT NOT pure Selective Repeat, cumulative ACK, buffering not required (free implementation choice)
  - Shares some aspects with GBN BUT
    - A single timer for the oldest unacked packet;
    - when the timer experises ONLY that packet is retransmitted

TCP: a reliable transport

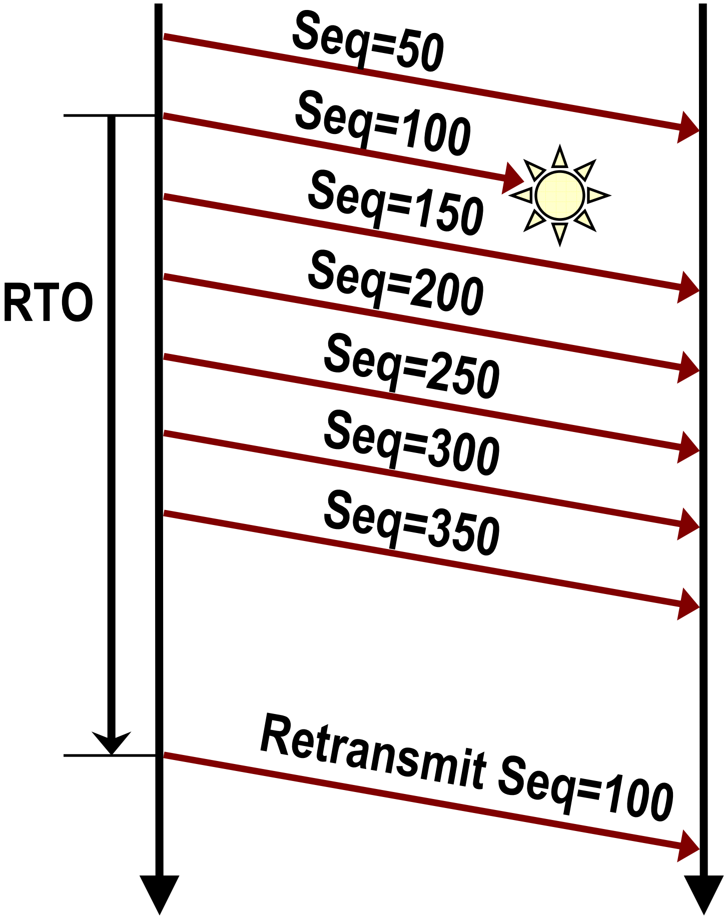
- TCP is a reliable protocol
  - all data sent are guaranteed to be received
  - ***very important feature, as IP is unreliable network layer***
- employs positive acknowledgement
  - cumulative ack
  - selective ack may be activated when both peers implement it (use option) → TCP SACKS
- does not employ negative ack
  - error discovery via timeout (retransmission timer)
  - …But “implicit NACK” is available (more later: fast retransmit)

Need for implicit NACKs

- ➔ TCP does not support negative ACKs
- ➔ This can be a serious drawback
  - ⇨ Especially in the case of single packet loss
- ➔ Necessary RTO expiration to start retransmit lost packet
  - ⇨ As well as following ones!!

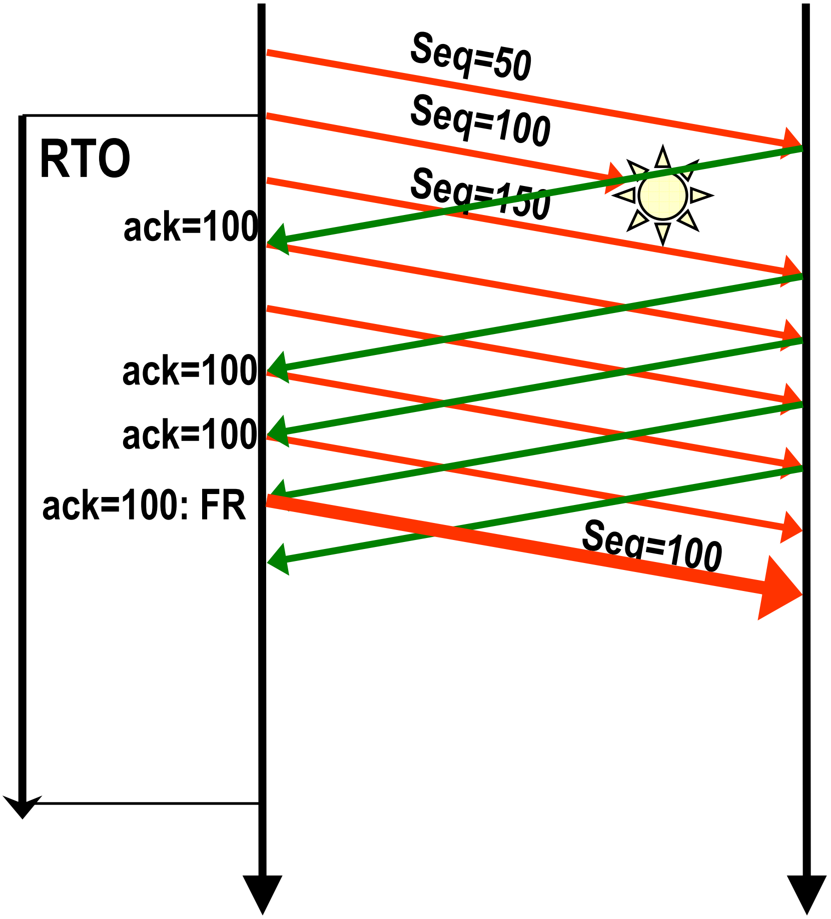
  May take too much time before retransmitting!!!

- ➔ ISSUE: is there a way to have NACKs in an implicit manner????



The Fast Retransmit Algorithm

- Idea: use duplicate ACKs!
  - Receiver responds with an ACK every time it receives an out-of-order segment
  - ACK value = last correctly received segment
- FAST RETRANSMIT algorithm:
  - if 3 duplicate acks are received for the same segment, assume that the next segment has been lost. Retransmit it right away.
  - Helps if single packet lost. Not very effective with multiple losses



Fast retransmit algorithm:

```
event: ACK received, with ACK field value of y
          if (y > SendBase) {
              SendBase = y
              if (there are currently not-yet-acknowledged segments)
                  start timer
            }
          else {
               increment count of dup ACKs received for y
               if (count of dup ACKs received for y = 3) {
                   resend segment with sequence number y
                }
```

a duplicate ACK for already ACKed segment

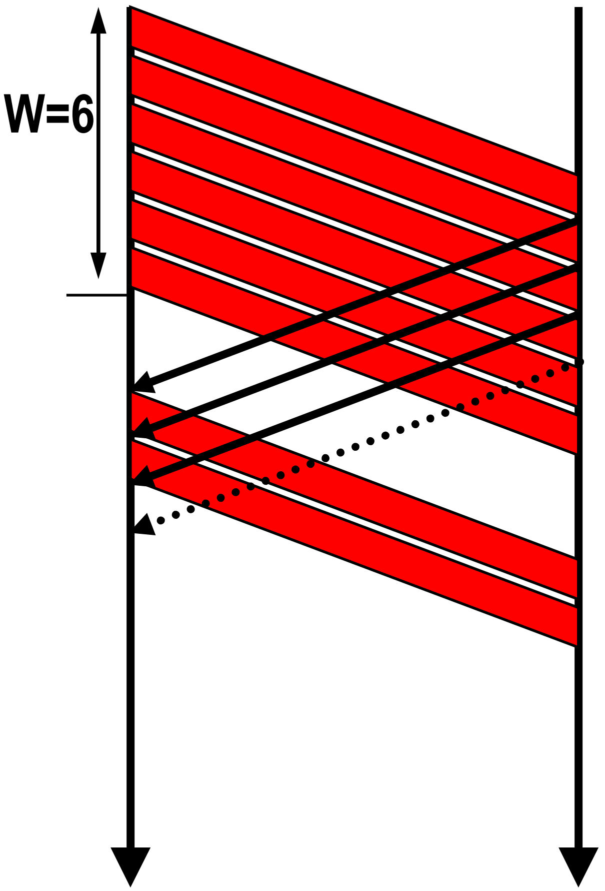
fast retransmit

TCP mechanisms for:

- flow control
- congestion control

Graphical examples (applet java) of several algorithms at:
http://www.ce.chalmers.se/~fcela/tcp-tour.html

TCP pipelining

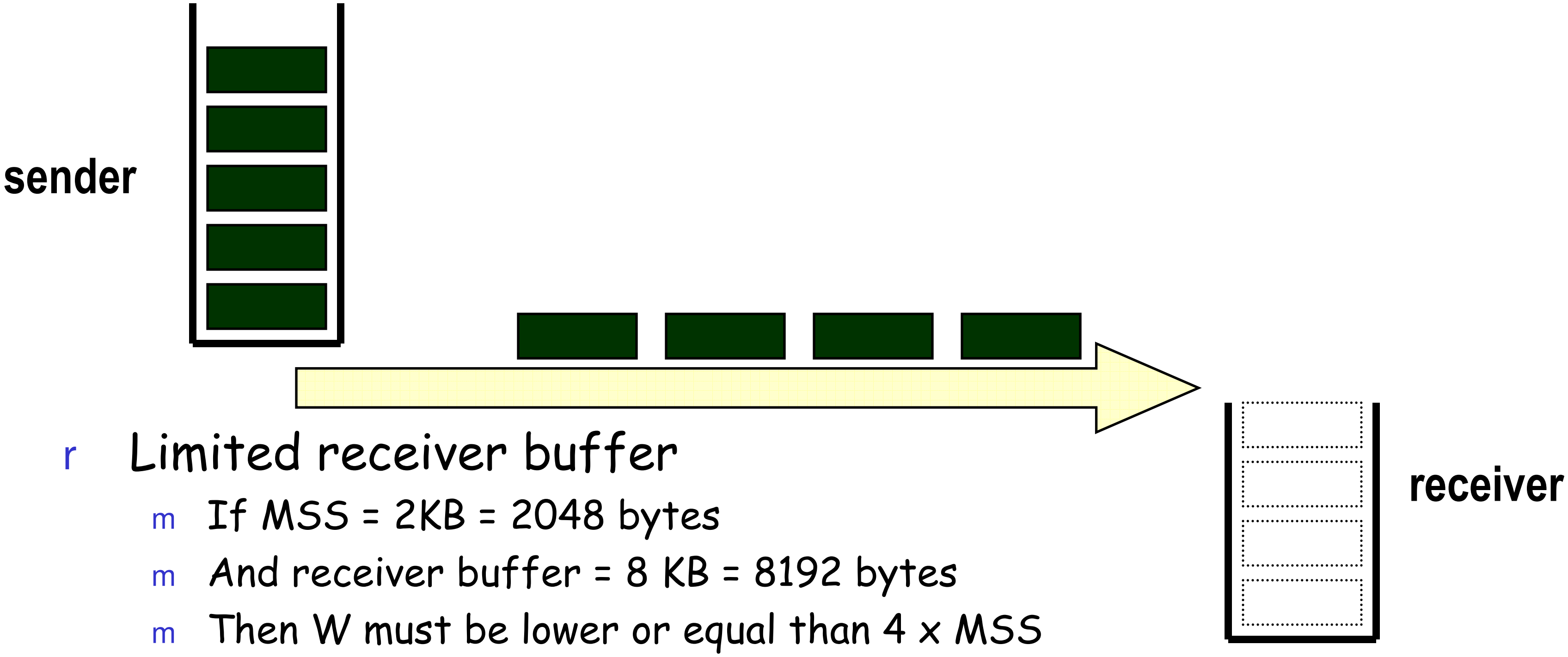


- More than 1 segment "flying" in the network
- Transfer efficiency increases with W

$$thr = \min\left(C, \frac{W \cdot MSS}{RTT + MSS / C}\right)$$

- So, why an upper limit on W?
  - Esempio: flow control

Why flow control?

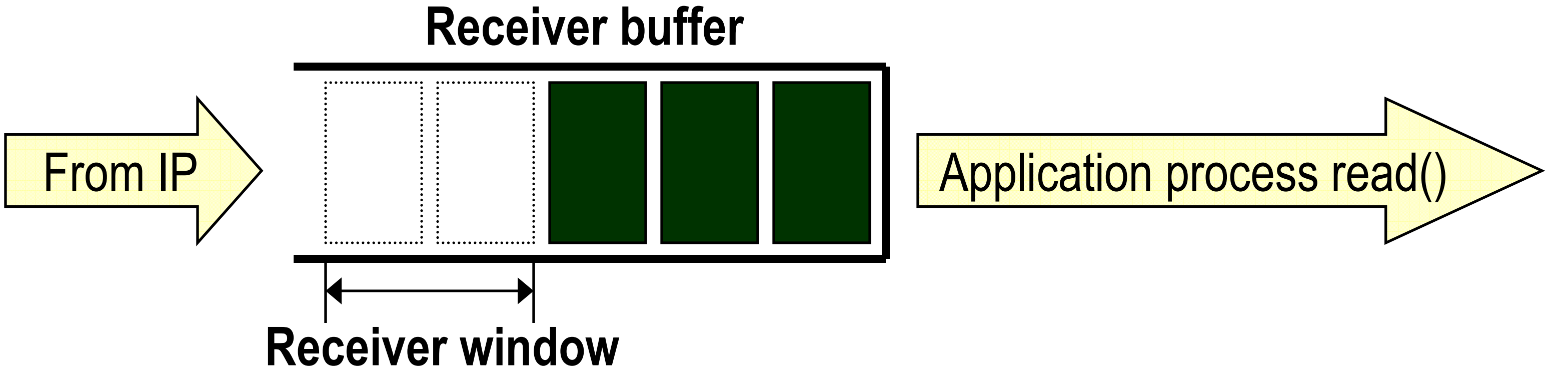


- Limited receiver buffer
  - If MSS = 2KB = 2048 bytes
  - And receiver buffer = 8 KB = 8192 bytes
  - Then W must be lower or equal than 4 x MSS

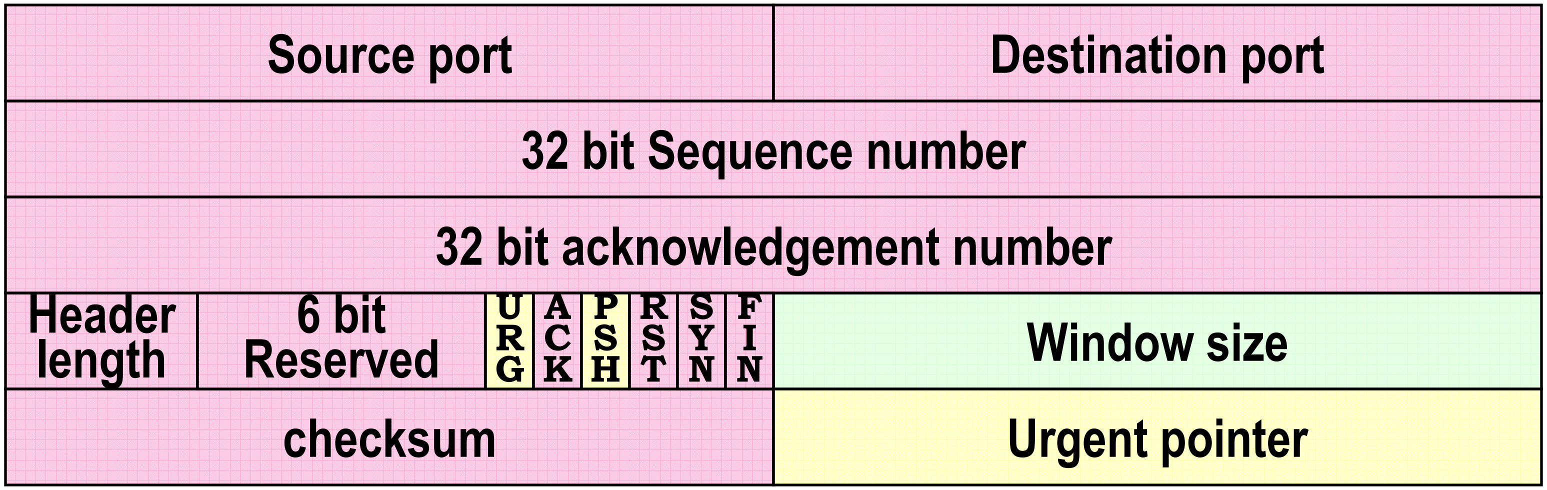
- A possible implementation:
  - During connection setup, exchange W value.
  - *DOES NOT WORK. WHY?*

Window-based flow control

- ➔ **receiver buffer capacity varies with time!**
  - ⇨ Upon application process read() [asynchronous, not depending on OS, not predictable]



- ➔ MSS = 2KB = 2048 bytes
- ➔ Receiver Buffer capacity = 10 KB = 10240 bytes
- ➔ TCP data stored in buffer: 3 segments
- ➔ Receiver window = Spare room: 10-6 = 4KB = 4096 bytes
  - ⇨ Then, at this time, W must be lower or equal than 2 x MSS



Source port								Destination port
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	URG	ACK	PSH	RST	SYN	FIN	Window size
checksum								Urgent pointer

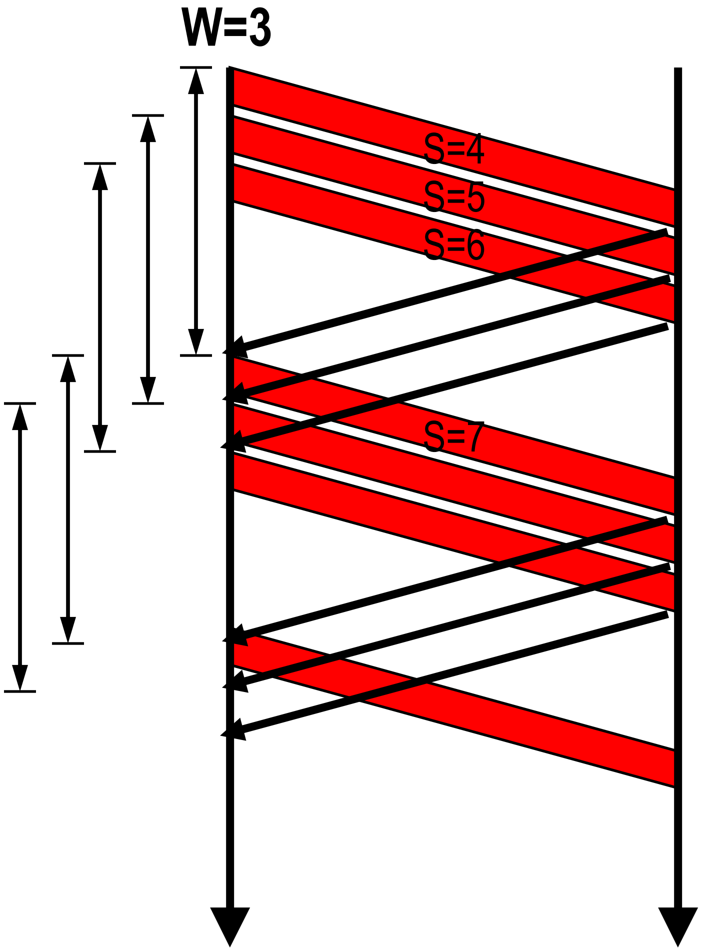
- Window size field: used to advertise receiver's remaining storage capabilities
  - 16 bit field, on every packet
  - Measure unit: bytes, from 0 (included) to 65535
  - Sender rule:

    `LastByteSent – LastByteAcked <= RcvWindow.`
  - W=2048 means:
    - I can accept other 2048 bytes since ack, i.e. bytes [ack, ack+W-1]
    - also means: sender may have 2048 bytes outstanding (in multiple segments)

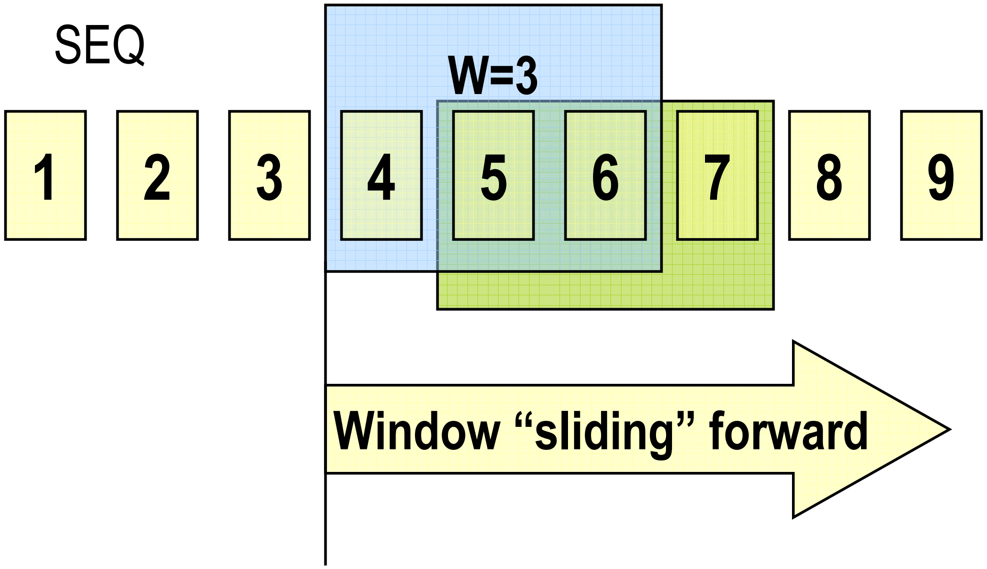
What is flow control needed for?

- Window flow control guarantees receiver buffer to be able to accept outstanding segments.
- When receiver buffer full, just send back win=0
- in essence, flow control guarantees that transmission bit rate never exceed receiver rate

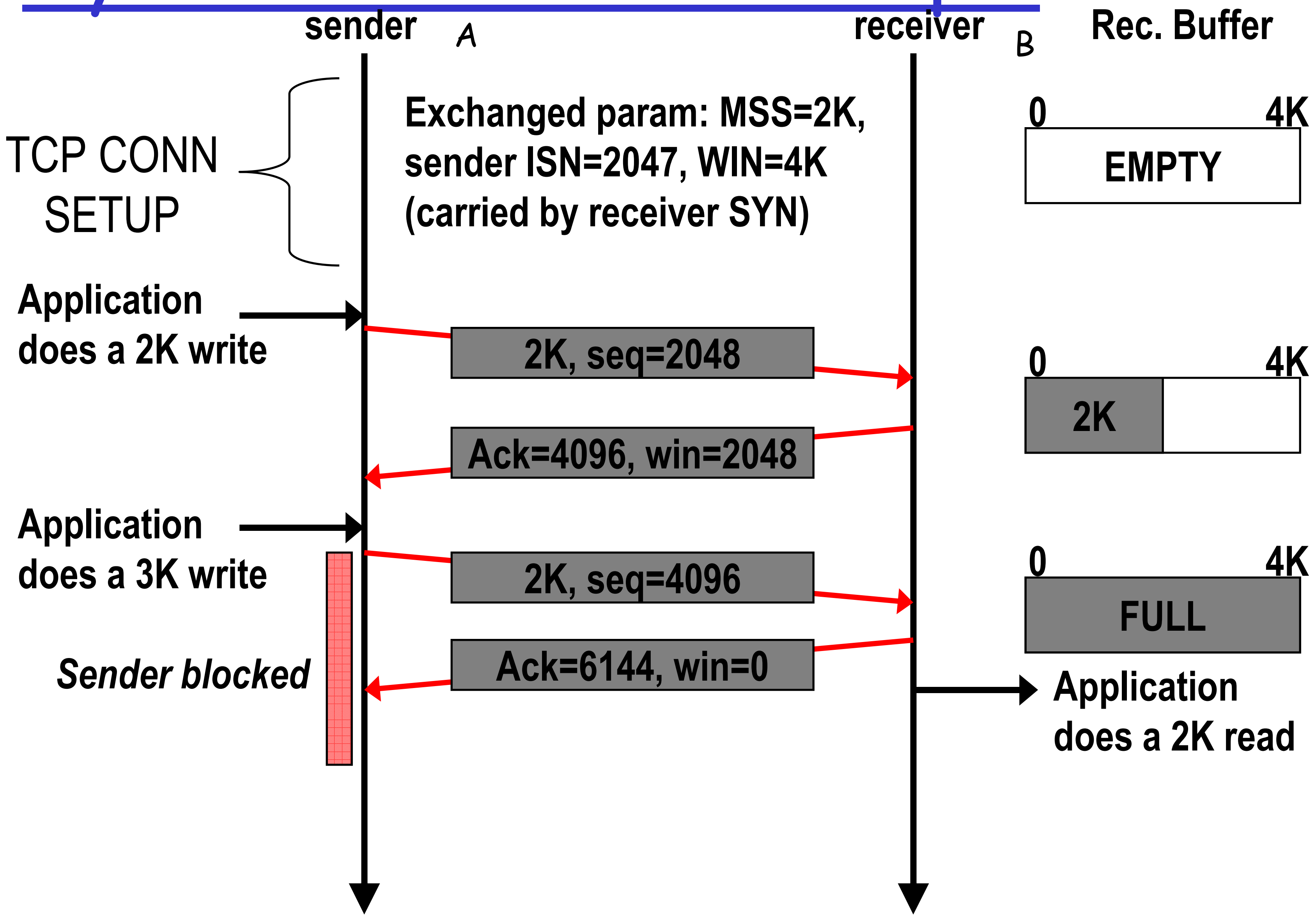
Sliding window



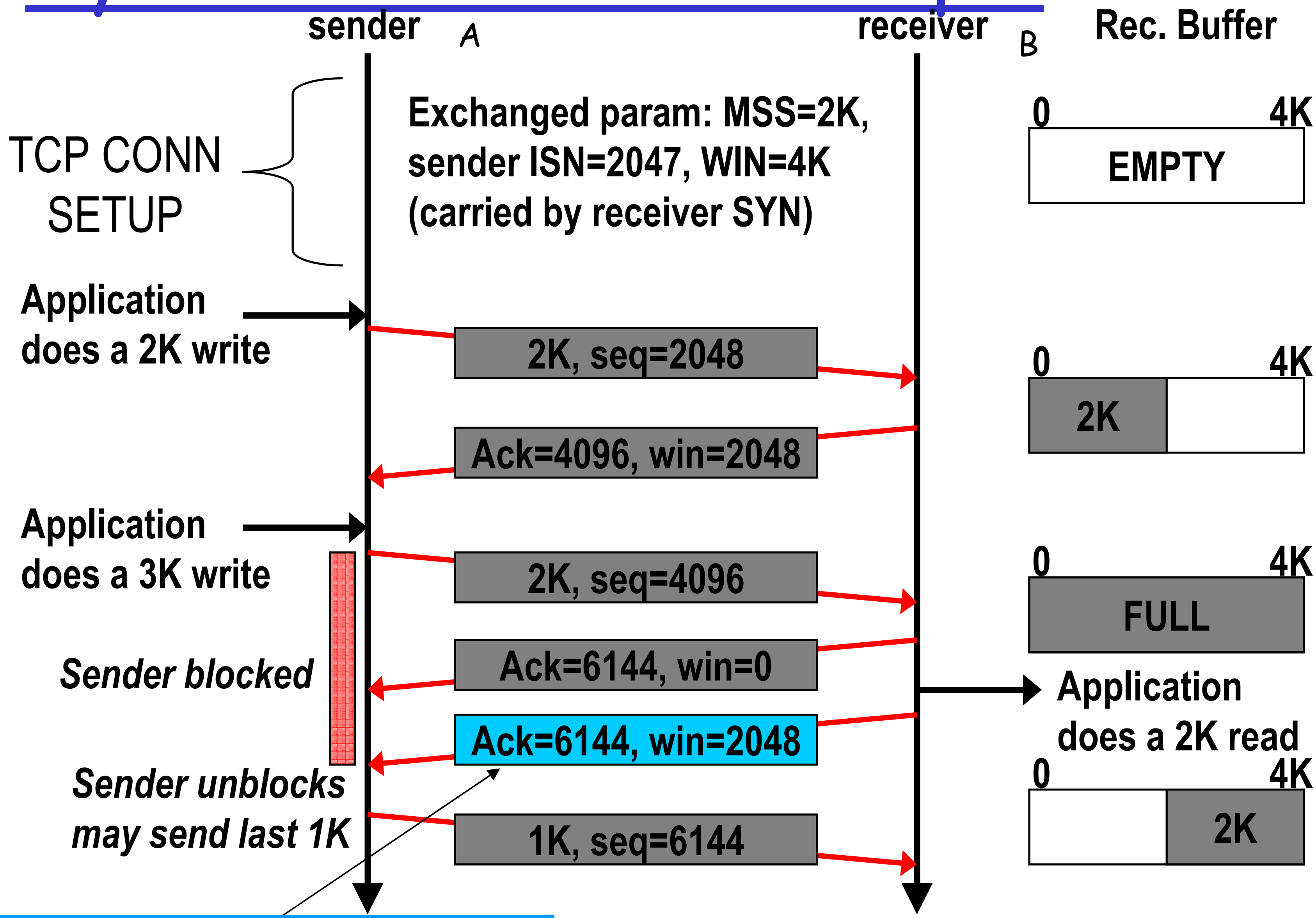
Dynamic window based reduces to pure sliding window when receiver app is very fast in reading data…



Dynamic window - example


sender
A
receiver
B
Rec. Buffer
TCP CONN SETUP
Exchanged param: MSS=2K, sender ISN=2047, WIN=4K (carried by receiver SYN)
0
4K
EMPTY
Application does a 2K write
2K, seq=2048
0
4K
2K
Ack=4096, win=2048
Application does a 3K write
2K, seq=4096
0
4K
FULL
Sender blocked
Ack=6144, win=0
Application does a 2K read

Dynamic window - example



sender A

receiver B

Rec. Buffer

TCP CONN SETUP

Exchanged param: MSS=2K, sender ISN=2047, WIN=4K (carried by receiver SYN)

0 4K EMPTY

Application does a 2K write

2K, seq=2048

0 4K 2K

Ack=4096, win=2048

Application does a 3K write

2K, seq=4096

0 4K FULL

Sender blocked

Ack=6144, win=0

Application does a 2K read

Ack=6144, win=2048

0 4K 2K

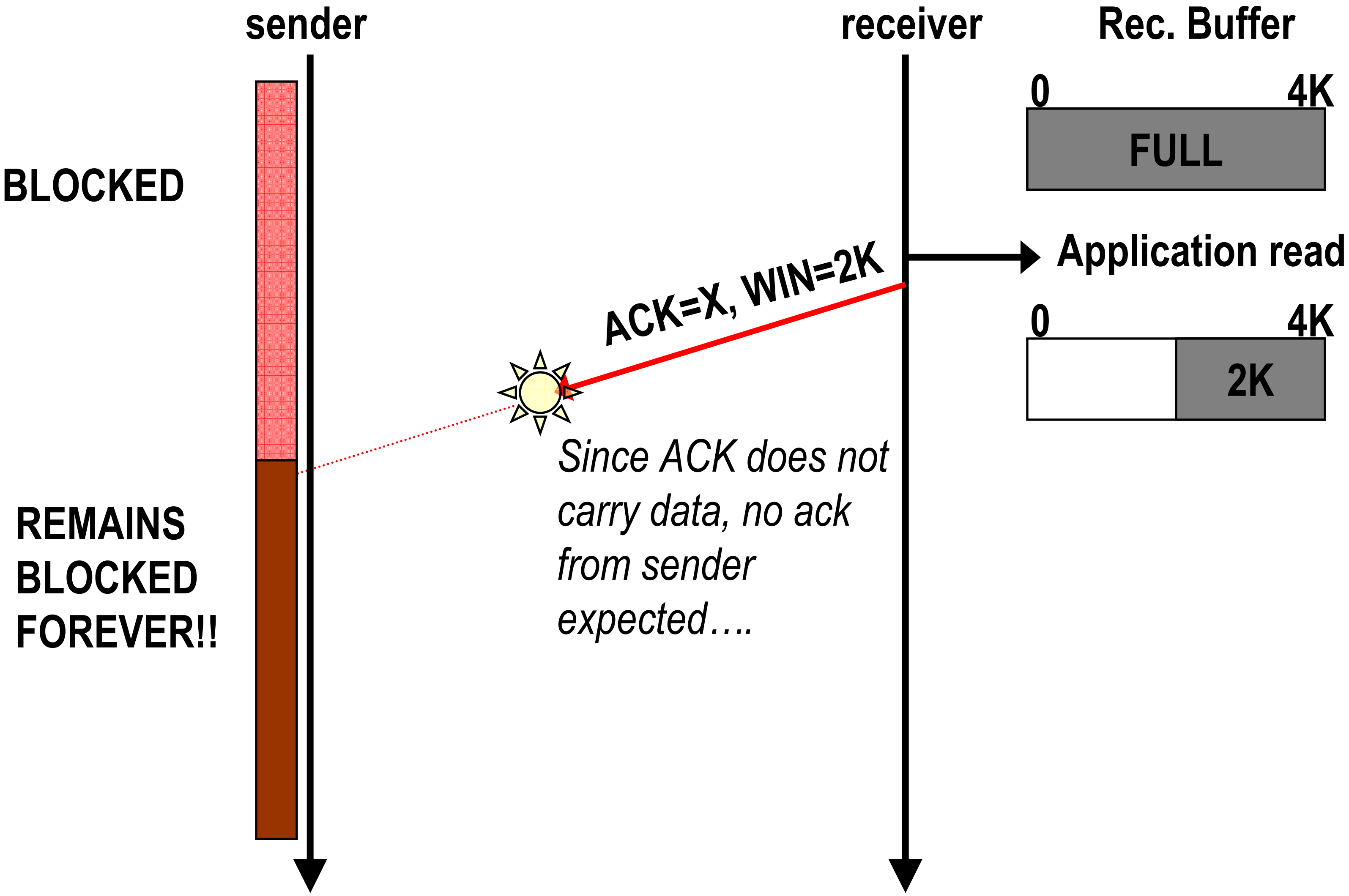
Sender unblocks may send last 1K

1K, seq=6144

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



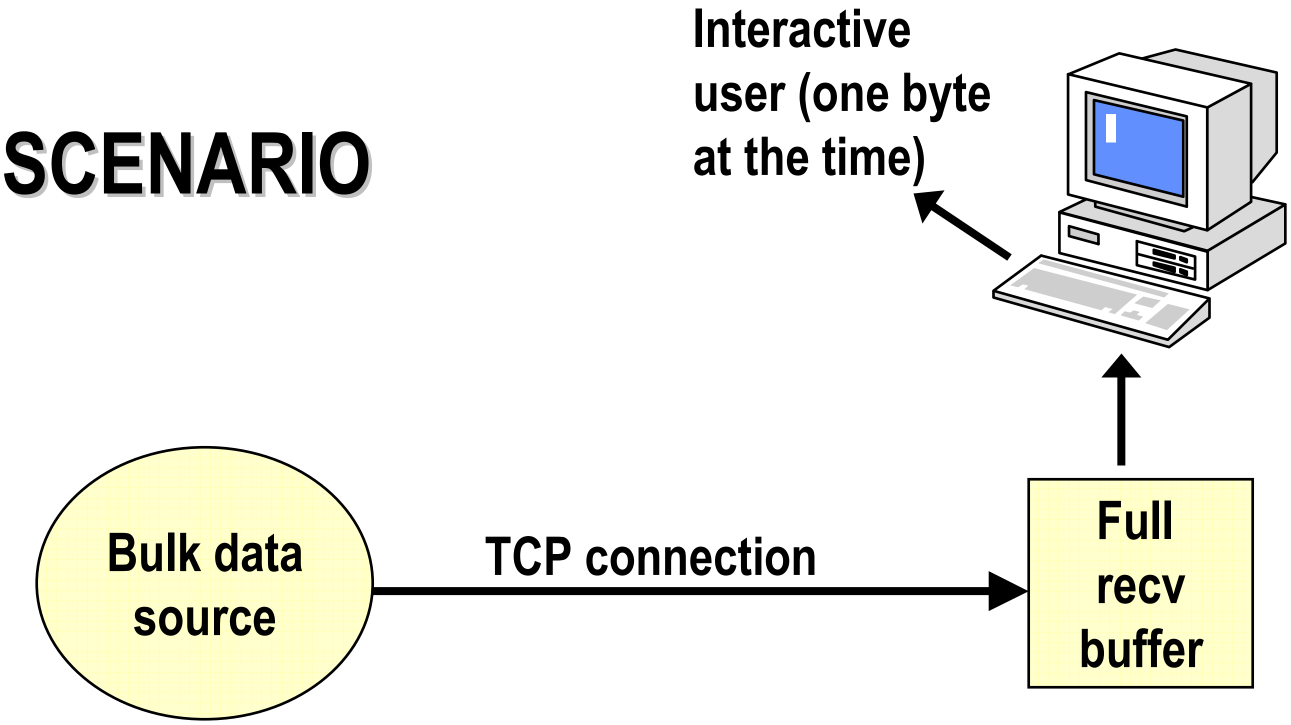
Since ACK does not carry data, no ack from sender expected….

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

SCENARIO



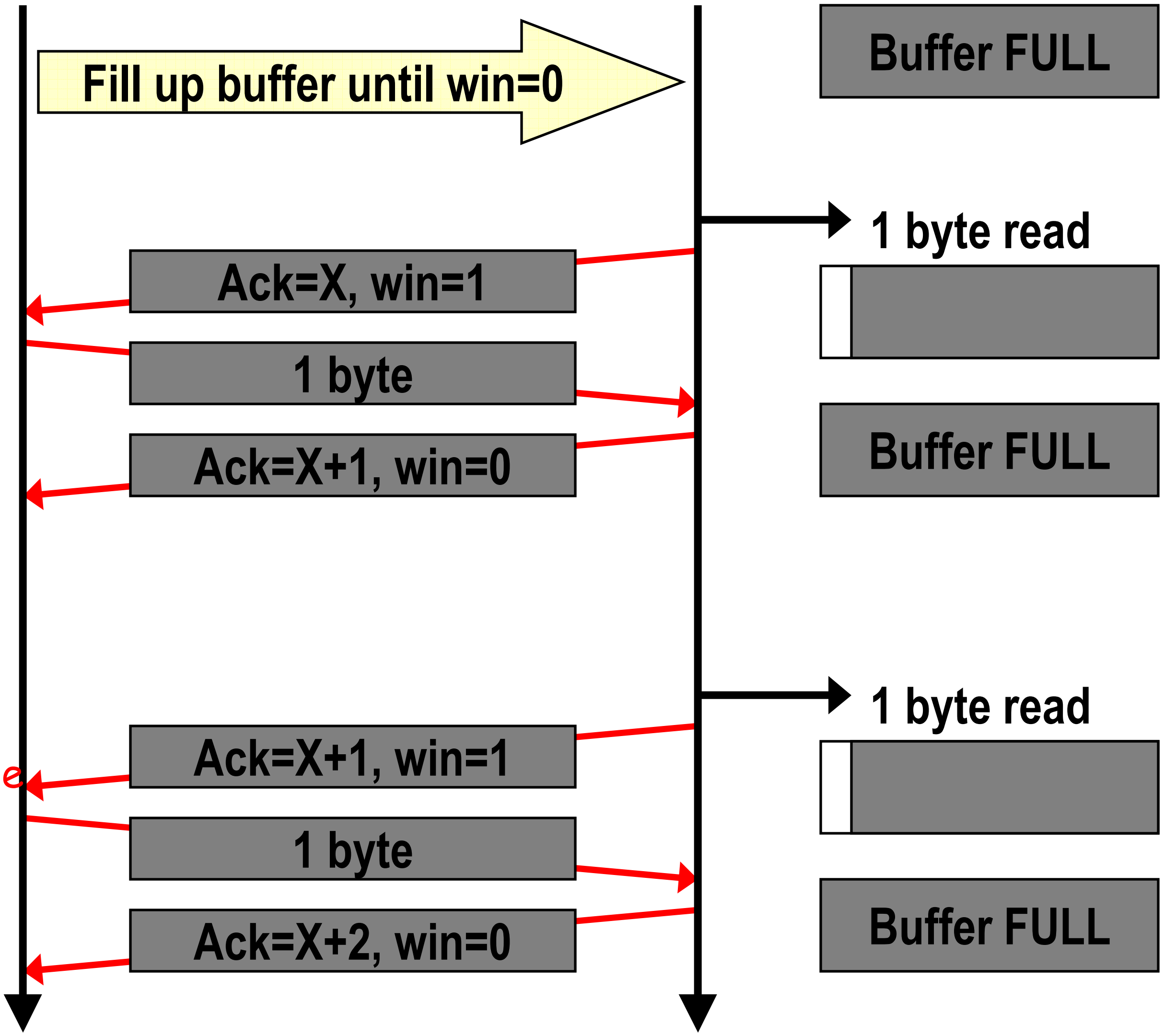
Interactive user (one byte at the time)

Bulk data source

TCP connection

Full recv buffer

The silly window syndrome



Fill up buffer until win=0

Buffer FULL

1 byte read

Ack=X, win=1

1 byte

Ack=X+1, win=0

Buffer FULL

1 byte read

Ack=X+1, win=1

1 byte

Ack=X+2, win=0

Buffer FULL

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

Forever!

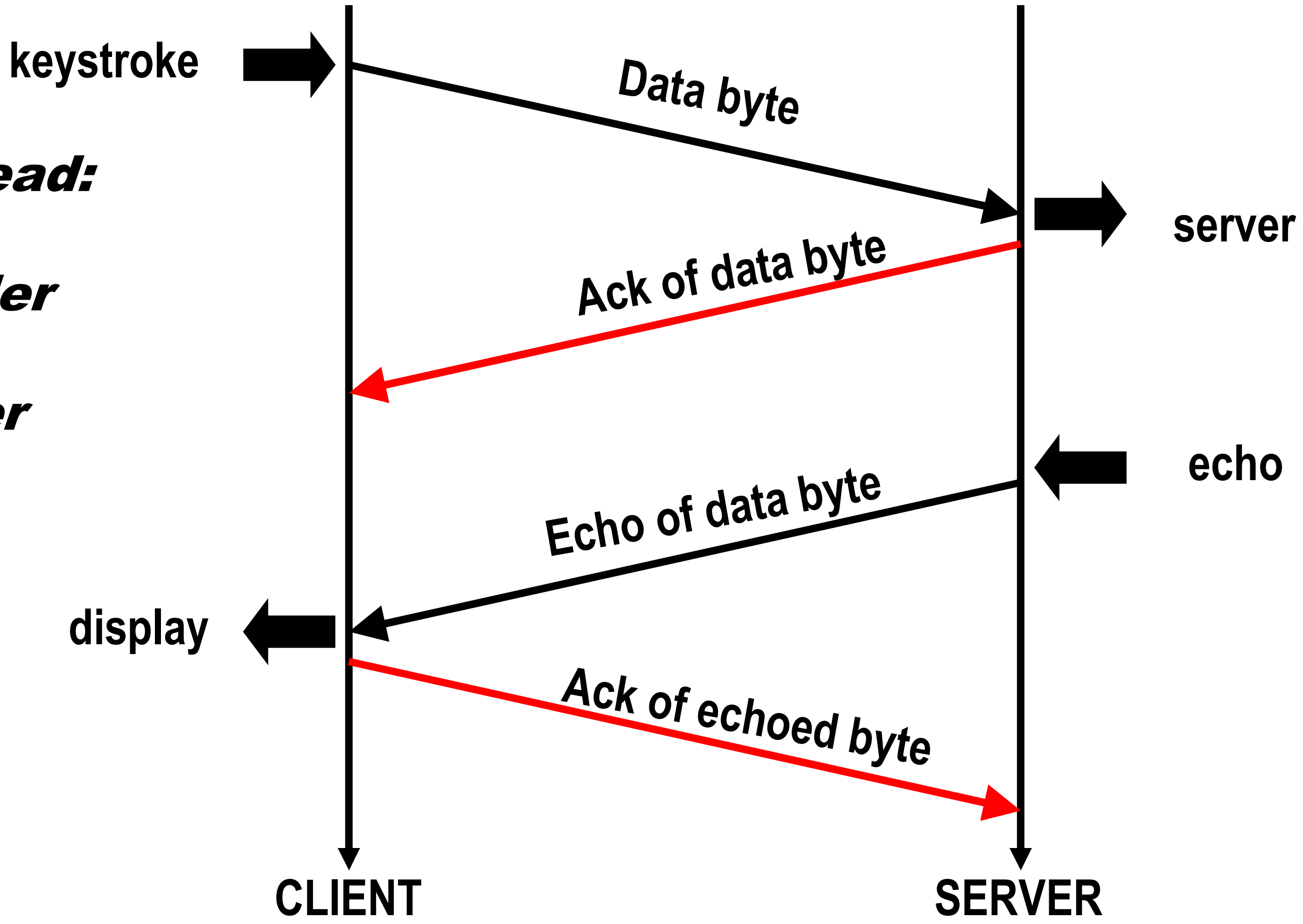
Anche se il ricevitore e' veloce
A passare i dati al livello
applicativo inviare segmenti
piccoli in un bulk di dati
ha questo effetto

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

Interactive applications



Header overhead:

20 TCP header
+
20 IP header
+
1 data

Interactive apps: create some tricky situations….

Nagle's algorithm (RFC 896, 1984)

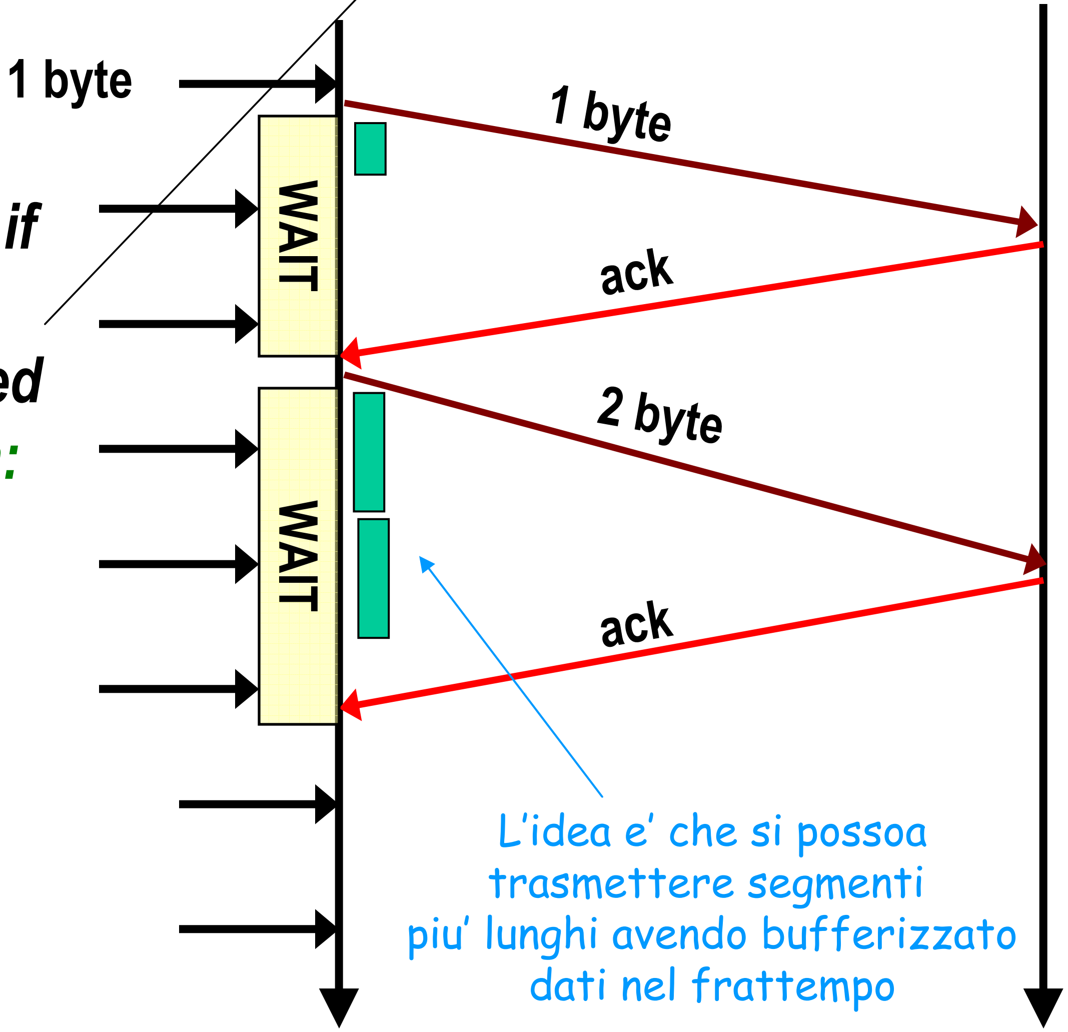
UNLESS MSS data (or at least half the window size bytes) are ready to be transmitted

NAGLE RULE: inhibit sending new segments if any previously transmitted data unacked

self-clocking algorithm:

on LANs, plenty of tynigrams

on slow WANs, data aggregation



L'idea e' che si possoa trasmettere segmenti piu' lunghi avendo bufferizzato dati nel frattempo

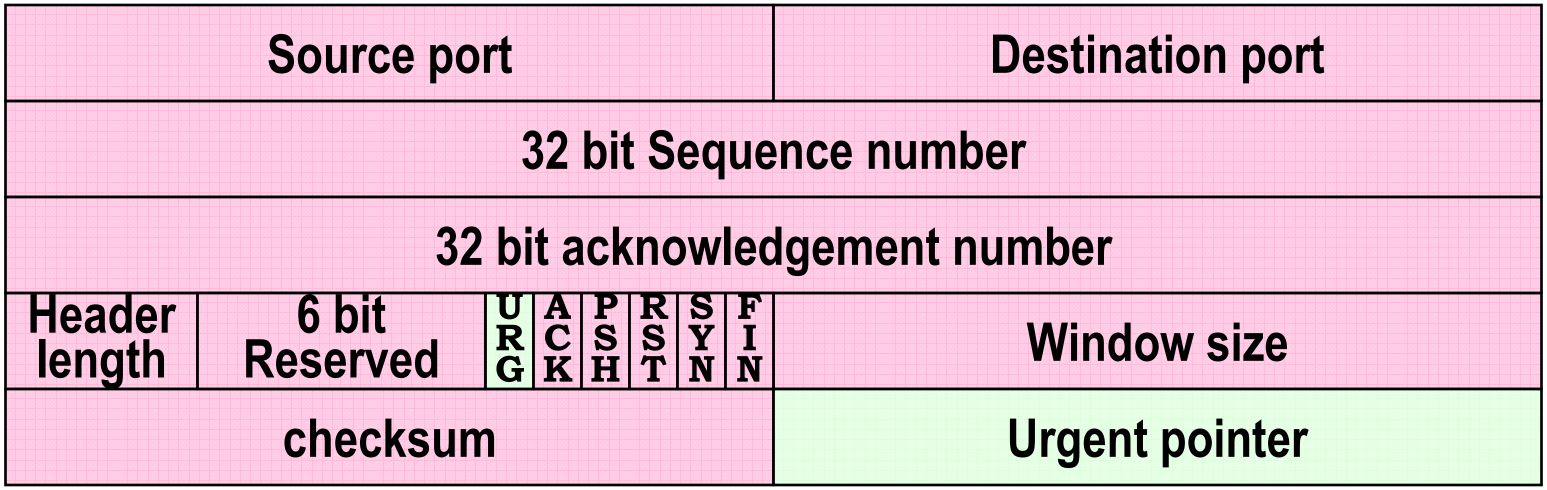
PUSH flag

<table>
<tr><td colspan="8">Source port</td><td>Destination port</td></tr>
<tr><td colspan="9">32 bit Sequence number</td></tr>
<tr><td colspan="9">32 bit acknowledgement number</td></tr>
<tr><td>Header length</td><td>6 bit Reserved</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Window size</td></tr>
<tr><td colspan="8">checksum</td><td>Urgent pointer</td></tr>
</table>

r Used to notify

- m TCP sender to send data
  - but for this an header flag NOT needed! Sufficient a “push” type indication in the TCP sender API
- m TCP receiver to pass received data to the application

Urgent data



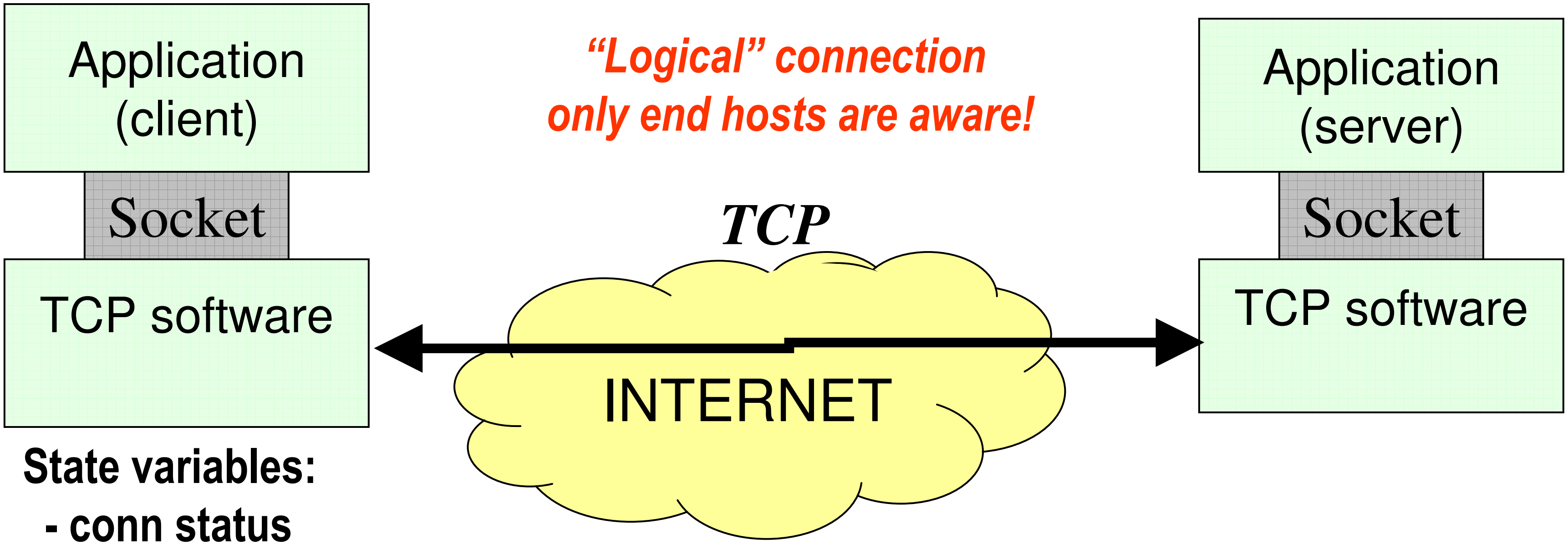
Source port								Destination port
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	URG	ACK	PSH	RST	SYN	FIN	Window size
checksum								Urgent pointer

- URG on: notifies rx that “urgent” data placed in segment.
- When URG on, *urgent pointer* contains position of *the last octet* of urgent data
  - *indeed it contains the positive offset from the segment sequence number*
  - *and the position of the first octet of urgent data? No way to specify it!*
  - *Changed wrt RFC 793*
- receiver is expected to pass all data up to urgent ptr to app
  - interpretation of urgent data is left to the app
- typical usage: ctrlC (interrupt) in rlogin & telnet; abort in FTP
- urgent data is a *second* exception to blocked sender

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 connection



"Logical" connection
only end hosts are aware!

State variables:
- **conn status**
- **MSS**
- **windows**
- **...**

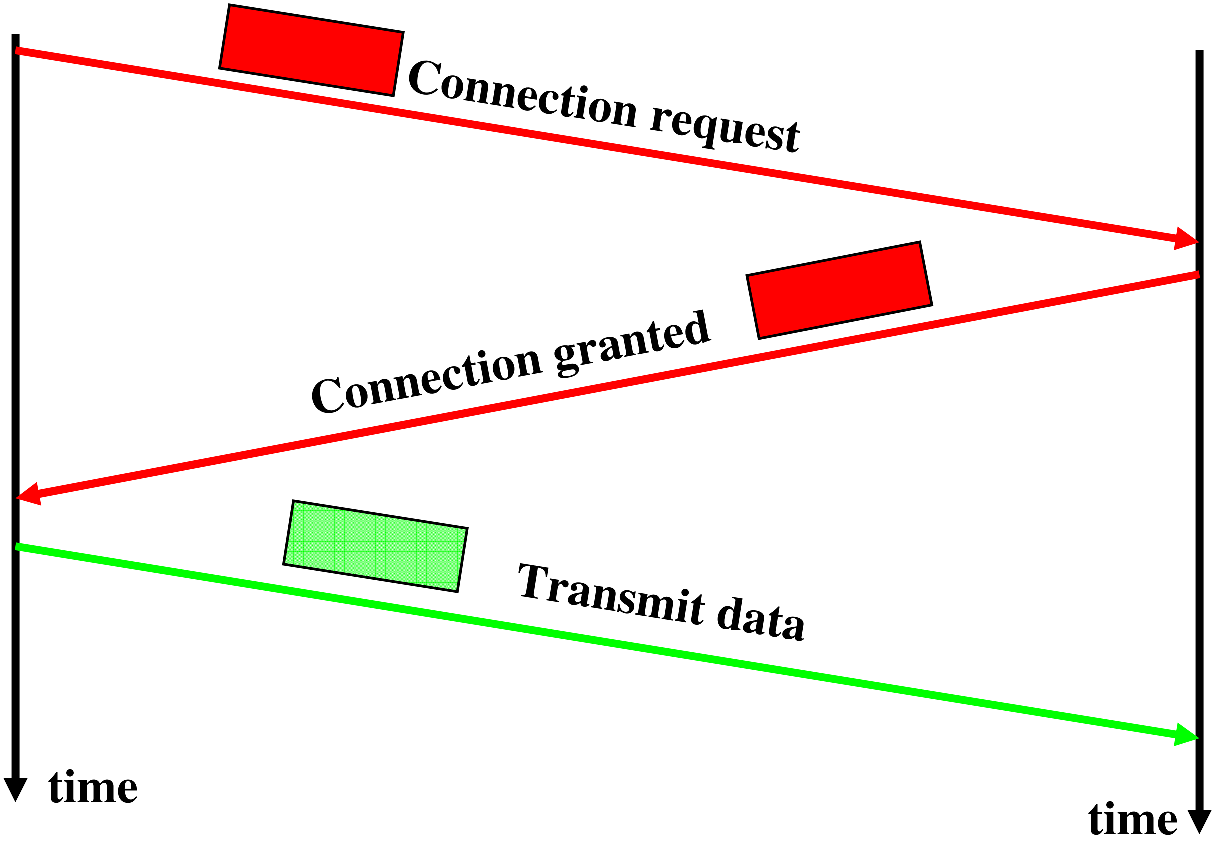
buffer space
normally 4 to 16 Kbytes
64+ Kbytes possible

Connection described by client&server <u>status</u>
Connection SET-UP duty:
1) initializes state variables
2) reserves buffer space

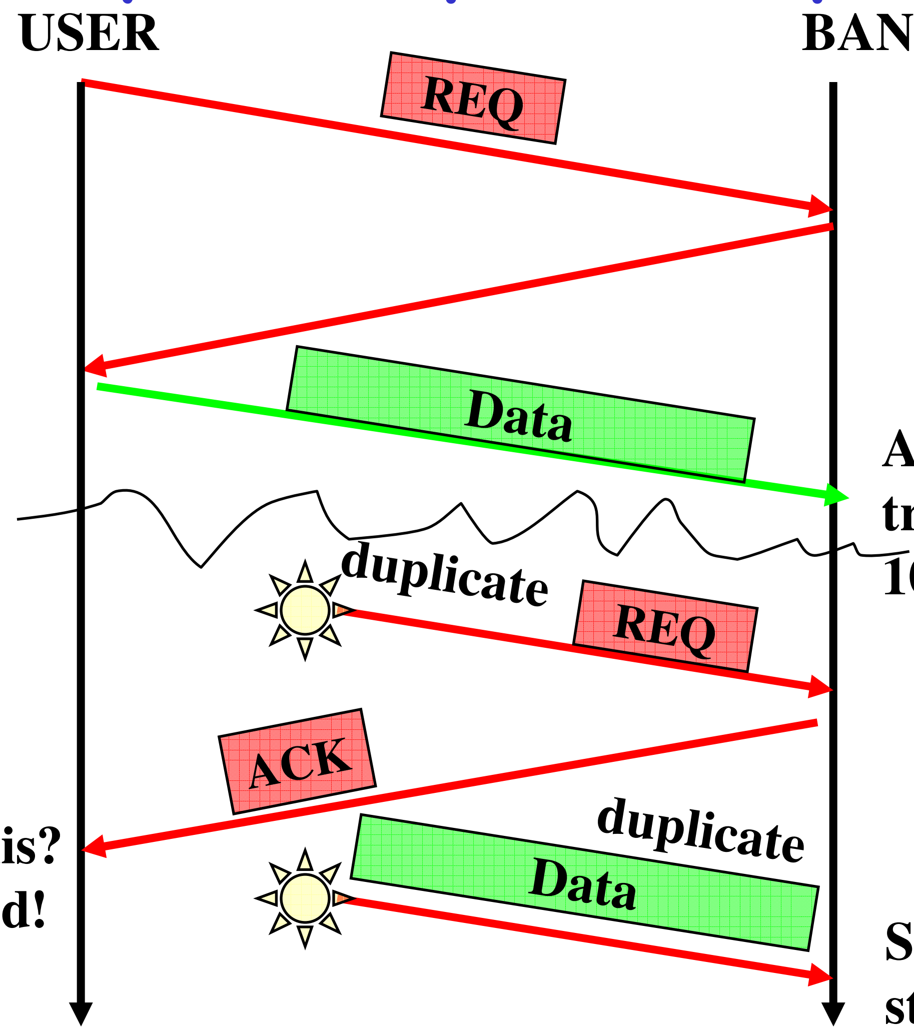
Transmission control block

Contains also info on: sockets, pointers to the users' send and receive buffers, to the retransmit queue and to the current segment

Connection establishment: simplest approach (non TCP)



Delayed duplicate problem



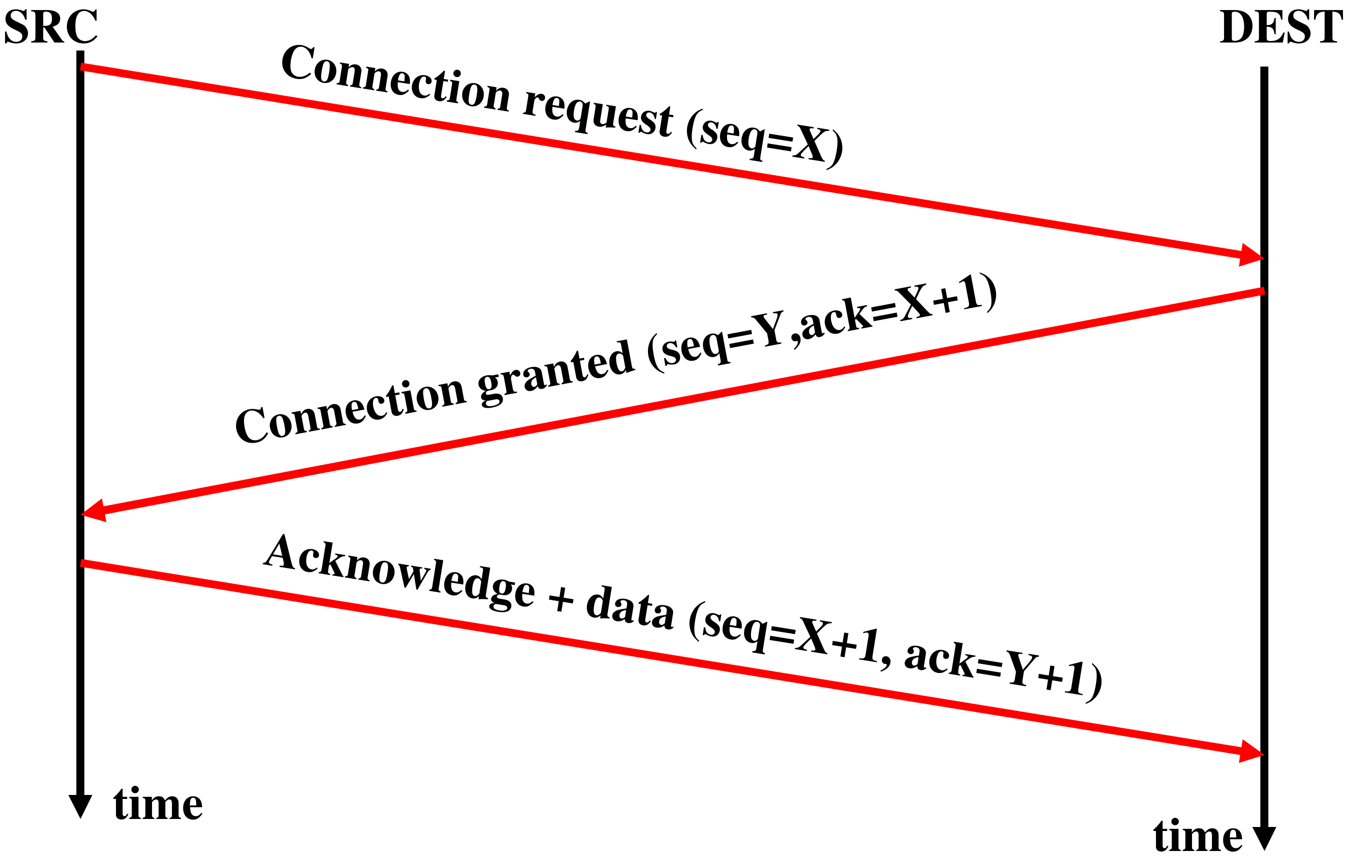
Application: transactional (sell 100000$ stocks)

What is this?
Oh my God!
Too late!!!

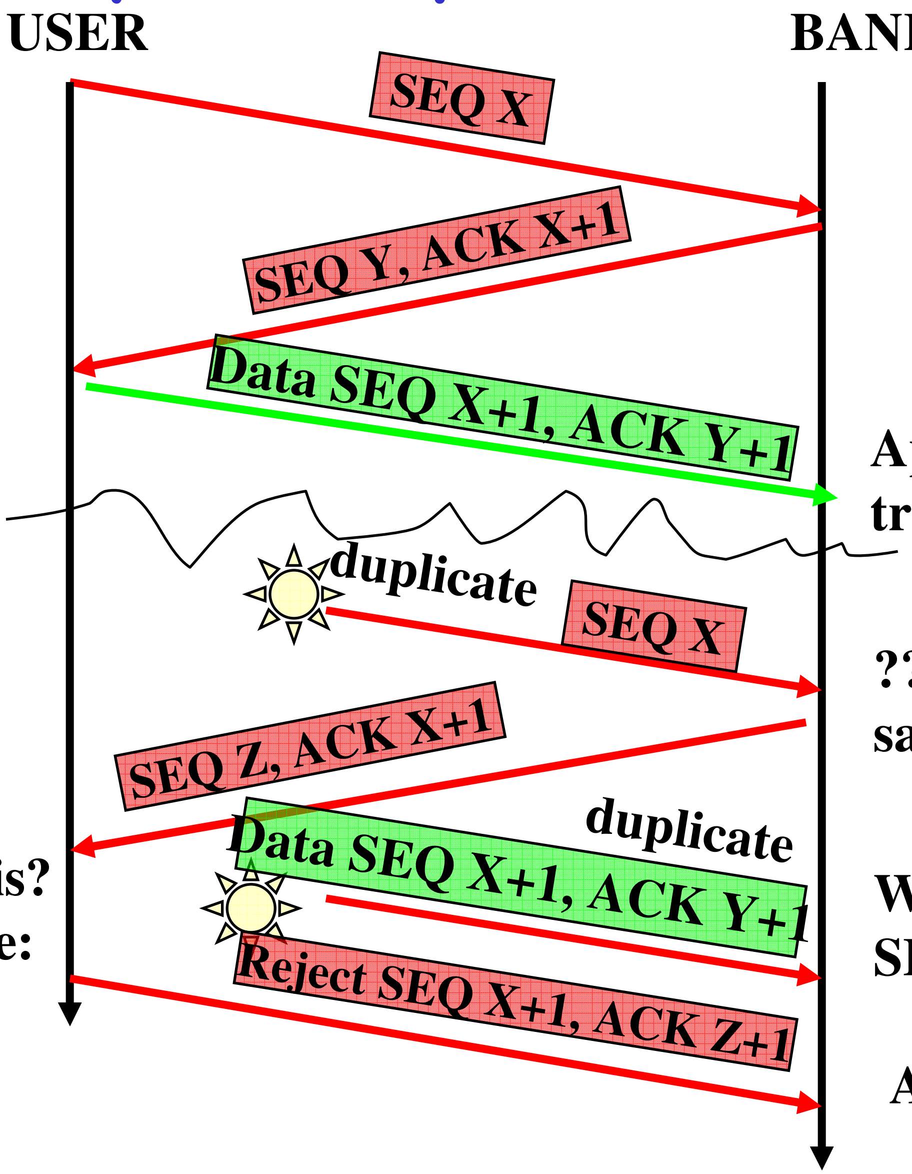
Selling other 100000$ stocks!!!!!

Solution: three way handshake

Tomlinson 1975


SRC
DEST
Connection request (seq=X)
Connection granted (seq=Y,ack=X+1)
Acknowledge + data (seq=X+1, ack=Y+1)
time
time

Delayed duplicate detection



USER

BANK

SEQ X

SEQ Y, ACK X+1

Data SEQ X+1, ACK Y+1

duplicate

SEQ X

SEQ Z, ACK X+1

duplicate

Data SEQ X+1, ACK Y+1

Reject SEQ X+1, ACK Z+1

Application:
transactional (selling stocks)

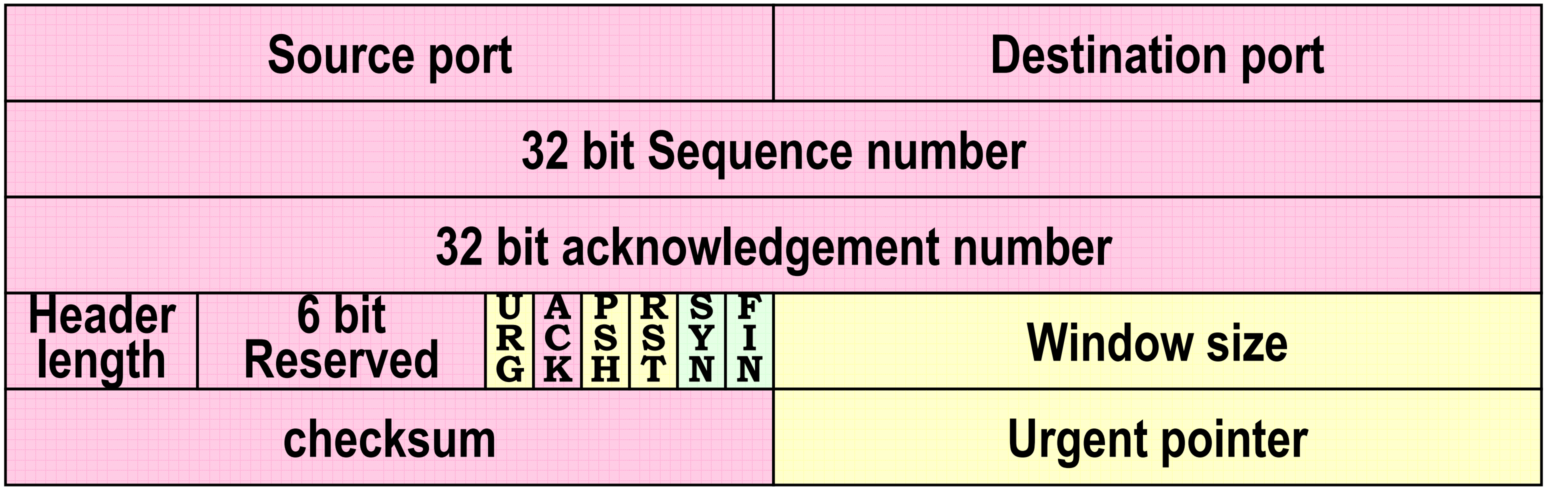
??? What a case: request with same indicator X? anyway...

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



Source port

Destination port

32 bit Sequence number

32 bit acknowledgement number

Header length

6 bit Reserved

URG

ACK

PSH

RST

SYN

FIN

Window size

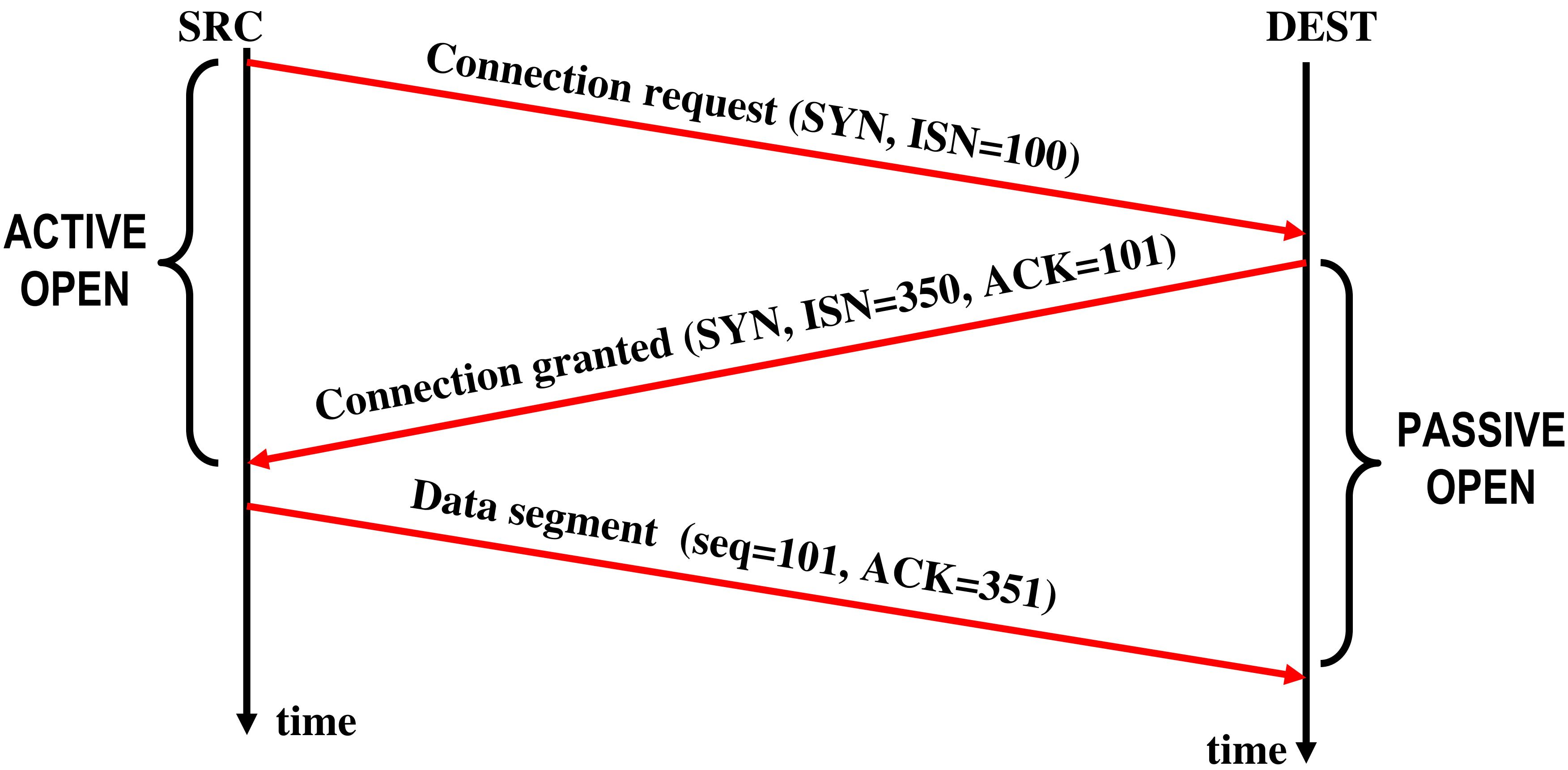
checksum

Urgent pointer

- SYN (synchronize sequence numbers): used to open connection
  - SYN present: this host is setting up a connection
  - SEQ with SYN: means initial sequence number (ISN)
  - data bytes numbered from ISN+1.
- FIN: no more data to send
  - used to close connection

...more later about connection closing...

Three way handshake in TCP


SRC
DEST
Connection request (SYN, ISN=100)
ACTIVE OPEN
Connection granted (SYN, ISN=350, ACK=101)
PASSIVE OPEN
Data segment (seq=101, ACK=351)
time
time

Full ***duplex connection: opened in both ways***

SRC: performs ACTIVE OPEN

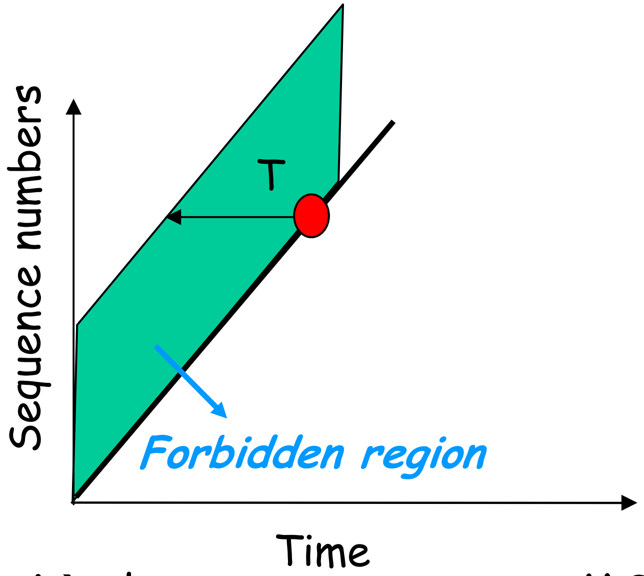
DEST: Performs PASSIVE OPEN

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

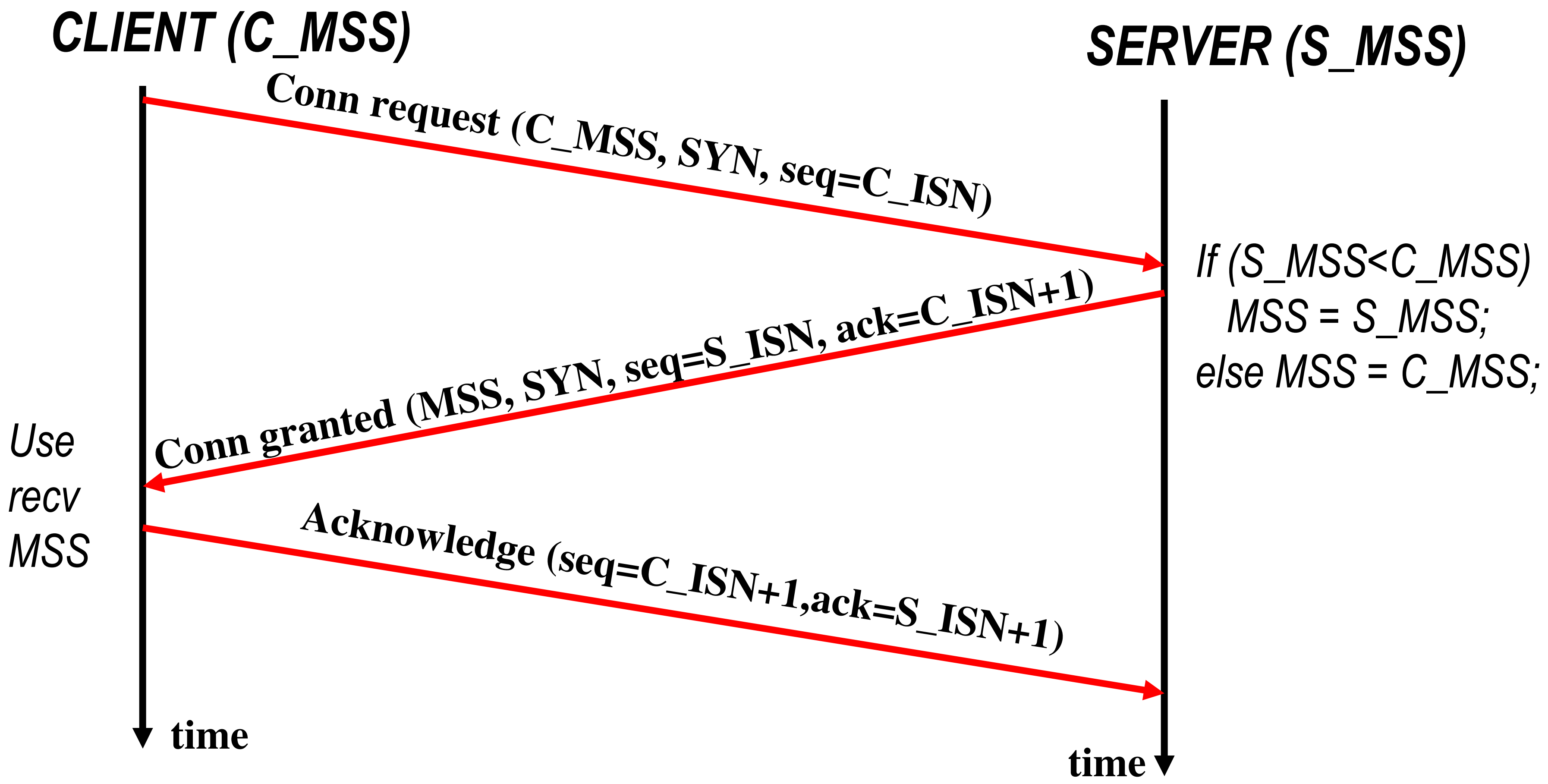


- Aging dei pacchetti→ dopo un certo tempo MSL (Maximum Segment Lifetime) i pacchetti eliminati dalla rete
- Sequence numbers basati sul clock
- Un ciclo del clock circa 4 ore; MSL circa 2 minuti.
- → Se non ci sono crash che fanno perdere il valore dell'ultimo sequence number usato NON ci sono problemi (si riusa lo stesso sequence number ogni 4 ore circa, quando il segmento precedentemente trasmesso con quel sequence number non è più in rete)
- → 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).

Maximum Segment Size - MSS

- Announced at setup by both ends.
- Lower value selected (indeed min of lower value and largest size permitted by IP layer).
- MSS sent in the Options header of the SYN segment
  - clearly cannot (=ignored if happens) send MSS in a non SYN segment, as connection has been already setup
  - when SYN has no MSS, default value 536 used
- goal: the larger the MSS, the better...
  - until fragmentation occurs
  - e.g. if host is on ethernet, sets MSS=1460
    - 1500 max ethernet size - 20 IP header - 20 TCP header

MSS advertise


CLIENT (C_MSS)
SERVER (S_MSS)
Conn request (C_MSS, SYN, seq=C_ISN)
If (S_MSS<C_MSS)
MSS = S_MSS;
else MSS = C_MSS;
Conn granted (MSS, SYN, seq=S_ISN, ack=C_ISN+1)
Use
recv
MSS
Acknowledge (seq=C_ISN+1,ack=S_ISN+1)
time
time

Does not avoid fragmentation to occur WITHIN the network!!

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

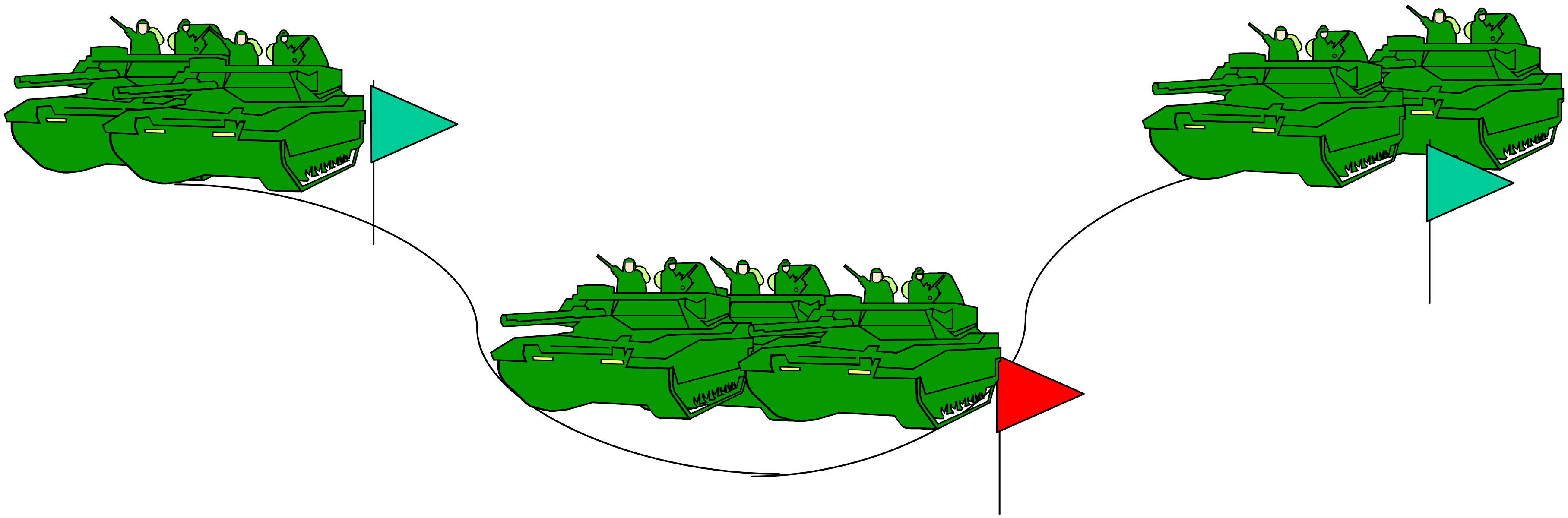
- server allocates buffers
- specifies server initial seq. #

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

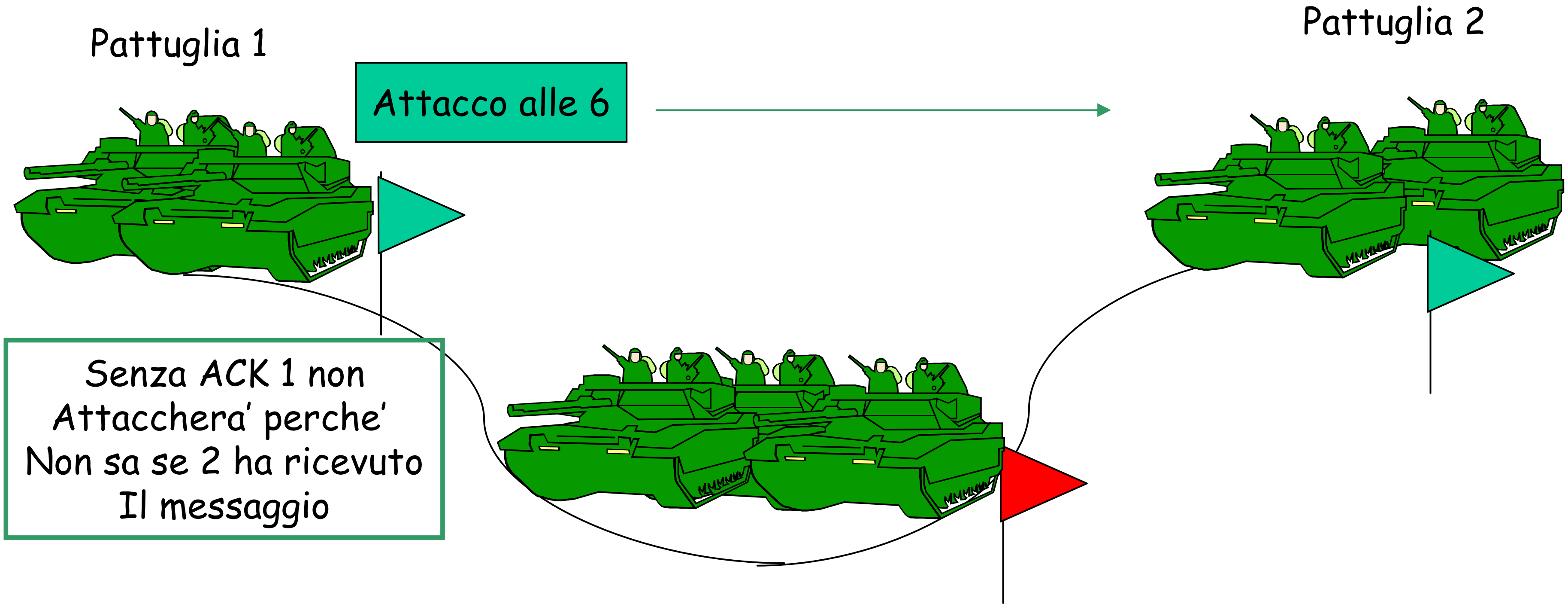
Problema dei due eserciti

- 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?



Pattuglia 1

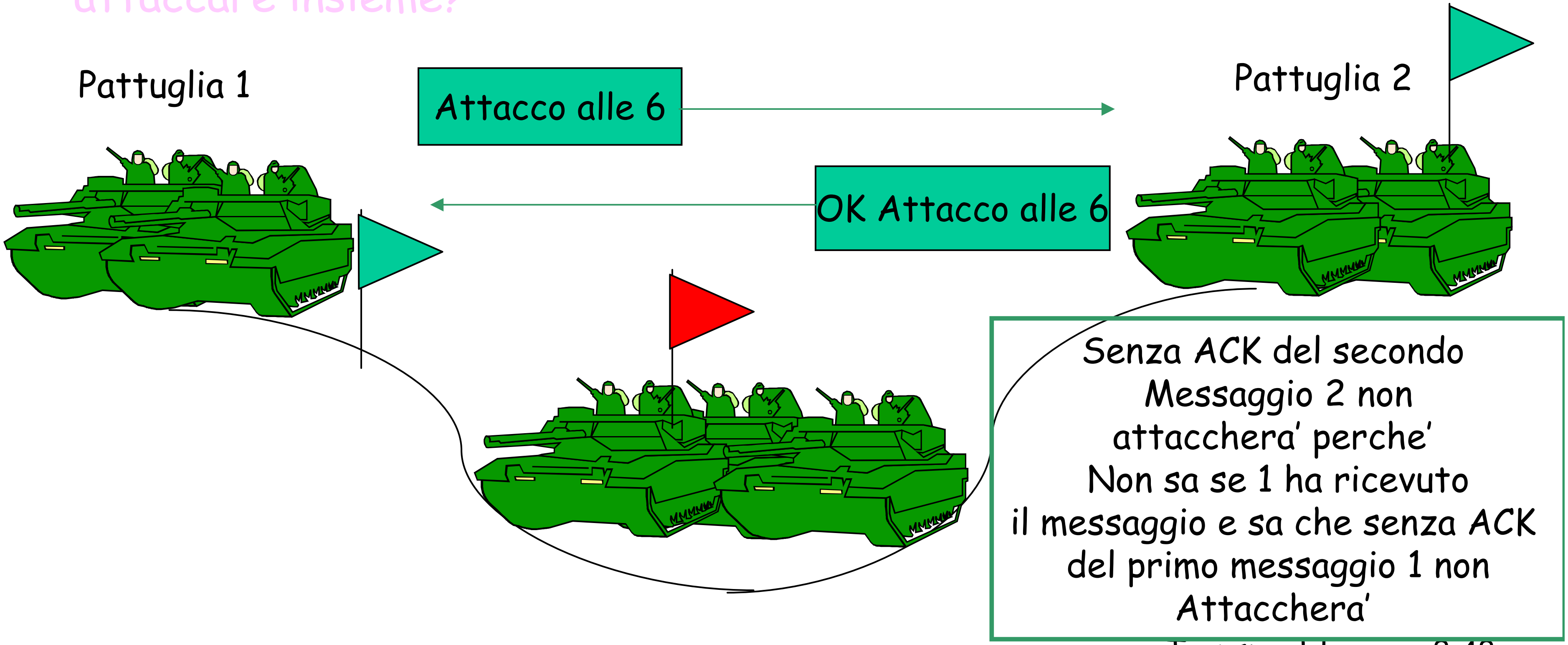
Pattuglia 2

Attacco alle 6

Senza ACK 1 non
Attacchera' perche'
Non sa se 2 ha ricevuto
Il messaggio

Problema dei due eserciti

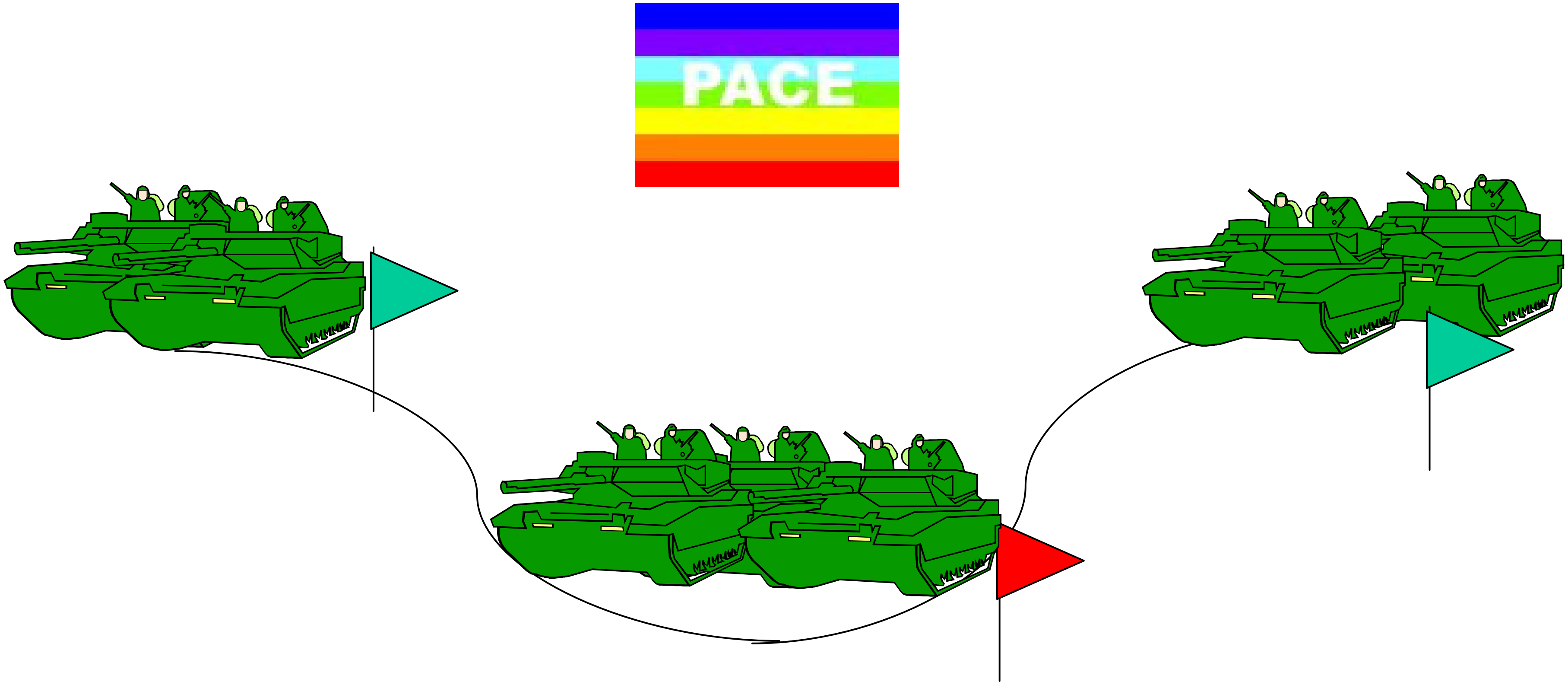
r L’esercito rosso e’ globalmente più debole. Se le due pattuglie verdi attaccano insieme lo sconfiggono, altrimenti perdono. Possono scambiarsi messaggi relativi all’orario in cui attaccheranno e di ACK di un messaggio ricevuto. I messaggeri che li portano possono pero’ essere catturati e quindi il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



Senza ACK del secondo
Messaggio 2 non
attacchera’ perche’
Non sa se 1 ha ricevuto
il messaggio e sa che senza ACK
del primo messaggio 1 non
Attacchera’

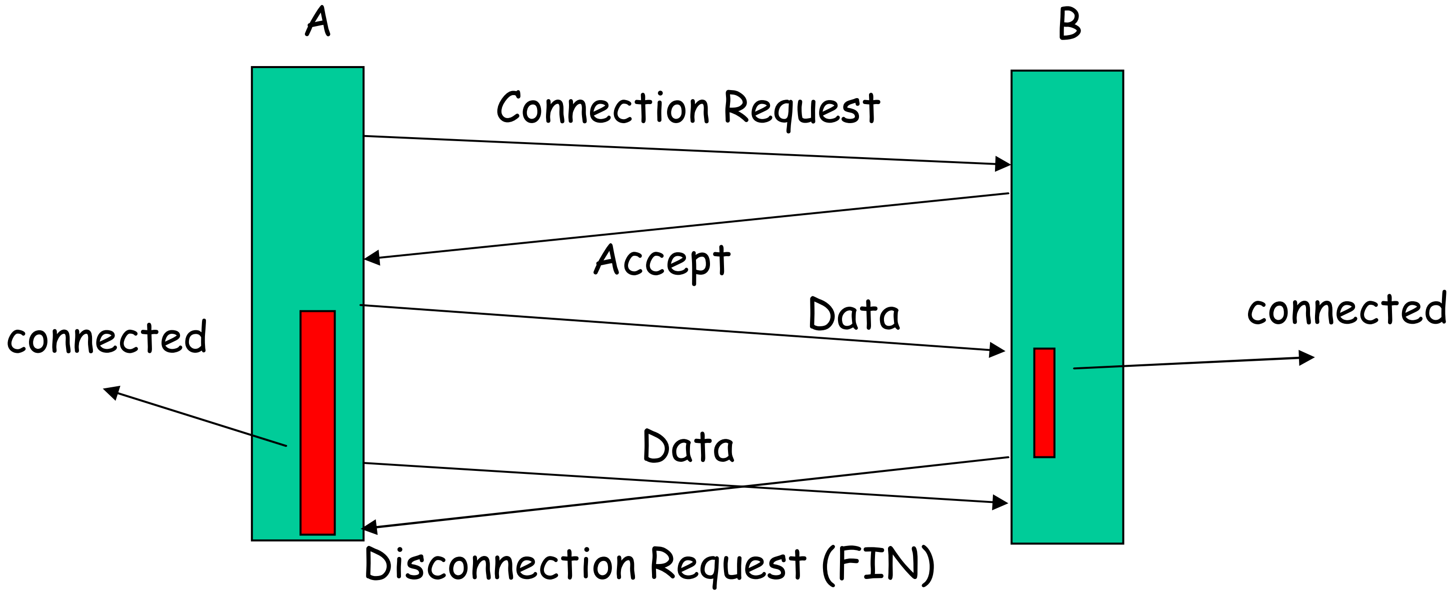
Problema dei due eserciti

- 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??

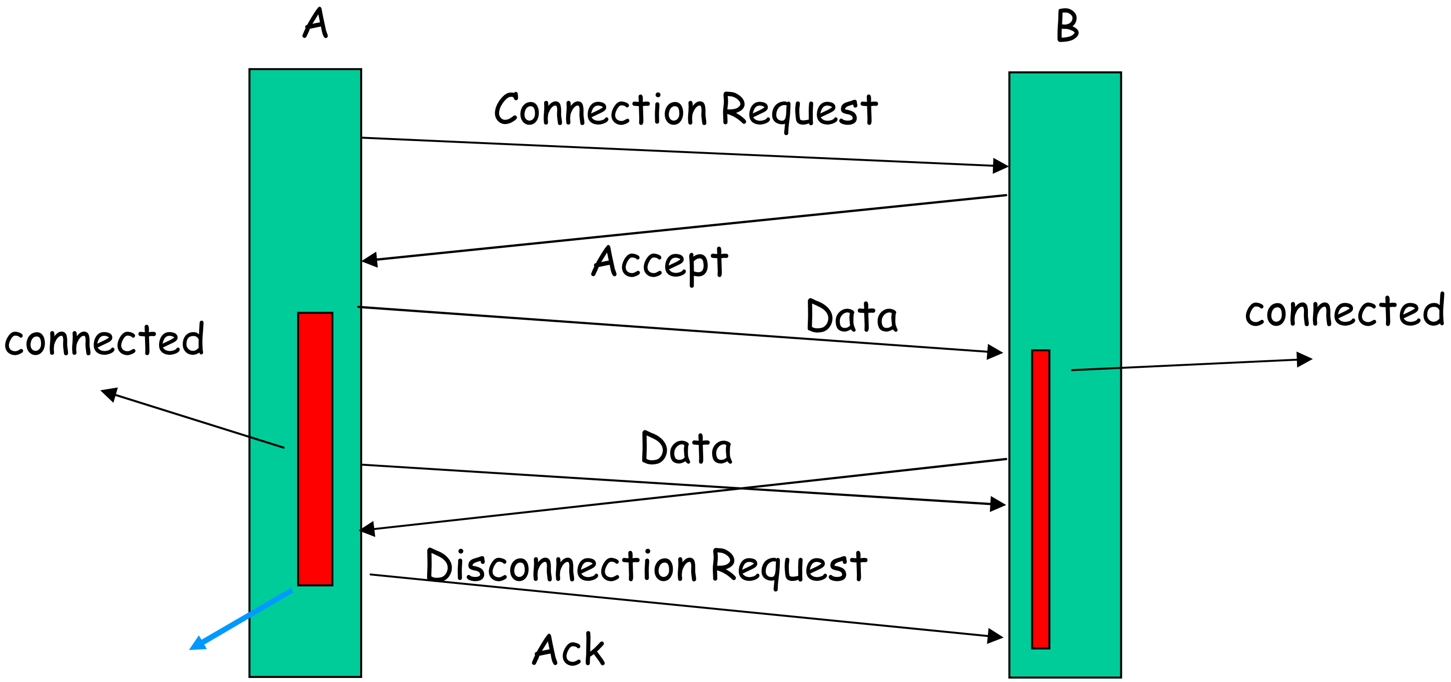
- Chiusura di una connessione. Vorremmo un accordo tra le due peer entity o rischiamo di perdere dati.



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 di quando si attacca un esercito nemico. Possibili malfunzionamenti. Soluzioni 'di recovery' in questi casi

TCP Connection Management (cont.)

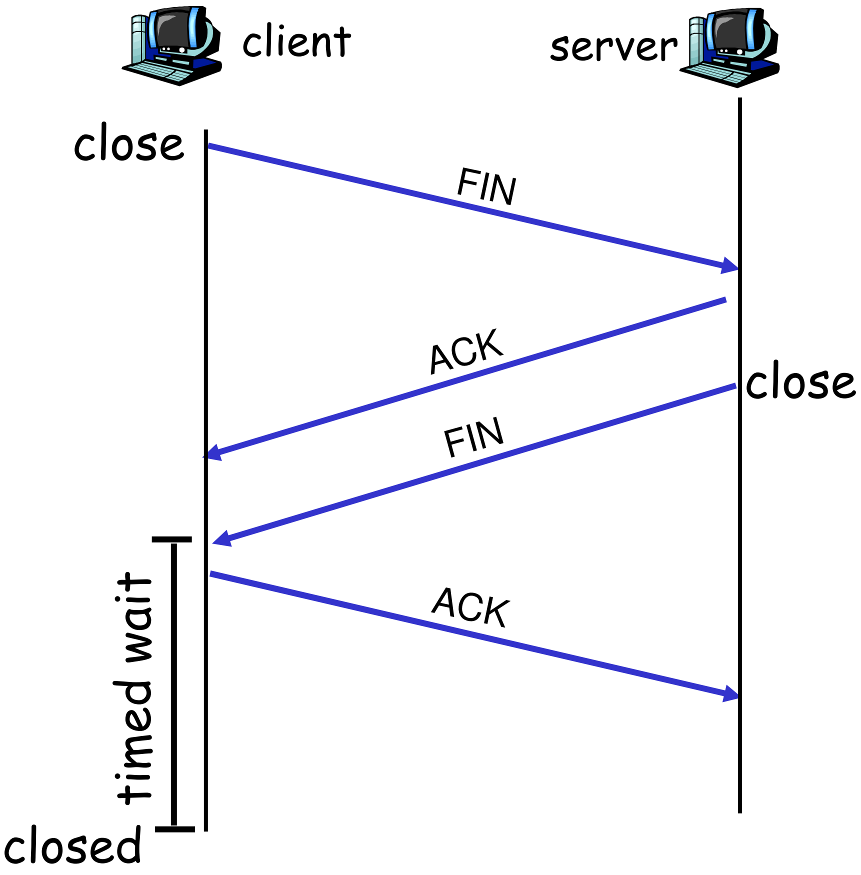
Since it is impossible to solvethe proble use simple solution: two way handshake

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.

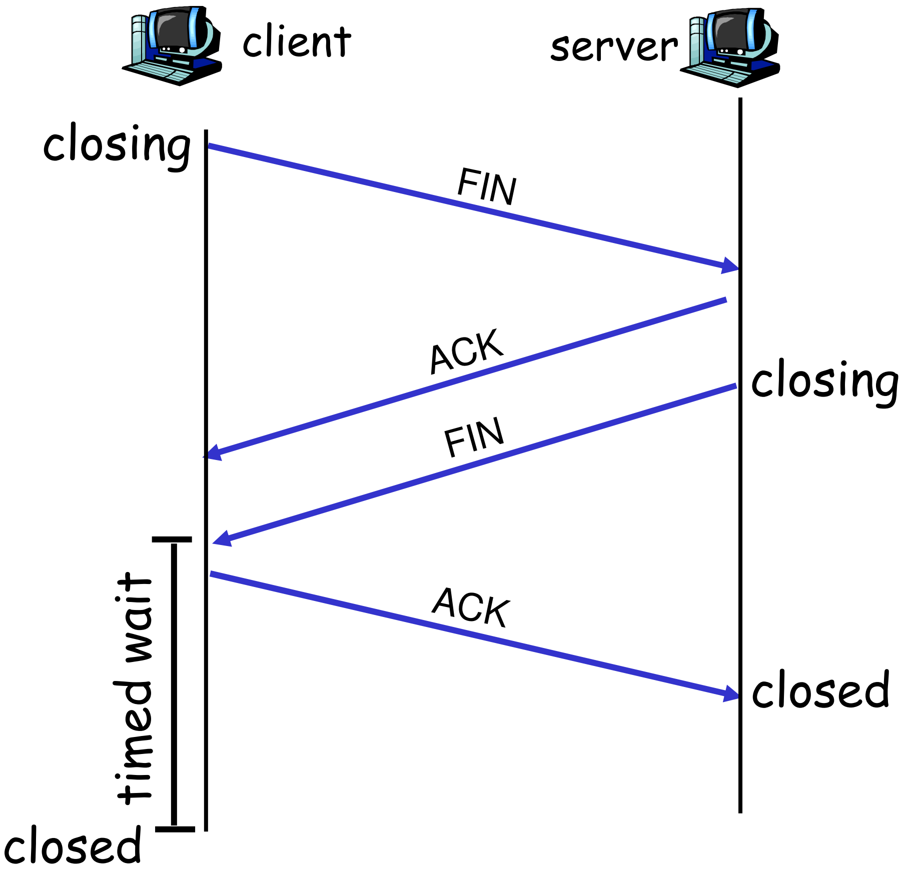


TCP Connection Management (cont.)

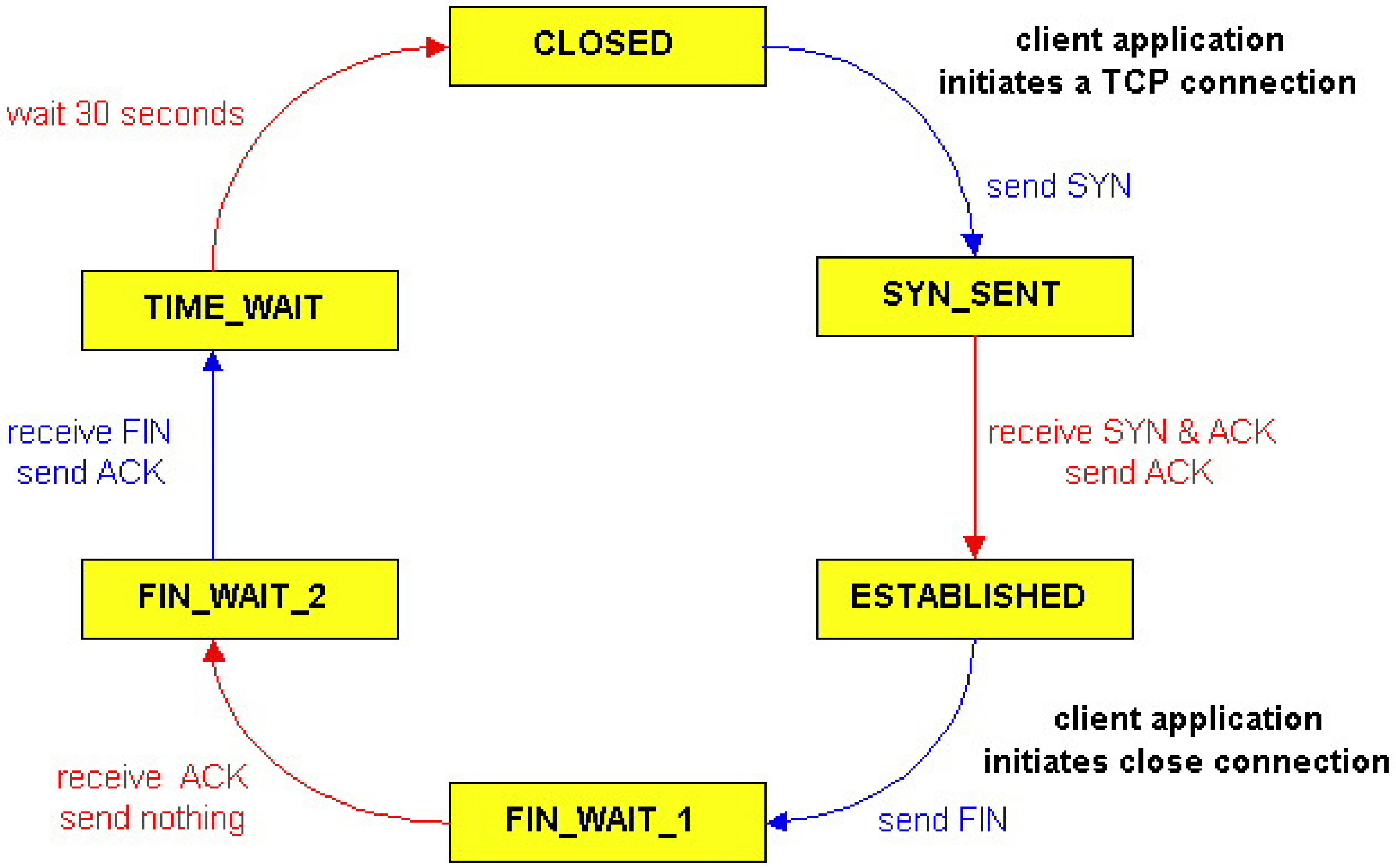
Step 3: client receives FIN, replies with ACK.

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

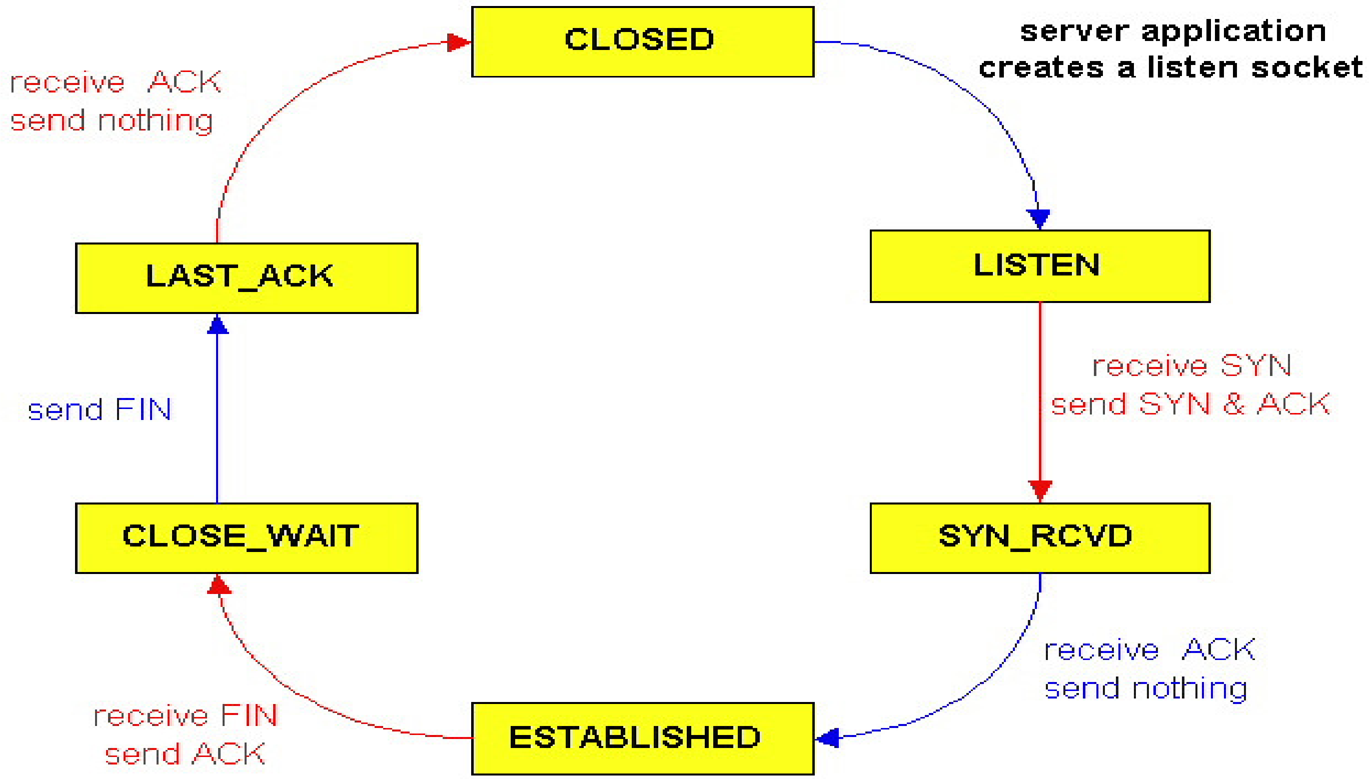
Step 4: server, receives ACK. Connection closed.



Connection states - Client



Connection States - Server



Why TIME_WAIT?

- MSL (Maximum Segment Lifetime): **maximum time a segment can live in the Internet**
    - no timers on IP packets! Only hop counter
    - RFC 793 specifies MSL=2min, but each implementation has its own value (from 30s to 2min)
- TIME_WAIT state: **2 x MSL**
    - **allows to "clean" the network of delayed packets belonging to the connection**
    - **2xMSL because a lost FIN_ACK implies a new FIN from server**
- during TIME_WAIT conn sock pair reserved
    - **many implementations even more restictive (local port non reusable)**
    - **clearly this may be a serious problem when restarting server daemon (must pause from 1 to 4 minutes…)**

<table>
<tr><td colspan="8">Source port</td><td>Destination port</td></tr>
<tr><td colspan="9">32 bit Sequence number</td></tr>
<tr><td colspan="9">32 bit acknowledgement number</td></tr>
<tr><td>Header length</td><td>6 bit Reserved</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>Window size</td></tr>
<tr><td colspan="8">checksum</td><td>Urgent pointer</td></tr>
</table>

- RST (Reset)
  - sent whenever a segment arrives and does not apparently belong to the connection
  - typical RST case: connection request arriving to port not in use
- Sending RST within an active connection:
  - allows ***aborting release*** of connection (versus ***orderly release***)
    - any queued data thrown away
    - receiver of RST can notify app that abort was performed at other end

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:

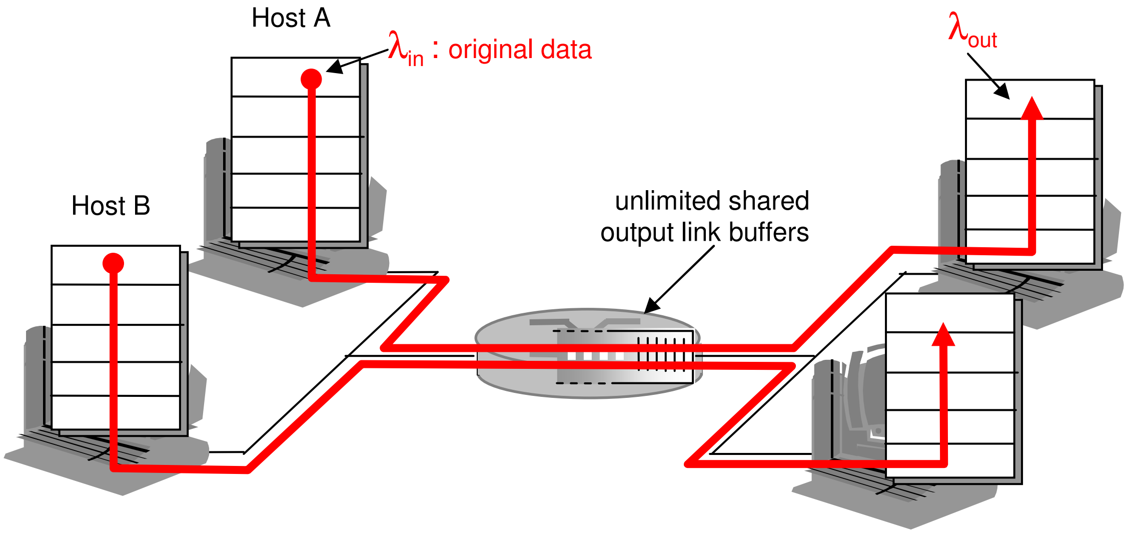
- informally: “too many sources sending too much data too fast for *network* to handle”
- different from flow control!
- manifestations:
  - lost packets (buffer overflow at routers)
  - long delays (queueing in router buffers)
- a top-10 problem!

Causes/costs of congestion: scenario 1

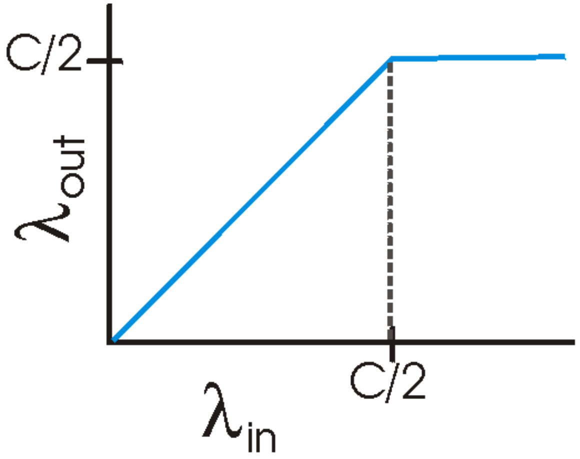
r two senders, two receivers

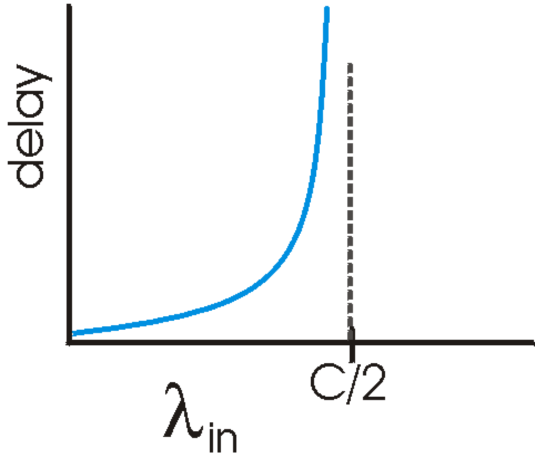
r one router, infinite buffers

r no retransmission



Host A
λ_{in} : original data
λ_{out}
Host B
unlimited shared output link buffers



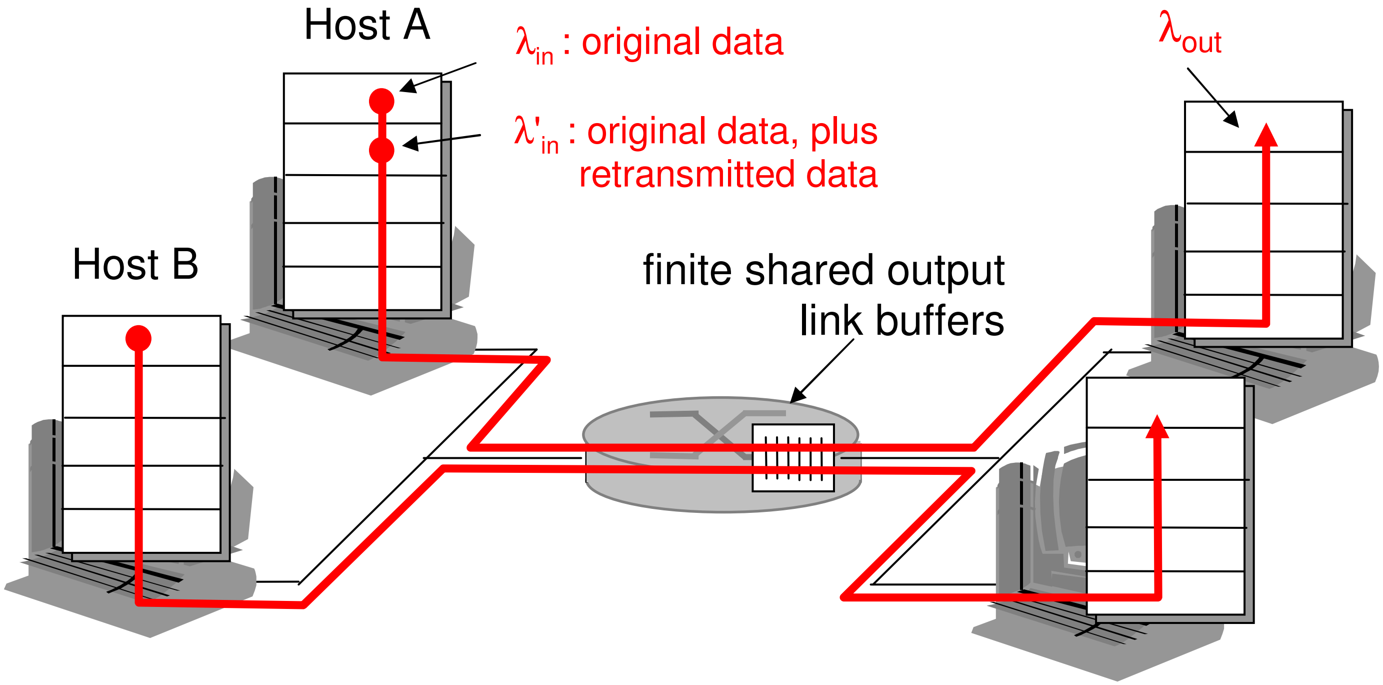


r large delays when congested

r maximum achievable throughput

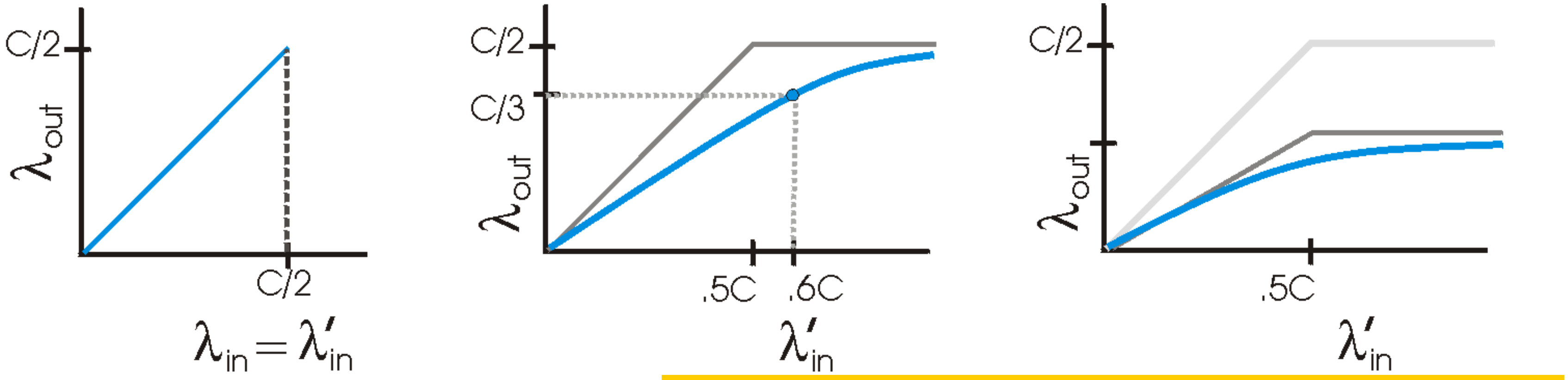
Causes/costs of congestion: scenario 2

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


Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
λ_{out}
Host B
finite shared output link buffers

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'_{in} 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

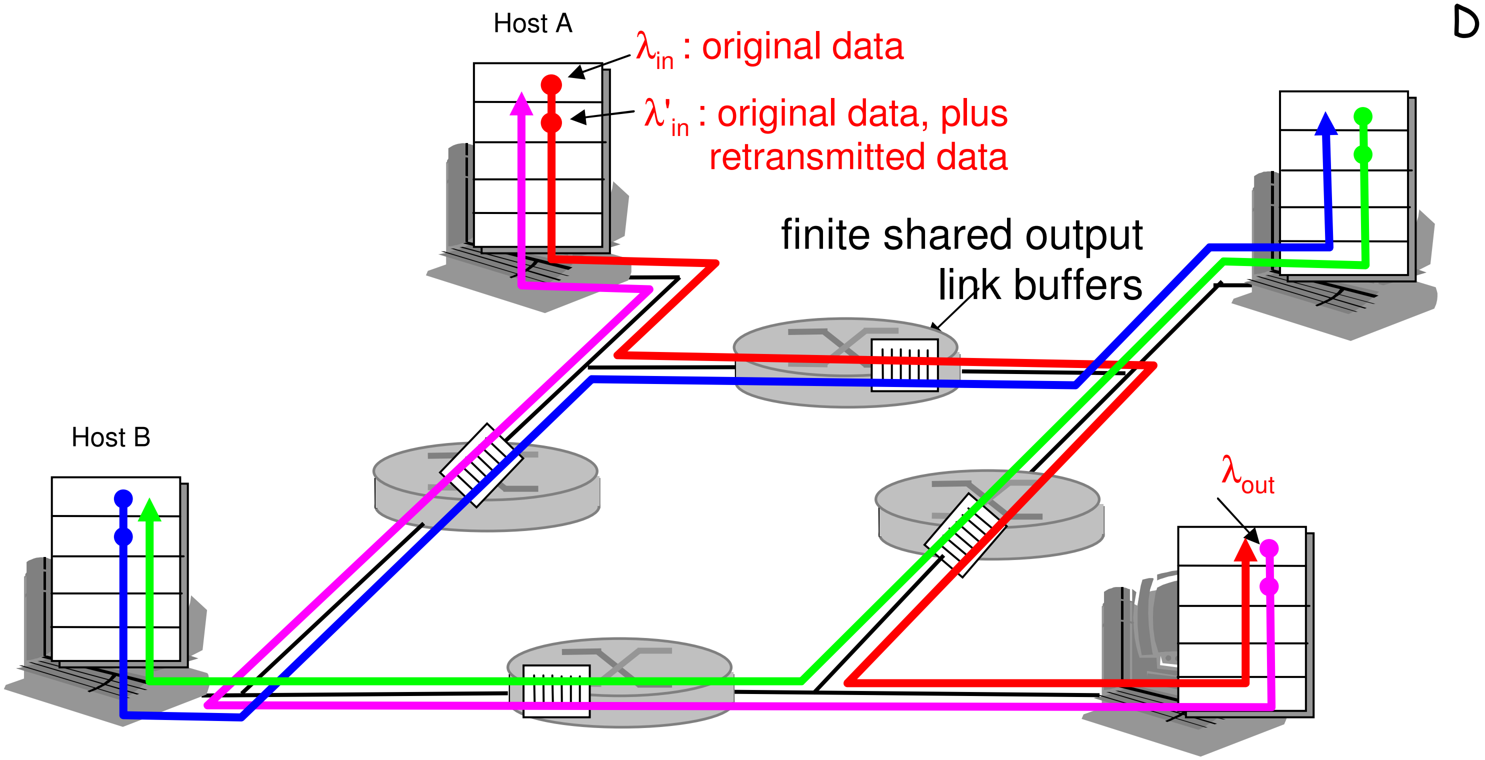
Causes/costs of congestion: scenario 3

r four senders

r multihop paths

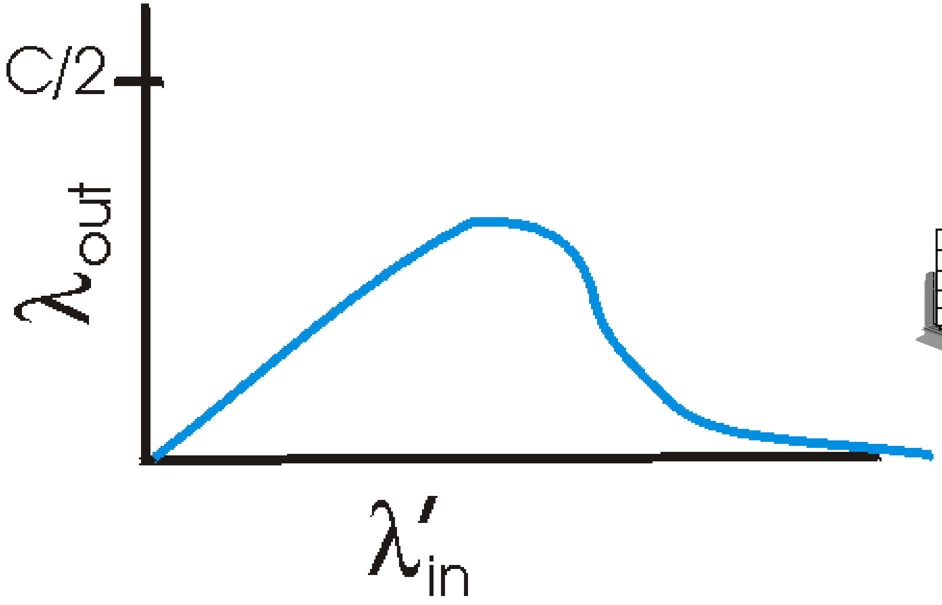
r timeout/retransmit

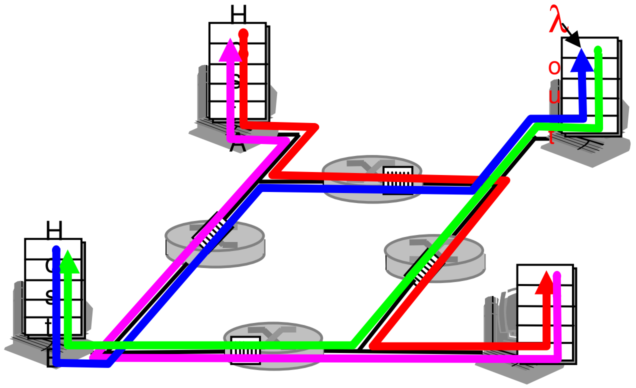
Q: what happens as λ_{in} and λ'_{in} increase ?


Host A
λ_{in} : original data
λ'_{in} : original data, plus retransmitted data
D
finite shared output link buffers
Host B
λ_{out}

D-B traffic high

Causes/costs of congestion: scenario 3





Another "cost" of congestion:

r 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 sender should send at

Chapter 3 outline

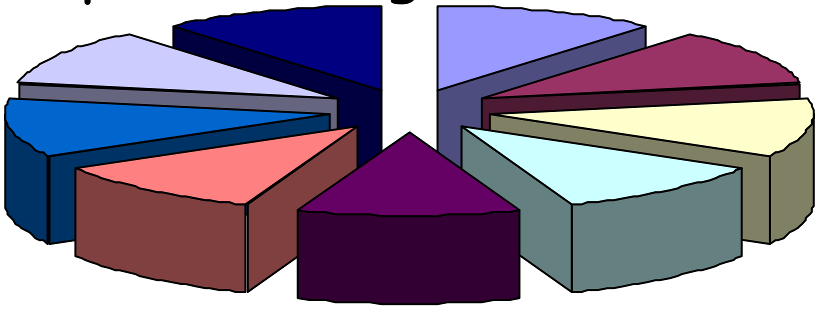
- 3.1 Transport-layer services
- 3.2 Multiplexing and demultiplexing
- 3.3 Connectionless transport: UDP
- 3.4 Principles of reliable data transfer
- 3.5 Connection-oriented transport: TCP
  - segment structure
  - reliable data transfer
  - flow control
  - connection management
- 3.6 Principles of congestion control
- 3.7 TCP congestion control

TCP: controllo di congestione

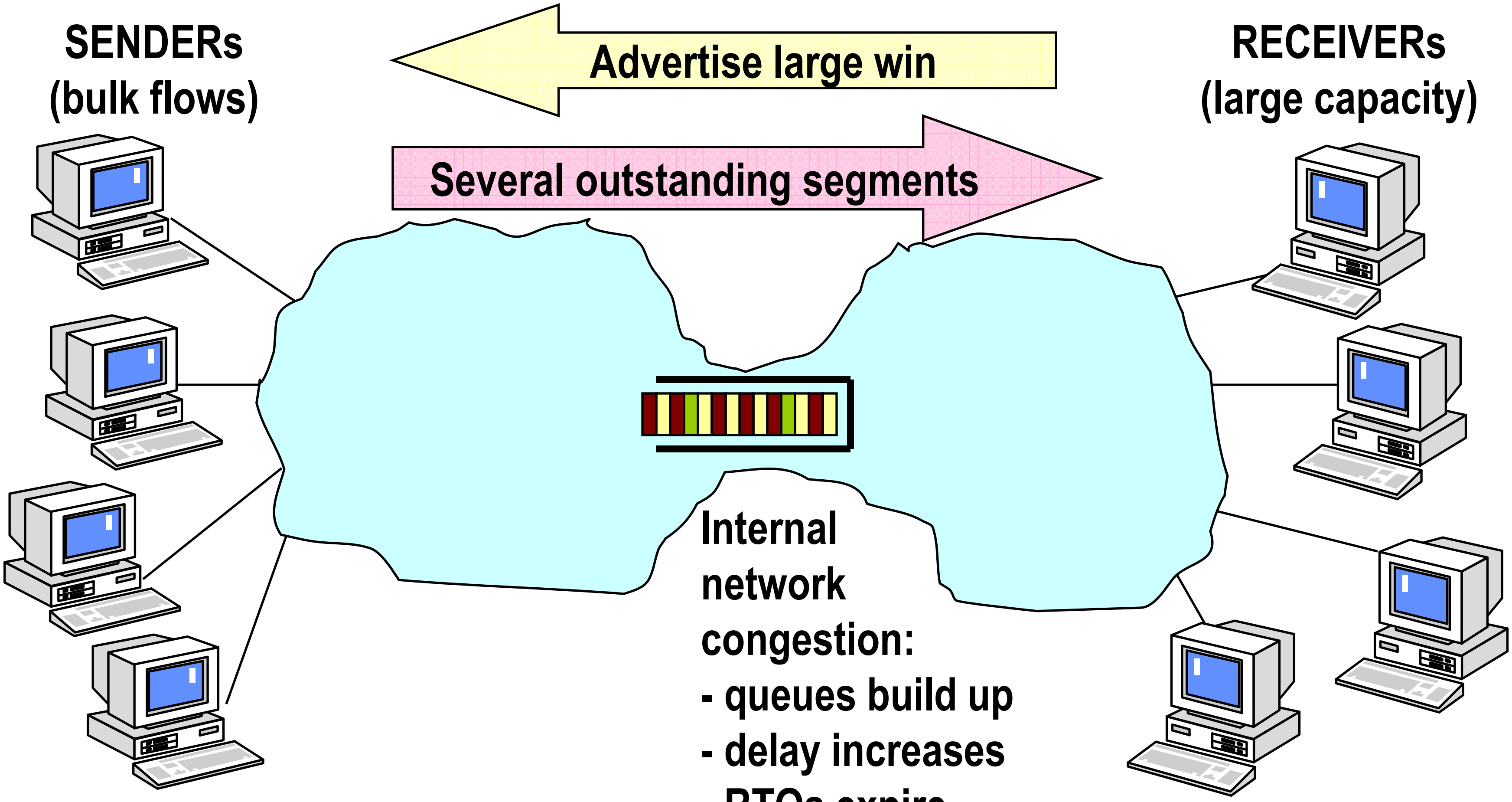
- Il TCP ha dei meccanismi di controllo della congestione
  - il flusso dei dati in ingresso in rete è anche regolato dalla situazione di traffico in rete
  - se il traffico in rete porta a situazioni di congestione il TCP riduce velocemente il traffico in ingresso
  - in rete non vi è nessun meccanismo per notificare esplicitamente le situazioni di congestione
  - il TCP cerca di scoprire i problemi di congestione sulla base degli eventi di perdita dei pacchetti

TCP: controllo di congestione

- il meccanismo si basa ancora sulla sliding window la cui larghezza viene dinamicamente regolata in base alle condizioni in rete
- in linea di principio scopo del controllo è far si che il flusso emesso da ciascuna sorgente venga regolato in modo tale che il flusso complessivo offerto a ciascun canale non superi la sua capacità
- tutti i flussi possono essere ridotti in modo tale che la capacità della rete venga condivisa da tutti in misura se possibile uguale



The problem of congestion

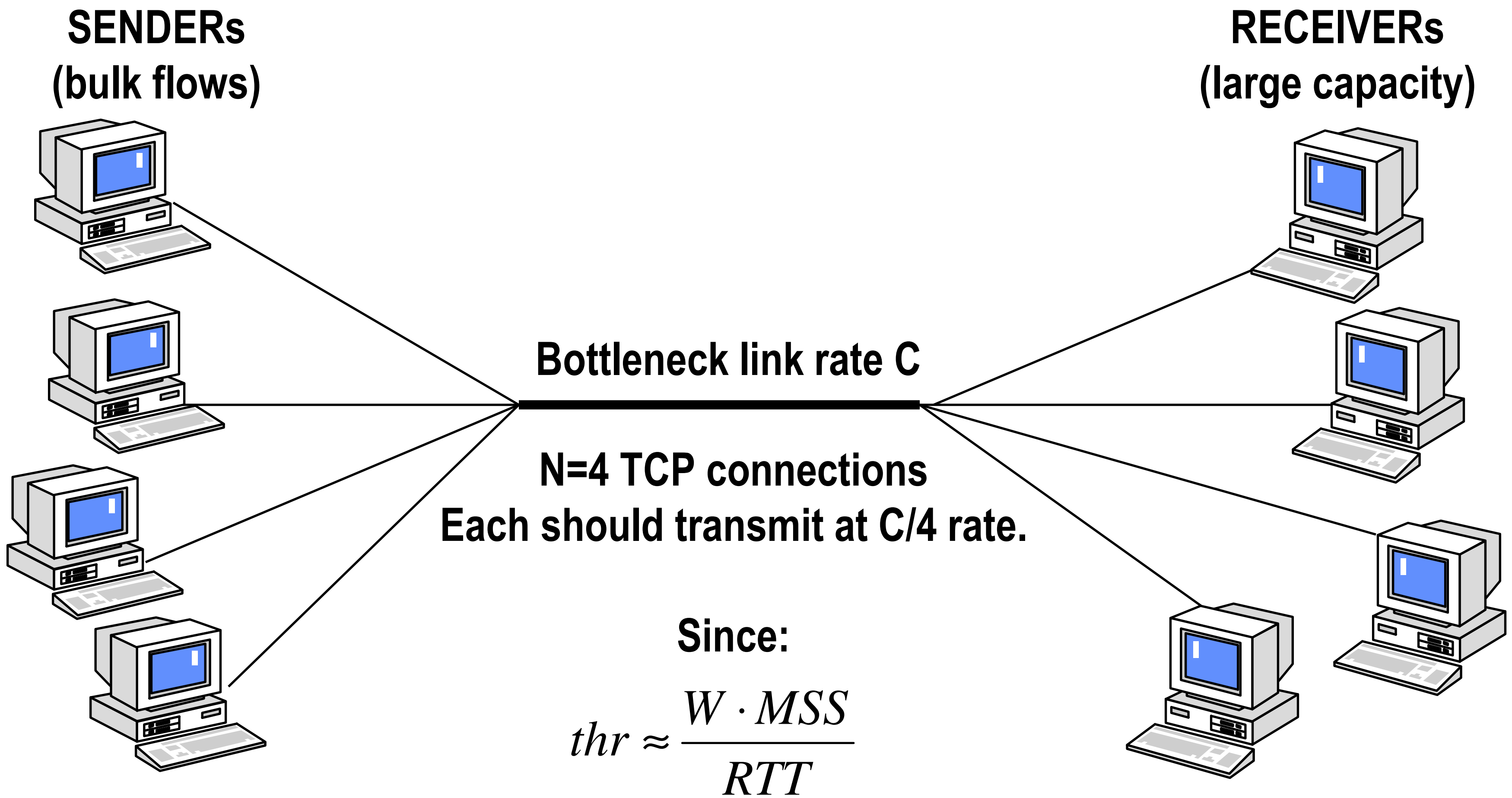


Internal network congestion:

- queues build up
- delay increases
- RTOs expire

-more segments transmitted, more Segments retransmitted -> more congestion!

The goal of congestion control



SENDERs
(bulk flows)

RECEIVERs
(large capacity)

Bottleneck link rate C

N=4 TCP connections
Each should transmit at C/4 rate.

Since:

$$thr \approx \frac{W \cdot MSS}{RTT}$$

Each should adapt W accordingly…
How sources can be lead to know the RIGHT value of W??

TCP approach for detecting and controlling congestion

- IP protocol does not implement mechanisms to detect congestion in IP routers
  - Unlike other networks, e.g. ATM
- necessary indirect means (TCP is an end-to-end protocol)
- TCP approach: congestion detected by lack of acks
  - couldn’t work efficiently in the 60s & 70s (error prone transmission lines)
  - OK in the 80s & 90s (reliable transmission)
  - what about wireless networks???
- Controlling congestion: use a SECOND window (congestion window)
  - Locally computed at sender
  - Outstanding segments: min(receiver_window, congestion_window)

TCP Congestion Control

- end-end control (no network assistance)
- sender limits transmission:

  LastByteSent-LastByteAcked
  ≤ 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

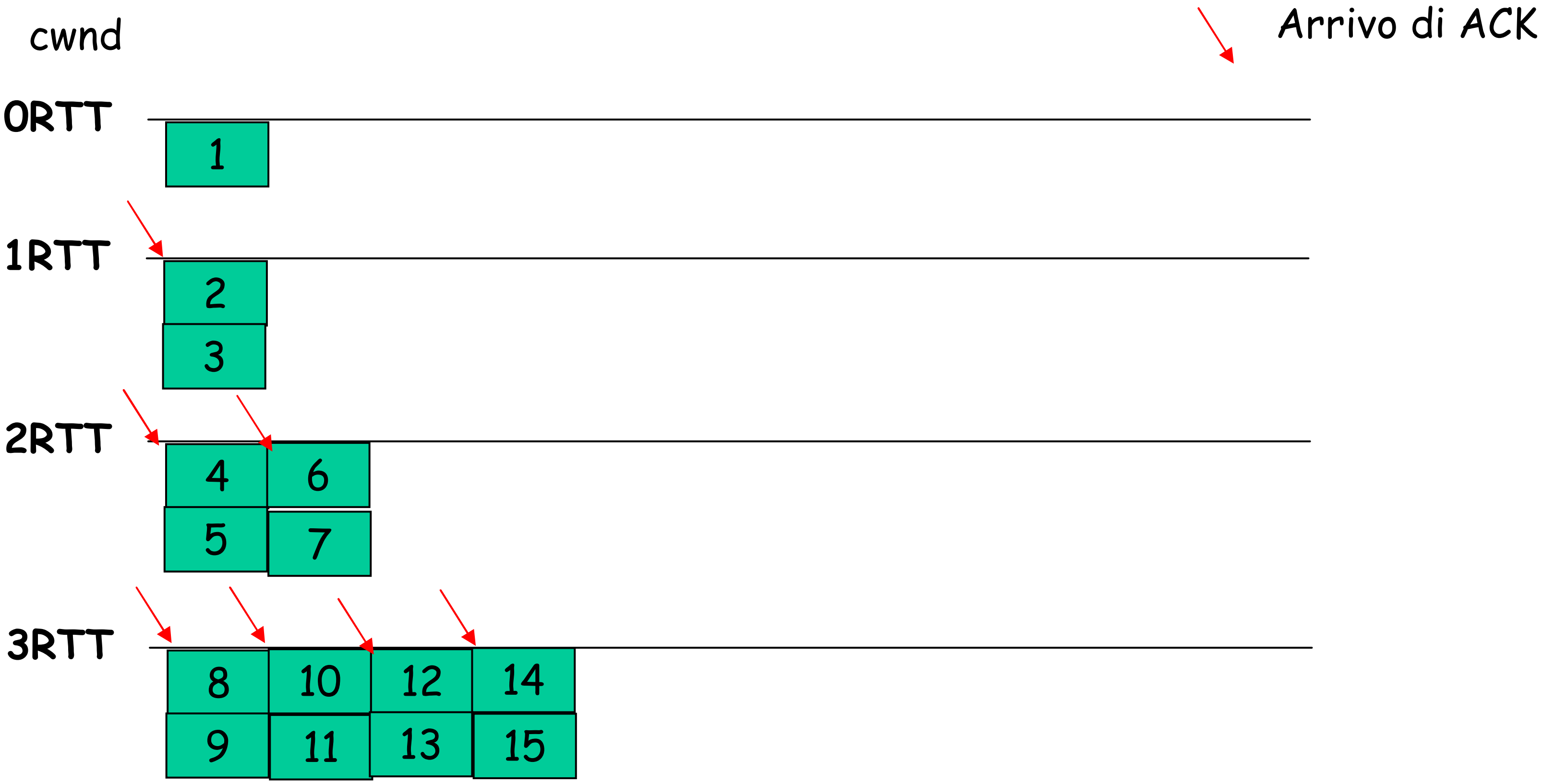
three mechanisms:

  - AIMD
  - slow start
  - conservative after timeout events

Starting a TCP transmission

- A new offered flow may suddenly overload network nodes
  - receiver window is used to avoid recv buffer overflow
  - But it may be a large value (16-64 KB)
- Idea: slow start
  - Start with small value of cwnd
  - And increase it as soon as packets get through
    - Arrival of ACKs = no packet losts = no congestion
- Initial cwnd size:
  - Just 1 MSS!
  - Recent (1998) proposals for more aggressive starts (up to 4 MSS) have been found to be dangerous

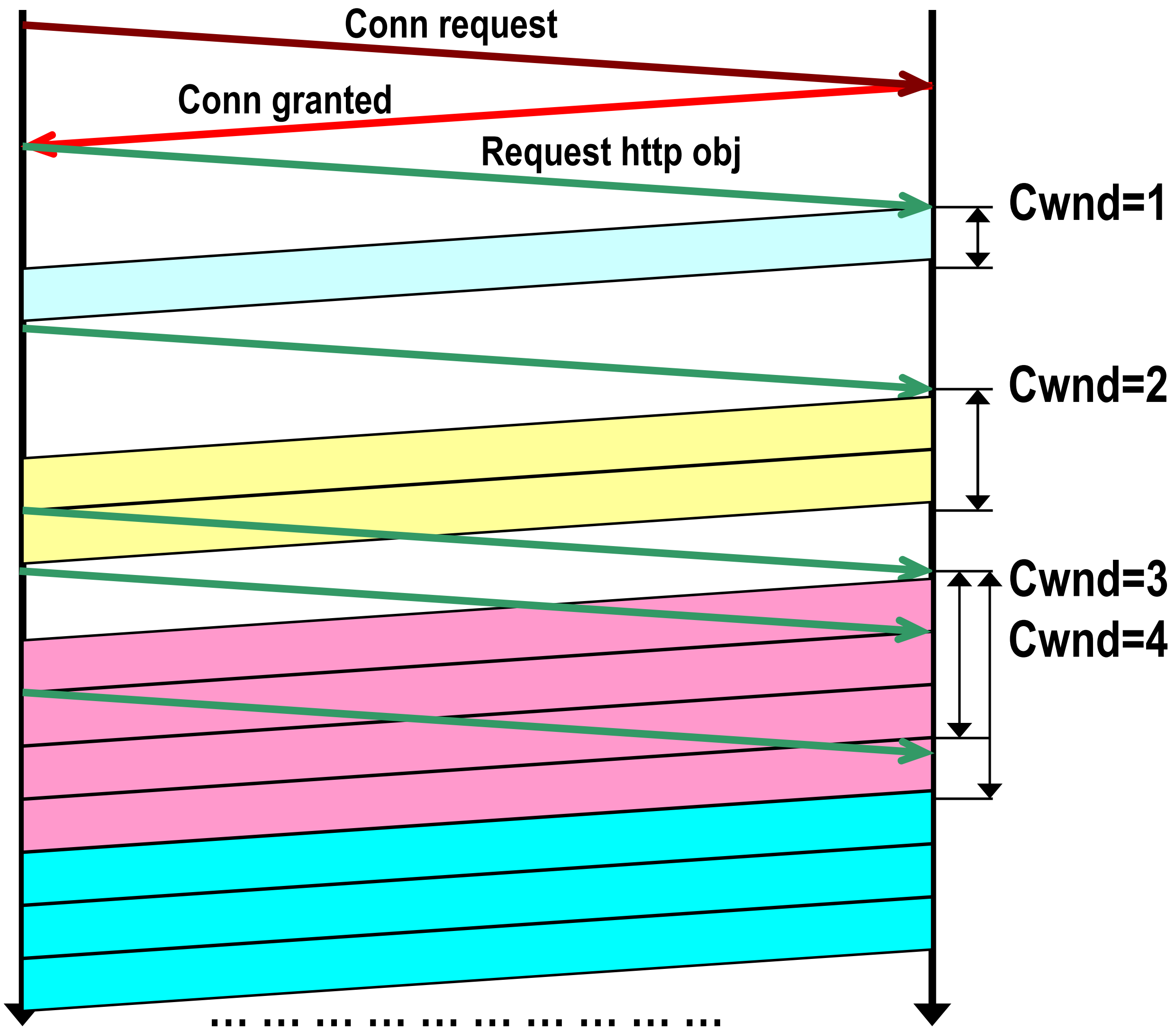
Slow start: the idea



Si trasmette il minimo tra window e cwd pacchetti

Slow start - exponential increase

- ➔ First start: set congestion window
  cwnd = 1MSS
- ➔ send cwnd segments
  - ⇨ assume cwnd <= receiver win
- ➔ upon successful reception:
  - ⇨ Cwnd +=1 MSS
  - ⇨ i.e. double cwnd every RTT
  - ⇨ until reaching receiver window advertisement
  - ⇨ <u>OR a segment gets lost</u>



Detecting congestion and restarting

- Segment gets lost
  - Detected via RTO expiration
  - Indirectly notifies that one of the network nodes along the path has lost segment
    - Because of full queue
- Restart from cwnd=1 (slow start)
- But introduce a supplementary control: slow start threshold
    - sstresh = max(min(cwnd,window)/2,2MSS)
  - The idea is that we now KNOW that there is congestion in the network, and we need to increase our rate in a more careful manner…
  - Ssthresh defines the "congestion avoidance" region

Congestion avoidance

- If cwnd < ssthresh
  - Slow start region: Increase rate exponentially
- If cwnd >= ssthresh
  - Congestion avoidance region : Increase rate linearly
  - At rate 1 MSS per RTT
    - Practical implementation:
      cwnd += MSS*MSS/cwnd
    - Good approximation for 1 MSS per RTT
    - Alternative (exact) implementations: count!!

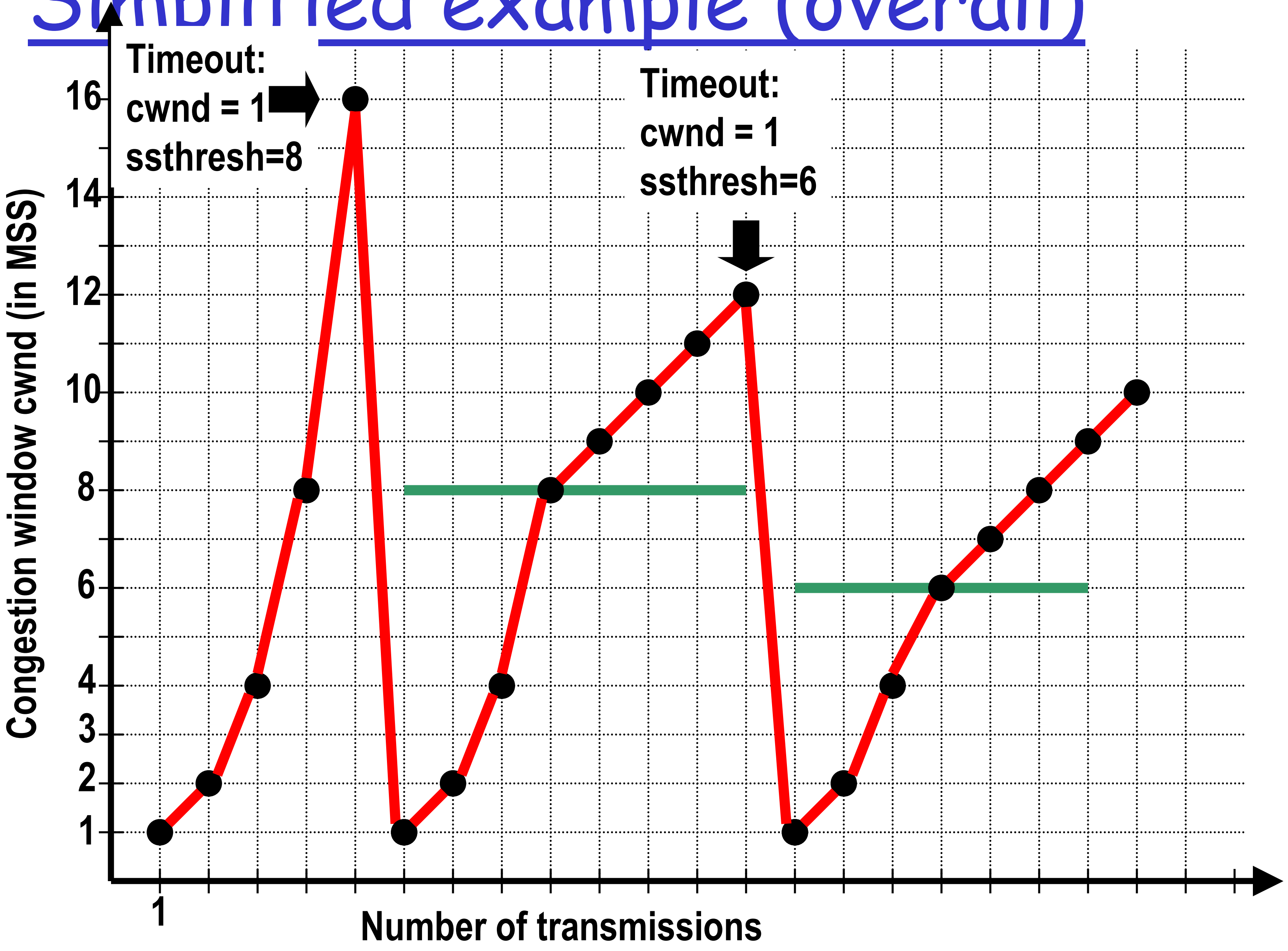
Corrisponde ad un segmento
per finestra

- Which initial ssthresh?

  – ssthresh initially set to 65535: unreachable!

In essence, congestion avoidance is flow control imposed by sender while advertised window is flow control imposed by receiver

Simplified example (overall)


Timeout:
cwnd = 1
ssthresh=8
Timeout:
cwnd = 1
ssthresh=6
Congestion window cwnd (in MSS)
Number of transmissions