

Chapter 3

Transport Layer

Reti di Elaboratori

Corso di Laurea in Informatica

Università degli Studi di Roma "La Sapienza"

Prof.ssa Chiara Petrioli

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

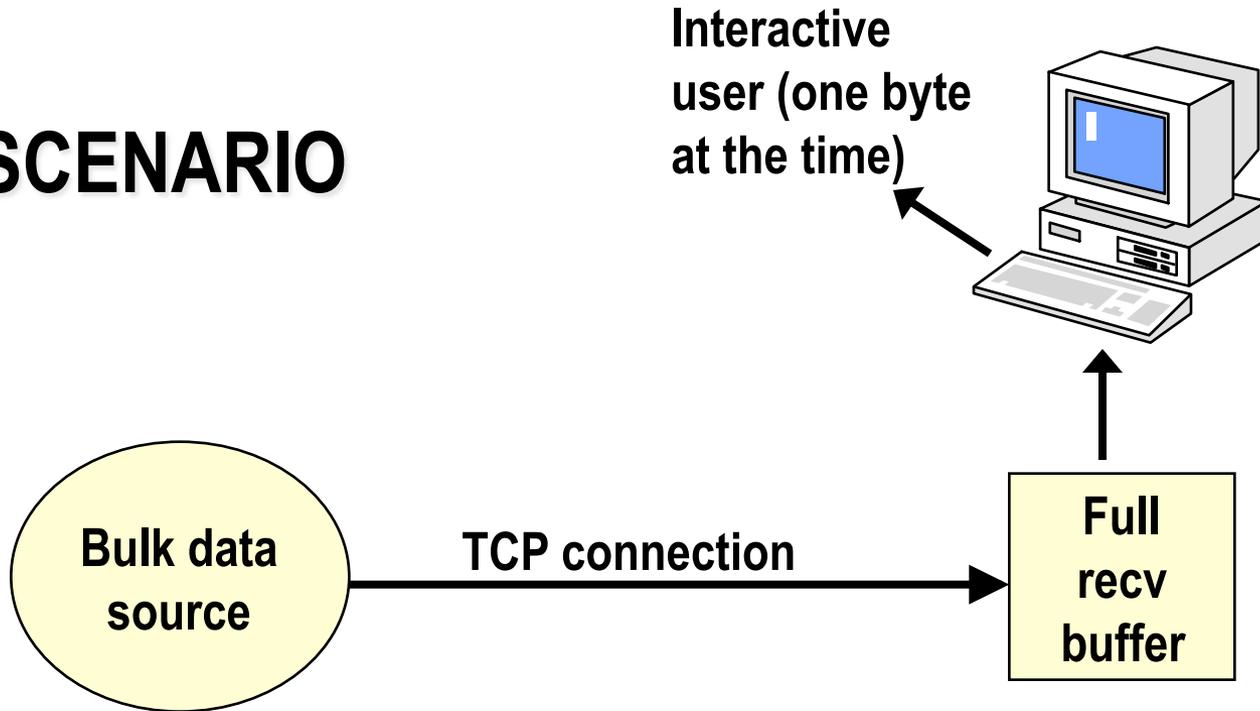
All material copyright 1996-2009

J.F Kurose and K.W. Ross, All Rights Reserved

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

The silly window syndrome

SCENARIO



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:

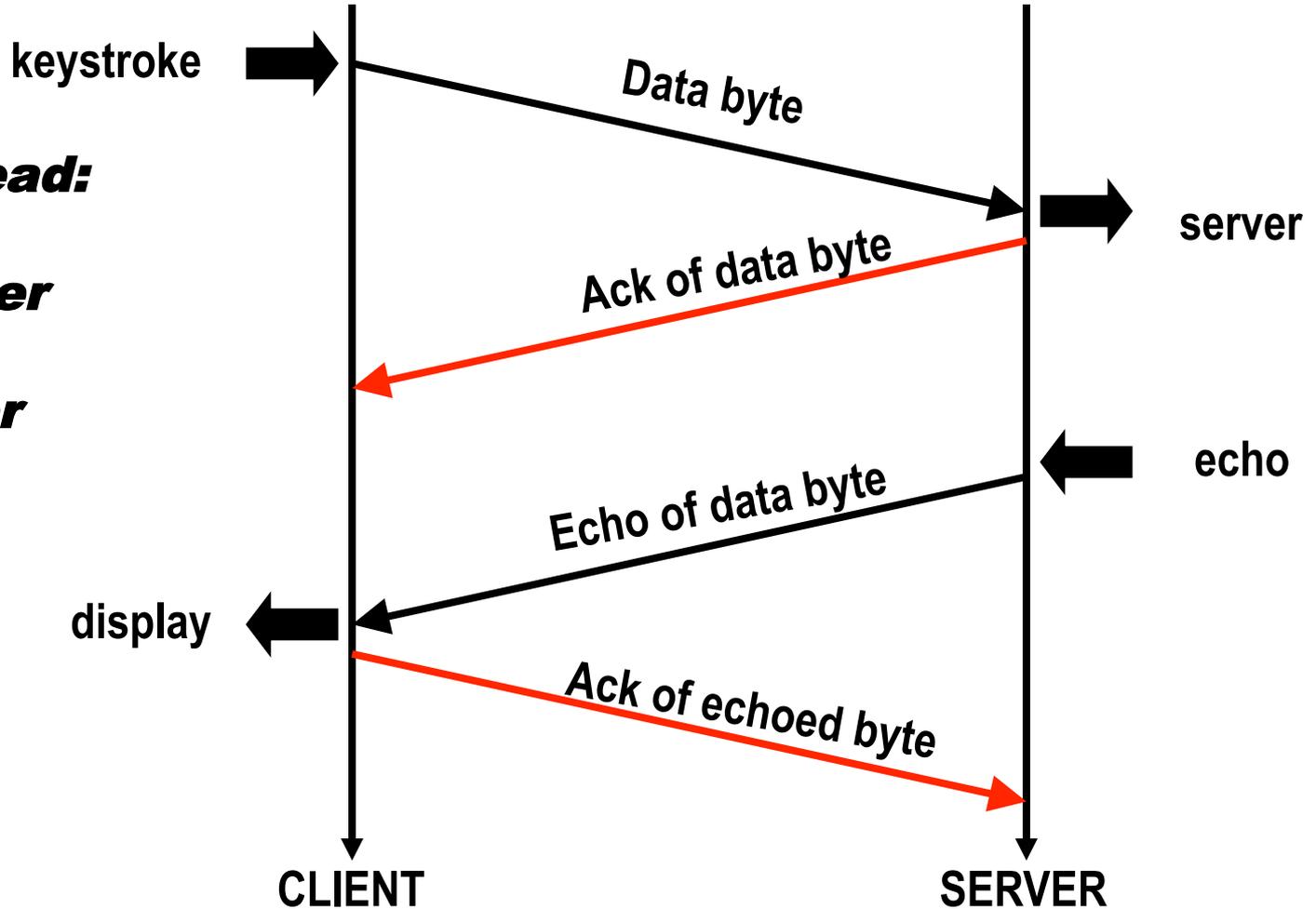
20 TCP header

+

20 IP header

+

1 data



Interactive apps: create some tricky situations....

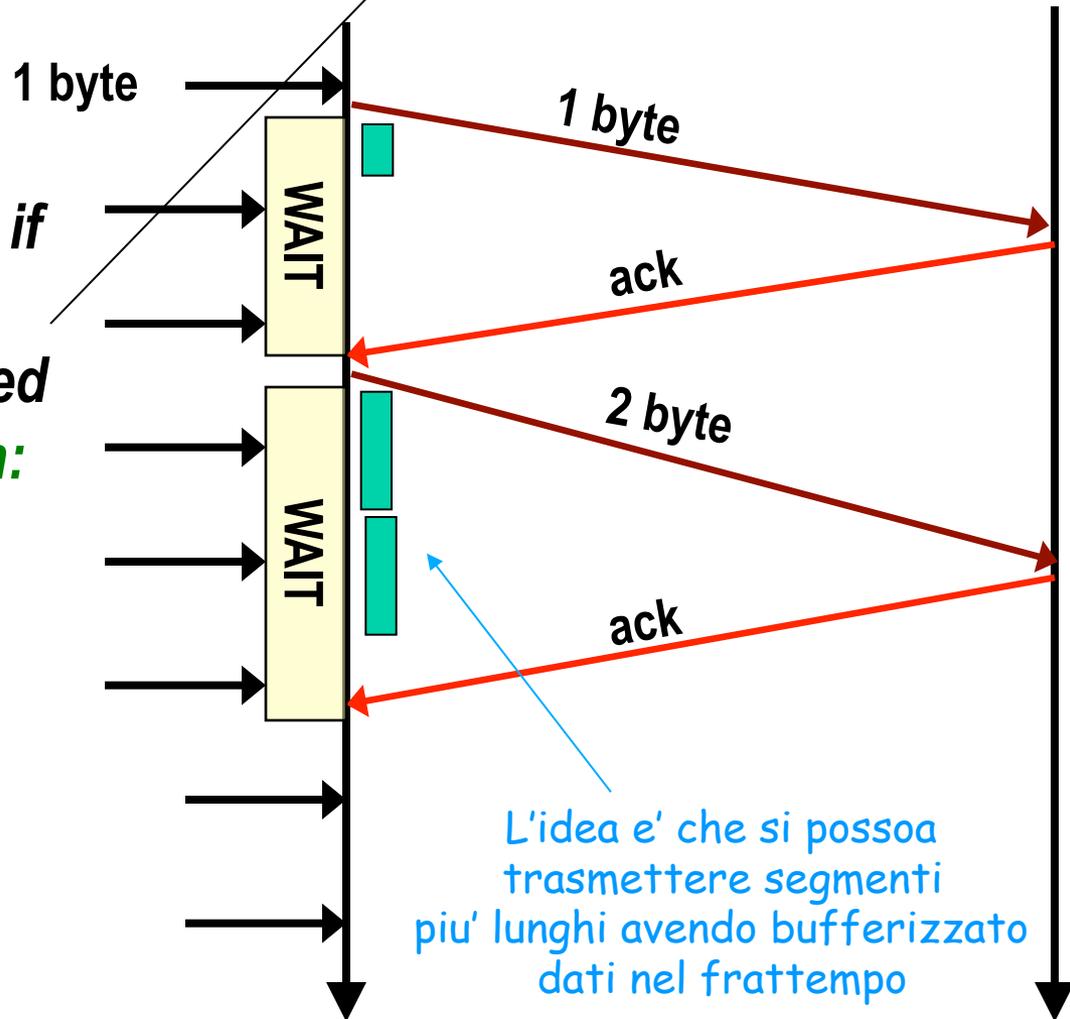
Nagle's algorithm (RFC 896, 1984)

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



PUSH flag

Source port			Destination port						
32 bit Sequence number									
32 bit acknowledgement number									
Header length	6 bit Reserved	U R G	A C K	P S H	R S T	S Y N	F I N	Window size	
checksum					Urgent pointer				

□ Used to notify

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

Urgent data

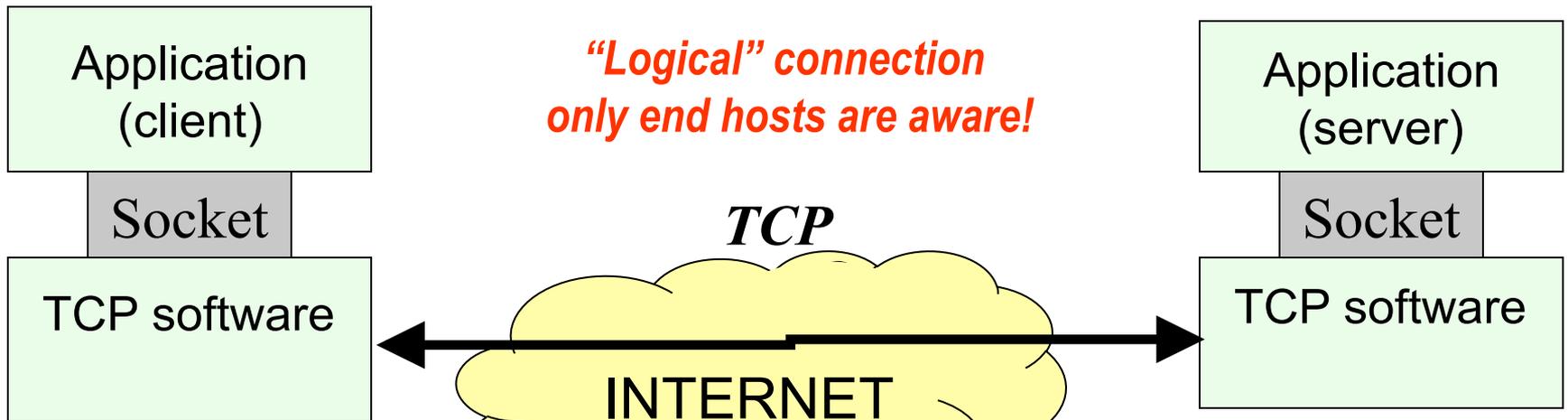
Source port				Destination port					
32 bit Sequence number									
32 bit acknowledgement number									
Header length	6 bit Reserved	U R G	A C K	P S H	R S T	S Y N	F I N	Window size	
checksum				Urgent pointer					

- ❑ 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



State variables:

- conn status
- MSS
- windows
- ...

buffer space

normally 4 to 16 Kbytes
64+ Kbytes possible

Connection described by client&server status

Connection SET-UP duty:

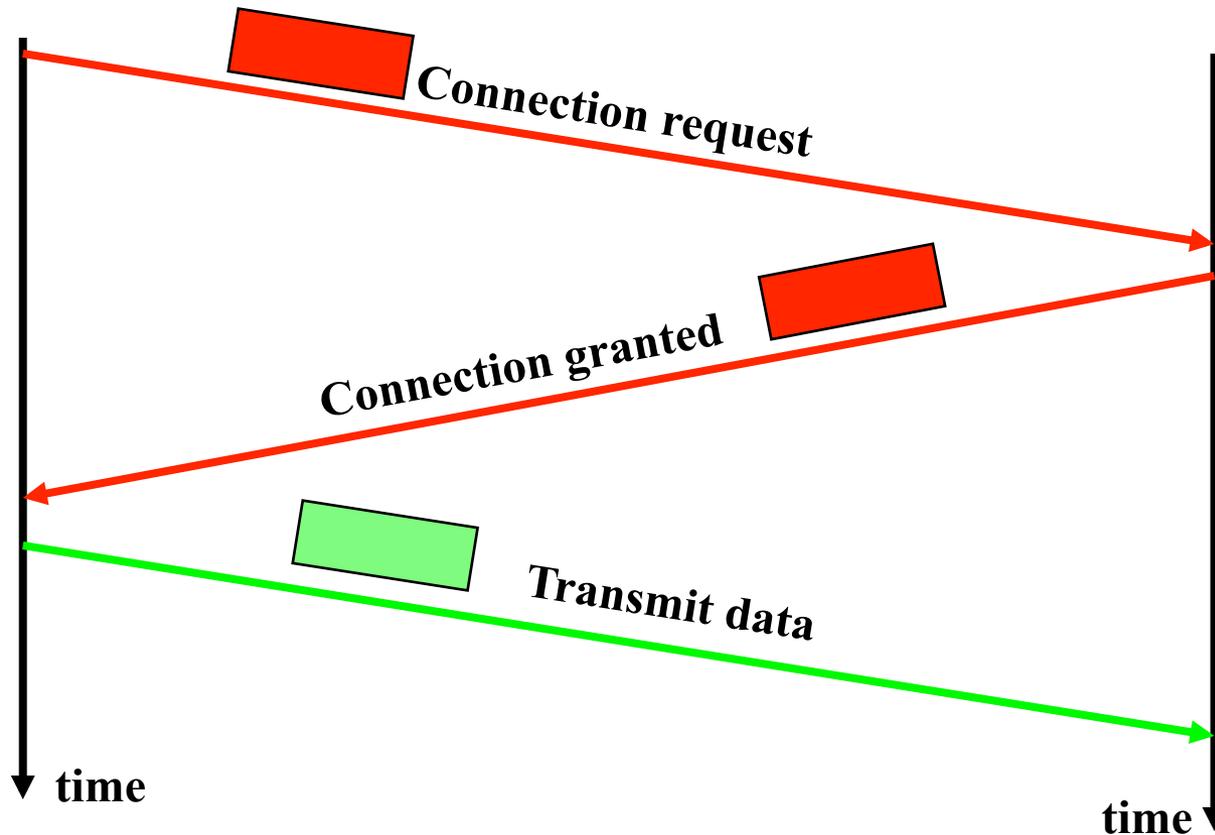
1) initializes state variables

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

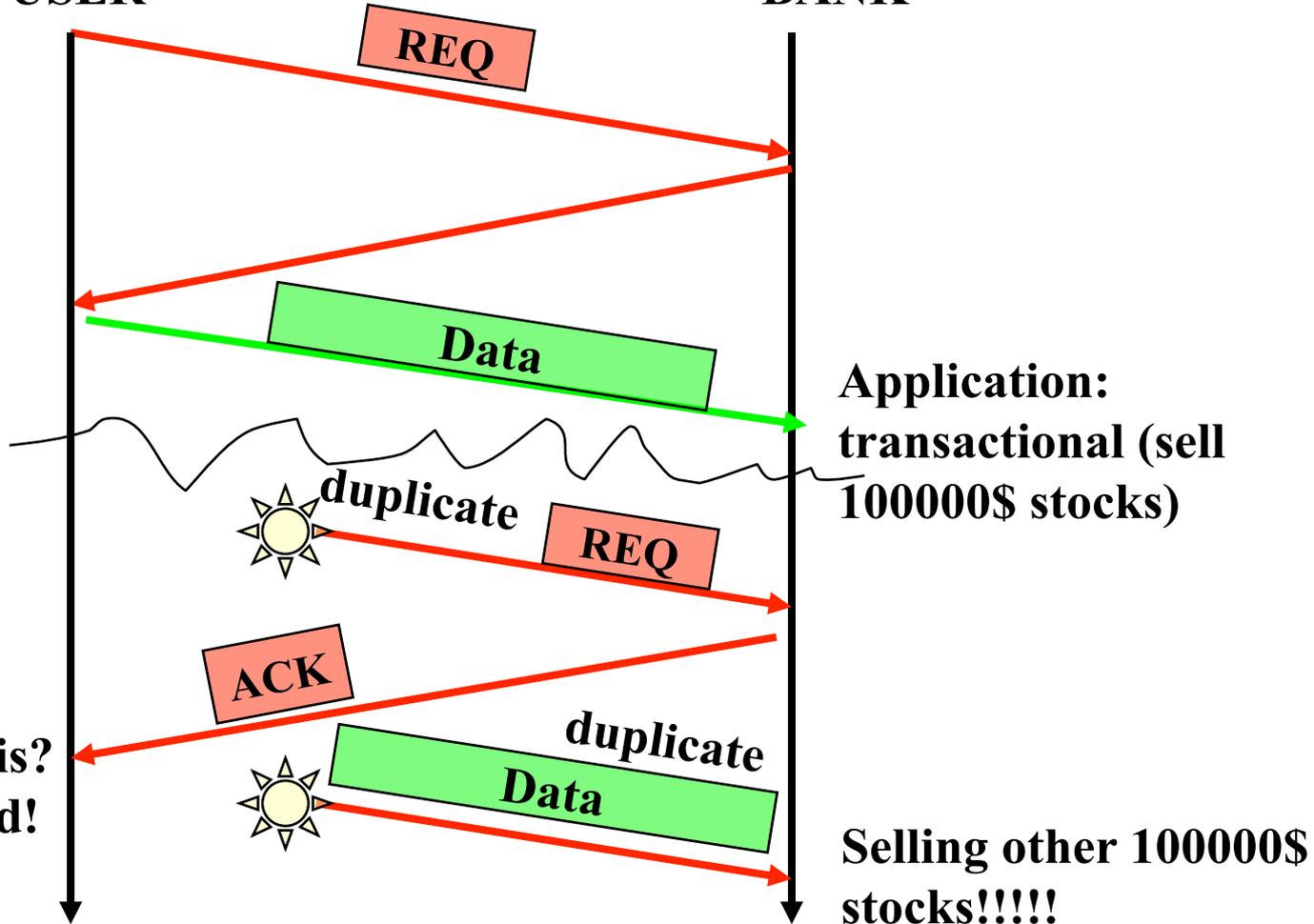
Connection establishment: simplest approach (non TCP)



Delayed duplicate problem

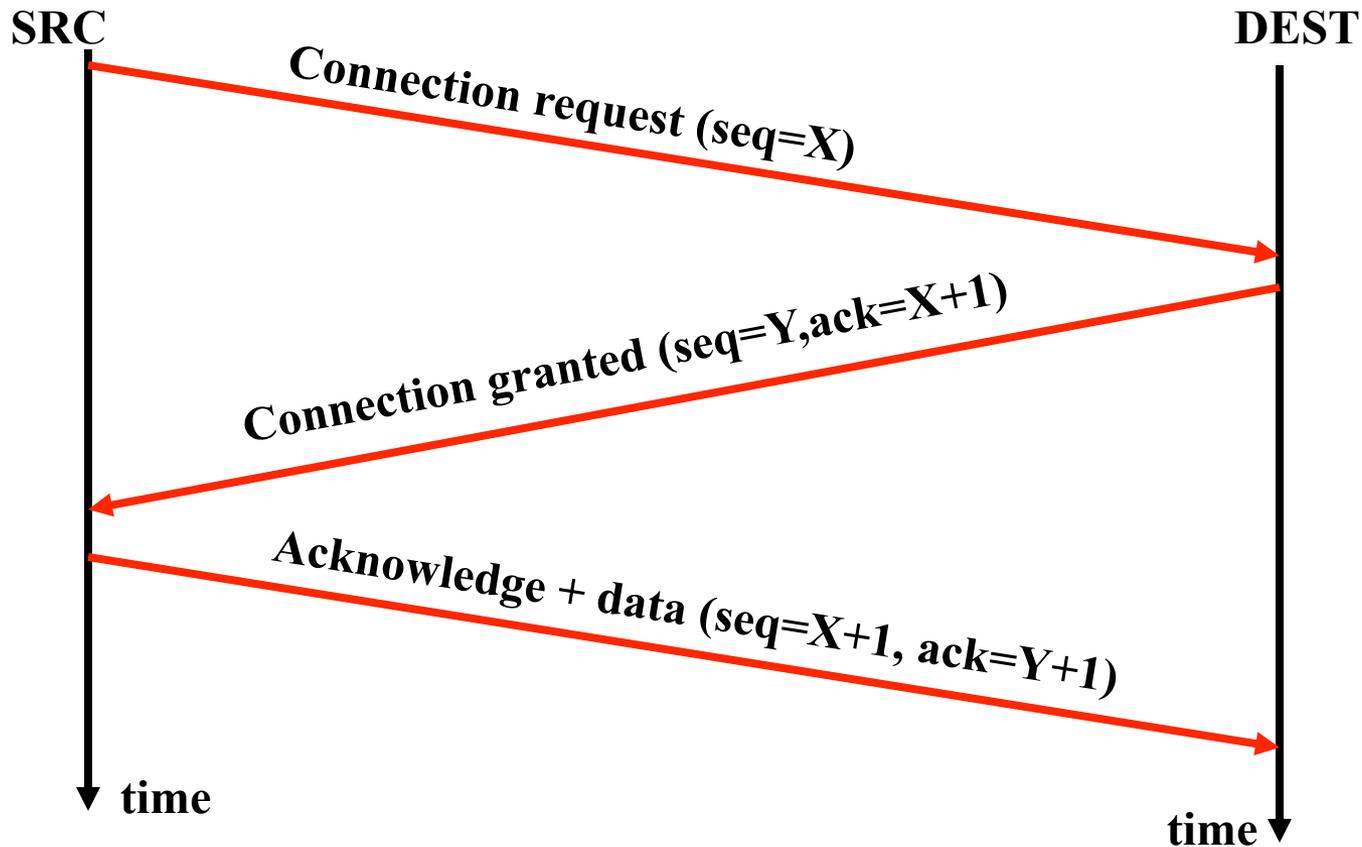
USER

BANK

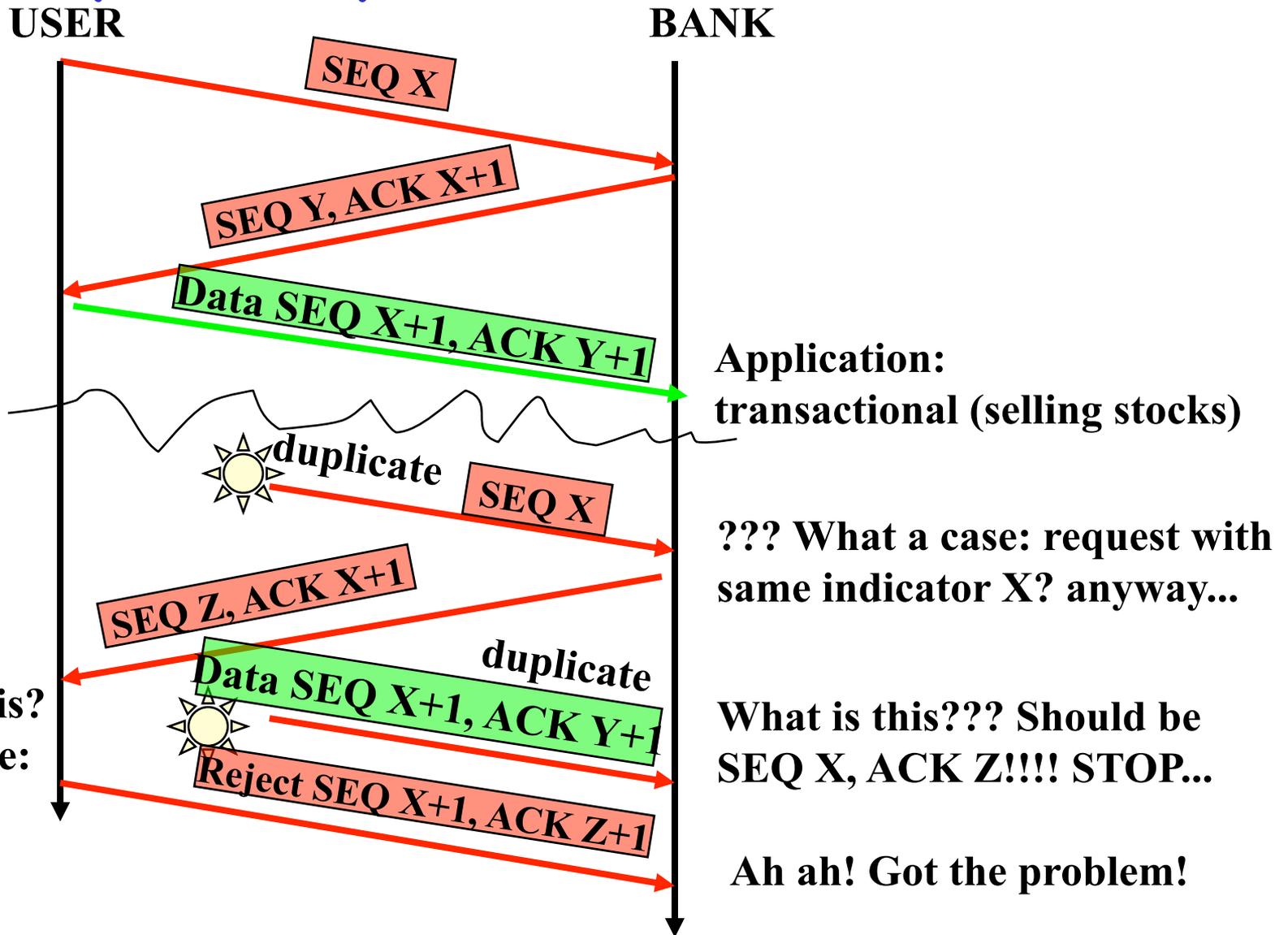


Solution: three way handshake

Tomlinson 1975



Delayed duplicate detection



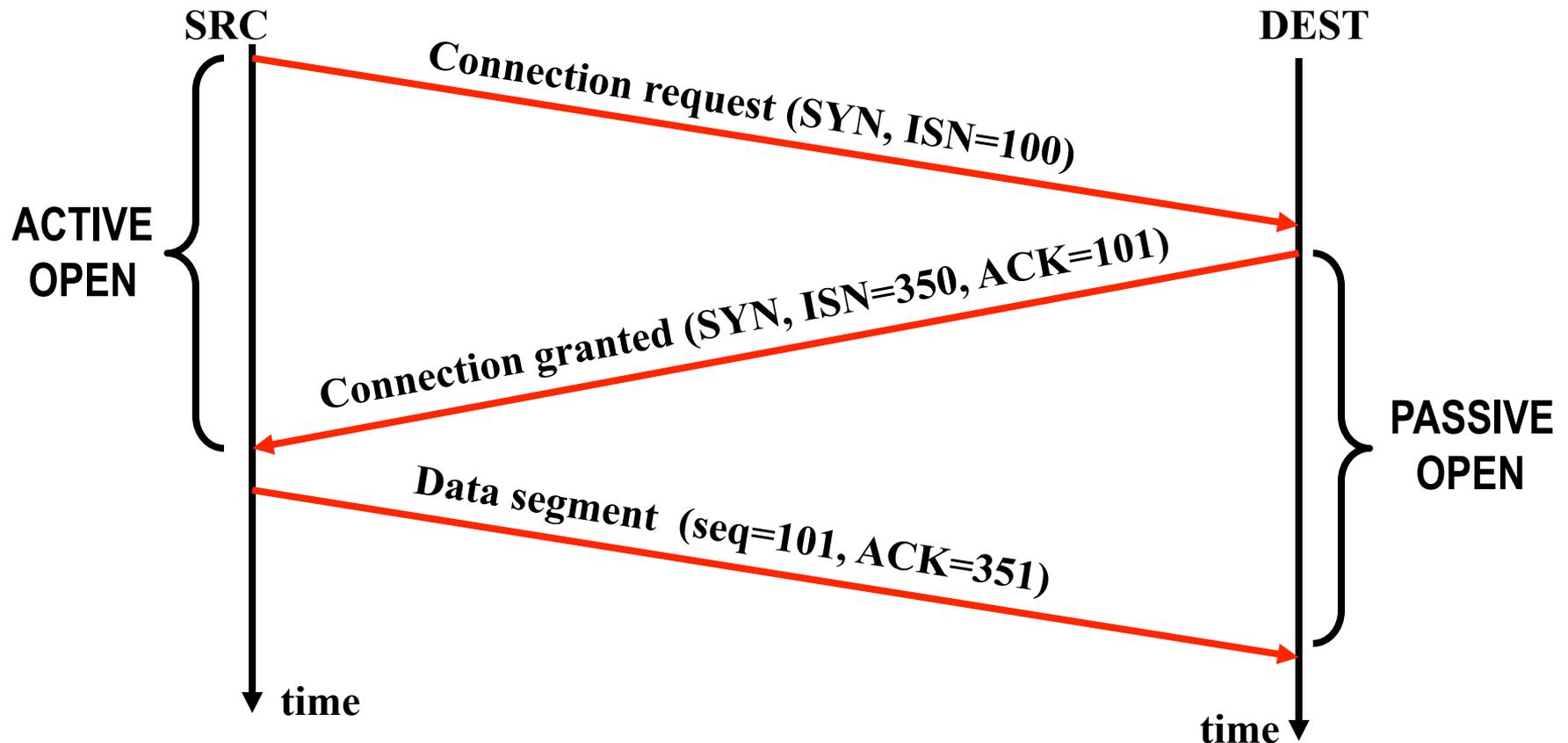
Disaster could not be avoided with a two-way handshake

Source port				Destination port				
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	U R G	A C K	P S H	R S T	S Y N	F I N	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



Full duplex connection: opened in both ways

SRC: performs ACTIVE OPEN

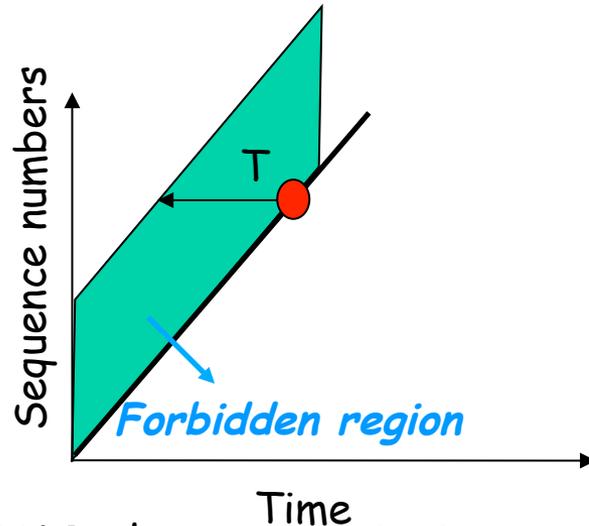
DEST: Performs PASSIVE OPEN

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 $4\mu\text{s}$ 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

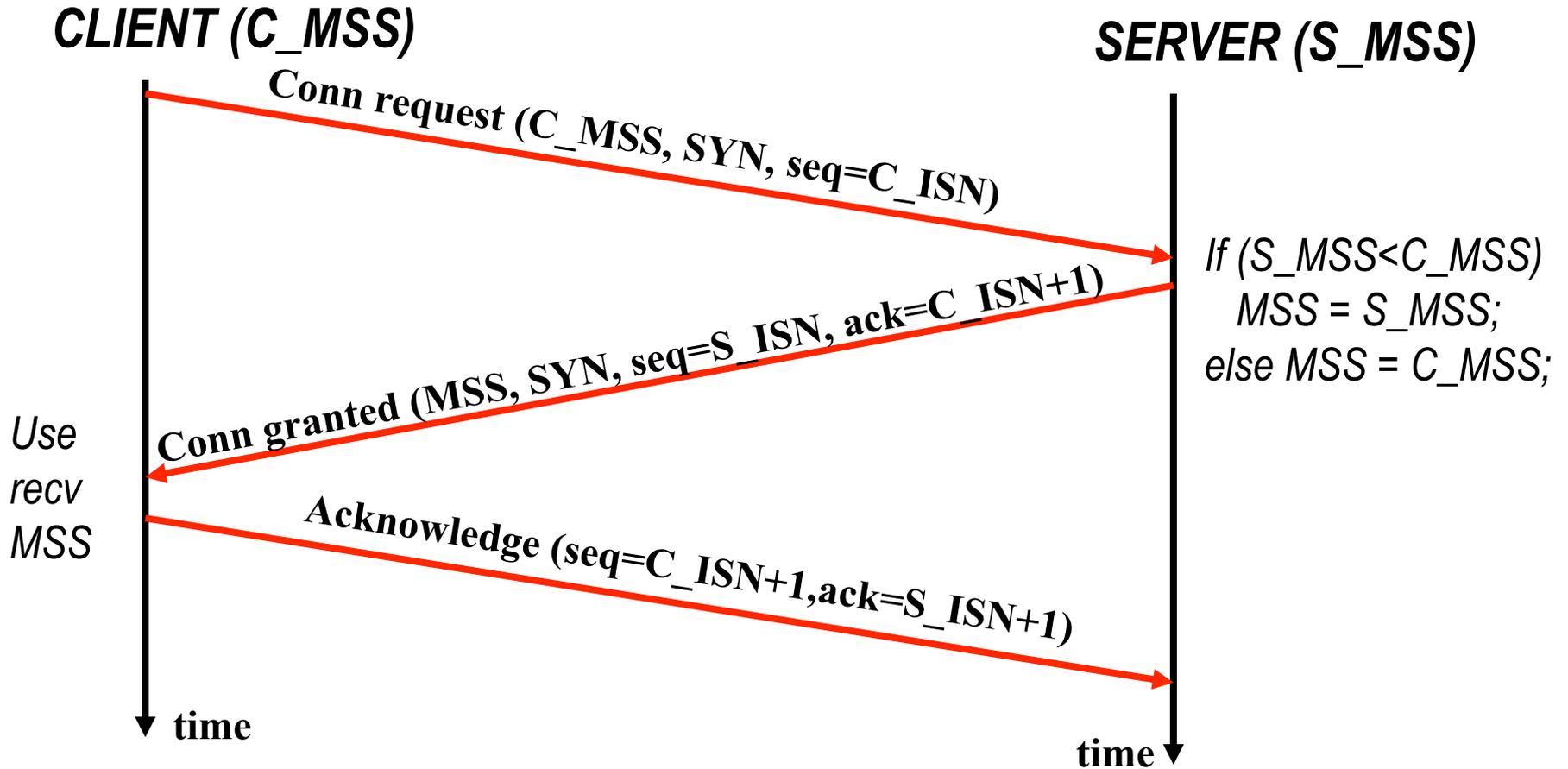


- Aging dei pacchetti → dopo un certo tempo MSL (Maximum Segment Lifetime) i pacchetti eliminati dalla rete
- Initial sequence numbers basati sul clock
- Un ciclo del clock circa 4 ore; MSL circa 2 minuti.
- → Se non ci sono crash che fanno perdere il valore dell'ultimo initial sequence number usato 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).

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



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

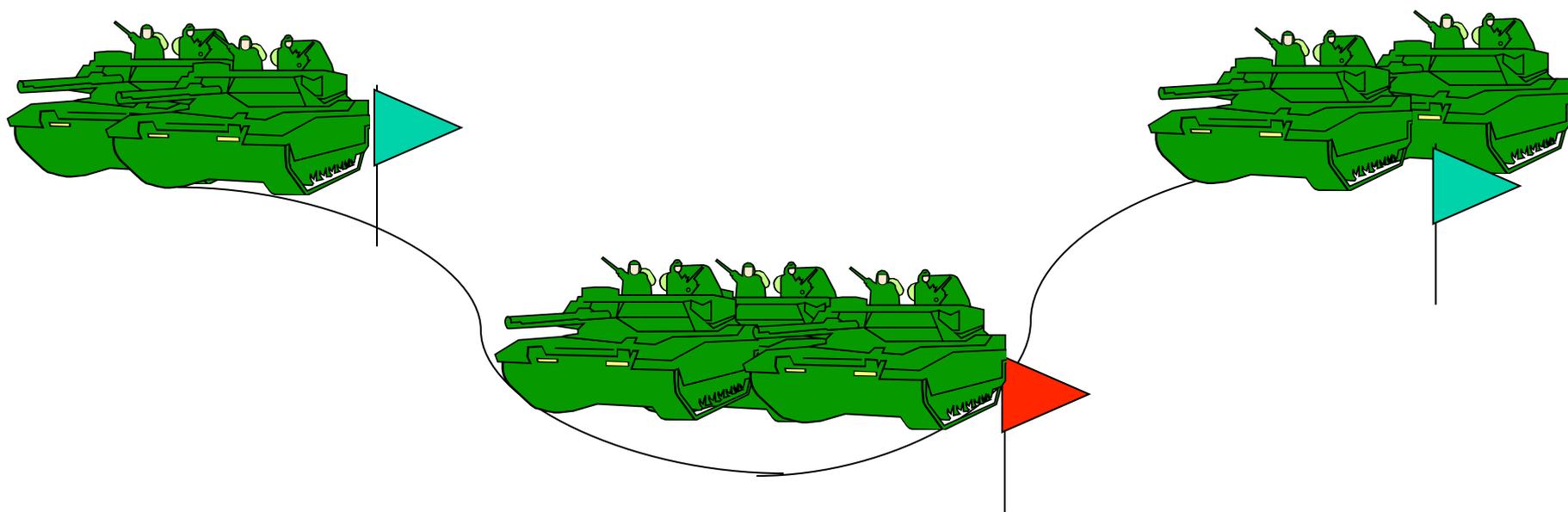
- server allocates buffers
- specifies server initial seq. #

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

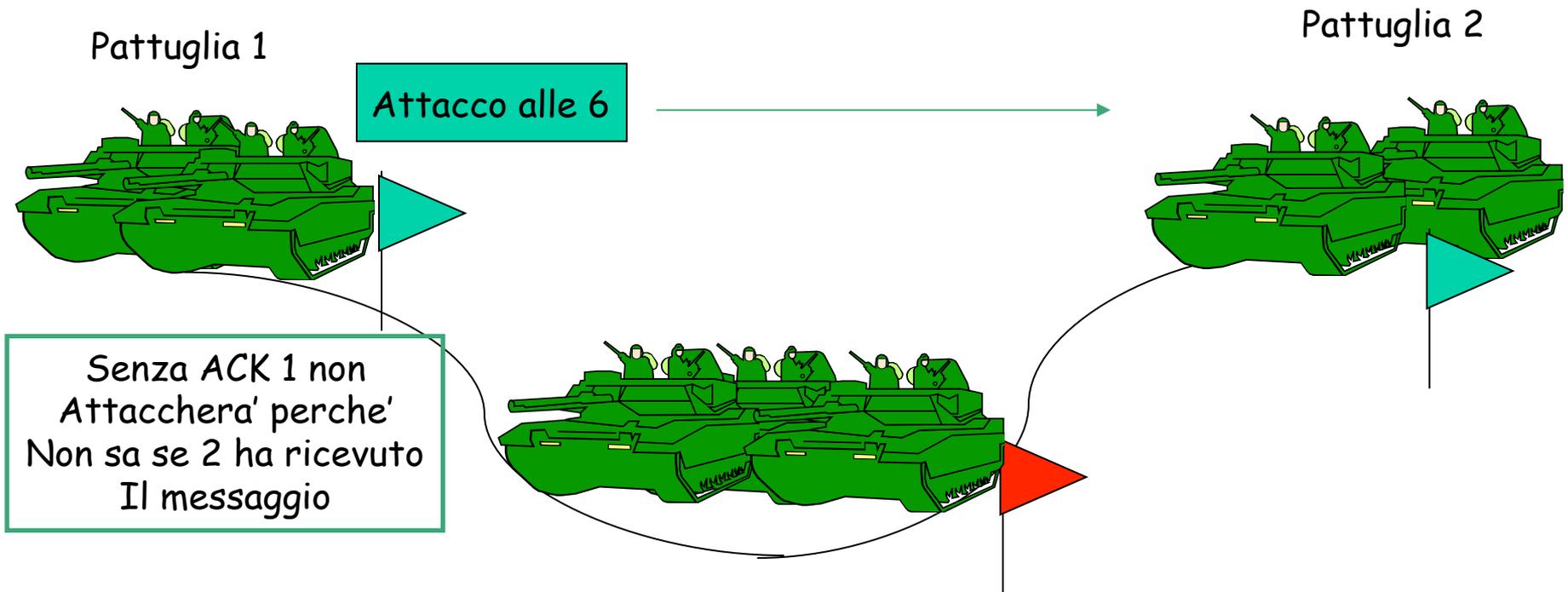
Problema dei due eserciti

- 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 però essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



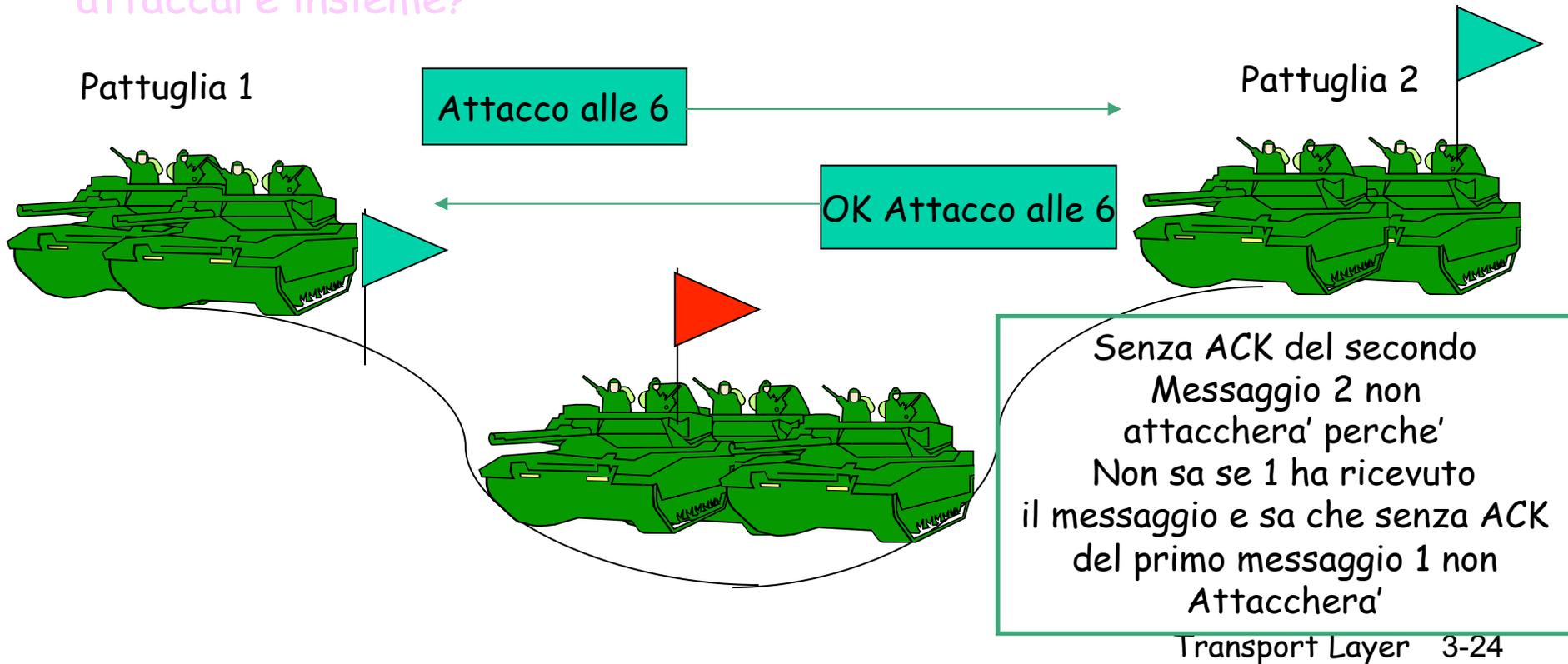
Problema dei due eserciti

- 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 però essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



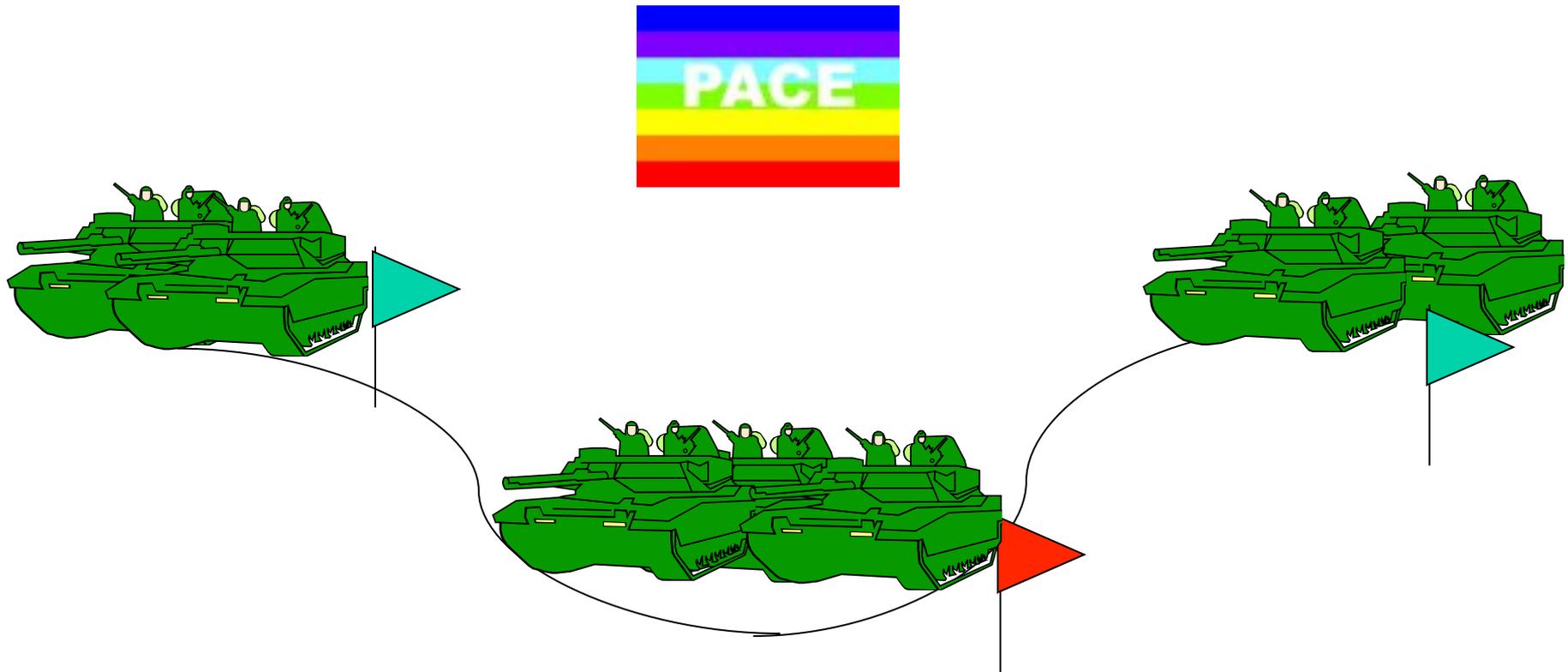
Problema dei due eserciti

- 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 però essere catturati e quindi il messaggio può non arrivare correttamente a destinazione. Come fanno a mettersi d'accordo per attaccare insieme?



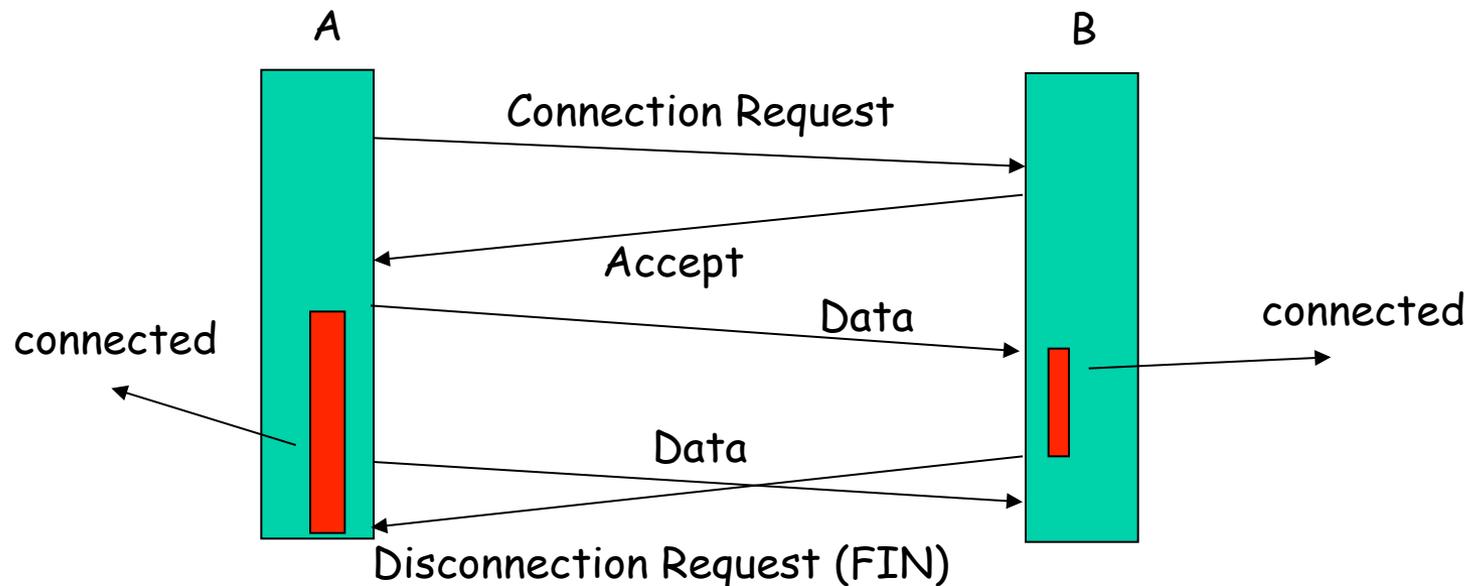
Problema dei due eserciti

- 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??

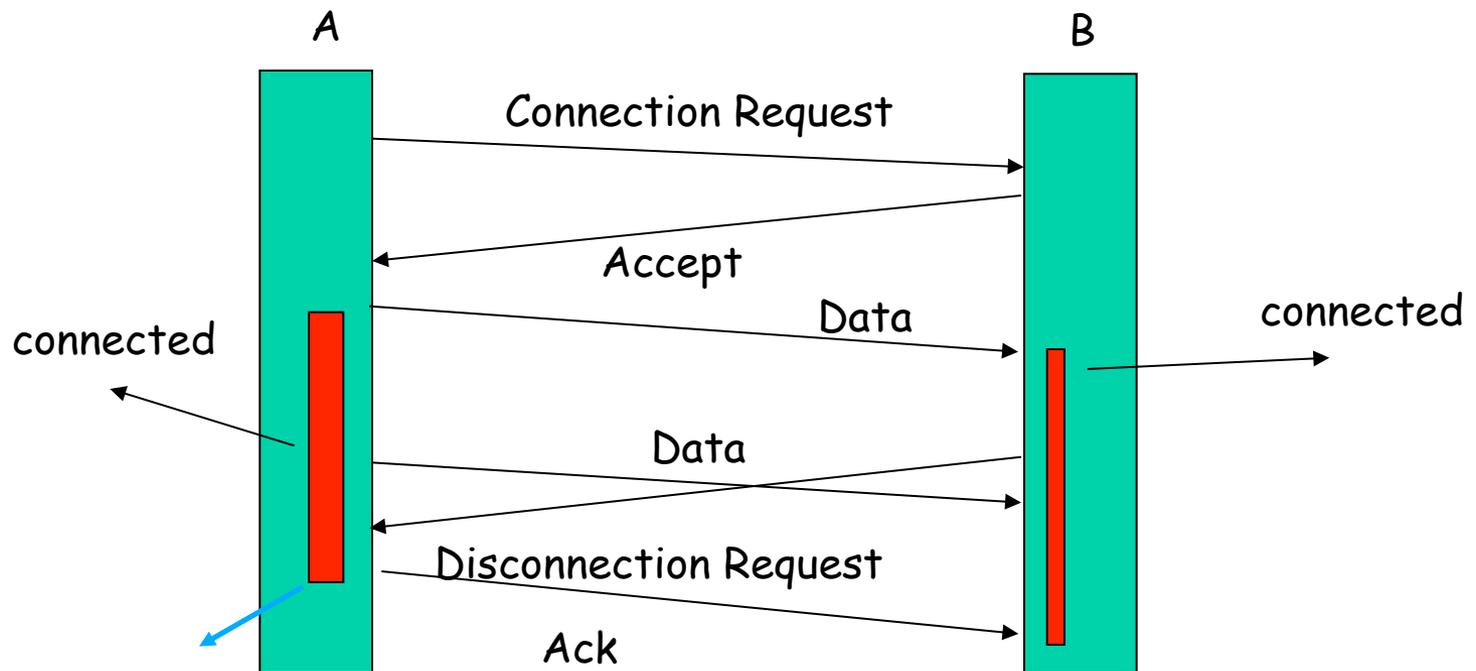
- Chiusura di una connessione. Vorremmo un **accordo** tra le due peer entity o rischiamo di perdere dati.



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 di 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 problem use simple solution:
two way handshake

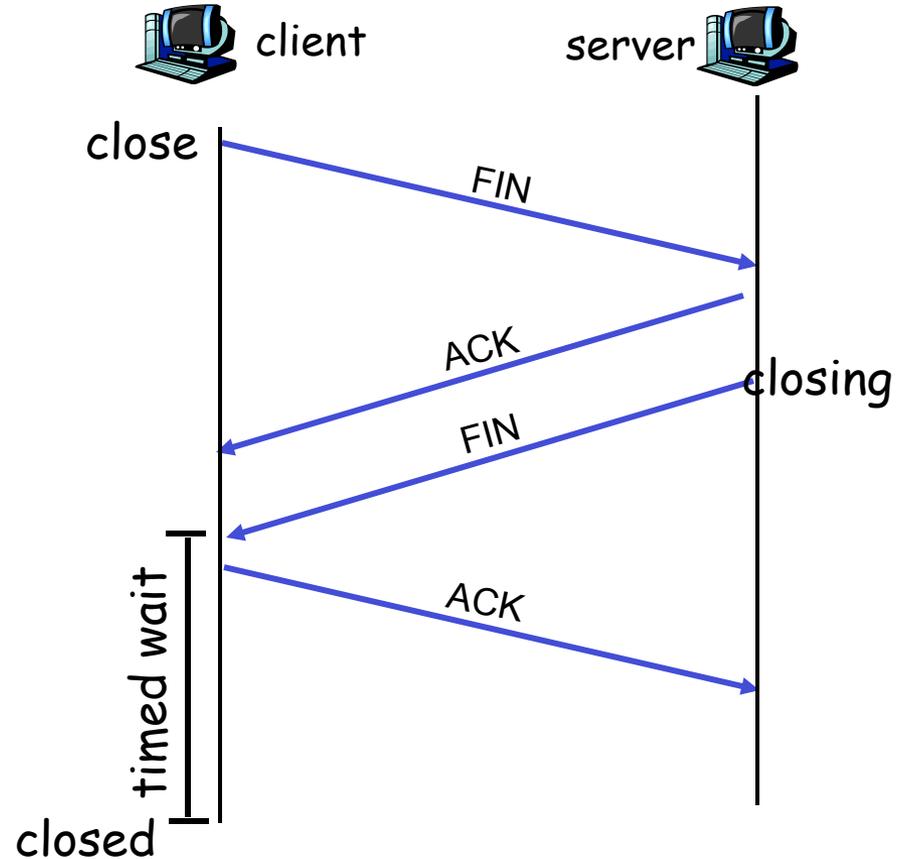
Closing a connection:

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.

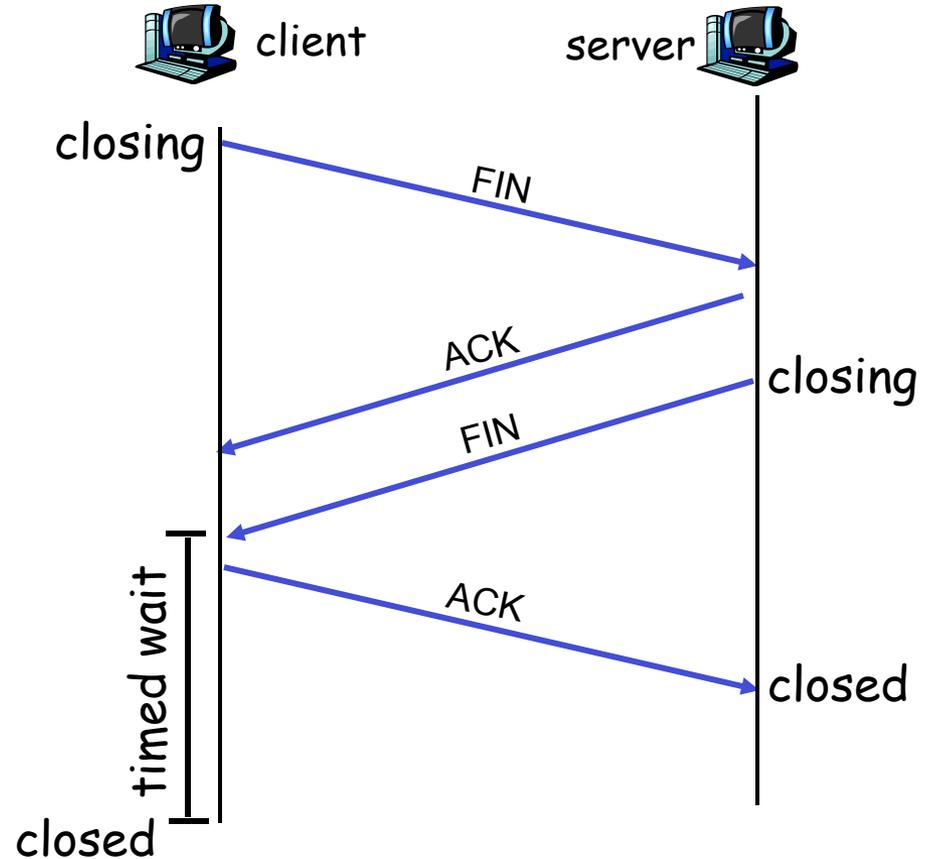


TCP Connection Management (cont.)

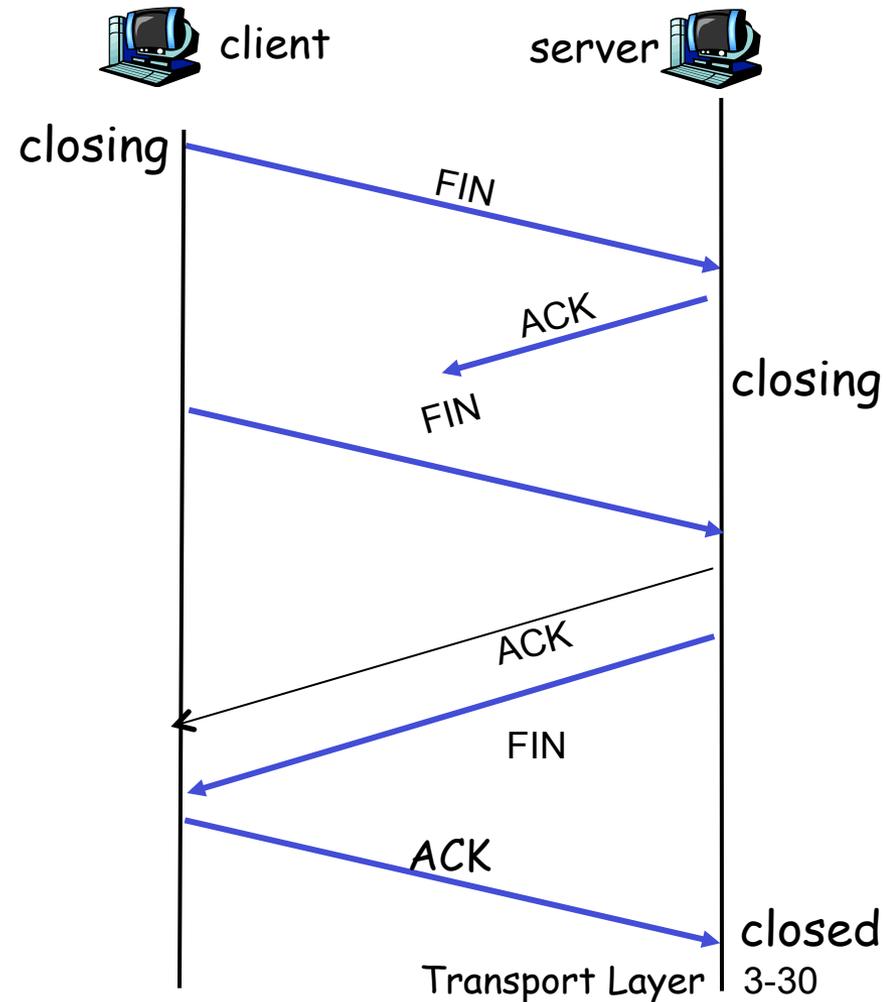
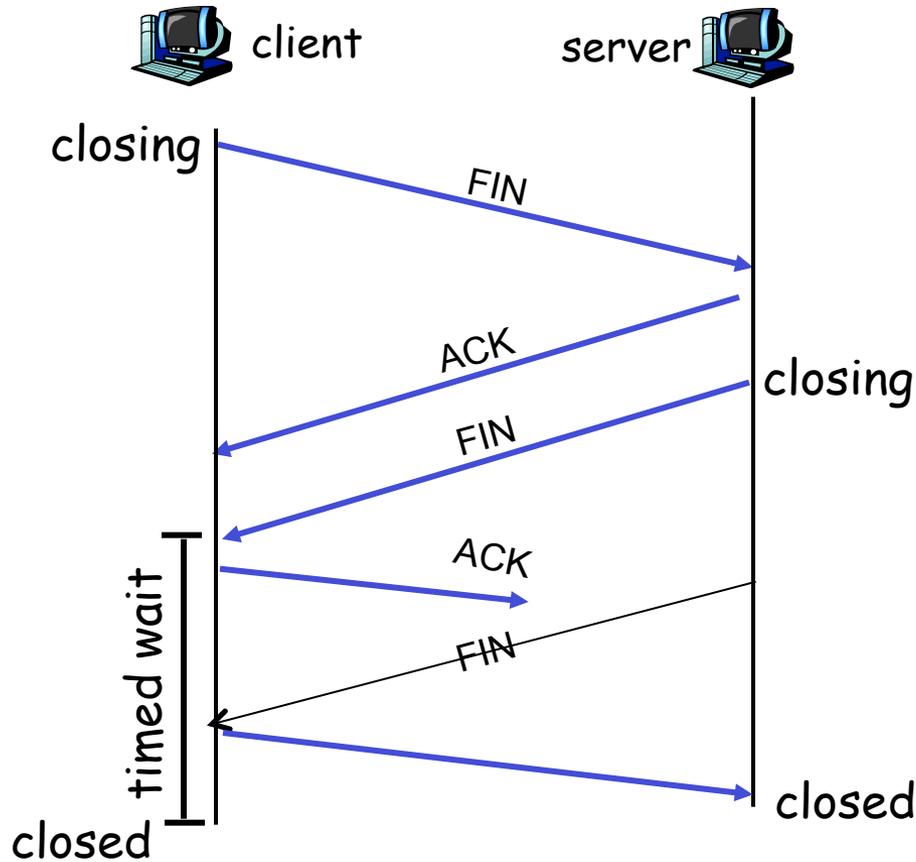
Step 3: client receives FIN, replies with ACK.

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

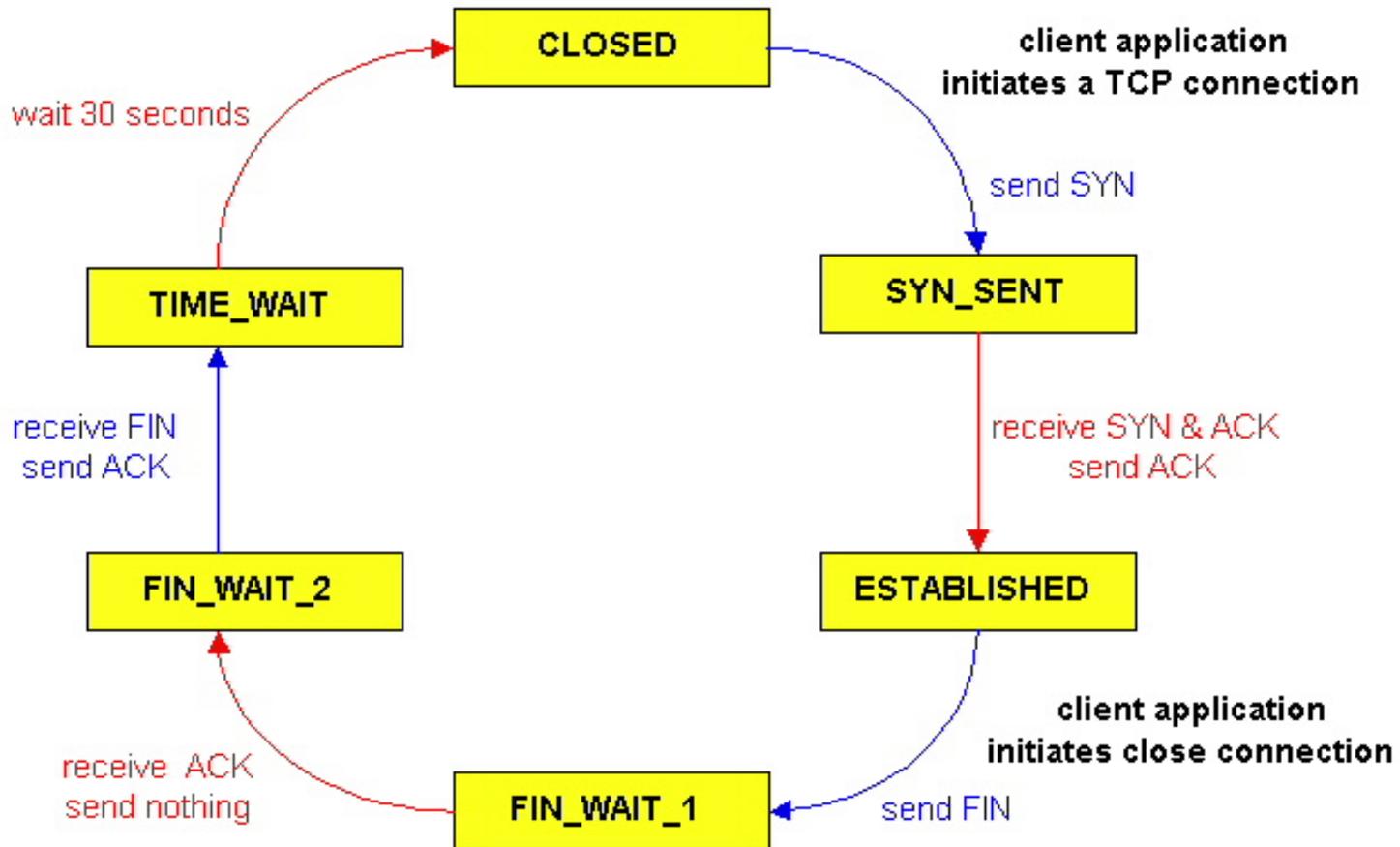
Step 4: server, receives ACK. Connection closed.



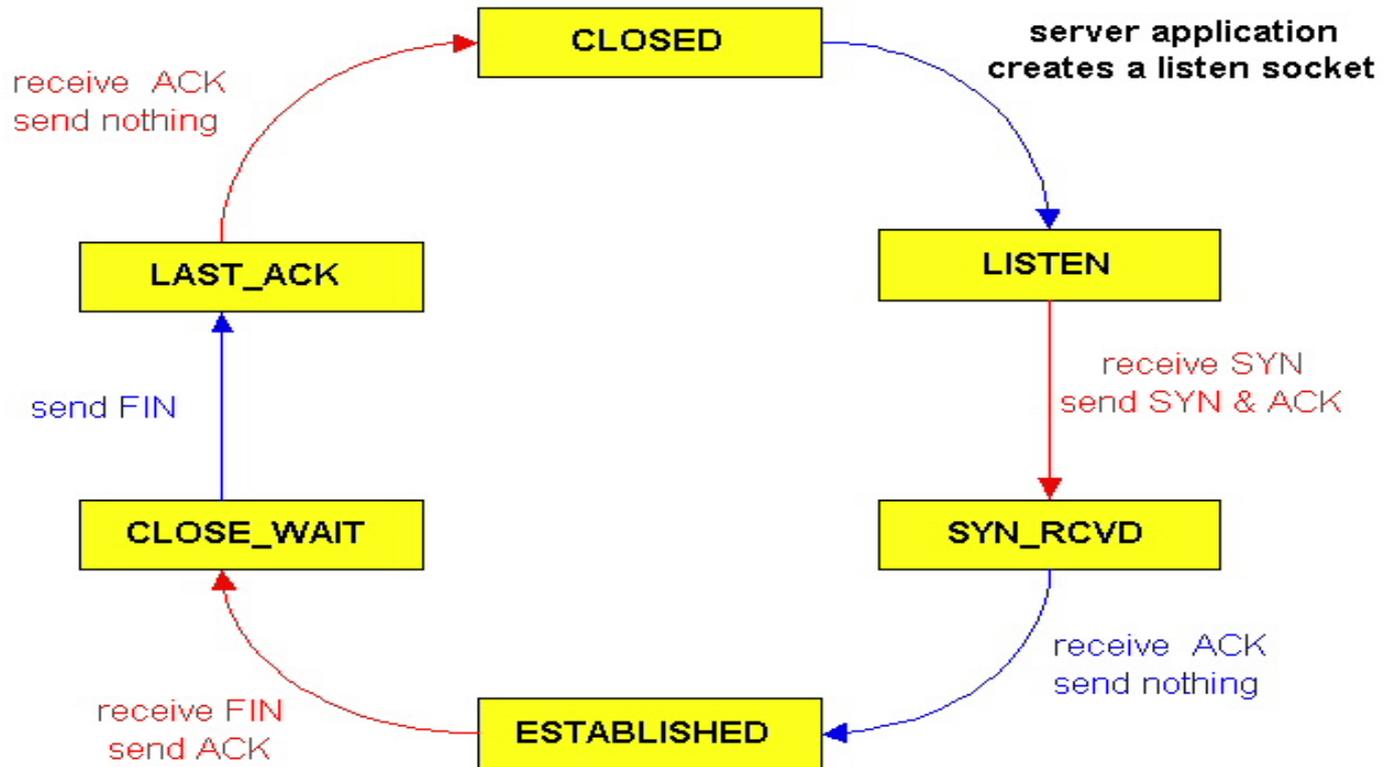
TCP Connection Management (examples)



Connection states - Client



Connection States - Server



Source port				Destination port				
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	U R G	A C K	P S H	R S T	S Y N	F I N	Window size
checksum				Urgent pointer				

- ❑ 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

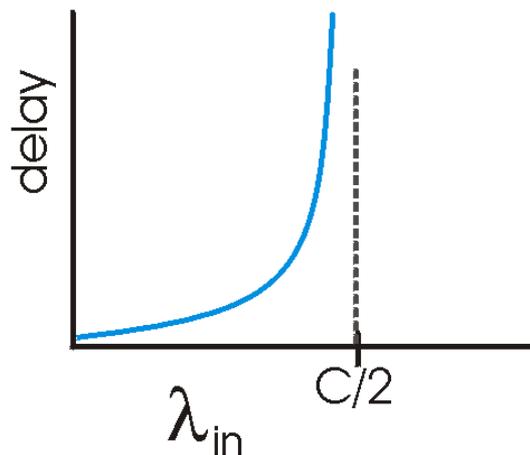
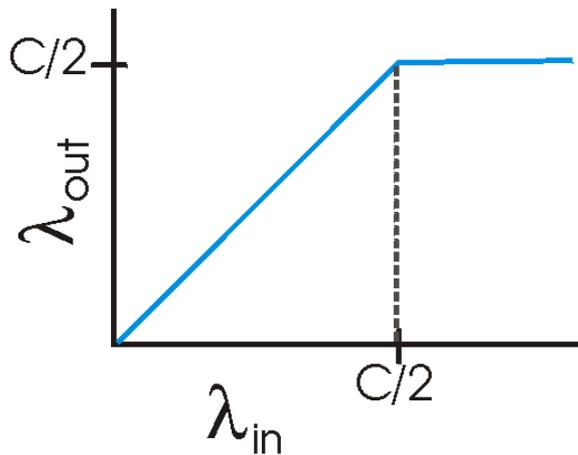
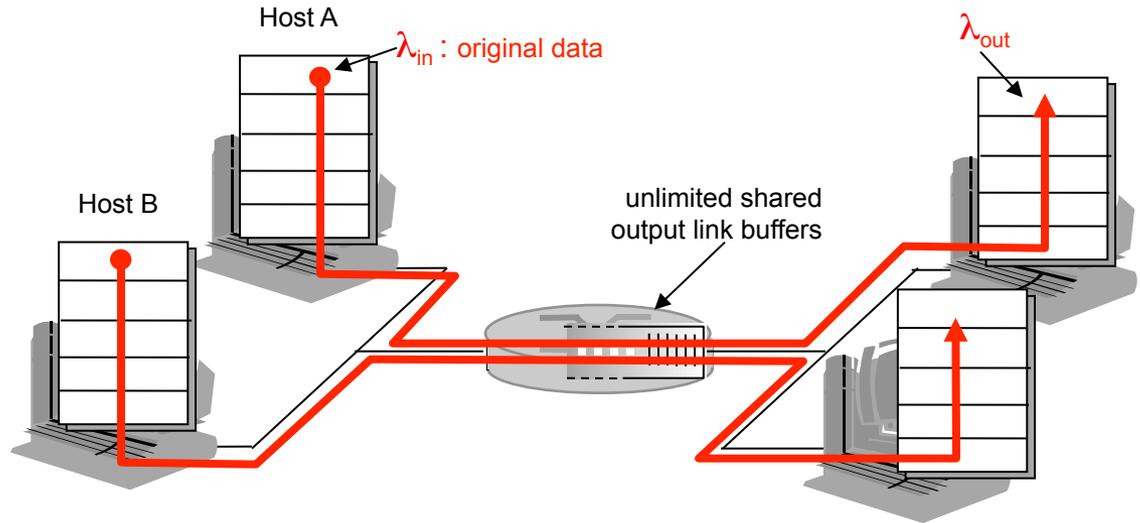
Principles of Congestion Control

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

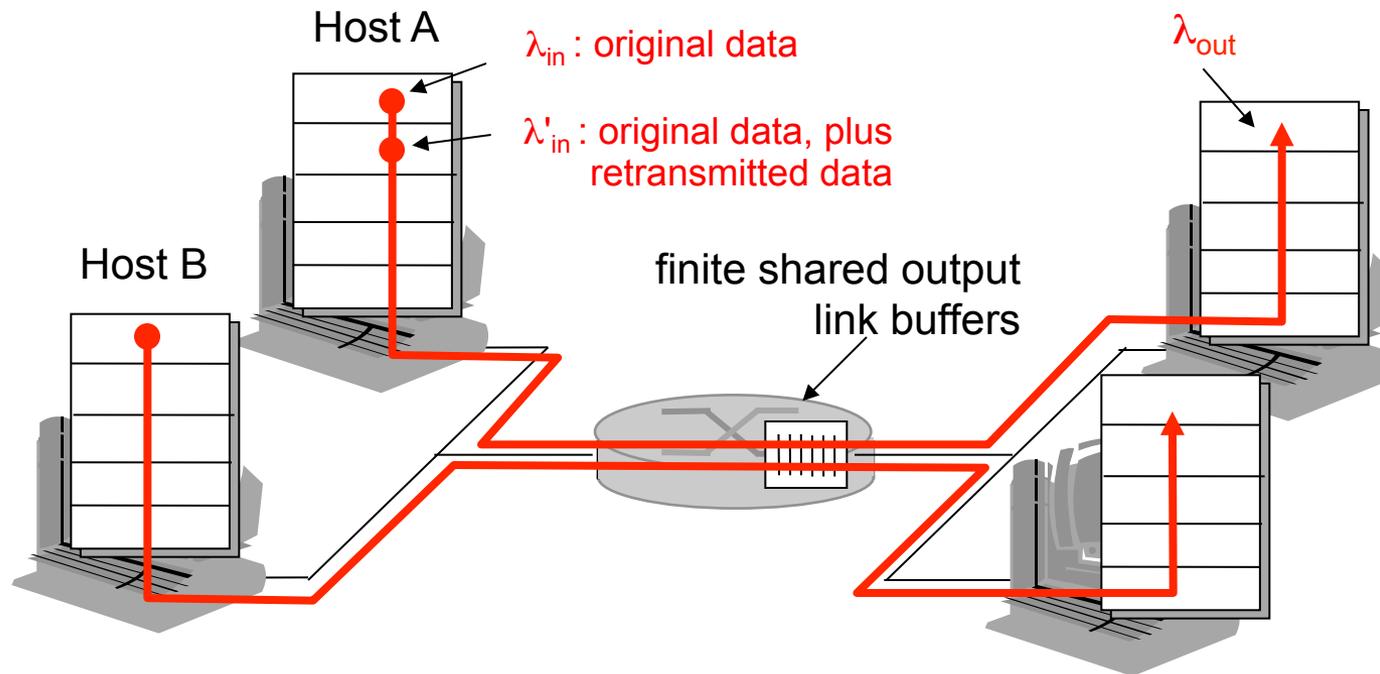
- ❑ two senders, two receivers
- ❑ one router, infinite buffers
- ❑ no retransmission



- ❑ large delays when congested
- ❑ maximum achievable throughput

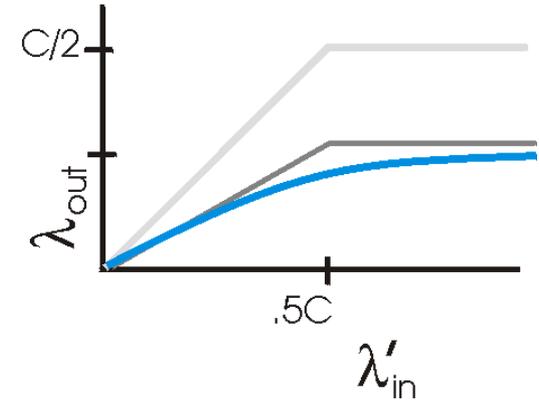
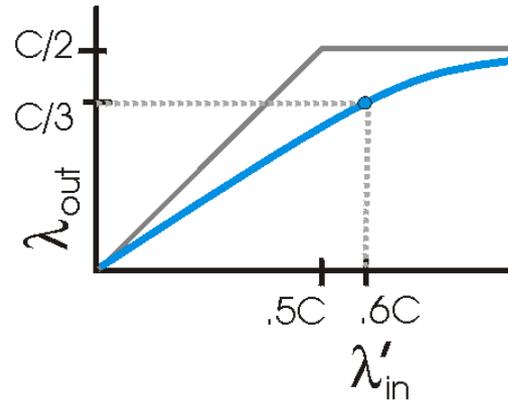
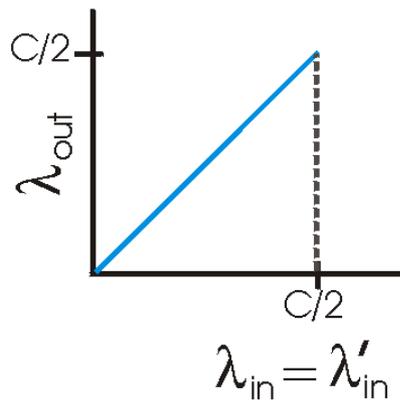
Causes/costs of congestion: scenario 2

- ❑ 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'_{in} larger (than second case) for same λ_{out}



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

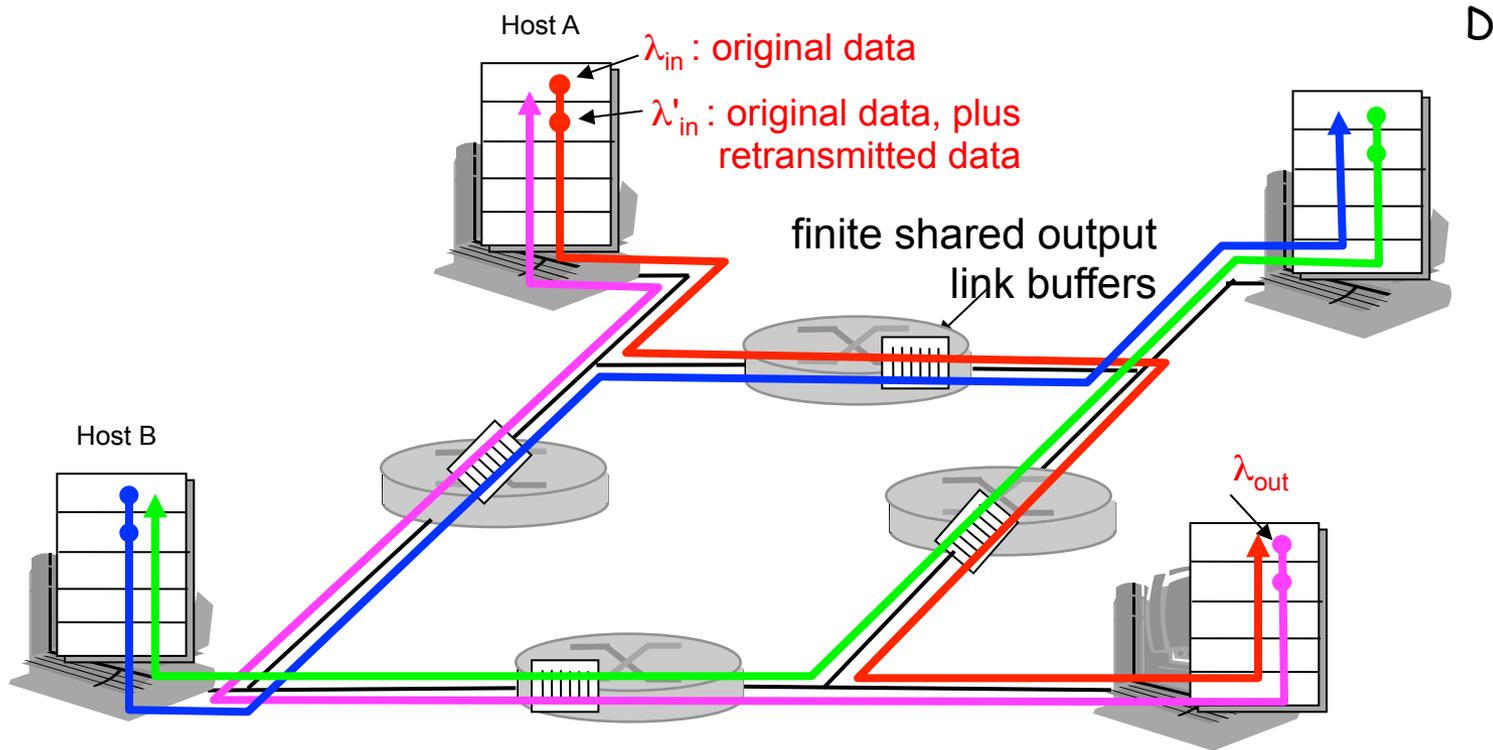
“costs” of congestion:

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

Causes/costs of congestion: scenario 3

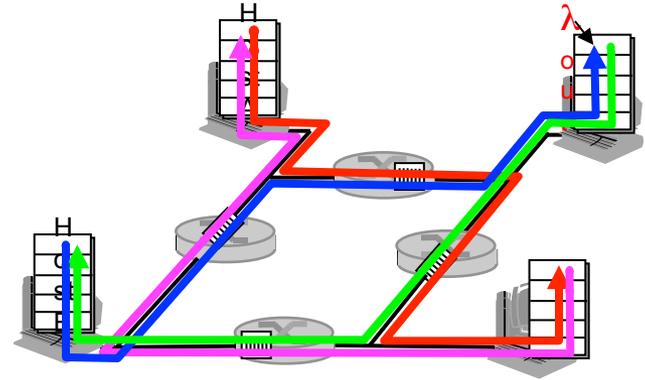
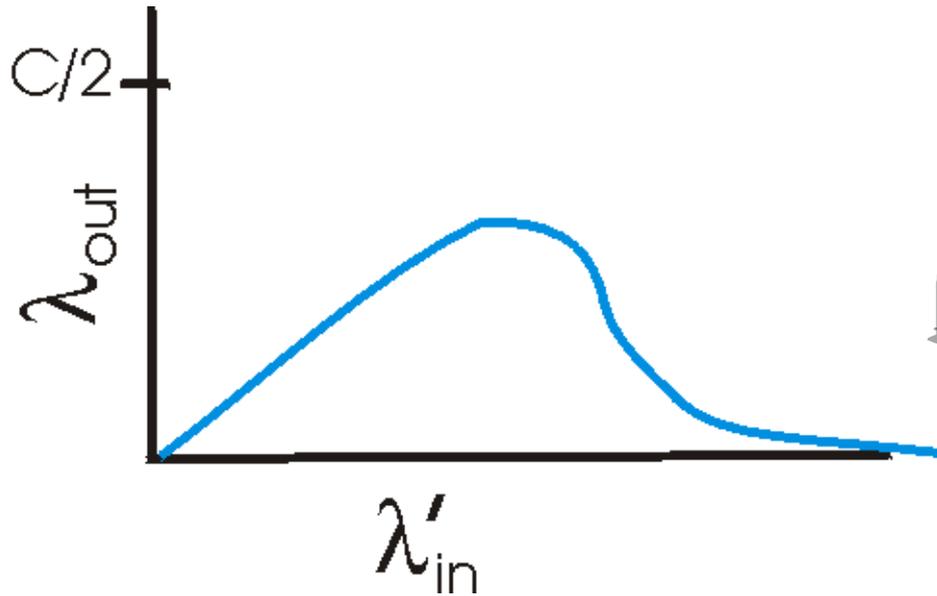
- ❑ four senders
- ❑ multihop paths
- ❑ timeout/retransmit

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 sender should send at