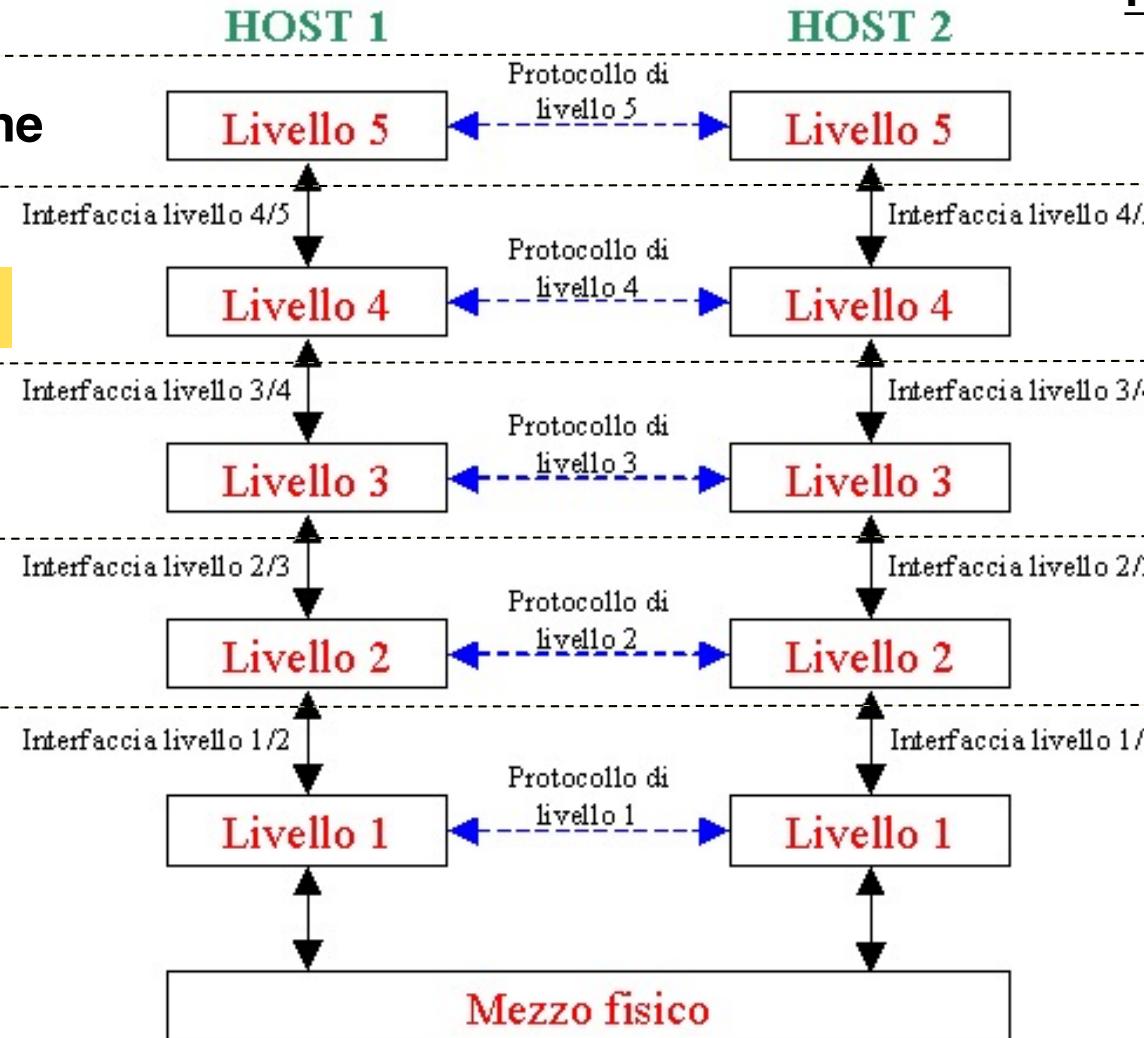
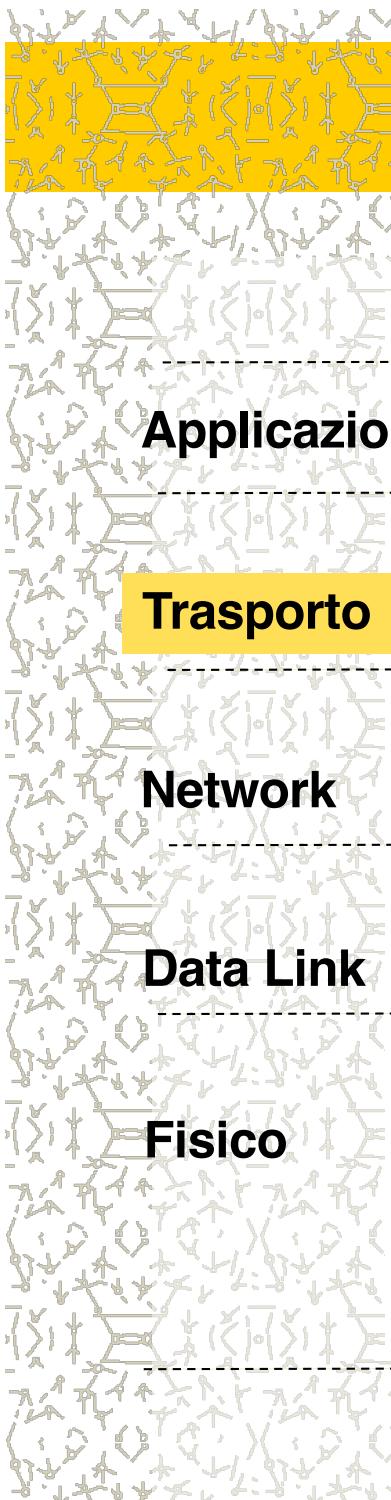


Reti di Calcolatori

Transport

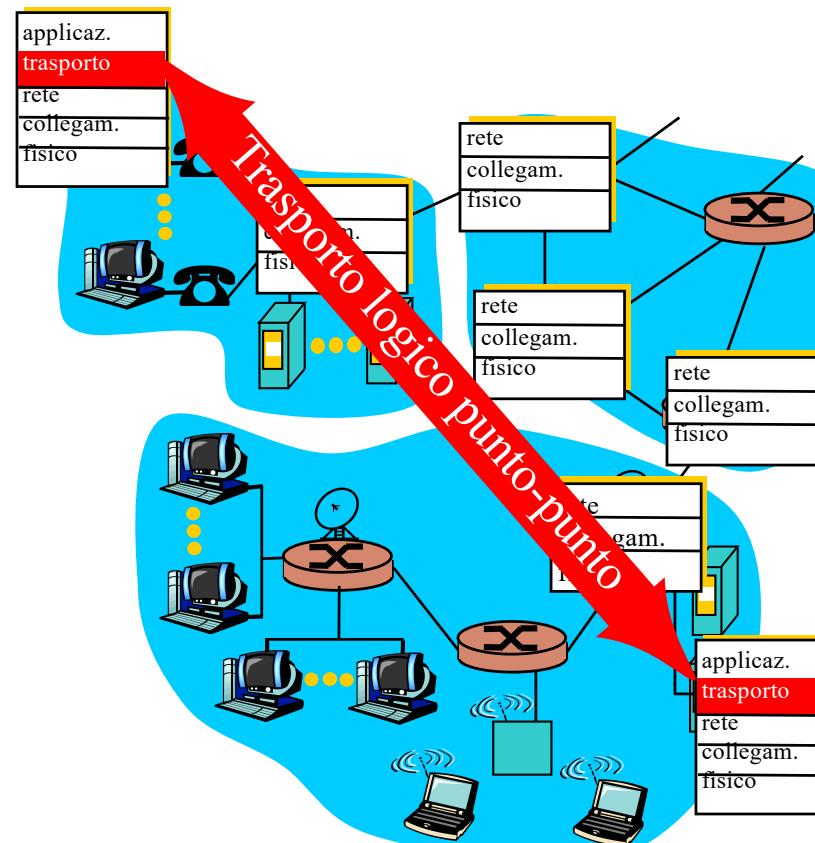


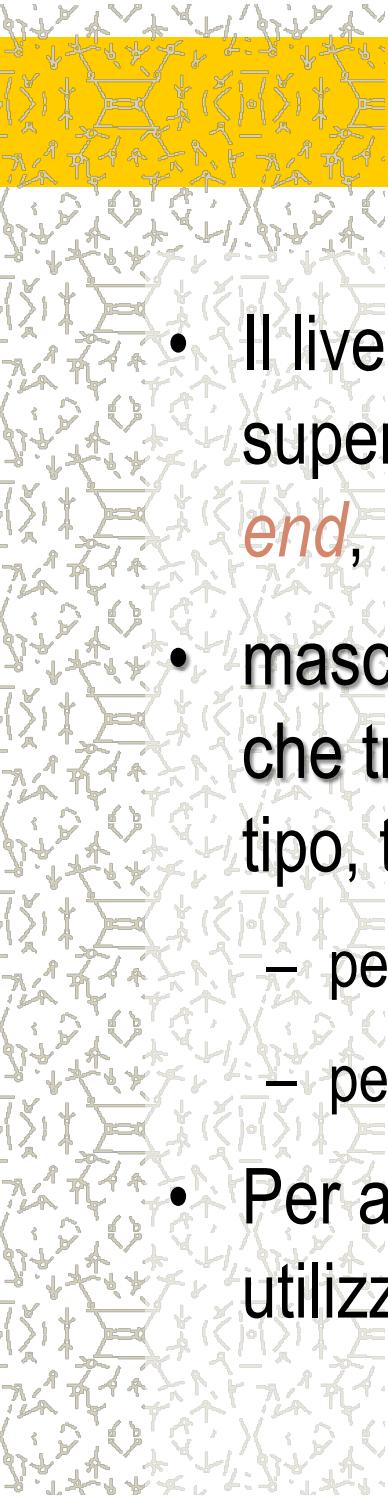
Modello ISO-OSI



Funzione del livello di trasporto

Un protocollo dello strato di trasporto fornisce una comunicazione logica fra i processi applicativi che girano su host differenti

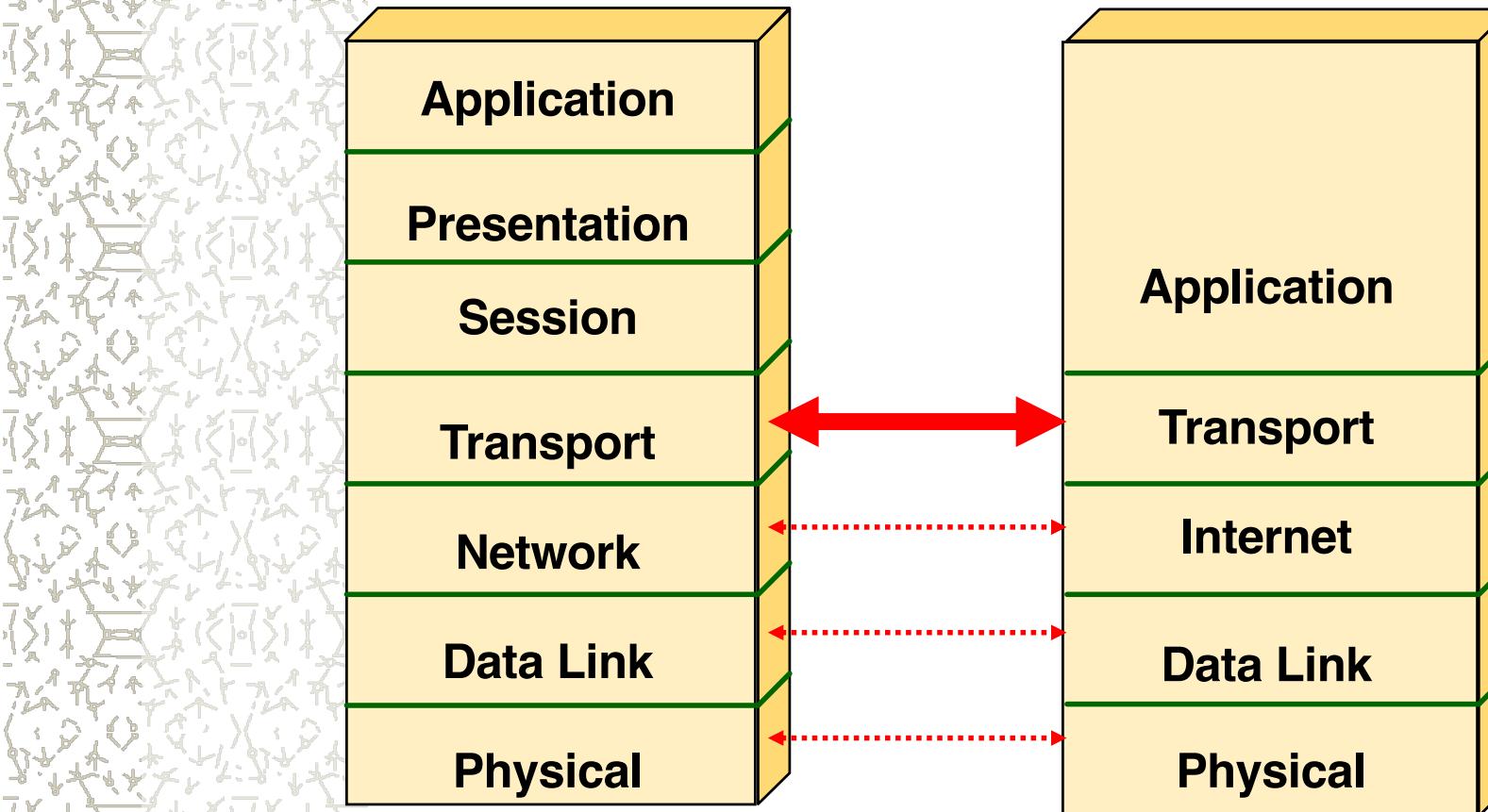




Funzione del livello di trasporto

- Il livello di **trasporto** ha lo scopo di fornire allo strato superiore un servizio di trasferimento dei dati ***end to end***,
- mascherando completamente al livello superiore il fatto che tra i due host terminali esista **una rete** di qualsiasi tipo, topologia, tecnologia e complessità
 - per OSI lo strato superiore è il **livello di sessione**
 - per TCP/IP lo strato superiore è il **livello di applicazione**
- Per assolvere le sue funzioni lo strato di trasporto utilizza i **servizi** dello **strato di rete**

Comunicazione fra TCP/IP stacks



- Il **TCP** su un computer usa **IP** e i livelli inferiori per comunicare con il **TCP** di un altro computer

Relazione tra livello di trasporto e livello di rete

- *livello di rete:* comunicazione logica tra host
- *livello di trasporto:* comunicazione logica tra processi di applicazioni che girano su host
 - si basa sui servizi del livello di rete

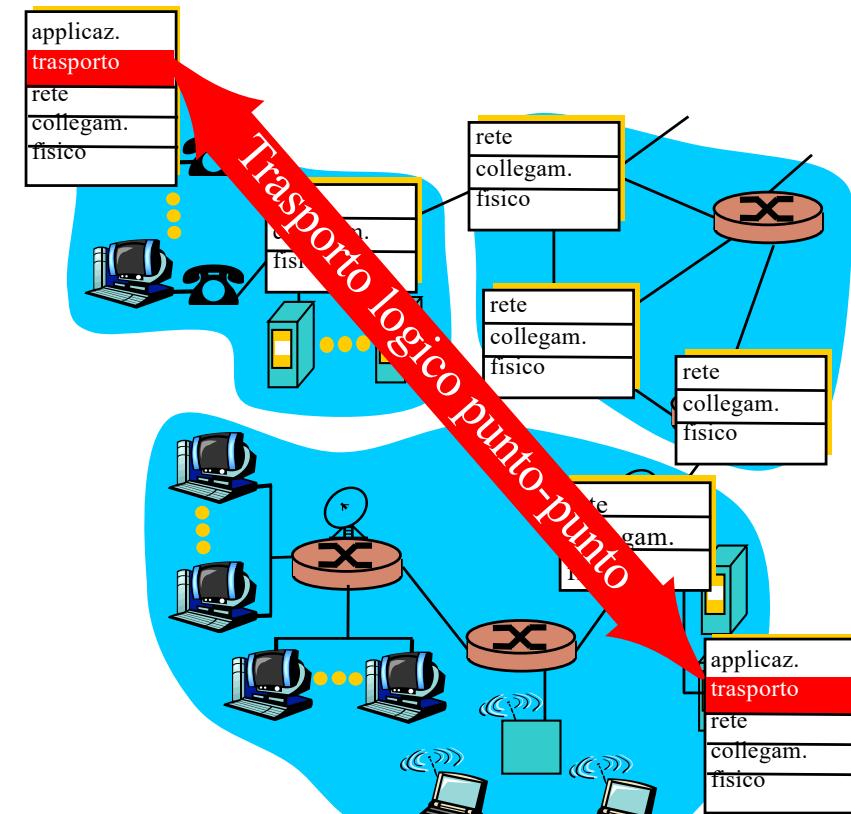
Analogia con la posta ordinaria:

12 ragazzi inviano lettere a 12 ragazzi

- processi = ragazzi
- messaggi delle applicazioni = lettere nelle buste
- host = case
- protocollo di trasporto = Anna e Andrea
- protocollo del livello di rete = servizio postale

Servizi e protocolli di trasporto

- Forniscono la *comunicazione logica* tra processi applicativi di host differenti
 - lato invio: scinde i messaggi in **segmenti** e li passa al livello di rete
 - lato ricezione: riassembra i segmenti in messaggi e li passa al livello di applicazione
 - Più protocolli di trasporto sono a disposizione delle applicazioni Internet: **TCP e UDP**



Servizio di trasporto

- **Affidabile (TCP)**

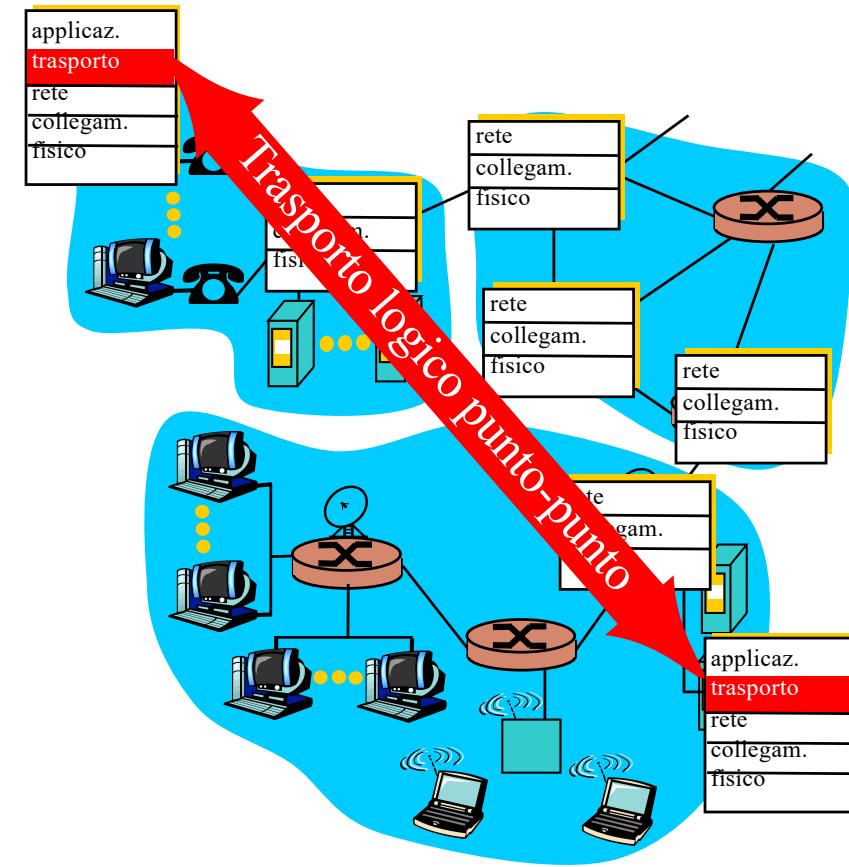
- consegne nell'ordine originario
- controllo di congestione
- controllo di flusso
- setup della connessione

- **Inaffidabile (UDP)**

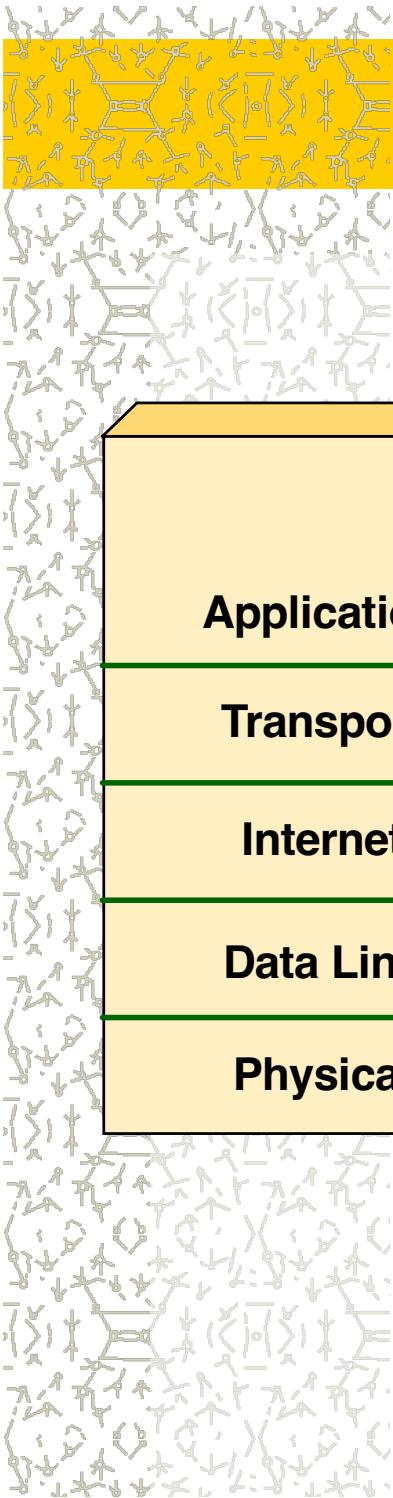
- consegne senz'ordine:
- estensione senza fronzoli del servizio di consegna a massimo sforzo

- **Servizi non disponibili:**

- garanzia su ritardi
- garanzia su ampiezza di banda



Protocolli di Trasporto



Servizio di trasporto e chiamate di procedure

- Può essere **con connessione (TCP)** o **senza connessione (UDP)**
- È disponibile al programmatore delle applicazioni come un insieme di chiamate di procedura disponibili in una libreria
 - Il trasporto con connessione fornisce un **canale affidabile** su cui scrivere o da cui leggere dati (come un file)
 - Per il servizio connection oriented si possono elencare in
 - **LISTEN**: lo strato superiore notifica al trasporto che è pronto a ricevere una connessione
 - **CONNECT**: lo strato superiore chiede allo strato di trasporto di effettuare una connessione (si traduce nell'invio da parte del trasporto di un messaggio "Connection Request" al destinatario)
 - **SEND**: lo strato superiore chiede al trasporto di inviare dati
 - **RECEIVE**: lo strato superiore chiede allo strato di trasporto di trasmettergli i dati in arrivo
 - **DISCONNECT**: lo strato superiore chiede di chiudere la connessione (si traduce nell'invio da parte dello strato di trasporto di un messaggio "Disconnection Request")
 - Per il servizio connectionless, le due primitive SEND e RECEIVE possono essere sufficienti

RFC di TCP e UDP

Protocolli di trasporto definiti su rete Internet (su IP)

- **Transmission Control Protocol (TCP)** definisce un protocollo di trasporto orientato alla connessione
 - definito in RFC 793, RFC 1122 e RFC 1323
 - progettato per fornire un flusso affidabile end-to-end su una internet inaffidabile
- **User Data Protocol (UDP)** definisce un protocollo senza connessione
 - descritto in RFC 768
 - permette di inviare datagram IP senza stabilire una connessione

Protocolli di trasporto: principali obiettivi

- **Indirizzamento** a livello trasporto (su uno stesso host possono essere disponibili più connessioni)
 - Viene introdotto il concetto di service port
- **Mult/Dem.** In generale il livello di trasporto su un host gestisce numerose connessioni
 - Il livello di trasporto provvede a **multiplare e demoltiplicare** i pacchetti provenienti dal livello di rete sulle diverse connessioni
- Gestiscono il **controllo degli errori**, i **numeri di sequenza** e il **controllo di flusso** per un collegamento attraverso una rete (situazione più complessa del caso del livello data link)
- **Controllo della congestione**. Devono **risolvere** il problema della **capacità di memorizzazione della rete** (un pacchetto può essere memorizzato in un router e consegnato dopo un certo ritardo)

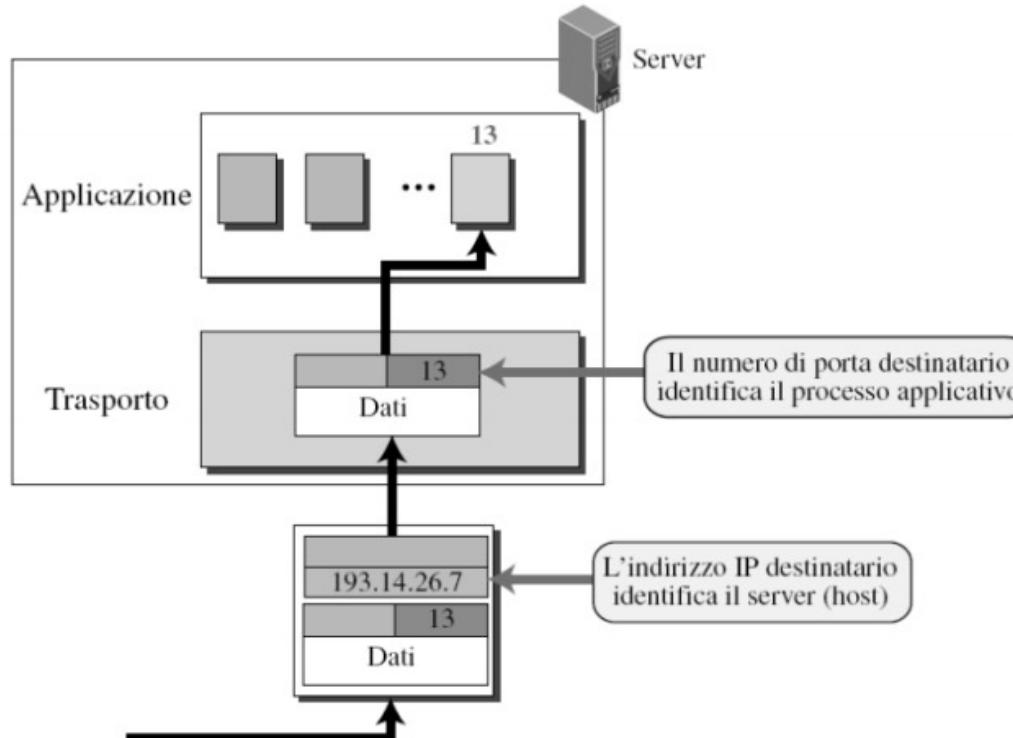
Indirizzamento a livello trasporto

- La maggior parte dei sistemi operativi è multiutente e multiprocesso
 - Diversi processi client attivi (host locale)
 - Diversi processi server attivi (host remoto)
- Per stabilire una comunicazione tra i due dispositivi è necessario un metodo per individuare:
 - Host locale
 - Host remoto
 - Processo locale
 - Processo remoto
- **Host → indirizzo IP**
- **Processo → numero di porta**

Indirizzamento a livello trasporto

- Host → indirizzo IP
- Processo → numero di porta

Indirizzo IP + porta = socket address



Multiplexing/demultiplexing

Analogia con la posta ordinaria:

12 ragazzi (processi) della casa A (Host A)
inviano lettere (Msg) a
12 ragazzi (processi) della casa B (Host B)

Anna raccoglie (Multiplexing) le 12 lettere della casa A e le consegna al servizio postale (Rete)

Andrea raccoglie le lettere dal servizio postale (Rete) e le consegna (demultiplexing) ai rispettivi ragazzi (processi) della casa B (Host B)

protocollo di trasporto =Anna e Andrea

Multiplexing/demultiplexing: Socket

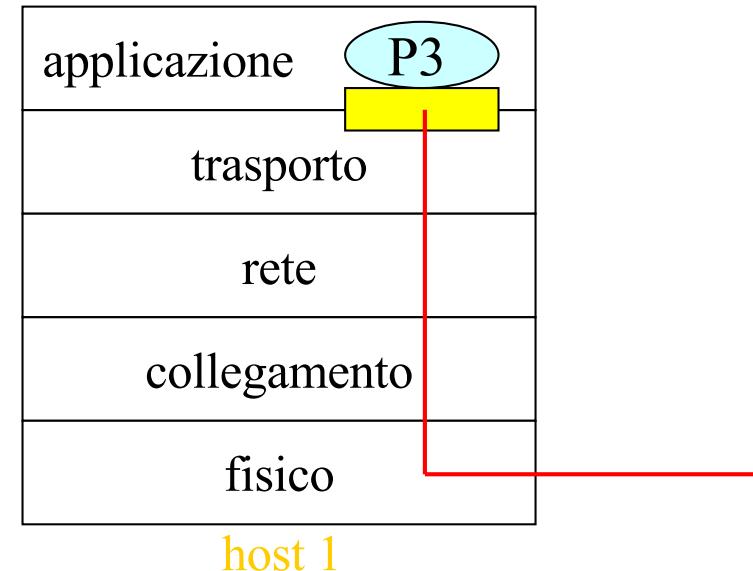
Ogni processo (ragazzo), come parte di un'applicazione , ha un
SOCKET

SOCKET:

*una porta attraverso la quale i dati passano dalla rete al processo e
attraverso la quale i dati passano dal processo alla rete*

= processo

= socket



Multiplexing/demultiplexing

Demultiplexing
nell'host ricevente:

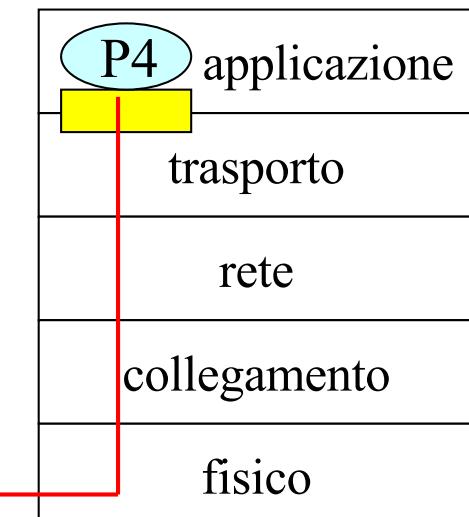
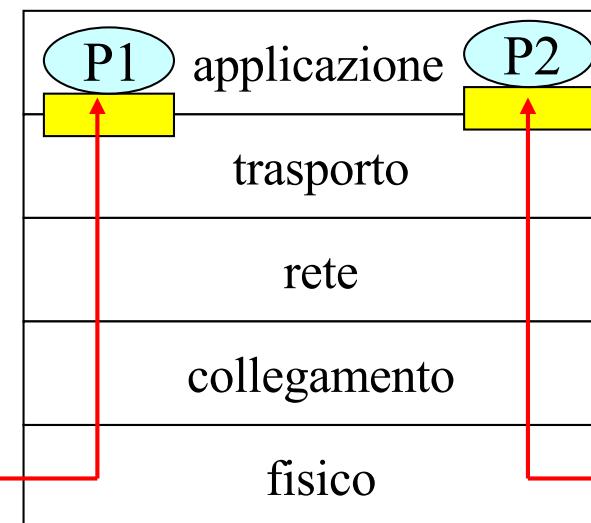
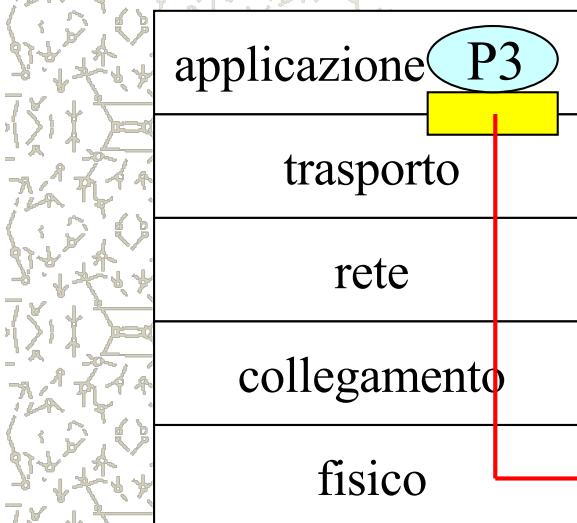
consegnare i segmenti ricevuti
alla socket appropriata

Multiplexing
nell'host mittente:

raccogliere i dati da varie
socket, incapsularli con
l'intestazione (utilizzati poi per il
demultiplexing)

= socket

= processo

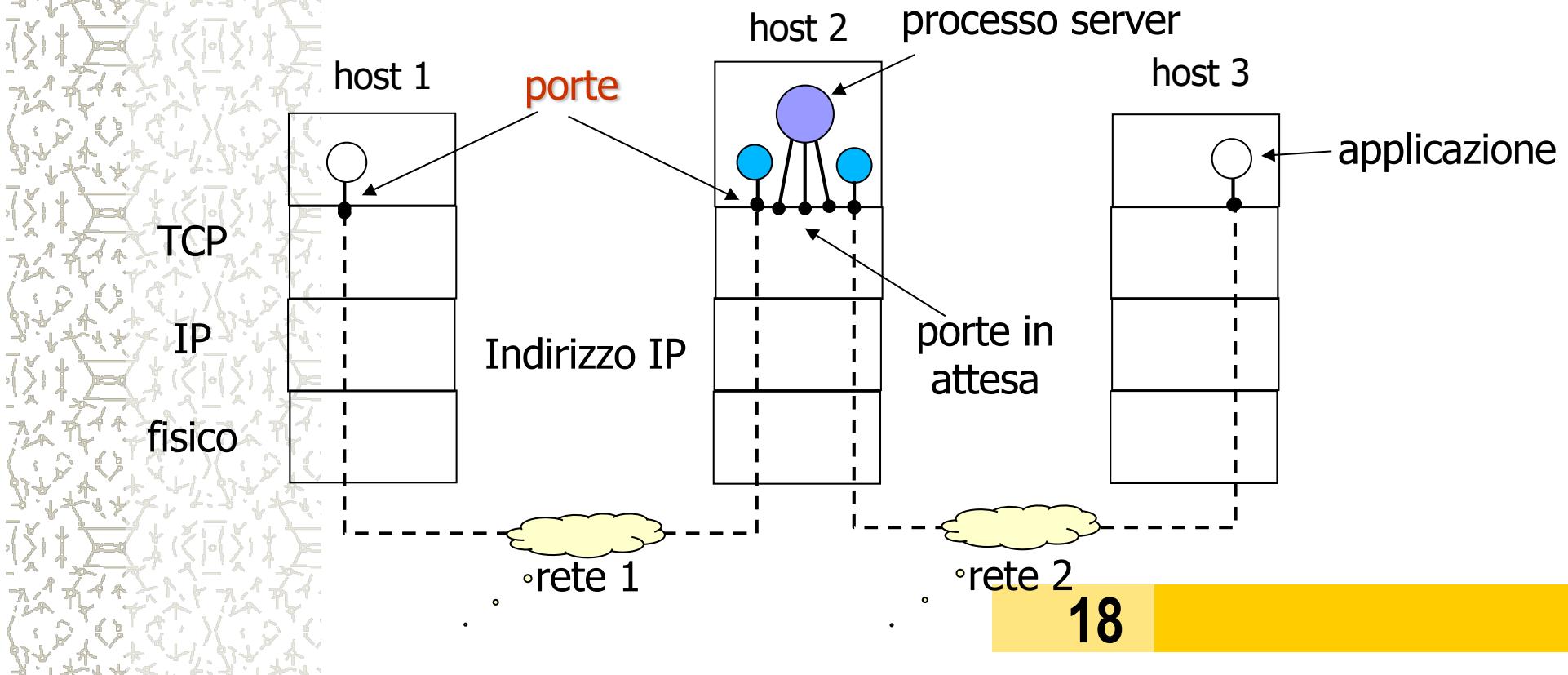


host 2

host 3

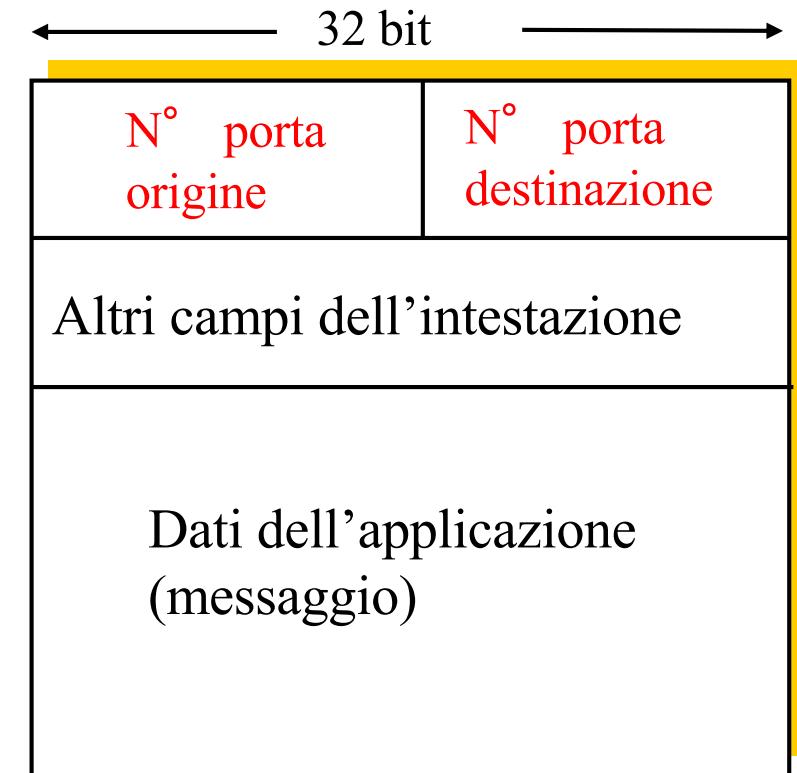
Multiplexing/demultiplexing: Indirizzamento

- La porta e l'indirizzo IP possono essere visti come un indirizzo a livello di trasporto
- Le **applicazioni** che utilizzano il TCP/IP si **registrano** sullo strato di trasporto ad un indirizzo specifico, detto **porta**
- La porta è il meccanismo che ha a disposizione un'applicazione per **identificare** l'applicazione remota a cui inviare i dati
- La porta è un numero di **16 bit** (da 1 a 65535; la porta 0 non è utilizzata)



Come funziona il demultiplexing

- L'host riceve i datagrammi IP
 - ogni datagramma ha un indirizzo IP di origine e un indirizzo IP di destinazione
 - ogni datagramma trasporta 1 segmento a livello di trasporto
 - ogni segmento ha un numero di porta di origine e un numero di porta di destinazione
- L'host usa gli indirizzi IP e i numeri di porta per inviare il segmento alla socket appropriata
 - L'indirizzo e la porta di sorgente servono al receiver per inviare una risposta al sender



Struttura del segmento TCP/UDP

Demultiplexing senza connessione (UDP)

- Crea le socket con i numeri di porta:

```
DatagramSocket mySocket1 =  
    new DatagramSocket(99111);
```

```
DatagramSocket mySocket2 =  
    new DatagramSocket(99222);
```

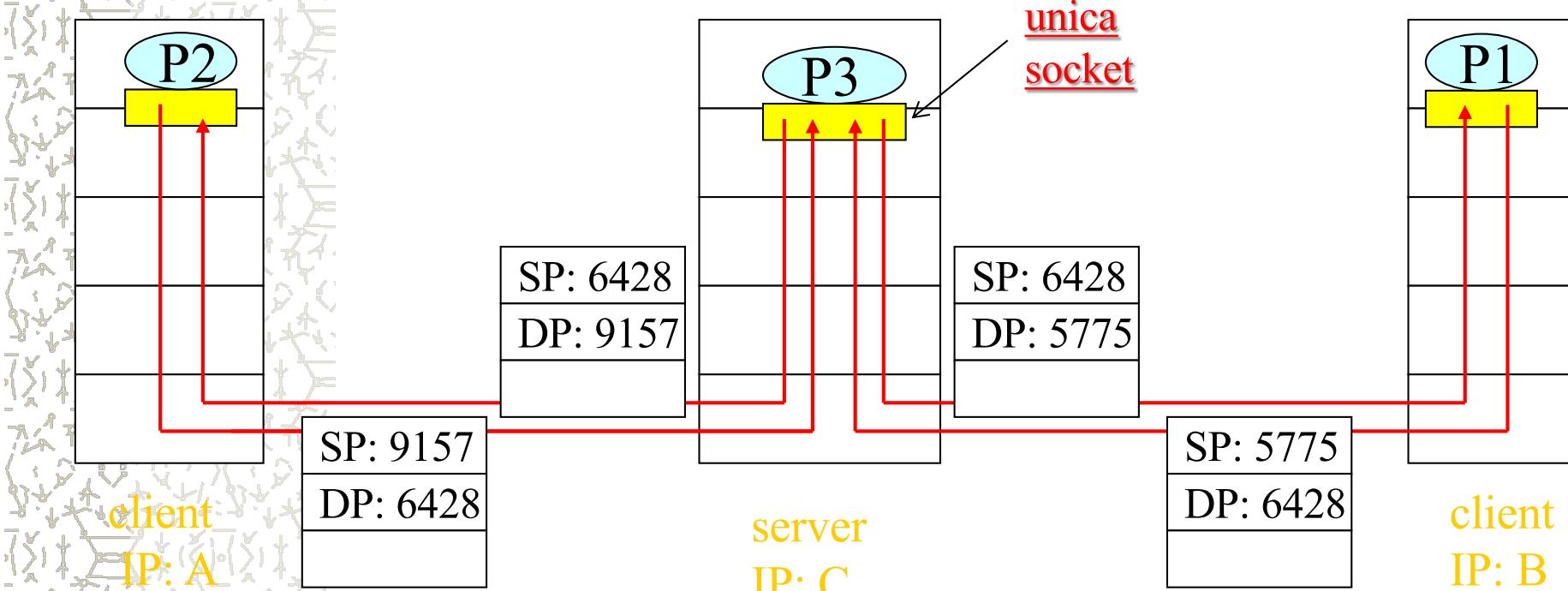
- La socket UDP è identificata da 2 parametri:

(indirizzo IP di destinazione,
numero della porta di destinazione)

- Quando l'host riceve il segmento UDP:
 - controlla il numero della porta di destinazione nel segmento
 - invia il segmento UDP alla socket con quel numero di porta
- I datagrammi IP con indirizzi IP di origine e/o numeri di porta di origine differenti vengono inviati alla stessa socket

Demultiplexing senza connessione (UDP)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

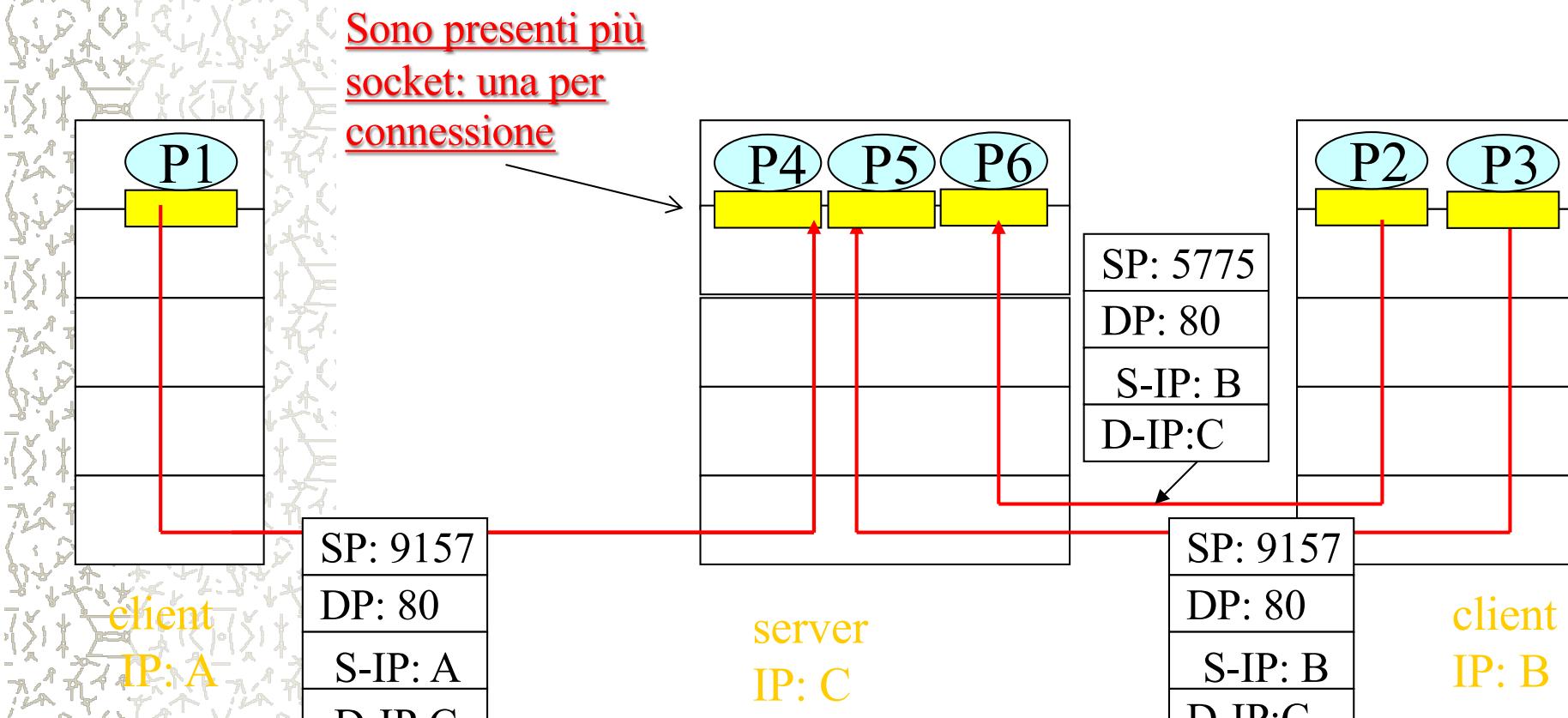


SP fornisce “l’indirizzo di ritorno”

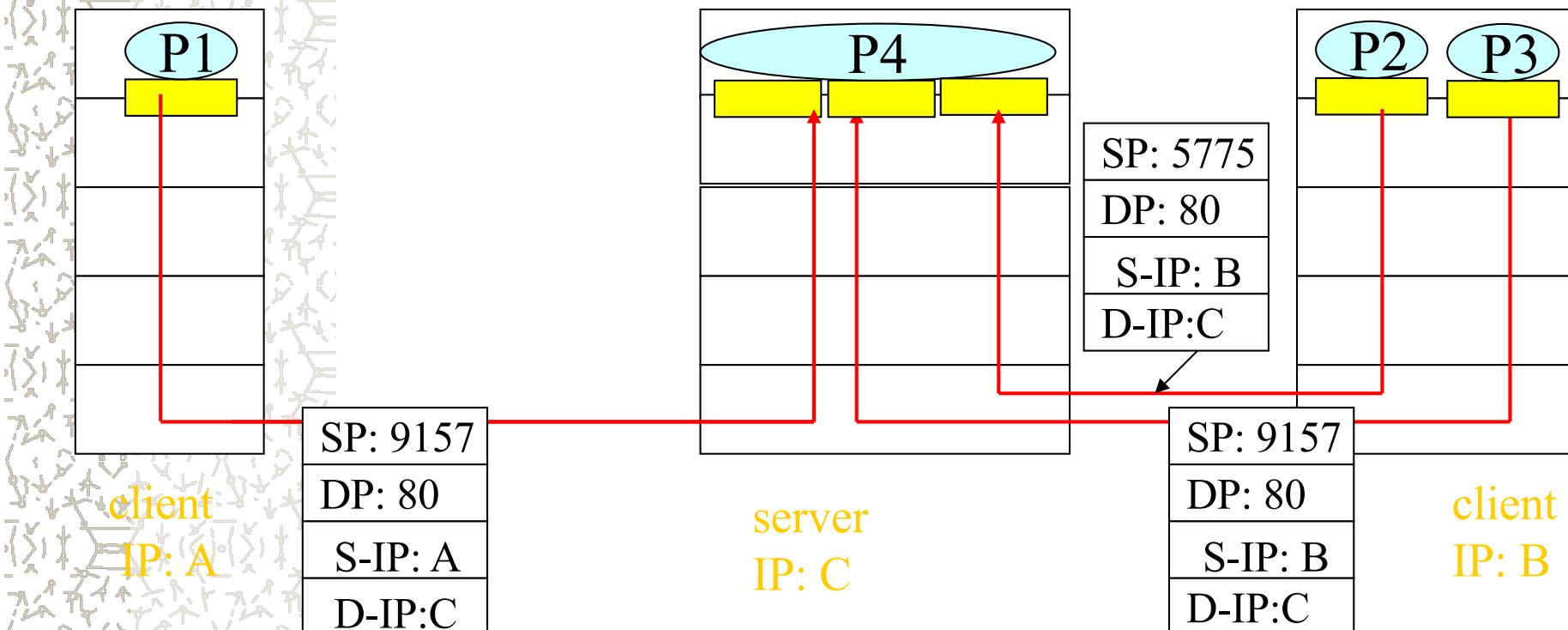
Demultiplexing orientato alla connessione (TCP)

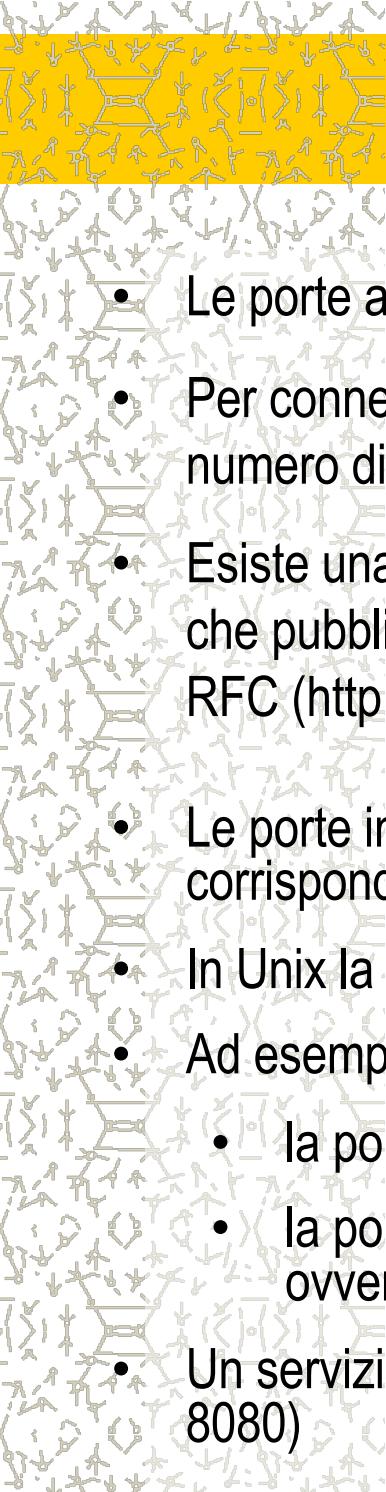
- La socket TCP è identificata da 4 parametri:
 - indirizzo IP di origine
 - numero di porta di origine
 - indirizzo IP di destinazione
 - numero di porta di destinazione
- L'host ricevente usa i quattro parametri per inviare il segmento alla socket appropriata
- Un host server può supportare più socket TCP contemporanee:
 - ogni socket è identificata dai suoi 4 parametri
- I server web hanno socket differenti per ogni connessione client
 - con HTTP non-persistente si avrà una socket differente per ogni richiesta

Demultiplexing orientato alla connessione (TCP)



Demultiplexing orientato alla connessione: thread dei server web

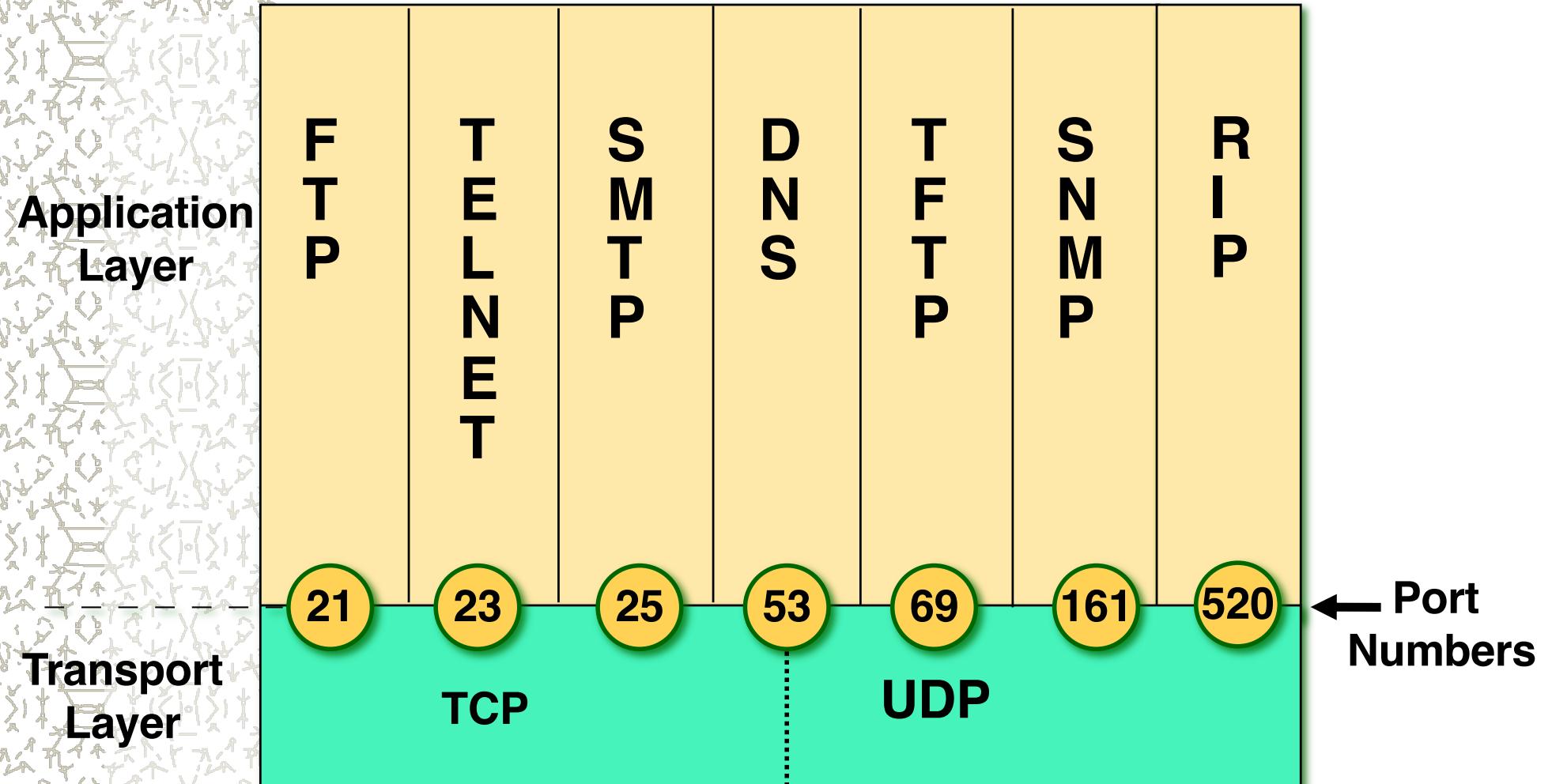




Le porte

- Le porte attive definiscono i servizi TCP disponibili
- Per connettersi ad un servizio specifico su un server si deve conoscere il numero di porta su cui il processo server accetta le connessioni
- Esiste una autorità centrale, lo IANA (**Internet Assigned Numbers Authority**), che pubblica la raccolta dei numeri di porta assegnati alle applicazioni negli RFC (<http://www.iana.org>)
- Le porte inferiori alla 1024 sono dette **porte ben note (well-known ports)** e corrispondono a servizi standard
- In Unix la lista dei servizi e delle porte è nel file </etc/services>
- Ad esempio
 - la porta 21 di TCP corrisponde al servizio **FTP** (File Transfer Protocol)
 - la porta 80 di TCP corrisponde al servizio **HTTP** (Hypertext Transfer Protocol) ovvero al server Web
 - Un servizio “standard” può anche essere attivato su una porta diversa (es. HTTP su 8080)

Port Numbers



Protocol Port Number

I numeri delle porte vengono divisi in tre gruppi:

- **Well-Known-Ports (0 – 1023)**: Queste porte vengono assegnate univocamente dall'IANA
- **Registered Ports (1024 – 49151)**: L'uso di queste porte viene registrato a beneficio degli utenti della rete, ma non esistono vincoli restrittivi
- **Dynamic and/or Private Ports (49152 – 65535)**: Non viene applicato nessun controllo all'uso di queste porte

Le porte del client

Il client definisce la porta di ogni sua connessione utilizzando numeri in genere elevati e scelti in modo da essere unici sull'host

Ad esempio

richiesta di connessione ad un server TELNET

client port 23443
server port 23

Le connessioni sono quindi punto-a-punto e full duplex

Incapsulamento/decapsulamento

I pacchetti a livello di trasporto sono chiamati:

- segmenti (TCP)
- datagrammi utente (UDP)



Gestione della connessione TCP

Ricordiamo: mittente e destinatario TCP stabiliscono una “connessione” prima di scambiare i segmenti di dati

- inizializzano le variabili TCP:
 - numeri di sequenza
 - buffer, informazioni per il controllo di flusso (per esempio, **RcvWindow**)

- *client*: avvia la connessione

```
Socket clientSocket = new  
Socket("hostname", "portnumber");
```

- *server*: contattato dal client

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

Handshake a tre vie:

Passo 1: il client invia un segmento SYN al server

- specifica il numero di sequenza iniziale
- nessun dato

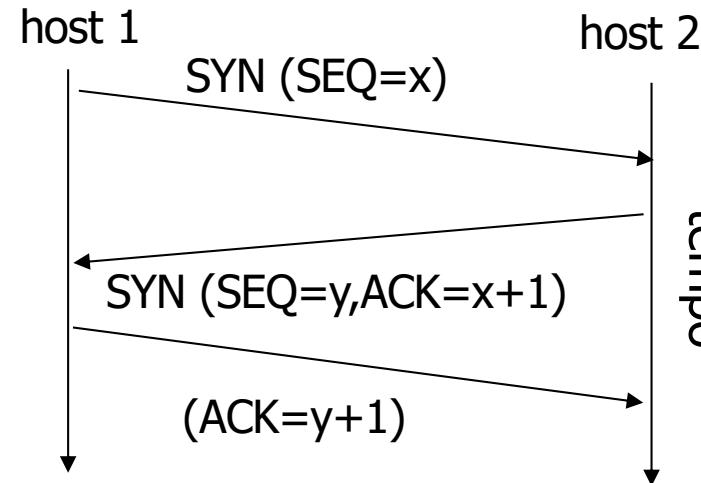
Passo 2: il server riceve SYN e risponde con un segmento SYNACK

- il server alloca i buffer
- specifica il numero di sequenza iniziale del server

Passo 3: il client riceve SYNACK e risponde con un segmento ACK, che può contenere dati

Apertura della connessione

Si utilizza un protocollo **3-way handshaking**



Se il TCP ricevente non verifica la presenza di nessun processo in attesa sulla porta destinazione manda un segmento di rifiuto della connessione (RST).

Un esempio di connessione

Connessione Telnet fra da 10.6.1.9 a 10.6.1.2 catturata con tcpdump *
porta client 4548 - porta server 23 (telnet)

```
10.6.1.9.4548 > 10.6.1.2.23: S 2115515278:2115515278(0) win 32120  
<mss 1460,nop,nop,sackOK,nop,wscale 0> (DF)
```

```
10.6.1.2.23 > 10.6.1.9.4548: S 1220480853:1220480853(0)  
ack 2115515279 win 32120 <mss 1460,nop,nop,sackOK,nop,wscale 0>  
(DF)
```

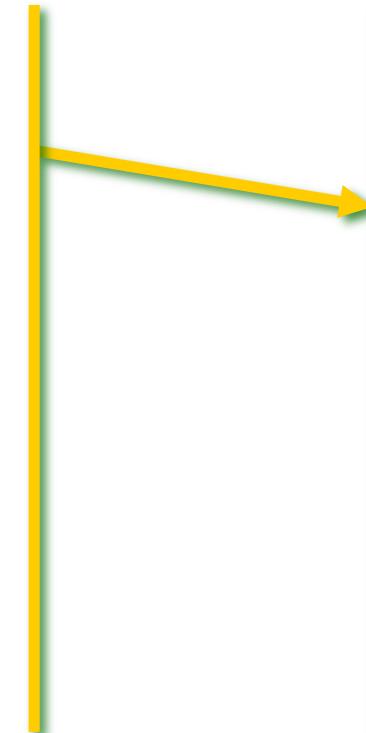
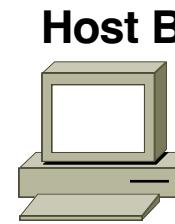
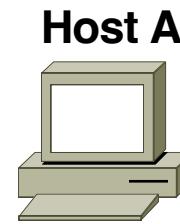
```
10.6.1.9.4548 > 10.6.1.2.23: . ack 1220480854 win 32120 (DF)
```

* tcpdump -S -n -t \dst 10.6.1.2 and src 10.6.1.9\ or \dst 10.6.1.9 and src 10.6.1.2\

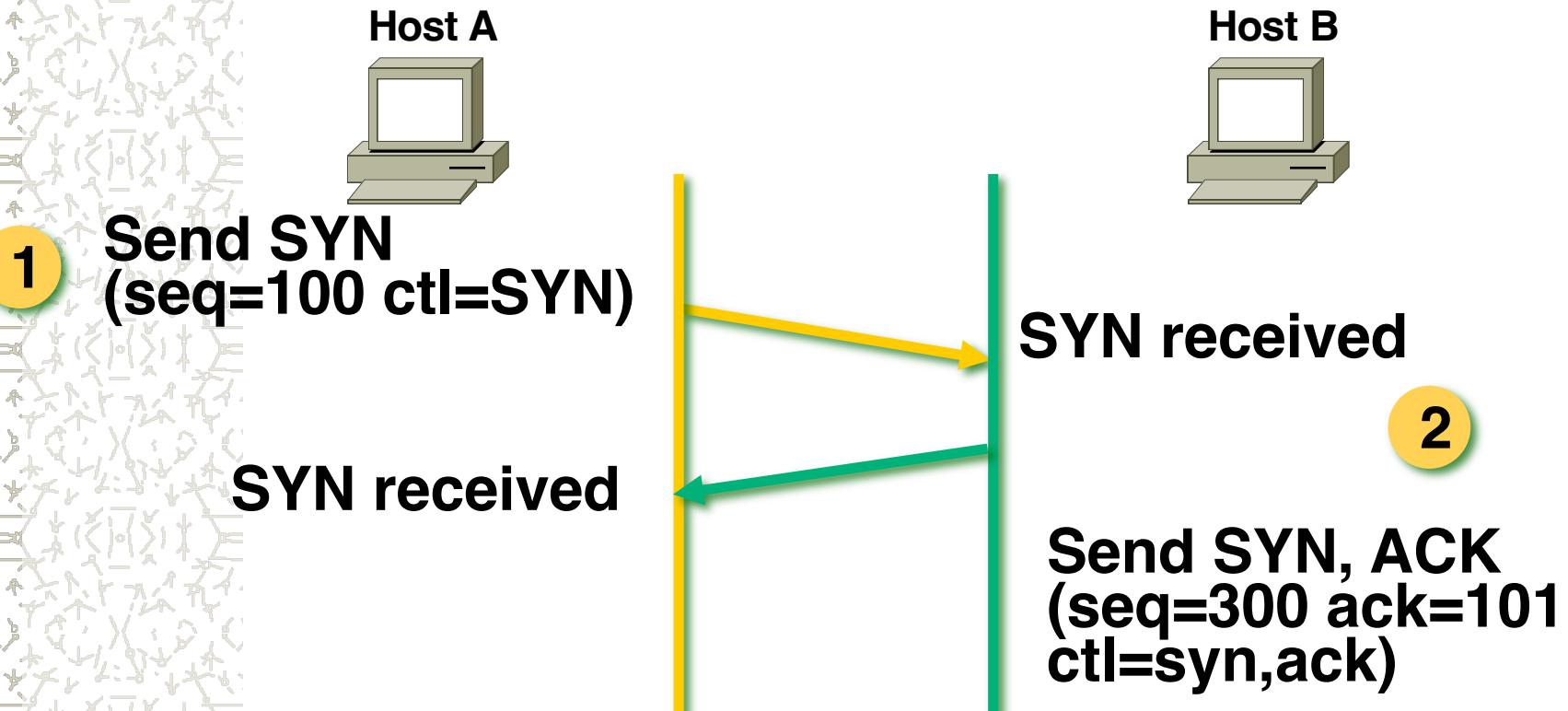
Three Way Handshake/Open

1

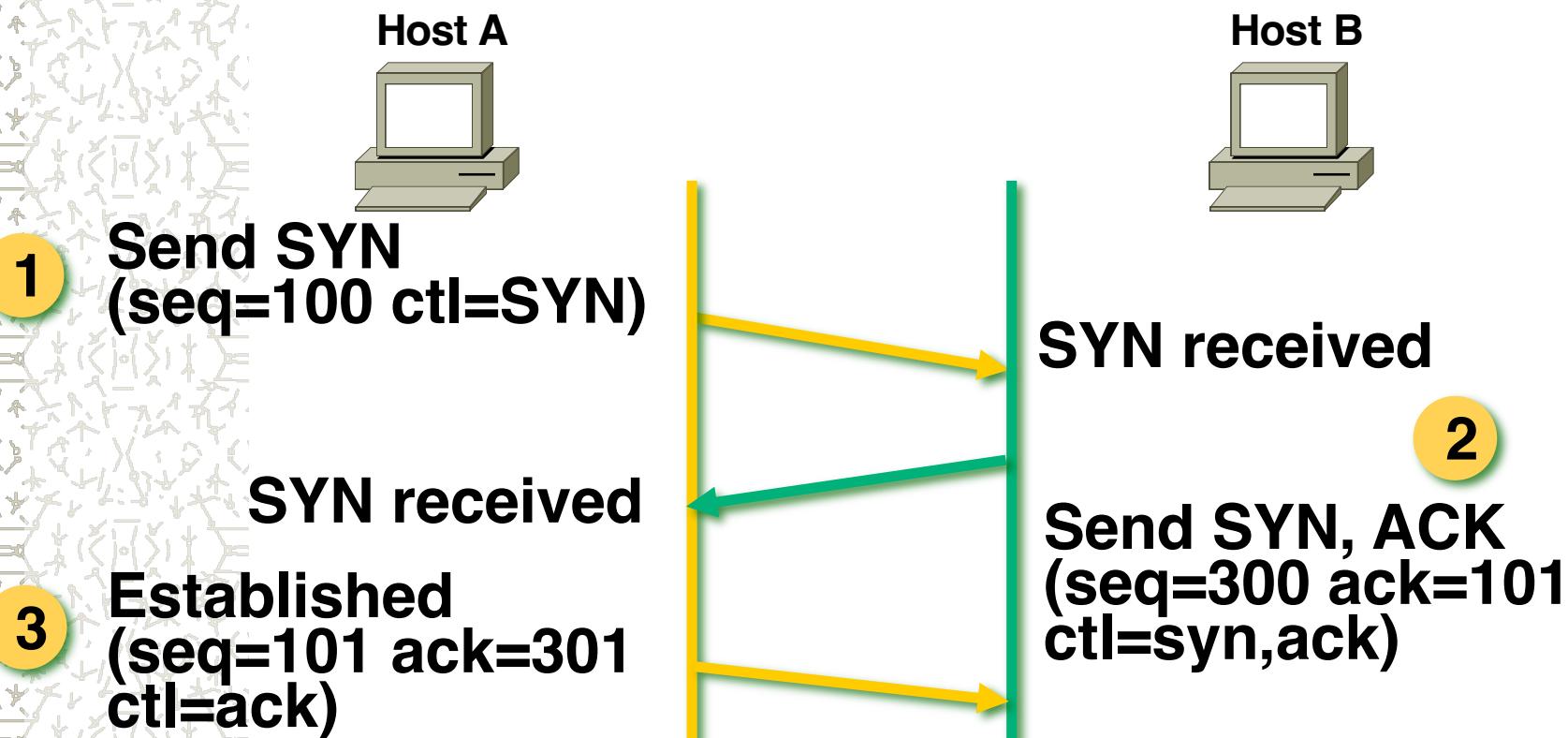
**Send SYN
(seq=100 ctl=SYN)**



Three Way Handshake/Open



Three Way Handshake/Open

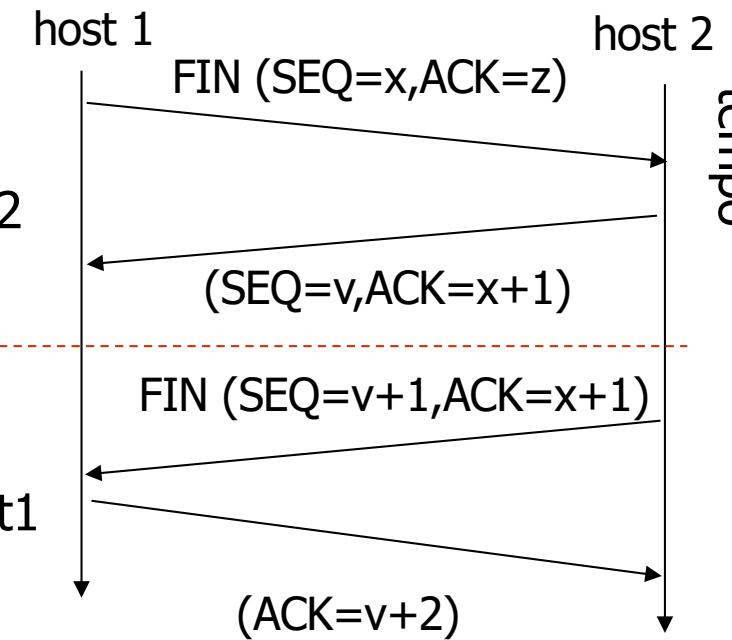


Chiusura della connessione

La connessione è full-duplex e le due direzioni devono essere chiuse indipendentemente

Chiusura da host1 a host2

Chiusura da host2 a host1



Host 2 può ancora inviare dati a host1

Se l'ack di un messaggio FIN si perde l'host mittente chiude comunque la connessione dopo un timeout

Chiusura della connessione

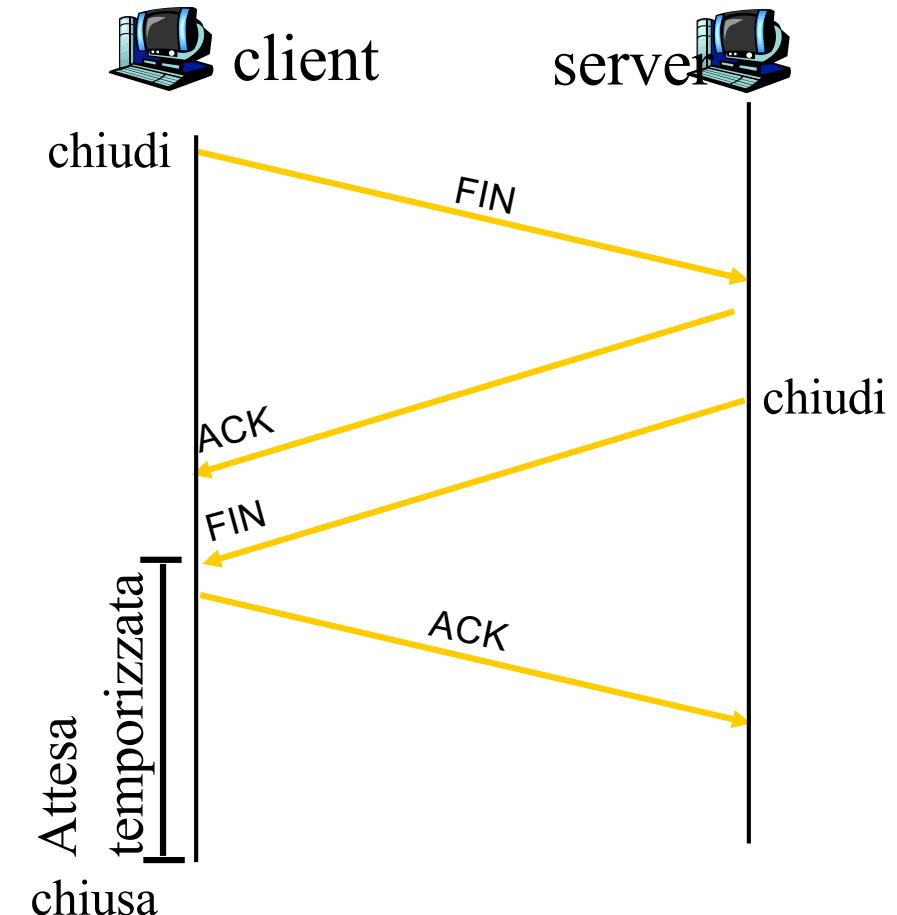
Chiudere una connessione:

Il client chiude la socket:

```
clientSocket.close();
```

Passo 1: il **client** invia un segmento di controllo FIN al server.

Passo 2: il **server** riceve il segmento FIN e risponde con un ACK. Chiude la connessione e invia un FIN.

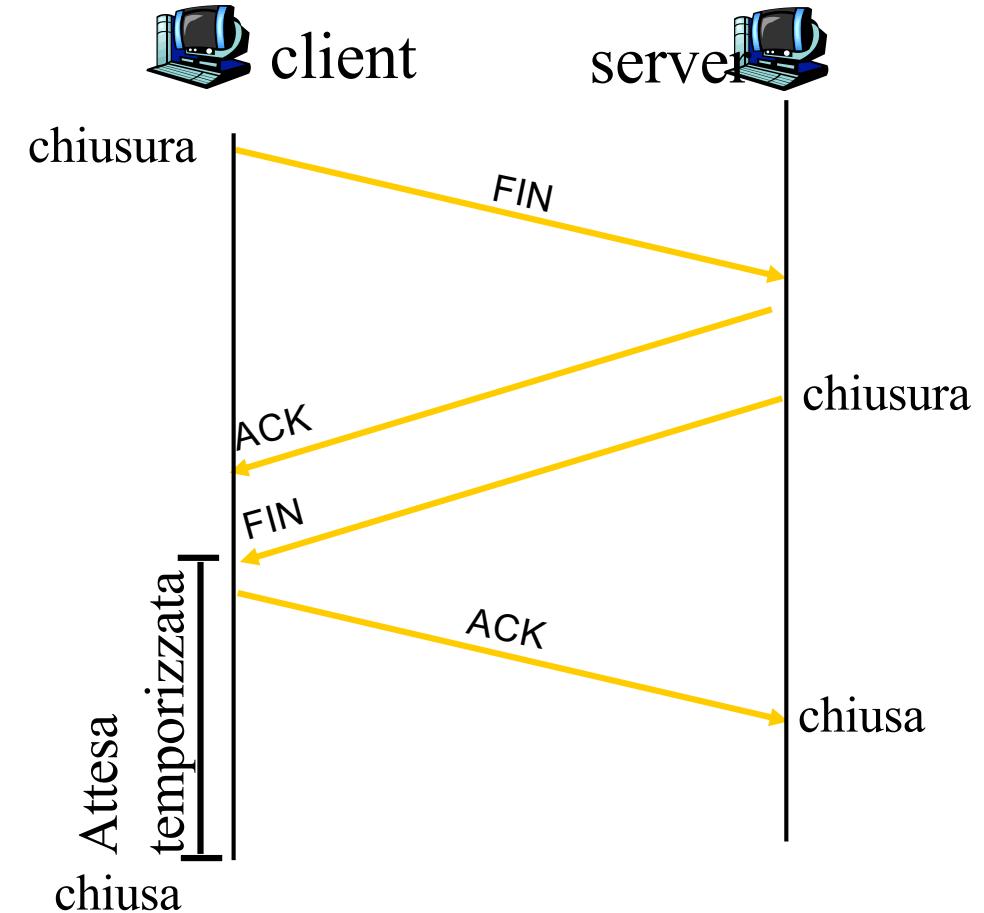


Chiusura della connessione

Passo 3: il **client** riceve FIN e risponde con un ACK.

- inizia l'attesa temporizzata - risponde con un ACK ai FIN che riceve

Passo 4: il **server** riceve un ACK. La connessione viene chiusa.



Un esempio di chiusura

**Chiusura Telnet da 10.6.1.9
porta client 4548 - porta server 23 (telnet)**

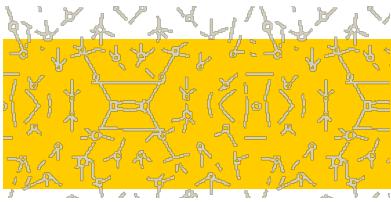
FIN

10.6.1.9.4548 > 10.6.1.2.23: F 2115515449:**2115515449**(0) ack
1220480986 win 32120 (DF)

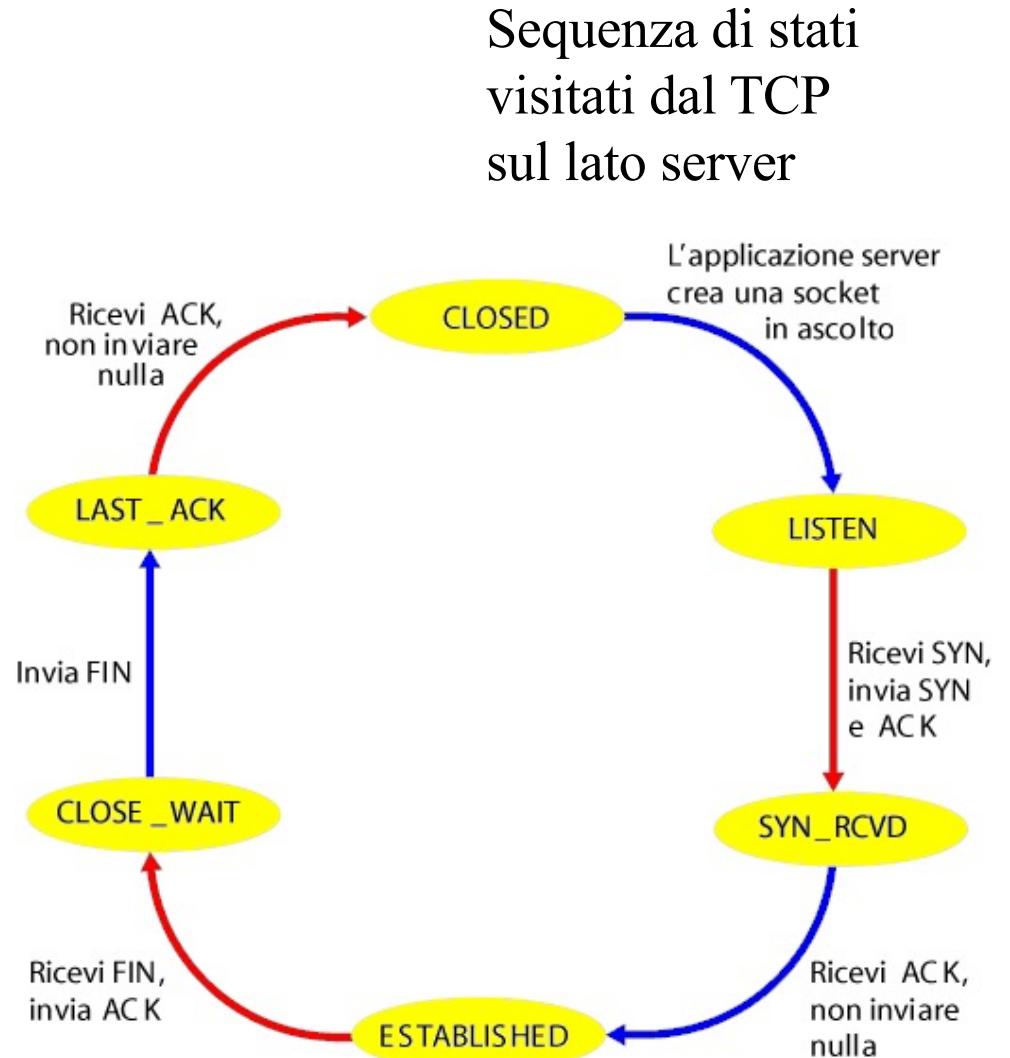
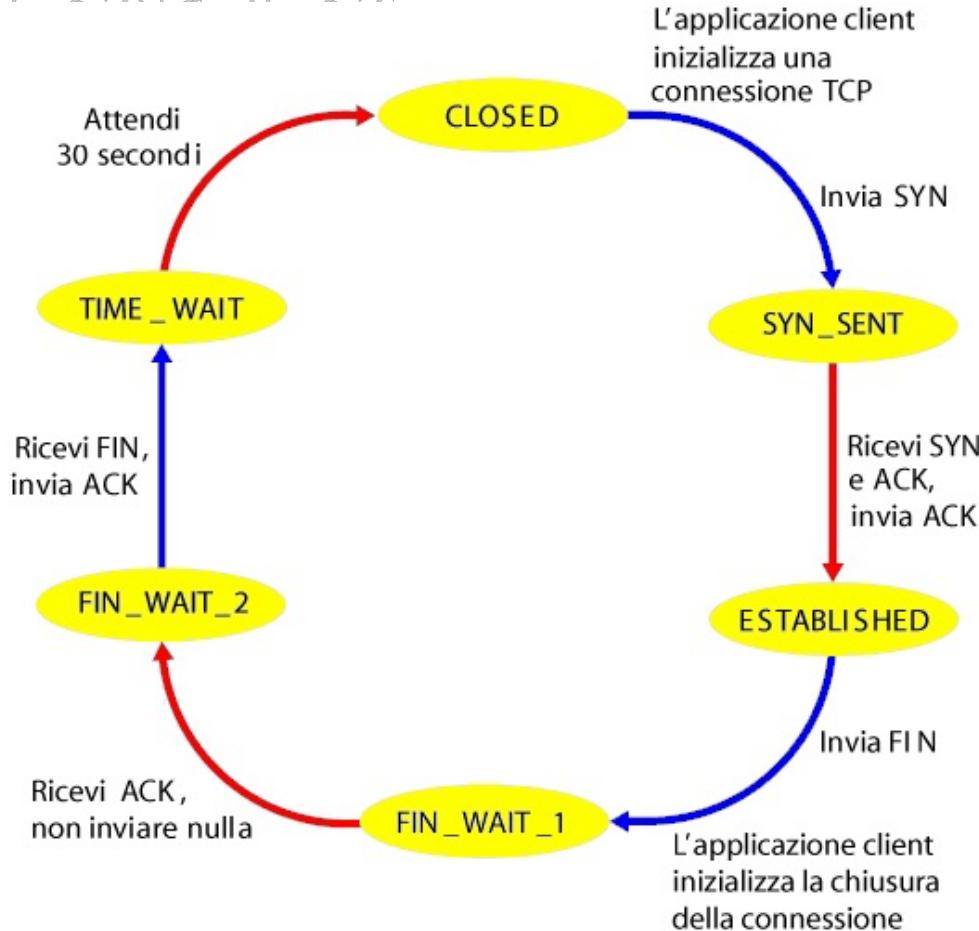
10.6.1.2.23 > 10.6.1.9.4548: . ack **2115515450** win 32120 (DF)

10.6.1.2.23 > 10.6.1.9.4548: F 1220480986:**1220480986**(0) ack
2115515450 win 32120 (DF)

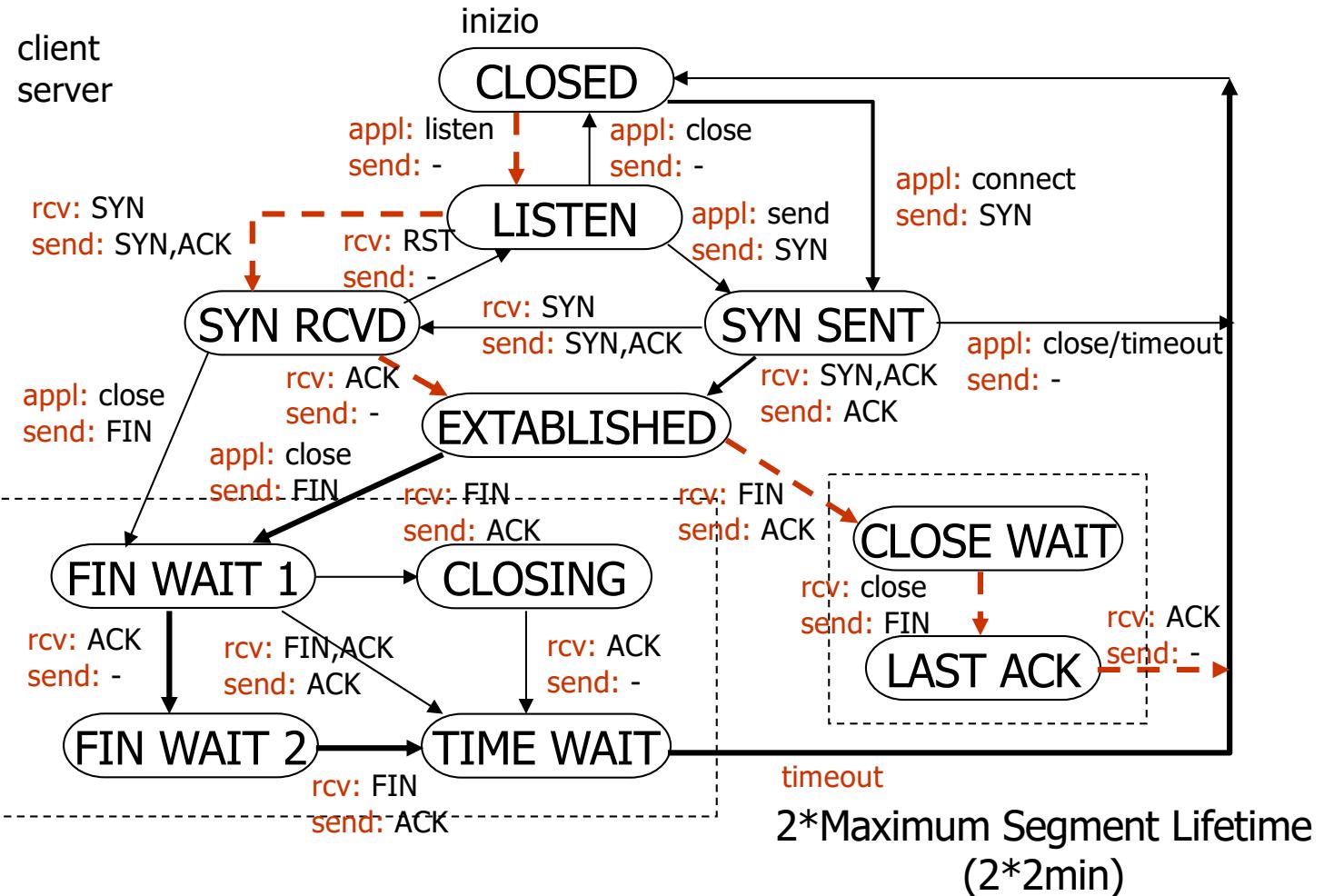
10.6.1.9.4548 > 10.6.1.2.23: . ack **1220480987** win 32120 (DF)



Il diagramma degli stati TCP



Il diagramma degli stati TCP



TCP: Panoramica

RFC: 793, 1122, 1323, 2018, 2581

- **punto-punto:**

- un mittente, un destinatario

- **flusso di byte affidabile, in sequenza:**

- nessun “confine ai messaggi”

- **pipeline:**

- il controllo di flusso e di congestione TCP definiscono la dimensione della finestra

- **buffer d'invio e di ricezione**

- **full duplex:**

- flusso di dati bidirezionale nella stessa connessione
 - MSS: dimensione massima di segmento (maximum segment size)

- **orientato alla connessione:**

- l'handshaking (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima di scambiare i dati

- **flusso controllato:**

- il mittente non sovraccarica il destinatario



Funzionalità del TCP

Trasmissione

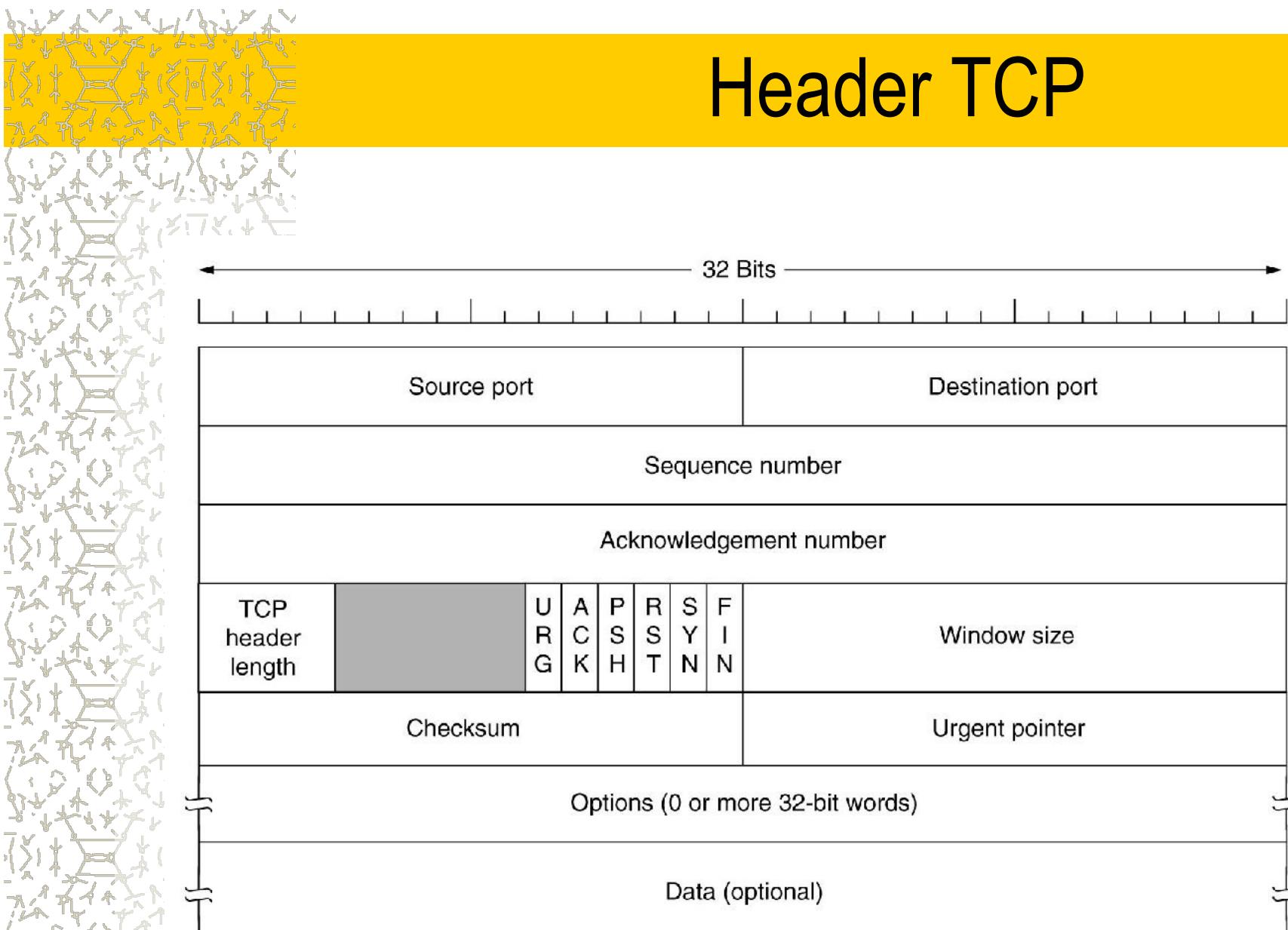
- Riceve un flusso di dati dall'applicazione
- Li organizza in unità lunghe al massimo 64Kb
- Spedisce le unità di dati come datagram IP

Ricezione

- Riceve i datagram IP
- Ricostruisce il flusso di byte originale nella sequenza corretta

Ritrasmissione dei datagram non ricevuti, riordinamento dei datagram arrivati in ordine sbagliato

Header TCP



Struttura dei segmenti TCP

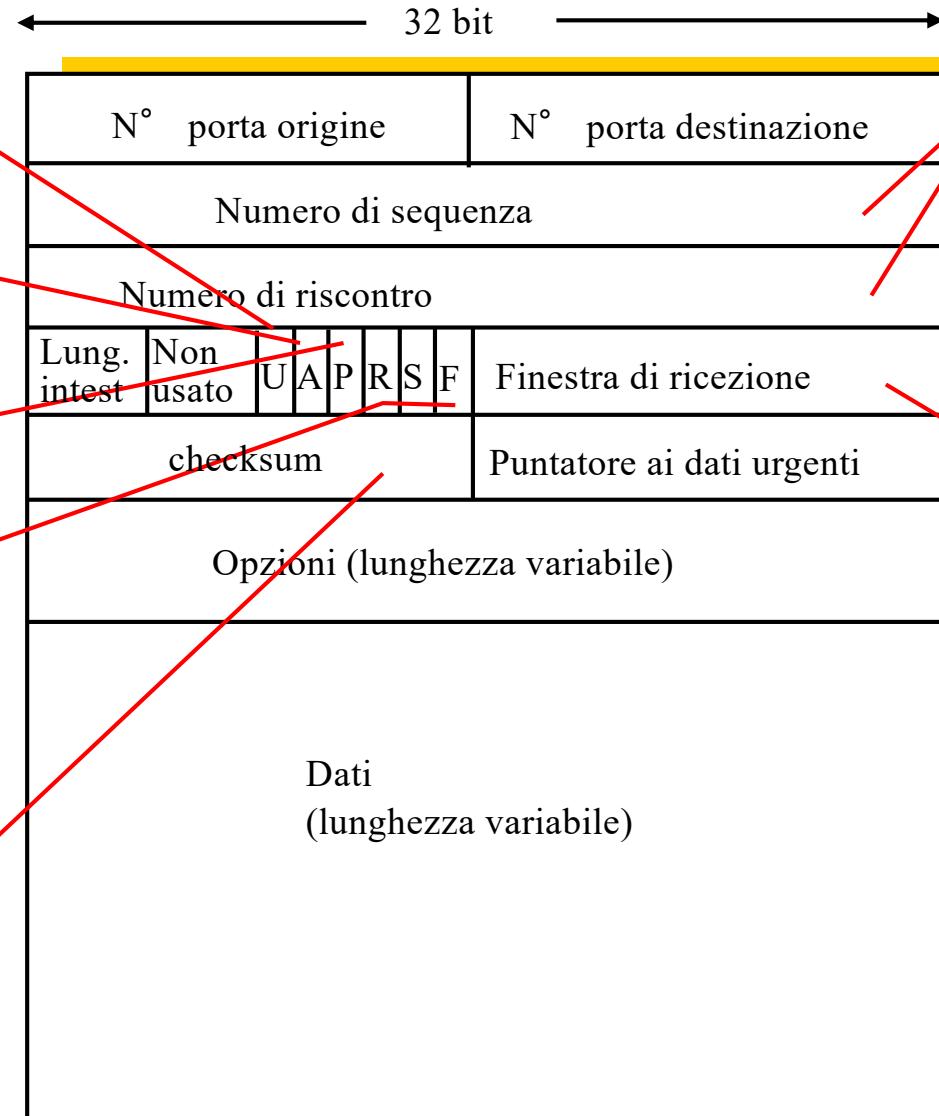
URG: dati urgenti
(generalmente non usato)

ACK: numero di
riscontro valido

PSH: invia i dati adesso
(generalmente non usato)

RST, SYN, FIN:
comandi per
impostare e chiudere
la connessione

Checksum
Internet
(come in UDP)



Conteggio per
byte di dati
(non segmenti!)

Numero di
byte che il
destinatario
desidera
accettare

Header TCP (cont.)

source e destination port

- le porte del sorgente e del destinatario, che permettono di identificare le applicazioni a cui sono destinati i dati (16 bit ciascuna)

sequence number (32 bit)

- il valore del primo byte trasmesso nel segmento; all'atto della connessione viene stabilito il **valore iniziale**, basato sul **clock** del trasmittente

acknowledge number (32 bit)

- il valore dell'ultimo byte riscontrato più uno (cioè del successivo atteso)

TCP header length (4 bit)

- il numero di **gruppi di 32 bit** contenuti nella intestazione; necessario perché sono previsti campi opzionali (non più di **60 byte**)

flag URG (urgent)

- il campo dati contiene **dati urgenti**, che devono essere passati alla applicazione **prima degli altri** ancora in attesa nei buffer (ad esempio: il CTRL^C in applicazioni di terminale remoto)

Header TCP (cont.)

- **flag ACK**

- il segmento trasporta un riscontro; tutti i segmenti **tranne il primo** dovrebbero averlo settato

- **flag PSH (push)**

- indica che l'applicativo ha richiesto l'invio dei dati **senza ulteriore attesa** (ed in ricezione deve essere fatto lo stesso)

- **flag RST (reset)**

- utilizzato per comunicare che la connessione deve essere **abortita**, o quando viene **rifiutata** una nuova connessione

- **flag SYN (synchronize)**

- utilizzato per stabilire una connessione; questi segmenti definiscono il **sequence number iniziale** per i due versi

- **flag FIN (finish)**

- utilizzato per comunicare alla controparte che non si hanno piu' dati da inviare e che si desidera **chiudere la connessione**; il doppio FIN con relativo riscontro genera il rilascio della connessione

Header TCP (cont.)

- **window size** (16 bit)

- la dimensione **in byte** dello **spazio disponibile** dei buffer in **ricezione**: il valore massimo è di **64 KB**
- le reti moderne molto veloci rendono questo limite **inefficiente**: è possibile utilizzare un **header opzionale** per accordarsi su una window size a **30 bit** (buffer fino ad **1 GB**)

- **checksum** (16 bit)

- obbligatoria per TCP (al contrario di UDP); anche in TCP la checksum viene calcolata su tutto il segmento più uno **pseudo header** che riporta gli indirizzi IP di sorgente e destinazione

- **urgent pointer** (16 bit)

- definisce **l'offset** dell'ultimo byte facente parte dei **dati urgenti** quando la flag URG è settata

Header opzionali

- Le opzioni sono definite da una **lunghezza**, un **tipo**, ed i **dati** relativi; sono definite diverse opzioni, tra cui:
 - **padding**: necessario in presenza di opzioni per rendere il **campo header** nel suo complesso un **multiplo di 32 bit**
 - **MSS**: utilizzato con i segmenti **SYN** per determinare il MSS scambiandosi i valori di **MTU** ed **MRU**
 - **window scale**: utilizzata per definire la dimensione della finestra fino a 30 bit
 - **selective acknowledge**: TCP utilizza normalmente il **go-back-N**; questa opzione permette di utilizzare il **selective reject**
 - **timestamp**: utilizzata per valutare (a livello di trasporto) il **round trip time** e poter definire valori **opportuni** per i **timer interni**

Flusso di dati interattivi

Si considera il caso di una connessione interattiva (es. telnet):

- Non si possono accumulare i dati ma occorre inviare segmenti piccoli
- Il 90% dei segmenti telnet porta circa 10 byte

Nel caso limite si ha un segmento per ogni carattere battuto con il ricevente che genera un echo del medesimo carattere:

Segmento dal client col carattere battuto ($26\text{IP} + 20\text{TCP} + 1\text{byte} = 47\text{byte}$)

Segmento di ack dal server al client (46 byte)

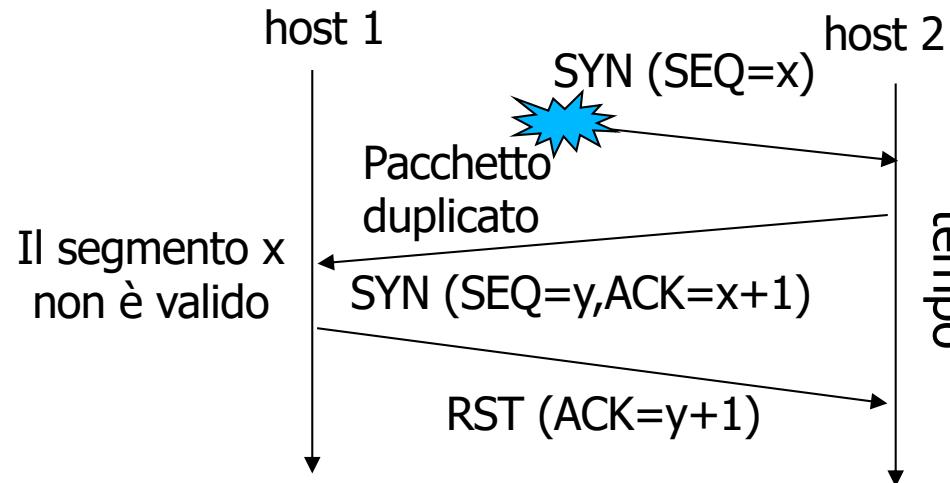
Segmento di echo dal server (47 byte)

Segmento di ack dal client (46 byte)

In totale si userebbero 186 byte in 4 segmenti TCP per 1 carattere!!

Duplicato della richiesta di attivazione

I pacchetti possono essere memorizzati e ricomparire nella rete

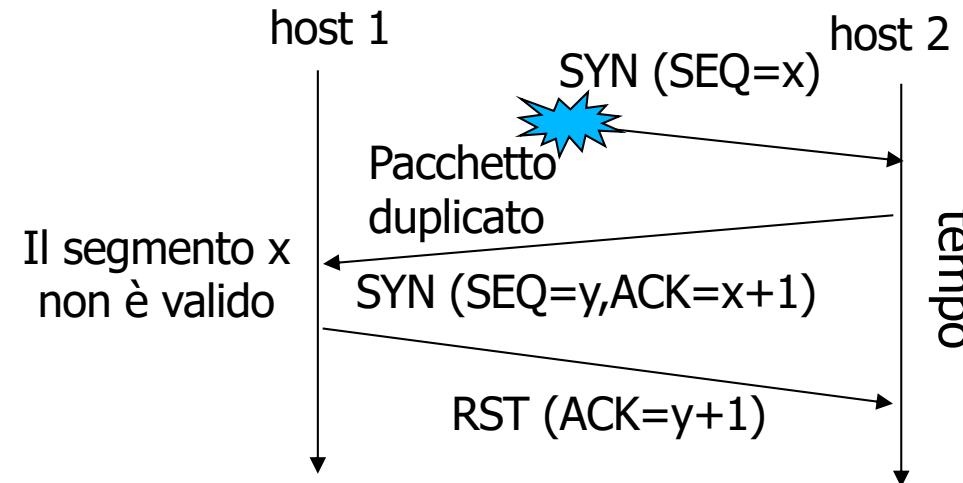


La numerazione iniziale di x e y è fatta con un orologio locale (tick=4 μ s)

L'intervallo dei numeri di sequenza (32 bit) garantisce che non venga riutilizzato lo stesso numero prima di qualche ora

A causa del **time to live** dei pacchetti IP, segmenti con lo stesso numero non possono coesistere sulla rete

Duplicato della richiesta di attivazione



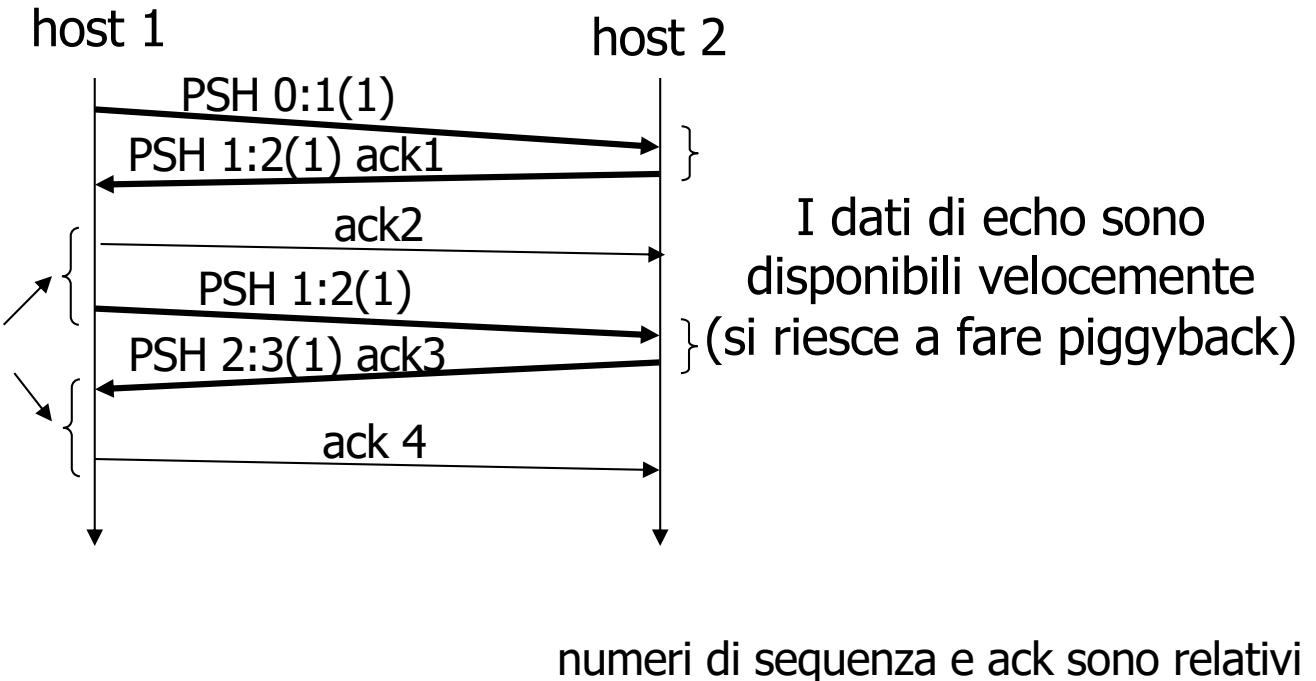
Se arriva a destinazione un duplicato della richiesta di attivazione, il destinatario risponde come prima, ma il mittente, che sa di non aver richiesto una seconda connessione, lo informa dell'errore.

Ack ritardati

Normalmente il TCP non invia un ack istantaneamente ma ritarda l'invio sperando di avere dati da spedire con l'ack (**ACK piggyback**).

Molte implementazioni usano un ritardo di 200ms.

ack ritardati
scade il timeout
prima che un dato
sia disponibile



Sequence e Ack Numbers

- TCP usa un meccanismo di sliding window di tipo go-back-n con timeout.
- Se il timeout scade, il segmento è ritrasmesso.
- Ogni byte del flusso TCP è numerato con un numero a 32 bit
- Si noti che le dimensioni della finestra scorrevole e i valori degli ack **sono espressi in numero di byte e non in numero di segmenti.**

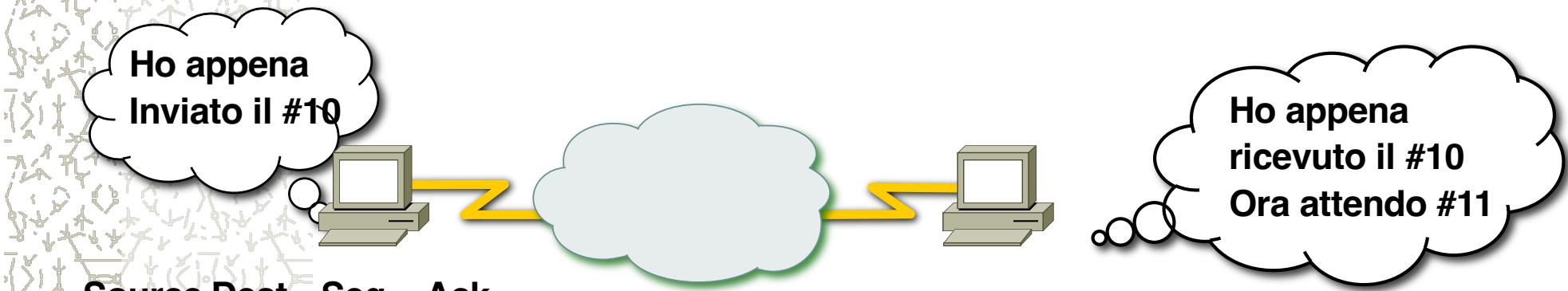
Sequence e Ack Numbers



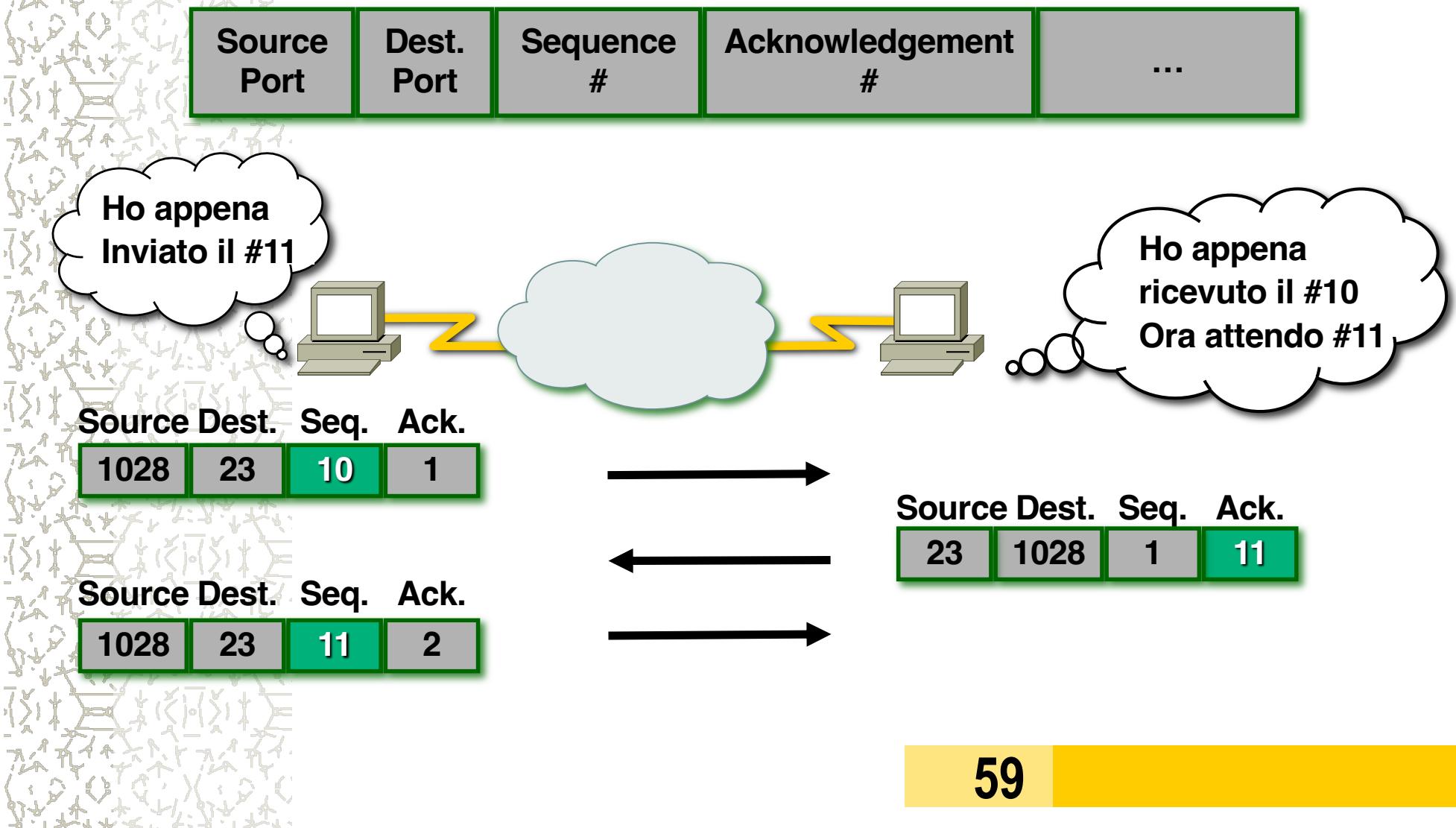
Ho appena
Inviato il #10

Source	Dest.	Seq.	Ack.
1028	23	10	1

Sequence e Ack Numbers



Sequence e Ack Numbers



Sequence e Ack Numbers



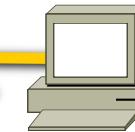
Ho appena
Inviato il #11



Source Dest. Seq. Ack.

1028	23	10	1
------	----	----	---

Ho appena
ricevuto il #11
Ora attendo #12



Source Dest. Seq. Ack.

1028	23	11	2
------	----	----	---



Source Dest. Seq. Ack.

23	1028	1	11
----	------	---	----

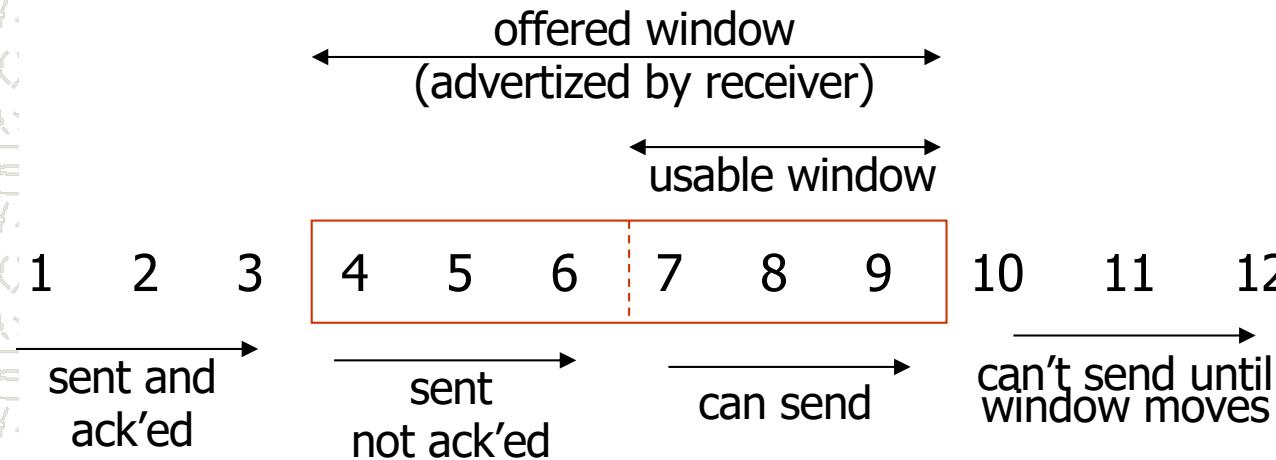
Source Dest. Seq. Ack.

23	1028	2	12
----	------	---	----

Trasmissione di flussi di dati

Viene utilizzato un protocollo a finestra scorrevole (sliding window)

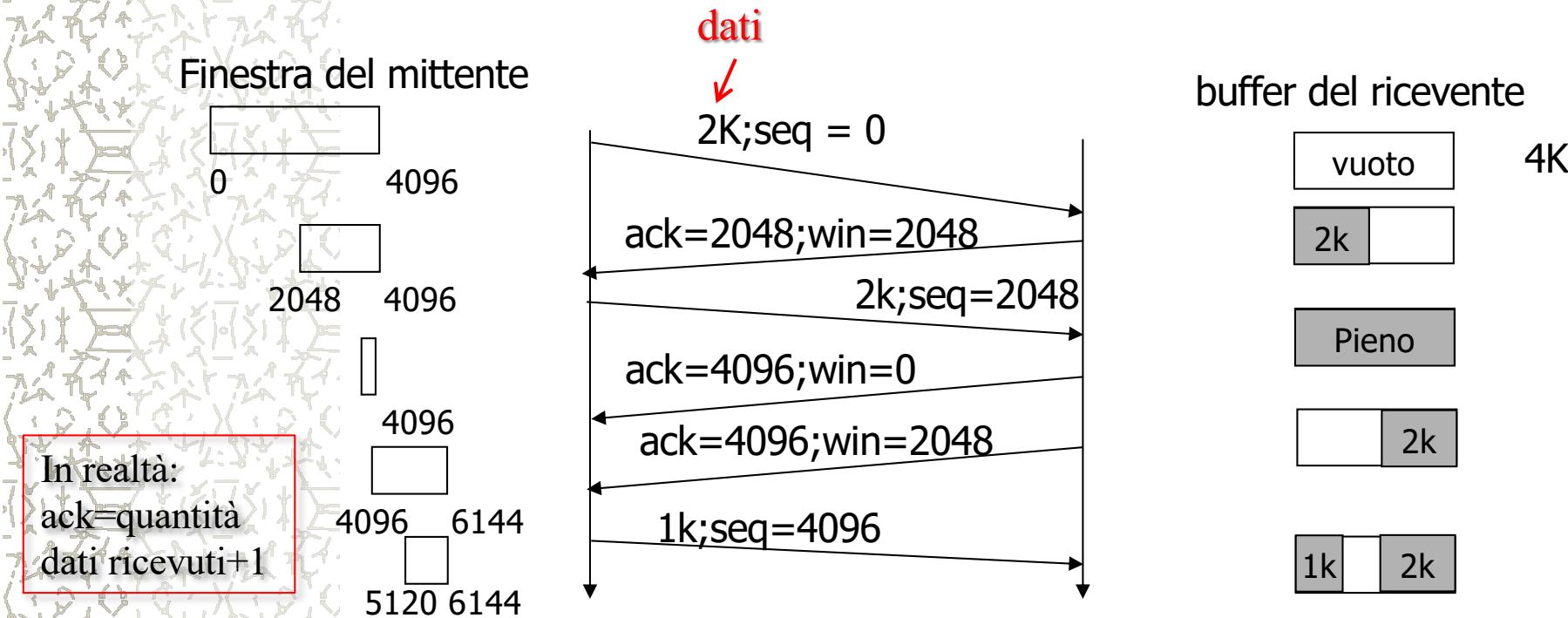
Il ricevente indica la dimensione della finestra che può gestire in un dato momento



La finestra di dati trasmissibili ancora senza aspettare l'ack è ottenuta dall'ampiezza della finestra e dal numero dell'ultimo byte ricevuto

Trasmissione di flussi di dati

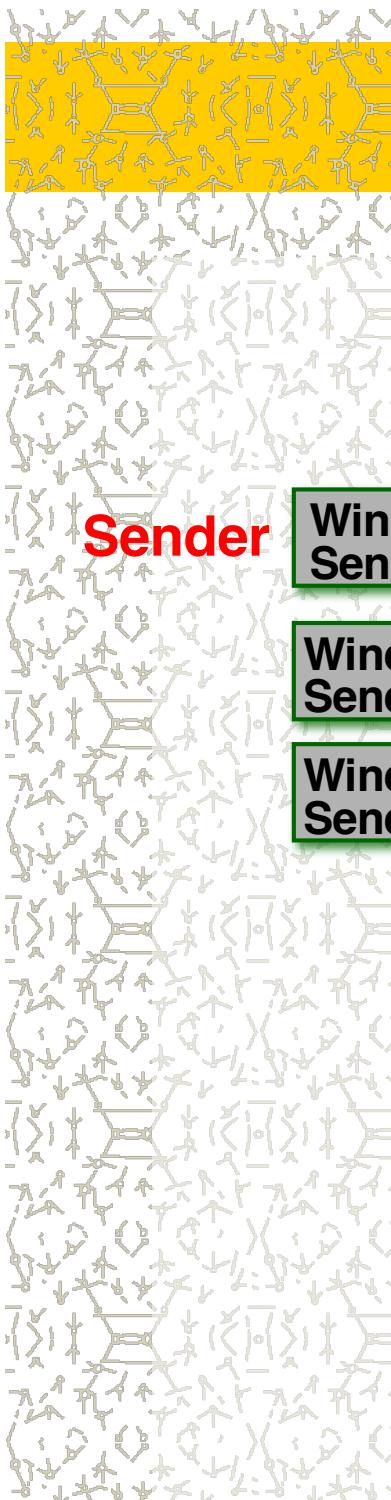
Nell'esempio, le parti si sono preventivamente accordate su un buffer di 4K a destinazione.



Se il ricevente indica una finestra 0 il mittente non può trasmettere dati.

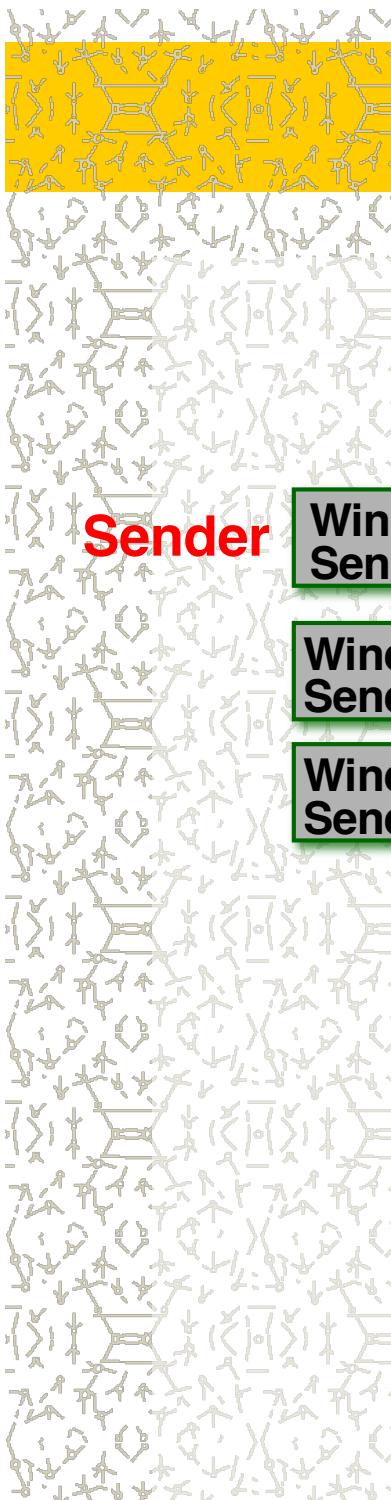
Il mittente può inviare un segmento di un byte per forzare il destinatario a indicare il prossimo byte atteso e l'ampiezza della finestra (per non rimanere in attesa infinita se si perdono pacchetti) - **timer di persistenza**

Windowing



Receiver

Windowing



Sender

Window size = 3
Send 1

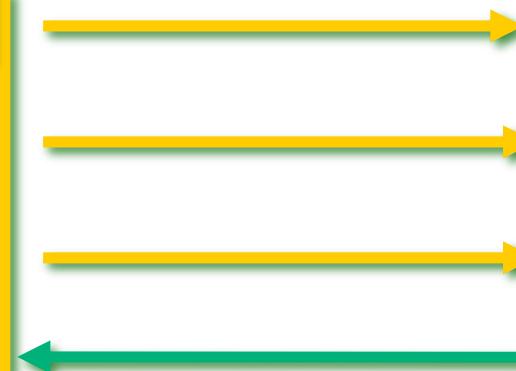
Window size = 3
Send 2

Window size = 3
Send 3

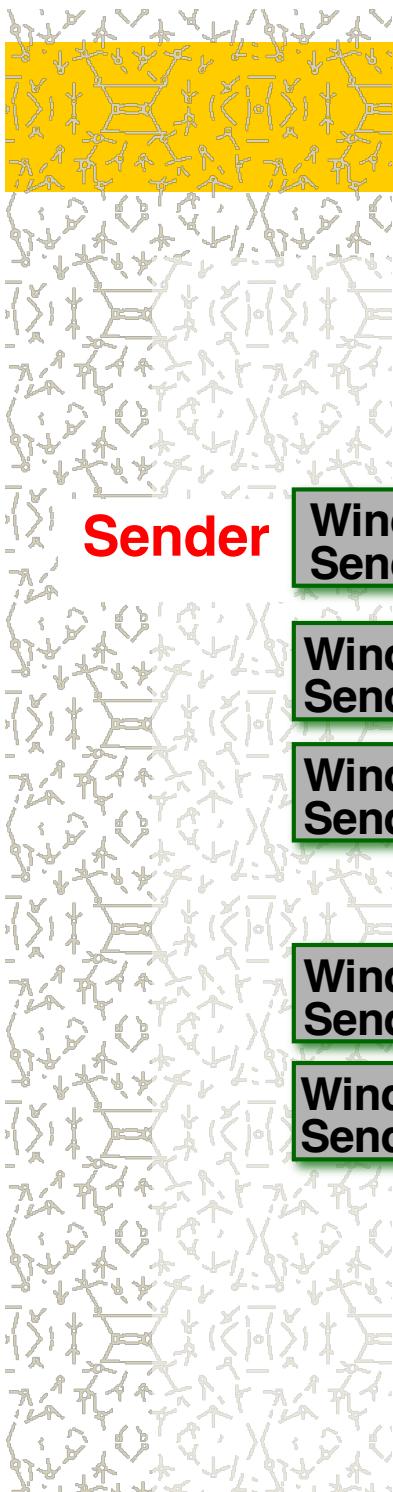
Receiver

ACK 3
Window size = 2

Invio 3 perduto



Windowing



Sender

Window size = 3
Send 1

Window size = 3
Send 2

Window size = 3
Send 3

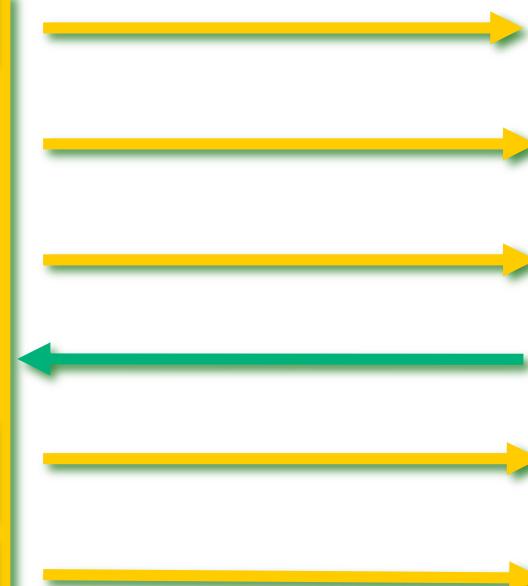
Window size = 3
Send 3

Window size = 3
Send 4

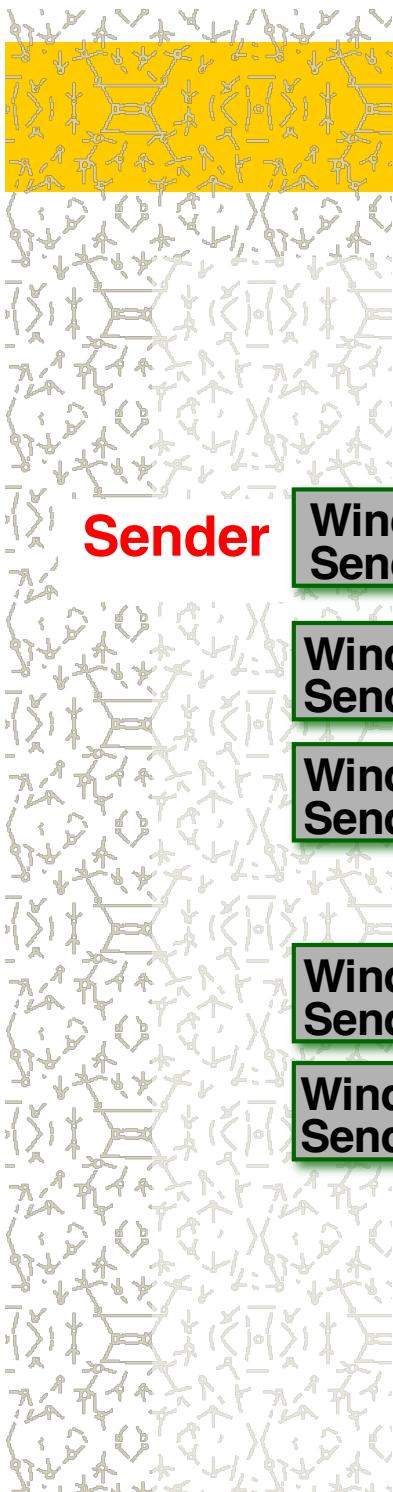
Receiver

ACK 3
Window size = 2

Invio 3
perduto



Windowing



Sender

Window size = 3
Send 1

Window size = 3
Send 2

Window size = 3
Send 3

Window size = 3
Send 3

Window size = 3
Send 4

Receiver

ACK 3
Window size = 2

ACK 5
Window size = 2

Invio 3
perduto

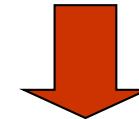
TCP Timeout e ritrasmissione

TCP utilizza un timeout di attesa dell'ack dopo di che provvede alla ritrasmissione dei dati.

Il problema è determinare il valore del timeout migliore in quanto i ritardi possono essere molto variabili nel tempo sulla rete:

Se il timeout è troppo piccolo si fanno ritrasmissioni inutili

Se il timeout è troppo elevato si avranno ritardi di trasmissione



Si utilizza un algoritmo di stima del migliore timeout basato sulla misura del Round-Trip Time (RTT)

Stima del Timeout

Il timeout non è un parametro fisso; anzi, il software di rete TCP ne esegue una stima ogni volta che arriva un ACK. In questo modo, il software di rete TCP si tiene aggiornato sullo stato della connessione e della rete. Il timeout è infatti calcolato come la media pesata di vari campionamenti, eseguiti durante la connessione dal software di rete del mittente, del tempo intercorso fra l'invio di un pacchetto e la ricezione del corrispondente ACK (RTT).

Per ogni connessione si calcola una stima di RTT, aggiornandola per ogni pacchetto con

$$\text{RTT stimato} \quad \text{RTT reale calcolato su un pacchetto}$$
$$\text{RTT}_i = \alpha \text{ RTT}_{i-1} + (1 - \alpha) T_{rtt}(\text{pkt}_i)$$

Si stima poi la deviazione media

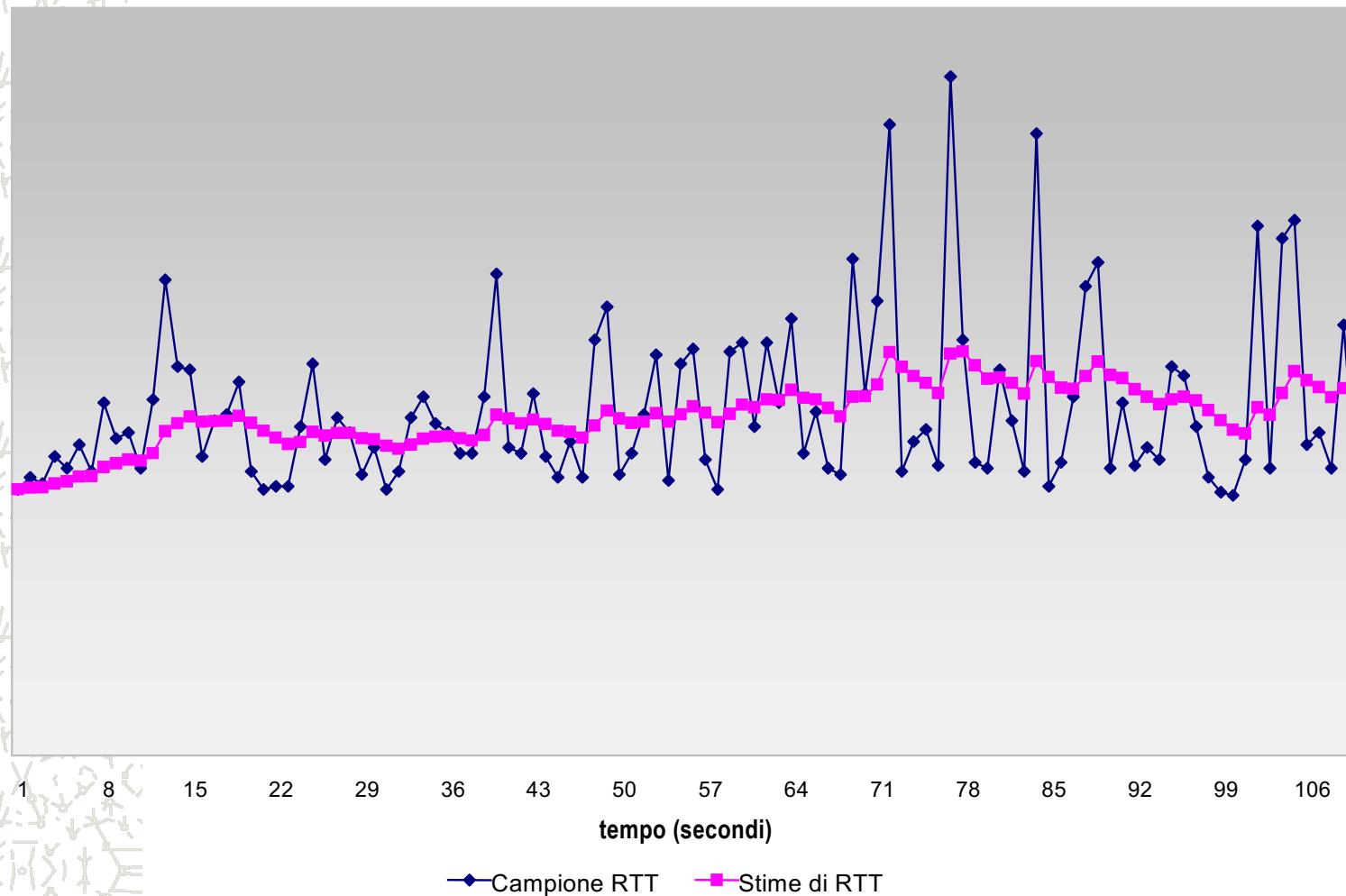
$$D_i = \alpha D_{i-1} + (1 - \alpha) |RTT_i - T_{rtt}(\text{pkt}_i)|$$

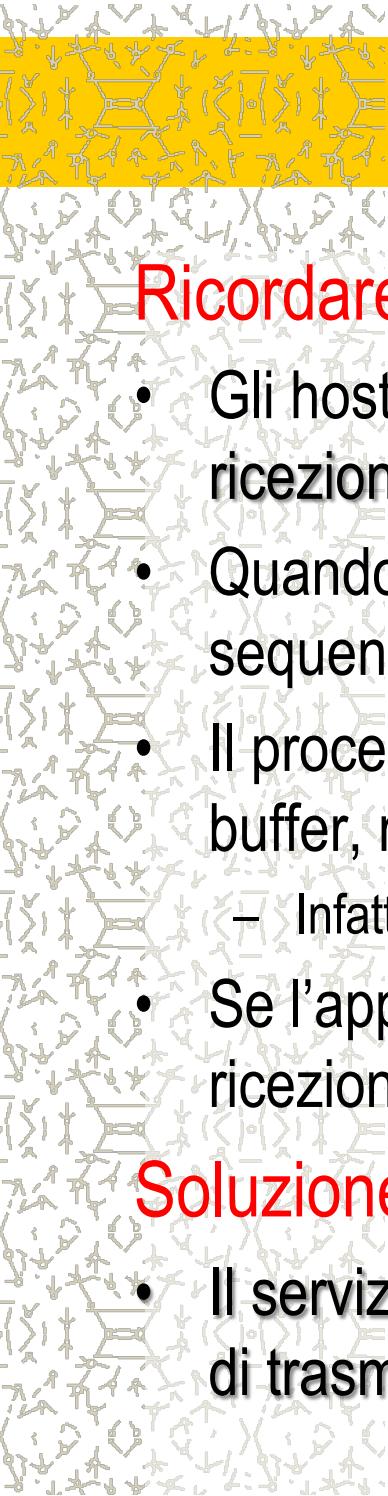
E si sceglie

$$\text{timeout} = \text{RTT} + 4 * D$$

Esempio di stima di RTT:

RTT: gaia.cs.umass.edu e fantasia.eurecom.fr





Controllo del flusso

Ricordare:

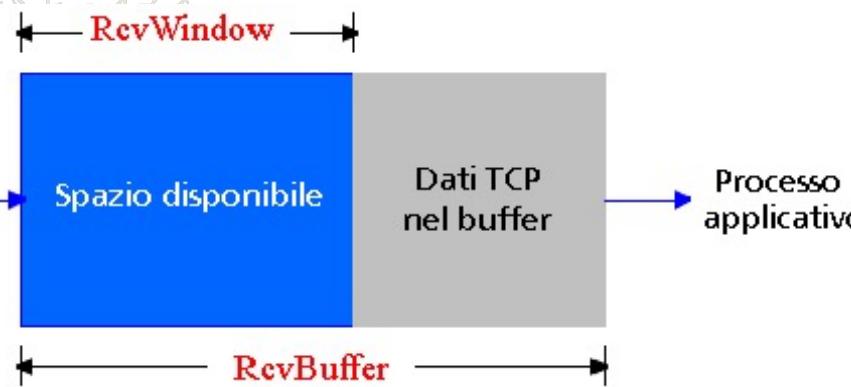
- Gli host della connessione TCP riservano ognuno un **buffer di ricezione** per la connessione
- Quando la connessione TCP riceve byte che sono corretti e in sequenza, colloca i dati nel buffer di ricezione.
- Il processo dell'applicazione associato leggerà i dati da questo buffer, ma non necessariamente nell'istante di arrivo.
 - Infatti l'applicazione potrebbe essere occupata in altri compiti
- Se l'applicazione è lenta, il sender può saturare il buffer di ricezione, inviando troppi dati e in fretta.

Soluzione

- Il servizio di controllo del flusso consente di adattare la velocità di trasmissione a quella in ricezione

TCP: controllo di flusso

- Il lato ricevente della connessione TCP ha un buffer di ricezione:



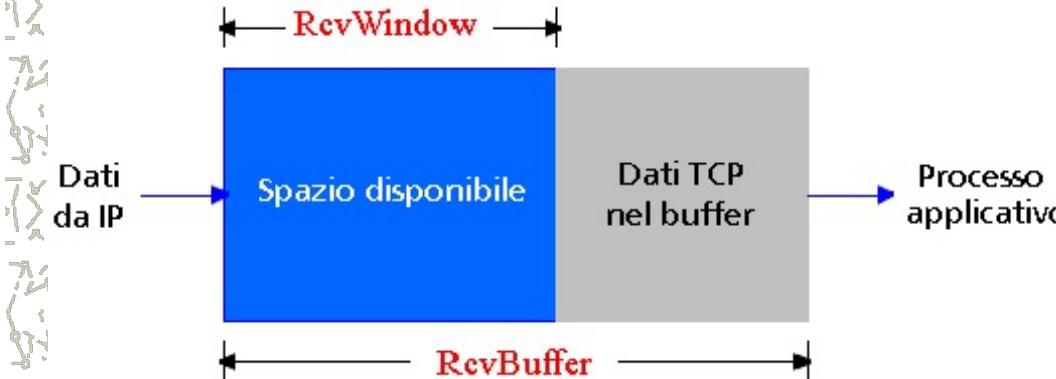
Il processo applicativo potrebbe essere rallentato dalla lettura nel buffer

Controllo di flusso

Il mittente non vuole sovraccaricare il buffer del destinatario trasmettendo troppi dati, troppo velocemente.

- Servizio di corrispondenza delle velocità: la frequenza d'invio deve corrispondere alla frequenza di lettura dell'applicazione ricevente

TCP: funzionamento del controllo di flusso



(supponiamo che il destinatario TCP scarti i segmenti fuori sequenza)

Spazio disponibile nel buffer =

= **RcvWindow** =

= **RcvBuffer** - [LastByteRcvd -
LastByteRead]

- Il receiver comunica lo spazio disponibile includendo il valore di **RcvWindow** (finestra di ricezione) nei segmenti
- Il sender limita i dati non riscontrati a **RcvWindow**
 - garantisce che il buffer di ricezione non vada in overflow

Remarking: Principi sul controllo di congestione

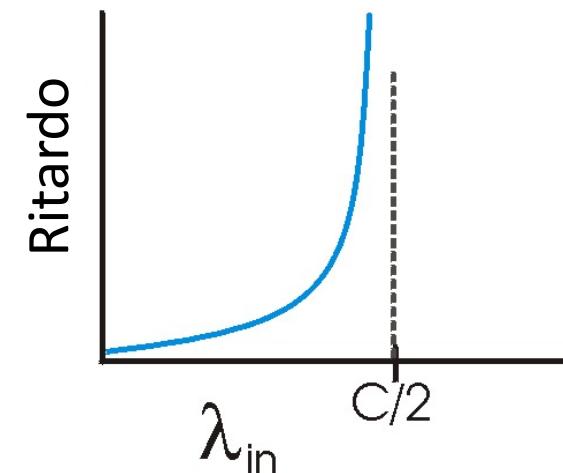
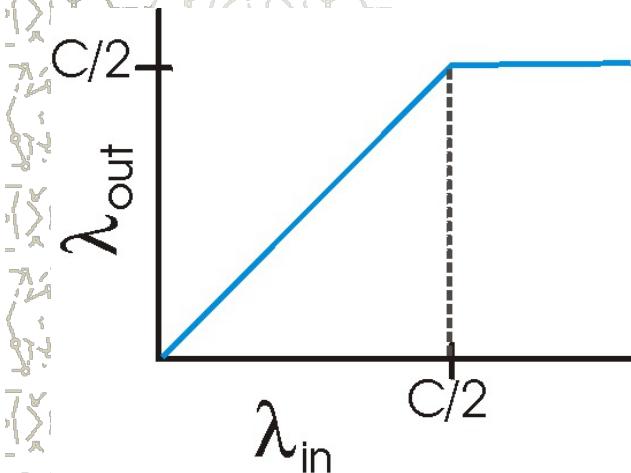
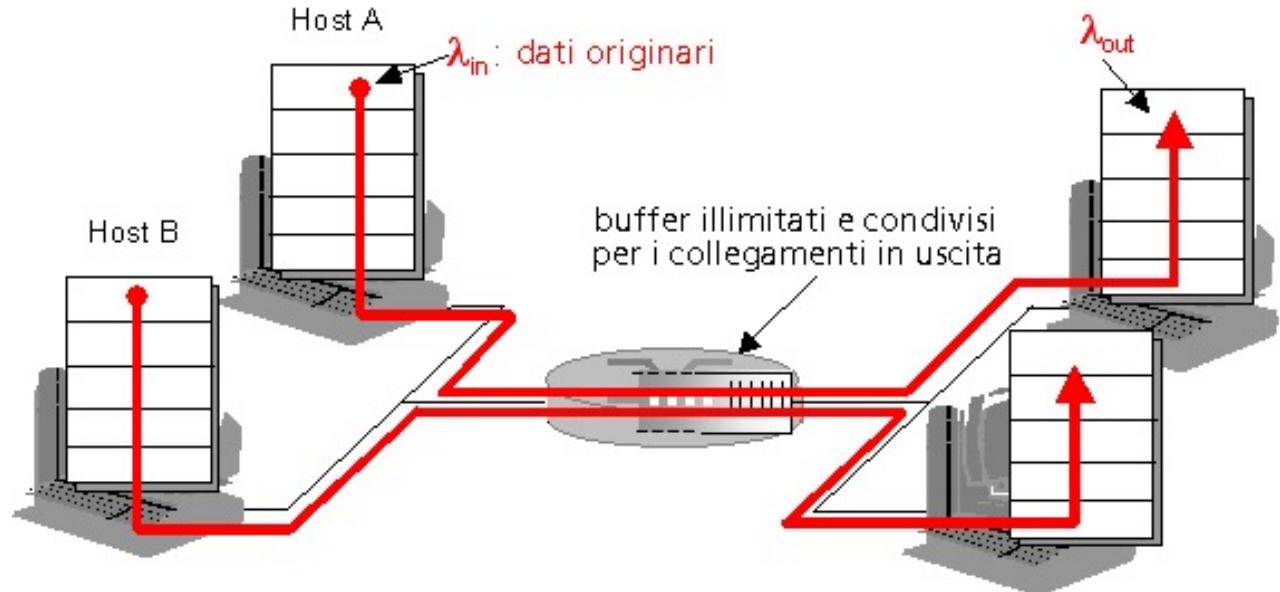
Congestione:

Un sender può anche essere «strozzato» a causa della congestione entro la rete IP.

- informalmente: “troppe sorgenti trasmettono troppi dati, a una velocità talmente elevata che la *rete* non è in grado di gestirli”
- manifestazioni:
 - pacchetti smarriti (overflow nei buffer dei router)
 - lunghi ritardi (accodamento nei buffer dei router)
- tra i dieci problemi più importanti del networking!

Cause/costi della congestione: scenario 1

- due mittenti, due destinatari
- un router con buffer illimitati
- nessuna ritrasmissione
- Canale a C byte/s
- A e B inviano a C byte/s
- Appena $\lambda_{in} >= C/2$ il throughput in uscita (λ_{out}) rimane $C/2$

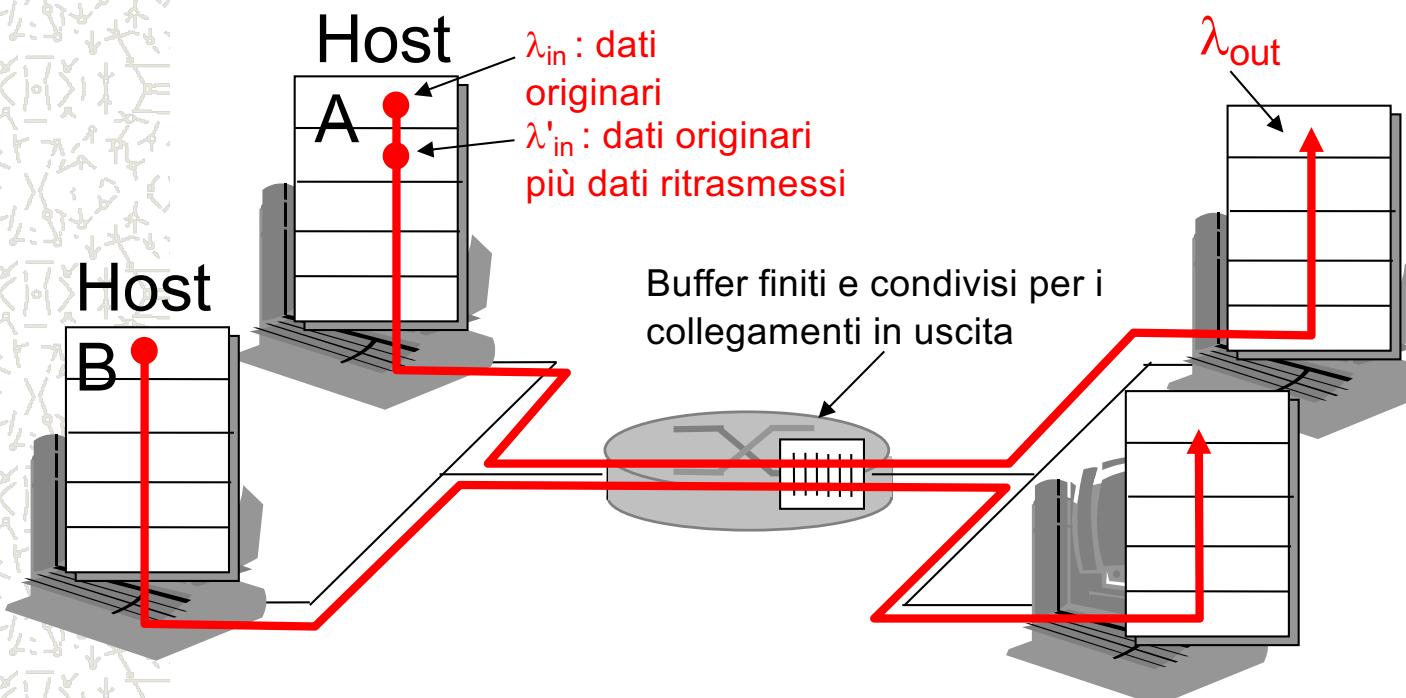


Nonostante il buffer è illimitato, il costo della congestione è:

- grandi ritardi per λ_{in} prossimo a $C/2$

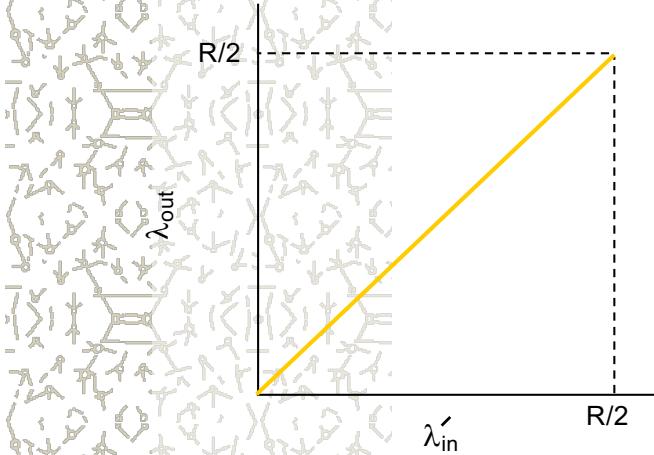
Cause/costi della congestione: scenario 2

- un router, buffer *finiti*
- I pacchetti verranno scartati quando il buffer sarà pieno
- il mittente ritrasmette il pacchetto perduto se la connessione è di tipo affidabile

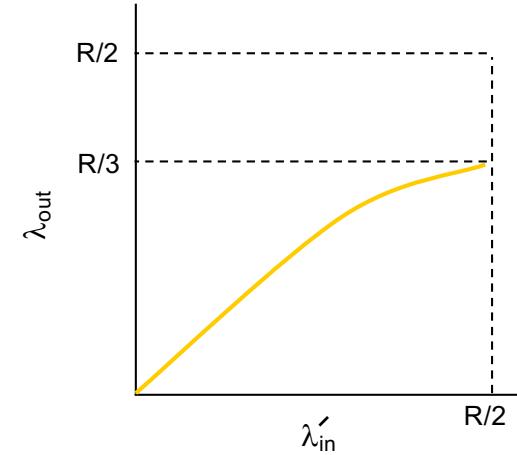


Cause/costi della congestione: scenario 2

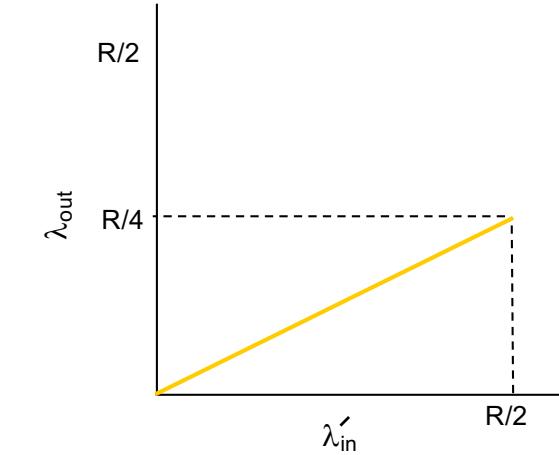
- a. Fig. a - nessuna perdita di pacchetti $\rightarrow \lambda_{\text{out}} = \lambda_{\text{in}}$
- b. Fig. b - Il sender ritrasmette solo quando è sicuro che il pacchetto è andato perso
- c. Fig. c - La ritrasmissione di un **pacchetto ritardato** (non perduto) rende λ'_{in} più grande (rispetto al caso perfetto) per lo stesso λ_{out} . Fig.c mostra il caso in cui ogni pacchetto è ritrasmesso 2 volte dal router.



a.



b.



c.

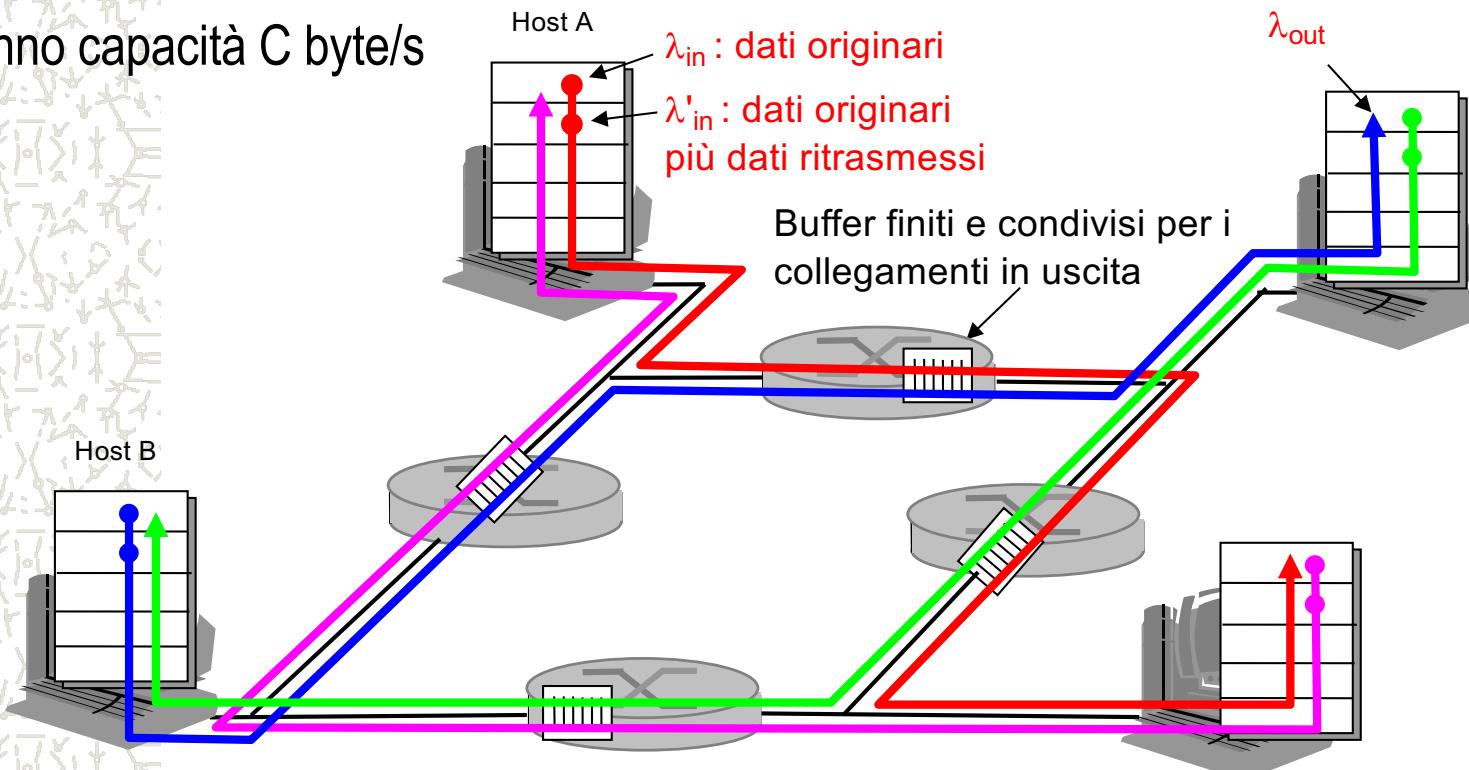
“Costi” della congestione:

- Ritrasmissioni per pacchetti scartati a causa del sovraccarico del buffer
- Ritrasmissioni non necessarie: il collegamento trasporta più copie non necessarie del pacchetto

Cause/costi della congestione: scenario 3

- Quattro mittenti
- Percorsi multihop
- timeout/ritrasmessione
- Tutti i link dei router
hanno capacità C byte/s

D: Che cosa accade quando λ_{in}
e λ'_{in} aumentano?

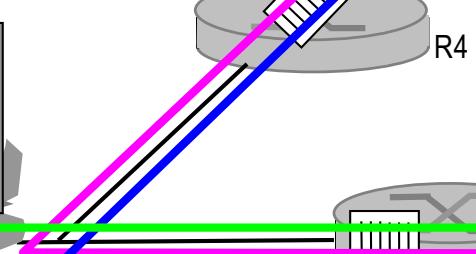
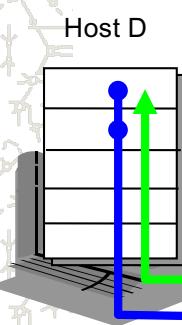
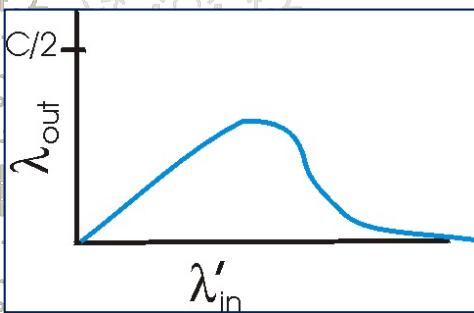


Cause/costi della congestione: scenario 3

Consideriamo la connessione da A a C passando per R1 e R2

Caso 1: Traffico basso

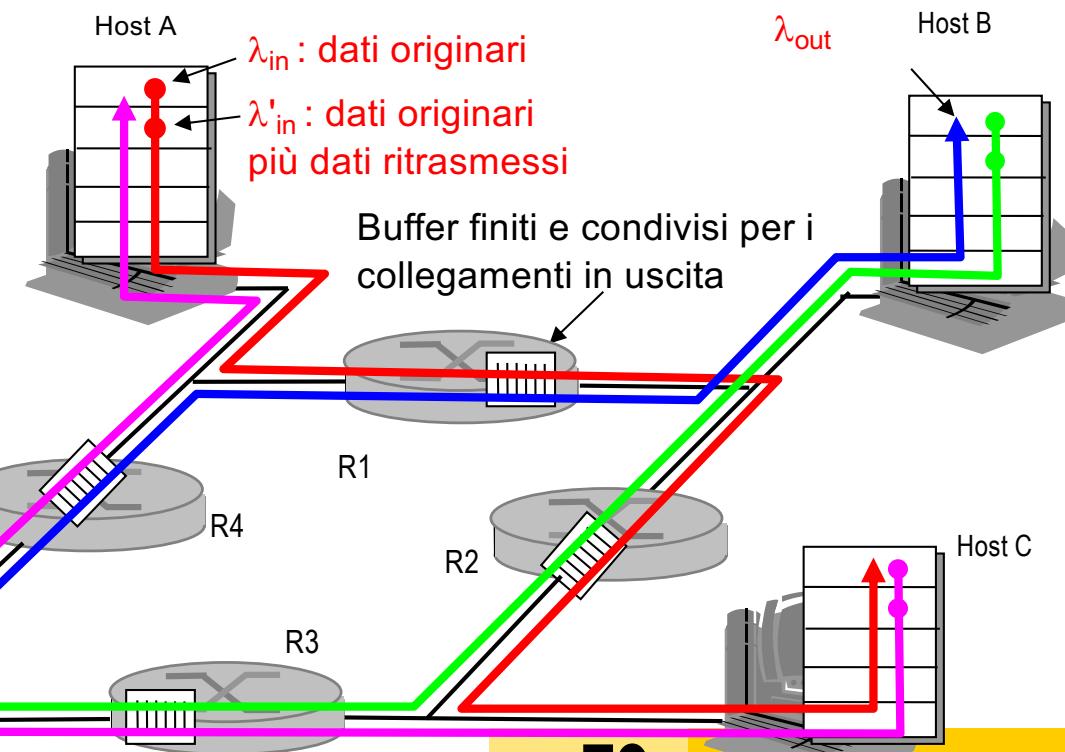
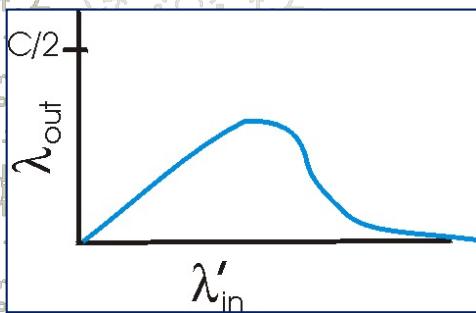
- La connessione A-C condivide R1 con D-B e R2 con B-D
 - Per λ_{in} piccoli $\lambda_{out} = \lambda_{in}$ in quanto il sovraccarico dei buffer è raro
 - Per λ'_{in} poco > di λ_{in} , $\lambda_{out} \approx \lambda_{in}$



Cause/costi della congestione: scenario 3

Caso 2: Traffico alto

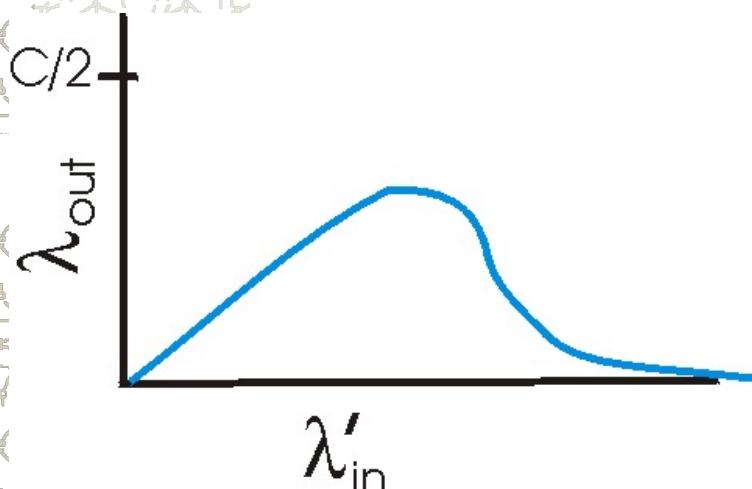
- Se λ'_{in} è grande per tutte le connessioni, la velocità di arrivo del traffico B-D a R2 può essere più grande di quella del traffico A-C.
- Entrambi i traffici competono in R2 (buffer limitato), quindi il traffico A-C che con successo attraversa R2 diminuisce sempre più quando il carico offerto da B-D continua ad aumentare
- Al limite il throughput di A-C in R2 va a zero.



Cause/costi della congestione: scenario 3

Un altro “costo” della congestione:

- Ogni volta che un pacchetto è perso nel router del secondo salto (R2), il lavoro fatto dal router del primo salto (R1) nell'instradare il pacchetto al router R2 finisce per essere inutile.
- Quando il pacchetto viene scartato, la capacità trasmissiva utilizzata sui collegamenti di upstream per instradare il pacchetto risulta sprecata!



Approcci al controllo della congestione

I due principali approcci al controllo della congestione:

Controllo di congestione punto-punto:

- nessun supporto esplicito dalla rete
- la congestione è dedotta osservando le perdite e i ritardi nei sistemi terminali
- metodo adottato da TCP

Controllo di congestione assistito dalla rete:

- i router forniscono un feedback ai sistemi terminali
 - un singolo bit per indicare la congestione (SNA, DECbit, TCP/IP ECN, ATM)
 - comunicare in modo esplicito al mittente la frequenza trasmissiva

Controllo di congestione nel TCP

Il TCP adatta la velocità di trasmissione in funzione della congestione in rete percepita.

Se un sender percepisce che c'è poca congestione tra sé e la destinazione, allora aumenta il suo ritmo di trasmissione.

Se il sender percepisce che c'è congestione lungo il percorso, allora riduce il suo ritmo di invio

3 problemi

Come il
sender limita il
ritmo di invio

Come il sender
percepisce la
congestione

Quale algoritmo usare per
cambiare il ritmo di invio in
funzione della congestione

Come il sender limita il ritmo di invio

Si introduce la finestra di congestione (CongWin)

- Essa impone una limitazione addizionale alla quantità di traffico che un host può inviare in una connessione.
- In particolare, l'ammontare dei dati non riscontrati che un host può avere non deve superare il minimo tra CongWin e RcvWin

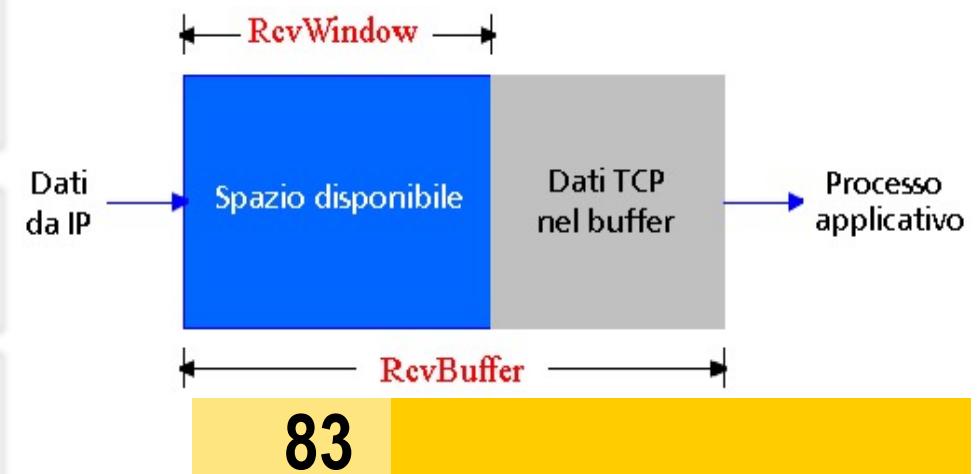
LastByteSent-LastByteAcked
 $\leq \min(\text{CongWin}, \text{RcvWindow})$

Questo vincolo limita la quantità di dati non riscontrati al mittente e quindi indirettamente limita il ritmo di invio del sender.

Variando CongWin, il sender può variare il ritmo di invio

CongWin è funzione della congestione percepita

Buffer di ricezione usato nel controllo del flusso



Come il sender percepisce la congestione

Quando c'è congestione uno o più buffer dei router lungo il percorso del collegamento traboccano, causando la perdita di pacchetti.

Il pacchetto perso, a sua volta, dà luogo ad un **evento di perdita** al sender, che si può esprimere con:

- timeout
- ricezione di 3 ACK duplicati

Questi eventi vengono percepiti dal sender come congestione nel percorso tra sender e receiver.

Algoritmo di controllo della cong. di TCP

L'Algoritmo ha tre componenti principali:

- Incremento additivo e decremento moltiplicativo (AIMD)
- Partenza lenta (slow start)
- Reazione a eventi di timeout

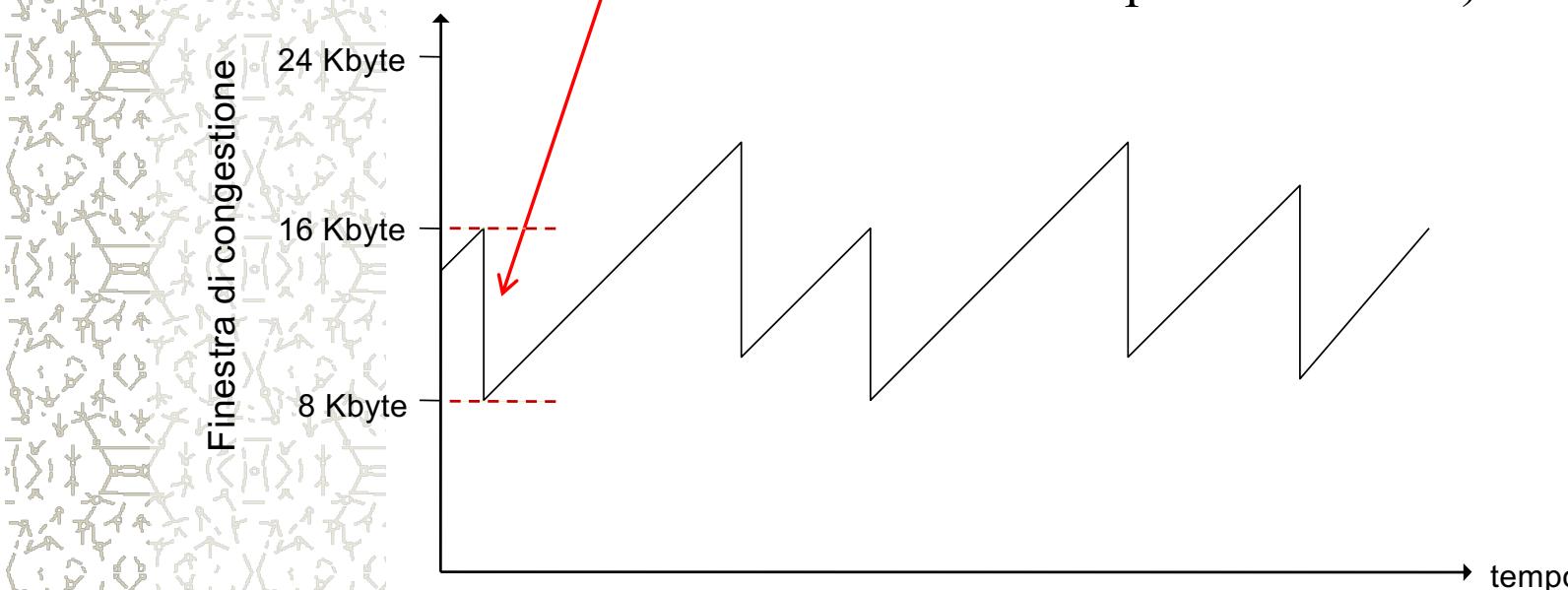
Incremento additivo e decremento moltiplicativo (AIMD)

Decremento moltiplicativo: riduce a metà **CongWin** dopo un evento di perdita.

CongWin non può scendere al di sotto di una soglia (MSS)

Incremento additivo: aumenta **CongWin** lentamente, con cautela.

In particolare, il sender aumenta CongWin di 1 MSS (ecco perché additivo)



Controllo di congestione AIMD

Partenza lenta

- Quando si inizia una connessione, **CongWin** è impostato ad 1 MSS
- Dando luogo a un ritmo iniziale di invio pari a MSS/RTT
 - Esempio: MSS = 500 byte
 - RTT = 200 msec
 - Ritmo iniziale = 20 kbps
- Ma la larghezza di banda disponibile potrebbe essere molto maggiore,
- e quindi sarebbe un peccato aumentare di 1 MSS alla volta
- Quindi il sender aumenta il suo ritmo in modo esponenziale **raddoppiando** il valore di **CongWin** ogni RTT
- Il sender continua ad aumentare il suo ritmo fino a quando si verifica un evento di perdita, al che CongWin viene dimezzato e poi cresce linearmente (seguendo AIMD)
- Quindi, inizialmente il sender inizia trasmettendo ad un ritmo lento (da cui Partenza lenta), ma aumenta il proprio ritmo di invio a velocità esponenziale.

Reazione a eventi di timeout

Riflessione:

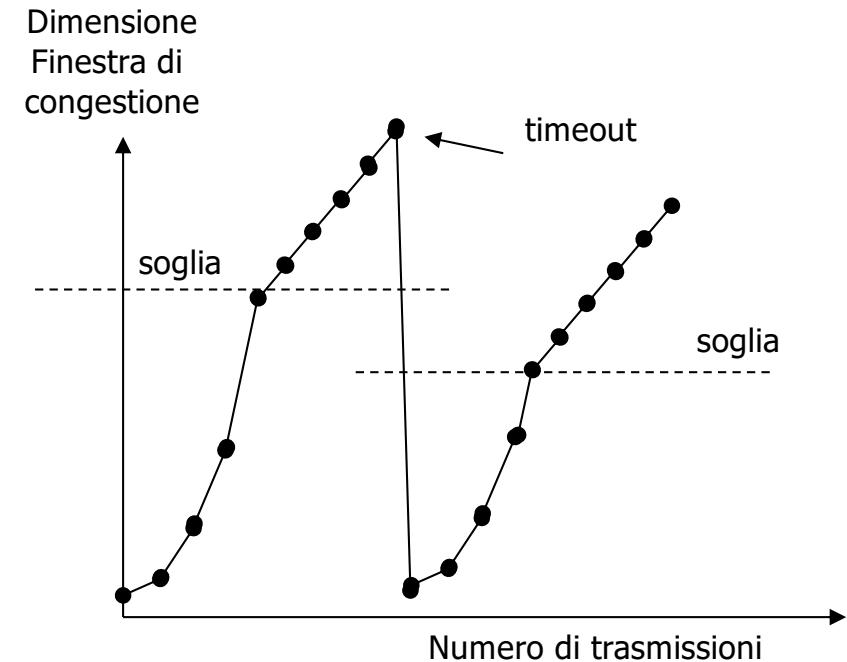
- 3 ACK duplicati indicano la capacità della rete di consegnare qualche segmento
- Un timeout prima di 3 ACK duplicati è “più allarmante”

Soluzione:

- Dopo un evento di timeout, il sender entra nella fase di partenza lenta
- Durante la partenza lenta, il sender aumenta il valore di CongWin a velocità esponenziale finché CongWin raggiunge una soglia (Threshold)
- A questo punto, il TCP entra in una fase di prevenzione, durante la quale CongWin cresce linearmente (AIMD)

Riassunto: il controllo della congestione TCP

- Quando **CongWin** è sotto la soglia (**Threshold**), il mittente è nella fase di **partenza lenta**; la finestra cresce in modo esponenziale.
- Quando **CongWin** è sopra la soglia, il mittente è nella fase di **congestion avoidance**; la finestra cresce in modo lineare.
- Quando si verificano **tre ACK duplicati**, il valore di **Threshold** viene impostato a **CongWin/2** e **CongWin** viene impostata al valore di **Threshold**.
- Quando si verifica un **timeout**, il valore di **Threshold** viene impostato a **CongWin/2** e **CongWin** è impostata a 1 MSS (di nuovo **partenza lenta**).



Ricordare:
Partenza Lenta in:
1. fase iniziale
2. dopo il timeout

UDP: User Datagram Protocol [RFC 768]

- Protocollo di trasporto “senza fronzoli”
- Servizio di consegna “a massimo sforzo”, i segmenti UDP possono essere:
 - perduti
 - consegnati fuori sequenza all’applicazione
- ***Senza connessione:***
 - no handshaking tra mittente e destinatario UDP
 - ogni segmento UDP è gestito indipendentemente dagli altri

Perché esiste UDP?

- Nessuna connessione stabilita (che potrebbe aggiungere un ritardo)
- Semplice: nessuno stato di connessione nel mittente e destinatario
- Intestazioni di segmento corte
- Senza controllo di congestione: UDP può sparare dati a raffica

User Datagram Protocol

- UDP implementa un servizio di consegna inaffidabile dei dati a destinazione
- UDP riceve i dati dalla applicazione e vi aggiunge un **header di 8 byte**, costruendo così il **segmento** da inviare
- L'applicazione specifica (l'indirizzo di rete e) la **porta di destinazione**, ed in ricezione UDP recapita il campo dati al destinatario
- UDP non si preoccupa di sapere nulla sul **destino** del segmento inviato, nè comunica alla applicazione qualsiasi informazione
- Di fatto costituisce semplicemente una **interfaccia ad IP** (che fornisce lo stesso tipo di servizio), con l'aggiunta di fare **multiplexing** del traffico delle applicazioni su IP
 - tramite il meccanismo delle porte a cui sono associate le applicazioni, di fatto UDP **realizza un multiplexing** dei dati delle diverse applicazioni su IP

Orientato al datagramma

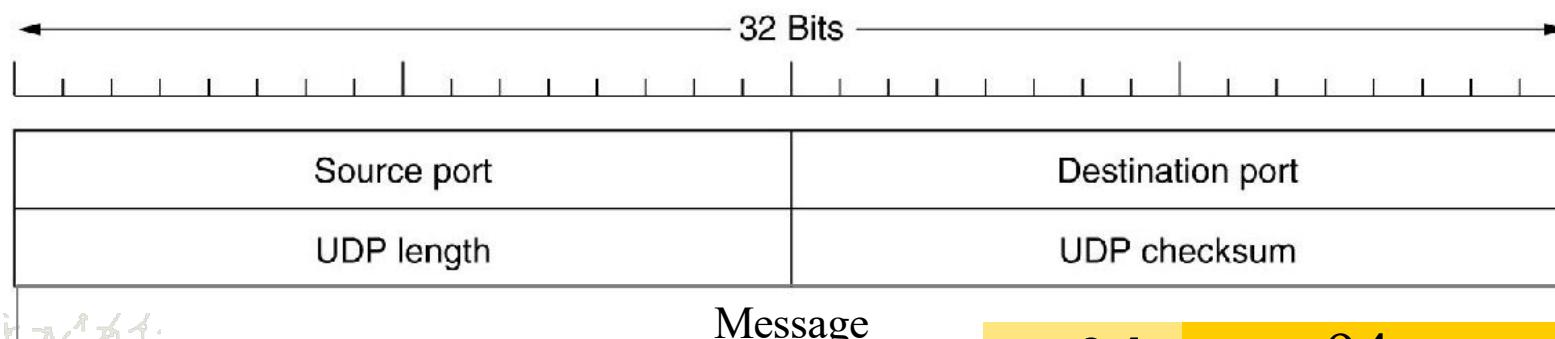
- A differenza di TCP, UDP si occupa di **un datagramma per volta**
 - quando un'applicazione passa dati ad UDP, UDP li maneggia in un **unico segmento**, senza suddividerlo in pezzi
 - il segmento di massime dimensioni che UDP può gestire **dove** stare interamente nel **campo dati del pacchetto IP**
 - il segmento viene passato ad IP che eventualmente lo frammenta, ma a destinazione UDP riceverà il **datagramma intero**
 - l'applicazione di destinazione riceverà quindi il **blocco completo** di dati inviato dalla applicazione che li ha trasmessi

Il segmento UDP

- Il segmento UDP è costituito da un header di lunghezza fissata (**8 byte**) più il campo dati, che deve avere dimensione massima tale da stare dentro il campo dati di IP
 - poichè il pacchetto IP può essere lungo 65535 byte, il campo dati UDP può essere lungo al massimo (**65535 – 8 – lunghezza header IP**) byte

UDP header

- L'header è costituito da quattro campi di due byte:
 - **source e destination port**: le porte di associazione alle applicazioni mittente e destinataria dei dati
 - **length**: lunghezza del segmento in byte (compreso l'header)
 - **checksum**: questo campo contiene una checksum del segmento completo (anzi: viene aggiunto uno **pseudo-header** con le informazioni degli **indirizzi IP** di sorgente e destinazione)
- l'utilizzo del campo checksum è **opzionale**, e l'applicativo può decidere di non utilizzarlo (in questo caso il campo è riempito con zeri)
- molti applicativi non lo utilizzano per motivi di **efficienza**
- se viene utilizzato, un errore provoca la **rimozione** del segmento senza che vengano prese altre iniziative





Caratteristiche di UDP

- Benchè inaffidabile, UDP ha caratteristiche che per molte applicazioni sono **appetibili**
 - può utilizzare trasmissione **multicast** o **broadcast**
 - TCP è un protocollo orientato alla connessione, quindi per definizione non può gestire una comunicazione tra più di due entità
- è molto **leggero**, quindi **efficiente**
 - la dimensione ridotta dell'header impone un **overhead minimo**, ed una **rapidità di elaborazione** elevata
 - la mancanza di **meccanismi di controllo** rende ancora più **rapida** l'elaborazione del segmento ed il recapito dei dati

Applicativi che utilizzano UDP

- Applicativi che necessitano di trasmissioni broadcast
- Applicativi per i quali la perdita di una porzione di dati non è essenziale, ma richiedono un protocollo rapido e leggero
 - stream voce/video
- Applicativi che si scambiano messaggi (e non flussi di byte) di piccole dimensioni, e che non risentono della perdita di un messaggio
 - interrogazione di database
 - sincronizzazione oraria
 - in questi casi la perdita della richiesta e della risposta provoca un nuovo tentativo di interrogazione
- Applicativi che necessitano di risparmiare l'overhead temporale provocato dalla connessione, ed implementano a livello di applicazione il controllo della correttezza dei dati
 - ad esempio applicativi che scambiano dati con molti host, rapidamente, per i quali dover stabilire ogni volta una connessione è peggio che ritentare se qualcosa va storto

Applicativi standard su UDP

- Sono molti, ed in aumento
- Gli applicativi che storicamente utilizzano UDP sono
 - DNS, sulla porta 53
 - TFTP (**Trivial File Transfer Protocol**), sulla porta 69
 - NetBIOS Name Service (anche **WINS**) sulla porta 137
 - SNMP (**Simple Network Management Protocol**) sulla porta 161
 - NTP (**Network Time Protocol**) sulla porta 123
 - NFS (**Network File System**) via portmap