

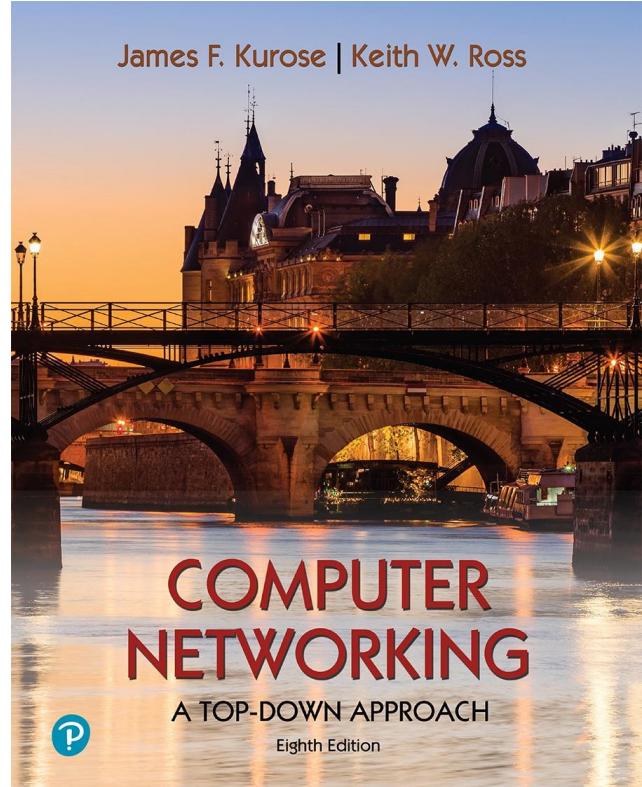


DeepL

Abbonati a DeepL Pro per tradurre file di maggiori dimensioni.
Per ulteriori informazioni, visita www.DeepL.com/pro.

Capitolo 3

Livello di trasporto



Reti di computer: Un approccio dall'alto verso il basso
8th edizione
Jim Kurose, Keith Ross

Pearson, 2020

Livello di trasporto: panoramica

Il nostro obiettivo:

- comprendere i principi alla base dei servizi del livello di trasporto:
 - multiplexing, demultiplexing
 - trasferimento dati affidabile
 - controllo del flusso
 - controllo della congestione

- conoscere i protocolli del livello di trasporto di Internet:
 - UDP: trasporto senza connessione
 - TCP: trasporto affidabile orientato alla connessione
 - Controllo della congestione TCP

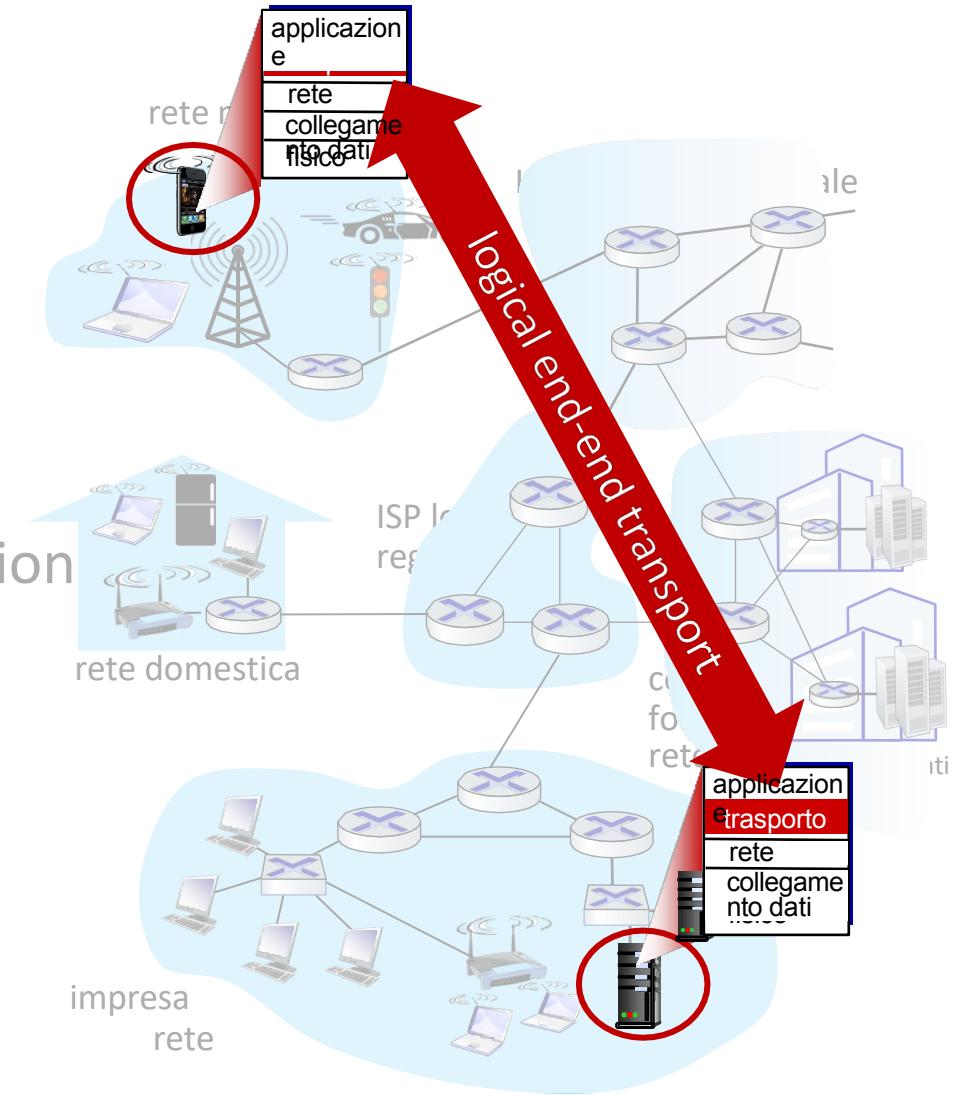
Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- Controllo della congestione TCP
- Evoluzione delle funzionalità del livello di trasporto



Servizi e protocolli di trasporto

- fornire una *comunicazione logica* tra i processi applicativi in esecuzione su host diversi
- azioni dei protocolli di trasporto nei sistemi finali:
 - mittente: interrompe i messaggi dell'applicazione in *segmenti*, passa al livello di rete
 - ricevitore: riassembra i segmenti in messaggi, passa al livello applicativo
- due protocolli di trasporto disponibili per le applicazioni Internet

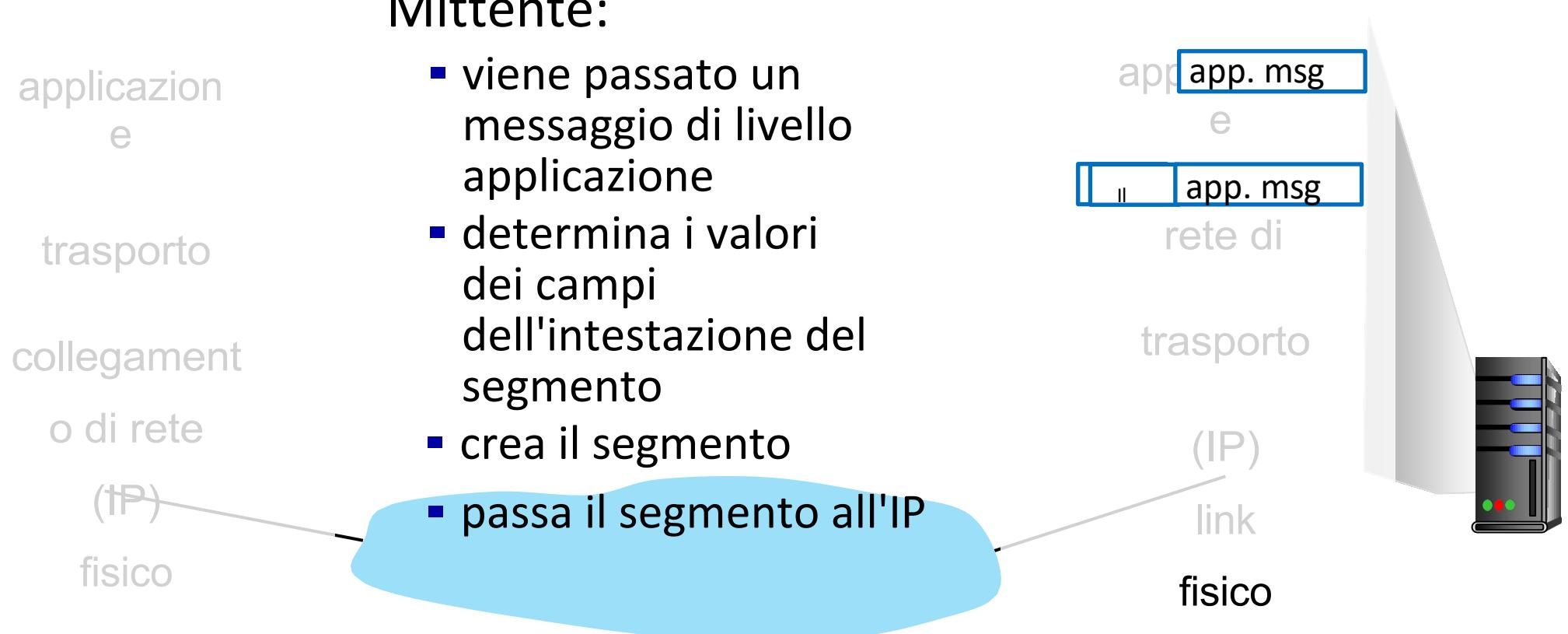


- TCP, UDP

Servizi e protocolli del livello di trasporto e del livello di rete

- **livello di trasporto:** comunicazione tra *processi*
 - si basa su, migliora i servizi di livello di rete
- **Livello di rete:** comunicazione tra *host*

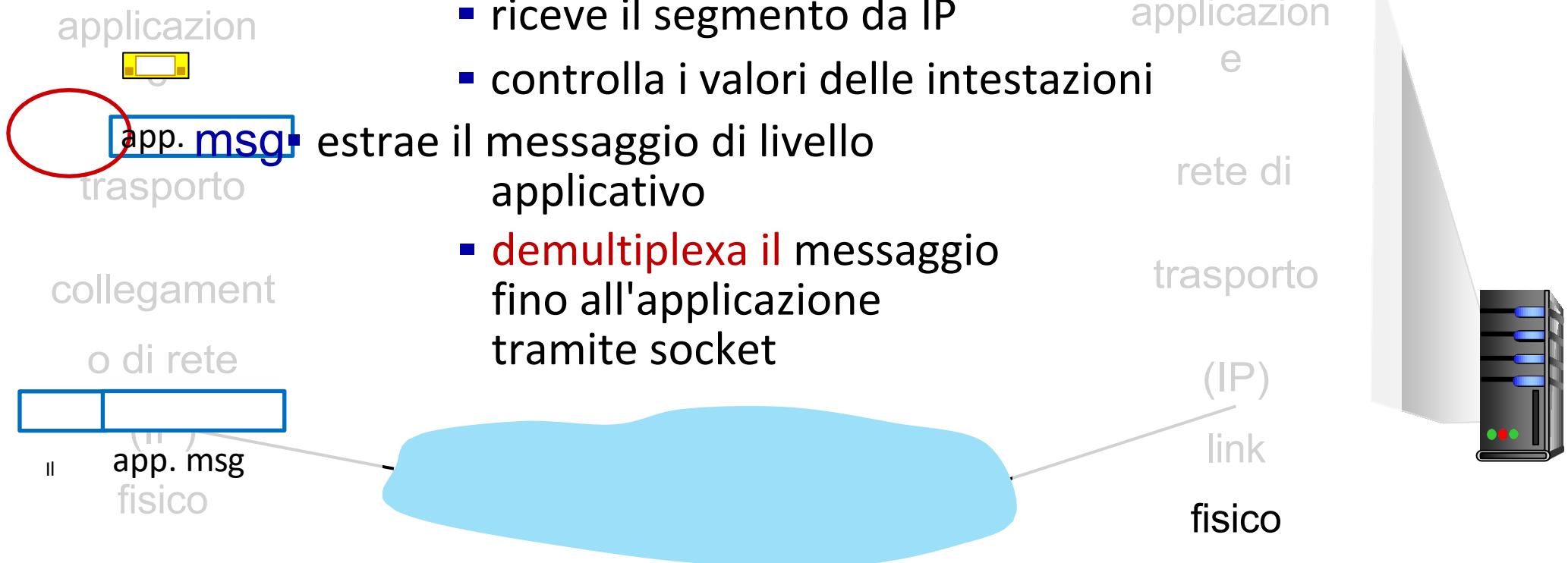
Azioni del livello di trasporto



Azioni del livello di trasporto

Ricevitore:

- riceve il segmento da IP
- controlla i valori delle intestazioni
- **estrae il messaggio di livello applicativo**
- **demultiplexa il messaggio fino all'applicazione tramite socket**



Due principali protocolli di trasporto Internet

- **TCP:** Protocollo di controllo della trasmissione

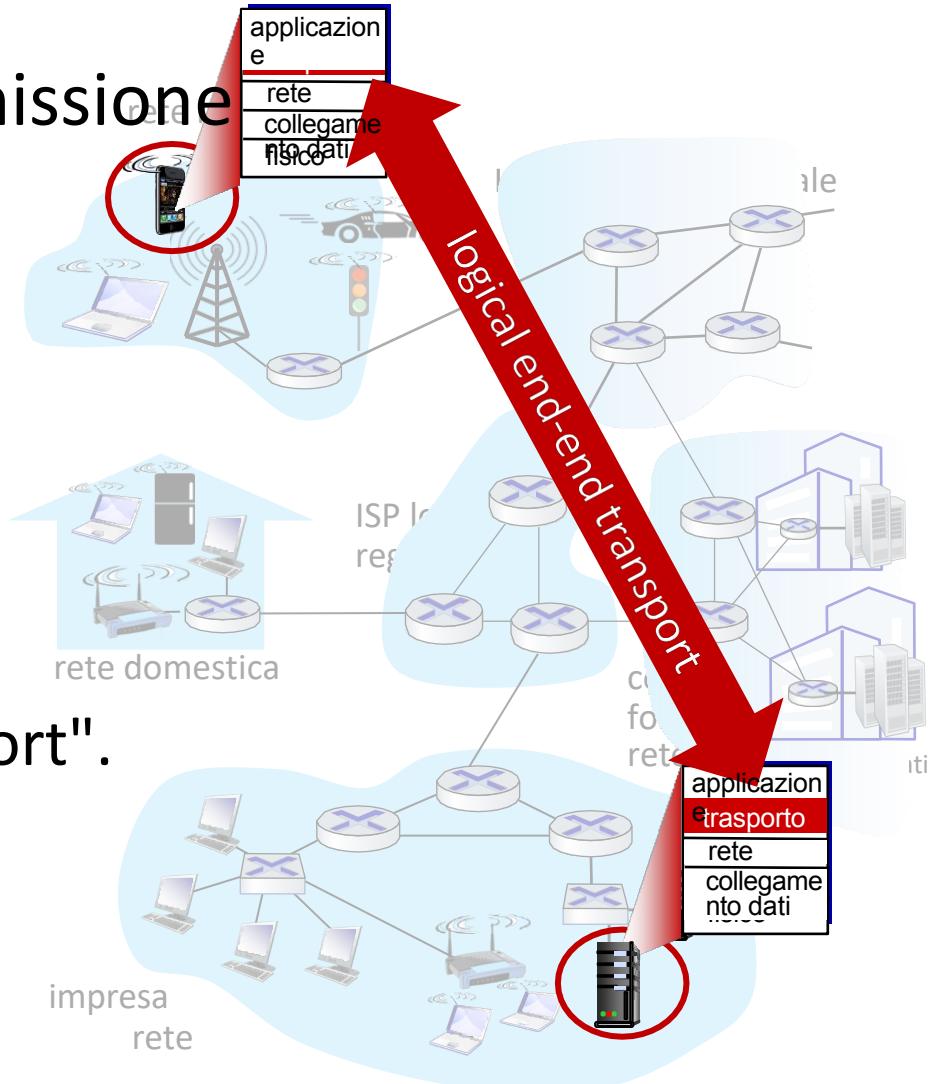
- consegna affidabile e puntuale
- controllo della congestione
- controllo del flusso
- impostazione della connessione

- **UDP:** Protocollo User Datagram

- consegna inaffidabile e non ordinata
- estensione senza fronzoli dell'IP "best-effort".

- servizi *non disponibili*:

- garanzie di ritardo
- garanzie di larghezza di banda



Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- **Multiplexing e demultiplexing**
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- Controllo della congestione TCP
- Evoluzione delle funzionalità del livello di trasporto

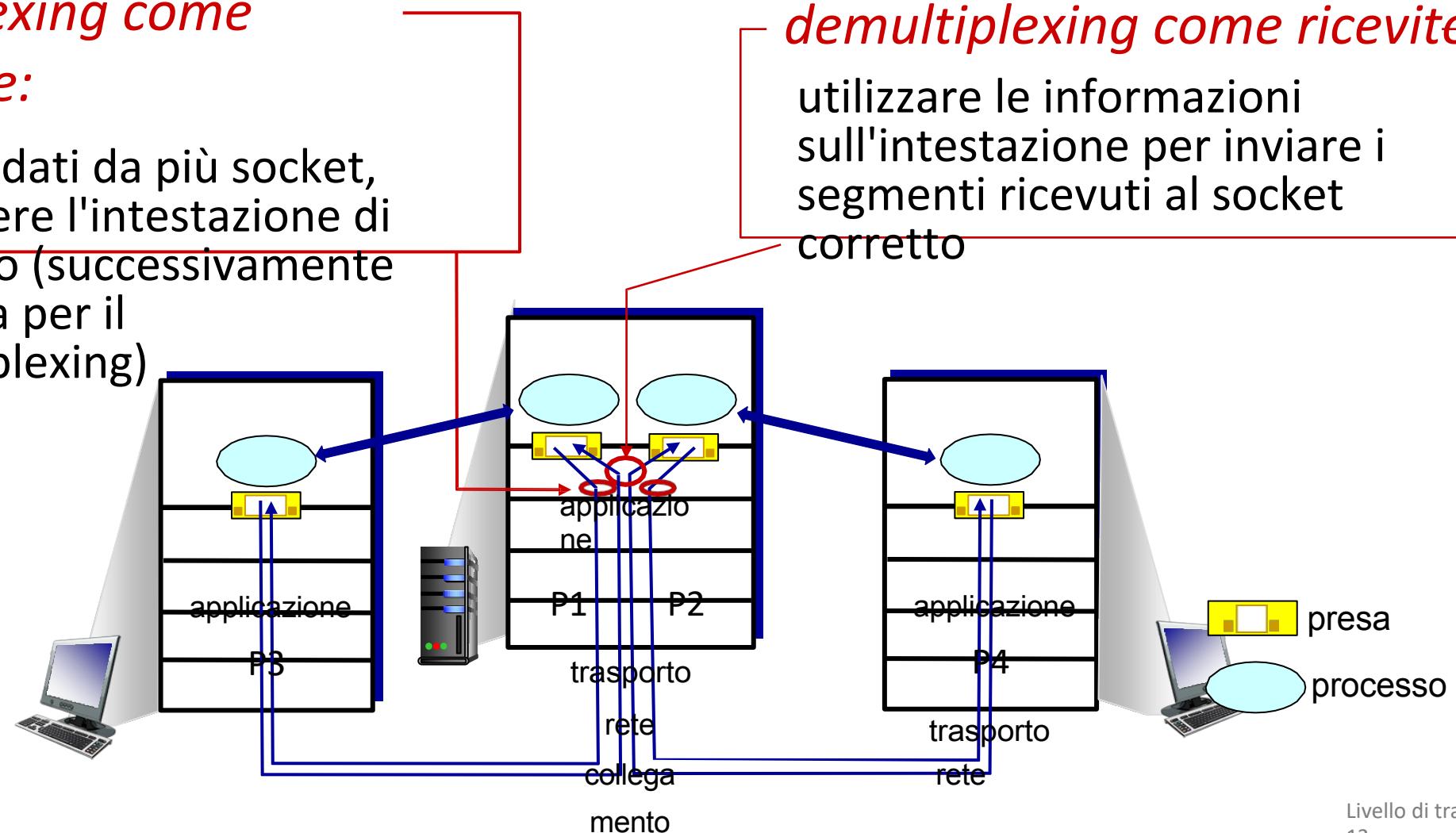


Multiplexing/demultiplexing

multiplexing come mittente:

gestire i dati da più socket,
aggiungere l'intestazione di
trasporto (successivamente
utilizzata per il
demultiplexing)

demultiplexing come ricevitore:
utilizzare le informazioni
sull'intestazione per inviare i
segmenti ricevuti al socket
corretto



trasporto

rete

collega

mento

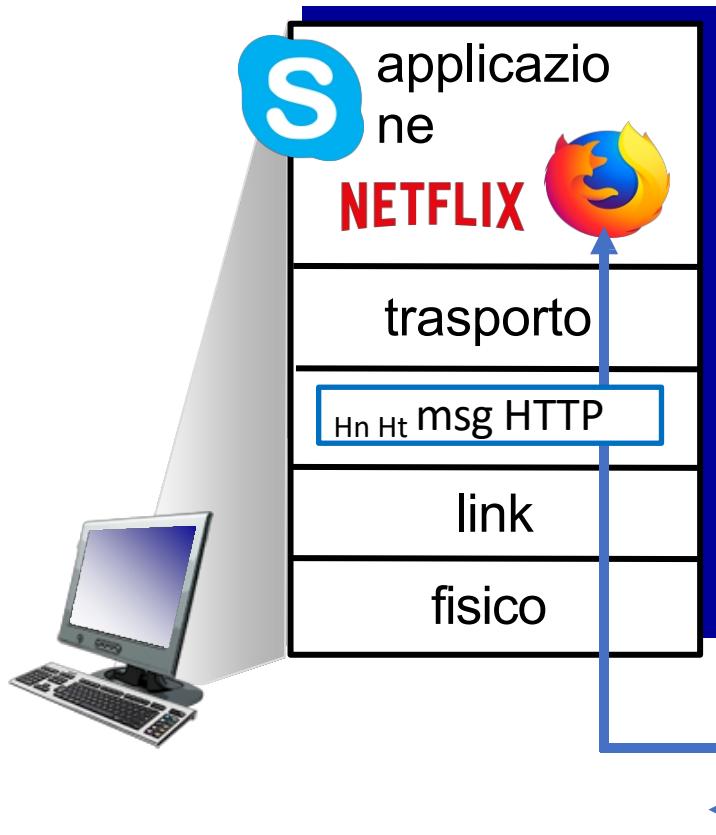
fisico

link

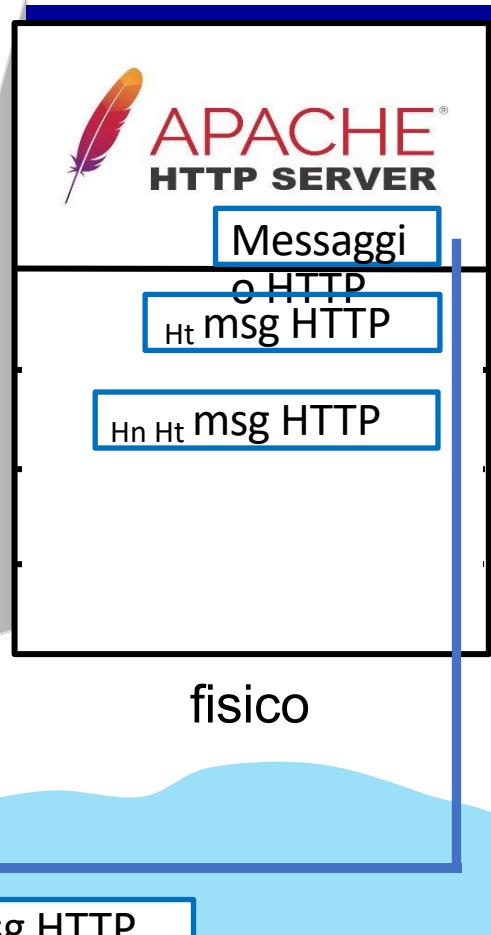
fisico

Server HTTP

cliente



server

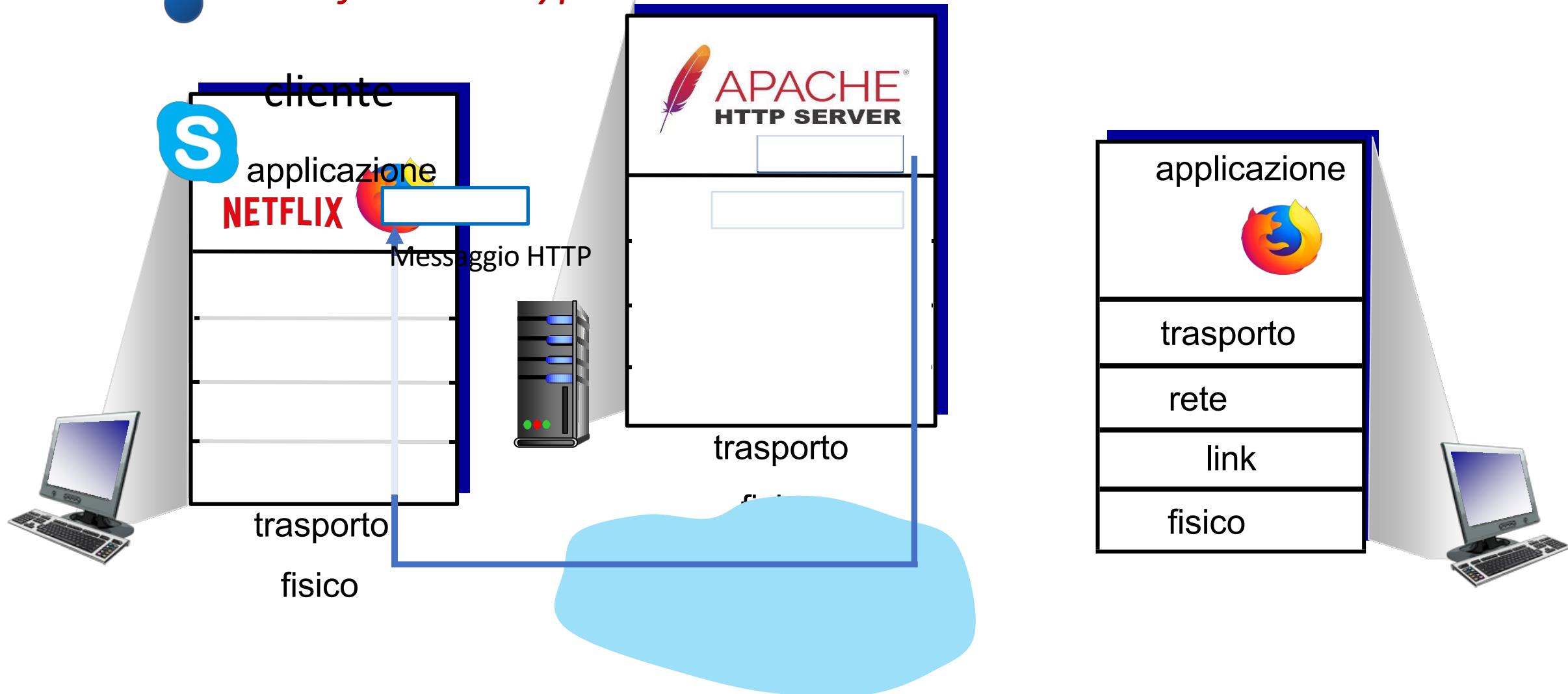


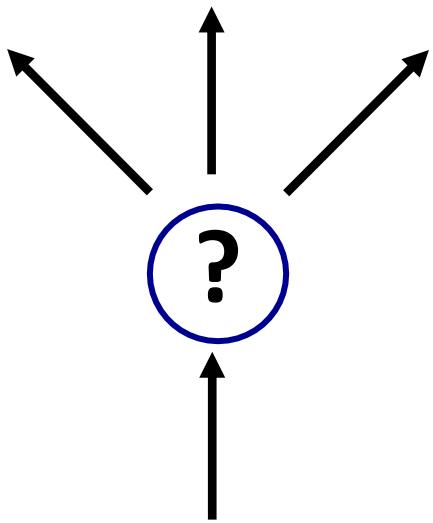
fisico



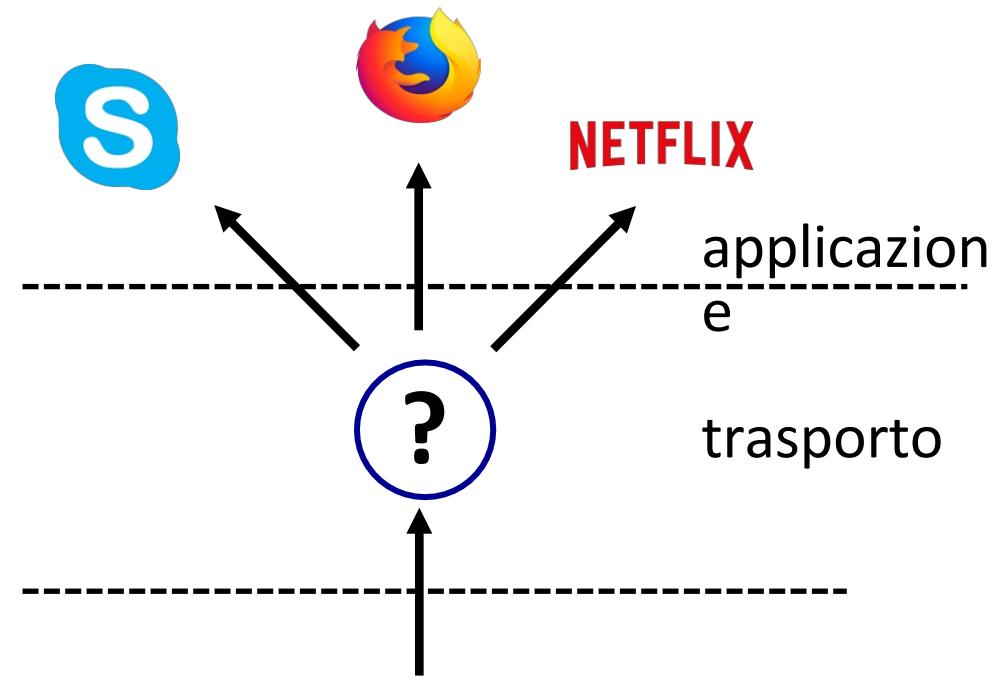


D: come ha fatto il livello di trasporto a sapere di dover consegnare il messaggio al processo del browser Firefox piuttosto che a quello di Netflix o di Skype?





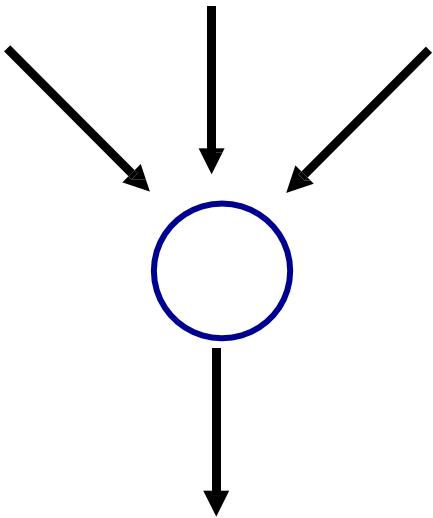
de-multiplexing



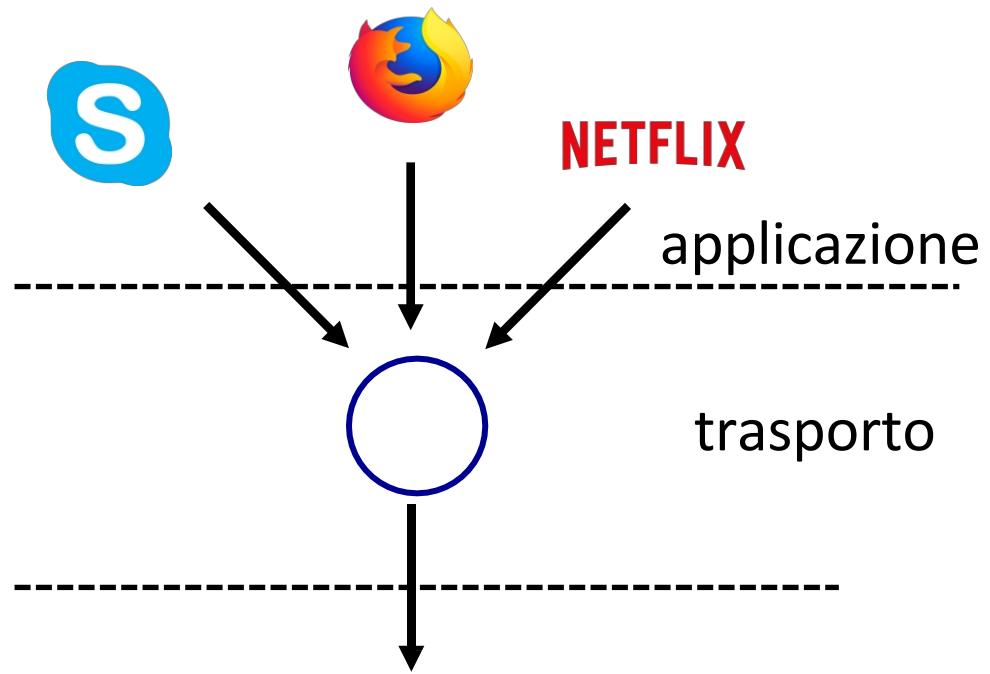
de-multiplexing



Demultiplexing



multiplexing



multiplexing



Multiplexing

Come funziona il demultiplexing

- l'host riceve i datagrammi IP
 - Ogni datagramma ha un indirizzo IP di origine, un indirizzo IP di destinazione e un indirizzo IP di destinazione.
 - ogni datagramma trasporta un segmento del livello di trasporto
 - ogni segmento ha un numero di porta di origine e di destinazione
- l'host utilizza gli *indirizzi IP e i numeri di porta* per indirizzare il segmento al socket appropriato



Formato del segmento TCP/UDP

Demultiplexing senza connessione

Richiamo:

- quando si crea un socket,
si deve specificare la porta
dell'host locale:

```
DatagramSocket mySocket1  
        = nuovo  
        DatagramSocket(12534);
```

- quando si crea il
datagramma da inviare al
socket UDP, si deve
specificare
 - indirizzo IP di destinazione

- porta di destinazione #

quando l'host ricevente riceve

allo *stesso socket* dell'host ricevente.

Segmento *UDP*:

- controlla la porta di destinazione # nel segmento
- indirizza il segmento UDP al socket con quella porta #

I datagrammi IP/UDP con la *stessa porta # di destinazione*, ma con indirizzi IP di origine e/o numeri di porta di origine diversi, vengono indirizzati

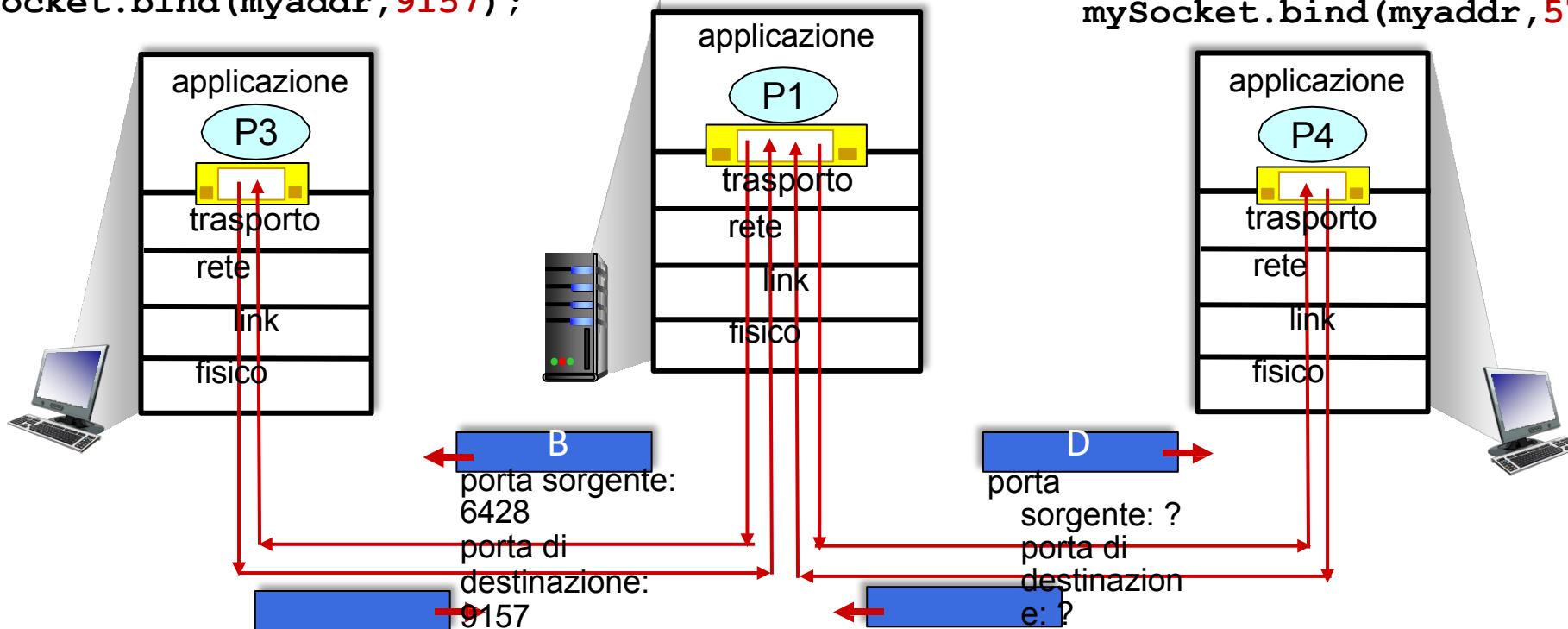


Demultiplexing senza connessione: un esempio

```
mySocket =  
    socket(AF_INET, SOCK_DGRAM)  
mySocket.bind(myaddr, 6428);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 9157);
```

```
mySocket =  
    socket(AF_INET, SOCK_STREAM)  
mySocket.bind(myaddr, 5775);
```



porta sorgente: 9157
Porta di destinazione: 6428

tinazione: ?

p
o
r
t
a

s
o
r
g
e
n
t
e
:

?

p
o
r
t
a

d
i

d
e
s

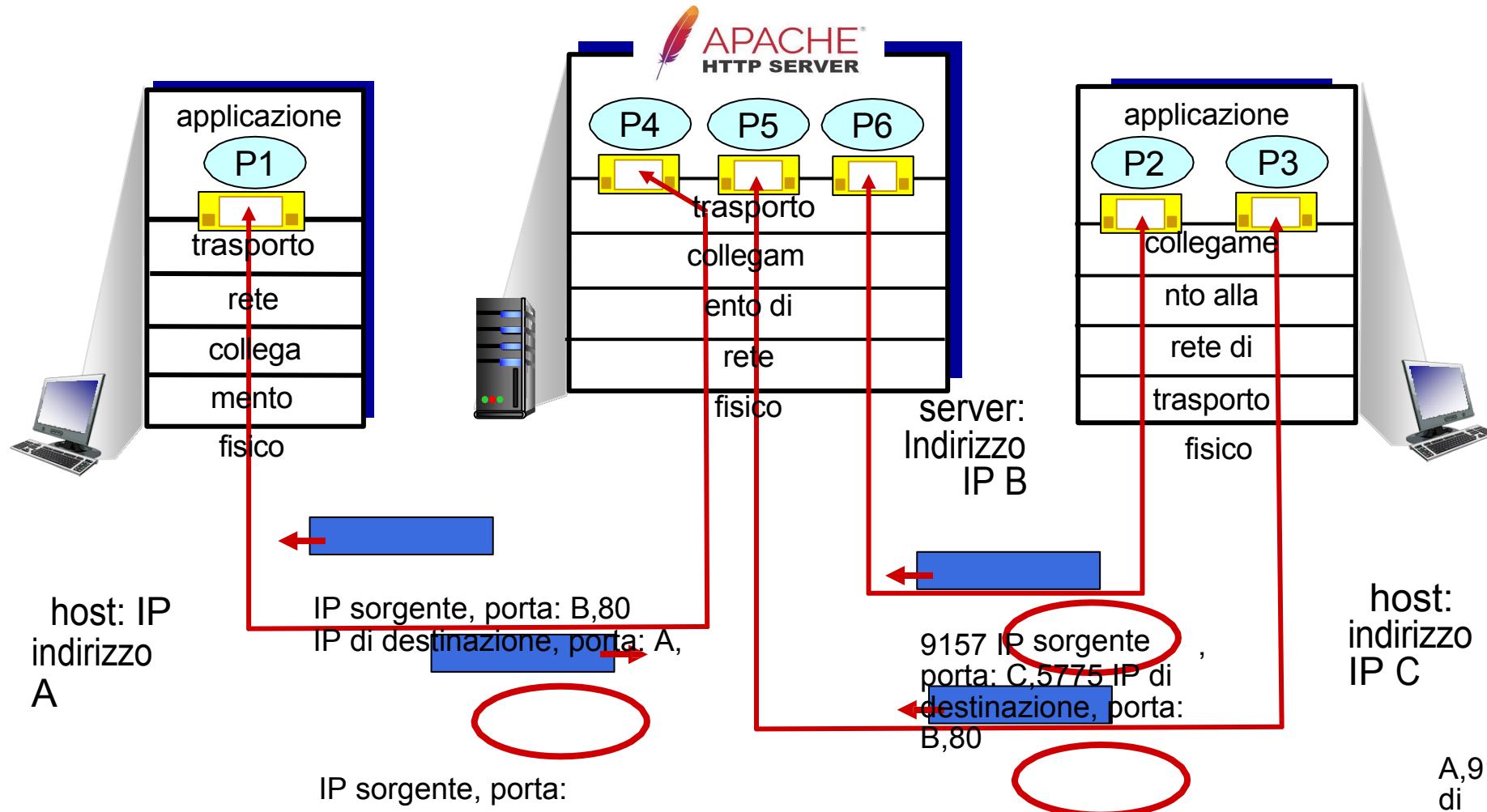
Demultiplexing orientato alla connessione

- socket TCP identificato da una **4-tupla**:
 - indirizzo IP di origine
 - numero di porta di origine
 - indirizzo IP di destinazione
 - numero della porta di destinazione
- demux: il ricevitore utilizza **tutti e quattro i**

valori (4-tuple) per indirizzare il segmento al socket appropriato

- Il server può supportare molti socket TCP simultanei:
 - ogni socket identificato dalla propria 4-tupla
 - ogni socket associato a un diverso client di connessione

Demultiplexing orientato alla connessione: esempio



destinazione,
porta: B,80

IP sorgente, porta:
C,9157 IP di
destinazione, porta:
B,80

Tre segmenti, tutti destinati all'indirizzo IP:
B,

porta dest: 80 vengono demultiplexati su socket *diversi*

Sintesi

- Multiplexing, demultiplexing: in base ai valori dei segmenti e dei campi dell'intestazione del datagramma
- **UDP:** demultiplexing con numero di porta di destinazione (solo)
- **TCP:** demultiplexing con 4 tuple: indirizzi IP di origine e destinazione e numeri di porta
- La multiplazione/demultiplazione avviene a *tutti i* livelli

Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- **Trasporto senza connessioni: UDP**
- Principi di trasferimento affidabile dei dati
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- Controllo della congestione TCP
- Evoluzione delle funzionalità del livello di trasporto



UDP: Protocollo User Datagram

- "senza fronzoli", "con pochi fronzoli".
Protocollo di trasporto Internet
- servizio "best effort", UDP
I segmenti possono essere:
 - perso
 - consegnato fuori ordine
all'app
- *senza connessione*:
 - nessun handshake tra il mittente e il destinatario UDP
 - ogni segmento UDP gestito indipendentemente dagli altri

Perché esiste un UDP?

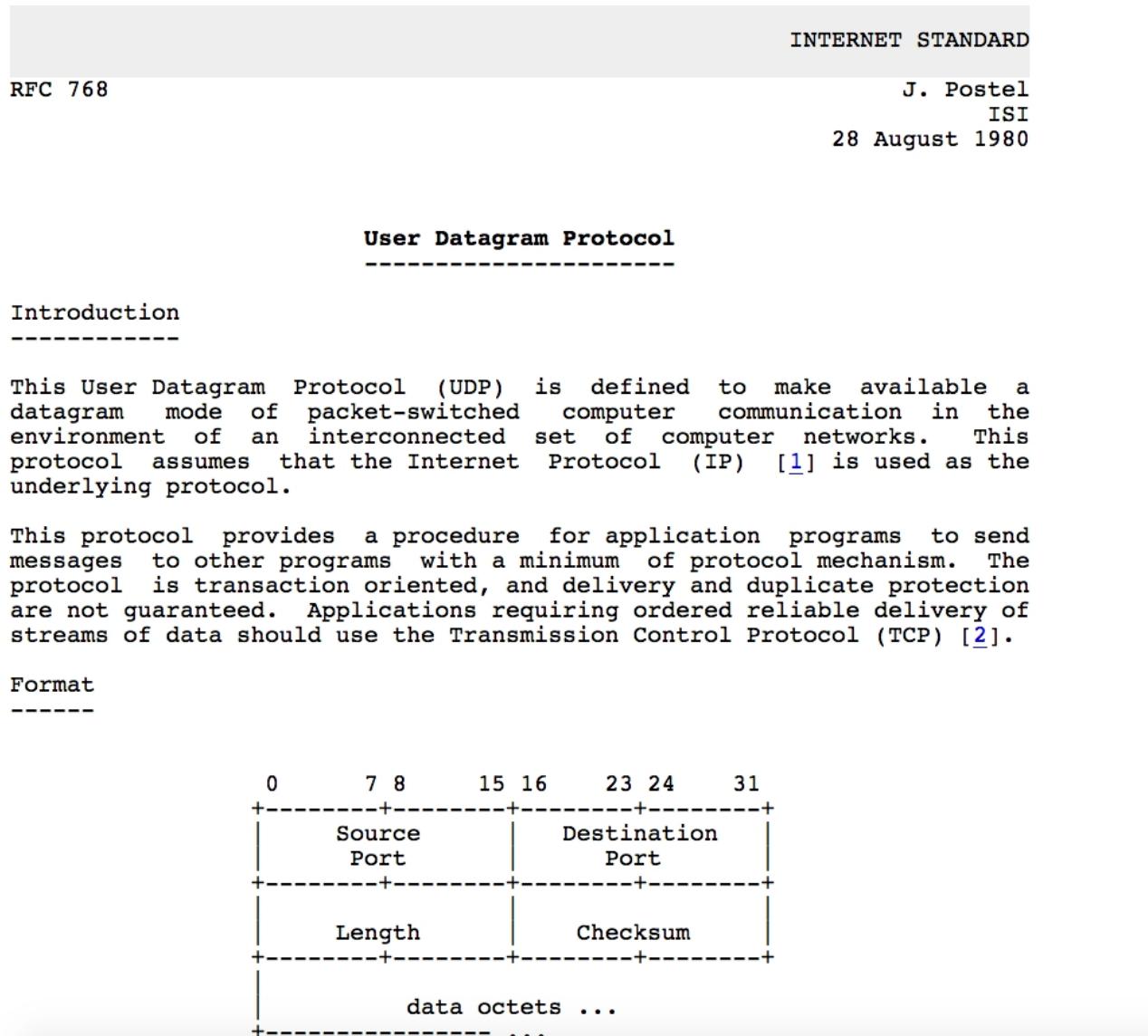
- nessuna creazione di connessione (che può aggiungere un ritardo RTT)
- semplice: nessuno stato di connessione al mittente e al destinatario
- dimensioni ridotte dell'intestazione
- nessun controllo della congestione
 - L'UDP può essere lanciato alla velocità desiderata!

- può funzionare in presenza di una congestione

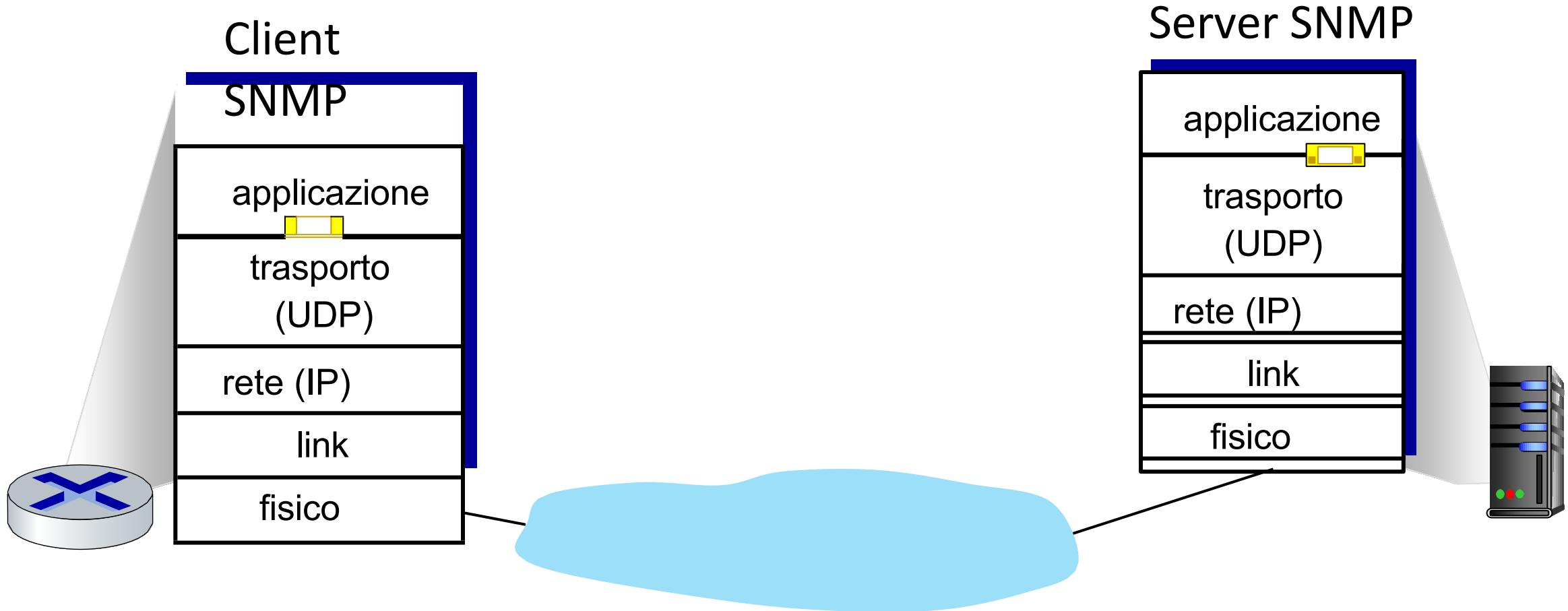
UDP: Protocollo User Datagram

- Utilizzo di UDP:
 - applicazioni multimediali in streaming (tolleranti alle perdite, sensibili alla velocità)
 - DNS
 - SNMP
 - HTTP/3
- se è necessario un trasferimento affidabile su UDP (ad esempio, HTTP/3):
 - aggiungere l'affidabilità necessaria a livello di applicazione
 - aggiungere il controllo della congestione a livello di applicazione

UDP: Protocollo User Datagram [RFC 768]



UDP: Azioni del livello di trasporto



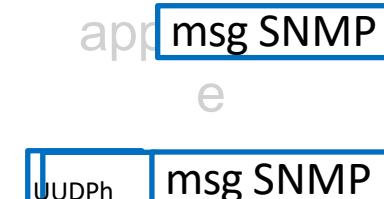
UDP: Azioni del livello di trasporto

Client
SNMP
applicazion
e
trasporto
(UDP)
collegament
o di rete
(IP)
fisico

Azioni del mittente UDP:

- viene passato un messaggio di livello applicazione
- determina i valori dei campi dell'intestazione del segmento UDP
- crea un segmento UDP
- passa il segmento all'IP

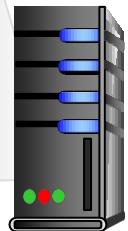
Server SNMP



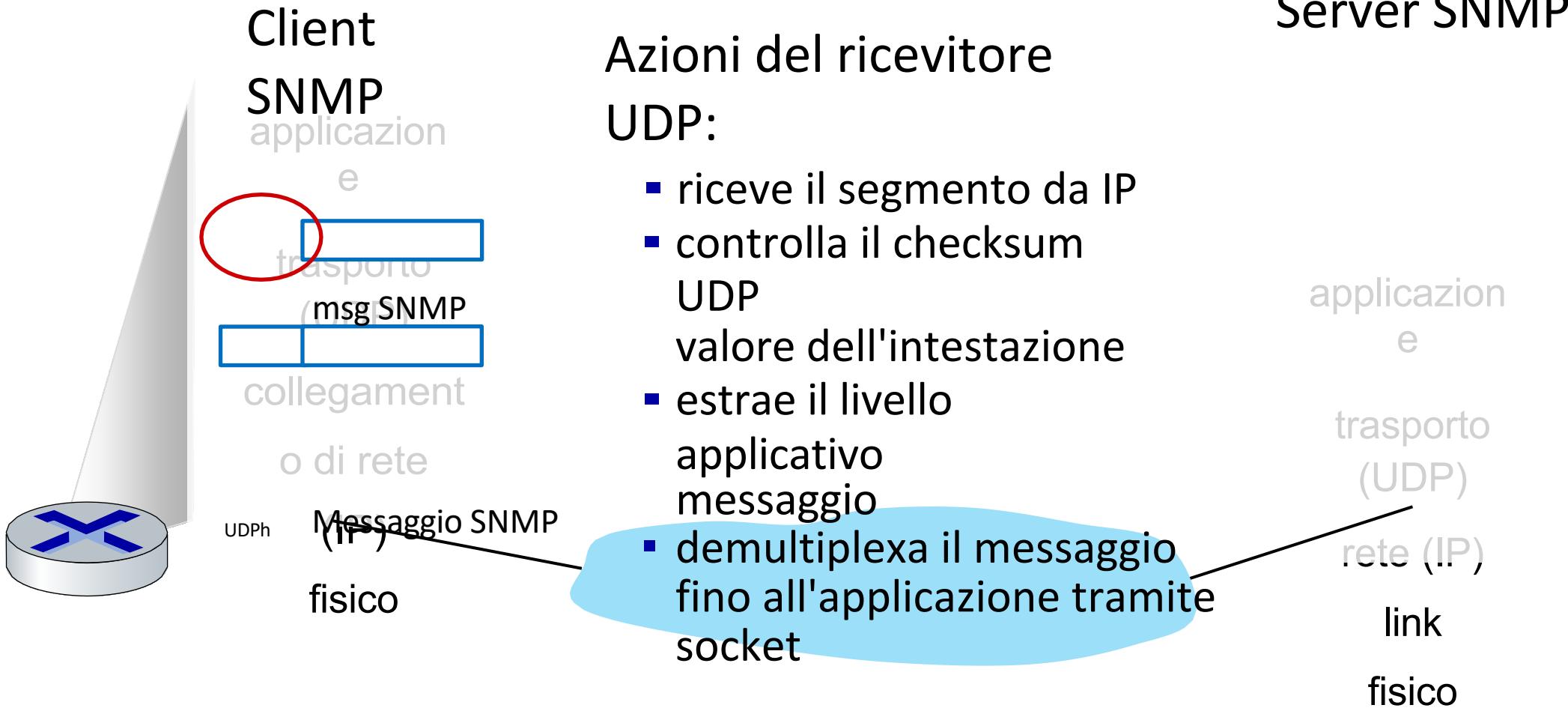
rete (IP)

link

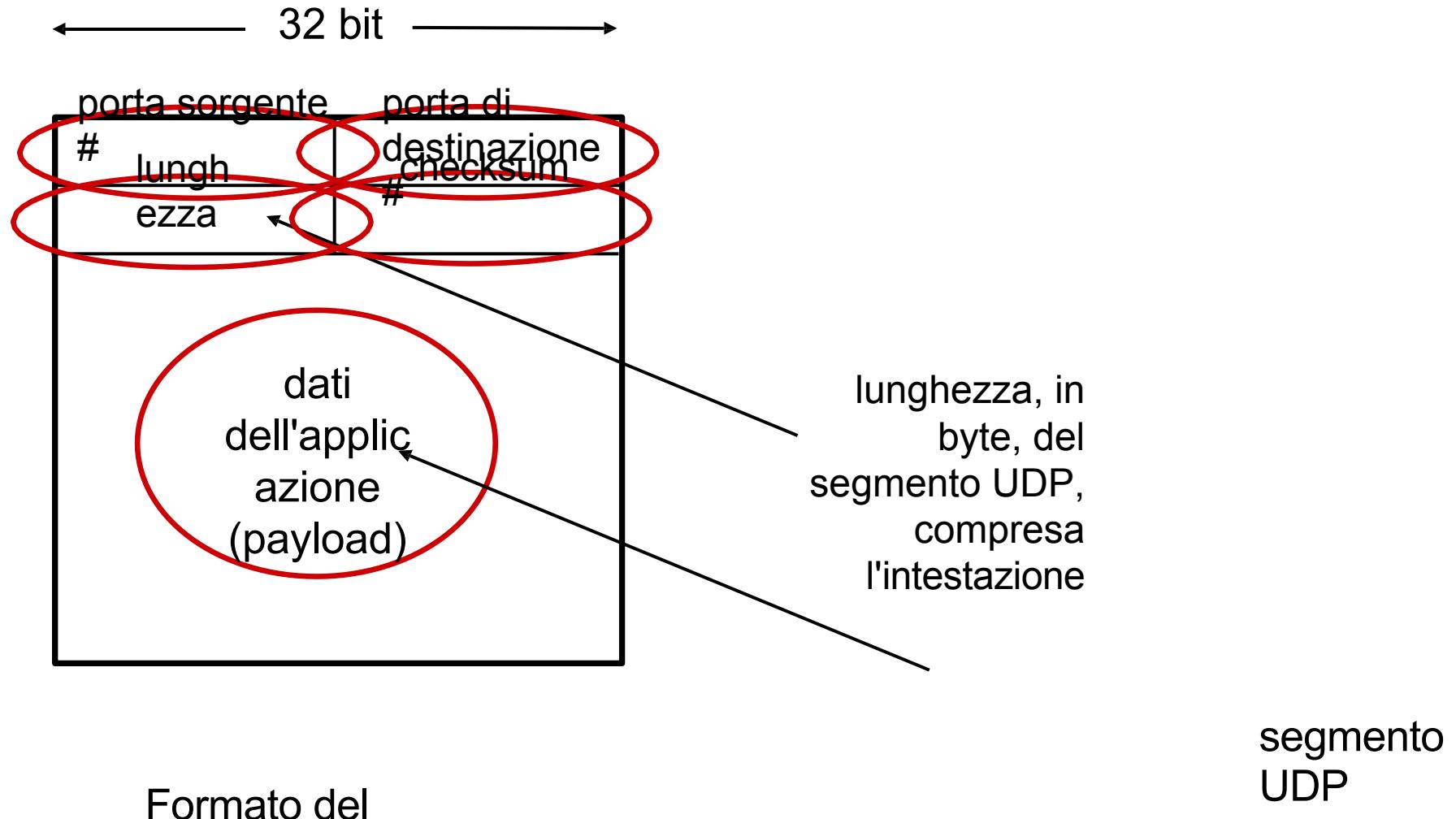
fisico



UDP: Azioni del livello di trasporto



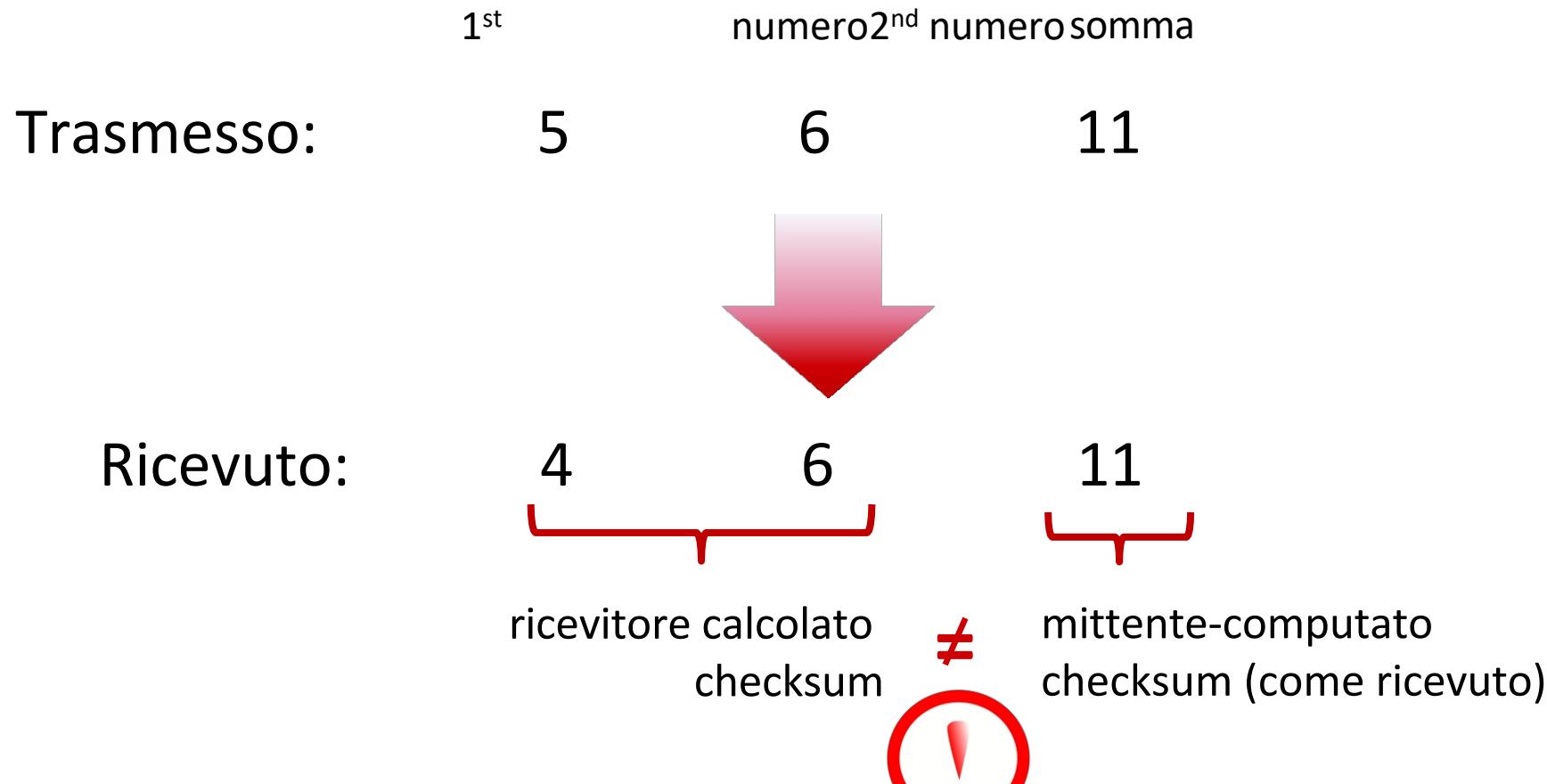
Intestazione del segmento UDP



dati da/per
livello applicativo

checksum UDP

Obiettivo: rilevare gli errori (cioè i bit capovolti) nel segmento trasmesso.



checksum di Internet

Obiettivo: rilevare gli errori (*cioè i bit capovolti*) nel segmento trasmesso.

mittente:

- trattare il contenuto del segmento UDP (compresi i campi dell'intestazione UDP e gli indirizzi IP) come una sequenza di numeri interi a 16 bit
- **checksum:** somma (complemento a uno) del contenuto del segmento

- valore di checksum inserito nel campo checksum UDP

ricevitore:

- calcolare il checksum
del segmento
ricevuto
- controlla se il checksum
calcolato corrisponde al
valore del campo
checksum:
 - non uguale - errore rilevato
 - uguale - nessun errore
rilevato. *Ma forse ci sono
comunque degli errori?*
Più tardi

Il checksum di Internet: un esempio

esempio: aggiungere due interi a 16 bit

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
<hr/>																
avvolgente	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
somma	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
checksum	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

Nota: quando si sommano i numeri, al risultato deve essere aggiunto il riporto del bit più significativo.

* Per ulteriori esempi, consultare gli esercizi interattivi online: http://gaia.cs.umass.edu/kurose_ross/interactive/

Il checksum di Internet: una protezione debole!

esempio: aggiungere due interi a 16 bit

	1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1	0 1
	1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1	1 0
avvolgente	1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1	
som	1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 0 0	
ma di controllo	0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 1 1	

Due numeri da sommare:

1 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1 0 1
1 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 1 0 1

Risultato della somma:

1 0 1 1 1 0 1 1 1 0 1 1 1 1 1 0 0 0

Che è diverso dal risultato atteso:

1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 0 1 1

Questo dimostra che il checksum non è un buon mezzo per la protezione dei dati.

Anche se i numeri sono cambiati (bit flips), **nessun** cambiamento nel checksum!

Sommario: UDP

- protocollo "senza fronzoli":
 - i segmenti possono essere persi, consegnati fuori ordine
 - servizio best effort: "invia e spera nel meglio"
- UDP ha i suoi vantaggi:
 - non è necessario alcun setup/handshaking (nessun RTT)
 - può funzionare quando il servizio di rete è compromesso
 - contribuisce all'affidabilità (checksum)
- costruire funzionalità aggiuntive su UDP nel livello applicazione (ad esempio, HTTP/3)

Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- **Principi di trasferimento affidabile dei dati**
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- Controllo della congestione TCP
- Evoluzione delle funzionalità del livello di trasporto

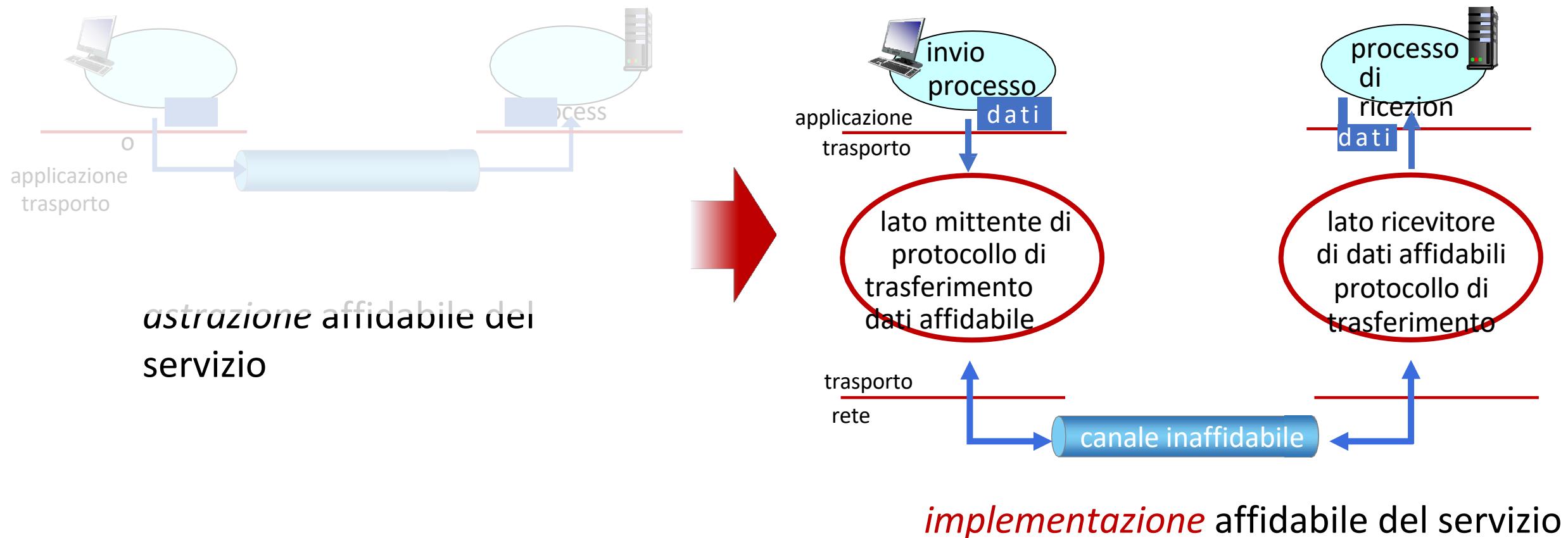


Principi di trasferimento affidabile dei dati



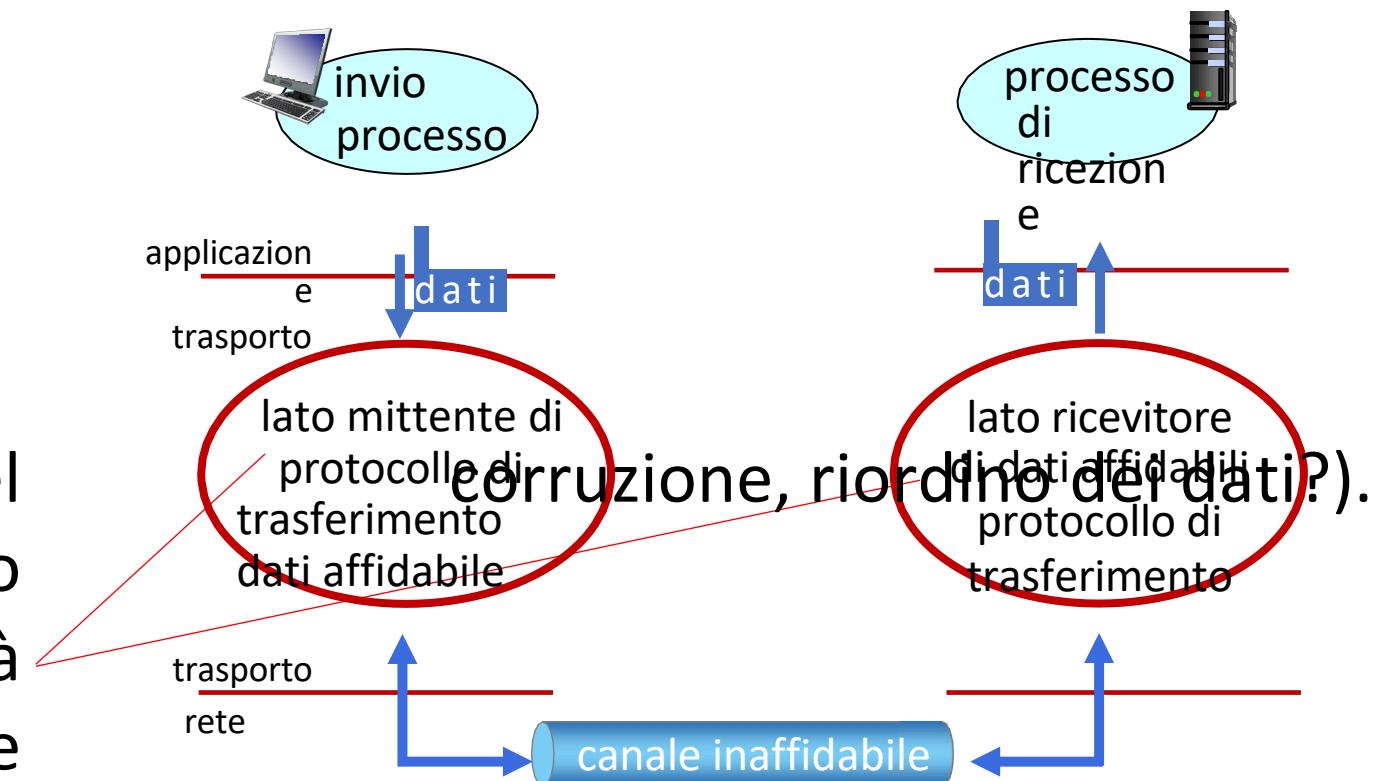
astrazione affidabile del servizio

Principi di trasferimento affidabile dei dati



Principi di trasferimento affidabile dei dati

La complessità del protocollo di trasferimento dati affidabile dipenderà (fortemente) dalle caratteristiche del canale inaffidabile (perdita,

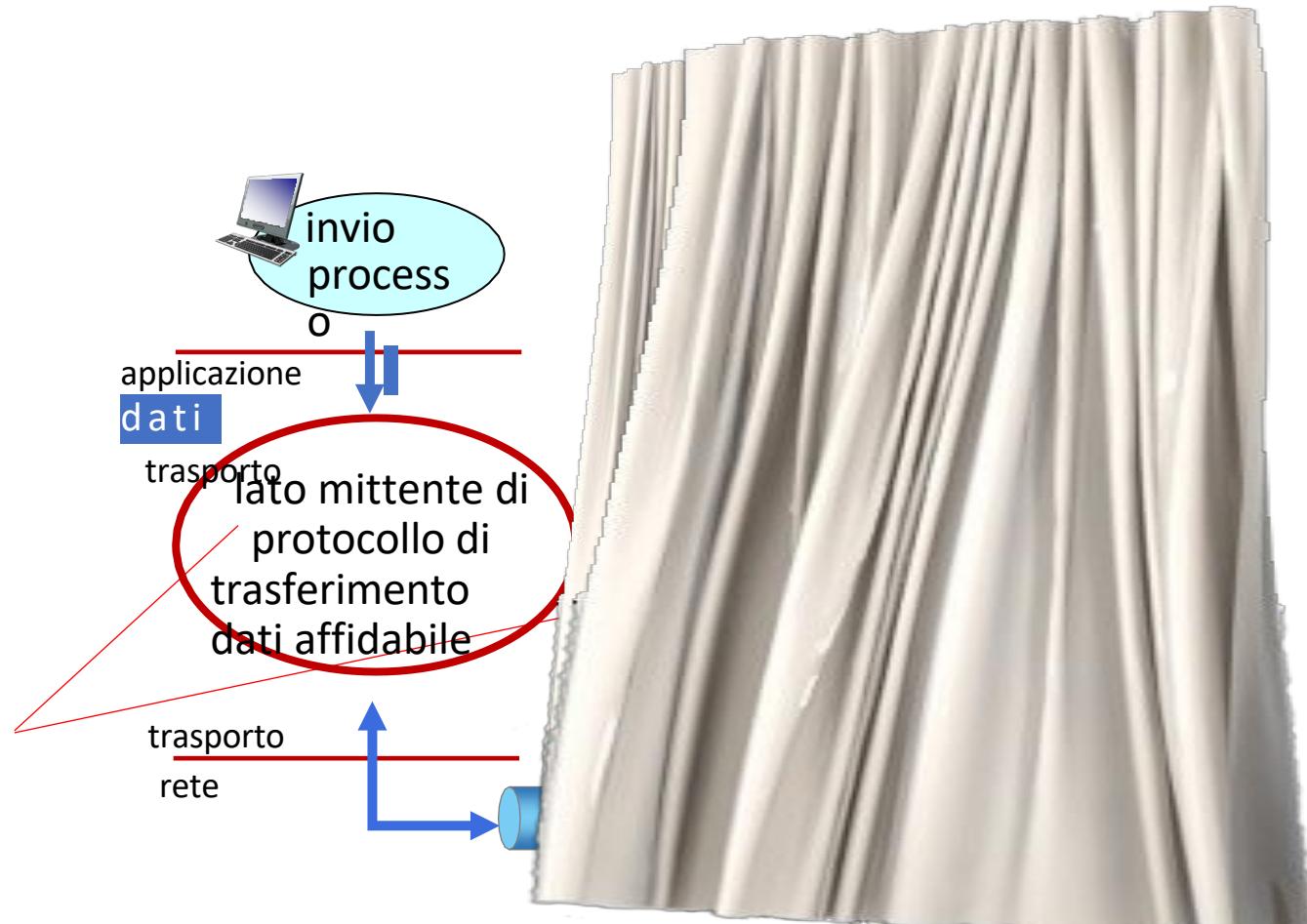


implementazione affidabile del servizio

Principi di trasferimento affidabile dei dati

Il mittente e il destinatario *non* conoscono lo "stato" dell'altro, ad esempio: il messaggio è stato ricevuto?

- a meno che non venga comunicato tramite un

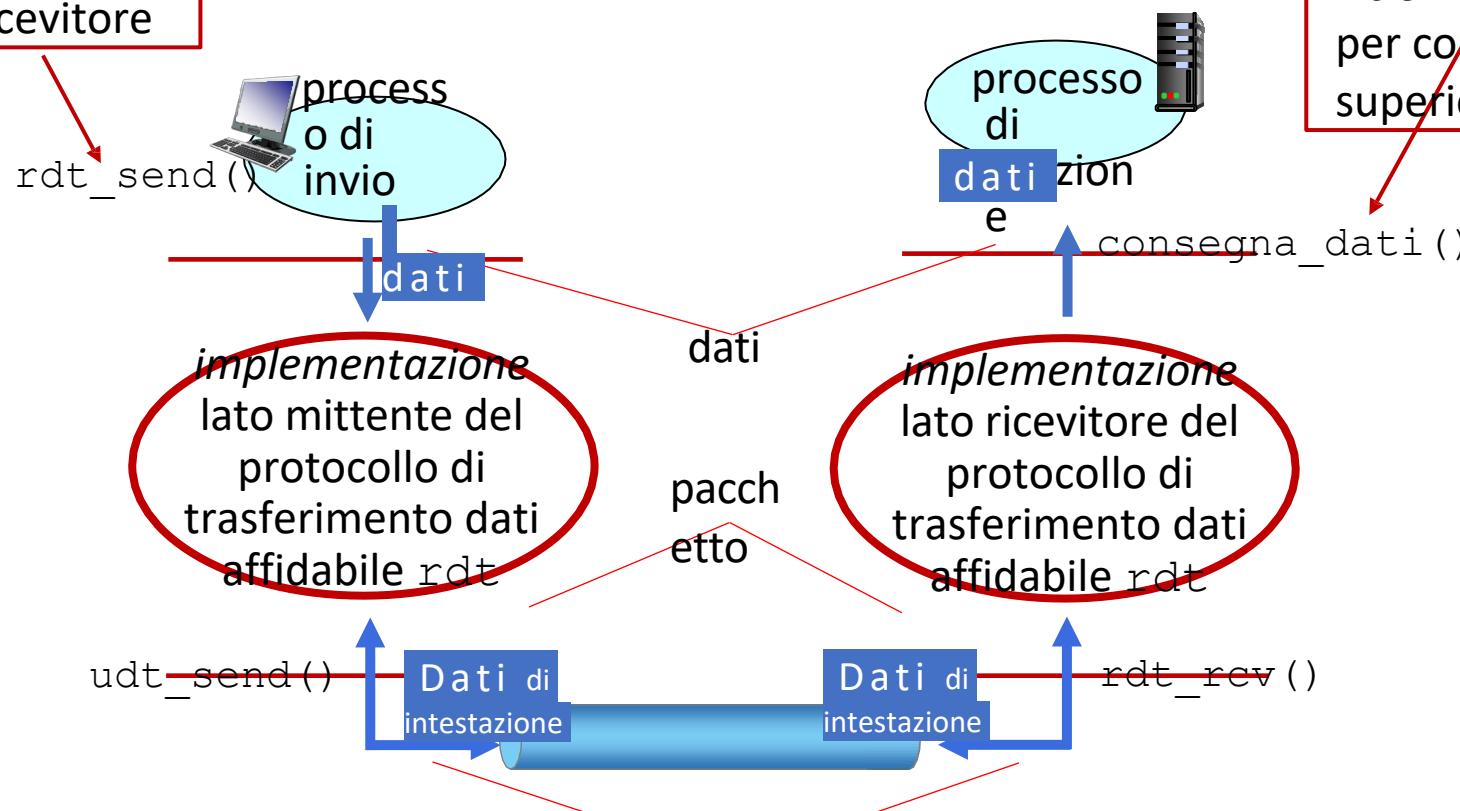


implementazione affidabile del servizio

messaggio

Protocollo di trasferimento dati affidabile (rdt): interfacce

rdt_send(): chiamato dall'alto (ad es. dall'applicazione). Dati passati da consegnare al livello superiore del ricevitore



udt_send(): chiamato da rdt
per trasferire il pacchetto su un
canale inaffidabile al ricevitore

Comunicazione bidirezionale su canale
inaffidabile

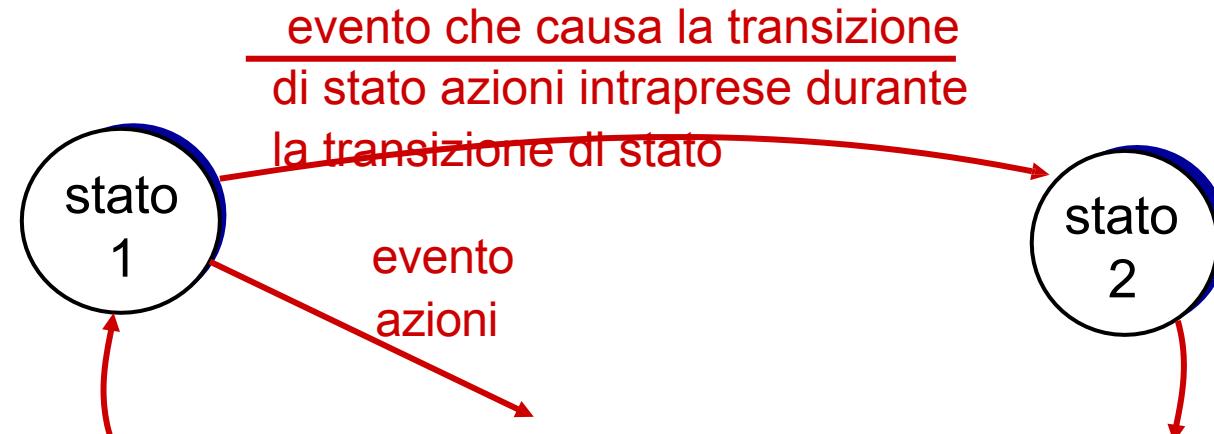
rdt_rcv(): chiamato quando
il pacchetto arriva sul lato
ricevente del canale

Trasferimento dati affidabile: come iniziare

Lo faremo:

- sviluppare in modo incrementale i lati mittente e ricevitore del protocollo di trasferimento dati affidabile (rdt)
- considerare solo il trasferimento unidirezionale dei dati
 - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- utilizzare macchine a stati finiti (FSM) per specificare mittente, destinatario

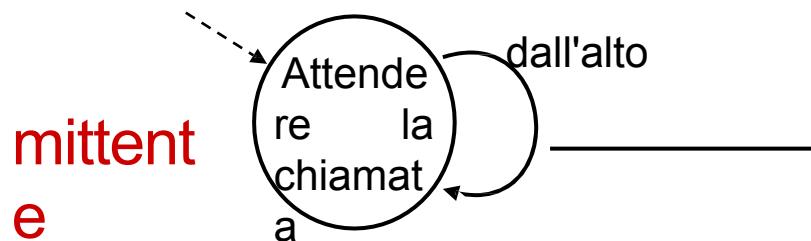
stato: quando si trova in questo "stato" lo stato successivo è determinato in modo univoco da



next
evento

rdt1.0: trasferimento affidabile su un canale affidabile

- canale sottostante perfettamente affidabile
 - nessun errore di bit
 - nessuna perdita di pacchetti
- FSM *separati* per mittente e destinatario:
 - il mittente invia i dati nel canale sottostante
 - il ricevitore legge i dati dal canale sottostante



rdt_send(da
ti)
pacchetto =
make_pkt(d

ati)
udt_se
nd(pac
chetto)

ricevitore

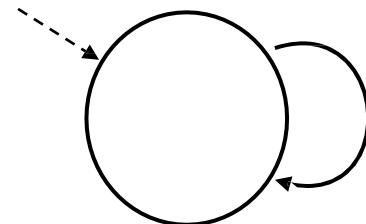
Attendere
la chiamata
dal basso

```
rdt_rc , dati)  
v(pac  
chetto  
)  
_____
```

```
e  
s  
t  
r  
a  
r  
r  
e
```

```
(  
p  
a  
c  
c  
h  
e  
t  
t  
o
```

```
consegnare_dati(  
dati)
```



rdt2.0: canale con errori di bit

- il canale sottostante può invertire i bit nel pacchetto
 - checksum (ad esempio, Internet checksum) per rilevare gli errori di bit
- *la domanda: come recuperare gli errori?*

Come fanno gli esseri umani a riprendersi dagli "errori" durante la conversazione?

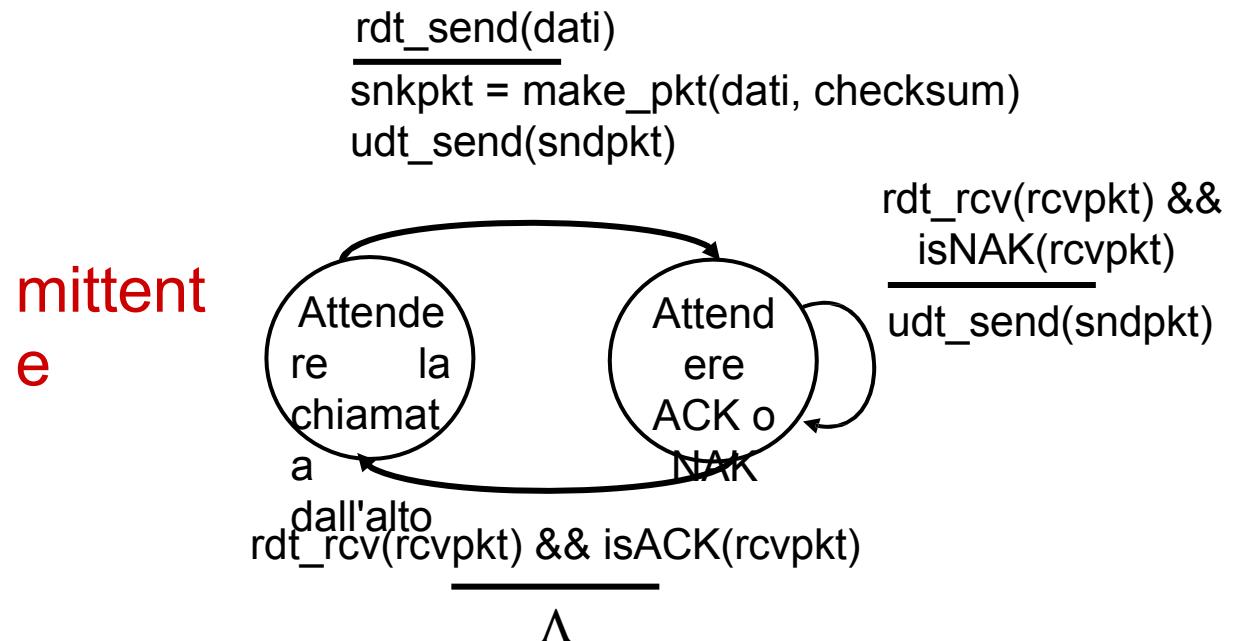
rdt2.0: canale con errori di bit

- il canale sottostante può invertire i bit nel pacchetto
 - checksum per rilevare gli errori di bit
- la domanda: come recuperare dagli errori?
 - *Riconoscimenti (ACK)*: il ricevitore comunica esplicitamente al mittente che il pkt è stato ricevuto OK.
 - *negative acknowledgements (NAK)*: il ricevitore comunica esplicitamente al mittente che il pkt ha avuto degli errori
 - il mittente *ritrasmette il* pkt al ricevimento di NAK

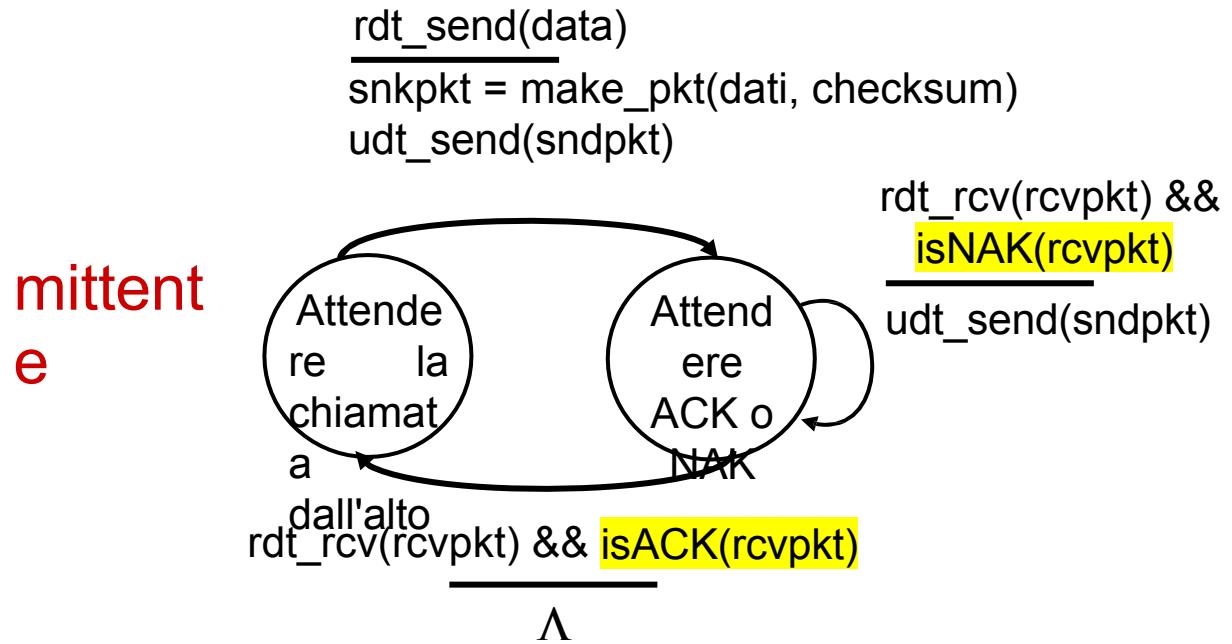
fermarsi e aspettare

il mittente invia un pacchetto, quindi attende la risposta del destinatario

rdt2.0: Specifiche FSM



rdt2.0: Specifiche FSM

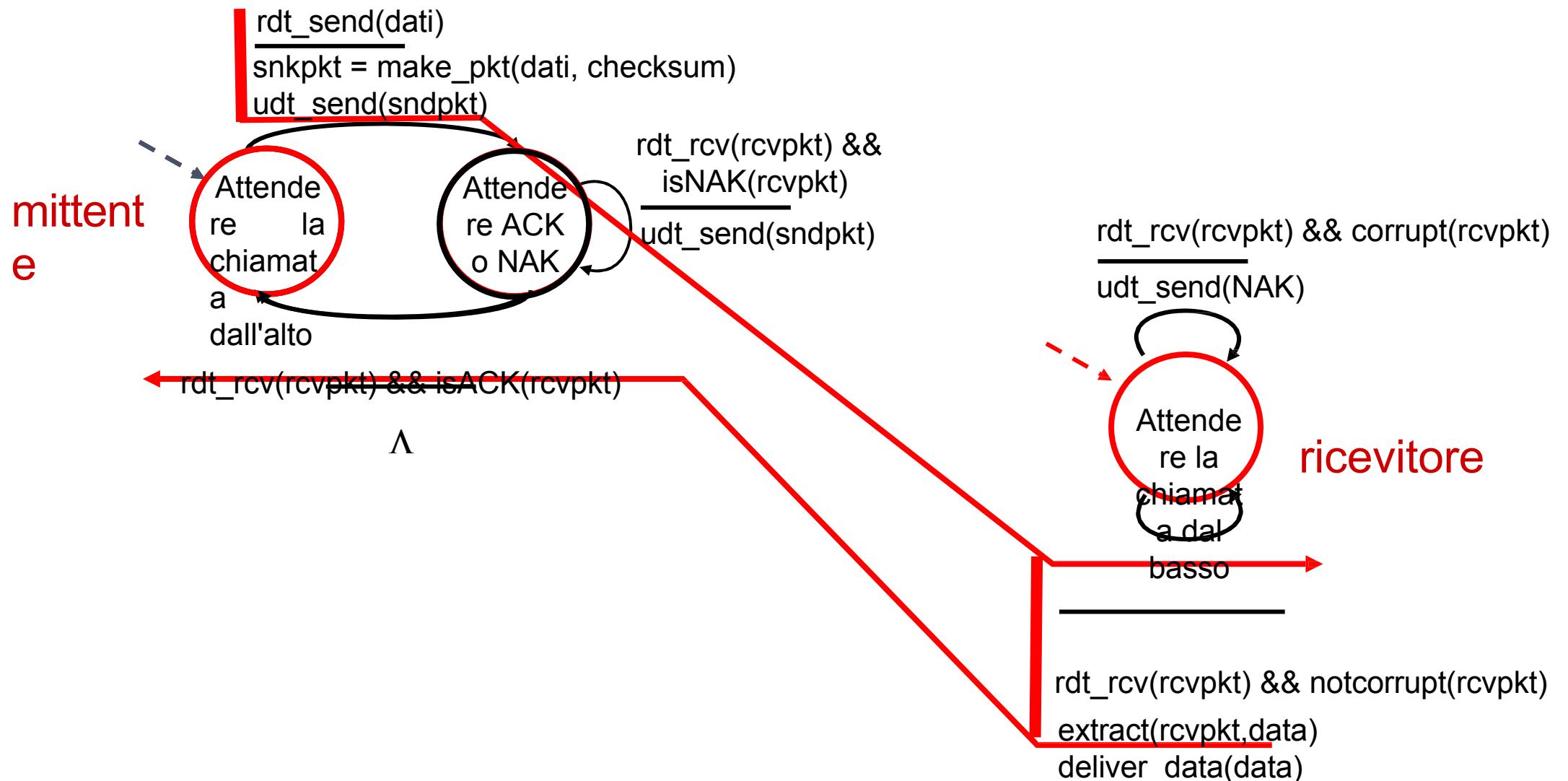


Nota: lo "stato" del destinatario (ha ricevuto correttamente il mio messaggio?) non è noto al mittente, a meno che non venga comunicato in qualche modo dal destinatario al mittente.

- Ecco perché abbiamo bisogno di un protocollo!

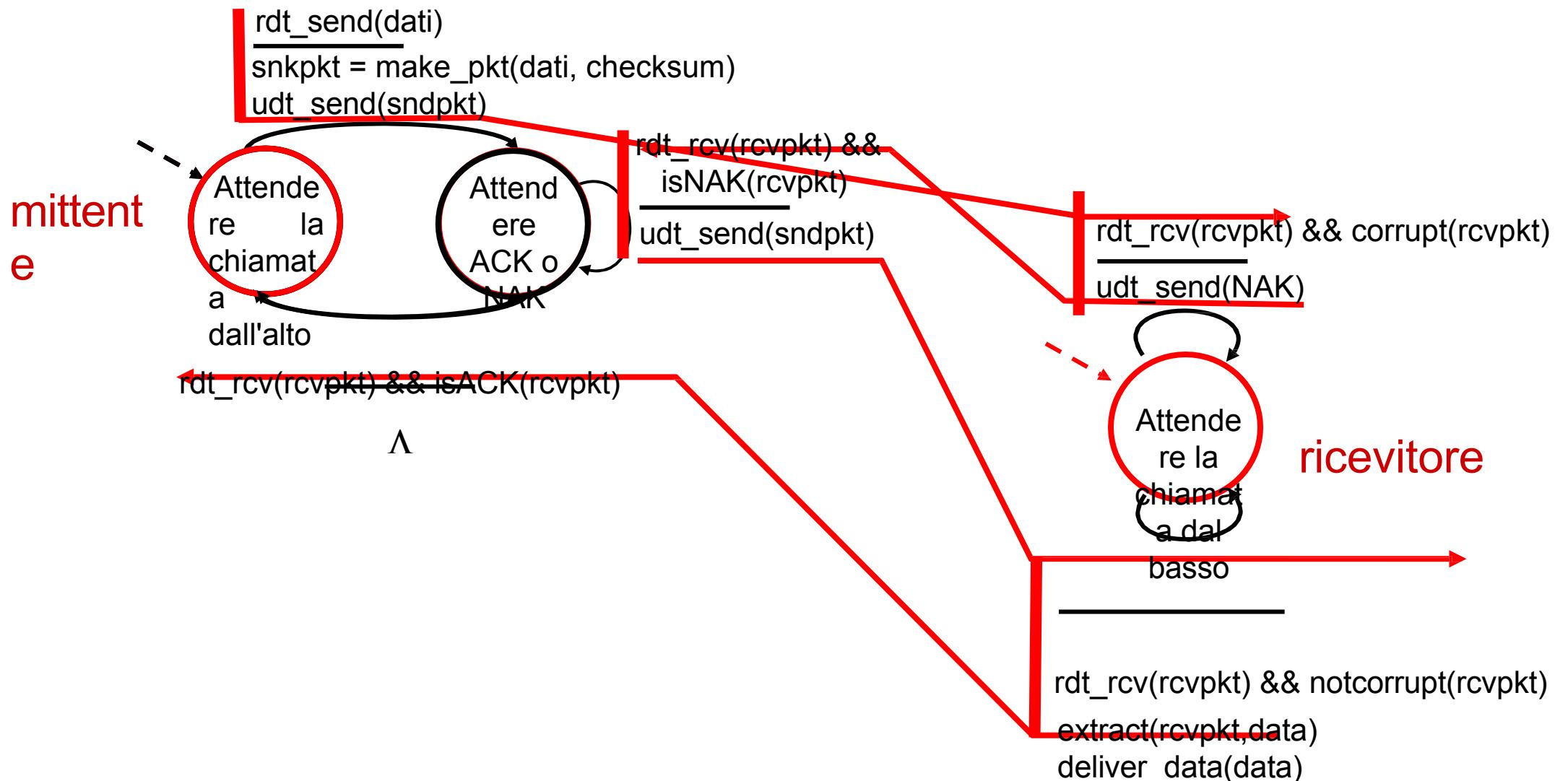


rdt2.0: operazione senza errori



`udt_send(ACK)`

rdt2.0: scenario pacchetto corrotto



`udt_send(ACK)`

rdt2.0 ha un difetto fatale!

cosa succede se ACK/NAK è corrotto?

- Il mittente non sa cosa sia successo al destinatario!
- non può semplicemente ritrasmettere: possibile duplicato

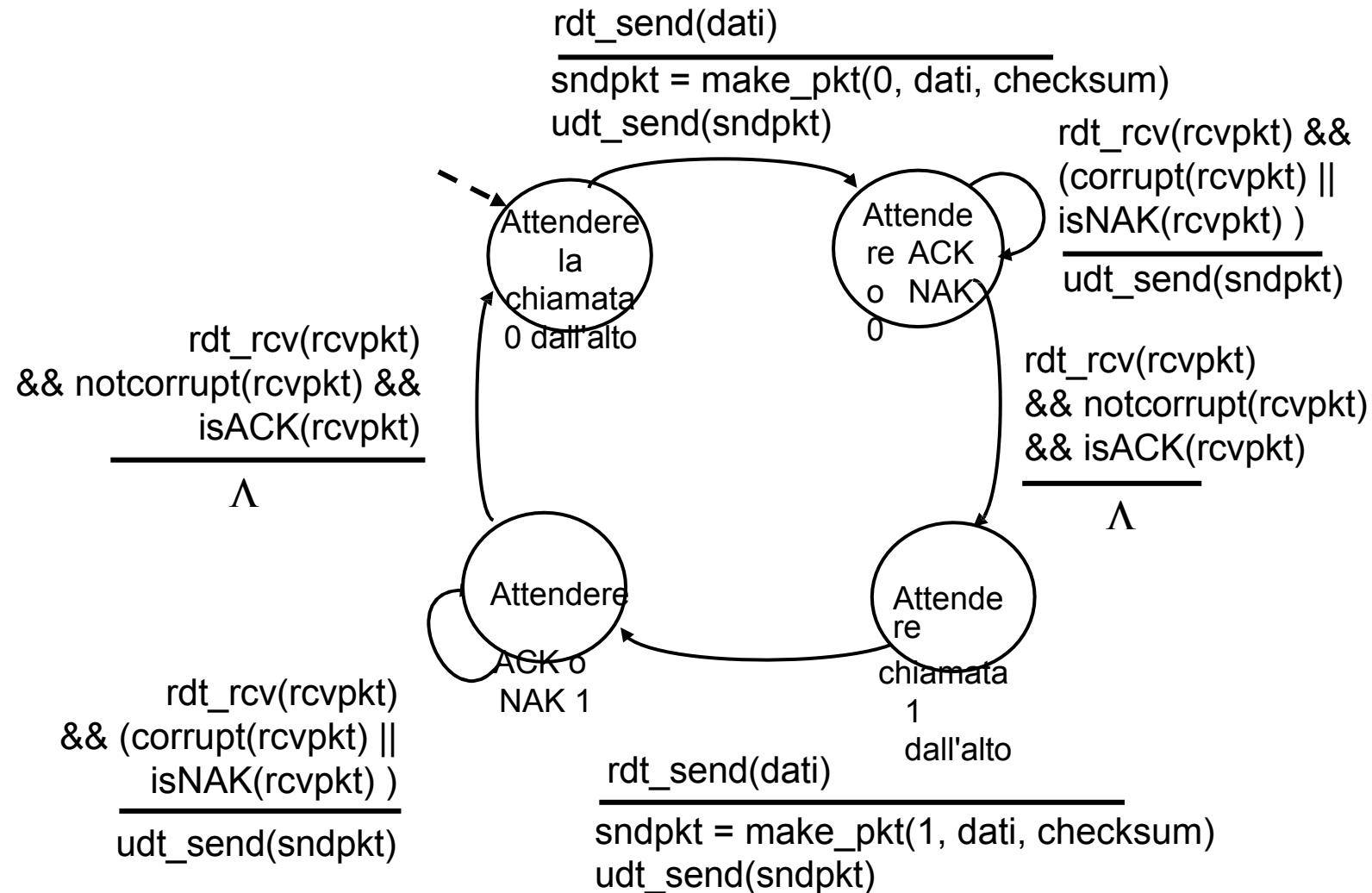
gestione dei duplicati:

- il mittente ritrasmette il pkt corrente se l'ACK/NAK è corrotto
- il mittente aggiunge *il numero di sequenza* a ciascun pkt
- il ricevitore scarta (non consegna) il pkt duplicato

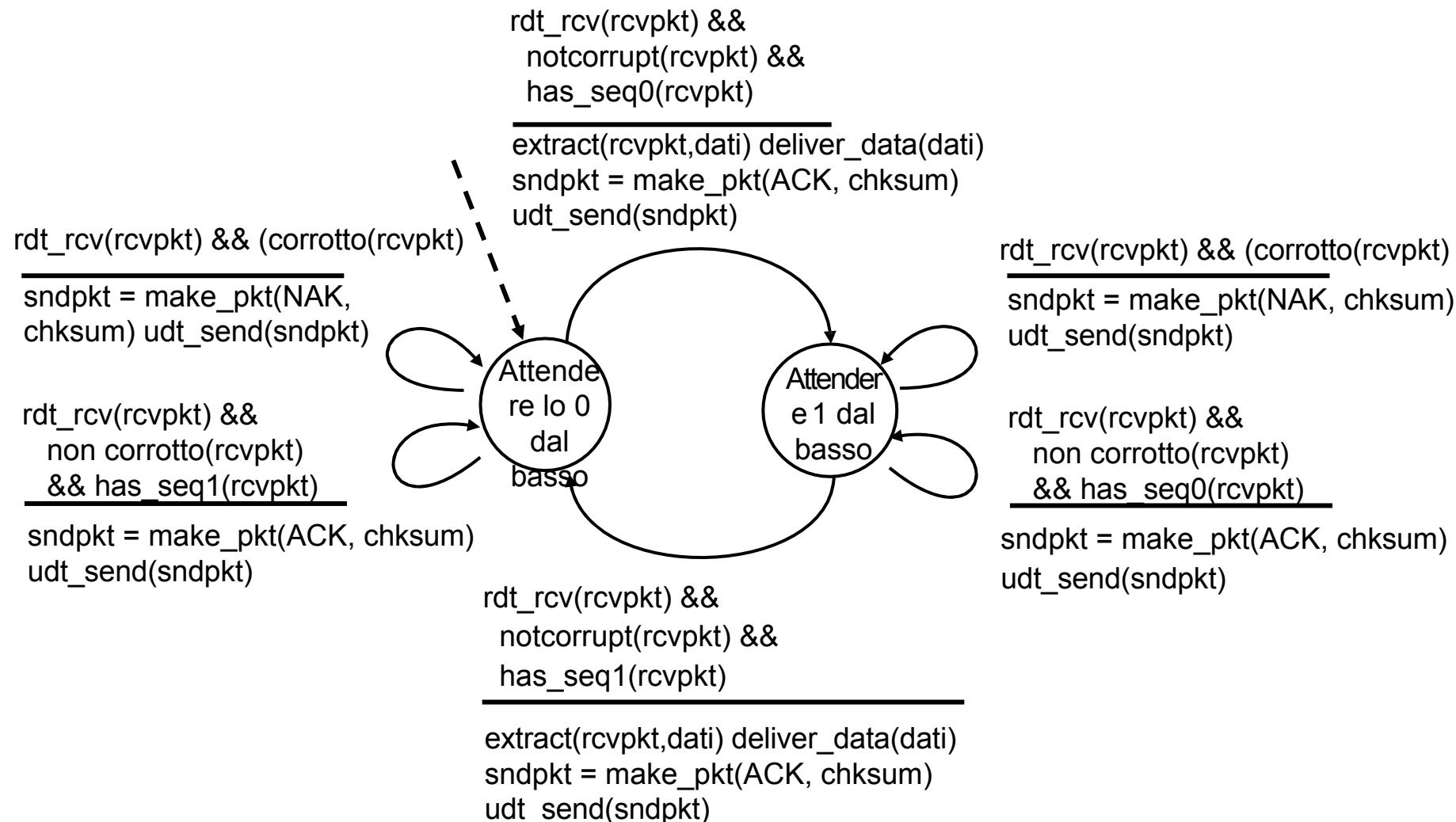
— fermarsi e aspettare

il mittente invia un pacchetto,
quindi attende la risposta del
destinatario

rdt2.1: mittente, gestione di ACK/NAK confusi



rdt2.1: ricevitore, gestione di ACK/NAK confusi



rdt2.1: discussione

mittente:

- seq # aggiunto al pkt
- due seq. #(0,1) saranno sufficienti. Perché?
- deve verificare se l'ACK/NAK ricevuto è danneggiato
- il doppio degli Stati
 - lo stato deve "ricordare" se il pkt "atteso" deve avere seq # di 0 o 1

ricevitore:

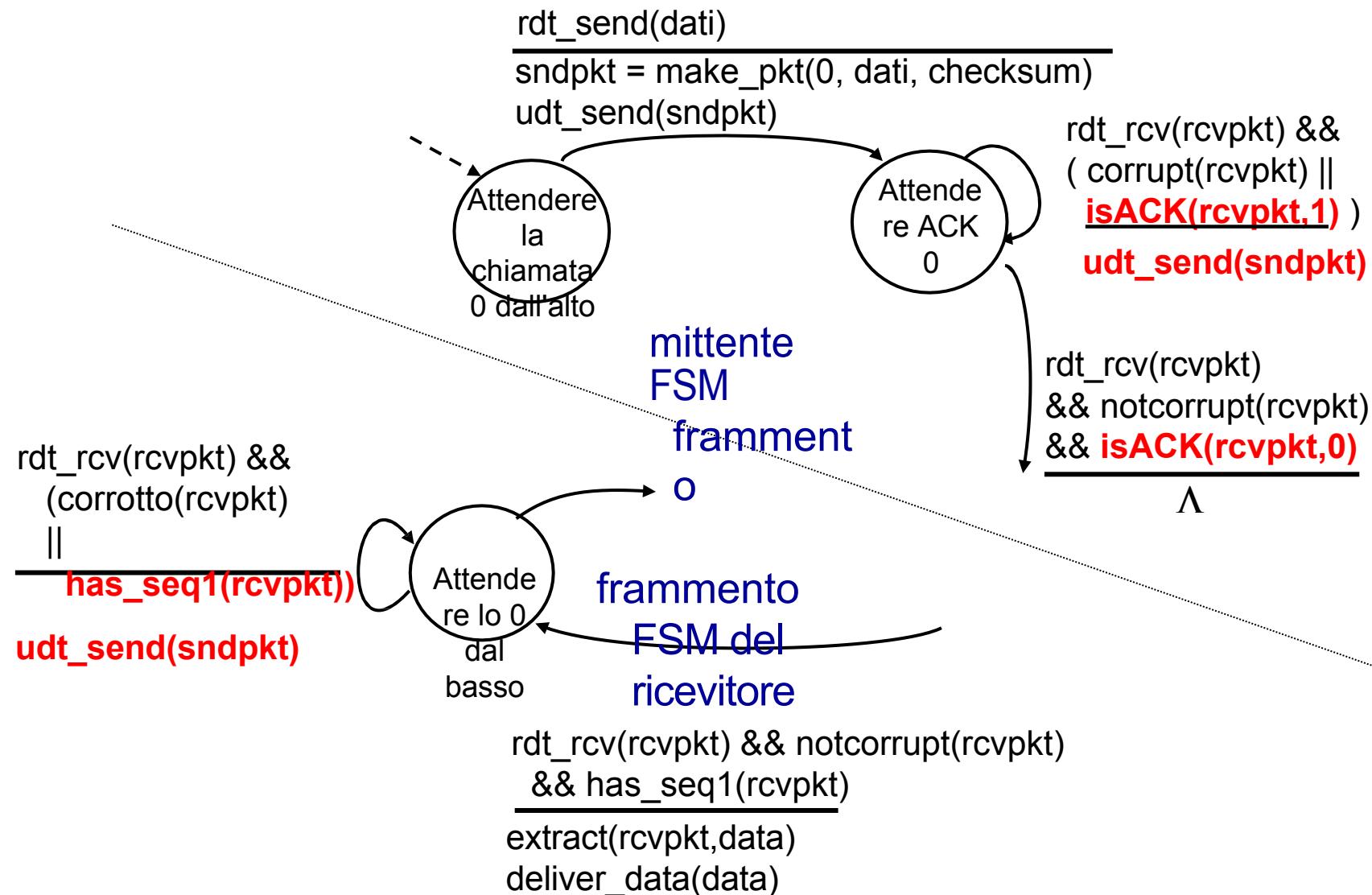
- deve verificare se il pacchetto ricevuto è duplicato
 - indica se 0 o 1 è pkt seq previsto #
- nota: il ricevitore *non* può sapere se il suo ultimo ACK/NAK è stato ricevuto dal mittente con esito positivo.

rdt2.2: un protocollo senza NAK

- stessa funzionalità di rdt2.1, utilizzando solo gli ACK
- invece di NAK, il ricevitore invia ACK per l'ultimo pkt ricevuto OK
 - il ricevitore deve includere *esplicitamente il* numero di seq del pkt in ACK
- un ACK duplicato al mittente comporta la stessa azione del NAK:
ritrasmettere il pkt corrente

Come vedremo, il protocollo TCP utilizza questo approccio per essere esente da NAK.

rdt2.2: frammenti di mittente e ricevitore



```
sndpkt = make_pkt(ACK1, cksum)
udt_send(sndpkt)
```

rdt3.0: canali con errori e perdite

Nuova ipotesi di canale: anche il canale sottostante può *perdere* pacchetti (dati, ACK)

- checksum, numeri di sequenza, ACK, ritrasmissioni saranno di aiuto... ma non abbastanza

D: Come fanno *gli esseri umani* a gestire le parole perse da mittente a destinatario nelle conversazioni?

rdt3.0: canali con errori e perdite

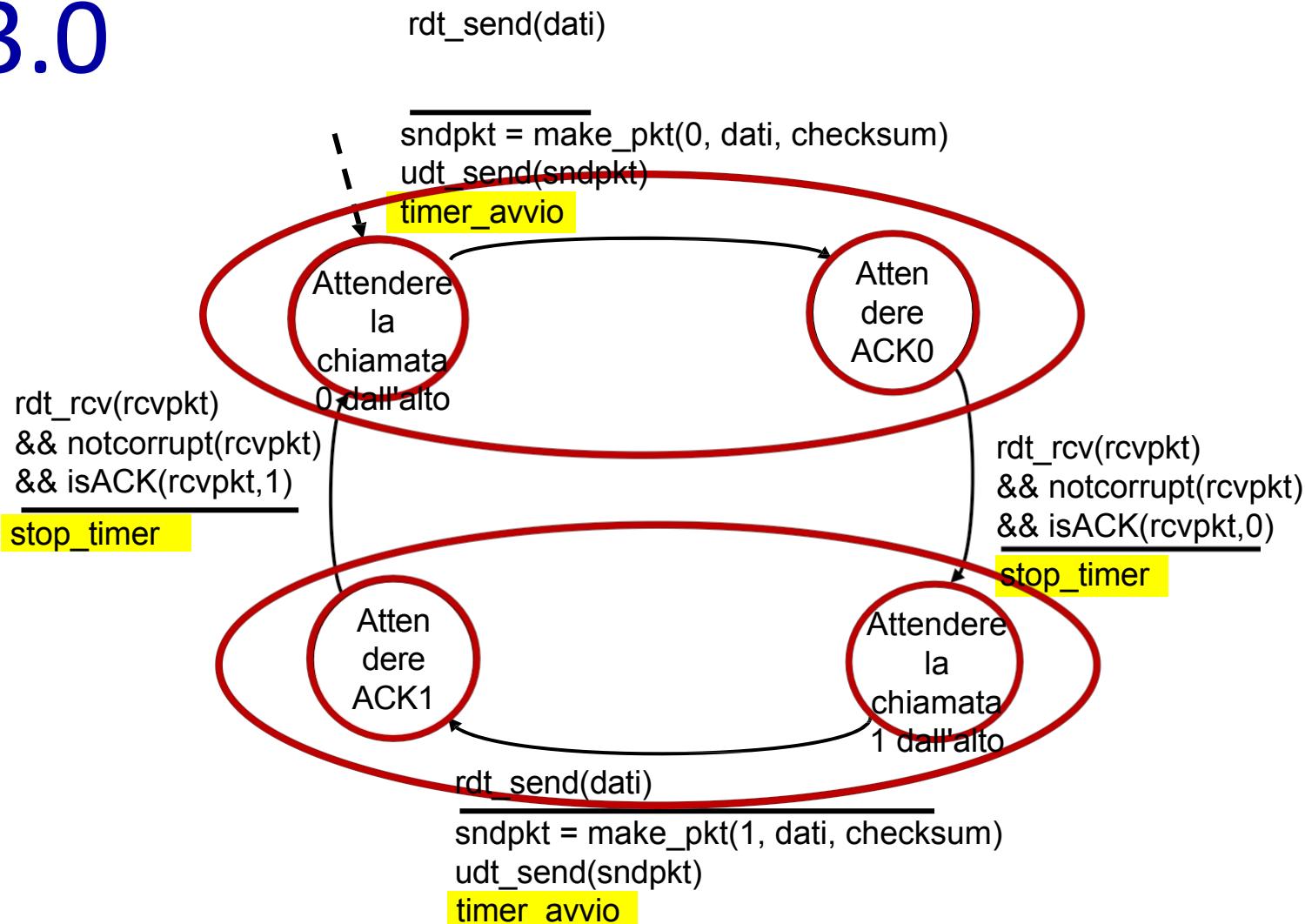
Approccio: il mittente attende un tempo "ragionevole" per l'ACK

- ritrasmette se non viene ricevuto un ACK in questo lasso di tempo
- se il pkt (o l'ACK) è solo ritardato (non perso):
 - la ritrasmissione sarà duplicata, ma seq #s gestisce già questo aspetto!
 - il ricevitore deve specificare il numero di seq del pacchetto che viene inviato come ACK
- utilizzare il timer per il conto alla rovescia per interrompere dopo un tempo "ragionevole" del tempo



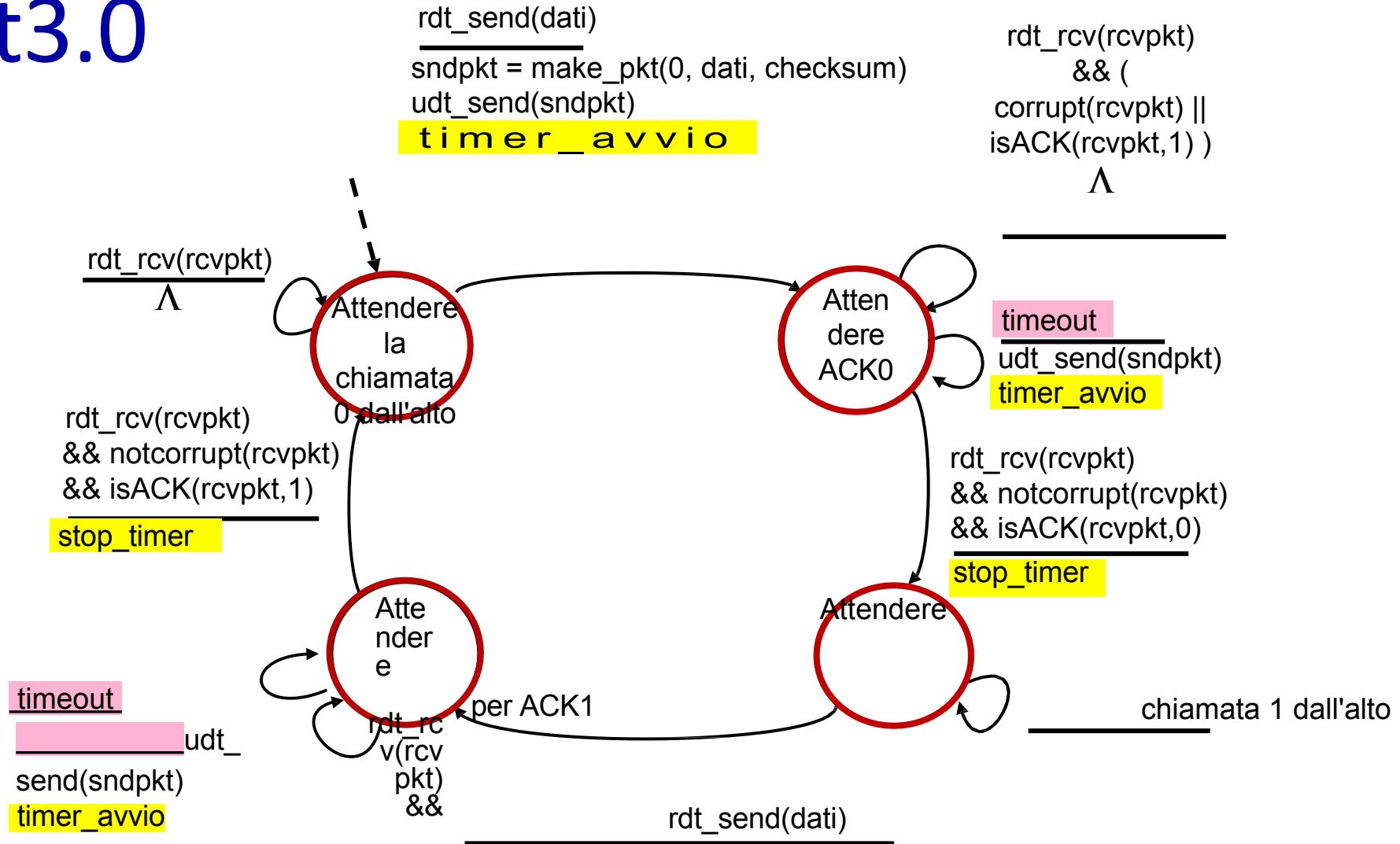
mittente

rdt3.0



mittente

rdt3.0

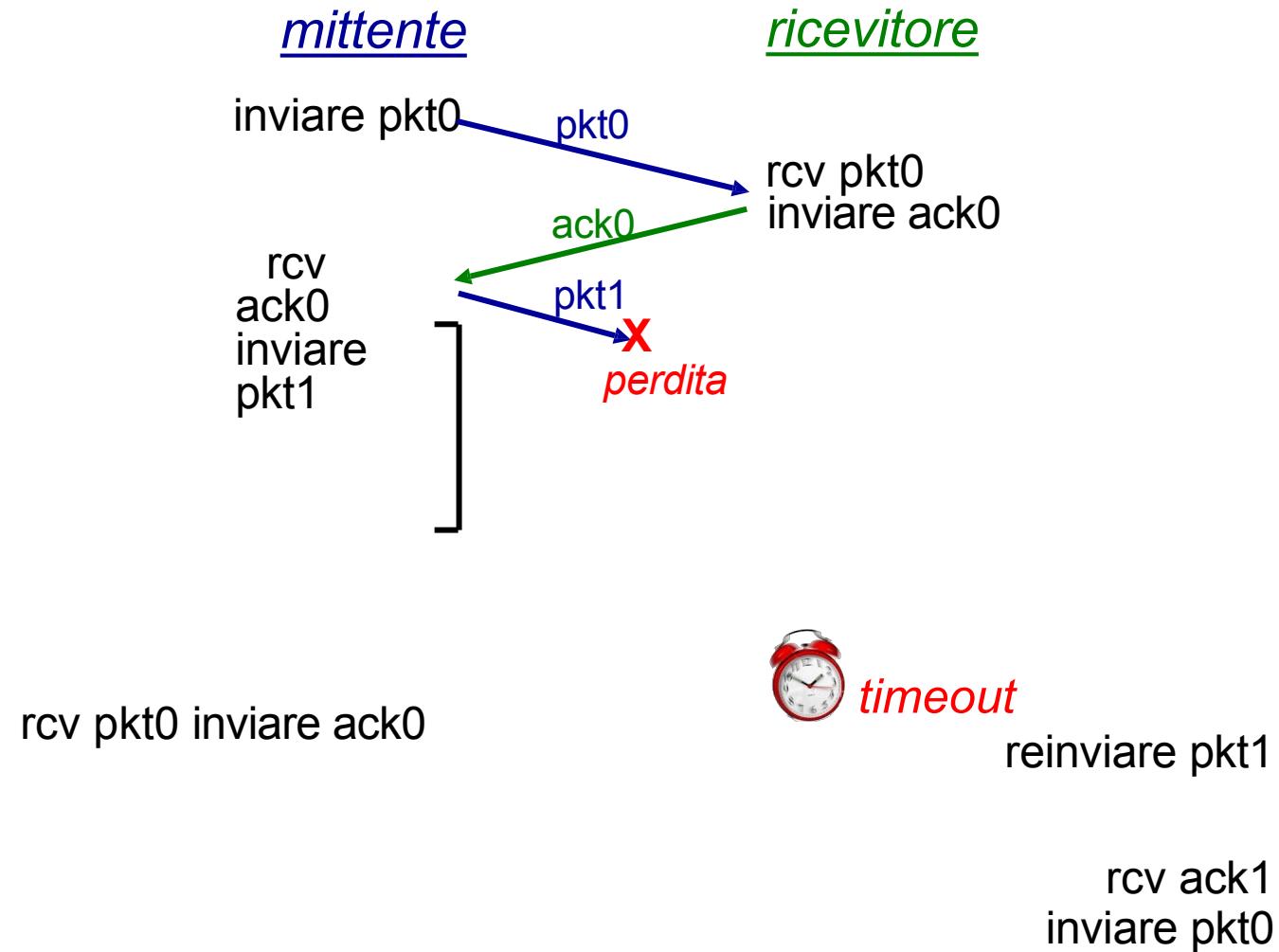
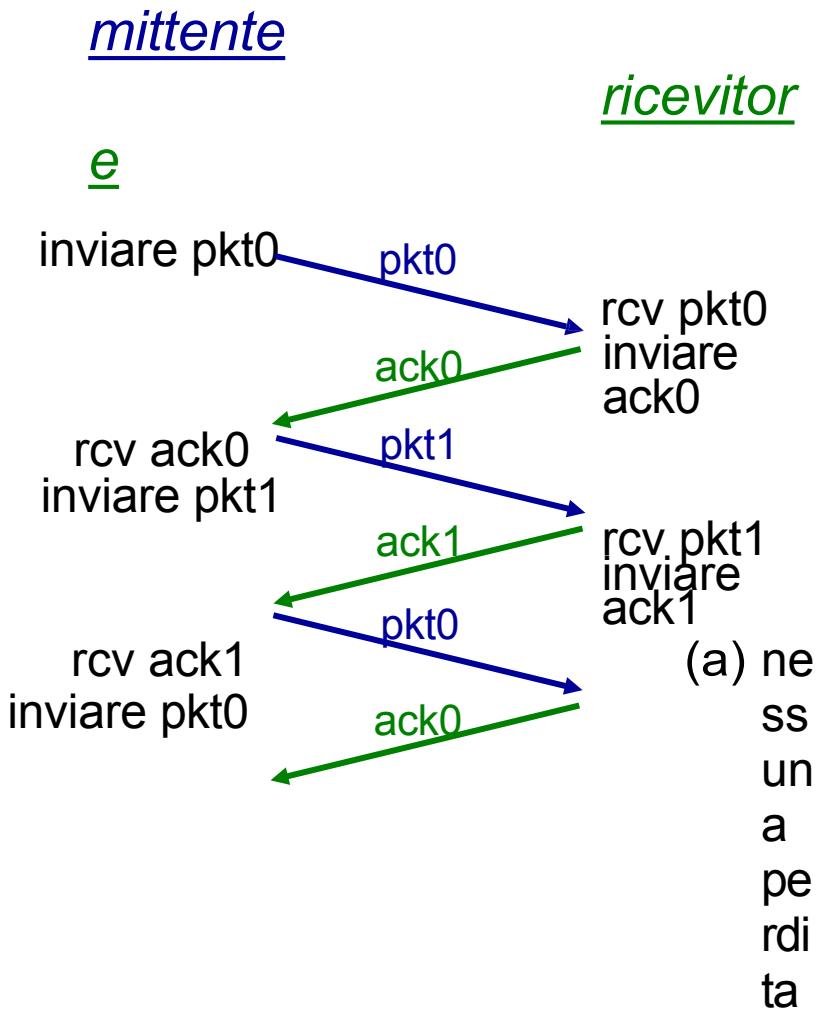


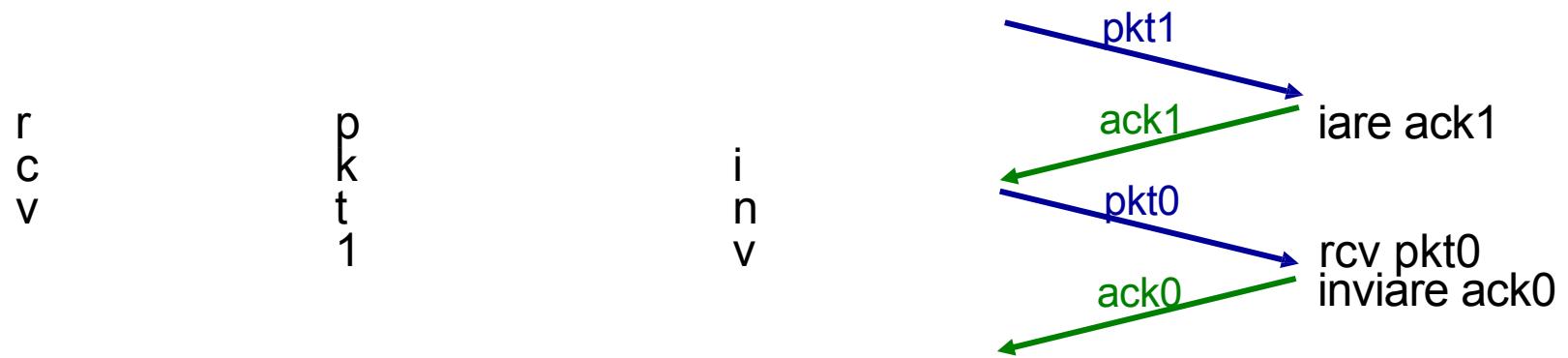
r dt_rcv(rcvp kt) Λ

(corrupt(rcvpkt) ||
isACK(rcvpkt,0)) Λ

 sndpkt = make_pkt(1, dati, checksum)
 udt_send(sndpkt)
 timer_avvio

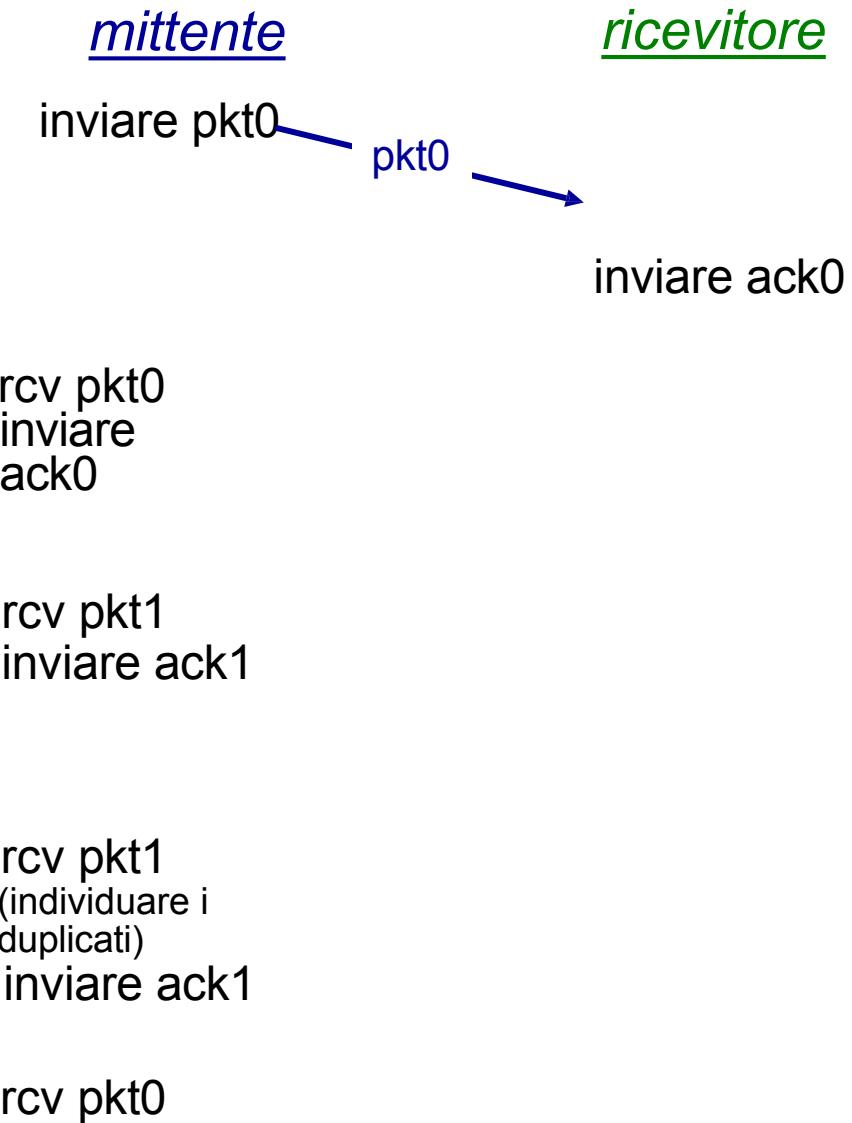
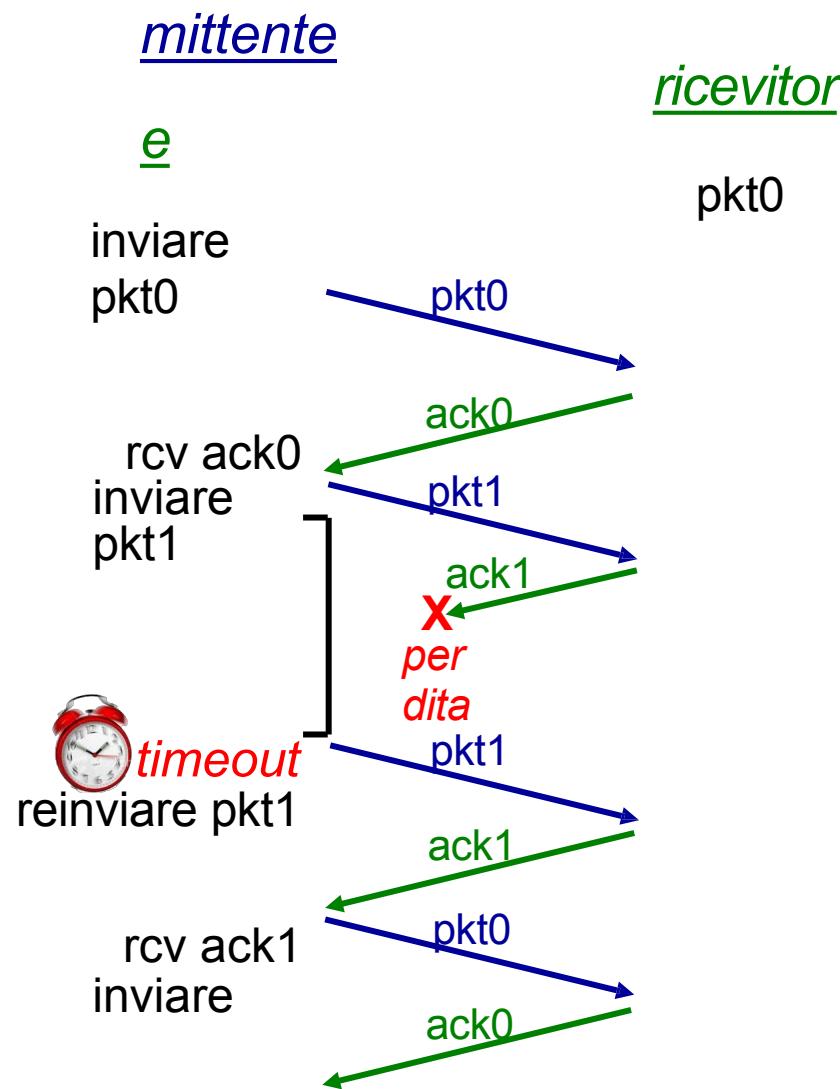
rdt3.0 in azione





(b) perdita di pacchetti

rdt3.0 in azione



rcv
ack0
inviare
pkt1

timeout
reinviare pkt1

rcv ack1
inviare
pkt0

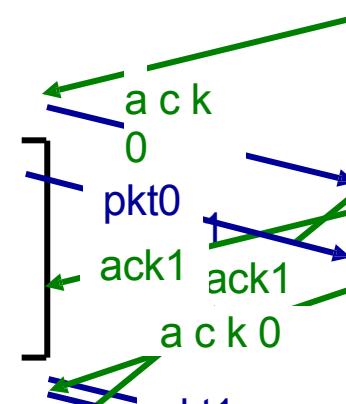
rcv ack1
(ignorare
)

r
c
v
p
k
t
0
i
n
v
i
a
r
e
a
c
k
0

r
c
v

p
k
t
1
i
n
v
i
a
r
e
a
c
k
1

rcv pkt1
(individuare i
duplicati)
inviare
ack1



(c) Perdita ACK

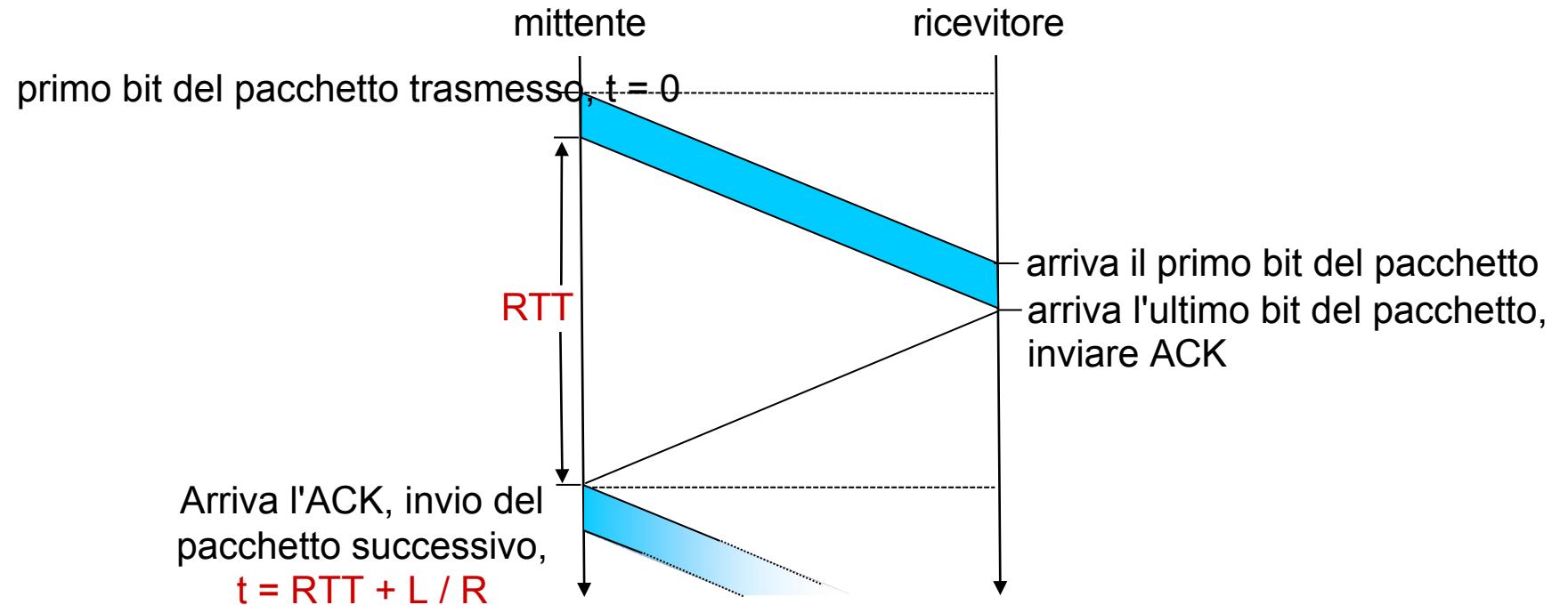
(d) timeout prematuro/ ACK ritardato

Prestazioni di rdt3.0 (stop-and-wait)

- $U_{mittente}$: *utilizzo* - frazione di tempo in cui il mittente è occupato a inviare
- esempio: collegamento a 1 Gbps, ritardo di prop. di 15 ms, pacchetto di 8000 bit
 - tempo di trasmissione del pacchetto nel canale:

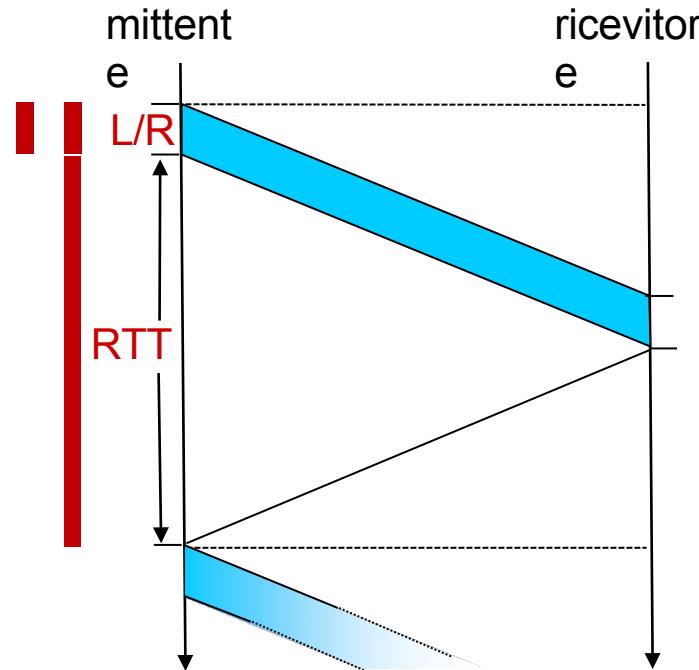
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bit}}{10^9 \text{ bit/sec}} = 8 \text{ microsec}$$

rdt3.0: operazione di arresto e attesa



rdt3.0: operazione di arresto e attesa

$$\begin{aligned}U_{\text{mittente}} &= \frac{L / R}{RTT + L / R} \\&= \frac{.008}{30.008} \\&= 0.00027\end{aligned}$$

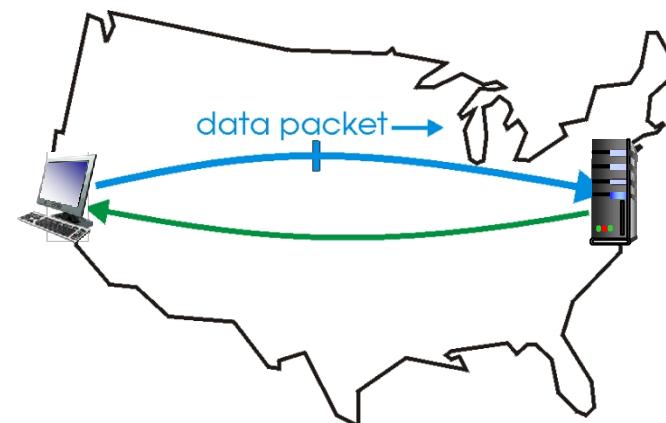


- Le prestazioni del protocollo rdt 3.0 fanno schifo!
- Il protocollo limita le prestazioni dell'infrastruttura sottostante (canale)

rdt3.0: funzionamento dei protocolli in pipeline

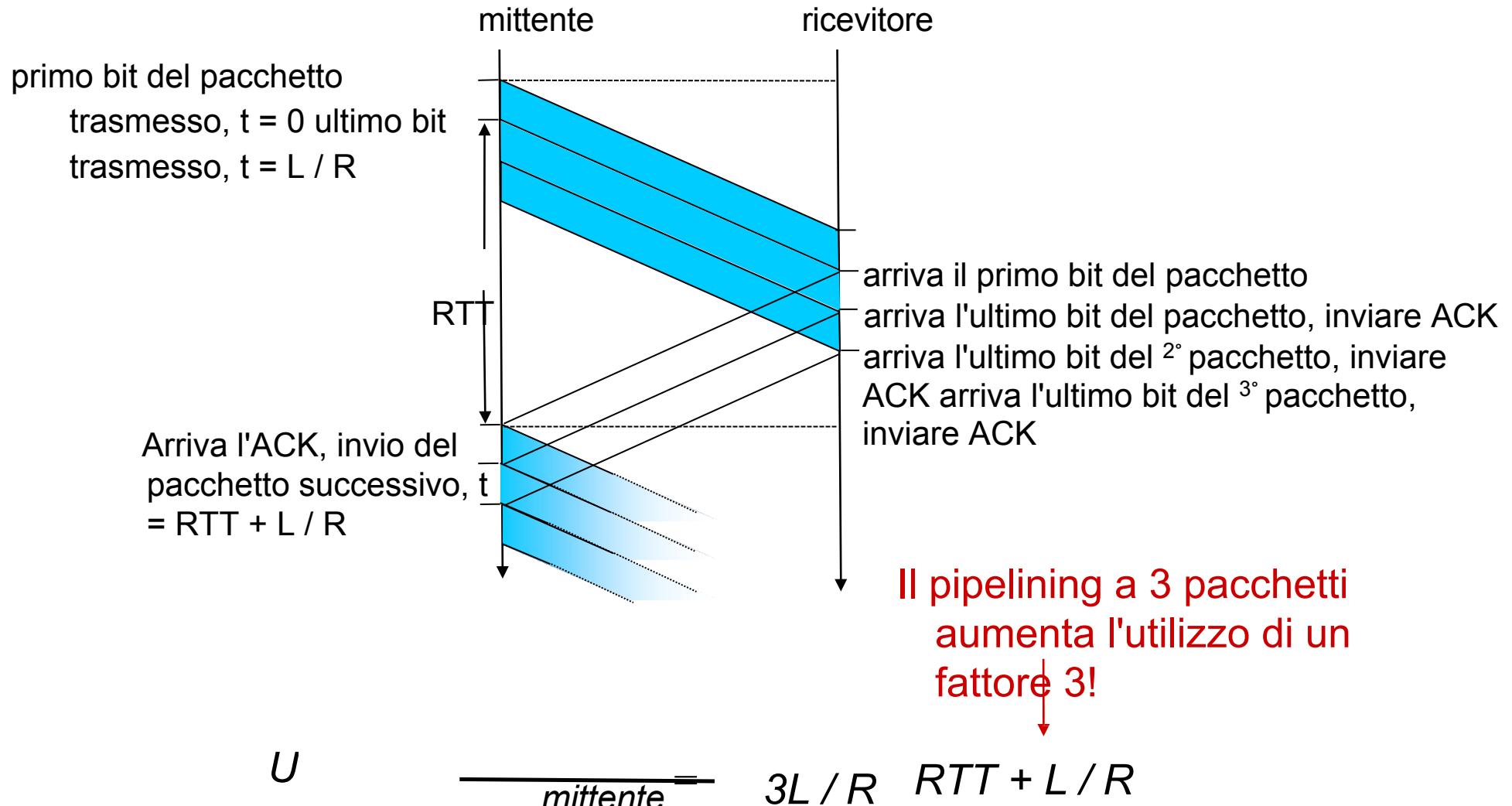
pipelining: il mittente consente l'invio di pacchetti multipli, "in volo", ancora da riconoscere

- l'intervallo dei numeri di sequenza deve essere aumentato
- buffering al mittente e/o al destinatario



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

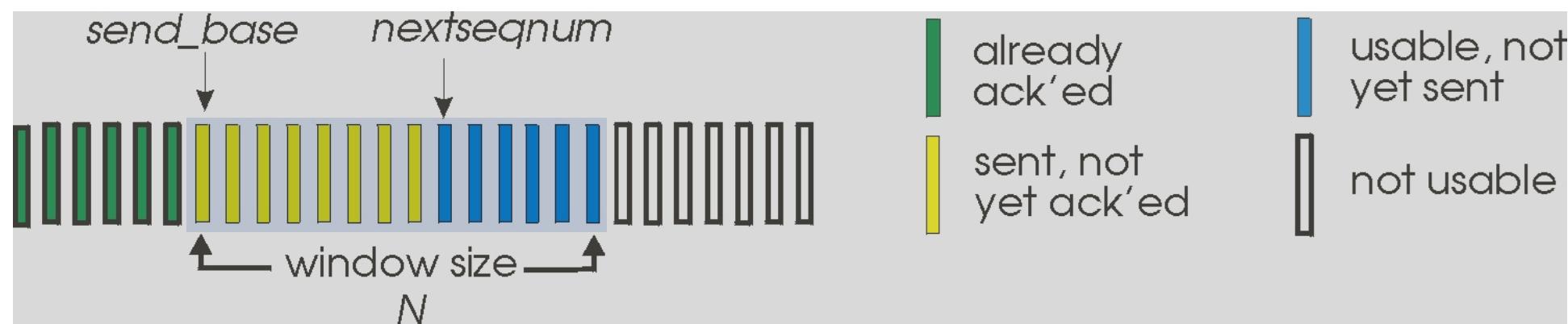
Pipelining: maggiore utilizzo



$$\begin{array}{r} .0024 \\ = 30.008 \\ \hline & 0.000 \\ & 81 \end{array}$$

Indietro-N: mittente

- mittente: "finestra" di un massimo di N pkts consecutivi trasmessi ma non accettati
 - k-bit seq # nell'intestazione del pkt



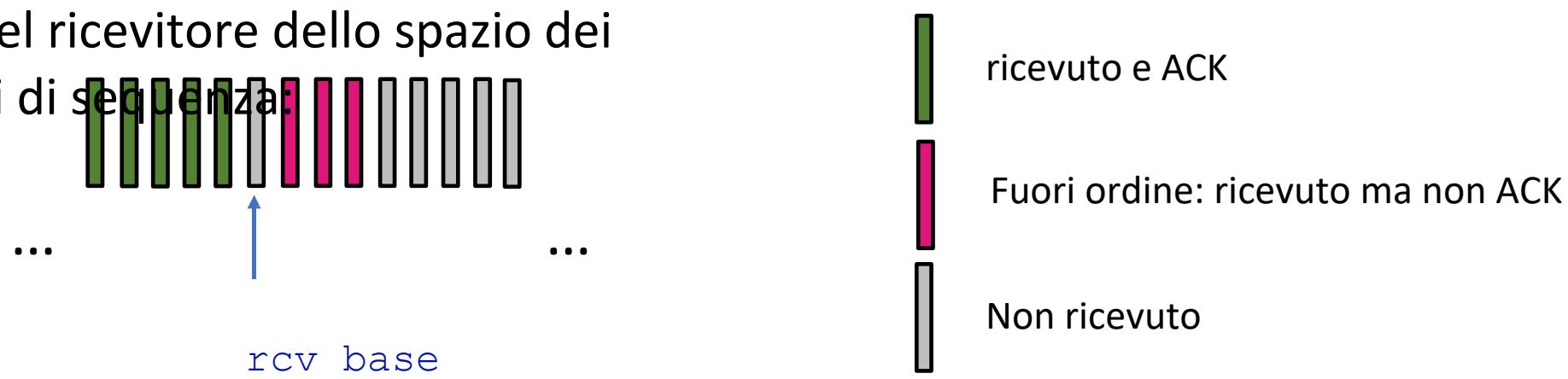
- ACK cumulativo:*** $\text{ACK}(n)$: ACK di tutti i pacchetti fino a, compreso il seq # n
 - alla ricezione dell' $\text{ACK}(n)$: sposta la finestra in avanti per iniziare da $n+1$

- timer per il pacchetto più vecchio in volo
- $\text{timeout}(n)$: ritrasmette il pacchetto n e tutti i pacchetti di seq # superiore nella finestra

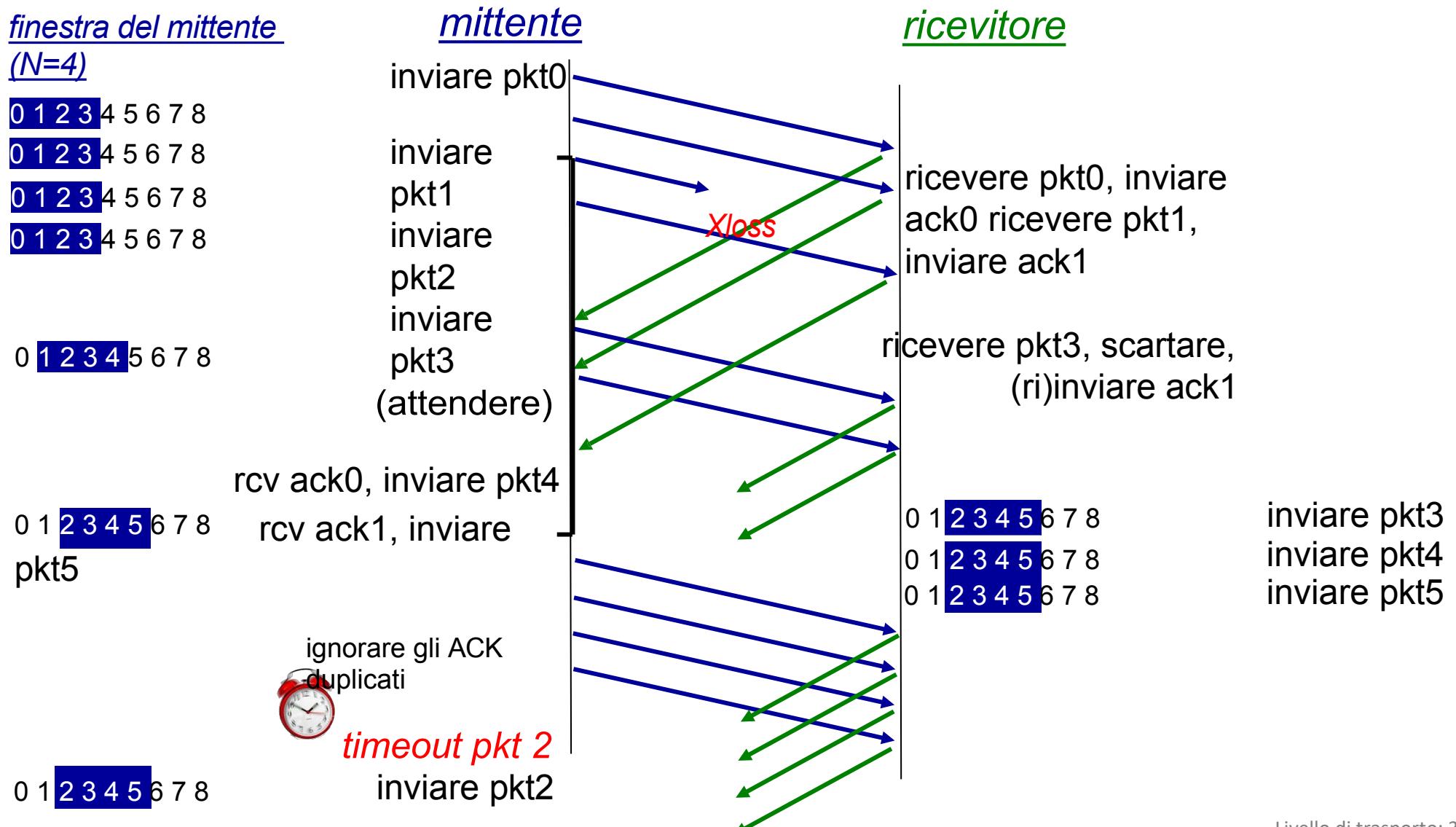
Go-Back-N: ricevitore

- Solo ACK: invia sempre ACK per i pacchetti ricevuti correttamente fino a quel momento, con il più alto numero di seq *in ordine*.
 - può generare ACK duplicati
 - deve solo ricordare `rcv_base`
- al ricevimento di un pacchetto fuori ordine:
 - può scartare (non bufferizzare) o bufferizzare: una decisione di implementazione.
 - ri-ACCETTA il pkt con il seq in-ordine più alto #

Vista del ricevitore dello spazio dei numeri di sequenza:



Go-Back-N in azione



ricevere pkt4, scartare,
(ri)inviare
ack1 ricevere
pkt5, scartare,
(ri)inviare ack1

rcv pkt2, consegnare,
inviare ack2 rcv pkt3,
consegnare, inviare
ack3 rcv pkt4,
consegnare, inviare
ack4 rcv pkt5,
consegnare, inviare
ack5

Ripetizione selettiva: l'approccio

- *pipelining*: più pacchetti in volo
- *Il ricevitore accetta singolarmente* tutti i pacchetti ricevuti correttamente.
 - bufferizza i pacchetti, se necessario, per consegnarli in ordine al livello superiore
- mittente:
 - mantiene (concettualmente) un timer per ogni pkt non accettato
 - timeout: ritrasmette il singolo pacchetto non accettato associato al timeout
 - mantiene (concettualmente) una "finestra" su N seq # consecutive

- limita i pacchetti in pipeline, "in volo", per rientrare in questa finestra

Ripetizione selettiva: finestre del mittente e del destinatario

Livello di trasporto: 3-73

Ripetizione selettiva: mittente e destinatario

mittente

dati di cui sopra:

- se il prossimo seq # disponibile nella finestra, invia il pacchetto

timeout(n):

- reinvia il pacchetto n , riavvia il timer **ACK(n)** in $[sendbase, sendbase+N-1]$:
- contrassegnare il pacchetto n come ricevuto
- se n è il più piccolo pacchetto non accettato, avanza la base della finestra alla successiva sequenza non accettata #

ricevitore

pacchetto n in $[rcvbase, rcvbase+N-1]$

- inviare ACK(n)
- fuori ordine: buffer
- in-order: consegna (anche consegna di pacchetti bufferizzati, in-order), avanzamento della finestra al prossimo pacchetto non ancora ricevuto

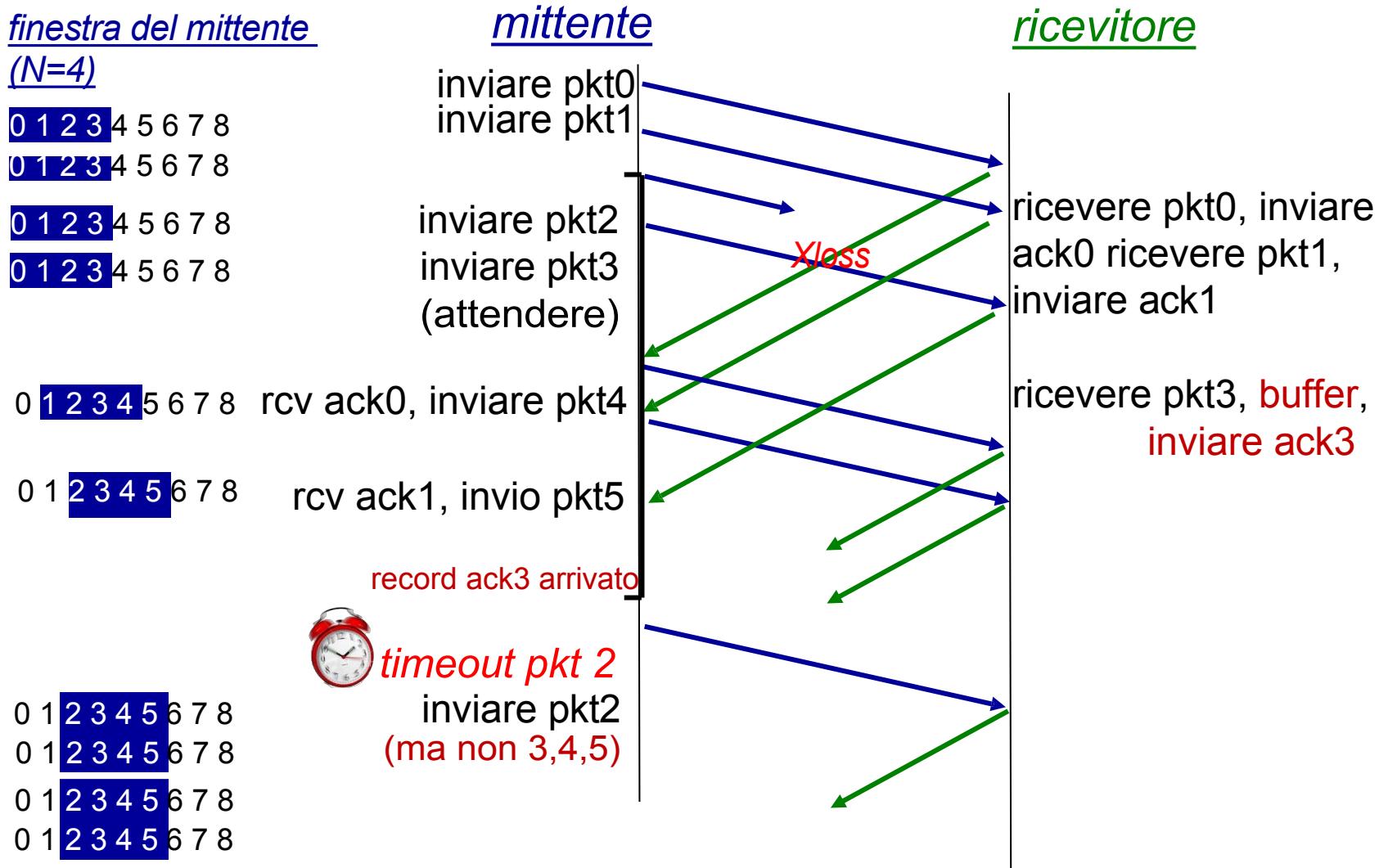
pacchetto n in $[rcvbase-N, rcvbase-1]$

- ACK(n)

altrimenti:

- ignorare

Ripetizione selettiva in azione



ccede quando arriva ack2?

ricevere pkt4, buffer,
inviare ack4
ricevere pkt5, buffer,
inviare ack5

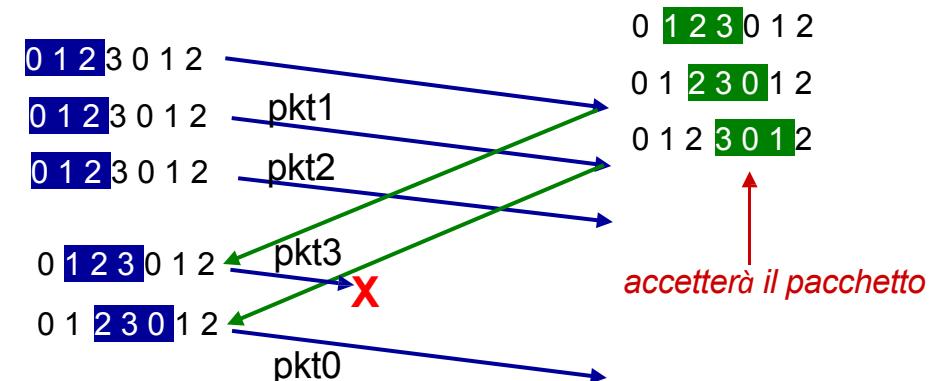
rcv pkt2; deliver pkt2, pkt3,
pkt4, pkt5; send ack2

Ripetizione selettiva: un dilemma!

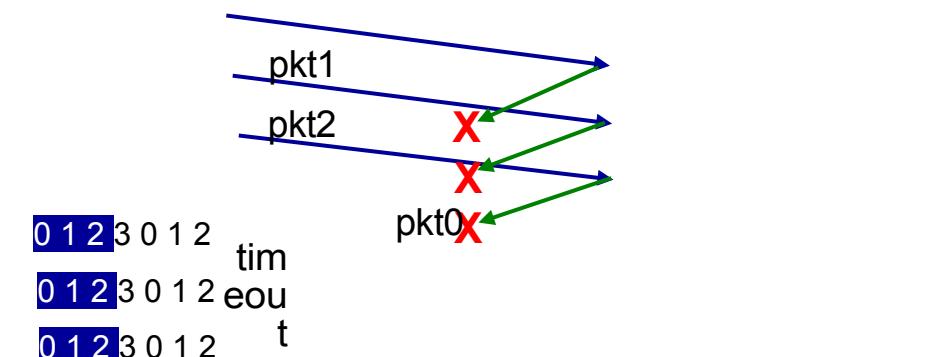
esempio:

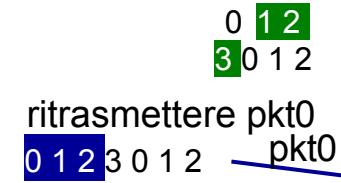
- seq #: 0, 1, 2, 3 (conteggio in base 4)
- dimensione finestra=3

finestra del mittente
(dopo il ricevimento)
pkt0



(a) Nessun problema





(b) oops!

0 1 2 3 0 1 2
0 1 2 3 0 1 2

accetterà i pacchetti con numero di seq 0

Ripetizione selettiva: un dilemma!

esempio:

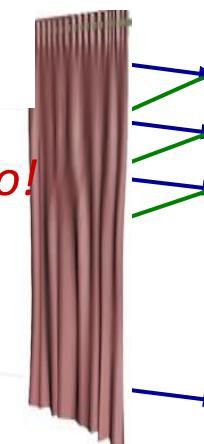
- seq #: 0, 1, 2, 3 (conteggio in base 4)
- dimensione finestra=3

D: Quale relazione è necessaria tra la dimensione della sequenza # e la dimensione

finestra del mittente
(dopo il



- *il ricevitore non può vedere il lato mittente*
- *comportamento del ricevitore identico in entrambi i casi!*
- ***qualcosa è (Molto) sbagliato!***



(b)oops!

nello scenario (b)?

X	0 1 2 3 0 1 2
X	0 1 2 3 0 1 2
X	0 1 2 3 0 1 2

↑

accetterà i pacchetti con numero di seq 0

Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- **Trasporto orientato alla connessione: TCP**
 - struttura del segmento
 - trasferimento dati affidabile
 - controllo del flusso
 - gestione delle connessioni
- Principi del controllo della congestione
- Controllo della congestione TCP



TCP: panoramica RFC: 793, 1122, 2018, 5681, 7323

- punto a punto:
 - un mittente, un destinatario
- *vapore di byte*
affidabile e in ordine:
 - nessun "confine del messaggio"
- dati full duplex:
 - flusso di dati bidirezionale nella stessa connessione
- MSS: dimensione massima del segmento

- ACK cumulativi
- pipelining:
 - Il controllo della congestione e del flusso TCP imposta la dimensione della finestra
- orientato alla connessione:
 - handshaking (scambio di messaggi di controllo): inizializza lo stato del mittente e del destinatario prima dello scambio di dati.
 - flusso controllato:
 - il mittente non deve sopraffare il destinatario

Struttura del segmento TCP



dell'applicazione
(lunghezza variabile)

dati inviati
dall'applicaz
ione nel
socket TCP

Numeri di sequenza TCP, ACK

Numeri di sequenza:

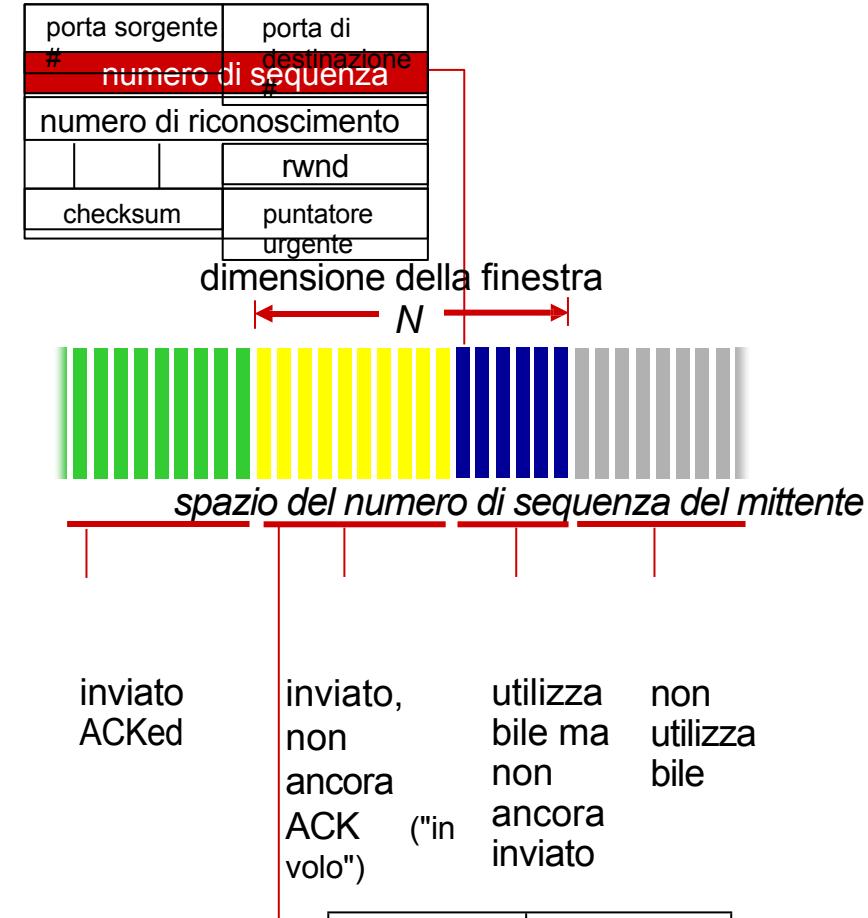
- flusso di byte "numero" di primo byte nei dati del segmento

Ringraziamenti:

- seq # del prossimo byte atteso dall'altro lato
- ACK cumulativo

D: come il ricevitore gestisce il fuori servizio
segmenti d'ordine

segmento in uscita dal mittente



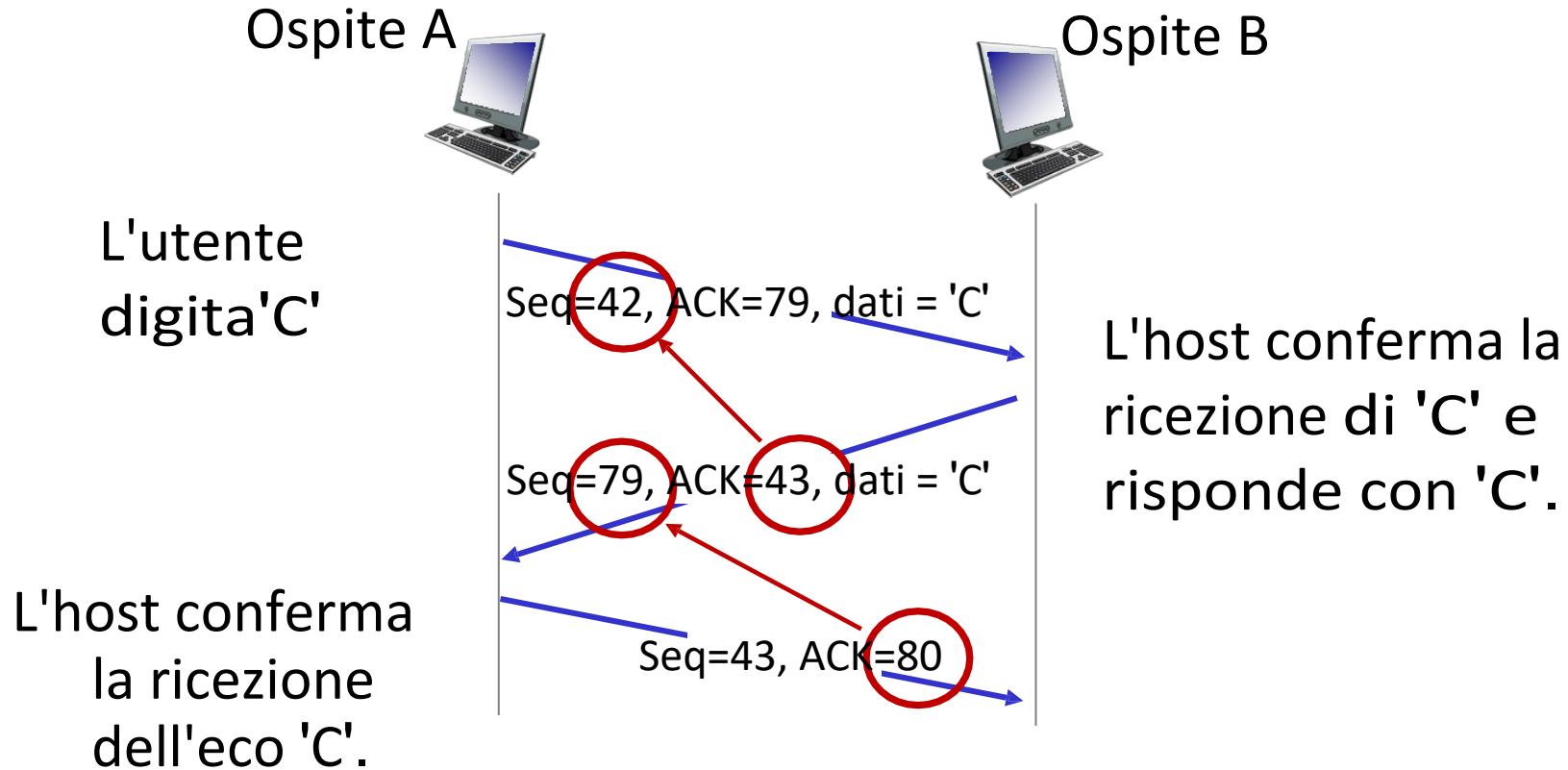
porta sorgente	porta di destinazione
#	numero di sequenza
numero di riconoscimento	
A	rwnd

- **R: Le specifiche TCP non lo**

dicono, - dipende
dall'implementatore.

segmento in uscita dal ricevitore

Numeri di sequenza TCP, ACK



semplice scenario telnet

Tempo di andata e ritorno TCP, timeout

- D:** Come impostare il valore di timeout TCP?
- più lungo dell'RTT, ma l'RTT varia!
 - *troppo breve*: timeout prematuro, ritrasmissioni non necessarie
 - *troppo a lungo*: reazione lenta alla perdita del segmento

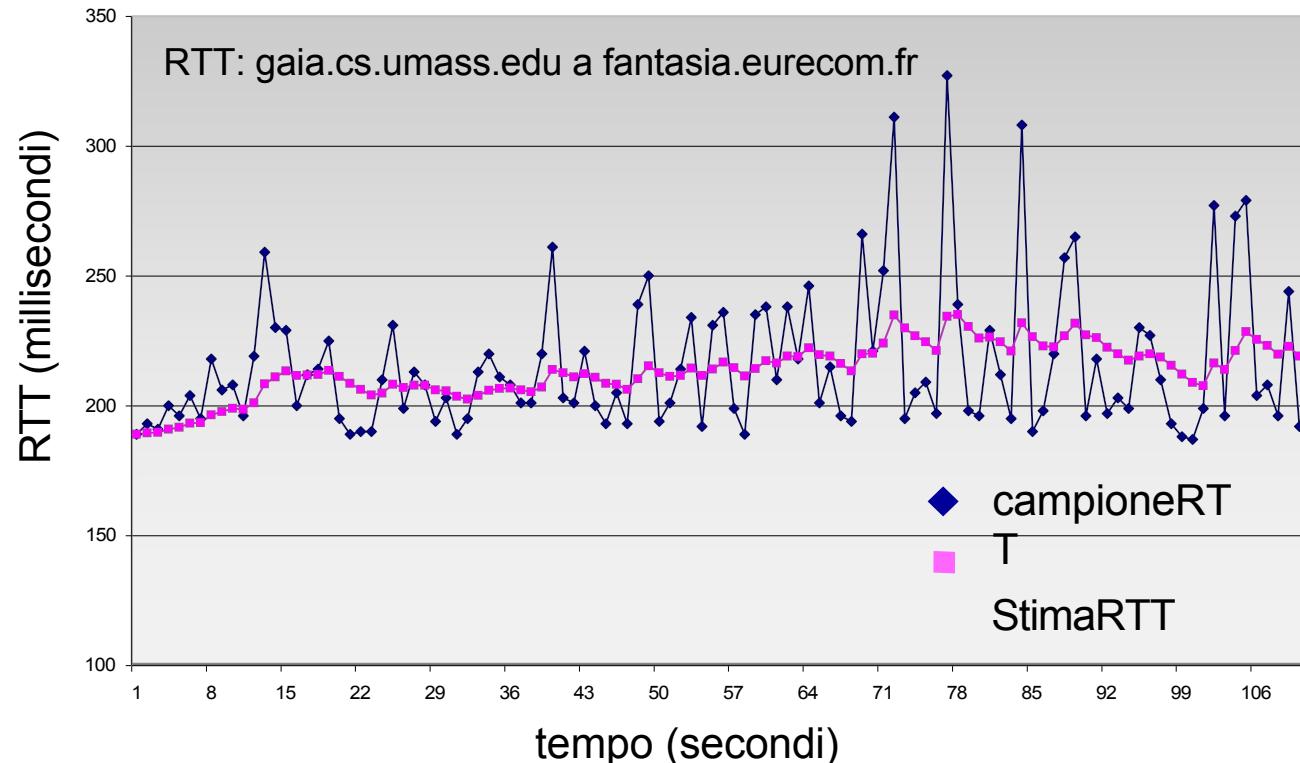
- D:** come stimare l'RTT?

- *SampleRTT*: tempo misurato dalla trasmissione del segmento fino alla ricezione dell'ACK
 - ignorare le ritrasmissioni
- Il *SampleRTT* varierà, si vuole RTT stimato "più morbido"
 - media di diverse misurazioni *recenti*, non solo del *SampleRTT* corrente

Tempo di andata e ritorno TCP, timeout

$$\text{EstimatedRTT} = (1 - A) * \text{EstimatedRTT} + \alpha * \text{SampleRTT}$$

- media mobile ponderata esponenziale (EWMA)
- l'influenza del campione passato diminuisce in modo esponenziale e veloce
- valore tipico: $A = 0,125$



Tempo di andata e ritorno TCP, timeout

- intervallo di timeout: **StimaRTT** più "margini di sicurezza"
 - Grande variazione nell'**EstimatedRTT**: si desidera un margine di sicurezza più ampio.

$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$



RTT stimato

"margine di
sicurezza"

- **DevRTT**: EWMA della deviazione di **SampleRTT** da **EstimatedRTT**:

$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * | \text{SampleRTT} - \text{EstimatedRTT} |$$

(in genere, $B = 0,25$)

* Per ulteriori esempi, consultate gli esercizi interattivi online: http://gaia.cs.umass.edu/kurose_ross/interactive/.

Mittente TCP (semplificato)

evento: dati ricevuti
dall'applicazione

- intervallo di scadenza:
TimeOutInterval
- creare un segmento con seq #
- seq # è il numero di byte-stream del primo byte di dati nel segmento
- avvia il timer se non è già in esecuzione
 - Il timer si riferisce al segmento non accettato più vecchio.

evento: timeout

- ritrasmettere il segmento che ha causato il timeout
- riavviare il timer

- avviare il timer se ci sono ancora segmenti non accettati

evento: ACK ricevuto

- se l'ACK riconosce i segmenti precedentemente non accettati
 - aggiornare ciò che è noto per essere ACKed

Ricevitore TCP: Generazione ACK [RFC 5681]

*Evento al
ricevitore*

*Azione del
ricevitore TCP*

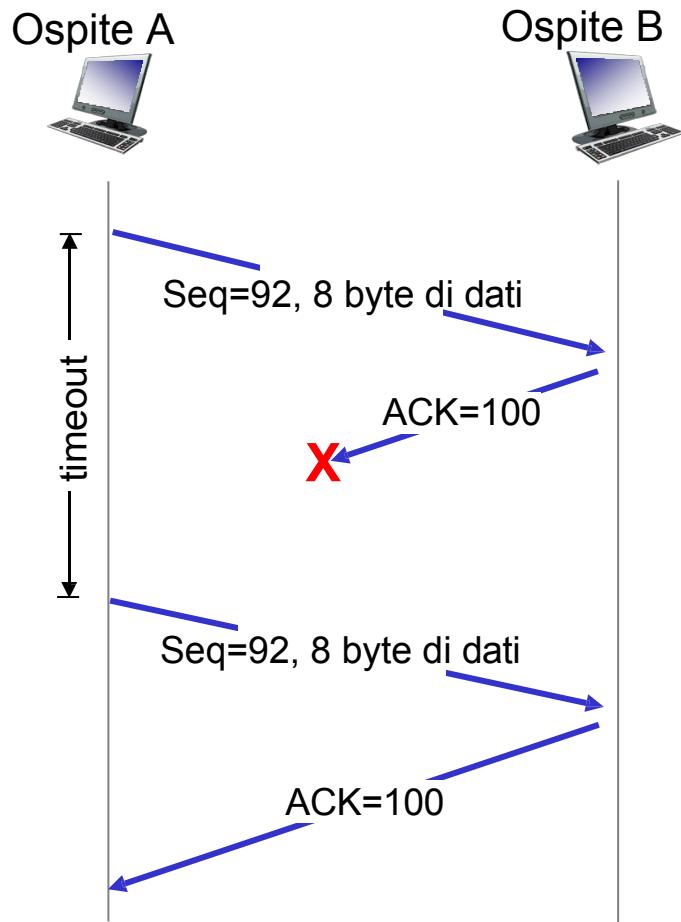
~~un segmento successivo, il seq # atteso e già stato inviato ACK.~~

~~ordine. il segmento ha un ACK in attesa.~~

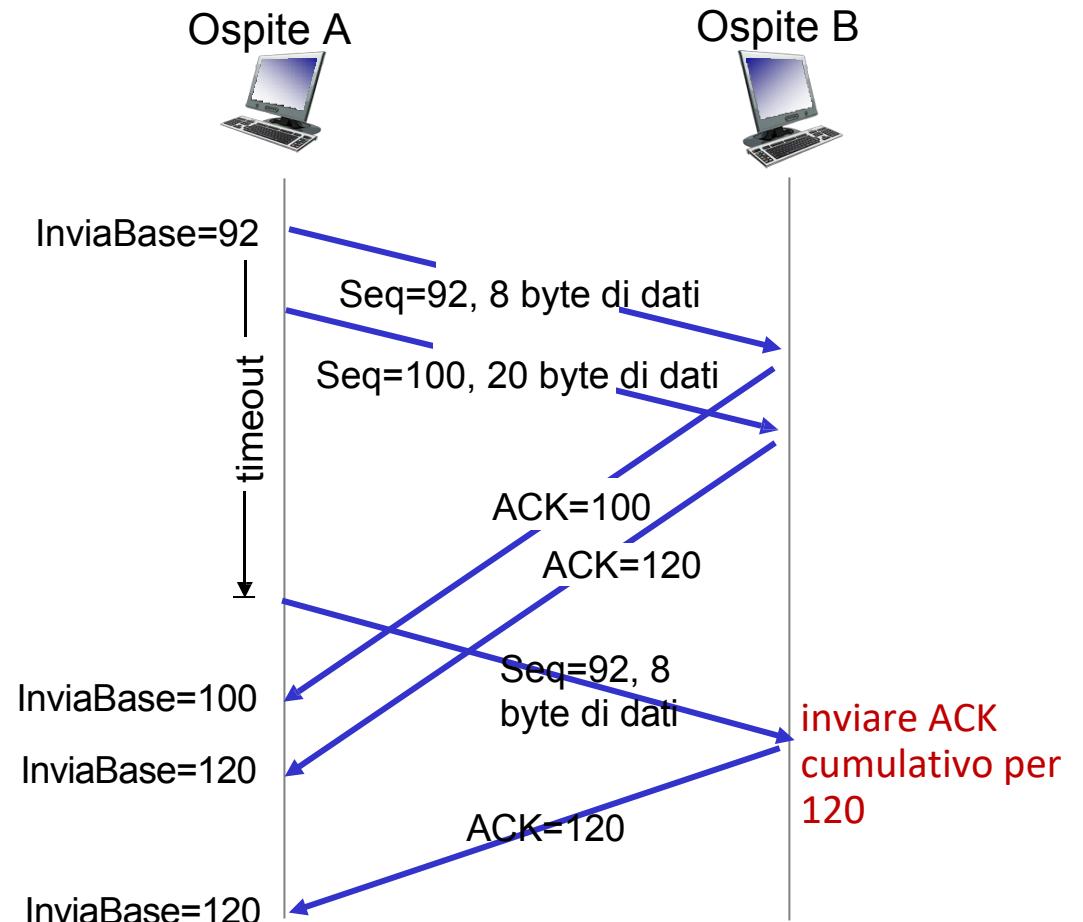
~~atteso~~

all'estremità inferiore della lacuna

TCP: scenari di ritrasmissione

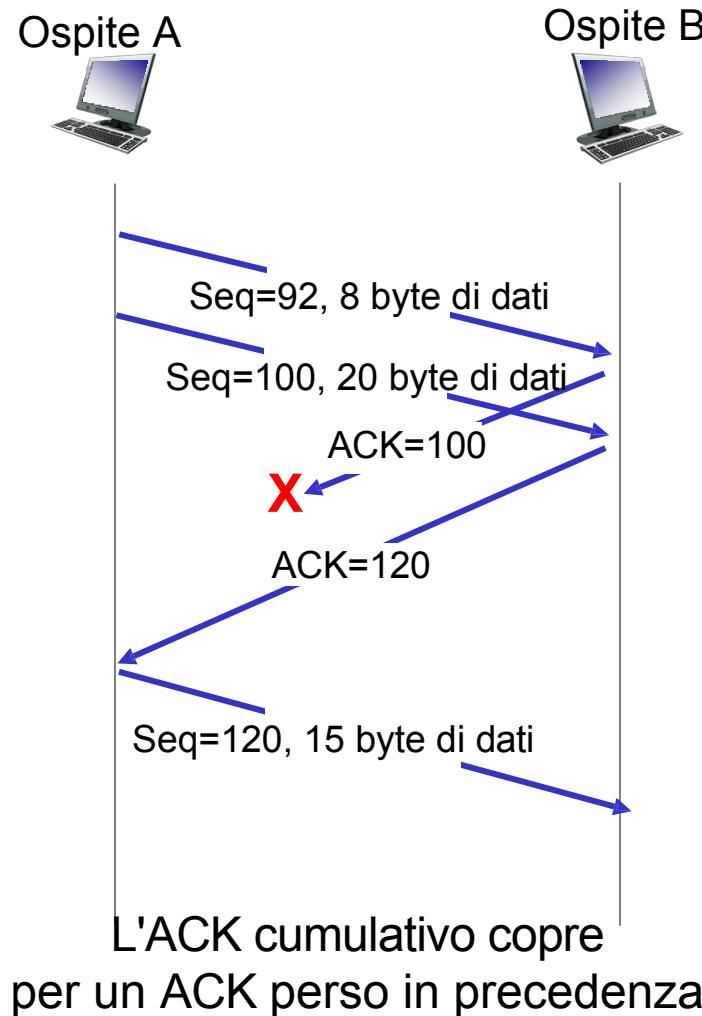


scenario ACK perso



timeout prematuro

TCP: scenari di ritrasmissione



Ritrasmissione rapida TCP

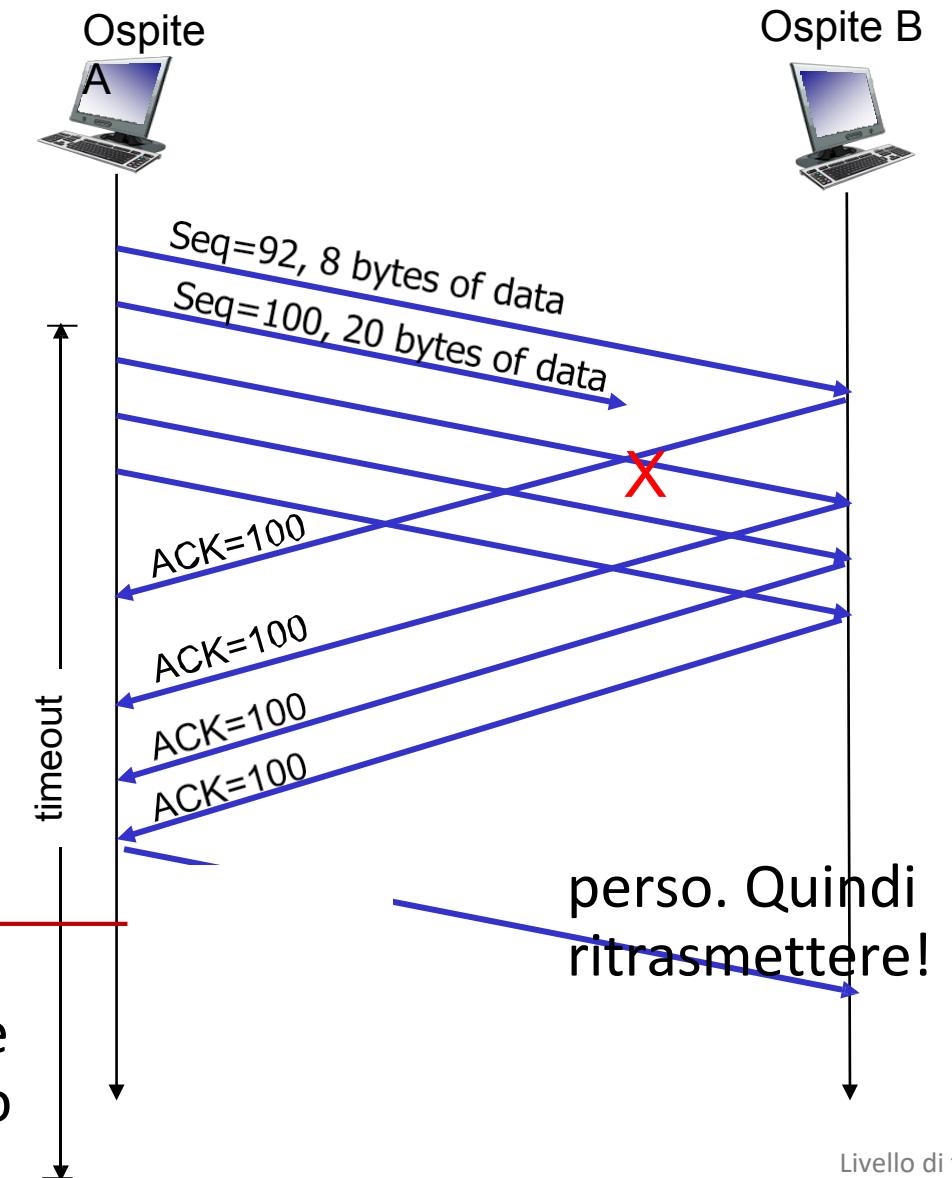
Ritrasmissione rapida TCP

se il mittente riceve 3 ACK aggiuntivi per gli stessi dati ("triplo").
ACK duplicati"), reinvia il segmento non accettato con il seq più piccolo.

- probabilità di perdita del segmento nonACKed,

quindi non aspettare il timeout

 La ricezione di tre ACK duplicati indica 3 segmenti ricevuti dopo un segmento mancante - è probabile che il segmento sia stato perso. Quindi ritrasmettere!



Seq=100, 20 byte di dati

Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- **Trasporto orientato alla connessione: TCP**
 - struttura del segmento
 - trasferimento dati affidabile
 - controllo del flusso
 - gestione delle connessioni
- Principi del controllo della congestione
- Controllo della congestione TCP

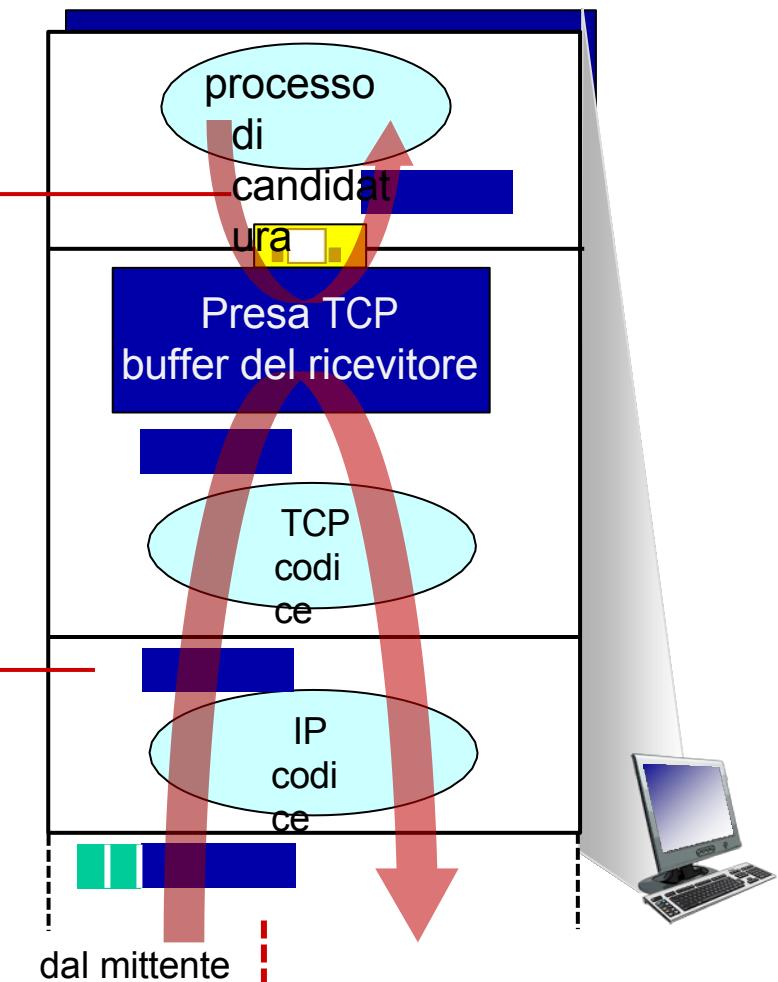


Controllo del flusso TCP

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer dei socket?

Applicazione che rimuove i dati dal socket TCP tamponi

Livello di rete che consegna il payload del datagramma IP nei buffer del socket TCP



stack di protocollo del ricevitore

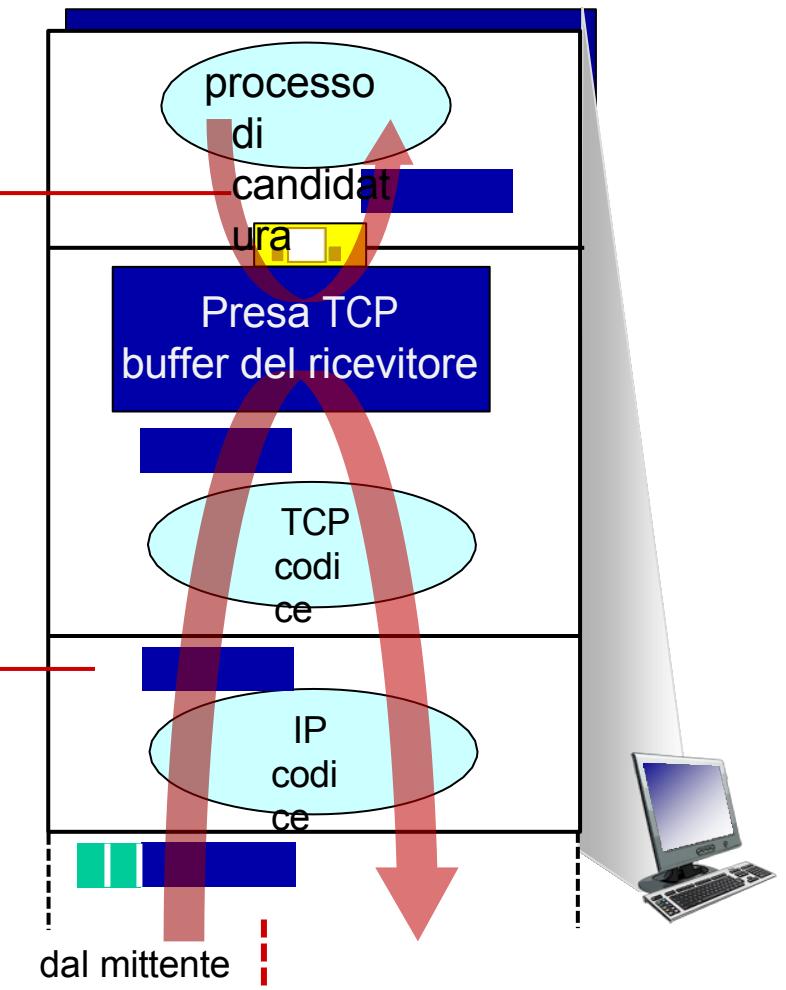
Controllo del flusso TCP

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer dei socket?



Applicazione che
rimuove i dati dal
socket TCP
tamponi

Livello di rete
che consegna il
payload del
datagramma IP nei
buffer del socket TCP



stack di protocollo del ricevitore

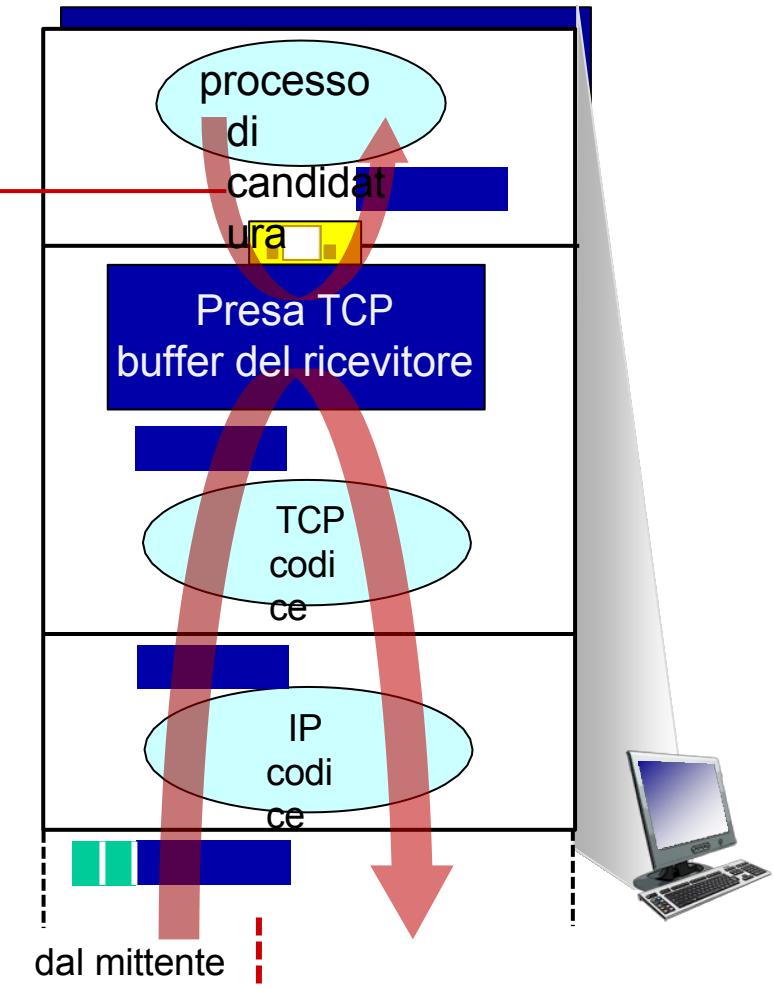
Controllo del flusso TCP

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer dei socket?



controllo del flusso: # byte che il ricevitore è disposto ad accettare

Applicazione che rimuove i dati dal socket TCP tamponi



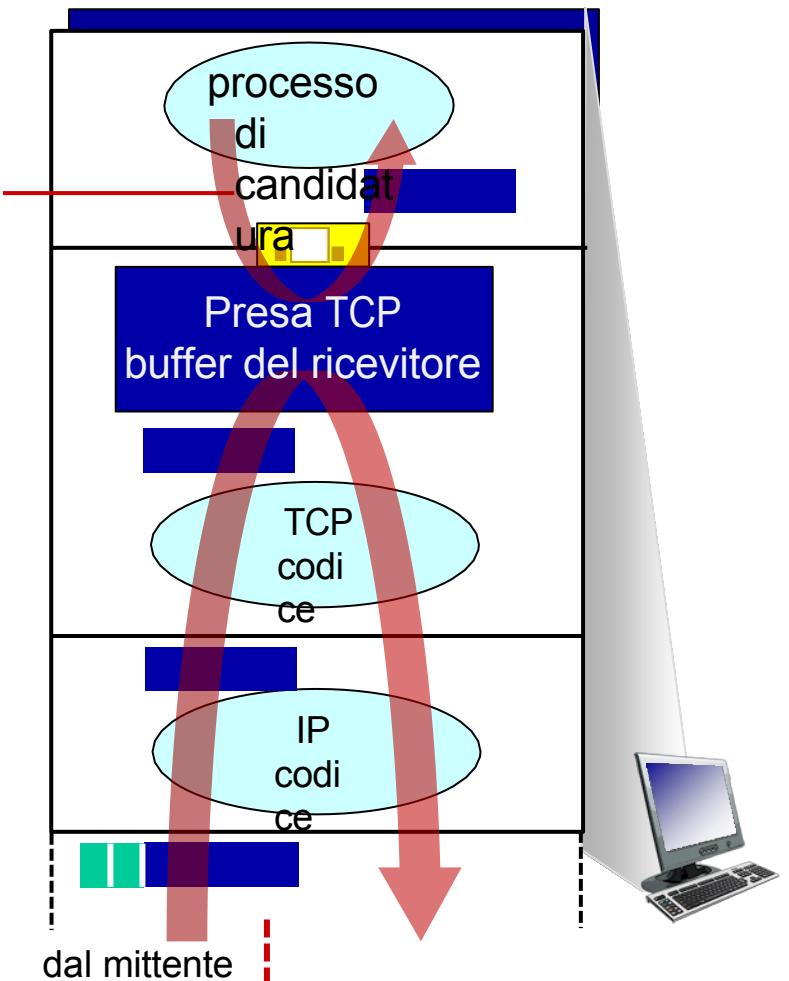
stack di protocollo del ricevitore

Controllo del flusso TCP

D: Cosa succede se il livello di rete fornisce i dati più velocemente di quanto il livello applicativo rimuova i dati dai buffer dei socket?

controllo del flusso
il ricevitore controlla il mittente, quindi il mittente non traboccherà buffer del ricevitore da trasmettere troppo, troppo velocemente

Applicazione che rimuove i dati dal socket TCP tamponi



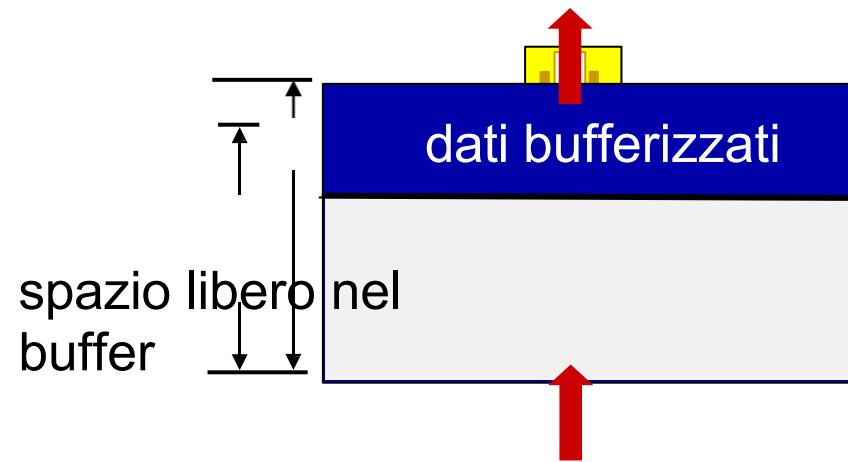
stack di protocollo del ricevitore

Controllo del flusso TCP

- Il ricevitore TCP "pubblicizza" il buffer libero spazio nel campo **rwnd** dell'intestazione TCP
 - Dimensione di **RcvBuffer** impostata tramite le opzioni del socket (l'impostazione predefinita tipica è 4096 byte)
 - molti sistemi operativi si regolano automaticamente **RcvBuffer**
- il mittente limita la quantità di messaggi non accettati ("in volo") ai dati ricevuti

RcvBuffer
rwnd

al processo di
candidatura



- garantisce che il buffer di ricezione non trabocchi

Carichi utili del
segmento TCP

Buffering lato ricevitore TCP

Controllo del flusso TCP

- Il ricevitore TCP "pubblicizza" il buffer libero spazio nel campo **rwnd** dell'intestazione TCP
 - Dimensione di **RcvBuffer** impostata tramite le opzioni del socket (l'impostazione predefinita tipica è 4096 byte)
 - molti sistemi operativi si regolano automaticamente

RcvBuffer

- il mittente limita la quantità di messaggi non accettati ("in volo") ai dati ricevuti
- rwnd**
- garantisce che il buffer di ricezione non trabocchi

controllo del flusso: # byte che il ricevitore è disposto ad accettare

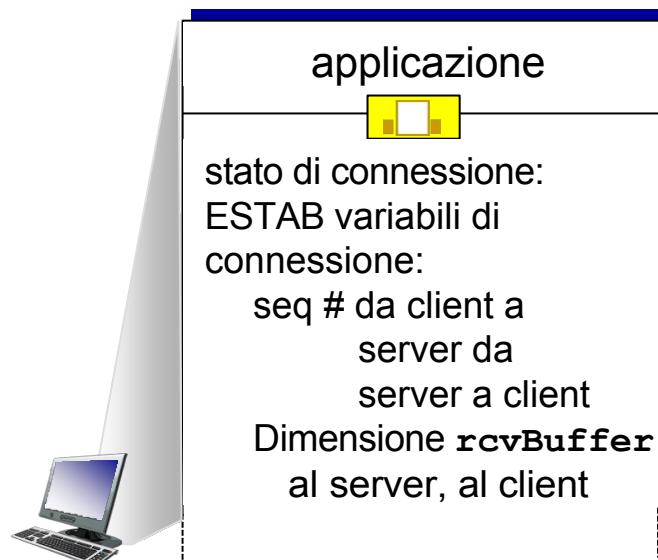


Formato del segmento TCP

Gestione delle connessioni TCP

prima dello scambio di dati, "handshake" tra mittente e destinatario:

- concordano di stabilire una connessione (ognuno sa che l'altro è disposto a stabilire una connessione)
- concordare i parametri di connessione (ad esempio, i numeri di seq iniziali)



`Socket clientSocket =`



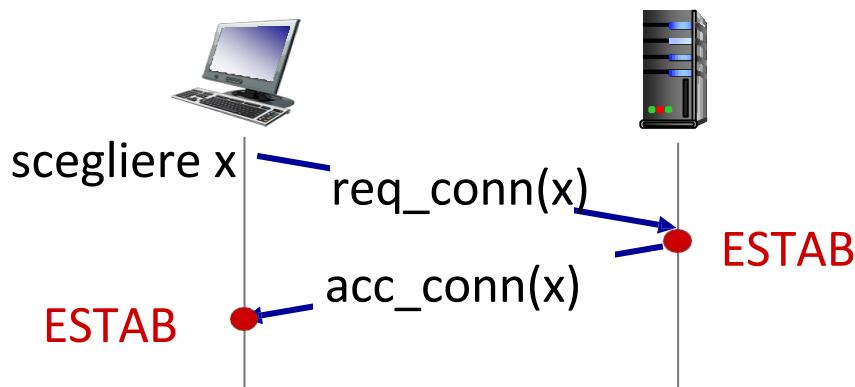
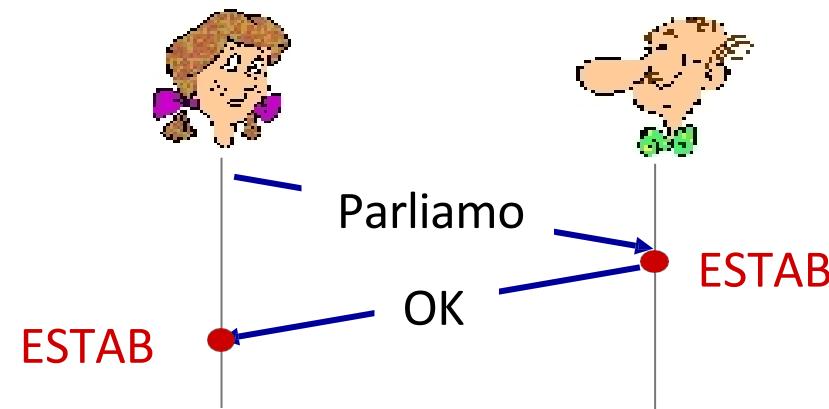
`newSocket ("hostname", "numero di`

```
porta");
```

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

Accettare di stabilire una connessione

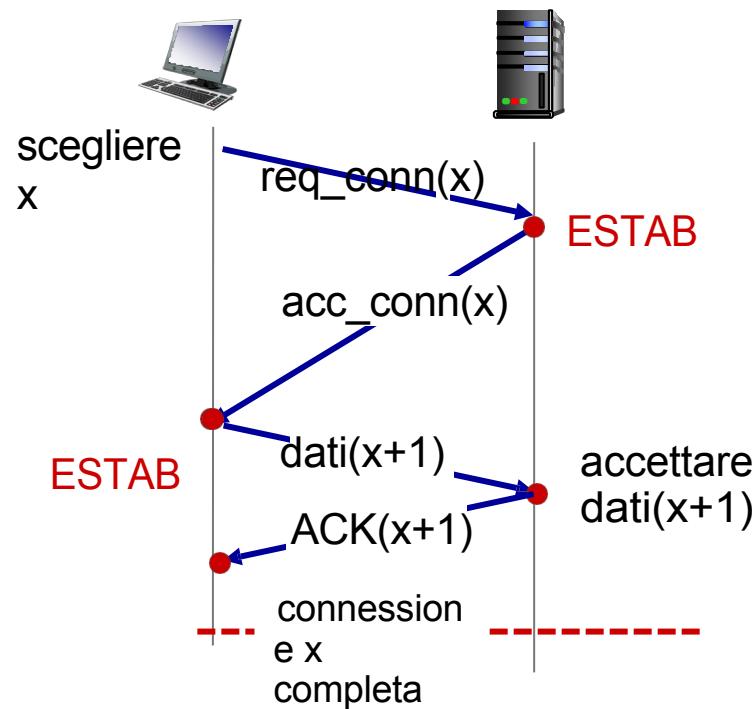
Stretta di mano a 2 vie:



D: L'handshake a 2 vie funziona sempre in rete?

- ritardi variabili
- messaggi ritrasmessi (ad es. $\text{req_conn}(x)$) a causa della perdita di un messaggio
- riordino dei messaggi
- non riesce a "vedere" l'altro lato

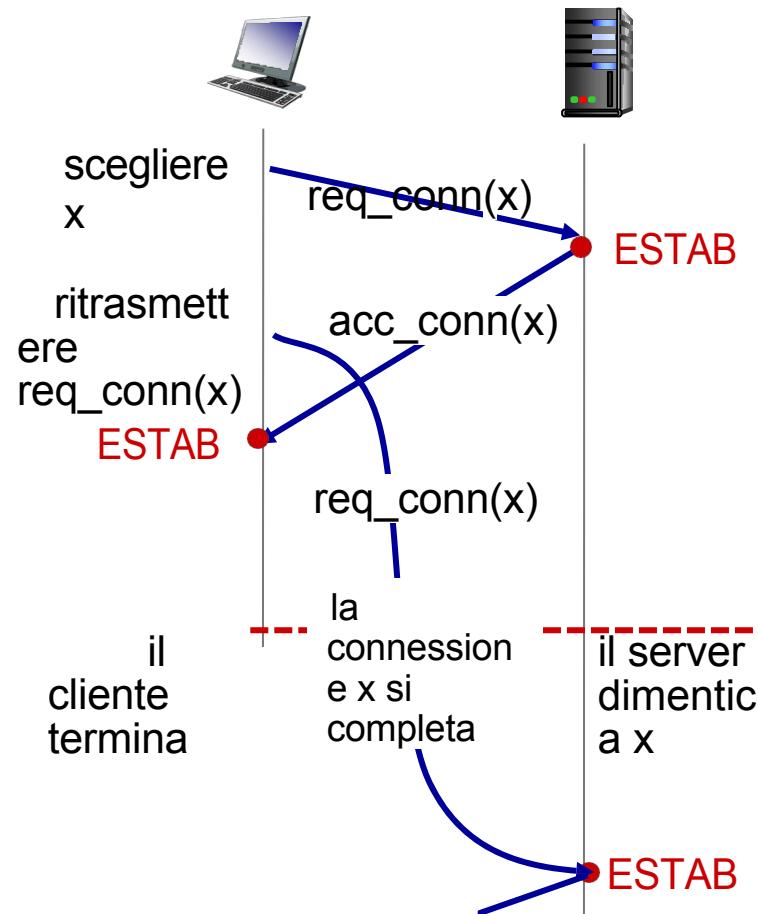
Scenari di handshake a 2 vie



Nessun problema!

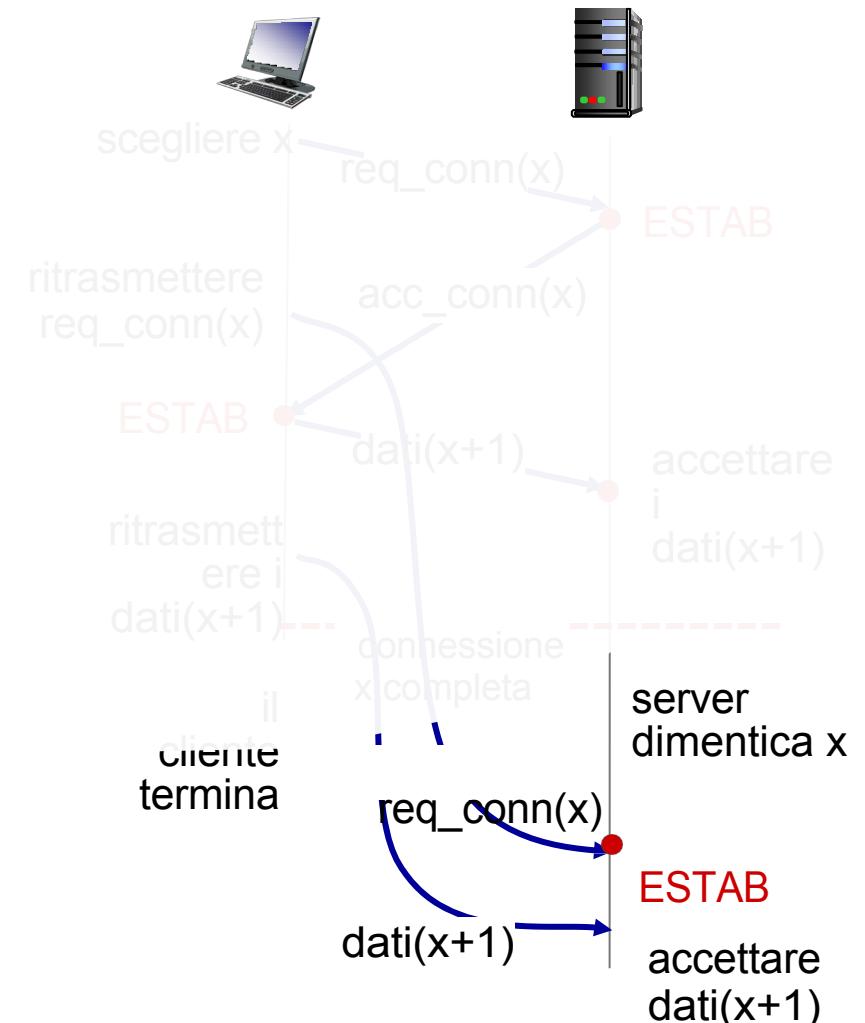


Scenari di handshake a 2 vie



Problema: mezzo
aperto
connessione! (nessun

Scenari di handshake a 2 vie



Problema: dati

duplicati accettati!

TCP 3way handshake

Stato del server

```
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

Stato del cliente

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

ASCOLTA

```
clientSocket.connect((serverName,serverPort))
```

SYNSENT

scegliere init seq num, x
inviare il messaggio TCP SYN



SYNbit=1, Seq=x

ESTAB

ricevuto SYNACK(x)
indica che il server è
attivo; inviare ACK per
SYNACK;

questo segmento può contenere
dati da client a server

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

ACKbit=1, ACKnum=y+1

ASCOLTA

SYN RCVD

scegliere il numero di
seq iniziale, y inviare il
messaggio TCP
SYNACK, accettare il
SYN

ricevuto ACK(y)
indica che il client è
attivo

ESTAB

Chiusura di una connessione TCP

- client e server chiudono ciascuno il proprio lato della connessione
 - inviare il segmento TCP con il bit FIN = 1
- rispondere al FIN ricevuto con un ACK
 - alla ricezione del FIN, l'ACK può essere combinato con il proprio FIN
- è possibile gestire scambi FIN simultanei

Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione**
- Controllo della congestione TCP
- Evoluzione delle funzionalità del livello di trasporto



Principi del controllo della congestione

Congestione:

- informalmente: "troppe fonti che inviano troppi dati troppo velocemente per *rete* da gestire".
- manifestazioni:
 - lunghi ritardi (accodamento nei buffer dei router)
 - perdita di pacchetti (buffer overflow nei router)
- diverso dal controllo di flusso!
- un problema da top 10!



controllo della congestione:

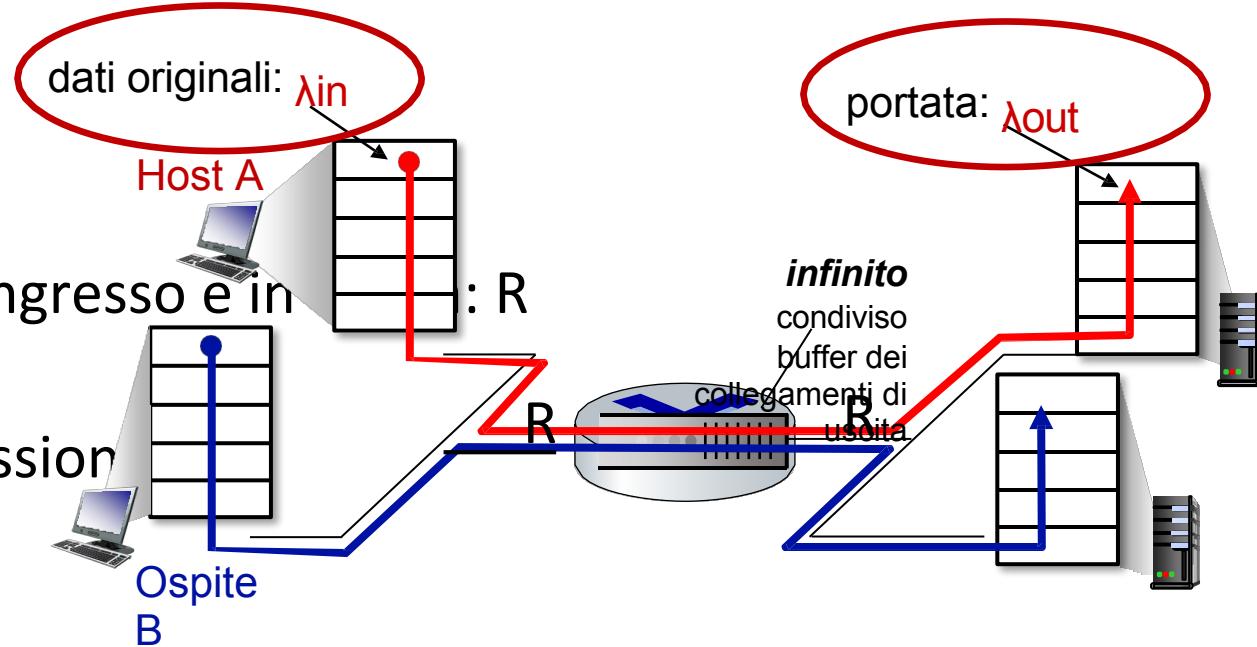
troppi mittenti,
invio troppo
veloce

controllo di flusso: un
mittente
troppo veloce per un solo ricevitore

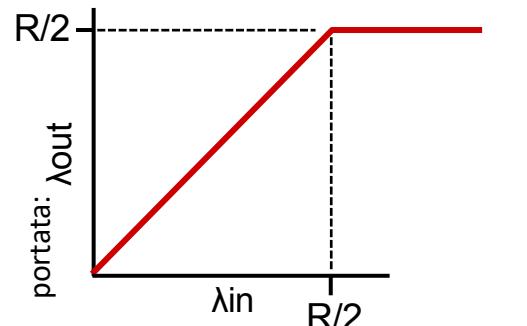
Cause/costi della congestione: scenario 1

Lo scenario più semplice:

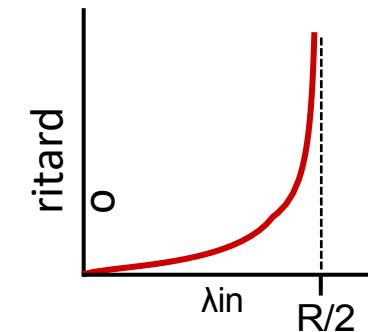
- un router, buffer infiniti
- capacità di collegamento in ingresso e in uscita infinita
- due flussi
- non sono necessarie ritrasmissioni



D: Cosa succede quando il tasso di arrivo λ_{in} si avvicina a $R/2$?

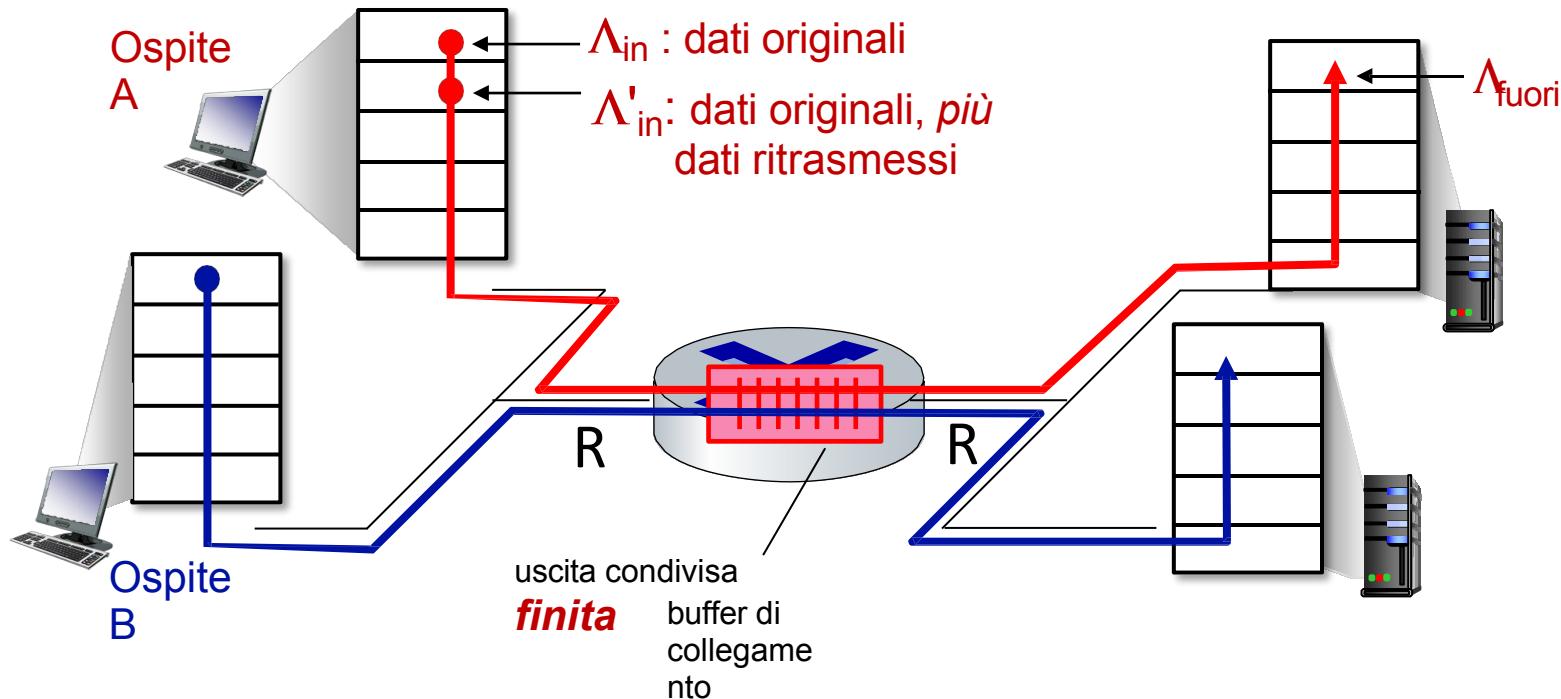


massimo per connessione
velocità di trasmissione:



Cause/costi della congestione: scenario 2

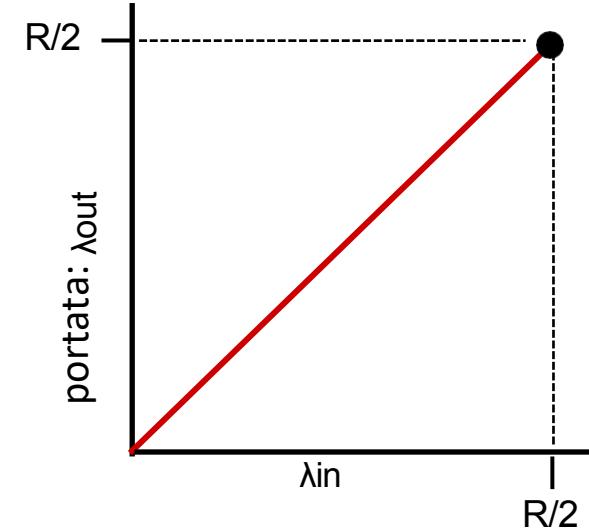
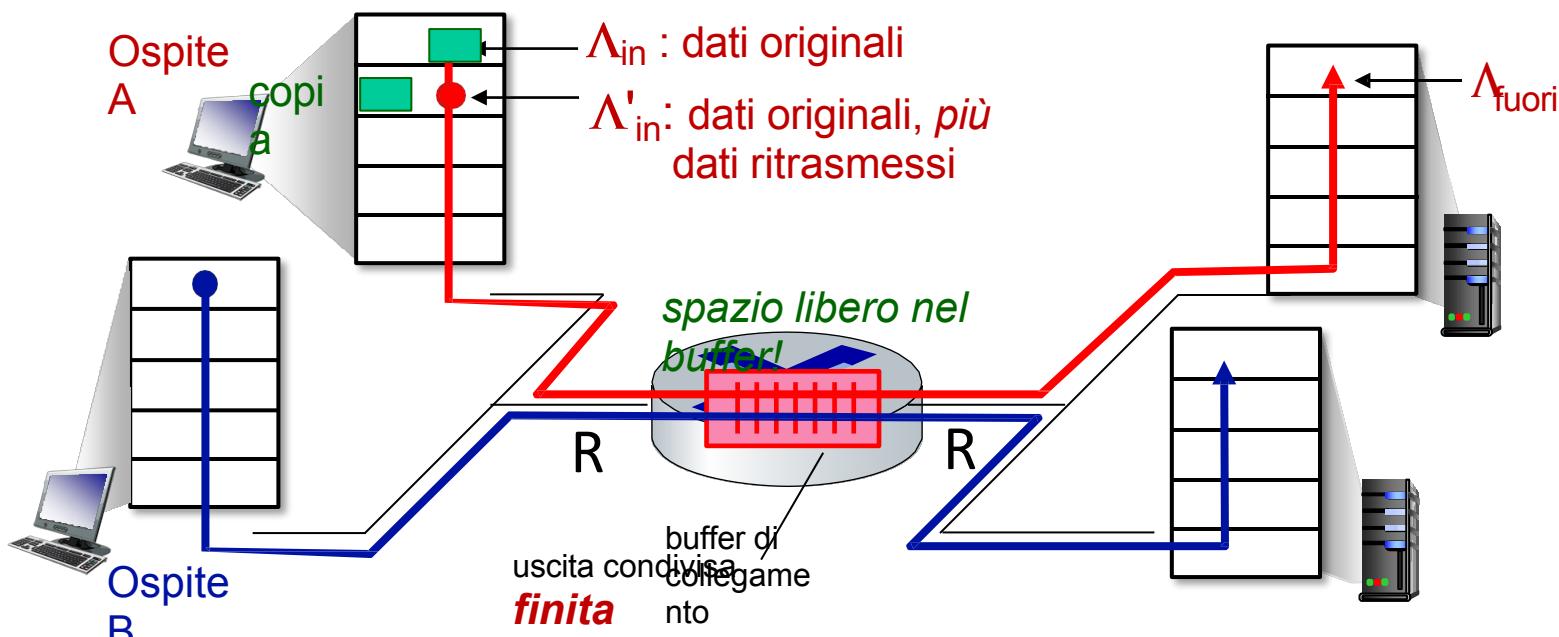
- un router, buffer *finito*
- il mittente ritrasmette il pacchetto perso e fuori tempo
 - input dello strato applicativo = output dello strato applicativo: $\Lambda_{in} = \Lambda_{out}$
 - l'input del livello di trasporto include le *ritrasmissioni*: $\Lambda'_{in} \geq \Lambda_{in}$



Cause/costi della congestione: scenario 2

Idealizzazione: conoscenza perfetta

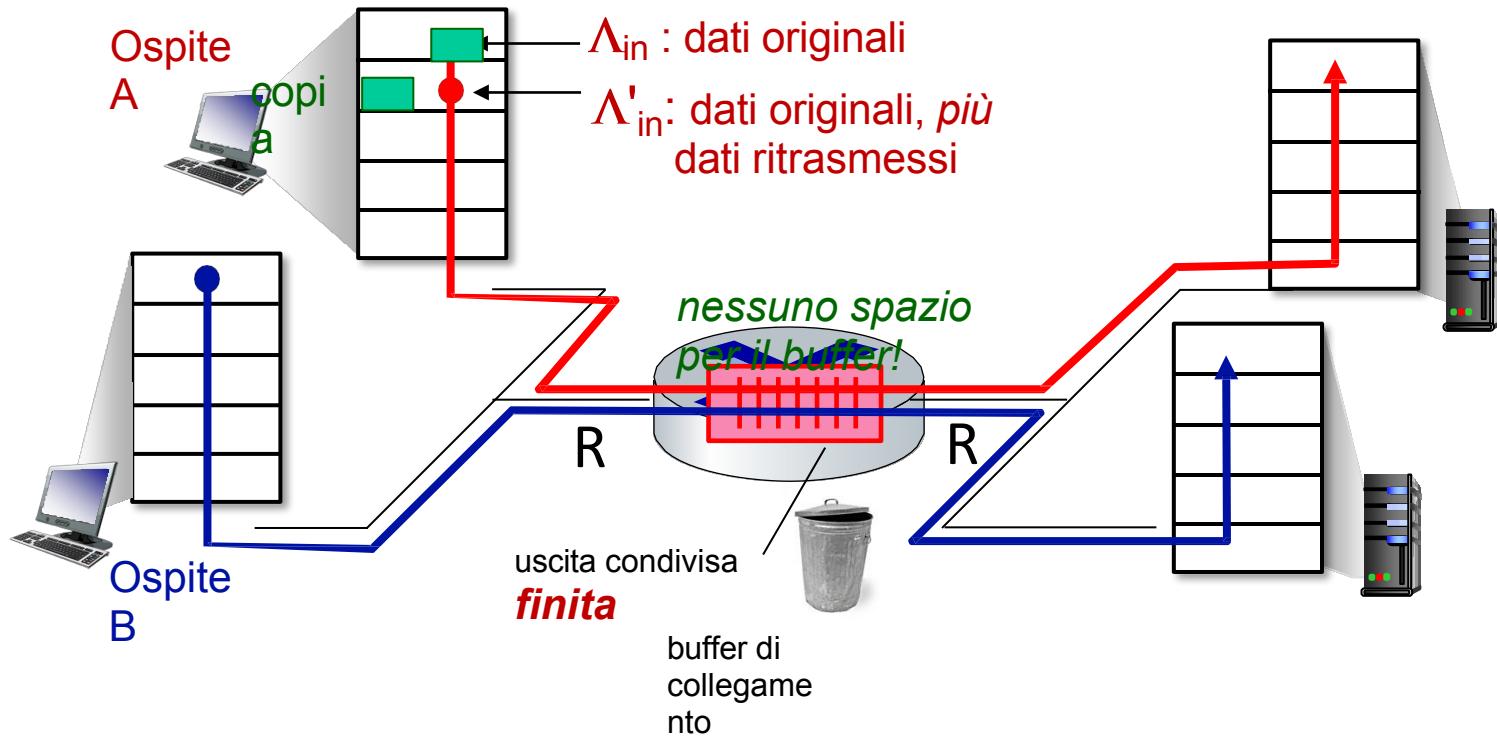
- il mittente invia solo quando i buffer del router sono disponibili



Cause/costi della congestione: scenario 2

Idealizzazione: *una* conoscenza perfetta

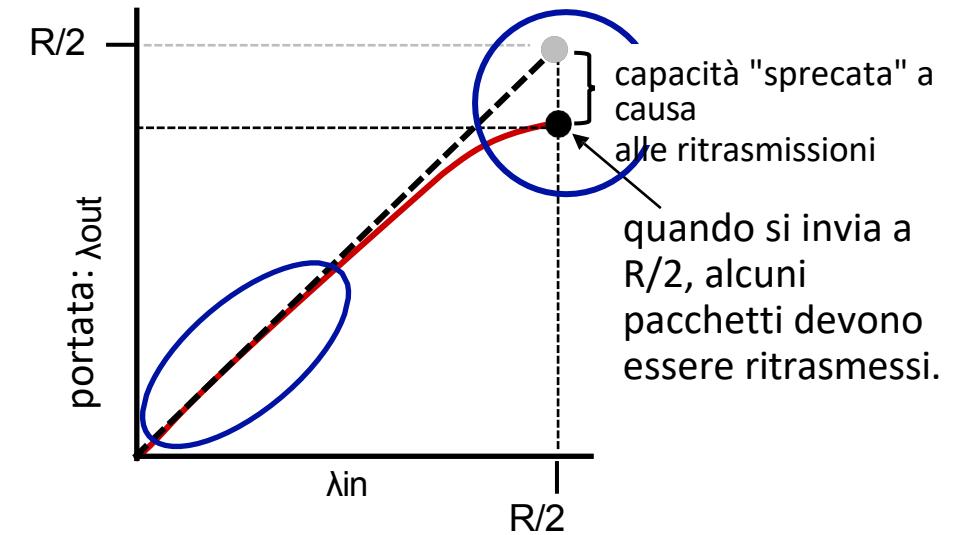
- I pacchetti possono essere persi (abbandonati dal router) a causa di buffer pieni
- il mittente sa quando il pacchetto è stato abbandonato: reinvia solo se il pacchetto è *noto* come perso

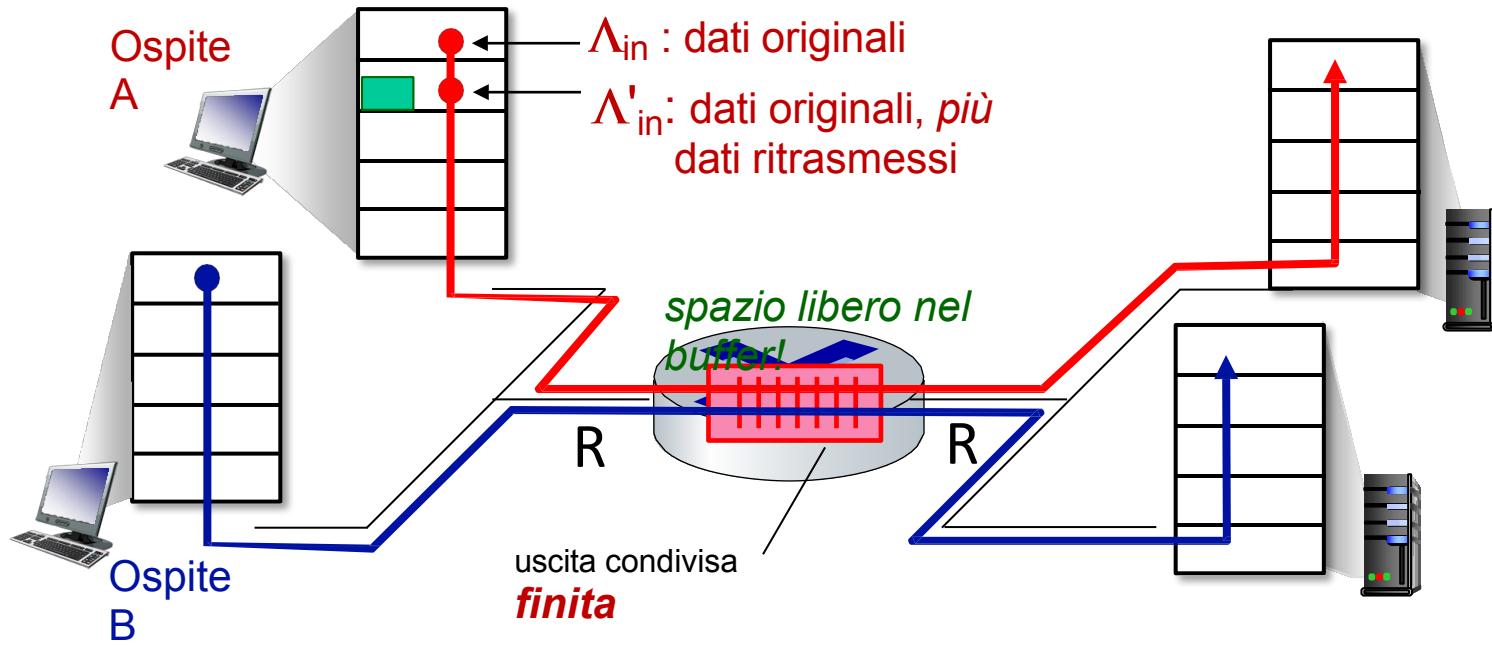


Cause/costi della congestione: scenario 2

Idealizzazione: *una conoscenza perfetta*

- I pacchetti possono essere persi (abbandonati dal router) a causa di buffer pieni
- il mittente sa quando il pacchetto è stato abbandonato: reinvia solo se il pacchetto è *noto* come perso



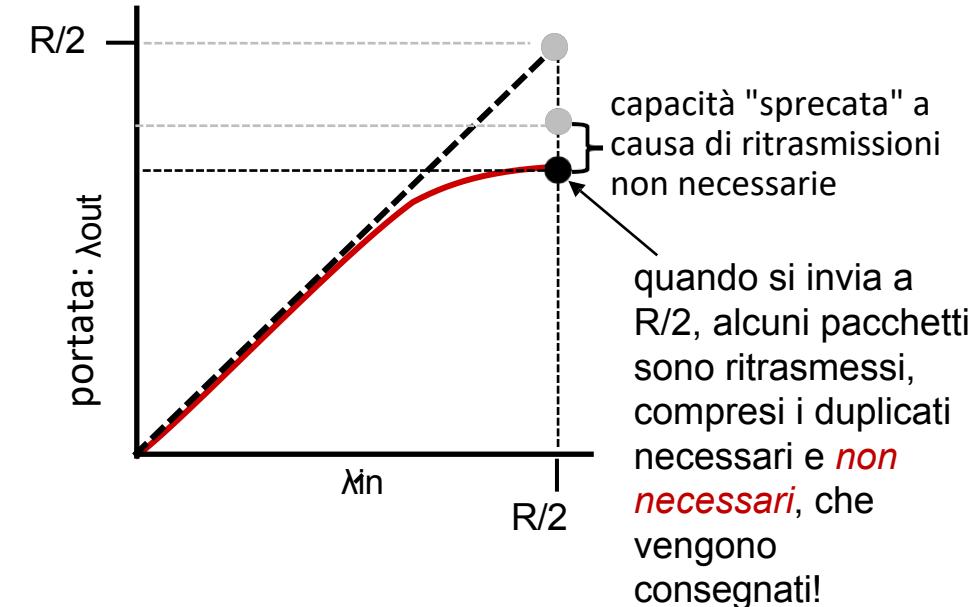
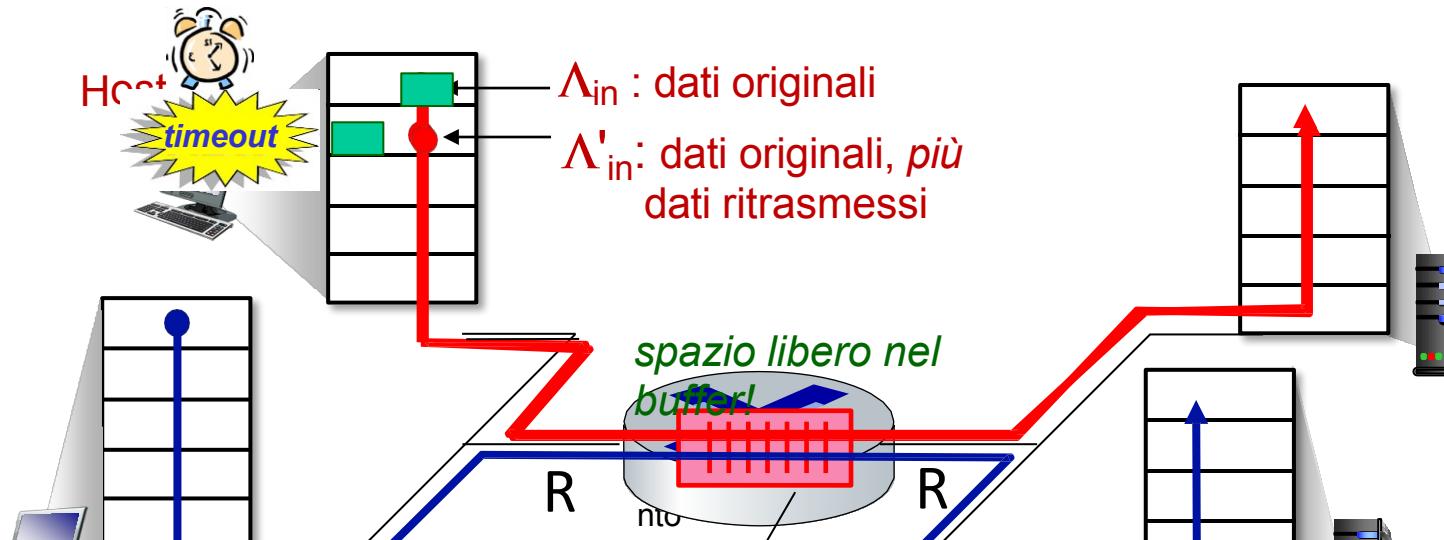


buffer di collegamento

Cause/costi della congestione: scenario 2

Scenario realistico: *duplicati non necessari*

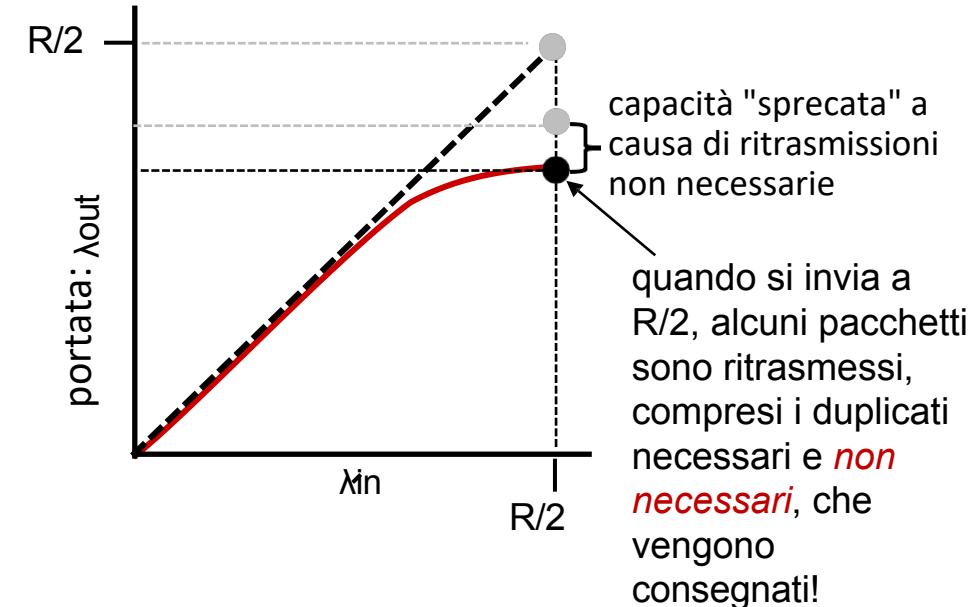
- I pacchetti possono essere persi, abbandonati dal router a causa dei buffer pieni, richiedendo ritrasmissioni.
- ma i tempi del mittente possono scadere prematuramente, invio di *due* copie, *entrambe* consegnate



Cause/costi della congestione: scenario 2

Scenario realistico: *duplicati non necessari*

- I pacchetti possono essere persi, abbandonati dal router a causa dei buffer pieni, richiedendo ritrasmissioni.
- ma i tempi del mittente possono scadere prematuramente, invio di *due* copie, *entrambe* consegnate



"costi" della congestione:

- più lavoro (ritrasmissione) per un dato throughput del ricevitore
- ritrasmissioni non necessarie: il collegamento trasporta più copie di un pacchetto

- diminuzione del throughput massimo raggiungibile

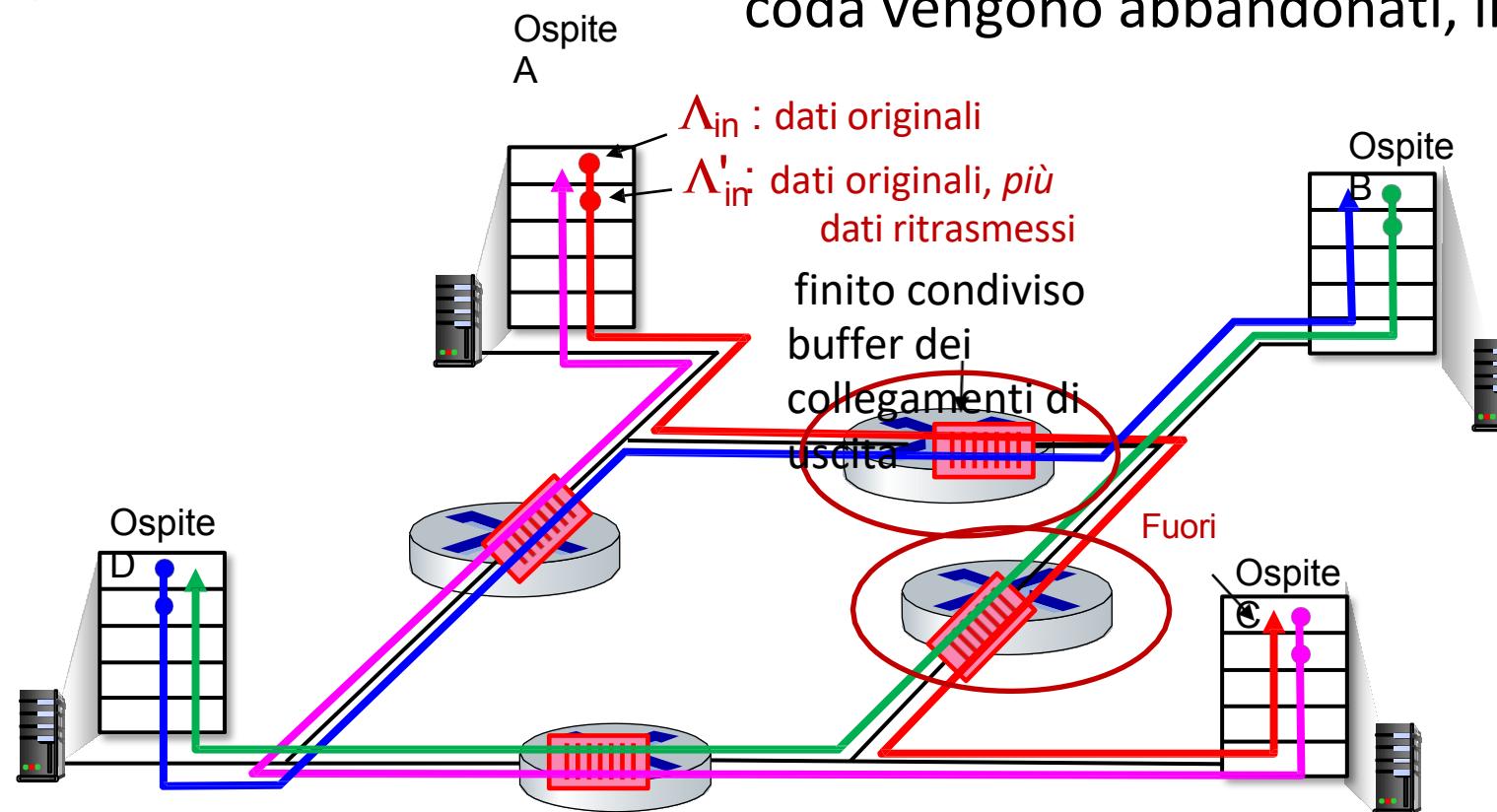
Cause/costi della congestione: scenario 3

- quattro mittenti
- percorsi *multi-hop*
- timeout/ritrasmissioni

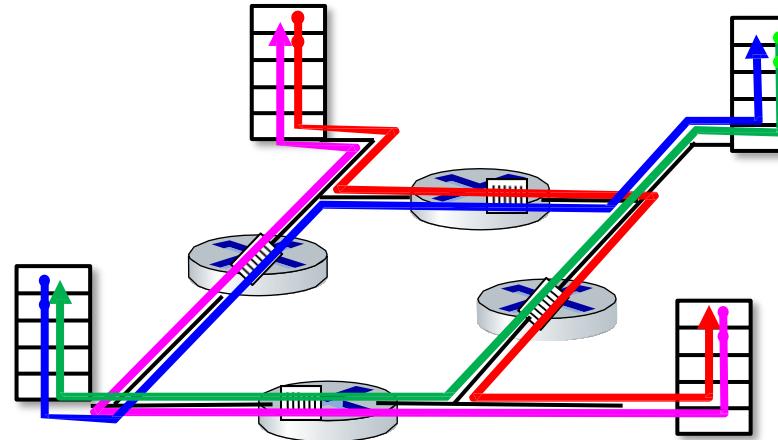
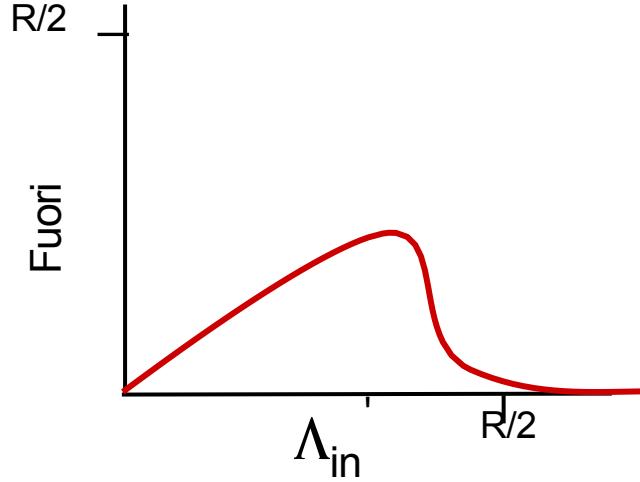
D: Cosa succede all'aumentare di Λ_{in} e Λ'_{in} ?

A: all'aumentare del Λ rosso_{in'}, tutti i pk blu in arrivo nella parte superiore

coda vengono abbandonati, il throughput blu $\rightarrow 0$



Cause/costi della congestione: scenario 3

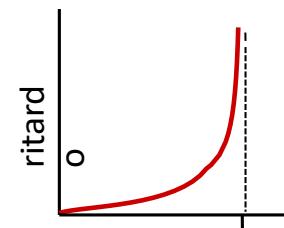
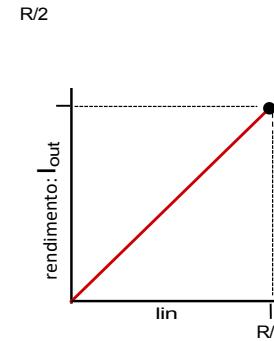


un altro "costo" della congestione:

- Quando il pacchetto cade, la capacità di trasmissione a monte e il buffering utilizzati per quel pacchetto vengono sprecati!

Cause/costi della congestione: approfondimenti

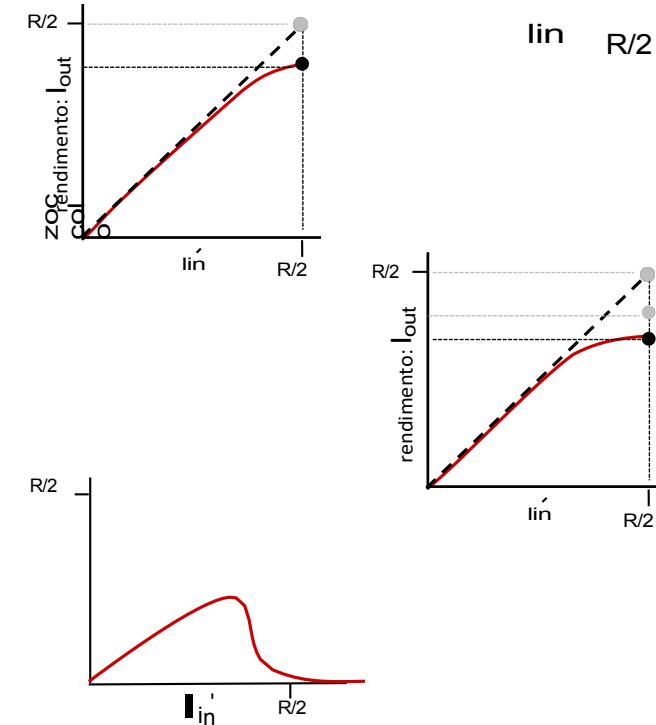
- Il flusso di lavoro non può mai superare la capacità
- il ritardo aumenta con l'avvicinarsi della capacità
- la perdita/ritrasmissione diminuisce il throughput effettivo
- i duplicati non necessari diminuiscono



u i n l o t e m r e i r o u t e h i r p

ut effettivo

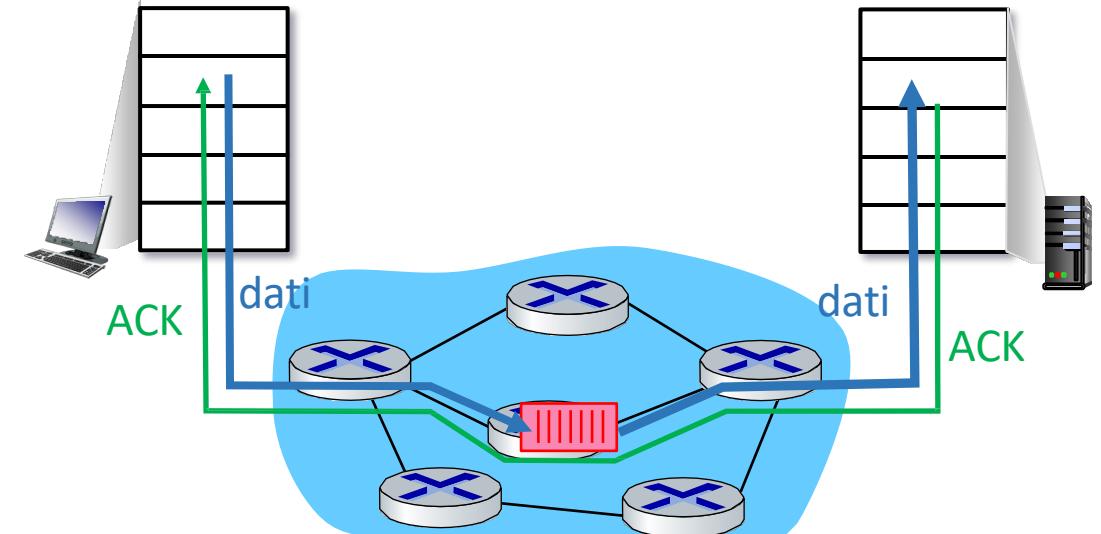
- capacità di trasmissione a monte / buffering sprecato per i pacchetti persi a valle



Approcci al controllo della congestione

Controllo della congestione all'estremità:

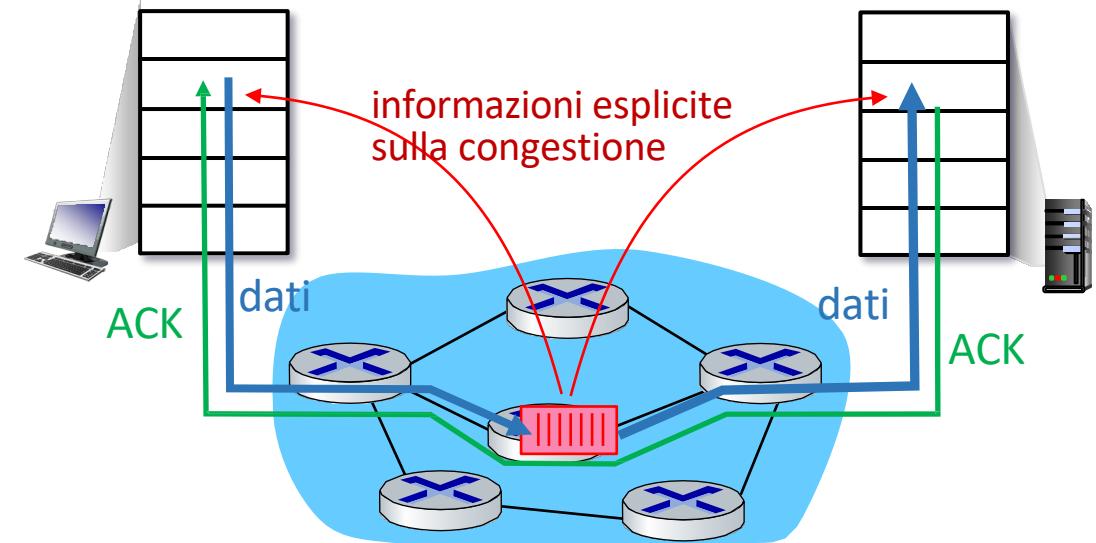
- nessun feedback esplicito dalla rete
- congestione *desunta* da perdite e ritardi osservati
- approccio adottato da TCP



Approcci al controllo della congestione

Controllo della congestione assistito dalla rete:

- I router forniscono un feedback *diretto* agli host che inviano/ricevono i flussi che passano attraverso i router congestionati.
- può indicare il livello di congestione o la velocità di invio impostata esplicitamente



- TCP ECN, ATM, protocolli DECbit

Capitolo 3: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- **Controllo della congestione TCP**
- Evoluzione delle funzionalità del livello di trasporto



Controllo della congestione TCP: AIMD

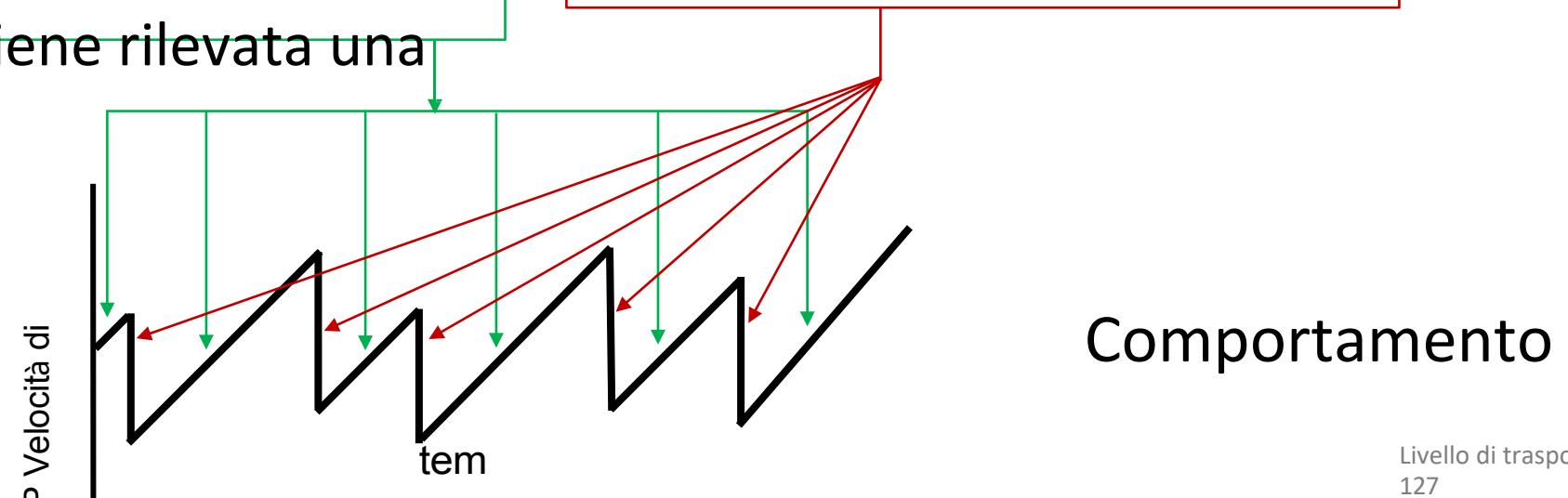
- *approccio:* i mittenti possono aumentare la velocità di invio fino a quando non si verifica una perdita di pacchetti (congestione), quindi diminuire la velocità di invio in caso di perdita

Aumento dell'additivo

aumentare la velocità di invio di 1 dimensione massima del segmento ogni RTT fino a quando non viene rilevata una perdita

Diminuzione moltiplicativa

dimezzare la velocità di invio ad ogni evento di perdita



a dente di sega
AIMD:
sondaggio della
larghezza di
banda

TCP AIMD: di più

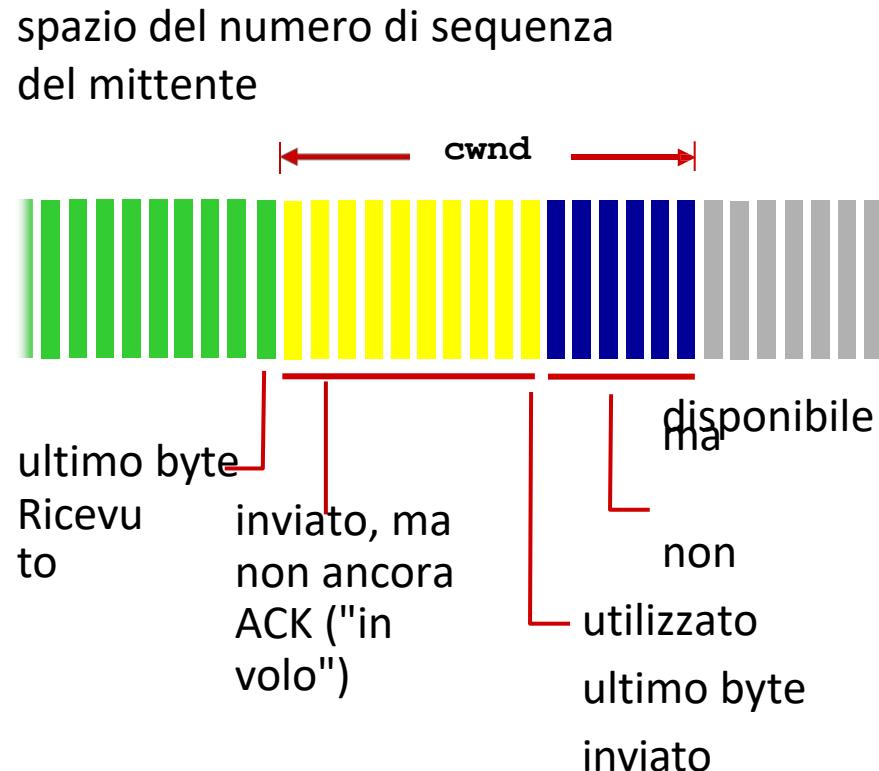
Dettaglio *diminuzione moltiplicativa*: la velocità di invio è

- Dimezzamento della perdita rilevata da un triplo ACK duplicato (TCP Reno)
- Taglio a 1 MSS (dimensione massima del segmento) in caso di perdita rilevata dal timeout (TCP Tahoe)

Perché AIMD?

- È stato dimostrato che l'AIMD - un algoritmo distribuito e asincrono - è in grado di:
 - ottimizzare le portate congestionate a livello di rete!
 - hanno proprietà di stabilità desiderabili

Controllo della congestione TCP: dettagli



Comportamento di invio TCP:

- *all'incirca*: inviare byte di cwnd, attendere RTT per ACKS, quindi inviare altri

$$\text{Velocità TCP} \frac{\text{cwnd}}{\tilde{\text{RTT}}} \text{ bytes/sec}$$

- Il mittente TCP limita la trasmissione: $\text{UltimoByteSent} - \text{UltimoByteAcked} \leq \text{cwnd}$

- cwnd viene regolato dinamicamente in risposta alla congestione di rete osservata (implementando il controllo della congestione TCP)

Avvio lento del TCP

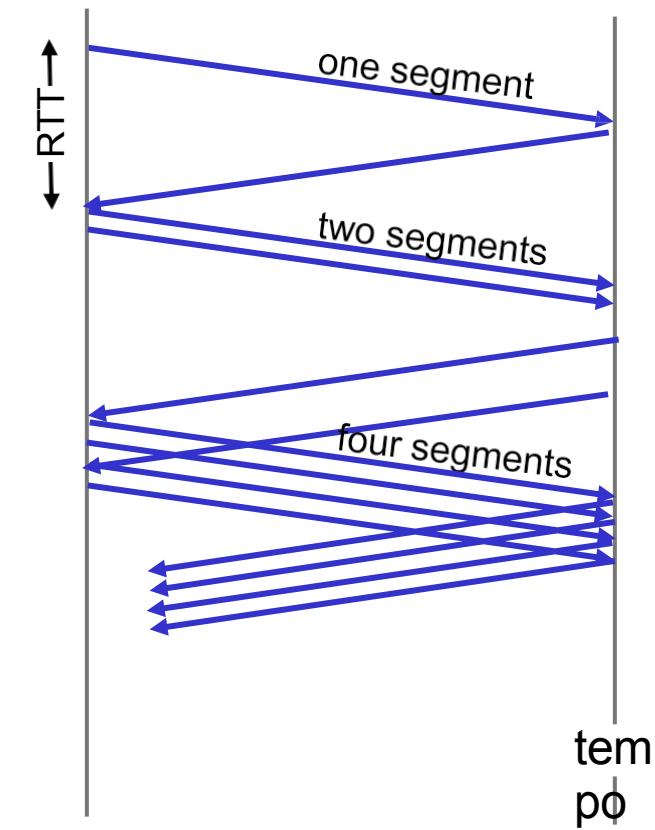
- quando inizia la connessione, aumentare la velocità in modo esponenziale fino al primo evento di perdita:
 - inizialmente **cwnd** = 1 MSS
 - doppio **cwnd** ogni RTT
 - incrementando **cwnd** per ogni ACK ricevuto

- *sintesi:* il tasso iniziale è lento, ma aumenta in modo esponenziale e veloce

Ospite A



Ospite B



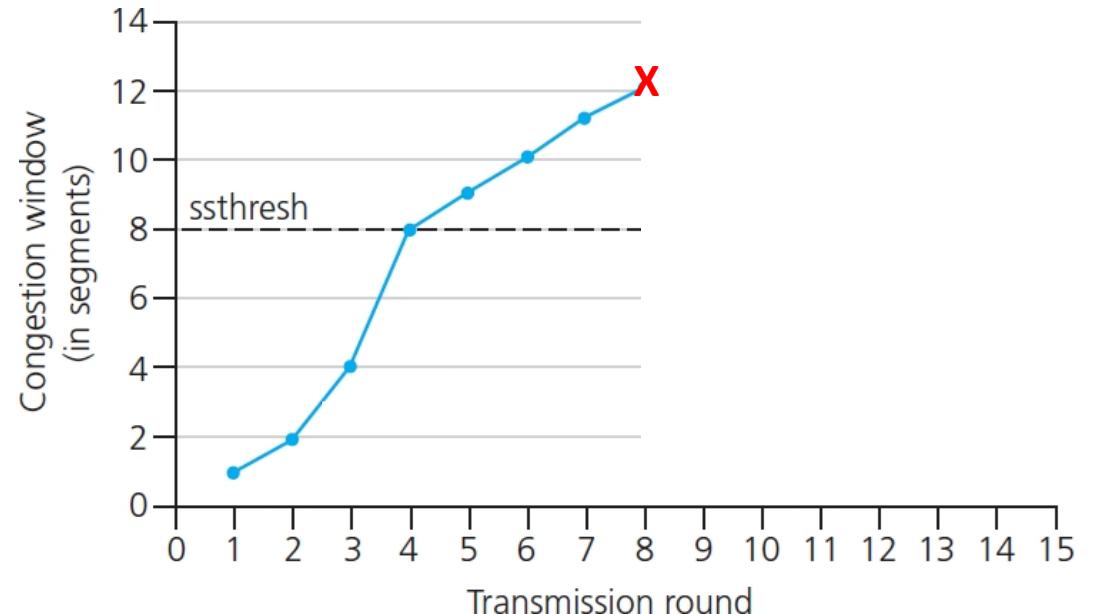
TCP: dall'avvio lento all'evitamento della congestione

D: quando l'aumento esponenziale dovrebbe passare a quello lineare?

R: quando **cwnd** raggiunge $1/2$ del suo valore prima del timeout.

Implementazione:

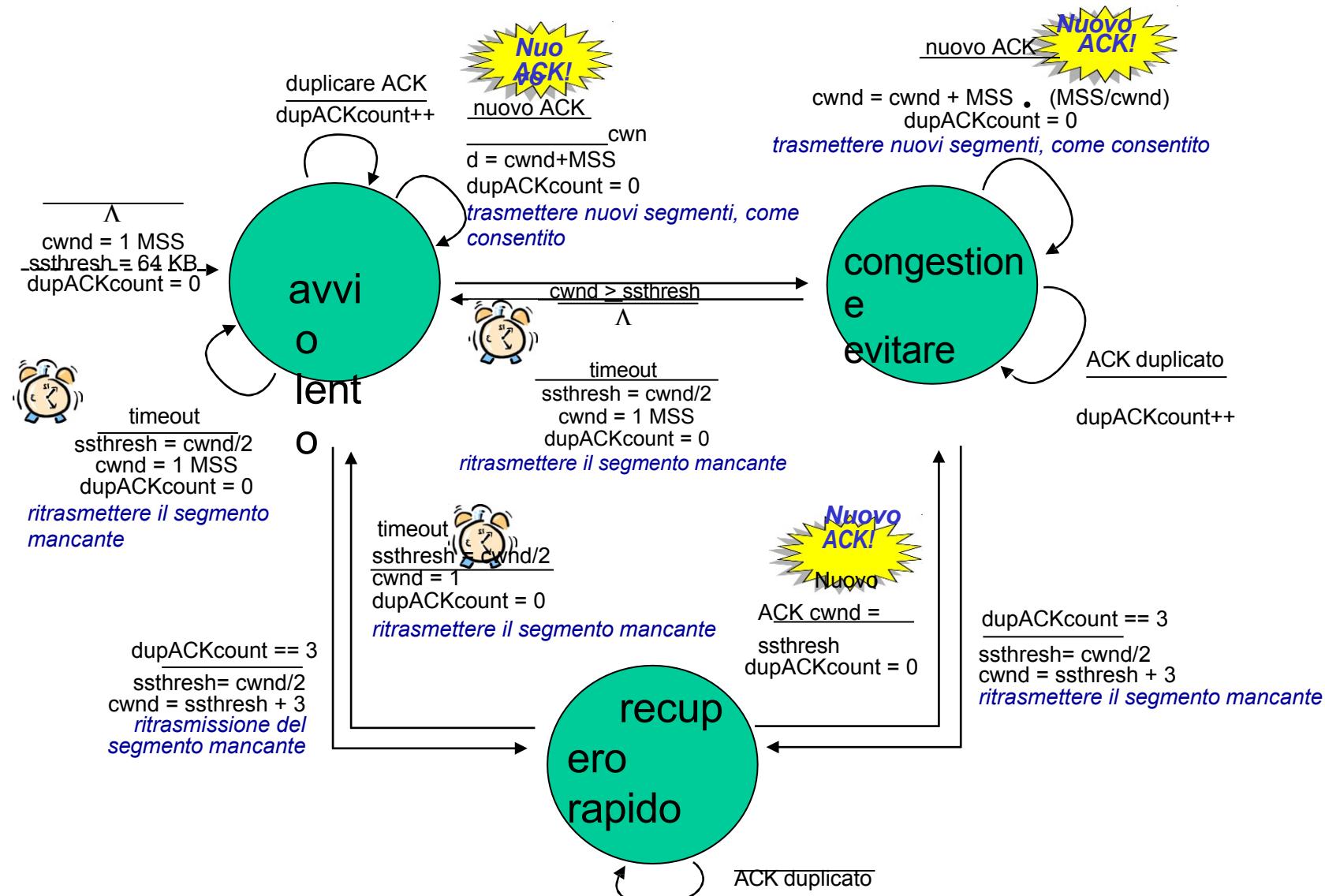
- variabile **ssthresh**
- su evento di perdita, **ssthresh**



viene impostato a 1/2 di **cwnd**
appena prima dell'evento di
perdita

* Per ulteriori esempi, consultate gli esercizi interattivi online: http://gaia.cs.umass.edu/kurose_ross/interactive/.

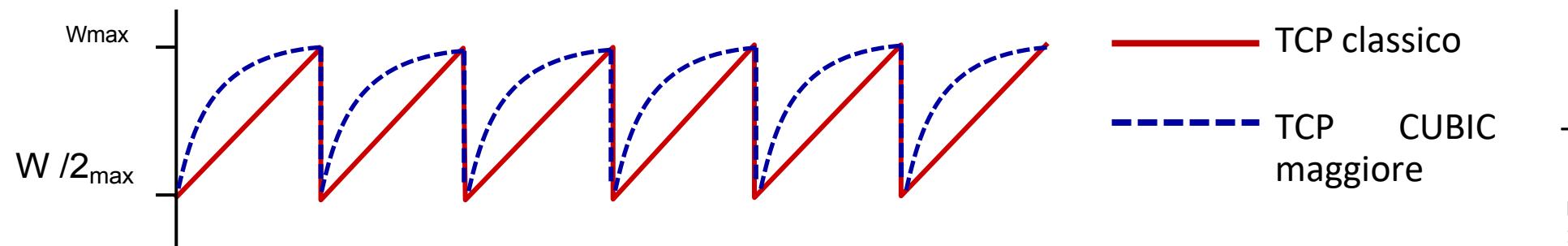
Sommario: Controllo della congestione TCP



$cwnd = cwnd + MSS$
trasmettere nuovi segmenti, come consentito

CUBO TCP

- Esiste un modo migliore di AIMD per "sondare" la larghezza di banda utilizzabile?
- Intuizione/intuizione:
 - w_{max} : velocità di invio alla quale è stata rilevata la perdita di congestione
 - lo stato di congestione del collegamento a collo di bottiglia probabilmente (?) non è cambiato molto
 - dopo aver dimezzato la velocità/finestra in caso di perdita, all'inizio la rampa di salita fino a w_{max} è *più veloce*, ma poi si avvicinano più *lentamente* a w_{max}



mpio

t
h
r
o
u
g
h
p
u
t

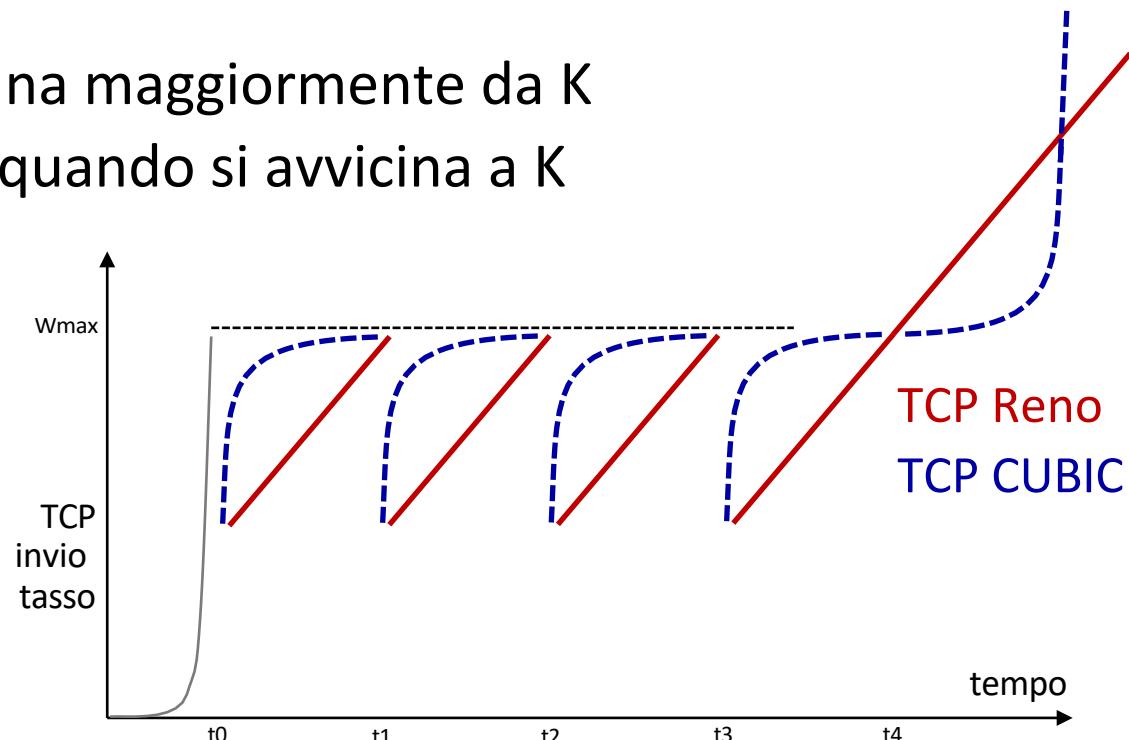
i
n

q
u
e
s
t
o

e
s
e

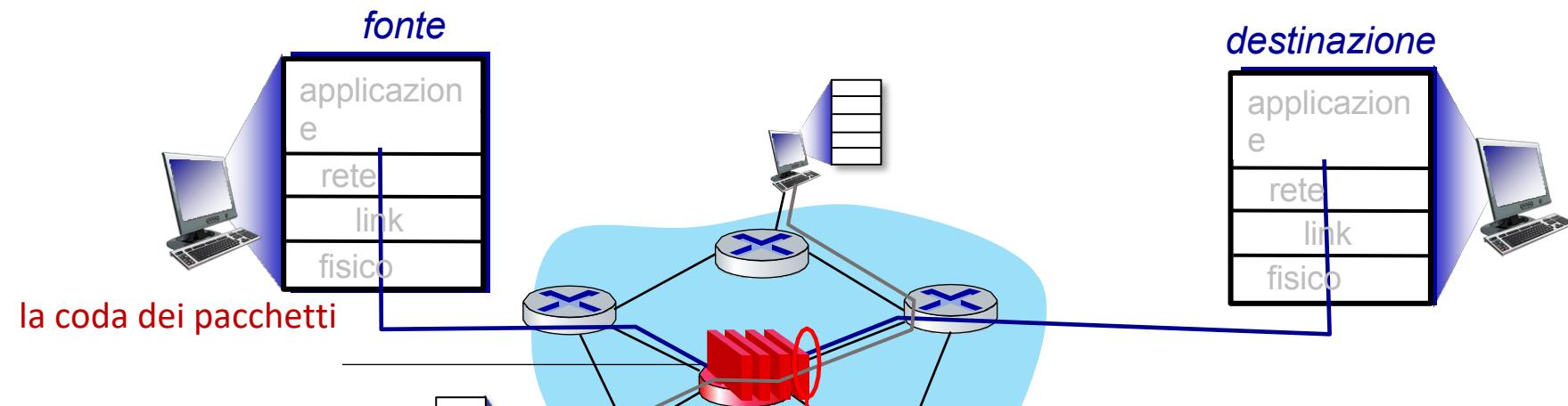
CUBO TCP

- K: momento in cui la dimensione della finestra TCP raggiungerà w_{max}
 - K è di per sé sintonizzabile
- aumentare W in funzione del *cubo* della distanza tra il tempo corrente e K
 - aumenta quando ci si allontana maggiormente da K
 - più piccolo aumenta (cauto) quando si avvicina a K
- TCP CUBIC predefinito
in Linux, il TCP più diffuso per i server Web più popolari



TCP e il "collegamento a collo di bottiglia" congestionato

- TCP (classico, CUBIC) aumenta la velocità di invio di TCP fino a quando non si verifica una perdita di pacchetti all'uscita di un router: il *collegamento del collo di bottiglia*.

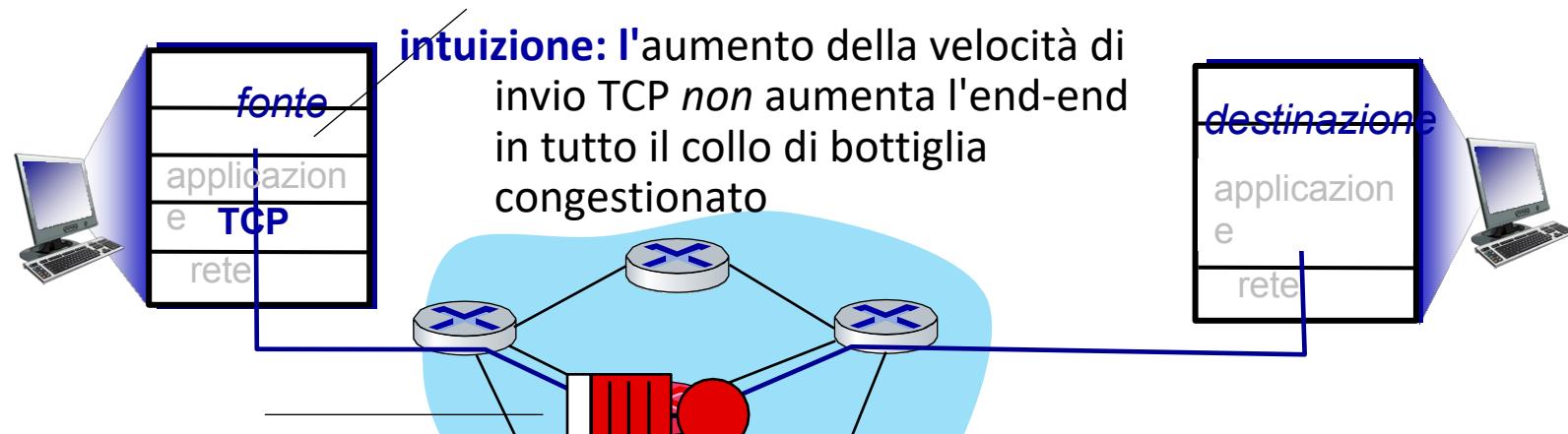


non è quasi mai vuota, a
volte si verifica
un'eccedenza di
pacchetti (perdita)

collegamento a collo di bottiglia (quasi sempre occupato)

TCP e il "collegamento a collo di bottiglia" congestionato

- TCP (classico, CUBIC) aumenta la velocità di invio di TCP fino a quando non si verifica una perdita di pacchetti all'uscita di un router: il *collegamento del collo di bottiglia*.
- capire la congestione: è utile concentrarsi sul collegamento con il collo di bottiglia congestionato



link
fisico

intuizione: l'aumento
della velocità di invio
TCP aumenterà l'RTT
misurato

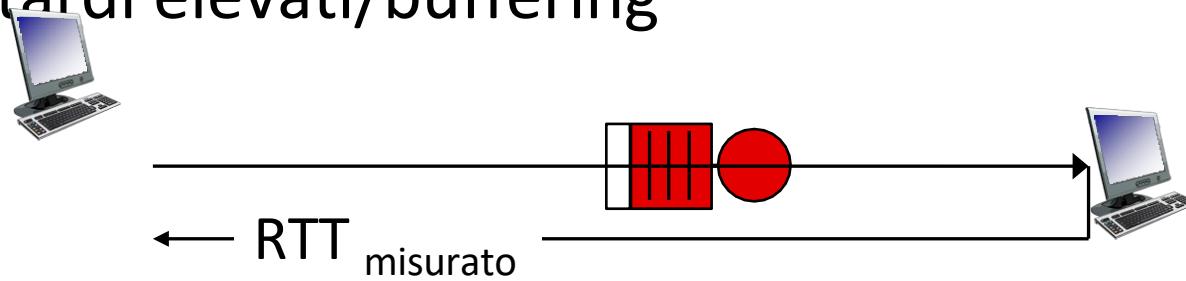
link
fisico

Obiettivo: "*mantenere il tubo finale appena pieno, ma*

← RTT → ~~non più pieno~~.

Controllo della congestione TCP basato sul ritardo

Mantenere il tubo da mittente a destinatario "abbastanza pieno, ma non di più": keep collegamento a collo di bottiglia impegnato nella trasmissione, ma evitare ritardi elevati/buffering



$$\text{rendimento misurato} = \frac{\text{\# byte inviati nell'ultimo intervallo RTT}}{\text{RTT misurato}}$$

Approccio basato sul ritardo:

- RTT_{\min} - RTT minimo osservato (percorso non congestionato)
- il throughput non congestionato con la finestra di congestione $cwnd$ è $cwnd/\text{RTT}_{\min}$

se il throughput misurato è "molto vicino" al throughput non congestionato
aumentare cwnd linearmente/* poiché il percorso non è congestionato */
altrimenti, se il throughput misurato è "molto inferiore" a quello di un'intera area non congestionata
diminuire linearmente cwnd/* poiché il percorso è congestionato */

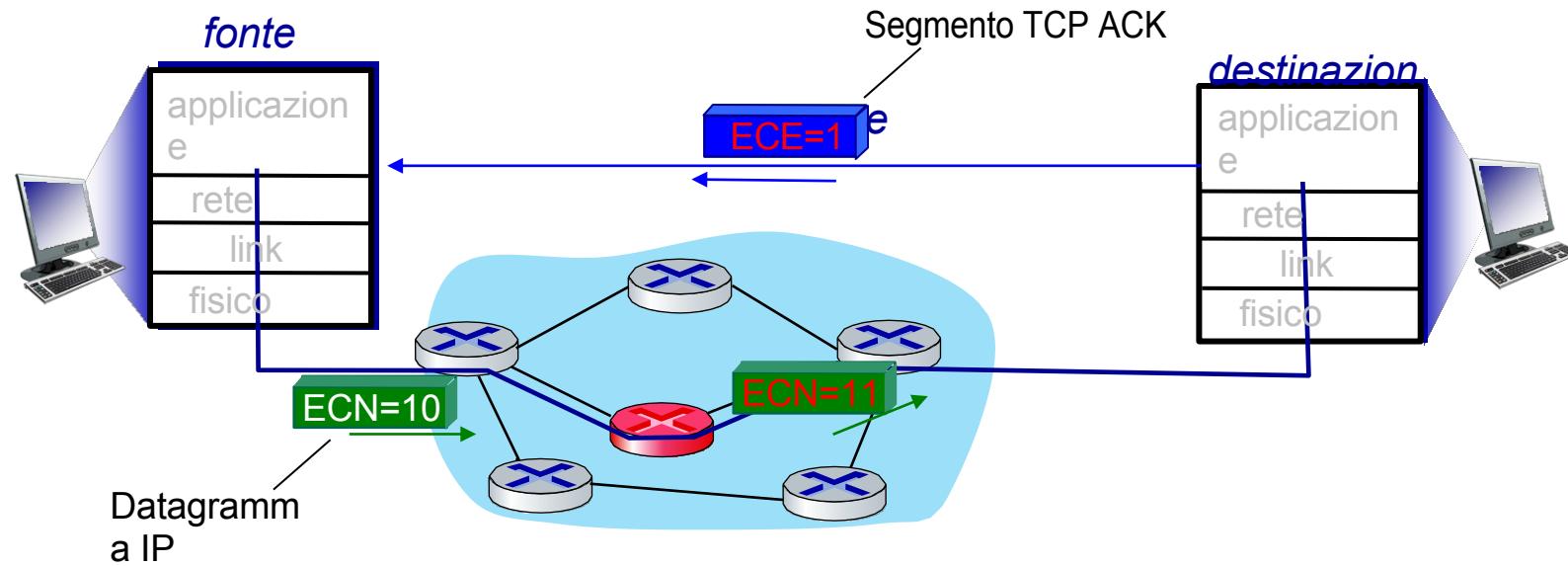
Controllo della congestione TCP basato sul ritardo

- controllo della congestione senza indurre/forzare perdite
- massimizzare l'intero percorso ("mantenere il tubo pieno...") mantenendo il ritardo basso ("ma non più pieno")
- alcuni TCP utilizzati adottano un approccio basato sul ritardo
 - BBR distribuito sulla rete dorsale (interna) di Google

Notifica esplicita di congestione (ECN)

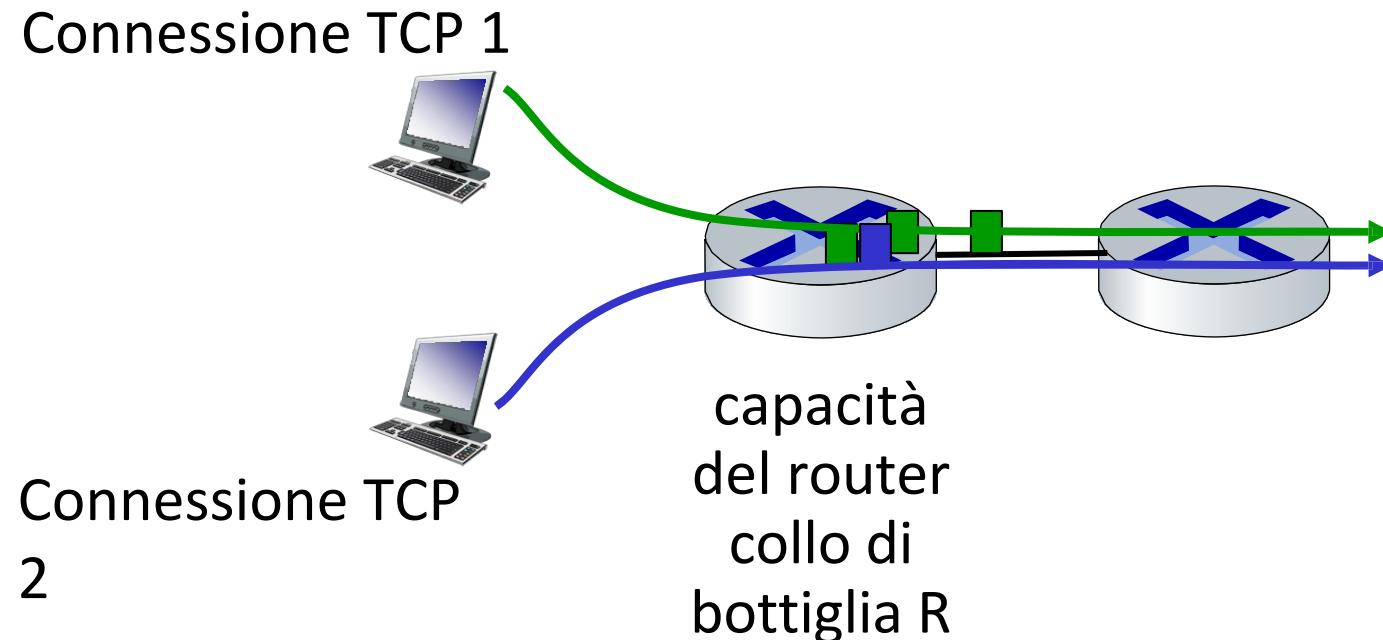
Le implementazioni di TCP spesso implementano un controllo della congestione *assistito dalla rete*:

- due bit nell'intestazione IP (campo ToS) contrassegnati *dal router di rete* per indicare una congestione
 - *politica* per determinare la marcatura scelta dall'operatore di rete
 - indicazione di congestione portata a destinazione
 - la destinazione imposta il bit ECE sul segmento ACK per notificare al mittente la presenza di una congestione
 - coinvolge sia l'IP (marcatura del bit ECN dell'intestazione IP) che il TCP (marcatura dei bit C,E dell'intestazione TCP).



Equità TCP

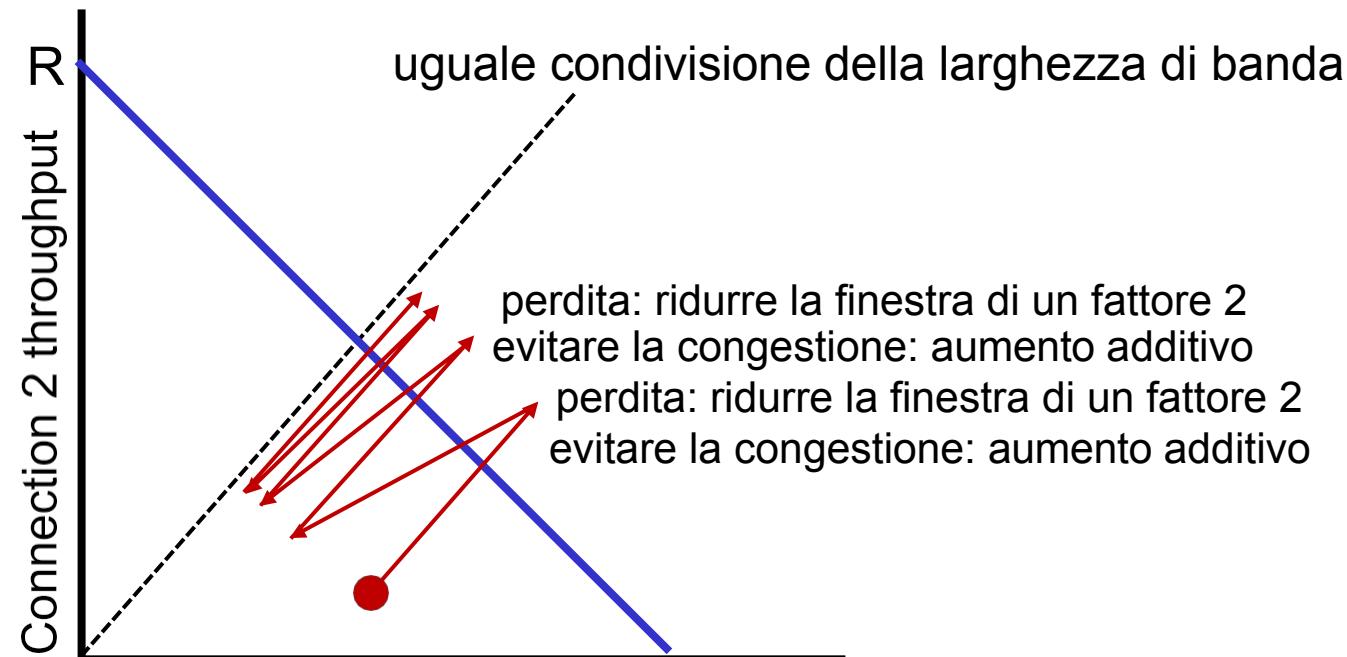
Obiettivo di equità: se K sessioni TCP condividono lo stesso collegamento a collo di bottiglia con larghezza di banda R , ciascuna dovrebbe avere una velocità media di R/K



D: Il TCP è equo?

Esempio: due sessioni TCP concorrenti:

- l'aumento additivo dà una pendenza pari a 1, con l'aumento di tutto il territorio
- la diminuzione moltiplicativa riduce il throughput in modo proporzionale



Il TCP è giusto?

R: Sì, in base a ipotesi idealizzate:

- stesso RTT
- numero fisso di sessioni solo in congestion avoidance

Portata della connessione 1 R

Correttezza: tutte le app di rete devono essere "corrette"?

Equità e UDP

- Le applicazioni multimediali spesso non utilizzano il protocollo TCP
 - non vogliono che la velocità sia limitata dal controllo della congestione
- utilizzare invece UDP:
 - inviare audio/video a velocità costante, tollerare la perdita di pacchetti
- non esiste una "polizia di

"Internet" che controlli l'uso del controllo della congestione

Equità, connessioni TCP parallele

- può aprire *più* connessioni parallele tra due host
- I browser web lo fanno, ad esempio il link del tasso R con 9 connessioni esistenti:
 - la nuova applicazione chiede 1 TCP, ottiene la tariffa $R/10$
 - la nuova applicazione chiede 11 TCP, ottiene $R/2$

Livello di trasporto: tabella di marcia

- Servizi del livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessioni: UDP
- Principi di trasferimento affidabile dei dati
- Trasporto orientato alla connessione: TCP
- Principi del controllo della congestione
- Controllo della congestione TCP
- **Evoluzione delle funzionalità del livello di trasporto**



Evoluzione delle funzionalità del livello di trasporto

- TCP, UDP: i principali protocolli di trasporto per 40 anni
- Sono stati sviluppati diversi "gusti" di TCP, per scenari specifici:

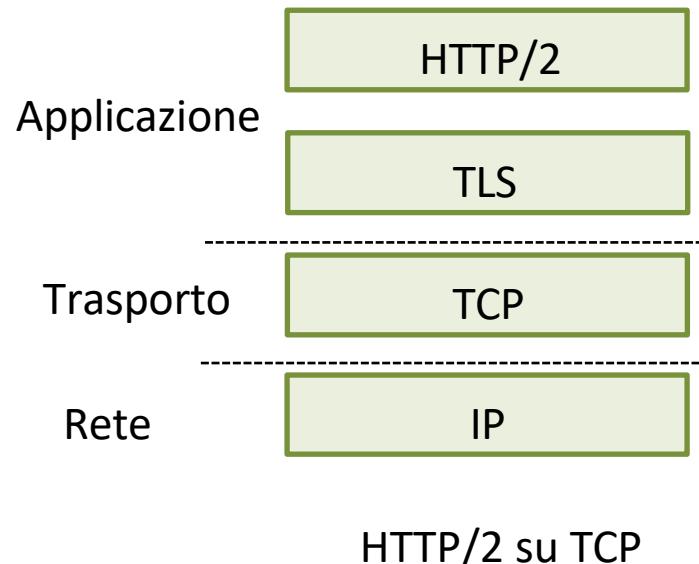
Scenario	Sfide
Tubi lunghi e grassi (dati di grandi dimensioni trasferimenti)	Molti pacchetti "in volo"; la perdita si interrompe condutture
Reti wireless	Perdita dovuta a collegamenti wireless rumorosi, mobilità; Il TCP lo tratta come una perdita di congestione
Collegamenti a lungo termine	RTT estremamente lunghi
Reti di centri dati	Sensibile alla latenza
Flussi di traffico di fondo	Flussi TCP a bassa priorità, "in

background"

- spostare le funzioni del livello di trasporto al livello applicativo, in cima a UDP
 - HTTP/3: QUIC

QUIC: Connessioni Internet UDP rapide

- protocollo di livello applicazione, in aggiunta a UDP
 - aumentare le prestazioni di HTTP
 - distribuito su molti server di Google, applicazioni (Chrome, app mobile di YouTube)



QUIC: Connessioni Internet UDP rapide

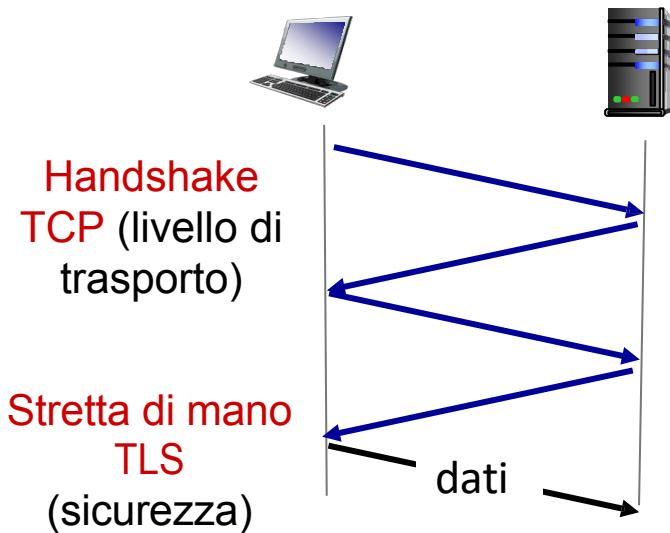
adotta gli approcci che abbiamo studiato in questo capitolo per creazione di connessioni, controllo degli errori, controllo della congestione

- **errore e controllo della congestione:** "I lettori che hanno familiarità con il rilevamento delle perdite e il controllo della congestione di TCP troveranno qui algoritmi paralleli a quelli ben noti di TCP." [da specifiche QUIC]
- **creazione della connessione:** affidabilità, controllo della congestione, autenticazione, crittografia, stato stabilito in un RTT
- flussi multipli a livello di applicazione multiplexati su un singolo QUIC

connessione

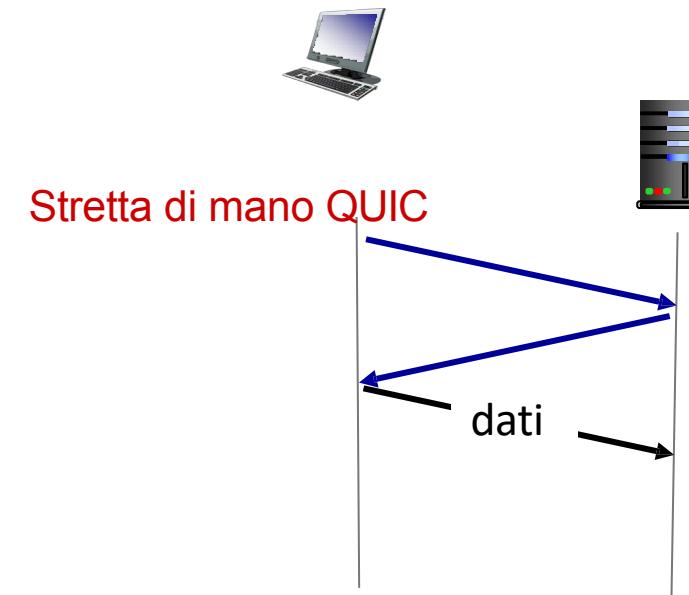
- separare il trasferimento affidabile dei dati, la sicurezza
- controllo comune della congestione

QUIC: creazione della connessione



TCP (affidabilità, stato di controllo della congestione) + TLS (autenticazione, stato di crittografia)

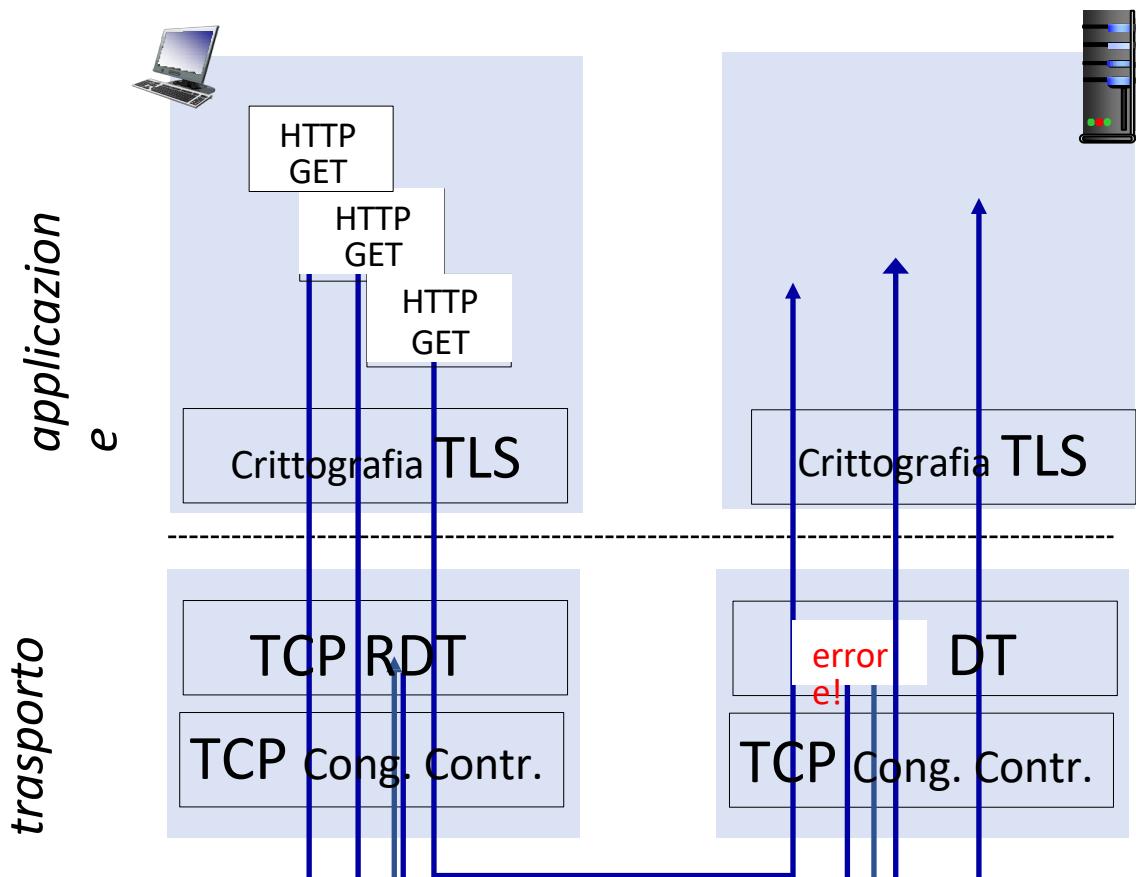
- 2 strette di mano seriali



QUIC: affidabilità, controllo della congestione, autenticazione, stato della crittografia

- 1 stretta di mano

QUIC: flussi: parallelismo, nessun blocco HOL



(a) HTTP 1.1

(b) HTTP/2 con QUIC: nessun blocco HOL

Capitolo 3: riassunto

- principi alla base dei servizi del livello di trasporto:
 - multiplexing, demultiplexing
 - trasferimento dati affidabile
 - controllo del flusso
 - controllo della congestione
- istanziazione, implementazione in Internet
 - UDP
 - TCP

Prossimamente:

- lasciando il "bordo" della rete (livelli di applicazione e trasporto)
- nel "cuore" della rete
- due capitoli sul livello di rete:
 - piano dati
 - piano di controllo