

Capitolo 3

Livello di trasporto

Nota per l'utilizzo:

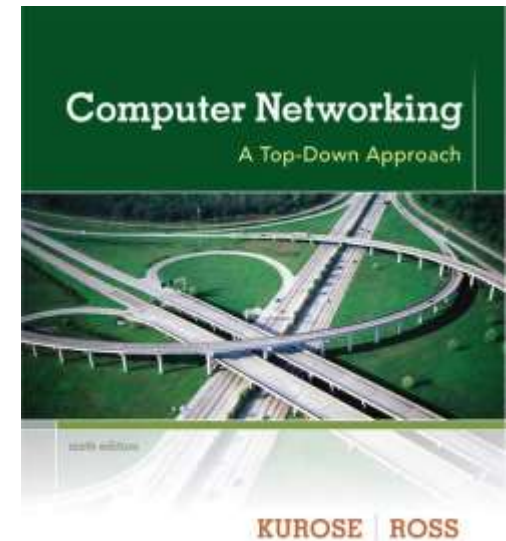
Abbiamo preparato queste slide con l'intenzione di renderle disponibili a tutti (professori, studenti, lettori). Sono in formato PowerPoint in modo che voi possiate aggiungere e cancellare slide (compresa questa) o modificarne il contenuto in base alle vostre esigenze.

Come potete facilmente immaginare, da parte nostra abbiamo fatto *un sacco* di lavoro. In cambio, vi chiediamo solo di rispettare le seguenti condizioni:

- ❑ se utilizzate queste slide (ad esempio, in aula) in una forma sostanzialmente inalterata, fate riferimento alla fonte (dopo tutto, ci piacerebbe che la gente usasse il nostro libro!)
- ❑ se rendete disponibili queste slide in una forma sostanzialmente inalterata su un sito web, indicate che si tratta di un adattamento (o che sono identiche) delle nostre slide, e inserite la nota relativa al copyright.

Thanks and enjoy! JFK/KWR

©All material copyright 1996-2012
J.F Kurose and K.W. Ross, All Rights Reserved



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

Capitolo 3: Livello di trasporto

obiettivi:

- ❖ capire i principi che sono alla base dei servizi del livello di trasporto:
 - multiplexing, demultiplexing
 - trasferimento dati affidabile
 - controllo di flusso
 - controllo di congestione
- ❖ descrivere i protocolli del livello di trasporto di Internet:
 - UDP: trasporto senza connessione
 - TCP: trasporto orientato alla connessione
 - controllo di congestione TCP

Capitolo 3: Livello di trasporto

3.1 servizi a livello di trasporto

3.2 multiplexing e demultiplexing

3.3 trasporto senza connessione: UDP

3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

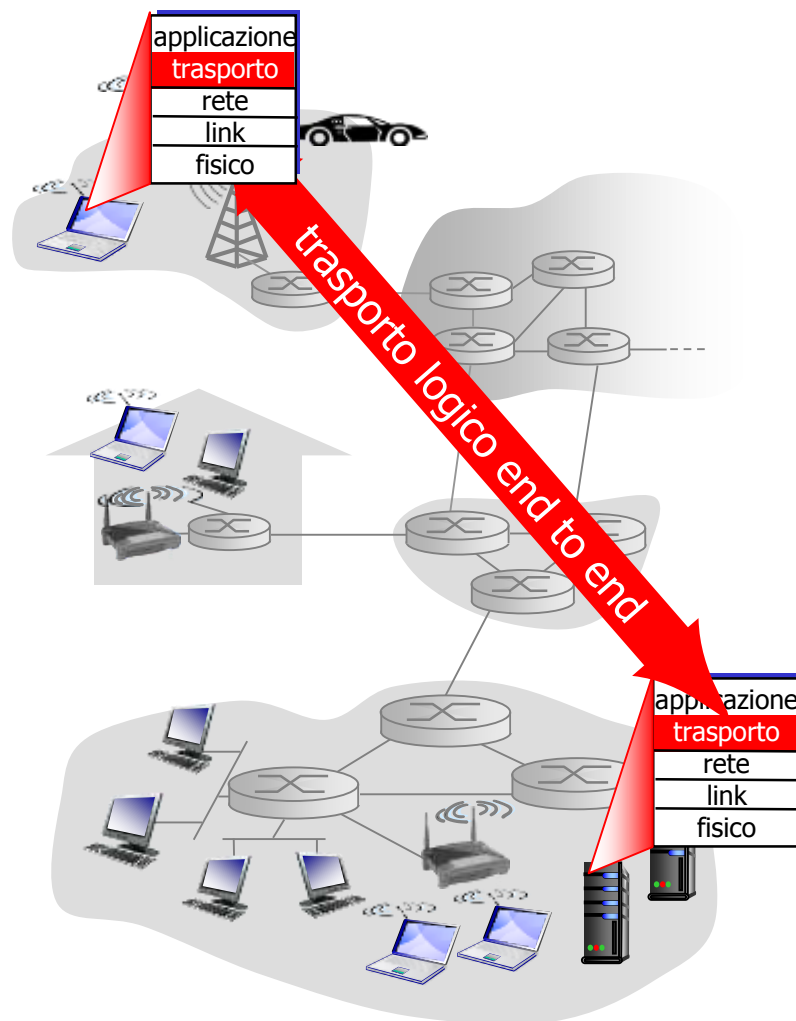
- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

Servizi e protocolli di trasporto

- ❖ forniscono la **comunicazione logica** tra processi applicativi su host differenti
- ❖ i protocolli di trasporto vengono eseguiti sugli end systems
 - lato invio: divide i messaggi in segmenti e li passa al livello di rete
 - lato ricezione: riassume i segmenti in messaggi e li passa al livello di applicazione
- ❖ più protocolli di trasporto sono a disposizione delle applicazioni
 - Internet: TCP e UDP



Confronto tra livello di trasporto e livello di rete

- ❖ *livello di rete:*
comunicazione logica tra host
- ❖ *livello di trasporto:*
comunicazione logica tra processi
 - si basa sui servizi (se può li migliora) del livello di rete

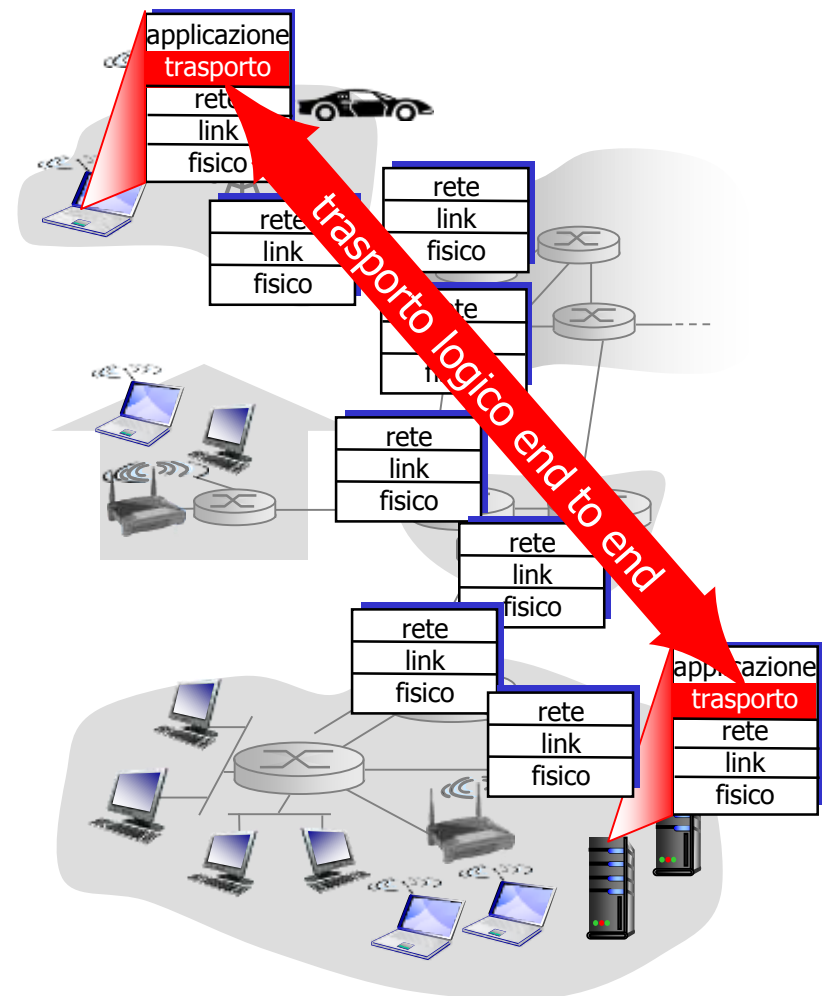
analogia con la posta ordinaria:

i ragazzi della casa di Ann inviano lettere ai ragazzi nella casa di Bill:

- ❖ host = case
- ❖ processi = ragazzi
- ❖ messaggi delle applicazioni = lettere nelle buste
- ❖ protocollo di trasporto = Ann e Bill che smistano le lettere
- ❖ protocollo del livello di rete = servizio postale

Protocolli del livello di trasporto in Internet

- ❖ affidabile, consegne nell'ordine originario (TCP)
 - controllo di congestione
 - controllo di flusso
 - setup della connessione
- ❖ inaffidabile, consegne senz'ordine: UDP
 - estensione senza fronzoli del servizio di consegna a massimo sforzo ("best-effort")
- ❖ servizi non disponibili:
 - garanzie sui ritardi
 - garanzie sulla bandwidth



Capitolo 3: Livello di trasporto

3.1 servizi a livello di trasporto

3.2 multiplexing e demultiplexing

3.3 trasporto senza connessione: UDP

3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

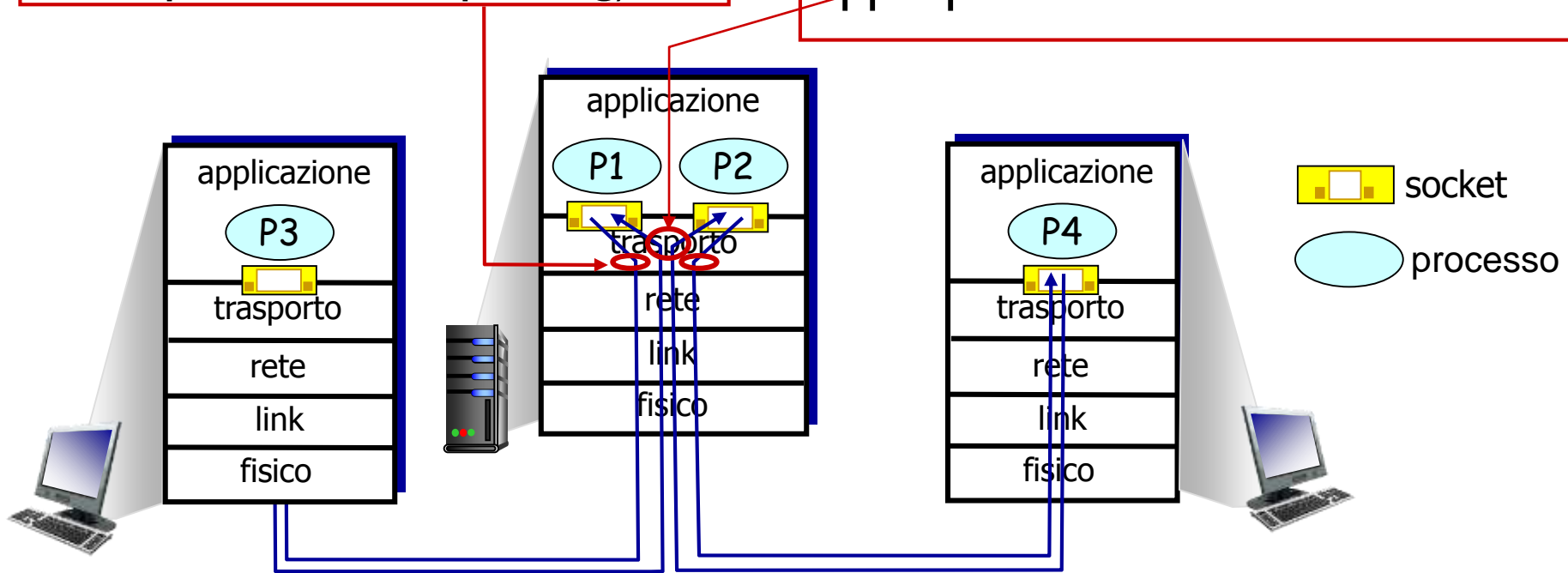
Multiplexing/demultiplexing

multiplexing in invio:

maneggia i dati provenienti dalle diverse socket, aggiunge le intestazioni di trasporto (poi usate per il demultiplexing)

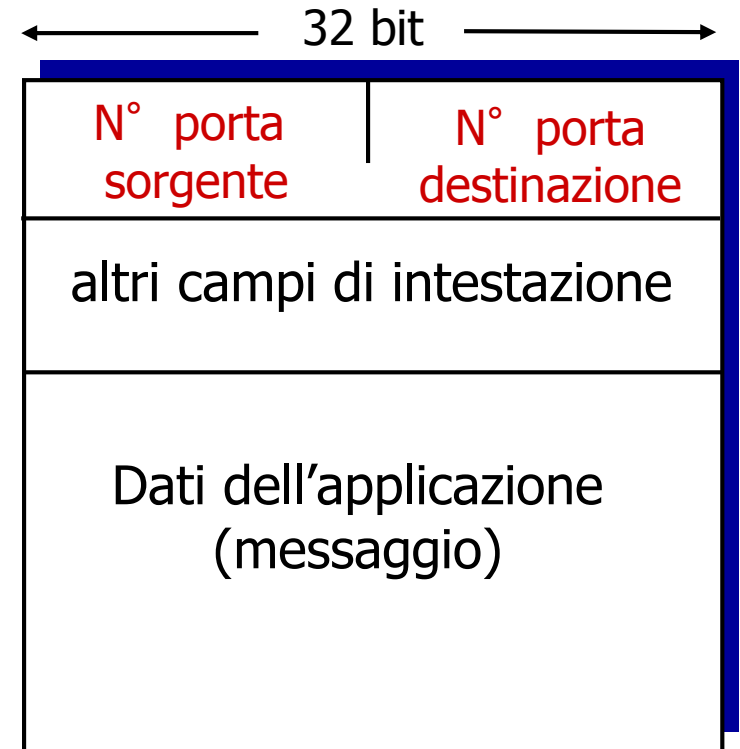
demultiplexing in ricezione:

usa le intestazioni per consegnare i segmenti ricevuti alla socket appropriata




Come funziona il demultiplexing

- ❖ l'host riceve i datagrammi IP
 - ogni datagramma ha un indirizzo IP sorgente 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



Struttura del segmento TCP/UDP

Demultiplexing senza connessione

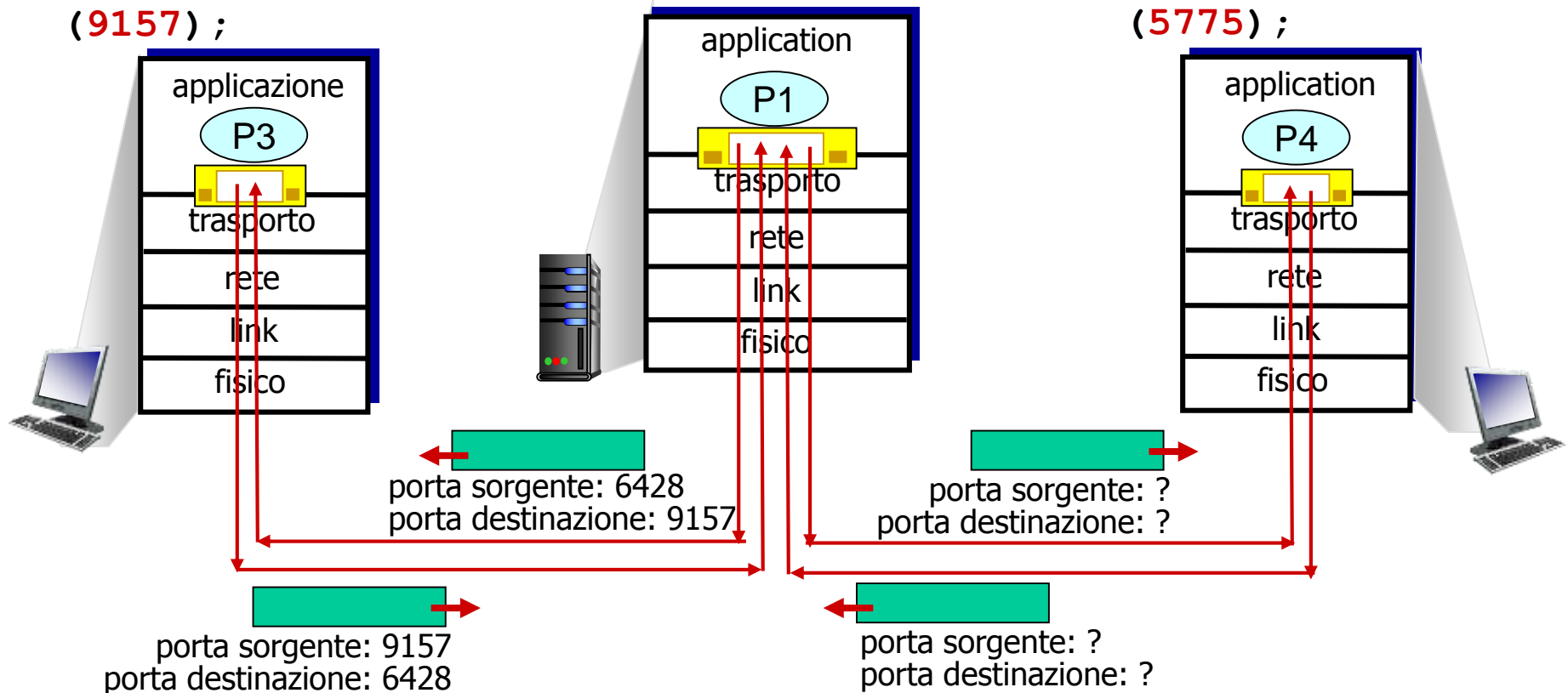
- ❖ la socket creata ha un numero di porta locale:
`DatagramSocket mySocket1 = new DatagramSocket(12534);`
 - ❖ quando si crea un datagramma da inviare a una socket UDP, vengono specificati
 - indirizzo IP di destinazione
 - numero di porta di destinazione
-
- ❖ quando un host riceve un segmento UDP:
 - controlla il numero di porta di destinazione
 - dirige il segmento UDP alla socket con quel numero di porta
- 
- i datagram IP con lo *stesso numero di porta di destinazione*, anche se con differenti indirizzi IP sorgenti e/o numeri di porta sorgente verranno diretti alla *stessa socket* nella destinazione

Demultiplexing senza connessione

```
DatagramSocket  
mySocket2 = new  
DatagramSocket  
(9157) ;
```

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

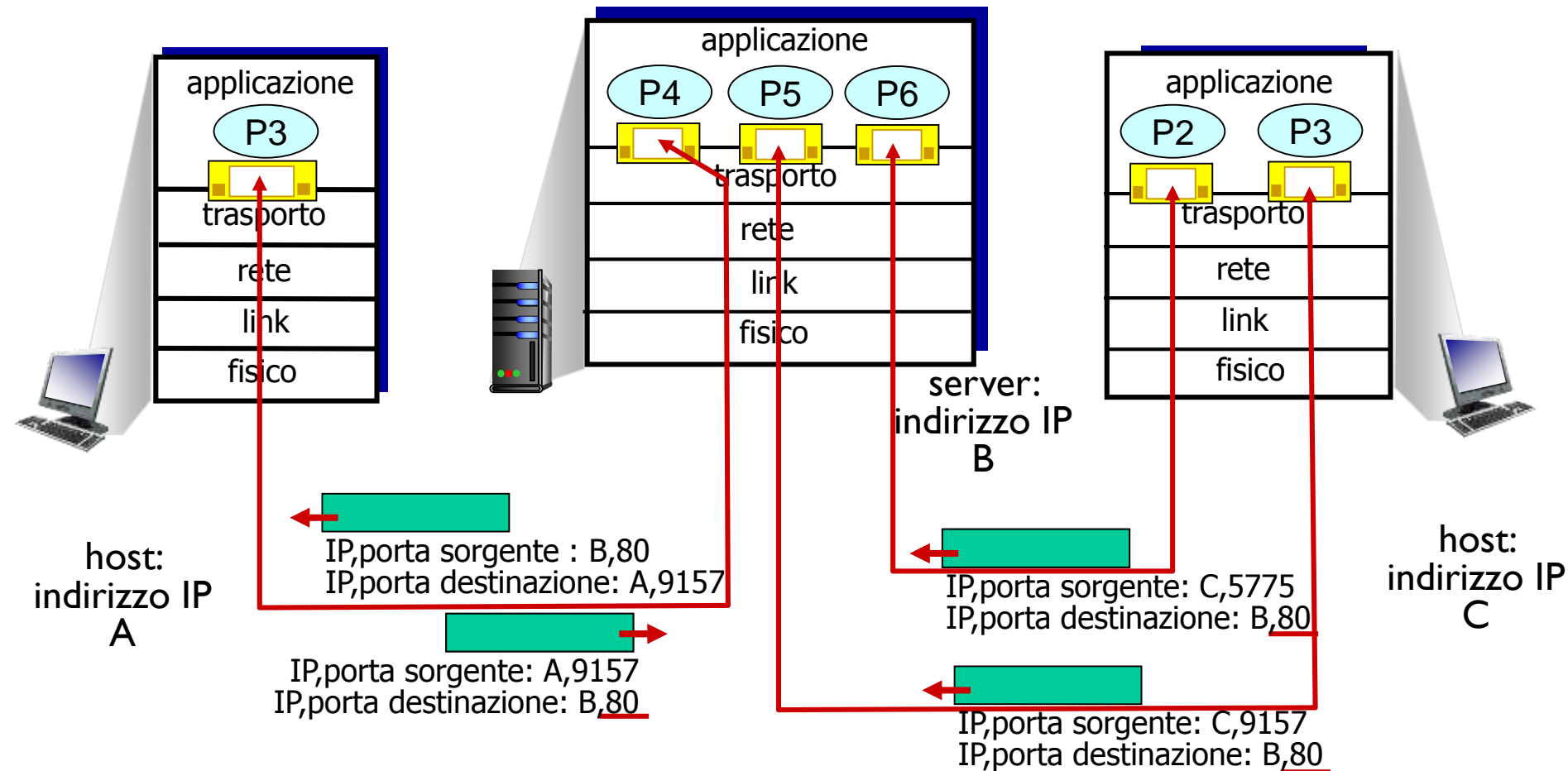
```
DatagramSocket  
mySocket1 = new  
DatagramSocket  
(5775) ;
```



Demultiplexing orientato alla connessione

- ❖ il socket TCP è identificato da 4 parametri:
 - indirizzo IP sorgente
 - numero di porta sorgente
 - indirizzo IP di destinazione
 - numero di porta di destinazione
- ❖ il ricevente usa tutti e quattro i parametri per inviare i segmenti 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

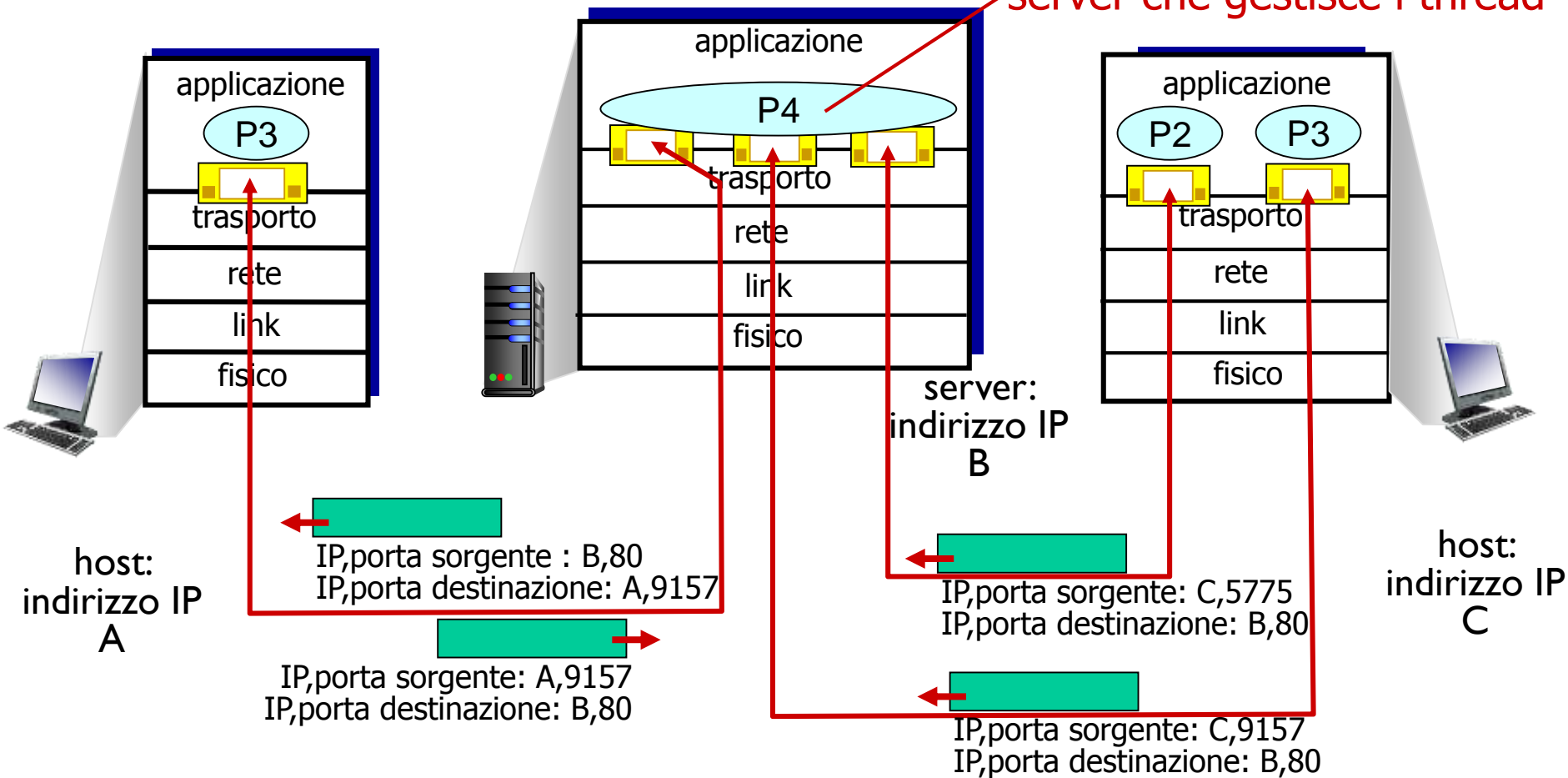
Demultiplexing orientato alla connessione



tre segmenti, tutti destinati all'indirizzo IP: B,
porta: 80 sono demultiplexati verso socket *differenti*

Demultiplexing orientato alla connessione

server che gestisce i thread



Capitolo 3: Livello di trasporto

3.1 servizi a livello di trasporto

3.2 multiplexing e demultiplexing

3.3 trasporto senza connessione: UDP

3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

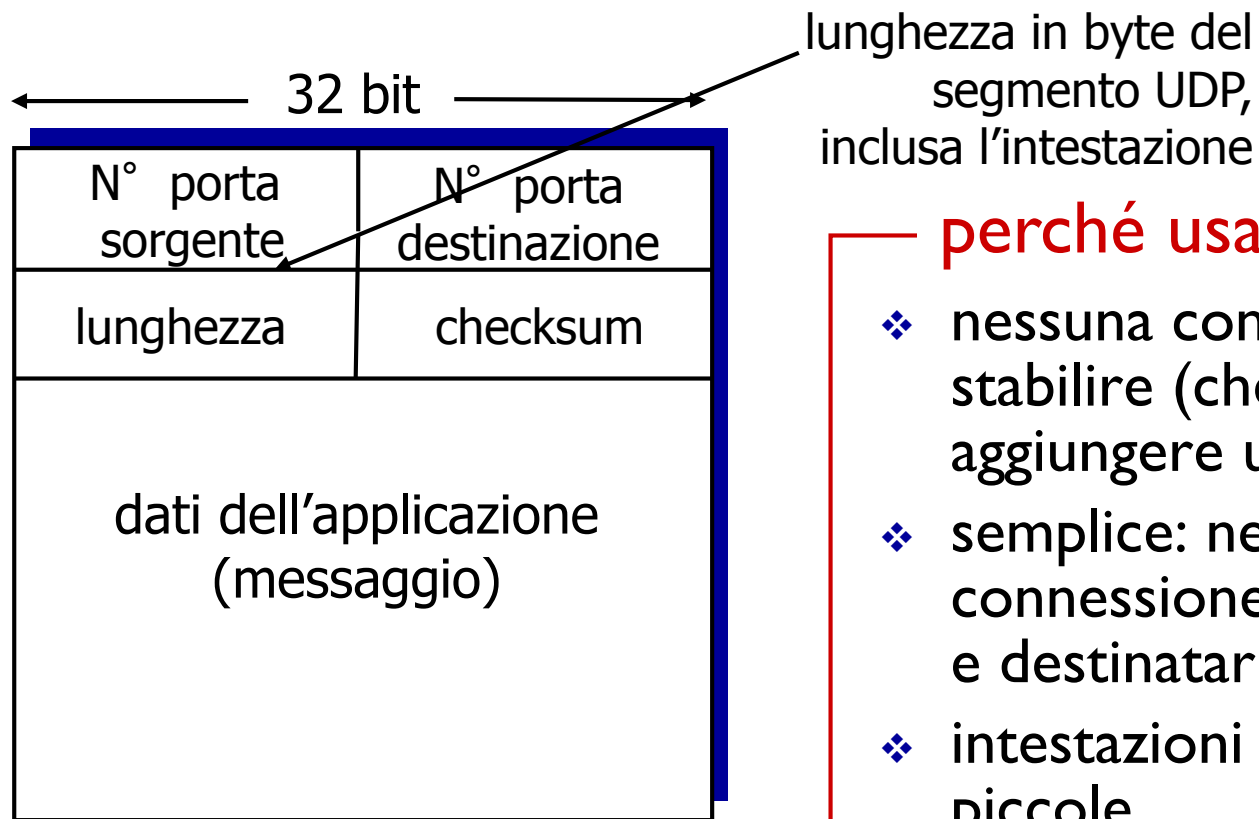
3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

UDP: User Datagram Protocol [RFC 768]

- ❖ protocollo di trasporto “senza fronzoli”
- ❖ servizio “best effort”, i segmenti UDP possono essere:
 - perduti
 - consegnati fuori sequenza all'applicazione
- ❖ *senza connessione*:
 - non c'è handshaking tra mittente e destinatario UDP
 - ogni segmento UDP è gestito indipendentemente dagli altri
- ❖ uso di UDP :
 - applicazioni di streaming multimediali (tolleranti alle perdite, sensibili al rate)
 - DNS
 - SNMP
- ❖ trasferimento affidabile con UDP:
 - l'affidabilità va aggiunta al livello di applicazione
 - il recupero degli errori deve essere gestito dalle applicazioni!

UDP: intestazione del segmento



struttura del segmento UDP

perché usare UDP?

- ❖ nessuna connessione da stabilire (che può aggiungere un ritardo)
- ❖ semplice: nessuno stato di connessione nel mittente e destinatario
- ❖ intestazioni di segmento piccole
- ❖ non c'è controllo di congestione: UDP può sparare dati a raffica

Checksum UDP

Obiettivo: rilevare gli “errori” (bit alterati) nel segmento trasmesso

mittente:

- ❖ tratta il contenuto del segmento, compresa l'intestazione, come una sequenza di interi a 16 bit
- ❖ checksum: somma (complemento a 1) dei contenuti del segmento
- ❖ il mittente pone il valore della checksum nel campo checksum del segmento UDP

ricevente:

- ❖ calcola la checksum del segmento ricevuto
- ❖ controlla se la checksum calcolata è uguale al valore del campo checksum:
 - NO - errore rilevato
 - YES - nessun errore rilevato. *Ma potrebbero esserci errori nonostante questo? Altro più avanti...*

Checksum: esempio

esempio: sommare 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/>																
ritorno	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1
<hr/>																
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, un riporto dal bit più significativo deve essere sommato al risultato

Capitolo 3: Livello di trasporto

3.1 servizi a livello di trasporto

3.2 multiplexing e demultiplexing

3.3 trasporto senza connessione: UDP

3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

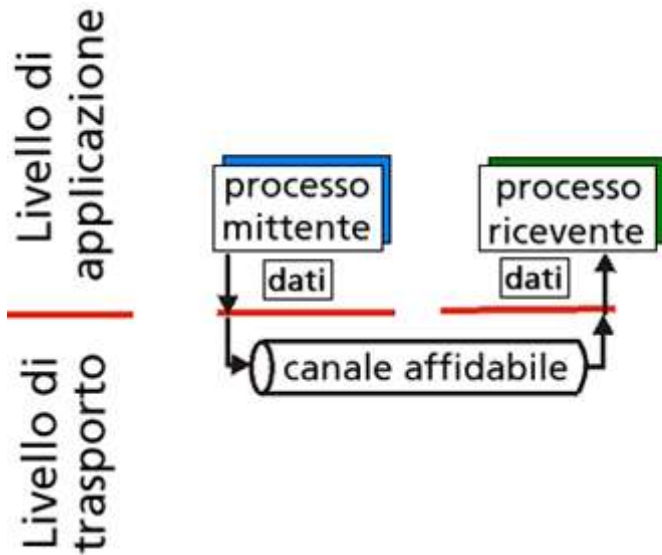
- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

Principi del trasferimento dati affidabile

- ❖ importante nei livelli di applicazione, trasporto e link
 - nella top-10 dei problemi più importanti del networking!

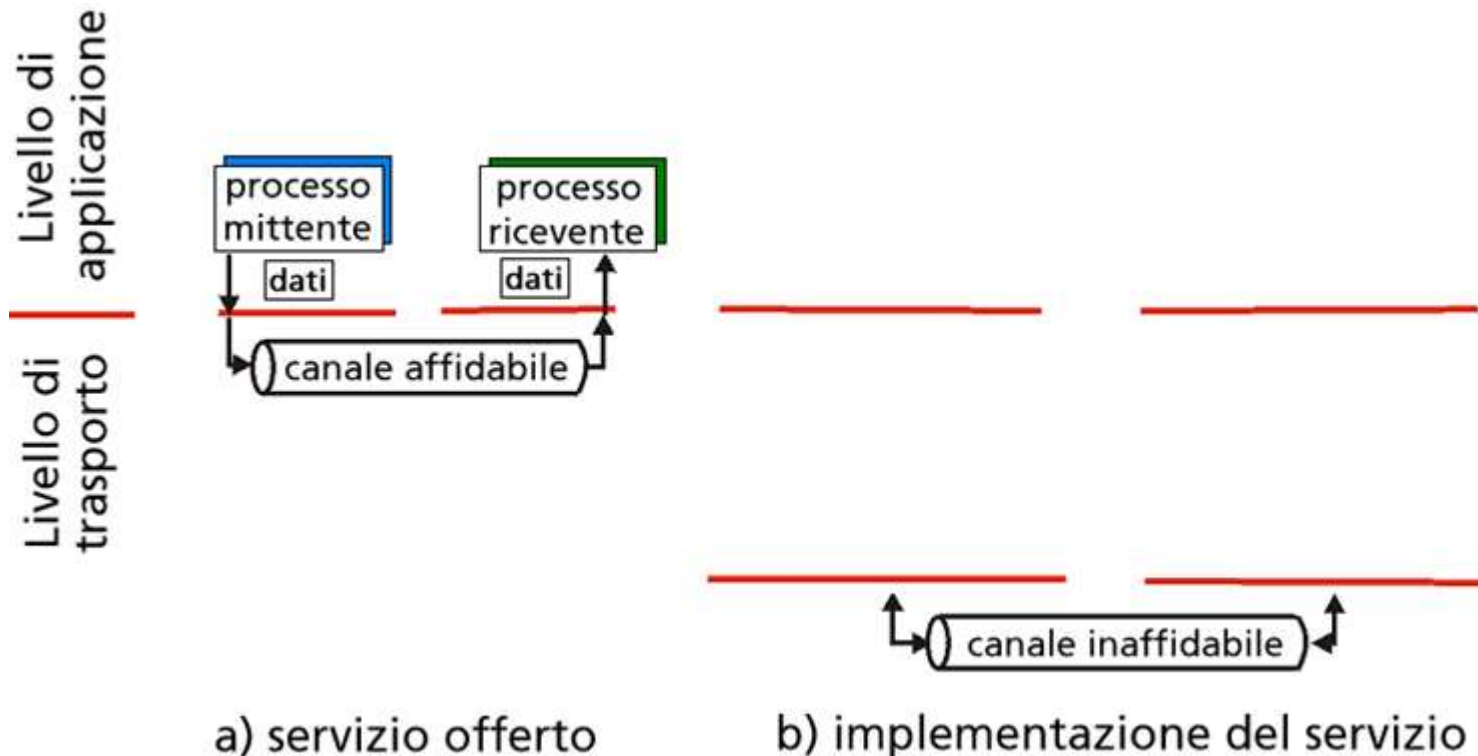


a) servizio offerto

- ❖ le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Principi del trasferimento dati affidabile

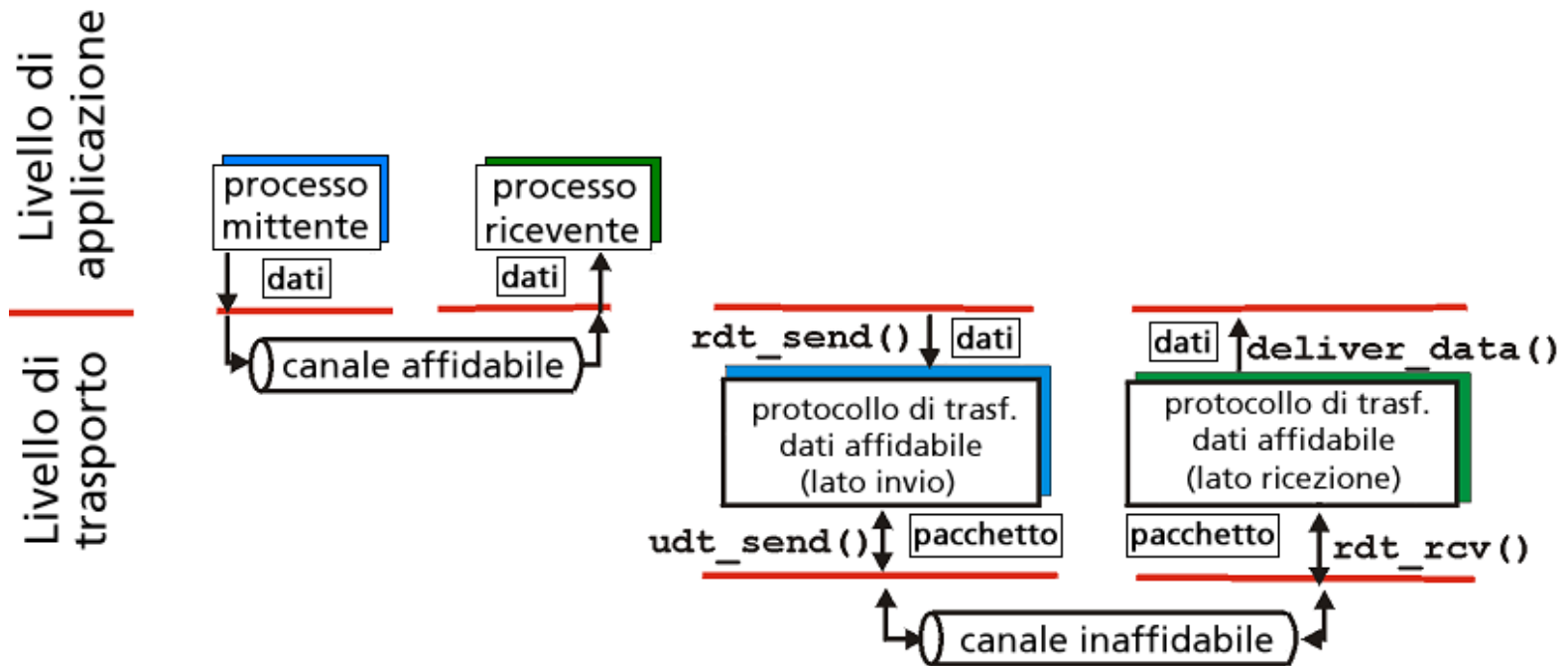
- ❖ importante nei livelli di applicazione, trasporto e link
 - nella top-10 dei problemi più importanti del networking!



- ❖ le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Principi del trasferimento dati affidabile

- ❖ importante nei livelli di applicazione, trasporto e link
 - nella top-10 dei problemi più importanti del networking!



a) servizio offerto

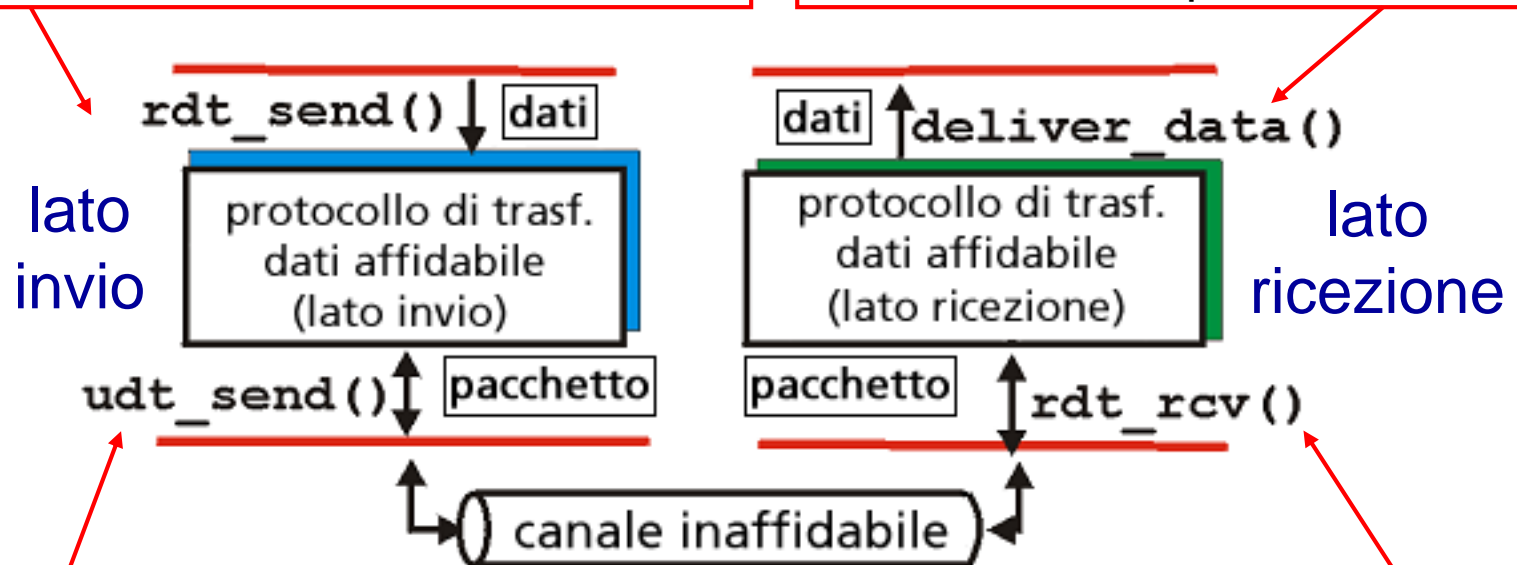
b) implementazione del servizio

- ❖ le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)

Trasferimento dati affidabile: preparazione

rdt_send() : chiamata dall'alto, (ad es. dall'applicazione). Trasferisce i dati da consegnare al livello superiore del ricevente

deliver_data() : chiamata da **rdt** per consegnare i dati al livello superiore



udt_send() : chiamata da rdt per trasferire il pacchetto al ricevente tramite il canale inaffidabile

rdt_rcv() : chiamata quando il pacchetto arriva nel lato ricezione del canale

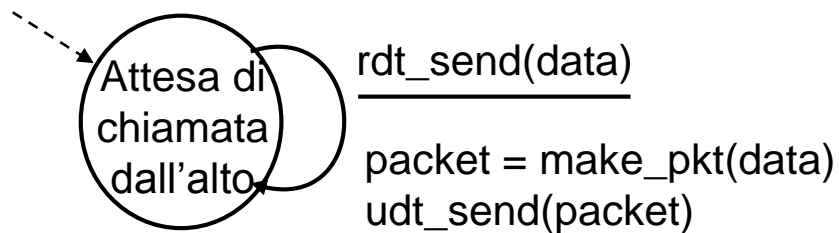
Trasferimento dati affidabile: preparazione

- ❖ svilupperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile (rdt)
- ❖ considereremo soltanto i trasferimenti dati unidirezionali
 - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- ❖ utilizzeremo automi a stati finiti (finite state machine, FSM) per specificare il mittente e il ricevente

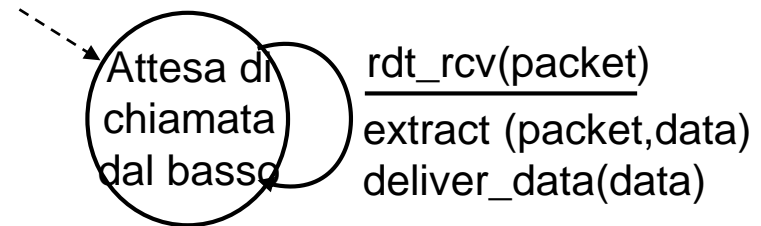


rdt1.0: trasferimento affidabile su canale affidabile

- ❖ canale sottostante perfettamente affidabile
 - nessun errore nei bit
 - nessuna perdita di pacchetti
- ❖ FSM distinti per il mittente e per il ricevente:
 - il mittente invia i dati nel canale sottostante
 - il ricevente legge i dati dal canale sottostante



mittente



ricevente

rdt2.0: canale con errori nei bit

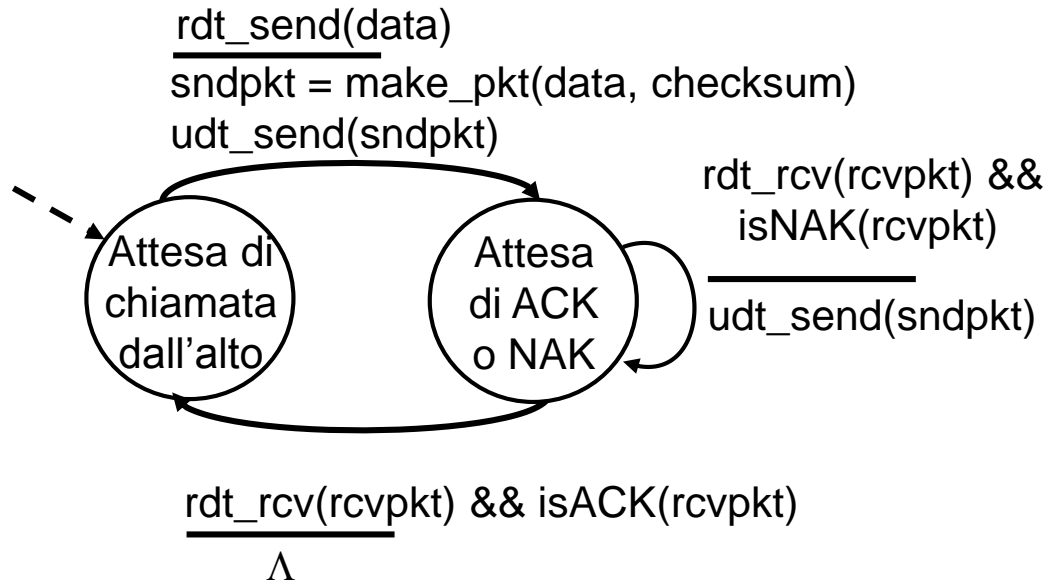
- ❖ il canale sottostante potrebbe confondere i bit nei pacchetti
 - checksum per rilevare gli errori nei bit
- ❖ *domanda*: come correggere gli errori:

Come fanno gli uomini a recuperare gli “errori” (incomprensioni) durante una conversazione?

rdt2.0: canale con errori nei bit

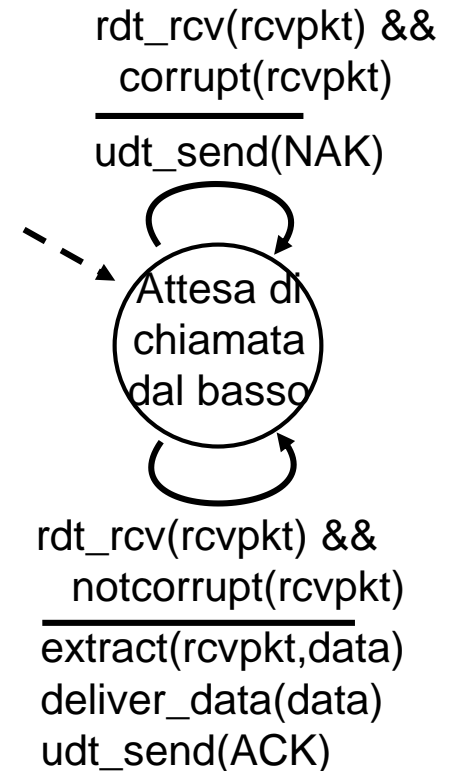
- ❖ il canale sottostante potrebbe confondere i bit nei pacchetti
 - checksum per rilevare gli errori nei bit
- ❖ *domanda*: come correggere gli errori:
 - *acknowledgements (ACKs)*: il ricevente dice esplicitamente al mittente che il pacchetto ricevuto è corretto
 - *negative acknowledgements (NAKs)*: il ricevente dice esplicitamente al mittente che il pacchetto ricevuto contiene errori
 - il mittente ritrasmette il pacchetto se riceve un NAK
- ❖ nuovi meccanismi in `rdt2.0` (rispetto a `rdt1.0`):
 - rilevamento degli errori
 - feedback del destinatario: messaggi di controllo (ACK, NAK) ricevente->mittente

rdt2.0: specifica del FSM

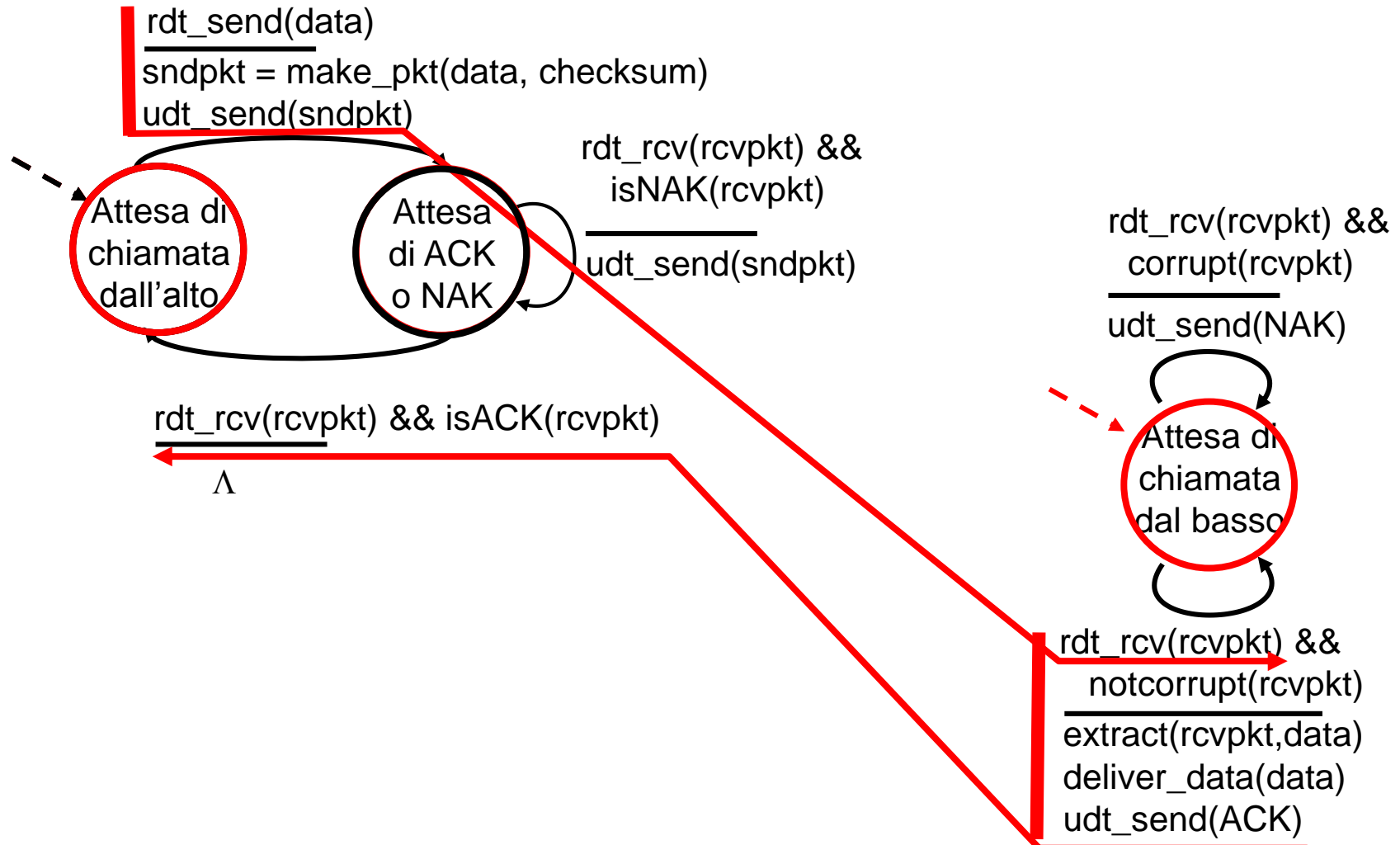


mittente

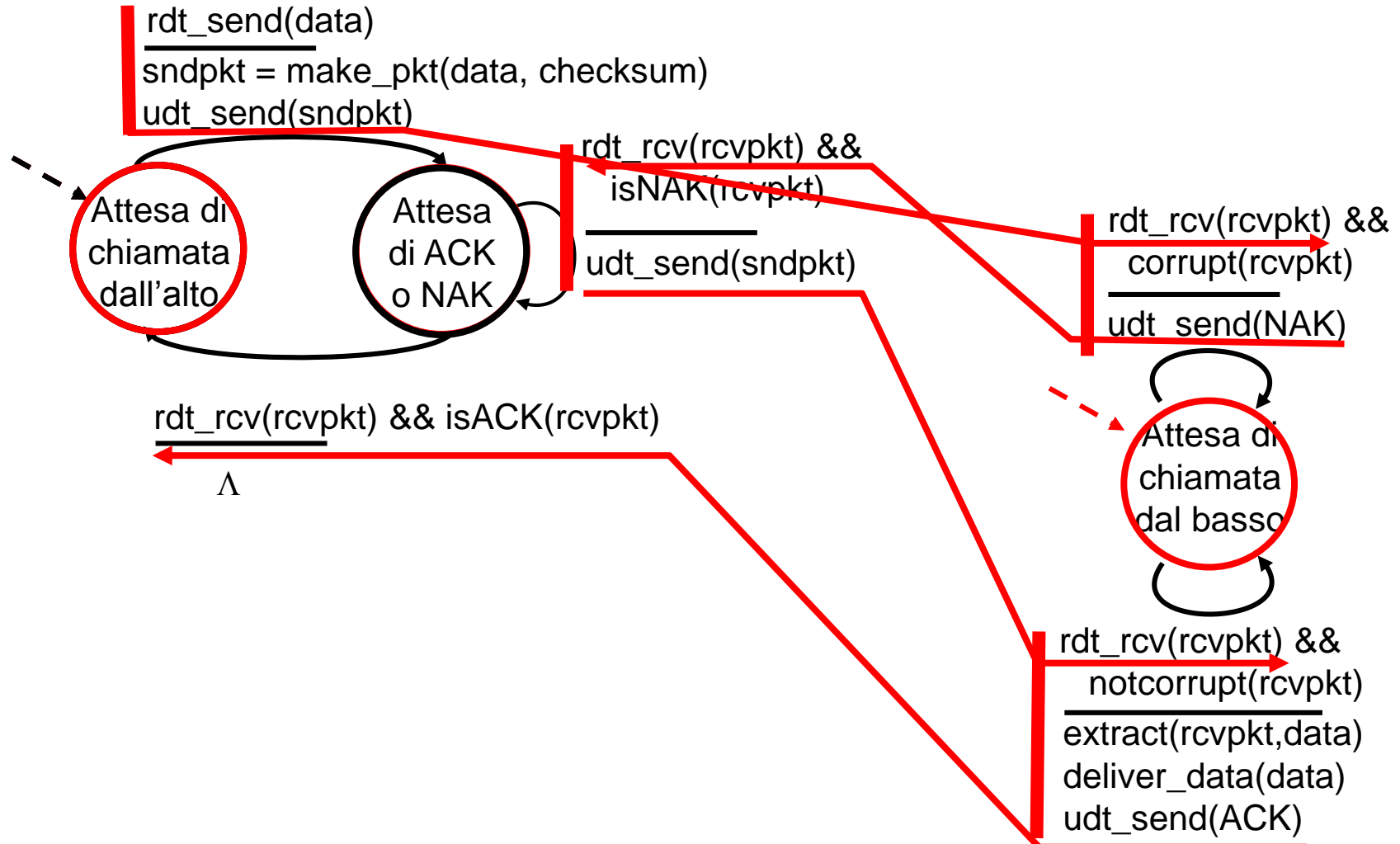
ricevente



rdt2.0: operazione senza errori



rdt2.0: scenario con errore



rdt2.0 ha un difetto fatale!

che accade se i pacchetti
ACK/NAK sono
danneggiati?

- ❖ il mittente non sa che cosa sia accaduto al destinatario!
- ❖ non può semplicemente ritrasmettere: possibili duplicati

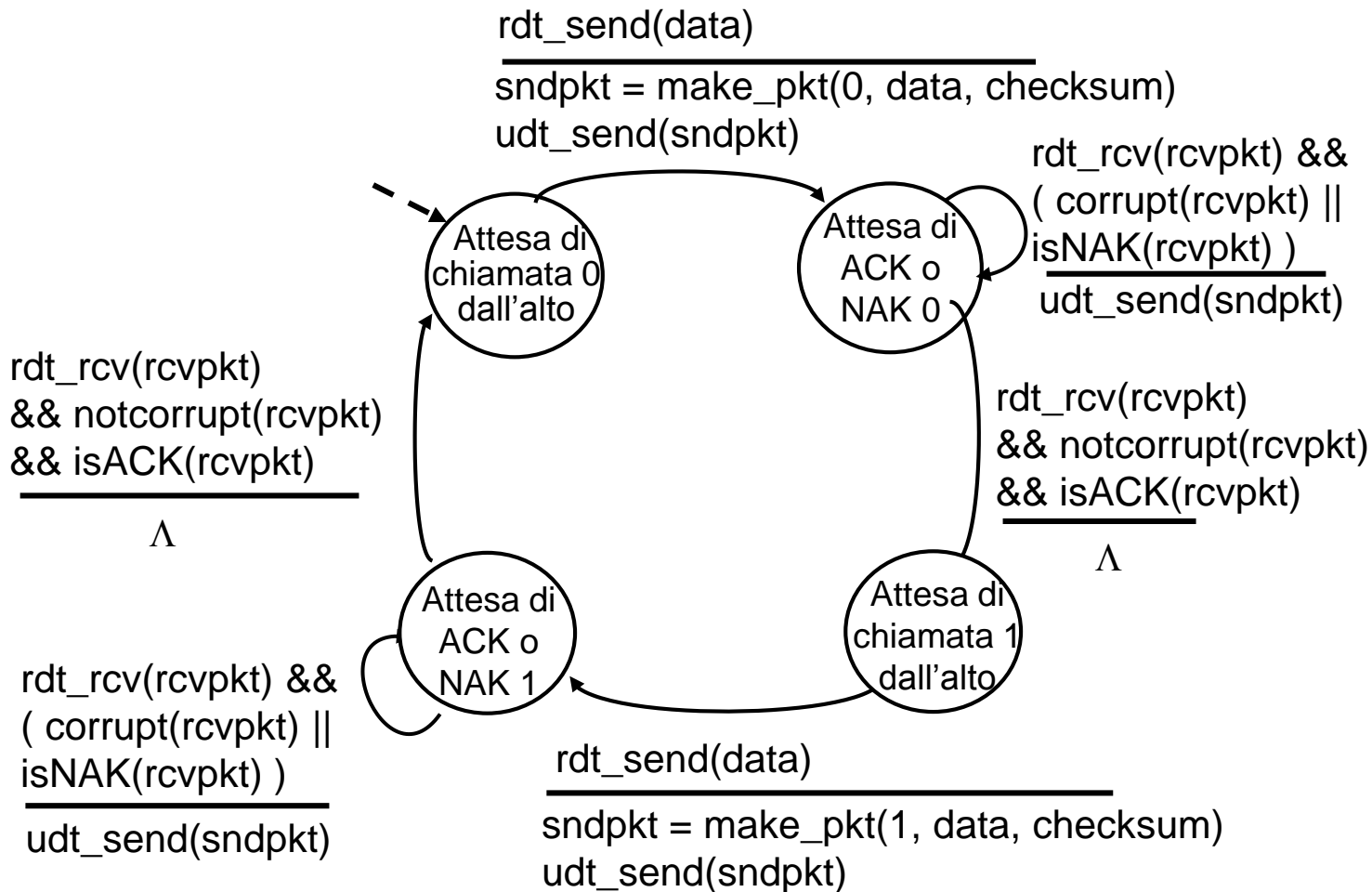
gestione dei duplicati:

- ❖ il mittente ritrasmette il pacchetto corrente se ACK/NAK è alterato
- ❖ il mittente aggiunge un *numero di sequenza* a ogni pacchetto
- ❖ il ricevente scarta il pacchetto duplicato

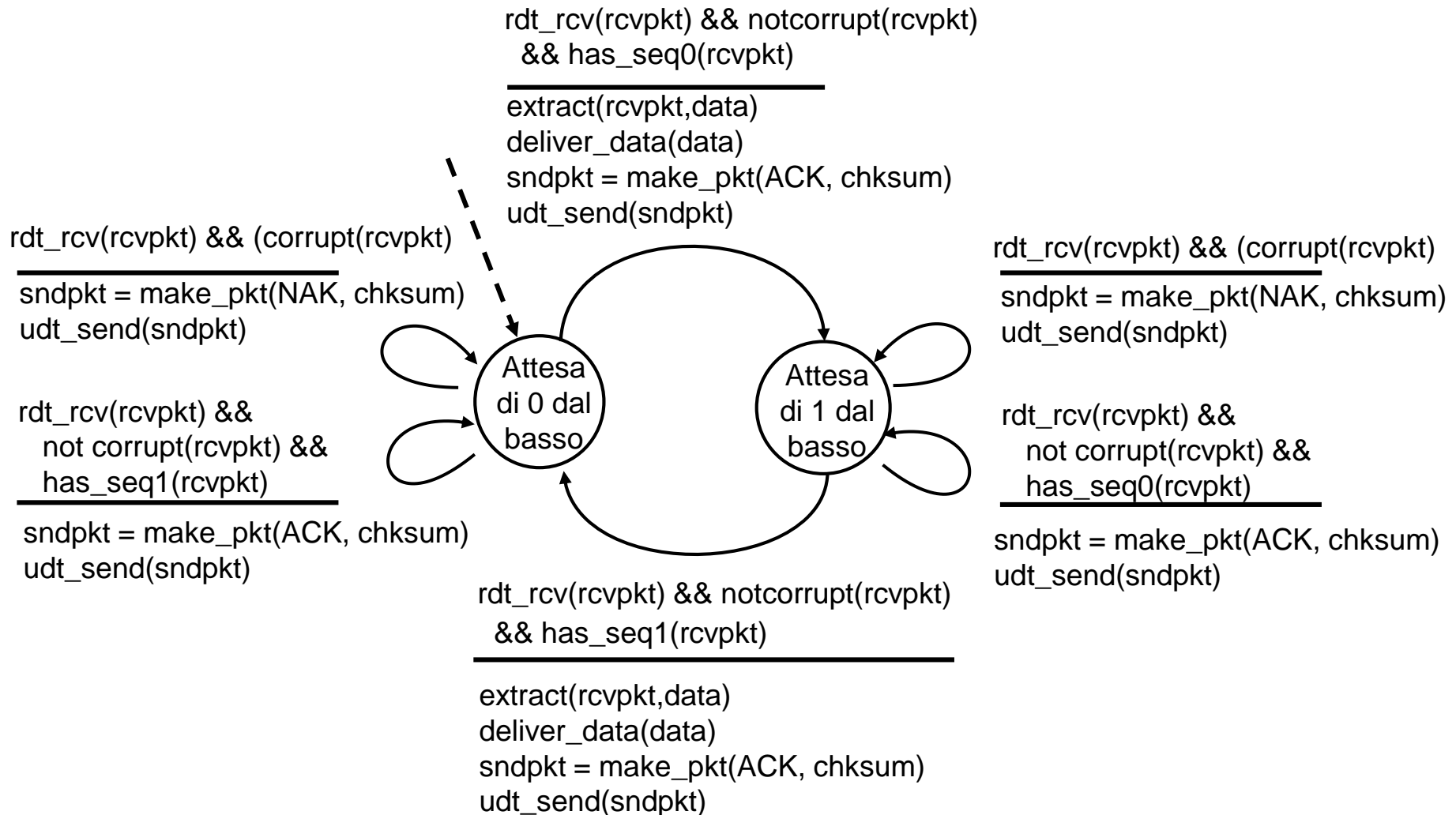
— stop and wait —

il mittente invia un
pacchetto, poi aspetta la
risposta del destinatario

rdt2.1: il mittente gestisce gli ACK/NAK corrotti



rdt2.1: il ricevente gestisce gli ACK/NAK corrotti



rdt2.1: discussione

mittente:

- ❖ aggiunge il numero di sequenza al pacchetto
- ❖ saranno sufficienti due numeri di sequenza (0,1). Perché?
- ❖ deve controllare se gli ACK/NAK sono danneggiati
- ❖ il doppio di stati
 - lo stato deve “ricordare” se il pacchetto “atteso” ha numero di sequenza 0 o 1

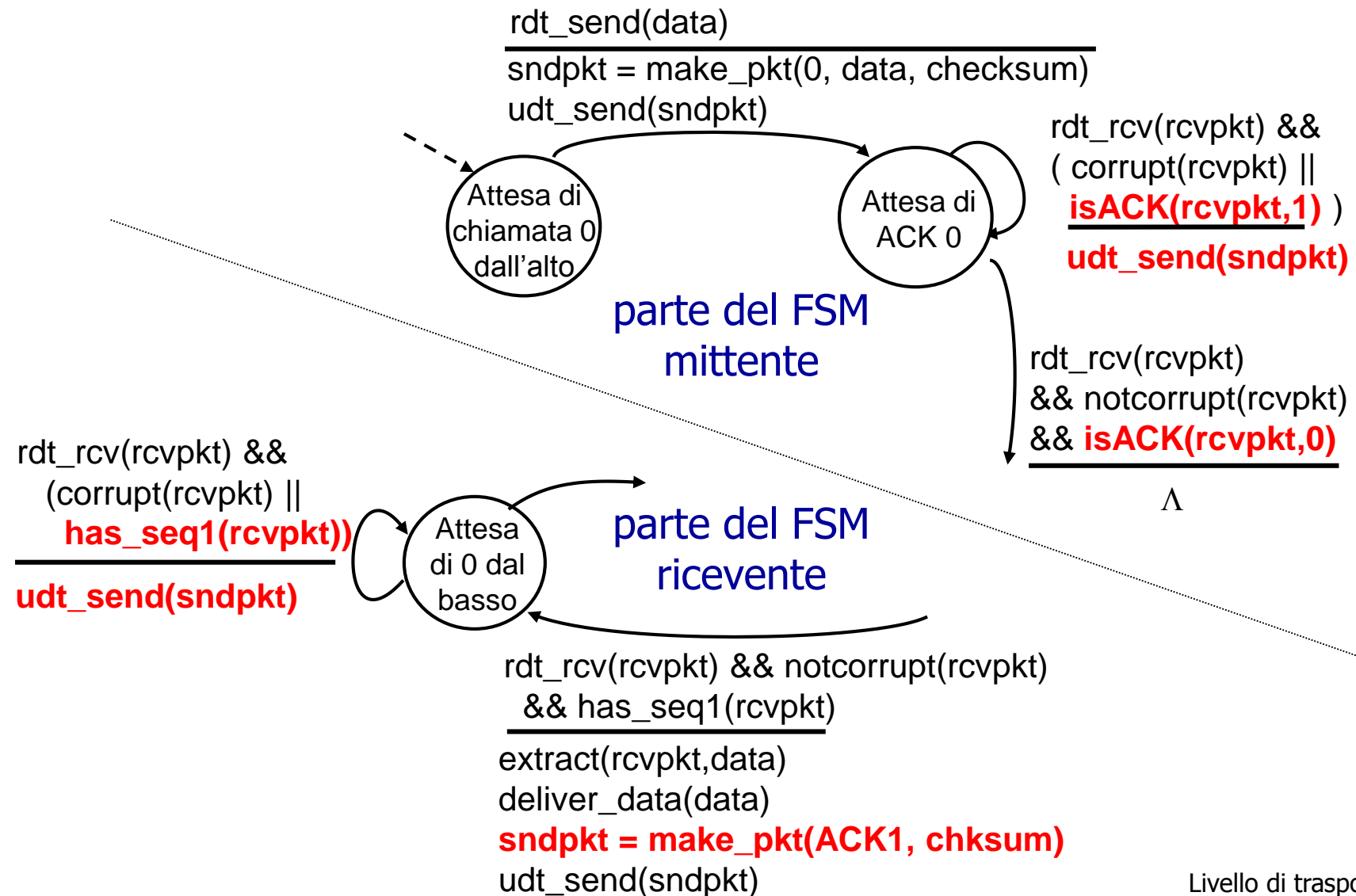
ricevente:

- ❖ deve controllare se il pacchetto ricevuto è duplicato
 - lo stato indica se il numero di sequenza previsto è 0 o 1
- ❖ nota: il ricevente *non* può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

rdt2.2: un protocollo senza NAK

- ❖ stessa funzionalità di rdt2.1, utilizzando soltanto gli ACK
- ❖ invece del NAK, il destinatario invia un ACK per l'ultimo pacchetto ricevuto correttamente
 - il destinatario deve includere *esplicitamente* il numero di sequenza del pacchetto con l'ACK
- ❖ un ACK duplicato presso il mittente determina la stessa azione del NAK: *ritrasmettere il pacchetto corrente*

rdt2.2: mittente e ricevente



rdt3.0: canali con errori e perdite

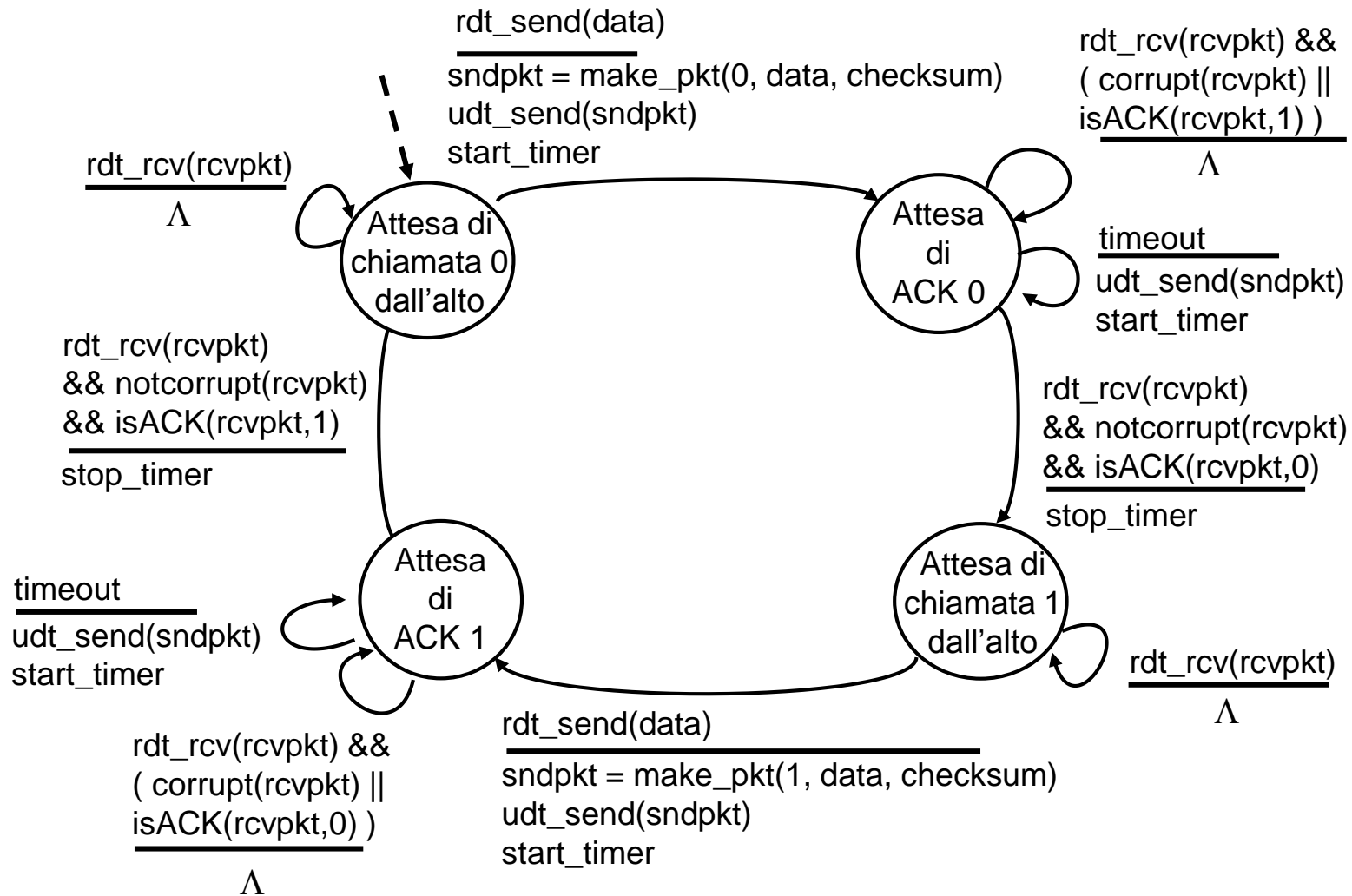
nuova ipotesi: il canale sottostante può anche perdere i pacchetti (dati o ACK)

- checksum, numero di sequenza, ACK e ritrasmissioni aiuteranno, ma non saranno sufficienti

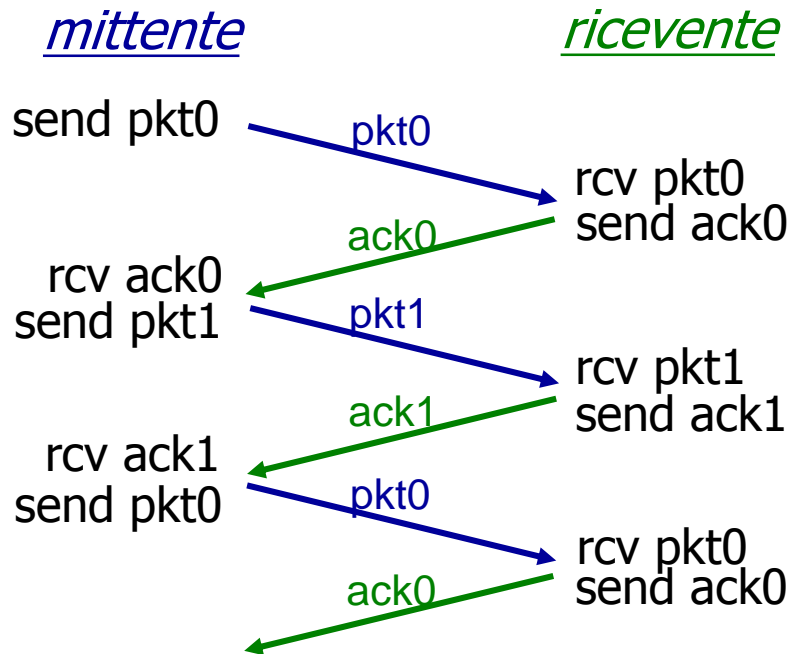
approccio: il mittente attende un ACK per un tempo “ragionevole”

- ❖ ritrasmette se non riceve un ACK in questo periodo
- ❖ se il pacchetto (o l'ACK) è soltanto in ritardo (non perso):
 - la ritrasmissione sarà duplicata, ma l'uso dei numeri di sequenza gestisce già questo
 - il destinatario deve specificare il numero di sequenza del pacchetto riscontrato
- ❖ occorre un contatore (countdown timer)

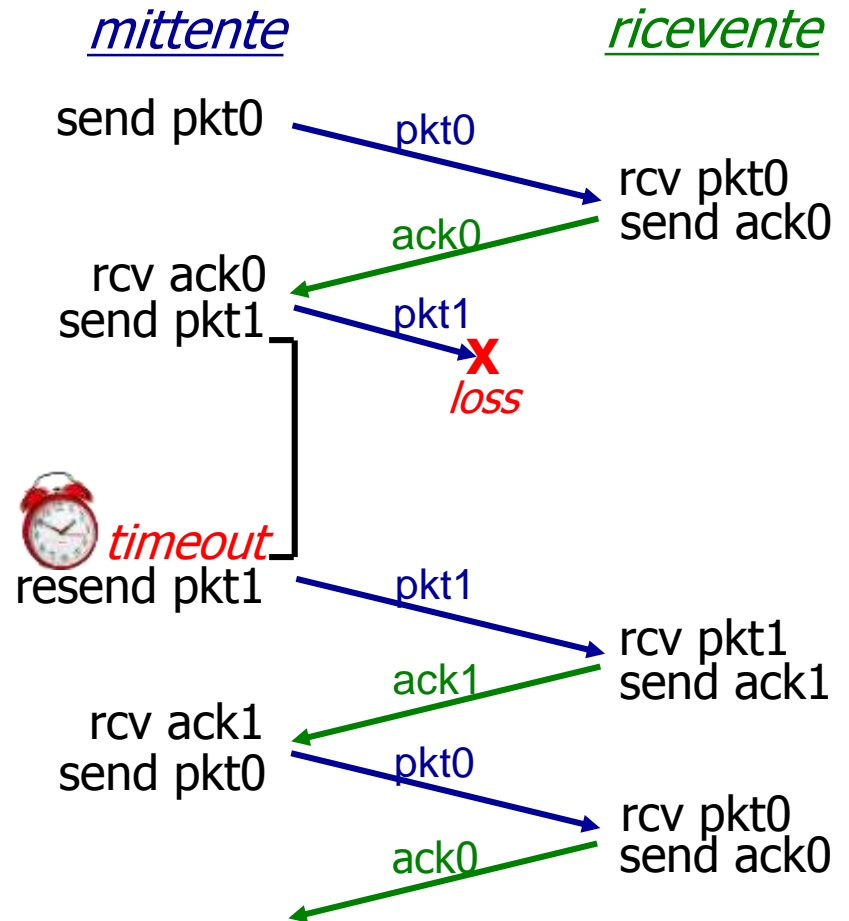
rdt3.0 mittente



rdt3.0 in azione



(a) nessuna perdita

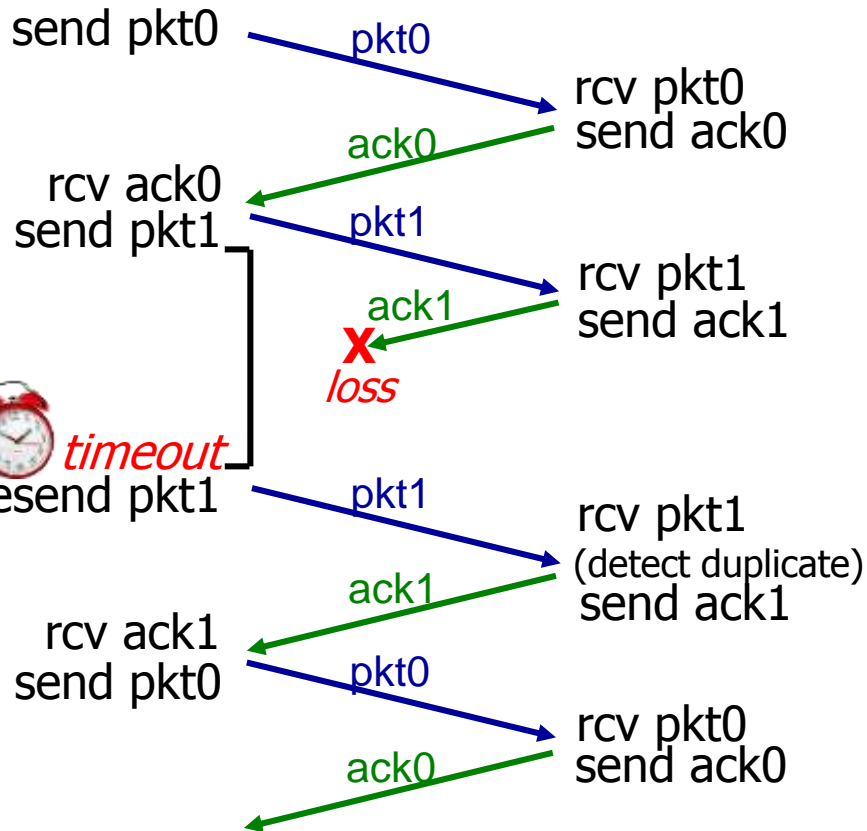


(b) perdita di pacchetto

rdt3.0 in azione

mittente

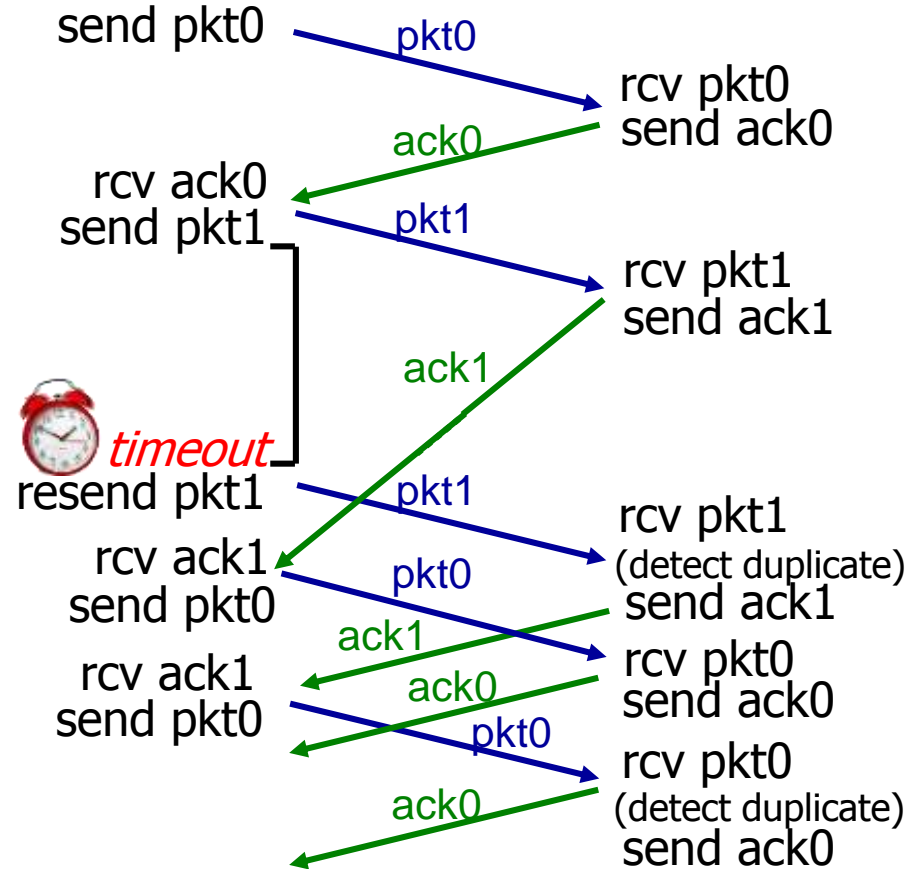
ricevente



(c) perdita di ACK

mittente

ricevente



(d) timeout prematuro per ACK in ritardo

Prestazioni di rdt3.0

- ❖ rdt3.0 funziona, ma le prestazioni non sono apprezzabili
- ❖ es.: link a 1 Gbps, 15 ms di ritardo di propagazione, 8000 pacchetti da 1 bit:

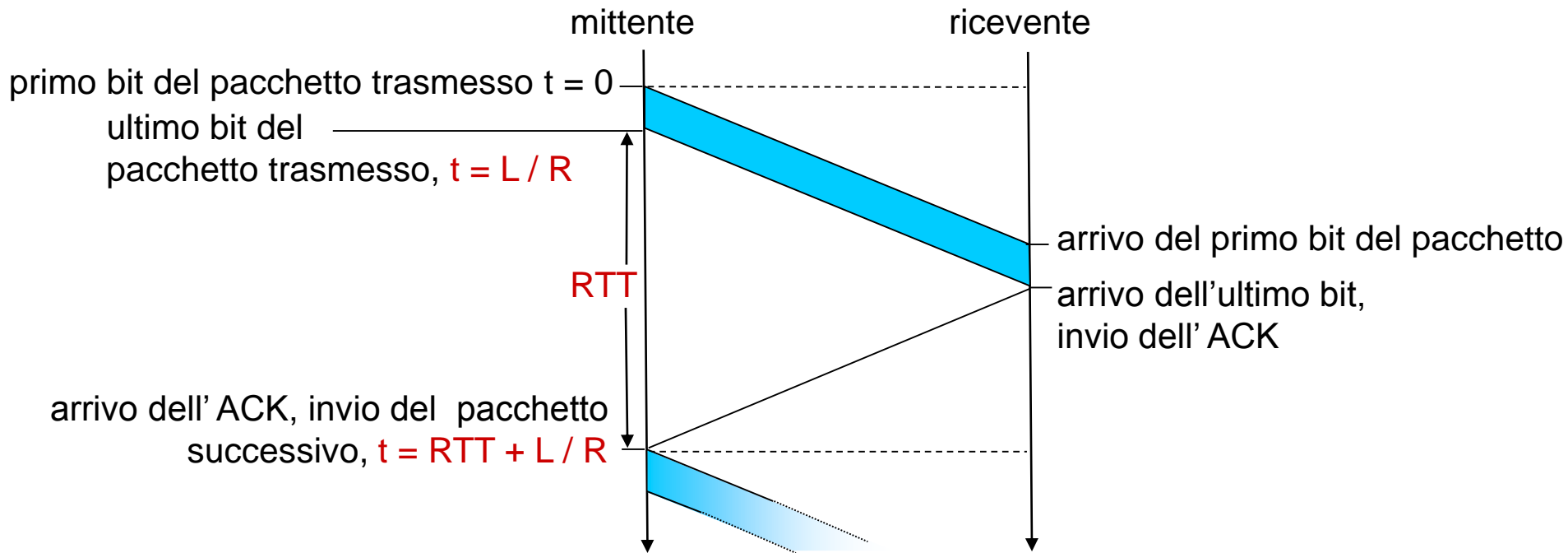
$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ bits}}{10^9 \text{ bits/sec}} = 8 \text{ microsecs}$$

- U_{sender} : **utilizzo** – la frazione di tempo in cui il mittente è occupato nell'invio

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

- se $RTT=30$ msec, 1 pacchetto da 1 KB 30 msec: throughput di 33 kB/sec in un link da 1 Gbps
- ❖ il protocollo di rete limita l'uso delle risorse fisiche!

rdt3.0: funzionamento con stop-and-wait

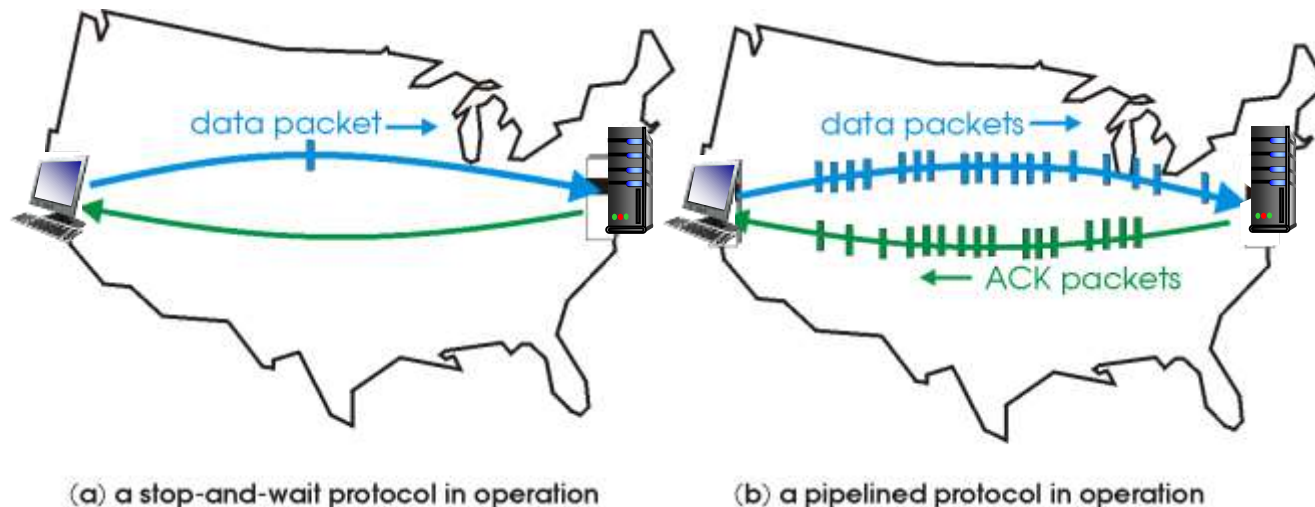


$$U_{\text{sender}} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

Protocolli con pipeline

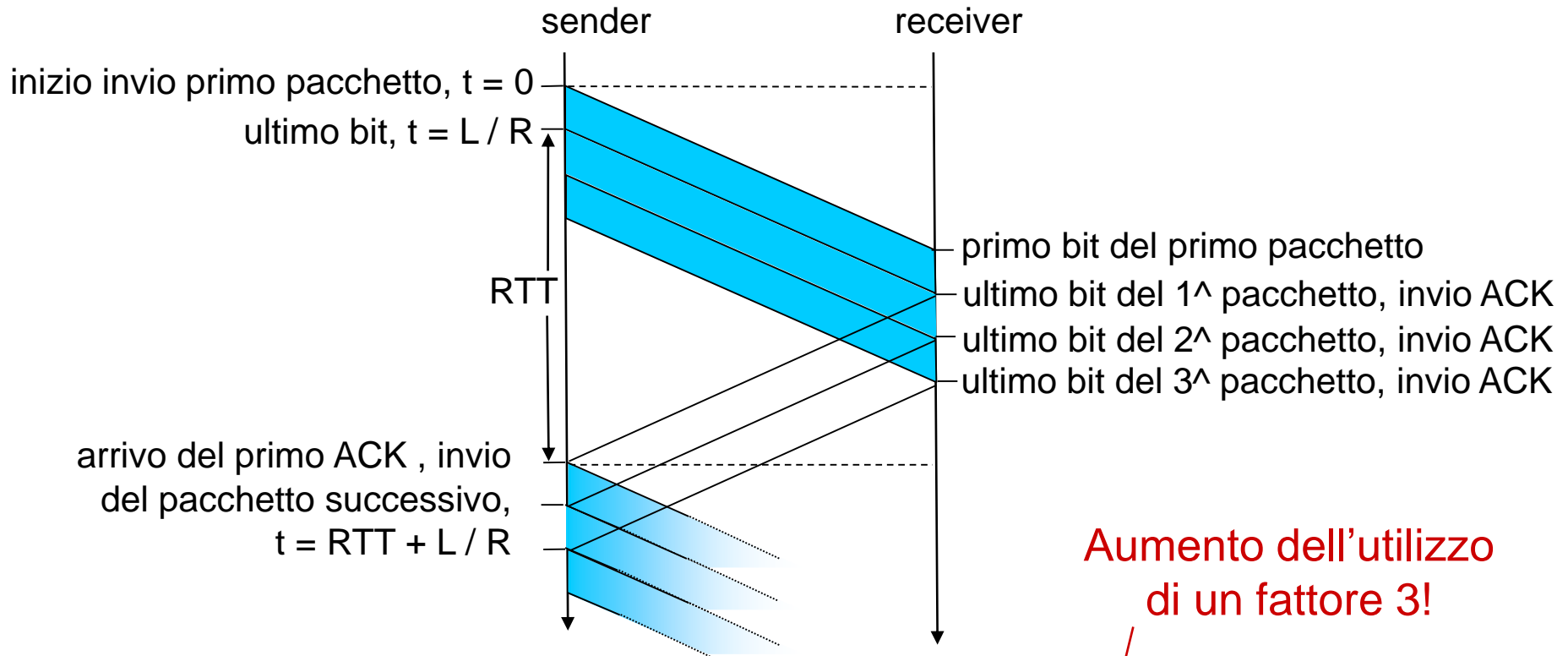
pipelining: il mittente ammette più pacchetti in transito, ancora da notificare

- l'intervallo dei numeri di sequenza deve essere incrementato
- buffering dei pacchetti presso il mittente e/o il ricevente



- ❖ due forme generiche di protocolli con pipeline:
go-Back-N, ripetizione selettiva

Pipelining: aumento dell'utilizzo



$$U_{\text{sender}} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Aumento dell'utilizzo
 di un fattore 3!

Protocolli con pipeline: panoramica

Go-back-N:

- ❖ il mittente può avere fino a N pacchetti consecutivi non riscontrati in pipeline
- ❖ il ricevente invia solo *ack cumulativi*
 - non dà riscontro a un pacchetto se c'è un 'buco'
- ❖ il mittente ha un timer per il più vecchio pacchetto non riscontrato
 - quando il timer scade, ritrasmette *tutti* i pacchetti non riscontrati

Ripetizione selettiva:

- ❖ il mittente può avere fino a N pacchetti consecutivi non riscontrati in pipeline
- ❖ il ricevente invia *ack individuali* per ogni pacchetto
- ❖ il mittente ha un timer per ogni pacchetto non riscontrato
 - quando un timer scade, ritrasmette solo il pacchetto non riscontrato

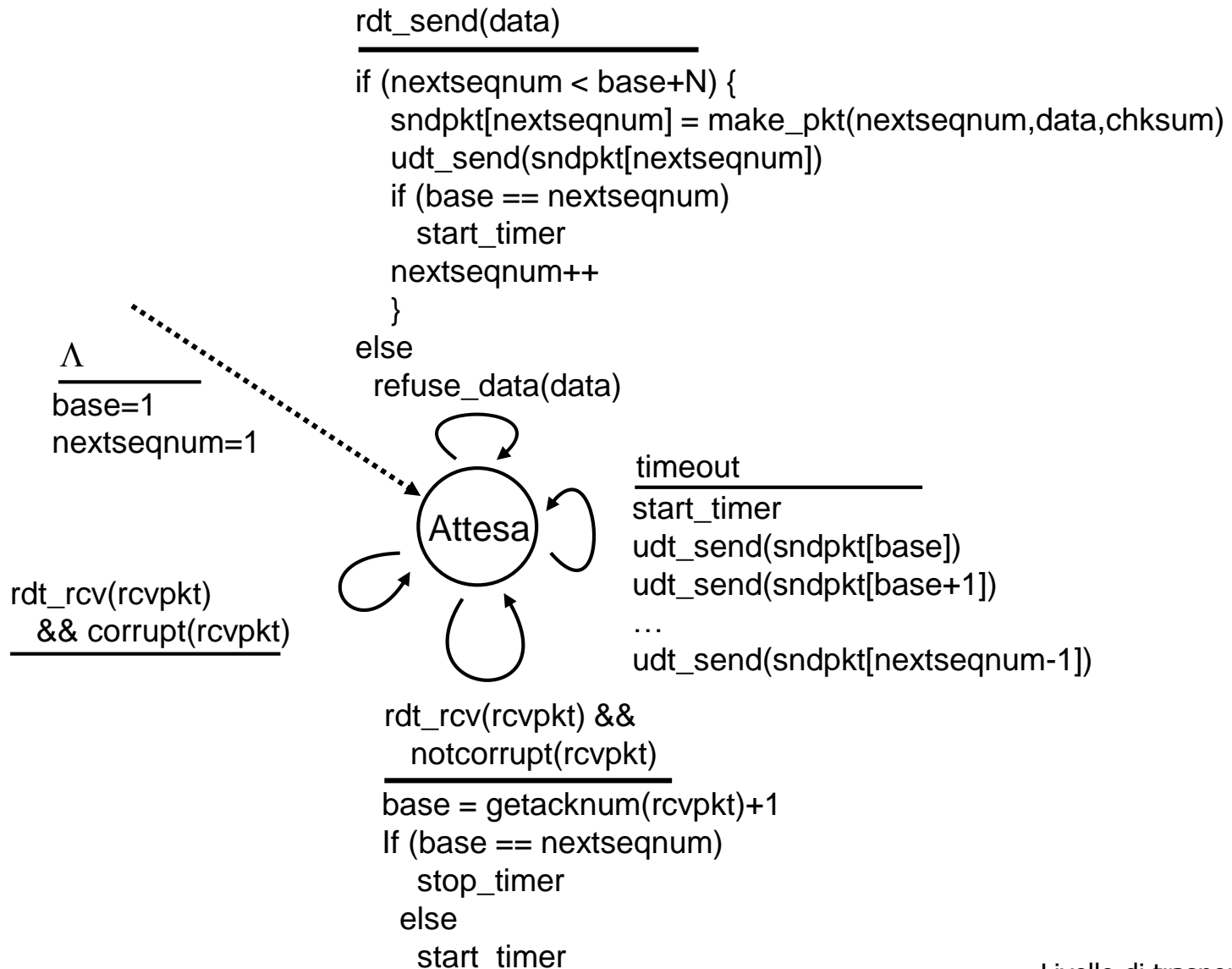
Go-Back-N: mittente

- ❖ numero di sequenza a k bit nell'intestazione del pacchetto
- ❖ “finestra” contenente fino a N pacchetti consecutivi non riscontrati

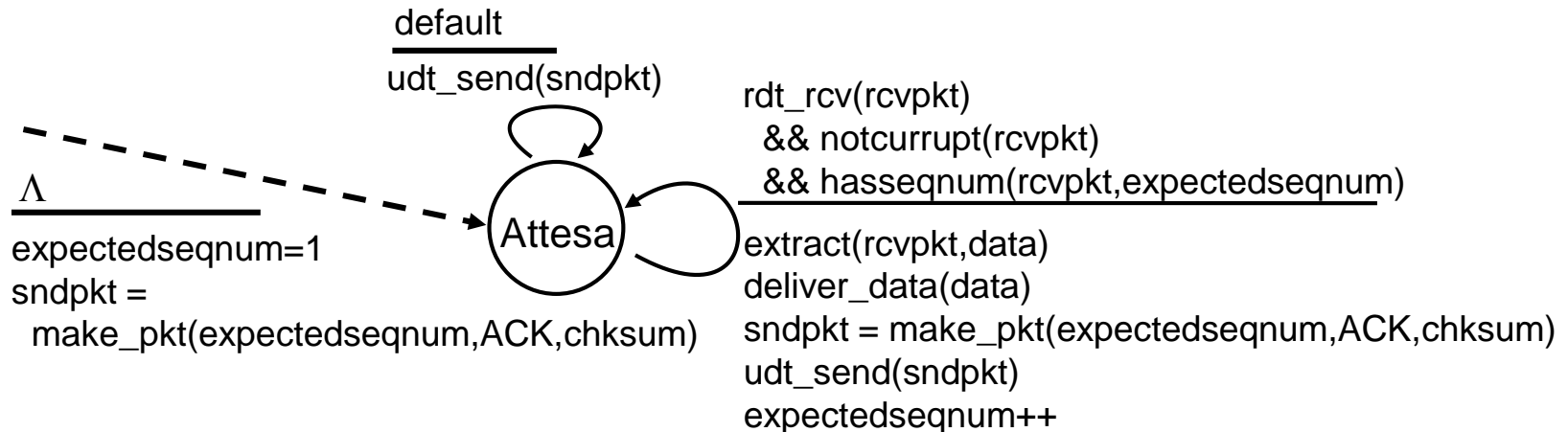


- ❖ ACK(n): riscontro di tutti i pacchetti con numero di sequenza minore o uguale a n - *“ACK cumulativi”*
 - può ricevere ACK duplicati (vedere il ricevente)
- ❖ timer per il primo pacchetto in transito
- ❖ *timeout(n)*: ritrasmette il pacchetto n e tutti i pacchetti con numero di sequenza più grande nella finestra

GBN: FSM esteso del mittente



GBN: FSM esteso del ricevente



ACK-only: invia sempre un ACK per un pacchetto ricevuto correttamente con il numero di sequenza più alto *in sequenza*

- può generare ACK duplicati
 - deve memorizzare soltanto **expectedseqnum**
- ❖ pacchetto fuori sequenza:
- scartato (non salvato): *non c'è buffering da parte del ricevente!*
 - rimanda un ACK per il pacchetto con il numero di sequenza più alto in sequenza

GBN in azione

finestra d'invio (N=4)

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8
0 1 2 3 4 5 6 7 8

mittente

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

rcv ack0, send pkt4
rcv ack1, send pkt5

ignora ACK duplicato



timeout pkt 2

send pkt2
send pkt3
send pkt4
send pkt5

ricevente

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1

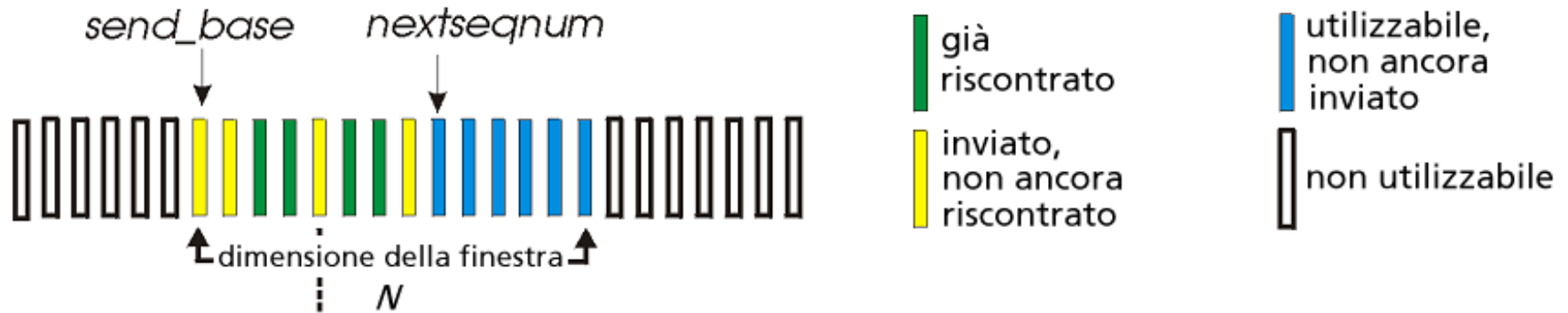
receive pkt5, discard,
(re)send ack1

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

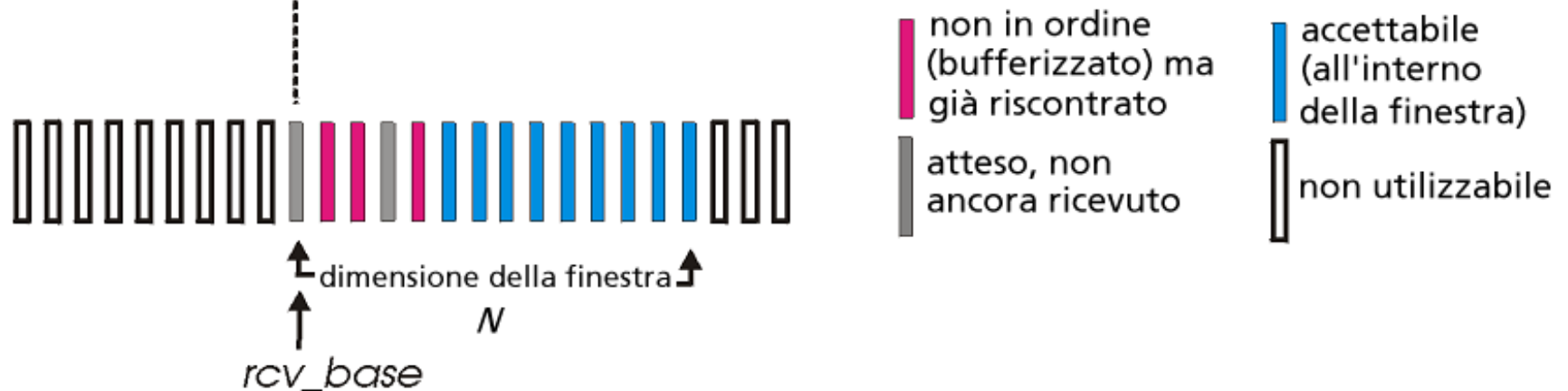
Ripetizione selettiva

- ❖ il ricevente invia riscontri *specifici* per tutti i pacchetti ricevuti correttamente
 - bufferizza i pacchetti, se necessario, per eventuali consegne in sequenza al livello superiore
- ❖ il mittente ritrasmette soltanto i pacchetti per i quali non ha ricevuto un ACK
 - timer del mittente per ogni pacchetto non riscontrato
- ❖ finestra di invio
 - N numeri di sequenza consecutivi
 - limita i numeri di sequenza dei pacchetti inviati non riscontrati

Ripetizione selettiva: finestre di invio e ricezione



a) Visione del mittente sui numeri di sequenza



b) Visione del ricevente sui numeri di sequenza

Ripetizione selettiva

mittente

dati dall'alto:

- ❖ se il successivo numero di sequenza disponibile è nella finestra, invia il pacchetto

timeout(n):

- ❖ ritrasmette il pacchetto n, riparte il timer

ACK(n) in [sendbase, sendbase+N]:

- ❖ marca il pacchetto n come ricevuto
- ❖ se n è il numero di sequenza non riscontrato più piccolo, avanza la base della finestra al successivo numero di sequenza non riscontrato

ricevente

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

- ❖ invia ACK(n)
- ❖ fuori sequenza: buffer
- ❖ in sequenza: consegna (vengono consegnati anche i pacchetti bufferizzati in sequenza); la finestra avanza al successivo pacchetto non ancora ricevuto

pacchetto n in [rcvbase N, rcvbase-1]

- ❖ ACK(n)

altrimenti: ignora

Ripetizione selettiva in azione

finestra d'invio (N=4)

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

0 1 2 3 4 5 6 7 8

mittente

send pkt0

send pkt1

send pkt2

send pkt3

(wait)

rcv ack0, send pkt4

rcv ack1, send pkt5

record ack3 arrived



timeout pkt 2

send pkt2

record ack4 arrived

record ack5 arrived

ricevente

receive pkt0, send ack0

receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4

receive pkt5, buffer,
send ack5

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

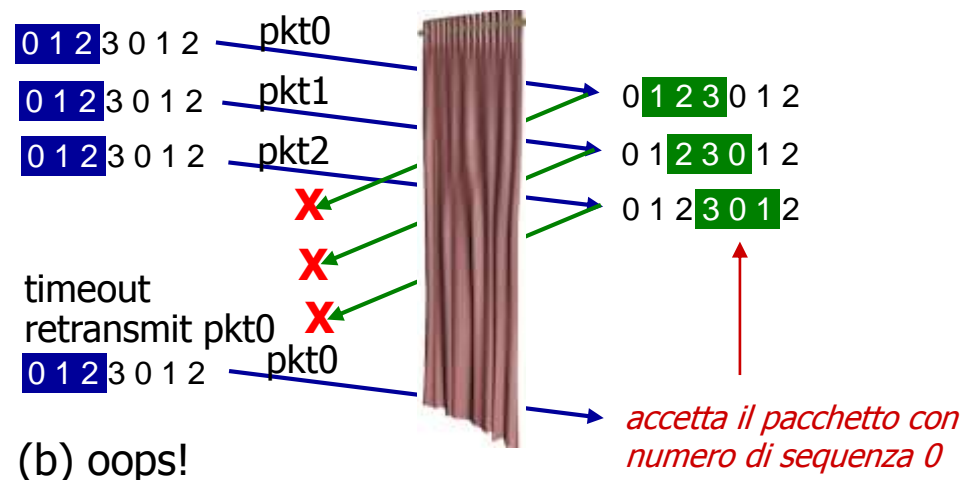
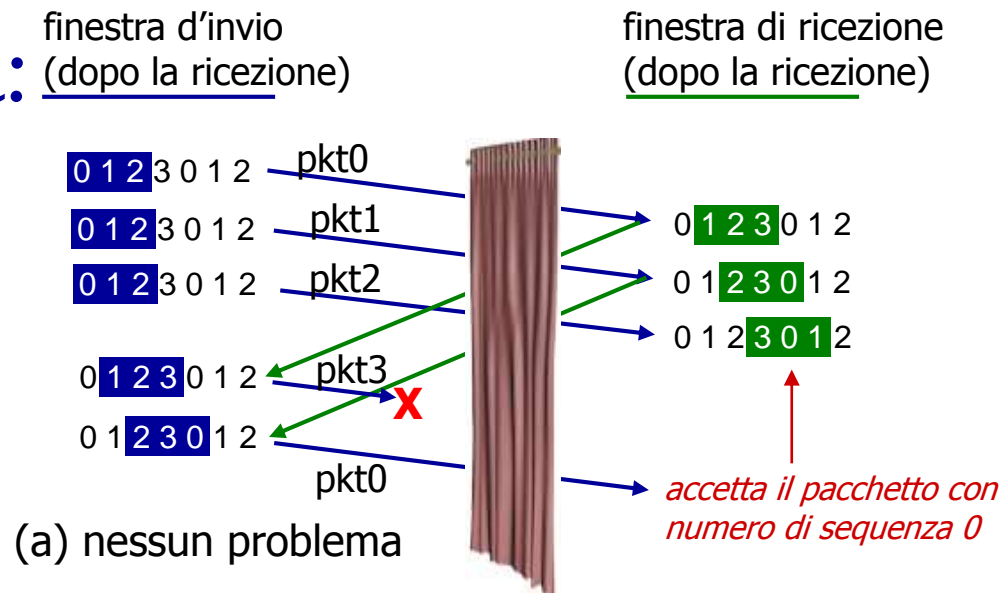
D: che succede quando arriva ack2 ?

Ripetizione selettiva: dilemma

esempio:

- ❖ numeri di sequenza: 0, 1, 2, 3
- ❖ dimensione finestra=3
- ❖ il ricevente non vede alcuna differenza fra i due scenari!
- ❖ dati duplicati accettati come nuovi in (b)

D: quale relazione deve esserci fra lo spazio dei numeri di sequenza e la dimensione della finestra per evitare i problemi in (b)?



Capitolo 3: Livello di trasporto

- 3.1 servizi a livello di trasporto
- 3.2 multiplexing e demultiplexing
- 3.3 trasporto senza connessione: UDP
- 3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

TCP: Panoramica

RFCs: 793, 1122, 1323, 2018, 2581

❖ punto-punto:

- un mittente, un destinatario

❖ *flusso di byte affidabile, in sequenza:*

- nessuna “limitazione ai messaggi”

❖ pipeline:

- controllo di congestione e controllo di flusso

❖ full duplex data:

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

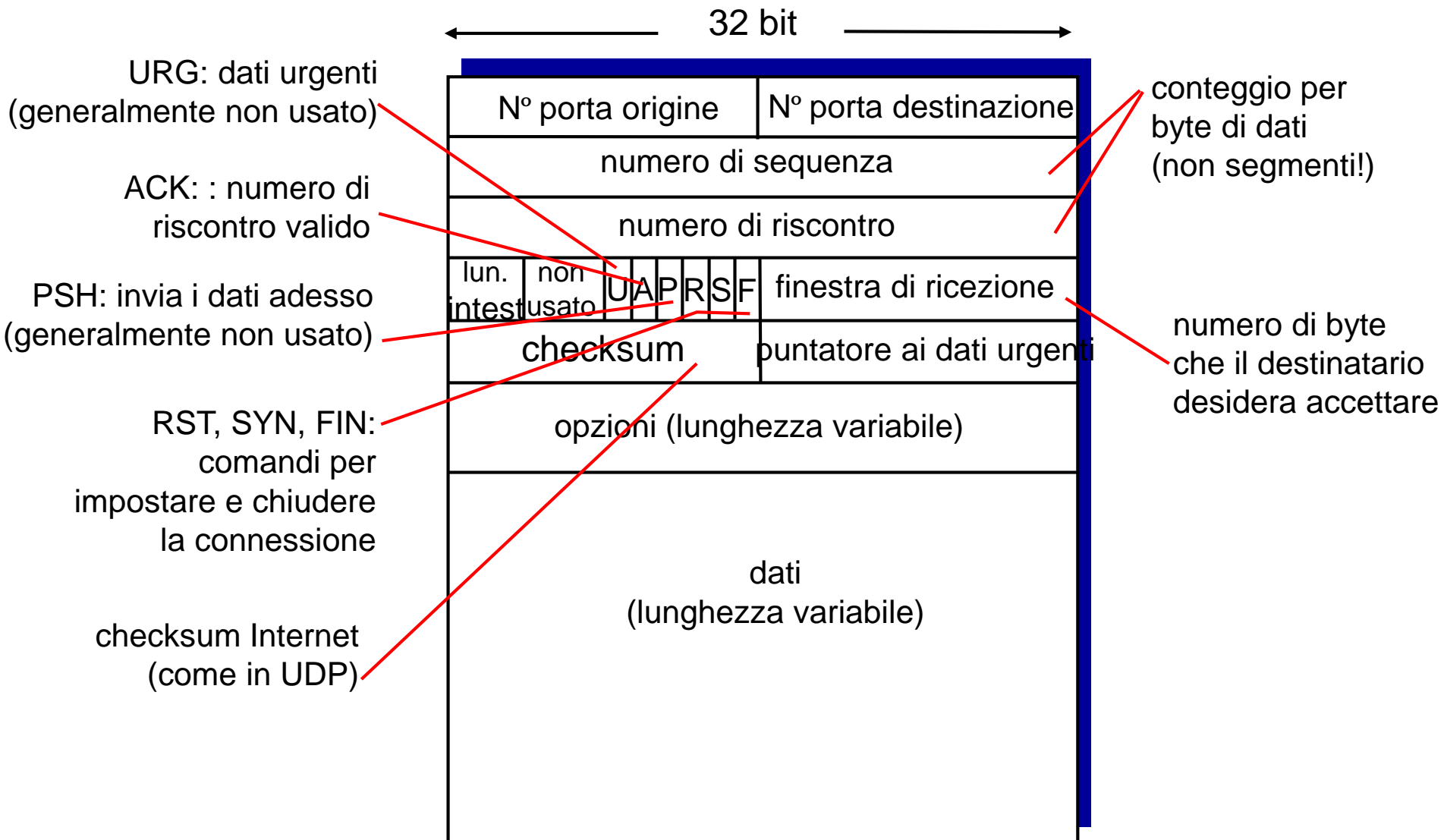
❖ orientato alla connessione:

- 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

Struttura dei segmenti TCP



Numeri di sequenza e ACK di TCP

numeri di sequenza:

- “numero” del primo byte del segmento nel flusso di byte

acknowledgement:

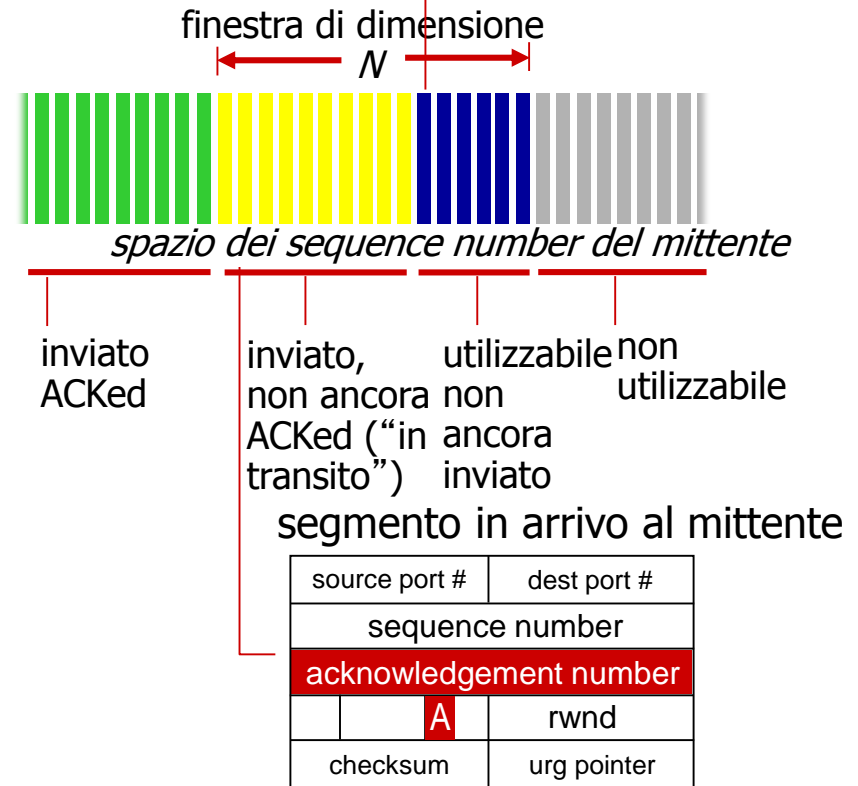
- numero di sequenza del successivo byte atteso dall'altro lato
- ACK cumulativi

D: come gestisce il destinatario i segmenti fuori sequenza?

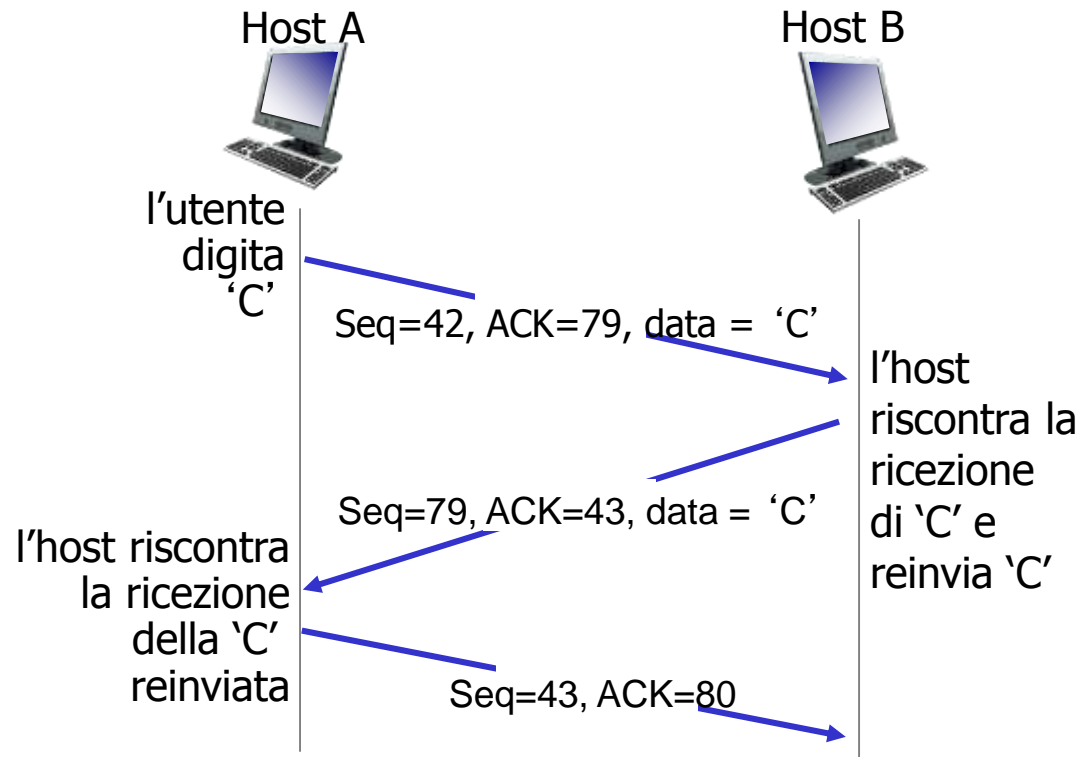
- R: la specifica TCP non lo dice – dipende dall'implementatore

segmento in uscita dal mittente

source port #		dest port #	
sequence number			
acknowledgement number			
			rwnd
checksum		urg pointer	



Numeri di sequenza e ACK di TCP



semplice scenario telnet

TCP round trip time, timeout

D: come impostare il valore del timeout di TCP?

- ❖ più grande di RTT
 - ma RTT varia
- ❖ *troppo breve*: timeout prematuri, ritrasmissioni non necessarie
- ❖ *troppo lungo*: reazione lenta alla perdita dei segmenti

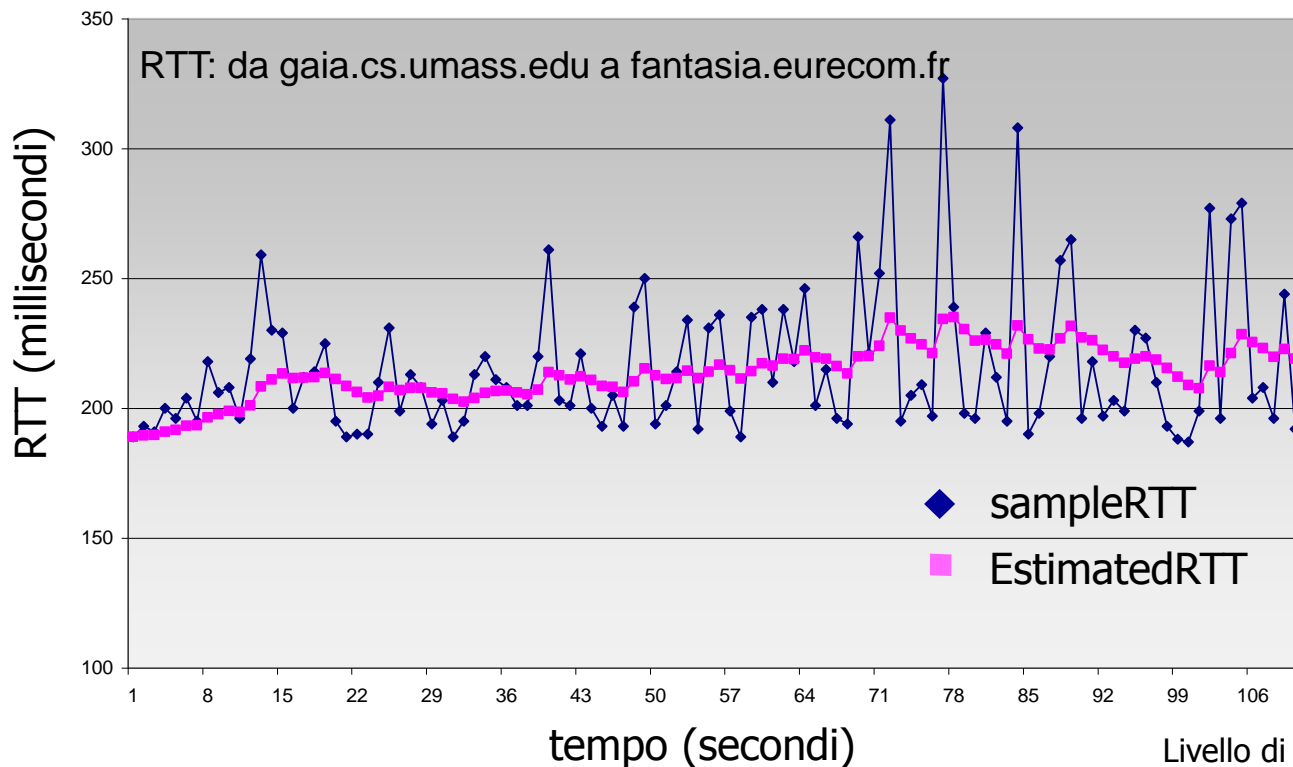
D: come stimare RTT?

- ❖ **SampleRTT**: tempo misurato dalla trasmissione del segmento fino alla ricezione di ACK
 - ignora le ritrasmissioni
- ❖ **SampleRTT** varia, quindi occorre una stima “più morbida” di RTT
 - media di più misure *recenti*, non semplicemente il valore corrente di **SampleRTT**

TCP round trip time, timeout

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

- ❖ media mobile esponenziale ponderata
- ❖ l'influenza dei vecchi campioni decresce esponenzialmente
- ❖ valore tipico: $\alpha = 0.125$



TCP round trip time, timeout

- ❖ intervallo di timeout : `EstimatedRTT` più un “margine di sicurezza”
 - grande variazione di `EstimatedRTT` → margine di sicurezza maggiore
- ❖ viene stimata la deviazione `SampleRTT` da `EstimatedRTT`:
$$\text{DevRTT} = (1-\beta) * \text{DevRTT} + \beta * |\text{SampleRTT} - \text{EstimatedRTT}|$$
(typically, $\beta = 0.25$)

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



↑
RTT stimato

↑
“margine di sicurezza”

Capitolo 3: Livello di trasporto

- 3.1 servizi a livello di trasporto
- 3.2 multiplexing e demultiplexing
- 3.3 trasporto senza connessione: UDP
- 3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- **trasferimento dati affidabile**
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

Trasferimento dati affidabile del TCP

- ❖ TCP crea un servizio di trasferimento dati affidabile sul servizio inaffidabile di IP

- pipeline dei segmenti
- ACK cumulativi
- un solo timer di ritrasmissione

- ❖ le ritrasmissioni avvengono in seguito a:

- eventi di timeout
- ACK duplicati

inizialmente consideriamo un mittente TCP semplificato:

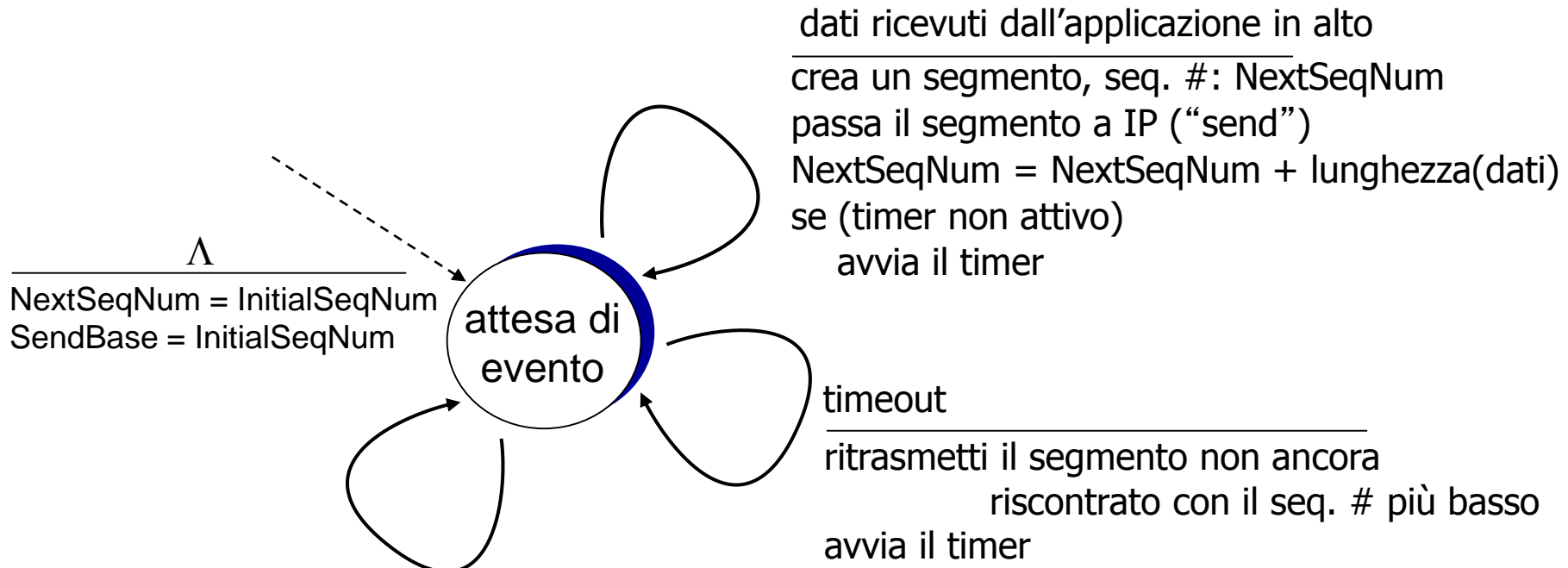
- ignoriamo gli ACK duplicati
- ignoriamo il controllo di flusso e il controllo di congestione

TCP: eventi del mittente

dati ricevuti dall'applicazione: *timeout:*

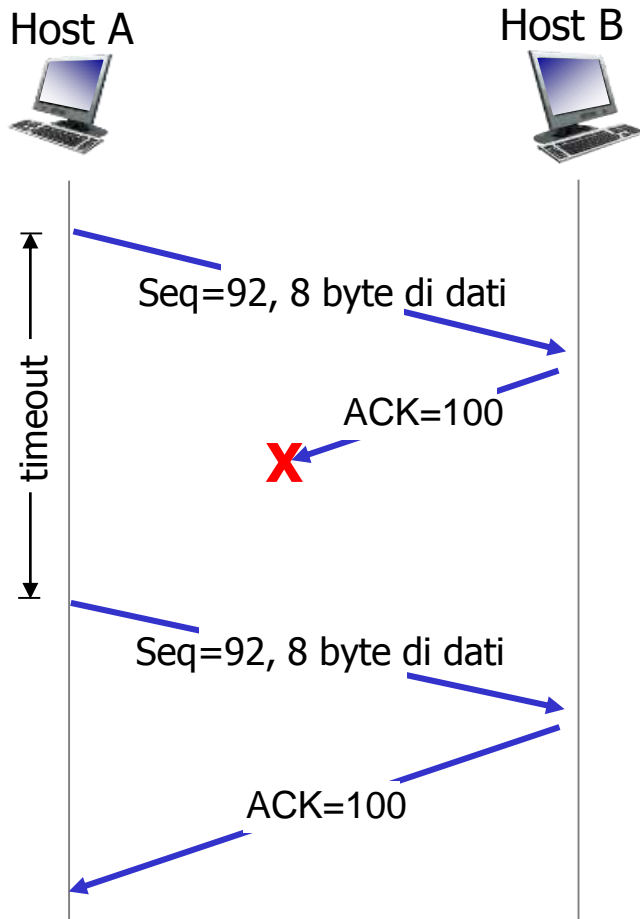
- ❖ crea un segmento con il numero di sequenza
 - ❖ il numero di sequenza è il numero del primo byte del segmento nel flusso di byte
 - ❖ avvia il timer, se non è già in funzione
 - pensate al timer come se fosse associato al più vecchio segmento non riscontrato
 - intervallo di scadenza: **TimeoutInterval**
 - ❖ ritrasmette il segmento che ha causato il timeout
 - ❖ riavvia il timer
- ## *ack ricevuto:*
- ❖ se riscontra segmenti precedentemente non riscontrati
 - aggiorna la traccia su ciò che è stato riscontrato
 - avvia il timer se ci sono altri segmenti da riscontrare

Mittente TCP (semplificato)

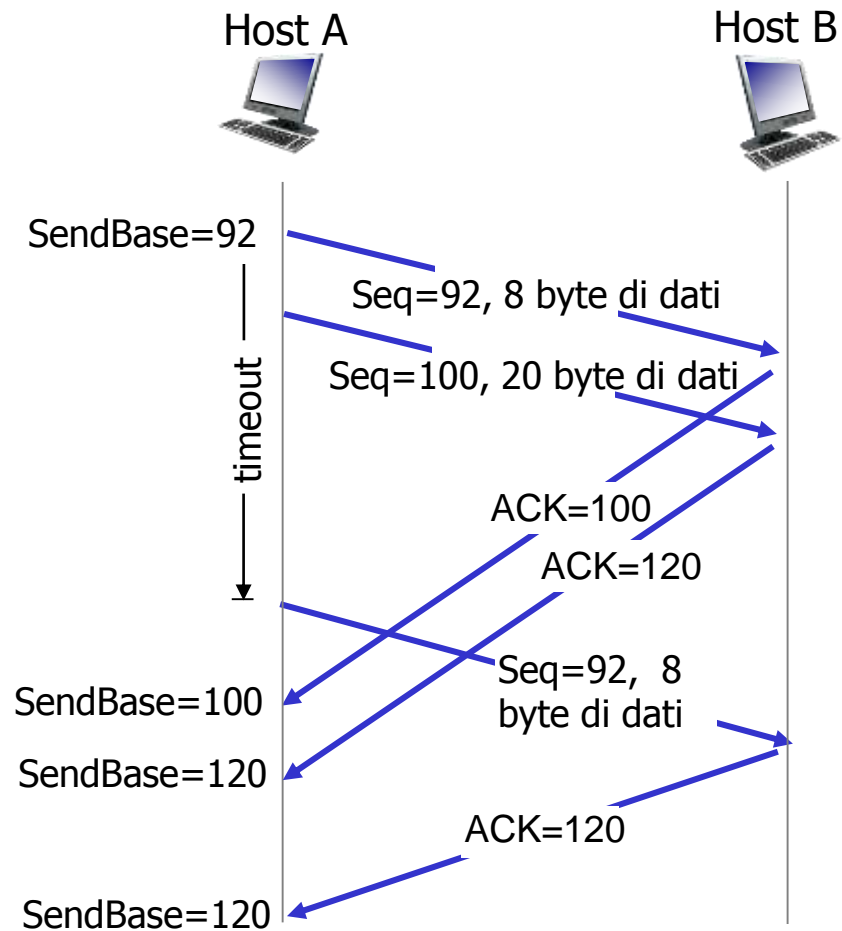


```
se (y > SendBase) {  
    SendBase = y  
    /* SendBase-1: ultimo byte ACKed */  
    se (ci sono segmenti non ancora riscontrati)  
        avvia il timer  
    altrimenti ferma il timer  
}
```

TCP: scenari di ritrasmissione

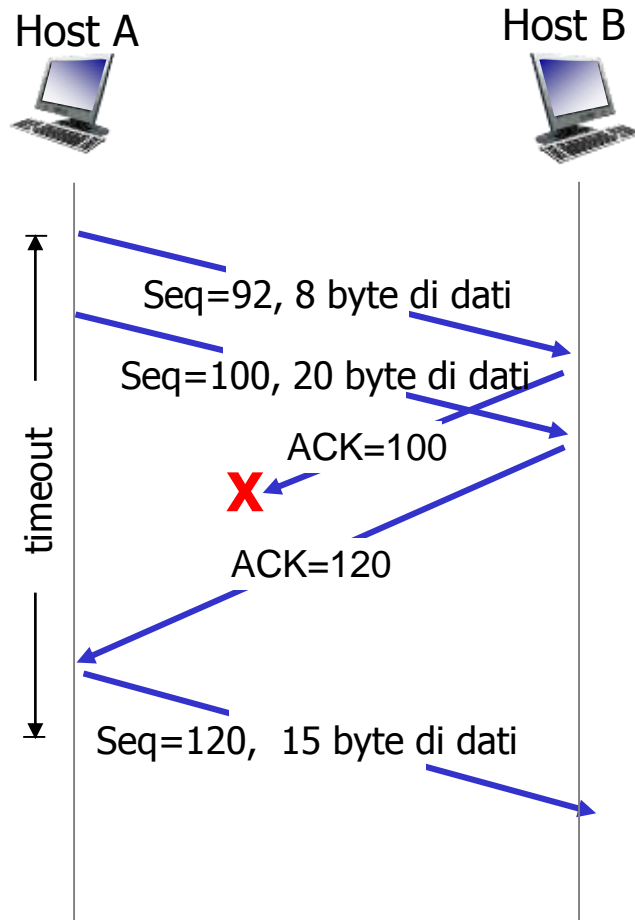


scenario con ACK perso



timeout prematuro

TCP: scenari di ritrasmissione



ACK cumulativo

TCP: generazione di ACK [RFC 1122, RFC 2581]

<i>evento nel destinatario</i>	<i>azione del ricevente TCP</i>
arrivo ordinato di un segmento con numero di sequenza atteso. Tutti i dati fino al numero di sequenza atteso sono già stati riscontrati	ACK ritardato. Attende fino a 500 ms l'arrivo del prossimo segmento. Se il segmento non arriva, invia un ACK
arrivo ordinato di un segmento con numero di sequenza atteso. Per un altro segmento è stato inviato l'ACK ma non ancora riscontrato	invia immediatamente un singolo ACK cumulativo, riscontrando entrambi i segmenti ordinati
arrivo non ordinato di un segmento con numero di sequenza superiore a quello atteso. Viene rilevato un buco	invia immediatamente un <i>ACK duplicato</i> , indicando il numero di sequenza del prossimo byte atteso
arrivo di un segmento che colma parzialmente o completamente il buco	invia immediatamente un ACK, ammesso che il segmento cominci all'estremità inferiore del buco

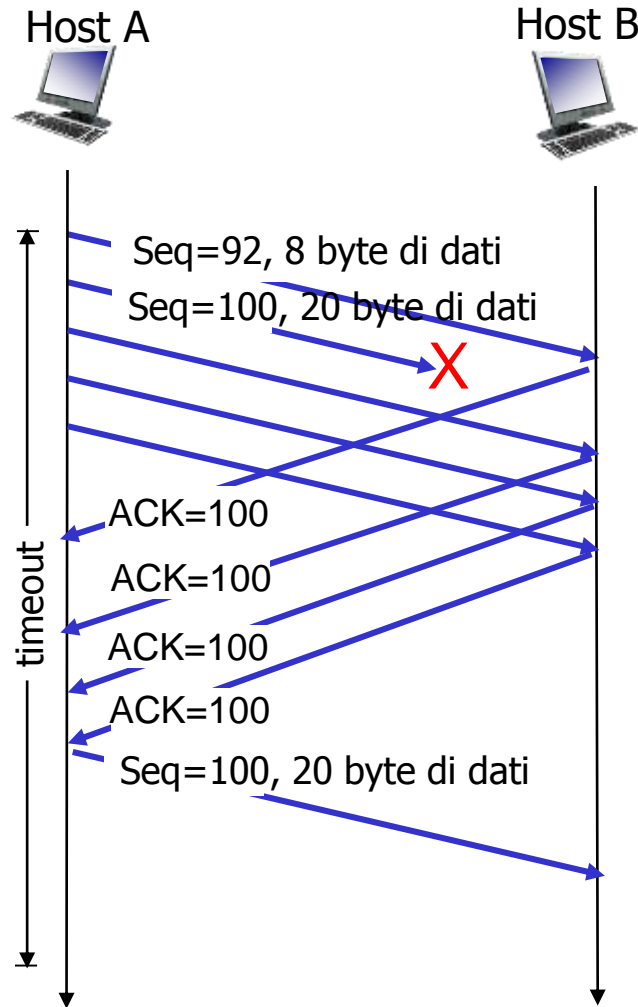
Ritrasmissione rapida

- ❖ il periodo di timeout spesso è relativamente lungo:
 - lungo ritardo prima di ritrasmettere il pacchetto perduto
- ❖ rileva i segmenti perduti tramite gli ACK duplicati.
 - il mittente spesso invia molti segmenti
 - se un segmento viene smarrito, è probabile che ci saranno molti ACK duplicati.

ritrasmissione rapida

- se il mittente riceve 3 ACK per lo stesso dato (“triple duplicate ACKs”), re-invia il segmento non acked con il numero di sequenza più basso
- suppone che il segmento non acked sia andato perso, e non aspetta il timeout

Ritrasmissione rapida



ritrasmissione rapida dopo che il mittente
ha ricevuto il triple duplicate ACK

Capitolo 3: Livello di trasporto

- 3.1 servizi a livello di trasporto
- 3.2 multiplexing e demultiplexing
- 3.3 trasporto senza connessione: UDP
- 3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

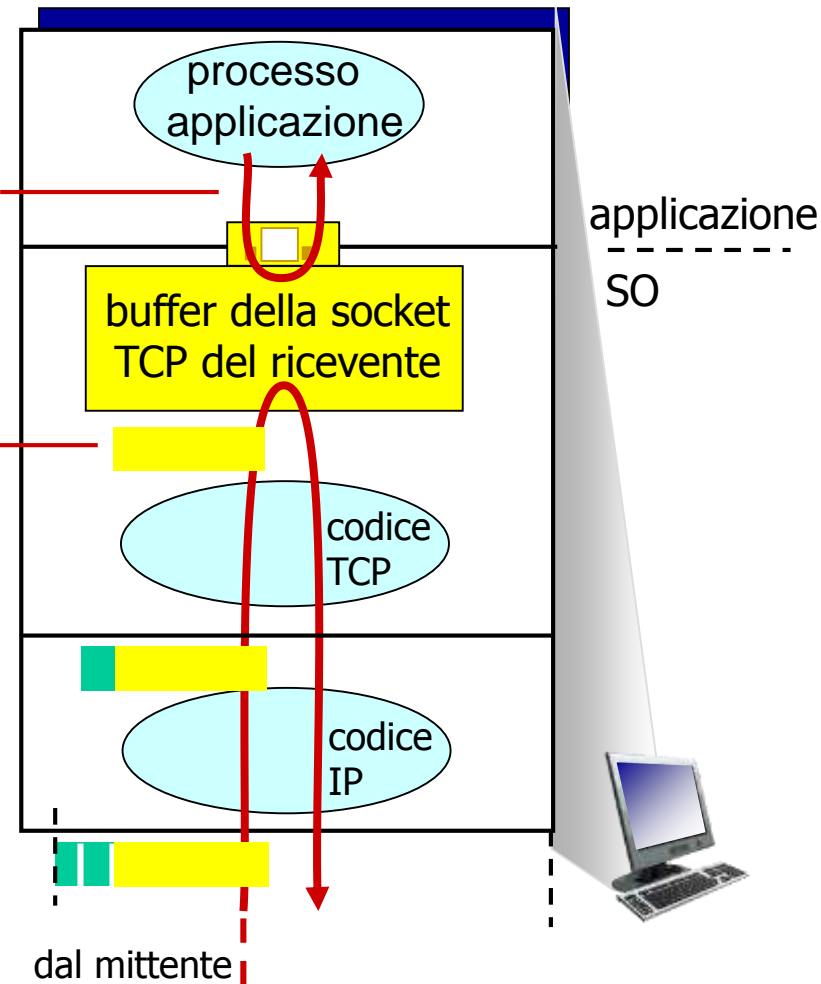
TCP: controllo di flusso

potrebbe accadere che
l'applicazione riesca a
prendere i dati dal
buffer della socket TCP

... più lentamente di
quanto il ricevente TCP
riesca a inviarglieli
(il mittente sta inviando)

controllo di flusso

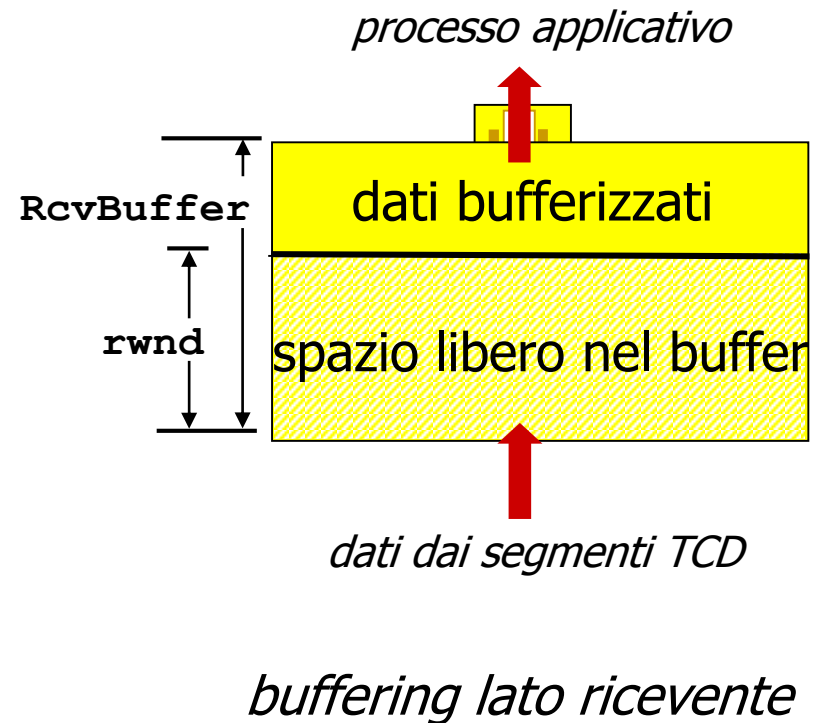
il destinatario modera il mittente,
così da non sovraccaricare il buffer
del destinatario trasmettendo
troppi dati, troppo velocemente



pila protocollare del ricevente

TCP: controllo di flusso

- ❖ il ricevente comunica lo spazio disponibile includendo il valore **rwnd** nelle intestazioni TCP dei suoi segmenti
 - la dimensione di **RcvBuffer** è impostata tramite opzioni della socket (il default tipico è 4096 byte)
 - molti sistemi operativi hanno un autoadjust di **RcvBuffer**
- ❖ il mittente limita la quantità di dati unacked (“in-flight”) al valore **rwnd** del ricevente
- ❖ garantisce che il buffer di ricezione non vada in overflow



Capitolo 3: Livello di trasporto

- 3.1 servizi a livello di trasporto
- 3.2 multiplexing e demultiplexing
- 3.3 trasporto senza connessione: UDP
- 3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

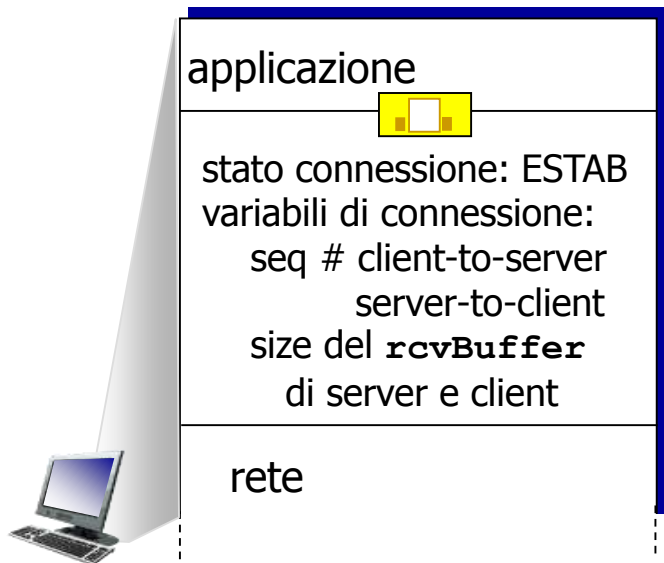
3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

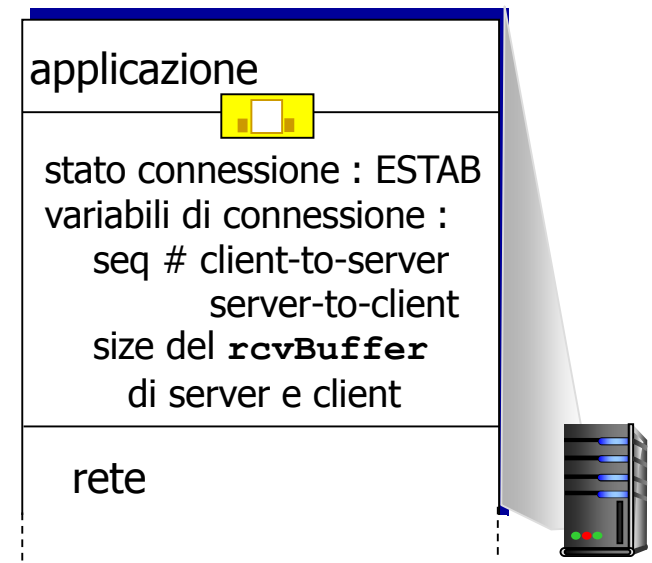
Gestione della connessione

prima di scambiarsi dati, sender e receiver effettuano l'“handshake” (stretta di mano):

- ❖ si accordano nello stabilire la connessione (ognuno sa che l'altro vuole stabilire la connessione)
- ❖ si accordano sui parametri della connessione



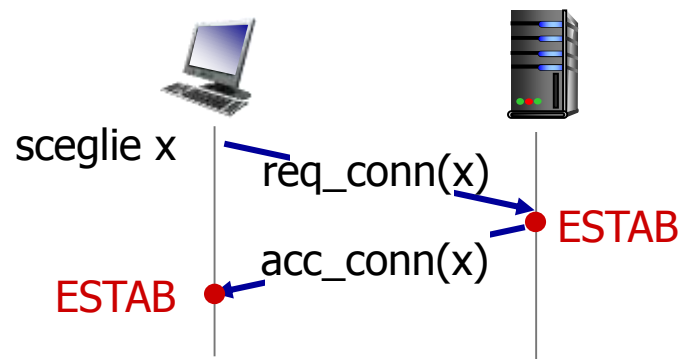
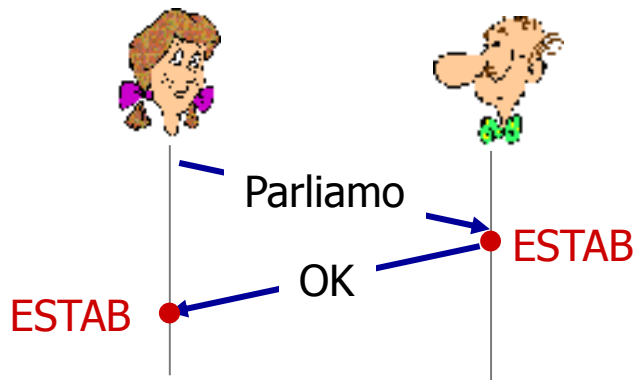
```
clientSocket.connect(  
    (serverName,serverPort))
```



```
connectionSocket, addr =  
    serverSocket.accept()
```

Accordo nello stabilire una connessione

2-way handshake:

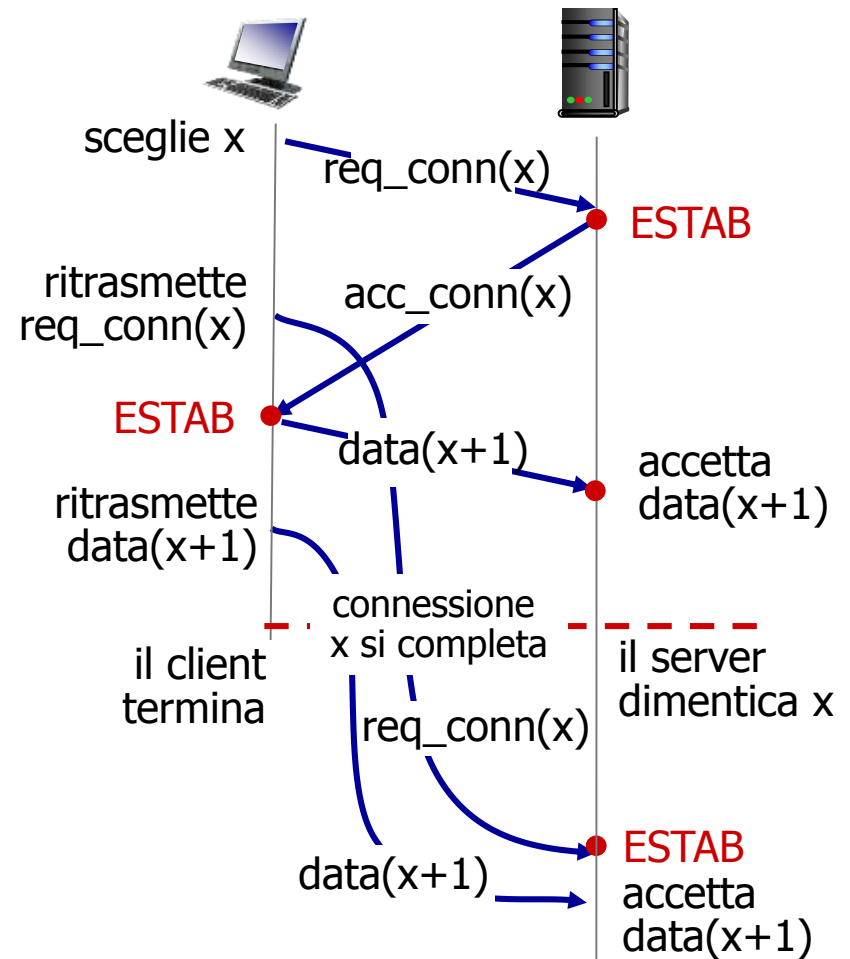
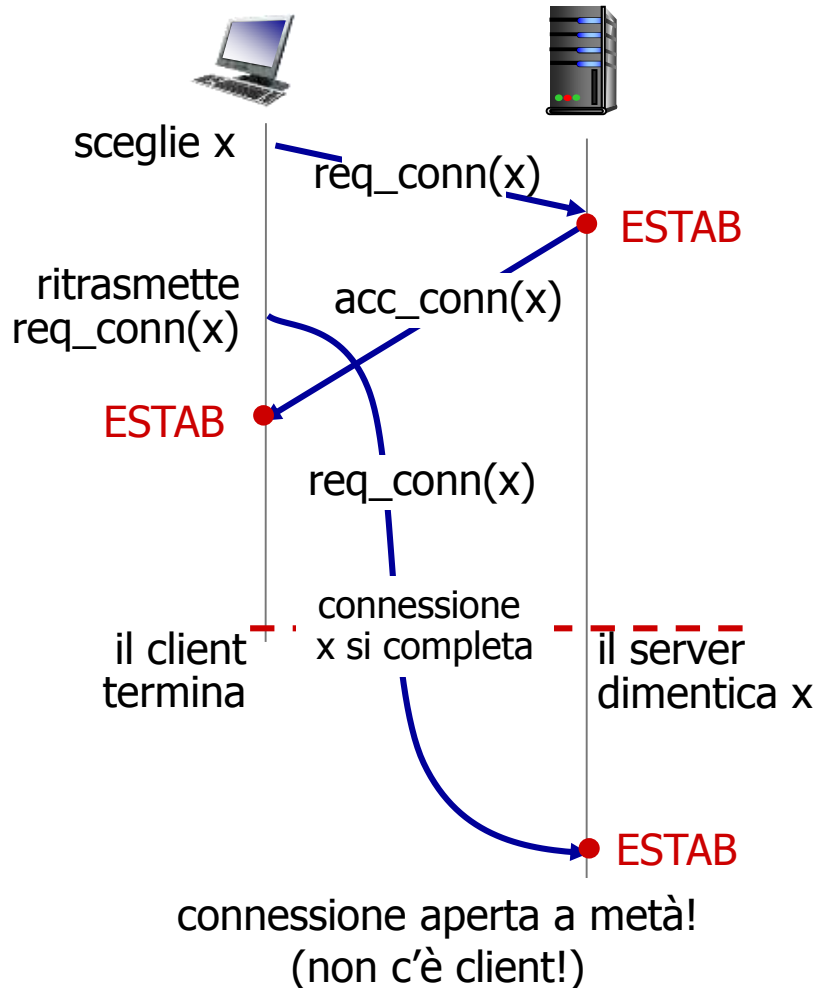


D: l'handshake a 2 vie funziona sempre nella rete?

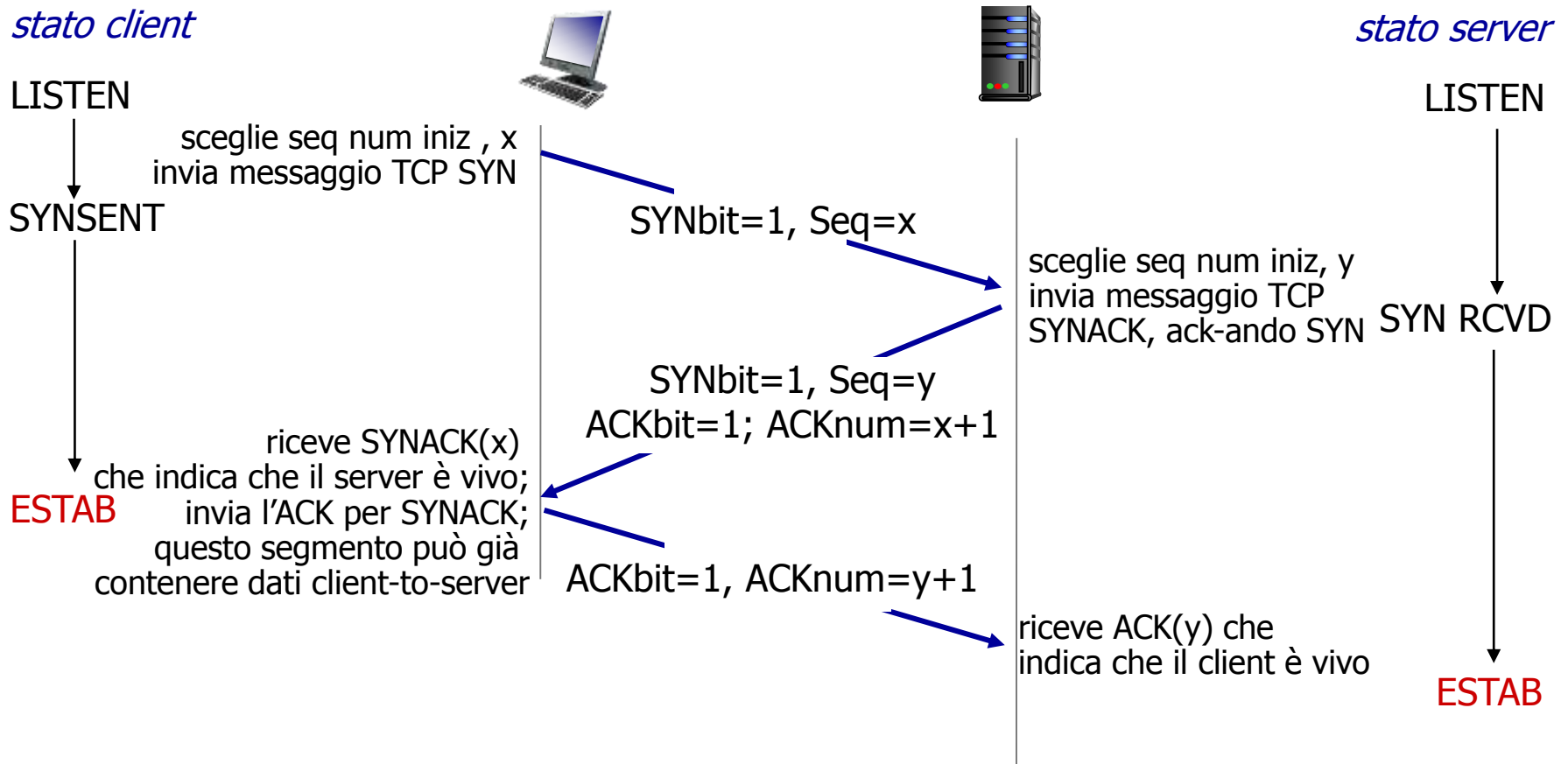
- ❖ ritardi variabili
- ❖ messaggi ritrasmessi (es. `req_conn(x)`) a causa di perdite
- ❖ riordino dei messaggi
- ❖ non si può “vedere” l'altro lato

Accordo nello stabilire una connessione

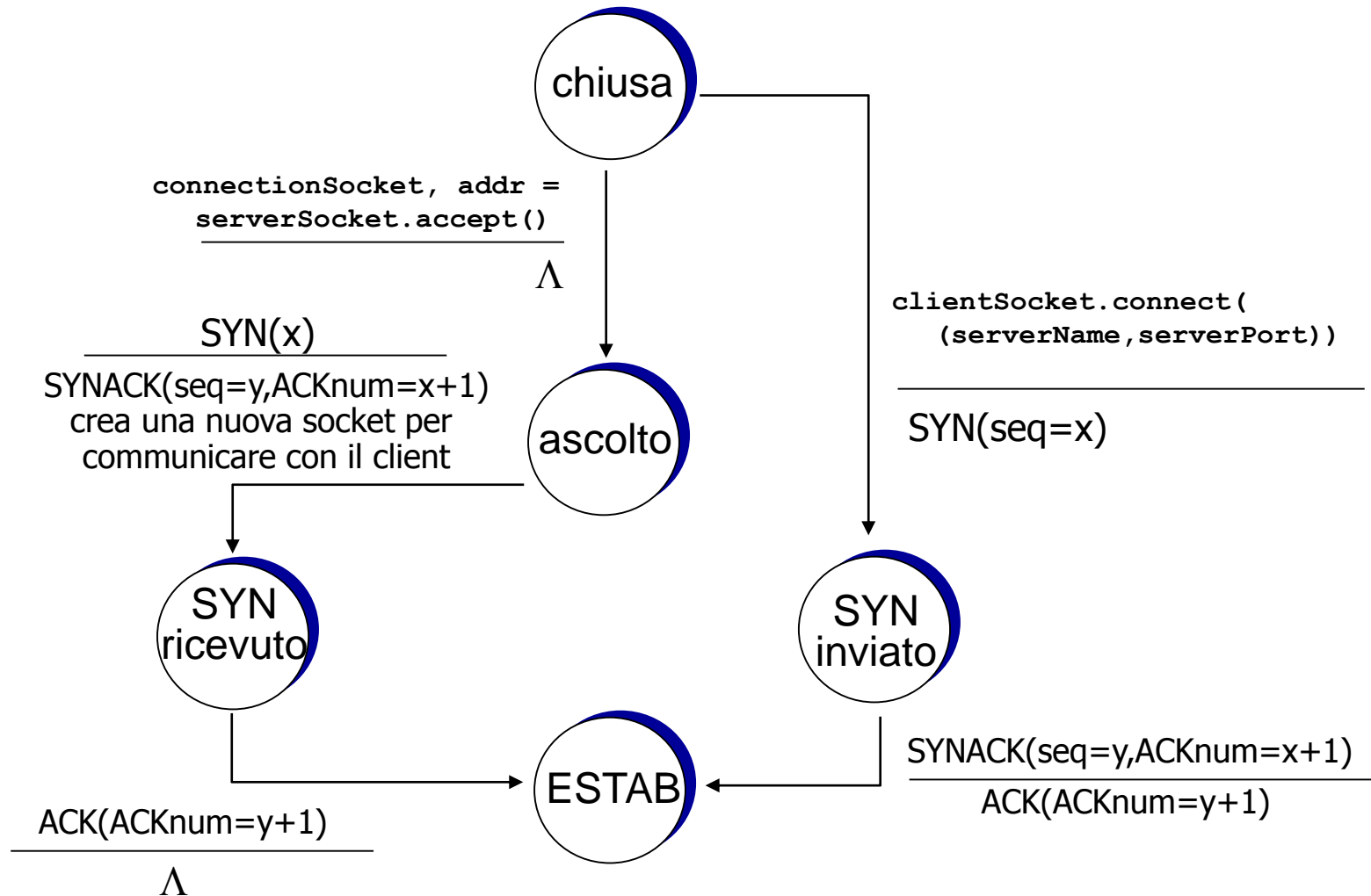
scenari di fallimento del 2-way handshake :



TCP 3-way handshake



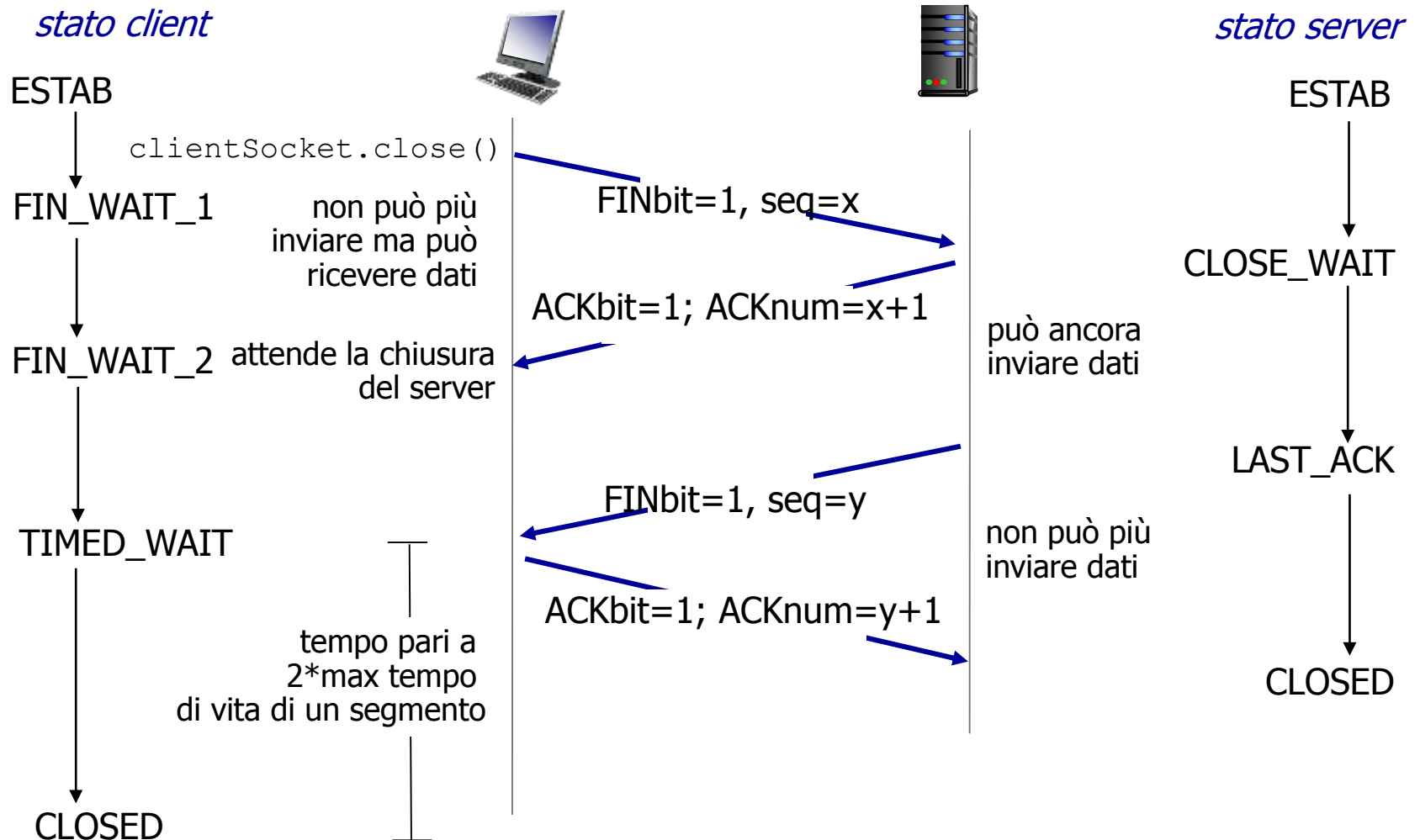
TCP 3-way handshake: FSM



TCP: chiudere una connessione

- ❖ client e server chiudono il loro lato della connessione
 - inviano un segmento TCP con bit FIN = 1
- ❖ rispondono al FIN ricevuto con un ACK
 - alla ricezione del FIN, l'ACK può essere combinato con il proprio FIN
- ❖ uno scambio di FIN simultanei può essere gestito

TCP: chiudere una connessione



Capitolo 3: Livello di trasporto

- 3.1 servizi a livello di trasporto
- 3.2 multiplexing e demultiplexing
- 3.3 trasporto senza connessione: UDP
- 3.4 principi del trasferimento dati affidabile

3.5 trasporto orientato alla connessione : TCP

- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 principi sul controllo di congestione

3.7 controllo di congestione TCP

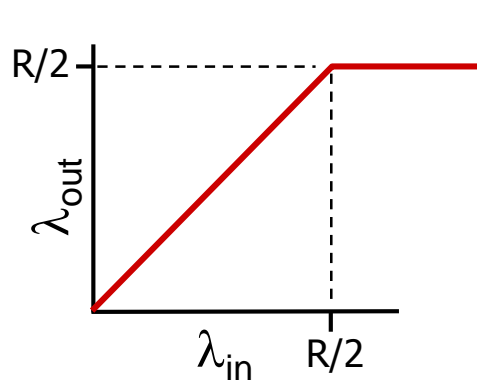
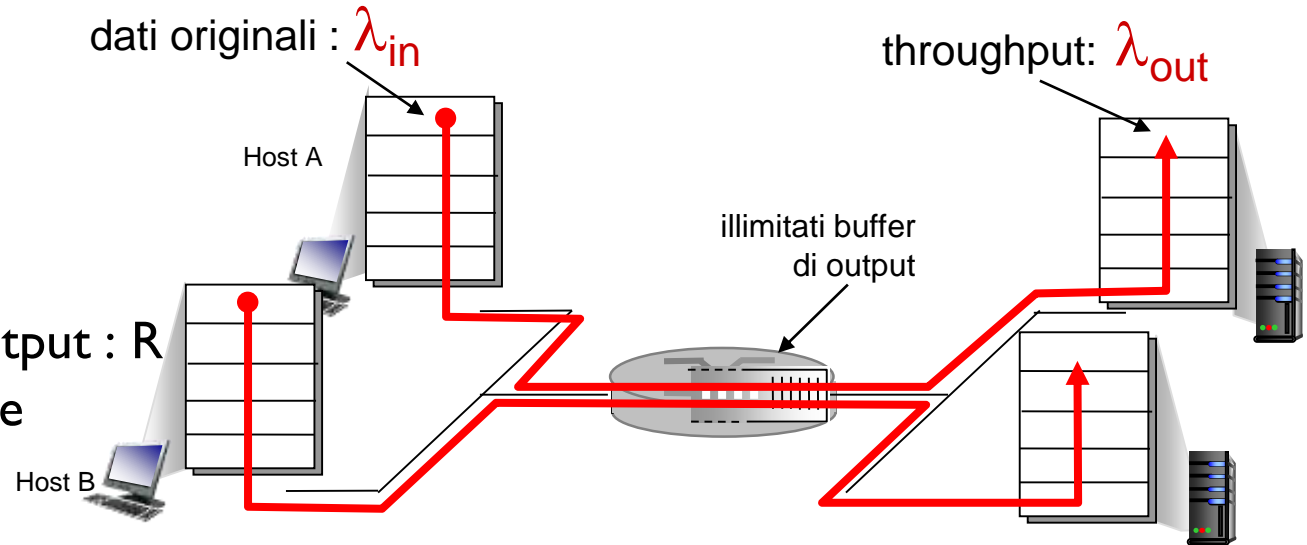
Principi del controllo di congestione

congestione:

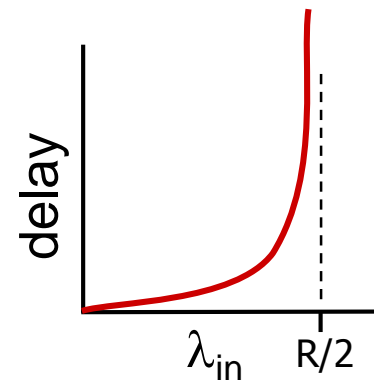
- ❖ informalmente: “troppe sorgenti inviano troppi dati troppo velocemente perché la *rete* li gestisca”
- ❖ differente dal controllo di flusso!
- ❖ sintomi:
 - pacchetti persi (buffer overflow nei router)
 - lunghi ritardi (accodamento nei buffer dei router)
- ❖ problema nella top-10!

Cause/costi della congestione: scenario I

- ❖ due mittenti, due destinatari
- ❖ un router, con buffer illimitati
- ❖ capacità del link di output : R
- ❖ nessuna ritrasmissione



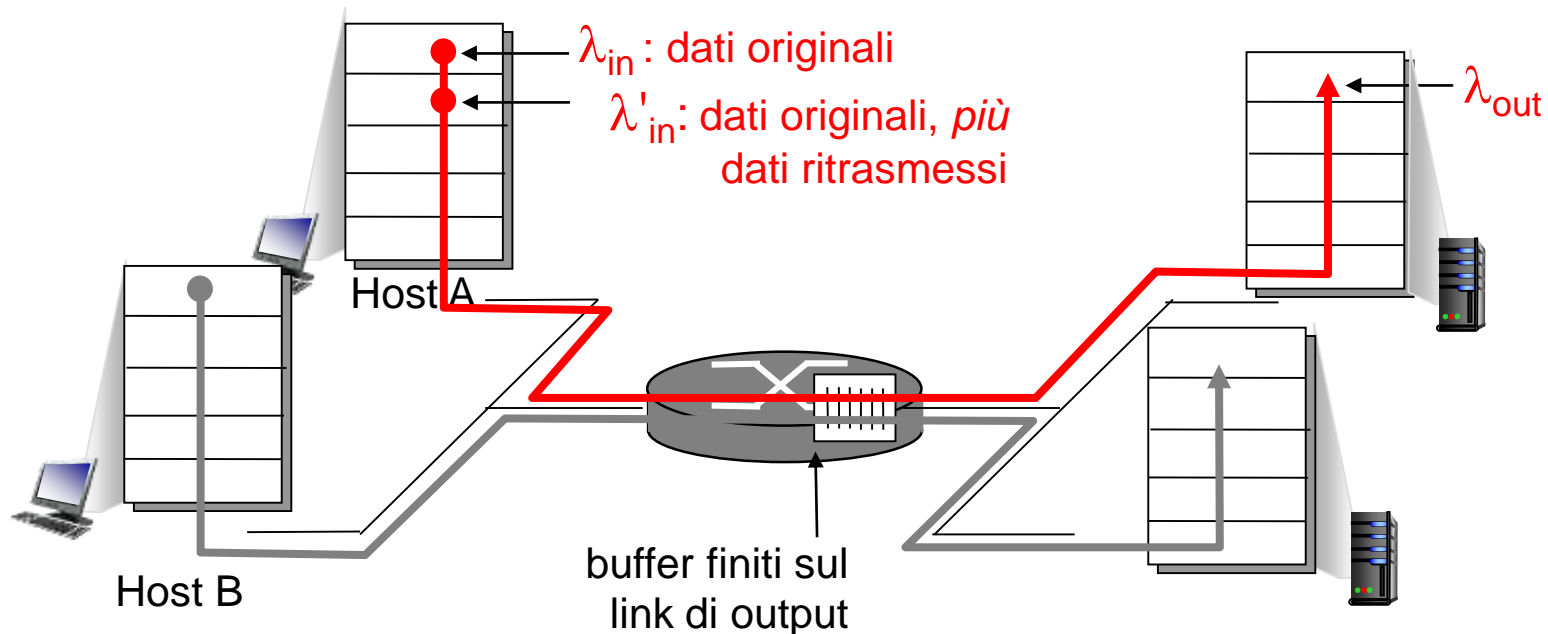
- ❖ throughput massimo per connessione : $R/2$



- ❖ lunghi ritardi se il rate in arrivo, λ_{in} , si avvicina alla capacità

Cause/costi della congestione: scenario 2

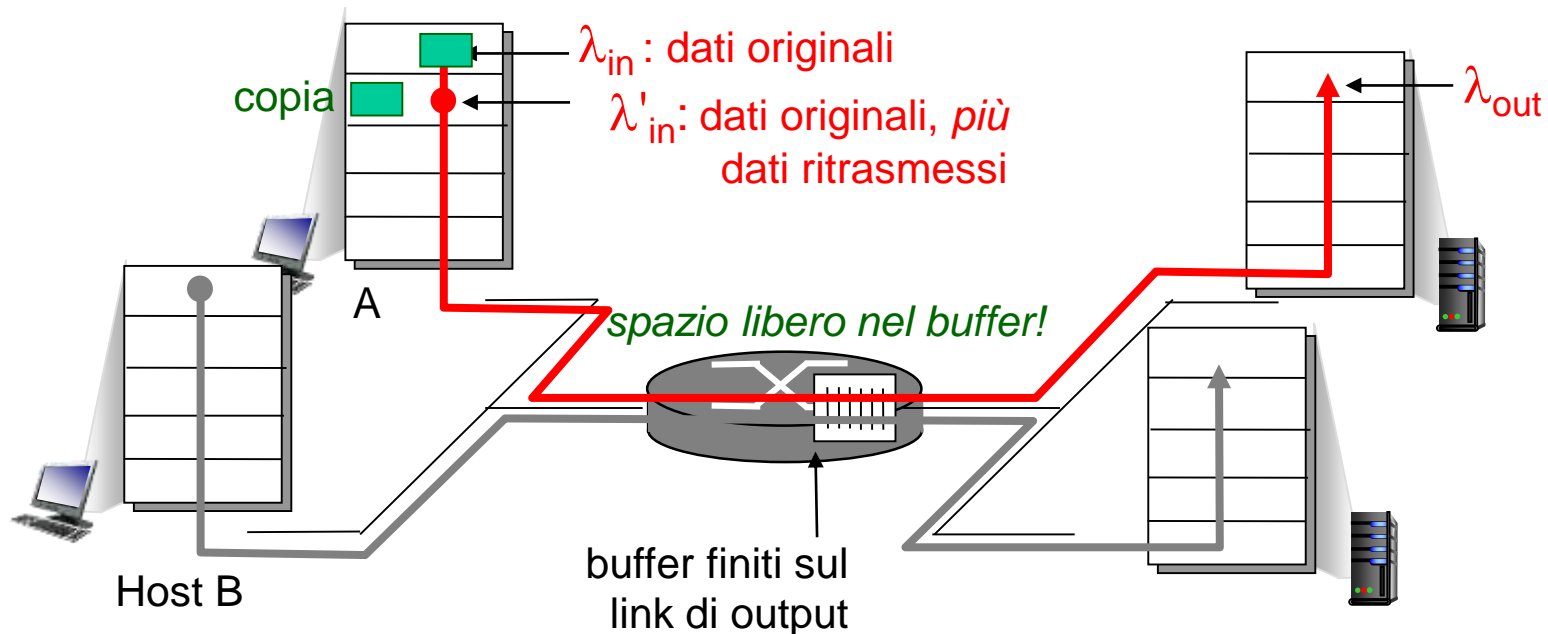
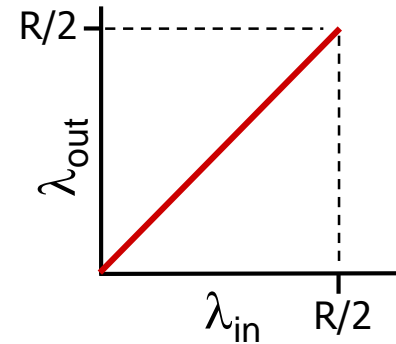
- ❖ un router, buffer *finiti*
- ❖ ritrasmissione dei pacchetti timed-out
 - input nell'application-layer = output dell'application-layer : $\lambda_{in} = \lambda_{out}$
 - l'input del transport-layer include le *ritrasmissioni* : $\lambda'_{in} \geq \lambda_{in}$



Cause/costi della congestione: scenario 2

idealizzazione: conoscenza perfetta

- ❖ il mittente invia solo quando i buffer dei router sono disponibili

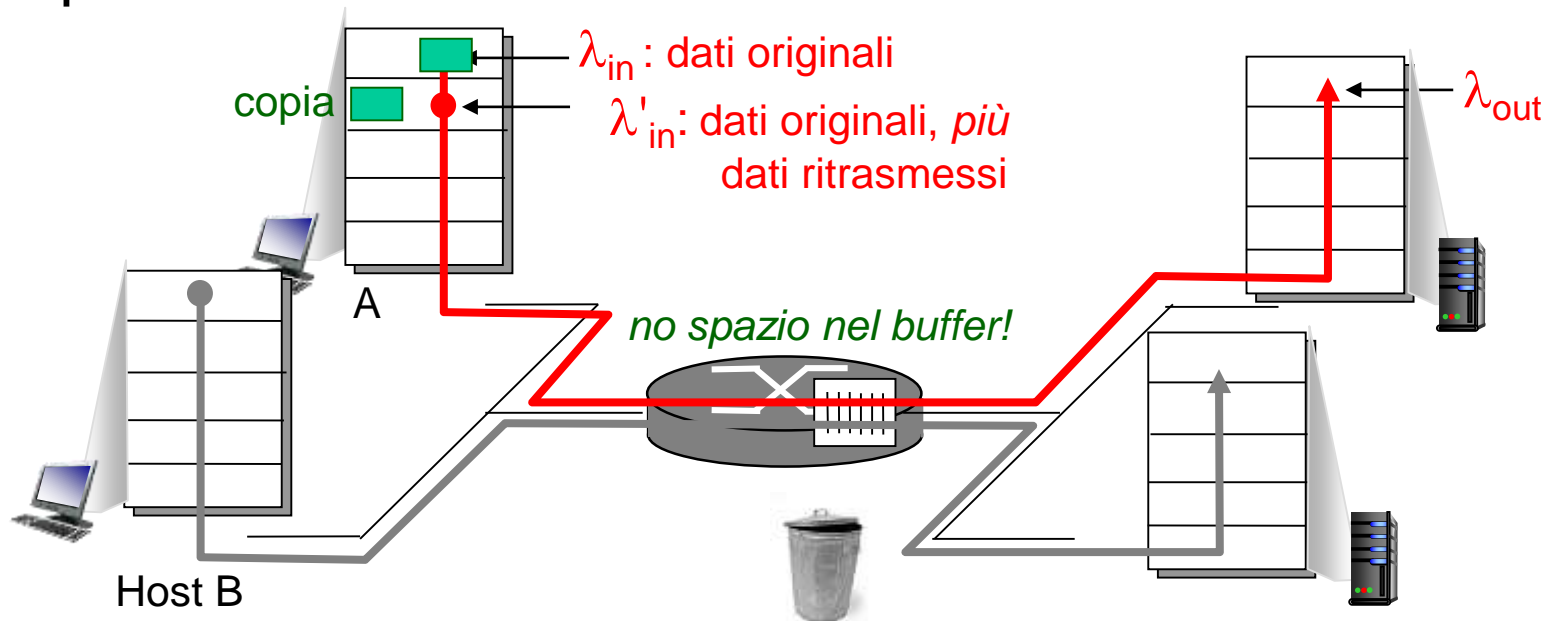


Cause/costi della congestione: scenario 2

idealizzazione: perdite note

i pacchetti possono essere persi, scartati dai router a causa dei buffer pieni

- ❖ il mittente re-invia solo se *sa* che il pacchetto è andato perso

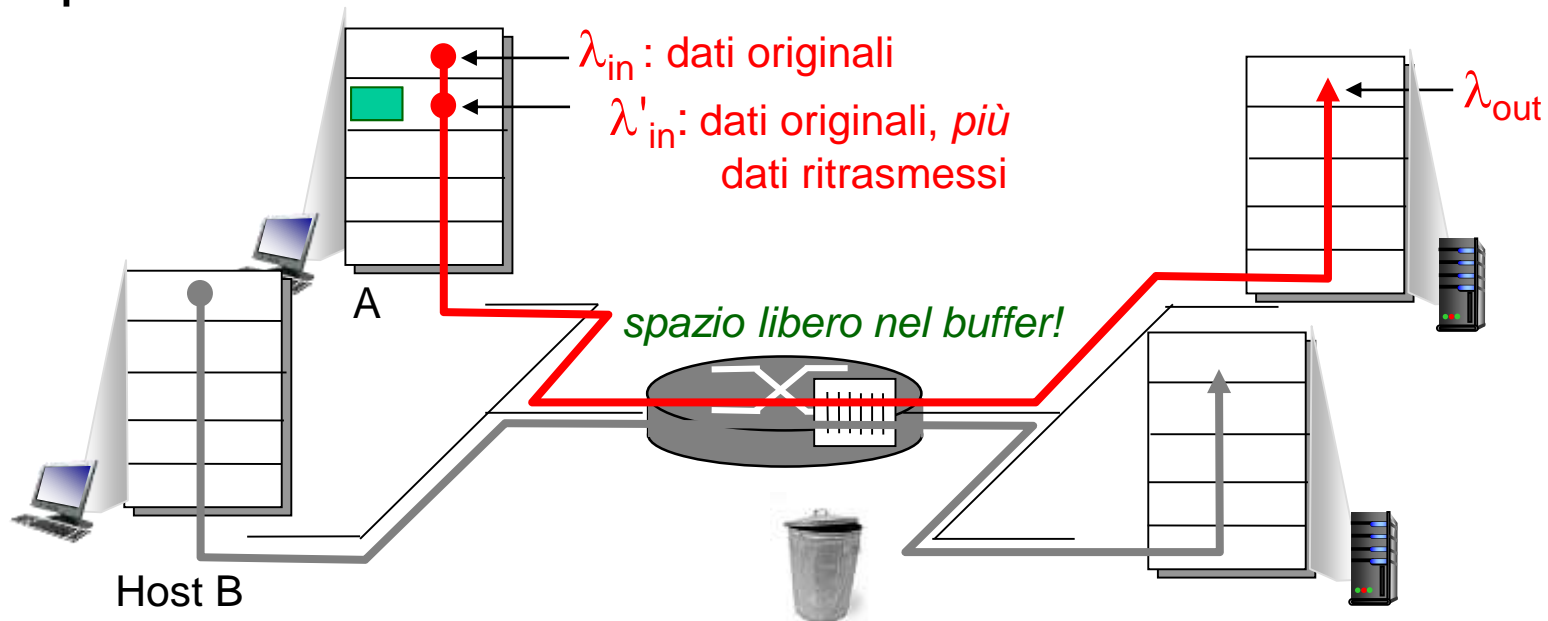
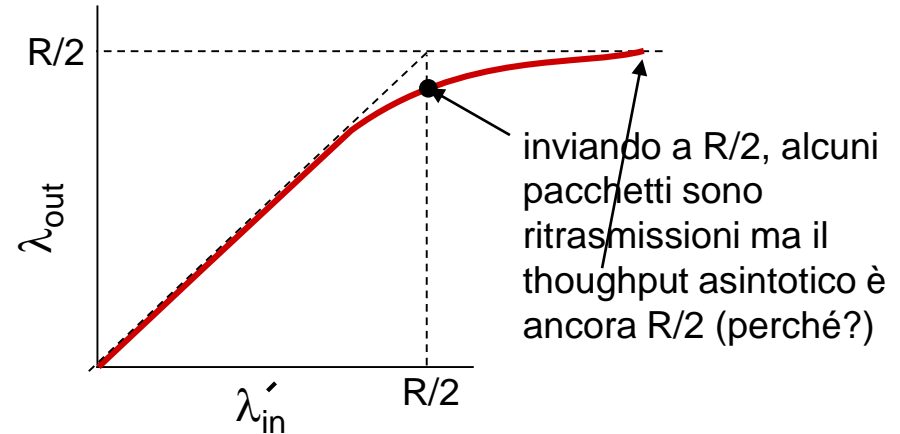


Cause/costi della congestione: scenario 2

idealizzazione: *perdite note*

i pacchetti possono essere persi, scartati dai router a causa dei buffer pieni

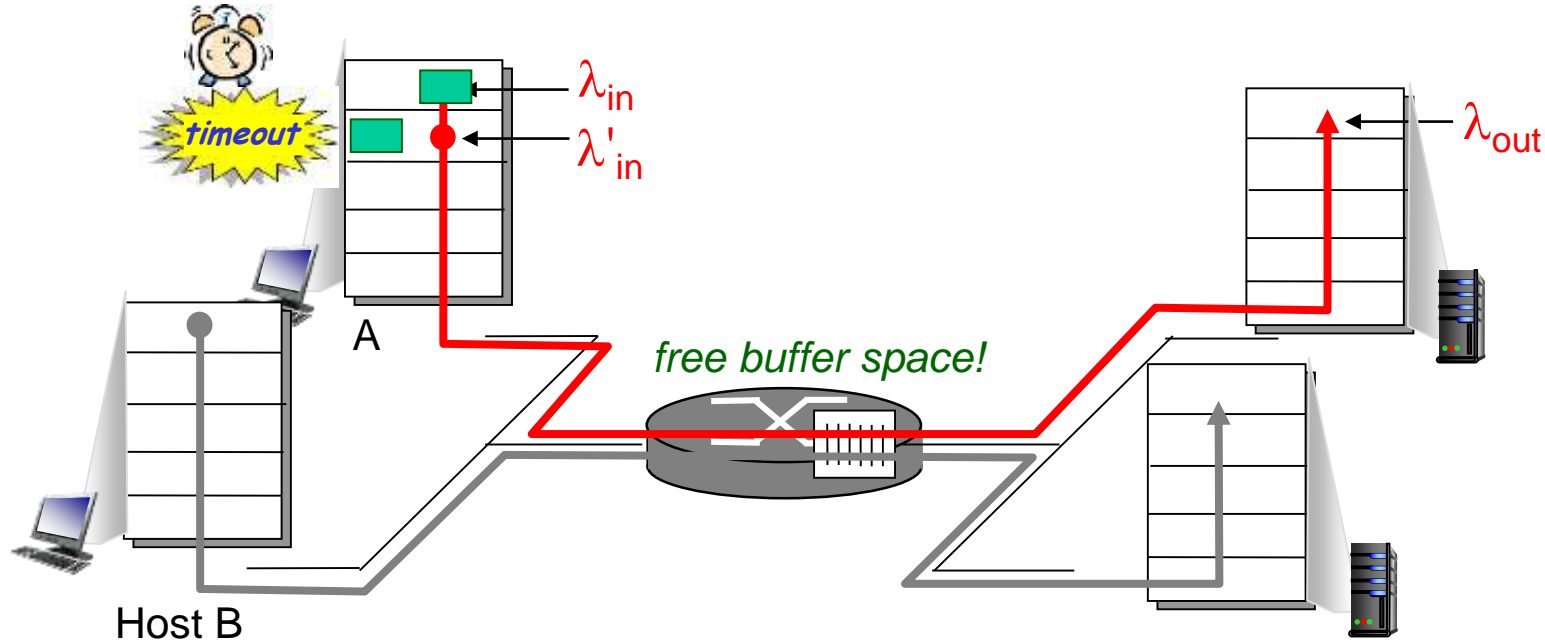
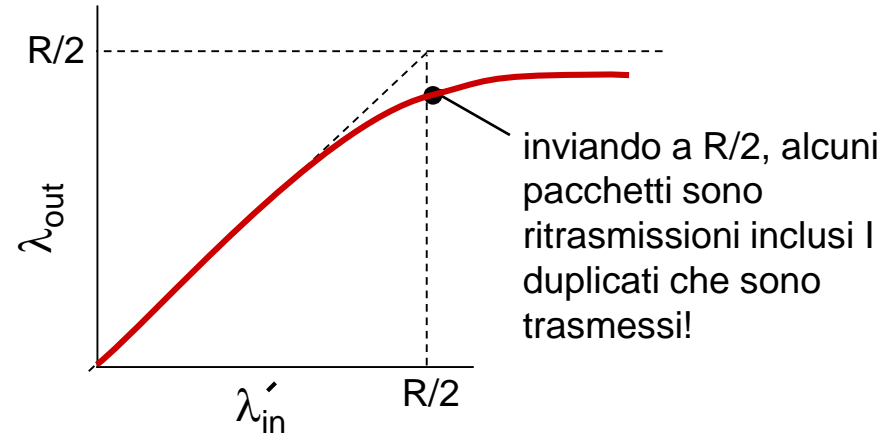
- ❖ il mittente re-invia solo se sa che il pacchetto è andato perso



Cause/costi della congestione: scenario 2

realistico: duplicati

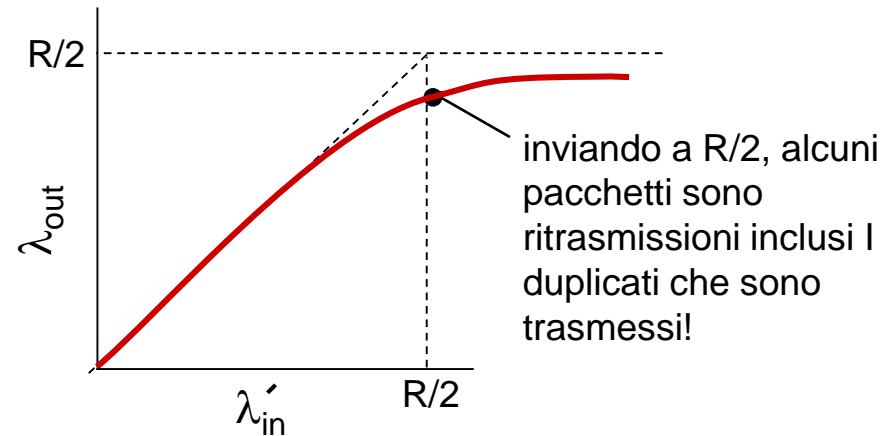
- ❖ i pacchetti possono essere persi, scartati dai router a causa dei buffer pieni
- ❖ il mittente, per time out premature, invia *due* copie, entrambe trasmesse



Cause/costi della congestione: scenario 2

realistico: *duplicati*

- ❖ i pacchetti possono essere persi, scartati dai router a causa dei buffer pieni
- ❖ il mittente, per time out premature, invia *due* copie, entrambi trasmesse



“costi” della congestione:

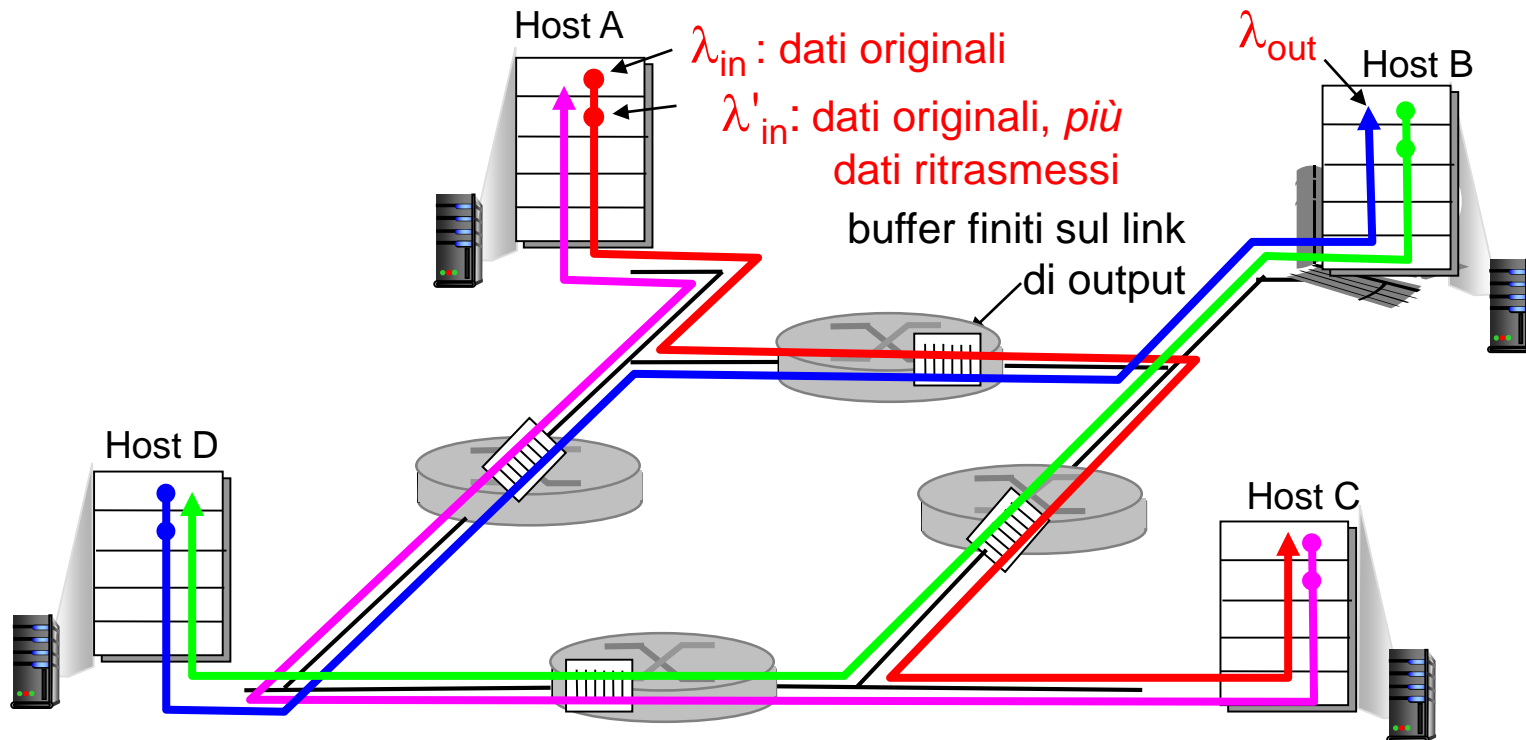
- ❖ più lavoro (ritrasmissioni) per avere “goodput”
- ❖ ritrasmissioni non necessarie: il link trasmette copie multiple di pacchetti
 - decresce il goodput

Cause/costi della congestione: scenario 3

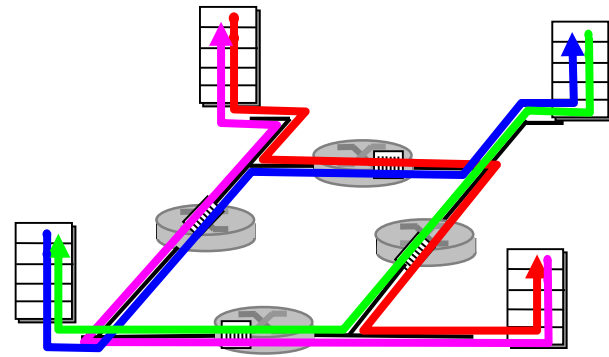
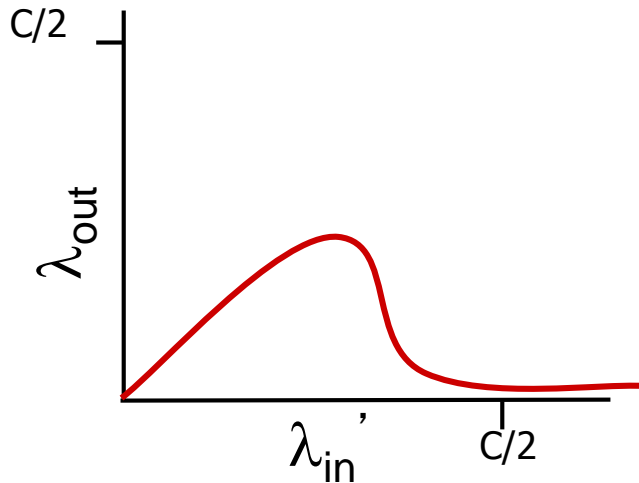
- ❖ quattro mittenti
- ❖ cammini multihop
- ❖ timeout/ritrasmissioni

D: che succede se λ_{in} e λ'_{in} aumentano ?

R: se il λ'_{in} rosso aumenta, tutti i pacchetti blu verranno scartati, throughput blu $\rightarrow 0$



Cause/costi della congestione: scenario 3



un altro “costo” della congestione:

- ❖ quando il pacchetto è scartato, parte della capacità di trasmissione è stata sprecata per quel pacchetto!

Approcci al controllo della congestione

due principali approcci al controllo della congestione:

controllo di congestione end-end:

- ❖ nessun feedback esplicito dalla rete
- ❖ la congestione è dedotta osservando le perdite e i ritardi negli end-system
- ❖ approccio adottato dal TCP

controllo di congestione assistito dalla rete:

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

Case study: controllo di congestione ATM ABR

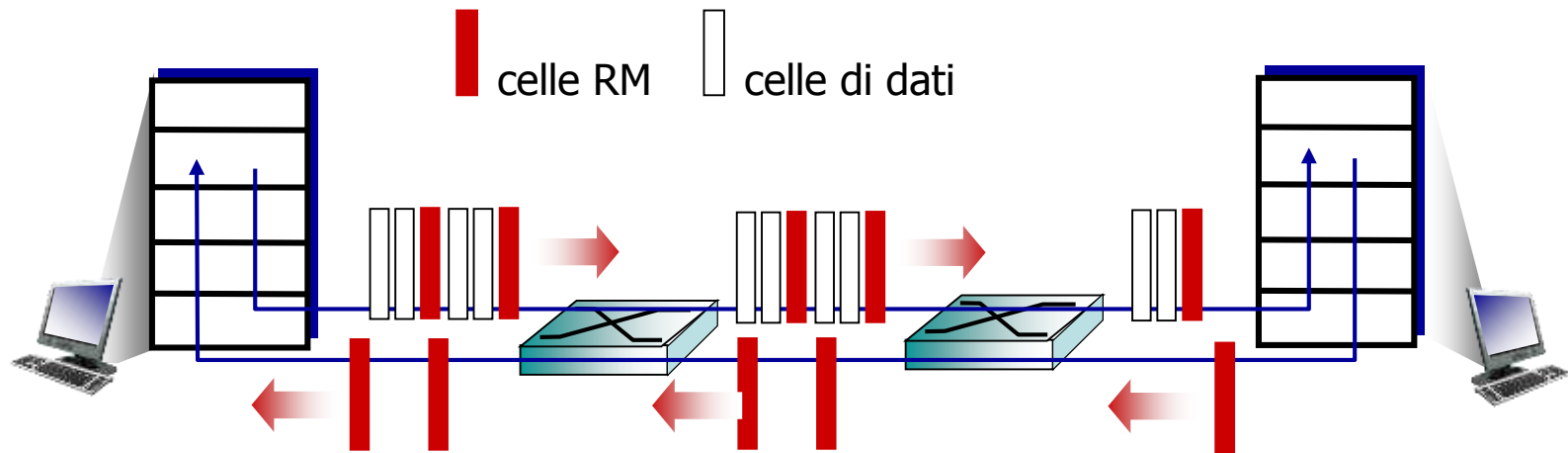
ABR: available bit rate:

- ❖ “servizio elastico”
- ❖ se il percorso del mittente è “sottoutilizzato”:
 - il mittente dovrebbe utilizzare la larghezza di banda disponibile
- ❖ se il percorso del mittente è congestionato:
 - il mittente dovrebbe ridurre al minimo il tasso trasmissivo

Celle RM (resource management) :

- ❖ inviate dal mittente, inframmezzate alle celle di dati
- ❖ i bit in una cella RM sono impostati dagli switch (“*network-assisted*”)
 - *bit NI* : No Increase nel rate (congestione moderata)
 - *bit CI*: Congestion Indication
- ❖ il destinatario restituisce le celle RM al mittente con i bit intatti

Case study: controllo di congestione ATM ABR



- ❖ campo esplicito di frequenza (ER, explicit rate) in ogni cella RM
 - lo switch congestionato può diminuire il valore del campo ER
 - in questo modo, il campo ER sarà impostato alla velocità minima supportabile da tutti gli switch sul percorso globale
- ❖ ogni cella di dati contiene un bit EFCl: impostato a 1 nello switch congestionato
 - se la cella di dati che precede la cella RM ha impostato il bit EFCl, il mittente imposta il bit CI nella cella RM restituita

Capitolo 3: Livello di trasporto

- 3.1 servizi a livello di trasporto
- 3.2 multiplexing e demultiplexing
- 3.3 trasporto senza connessione: UDP
- 3.4 principi del trasferimento dati affidabile

- 3.5 trasporto orientato alla connessione : TCP
 - struttura dei segmenti
 - trasferimento dati affidabile
 - controllo di flusso
 - gestione della connessione

- 3.6 principi sul controllo di congestione

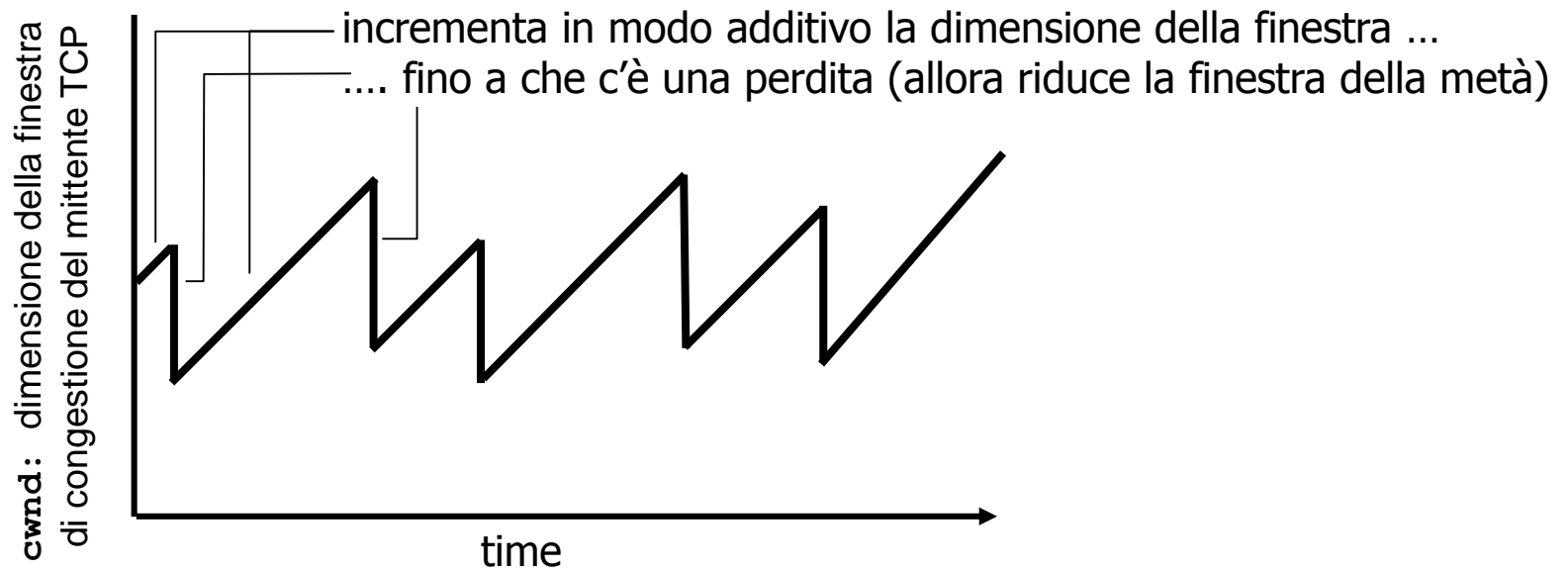
3.7 controllo di congestione TCP

Controllo di congestione nel TCP

AIMD: additive increase multiplicative decrease
(incremento additivo decremento moltiplicativo)

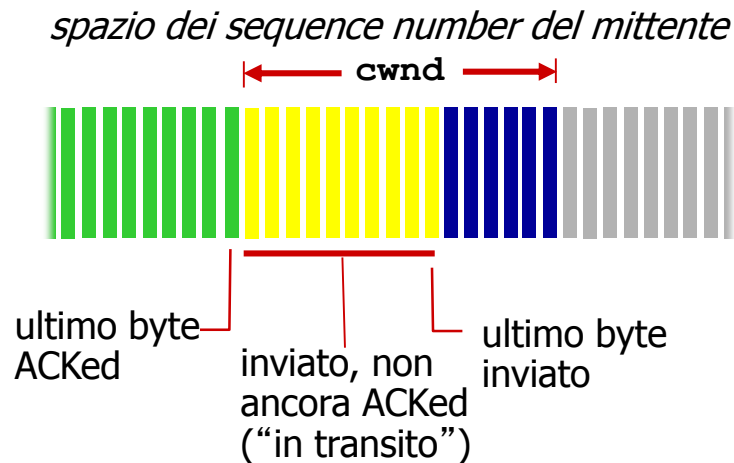
- ❖ *approccio*: il mittente incrementa il transmission rate (window size), sondando la bandwidth, fino a quando non si verifica una perdita
 - *incremento additivo*: incrementa **cwnd** di 1 MSS ogni RTT finché non ci sono perdite
 - *decremento moltiplicativo* : riduce **cwnd** della metà dopo una perdita

Controllo di congestione nel TCP



Comportamento a dente di sega dell'AIMD

Controllo di congestione TCP : dettagli



- ❖ il mittente limita la trasmissione:

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

- ❖ **cwnd** è dinamica, funzione della congestione della rete percepita

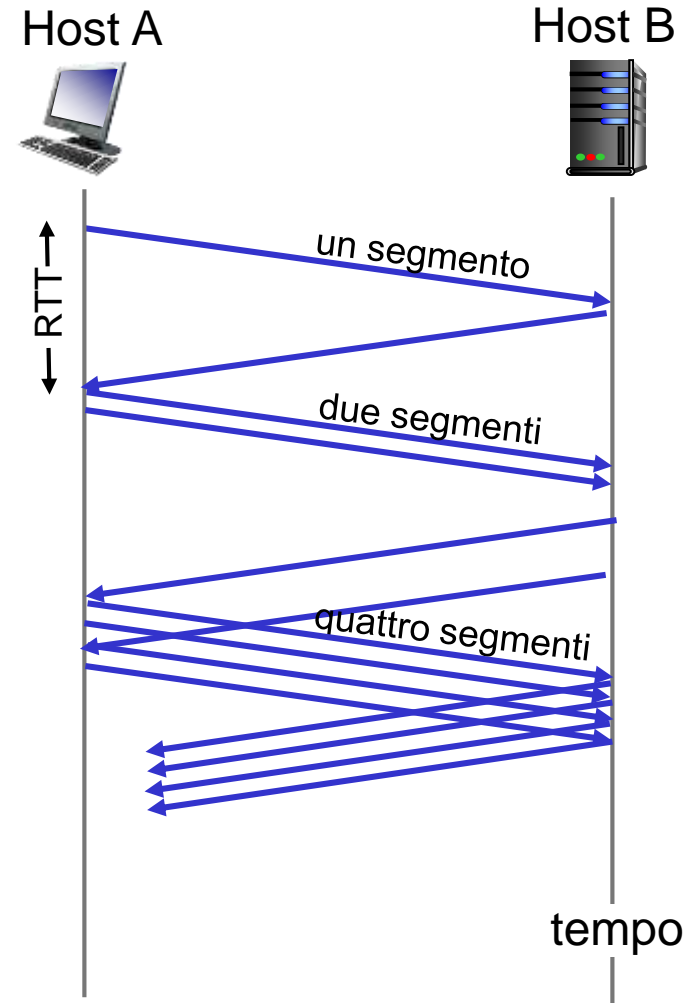
rate di invio TCP:

- ❖ *all'incirca*: invia **cwnd** byte, aspetta RTT per gli ACK, e poi invia altri byte

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

Partenza lenta

- ❖ quando si stabilisce una connessione, incrementa il rate esponenzialmente fino al primo evento di perdita:
 - inizialmente **cwnd** = 1 MSS
 - raddoppia **cwnd** ogni RTT
 - **cwnd** è incrementato a ogni ACK ricevuto
- ❖ risultato: il rate iniziale è lento ma si impenna con velocità esponenziale



TCP: rilevazione e reazione alle perdite

- ❖ perdite indicate dai timeout:
 - `cwnd` impostato a 1 MSS;
 - la finestra cresce esponenzialmente (come nella partenza lenta) fino alla soglia, poi cresce linearmente
- ❖ perdite indicate da 3 duplicate ACK: TCP RENO
 - gli ACK duplicati indicano la capacità della rete di consegnare segmenti
 - `cwnd` viene dimezzato poi cresce linearmente
- ❖ TCP Tahoe imposta sempre `cwnd` a 1 (timeout o 3 duplicate ack)

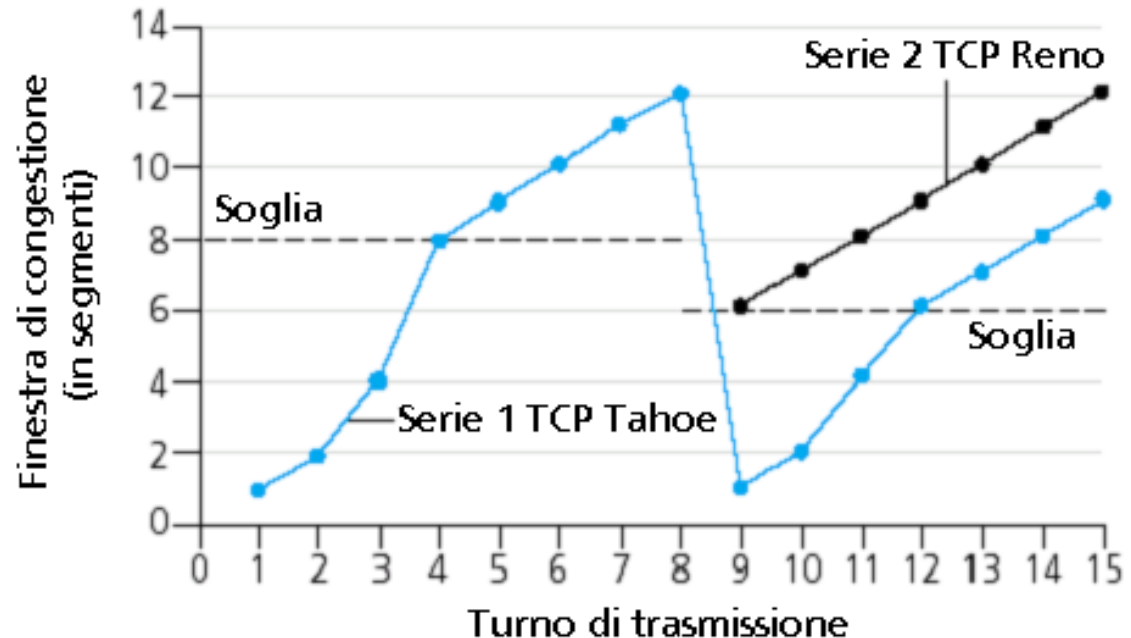
TCP: passaggio da partenza lenta a CA

D: quando la crescita da esponenziale deve passare a lineare?

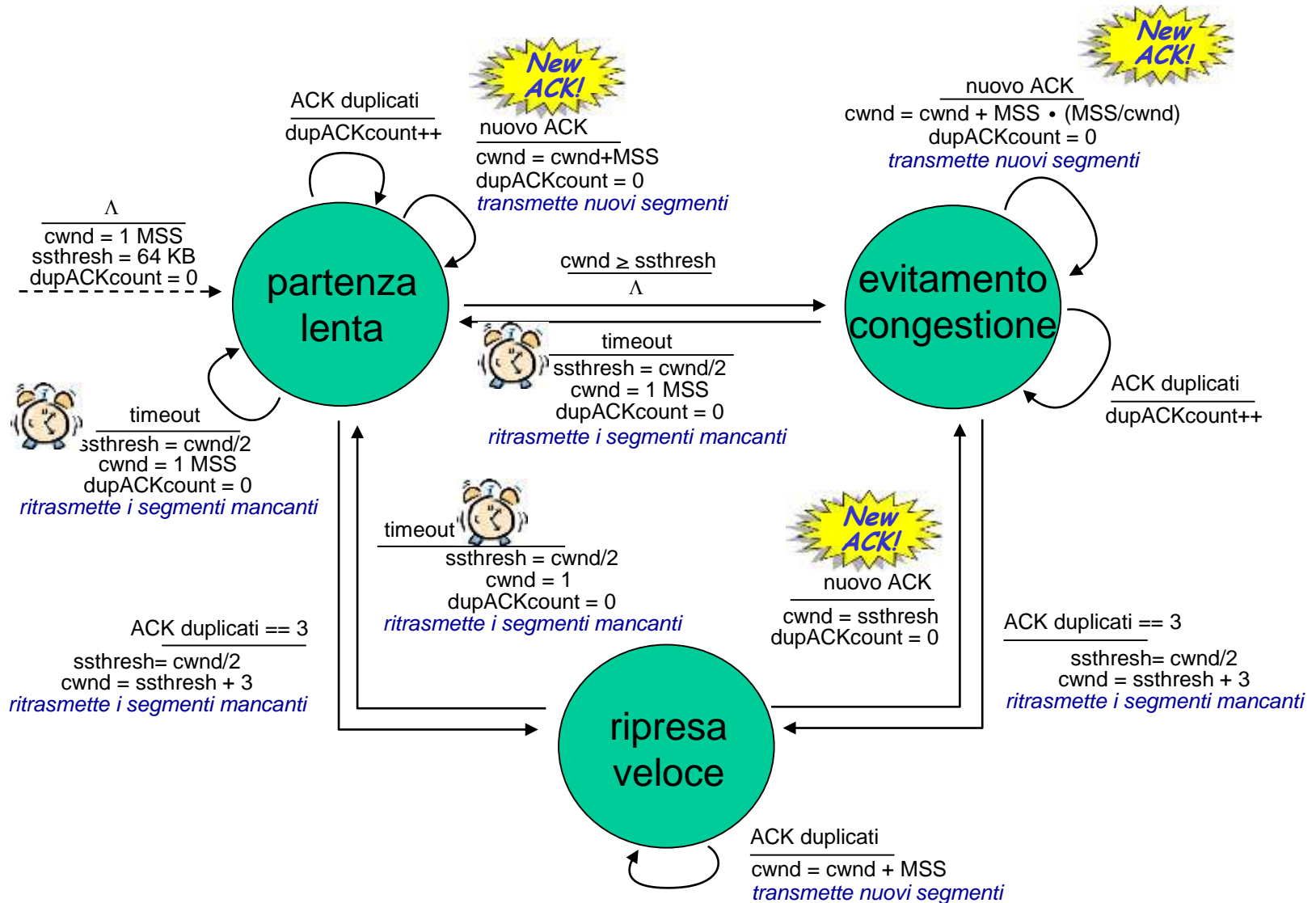
R: quando **cwnd** raggiunge 1/2 del suo valore prima del timeout.

implementazione:

- ❖ **ssthresh** variabile
- ❖ in caso di evento di perdita, **ssthresh** è impostata a 1/2 di **cwnd** appena prima dell'evento di perdita



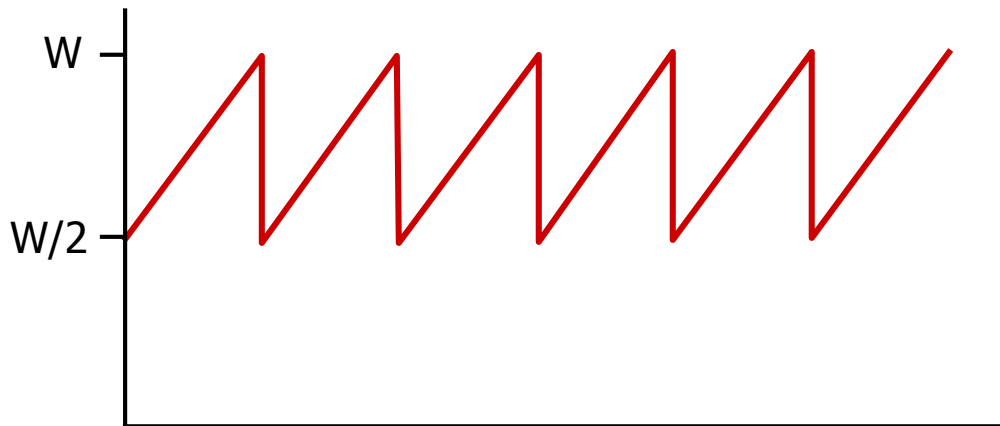
Riassunto: Controllo di congestione TCP



Throughput TCP

- ❖ qual è il throughput medio di TCP in funzione della dimensione della finestra e di RTT?
 - ignoriamo le fasi di partenza lenta
- ❖ W : dimensione della finestra (in byte) quando avviene una perdita
 - la dimensione media della finestra (# byte in transito) è $\frac{3}{4} W$
 - il throughput medio è $\frac{3}{4}W$ per RTT

$$\text{throughput medio TCP} = \frac{3}{4} \frac{W}{\text{RTT}} \text{ bytes/sec}$$



Futuro di TCP

- ❖ esempio: segmenti da 1500 byte, RTT di 100ms, vogliamo un throughput di 10 Gbps
- ❖ occorre una dimensione della finestra pari a $W = 83.333$ segmenti in transito
- ❖ throughput in funzione della frequenza di smarrimento, L [Mathis 1997]:

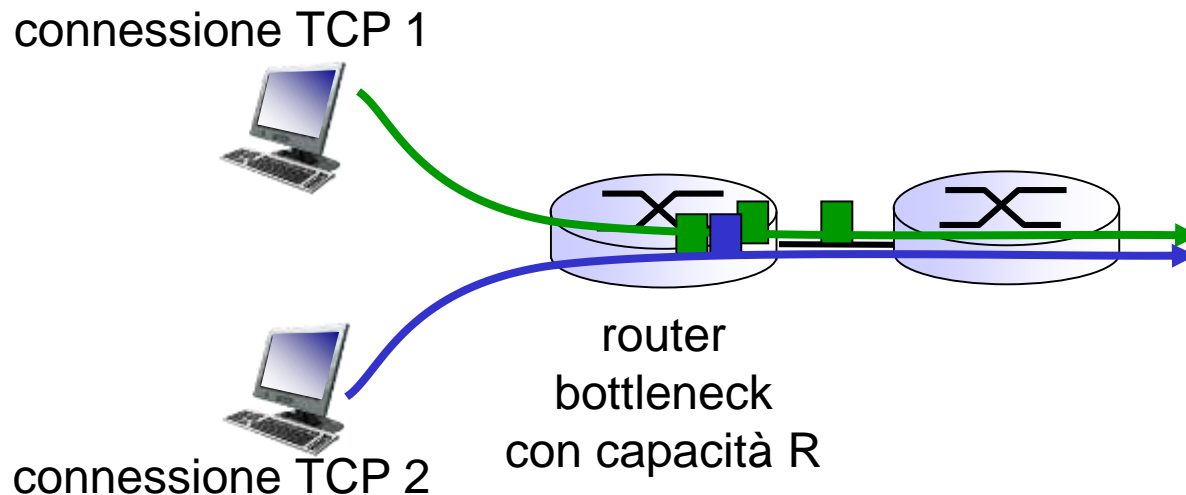
$$\text{throughput TCP} = \frac{1.22 \cdot \text{MSS}}{\text{RTT} \sqrt{L}}$$

→ per ottenere un throughput da 10 Gbps, occorre un rate di perdite pari a $L = 2 \cdot 10^{-10}$ – *davvero piccolo!*

- ❖ occorrono nuove versioni di TCP per ambienti ad alta velocità

Equità di TCP

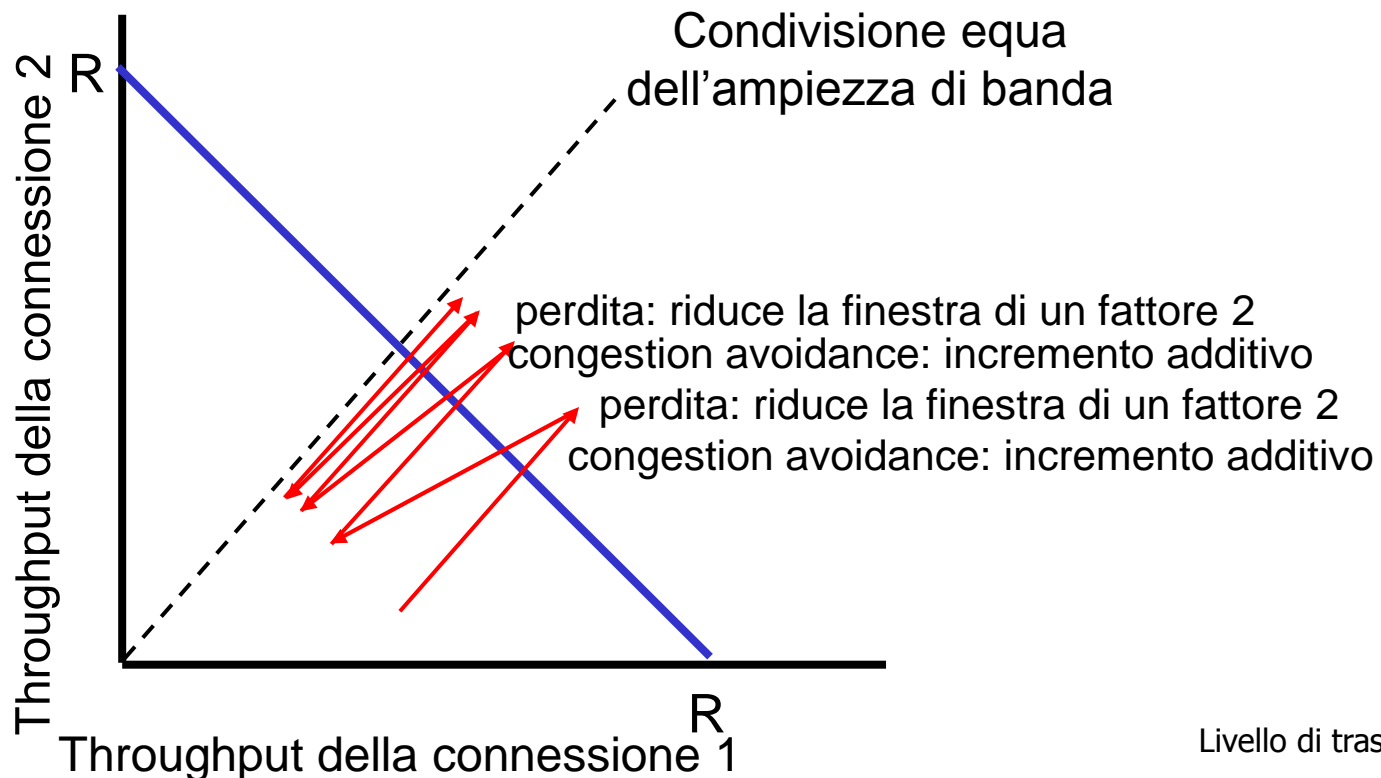
obiettivo di equità: se K sessioni TCP condividono lo stesso link bottleneck con bandwidth R , ogni sessione dovrebbe avere un rate di R/K



Perché TCP è equo?

due connessioni:

- ❖ l'incremento additivo determina una pendenza pari a 1, all'aumentare del throughput
- ❖ il decremento moltiplicativo riduce il throughput in modo proporzionale



Equità (altro)

Equità e UDP

- ❖ le applicazioni multimediali spesso non usano TCP
 - non vogliono che il loro rate venga ridotto dal controllo di congestione
- ❖ utilizzano UDP:
 - immettono audio/video a frequenza costante, tollerano la perdita di pacchetti

Equità e connessioni TCP parallele

- ❖ le applicazioni possono aprire più connessioni parallele tra due host
- ❖ i web browsers lo fanno
- ❖ es., link con rate R e 9 connessioni esistenti:
 - se una nuova applicazione chiede 1 connessione TCP, ottiene un rate $R/10$
 - se la nuova applicazione chiede 11 connessioni TCP, ottiene un rate superiore a $R/2$

Capitolo 3: Riassunto

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

prossimamente:

- ❖ lasciare la “periferia” della rete (livelli di applicazione e di trasporto)
- ❖ entrare nel “cuore” della rete