Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di trasporto

dr. Manuel Fiorelli
manuel.fiorelli@uniroma2.it
https://art.uniroma2.it/fiorelli

Basate sulle slide del libro di testo:
https://gaia.cs.umass.edu/kurose_ross/ppt.php

Livello di trasporto: panoramica

Obiettivi:

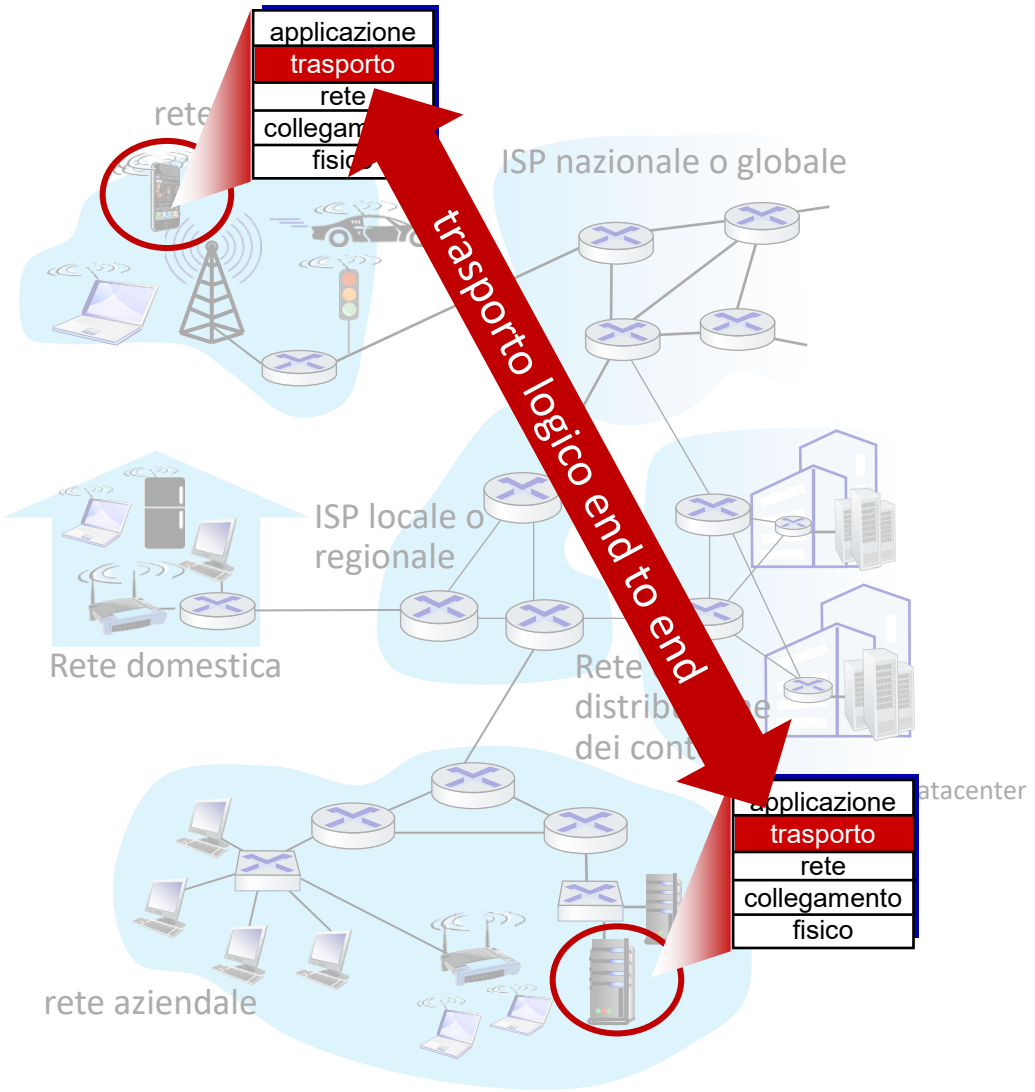
- capire i principi che sono alla base dei servizi del livello di trasporto:
  - multiplexing, demultiplexing
  - trasferimento dati affidabile
  - controllo di flusso
  - controllo della congestione
- conoscere i protocolli del livello di trasporto di Internet:
  - UDP: trasporto senza connessione
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

Livello di trasporto: tabella di marcia

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

Servizi e protocolli di trasporto

- Forniscono la *comunicazione logica* tra processi applicativi di host differenti
- I protocolli di trasporto vengono eseguiti nei sistemi periferici:
  - lato invio: scinde (se necessario) i messaggi dell'applicazione, combinando ciascuna parte con un'intestazione per creare un *segmento* e lo passa al livello di rete
  - lato ricezione: estrae i dati contenuti del segmento e li passa al livello di applicazione
- I router nel cammino da un host all'altro operano solo sull'intestazione del datagramma, ignorando il segmento incapsulato al suo interno
- Più protocolli di trasporto sono a disposizione delle applicazioni
  - TCP, UDP



Relazione tra livello di trasporto e livello di rete

Analogia domestica:

12 ragazzi a casa di Ann inviano lettere a 12 ragazzi a casa di Bill:

- host = case
- processi = ragazzi
- messaggi delle applicazioni= lettere nelle buste

Relazione tra livello di trasporto e livello di rete

- livello di trasporto: comunicazione logica tra *processi*
  - si basa sui servizi del livello di rete e li potenzia
- livello di rete: comunicazione logica tra *host*

➡ multiplexing/demultiplexing

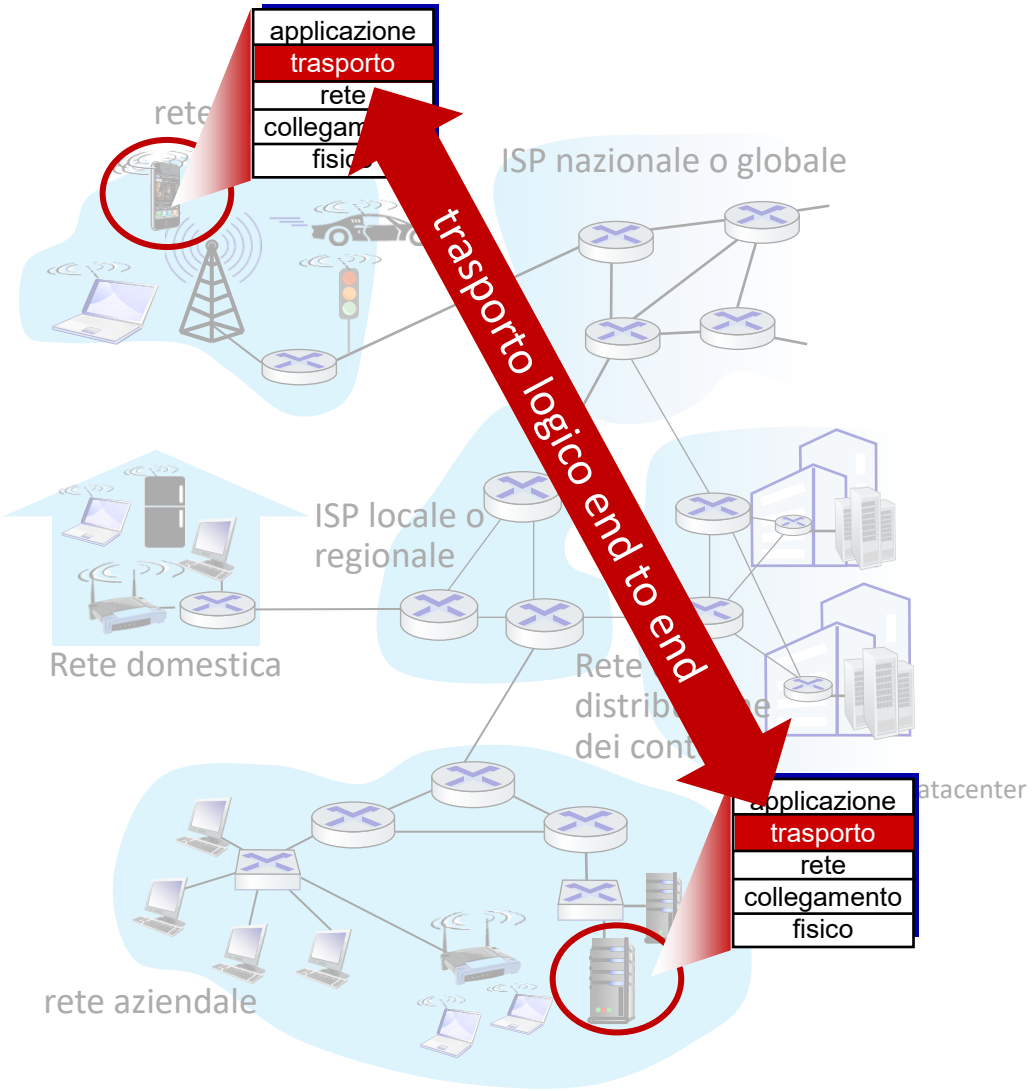
Analogia domestica:

12 ragazzi a casa di Ann inviano lettere a 12 ragazzo a casa do Bill:

- host = case
- processi = ragazzi
- messaggi delle applicazioni= lettere nelle buste
- protocollo di trasporto = Ann e Bill che raccolgono e distribuiscono la posta tra i fratelli
- protocollo a livello di rete = servizio postale (compresi i postini)

Protocolli del livello di trasporto in Internet

- **UDP:** User Datagram Protocol
  - inaffidabile, consegne senz'ordine
  - estensione "senza fronzoli" di IP: solo comunicazione tra processi e controllo degli errori
- **TCP:** Transmission Control Protocol
  - comunicazione tra processi affidabile, consegne nell'ordine originario
  - controllo di flusso
  - controllo della congestione
  - instaurazione della connessione
- servizi *non* disponibili (impossibile realizzarli a livello di trasporto date le caratteristiche del servizio di rete di Internet):
  - garanzie su ritardi
  - garanzie su banda



Capitolo 3: tabella di marcia

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

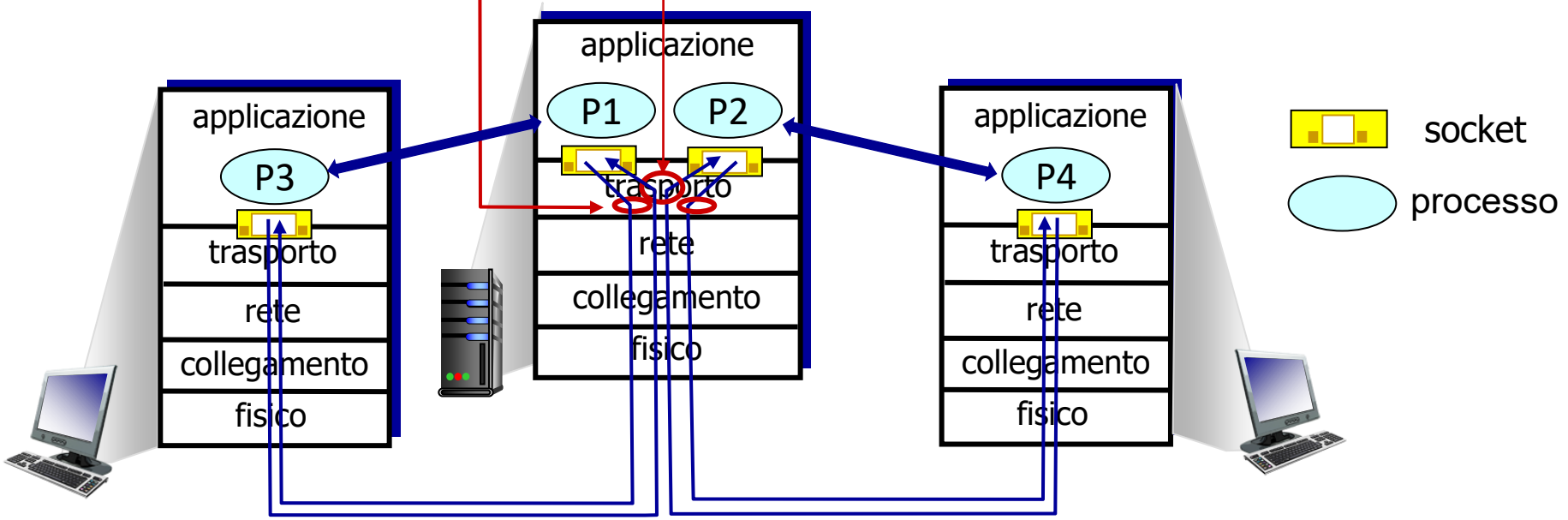
Multiplexing/demultiplexing

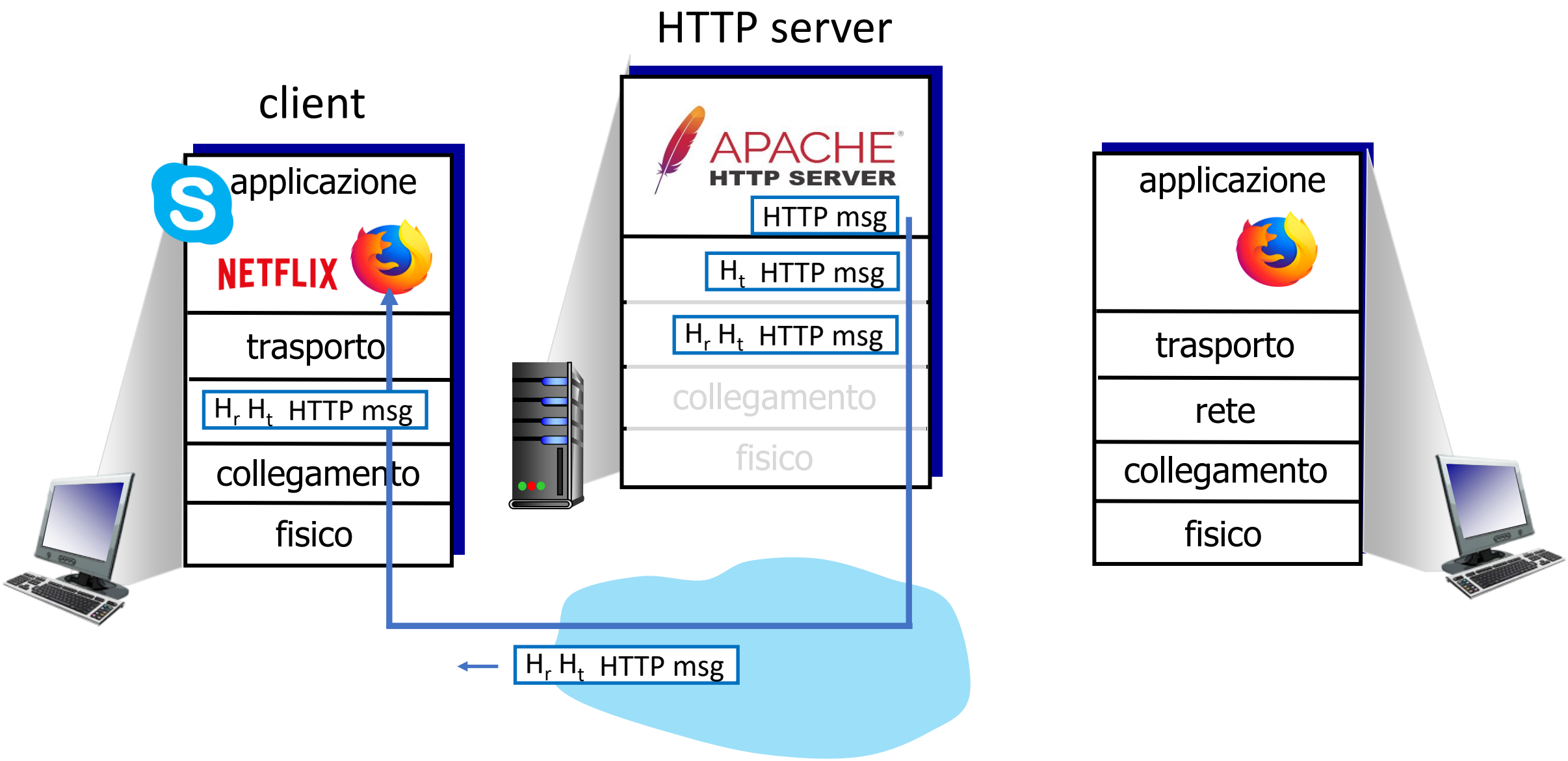
Multiplexing lato mittente:

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

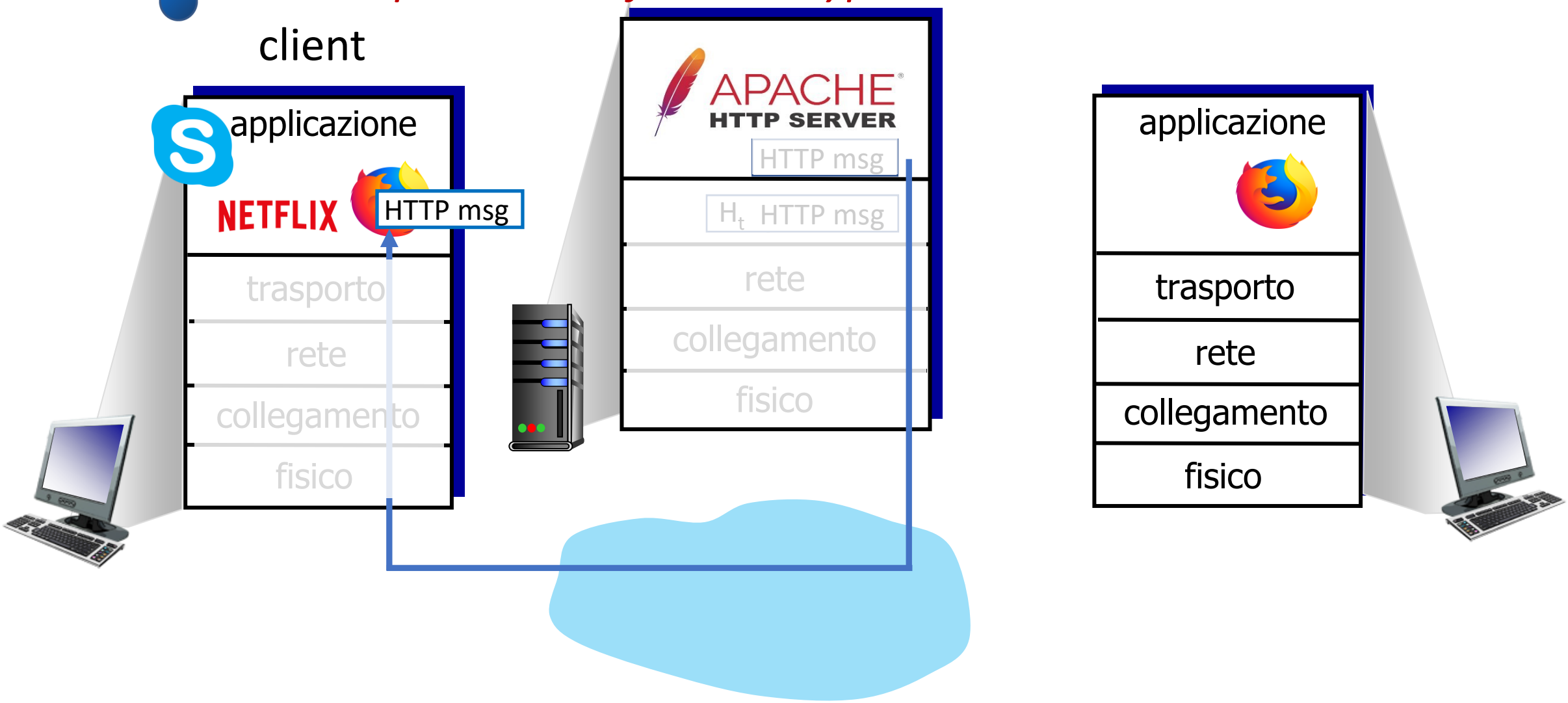
demultiplexing lato ricevente:

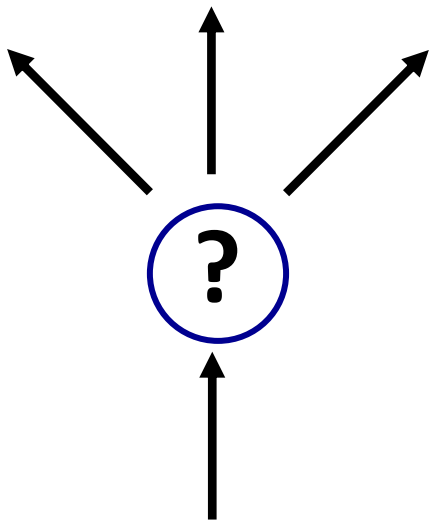
utilizzare le informazioni nell'intestazione per consegnare i segmenti ricevuti alla socket corretta



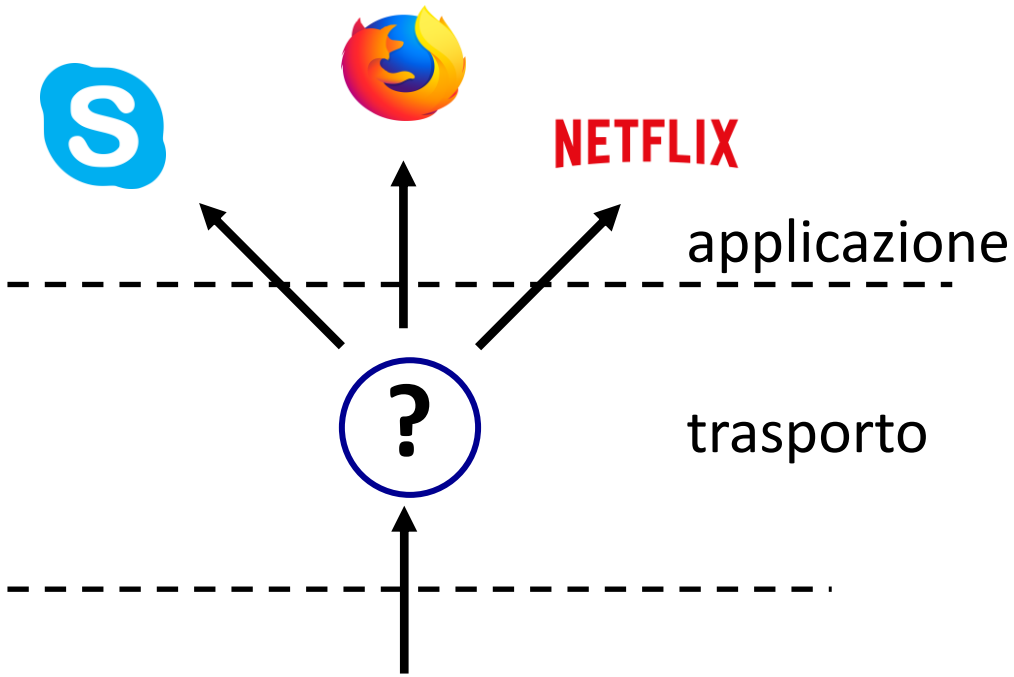

HTTP server
client
applicazione
NETFLIX
trasporto
H_r H_t HTTP msg
collegamento
fisico
APACHE
HTTP SERVER
HTTP msg
H_t HTTP msg
H_r H_t HTTP msg
collegamento
fisico
applicazione
trasporto
rete
collegamento
fisico
H_r H_t HTTP msg

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?





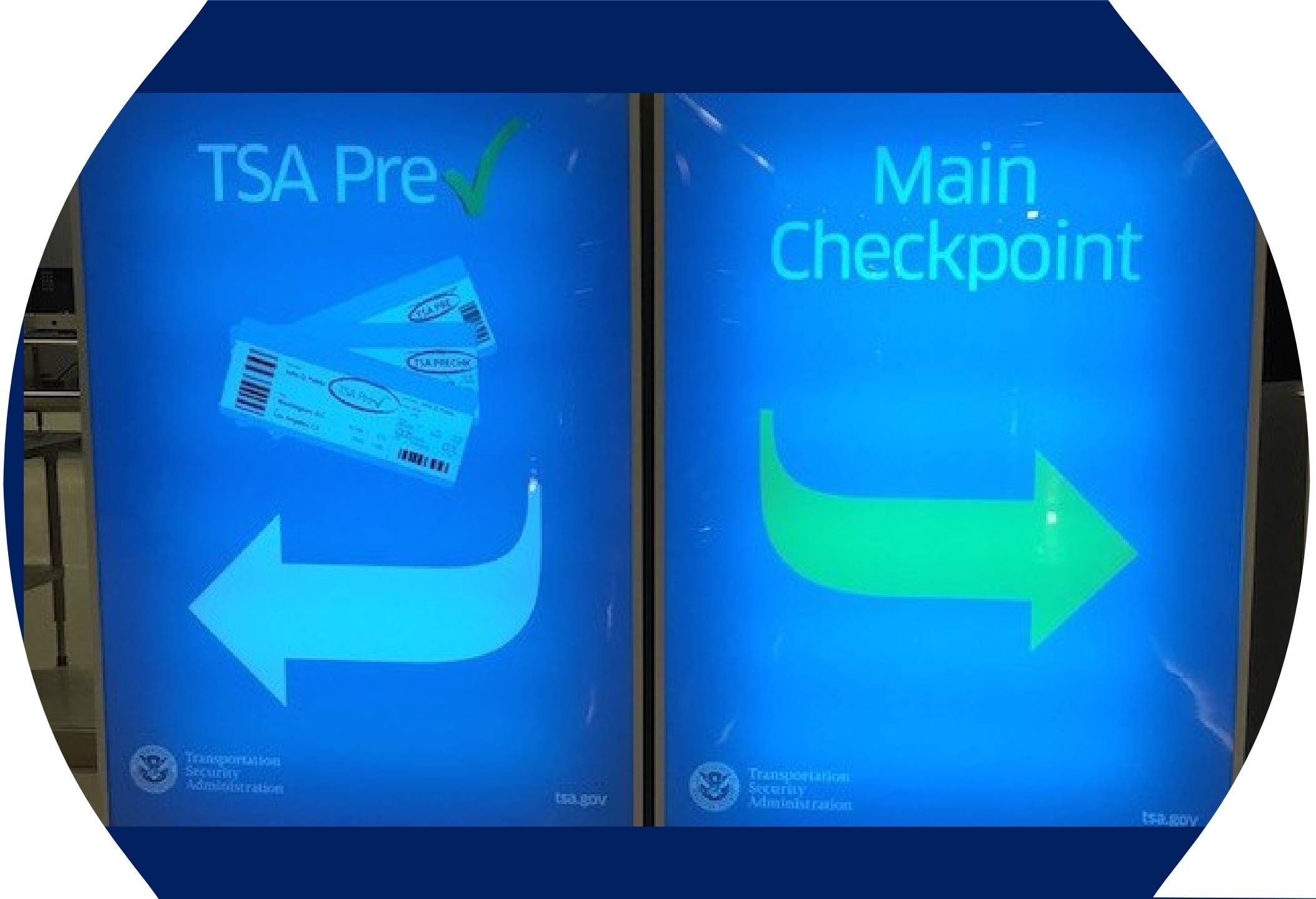
de-multiplexing

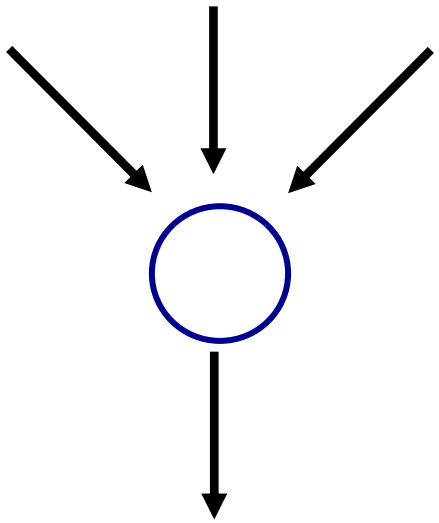


de-multiplexing

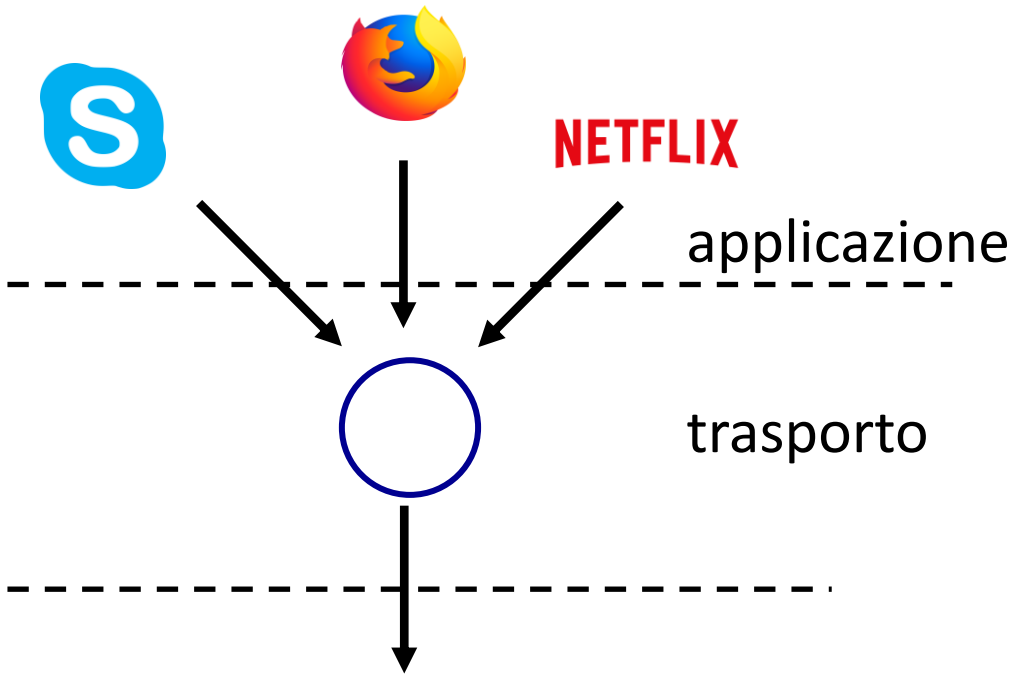
Demultiplexing

AIRFRANCE
ECONOMY
SKYTEAM
AIRFRANCE
SKY
PRIORITY™
SKYTEAM

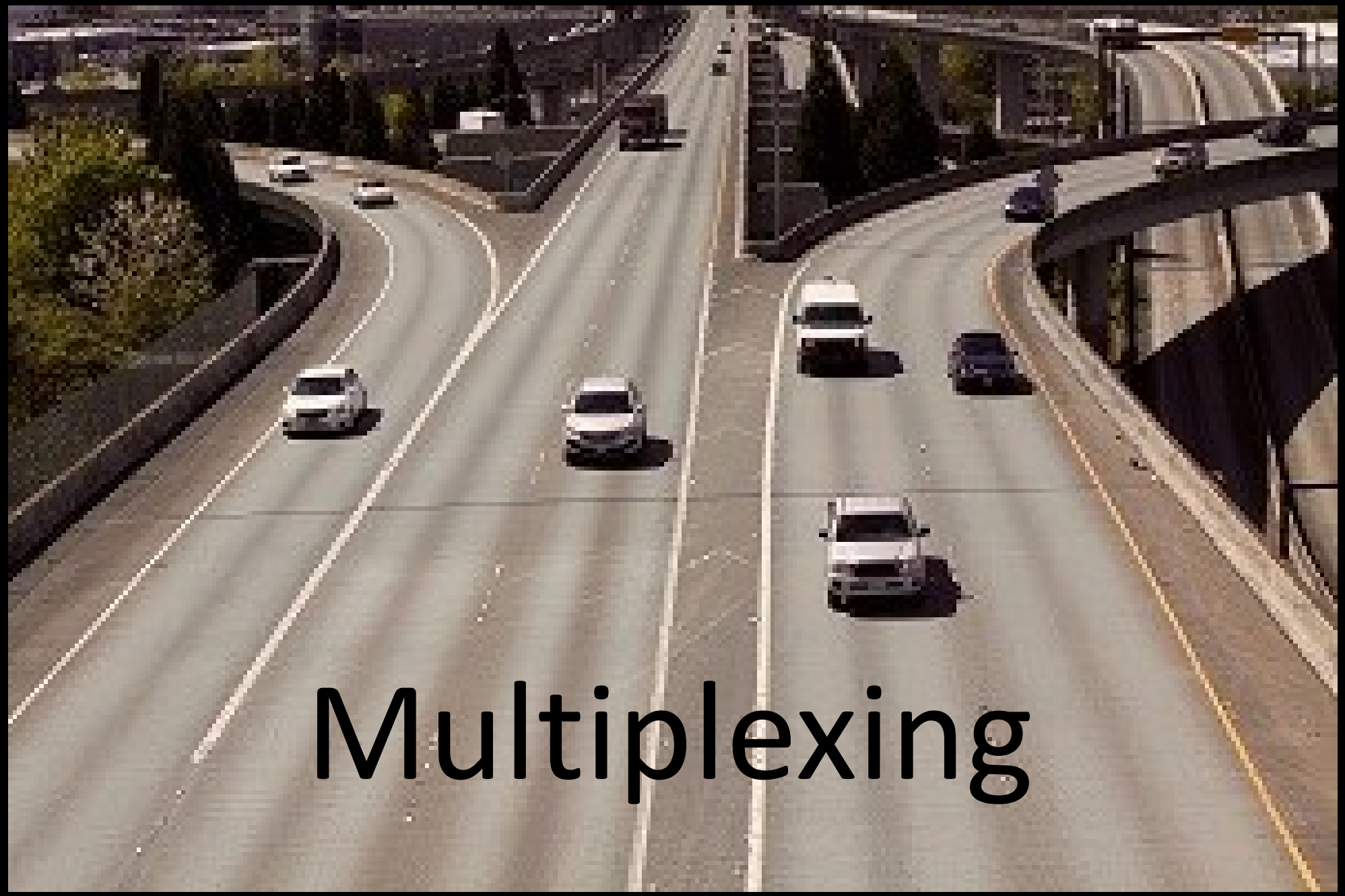

TSA Pre
Transportation Security Administration
tsa.gov
Main Checkpoint
Transportation Security Administration
tsa.gov



multiplexing



multiplexing



Multiplexing

Come funziona il demultiplexing

- l'host riceve i datagrammi IP
  - ogni datagramma ha un indirizzo IP di origine e un indirizzo IP di destinazione
  - ogni datagramma trasporta 1 segmento a livello di trasporto
  - ogni segmento ha un numero di porta di origine e un numero di porta di destinazione
- l'host usa *gli indirizzi IP e i numeri porta* per inviare il segmento alla socket appropriata

formato dei segmenti TCP/UDP

Demultiplexing senza connsessione

- quando si crea una socket, si può specificare il numero di porta:

```
mySocket = socket(AF_INET, SOCK_DGRAM)
mySocket.bind(('',9157))
```

oppure, soprattutto per il lato client, lasciare che il sistema operativo le assegni un numero di porta *effimero*

- quando si crea il datagramma da inviare alla socket UDP, si deve specificare
  - indirizzo IP di destinazione
  - numero di porta di destinazione
- il segmento viene passato al livello di rete

quando l'host riceve il segmento UDP:

- controlla il numero della porta di destinazione nel segmento
- invia il segmento UDP alla socket con quel numero di porta

Datagrammi IP/UDP con lo *stesso indirizzo IP e numero di porta di destinazione*, ma indirizzi IP e/o numeri di porta di origine differenti vengono inviati alla *stessa socket* sull'host ricevente

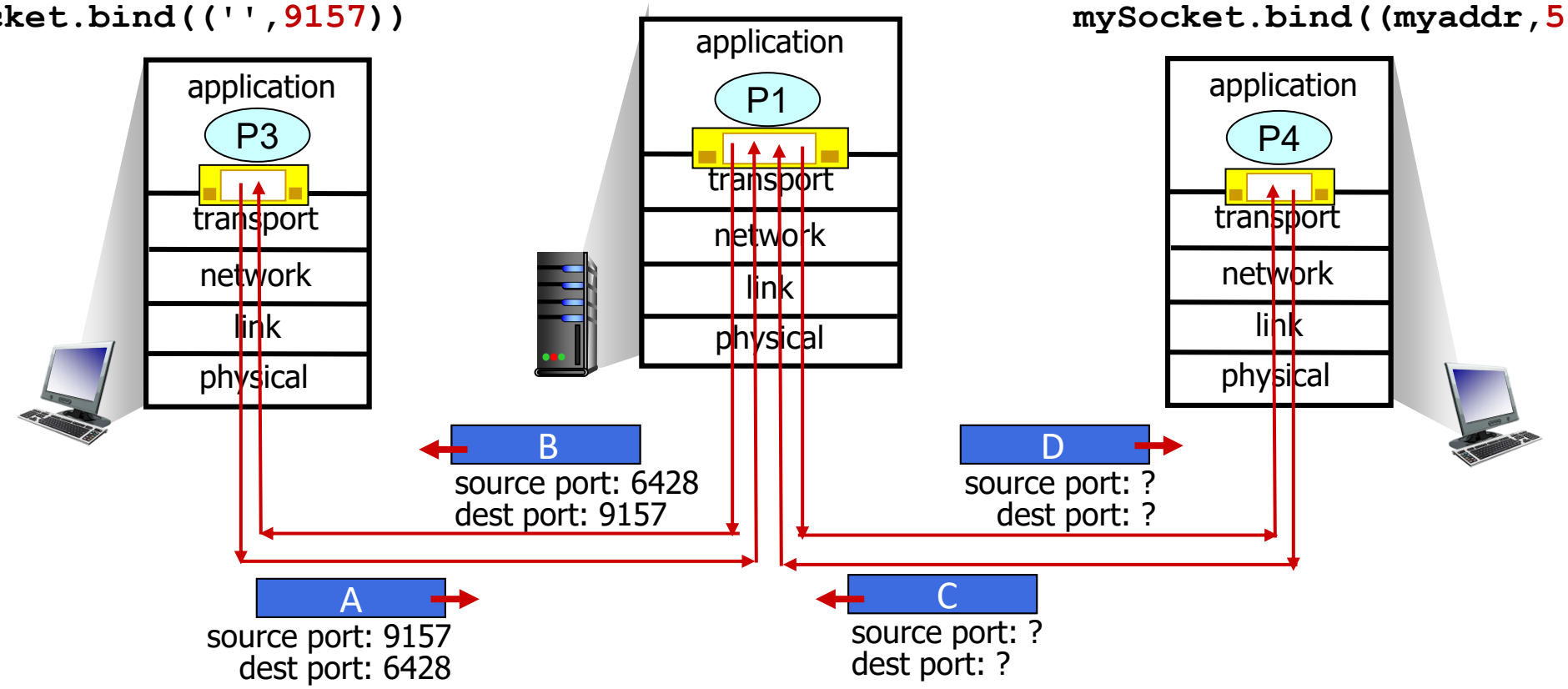
Indirizzo IP e numero di porta di origine servono come "indirizzo di ritorno" per una eventuale risposta

Demultiplexing senza connsessione

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

```
mySocket =
 socket(AF_INET, SOCK_DGRAM)
mySocket.bind(('',9157))
```

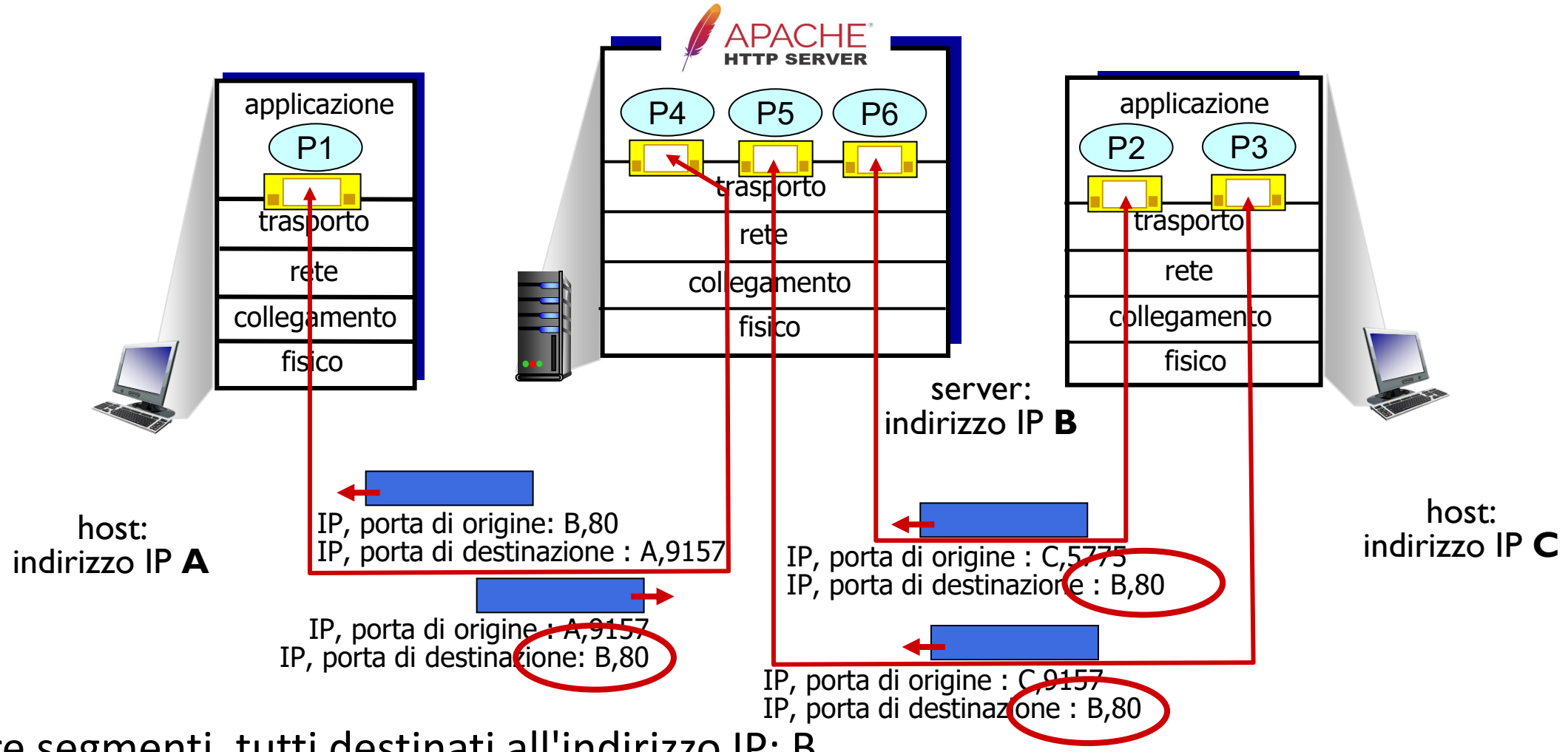
```
mySocket =
  socket(AF_INET, SOCK_DGRAM)
mySocket.bind((myaddr,5775))
```



Demultiplexing orientato alla connessione

- la socket TCP è identificata da quattro parametri:
  - indirizzo IP di origine
  - numero di porta di origine
  - indirizzo IP di destinazione
  - numero di porta di destinazione
- demux: il lato ricevente usa i *quattro valori (quadrupla)* per inviare il segmento alla socket appropriata
- Un host server crea una *socket passiva* specificando un numero di porta
- La socket passiva viene usata per accettare le richieste di connessione, per ciascuna delle quali verrà creata una nuova *socket connessa* (con la medesima porta e indirizzo IP locale, ma diversa porta e indirizzo remoto, discriminando pertanto le socket connesse di client diversi)

Demultiplexing orientato alla connessione



Tre segmenti, tutti destinati all'indirizzo IP: B,
porta di destinazione: 80: ne viene fatto il demultiplexing verso socket *differenti*

Riassunto

- Multiplexing, demultiplexing: basato sui valori dei campi dell'intestazione del segmento o del datagramma
- **UDP:** demultiplexing usando (solo) il numero di porta e indirizzo IP di destinazione
- **TCP:** demultiplexing usando la quadrupla di valori: indirizzi di origine e di destinazione, e numeri di porta
- Multiplexing/demultiplexing avviene a *tutti* i livelli (ogni volta che entità diverse vogliono usare i servizi del protocollo di livello inferiore)

Nella creazione di una socket abbiamo usato `''` *(che in Python equivale a '0.0.0.0' nel caso di IPv4) per indicare qualunque indirizzo IP dell'host; tuttavia, avremmo potuto specificare uno in particolare.*

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di trasporto (parte2)

dr. Manuel Fiorelli
manuel.fiorelli@uniroma2.it
https://art.uniroma2.it/fiorelli

Basate sulle slide del libro di testo:
https://gaia.cs.umass.edu/kurose_ross/ppt.php

Capitolo 3: tabella di marcia

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

UDP: User Datagram Protocol

- protocollo di trasporto di Internet "senza fronzoli"
- servizio di consegna “best effort” (massimo sforzo), i segmenti UDP possono essere:
  - perduti
  - consegnati fuori sequenza all'applicazione
- *senza connessione*
  - no handshaking tra mittente e destinatario UDP
  - ogni segmento UDP è gestito indipendentemente

Perché esiste UDP?

- nessuna connessione stabilita (che potrebbe aggiungere ritardo)
- semplice: nessuno stato di connessione nel mittente e nel destinatario (perciò un server può gestire più client)
- intestazioni di segmento corte
- senza controllo della congestione
  - UDP può sparare dati a raffica!
- controllo più preciso a livello di applicazione su quali dati sono inviati e quando (utile per applicazioni con requisiti di velocità di invio minima e ritardi limitati, ma che sono tolleranti alle perdite)

UDP: User Datagram Protocol

- utilizzo di UDP:
  - applicazioni per lo streaming multimediale (tolleranti alle perdite, sensibili alla banda)
  - DNS
  - SNMP
  - HTTP/3
- trasferimento affidabile con UDP (ad esempio, HTTP/3):
  - aggiungere affidabilità a livello di applicazione
  - aggiungere controllo della congestione a livello di applicazione

UDP: User Datagram Protocol [RFC 768]

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

User Datagram Protocol

Introduction

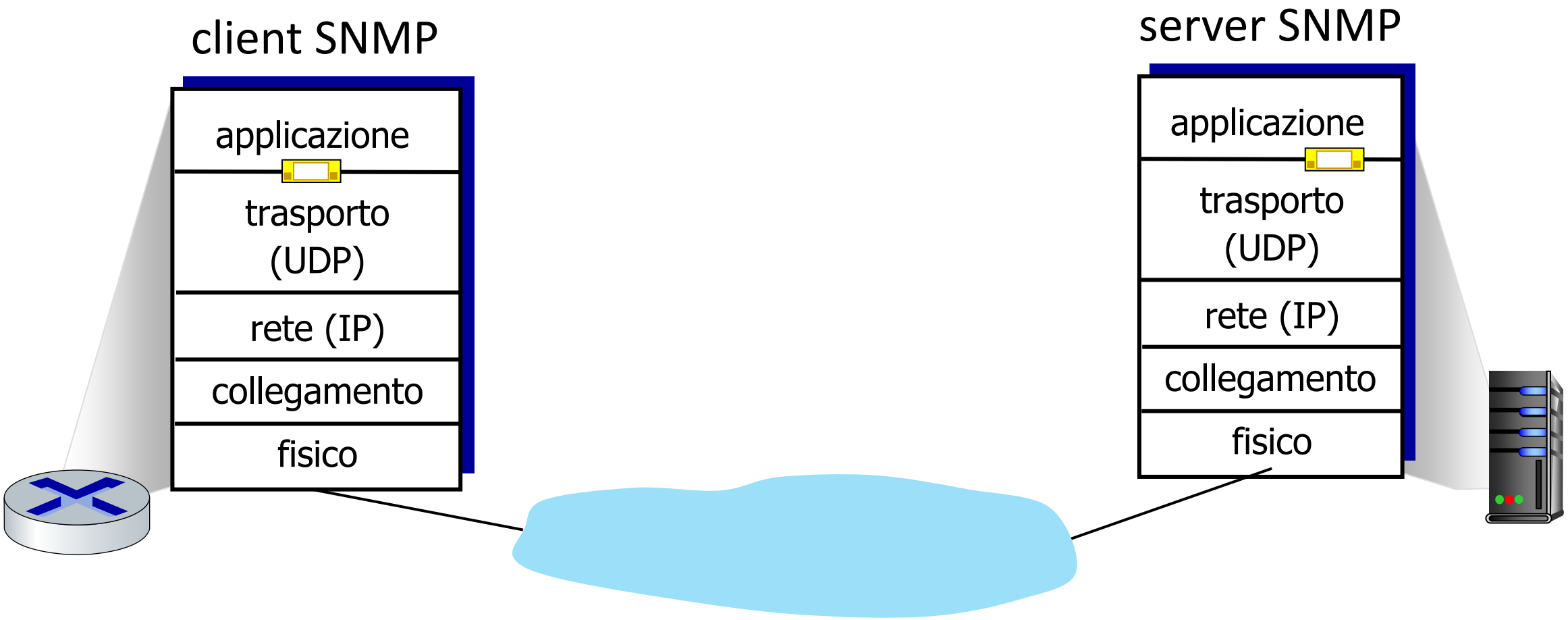
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

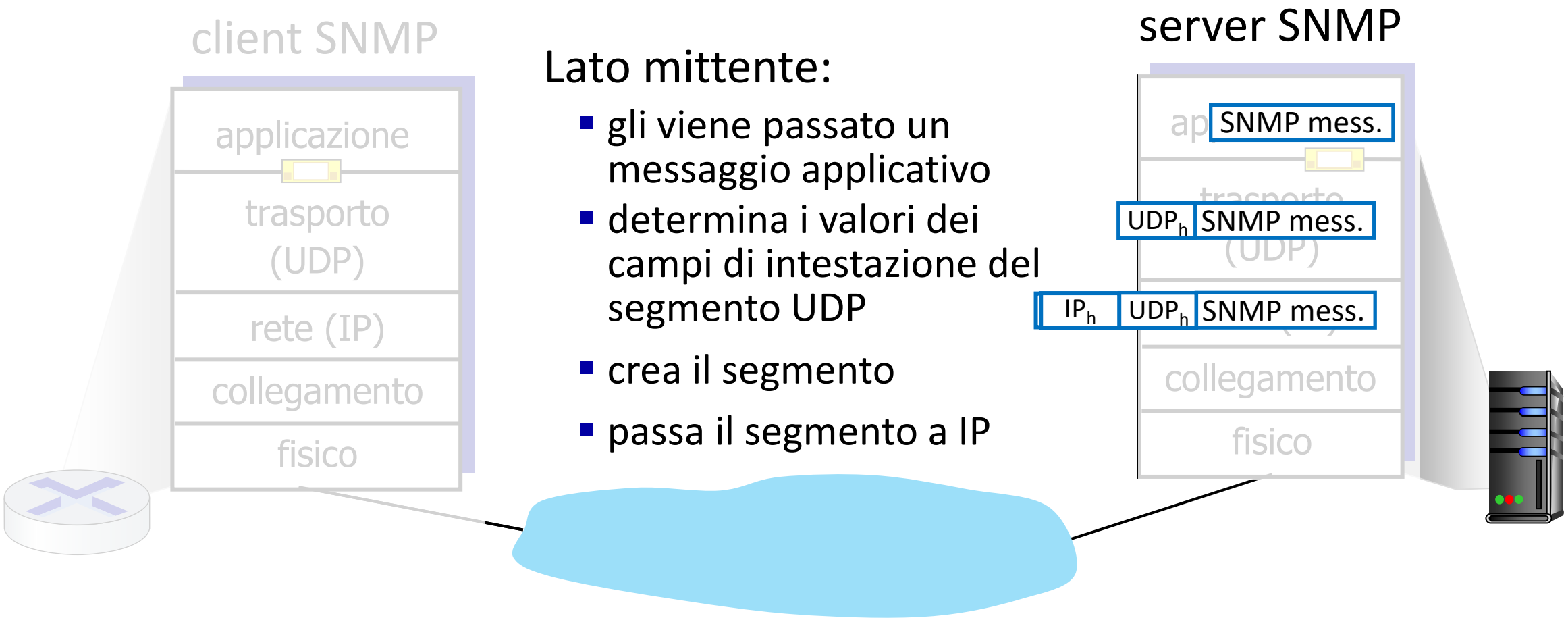
Format

```
 0      7 8     15 16    23 24    31
+--------+--------+--------+--------+
|     Source      |   Destination   |
|      Port       |      Port       |
+--------+--------+--------+--------+
|                 |                 |
|     Length      |    Checksum     |
+--------+--------+--------+--------+
|
|          data octets ...
+---------------- ...
```

UDP: azioni del livello di traporto



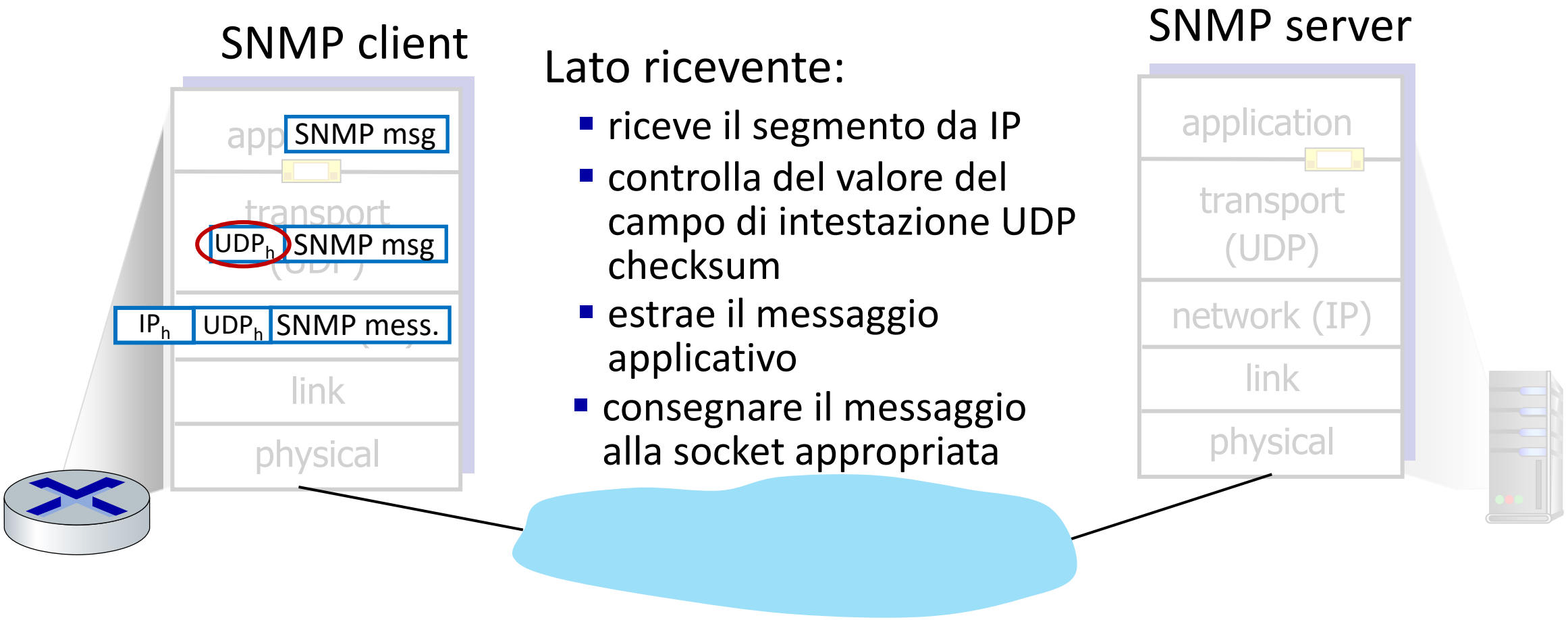
UDP: azioni del livello di traporto



Lato mittente:

- gli viene passato un messaggio applicativo
- determina i valori dei campi di intestazione del segmento UDP
- crea il segmento
- passa il segmento a IP

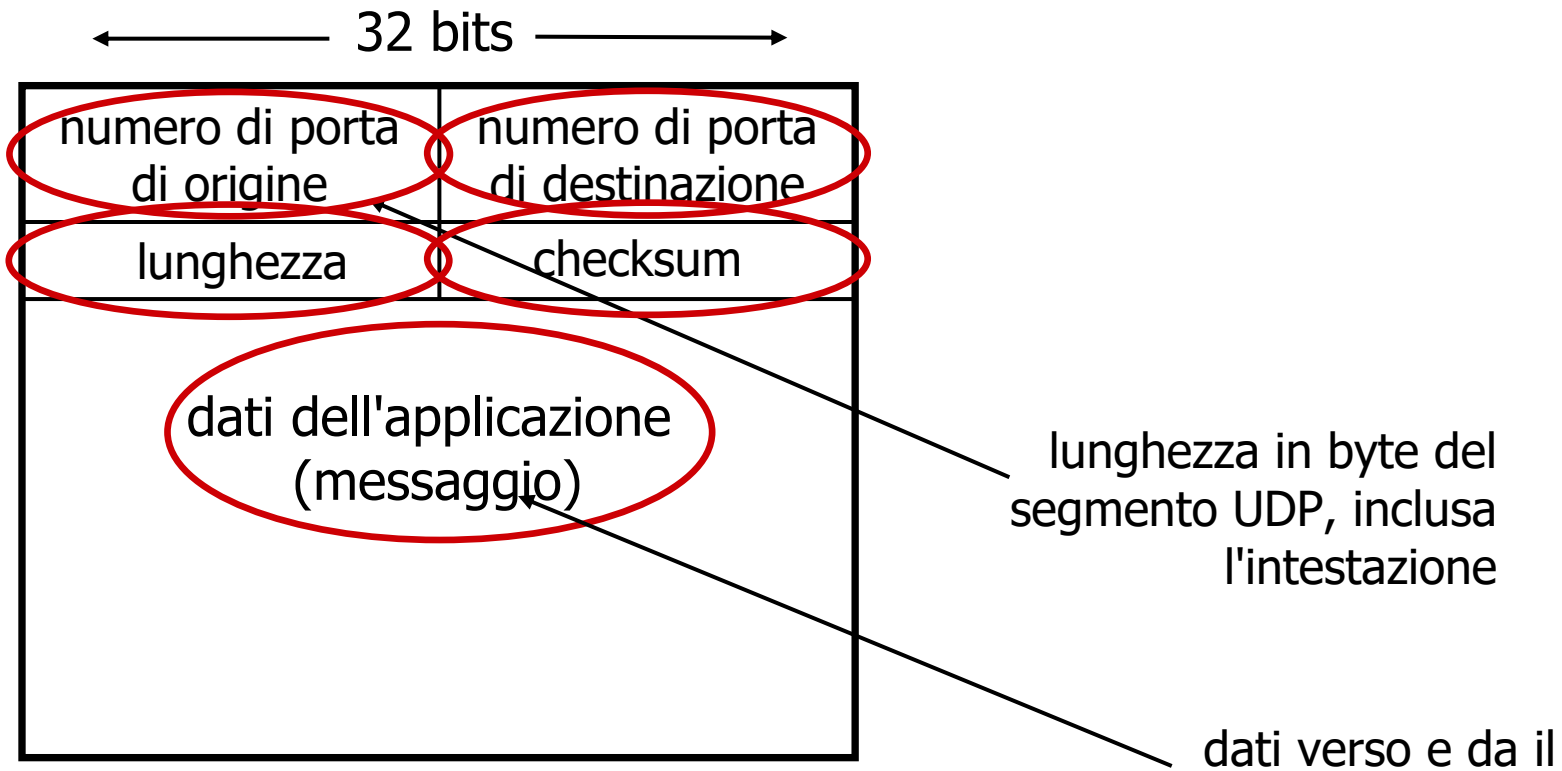
UDP: azioni del livello di traporto



Lato ricevente:

- riceve il segmento da IP
- controlla del valore del campo di intestazione UDP checksum
- estrae il messaggio applicativo
- consegnare il messaggio alla socket appropriata

Struttura dei segmenti UDP



Struttura dei segmenti UDP

Checksum UDP

Obiettivo: rilevare gli "errori" (*bit alternati*) nel segmento trasmesso

	1º numero		2º numero		somma		
messaggio:	5		6		11		
trasmesso:	5	+	6	+	-11	=	0
						≠	
ricevuto:	4	+	6	+	-11	=	-1

errore rilevato dal ricevente

Checksum Internet [RFC 1071, RFC 1624]

Obiettivo: rilevare gli "errori" (*bit alternati*) nel segmento trasmesso

mittente:

- tratta il contenuto del segmento come una sequenza di interi da 16 bit (inclusi i campi dell'intestazione UDP e gli indirizzi IP)
- checksum: complemento a 1 della somma (in complemento a 1) della sequenza di interi a 16 bit (considerando il campo checksum uguale a 0)
- pone il valore della checksum nel campo checksum del segmento UDP

ricevente:

- calcola la checksum allo stesso modo del mittente (ma sommando anche il valore ricevuto del checksum)
- Il risultato è costituito da tutti bit 1 (-0 nell'aritmetica del complemento a 1)?
  - Sì - nessun errore rilevato. Ma potrebbero esserci errori nonostante questo? Lo scopriremo più avanti....
  - No - errore rilevato
- Altre implementazioni verificano la checksum calcolandola (allo stesso modo del mittente) e confrontandola col valore ricevuto

Checksum Internet: un esempio

esempio: sommare due interi da 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
a capo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	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, un riporto dal bit più significativo deve essere sommato al risultato

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Checksum Internet: protezione debole

esempio: sommare due interi da 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
a capo	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	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

0 1
1 0

Nonostante i numeri siano cambiati (bit alternati), *nessun* cambiamento nella checksum!

Rappresentazione interi relativi in complemento a 1

n bit

a_{n-1}	a_{n-2}	a_{n-3}	...	a_2	a_1	a_0

Se a_{n-1} = 0, rappresenta un numero positivo (effettivamente di n-1 bit):

$$\sum_{i=0}^{n-2} a_i 2^i$$

Se a_{n-1} = 1, rappresenta un numero negativo, ottenuto invertendo tutti i bit:

$$-\sum_{i=0}^{n-2} \overline{a_i} 2^i$$

Possiamo rappresentare i numeri *x* nell'intervallo

$$-(2^{n-1} - 1) \leq x \leq 2^{n-1} - 1$$

due rappresentazioni dello zero

a_2	a_1	a_0	**Valore rappresentato in complemento a 1**	**Valore rappresentato senza segno**
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	-3	4
1	0	1	-2	5
1	1	0	-1	6
1	1	1	-0	7

provate a sommare -2 e -1 per capire l'importanza del bit di riporto!

Se sommiamo un numero e il suo complemento a 1, otteniamo sempre tutti 1

Rappresentazione interi relativi in complemento a 1

n bit

a_{n-1}	a_{n-2}	a_{n-3}	...	a_2	a_1	a_0

Se a_{n-1} = 1, rappresenta un numero negativo, ottenuto invertendo tutti i bit:

$$-\sum_{i=0}^{n-2} \overline{a_i} 2^i$$

Cioè

$$-\sum_{i=0}^{n-2} \overline{a_i} 2^i = -\sum_{i=0}^{n-2} (1 - a_{i)} 2^i$$

$$= -\sum_{i=0}^{n-2} 2^i + \sum_{i=0}^{n-2} a_i 2^i$$

$$= (-2^{n-1}+1) + \sum_{i=0}^{n-2} a_i 2^i$$

a_2	a_1	a_0	Valore rappresentato in complemento a 1	Valore rappresentato senza segno
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	-3	4
1	0	1	-2	5
1	1	0	-1	6
1	1	1	-0	7

Rappresentazione interi relativi in complemento a 1

n bit

a_{n-1}	a_{n-2}	a_{n-3}	...	a_2	a_1	a_0

Rappresenta il numero relativo

$$a_{n-1}(-2^{n-1}+1)+\sum_{i=0}^{n-2} a_i 2^i$$

a_2	a_1	a_0	**Valore rappresentato in complemento a 1**	**Valore rappresentato senza segno**
0	0	0	0	0
0	0	1	1	1
0	1	0	2	2
0	1	1	3	3
1	0	0	-3	4
1	0	1	-2	5
1	1	0	-1	6
1	1	1	-0	7

Riassunto: UDP

- protocollo “senza fronzoli” :
  - i segmenti possono essere persi, consegnati fuori sequenza
  - servizio best effort: “invia e spera per il meglio”
- UDP ha i propri vantaggi:
  - nessun bisogno di setup/handshaking (non si incorre in alcun RTT)
  - può funzionare quando il servizio di rete è compromesso
  - aiuta con l'affidabilità (checksum)
- costruire funzionalità aggiuntive su UDP nel livello di applicazione (ad esempio, HTTP/3)

Capitolo 3: tabella di marcia

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

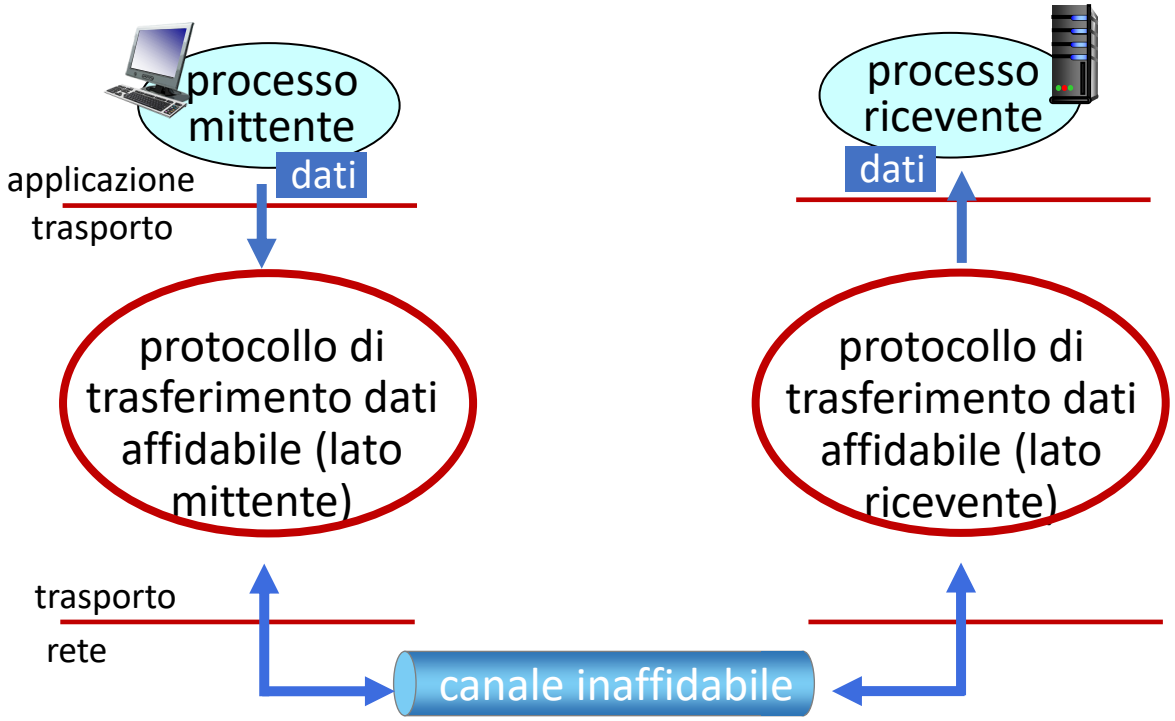
Principi del trasferimento dati affidabile

astrazione di un servizio affidabile

per il momento ci concentriamo su un canale unidirezionale, per l'invio affidabile di dati da un mittente a un destinatario.

Principi del trasferimento dati affidabile

astrazione di un servizio affidabile

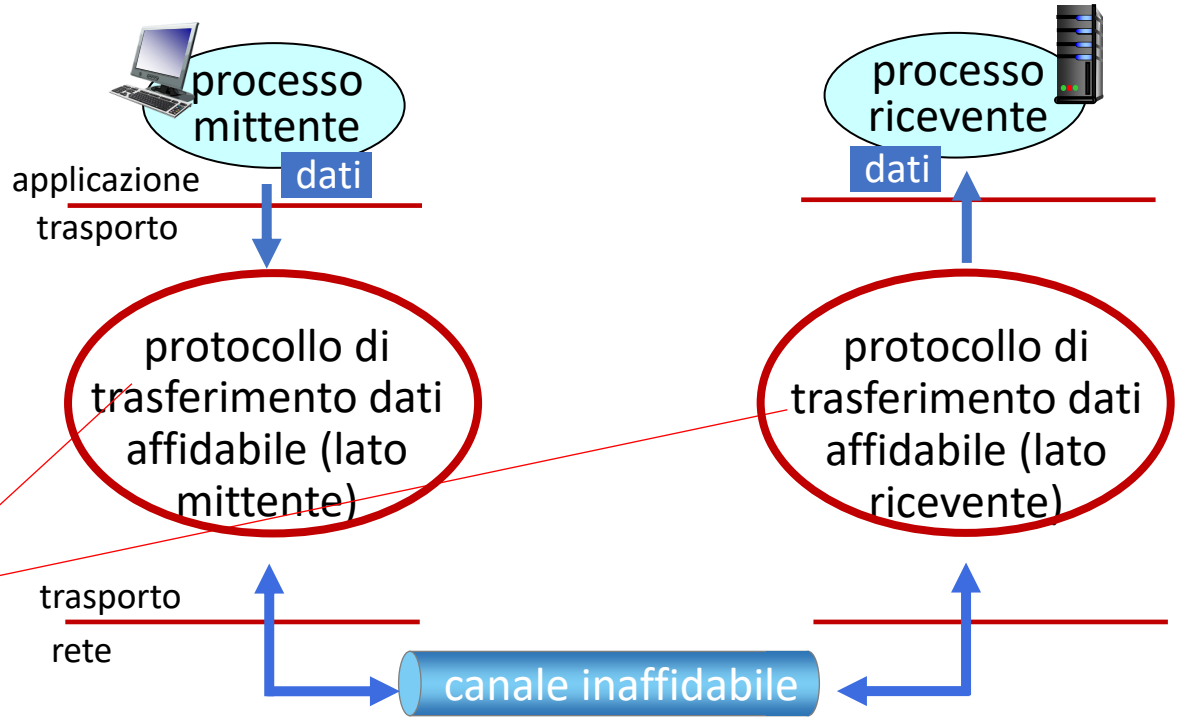


nonostante il nostro sia a livello astratto un canale di comunicazione unidirezionale, le due parti dentro al livello di trasporto devono comunicare in entrambi i versi usando il canale *inaffidabile* fornito dal livello di rete.

Implementazione di un servizio affidabile

Principi del trasferimento dati affidabile

la complessità del protocollo di trasferimento dati affidabile (reliable data transfer, rdt) dipende (fortemente) dalle caratteristiche del canale inaffidabile (perdite, corruzione, riordino dei dati?)

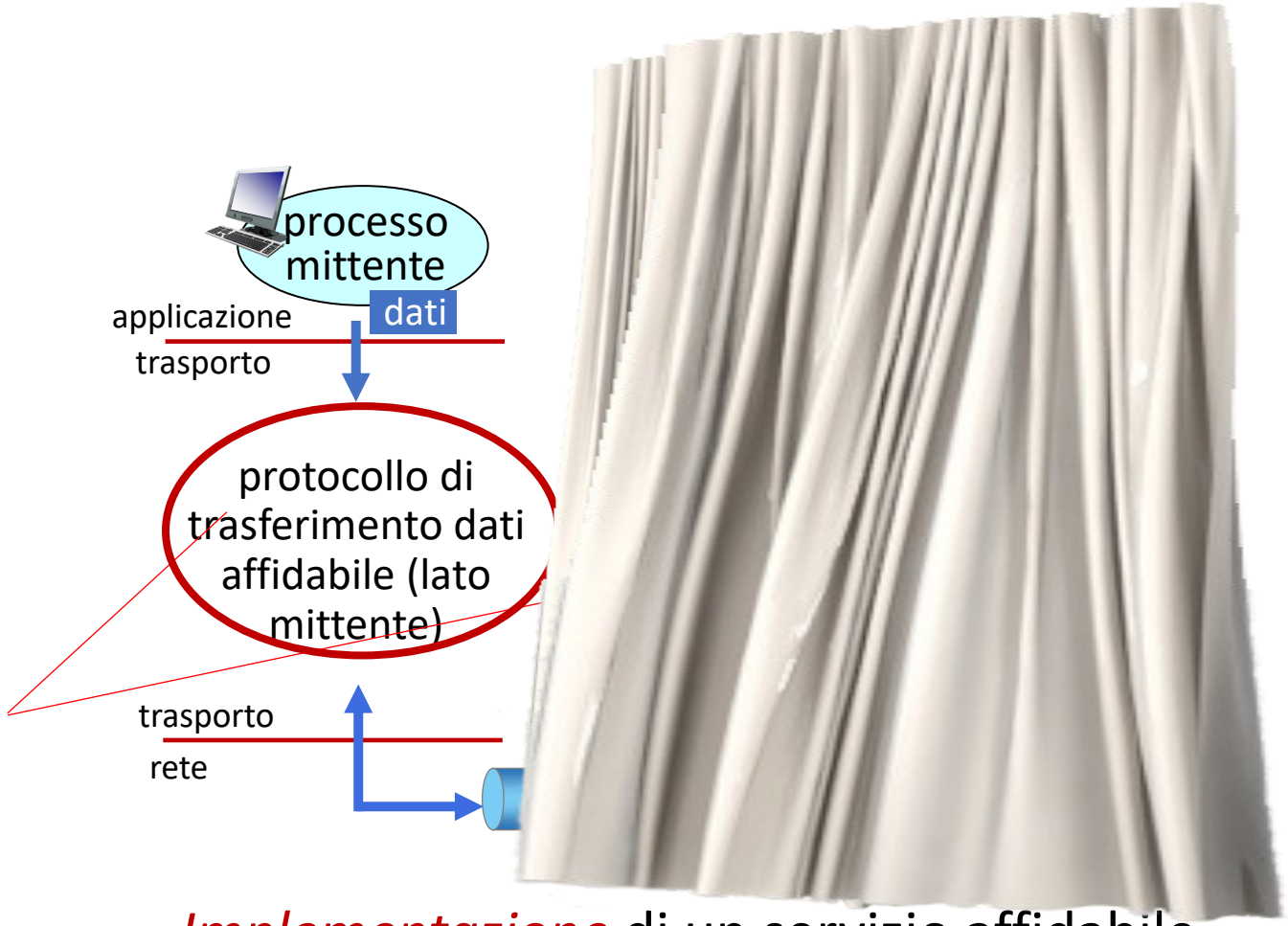


Implementazione di un servizio affidabile

Principi del trasferimento dati affidabile

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

- a meno che non venga comunicato attraverso un messaggio

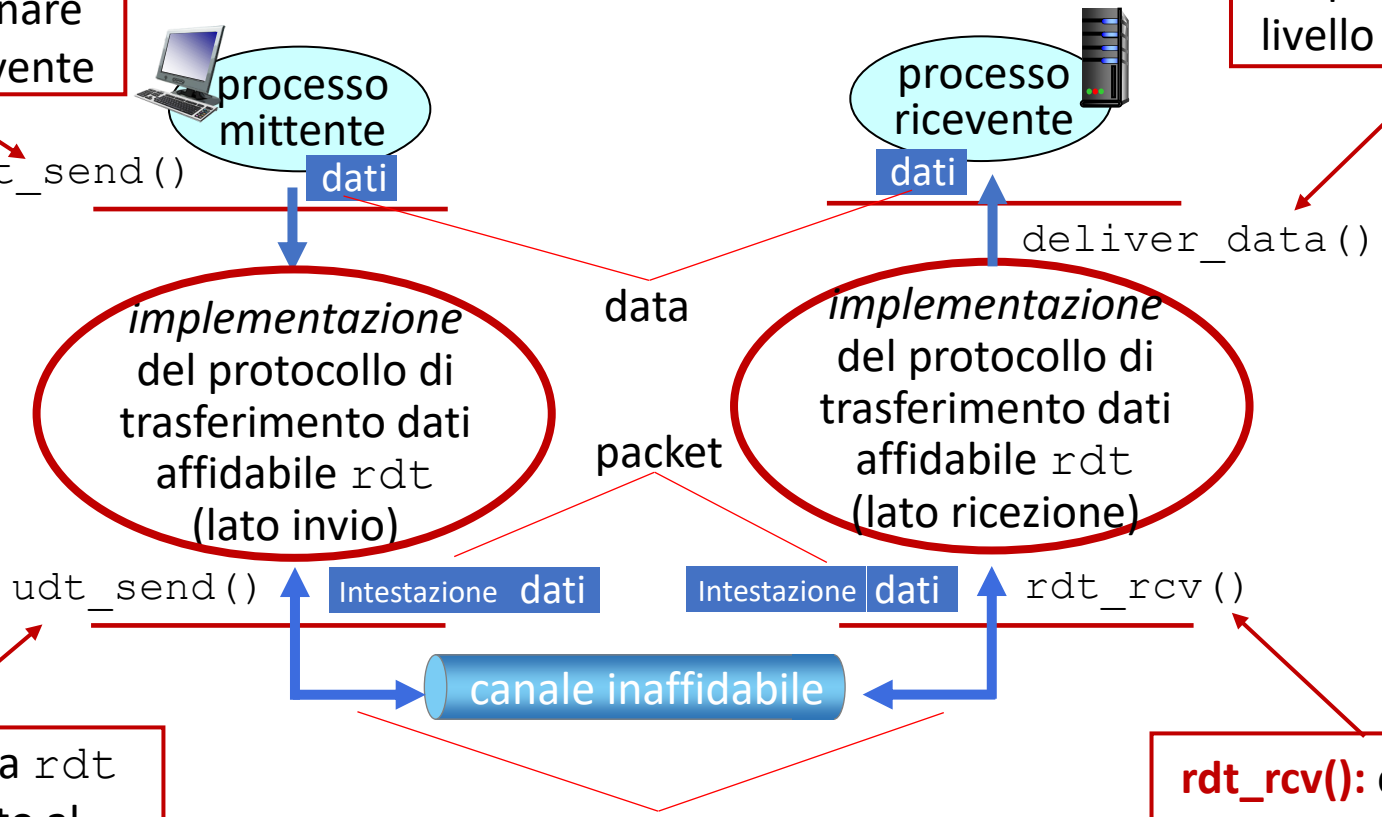


Implementazione di un servizio affidabile

Trasferimento dati affidabile (rdt): interfacce

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

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



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

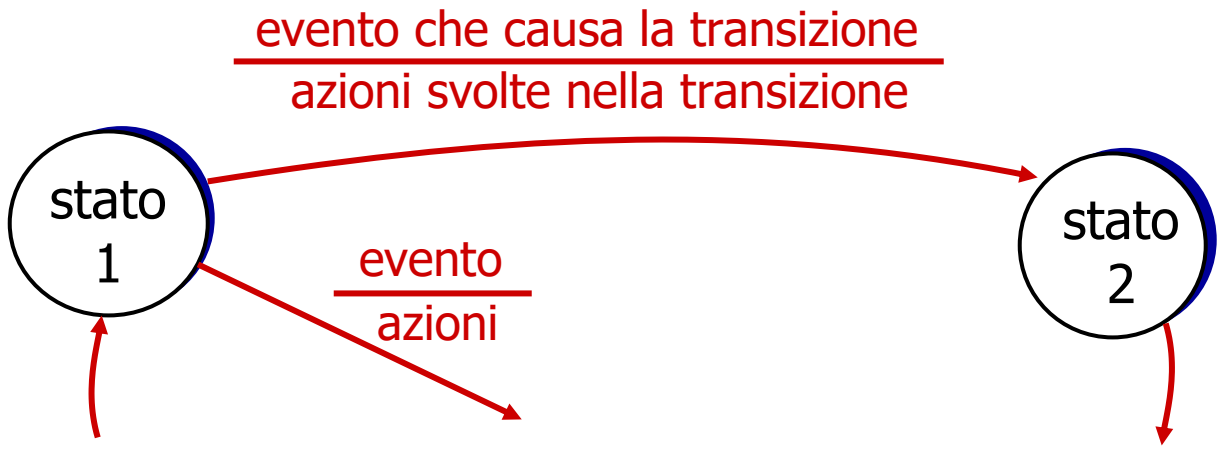
comunicazione bidirezionale tramite un canale inaffidabile

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

Trasferimento dati affidabile: come iniziare

- sviluppperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile, reliable data transfer protocol (`rdt`)
- considereremo soltanto i trasferimenti dati unidirezionali
  - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- utilizzeremo macchine a stati finiti, finite state machines (FSM) per specificare il mittente e il ricevente

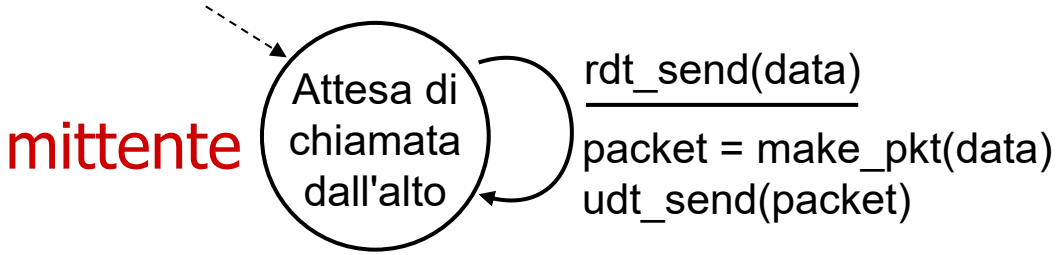
stato: lo stato successivo a questo è determinato unicamente dall'evento successivo

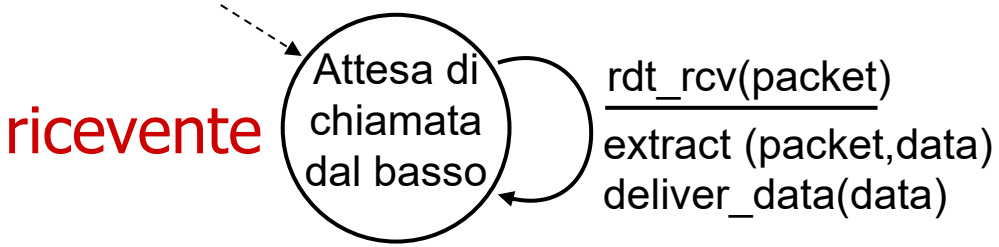


Una specifica formale di un protocollo è meno soggetta a fraintendimenti e incompletezza rispetto a una specifica testuale. Sulla base di una specifica formale è inoltre possibile **provare** che il protocollo specificato abbia determinate proprietà.

rdt1.0: trasferimento affidabile su canale affidabile

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





rdt2.0: canale con errori nei bit

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

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

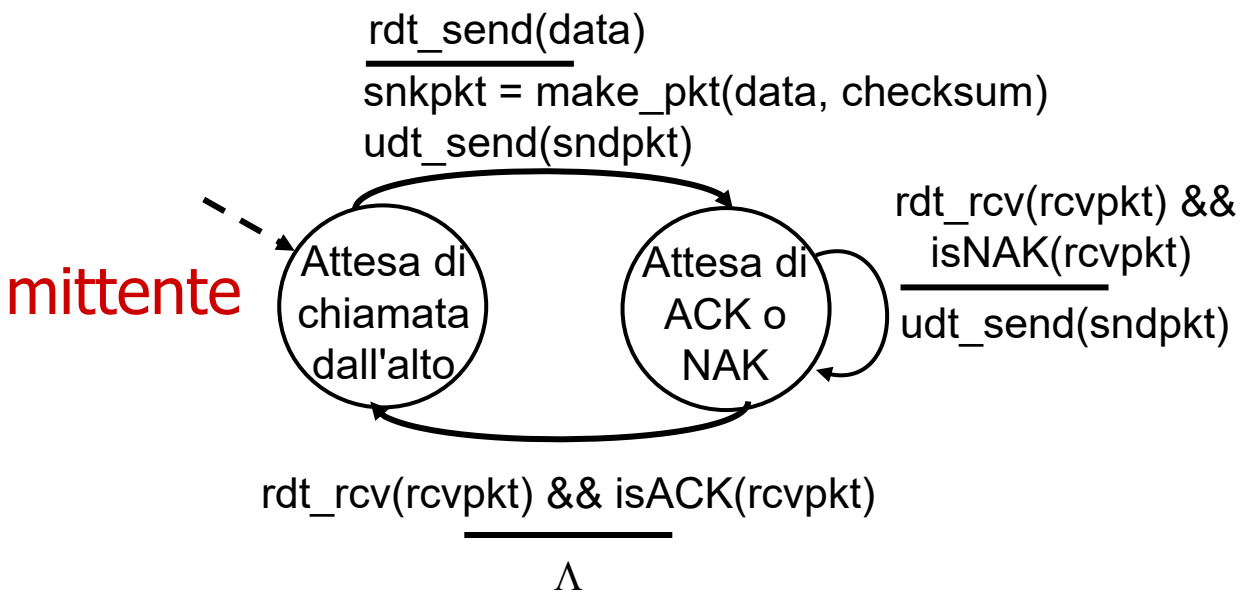
rdt2.0: canale con errori nei bit

- il canale sottostante può invertire (*flip*) i bit nel pacchetto
  - checksum per rilevare gli errori nei bit (*error detection*)
- *la* domanda: come recuperare (*recover*) dagli errori?
  - *notifica positiva, acknowledgements (ACKs):* il ricevente comunica espressamente al mittente che il pacchetto ricevuto è corretto
  - *notifica negativa, negative acknowledgements (NAKs):* il ricevente comunica espressamente al mittente che il pacchetto contiene errori
  - il mittente *ritrasmette* il pacchetto se riceve un NAK

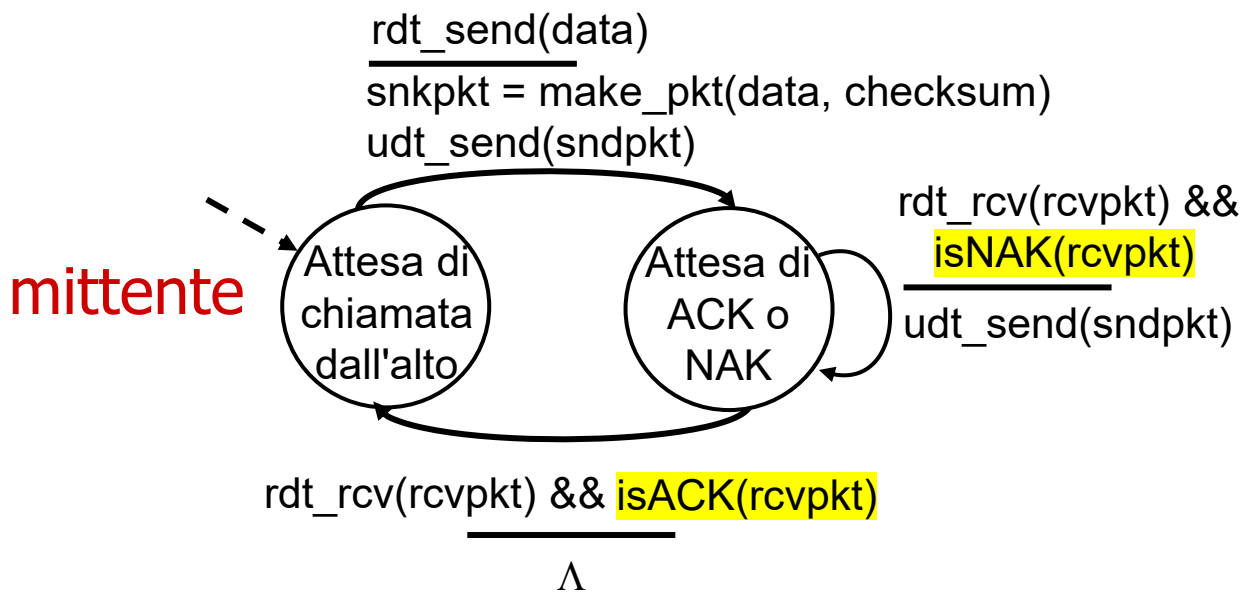
stop and wait

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

rdt2.0: specifica delle FSM



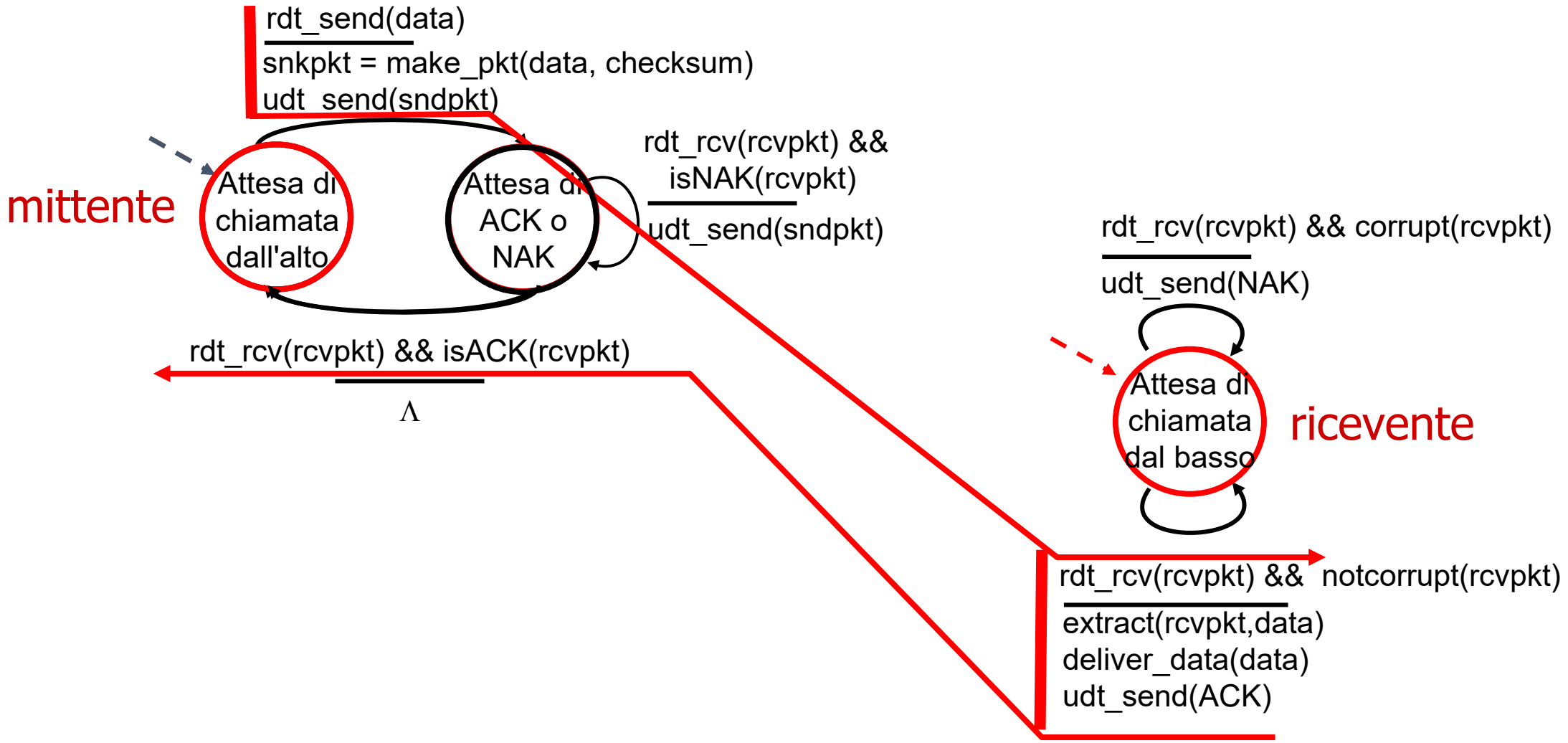
rdt2.0: specifica delle FSM



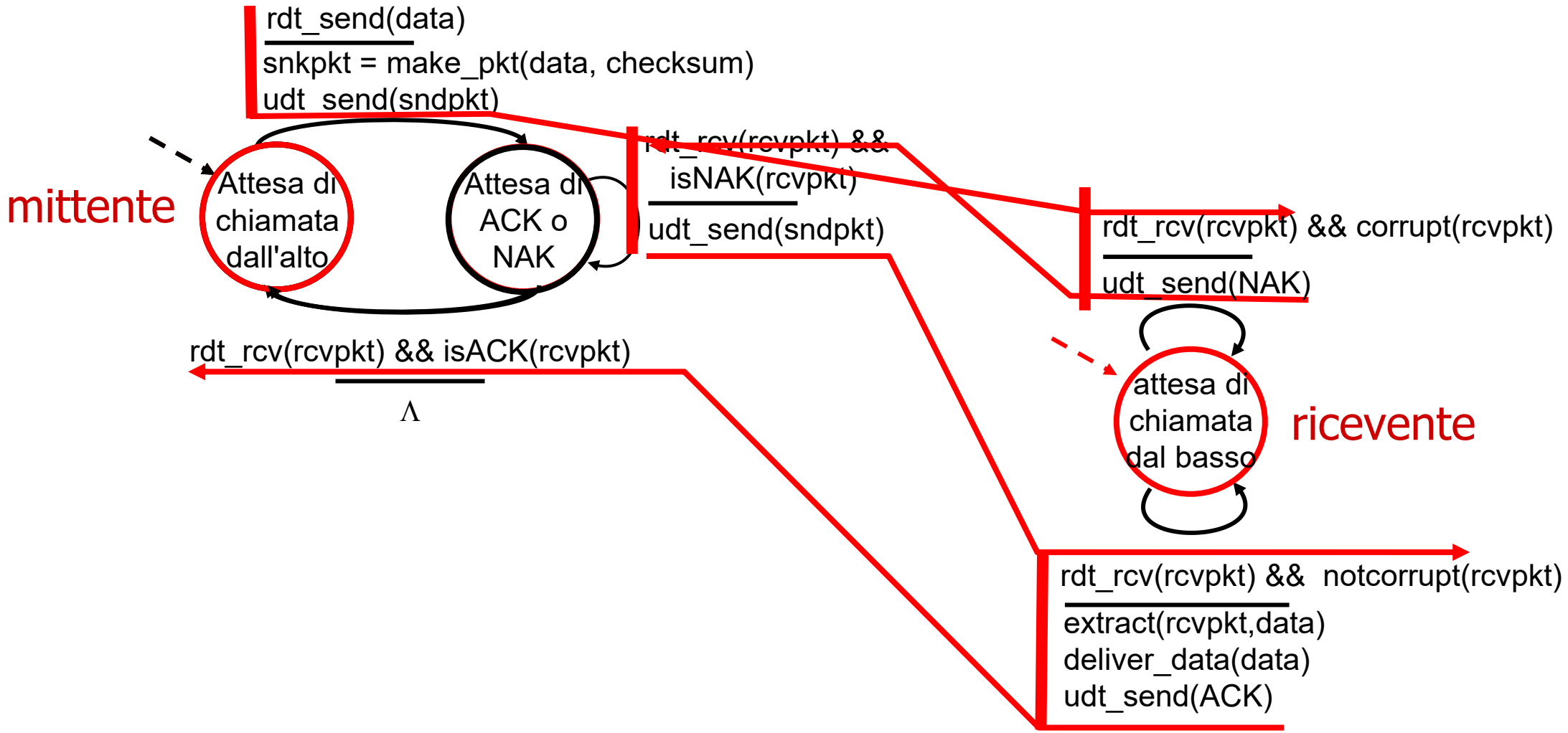
Nota: lo "stato" del destinatario (ha ricevuto correttamente il mio messaggio?) non è noto al mittente a meno che non venga comunicato in qualche modo dal destinatario al mittente.

- ecco perché abbiamo bisogno di un protocollo!

rdt2.0: operazione senza errori



rdt2.0: scenario di errore



rdt2.0 ha un difetto fatale!

Che cosa accade se i pacchetti ACK/NAK sono danneggiati?

- il mittente non sa che cosa sia accaduto
- non basta ritrasmettere: possibili duplicati

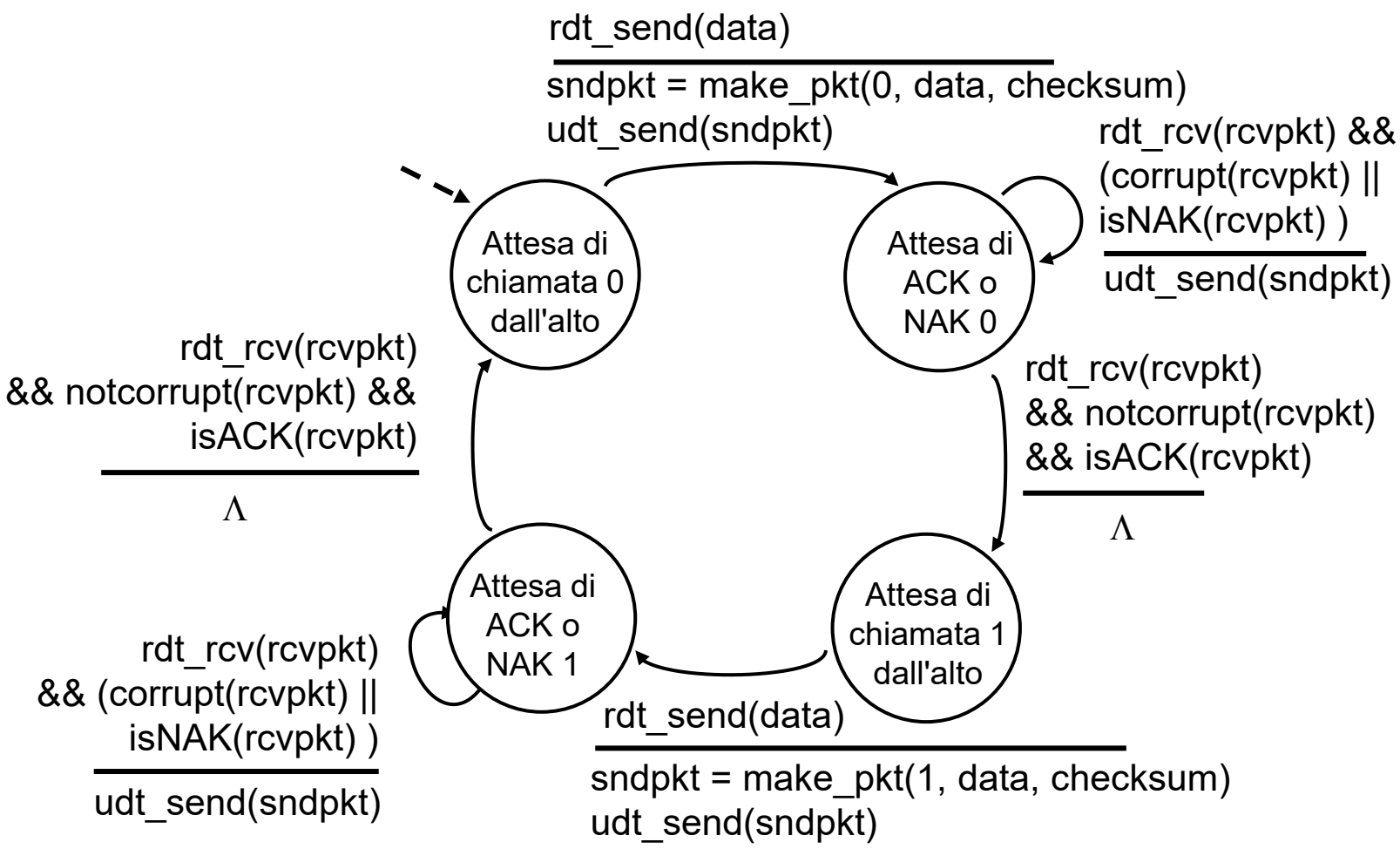
gestione dei duplicati

- il mittente ritrasmette il pacchetto corrente se ACK/NAK è alterato
- il mittente aggiunge un *numero di sequenza* a ogni pacchetto
- il ricevitore scarta (non consegna) il pacchetto duplicato

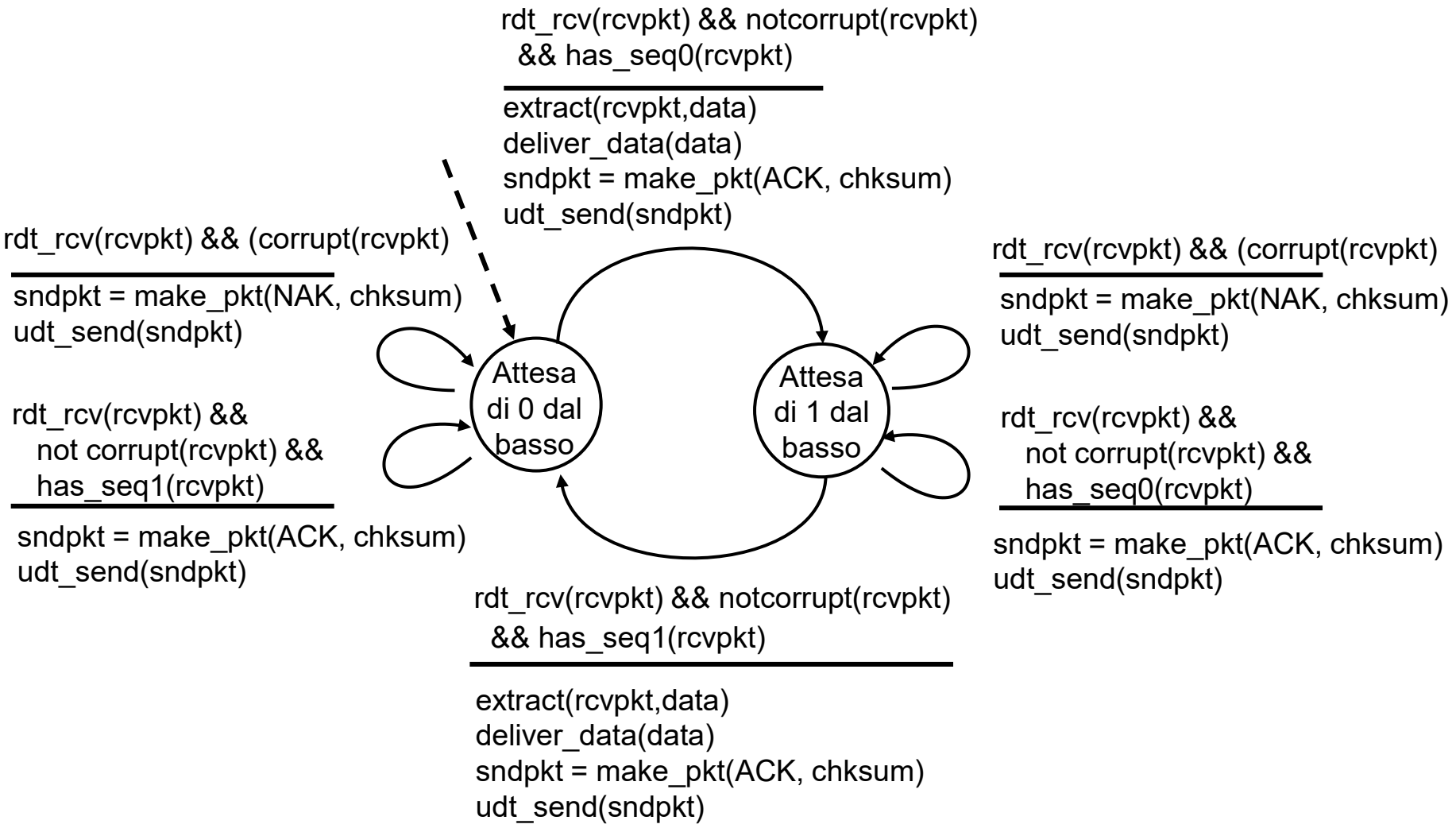
stop and wait

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

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



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



rdt2.1: discussion

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 "ricordarsi" se il pacchetto "corrente" ha numero di sequenza 0 o 1

ricevente:

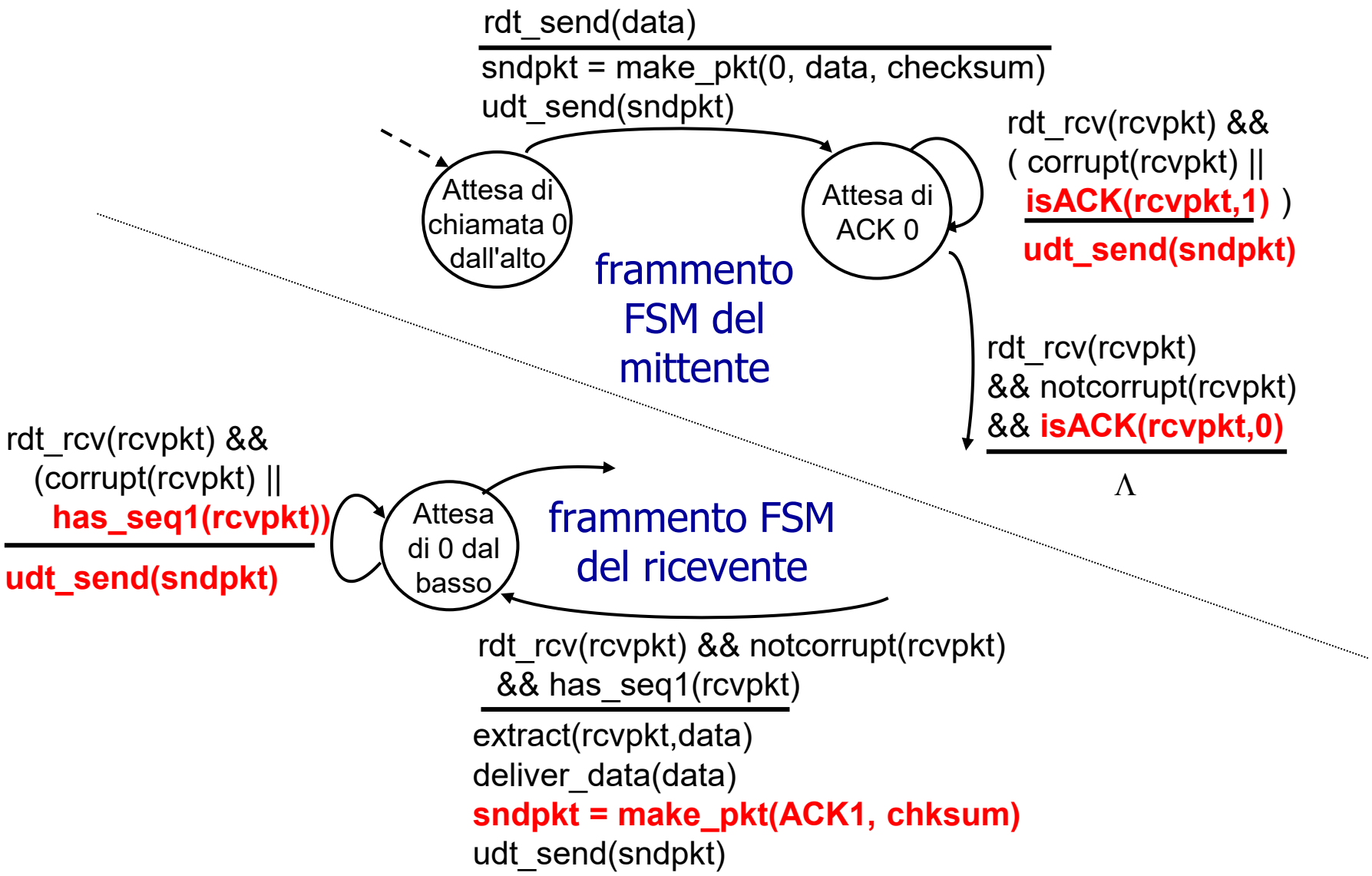
- deve controllare se il pacchetto ricevuto è duplicato
  - lo stato indica se il numero di sequenza previsto è 0 o 1
- nota: il ricevente *non* può sapere se il suo ultimo ACK/NAK è stato ricevuto correttamente dal mittente

rdt2.2: un protocollo senza NAK

- stessa funzionalità di rdt2.1, utilizzando soltanto gli ACK
- al posto 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*

Come vedremo, il protocollo TCP utilizza questo approccio senza NAK.

rdt2.2: frammenti del mittente e del ricevente



rdt3.0: canali con errori *e perdite*

Nuova ipotesi: il canale sottostante può anche *smarrire* pacchetti (dati o ACK)

- checksum, numeri di sequenza, ACK, ritrasmissioni aiuteranno… ma non sono sufficienti

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

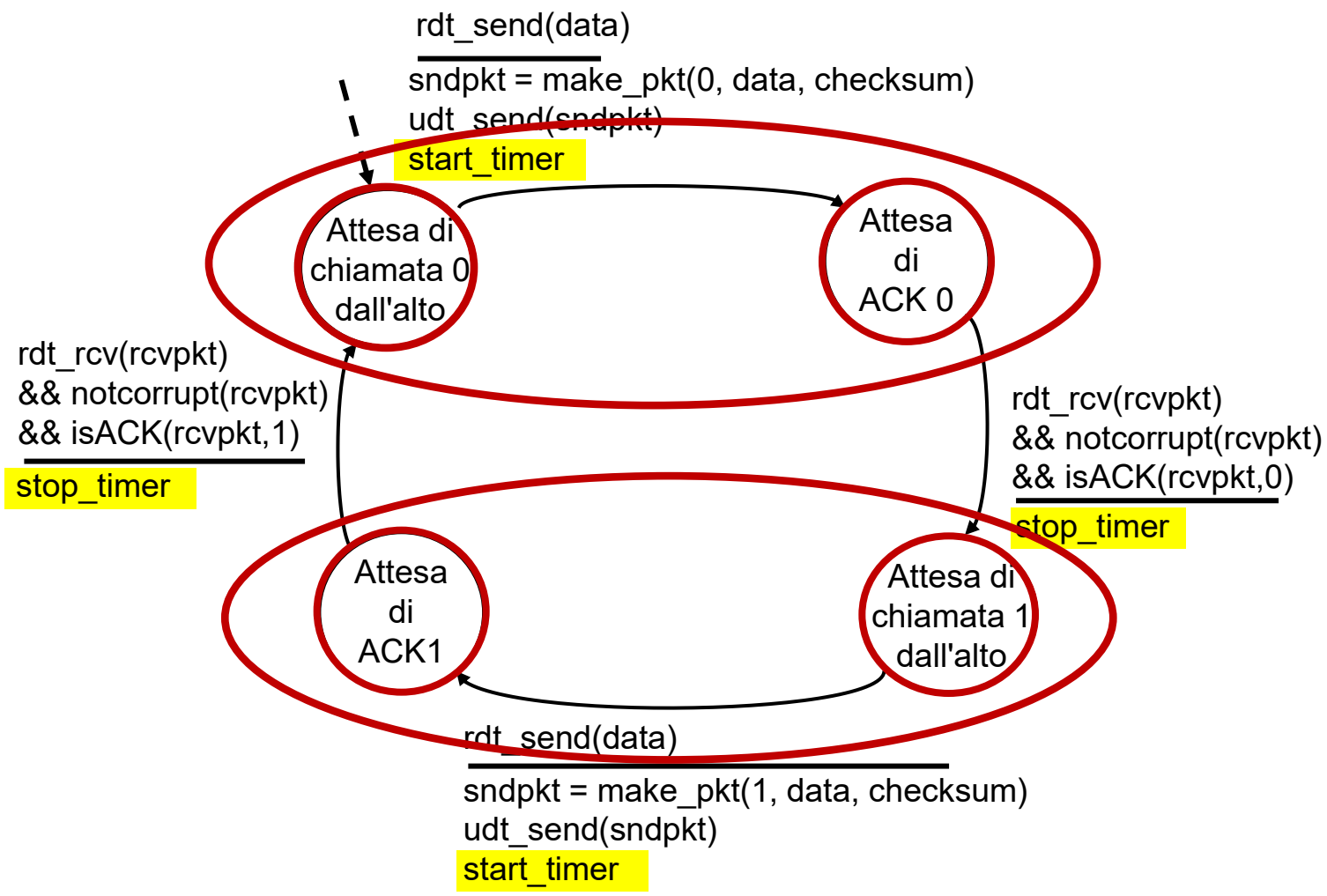
rdt3.0: canali con errori *e perdite*

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

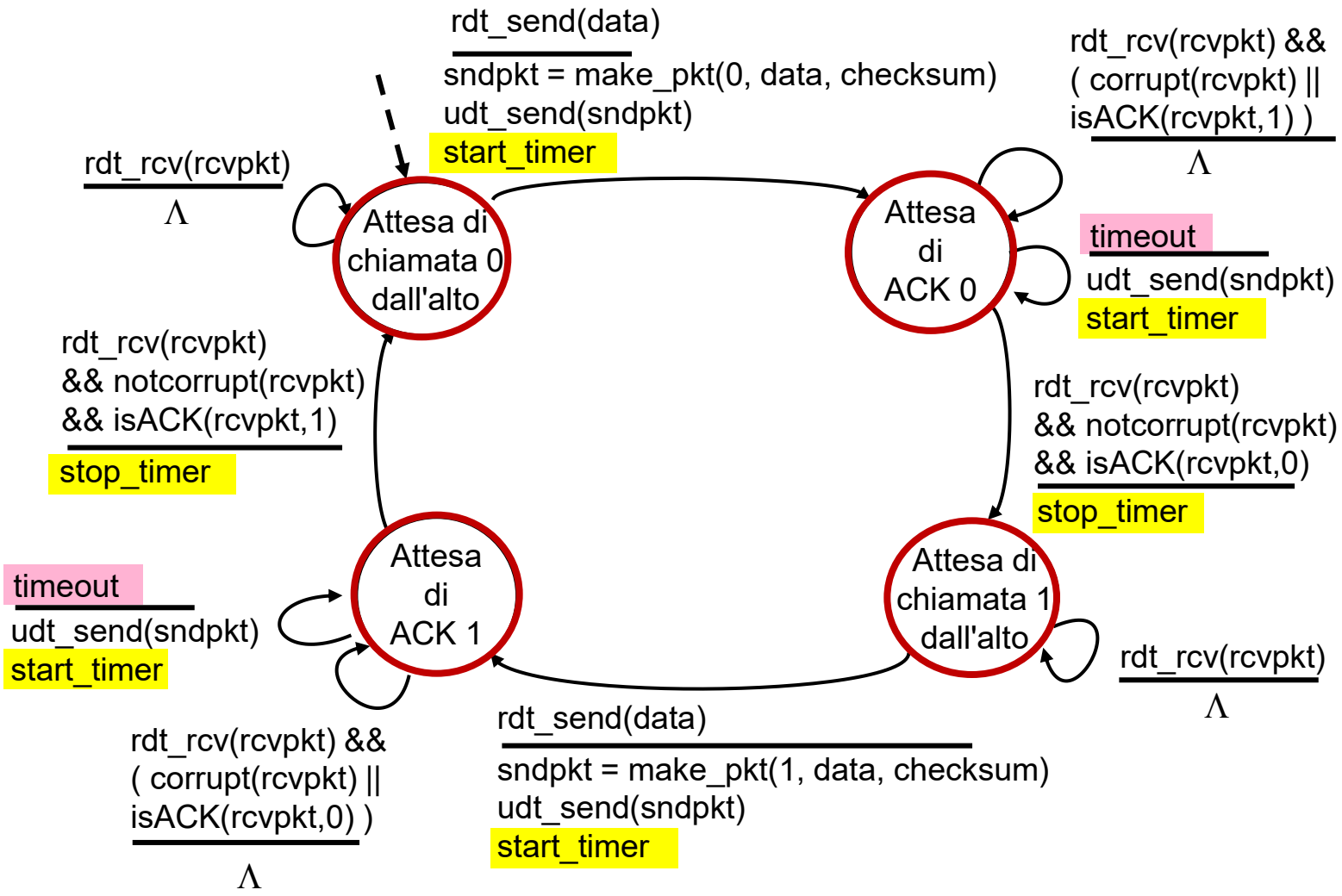
- ritrasmette se non riceve un ACK in questo periodo
- se il pacchetto (o l'ACK) è soltanto in ritardo (non perso)
  - la ritrasmissione sarà duplicata, ma l'uso dei numeri di sequenza gestisce già questo
  - il destinatario deve specificare il numero di sequenza del pacchetto da riscontrare
- utilizzare un timer per il conto alla rovescia (countdown timer) per interrompere (interrupt) dopo un periodo di tempo "ragionevole

timeout

rdt3.0 mittente



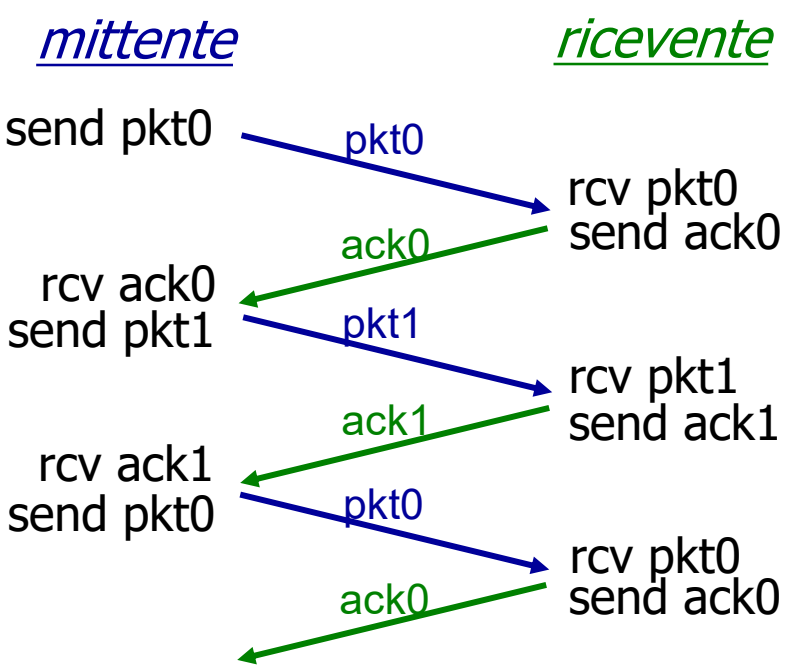
rdt3.0 mittente



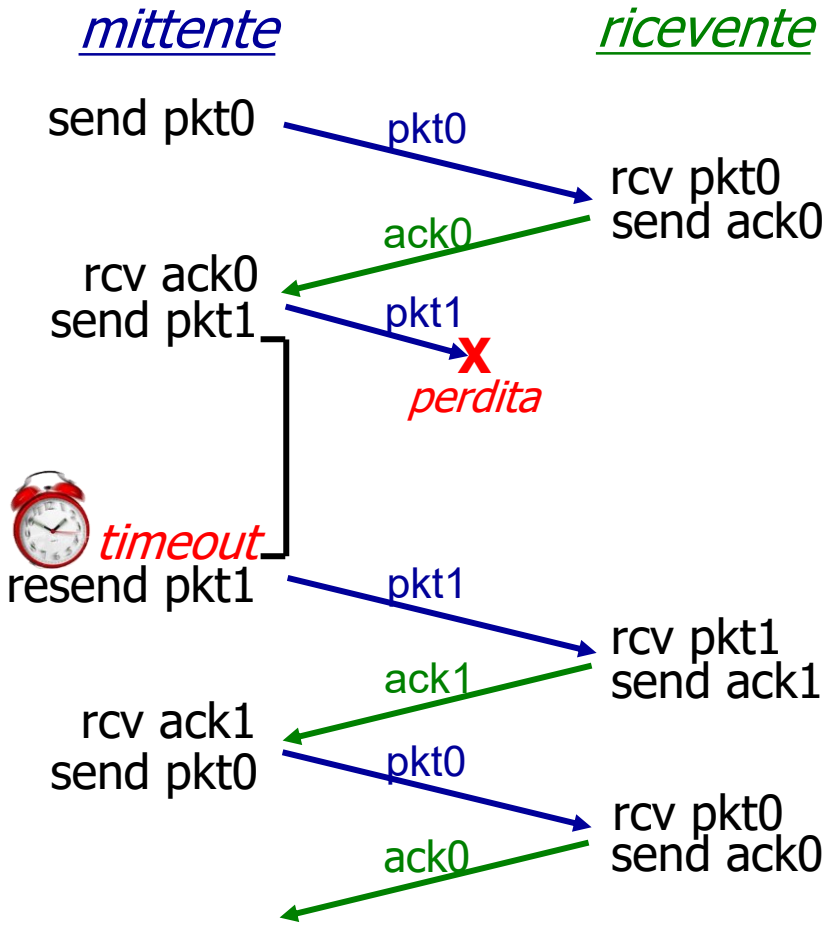
rdt3.0 ricevente

- vedi rdt 2.1

rdt3.0 in azione

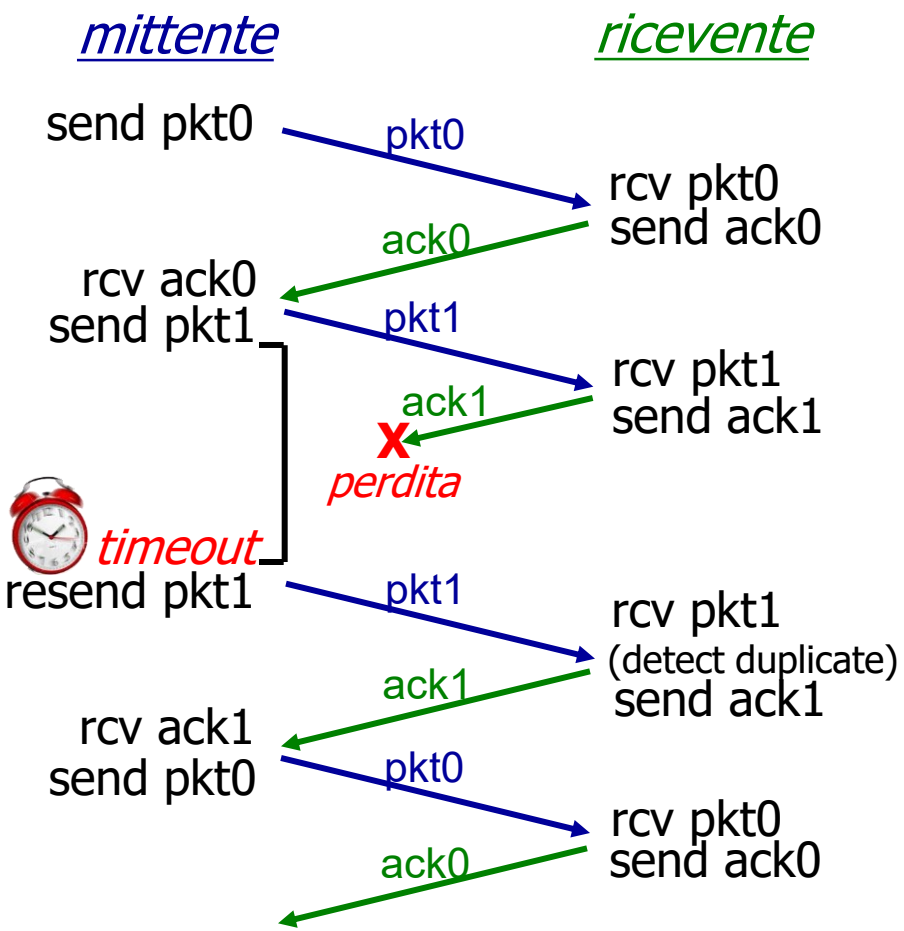


(a) operazioni senza perdita

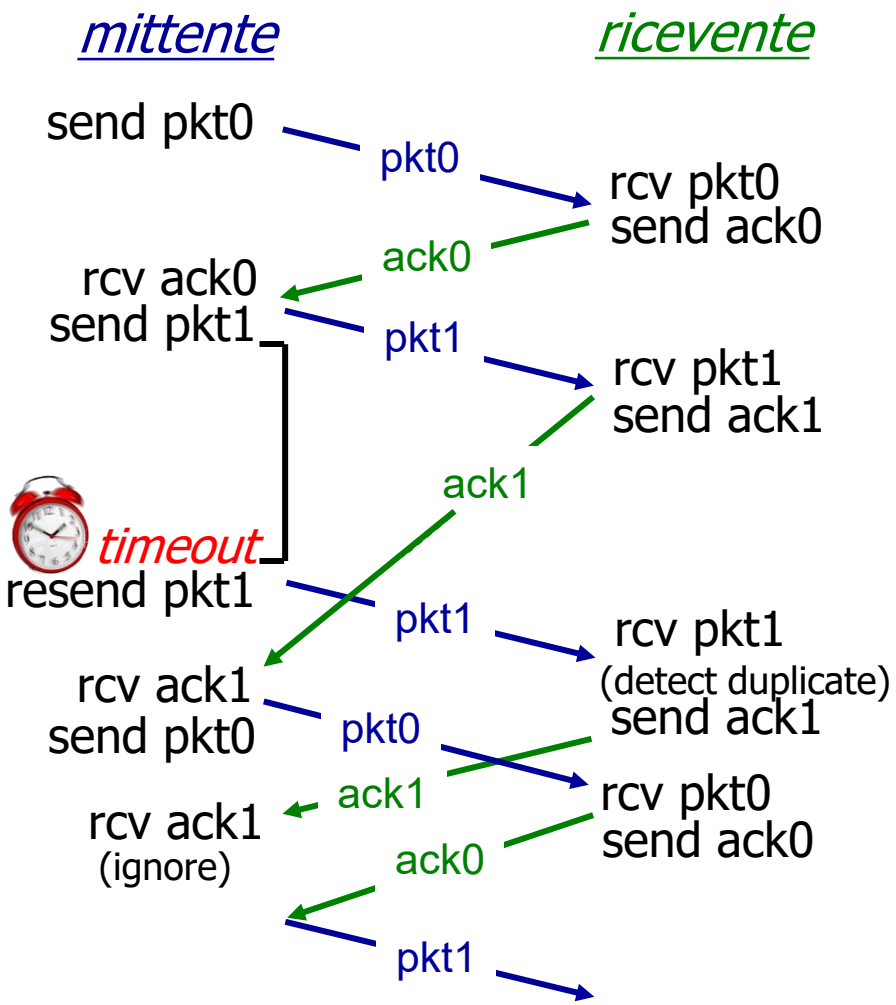


(b) perdita di pacchetto

rdt3.0 in azione



(c) perdita di ACK



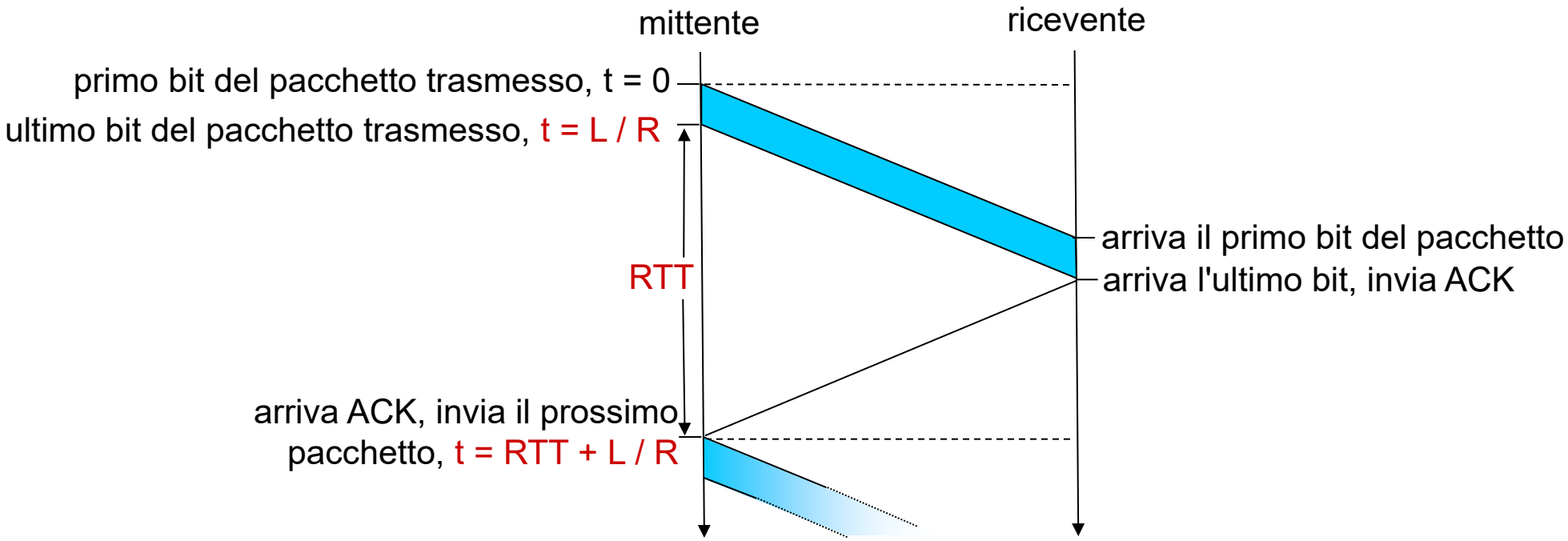
(d) timeout prematuro / ACK ritardato

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

- $U_{mittente}$: *utilizzazione* –la frazione di tempo in cui il mittente è stato effettivamente occupato nell'invio di bit sul canale
- esempio: collegamento da 1 Gbps, ritardo di propagazione 15 ms, pacchetti da 1000 byte (8000 bit)
  - Tempo per trasmettere un pacchetto sul collegamento:

$$D_{trasm} = \frac{L}{R} = \frac{8000\ bits}{10^9\ bits/sec} = 8\ \mu s$$

rdt3.0: funzionamento con stop-and-wait

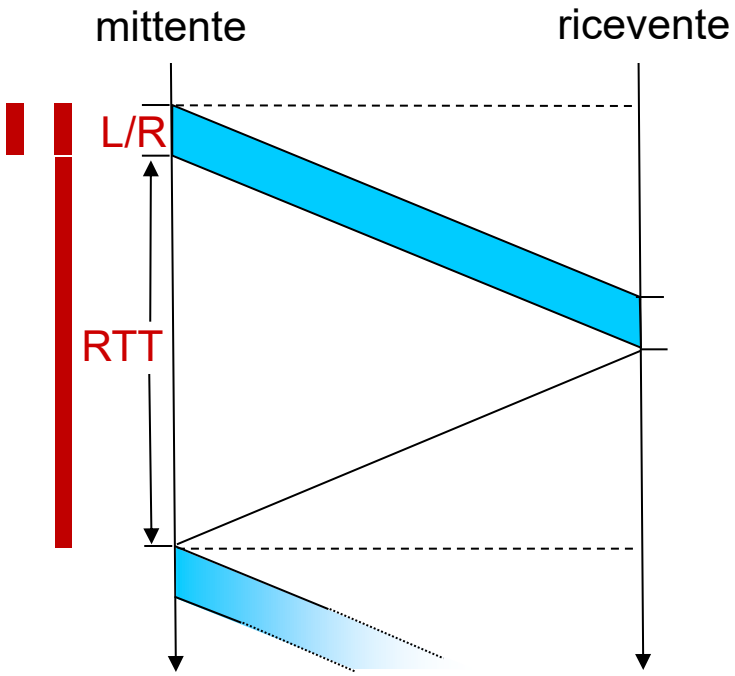


rdt3.0: funzionamento con stop-and-wait

$$U_{mittente} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.000267$$



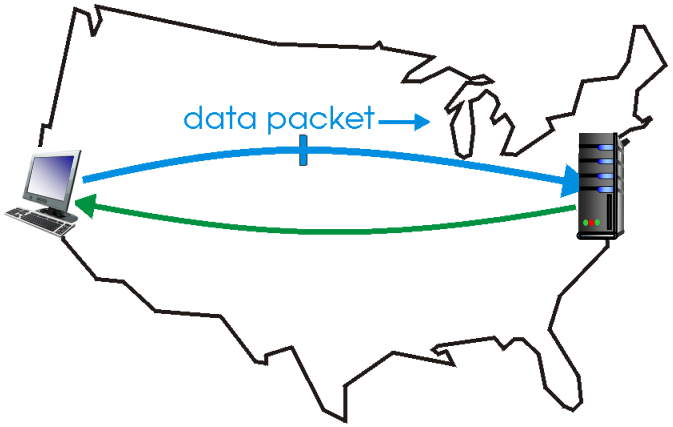
- il throughput effettivo generato dal mittente:

  $L / (RTT + L/R) = U_{mittente} \cdot R = 267$ kbps
- le prestazioni del protocollo rdt 3.0 fanno schifo!
- il protocollo limita le prestazioni dell'infrastruttura sottostante (canale)

rdt3.0: funzionamento con pipeline

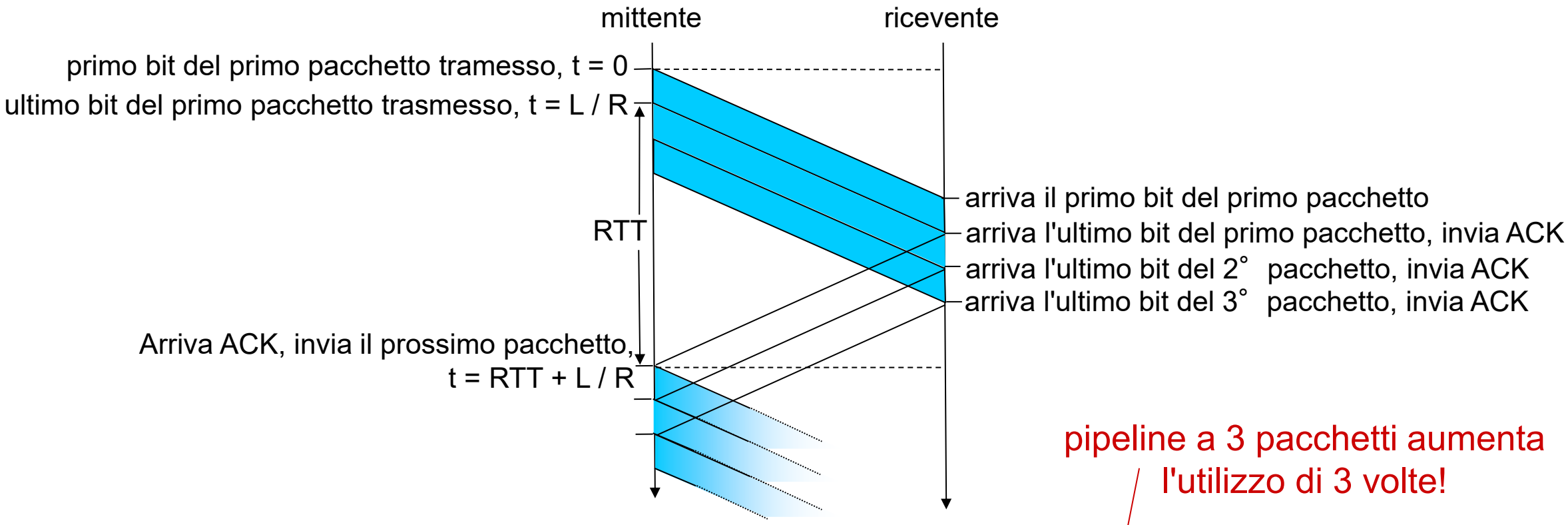
pipelining: il mittente ammette più pacchetti in transito, ancora da notificare

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



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

Pipelining: aumento dell'utilizzo

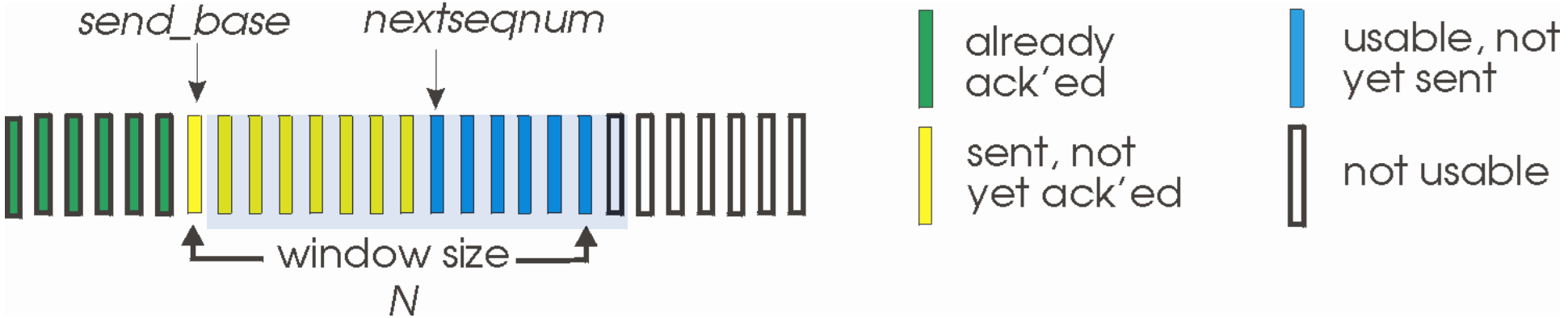


pipeline a 3 pacchetti aumenta l'utilizzo di 3 volte!

$$U_{mittente} = \frac{3L/R}{RTT + L/R} = \frac{0.0024}{30.008} = 0.00081$$

Protocolli con pipeline: Go-Back-N (mittente)

- mittente: "finestra" contenente fino a N pacchetti consecutivi trasmessi ma non riscontrati
  - numero di sequenza a k bit nell'intestazione del pacchetto

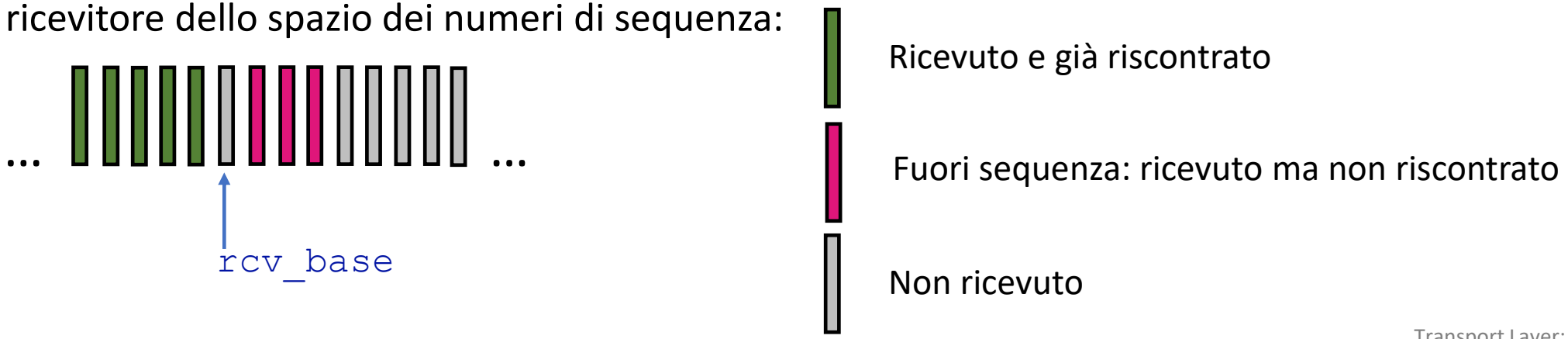


- *Riscontro cumulativo:* ACK(*n*): riscontro di tutti i pacchetti con numero di sequenza minore o uguale a *n*
  - alla ricezione dell'ACK(n): sposta la finestra in avanti per iniziare da n+1
- timer per il pacchetto più vecchio in transito
- *timeout(n):* ritrasmette il pacchetto n e tutti i pacchetti con i numeri di sequenza più grandi

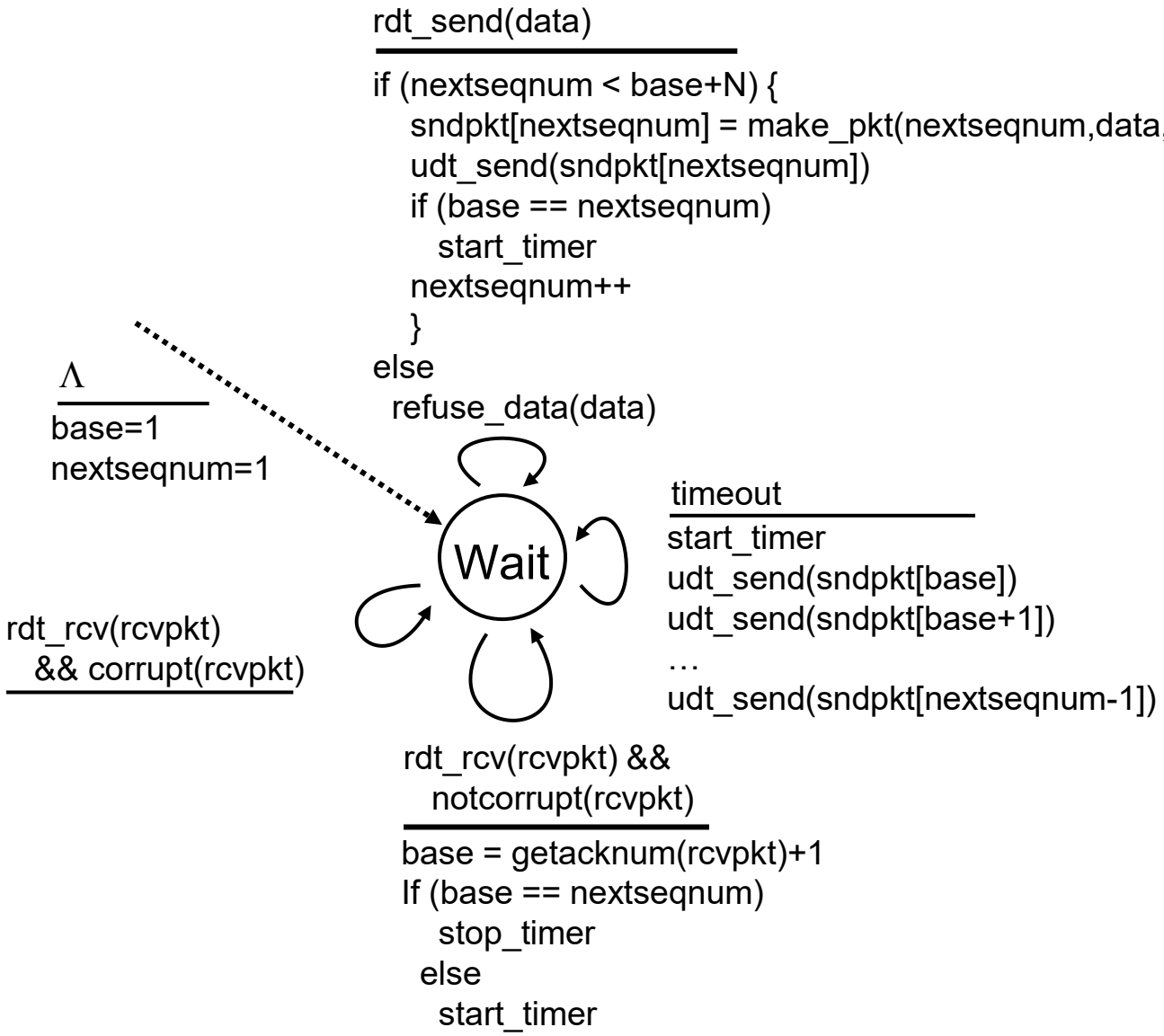
Protocolli con pipeline: Go-Back-N (ricevente)

- Solo ACK: invia sempre un ACK per un pacchetto ricevuto correttamente con il numero di sequenza più alto *in sequenza*
  - potrebbe generare ACK duplicati
  - deve memorizzare soltanto `rcv_base`
- Alla ricezione di un pacchetto fuori sequenza:
  - può scartarlo (non è salvato) o inserirlo in un buffer: la implementazione che discuteremo scarta i pacchetti fuori sequenza e deve solo gestire la variabile *rcv_base*.
  - rimanda un ACK per il pacchetto con il numero di sequenza più alto in sequenza

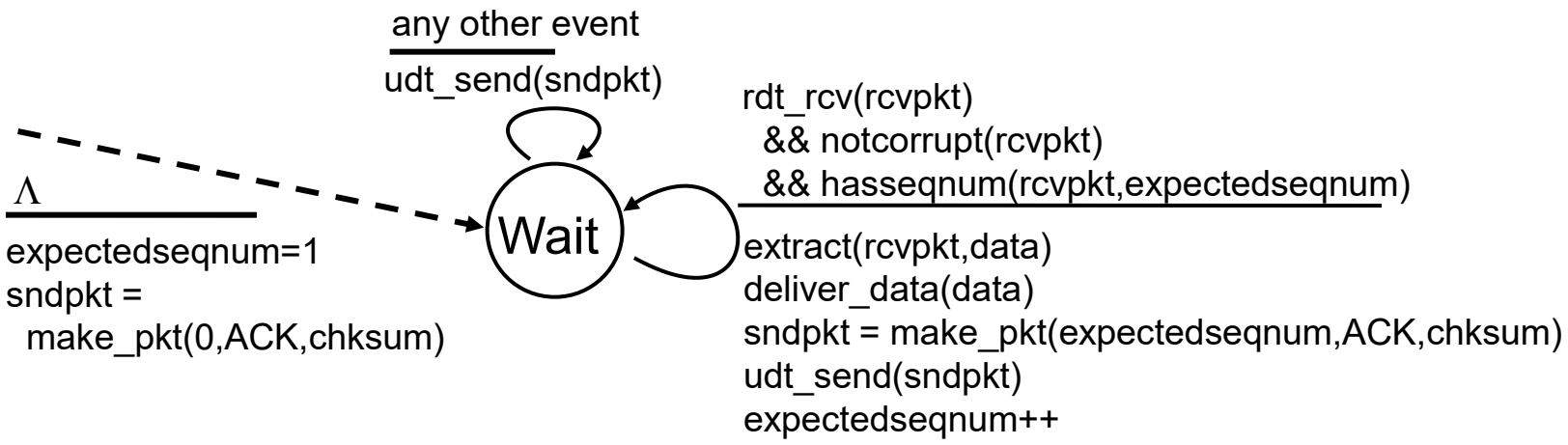
Vista del ricevitore dello spazio dei numeri di sequenza:



Go-Back-N: FSM esteso del mittente



Go-Back-N: FSM esteso del rivente

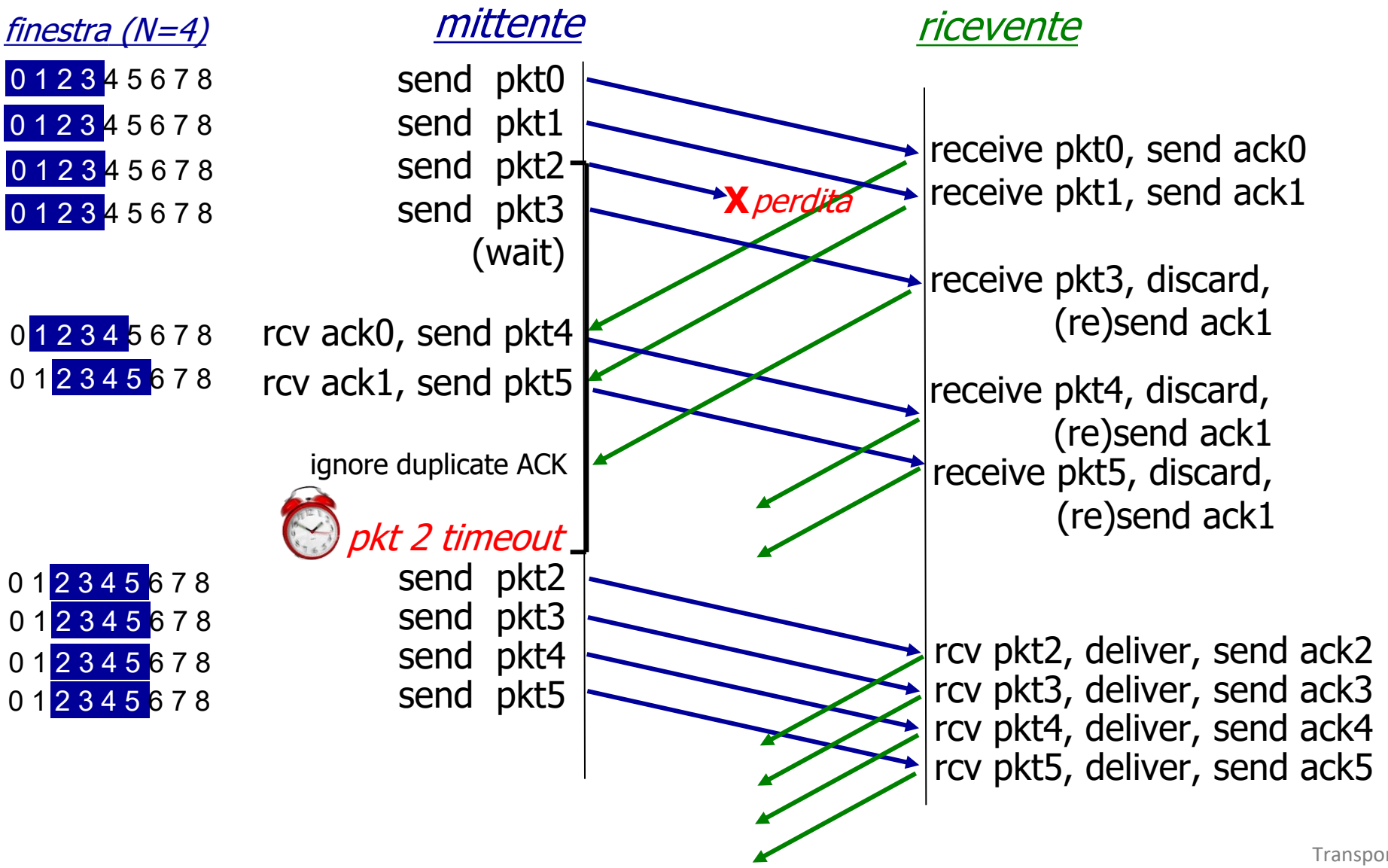


Solo ACK: invia sempre un ACK per pacchetti ricevuti correttamente col più alto numero di sequenza *in-ordine*

- può generare ACK duplicati
- deve solo ricordare **`expectedseqnum`**

- pacchetti fuori ordine:
  - scarta (non salva nel buffer): *nessun buffering lato ricevente!*
  - Re-invia ACK per il pacchetto con il più alto numero di sequenza in ordine

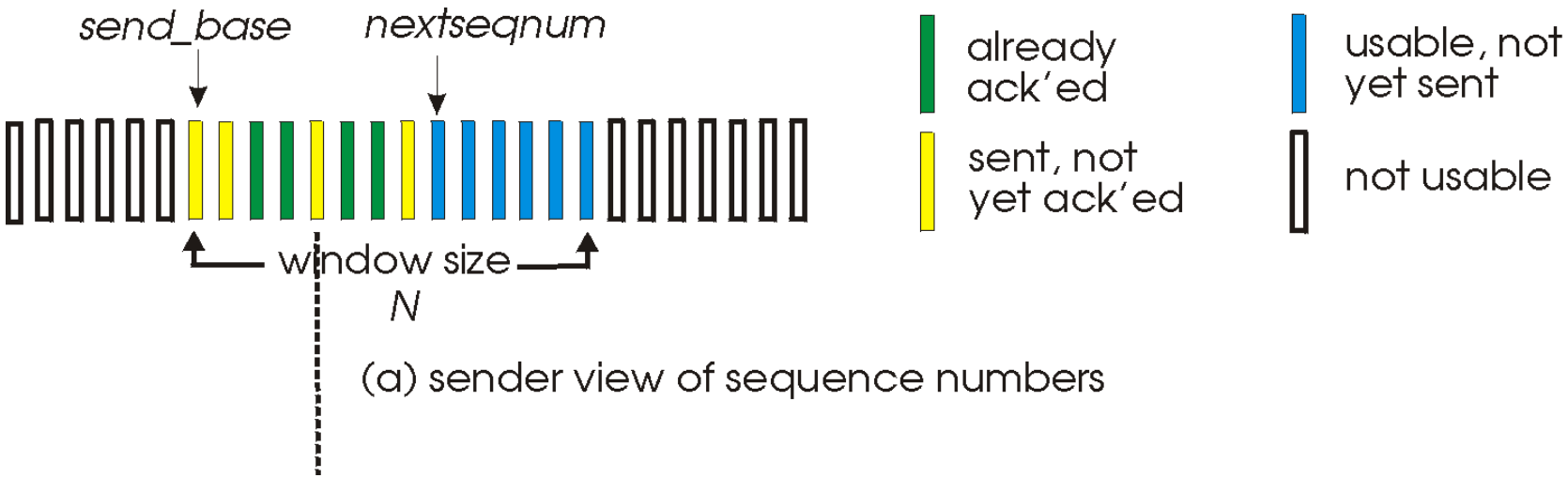
Go-Back-N in azione



Protocolli con pipeline: selective repeat

- *pipelining*: più pacchetti in transito
- *il ricevente riscontra individualmente* tutti i pacchetti ricevuti correttamente
  - buffer dei pacchetti, se necessario, per eventuali consegna in sequenza al livello superiore
- mittente:
  - mantiene (concettualmente) un timer per ogni pacchetto non riscontrato
    - timeout: ritrasmette il singolo pacchetto non riscontrato associato al timeout
  - mantiene (concettualmente) una "finestra" su *N* numeri di sequenza consecutivi
    - limita i pacchetti in pipeline, "in transito", per rientrare in questa finestra

Selective repeat: finestre del mittente e del ricevente



(a) sender view of sequence numbers

Selective repeat: mittente e ricevente

mittente

dati dall'alto:

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

timeout(*n*):

- ritrasmette il pacchetto n, riparte il timer

ACK(*n*) in [sendbase,sendbase+N-1]:

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

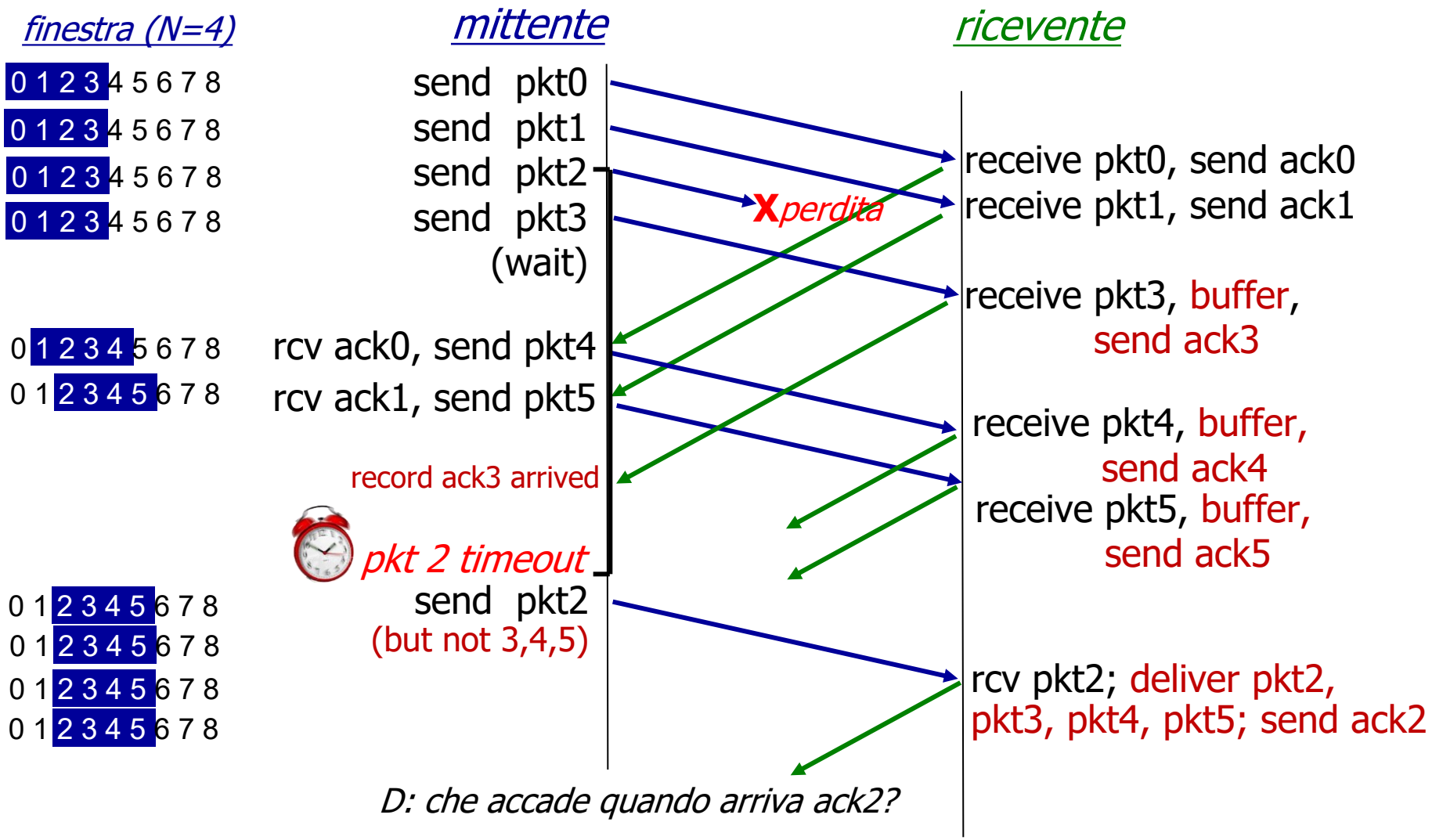
- ACK(*n*)

altrimenti:

- ignora

Il ricevente deve riscontrare pacchetti con numero di sequenza al di sotto di `rcvbase` (già consegnati all'applicazione), per far avanzare la finestra del mittente (rimasta indietro perché gli ACK non sono stati ricevuti)

Selective Repeat in azione

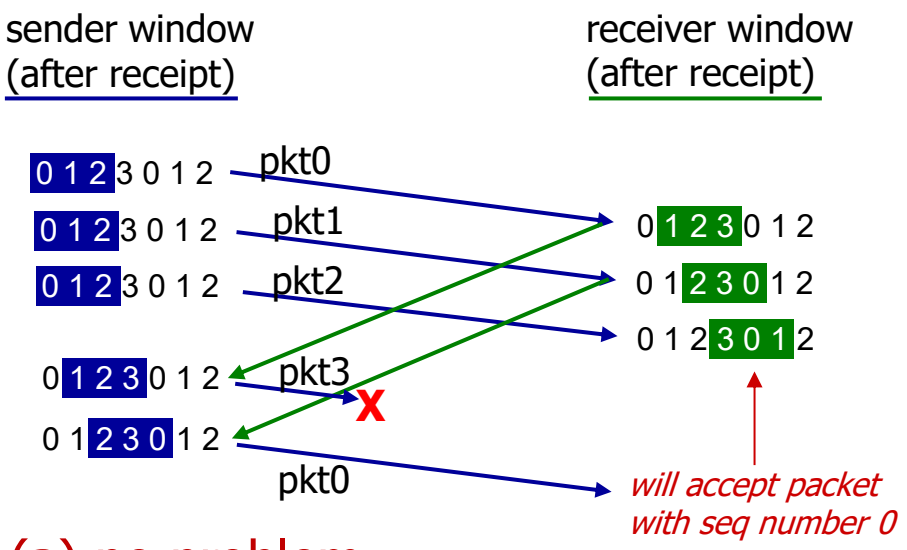


Selective repeat: dilemma!

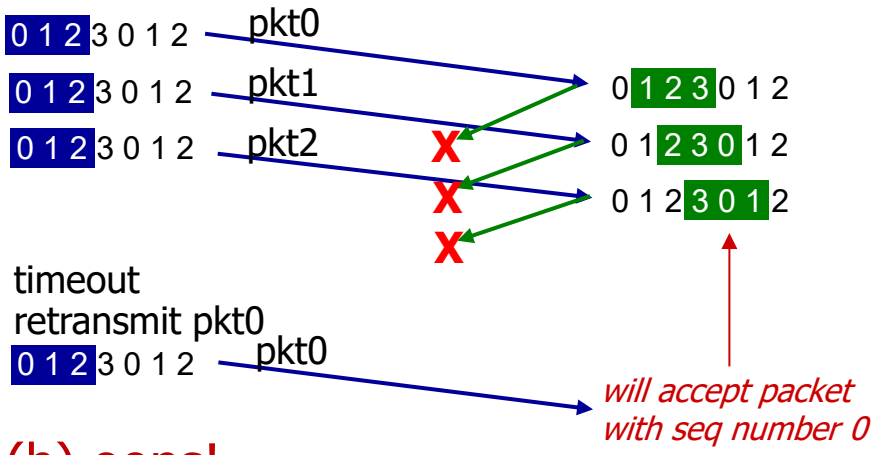
esempio:

- numeri di sequenza: 0, 1, 2, 3
- dimensione della finestra = 3

I numeri di sequenza sono usati *ciclicamente -> aritmetica modulare*



(a) no problem



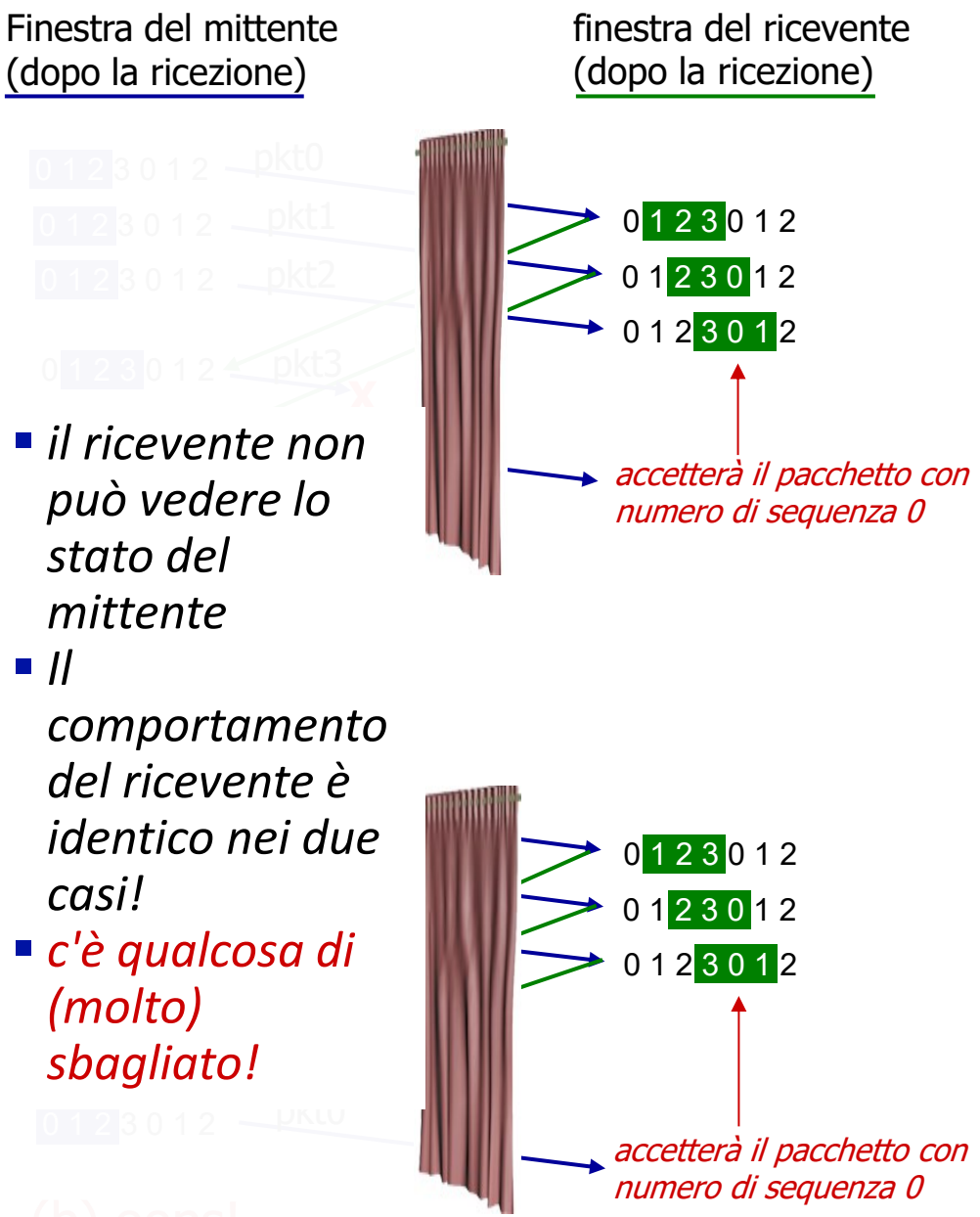
(b) oops!

Selective repeat: dilemma!

esempio:

- numeri di sequenza: 0, 1, 2, 3
- dimensione della finestra = 3

Q: Quale relazione è necessaria tra la dimensione dei numeri di sequenza e la dimensione della finestra per evitare il problema nello scenario (b)?



- *il ricevente non può vedere lo stato del mittente*
- *Il comportamento del ricevente è identico nei due casi!*
- *c'è qualcosa di (molto) sbagliato!*

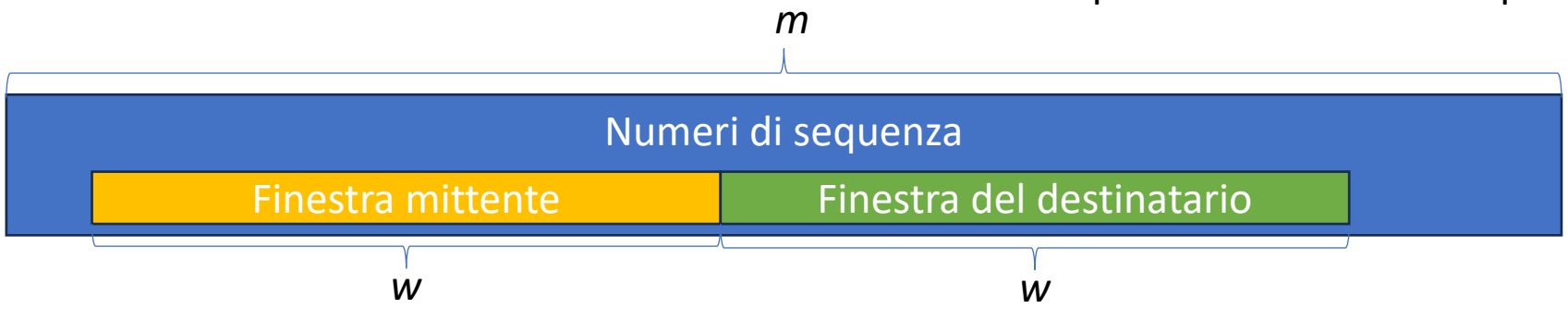
Relazione tra dimensione della finestra e dei numeri di sequenza (sia SR sia GBN)

Q: Quale relazione è necessaria tra la dimensione dei numeri di sequenza e la dimensione della finestra per evitare il problema nello scenario (b)?

Il problema è che a causa della mancata ricezione degli ACK la finestra del ricevente può andare avanti rispetto a quella del mittente: si consideri il caso peggiore associato all'invio di un'intera finestra di pacchetti e la mancata ricezione di tutti gli ACK.

È necessario che lo spazio dei numeri di sequenza sia sufficientemente grande da contenere sia la finestra del mittente sia la finestra del destinatario senza che si sovrappongano (considerando l'operazione di modulo)

Sia w la dimensione della finestra e m la dimensione dello spazio dei numeri di sequenza:



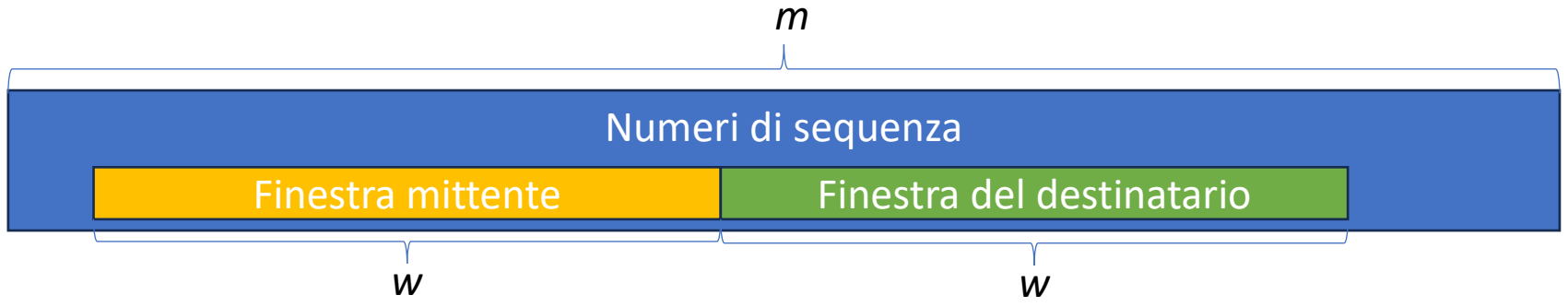
$2w \leq m$

cioè

$w \leq m / 2$

Relazione tra dimensione della finestra e dei numeri di sequenza (sia SR sia GBN)

Q: Quale relazione è necessaria tra la dimensione dei numeri di sequenza e la dimensione della finestra per evitare il problema nello scenario (b)?



$2w \leq m$
cioè
$w \leq m / 2$

Se i numeri di sequenza sono espressi tramite k bit, allora $m = 2^k$
$w \leq 2^k/2$ cioè $w \leq 2^{k-1}$

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di trasporto (parte3)

dr. Manuel Fiorelli
manuel.fiorelli@uniroma2.it
https://art.uniroma2.it/fiorelli

Basate sulle slide del libro di testo:
https://gaia.cs.umass.edu/kurose_ross/ppt.php

Capitolo 3: tabella di marcia

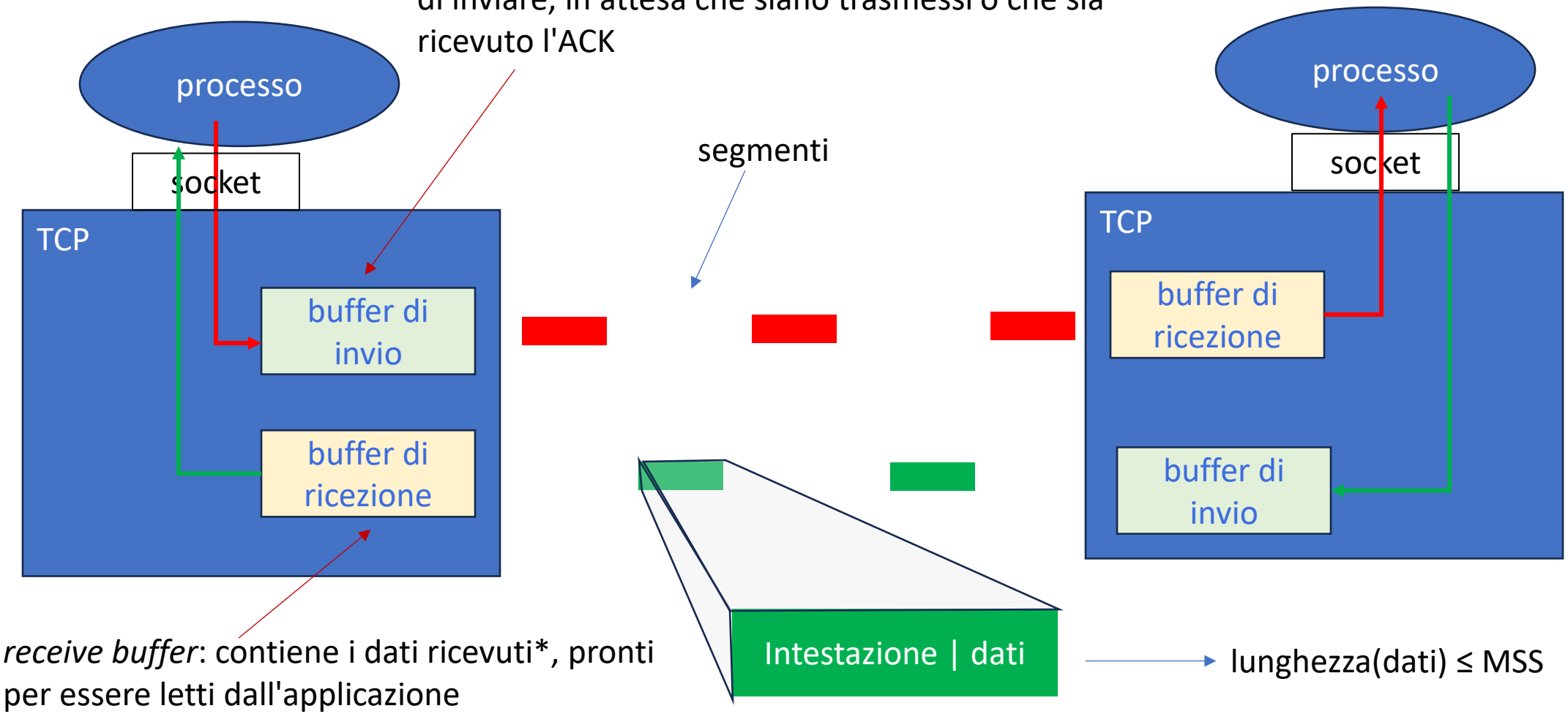
- Servizi a livello di trasporto
- Multiplexing e demultiplexing
- Trasporto senza connessione: UDP
- Principi del trasferimento dati affidabile
- Trasporto orientato alla connessione: TCP
  - struttura dei segmenti
  - trasferimento dati affidabile
  - controllo di flusso
  - controllo della congestione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

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

- punto a punto (point-to-point):
  - singolo mittente, singolo destinatario
- flusso di byte affidabile (reliable), in sequenza (in order):
  - nessun “confine ai messaggi"
- full duplex data:
  - i dati possono fluire in direzioni opposte allo stesso istante nella stessa connessione
  - MSS: dimensione massima del segmento (maximum segment size); in realtà si riferisce ai dati applicativi nel segmento (intestazione esclusa)
- ACK cumulativi
- pipelining:
  - il controllo di flusso e della congestione definiscono la dimensione della finestra
- orientato alla connessione:
  - l'handshaking (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima di scambiare i dati
  - una connessione TCP non definisce un circuito end-to-end del tipo che caratterizza le reti a commutazione di circuito. Infatti, le connessioni TCP sono interamente implementate nei sistemi periferici, mentre i commutatori di pacchetto vedono solo i pacchetti di rete in transito
- flusso controllato:
  - il mittente non sovraccarica il destinatario
- controllo della congestione:
  - il mittente riduce la velocità di invio in funzione della congestione della rete

TCP: Buffer di invio e ricezione TCP.

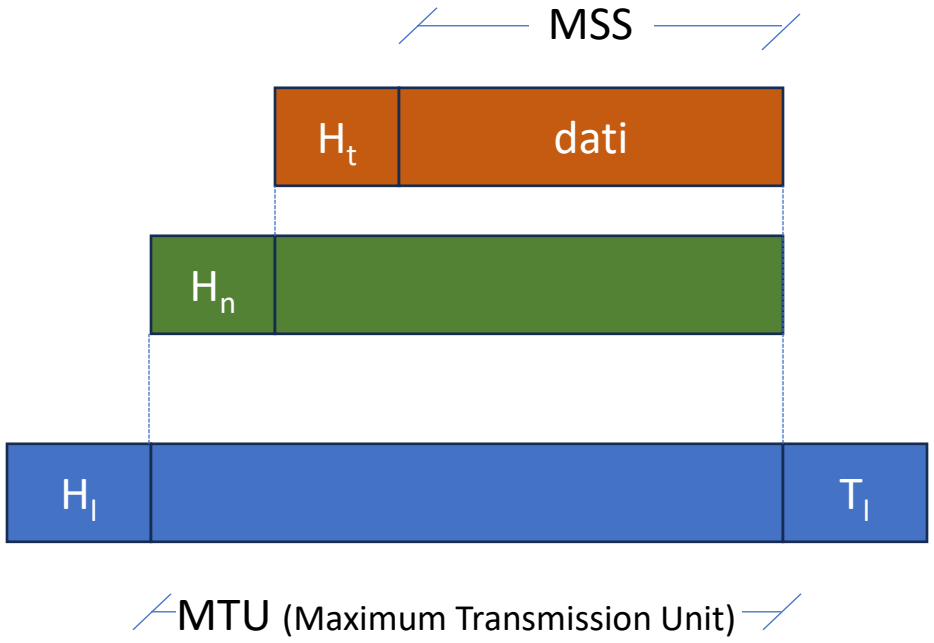
send buffer: contiene i dati che il processo ha chiesto di inviare, in attesa che siano trasmessi o che sia ricevuto l'ACK



receive buffer: contiene i dati ricevuti*, pronti per essere letti dall'applicazione

**la gestione dei segmenti fuori ordine richiede l'uso di ulteriori strutture dati e variabili (non mostrate); a prescindere da dove questi dati siano salvati dalla implementazione di TCP, il processo può accedere solo alla parte del buffer contenente dati in ordine*

TCP: MSS



La dimensione massima del segmento TCP predefinita per IPv4 è di 536. Per IPv6 è 1220. Se un host desidera impostare MSS su un valore diverso da quello predefinito, il suo valore viene specificata come opzione TCP, inizialmente nel pacchetto TCP SYN durante l'handshake TCP.

- MSS + lunghezza(H_t) + lunghezza(H_n) ≤ MTU

Valori tipici:

lunghezza(H_t) = 20 B

lunghezza(H_n) = 20 B

$MTU_{Ethernet}$= 1500 -> MSS = 1460

- Un host può determinare il MSS guardando la MTU del collegamento locale; ma ciò non offre garanzie circa altri collegamenti intermedi
- Può essere negoziato durante la connessione con l'opzione MSS
- Path MTU Discovery: permette di scoprire il valore più piccolo della MTU lungo il percorso da mittente a destinatario

Se un pacchetto IP eccede la MTU su un collegamento di uscita, il router dovrà frammentarlo (in IPv4) oppure scartalo (in IPv6).

Struttura dei segmenti TCP

ACK: numero di sequenza del prossimo byte atteso; flag A: numero di riscontro valido

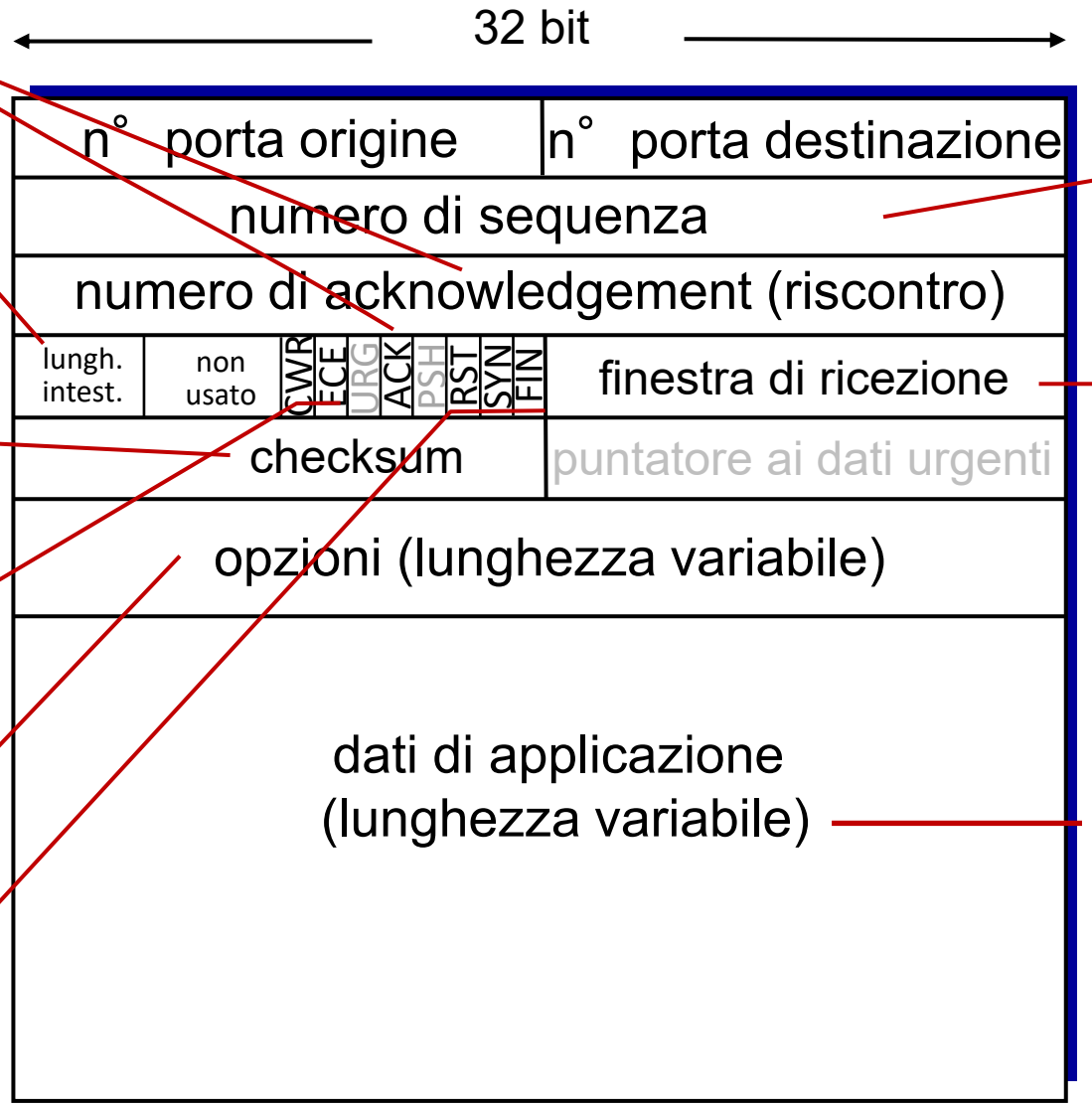
Lunghezza dell'intestazione TCP (in multipli di 32 bit)

checksum Internet (come in UDP)

CRW, ECE: flag usati per il controllo di congestione esplicito

opzioni TCP (generalmente e non usate)

RST, SYN, FIN: gestione della connessione



conteggio per byte di dati (non segmenti!)

controllo di flusso: numero di byte che il destinatario desidera accettare

dati inviati dall'applicazione nella socket TCP

(possibilmente suddivisi in più segmenti, a differenza di UDP in cui l'applicazione legge/scrivi messaggi che devono corrispondere a un segmento)

Lunghezza dell'intestazione variabile, ma valore tipico è 20 byte (senza alcuna opzione).

Numeri di sequenza e ACK di TCP

Numeri di sequenza:

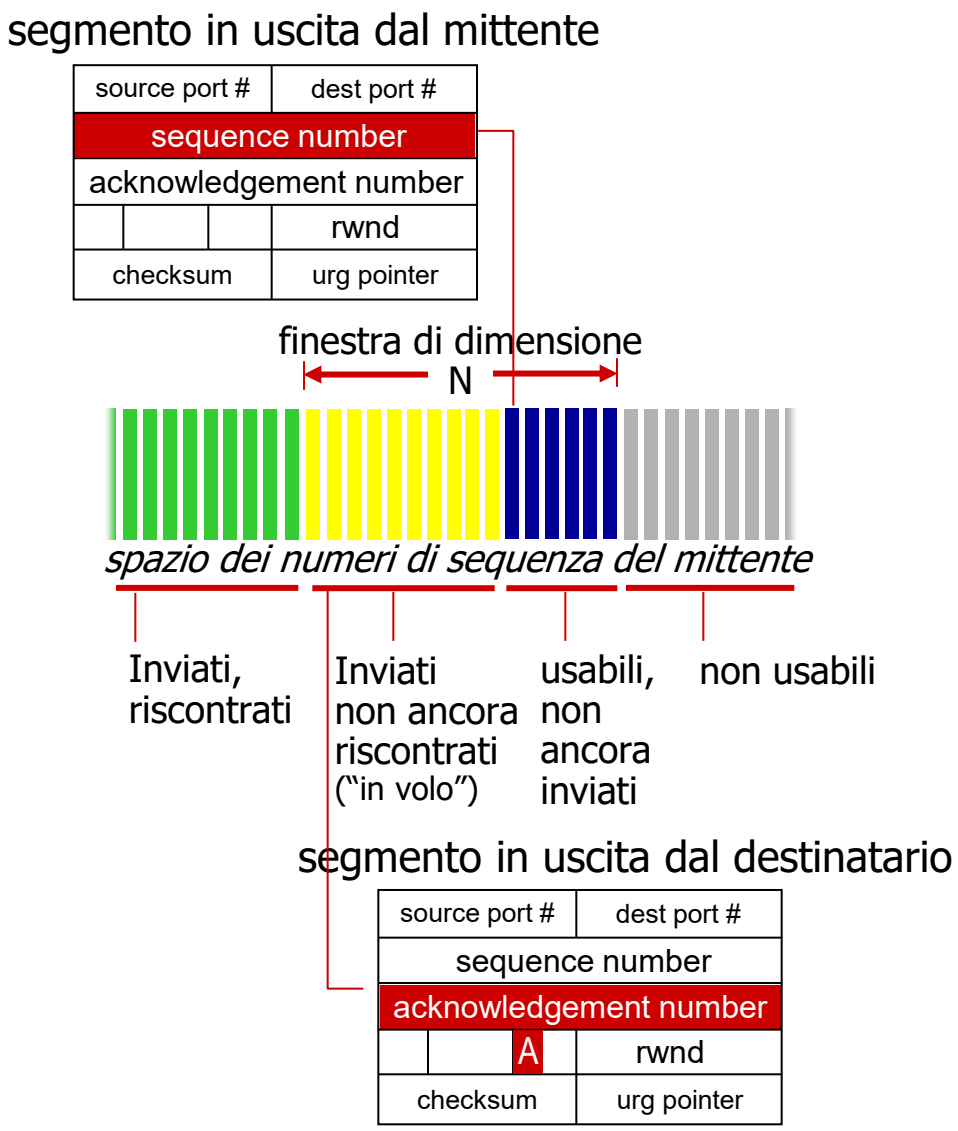
- "numero" del primo byte nel segmento nel flusso di byte

ACK:

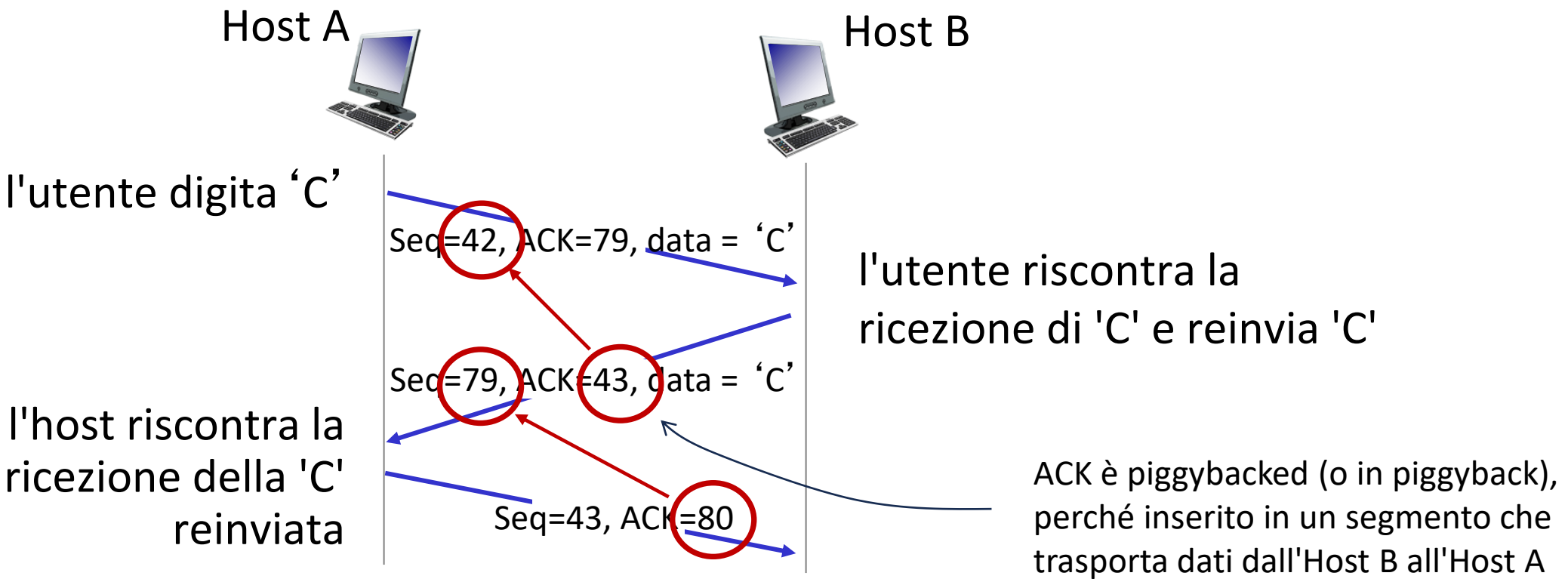
- numero di sequenza del prossimo byte atteso dall'altro lato
- ACK cumulativo
- RFC 2018: Acknowledgment Selettivo (che non studieremo)

D: come gestisce il destinatario i segmenti fuori sequenza?

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



Numeri di sequenza e ACK di TCP



un semplice serve echo

TCP: tempo di andata e ritorno (round trip time) e timeout

D: come impostare il valore del timeout di TCP?

- più grande di RTT (tempo trascorso da quando si invia un segmento a quando se ne riceve l'acknowledgment), ma RTT non è noto varia!
- *troppo piccolo:* timeout prematuro, ritrasmissioni non necessarie
- *troppo grande:* reazione lenta alla perdita di segmenti

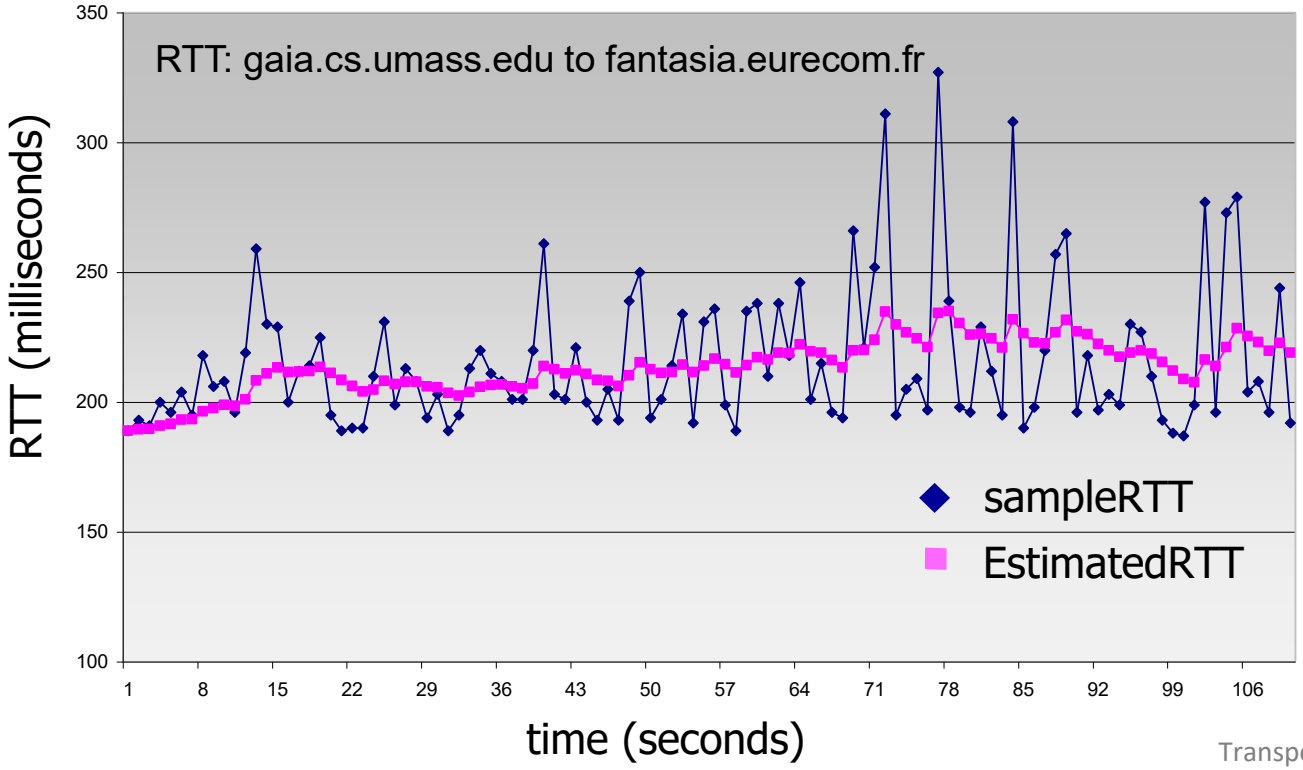
D: come stimare RTT?

- `SampleRTT:` tempo misurato dalla trasmissione del segmento fino alla ricezione di ACK
  - ignora le ritrasmissioni
- `SampleRTT` varia, quindi occorre una stima più "livellata" di RTT
  - *media di più misure recenti*, non semplicemente il calore corrente di `SampleRTT`

TCP: tempo di andata e ritorno (round trip time) e timeout

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

- exponential weighted moving average (EWMA): media mobile esponenziale pesata
- l'influenza dei vecchi campioni decresce esponenzialmente
  - i campioni più recenti riflettono meglio la congestione attuale della rete
- Valore raccomandato: $\alpha = 0.125$



TCP: tempo di andata e ritorno (round trip time) e timeout

```
EstimatedRTT = (1- α)*EstimatedRTT + α*SampleRTT
```

$$\mathbf{EstimatedRTT_n = (1-\alpha)*EstimatedRTT_{n-1} + \alpha*SampleRTT_n}$$

$$EstimatedRTT_n = (1-\alpha)\cdot EstimatedRTT_{n-1} + \alpha \cdot SampleRTT_n$$

$$= (1-\alpha)\cdot\left((1-\alpha)\cdot EstimatedRTT_{n-2} + \alpha\cdot SampleRTT_{n-1}\right) + \alpha\cdot SampleRTT_n$$

$$= (1-\alpha)^2\cdot EstimatedRTT_{n-2} + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha\cdot SampleRTT_n$$

$$= (1-\alpha)^2\cdot\left((1-\alpha)\cdot EstimatedRTT_{n-3} + \alpha\cdot SampleRTT_{n-2}\right) + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha\cdot SampleRTT_n$$

$$= (1-\alpha)^3\cdot EstimatedRTT_{n-3} + \alpha(1-\alpha)^2\cdot SampleRTT_{n-2} + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha\cdot SampleRTT_n$$

$$= (1-\alpha)^n\cdot EstimatedRTT_0 + \alpha(1-\alpha)^{n-1}\cdot SampleRTT_1 + \cdots + \alpha(1-\alpha)\cdot SampleRTT_{n-1} + \alpha\cdot SampleRTT_n$$

$$= (1-\alpha)^n\cdot EstimatedRTT_0 + \sum_{i=0}^{n-1}\alpha(1-\alpha)^i\cdot SampleRTT_{n-i}$$

Essendo 0 < α < 1, il peso dei campioni vecchi decresce in modo esponenziale

TCP: tempo di andata e ritorno (round trip time) e timeout

- Intervallo di timeout: **`EstimatedRTT`** più un "margine di sicurezza"
  - grande variazione di **`EstimatedRTT`**: margine di sicurezza maggiore

```
TimeoutInterval = EstimatedRTT + 4*DevRTT
```

RTT stimato

"margine di sicurezza"

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

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

(tipicamente, β = 0.25)

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Mittente TCP (semplificato)

evento: ricevuti dati dall'applicazione

- 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 già in funzione
  - pensate al timer come se fosse associato al più vecchio segmento non ancora riscontrato
  - intervallo di scadenza: **`TimeOutInterval`**

evento: timeout

- ritrasmette il segmento che ha causato il timeout (cioè il segmento in attesa di ACK con il più piccolo numero di sequenza)
- riavvia il timer

la durata del timeout viene raddoppiata, ma torna a usare la formula standard non appena viene ricevuto un ACK per un nuovo segmento

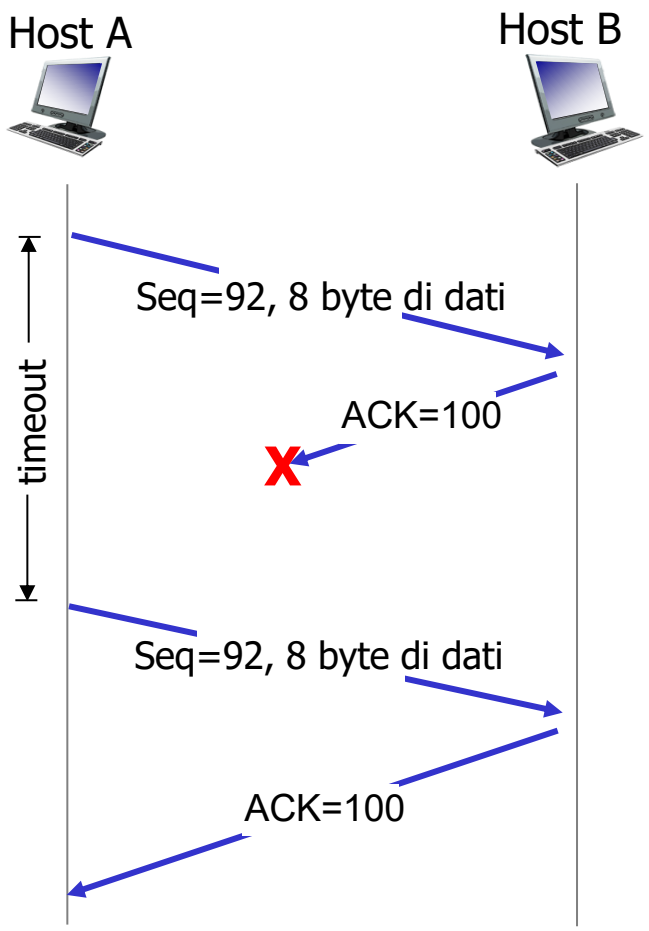
evento: ACK ricevuto

- se riscontra segmenti precedentemente non riscontrati (ACK y dove y > SendBase)
  - aggiorna ciò che si sa essere stato riscontrato (SendBase = y)
  - avvia il timer se ci sono altri segmenti ancora non riscontrati

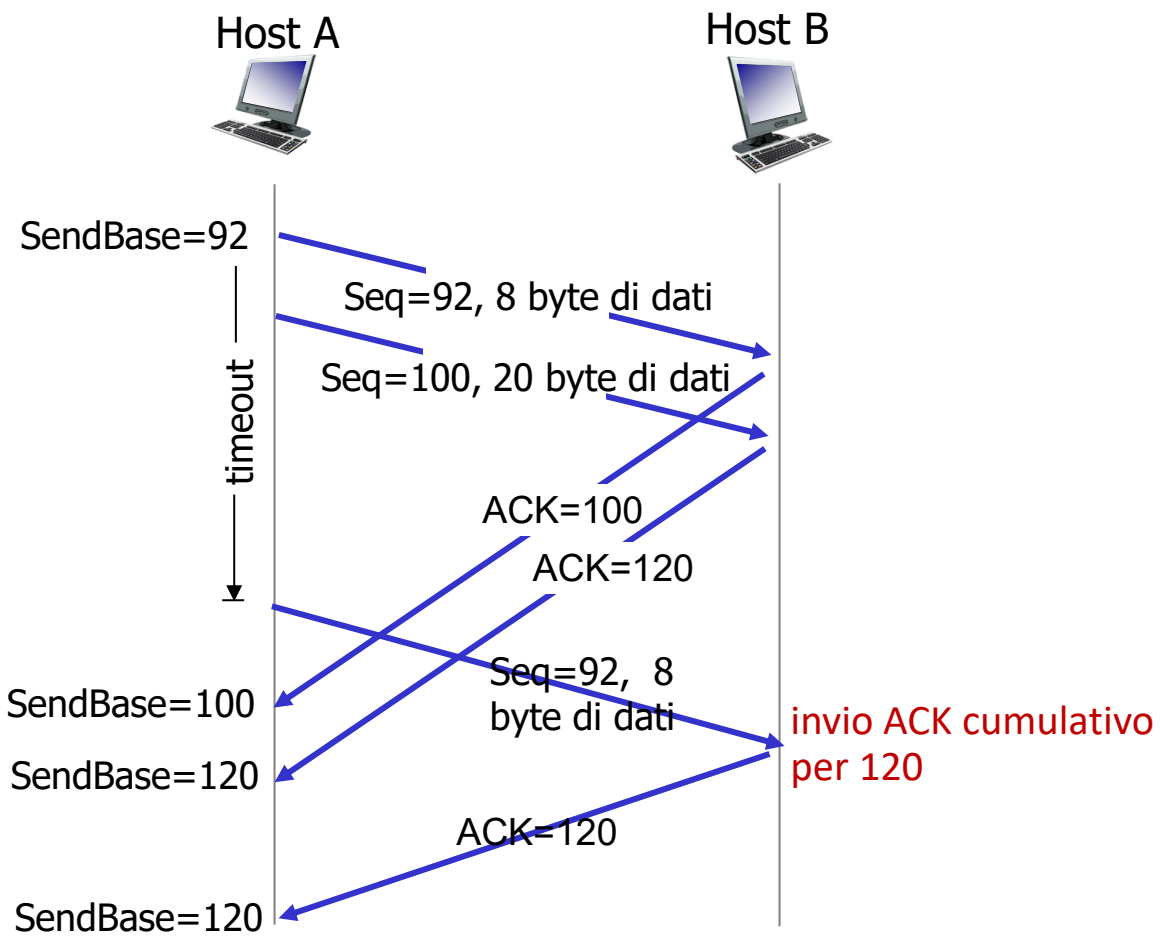
Ricevente TCP: Generazione degli ACK [RFC 5681]

Evento presso il destinatario	*Azione del ricevente TCP*
Arrivo ordinato di segmento con numero di sequenza atteso. Tutti i dati fino al numero di sequenza attivo sono già stati riscontrati.	ACK ritardato. Attende fino a 500 millisecondi per l'arrivo ordinato di un altro segmento. Se in questo intervallo non arriva il successivo segmento, invia un ACK.
Arrivo ordinato di segmento con numero di sequenza atteso. Un altro segmento ordinato è in attesa di trasmissione dell'ACK.	Invia immediatamente un singolo ACK cumulativo, riscontrando entrambi i segmenti ordinati.
Arrivo non ordinato di segmento con numero di sequenza superiore a quello atteso. Viene rilevato un buco.	Invia immediatamente un *ACK duplicato*, indicando il numero di sequenza del prossimo byte atteso (che è l'estremità inferiore del buco)
Arrivo di segmento che colma parzialmente o completamente il buco nei dati ricevuti.	Invia immediatamente un ACK, ammesso che il segmento cominci all'estremità inferiore del buco

TCP: scenari di ritrasmissione

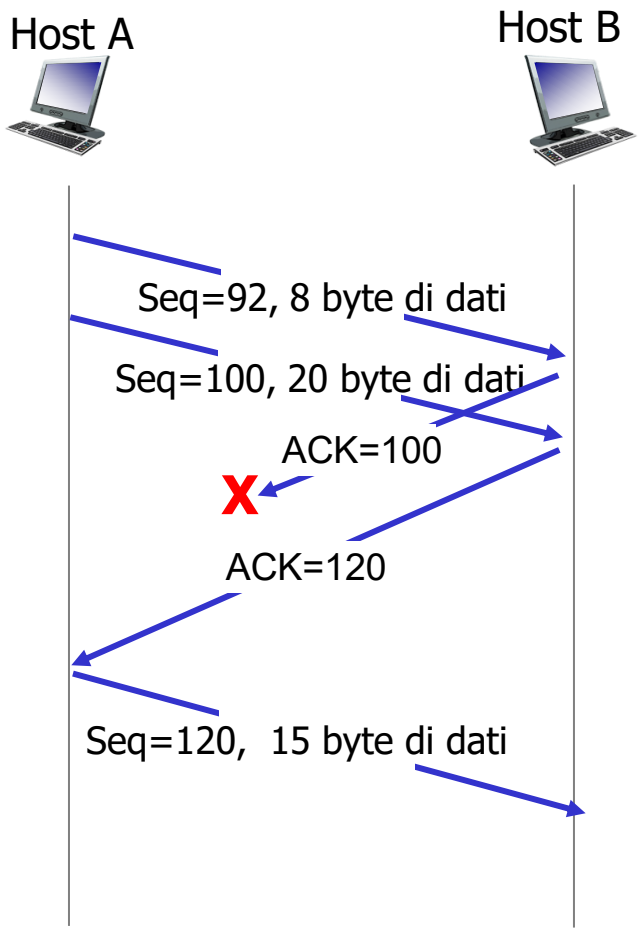


scenario con ACK perso



timeout prematuro

TCP: scenari di ritrasmissione



l'ACK cumulativo copre l'ACK precedente perso

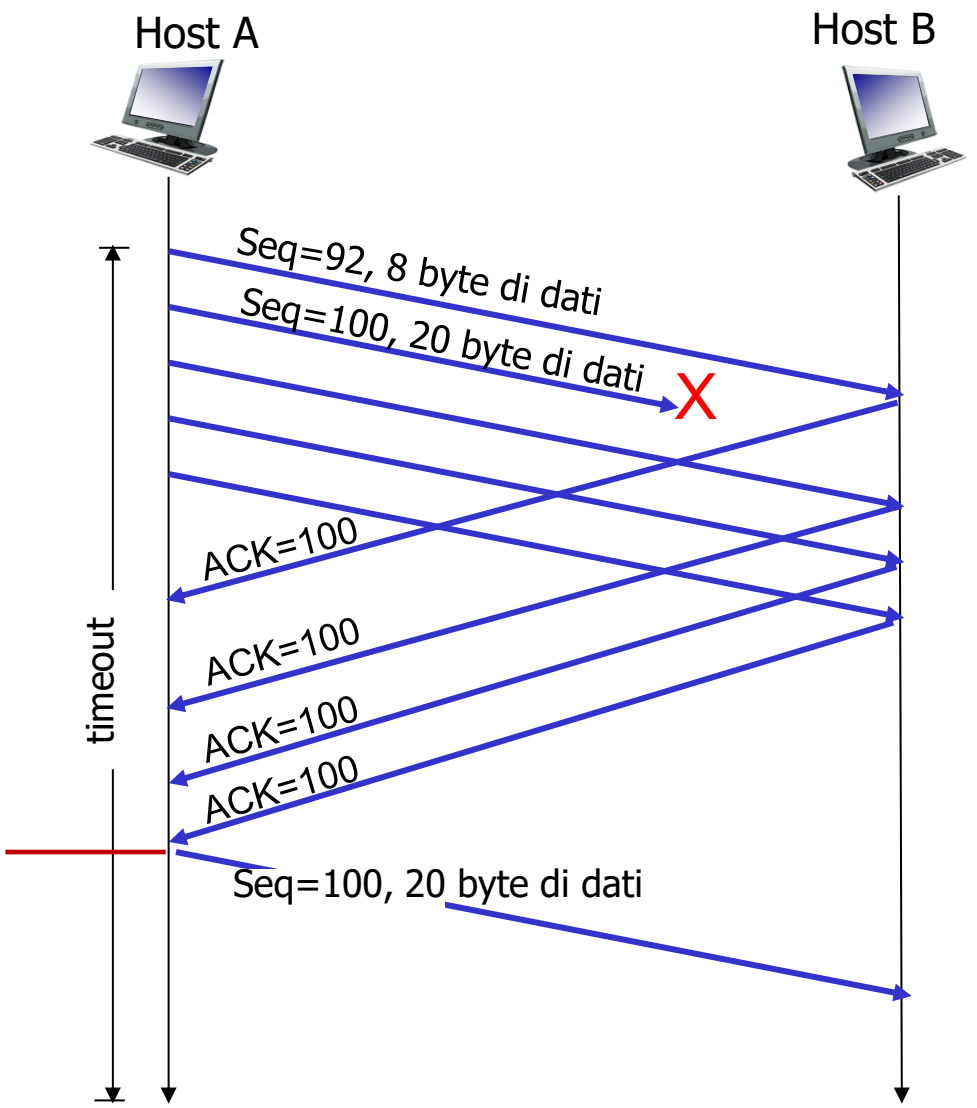
Ritrassmissione rapida

Ritrasmissione rapida

Se il mittente riceve 3 ACK addizionali per gli stessi dati ("3 ACK duplicati"), rispedisce il segmento non riscontrato con il più piccolo numero di sequenza

- è probabile che il segmento non riscontrato sia stato perso, 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!

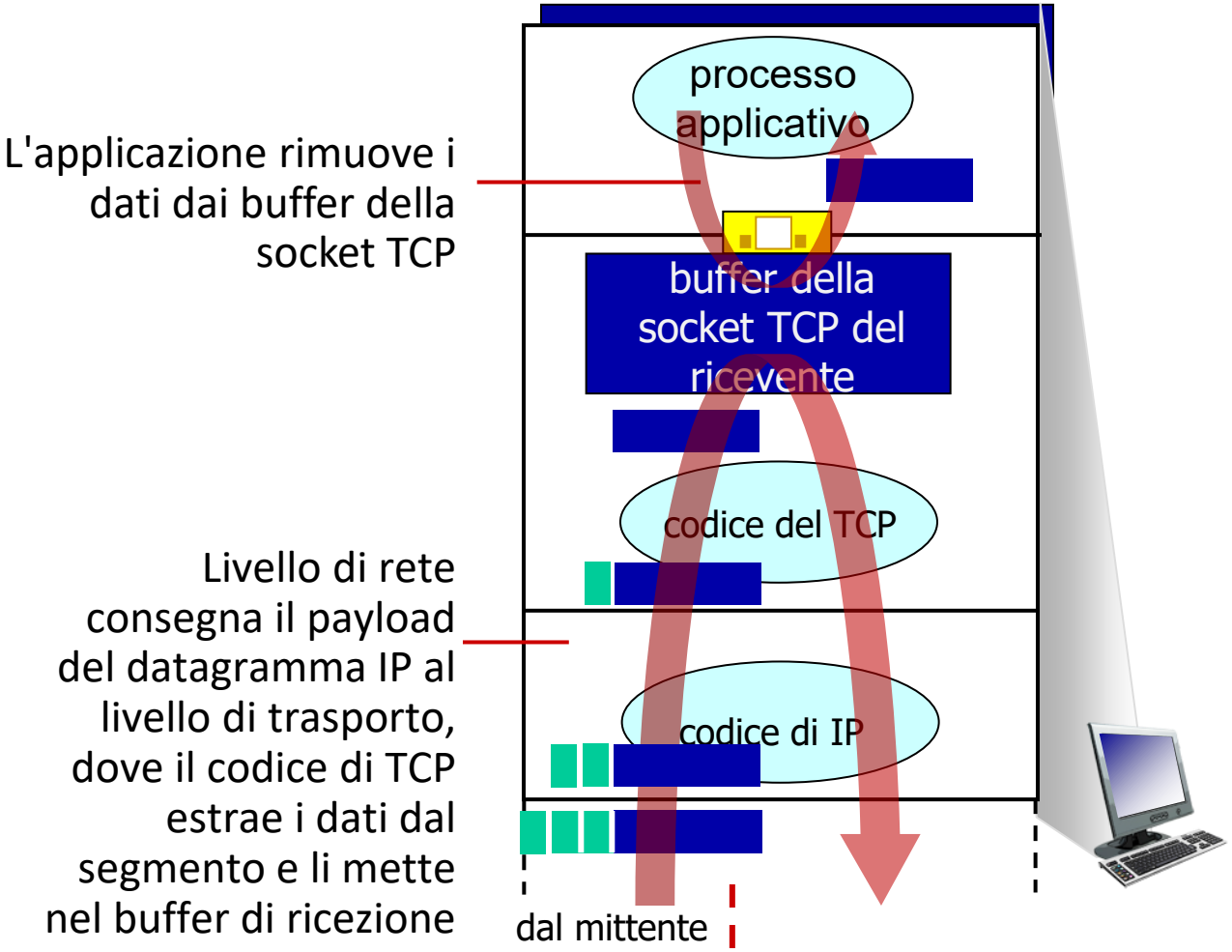


Chapter 3: roadmap

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

TCP: controllo di flusso

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



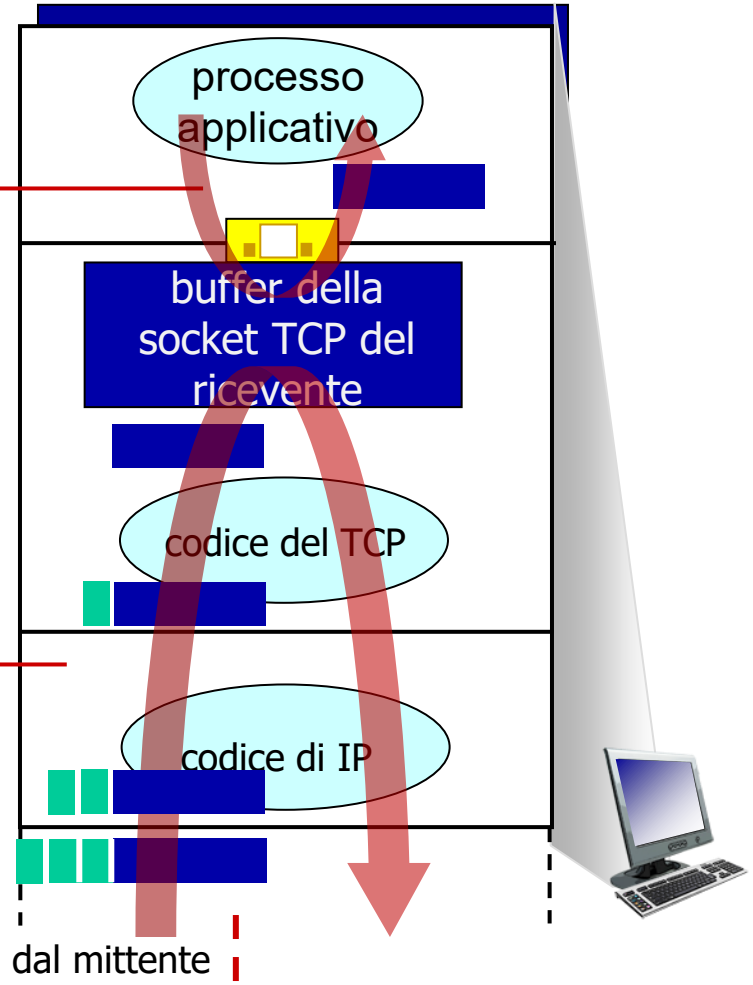
pila protocollare del ricevente

TCP: controllo di flusso

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

L'applicazione rimuove i dati dai buffer della socket TCP

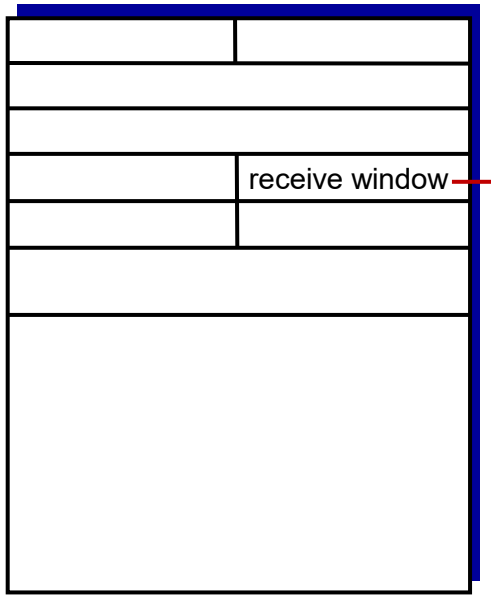
Livello di rete consegna il payload del datagramma IP al livello di trasporto, dove il codice di TCP estrae i dati dal segmento e li mette nel buffer di ricezione



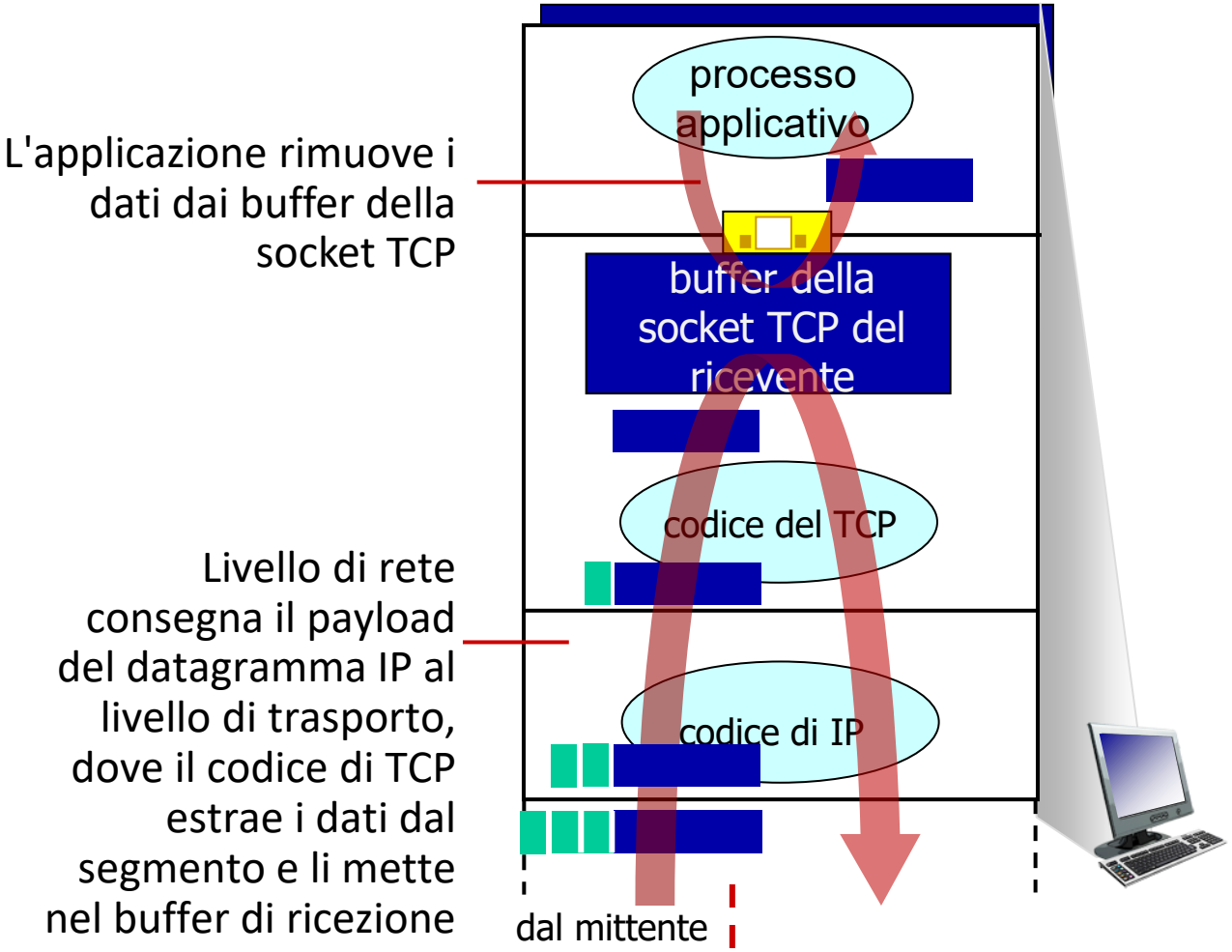
pila protocollare del ricevente

TCP: controllo di flusso

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



controllo di flusso: numero di byte che il ricevente è disposto ad accettare



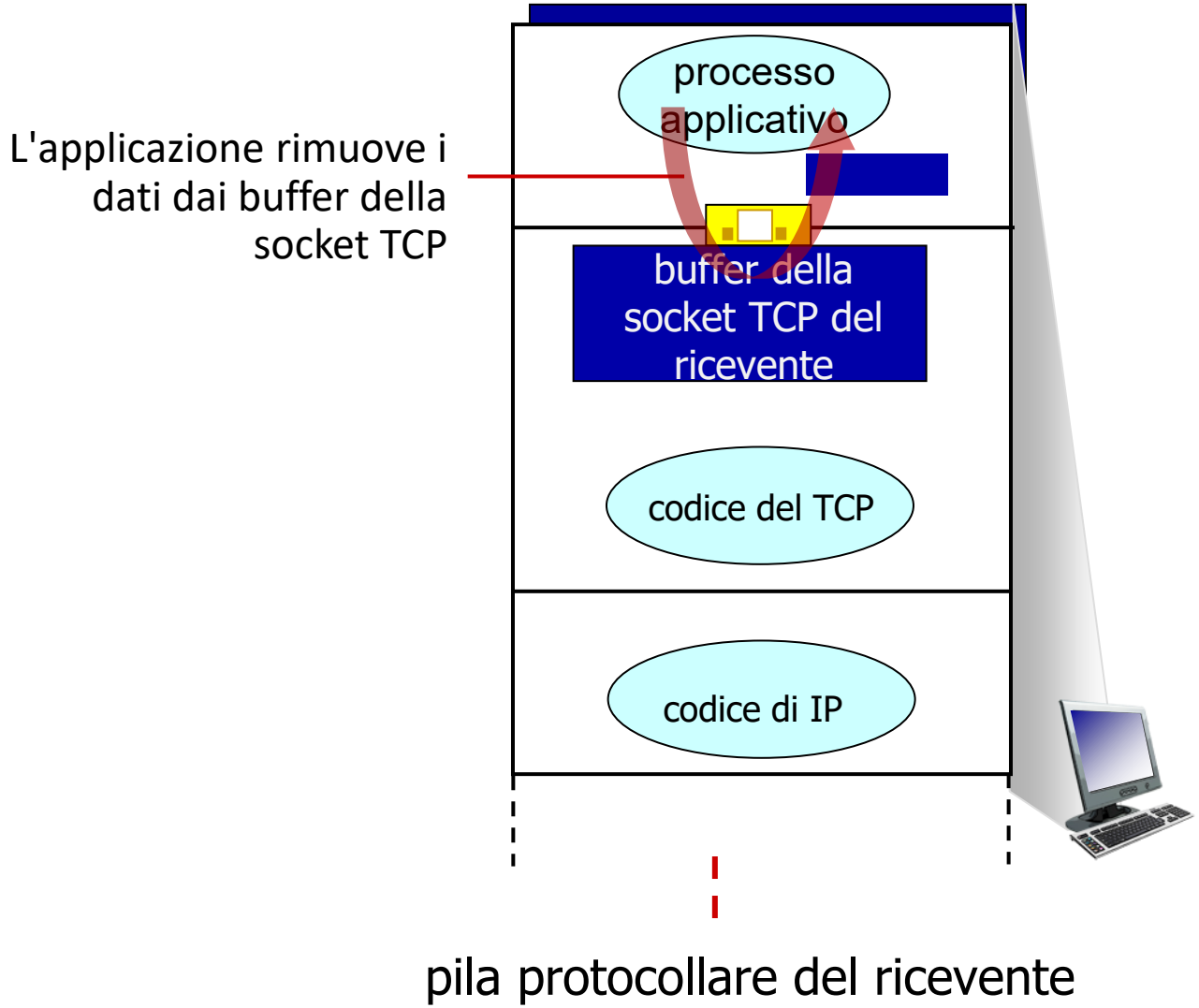
pila protocollare del ricevente

TCP: controllo di flusso

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

Controllo di flusso

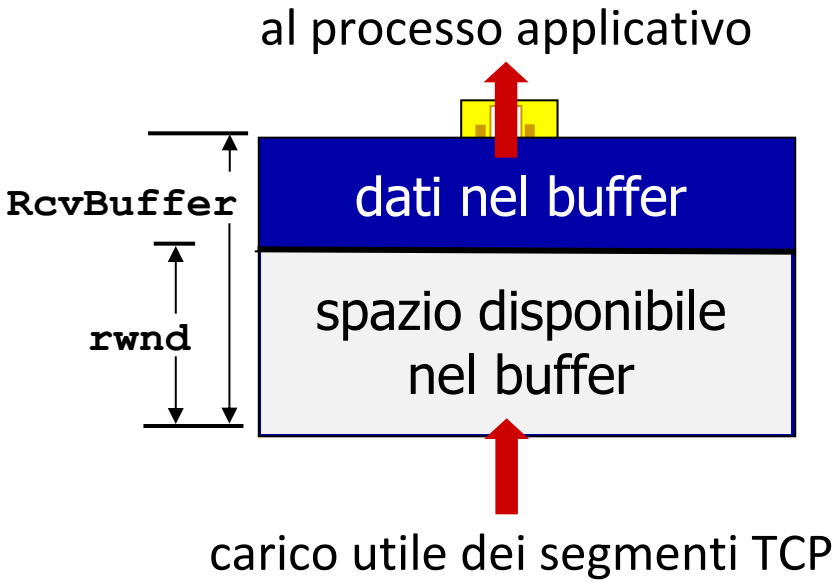
Il destinatario controlla il mittente, cosicché il mittente non ecceda il buffer del destinatario, inviando troppo e troppo velocemente



TCP: controllo di flusso

`rwnd = RcvBuffer - (LastByteRcvd - LastByteRead)`

- Il mittente comunica lo spazio disponibile nel buffer nel campo **`rwnd`** (*receive window*, finestra di ricezione) nell'intestazione TCP
  - **`RcvBuffer`** dimensione impostata attraverso opzioni della socket (valore predefinito è tipicamente 4096 byte)
  - molti sistemi operativi regolano automaticamente **`RcvBuffer`**
- Il mittente limita i dati non riscontrati a **`rwnd`**
  - garantisce che il buffer di ricezione non vada in overflow

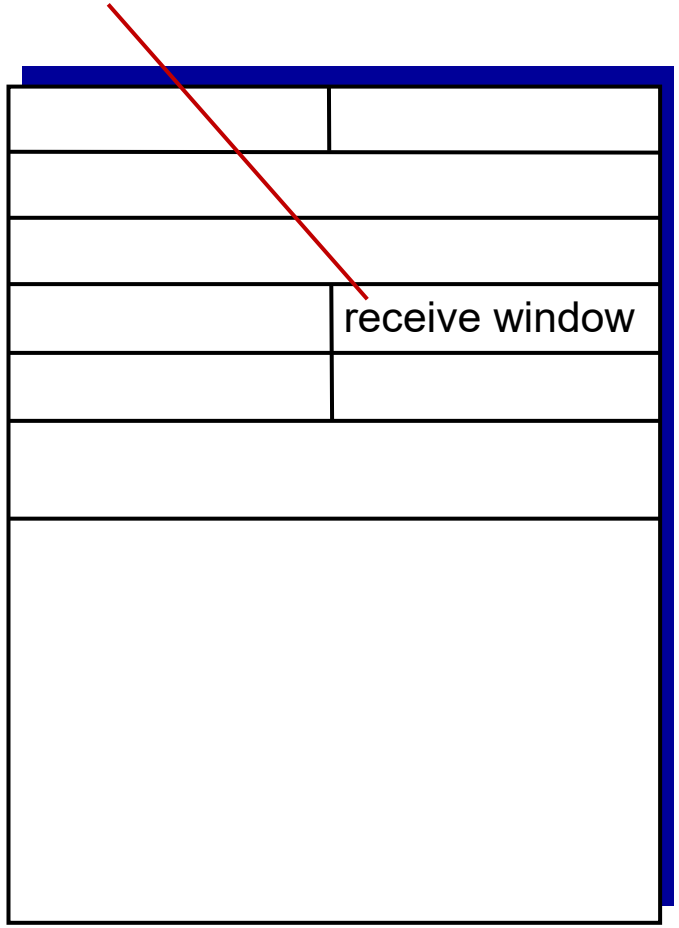


buffering dal lato del ricevente TCP

TCP: controllo di flusso

- Il mittente comunica lo spazio disponibile nel buffer nel campo **`rwnd`** (*receive window*, finestra di ricezione) nell'intestazione TCP
  - **`RcvBuffer`** dimensione impostata attraverso opzioni della socket (valore predefinito è tipicamente 4096 byte)
  - molti sistemi operativi regolano automaticamente **`RcvBuffer`**
- Il mittente limita i dati non riscontrati a **`rwnd`**
  - garantisce che il buffer di ricezione non vada in overflow

controllo di flusso: numero di byte che il destinatario è disposto a accettare

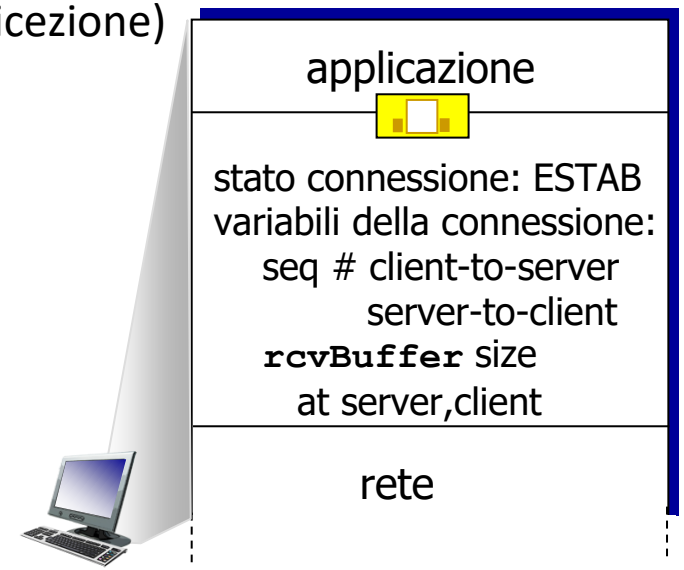


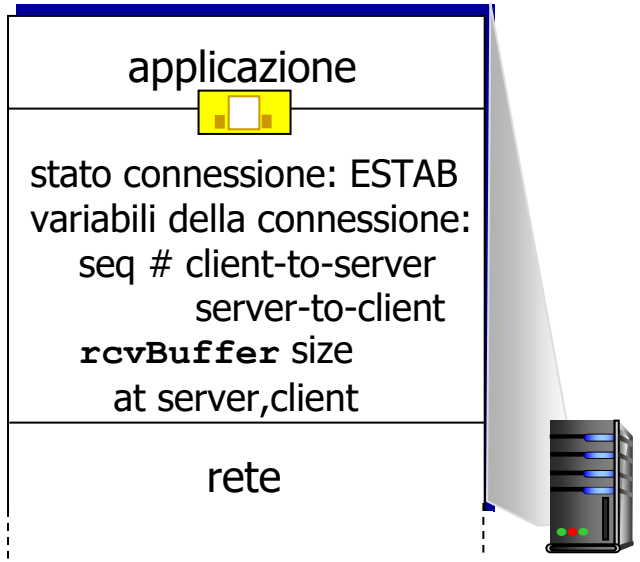
Formato del segmento TCP

Gestione della connessione TCP

Prima di scambiare i dati, il mittente e il destinatario si "stringono la mano" (“handshake”):

- accettano 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 sequenza iniziali e la dimensione del buffer di ricezione)



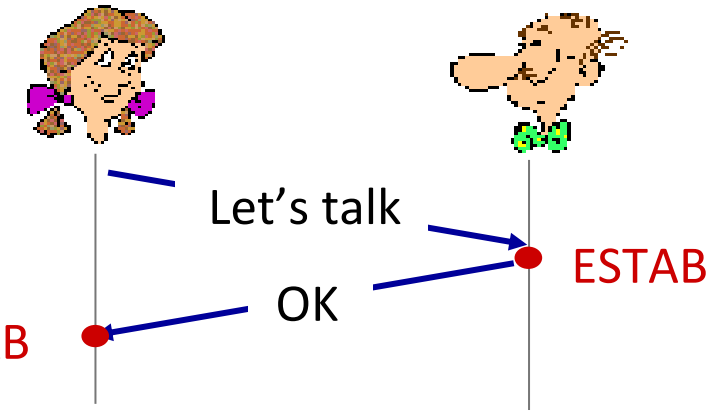


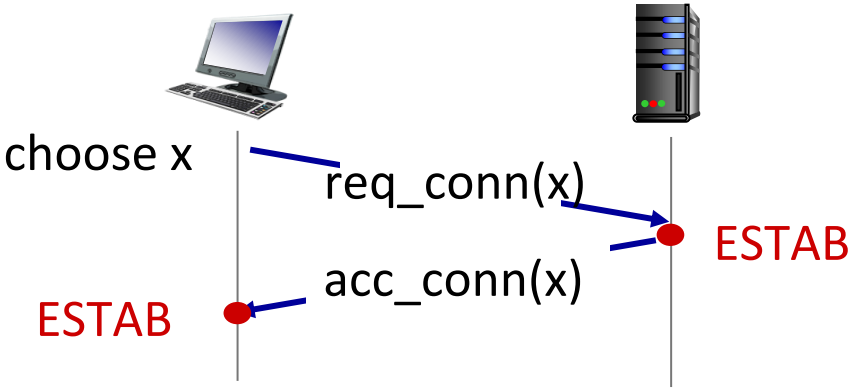

```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

Accettare di stabilire una connessione

handshake a 2 vie:

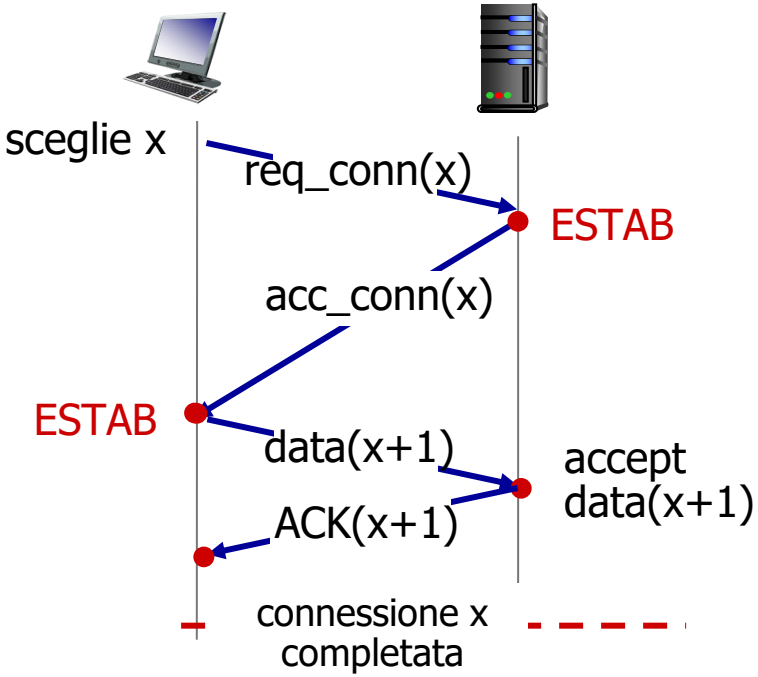




D: l'handshake a 2 vie funzionerebbe sempre?

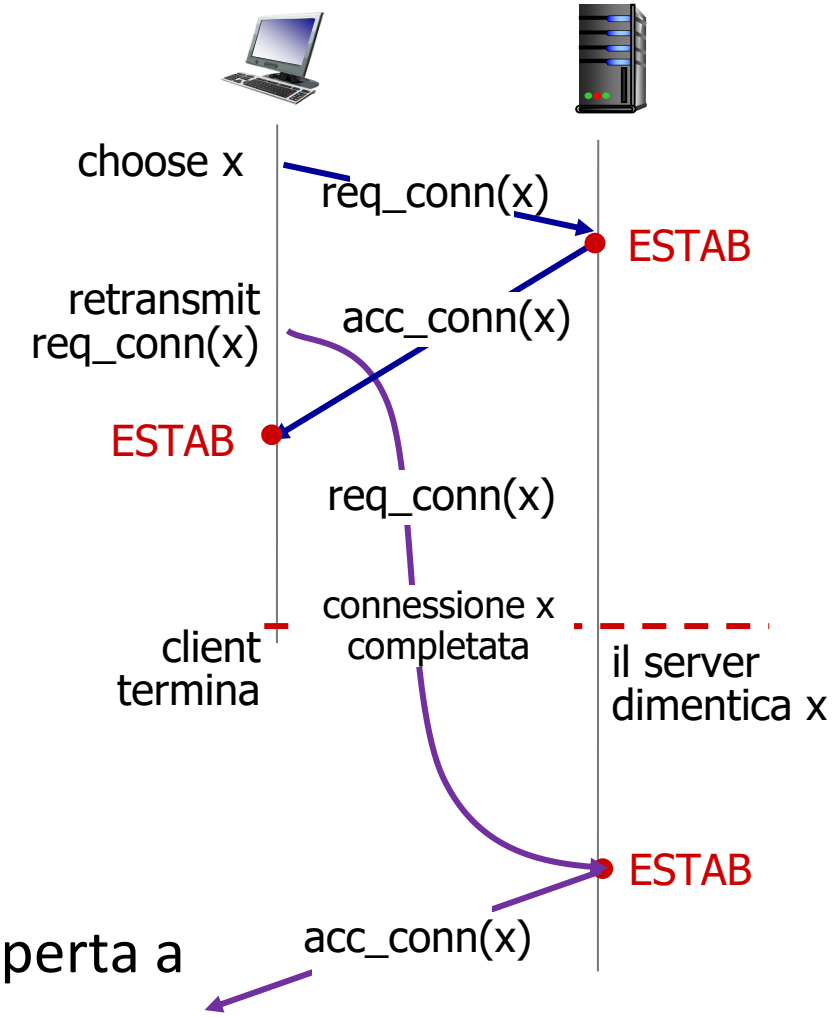
- ritardi variabili
- messaggi ritrasmessi (ad esempio, req_conn(x)) a causa della perdita di messaggi
- riordino dei messaggi
- impossibilità di "vedere" l'altro lato

Scenari per l'handshake a 2 vie



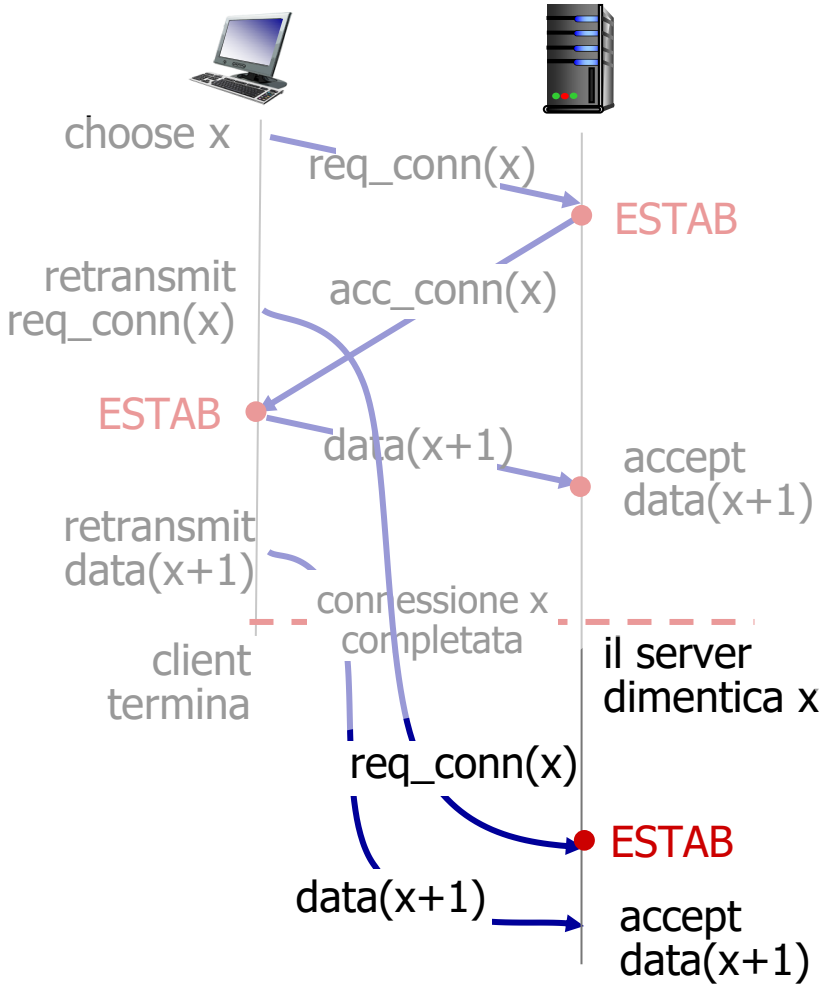
Nessun problema!

Scenari per l'handshake a 2 vie



Problema: connessione aperta a metà! (nessun client)

Scenari per l'handshake a 2 vie



Problema: dati duplicati accettati!

TCP 3-way handshake

Stato del Client

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

CLOSED

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

sceglie num. seq. iniziale x, invia segmento TCP SYN

SYNSENT

SYNbit=1, Seq=x

Stato del Server

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

LISTEN

sceglie num. seq. iniziale y, invia segmento TCP SYN ACK, riscontrando SYN

SYN RCVD

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

ricevuto SYNACK(x) indica che il server è vivo; invia ACK per il SYNACK; questo segmento può contenere dati da client a serve

ESTAB

ACKbit=1, ACKnum=y+1

ricevuto ACK(y) indica che il client è vivo

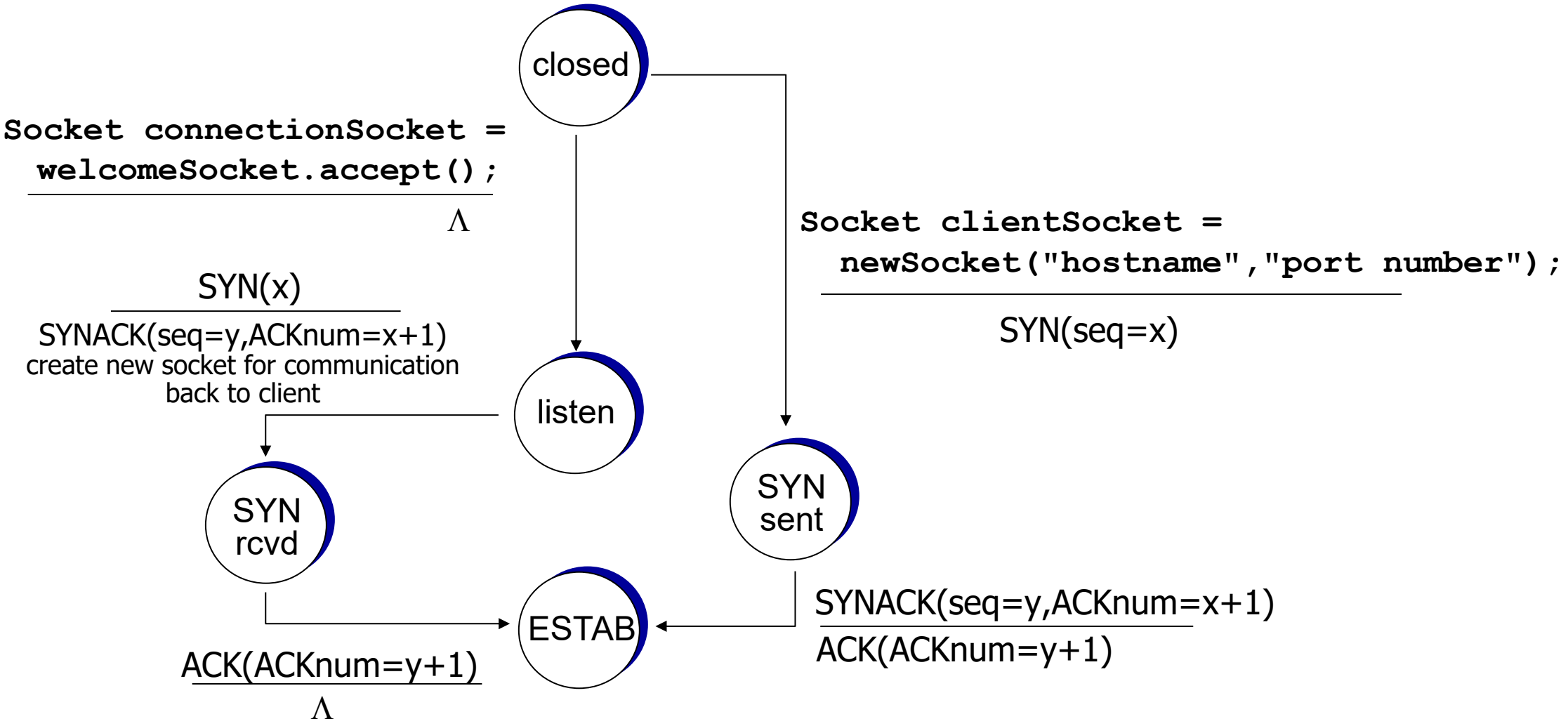
ESTAB

Entrambe le parti inizializzano il *sequence number* a caso, per evitare interferenze tra incarnazioni precedenti della stessa connessione

TCP 3-way handshake

- se un host riceve una richiesta di connessione per una porta su cui nessun socket è in ascolto, l'host invia al mittente un segmento con bit RST uguale a 1
- la ricezione di un segmento TCP RST informa un potenziale aggressore che la porta incriminata non è usata da alcun processo e che il segmento destinato a quella porta non è stato bloccato da un firewall lungo il percorso

TCP 3-way handshake: Macchina a Stati Finiti



Attacco SYN FLOOD

- L'aggressore invia un segmento TCP SYN ad un server con un indirizzo IP fasullo
- Il server alloca e inizializza le variabili e i buffer della connessione e risponde inviando un segmento TCP SYNACK alla porta e all'indirizzo IP (fasullo) di origine e transita nella stato SYN_RCVD
- La rete tenta di consegnare il segmento TCP SYNACK all'indirizzo IP fasullo, possibilmente raggiungendo un altro host che non c'entra nulla e che non risponderà
- Il server, nello strato SYN_RCVD, non ricevendo un ACK, eventualmente (dopo un minuto o più) rilascerà tutte le risorse associate a quella connessione mezza aperta
- Nel frattempo, però, l'aggressore è riuscito a consumare delle risorse del server. Inviando numerosi segmenti SYN ad un server obiettivo, un aggressore può montare un attacco DoS (spesso sottoforma di attacco DDoS)

Attacco SYN FLOOD: contromisure "SYN cookie" (semplificato)

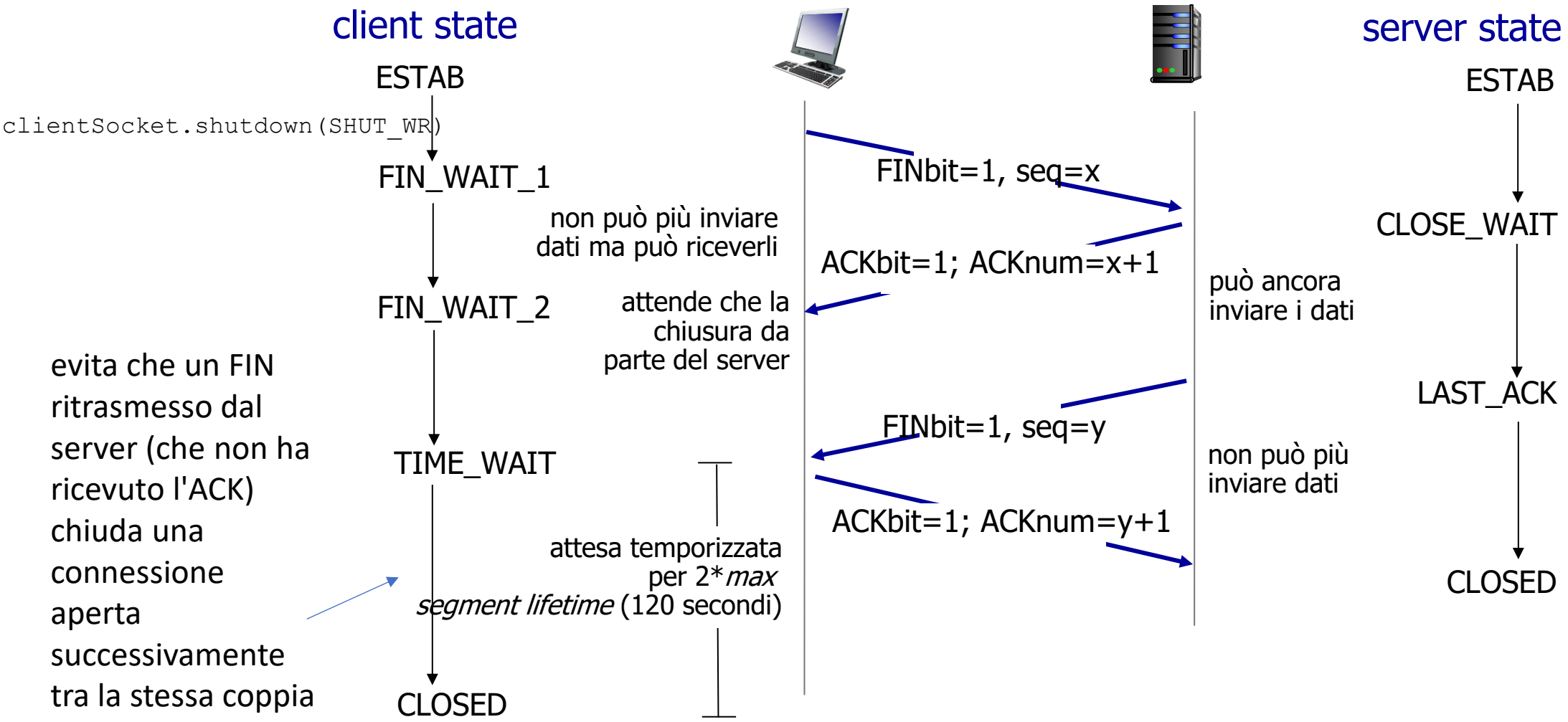
- Alla ricezione del segmento SYN iniziale, il server:
  - Calcola l'*hash* degli indirizzi IP e numeri di porta di origine e di destinazione e di una chiave segreta, producendo un *cookie*
    $cookie = hash(IP\ sorgente, IP\ destinazione, porta\ sorgente, porta\ destinazione, chiave\ segreta)$
  - usa il cookie con numero di sequenza iniziale da inserire dentro al segmento SYNACK
  - non alloca alcuna risorsa né memorizza il cookie
- Se il segmento SYN ricevuto in precedenza era legittimo, il client alla ricezione del segmento SYNACK risponde con un segmento ACK, che usa come numero di Acknowledgment *cookie + 1*
- Alla ricezione di un segmento ACK, il server può capire che è legato al SYN precedente perché gli basta calcolare Acknowledgment – 1 per ottenere il cookie. Quindi, riesegue il calcolo del cookie per vedere se ottiene lo stesso valore. Si noti che l'inclusione di una chiave segreta impedisce a un aggressore di creare un cookie valido.
- Per evitare attacchi replay, la funzione l'input hash dovrebbe includere una marca temporale, oppure la chiave segreta deve cambiare periodicamente.

Un protocollo umano di handshake a 3 vie

Chiusura di una connessione TCP

- client e server chiudono ciascuno il proprio lato della connessione
  - inviano 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

Chiusura di una connessione TCP



evita che un FIN ritrasmesso dal server (che non ha ricevuto l'ACK) chiuda una connessione aperta successivamente tra la stessa coppia di porte

Università degli Studi di Roma "Tor Vergata"
Laurea in Informatica

Sistemi Operativi e Reti
(modulo Reti)
a.a. 2024/2025

Livello di trasporto (parte4)

dr. Manuel Fiorelli
manuel.fiorelli@uniroma2.it
https://art.uniroma2.it/fiorelli

Basate sulle slide del libro di testo:
https://gaia.cs.umass.edu/kurose_ross/ppt.php

Capitolo 3: tabella di marcia

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

Principi del controllo della congestione

Congestione:

- informalmente: “troppe sorgenti inviano troppi dati troppo velocemente perché la *rete* li gestisca”
- sintomi:
  - lunghi ritardi (accodamento nei buffer dei router)
  - pacchetti persi (overflow nei buffer dei router)
- differisce dal controllo di flusso!
- tra i dieci problem più important del networking

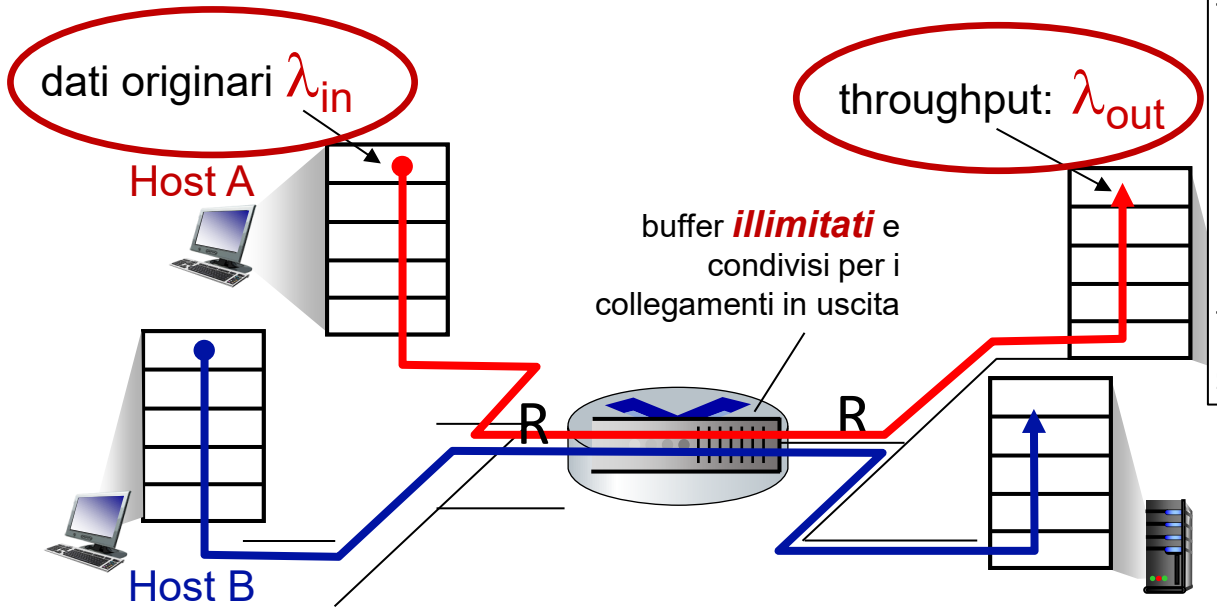
controllo della congestione: troppi mittenti, che trasmettono troppo velocemente

controllo di flusso: un mittente troppo veloce per il destinatario

Cause/costi della congestione: scenario 1

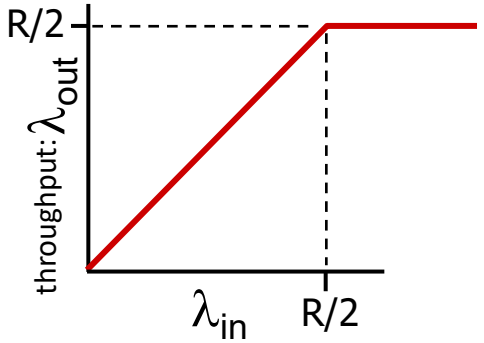
Caso più semplice:

- un router con buffer illimitati
- capacità dei collegamenti di ingresso e uscita: R
- due flussi
- nessuna ritrasmissione

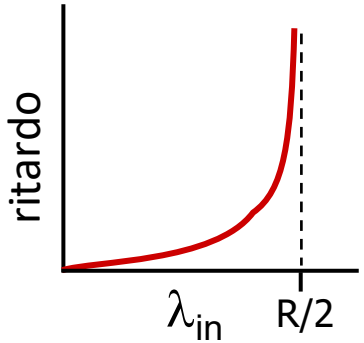


throughput a livello di applicazione (bit di informazione **utili** consegnati all'applicazione nell'unità di tempo), detto *goodput*

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



throughput massimo per connessione: R/2

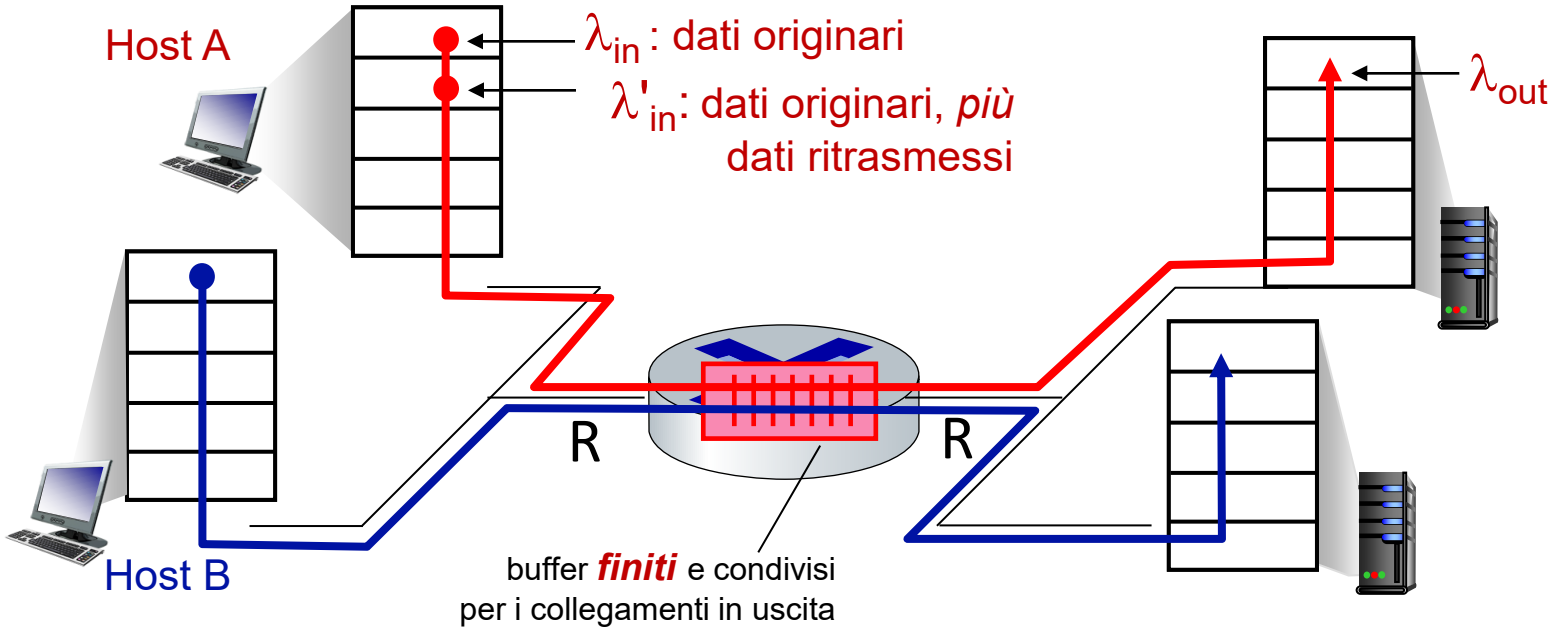


grandi ritardi quando il tasso di arrivo si avvicina alla capacità del collegamento

Cause/costi della congestione: scenario 2

- un router, buffer *finiti*
- il mittente ritrasmette pacchetti perduti (scartati perché il buffer era pieno)
  - input del livello di applicazione (tasso di trasmissione verso la socket): λ_{in}
  - input del livello di traporto (tasso di trasmissione verso la rete) include le *ritrasmissioni* : $\lambda'_{in} \geq \lambda_{in}$

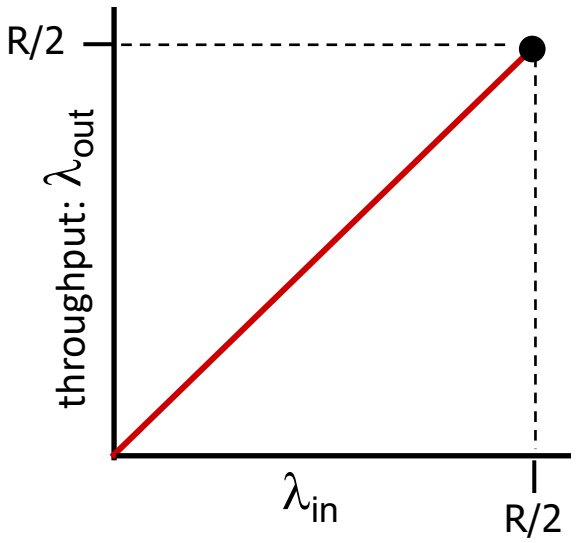
λ'_{in} detto anche carico offerto (offered load) alla rete

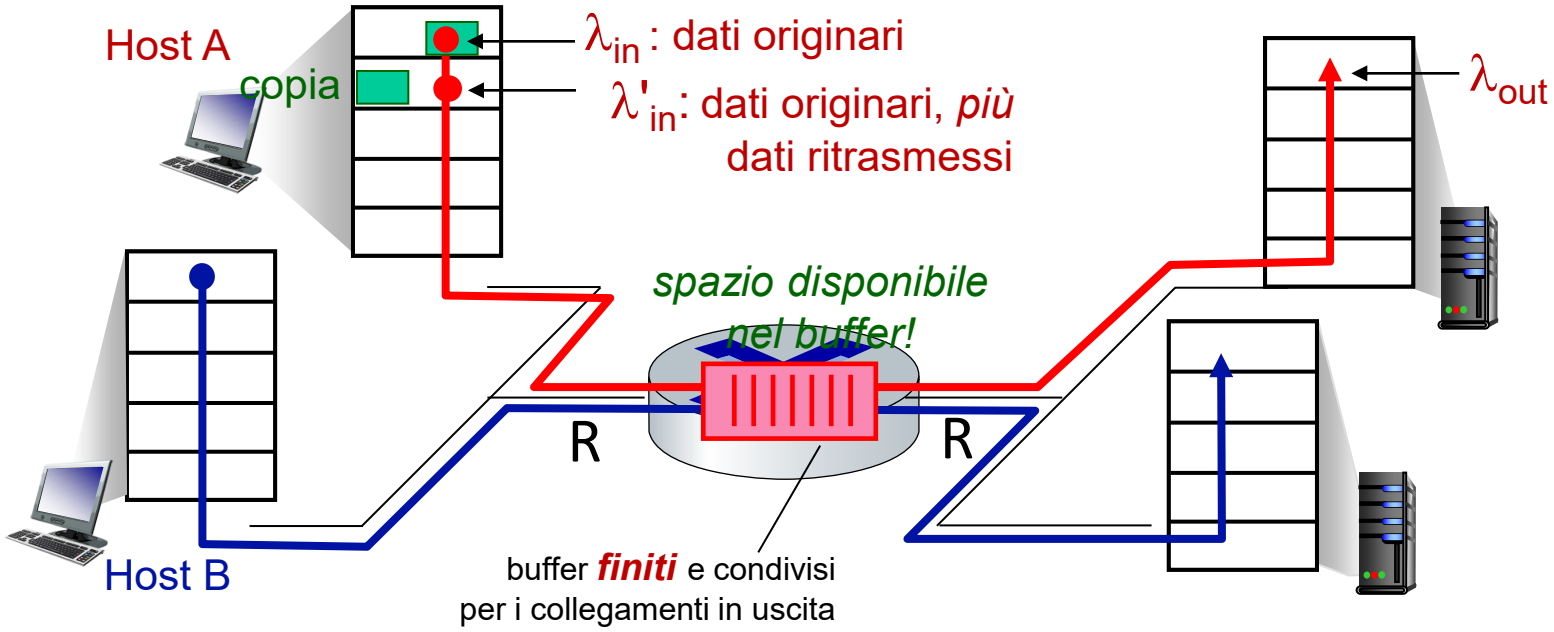


Cause/costi della congestione: scenario 2

idealizzazione: conoscenza perfetta

- il mittente invia solo quando c'è spazio disponibile nei buffer del router (nessuna perdita, $\lambda_{in} = \lambda'_{in}$)

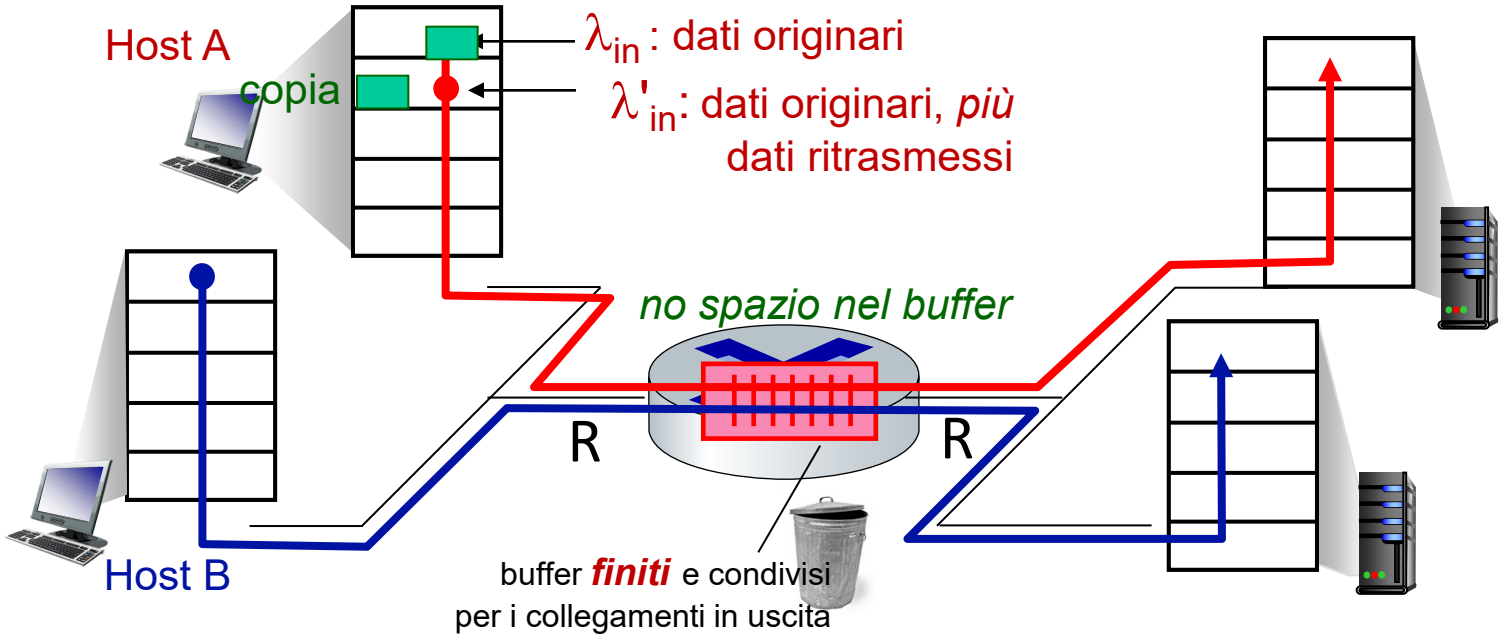




Cause/costi della congestione: scenario 2

idealizzazione: *un po' di conoscenza perfetta*

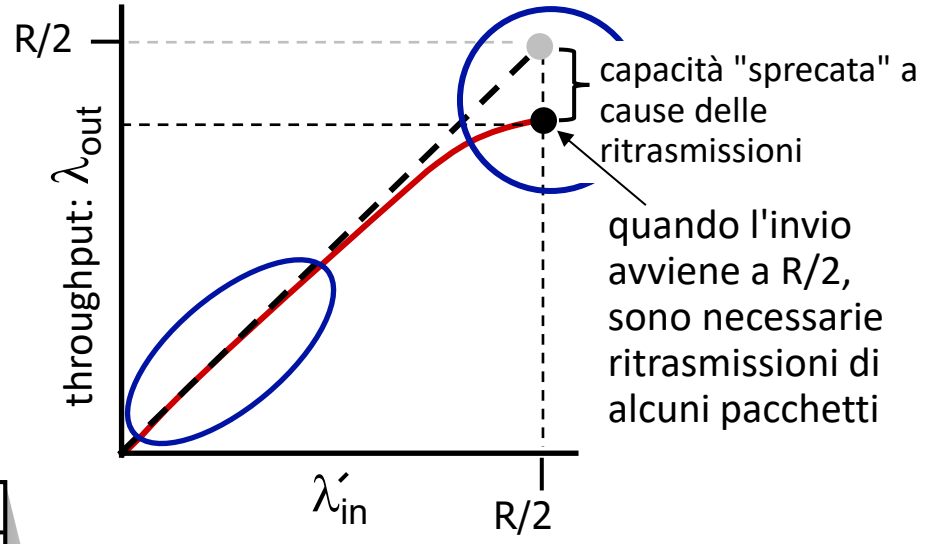
- i pacchetti possono essere persi (scartati nel router) a causa di buffer pieni
- il mittente sa quando un pacchetto è stato scartato: ritrasmette un pacchetto solo se *sa* che è stato perso

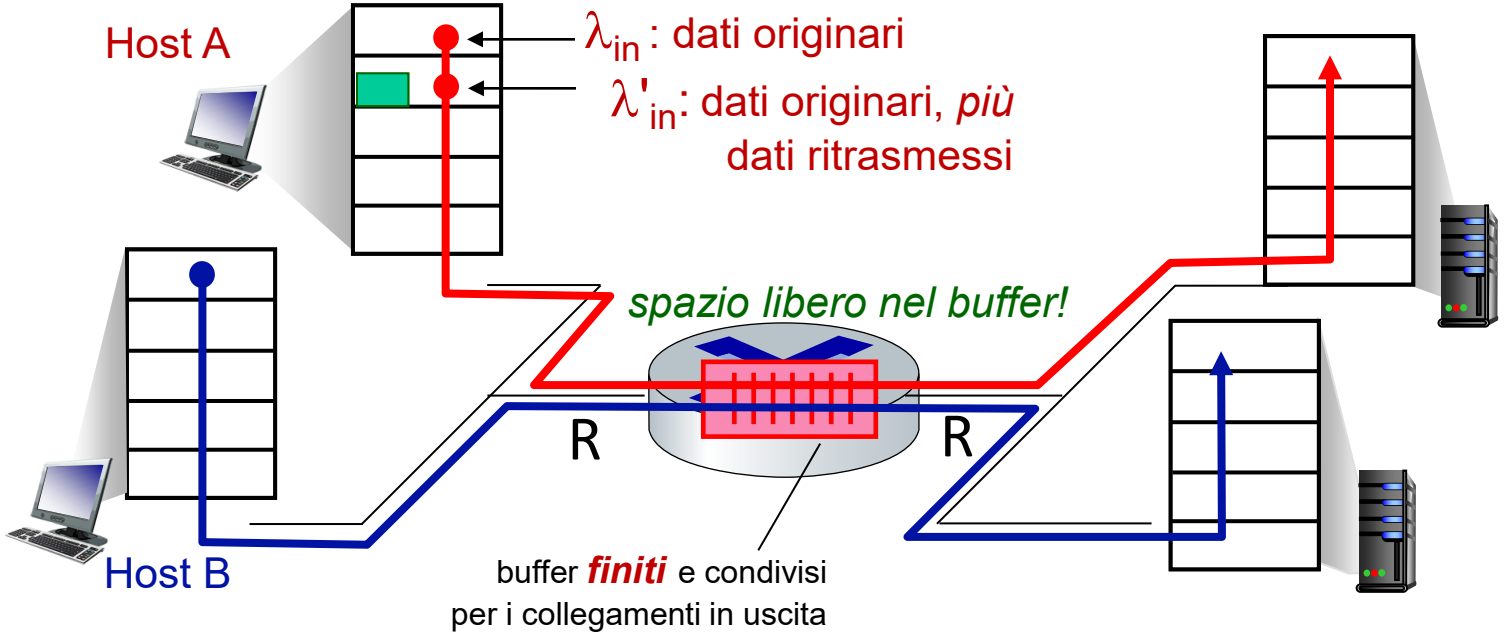


Cause/costi della congestione: scenario 2

idealizzazione: *un po' di conoscenza perfetta*

- i pacchetti possono essere persi (scartati nel router) a causa di buffer pieni
- il mittente sa quando un pacchetto è stato scartato: ritrasmette un pacchetto solo se *sa* che è stato perso

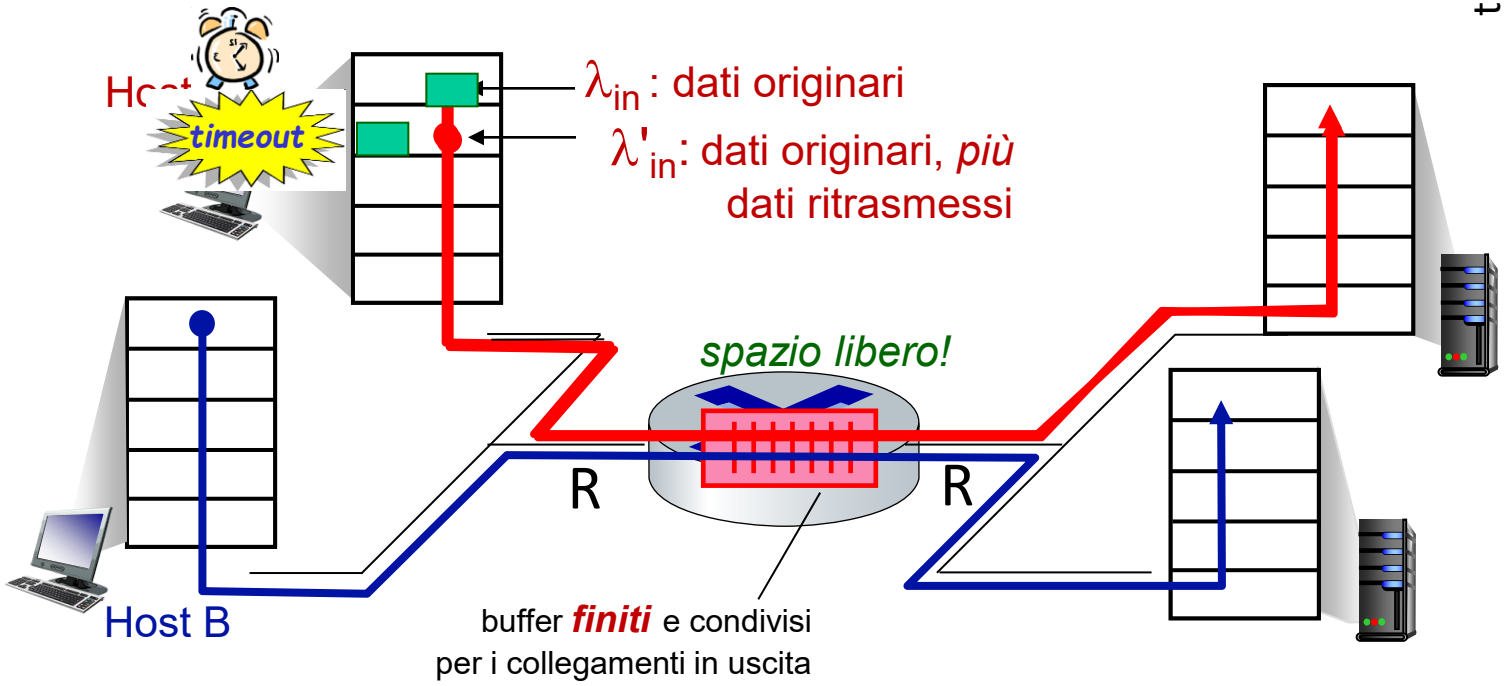


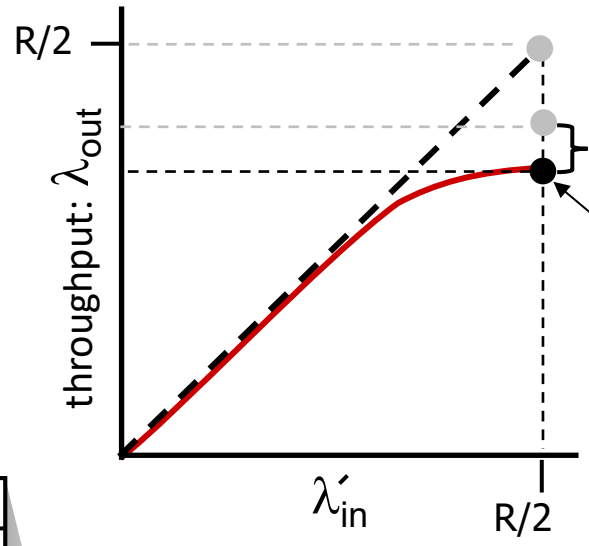


Cause/costi della congestione: scenario 2

scenario realistico: *duplicati non necessari*

- i pacchetti possono essere persi, scartati dal router a causa dei buffer pieni, richiedendo ritrasmissioni
- ma il mittente può andare in timeout prematuramente, inviando *due* copie, che vengono *entrambe* consegnate





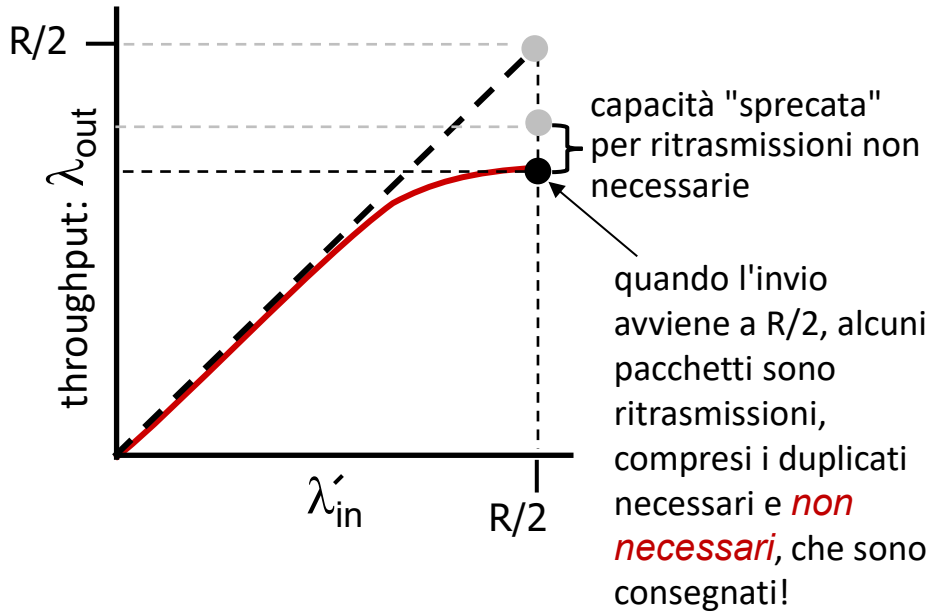
capacità "sprecata" per ritrasmissioni non necessarie

quando l'invio avviene a R/2, alcuni pacchetti sono ritrasmissioni, compresi i duplicati necessari e *non necessari*, che sono consegnati (al livello di trasporto che poi scarterà i duplicati)!

Cause/costi della congestione: scenario 2

scenario realistico: *duplicati non necessari*

- i pacchetti possono essere persi, scartati dal router a causa dei buffer pieni, richiedendo ritrasmissioni
- ma il mittente può andare in timeout prematuramente, inviando *due* copie, che vengono *entrambe* consegnate



"costi" della congestione;

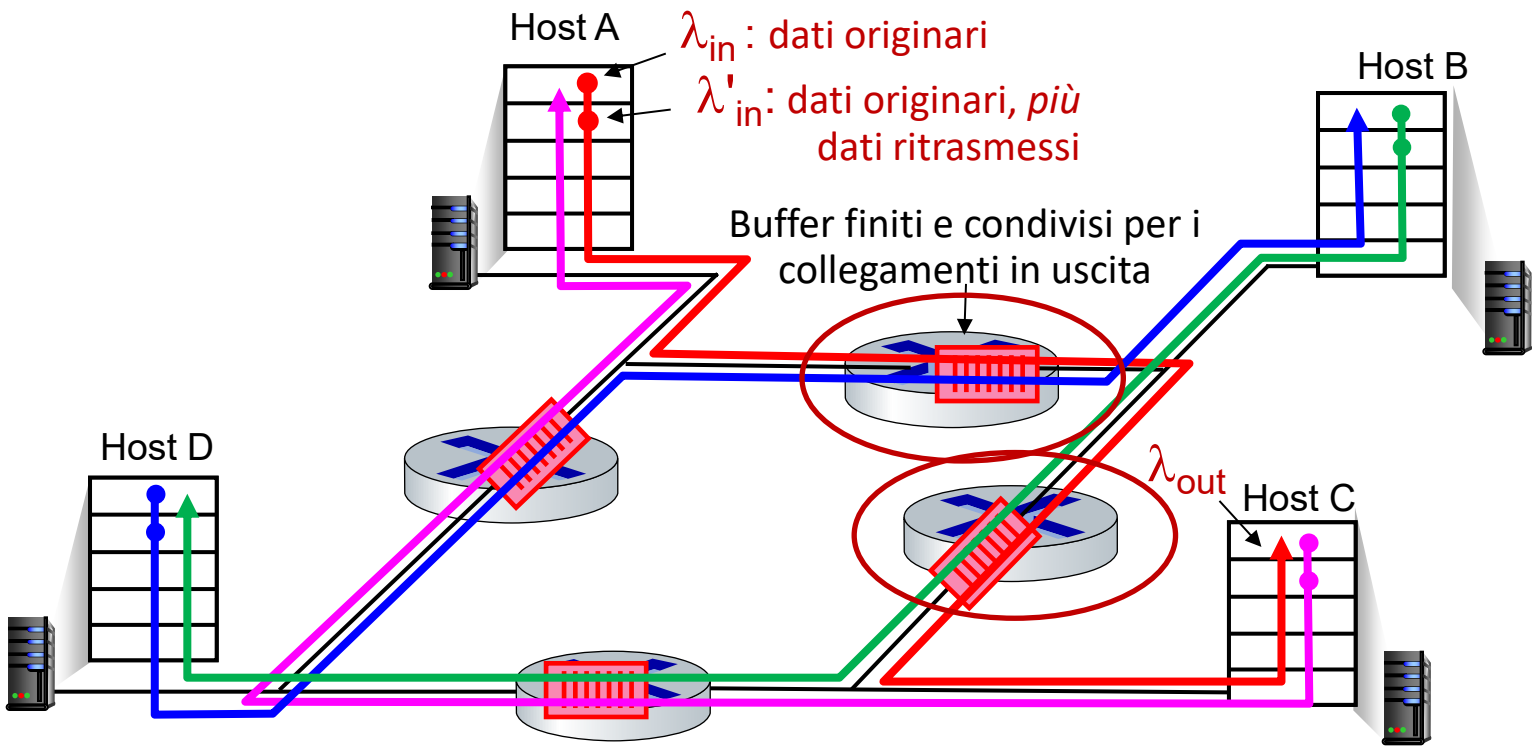
- più lavoro (ritrasmissioni) per un dato throughput di ricezione
- ritrasmissioni non necessarie: il collegamento trasporta più copie del pacchetto
  - diminuzione del throughput massimo raggiungibile

Cause/costi della congestione: scenario 3

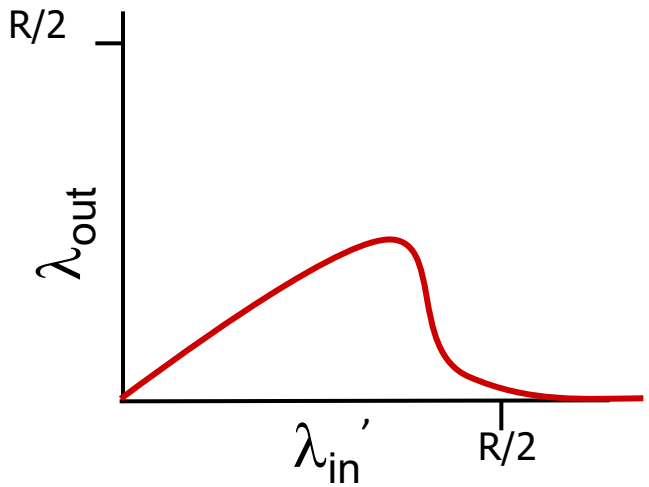
- *quattro mittenti*
- percorsi *multi-hop*
- timeout/ritrasmissione

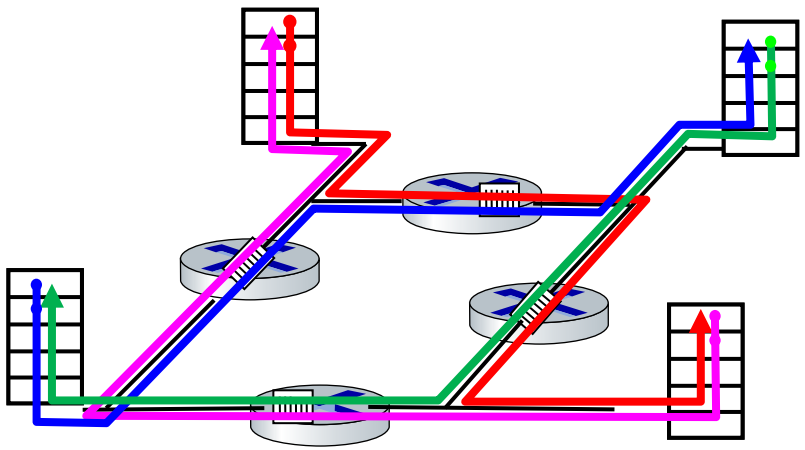
D: che cosa accade quando λ_{in} e λ_{in}' aumentano?

R: quando λ_{in}' aumenta, tutti i pacchetti blu in arrivo alla coda in alto sono scartati, throughput blu → 0



Cause/costi della congestione: scenario 3



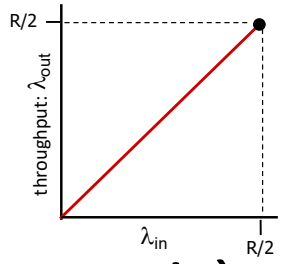


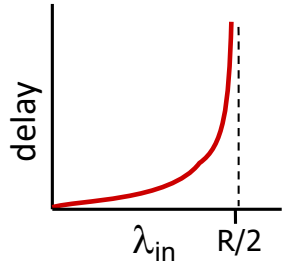
Un altro "costo" della congestione:

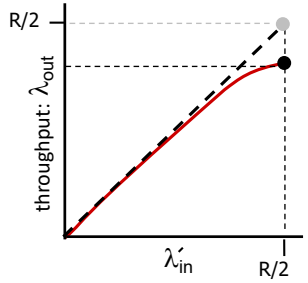
- Quando il pacchetto viene scartato, la capacità trasmissiva utilizzata sui collegamenti di upstream per instradare il pacchetto risulta sprecata!

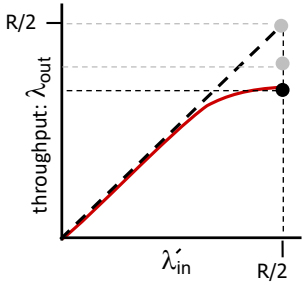
Cause/costi della congestione: intuizioni

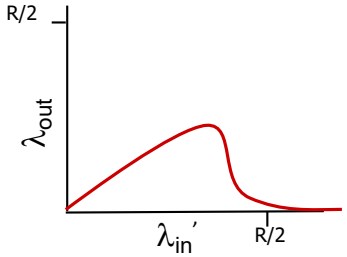
- il throughput non può mai superare la capacità
- il ritardo aumenta mentre ci si avvicina alla capacità
- la perdita/ritrasmissione diminuisce il throughput effettivo
- i duplicati non necessari diminuiscono ulteriormente il throughput effettivo
- capacità di trasmissione a monte / buffering sprecato per i pacchetti persi a valle







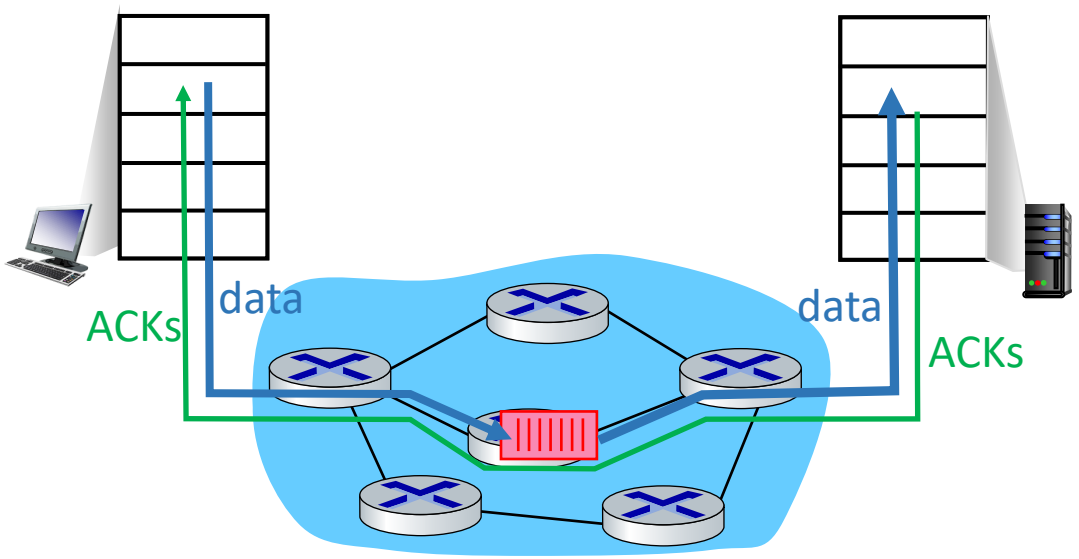




Approcci al controllo della congestione

Controllo della congestione end-to-end

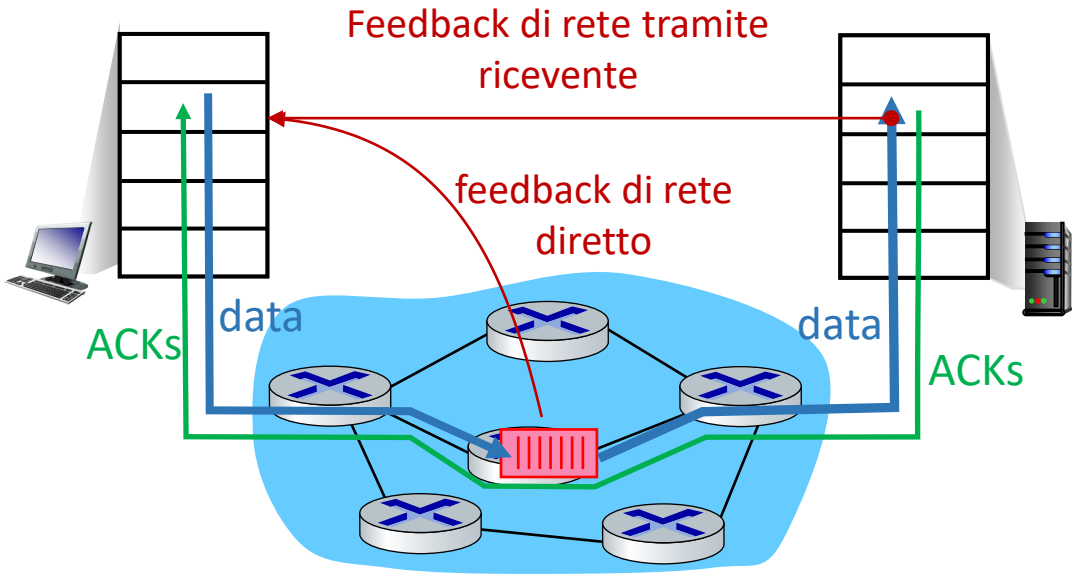
- nessun supporto esplicito dalla rete
- la congestione è *dedotta* osservando le perdite e i ritardi nei sistemi terminali
- metodo adottato da TCP



Approcci al controllo della congestione

Controllo della congestione assistito dalla rete:

- i router forniscono un feedback *diretto* all'host mittente tramite un *chokepacket* che lo avvisa dello stato di congestione
- oppure, un router può marcare i pacchetti che lo attraversano in modo tale che il destinatario alla sua ricezione informi il mittente
- possono indicare il livello di congestione o impostare esplicitamente un tasso di invio



- protocolli TCP ECN, ATM, DECbit

Capitolo 3: tabella di marcia

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

Trasmissione dati affidabile v. Controllo della congestione

- Trasmissione dati affidabile
  - reagisce alla perdita (e alla corruzione) dei pacchetti, possibilmente causata dalla congestione
  - "tratta i sintomi della congestione"
- Controllo della congestione
  - "cura la malattia"
  - evita che la malattia si aggravi fino a degenerare fino allo scenario di "collasso di congestione"

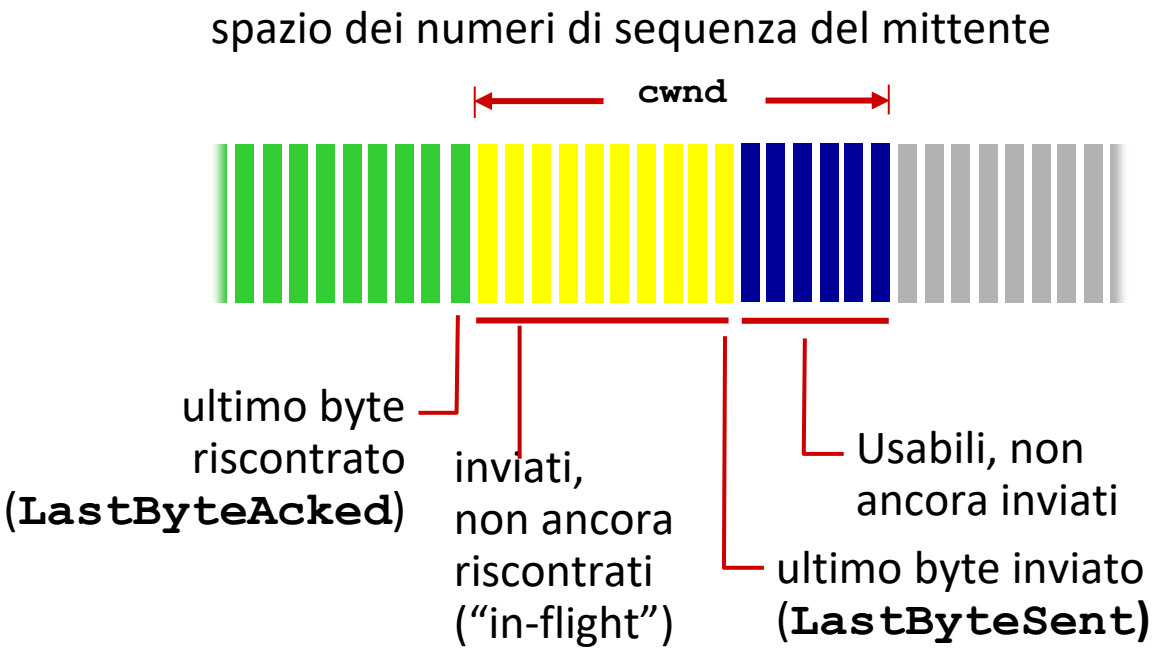
Controllo di congestione TCP

- TCP "classico" (come sviluppato da RFC 2581 e successivamente da RFC 5681)
  - controllo di congestione end-to-end
  - sviluppato in risposta alla serie di collassi di collasso di congestione subiti da Internet agli albori

    > Nell'ottobre dell'86, Internet subì il primo di una serie di "crolli di congestione". Durante questo periodo, il throughput dei dati dal LBL alla UC Berkeley ([...]) scese da 32 kbps a 40 bps.

    Van Jacobson; Michael J. Karels (November 1988), Congestion Avoidance and Control: https://ee.lbl.gov/papers/congavoid.pdf
- Evoluzioni recenti di TCP
  - indicazione di congestione esplicita fornita dalla rete
  - atre modifiche rispetto a TCP classico

Controllo della congestion TCP: dettagli



Comportamento di invio di TCP:

- *all'incirca:* invia `cwnd` byte, attende RTT per gli ACKS, quindi invia ulteriori byte

$$\text{tasso di invio} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ byte/s}$$

- Il mittente limita la trasmissione: `LastByteSent - LastByteAcked ≤ cwnd`
- `cwnd` viene regolata dinamicamente in risposta alla congestione della rete osservata (implementando il controllo della congestione TCP)

Relazione col controllo di flusso

- Il controllo del flusso regola la quantità e la velocità dei dati inviati in funzione della finestra di ricezione comunicata dal destinatario **`rwnd`** (byte liberi nel buffer di ricezione del destinatario)
- Quindi,

  `LastByteSent - LastByteAcked ≤ rwnd`
- Combinando questo vincolo con quello visto in precedenza

  `LastByteSent - LastByteAcked ≤ min{rwnd, cwnd}`

Assumendo che il buffer di ricezione sia sufficiente grande, possiamo trascurare il vincolo della finestra di ricezione (che assumiamo sempre maggiore della finestra di congestione)

Controllo di congestione TCP: incremento additivo e decremento moltiplicativo (AIMD)

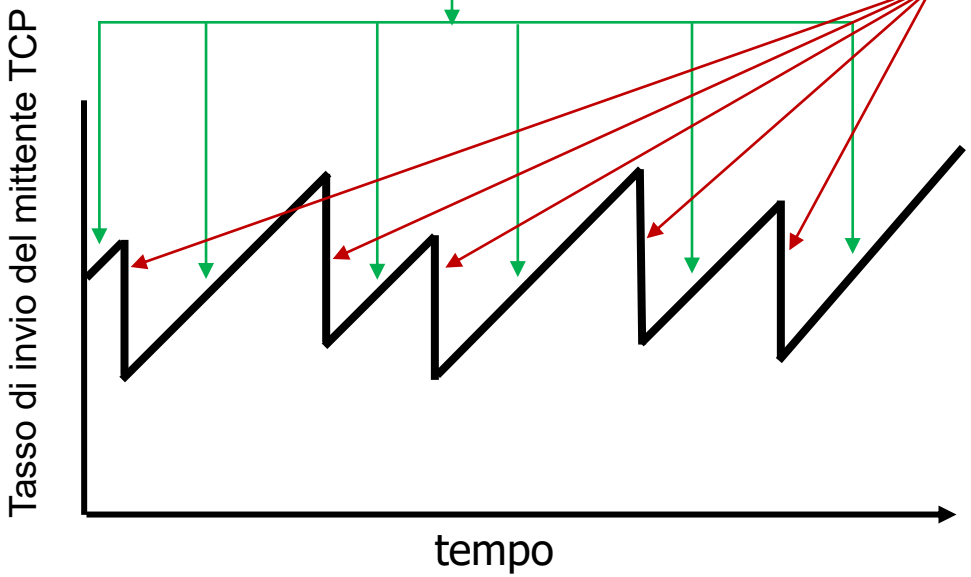
- *approccio:* i mittenti possono aumentare il tasso di invio, fino a quando non si verifica la perdita di pacchetti (congestione), quindi diminuire la velocità di invio in caso di perdita

Incremento additivo

aumentare la velocità di invio di 1 MSS ogni RTT fino a quando non viene rilevata una perdita

Decremento moltiplicativo

dimezzare la velocità di invio ad ogni evento di perdita



AIMD dente di sega: *sondare* la larghezza di banda

Questa è una descrizione di alto livello, che ignora la fase iniziale di slow start e assume che le perdite siano indicate da un triplo ACK duplicato in presenza di fast recovery

TCP AIMD: di più

Dettaglio decremento moltiplicativo:

In TCP Reno, il mittente riduce il tasso di invio in risposta a eventi di perdita:

- dimezzamento in caso di perdita rilevata da un triplo ACK duplicato (passando poi – per beve tempo – nella fase di fast recovery)
- taglio a 1 MSS ("maximum segment size") quando la perdita è rilevata dal timeout (ritornando poi nella fase di slow start)

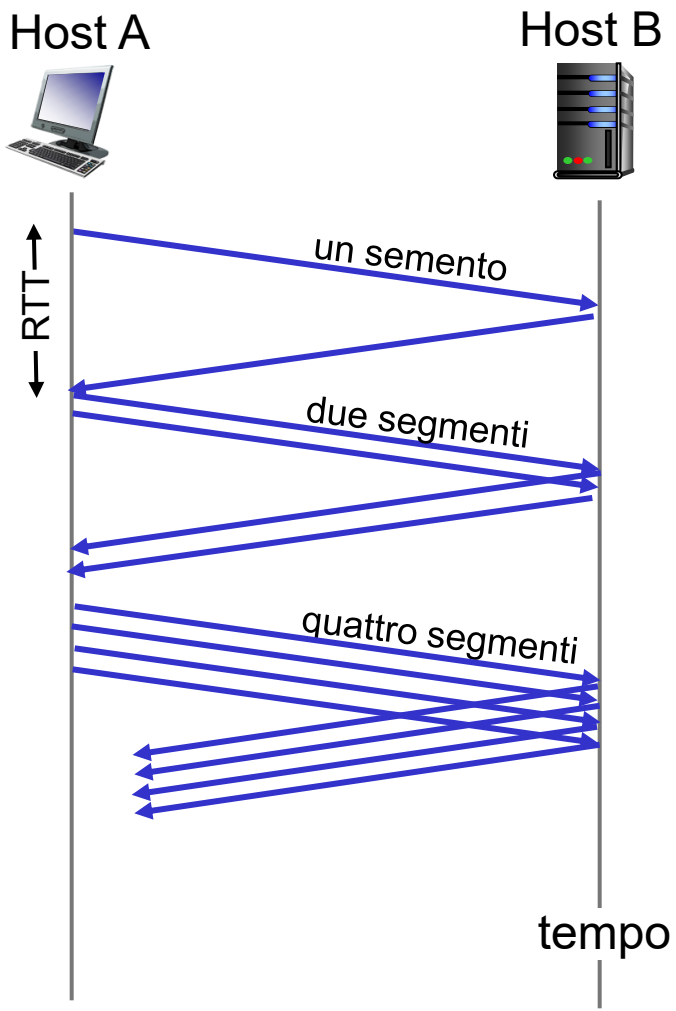
Una versione precedente, detta TCP Tahoe, la risposta a qualsiasi evento di perita era il taglio a 1 MSS e il passaggio a slow start

Perché AIMD?

- AIMD – un algoritmo asincrono, distribuito – è stato dimostrato che:
  - ottimizza i flussi congestionati in tutta la rete!
  - ha proprietà desiderabili di stabilità

Slow start (partenza lenta)

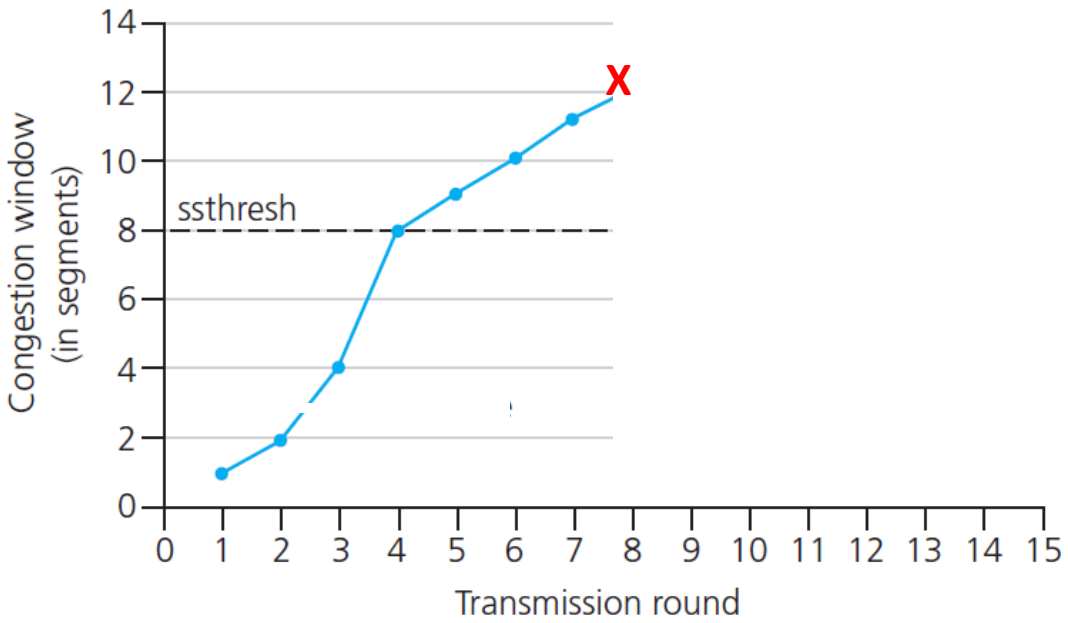
- Quando inizia la connessione, la frequenza aumenta in modo esponenziale fino a quando non si verifica un evento di perdita:
  - inizialmente **`cwnd`** = 1 MSS
  - raddoppia **`cwnd`** ogni RTT
  - fatto incrementando cwnd per ogni ACK ricevuto
- *sintesi*: il tasso iniziale è lento, ma aumenta in modo esponenziale e veloce



TCP: da slow start a congestion avoidance

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

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

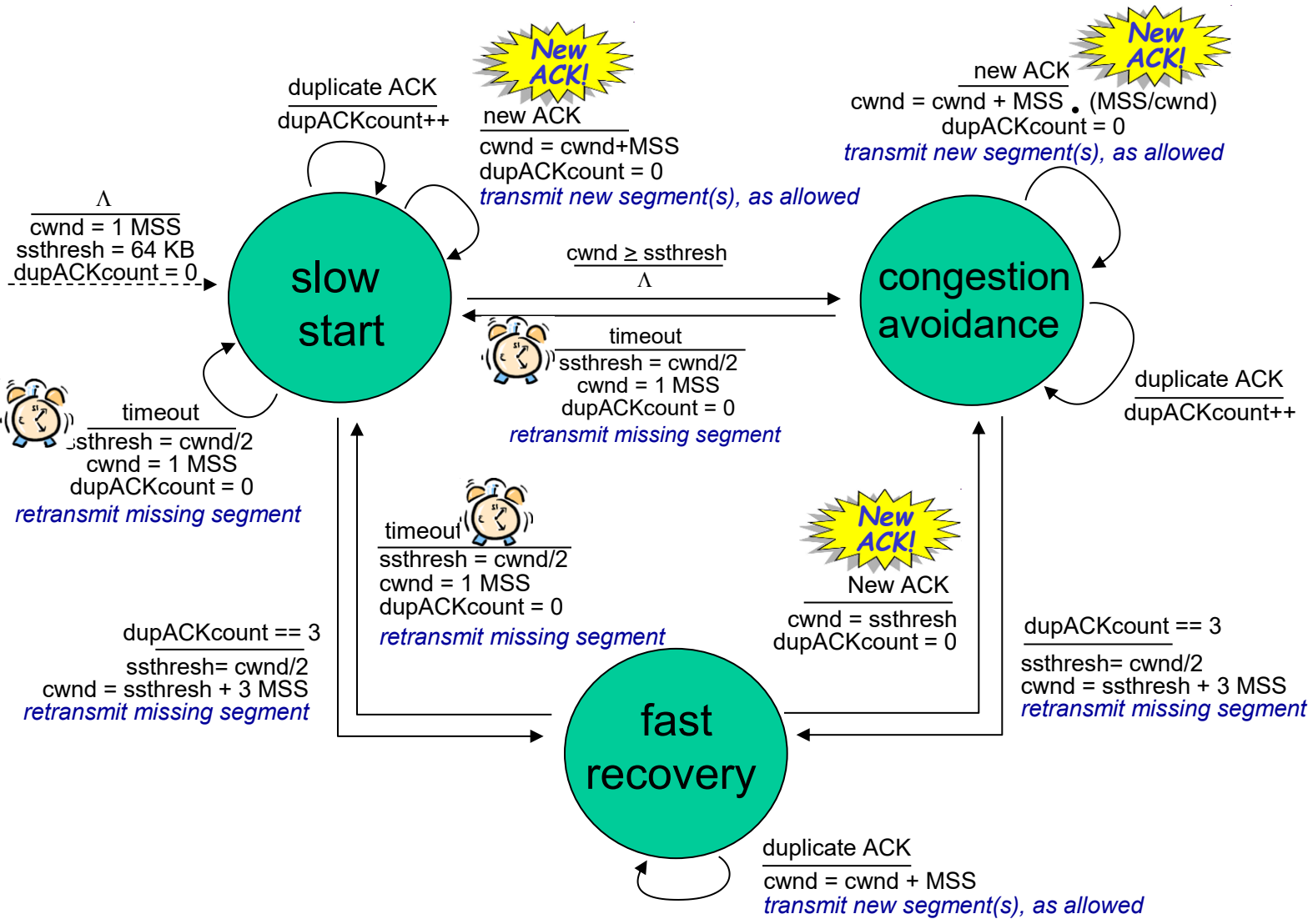


implementazione:

- **`ssthresh`** variabile (all'inizio 64 KB)
- in caso di evento di perdita, **`ssthresh`** è impostato a 1/2 **`cwnd`** giusto prima dell'evento di perdita

* Check out the online interactive exercises for more examples: http://gaia.cs.umass.edu/kurose_ross/interactive/

Riassunto: controllo della congestione TCP

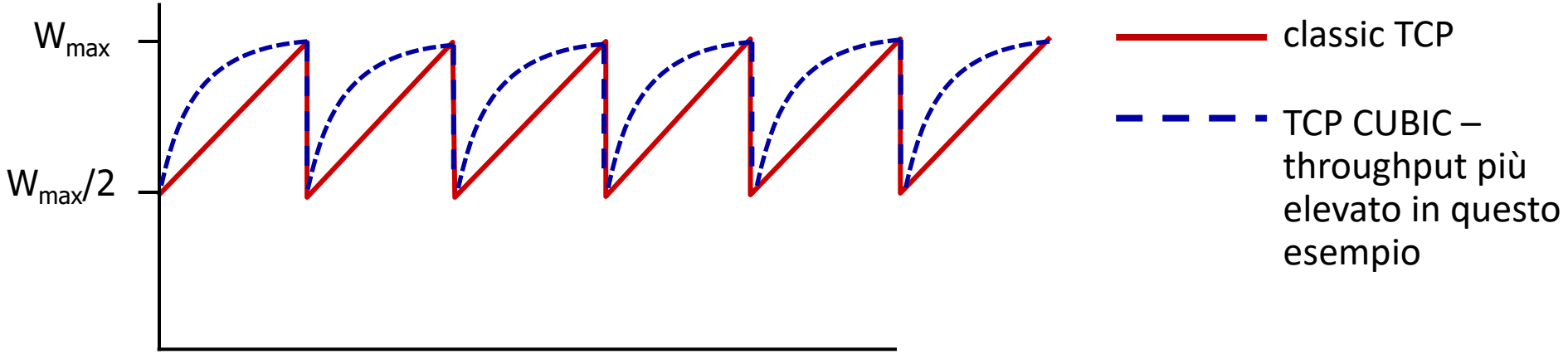


Riassunto: controllo della congestione TCP

- quando arriva un *nuovo ACK*, la finestra del mittente scorre verso destra e (a meno di recente riduzione della finestra di congestione), dovrebbe essere possibile l'invio nuovi segmenti (un segmento è uscito dalla rete e uno ne entra); quando la finestra di congestione viene incrementata sufficientemente può essere possibile l'invio di ulteriori segmenti (rivedete ciò che accade con slow start)
- nella fase di *congestion avoidance,* la finestra di congestione viene incrementata di $MSS \cdot \frac{MSS}{cwnd}$ per ogni nuovo ACK; poiché in un RTT ci si aspettano $\frac{cwnd}{MSS}$ ACK, il risultato finale (ottenuto moltiplicando queste due quantità) è un incremento di 1 MSS ogni RTT
- nella fase di *fast recovery*, in assenza di *nuovi ACK*, la finestra del mittente non avanza verso destra, bloccando *temporaneamente* l'invio di nuovi segmenti; tuttavia, la *cwnd* viene incrementata per ogni ACK duplicato (che indica l'arrivo a destinazione di un segmento successivo a quello potenzialmente perso) finché non diventa sufficientemente grande da permettere l'invio di nuovi segmenti (in media, in $\frac{1}{2}RTT$)
- la fase di *fast recovery* dura all'incirca un RTT affinché arrivi l'ACK del segmento ritrasmesso

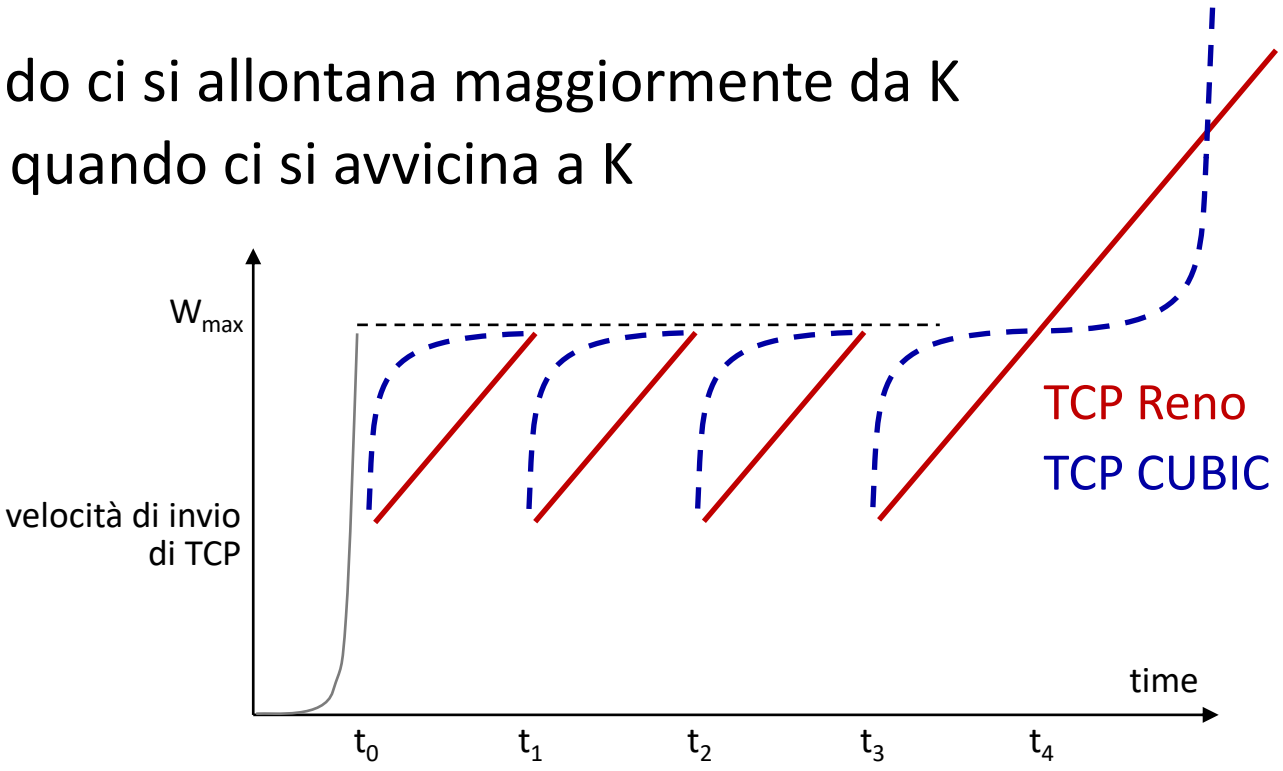
TCP CUBIC

- Esiste un modo migliore di AIMD per "sondare" la larghezza di banda utilizzabile?
- Intuizione:
  - W_{max}: la dimensione della finestra del controllo di congestione all'istante in cui viene rilevata la perdita
  - lo stato di congestione del collegamento bottleneck probabilmente (?) non è cambiato molto
  - dopo aver dimezzato la velocità/finestra in caso di perdita, inizialmente si sale verso W_{max} *più velocemente*, ma poi ci si avvicina a W_{max} *più lentamente*



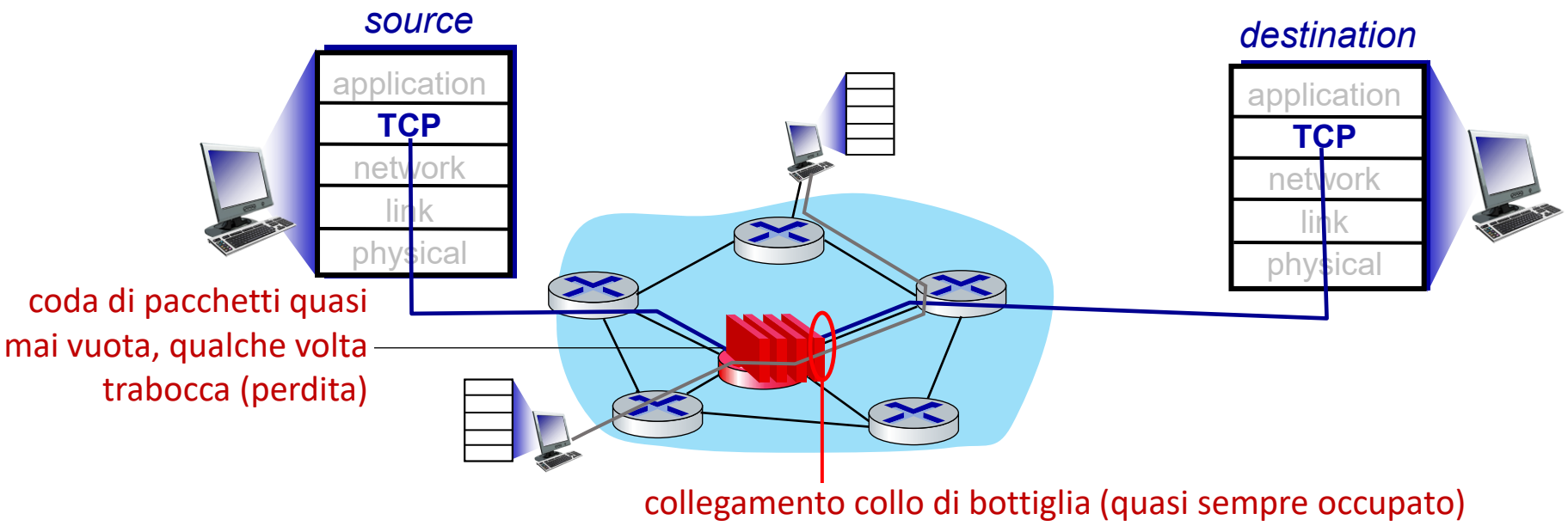
TCP CUBIC

- K: l'istante nel futuro in cui la finestra TCP raggiungerà nuovamente W_{max}
  - K è determinato da diversi parametri
- aumenta W come una funzione del *cubo* della distanza tra l'istante corrente e K
  - aumenti maggiori quando ci si allontana maggiormente da K
  - aumenti minori (cauti) quando ci si avvicina a K
- TCP CUBIC predefinito in Linux, il TCP più diffuso per i server Web più comuni



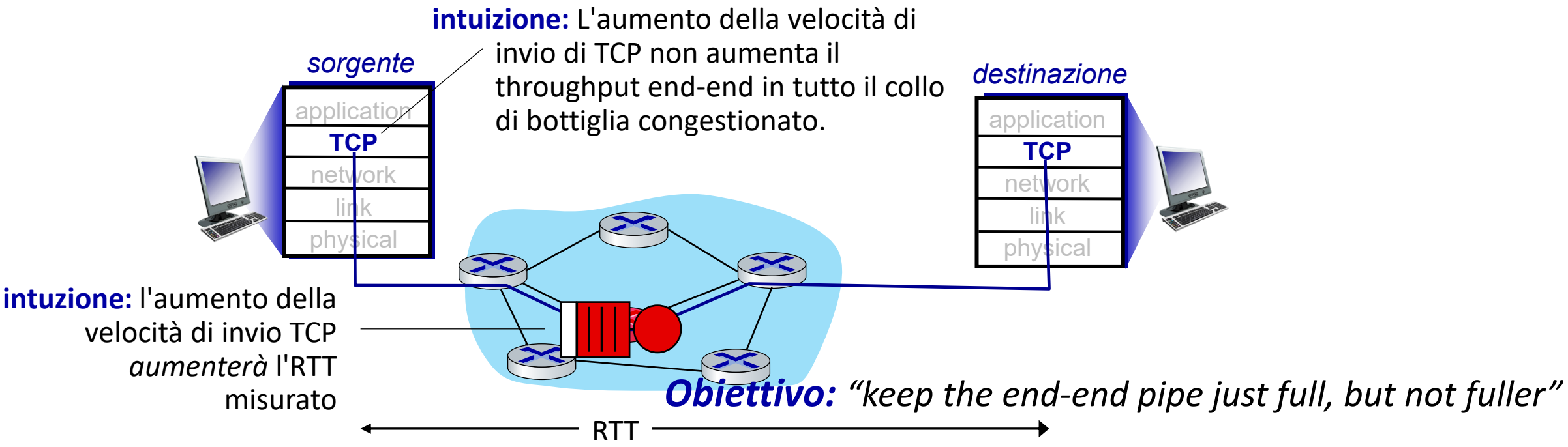
TCP e il collegamento "collo di bottiglia" congestionato

- TCP (classic, CUBIC) aumenta la velocità di invio di TCP finché non si verifica una perdita di pacchetti all'uscita di un router: il *collegamento "collo bottiglia" (bottleneck)*



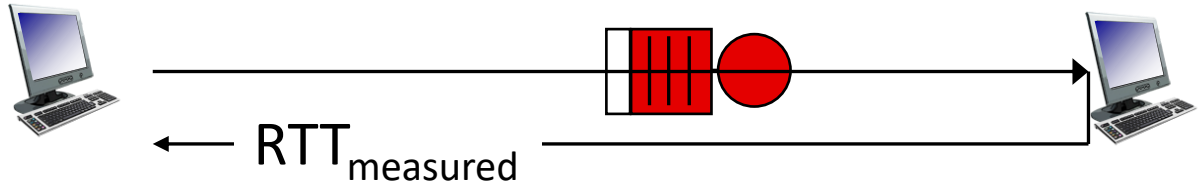
TCP e il collegamento "collo di bottiglia" congestionato

- TCP (classic, CUBIC) aumenta la velocità di invio di TCP finché non si verifica una perdita di pacchetti all'uscita di un router: il collegamento "collo bottiglia" (bottleneck)
- comprendere la congestione: utile concentrarsi sul collegamento con collo di bottiglia congestionato



Controllo di congestion basato sul ritardo

Mantenere la tubazione da mittente a destinatario "sufficientemente piena, ma non di più": mantenere il collegamento a collo di bottiglia impegnato nella trasmissione, ma evitare ritardi elevati/buffering.



$$\text{measured throughput} = \frac{\text{\# bytes sent in last RTT interval}}{RTT_{measured}}$$

Approccio basato sul ritardo (TCP Vegas):

- RTT_{min} - RTT minimo osservato (percorso non congestionato)
- throughput non congestionato con finestra di congestione `cwnd` è $cwnd/RTT_{min}$

se il throughput misurato è "molto vicino" al throughput non congestionato
    aumentare linearmente il cwnd /* poiché il percorso non è congestionato */
altrimenti se il throughput misurato è "molto inferiore" al throughput non congestionato
    diminuire linearmente cwnd /* poiché il percorso è congestionato */

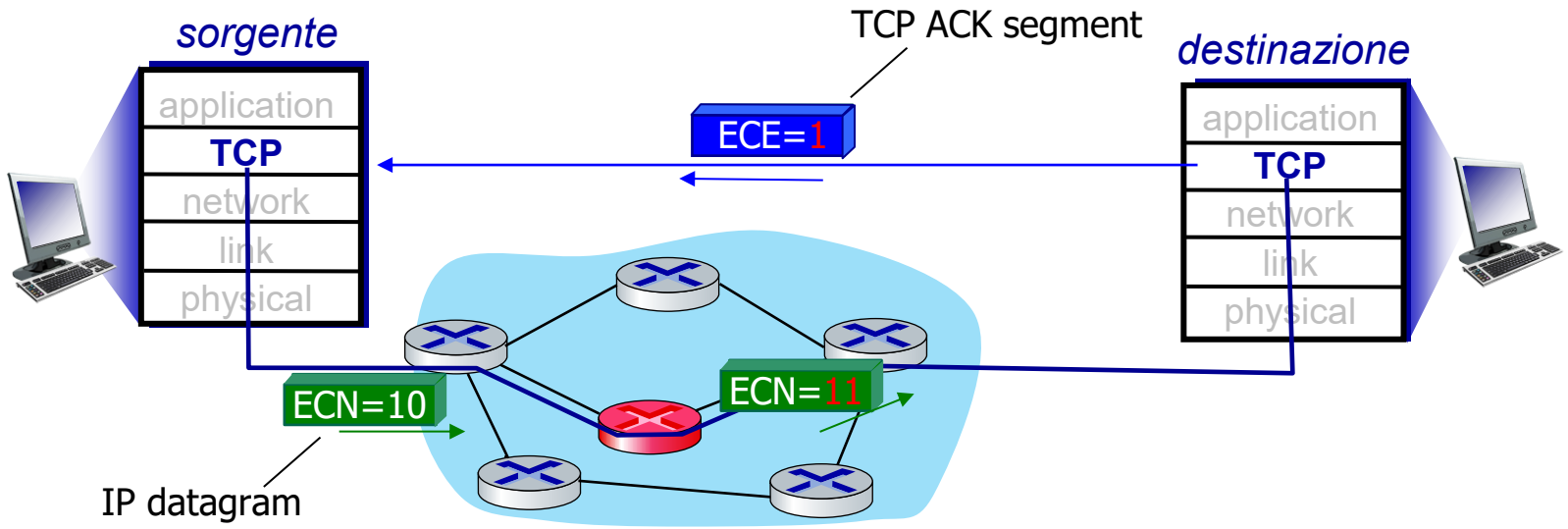
Controllo di congestion basato sul ritardo

- controllo di congestion senza indurre/forzare perdite
- massimizzare il throughput (“keeping the just pipe full… ”) ma mantenendo il ritardo basso (“…but not fuller”)
- un certo numero di TCP distribuiti adottano un approccio basato sui ritardi
  - BBR (Bottleneck Bandwidth and Round-trip propagation time) impiegato sulla rete dorsale (interna) di Google

Explicit congestion notification (ECN)

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

- due bit (ECN) nell'intestazione IP (all'interno del campo ToS) impostati *da un router di rete* per indicare la congestione
  - *policy* 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 dei bit ToS dell'intestazione IP) che il TCP (marcatura dei bit CWR,ECE dell'intestazione TCP).

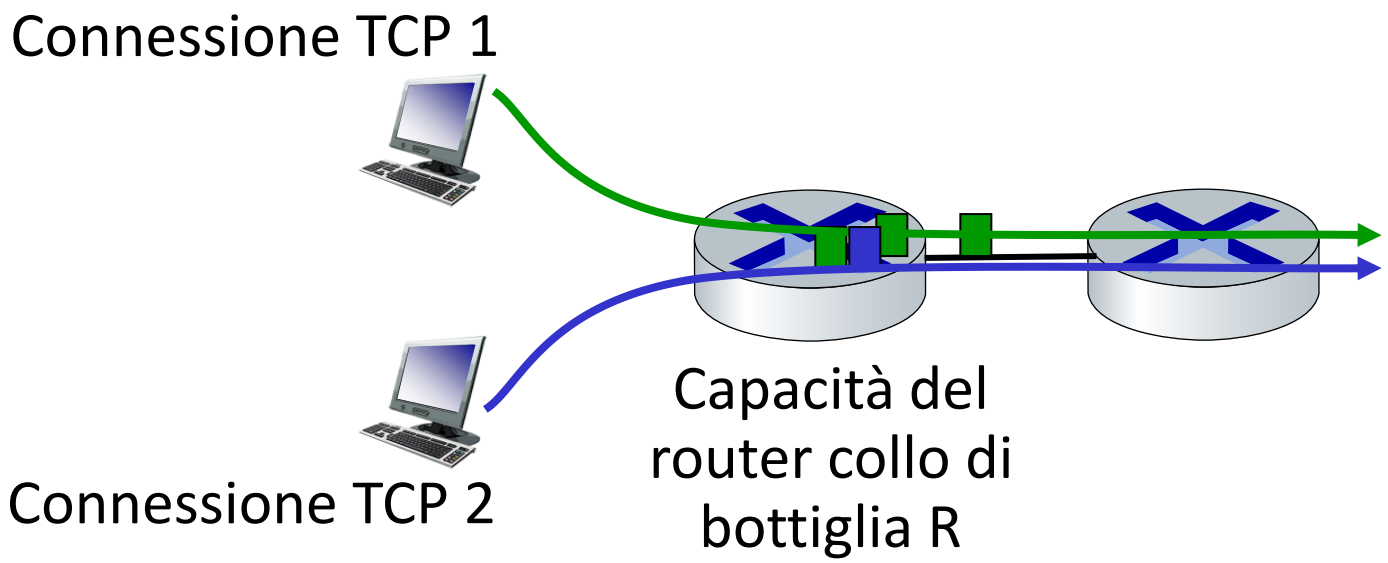


Explicit congestion notification (ECN)

- ECN viene negoziato in fase di instaurazione di una connessione TCP mediante opportune opzioni
- Il mittente imposta i bit ECN nell'intestazione IP per indicare che contengono i segmenti di una connessione in grado di gestire ECN

TCP fairness

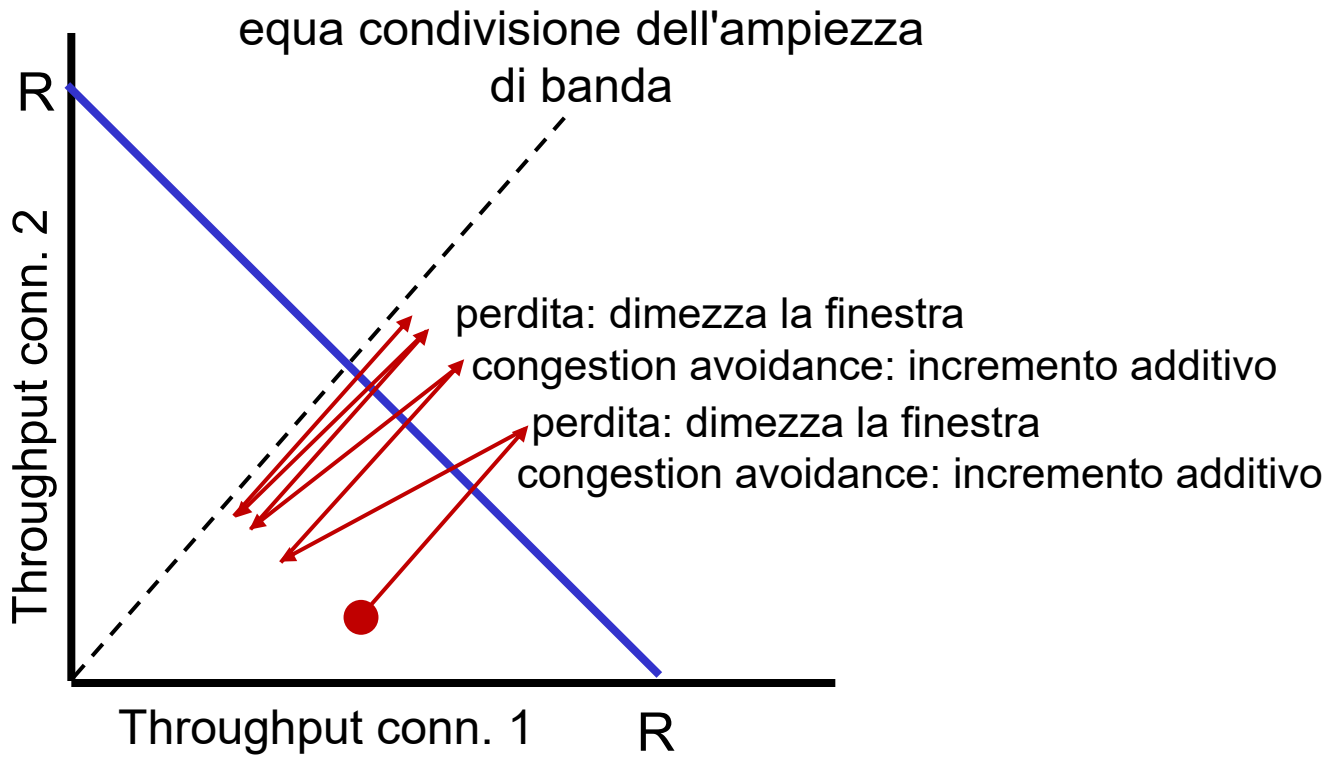
Obiettivo di equità: se *K* sessioni TCP condividono lo stesso collegamento a collo di bottiglia con larghezza di banda *R*, ciascuna dovrebbe avere una velocità media di *R/K*



D: TCP è Fair?

Esempio: due sessioni TCP in competizione:

- L'aumento additivo dà una pendenza di 1, quando il throughput aumenta
- la diminuzione moltiplicativa riduce il throughput in modo proporzionale



TCP è fair?

R: Sì, sotto assunzioni idealizzate:

- stesso RTT
- numero fisso di sessioni in congestion avoidance

Firness: tutte le app di rete devono essere "fair"?

Fairness e UDP

- Le app multimediali spesso non usano TCP
  - Non vogliono che la velocità sia ridotta dal controllo della congestione
- Usano invece UDP:
  - Inviano audio/video a velocità costante, tollerano la perdita di pacchetti
- Non esiste una "polizia di Internet" che controlli l'uso del controllo della congestione

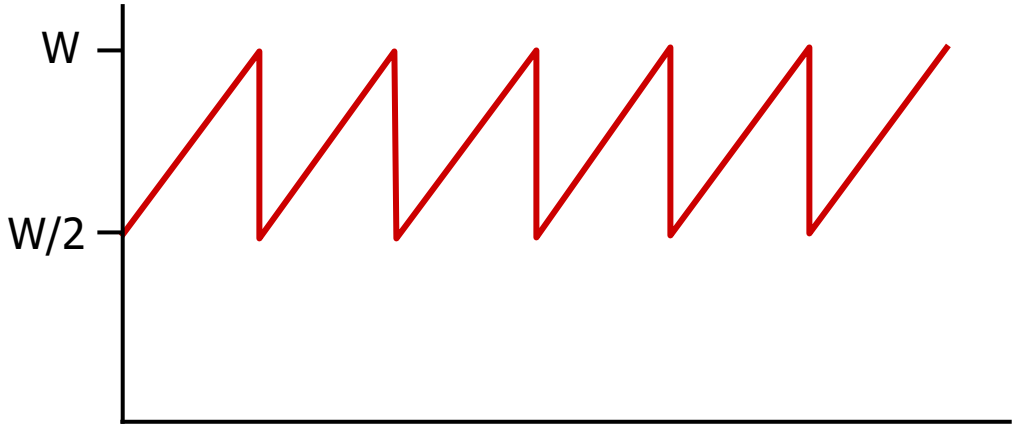
Fairness, connessioni TCP parallele

- L'applicazione può aprire *più* connessioni parallele tra due host
- I browser web lo fanno, ad esempio il link di velocità R con 9 connessioni esistenti:
  - nuova applicazione usa 1 connessione TCP, ottiene tasso R/10
  - nuova applicazione usa 11 connessioni TCP, ottiene tasso maggiore di R/2

Descrizione macroscopica del throughput di TCP

- Valore medio del throughput come funzione della dimensione della finestra e di RTT?
  - ignoriamo slow start, assumiamo che ci siano sempre dati da inviare
- W: dimensione della finestra (misurata in byte) quando si verifica una perdita
  - dimensione media della finestra (numero di byte "in volo") è ¾ W
  - throughput medio è 3/4W ogni RTT

$$\text{throughput TCP medio} = \frac{3}{4}\frac{W}{RTT} \text{ byte/s}$$



Livello di trasporto: tabella di marcia

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

Evoluzione della funzionalità del livello di trasporto

- TCP, UDP: i principali protocolli di trasporto per 40 anni
- Sono stati sviluppate diversi "varianti" (*flavor*)di TCP, per scenari specifici:

Scenario	Sfide
Long, fat pipes (trasferimenti di dati di grandi dimensioni)	Molti pacchetti "in volo"; la perdita interrompe la pipeline
Reti wireless	Perdita dovuta a collegamenti wireless rumorosi, mobilità; il TCP la tratta come perdita di congestione
Long-delay links	RTT estremamente elevato
Reti dei data center	Sensibili alla latenza
Background traffic flows	Flussi TCP a bassa priorità, "in background"

- Spostamento delle funzioni del livello di trasporto al livello di applicazione, in cima a UDP
  - HTTP/3: QUIC

TCP su "long, fat pipes"

- esempio: segmenti di 1500 byte, RTT di 100ms, vogliamo un throughput di 10 Gbps
- richiede W = 83˙333 segmenti in volo
- throughput in termini della probabilità di perdita dei pacchetti, L [Mathis 1997]:

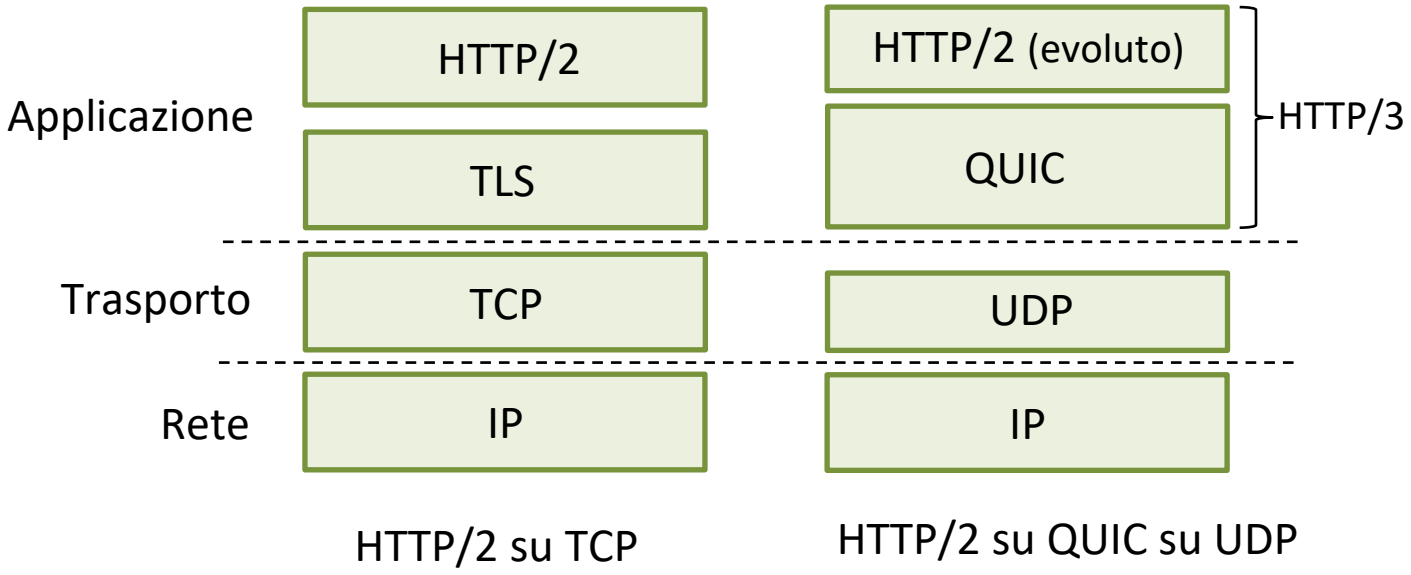
$$\text{TCP throughput} = \frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

➜ per raggiungere un throughput di 10 Gbps, occorre un tasso di perdita $L = 2 \cdot 10^{-10}$ *– un tasso di perdita davvero piccolo!*

- versioni di TCP per scenari lungi, ad alta velocità

QUIC: Quick UDP Internet Connections

- protocollo di livello applicazione, sopra a UDP
  - aumenta le prestazioni di HTTP
  - impiegati in molti server e app di Google (Chrome, app mobile di YouTube)



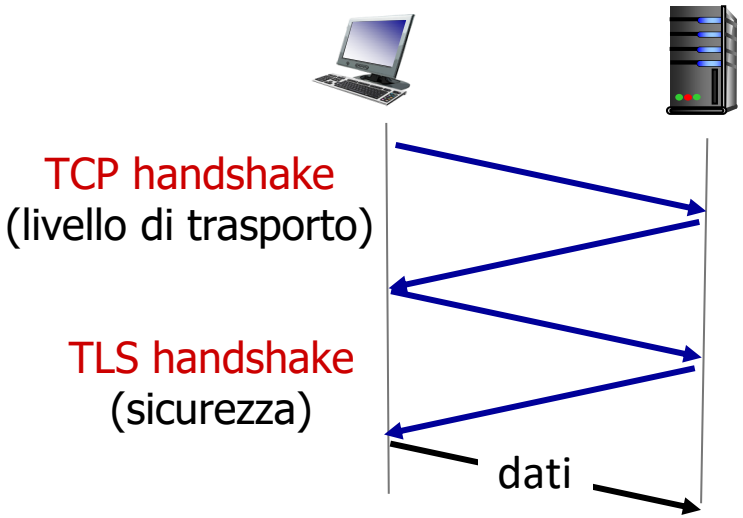
QUIC: Quick UDP Internet Connections

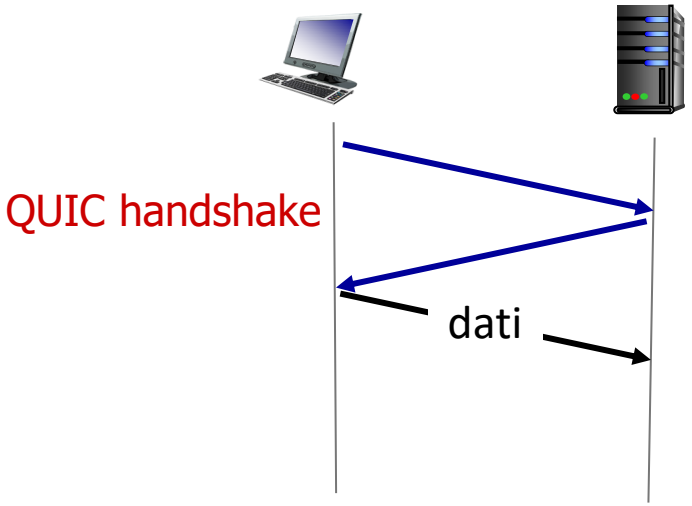
Adotta gli approcci che abbiamo studiato in queste lezioni per l'instaurazione della connessione, il controllo degli errori e il controllo della congestione

- **Controllo degli errori e della congestione:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [from QUIC specification]
- **Instaurazione della connessione:** affidabilità, controllo della congestione, autenticazione, cifratura, stato stability in un solo RTT

- multiplexing di molteplici "flussi" (*stream*) a livello di applicazione su una singola connessione QUIC
  - stato del trasferimento dati affidabile e della sicurezza separati
  - stato del controllo della congestione condiviso

QUIC: Instaurazione della connessione





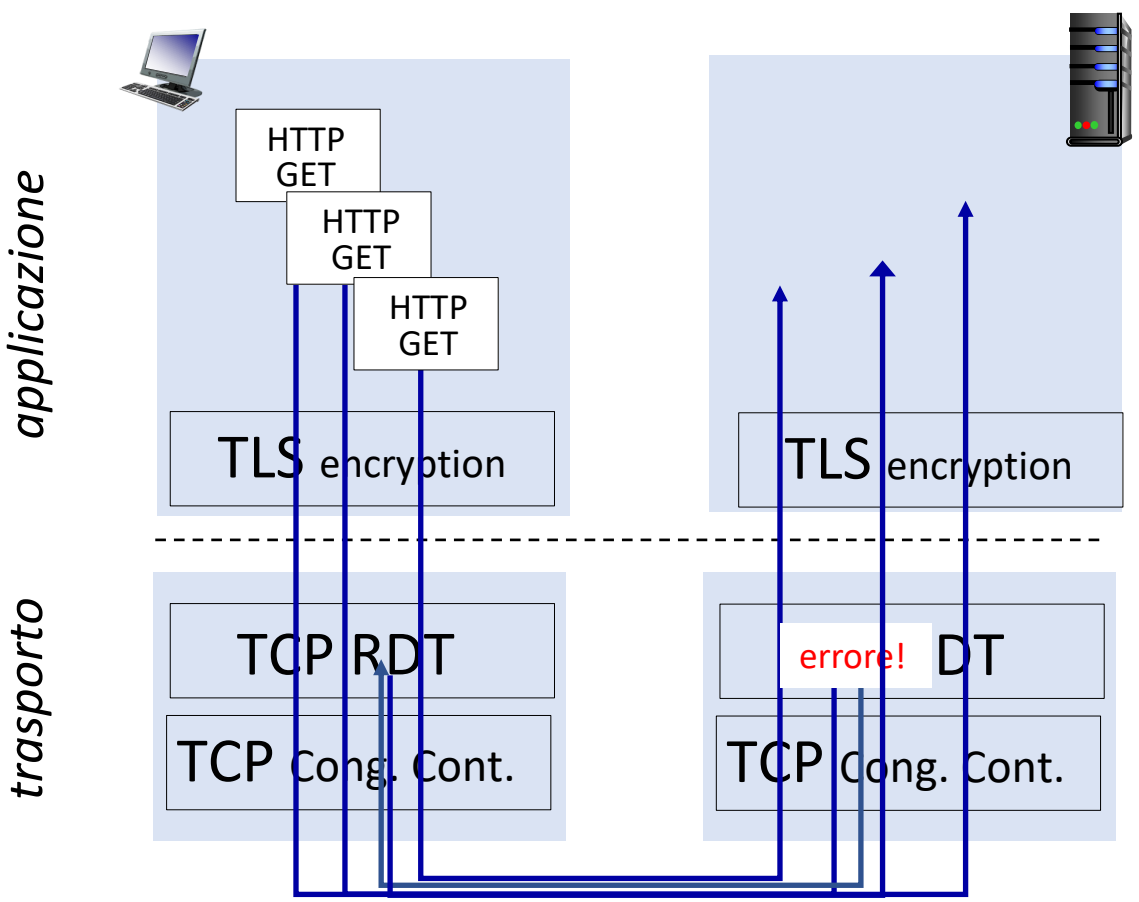
TCP (stato per il trasferimento affidabile e per il controllo della congestione) + TLS (stato per l'autenticazione autenticazione e per la cifratura)

- 2 handshake in successione

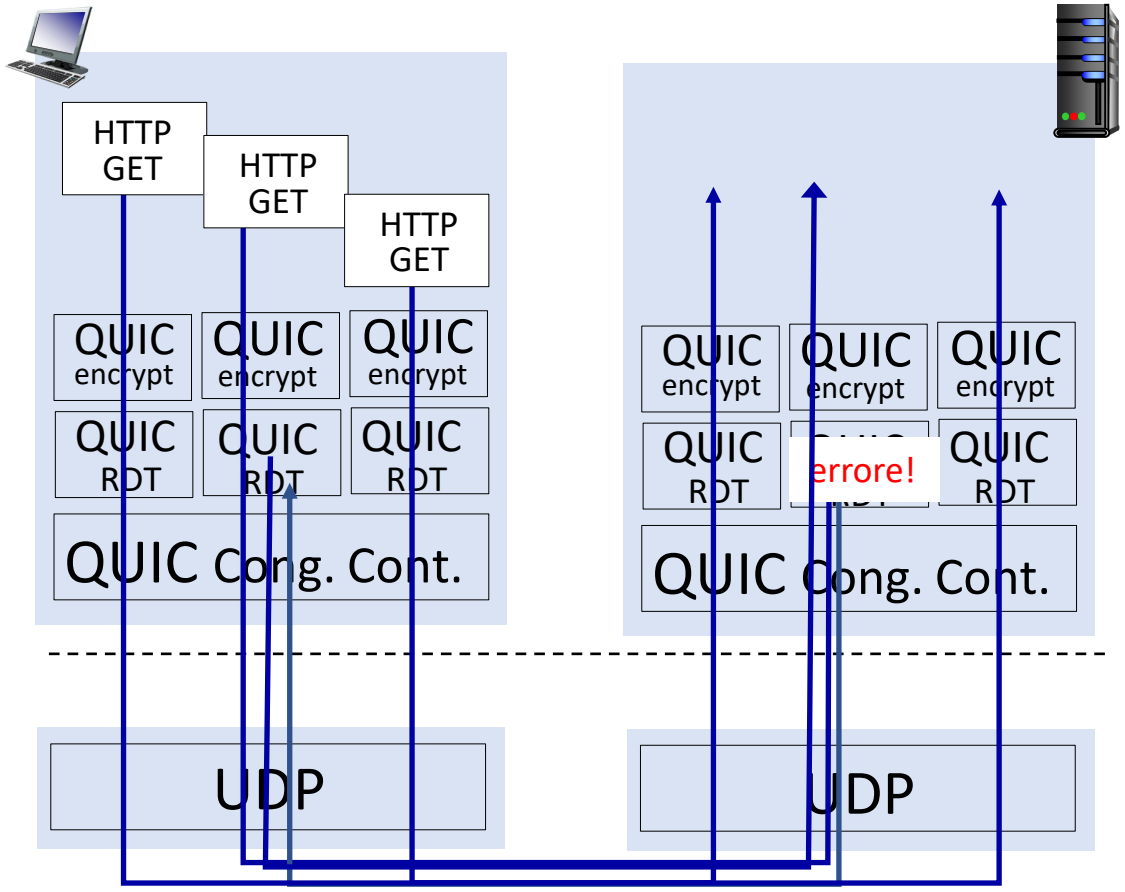
QUIC: stato della affidabilità, controllo della congestion, autenticazione e cifratura

- 1 handshake

QUIC: flussi: parallelismo, no blocco HOL



(a) HTTP 1.1



(b) HTTP/2 con QUIC: no HOL blocking

Capitolo 3: riassunto

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

Prossimamente :

- abbandonare la "periferia" (livelli di applicazione e di trasporto)
- per addentrarci nel "nucleo" della rete
  - piano dei dati
  - piano di controllo