Reti di Calcolatori A.A. 2012-2014

Capitolo 3

Livello di Trasporto

Luigi Vetrano

TechnoLabs

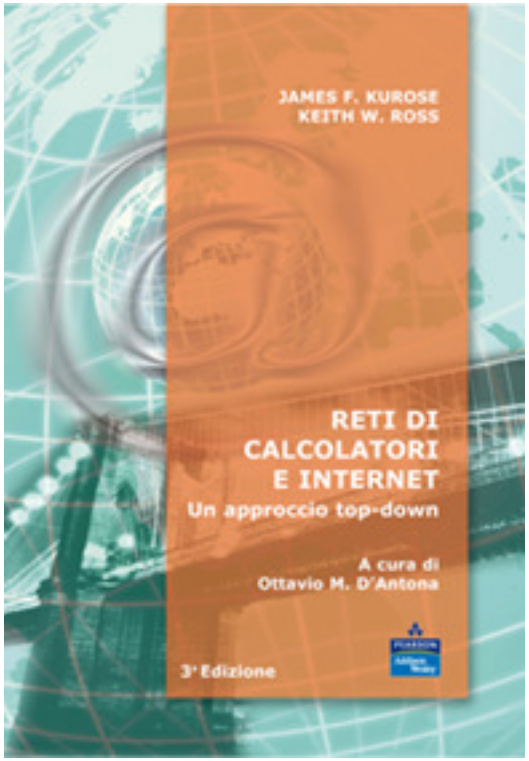
Capitolo 3: Livello di trasporto

Nota per l'utilizzo:

Abbiamo preparato queste slide con l'intenzione di renderle disponibili a tutti (professori, studenti, lettori). Sono in formato PowerPoint in modo che voi possiate aggiungere e cancellare slide (compresa questa) o modificarne il contenuto in base alle vostre esigenze.
Come potete facilmente immaginare, da parte nostra abbiamo fatto *un sacco* di lavoro. In cambio, vi chiediamo solo di rispettare le seguenti condizioni:

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

Thanks and enjoy! JFK/KWR



Reti di calcolatori e Internet: Un approccio top-down

3ª edizione
Jim Kurose, Keith Ross
Pearson Education Italia ©2005

Capitolo 3: Livello di trasporto

Obiettivi:

- **Capire i principi che sono alla base dei servizi del livello di trasporto:**
  - multiplexing/demultiplexing
  - trasferimento dati affidabile
  - controllo di flusso
  - controllo di congestione

- **Descrivere i protocolli del livello di trasporto di Internet:**
  - UDP: trasporto senza connessione
  - TCP: trasporto orientato alla connessione

Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

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

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

Servizi e protocolli di trasporto

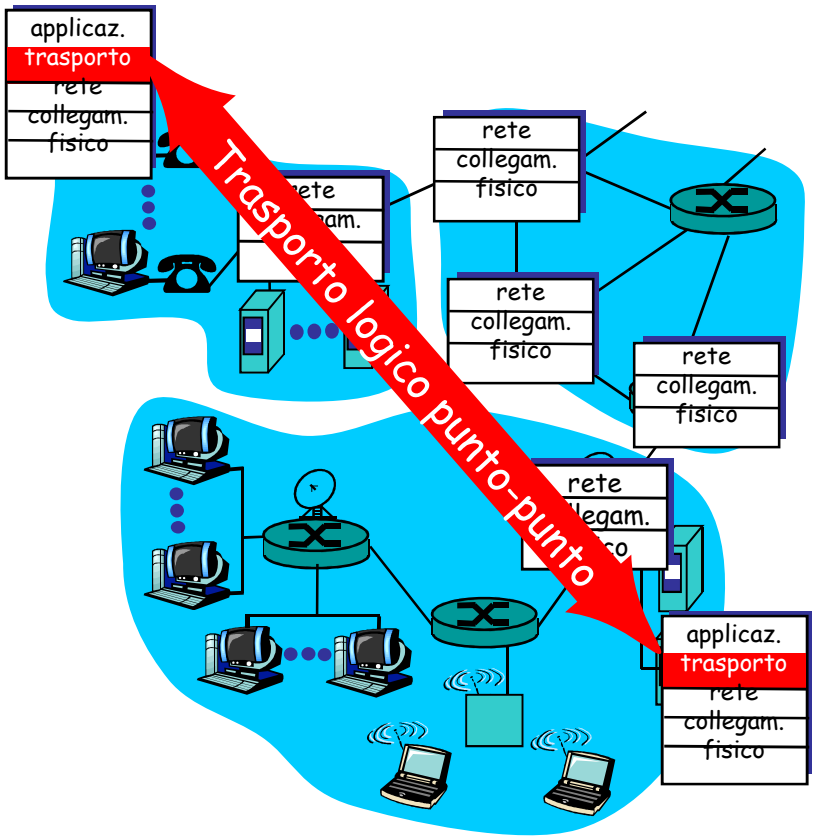
Forniscono la *comunicazione logica* tra processi applicativi di host differenti

I protocolli di trasporto vengono eseguiti nei sistemi terminali

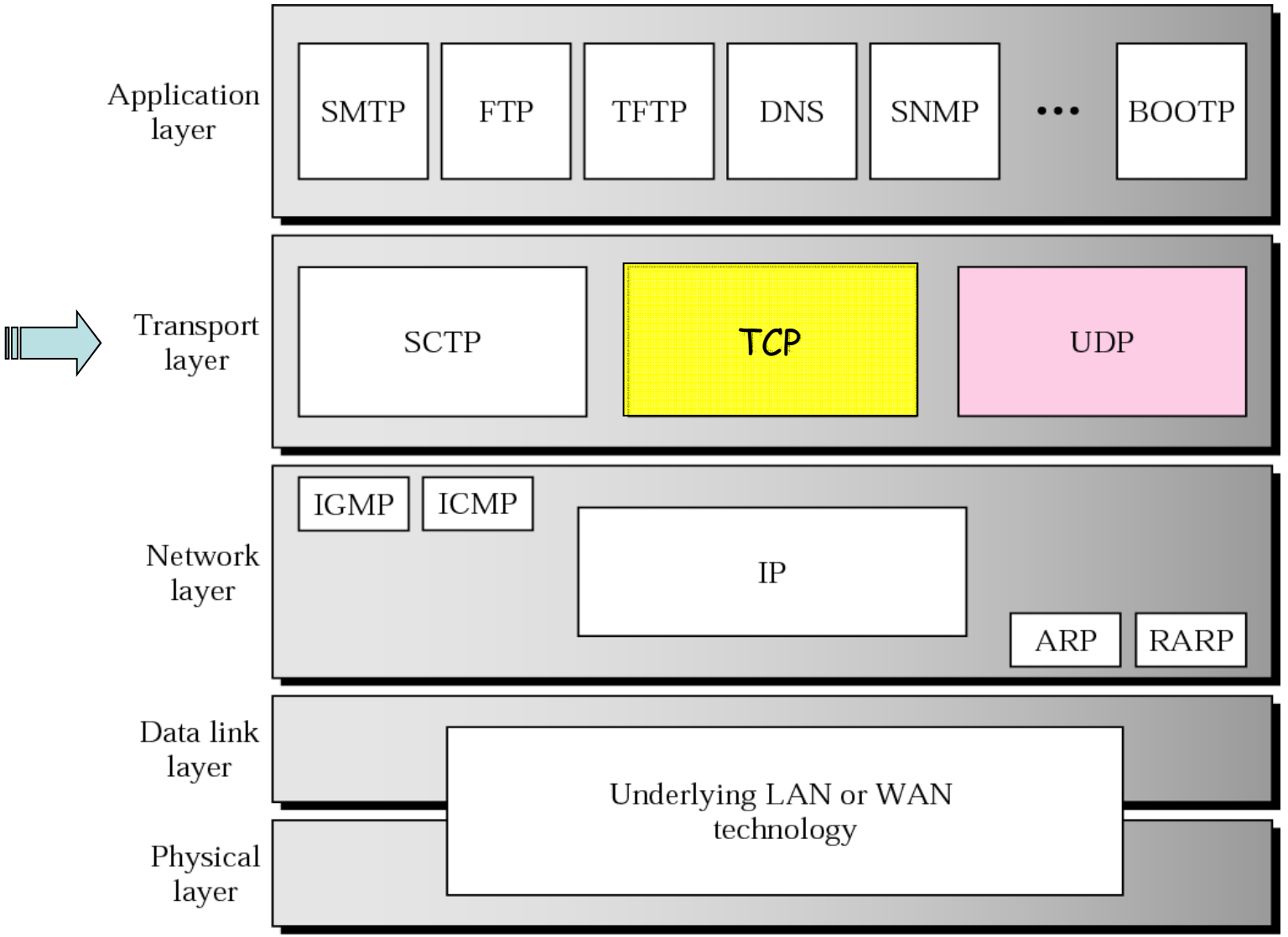
- lato invio: scinde i **messaggi** in **segmenti** e li passa al livello di rete
- lato ricezione: riassembla i segmenti in messaggi e li passa al livello di applicazione

Più protocolli di trasporto sono a disposizione delle applicazioni

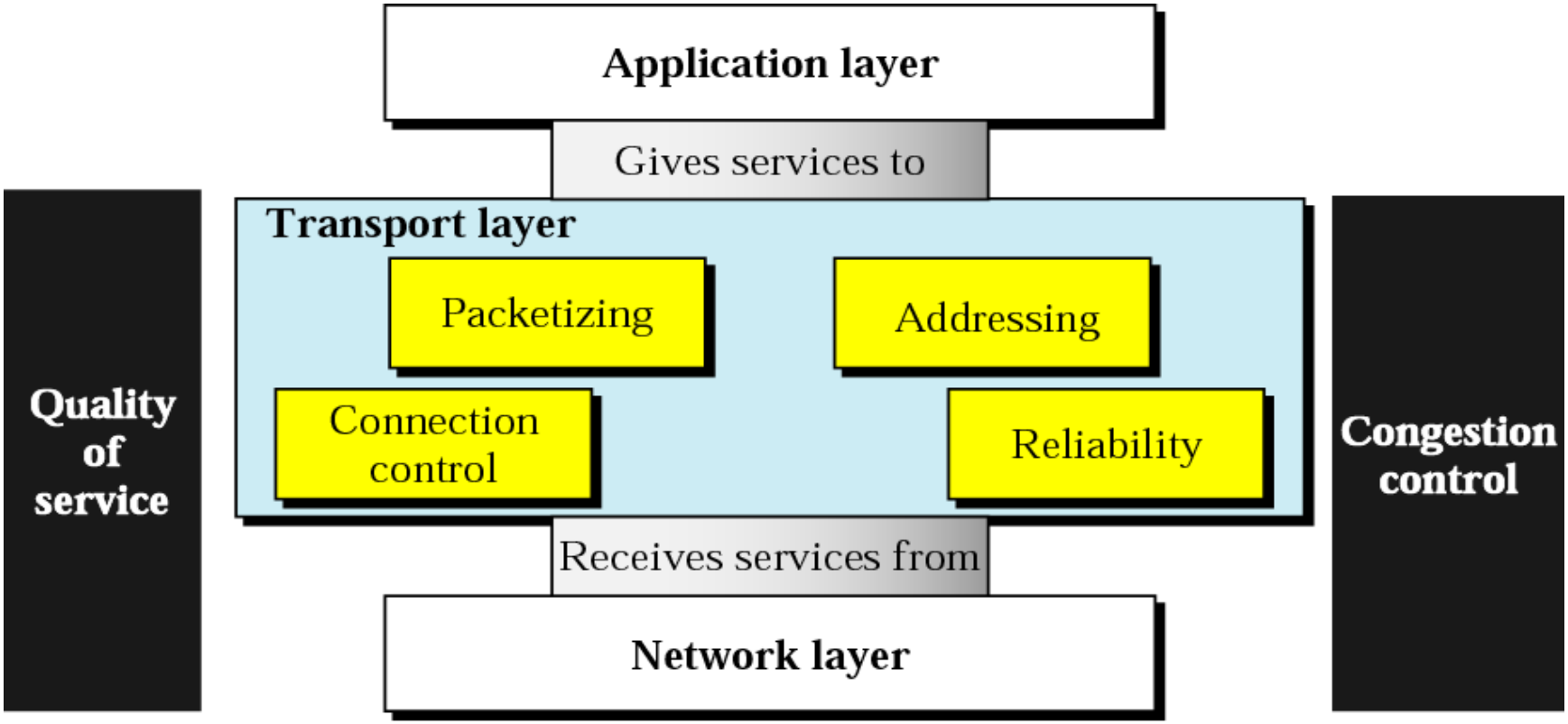
- Internet: TCP e UDP

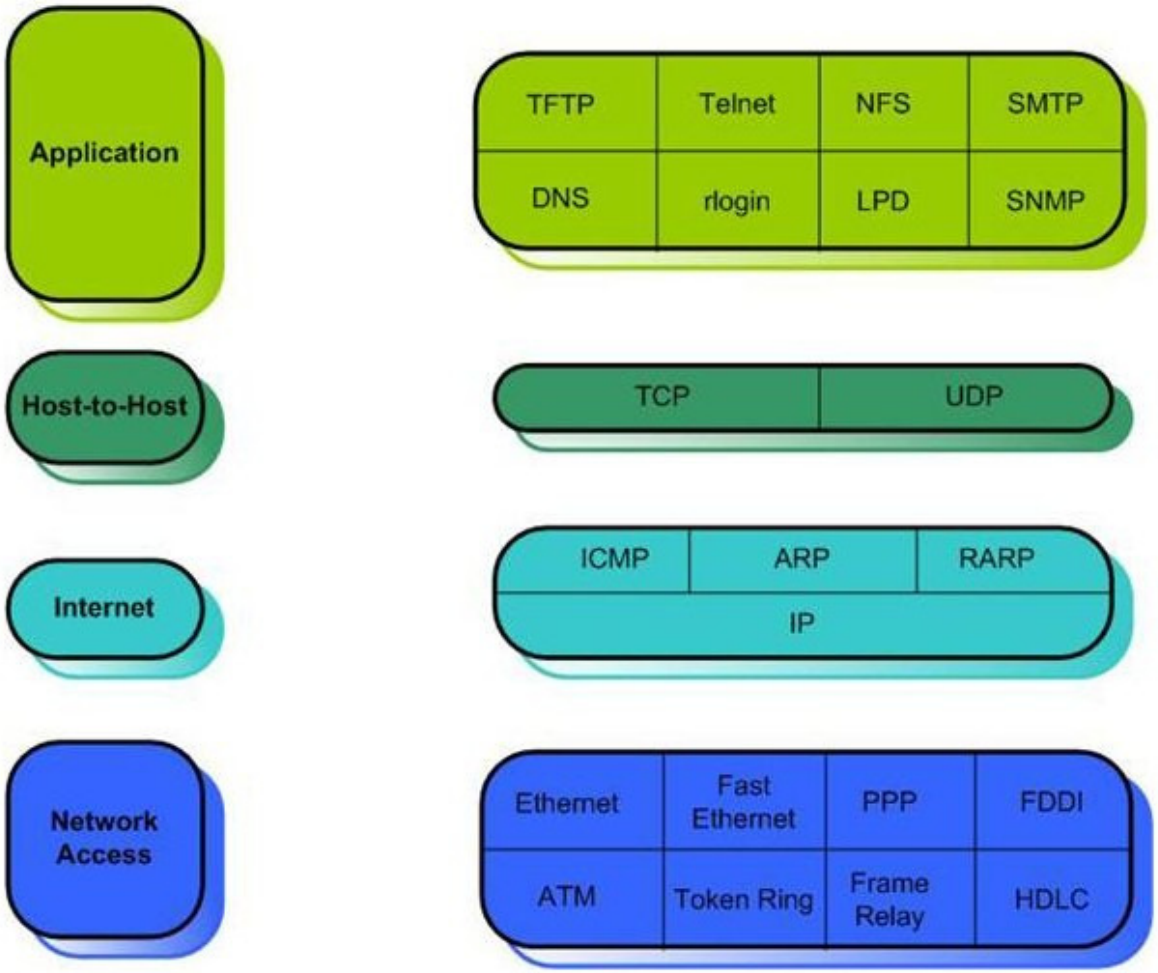


Posizione del trasporto nella suite TCP/IP



Posizione del Transport Layer




Application
TFTP
Telnet
NFS
SMTP
DNS
rlogin
LPD
SNMP
Host-to-Host
TCP
UDP
Internet
ICMP
ARP
RARP
IP
Network Access
Ethernet
Fast Ethernet
PPP
FDDI
ATM
Token Ring
Frame Relay
HDLC

Relazione tra livello di trasporto e livello di rete

livello di rete:
comunicazione logica tra host

livello di trasporto:
comunicazione logica tra processi che girano su host diversi

- si basa sui servizi del livello di rete
- i protocolli di questo livello vivono nei sistemi terminali

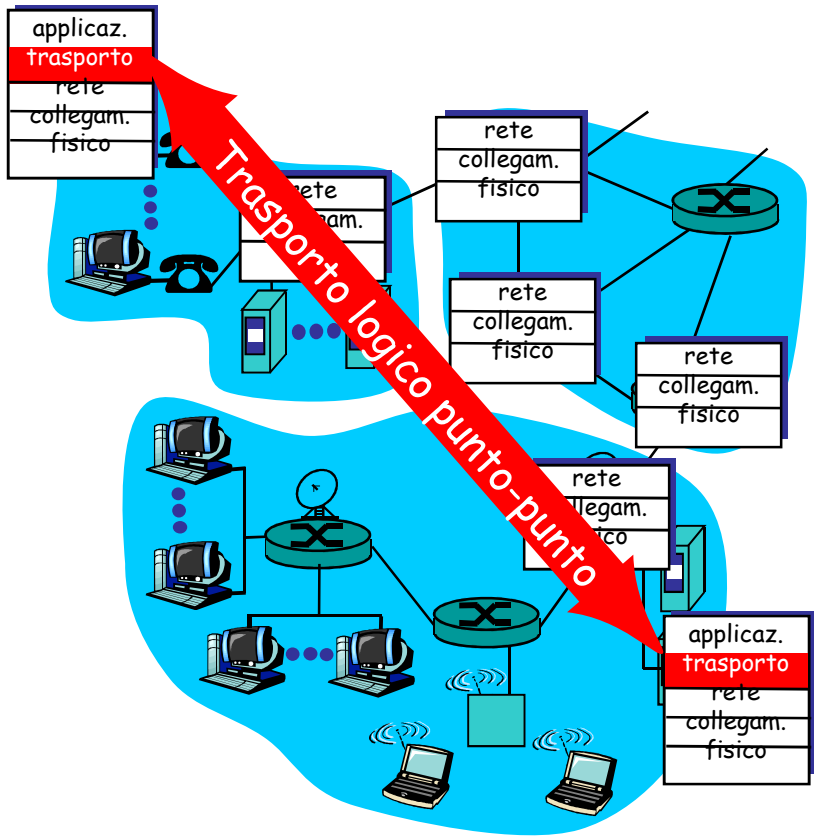
Analogia con la posta ordinaria:

12 cugini di Roma inviano lettere a 12 cugini di Milano

- **processi = ragazzi**
- **messaggi delle applicazioni = lettere nelle buste**
- **host (end systems) = case/condomini**
- **protocollo di trasporto = Anna e Andrea**
- **protocollo del livello di rete = servizio postale**
  - Si noti che Anna e Andrea svolgono tutto il proprio lavoro localmente e non sono coinvolti nello smistamento della posta tra uffici postali
- **se Anna/Andrea sono in vacanza potrebbero sustituirli Lucia e Mattia (che sono piccoli e non sono accurati come i fratelli)**

Protocolli del livello di trasporto in Internet

- **Affidabile, consegne nell'ordine originario (TCP)**
  - controllo di congestione
  - controllo di flusso
  - setup della connessione
- **Inaffidabile, consegne senz'ordine: UDP**
  - estensione senza fronzoli del servizio di consegna a massimo sforzo
- **Servizi non disponibili:**
  - garanzia su ritardi
  - garanzia su ampiezza di banda

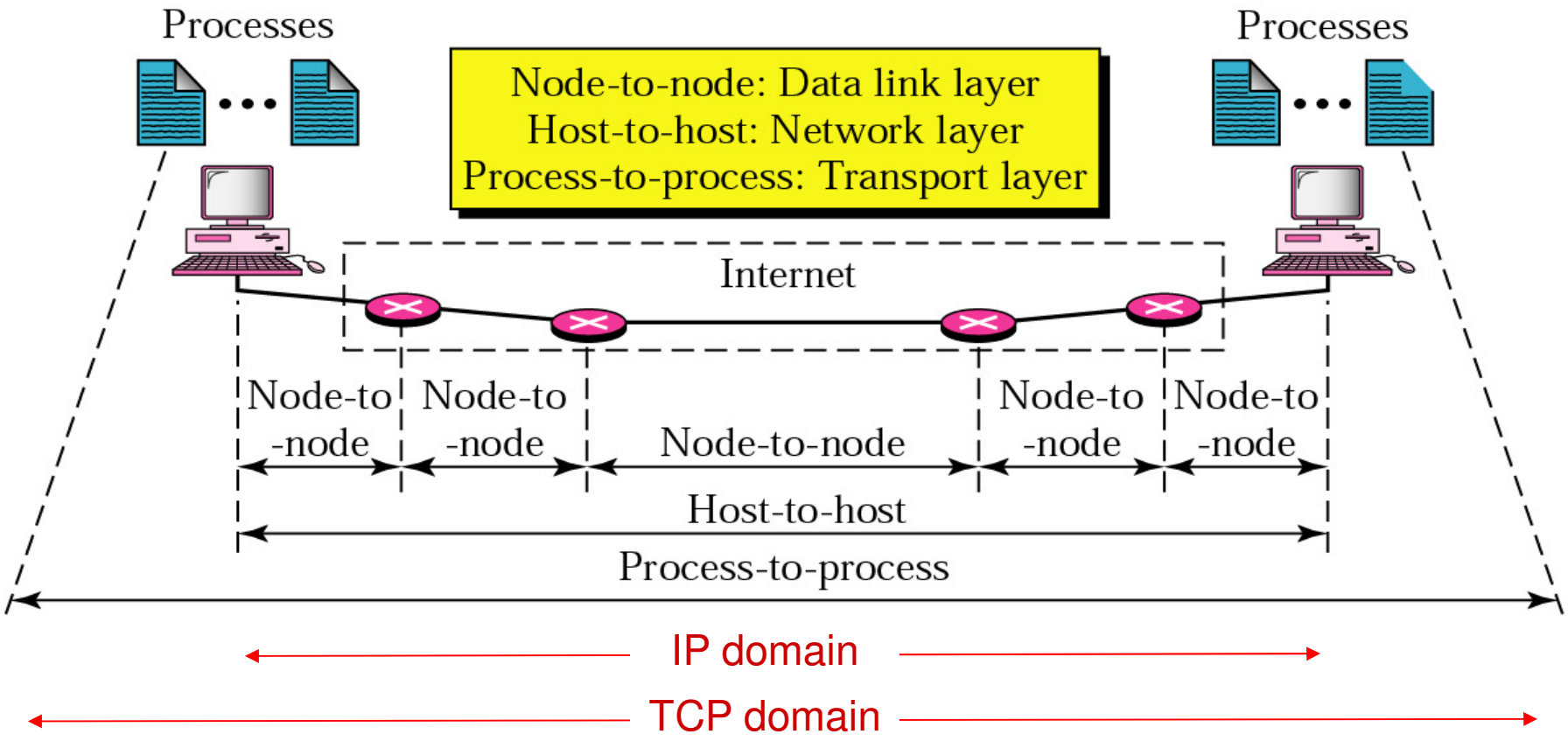


Come abbiamo visto, lo sviluppatore sceglie TCP o UDP quando crea la socket

Il livello sottostante

- **Il livello di rete usa IP che, come modello di servizio, è a "Best-Effort" ovvero** non affidabile
  - Non assicura la consegna
  - Non assicura l'ordine
  - Non assicura l'integrità
- **Il livello trasporto ha il compito di estendere i servizi di consegna di IP "tra terminali" a uno di consegna "tra processi in esecuzione sui sistemi terminali".**
- **Questo passaggio da consegna** host-to-host **a consegna** process-to-process **viene detto** multiplexing **e** demultiplexing **a** livello di trasporto
- **TCP converte il sistema inaffidabile, IP, a livello di rete in un sistema affidabile a livello trasporto**
- **UDP invece estende l'inaffidabilità di IP a livello trasporto**

Types of data deliveries



Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

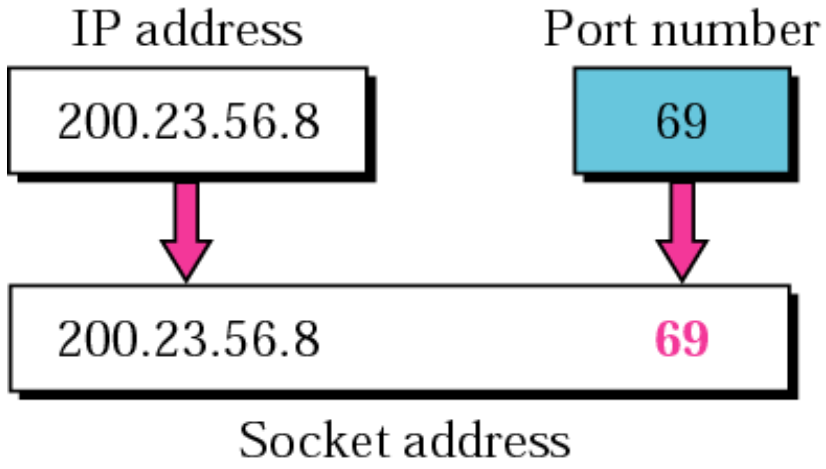
- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 Principi sul controllo di congestione

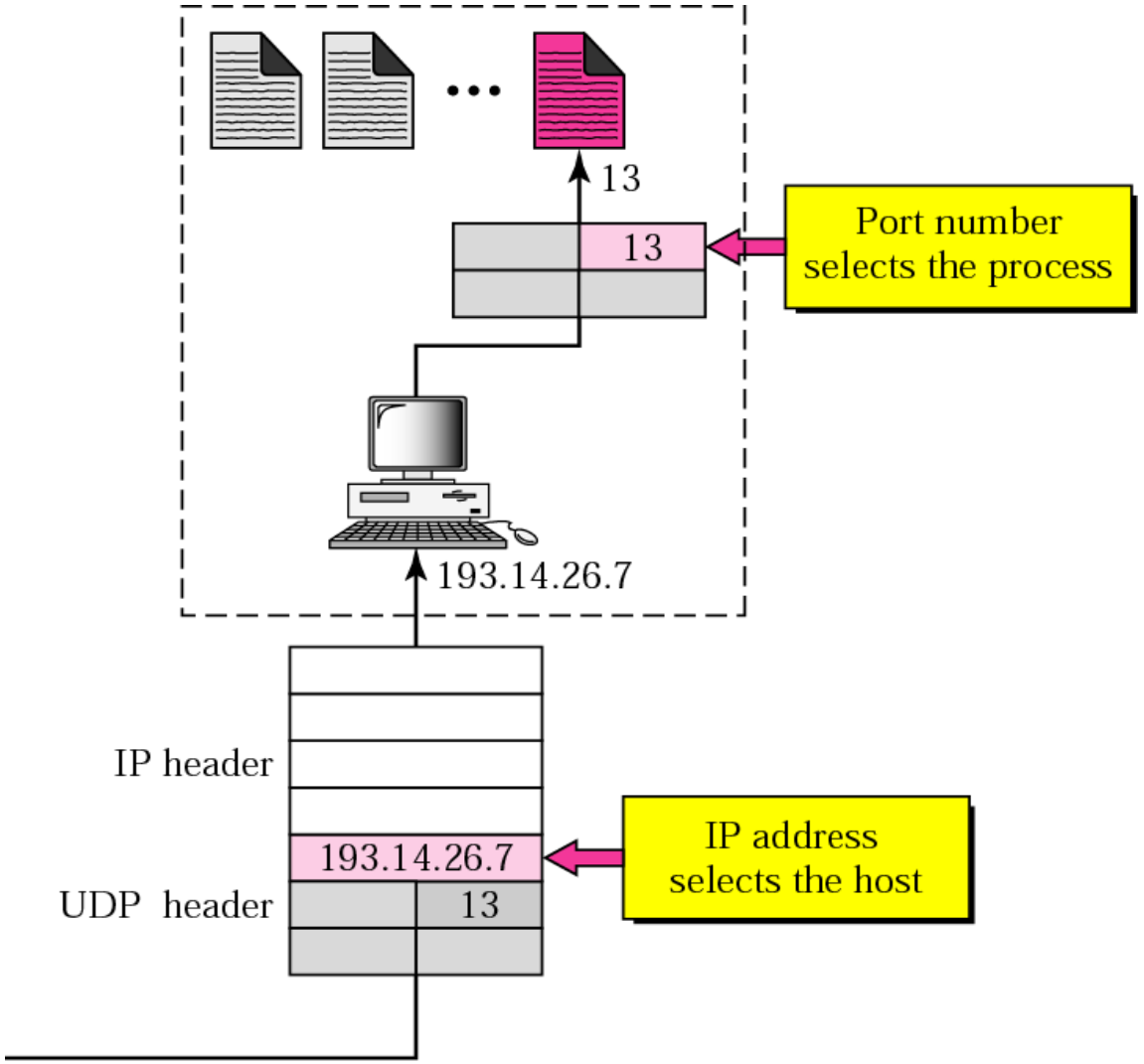
3.7 Controllo di congestione TCP

Sockets

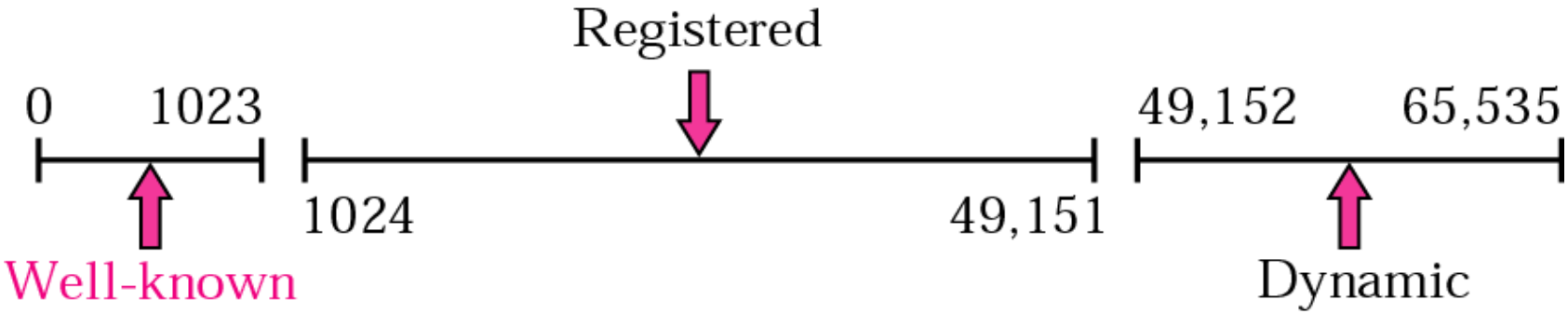
- **Un processo, come parte di una applicazione di rete, può presentare una o più socket**
  - porte attraverso le quali i dati fluiscono dal processo alla rete e viceversa
- **Il livello di trasporto nell'host ricevente non trasferisce i dati direttamente al processo ma ad una socket intermedia**
- **Ogni socket è identificata univocamente dal servizio, dall'indirizzo IP e dal numero di porta**



Indirizzi IP e indirizzi di porta



Numeri di porta

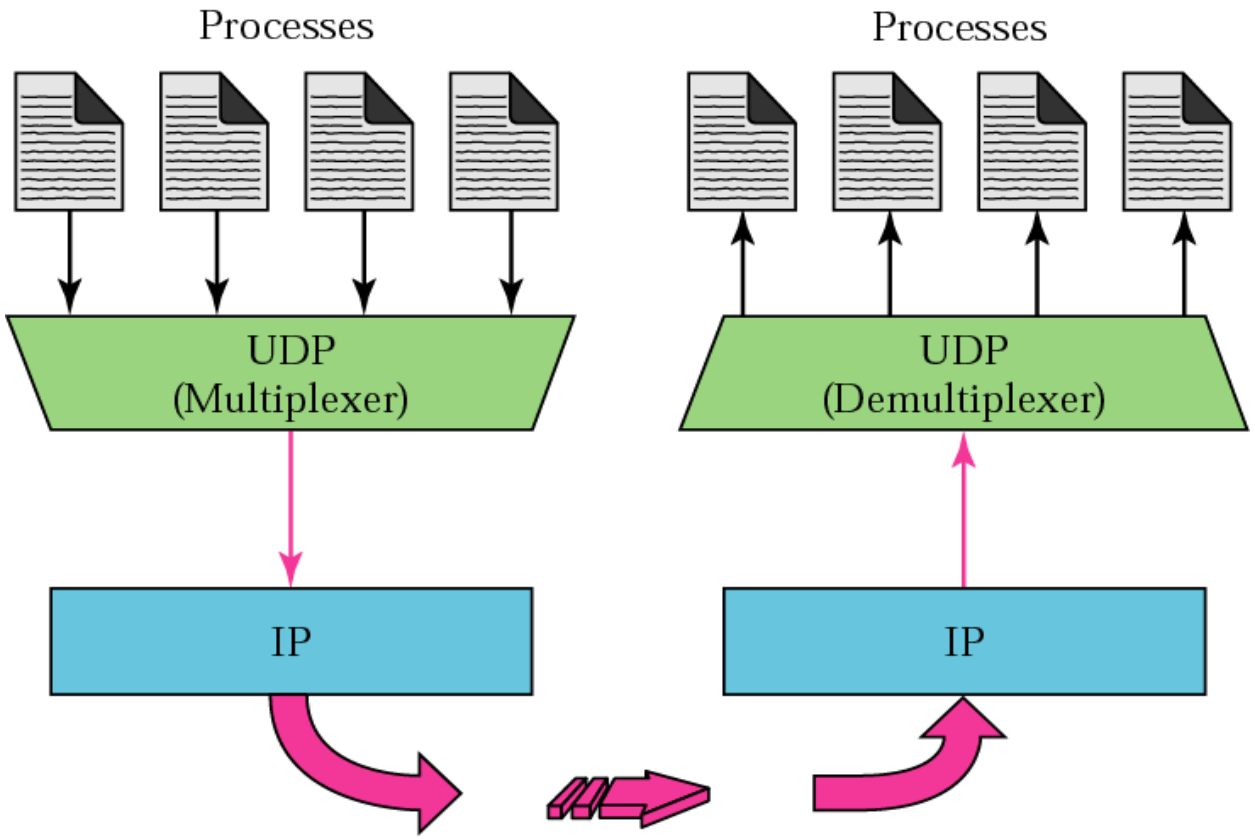


IANA: Internet Assigned Number Authority

- **Da 0 a 1023**
  - Porte “ben note”, gestite da IANA
- **Da 1024 a 49152**
  - Porte che si possono “registrare” con IANA
- **Da 49152 a 65535**
  - Dinamiche (si possono usare liberamente)

Numeri di porta e multiplexing

- **Utilizzo dei numeri di porta**
  - Forma di multiplexing



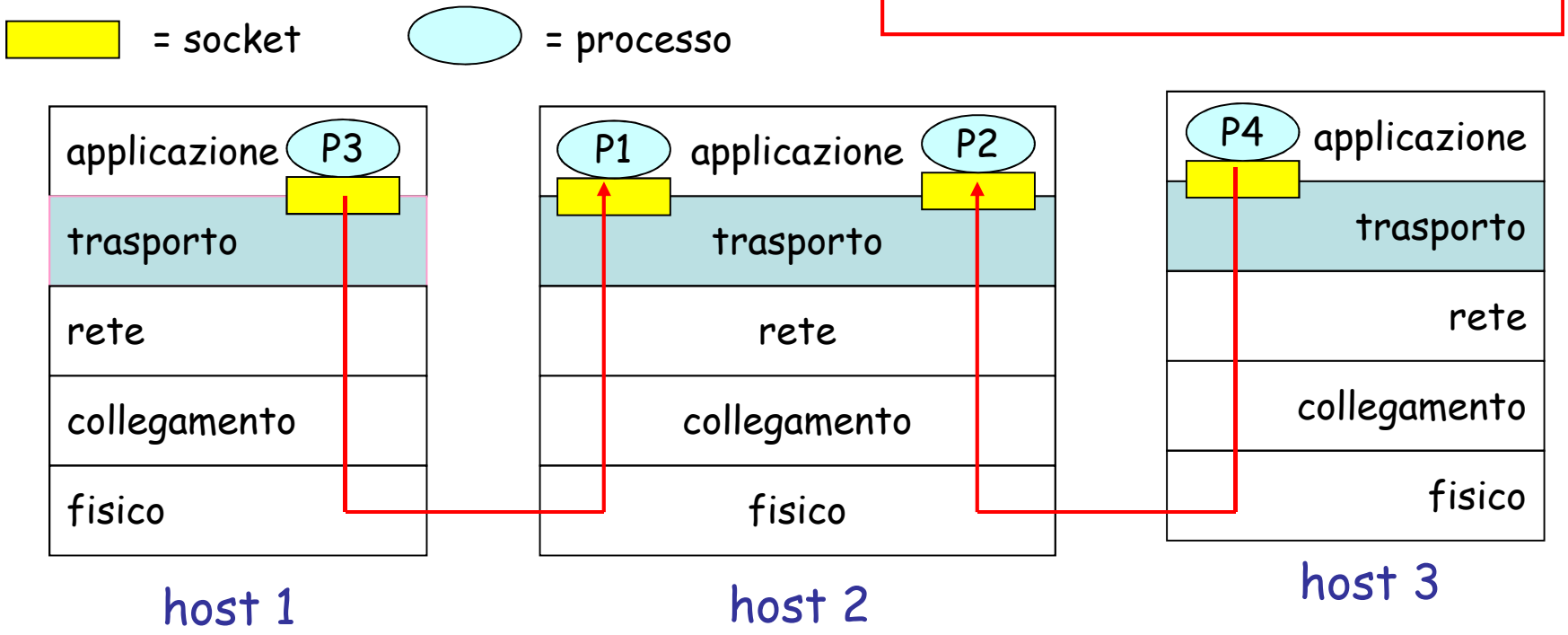
Multiplexing/demultiplexing

Demultiplexing nell'host ricevente:

consegnare i segmenti ricevuti alla socket appropriata

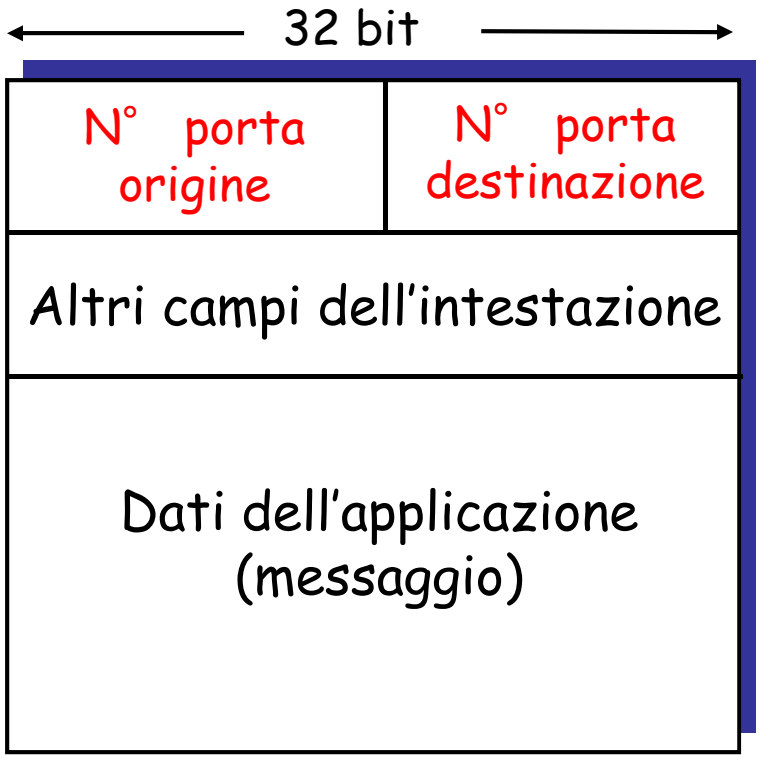
Multiplexing nell'host mittente:

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



Come funziona il demultiplexing

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



Struttura del segmento TCP/UDP

Demultiplexing senza connessione

- **Crea la socket**

```
DatagramSocket mySocket = new
    DatagramSocket();
```

Il livello di trasporto assegna un numero di porta random compreso tra 1024 e 65535 non in uso

- **Crea le socket con i numeri di porta:**

```
DatagramSocket mySocket1 = new
    DatagramSocket(11111);
DatagramSocket mySocket2 = new
    DatagramSocket(22222);
```

- **La socket UDP è identificata da 2 parametri:**

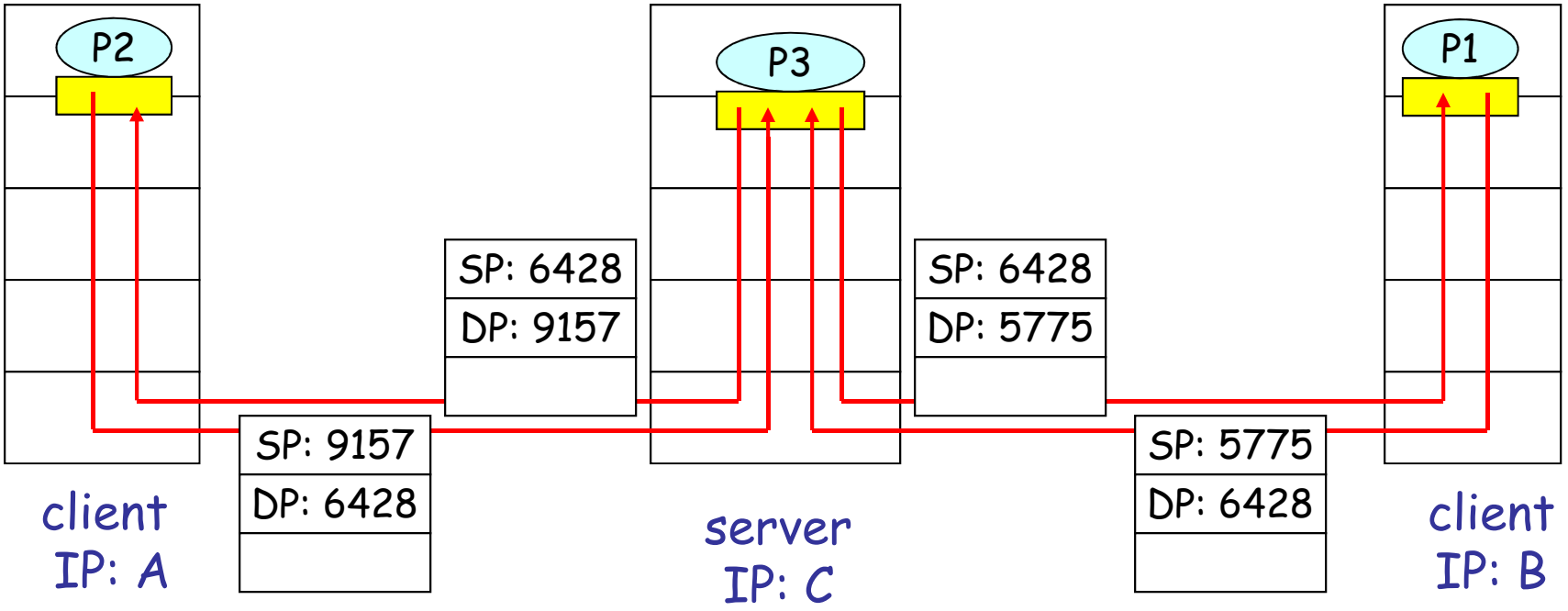
(indirizzo IP di destinazione, numero della porta di destinazione)

- **Quando l'host riceve il segmento UDP:**
  - controlla il numero della porta di destinazione nel segmento
  - invia il segmento UDP alla socket con quel numero di porta

- **I datagrammi IP con indirizzi IP di origine e/o numeri di porta di origine differenti vengono inviati alla stessa socket**

Demultiplexing senza connessione (continua)

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



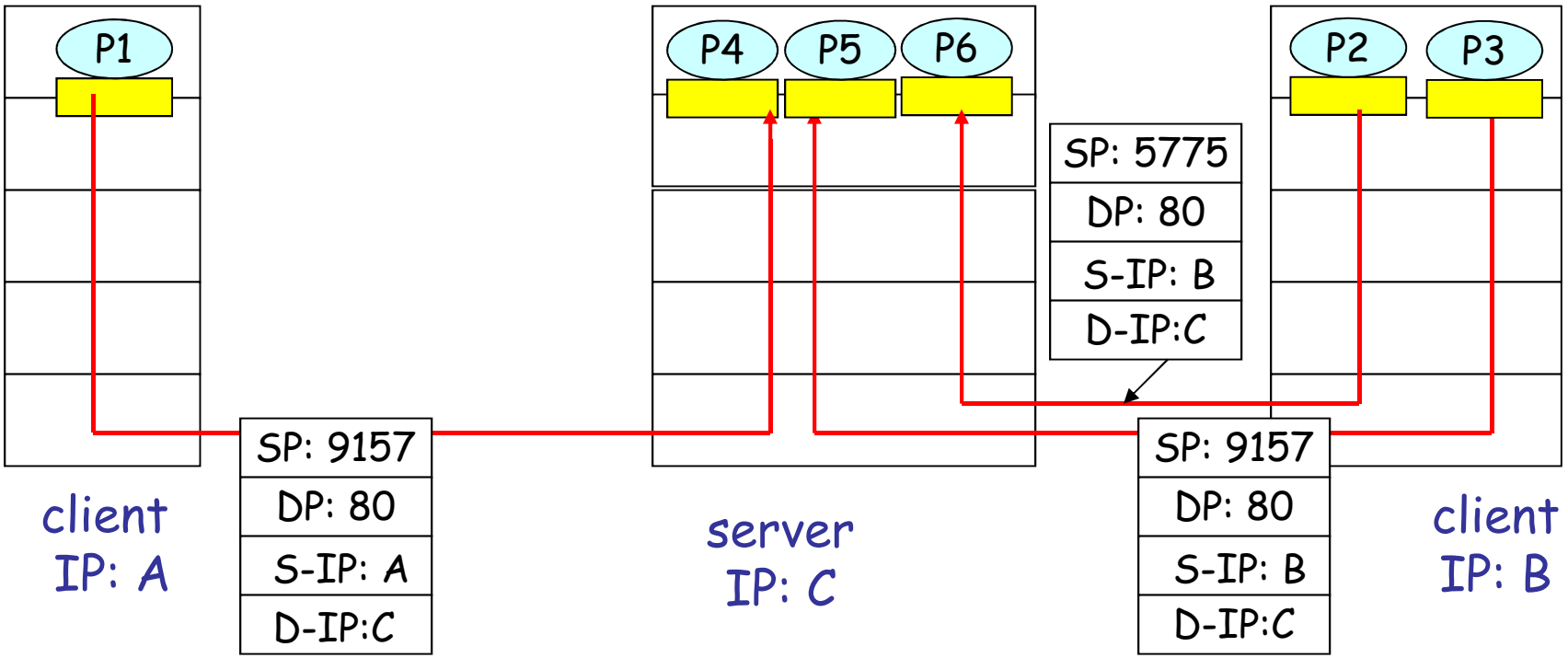
SP fornisce "l'indirizzo di ritorno"

Demultiplexing orientato alla connessione

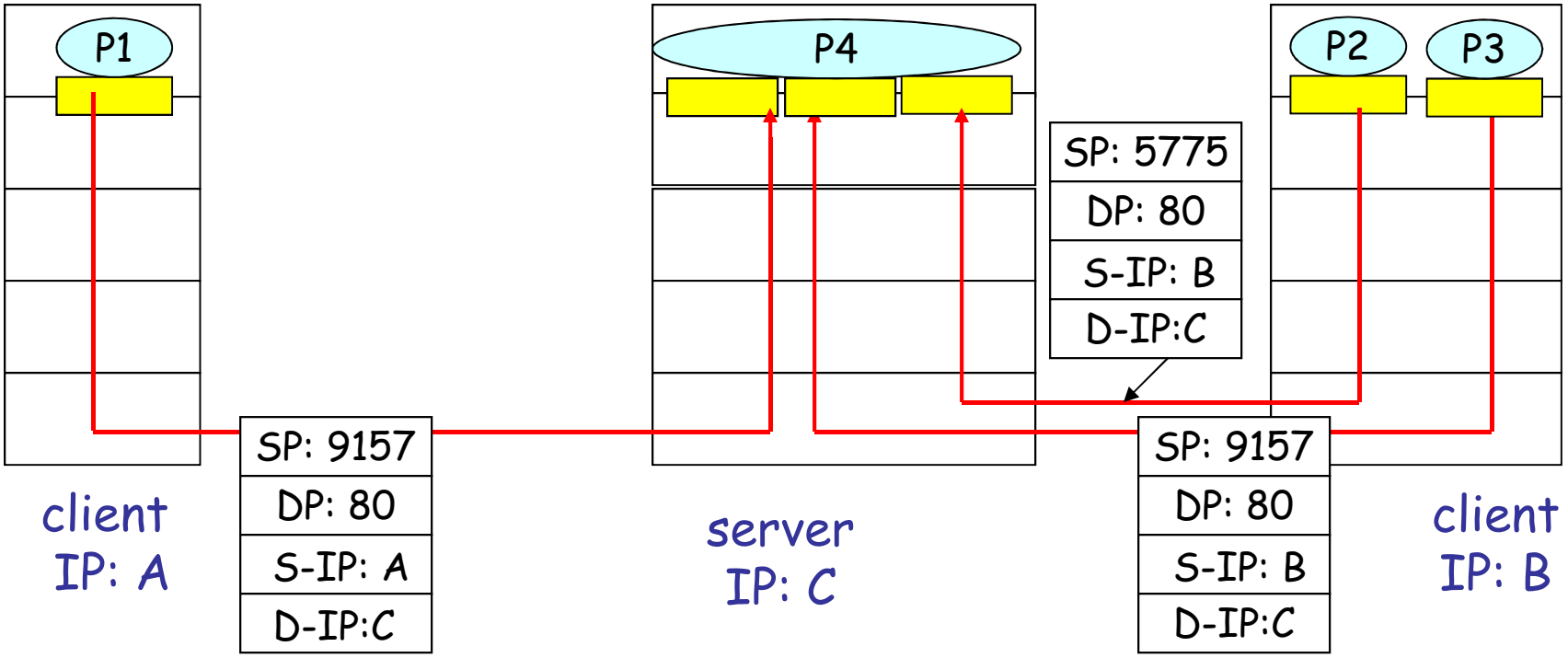
- **La socket TCP è identificata da 4 parametri:**
  - indirizzo IP di origine
  - numero di porta di origine
  - indirizzo IP di destinazione
  - numero di porta di destinazione
- **L'host ricevente usa i quattro parametri per inviare il segmento alla socket appropriata**
- **Un host server può supportare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

```
Socket ClientSocket = new Socket("serverHostName",80);
```

Demultiplexing orientato alla connessione (continua)



Demultiplexing orientato alla connessione: thread dei server web



Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

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

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

UDP: User Datagram Protocol [RFC 768]

- **Protocollo di trasporto "senza fronzoli"**
- **Servizio di consegna "best effort", i segmenti UDP possono essere:**
  - perduti
  - consegnati fuori sequenza all'applicazione

Senza connessione:

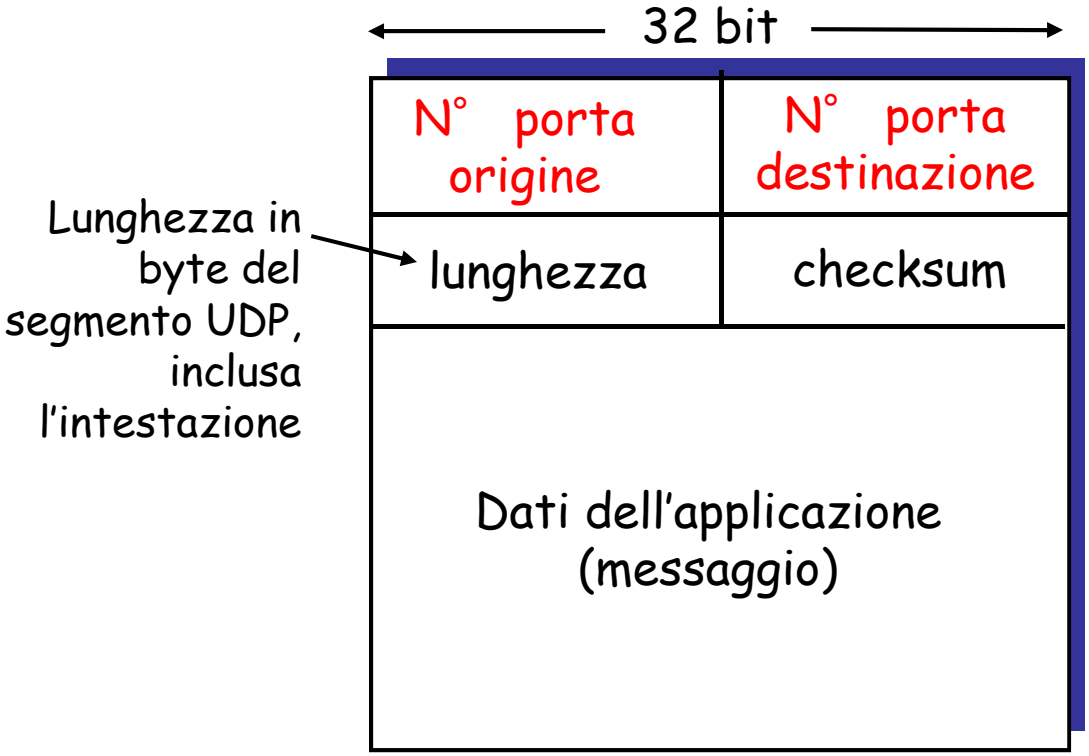
  - no handshaking tra mittente e destinatario UDP
  - ogni segmento UDP è gestito indipendentemente dagli altri

Perché esiste UDP?

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

UDP: altro

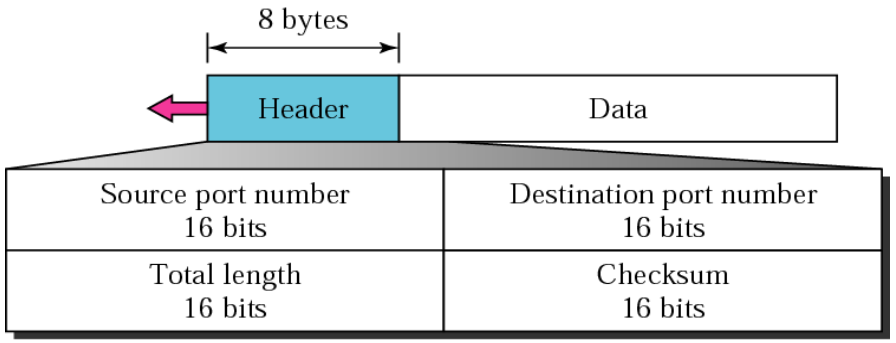
- **Utilizzato spesso dalle applicazioni multimediali**
  - tollera piccole perdite
  - sensibile alla frequenza
- **Altri impieghi di UDP**
  - DNS
  - SNMP
- **Trasferimento affidabile con UDP: aggiungere affidabilità al livello di applicazione**
  - Recupero degli errori a carico delle applicazioni!



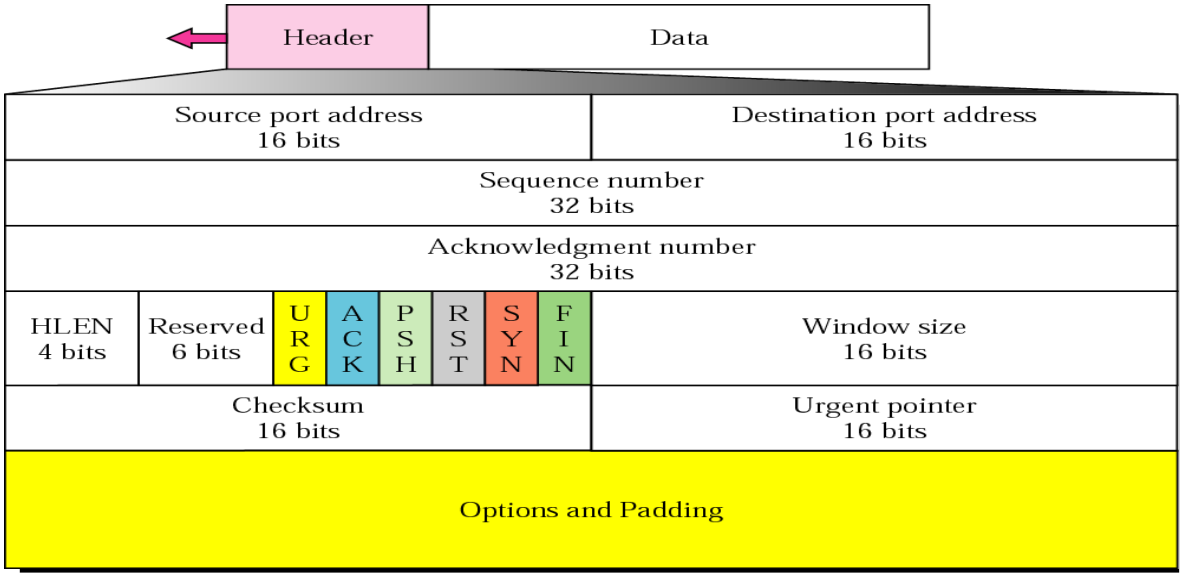
Struttura del segmento UDP

UDP vs TCP

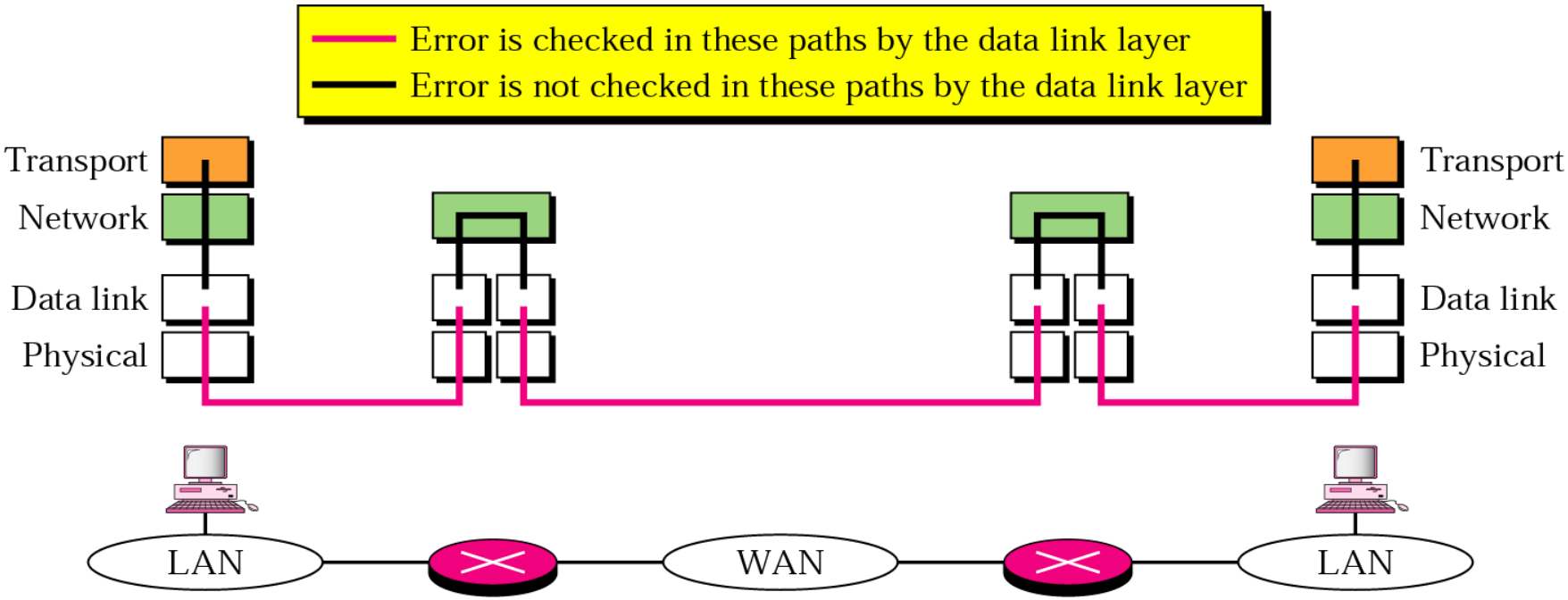
UDP header
8 bytes



TCP header
20 bytes



Error control



Checksum UDP

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

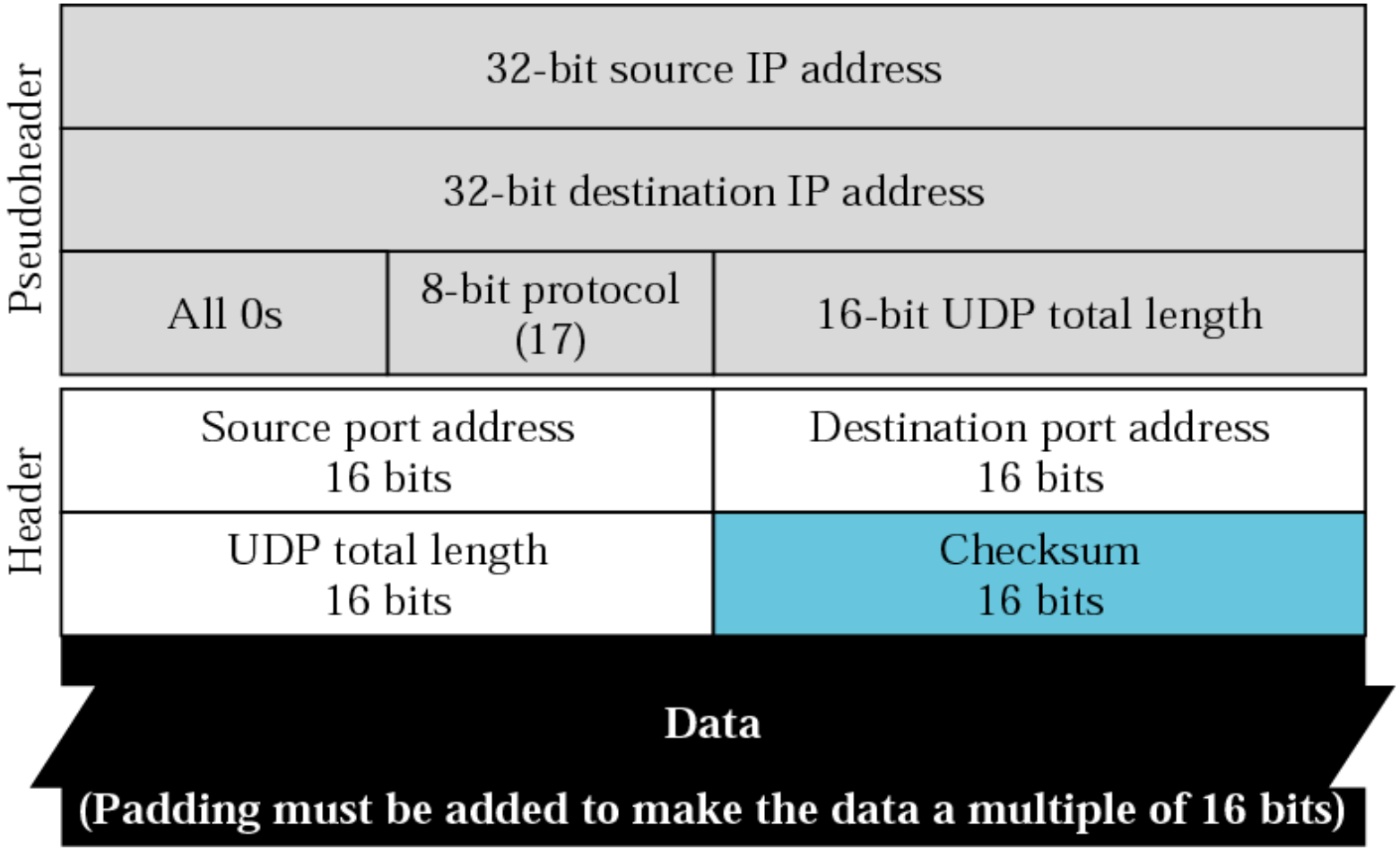
Mittente:

- Tratta il contenuto del segmento come una sequenza di interi da 16 bit
- checksum: somma (complemento a 1) i contenuti del segmento
- Il mittente pone il valore della checksum nel campo checksum del segmento UDP

Ricevente:

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

Checksum pseudoheader



Esempio di calcolo checksum

To calculate UDP checksum a "pseudo header" is added to the UDP header. This includes:

1. IP Source Address 4 bytes
2. IP Destination Address 4 bytes
3. Protocol 2 bytes
4. UDP Length 2 bytes

The checksum is calculated over all the octets of the pseudo header, UDP header and data. If the data contains an odd number of octets a pad, zero octet is added to the end of data. The pseudo header and the pad are not transmitted with the packet.

153.18.8.105			
171.2.14.10			
All 0s	17	15	
1087		13	
15		All 0s	
T	E	S	T
I	N	G	**All 0s**

Padding

10011001	00010010	153.18
00001000	01101001	8.105
10101011	00000010	171.2
00001110	00001010	14.10
00000000	00010001	0 and 17
00000000	00001111	15
00000100	00111111	1087
00000000	00001101	13
00000000	00001111	15
00000000	00000000	0 (checksum)
01010100	01000101	T and E
01010011	01010100	S and T
01001001	01001110	I and N
01000111	00000000	G and 0 (padding)
10010110	**11101011**	Sum
01101001	**00010100**	Checksum

Esempio di checksum

- **Nota**
  - Quando si sommano i numeri, un riporto dal bit più significativo deve essere sommato al risultato
- **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

Perchè UDP calcola la checksum ?

- **Anzitutto UDP non se ne fa nulla della informazione di Checksum**
  - Alcune implementazioni scartano il segmento errato
  - Altre si limitano a segnalare la cosa al livello applicativo
- **Anche se i livelli sottostanti controllano gli errori, si potrebbero verificare errori all'interno dei router**
- **Principio punto-punto secondo il quale determinate funzionalità devono essere implementate su base punto-punto**

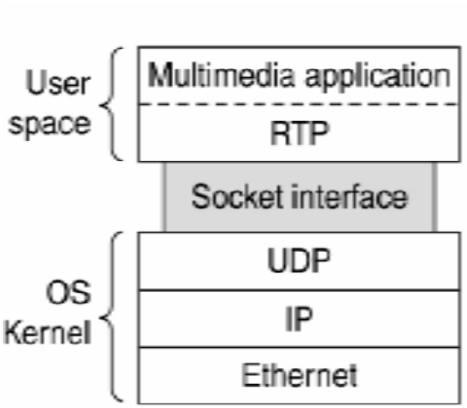
```
/*
*************************************************************************
Function: udp_sum_calc()
Description: Calculate UDP checksum
*************************************************************************
*/
typedef unsigned short u16;
typedef unsigned long u32;

u16 udp_sum_calc(u16 len_udp, u16 src_addr[],u16 dest_addr[], BOOL padding, u16 buff[])
{
u16 prot_udp=17;
u16 padd=0;
u16 word16;
u32 sum;

        // Find out if the length of data is even or odd number. If odd,
        // add a padding byte = 0 at the end of packet
        if (padding&1==1){
                padd=1;
                buff[len_udp]=0;
        }

        //initialize sum to zero
        sum=0;

        // make 16 bit words out of every two adjacent 8 bit words and
        // calculate the sum of all 16 vit words
        for (i=0;i<len_udp+padd;i=i+2){
                word16 =((buff[i]<<8)&0xFF00)+(buff[i+1]&0xFF);
                sum = sum + (unsigned long)word16;
        }
        // add the UDP pseudo header which contains the IP source and destinationn addresses
        for (i=0;i<4;i=i+2){
                word16 =((src_addr[i]<<8)&0xFF00)+(src_addr[i+1]&0xFF);
                sum=sum+word16;
        }
        for (i=0;i<4;i=i+2){
                word16 =((dest_addr[i]<<8)&0xFF00)+(dest_addr[i+1]&0xFF);
                sum=sum+word16;
        }
        // the protocol number and the length of the UDP packet
        sum = sum + prot_udp + len_udp;

        // keep only the last 16 bits of the 32 bit calculated sum and add the carries
        while (sum>>16)
                sum = (sum & 0xFFFF)+(sum >> 16);

        // Take the one's complement of sum
        sum = ~sum;

return ((u16) sum);
}
```

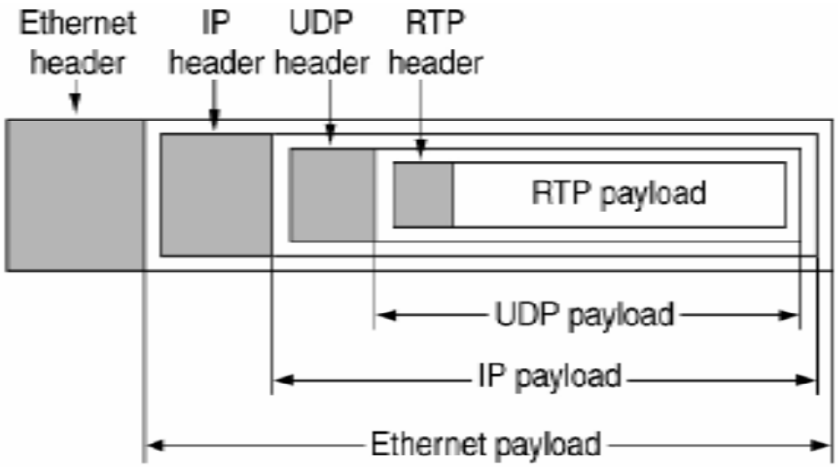
Protocolli/servizi che utilizzano UDP

Protocollo/Servizio	Porta	Descrizione
Echo	7	Risponde con gli stessi dati arrivati
Elimina	9	Elimina i datagram
Utenti	11	Fornisce il numero di utenti collegati
Daytime	13	Fornisce la data e l'ora
Quote	17	Fornisce la massima del giorno
Chargen	19	Risponde con una stringa di caratteri
DNS	53	Server DNS
BOOTP	67	Server BOOTP
BOOTP	68	Client BOOTP
TFTP	69	Trivial File Transfer Protocol
RPC	111	Server Remote Procedure Call
NTP	123	Network Time Protocol
SNMP	161	Server Simple Mail Network Protocol
SNMP	162	Porta di servizio SNMP

UDP: conclusione

- Per applicazioni che non necessitano di un controllo preciso sul flusso dei pacchetti, controllo di errore, di congestione e/o il timing, UDP rappresenta la scelta di elezione
- Con UDP tutto ciò che deve essere fatto è compito dell'applicazione
- Tipicamente tra applicazioni multimediali e UDP si ha RTP





Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

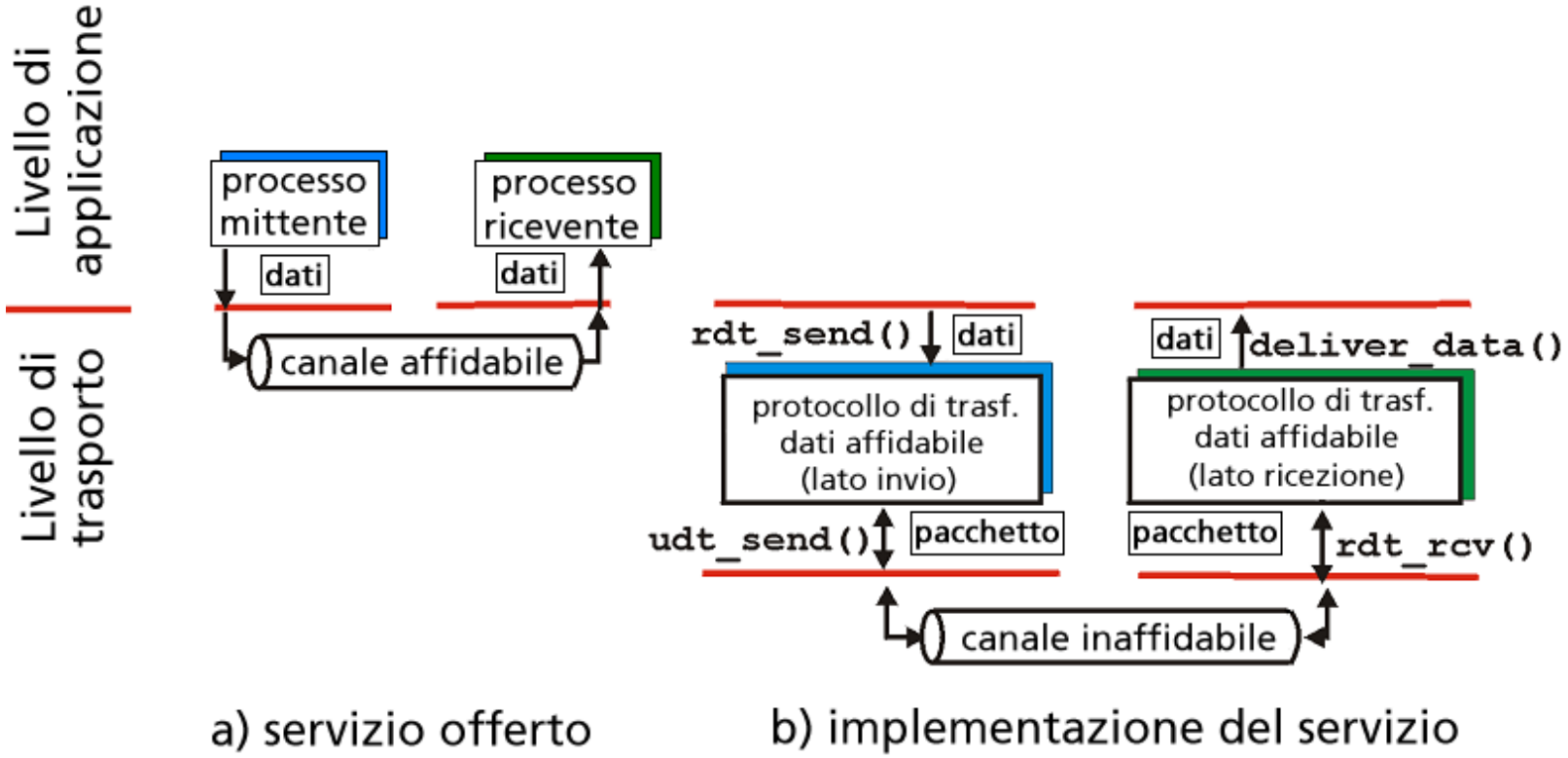
- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

Principi del trasferimento dati affidabile

- **Importante nei livelli di applicazione, trasporto e collegamento**
- **Tra i dieci problemi più importanti del networking!**

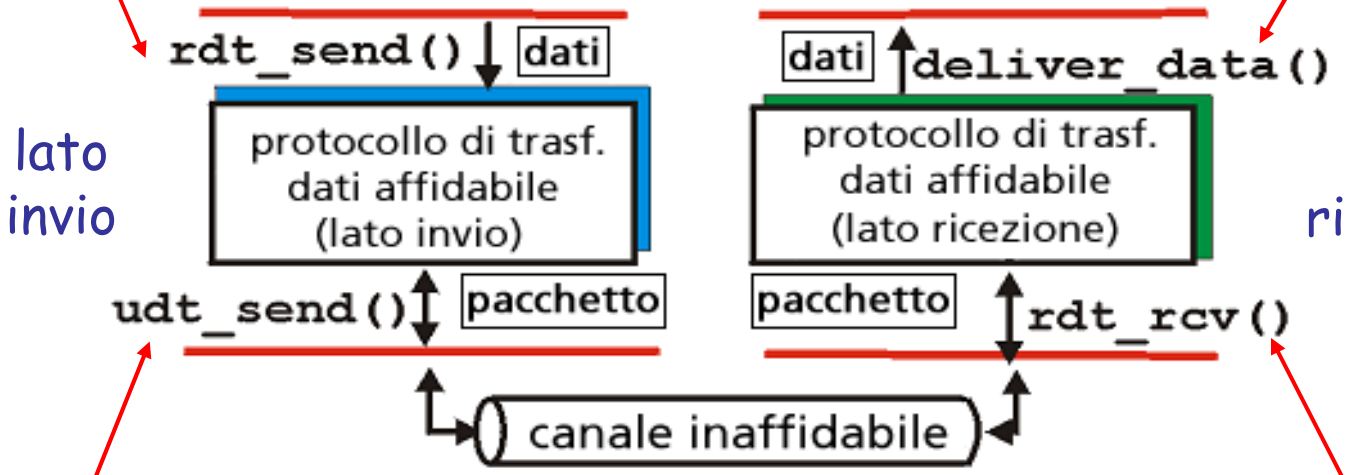


- **Le caratteristiche del canale inaffidabile determinano la complessità del protocollo di trasferimento dati affidabile (reliable data transfer o rdt)**

Trasferimento dati affidabile: preparazione

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

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



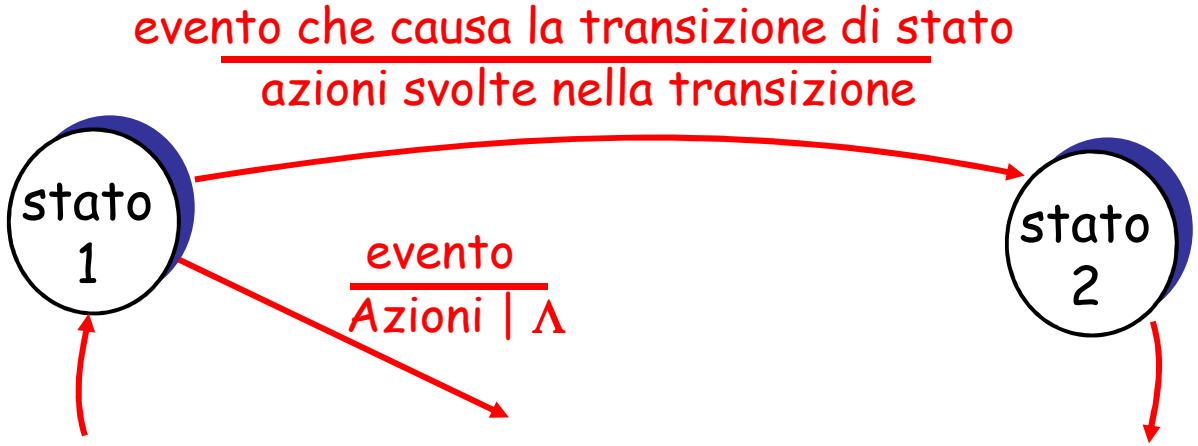
`udt_send()`: chiamata da rdt per trasferire il pacchetto al ricevente tramite il canale inaffidabile

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

Trasferimento dati affidabile: preparazione

- **Svilupperemo progressivamente i lati d'invio e di ricezione di un protocollo di trasferimento dati affidabile (rdt)**
- **Considereremo soltanto i trasferimenti dati unidirezionali**
  - ma le informazioni di controllo fluiranno in entrambe le direzioni!
- **Utilizzeremo automi a stati finiti per specificare il mittente e il ricevente**

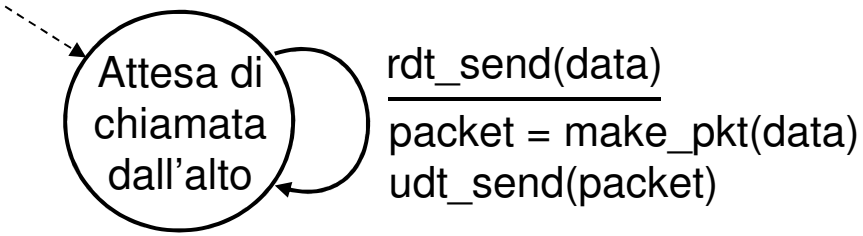
stato: lo stato successivo a questo è determinato unicamente dall'evento successivo



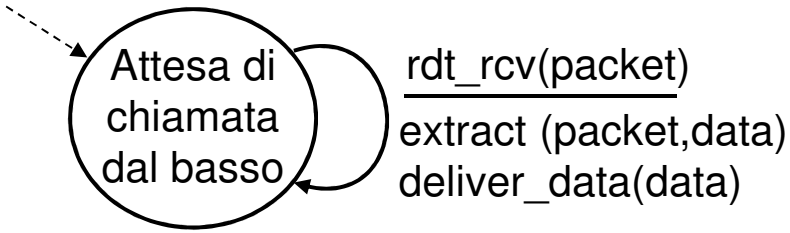
Λ significa "assenza di una azione o di un evento"

Rdt1.0: trasferimento affidabile su canale affidabile

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



mittente



ricevente

La freccia tratteggiata individua lo stato iniziale
In questo canale nulla può andare storto e mittente e ricevente operano alla stessa velocità per cui non c'è esigenza di controllo di flusso

Rdt2.0: canale con errori nei bit

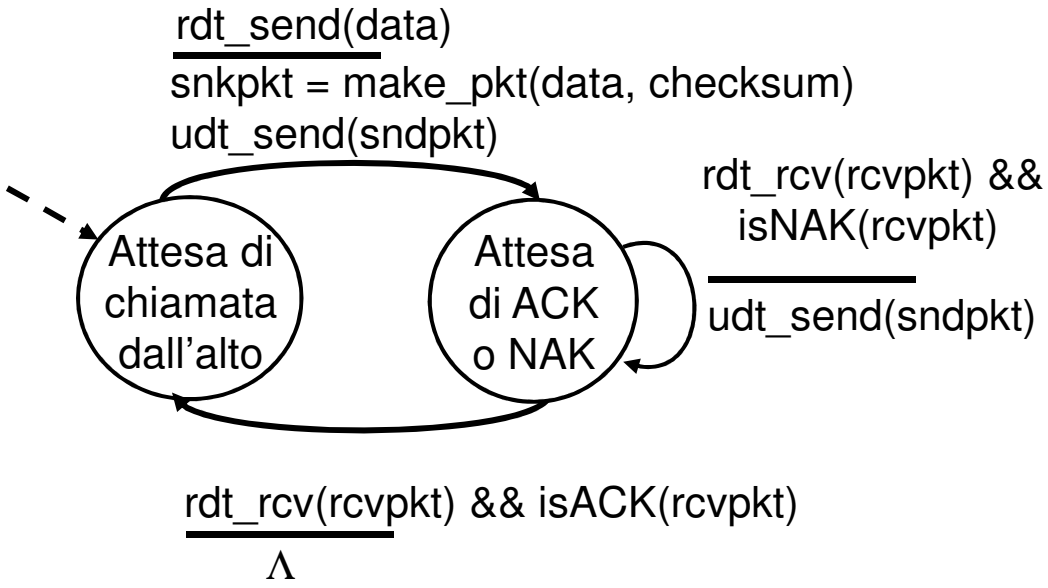
- **Il canale sottostante potrebbe confondere i bit nei pacchetti (l'ordine è ancora garantito)**
  - Metodo per rilevare gli errori nei bit (checksum)
- ***Domanda*: come correggere gli errori ?**
  - *Notifica positiva (ACK):* il ricevente comunica espressamente al mittente che il pacchetto ricevuto è corretto
  - *Notifica negativa (NAK):* il ricevente comunica espressamente al mittente che il pacchetto contiene errori
  - Il mittente ritrasmette il pacchetto se riceve un NAK
- **Questi protocolli necessitano di nuovi meccanismi in `rdt2.0` (oltre a quelli di `rdt1.0`):**
  - Rilevamento di errore
  - Feedback del destinatario: messaggi di controllo (ACK, NAK) ricevente → mittente
  - Ritrasmissioni

- **Analogia della dettatura al telefono**

Protocolli ARQ (Automatic Repeat reQuest)

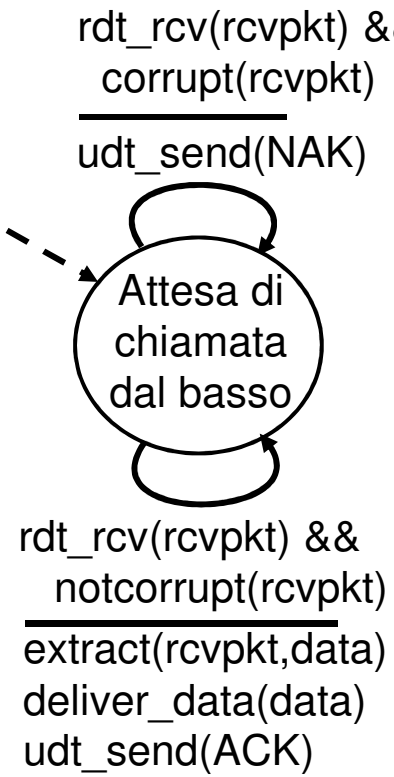
- **Nel networking i protocolli basati su ritrasmissioni si chiamano protocolli ARQ.**
- **Essi devono avere**
  - Rilevamento dell'errore
    - Le tecniche di rilevamento richiedono la trasmissione di bits extra
  - Feedback del destinatario
    - ACK
    - NACK
  - Ritrasmissione
    - Ogni pacchetto ricevuto con errori sarà ritrasmesso

Rdt2.0: specifica dell'automa

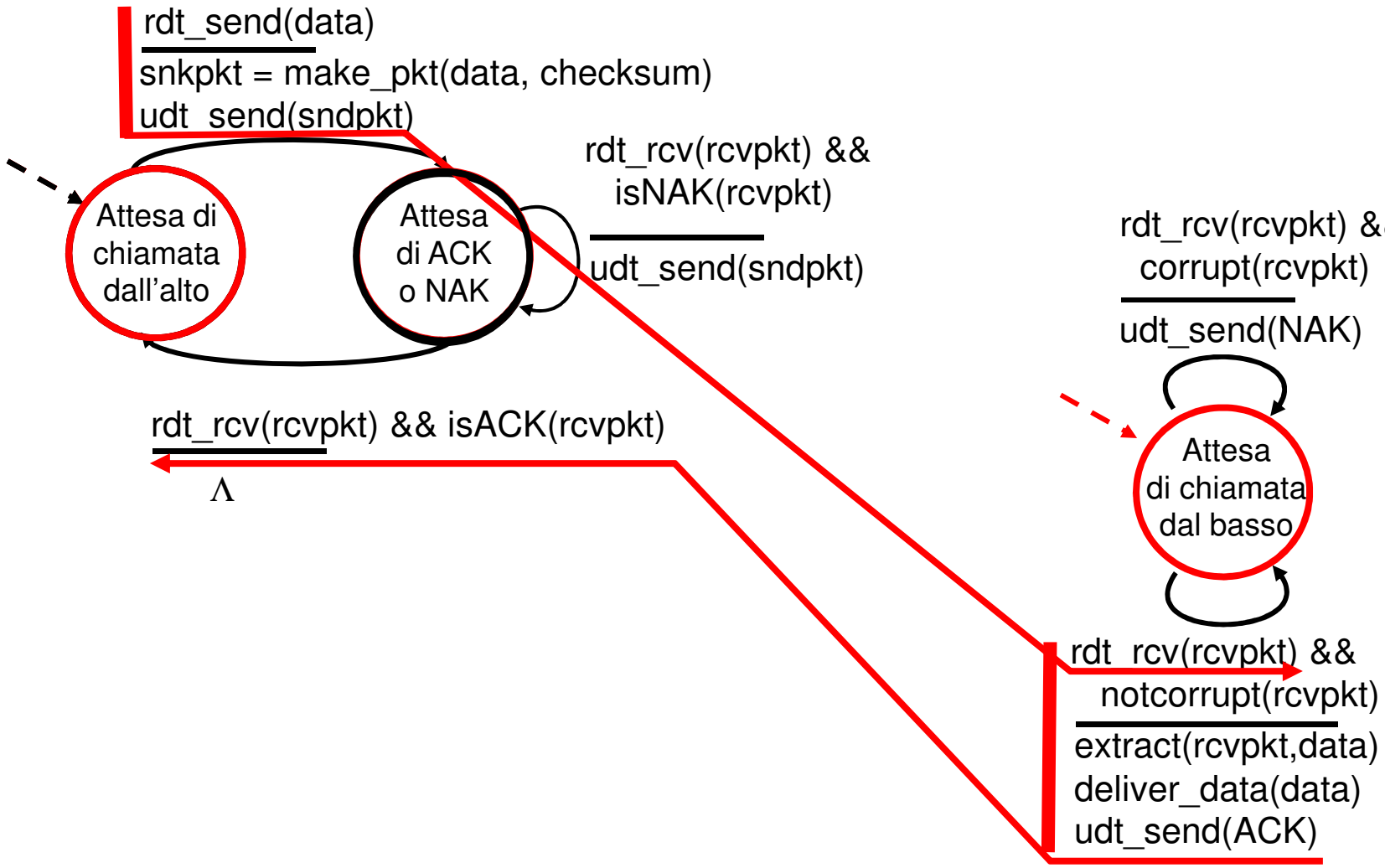


Automa mittente

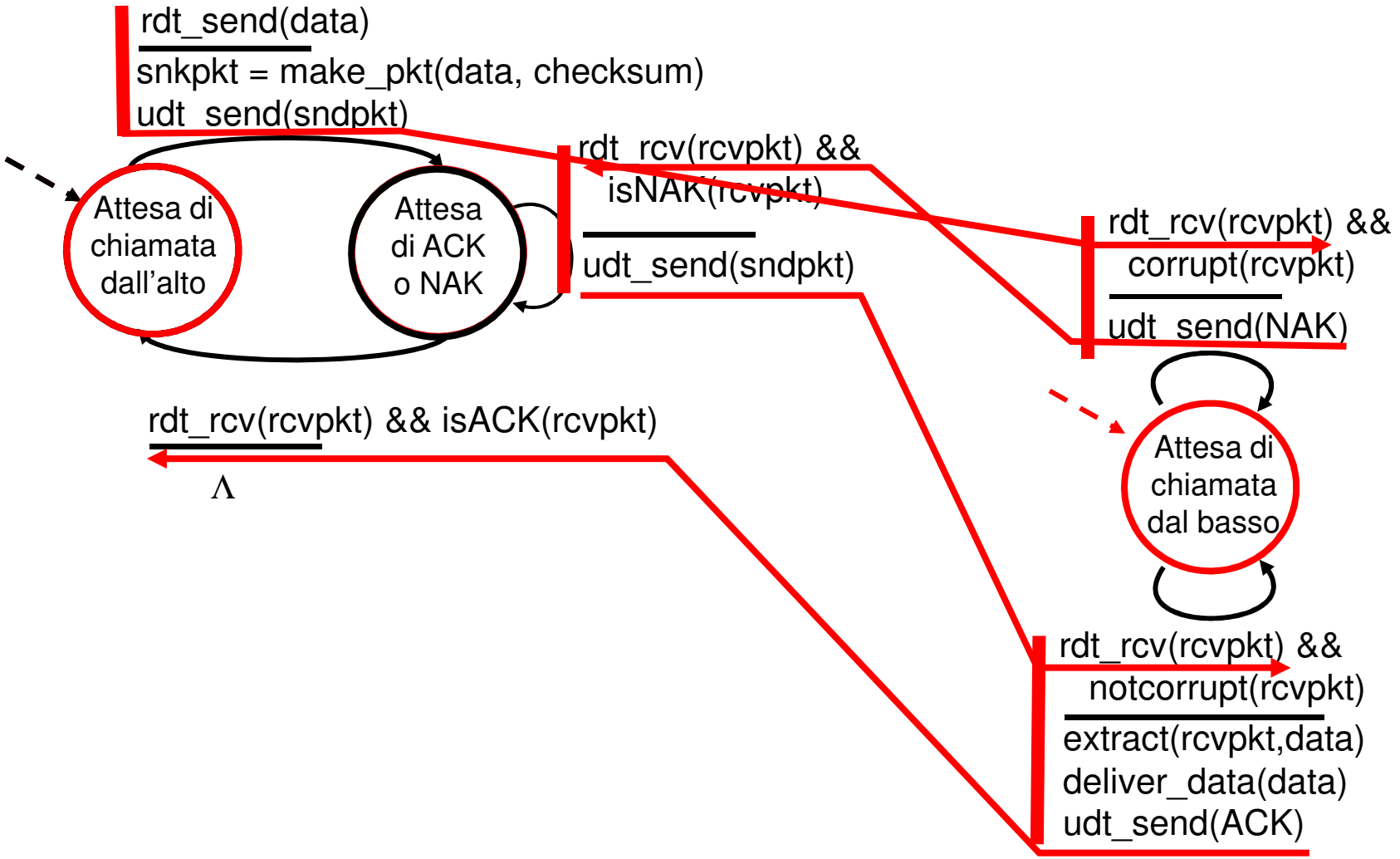
Automa ricevente



Rdt2.0: operazione senza errori



Rdt2.0: scenario di errore



Da notare

- **Quando il mittente è in attesa di ACK/NACK non può accettare dati dal livello superiore**
  - Non può aver luogo l'evento **rdt_send()** senza la ricezione di un **ACK** che permette al mittente di cambiare stato
  - Il mittente non invierà nuovi dati finchè non è certo che il ricevente abbia ricevuto correttamente il pacchetto corrente
  - Rdt 2.0 è un protocollo di tipo **stop-and-wait**
- **L'automa lato rx ha un solo stato**

Rdt2.0 ha un difetto fatale!

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

- Il mittente non sa se il destinatario abbia ricevuto correttamente o meno l'ultimo pacchetto trasmesso !
- Non basta ritrasmettere: possibili duplicati

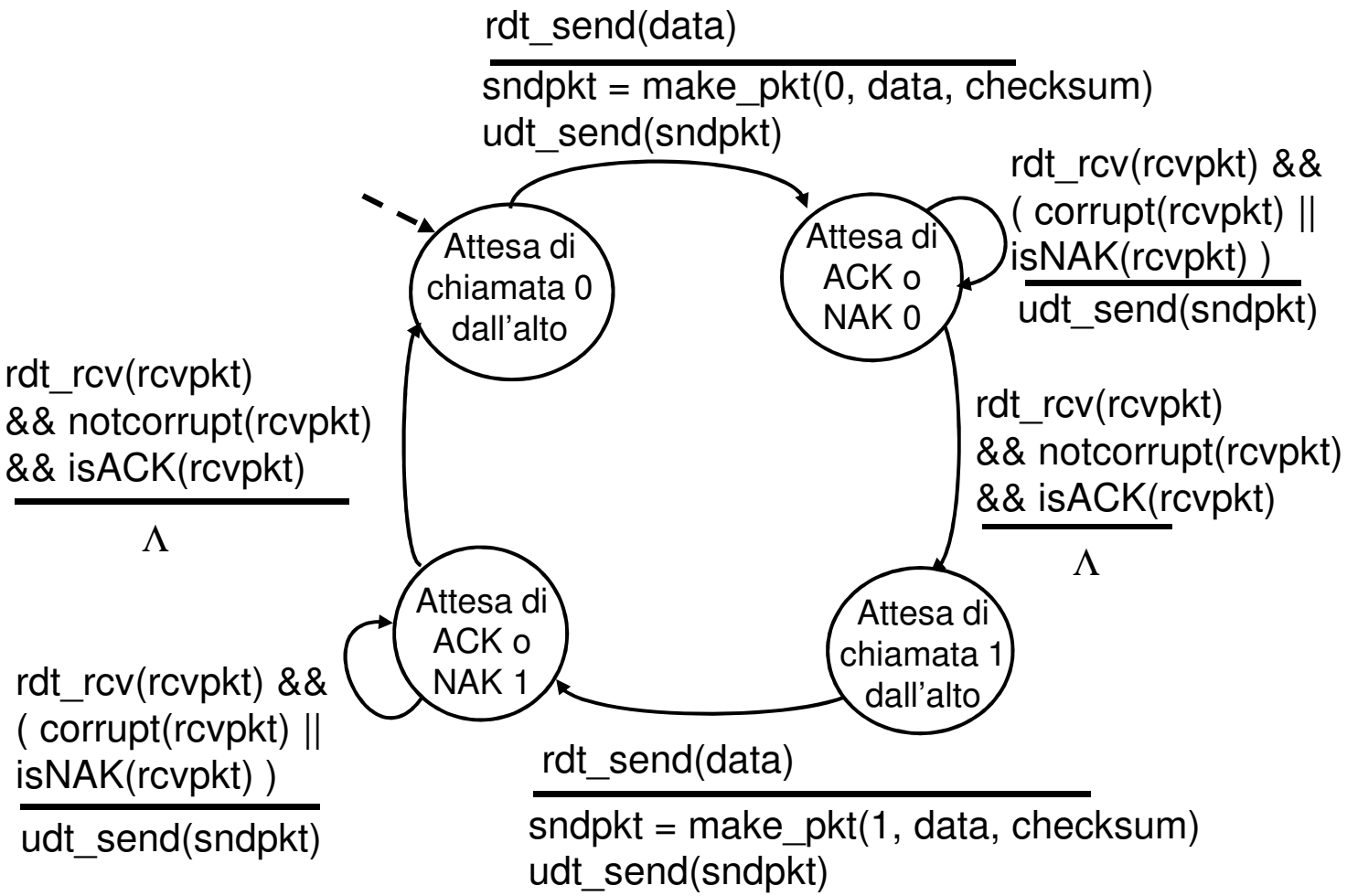
Gestione dei duplicati:

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

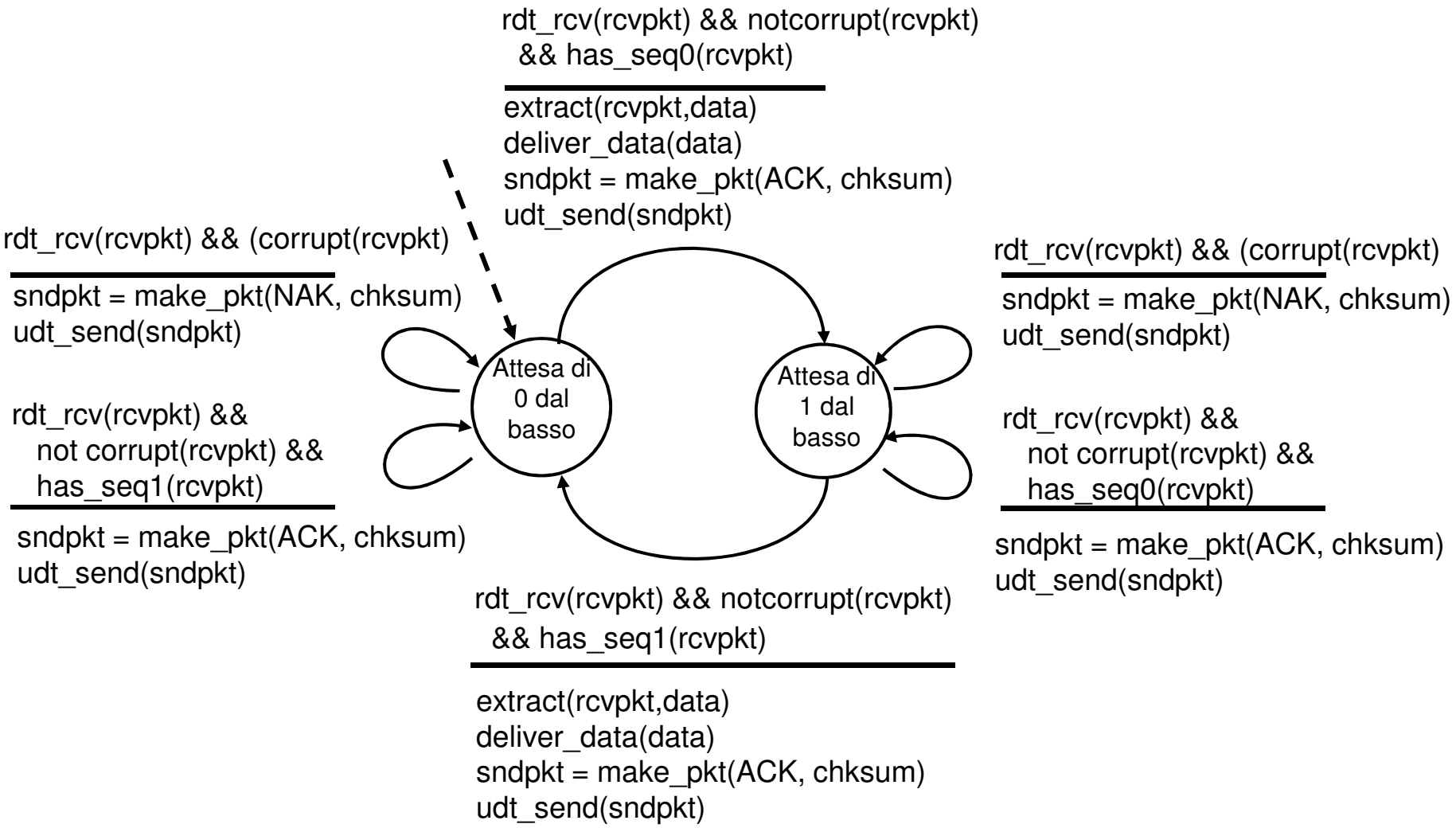
Numero di sequenza

In questo modello di canale con errori ma senza perdita, basta un bit per numerare i pacchetti

Rdt2.1: il mittente gestisce gli ACK/NAK alterati



Rdt2.1: il ricevente gestisce gli ACK/NAK alterati



Rdt2.1: discussione

Mittente:

- **Aggiunge il numero di sequenza al pacchetto**
- **Saranno sufficienti due numeri di sequenza (0,1). Perché?**
- **Deve controllare se gli ACK/NAK sono danneggiati**
- **Il doppio di stati**
  - lo stato deve "ricordarsi" se il pacchetto "corrente" ha numero di sequenza 0 o 1

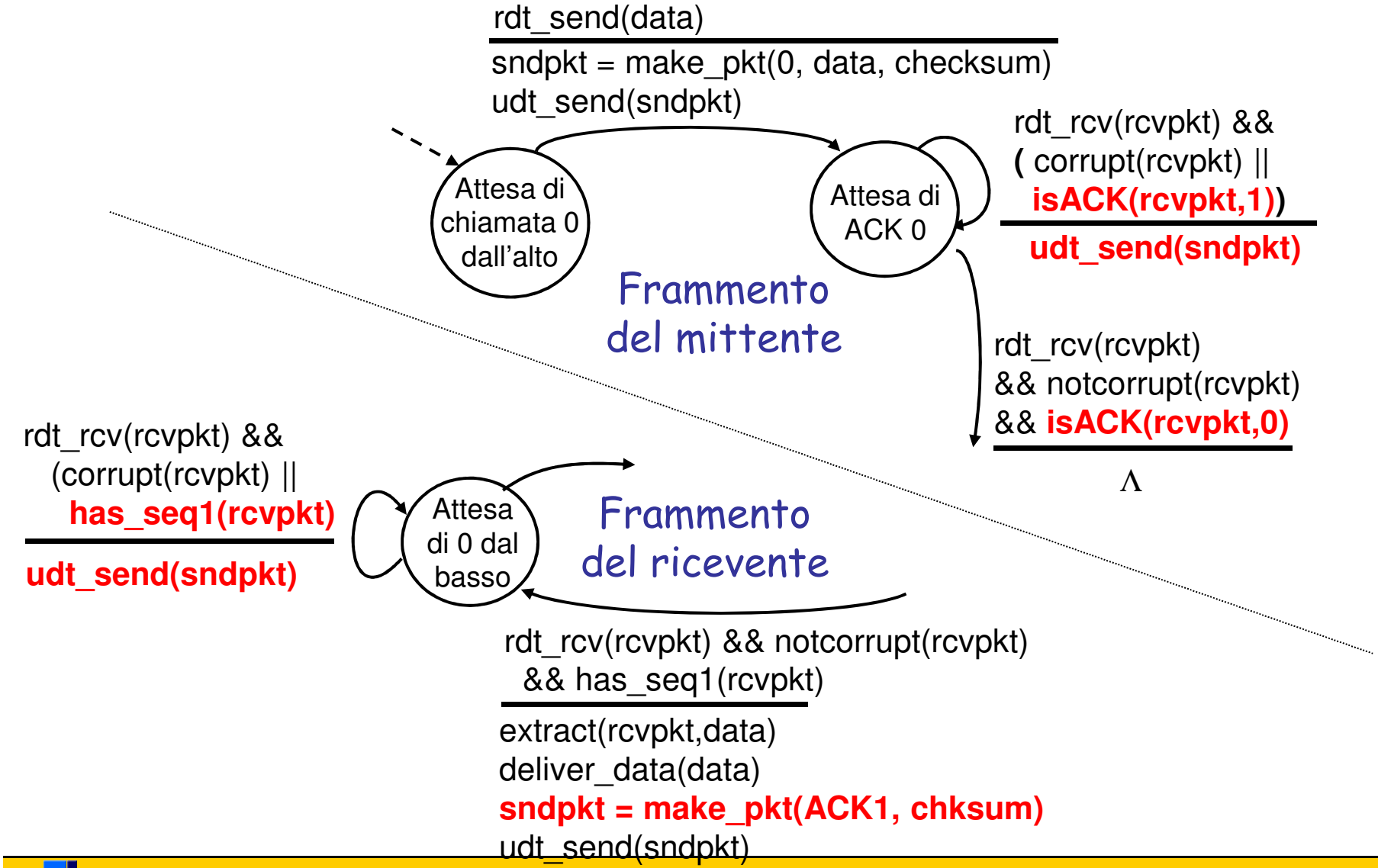
Ricevente:

- **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*

Rdt2.2: frammenti del mittente e del ricevente



Rdt3.0: canali con errori *e* perdite

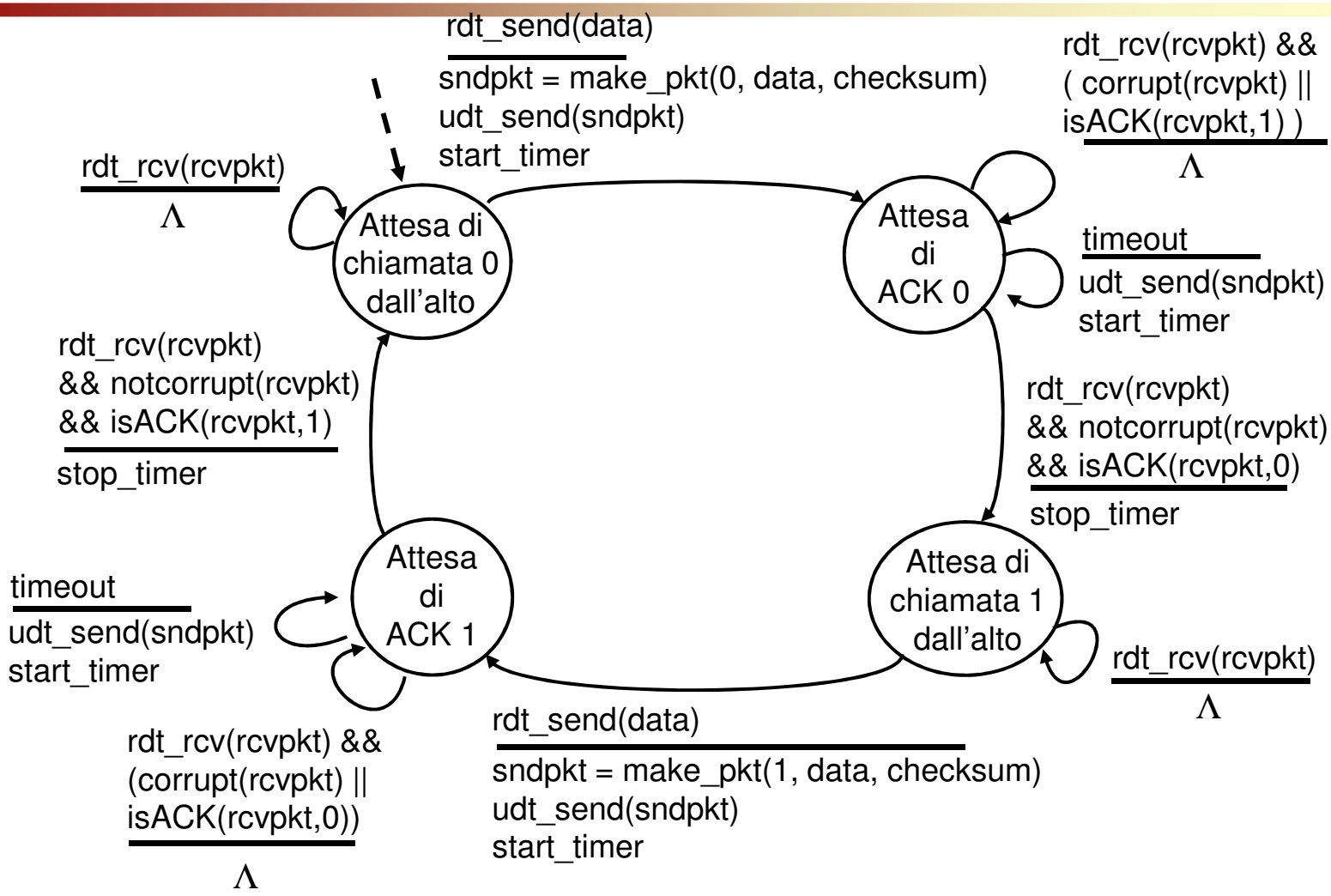
Nuova ipotesi: **il canale sottostante può anche smarrire i pacchetti (dati o ACK)**

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

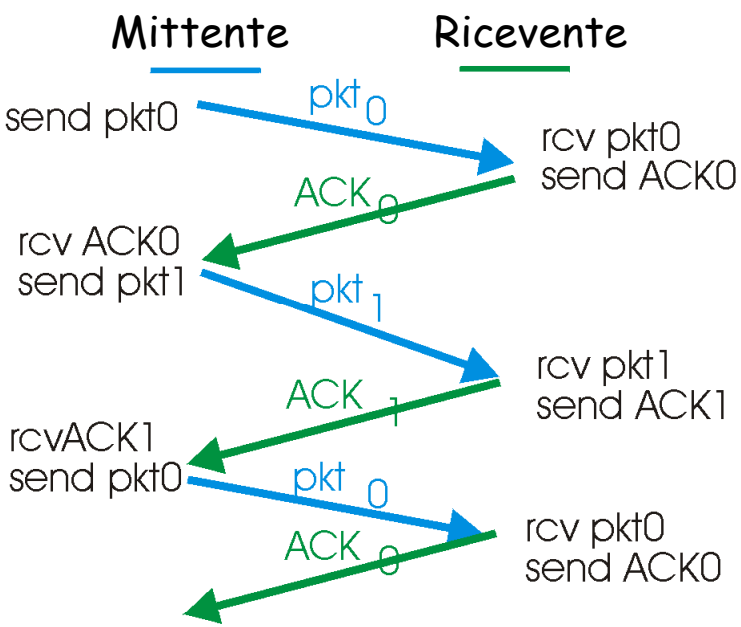
Approccio: **il mittente attende un ACK per un tempo "ragionevole"**

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

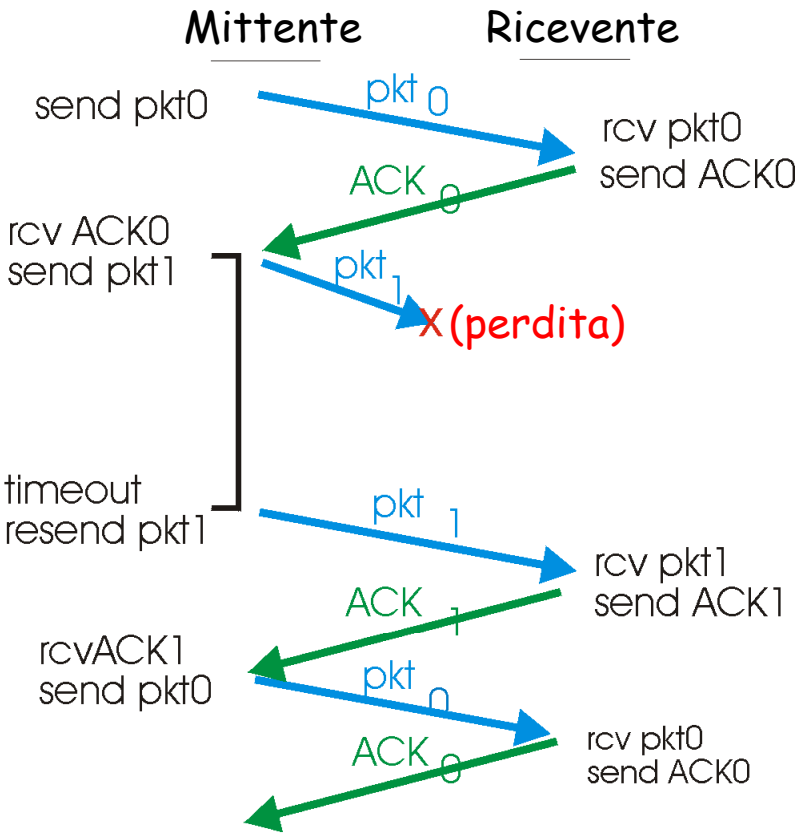
Rdt3.0 mittente



Rdt3.0 in azione

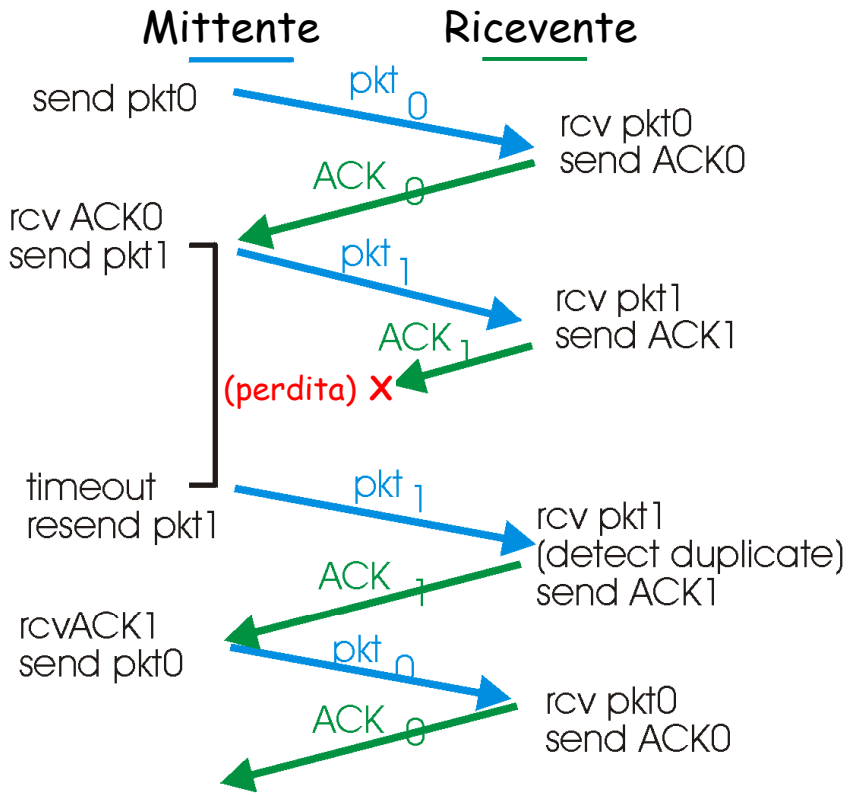


a) Operazioni senza perdite

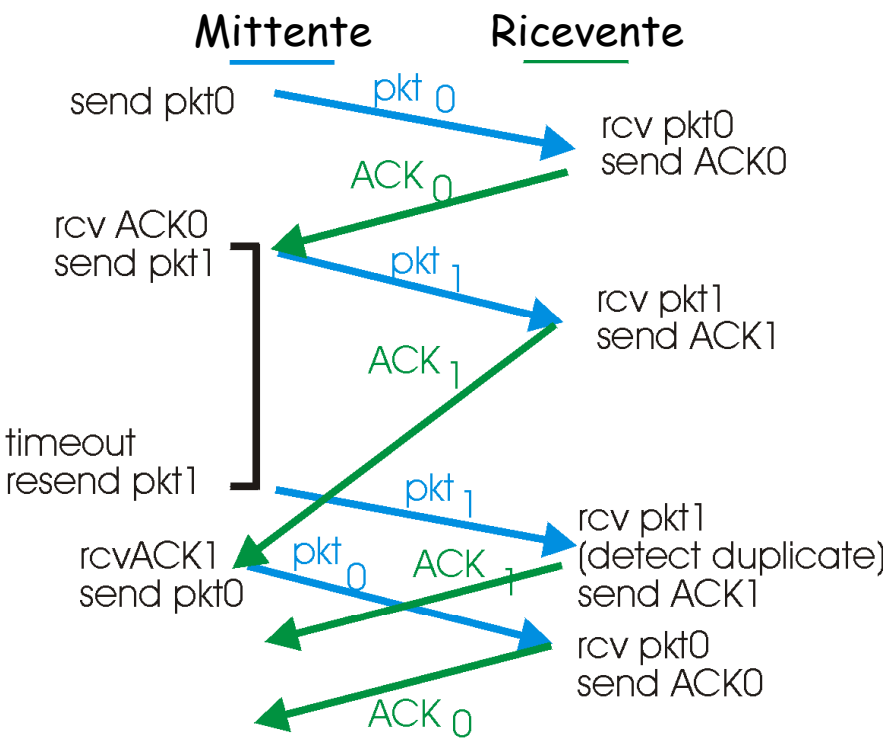


b) Perdita di pacchetto

Rdt3.0 in azione



c) Perdita di ACK



d) Timeout prematuro

Rdt 3.0 viene anche detto protocollo ad alternanza di bit

Key features di un protocollo affidabile

- **Checksum**
  - Per rilevare gli errori
- **Numeri di sequenza**
  - Per evitare duplicazioni
- **Contatori/timer**
  - Per ritrasmissione automatica
- **Ack/Nack**
  - Feedbacks

Prestazioni di rdt3.0

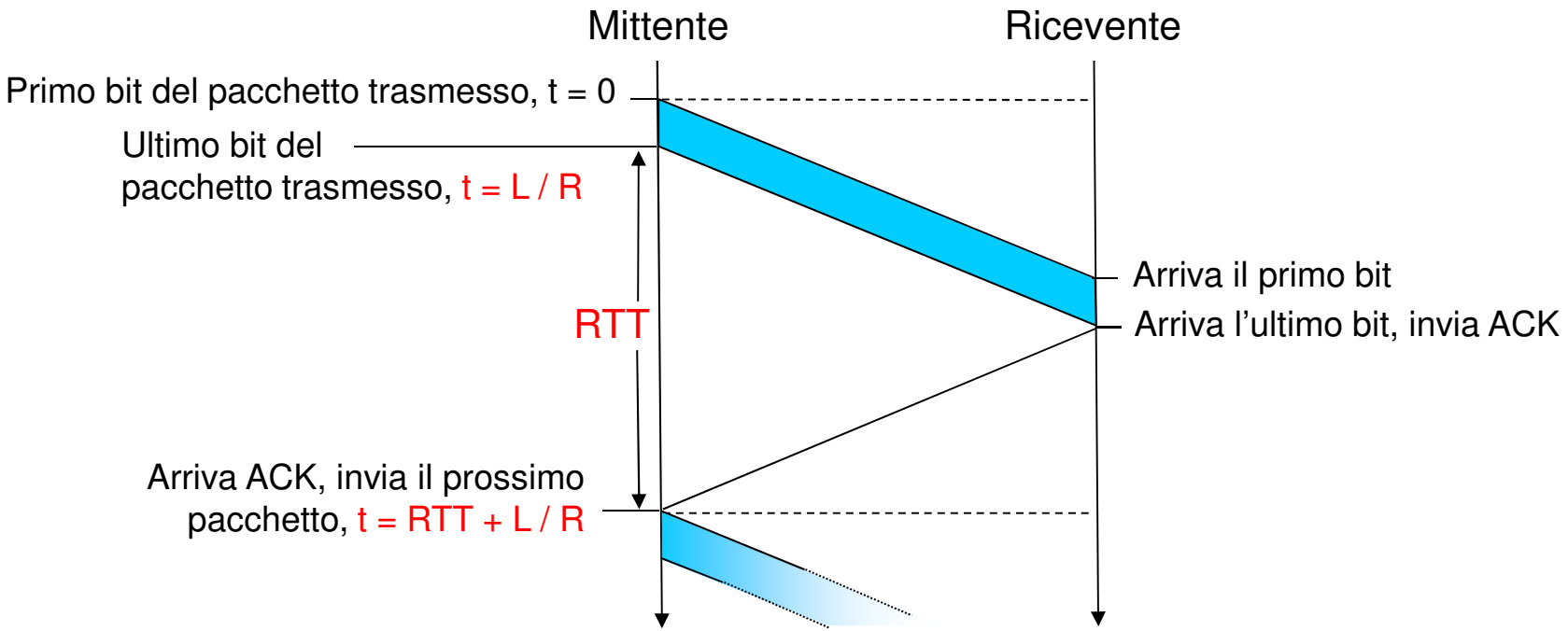
- **rdt3.0 funziona, ma le prestazioni non sono apprezzabili**
- **esempio: collegamento da 1 Gbps, due hosts uno sulla costa occidentale degli USA e uno su quella orientale (ritardo di propagazione 15 ms), pacchetti di dimensione L= 1000 Bytes:**

$$T_{trasm} = \frac{L \text{ (lunghezza del pacchetto in bit)}}{R \text{ (tasso trasmissivo, bps)}} = \frac{8000 \text{ bit/pacc}}{10^9 \text{ bit/sec}} = 8 \text{ microsec}$$

$$U_{mitt} = \frac{L/R}{RTT + L/R} = \frac{0{,}008}{30{,}008} = 0{,}00027$$

- U_{mitt} : **utilizzo** è la frazione di tempo in cui il mittente è occupato nell'invio di bit
- Un pacchetto da 1 KB ogni 30 msec → throughput di 33 kB/sec ovvero 267 Kbps in un collegamento da 1 Gbps (1000000 Kbps)
- Il protocollo di rete limita l'uso delle risorse fisiche!

Rdt3.0: funzionamento con stop-and-wait

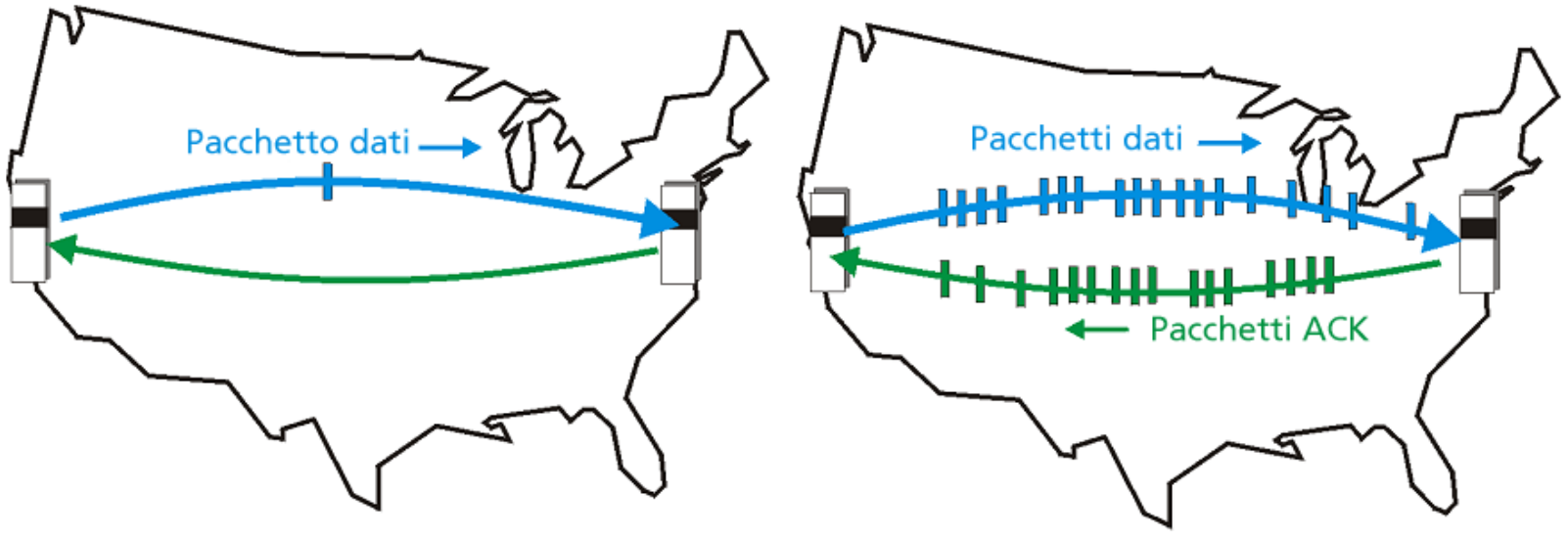


$$U_{mitt} = \frac{L / R}{RTT + L / R} = \frac{0{,}008}{30{,}008} = 0{,}00027 \text{ microsec}$$

Protocolli con pipeline

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

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

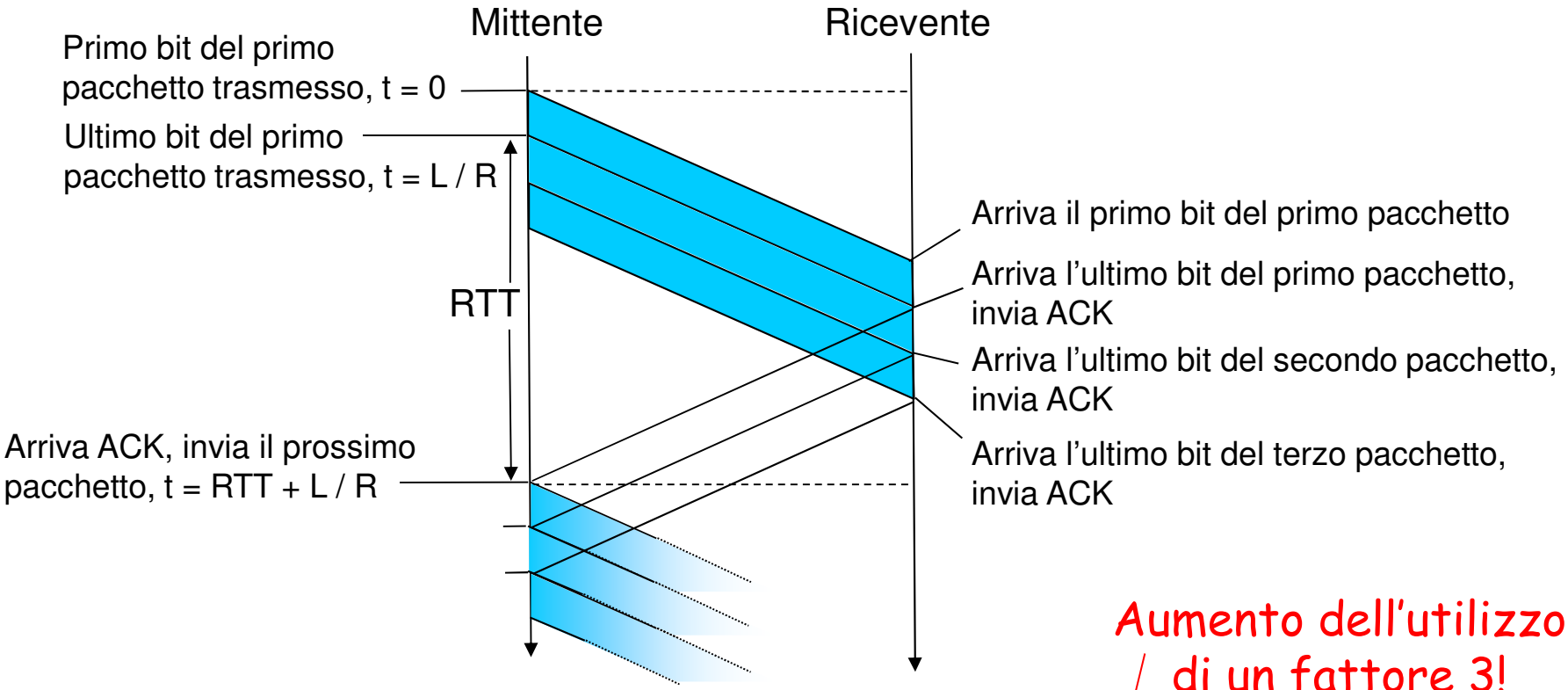


a) Protocollo stop-and-wait all'opera

b) Protocollo con pipeline all'opera

- **Due forme generiche di protocolli con pipeline: *Go-Back-N* e *ripetizione selettiva***

Pipelining: aumento dell'utilizzo



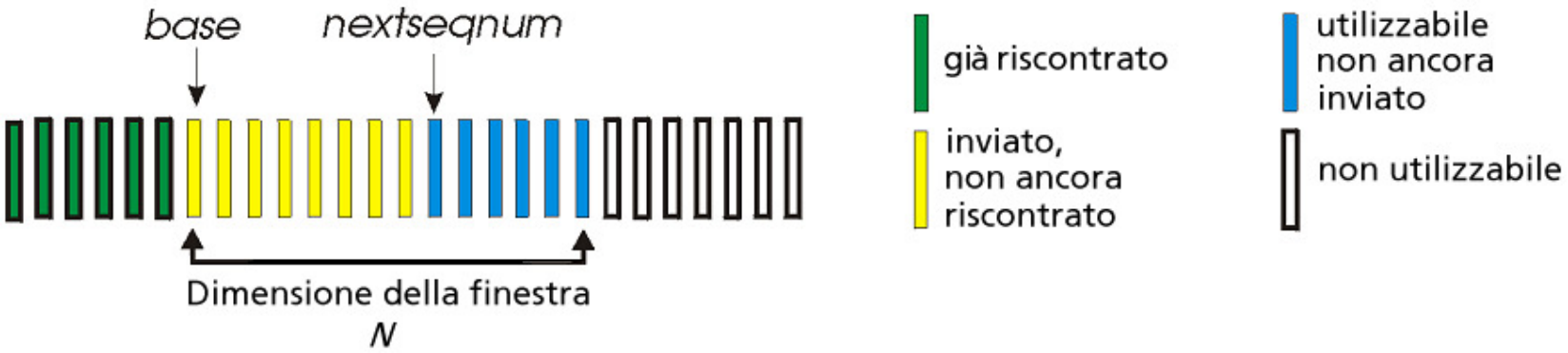
Aumento dell'utilizzo di un fattore 3!

$$U_{mitt} = \frac{3 * L / R}{RTT + L / R} = \frac{0{,}024}{30{,}008} = 0{,}0008$$

Go-Back-N

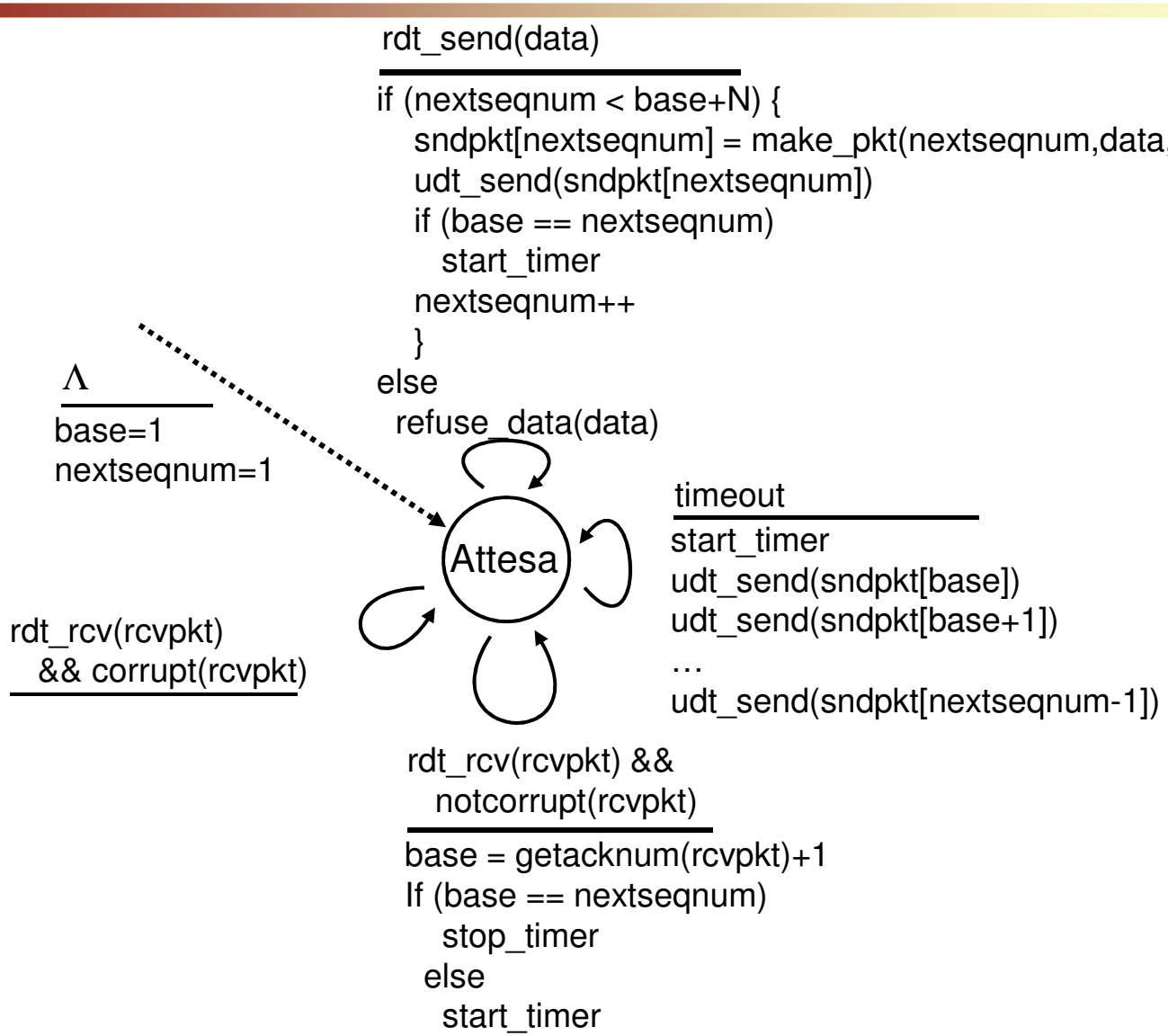
Mittente:

- **Numero di sequenza a k bit nell'intestazione del pacchetto**
- **"Finestra" contenente fino a N pacchetti consecutivi non riscontrati**

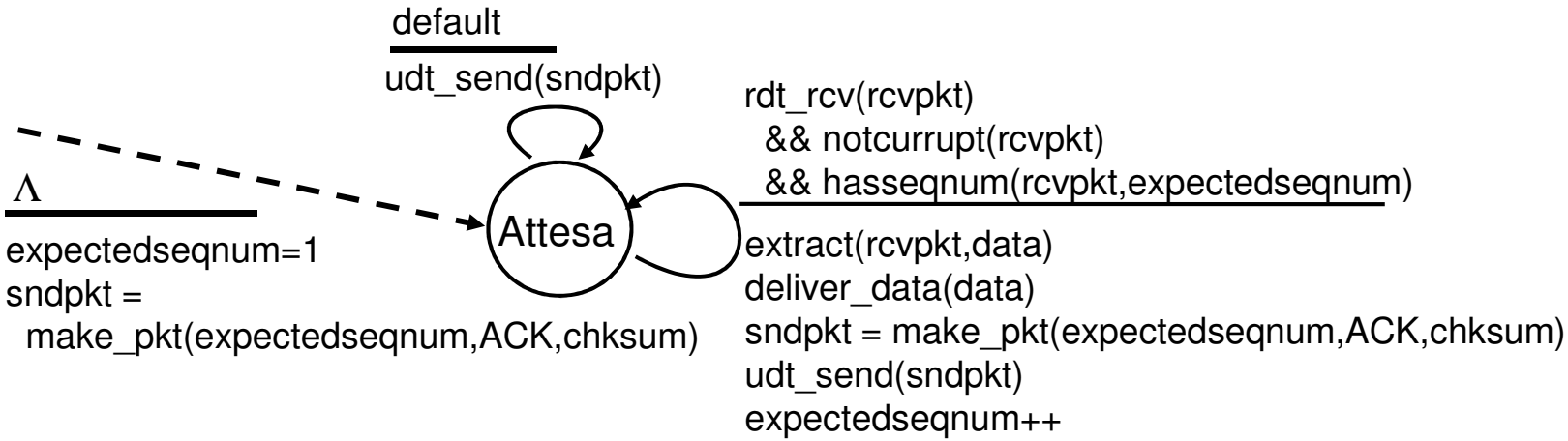


- **ACK(n): riscontro di tutti i pacchetti con numero di sequenza minore o uguale a n - "riscontri cumulativi"**
  - pacchetti duplicati potrebbero essere scartati (vedere il ricevente)
- **timer per ogni pacchetto in transito**
- ***timeout(n):* ritrasmette il pacchetto n e tutti i pacchetti con i numeri di sequenza più grandi nella finestra**

GBN: automa esteso del mittente



GBN: automa esteso del ricevente

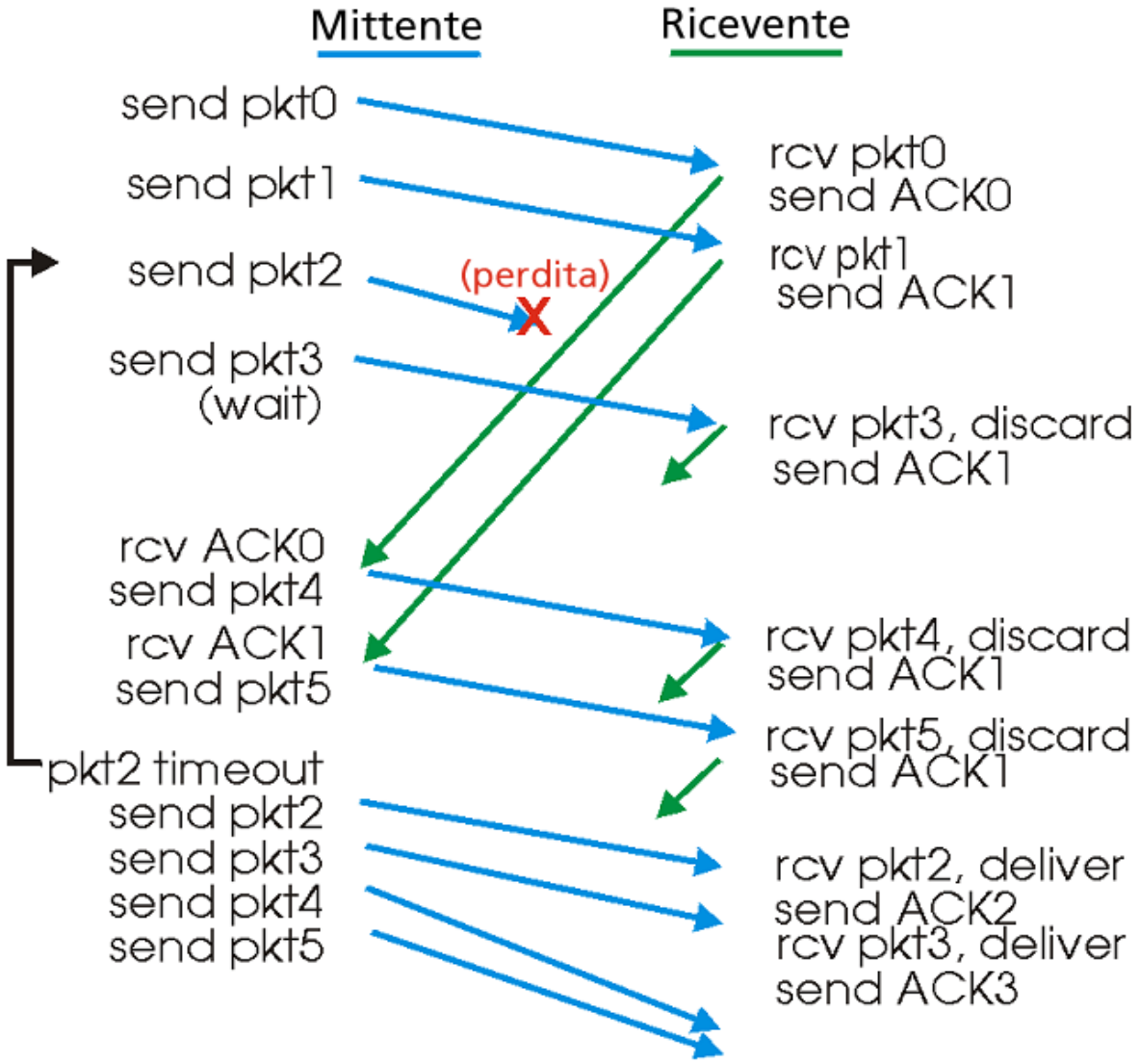


ACK-soltanto: invia sempre un ACK per un pacchetto ricevuto correttamente con il numero di sequenza più alto *in sequenza*

- potrebbe generare ACK duplicati
- deve memorizzare soltanto **`expectedseqnum`**

- **Pacchetto fuori sequenza:**
  - scartato (non è salvato) -> senza buffering del ricevente!
  - rimanda un ACK per il pacchetto con il numero di sequenza più alto *in sequenza*

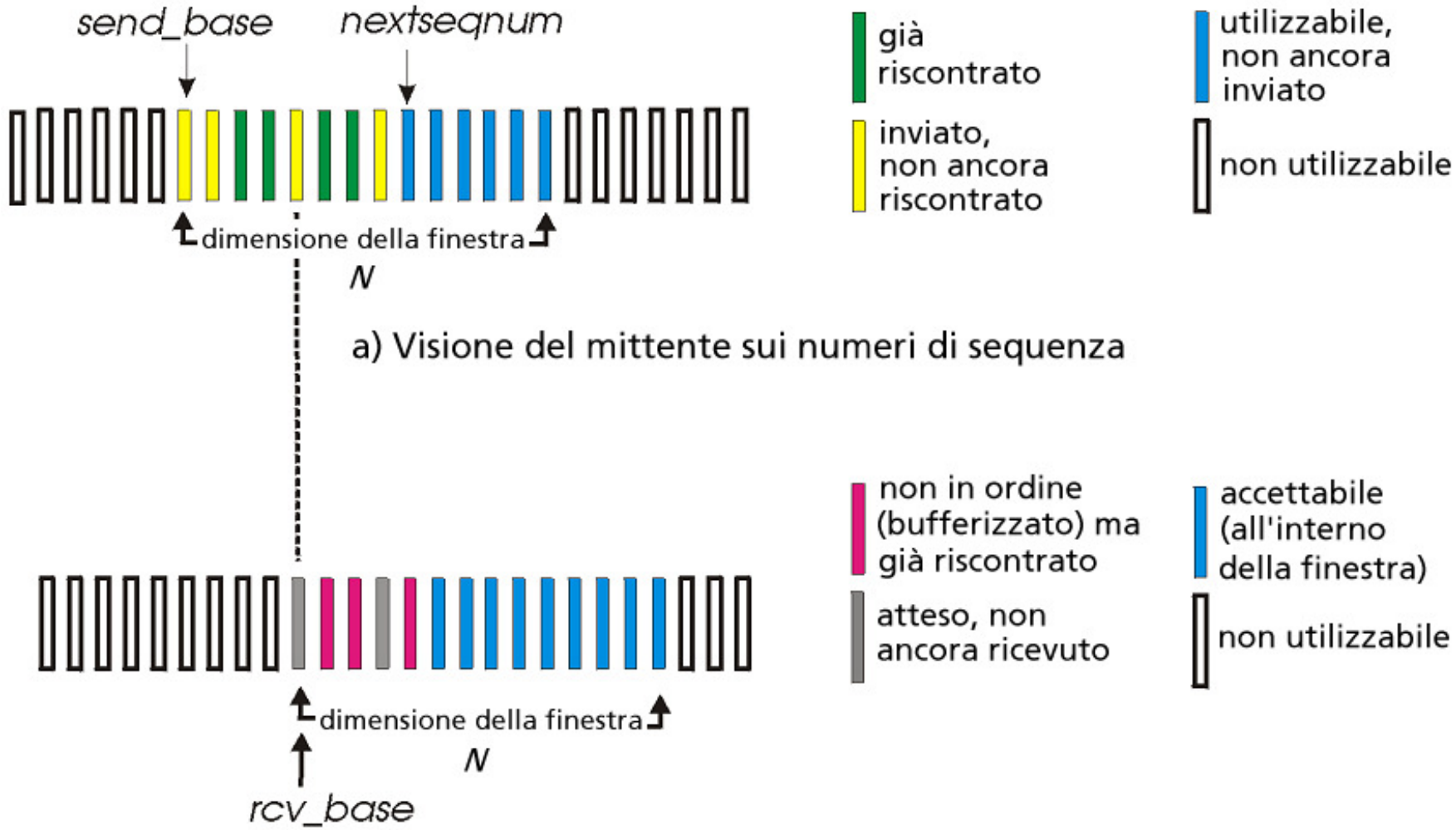
GBN in azione



Ripetizione selettiva

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

Ripetizione selettiva: finestre del mittente e del ricevente



Ripetizione selettiva

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]:

- 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

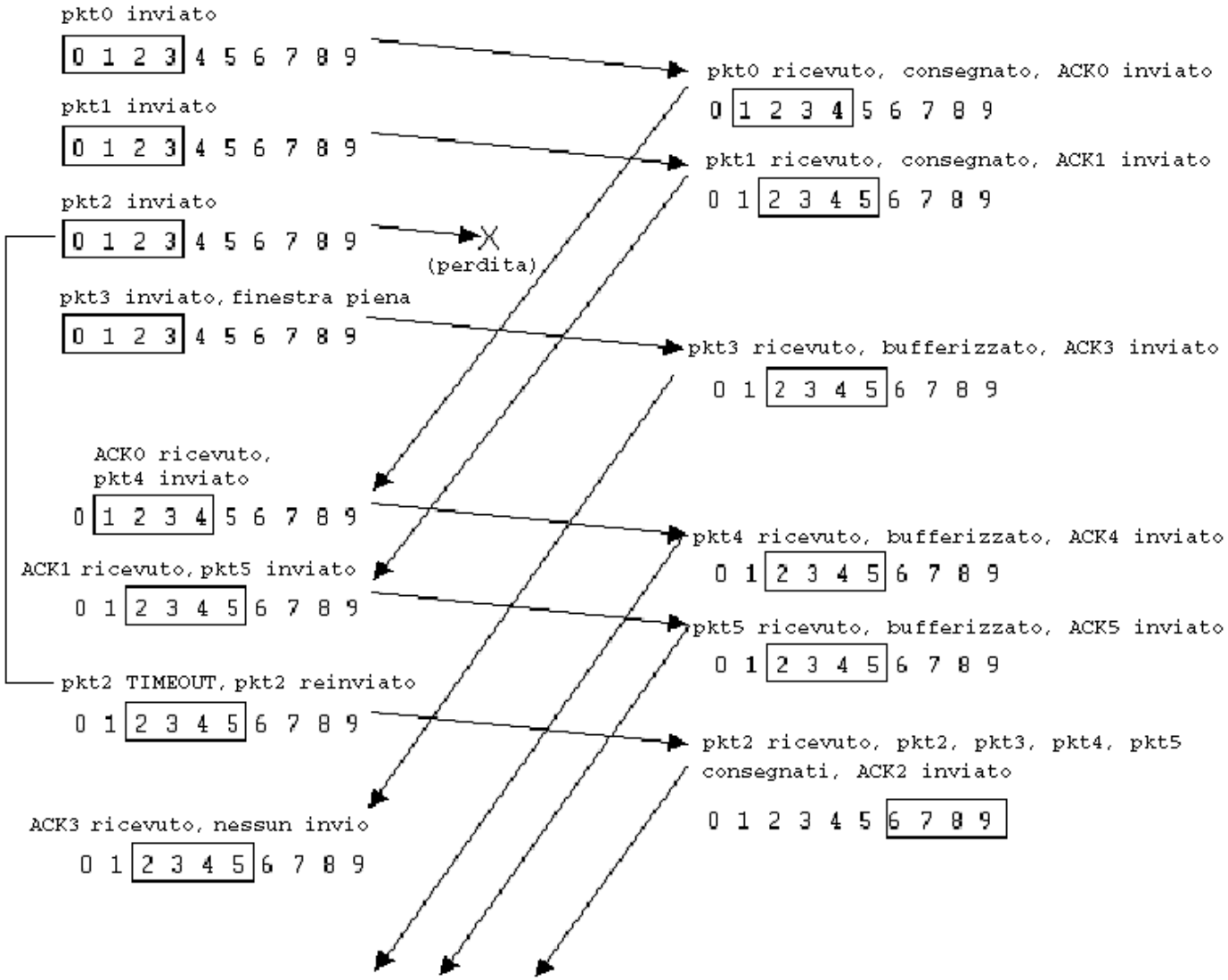
Pacchetto n in [rcvbase-N, rcvbase-1]

- ACK(n)

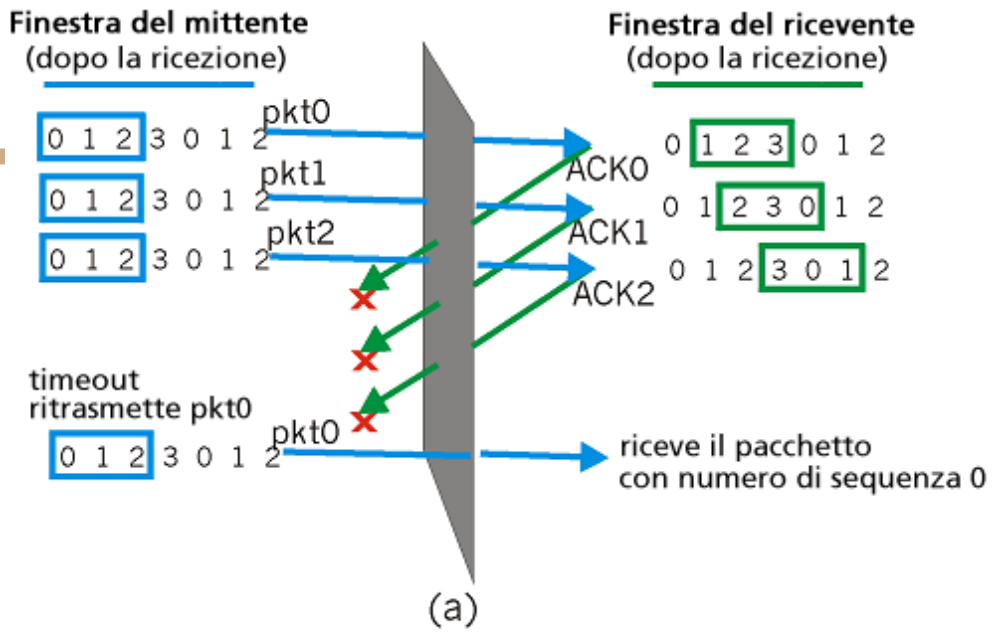
altrimenti:

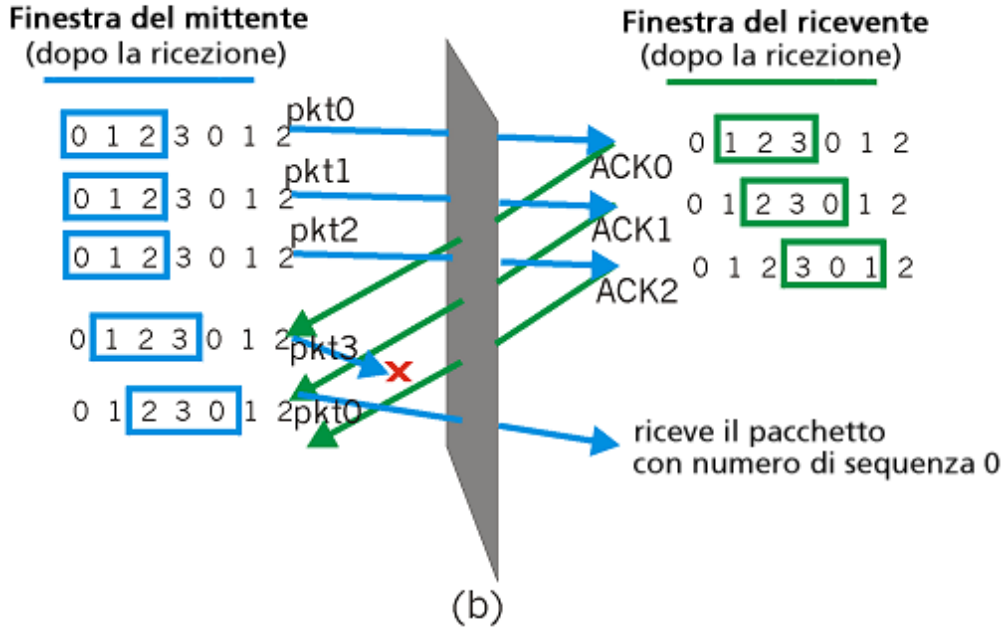
- ignora

Ripetizione selettiva in azione



Ripetizione selettiva: dilemma





Esempio:

- Numeri di sequenza: 0, 1, 2, 3
- Dimensione della finestra = 3

- Il ricevente non vede alcuna differenza fra i due scenari!
- Passa erroneamente i dati duplicati come nuovi in (a)

D: Qual è la relazione fra lo spazio dei numeri di sequenza e la dimensione della finestra?

Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

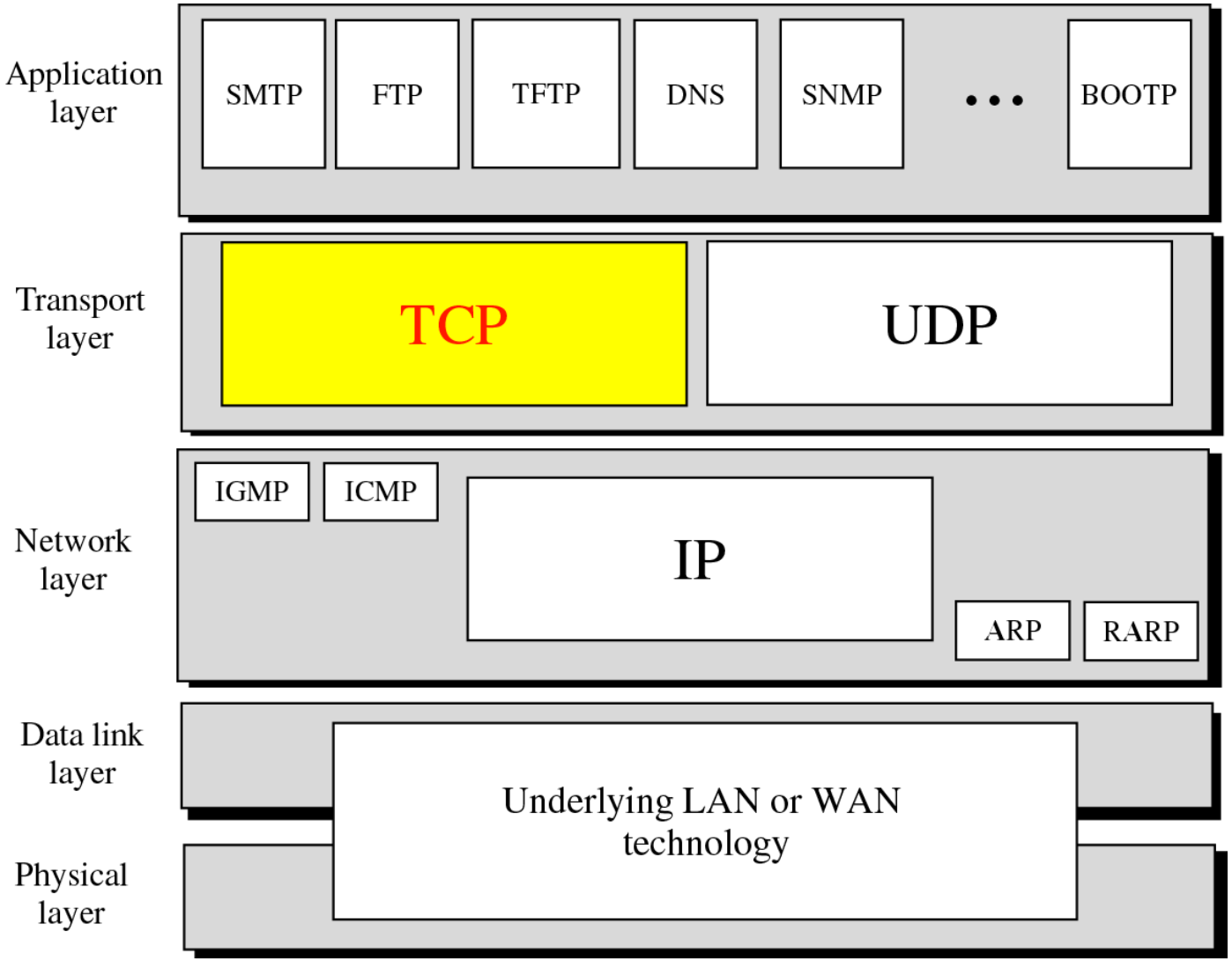
3.5 Trasporto orientato alla connessione: TCP

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

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

Transmission Control Protocol



TCP: RFC: 793, 1122, 1323, 2018, 2581

- **Comunicazione da-processo-a-processo**
- **Una connessione TCP è punto-punto:**
  - un mittente, un destinatario
  - il multicast non è possibile con TCP
- **flusso di byte affidabile, in sequenza*:***
  - nessun “confine ai messaggi”
- **pipeline:**
  - il controllo di flusso e di congestione TCP definiscono la dimensione della finestra
- ***buffer d’invio e di ricezione***
  - Una connessione TCP consiste di buffers, variabili e di 2 sockets verso i due end-systems

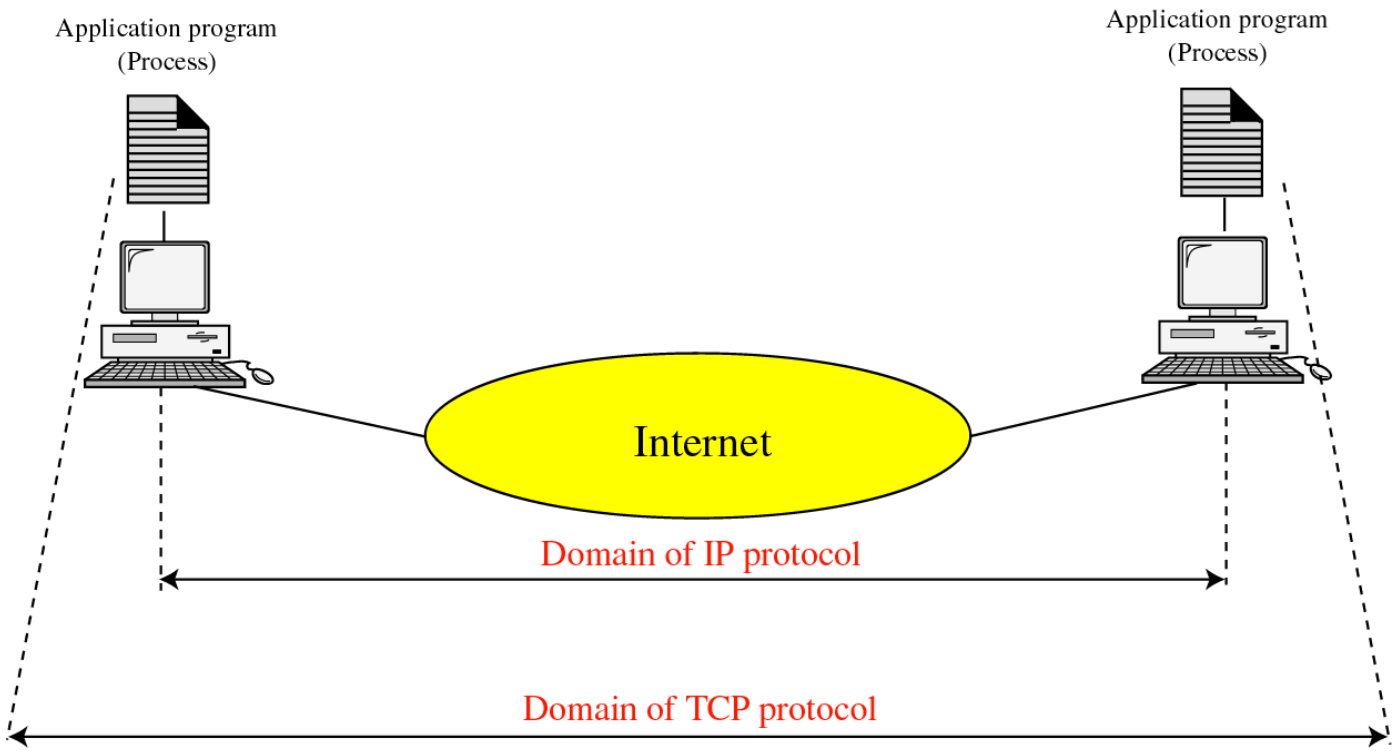
- **Una connessione TCP offre un servizio full duplex:**
  - flusso di dati bidirezionale nella stessa connessione
  - la massima quantità di **dati** che vengono imbustati in un **segmento** è detta **MSS** (maximum segment size)
- **orientato alla connessione:**
  - l’handshaking (scambio di messaggi di controllo) inizializza lo stato del mittente e del destinatario prima di scambiare i dati
- **flusso controllato:**
  - il mittente non sovraccarica il destinatario
- **Processo tx ≠ da processo Rx:**
  - Due buffer: uno per lato

Attenzione

- **Una connessione TCP non è un circuito**
- **..... e neanche un circuito virtuale**
  - lo stato della connessione risiede solo nei sistemi terminali
  - I router non eseguono TCP per cui sono del tutto ignari delle connessioni
- **I router vedono datagrammi non connessioni.**

Comunicazione Process-to-Process

- **IP: Host-to-Host**
- **TCP: Process-to-Process**



Porte

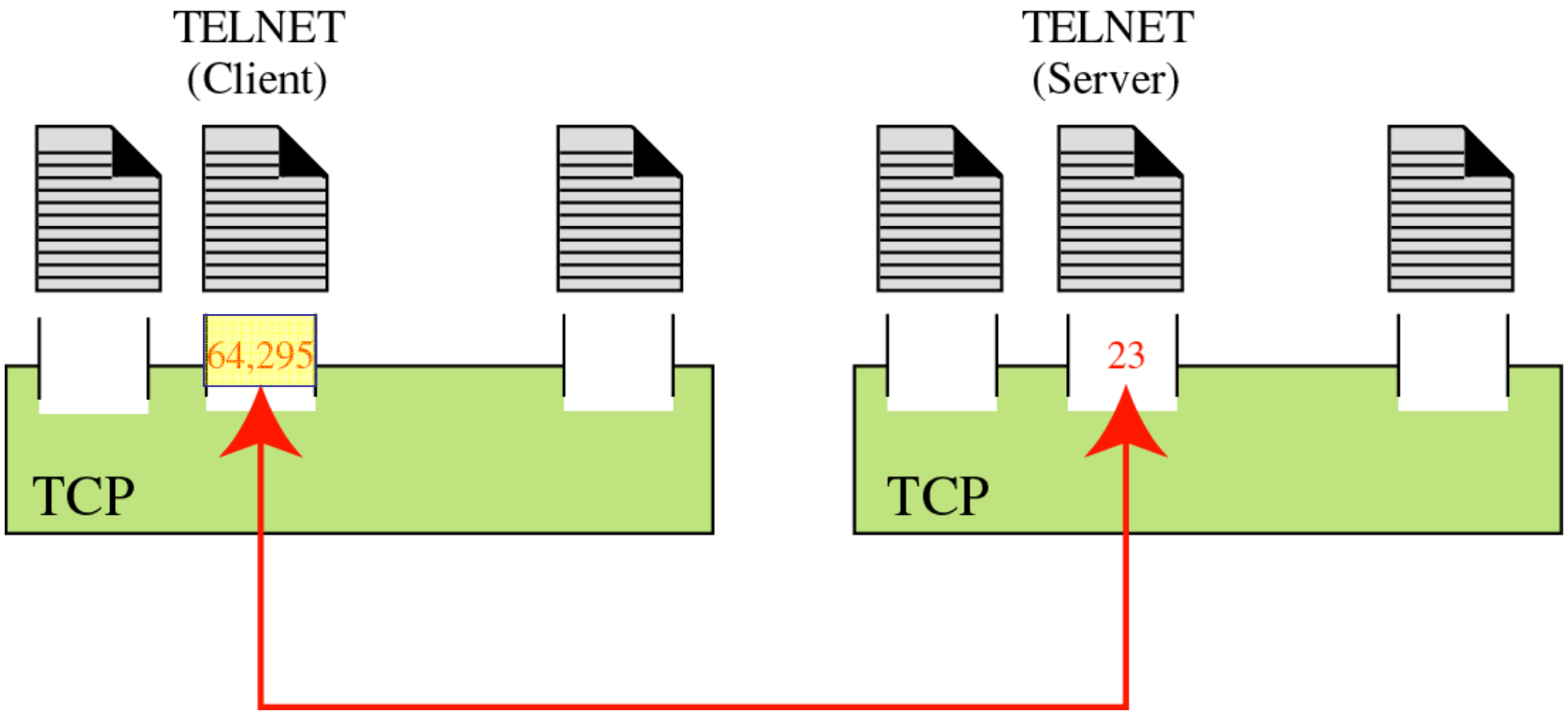
- **Per definire un processo è necessaria una porta.**

```
// create a TCP socket
$socket = socket_create(AF_INET, SOCK_STREAM, 0)
// bind socket to port
$result = socket_bind($socket, $host, $port)
```

- **Tipi:**
  - Effimera: random port number definito dal client.
  - Well-known port number: numeri di porta definiti da IANA e usate dal server.

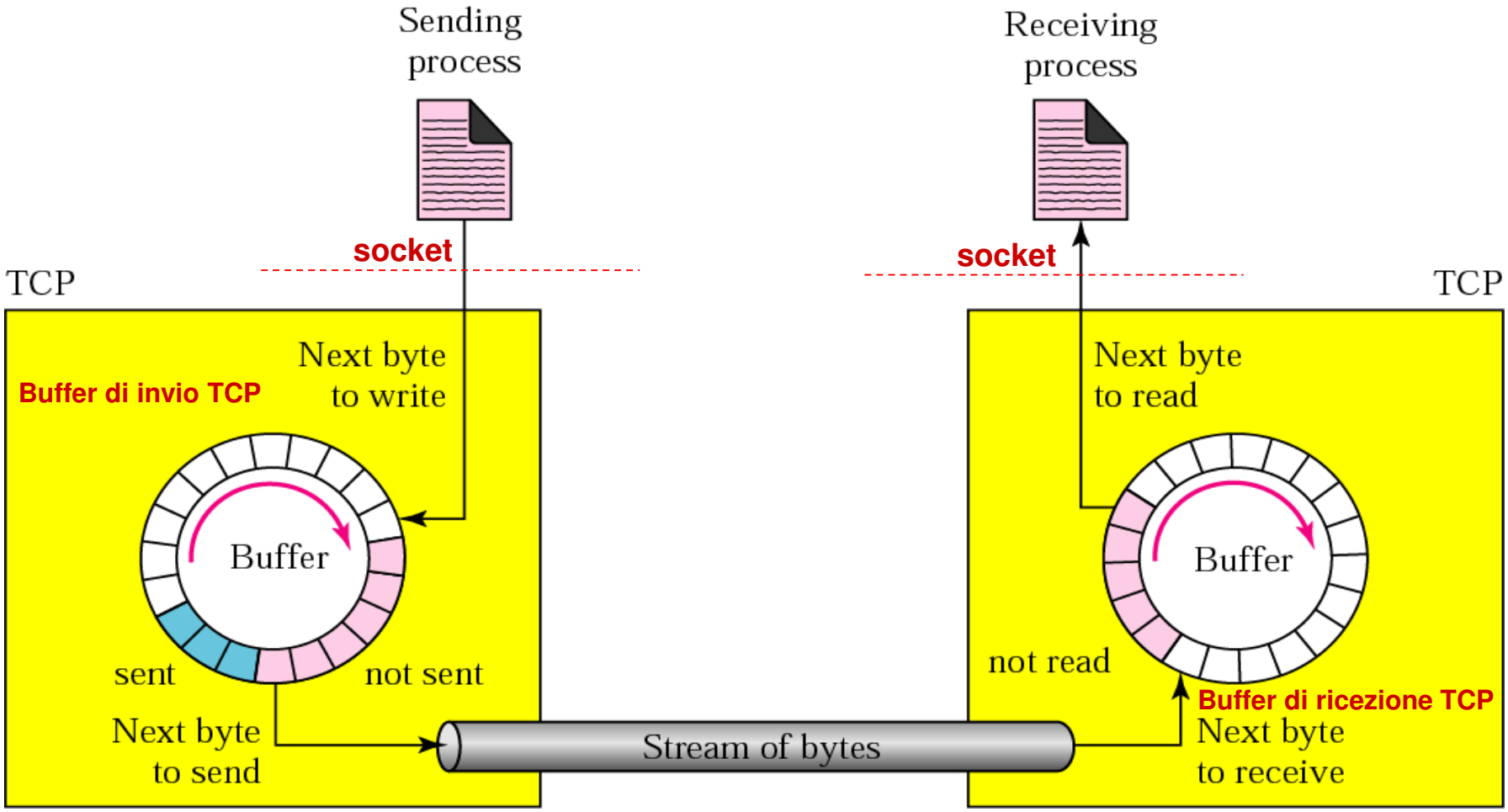
Telnet: porte TCP

Una sessione Telnet



Connessione TCP

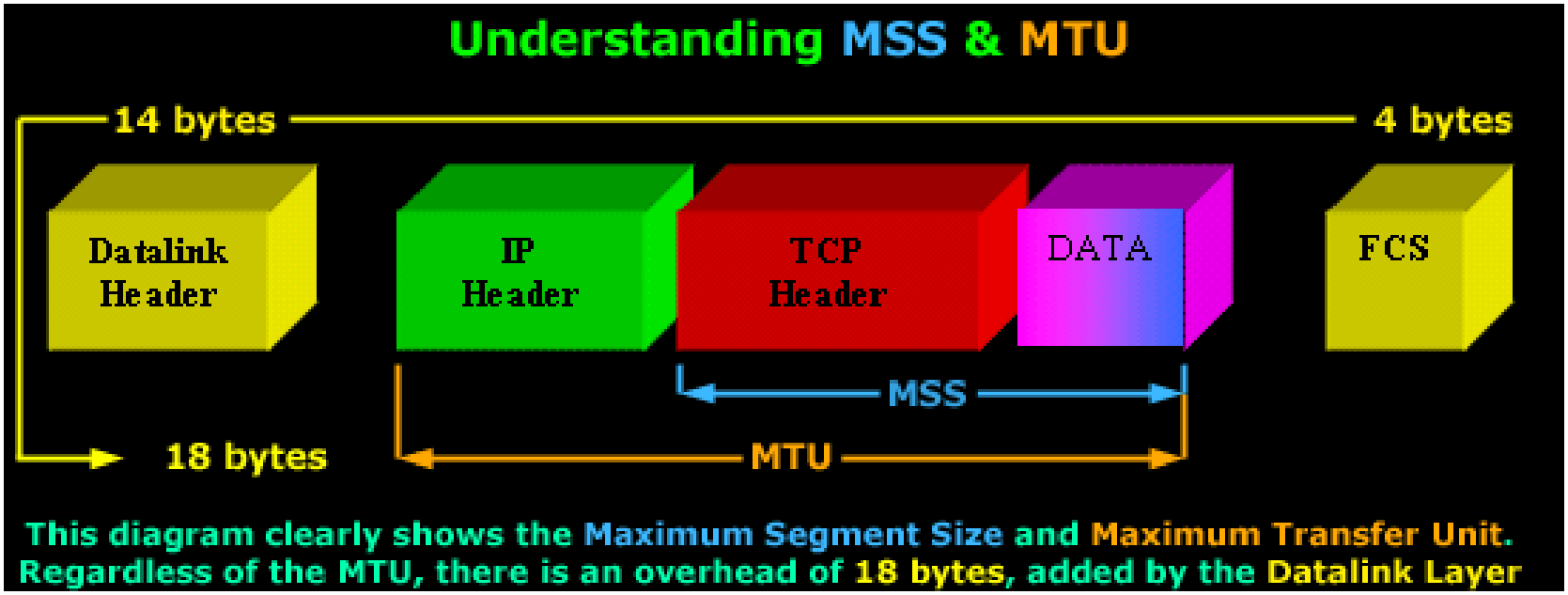
Una connessione TCP consiste di buffers, variabili e di 2 sockets verso i due end-systems



MSS e MTU

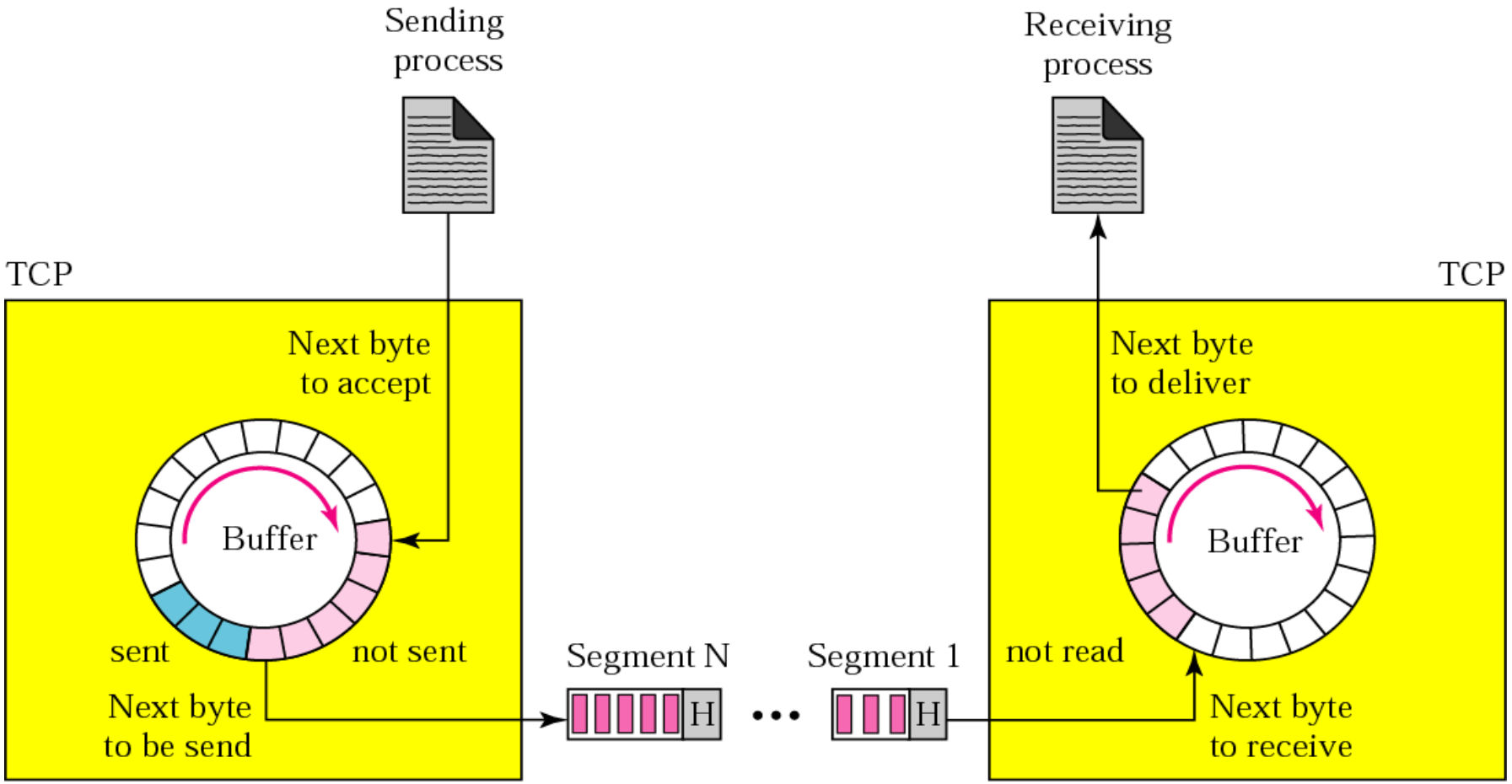
- **Instaurata la connessione i processi si scambiano i dati.**
- **I dati attraversano la socket e vengono consegnati a TCP**
  - La RFC dice: “spedire i dati in segmenti quando è più conveniente”
- **La massima quantità di dati da prelevare è delimitata da MSS (maximum segment size)**
  - MSS è relazionata a MTU (maximum transmission unit) che definisce il frame più grande a livello di datalink
  - Valori comuni di MTU sono: 1460, 536, 512
- **MSS viene scelto tale da permettere ad un segmento TCP di stare all’interno di un frame**

MSS e MTU

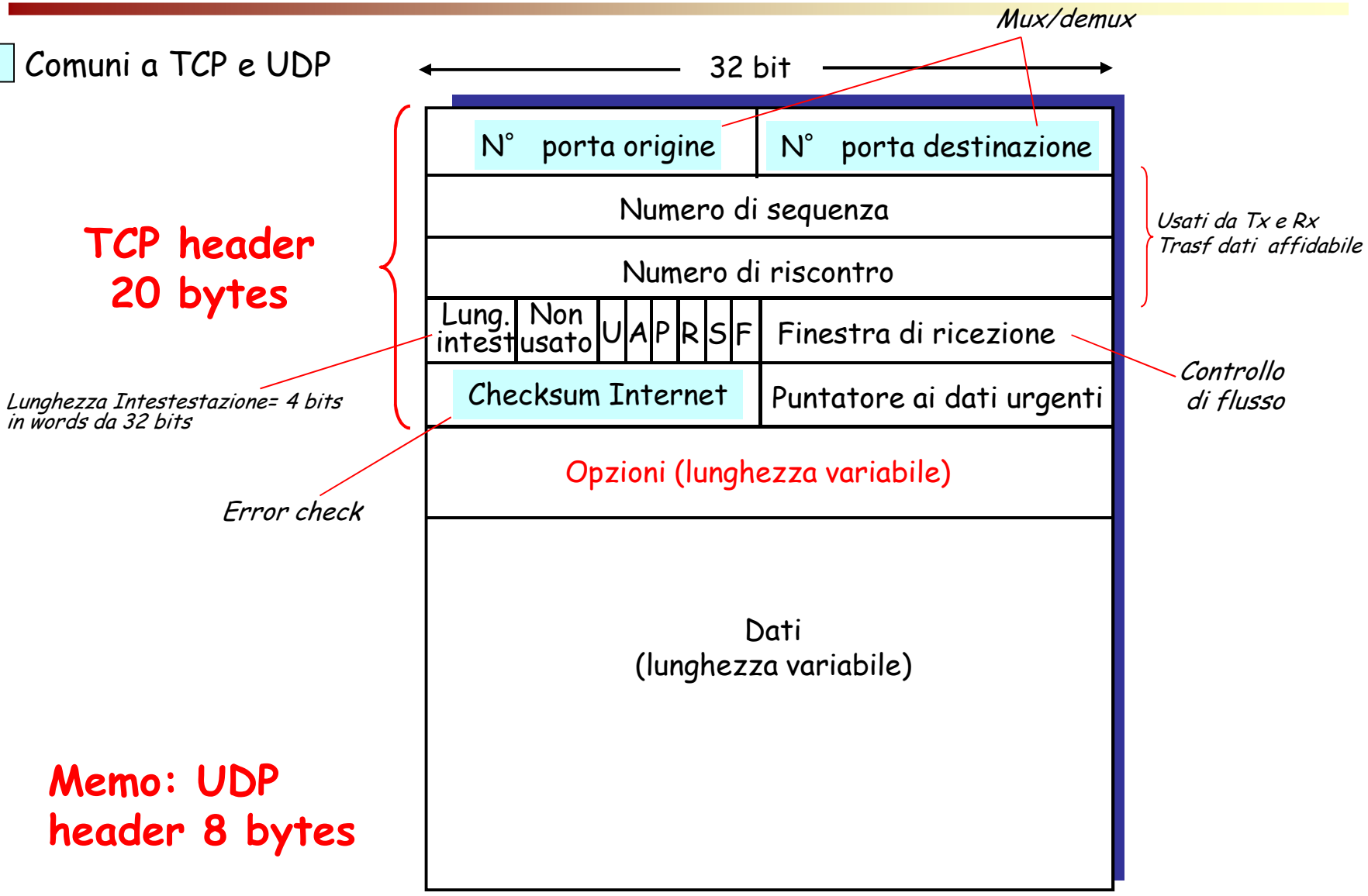

Understanding MSS & MTU
14 bytes
4 bytes
Datalink Header
IP Header
TCP Header
DATA
FCS
MSS
18 bytes
MTU
This diagram clearly shows the Maximum Segment Size and Maximum Transfer Unit.
Regardless of the MTU, there is an overhead of 18 bytes, added by the Datalink Layer

Segmenti TCP

TCP associa ad ogni porzione di dati utente un header TCP formando i segmenti TCP

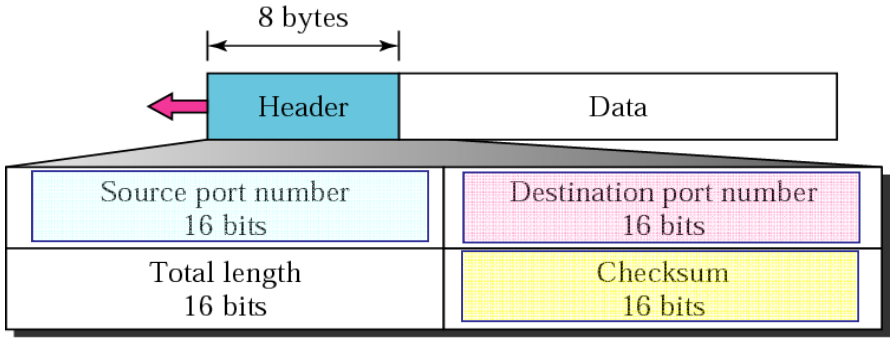


Struttura dei segmenti TCP

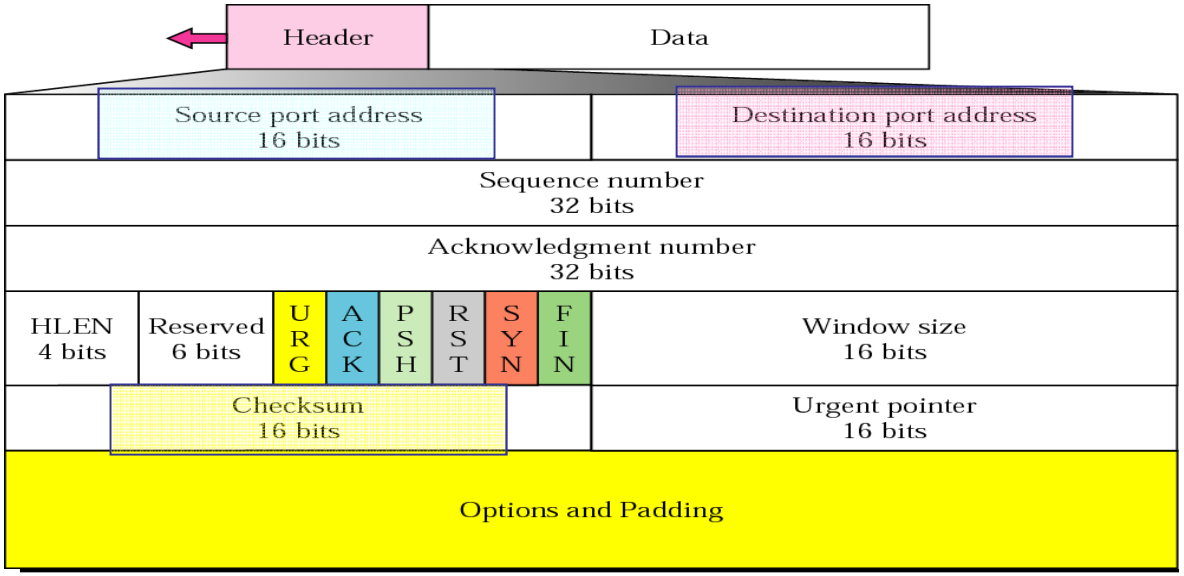


Recall: UDP vs TCP

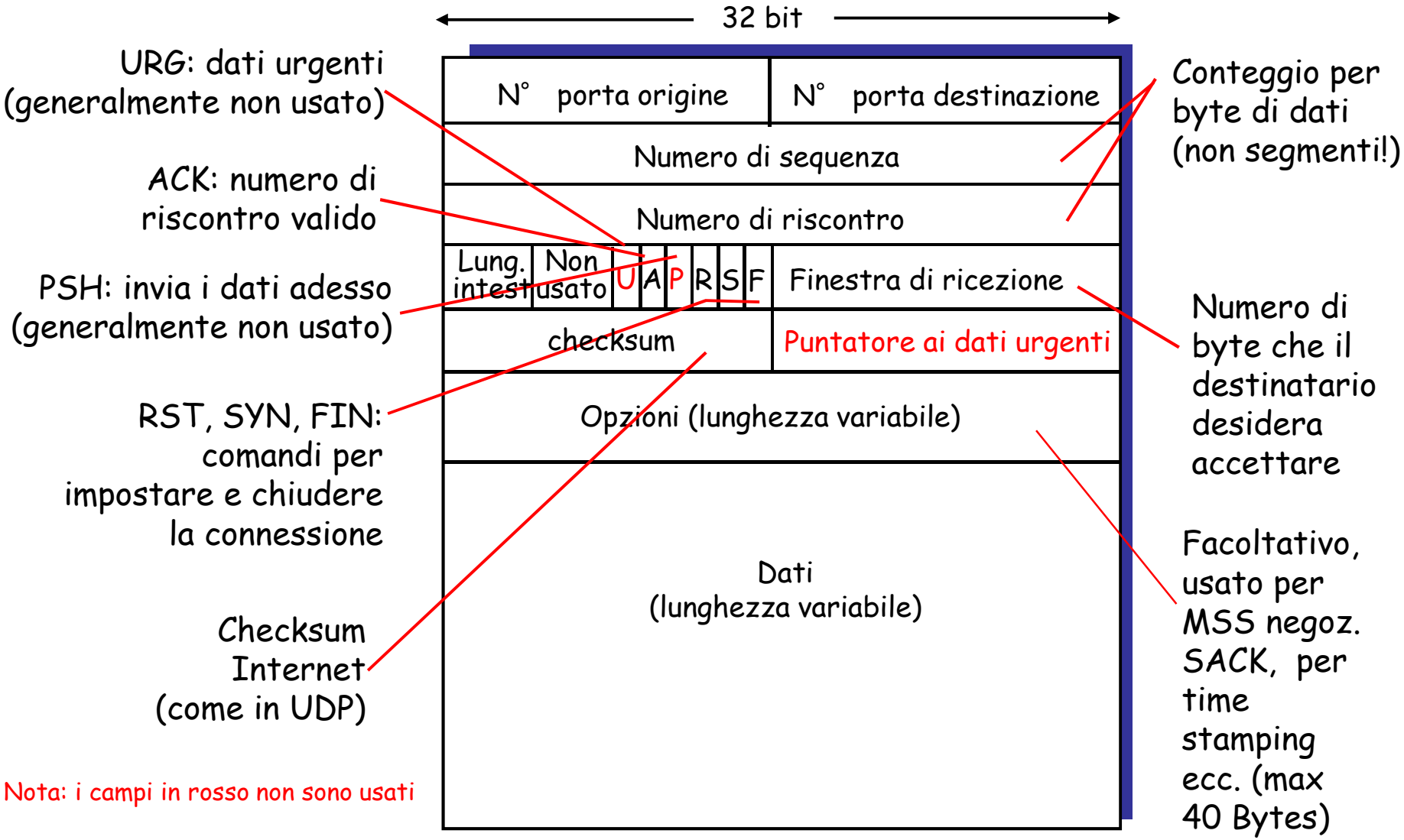
UDP header
8 bytes



TCP header
20 bytes

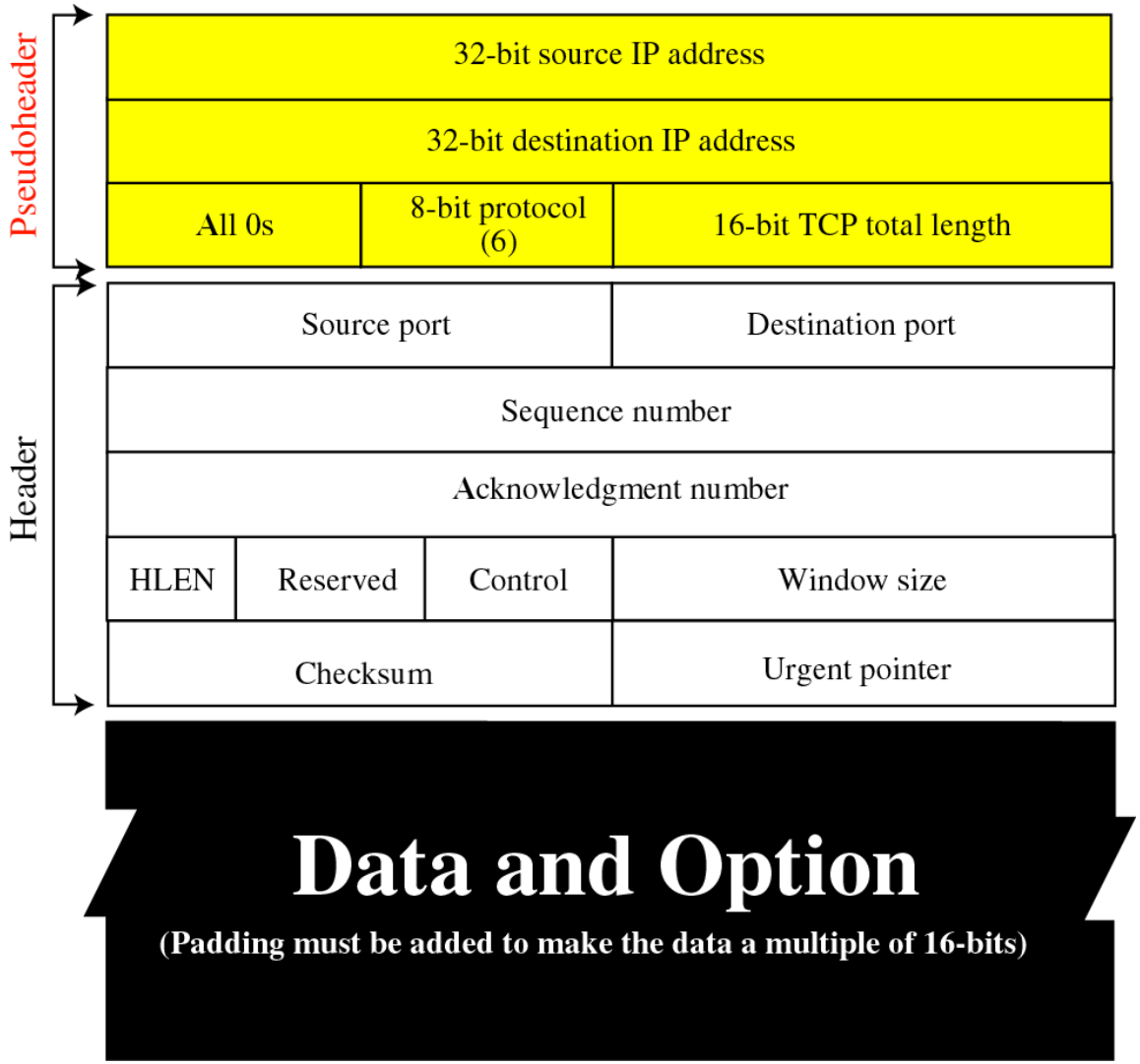


Struttura dei segmenti TCP



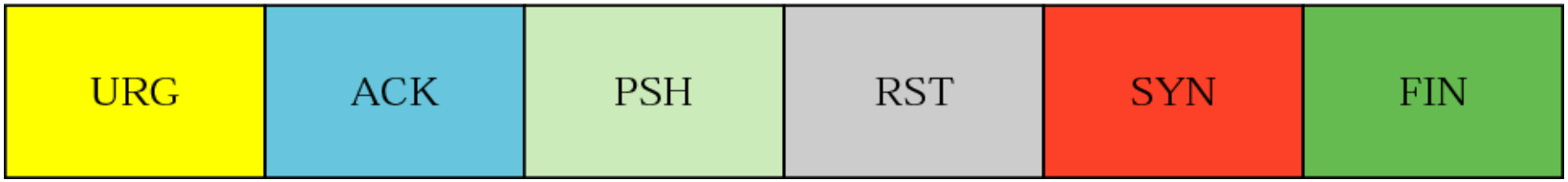
Nota: i campi in rosso non sono usati

Pseudo header di un segmentoTCP



Flags di controllo

URG: Urgent pointer is valid
ACK: Acknowledgment is valid
PSH: Request for push
RST: Reset the connection
SYN: Synchronize sequence numbers
FIN: Terminate the connection



Flag	*Description*
URG	The value of the urgent pointer field is valid
ACK	The value of the acknowledgment field is valid
PSH	Push the data
RST	The connection must be reset
SYN	Synchronize sequence numbers during connection
FIN	Terminate the connection

Encapsulation e decapsulation

- **Il segmento TCP viene poi incapsulato in un datagramma IP, a sua volta incapsulato in una trama a livello data-link (per esempio Ethernet)**

Numeri di sequenza e ACK di TCP

Numeri di sequenza:

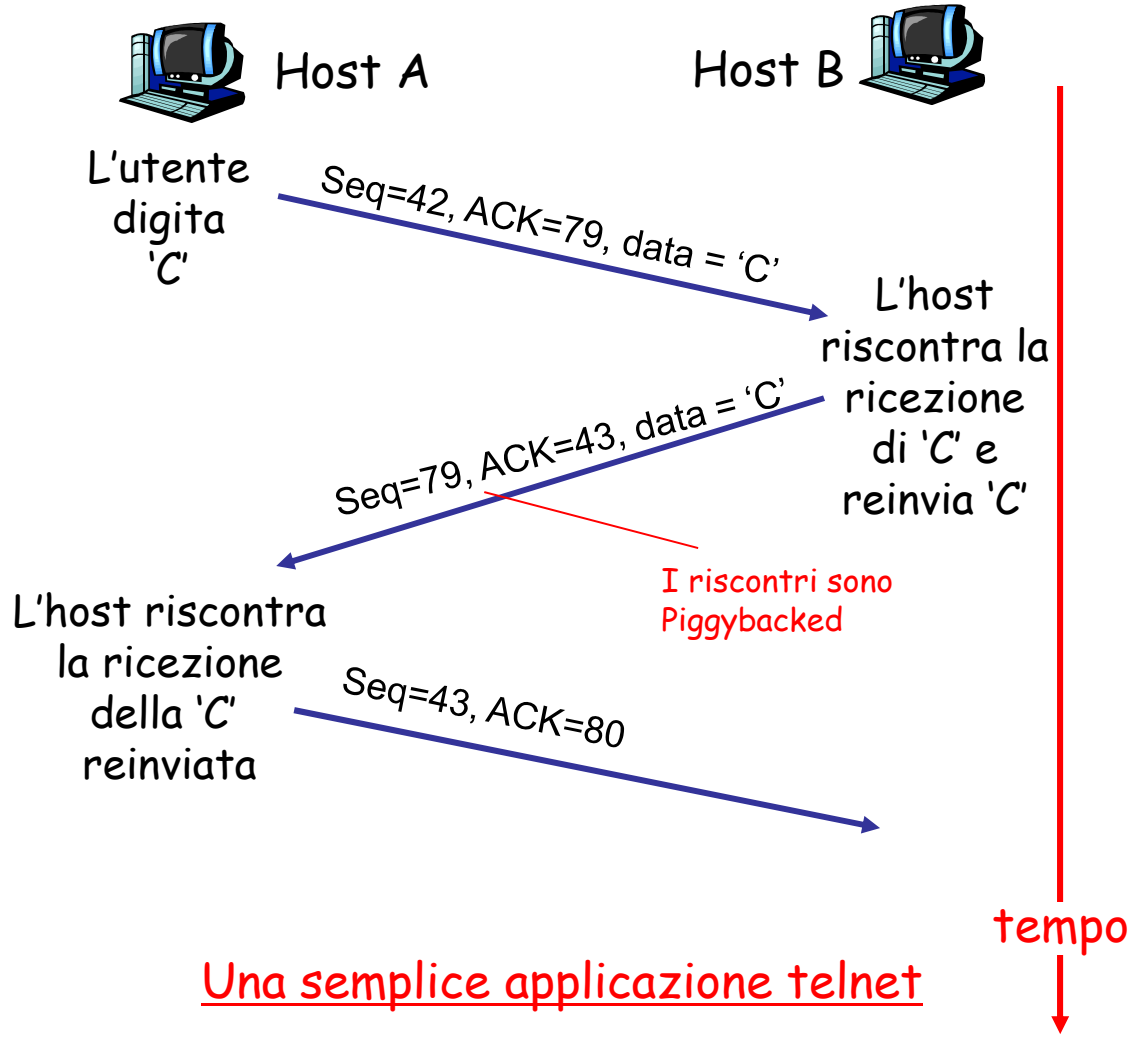
- "numero" del primo byte del segmento nel flusso di byte

ACK:

- numero di sequenza del prossimo byte atteso dall'altro lato
- ACK cumulativo

D: come gestisce il destinatario i segmenti fuori sequenza?

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



Una semplice applicazione telnet

The TELNET Protocol

Reference: RFC 854

Luigi Vetrano

TELNET vs. `telnet`

- **TELNET is a *protocol* that provides "a general, bi-directional, eight-bit byte oriented communications facility".**
- **`telnet` is a *program* that supports the TELNET protocol over TCP.**
- **Many application protocols are built upon the TELNET protocol.**

Command Structure

- All TELNET commands and data flow through the same TCP connection.
- Commands start with a special character called the Interpret as Command *escape* character (IAC).
- The IAC code is 255.
- If a 255 is sent as data - it must be followed by another 255.

Playing with TELNET

- **You can use the `telnet` program to play with the TELNET protocol.**

- **`telnet` is a *generic* TCP client.**
  - Sends whatever you type to the TCP socket.
  - Prints whatever comes back through the TCP socket.
  - Useful for testing TCP servers (ASCII based protocols).

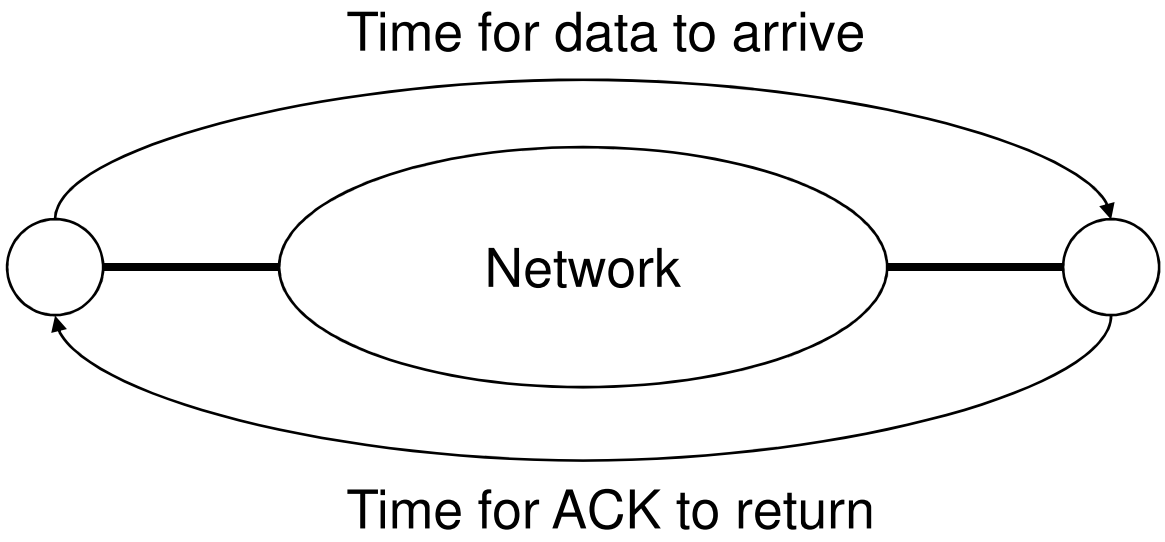
Ritrasmissione in TCP

- Quando un segmento rimane non riscontrato (unacknowledged) per un certo periodo di tempo, il TCP assume che sia andato perso e lo ritrasmette.

- TCP cerca di calcolare il “round trip time” (RTT) richiesto da un segmento e dal suo ACK

- Dalla conoscenza di RTT, TCP può valutare quanto tempo aspettare prima di andare in timeout e di ritrasmettere

Round Trip Time (RTT)



RTT = Time for packet to arrive at destination
+
Time for ACK to return from destination

TCP: tempo di andata e ritorno e timeout

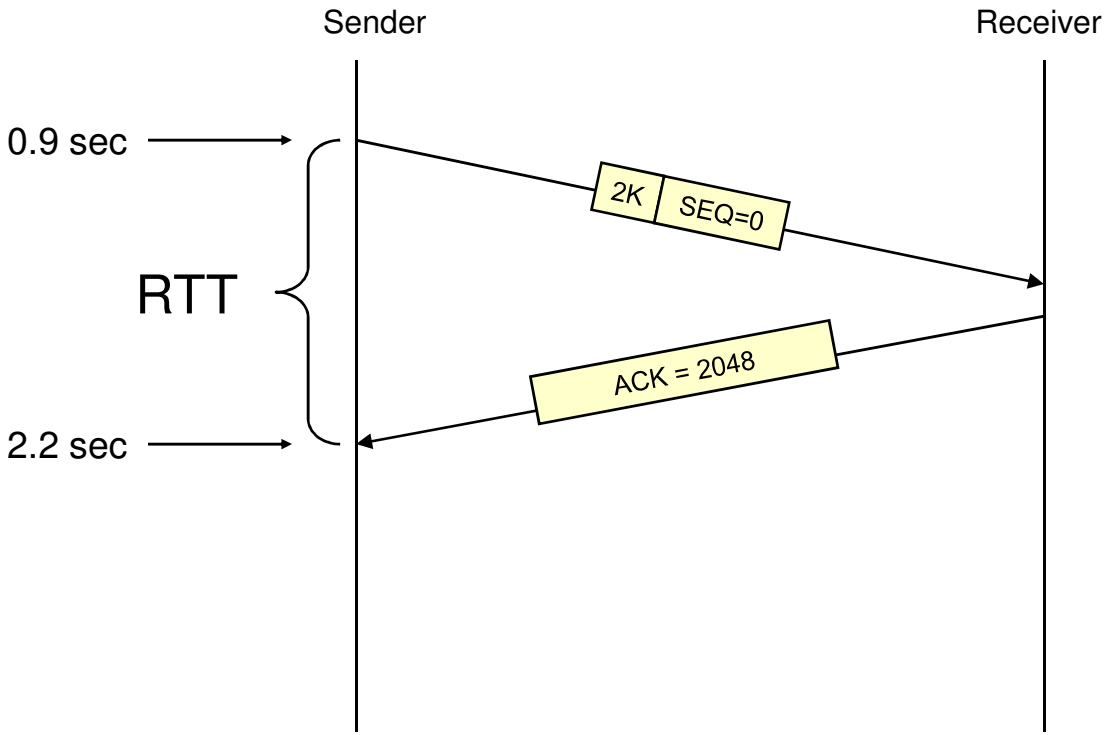
D: come impostare il valore del timeout di TCP?

- **Più grande di RTT**
  - ma RTT varia
- **Troppo piccolo: timeout prematuro**
  - ritrasmissioni non necessarie
- **Troppo grande: reazione lenta alla perdita dei segmenti**

D: come stimare RTT?

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

Calcolo di RTT



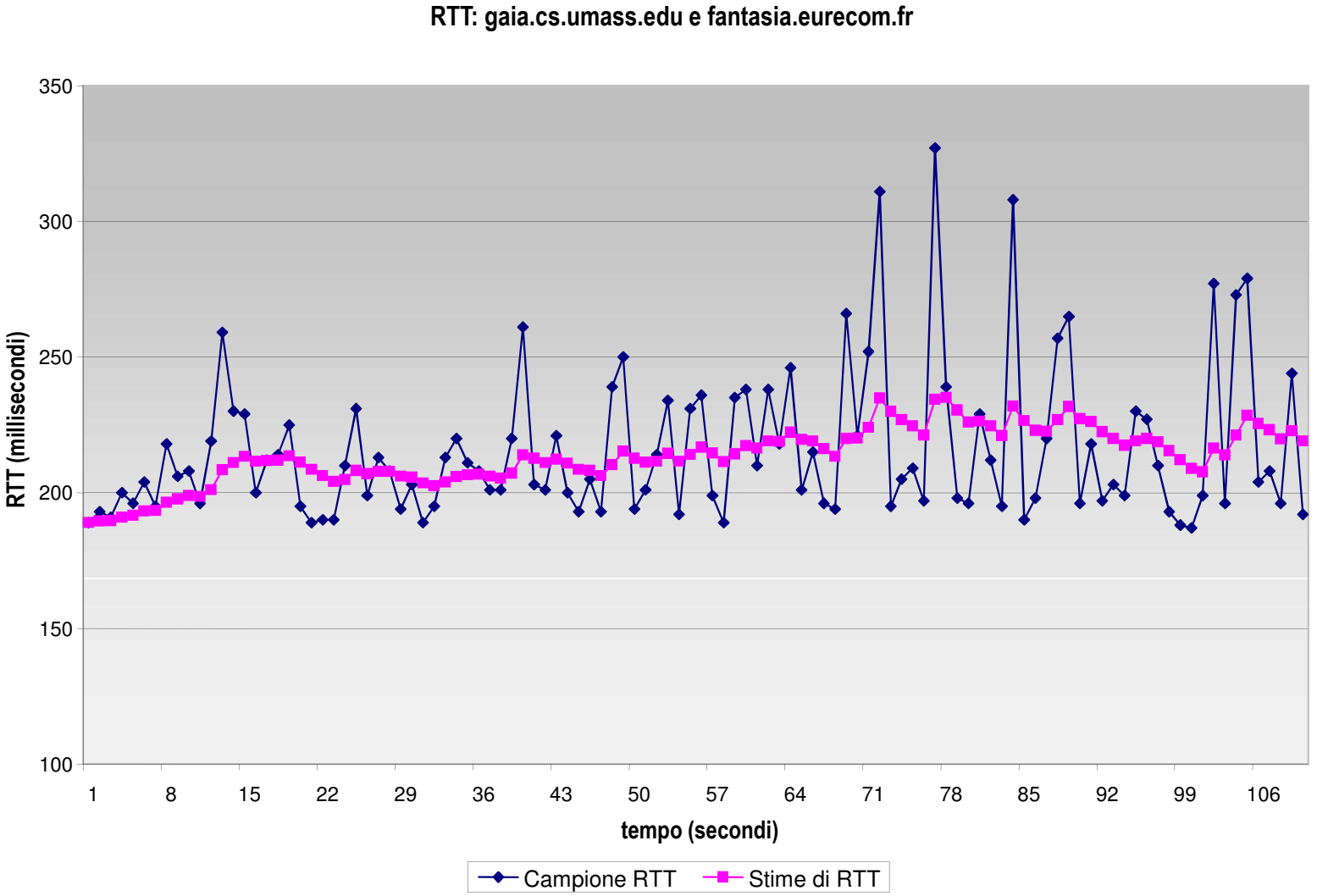
RTT = 2.2 sec - 0.9 sec. = 1.3 sec

TCP: tempo di andata e ritorno e timeout

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

- **Media mobile esponenziale ponderata (EWMA)**
- **L'influenza dei vecchi campioni decresce esponenzialmente**
- **Valore tipico: $\alpha = 0{,}125$**

Esempio di stima di RTT:



TCP: tempo di andata e ritorno e timeout

Impostazione del timeout

- `EstimtedRTT` **più un "margine di sicurezza"**
  - grande variazione di **`EstimatedRTT`** -> margine di sicurezza maggiore
- **Stimare innanzitutto di quanto SampleRTT si discosta da EstimatedRTT:**

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

(tipicamente, $\beta = 0,25$)

Poi impostare l'intervallo di timeout:

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

DevRTT è un EWMA della differenza tra **SampleRTT e EstimatedRTT**

Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

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

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

TCP: trasferimento dati affidabile

- **TCP crea un servizio di trasferimento dati affidabile sul servizio inaffidabile di IP**
- **Pipeline dei segmenti**
- **ACK cumulativi**
- **TCP usa un solo timer di ritrasmissione (RFC 2988)**
- **Le ritrasmissioni sono avviate da:**
  - eventi di timeout
  - ACK duplicati
- **Inizialmente consideriamo un mittente TCP semplificato:**
  - ignoriamo gli ACK duplicati
  - ignoriamo il controllo di flusso e il controllo di congestione

TCP: eventi del mittente

Dati ricevuti dall'applicazione:

- Crea un segmento con il numero di sequenza
- Il numero di sequenza è il numero del primo byte del segmento nel flusso di byte
- Avvia il timer, se non è già in funzione (pensate al timer come se fosse associato al più vecchio segmento non riscontrato)
- Intervallo di scadenza: `TimeOutInterval`

Timeout:

- Ritrasmette il segmento che ha causato il timeout
- Riavvia il timer

ACK ricevuti:

- Se riscontra segmenti precedentemente non riscontrati
  - aggiorna ciò che è stato completamente riscontrato
  - avvia il timer se ci sono altri segmenti da completare

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (per_sempre) {
  switch(evento)

  evento_1: i dati ricevuti dall'applicazione superiore
      creano il segmento TCP con numero di sequenza NextSeqNum
      if (il timer attualmente non funziona)
          avvia il timer
      passa il segmento a IP
      NextSeqNum = NextSeqNum + lunghezza(dati)
      break;
  evento_2: timeout del timer
      ritrasmetti il segmento non ancora riscontrato con
          il più piccolo numero di sequenza
      avvia il timer
      break;
  evento_3: ACK ricevuto, con valore del campo ACK pari a y
      if (y > SendBase) {
          SendBase = y
          if (esistono attualmente segmenti non ancora riscontrati)
              avvia il timer
          }
      break;
} /* fine del loop */
```

Mittente TCP (semplificato)

Commento:

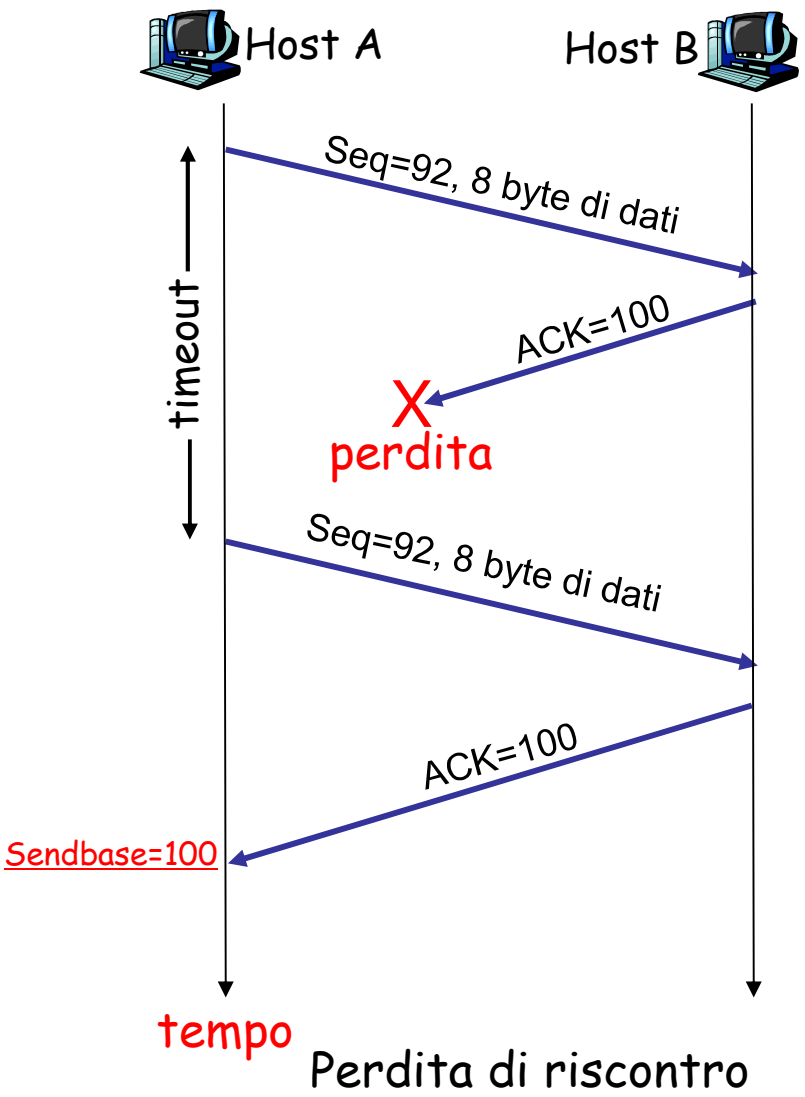
- SendBase-1: ultimo byte cumulativamente riscontrato

Esempio:

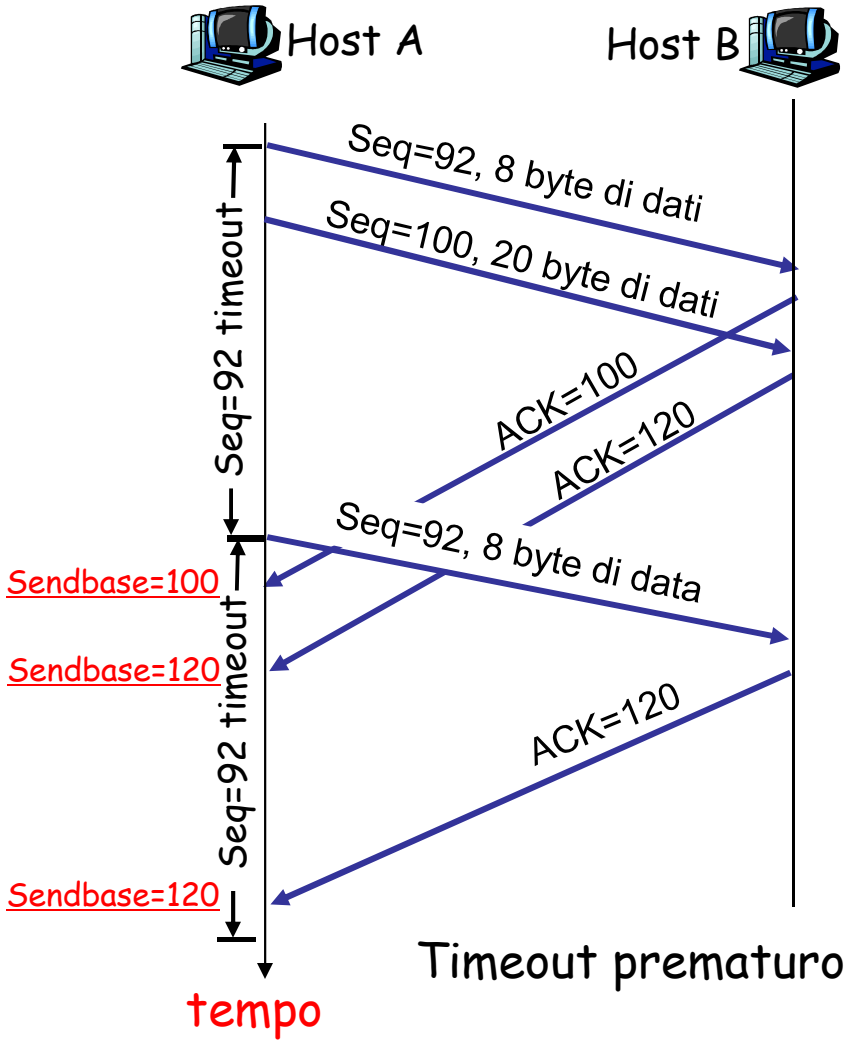
- SendBase-1 = 71;

y = 73, quindi il destinatario vuole 73+ ;
y > SendBase, allora vengono riscontrati tali nuovi dati

TCP: scenari di ritrasmissione

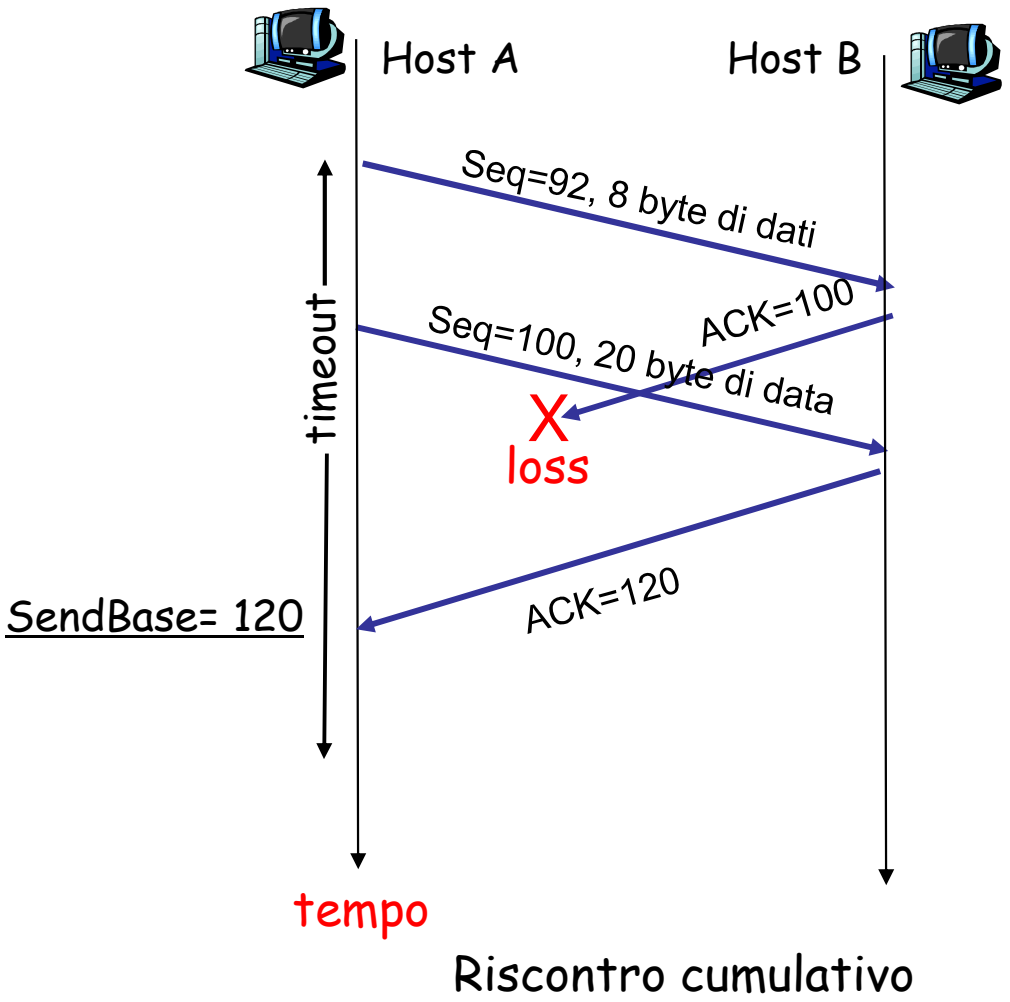


Perdita di riscontro



Timeout prematuro

TCP: scenari di ritrasmissione (altro)



Riscontro cumulativo

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

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

Ritrasmissione rapida

- **Il periodo di timeout spesso è relativamente lungo:**
  - lungo ritardo prima di ritrasmettere il pacchetto perduto.
- **Rileva i segmenti perduti tramite gli ACK duplicati.**
  - Il mittente spesso invia molti segmenti.
  - Se un segmento viene smarrito, è probabile che ci saranno molti ACK duplicati.
- **Se il mittente riceve 3 ACK per lo stesso dato, suppone che il segmento che segue il dato riscontrato è andato perduto:**
  - ritrasmissione rapida: rispedisce il segmento prima che scada il timer.

Algoritmo della ritrasmissione rapida:

```
evento: ACK ricevuto, con valore del campo ACK pari a y
            if (y > SendBase) {
                SendBase = y
                if (esistono attualmente segmenti non ancora riscontrati)
                    avvia il timer
              }
            else {
                incrementa il numero di ACK duplicati ricevuti per y
                if (numero di ACK duplicati ricevuti per y = 3) {
                    rispedisci il segmento con numero di sequenza y
                 }
```

un ACK duplicato per un segmento già riscontrato

ritrasmissione rapida

Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

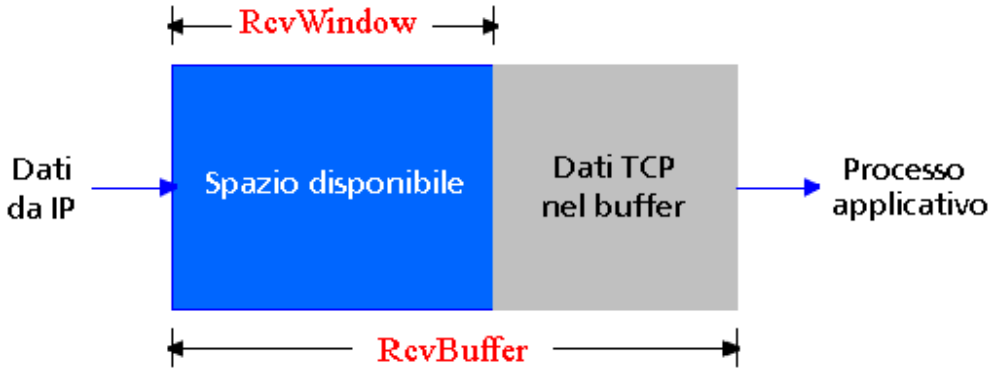
- struttura dei segmenti
- trasferimento dati affidabile
- controllo di flusso
- gestione della connessione

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

TCP: controllo di flusso

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



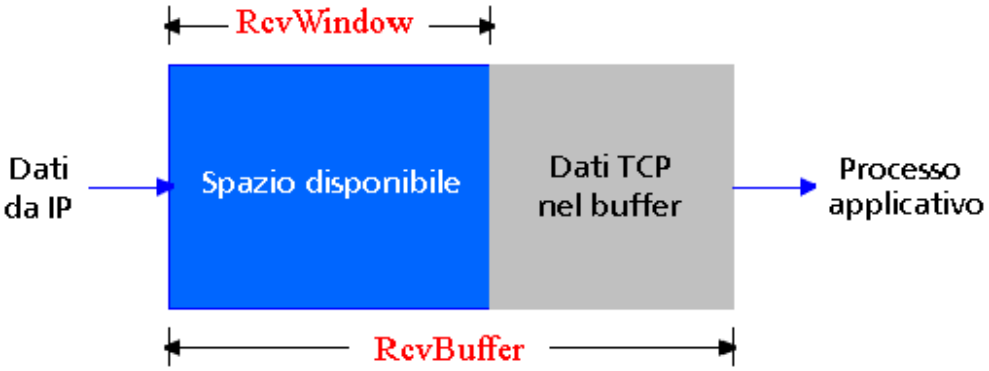
- **Il processo applicativo potrebbe essere rallentato dalla lettura nel buffer**

Controllo di flusso

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

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

TCP: funzionamento del controllo di flusso



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

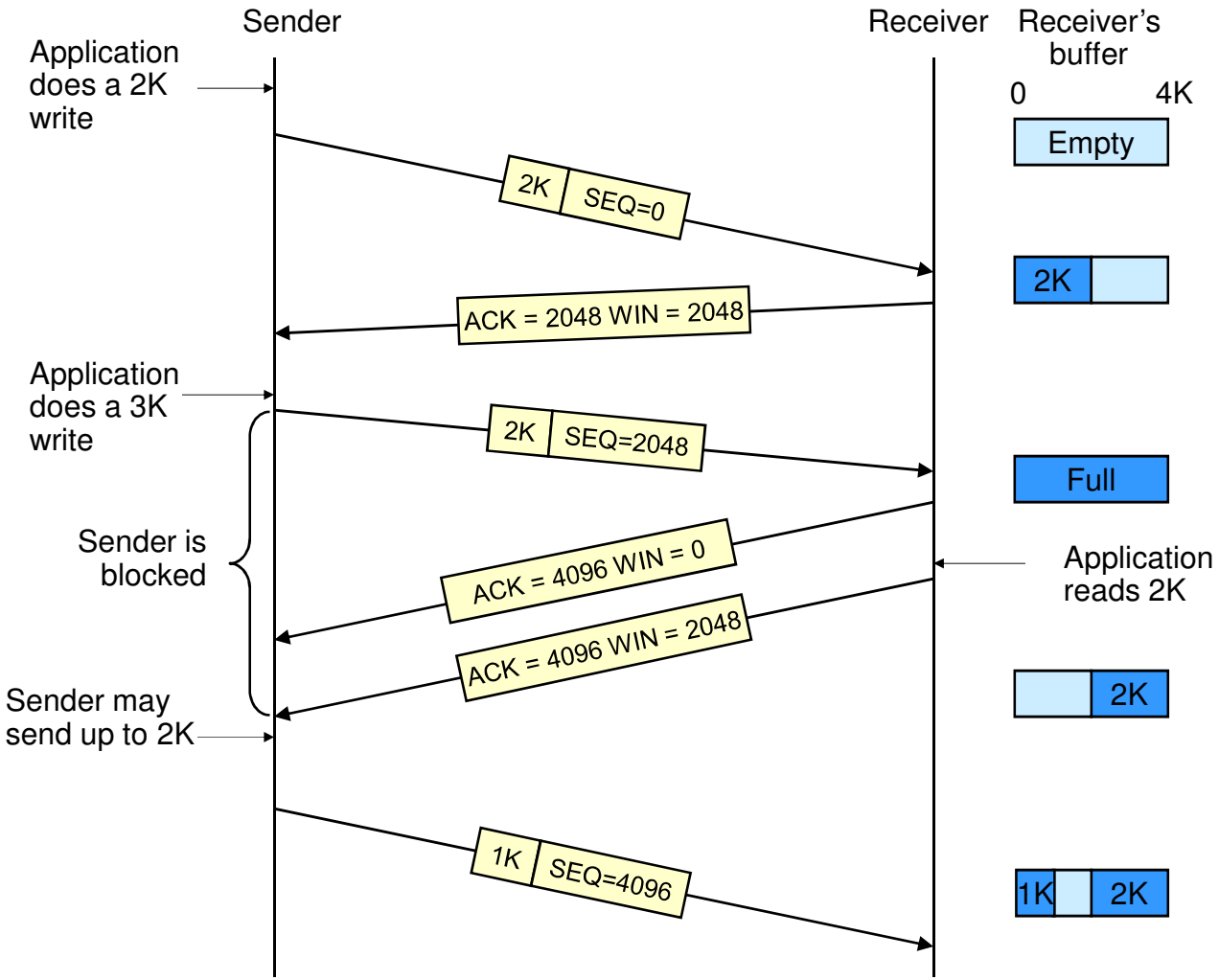
- **Spazio disponibile nel buffer**

= `RcvWindow`

= `RcvBuffer – [LastByteRcvd – LastByteRead]`

- **Il mittente comunica lo spazio disponibile includendo il valore di** `RcvWindow` **nei segmenti**
- **Il mittente limita i dati non riscontrati a** `RcvWindow`
  - garantisce che il buffer di ricezione non vada in overflow

TCP Flow Control



Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

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

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

TCP e connessione

- **Un protocollo di trasporto connection-oriented stabilisce un virtual path tra source e destination.**
- **Tutti i segmenti appartenenti ad un messaggio sono spediti su questo virtual path.**
- **Una trasmissione connection-oriented richiede tre fasi:**
  1. connection establishment,
  2. data transfer, e
  3. connection termination.

Gestione della connessione TCP

Ricordiamo: mittente e destinatario TCP stabiliscono una "connessione" prima di scambiarsi i segmenti di dati

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

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

- ***server:* contattato dal client**

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

Handshake a tre vie:

Passo 1: il client invia un segmento SYN al server

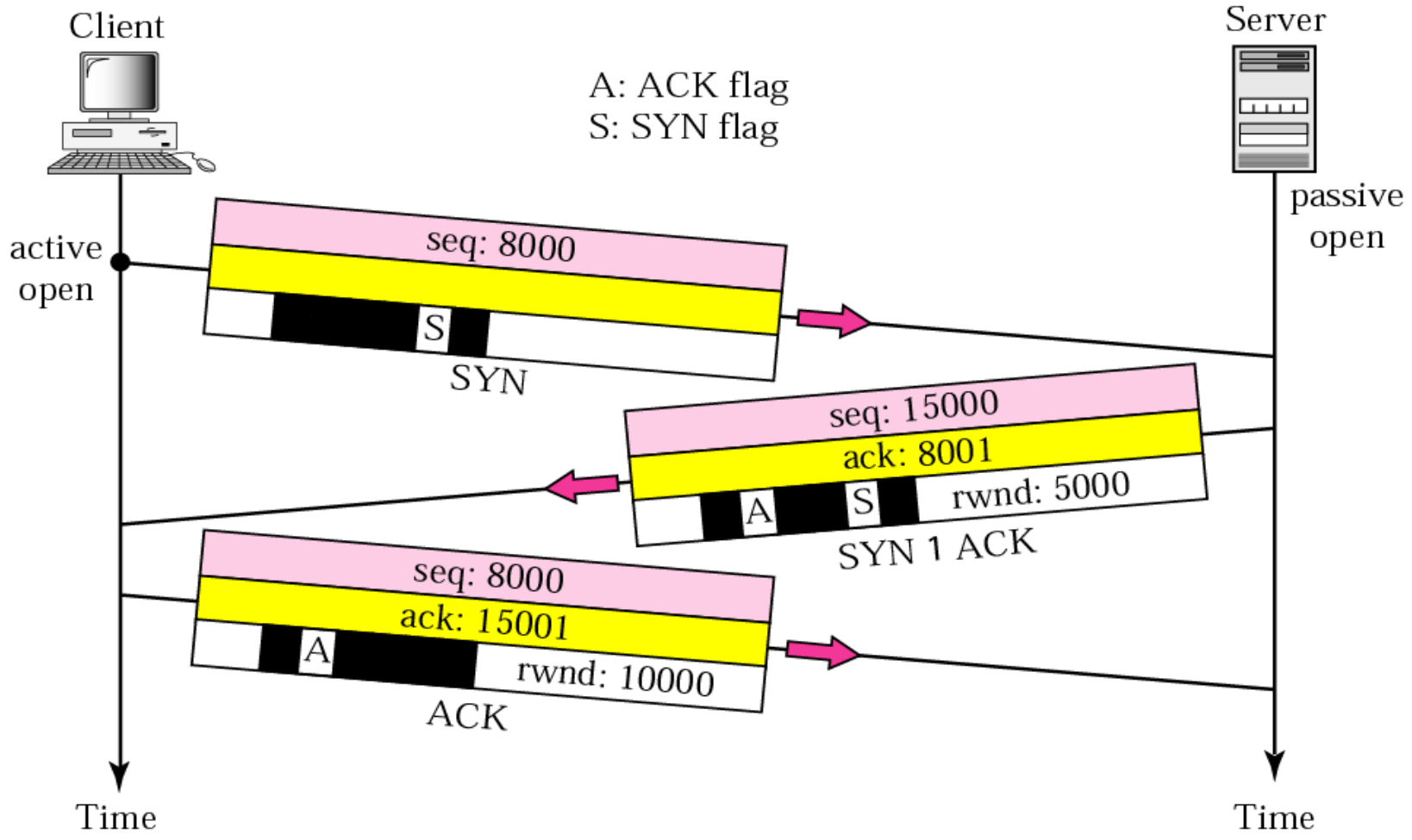
- specifica il numero di sequenza iniziale
- nessun dato

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

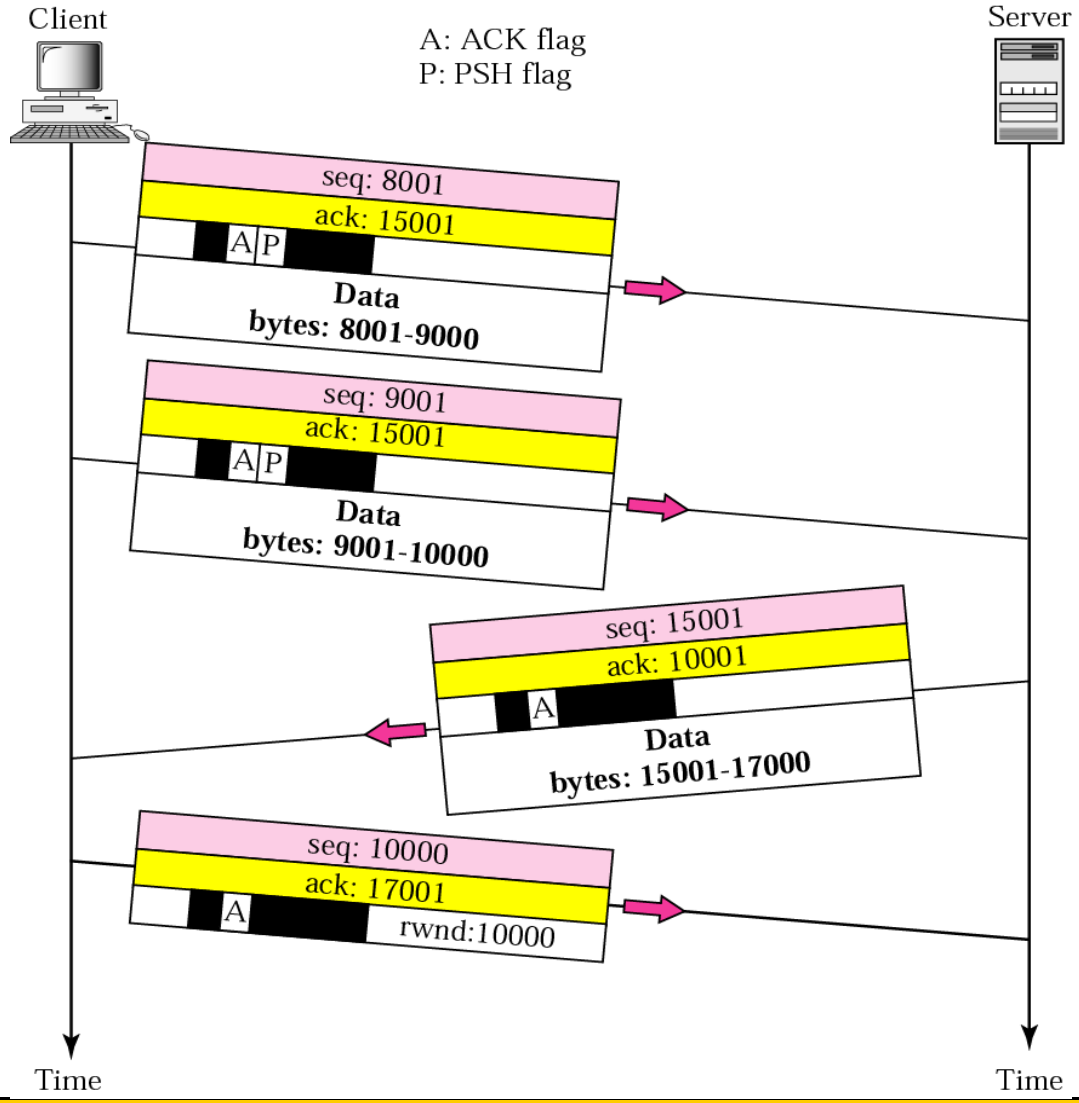
- il server alloca i buffer
- specifica il numero di sequenza iniziale del server

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

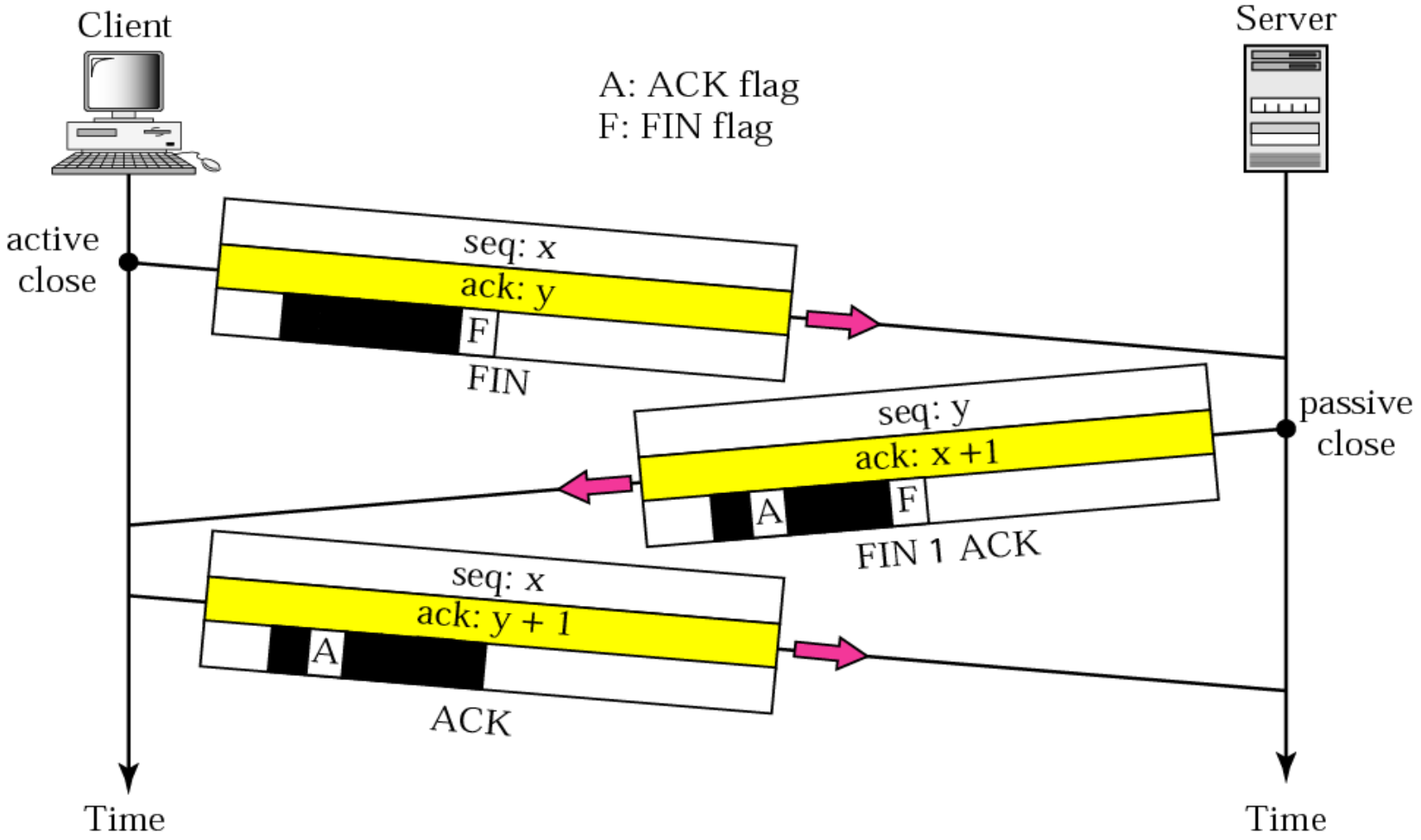
Connection establishment: three-way handshaking



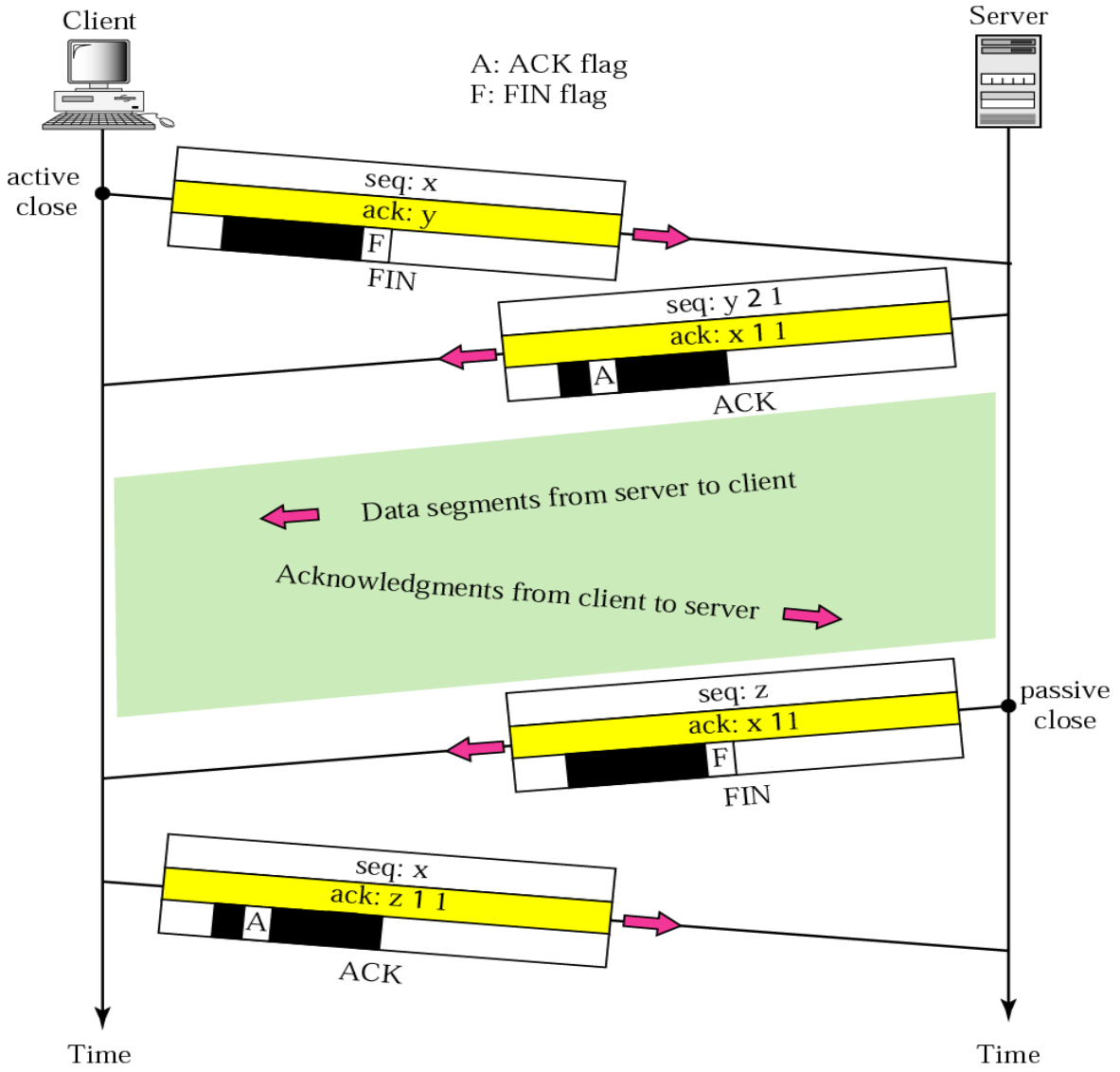
Data transfer



Terminazione di una Connessione (three-way handshaking)



Half-close


Client
Server
A: ACK flag
F: FIN flag
active close
seq: x
ack: y
F
FIN
seq: y 2 1
ack: x 1 1
A
ACK
Data segments from server to client
Acknowledgments from client to server
seq: z
ack: x 11
F
FIN
passive close
seq: x
ack: z 1 1
A
ACK
Time
Time

Gestione della connessione TCP (continua)

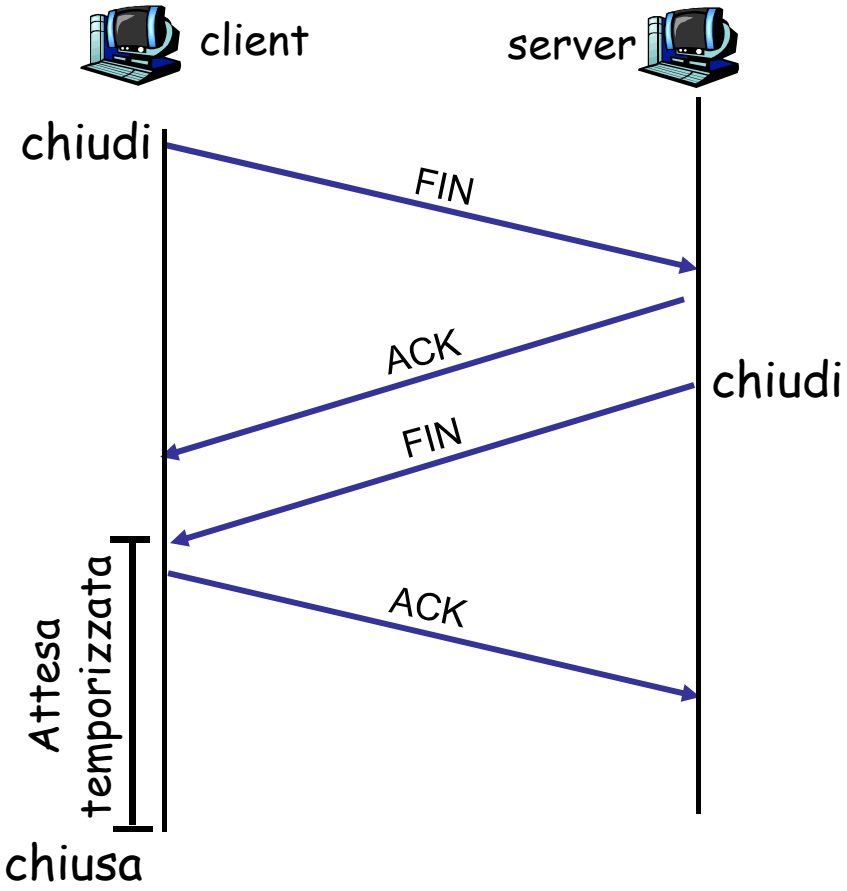
Chiudere una connessione:

Il client chiude la socket:

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

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

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



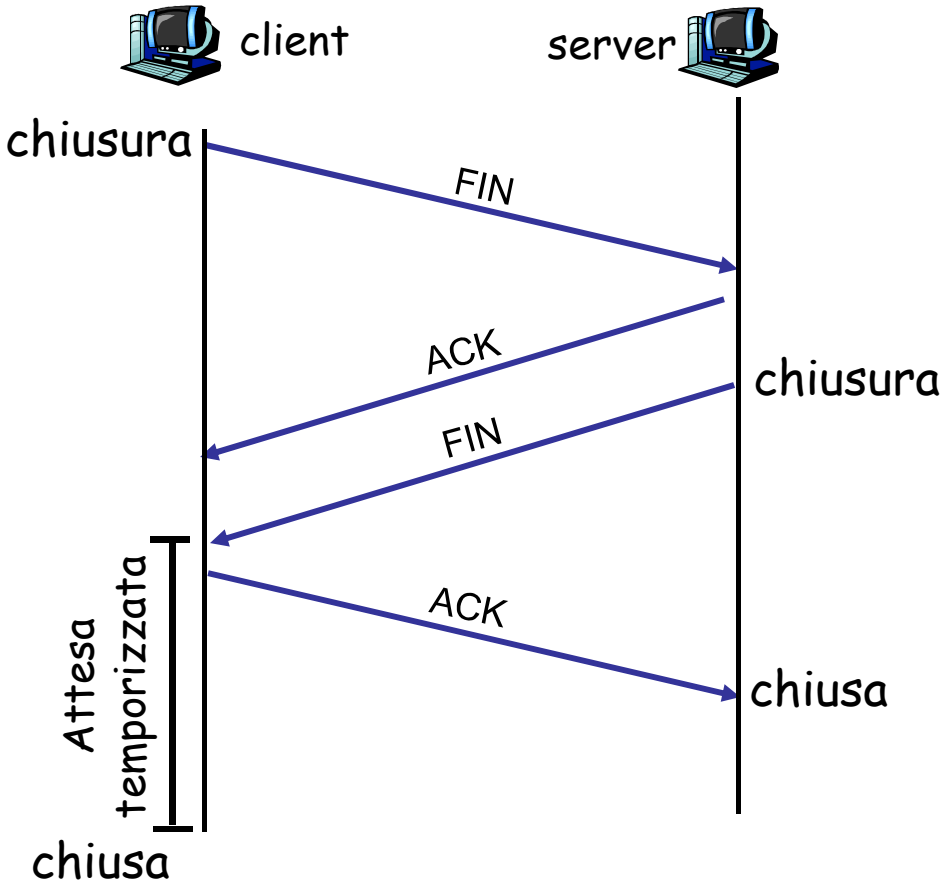
Gestione della connessione TCP (continua)

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

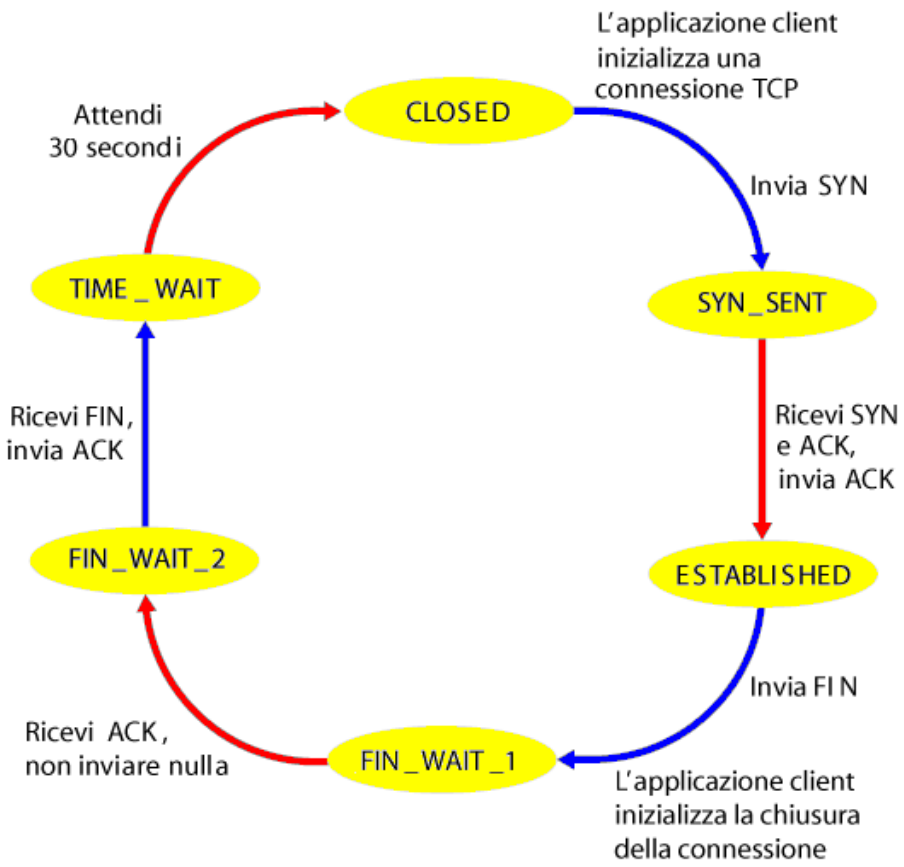
- inizia l'attesa temporizzata - risponde con un ACK ai FIN che riceve

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

Nota: con una piccola modifica può gestire segmenti FIN simultanei.

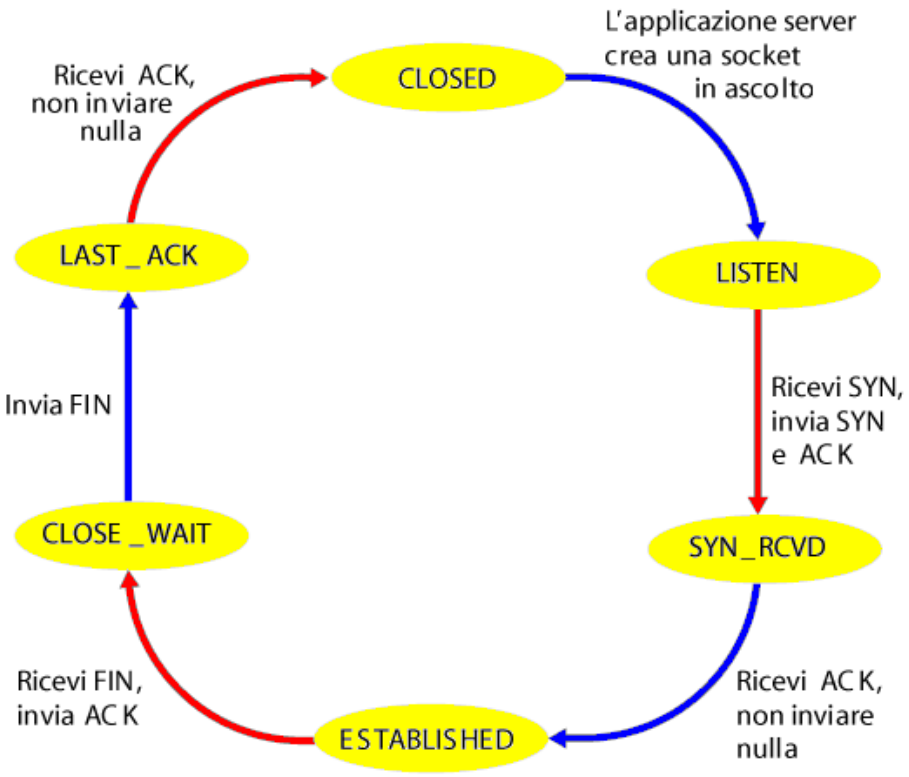


Gestione della connessione TCP (continua)



Sequenza di stati visitati da un client TCP

Sequenza di stati visitati dal TCP sul lato server



Capitolo 3: Livello di trasporto

3.1 Servizi a livello di trasporto

3.2 Multiplexing e demultiplexing

3.3 Trasporto senza connessione: UDP

3.4 Principi del trasferimento dati affidabile

3.5 Trasporto orientato alla connessione: TCP

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

3.6 Principi sul controllo di congestione

3.7 Controllo di congestione TCP

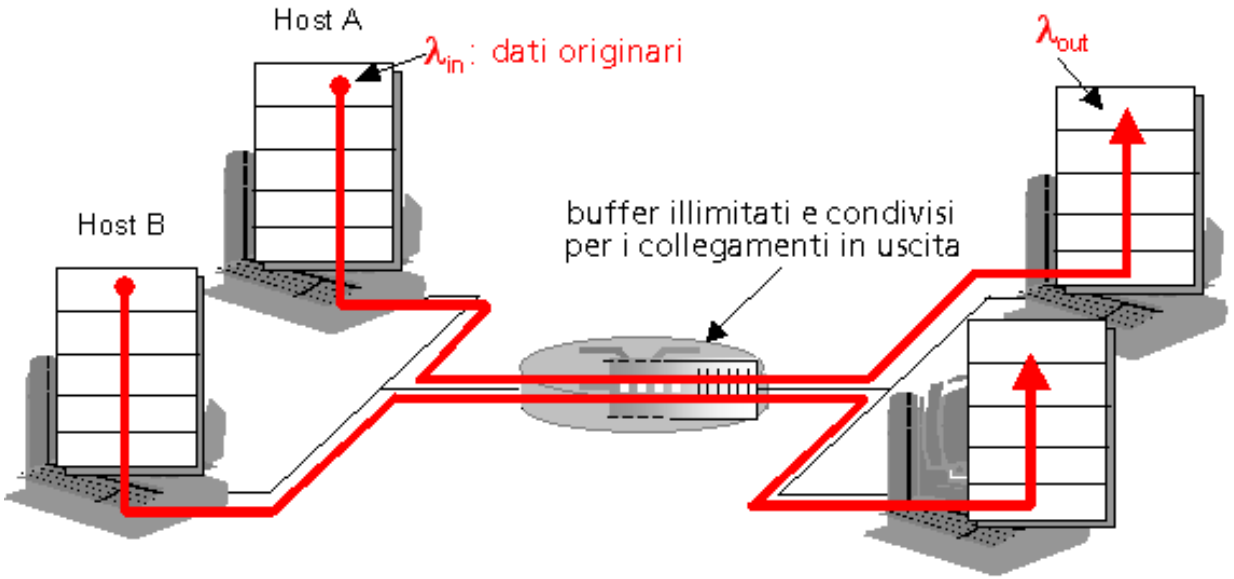
Principi sul controllo di congestione

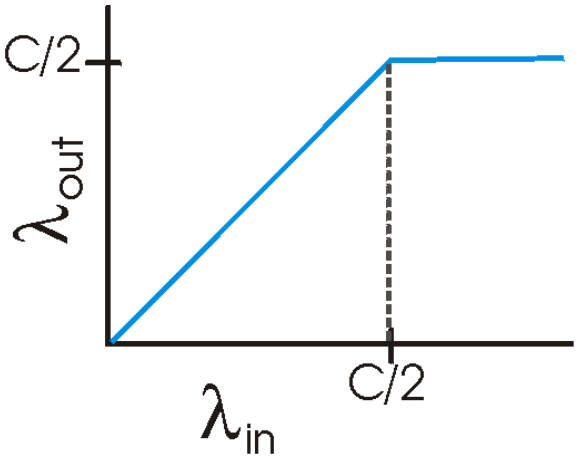
Congestione:

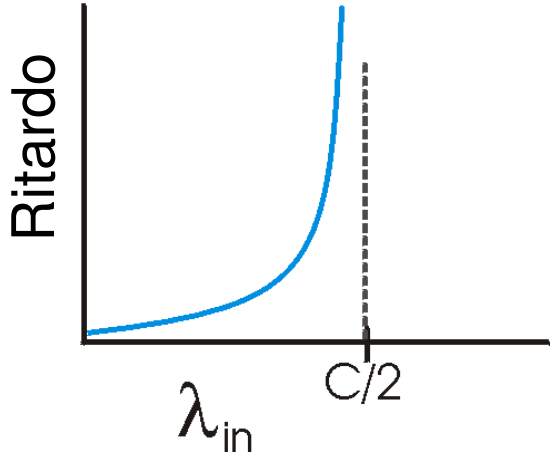
- **informalmente: "troppe sorgenti trasmettono troppi dati, a una velocità talmente elevata che la *rete* non è in grado di gestirli"**
- **differisce dal controllo di flusso!**
- **manifestazioni:**
  - pacchetti smarriti (overflow nei buffer dei router)
  - lunghi ritardi (accodamento nei buffer dei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

Cause/costi della congestione: scenario 1

- due mittenti, due destinatari
- un router con buffer illimitati
- nessuna ritrasmissione



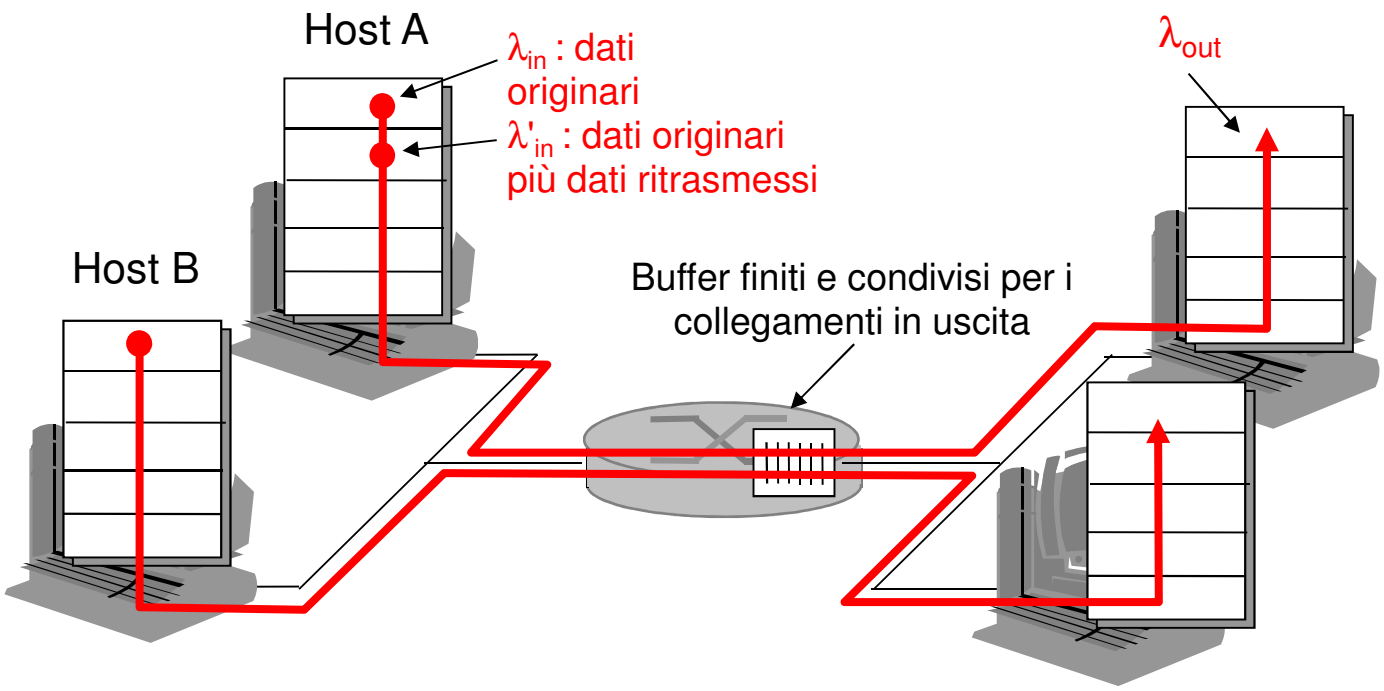




- grandi ritardi se congestionati
- throughput massimo

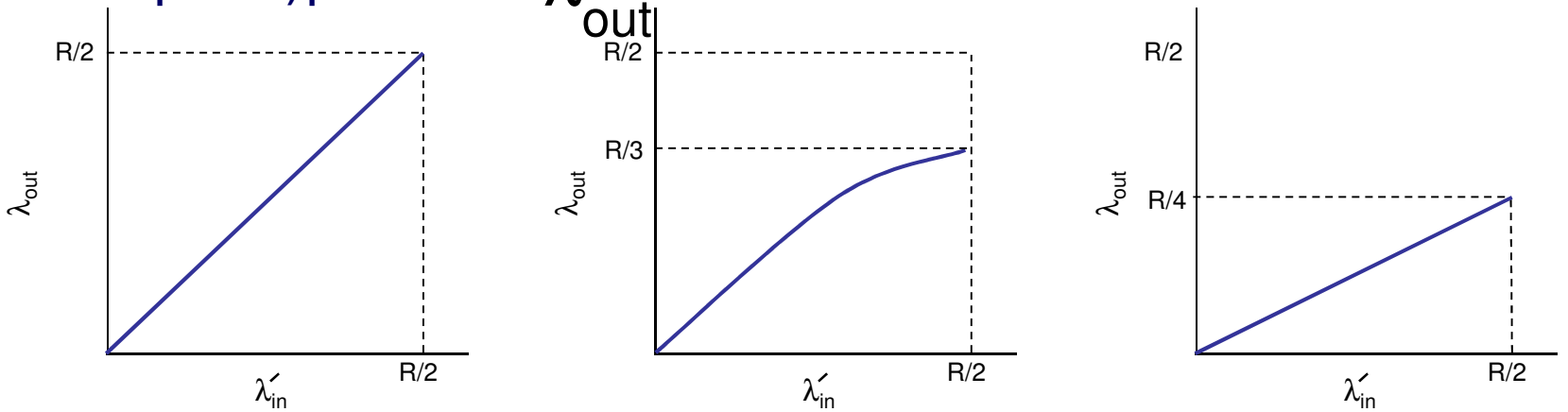
Cause/costi della congestione: scenario 2

- un router, buffer *finiti*
- il mittente ritrasmette il pacchetto perduto



Cause/costi della congestione: scenario 2

- **Sempre:** $\lambda_{in} = \lambda_{out}$ **(goodput)**
- **Ritrasmissione "perfetta" solo quando la perdita:** $\lambda'_{in} > \lambda_{out}$
- **La ritrasmissione del pacchetto ritardato (non perduto) rende** λ'_{in} **più grande (rispetto al caso perfetto) per lo stesso** λ_{out}



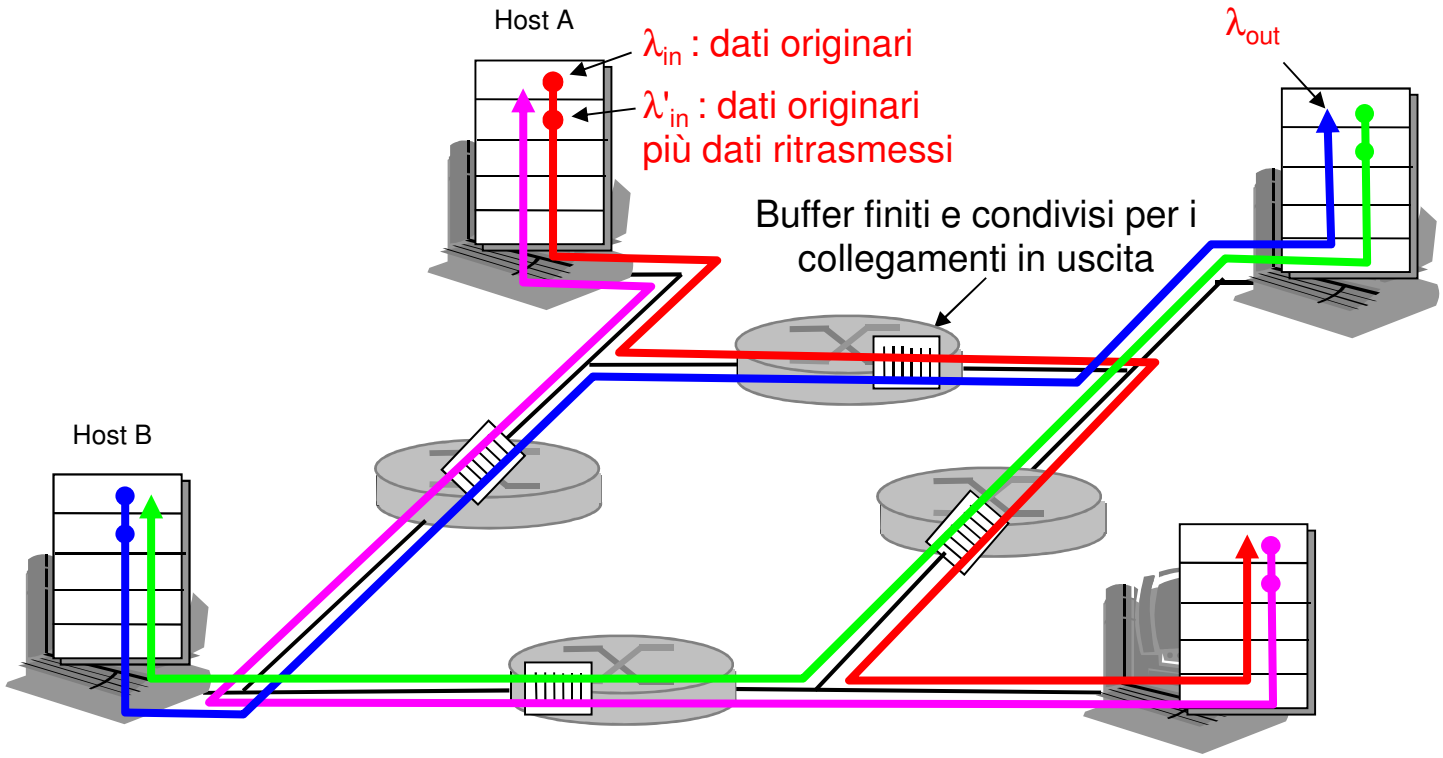
"Costi" della congestione:

- **Più lavoro (ritrasmissioni) per un dato "goodput"**
- **Ritrasmissioni non necessarie: il collegamento trasporta più copie del pacchetto**

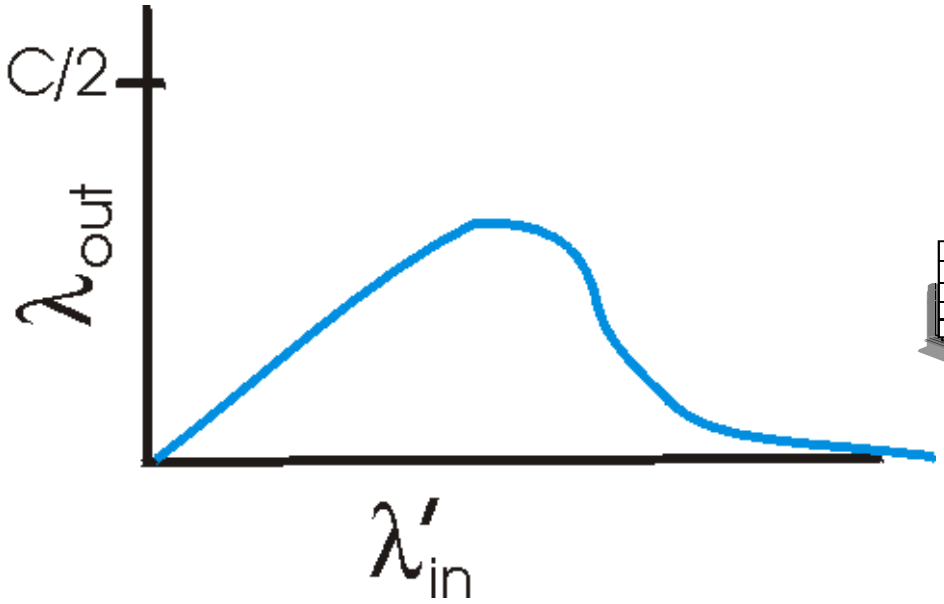
Cause/costi della congestione: scenario 3

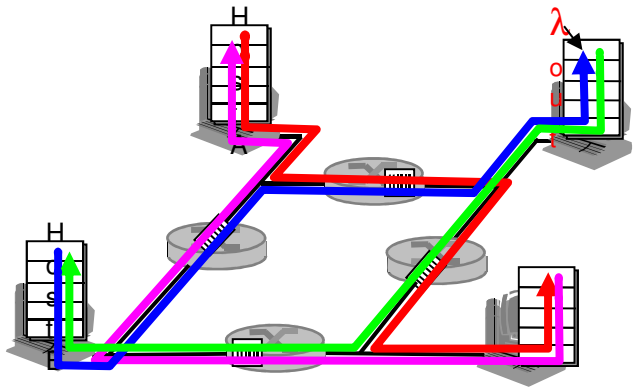
- Quattro mittenti
- Percorsi multihop
- timeout/ritrasmissione

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



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!**

Approcci al controllo della congestione

I due principali approcci al controllo della congestione:

Controllo di congestione punto-punto:

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

Controllo di congestione assistito dalla rete:

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

ATM

- **ATM si basa sulla commutazione di pacchetto orientato al circuito virtuale (VC):**
  - Ogni switch sul percorso origine-destinazione mantiene lo stato del VC in modo da conservare traccia dei singoli mittenti.
  - Questa conoscenza permette di intraprendere azioni di controllo di congestione assistito dalla rete

Esempio : controllo di congestione ATM ABR

Approccio assistito dalla rete

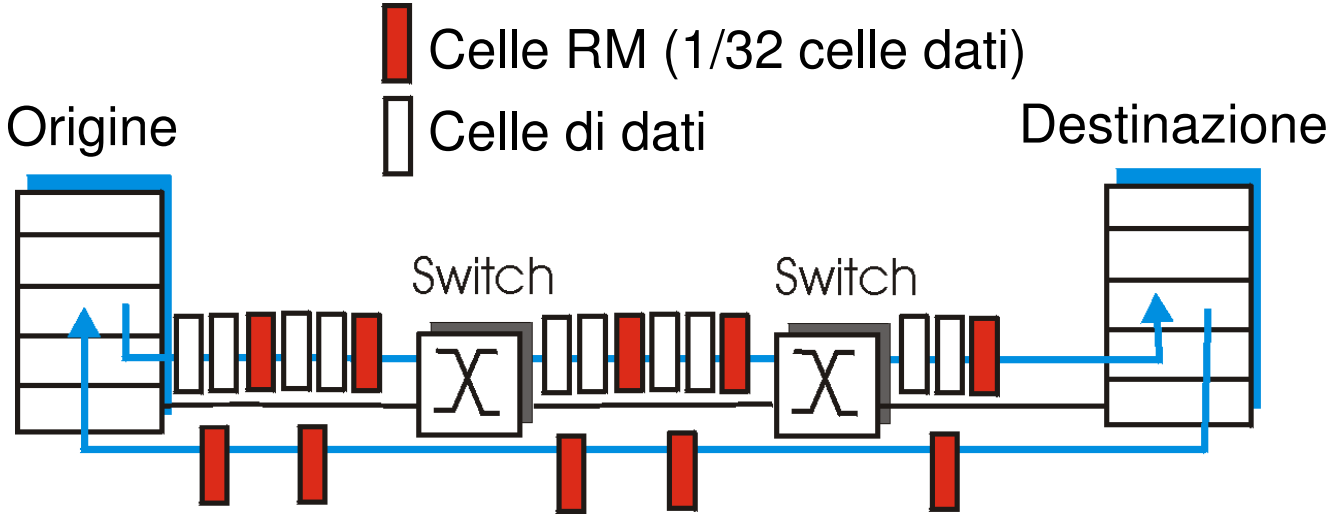
ABR: available bit rate:

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

Celle RM (resource management):

- **inviate dal mittente, inframmezzate alle celle di dati**
- **i bit in una cella RM sono impostati dagli switch ("*assistenza dalla rete*")**
  - bit NI: nessun aumento del tasso trasmissivo (congestione moderata)
  - bit CI: indicazione di congestione (traffico intenso)
- **il destinatario restituisce le celle RM al mittente con i bit intatti**

Esempio : controllo di congestione ATM ABR



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

Capitolo 3: Livello di trasporto

- 3.1 Servizi a livello di trasporto
- 3.2 Multiplexing e demultiplexing
- 3.3 Trasporto senza connessione: UDP
- 3.4 Principi del trasferimento dati affidabile
- 3.5 Trasporto orientato alla connessione: TCP
  - struttura dei segmenti
  - trasferimento dati affidabile
  - controllo di flusso
  - gestione della connessione
- 3.6 Principi sul controllo di congestione
- 3.7 Controllo di congestione TCP

Controllo di congestione TCP

- **Il TCP implementa il controllo della congestione, congiuntamente al controllo di flusso, solo agli estremi della comunicazione, e non richiede nessun supporto da parte dei router intermedi per realizzare questa funzione. Questo è coerente con il modello progettuale di IP, che prevede di aggiungere intelligenza ai nodi terminali lasciando i router il più possibile semplici.**

- **TCP impone a ciascun mittente un limite alla frequenza di invio sulla propria connessione in funzione della congestione di rete percepita**

  - Come fa a limitare la frequenza ?
  - Come si accorge della congestione ?
  - Quale algoritmo usa per modulare la Frequenza trasmissiva ?

Controllo di congestione TCP

- **Controllo punto-punto (senza assistenza dalla rete)**
- **Il mittente limita la trasmissione:**

  `LastByteSent-LastByteAcked`
  `≤ CongWin`
- **Approssimativamente:**

$$\text{Frequenza d'invio} = \frac{\text{CongWin}}{\text{RTT}} \text{ byte/sec}$$

- `CongWin` **è una funzione dinamica della congestione percepita**

In che modo il mittente percepisce la congestione?

- **Evento di perdita = timeout *o* ricezione di 3 ACK duplicati**
- **Il mittente TCP riduce la frequenza d'invio (`CongWin`) dopo un evento di perdita**

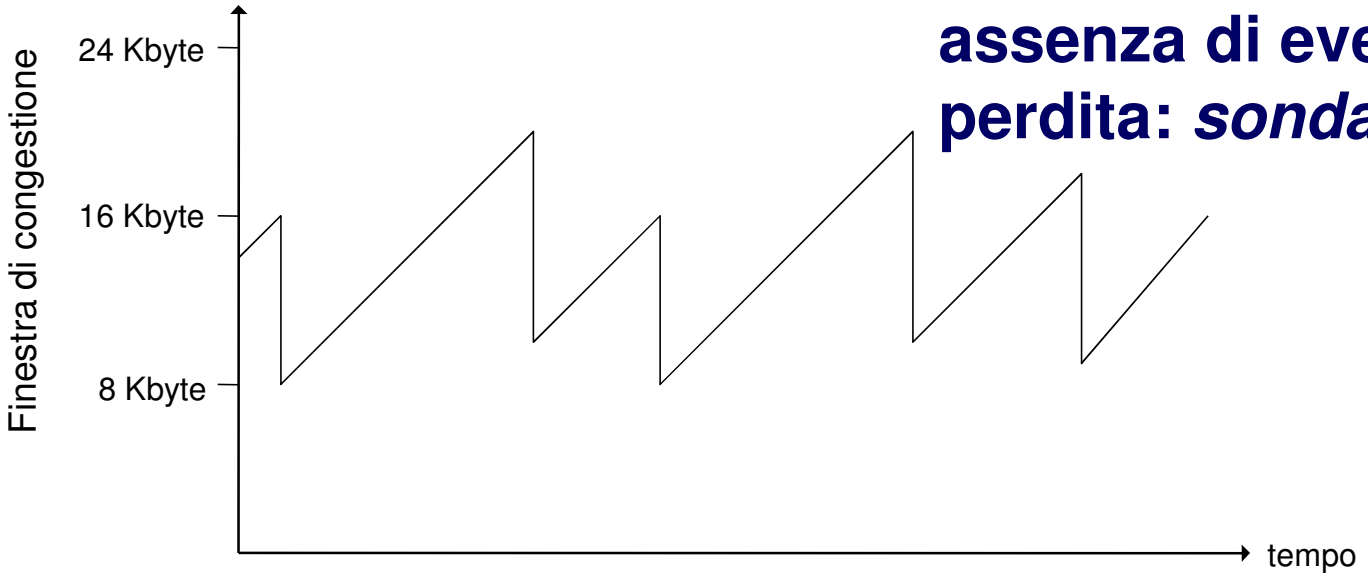
Algoritmo di controllo della congestione basato su tre meccanismi principali:

- AIMD
- Partenza lenta
- Reazione agli eventi di timeout

Incremento additivo e decremento moltiplicativo (AIMD)

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

Incremento additivo: aumenta `CongWin` di 1 MSS a ogni RTT in assenza di eventi di perdita: *sondaggio*



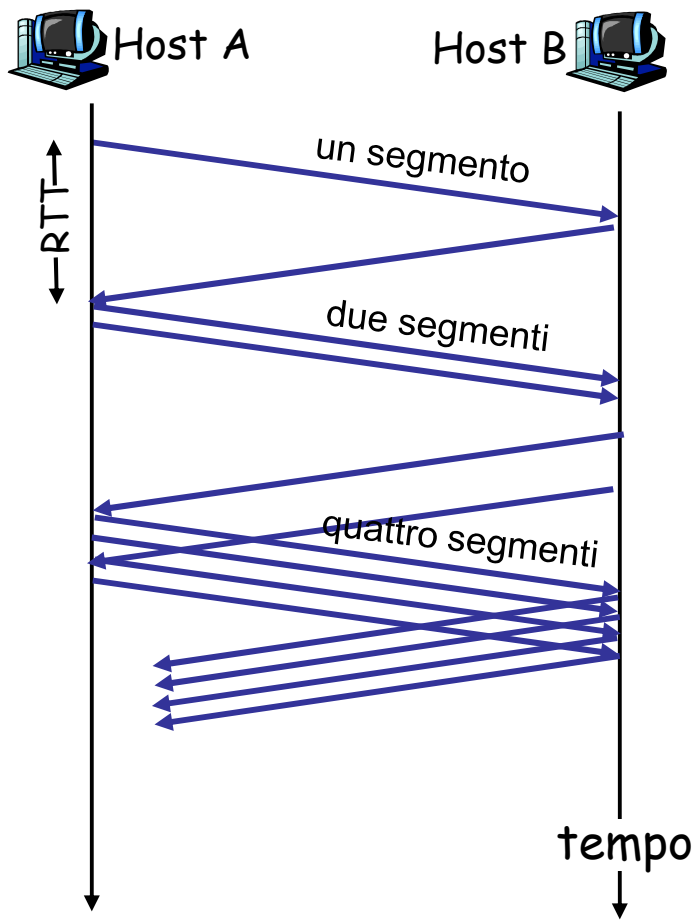
Controllo di congestione AIMD

Partenza lenta

- **Quando si stabilisce una connessione,** `CongWin` **= 1 MSS**
  - Esempio: MSS = 500 byte RTT = 200 msec
  - Frequenza iniziale = 20 kbps
- **La larghezza di banda disponibile potrebbe essere >> MSS/RTT**
  - Consente di raggiungere rapidamente una frequenza d'invio significativa
- **Quando inizia la connessione, la frequenza aumenta in modo esponenziale, fino a quando non si verifica un evento di perdita**

Partenza lenta (altro)

- **Quando inizia la connessione, la frequenza aumenta in modo esponenziale, fino a quando non si verifica un evento di perdita:**
  - raddoppia `CongWin` a ogni RTT
  - ciò avviene incrementando `CongWin` per ogni ACK ricevuto
- **Riassunto: la frequenza iniziale è lenta, ma poi cresce in modo esponenziale**



Affinamento

- **Dopo 3 ACK duplicati:**
  - **`CongWin`** è ridotto a metà
  - la finestra poi cresce linearmente
- **Ma dopo un evento di timeout:**
  - **`CongWin`** è impostata a 1 MSS;
  - poi la finestra cresce in modo esponenziale
  - fino a un valore di soglia, poi cresce linearmente

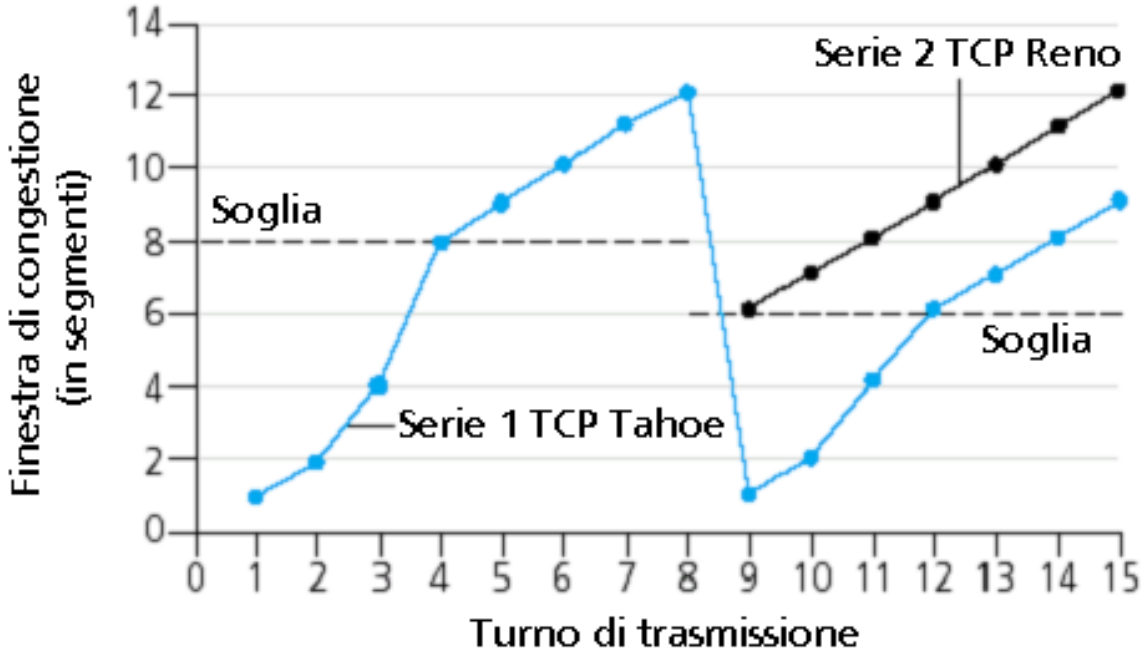
Filosofia:

- 3 ACK duplicati indicano la capacità della rete di consegnare qualche segmento
- un timeout prima di 3 ACK duplicati è "più allarmante"

Affinamento (altro)

D: Quando l'incremento esponenziale dovrà diventare lineare?

R: Quando `CongWin` raggiunge 1/2 del suo valore prima del timeout.



Implementazione:

- **Soglia variabile**
- **In caso di evento di perdita, la soglia è impostata a 1/2 di CongWin, appena prima dell'evento di perdita**

Riassunto: il controllo della congestione TCP

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

Controllo di congestione del mittente TCP

Stato	Evento	Azione del mittente TCP	Commenti
Slow Start (SS)	**Ricezione di ACK per dati precedente-mente non riscontrati**	**CongWin = CongWin + MSS, If (CongWin > Threshold) imposta lo stato a "Congestion Avoidance"**	**CongWin raddoppia a ogni RTT**
Congestion Avoidance (CA)	**Ricezione di ACK per dati precedente-mente non riscontrati**	**CongWin = CongWin + MSS * (MSS/CongWin)**	**Incremento additivo: CongWin aumenta di 1 MSS a ogni RTT**
SS o CA	**Rilevato un evento di perdita da tre ACK duplicati**	**Threshold = CongWin/2, CongWin = Threshold, imposta lo stato a "Congestion Avoidance"**	**Ripristino rapido con il decremento moltiplicativo. CongWin non sarà mai minore di 1 MSS**
SS o CA	**Timeout**	**Threshold = CongWin/2, CongWin = 1 MSS, imposta lo stato a "Slow Start"**	**Entra nello stato "Slow Start"**
SS o CA	**ACK duplicato**	**Incrementa il conteggio degli ACK duplicati per il segmento in corso di riscontro**	**CongWin e Threshold non variano**

Throughput TCP

- **Qual è il throughput medio di TCP in funzione della dimensione della finestra e di RTT?**
  - Ignoriamo le fasi di partenza lenta
- **Sia W la dimensione della finestra quando si verifica una perdita.**
- **Quando la finestra è W, il throughput è W/RTT**
- **Subito dopo la perdita, la finestra si riduce a W/2, il throughput a W/2RTT.**
- **Throughput medio: 0,75 W/RTT**

Futuro di TCP

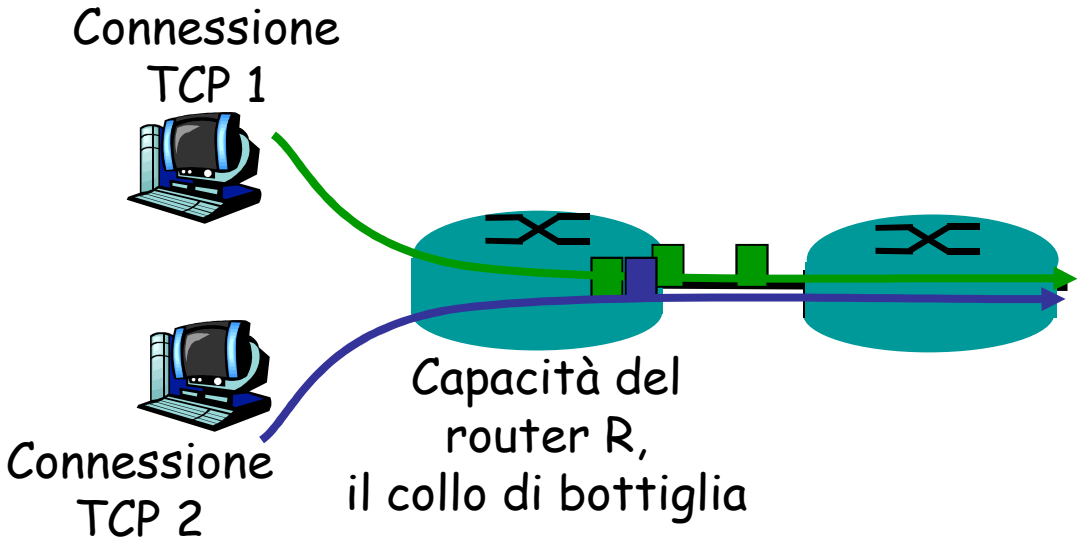
- Esempio: segmenti da 1500 byte, RTT di 100 ms, vogliamo un throughput da 10 Gbps
- Occorre una dimensione della finestra pari a W = 83.333 segmenti in transito
- Throughput in funzione della frequenza di smarrimento:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

- ➜ $L = 2 \cdot 10^{-10}$ *Wow*
- Occorrono nuove versioni di TCP per ambienti ad alta velocità!

Equità di TCP

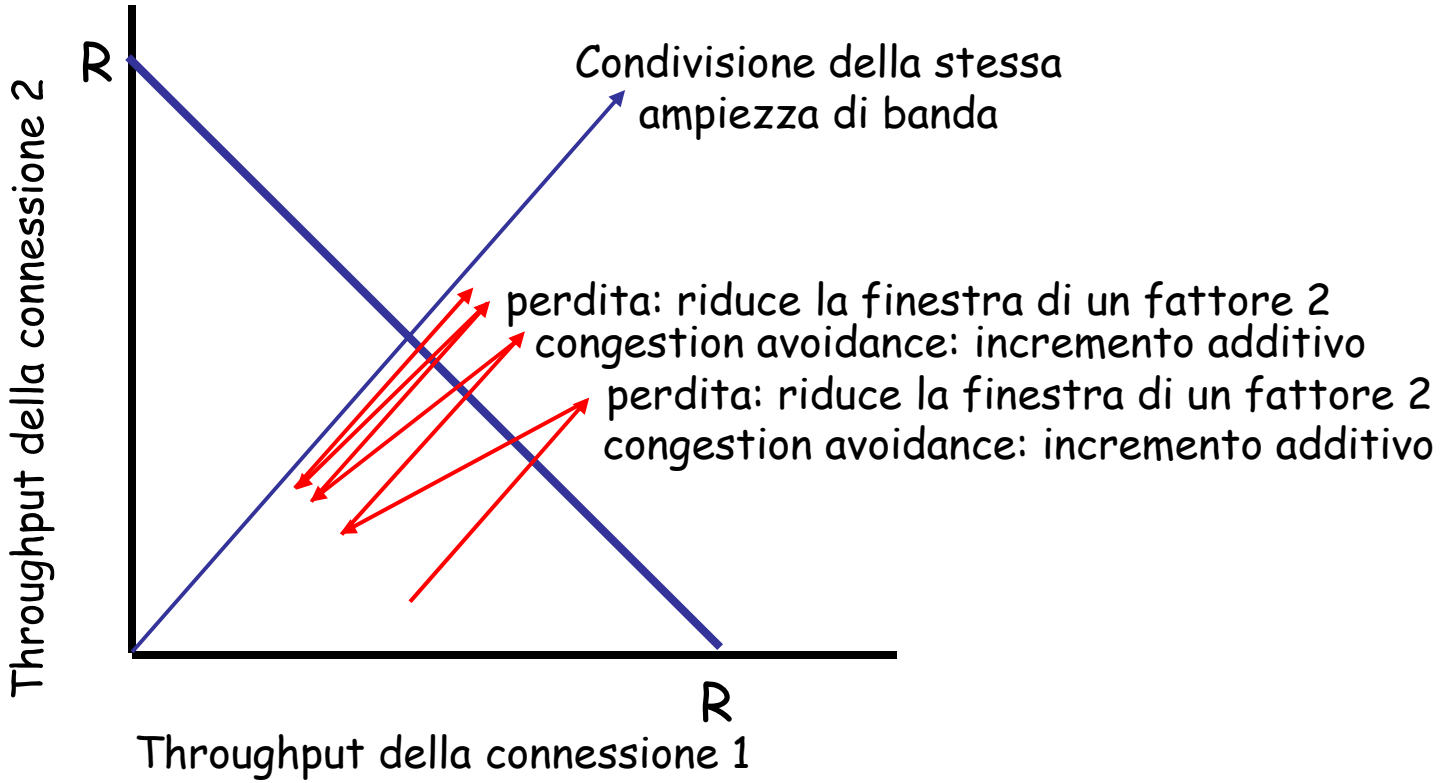
Equità: se K sessioni TCP condividono lo stesso collegamento con ampiezza di banda R, che è un collo di bottiglia per il sistema, ogni sessione dovrà avere una frequenza trasmissiva media pari a R/K.



Perché TCP è equo?

Due connessioni:

- **L'incremento additivo determina una pendenza pari a 1, all'aumentare del throughout**
- **Il decremento moltiplicativo riduce il throughput in modo proporzionale**



Equità (altro)

Equità e UDP

- **Le applicazioni multimediali spesso non usano TCP**
  - non vogliono che il loro tasso trasmissivo venga ridotto dal controllo di congestione
- **Utilizzano UDP:**
  - immettono audio/video a frequenza costante, tollerano la perdita di pacchetti
- **Area di ricerca: TCP friendly**

Equità e connessioni TCP in parallelo

- **Nulla può impedire a un'applicazione di aprire connessioni in parallelo tra 2 host**
- **I browser web lo fanno**
- **Esempio: un collegamento di frequenza R che supporta 9 connessioni;**
  - Se una nuova applicazione chiede una connessione TCP, ottiene una frequenza trasmissiva pari a R/10
  - Se la nuova applicazione chiede 11 connessioni TCP, ottiene una frequenza trasmissiva pari a R/2 !

Modellazione dei ritardi TCP

D: Quanto tempo occorre per ricevere un oggetto da un server web dopo avere inviato la richiesta?

Ignorando la congestione, il ritardo è influenzato da:

- Inizializzazione della connessione TCP
- Ritardo nella trasmissione dei dati
- Partenza lenta

Notazioni, ipotesi:

- Supponiamo che ci sia un solo collegamento tra il client e il server con tasso R
- S: dimensione massima dei segmenti o MSS (bit)
- O: dimensione dell'oggetto (bit)
- nessuna ritrasmissione (nessuna perdita né alterazione)

Dimensione della finestra:

- Prima supponiamo che la finestra di congestione sia statica, W segmenti
- Poi supponiamo che la finestra di congestione sia dinamica, per modellare la partenza lenta

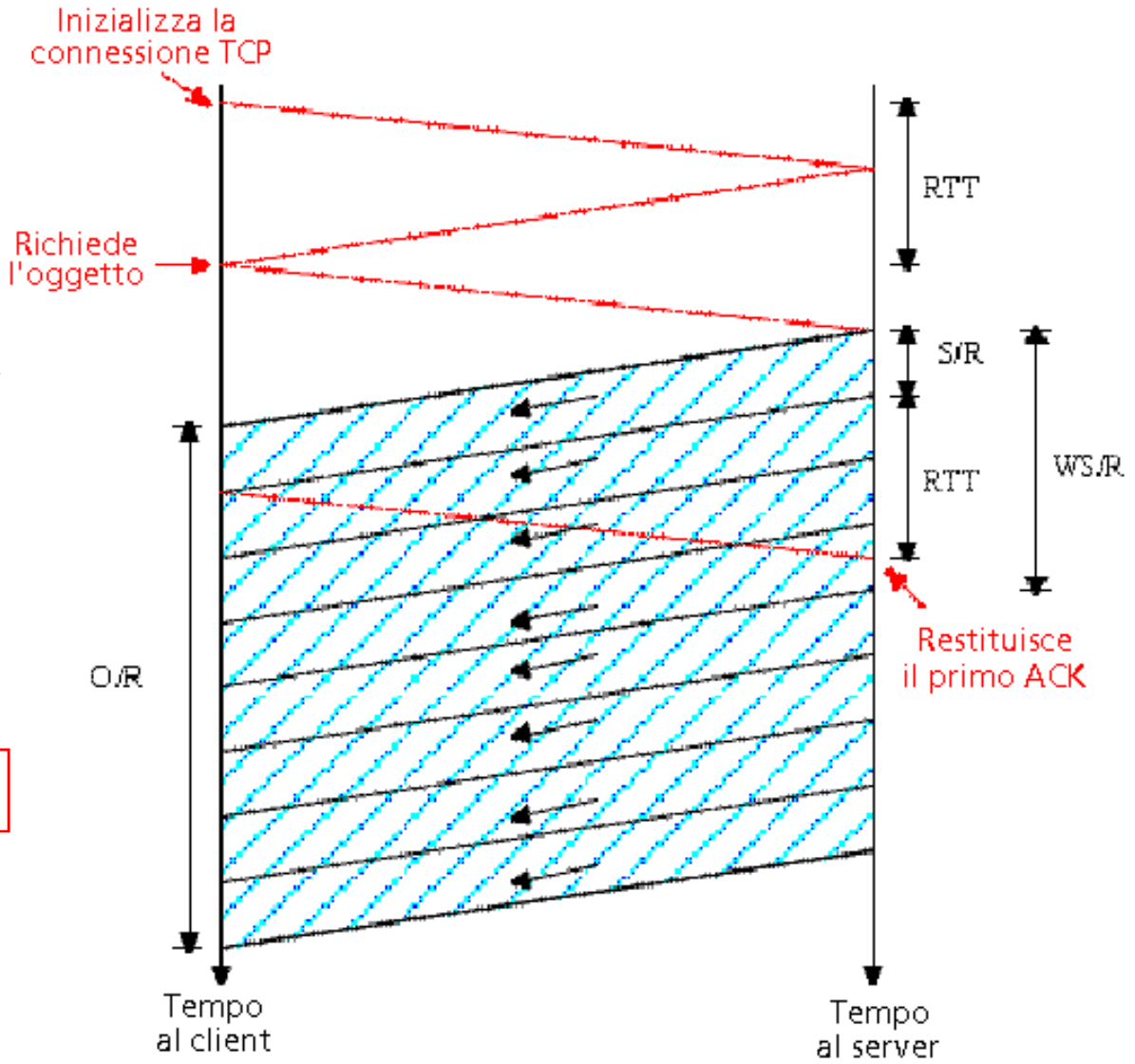
Finestra di congestione statica (1)

Primo caso:

WS/R > RTT + S/R:

il server riceve un ACK per il primo segmento nella finestra prima di completare la trasmissione della finestra

latenza = 2RTT + O/R

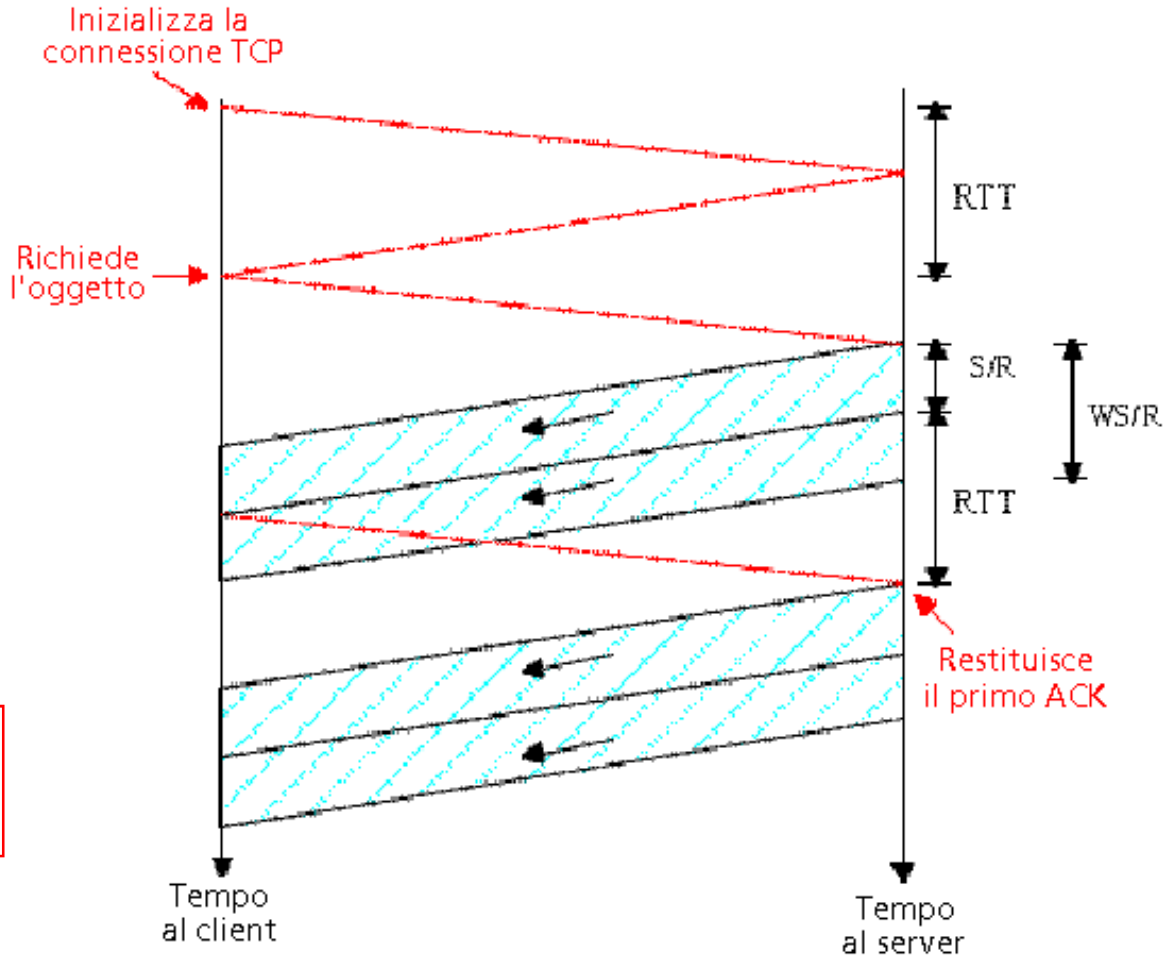


Finestra di congestione statica (2)

Secondo caso:

- **WS/R < RTT + S/R:**
  il server trasmette tanti segmenti quanti ne consente la dimensione della finestra prima che il server riceva un riscontro per il primo segmento nella finestra

latenza = 2RTT + O/R + (K-1)[S/R + RTT - WS/R]



Modellazione dei ritardi: Partenza lenta (1)

Adesso supponiamo che la finestra cresca secondo la partenza lenta

Dimostreremo che il ritardo per un oggetto è:

$$Latenza = 2RTT + \frac{O}{R} + P\left[RTT + \frac{S}{R}\right] - (2^P - 1)\frac{S}{R}$$

dove P è il numero di volte in cui il server entra in stallo:

$$P = \min\{Q, K - 1\}$$

- dove Q è il numero di volte in cui il server andrebbe in stallo se l'oggetto contenesse un numero infinito di segmenti.

- e K è il numero di finestre che coprono l'oggetto.

Modellazione dei ritardi: Partenza lenta (2)

Componenti della latenza:

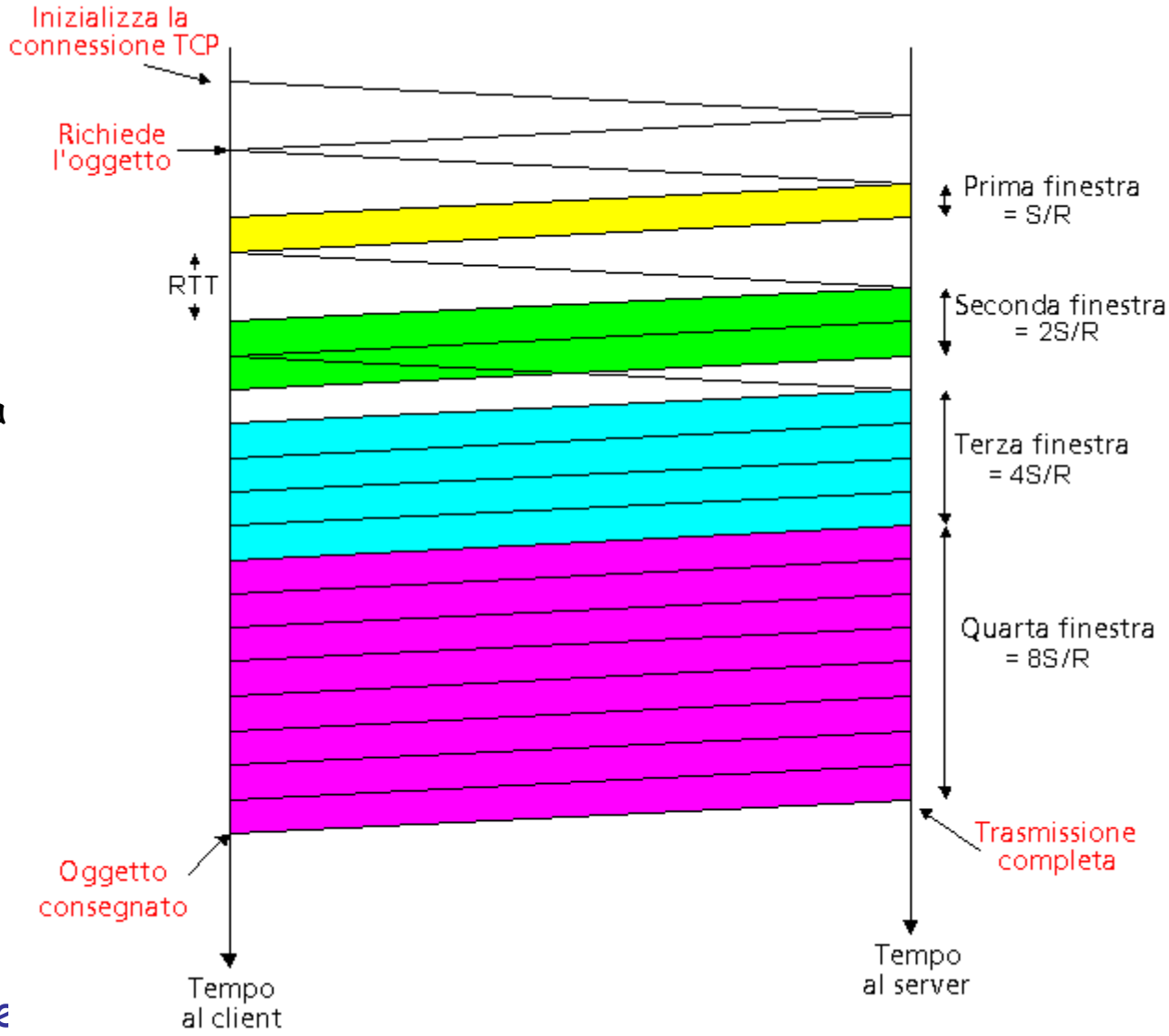
- 2 RTT per inizializzare la connessione e per la richiesta
- O/R per trasmettere l'oggetto
- periodo di stallo del server a causa della partenza lenta

Stalli del server:
P = min{K-1,Q} volte

Esempio:

- O/S = 15 segmenti
- K = 4 finestre
- Q = 2
- P = min{K-1,Q} = 2

Stalli del server P=2 volte



Modellazione dei ritardi TCP (3)

$\frac{S}{R} + RTT =$ tempo che passa da quando il server inizia a trasmettere un segmento fino a quando riceve un riscontro

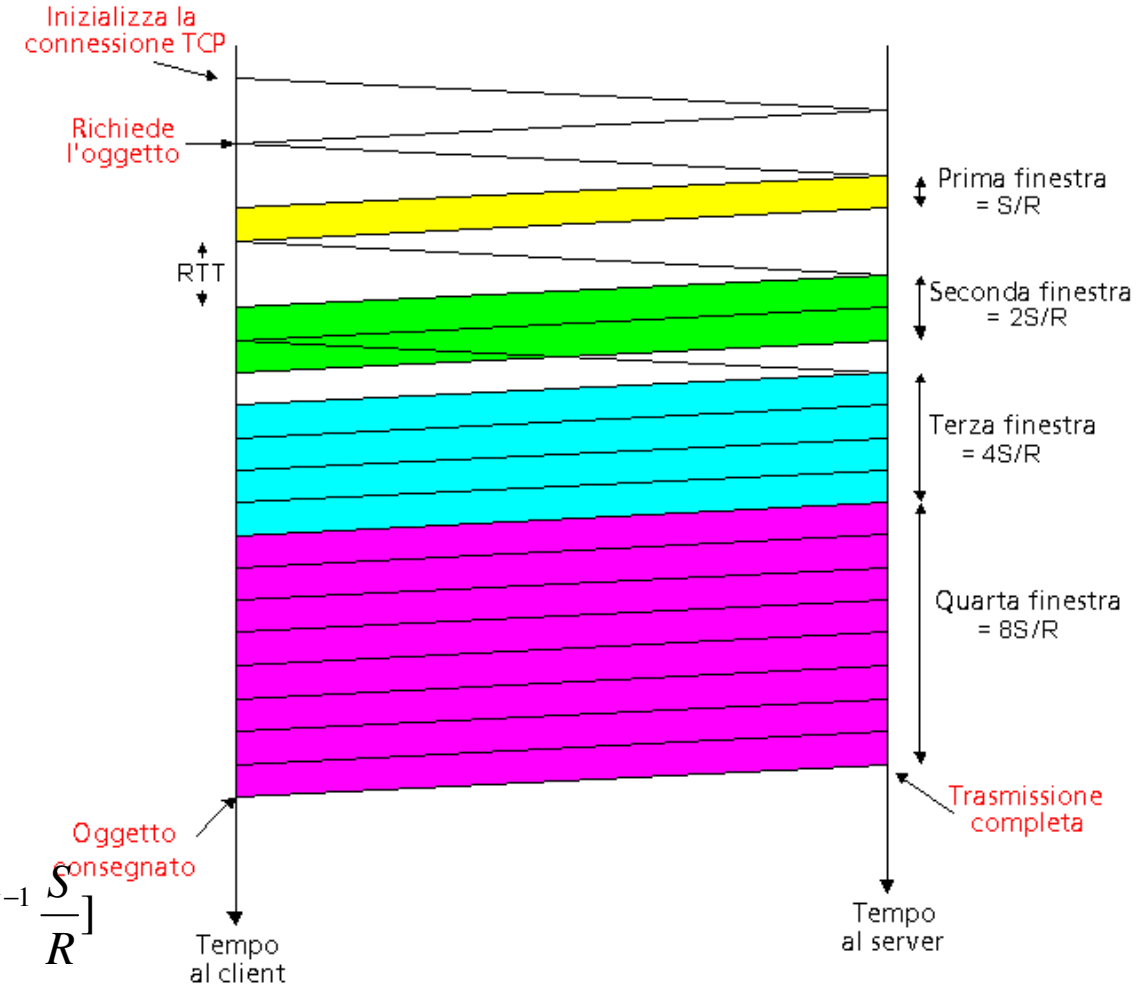
$2^{k-1}\frac{S}{R} =$ tempo per trasmettere la k-esima finestra

$\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]^{+} =$ tempo di stallo dopo la k-esima finestra

$$\textbf{latenza} = \frac{O}{R} + 2RTT + \sum_{p=1}^{P} PeriodidiStallo_p$$

$$= \frac{O}{R} + 2RTT + \sum_{k=1}^{P}\left[\frac{S}{R} + RTT - 2^{k-1}\frac{S}{R}\right]$$

$$= \frac{O}{R} + 2RTT + P\left[RTT + \frac{S}{R}\right] - (2^{P} - 1)\frac{S}{R}$$



Modellazione dei ritardi TCP (4)

Ricordiamo che K = numero di finestre che coprono l'oggetto

Come si calcola K?

$$
\begin{aligned}
K &= \min\{k : 2^0 S + 2^1 S + \cdots + 2^{k-1} S \geq O\} \\
&= \min\{k : 2^0 + 2^1 + \cdots + 2^{k-1} \geq O / S\} \\
&= \min\{k : 2^k - 1 \geq \frac{O}{S}\} \\
&= \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} \\
&= \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil
\end{aligned}
$$

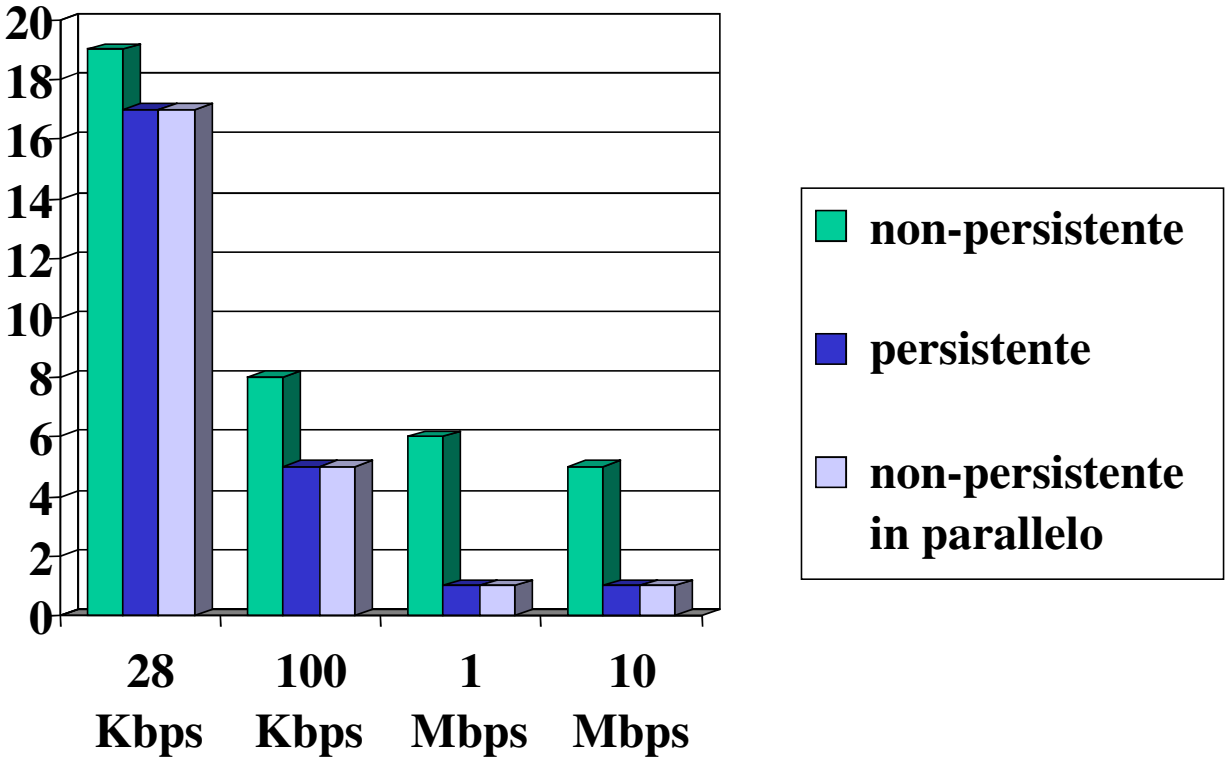
Il calcolo di Q (numero di volte in cui il server andrebbe in stallo se l'oggetto avesse dimensione infinita) è simile.

Esempio di modellazione: HTTP

- **Supponiamo che la pagina web sia formata da:**
  - *1* pagina HTML di base (di dimensione *O* bit)
  - *M* immagini (ciascuna di dimensione *O* bit)
- **HTTP non persistente:**
  - *M+1* connessioni TCP in serie
  - *Tempo di risposta = (M+1)O/R + (M+1)2RTT + somma degli stalli*
- **HTTP persistente:**
  - *2 RTT* per la richiesta e per ricevere il file HTML di base
  - *1 RTT* per la richiesta e per ricevere M immagini
  - *Tempo di risposta = (M+1)O/R + 3RTT + somma degli stalli*
- **HTTP non persistente con X connessioni in parallelo**
  - Supponiamo M/X intero
  - Una connessione TCP per il file di base
  - M/X gruppi di connessioni parallele per le immagini
  - *Tempo di risposta = (M+1)O/R + (M/X + 1)2RTT + somma degli stalli*

Tempo di risposta HTTP (in secondi)

RTT = 100 msec, O = 5 Kbyte, M = 10 e X = 5

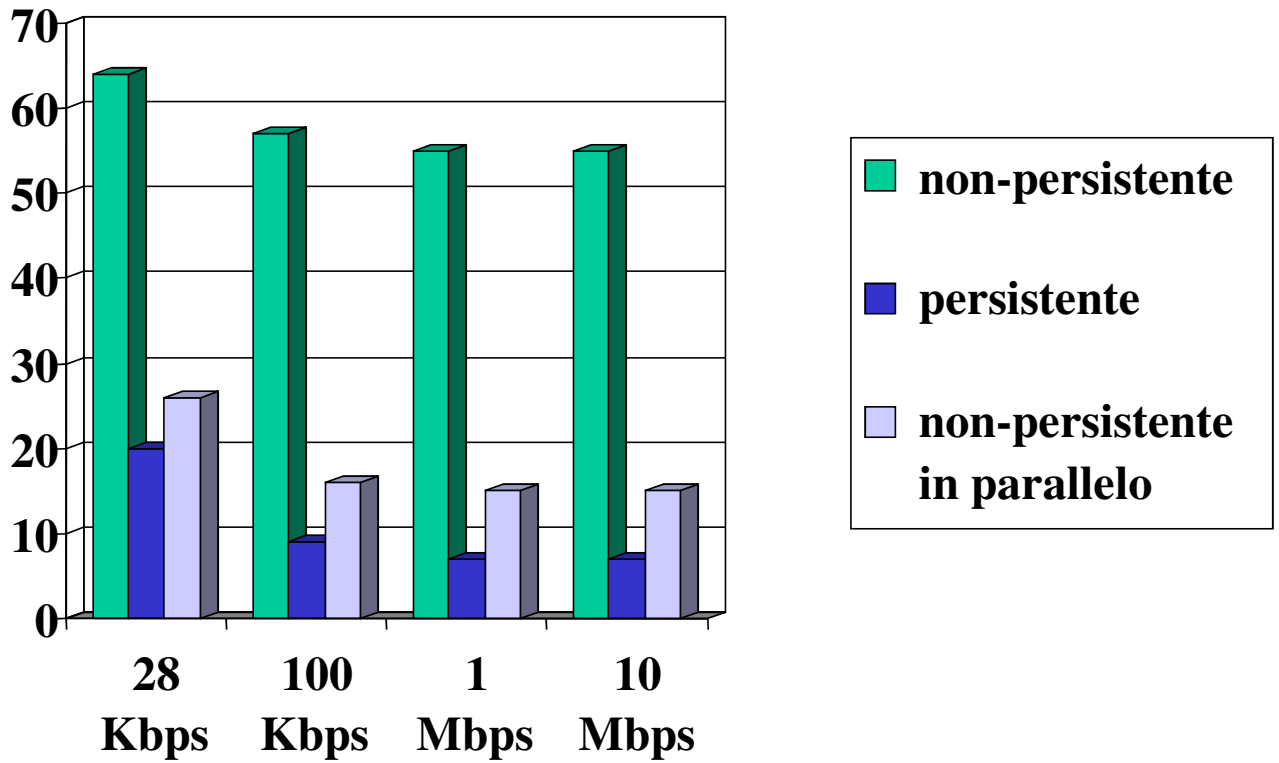


Se l'ampiezza di banda è limitata, la connessione e il tempo di risposta sono dominati dal tempo di trasmissione.

Le connessioni persistenti apportano soltanto un modesto miglioramento sulle connessioni parallele.

Tempo di risposta HTTP (in secondi)

RTT = 1 sec, O = 5 Kbyte, M = 10 e X = 5



Per RTT più grandi, il tempo di risposta è dominato dalla inizializzazione della connessione TCP e dai ritardi per partenze lente. Le connessioni persistenti adesso apportano un miglioramento significativo: in particolare nelle reti con un valore elevato del prodotto ritardo•ampiezzadibanda.

Capitolo 3: Riassunto

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

Prossimamente:

- **lasciare la "periferia" della rete (livelli di applicazione e di trasporto)**
- **nel "cuore" della rete**