



SAPIENZA  
UNIVERSITÀ DI ROMA

UNIVERSITÀ "SAPIENZA" DI ROMA  
FACOLTÀ DI INFORMATICA

---

## Reti di Elaboratori

---

Appunti integrati con il libro "Computer Networking: A Top-Down Approach", J. F. Kurose, K. W. Ross

*Author*  
Simone Bianco

29 marzo 2023

# Indice

<b>0 Introduzione</b>	<b>1</b>
<b>1 Introduzione alle reti</b>	<b>2</b>
1.1 Rete, Host e Collegamenti . . . . .	2
1.2 Struttura di Internet . . . . .	4
1.3 Pacchetti, Commutazione e Instradamento . . . . .	7
1.4 Misura delle prestazioni . . . . .	9
1.5 Stack protocollare TCP/IP . . . . .	14
<b>2 Livello di Applicazione</b>	<b>18</b>
2.1 Principi delle applicazioni di rete . . . . .	18
2.2 Web e Protocollo HTTP . . . . .	21
2.2.1 Messaggi di richiesta e risposta . . . . .	23
2.2.2 Versioni di HTTP . . . . .	26
2.2.3 Cookies e Web Caching . . . . .	27
2.3 Posta elettronica . . . . .	28
2.3.1 Protocolli SMTP e MIME . . . . .	29
2.3.2 Protocolli POP3 e IMAP . . . . .	31
2.4 Domain Name System (DNS) . . . . .	33
2.4.1 Gerarchia server DNS . . . . .	33
2.4.2 Protocollo DNS . . . . .	36
2.5 Trasferimento di file . . . . .	37
2.5.1 Protocollo FTP . . . . .	37
2.5.2 Protocollo BitTorrent . . . . .	39
<b>3 Livello di Trasporto</b>	<b>40</b>
3.1 Multiplexing e Demultiplexing . . . . .	40
3.2 Trasporto senza connessione (protocollo UDP) . . . . .	42
3.3 Trasferimento affidabile dei dati . . . . .	44
3.3.1 Protocollo RDT 1.0 e 2.0 . . . . .	46
3.3.2 Protocollo RDT 2.1 e 2.2 . . . . .	47
3.3.3 Protocollo RDT 3.0 . . . . .	49
3.3.4 Go-back-N e Selective repeat . . . . .	51
3.4 Trasporto orientato alla connessione (protocollo TCP) . . . . .	56
3.4.1 Gestione del timeout e stima del RTT . . . . .	58
3.4.2 Controllo del flusso . . . . .	60
3.4.3 Gestione della connessione . . . . .	61
3.5 Controllo della congestione . . . . .	64

3.5.1	Cause e costi della congestione . . . . .	64
3.5.2	Controllo della congestione nel TCP . . . . .	67
3.6	Equità nei protocolli di trasporto . . . . .	72

# Capitolo 0

## Introduzione

# Capitolo 1

## Introduzione alle reti

### 1.1 Rete, Host e Collegamenti

#### Definition 1. Rete e Link

Una **rete** è un'infrastruttura composta da dispositivi detti **nodi della rete** in grado di scambiarsi informazioni tramite dei mezzi di comunicazione, wireless o cablati, detti **link (o collegamenti)**

#### Definition 2. Nodi di una rete

I **nodi** costituenti una rete vengono differenziati in **due macro-categorie**:

- **Sistemi terminali**, differenziati a loro volta in
  - **Host**, ossia un dispositivo di proprietà dell'utente dedicato ad eseguire applicazioni utente
  - **Server**, ossia un dispositivo di elevate prestazioni destinato ad eseguire programmi che forniscono un servizio a diverse applicazioni utente
- **Dispositivi di interconnessione**, ossia dei dispositivi atti a modificare o prolungare il segnale ricevuto, differenziati a loro volta in:
  - **Router**, ossia dispositivi che collegano una rete ad una o più reti
  - **Switch**, ossia dispositivi che collegano più sistemi terminali all'interno di una rete
  - **Modem**, ossia dispositivi in grado di trasformare la codifica dei dati in segnale e viceversa

In particolare, classifichiamo le varie tipologie di rete in:

- **Personal Area Network (PAN)**, avente scala ridotta, solitamente equivalente a pochi metri (es: una rete Bluetooth)

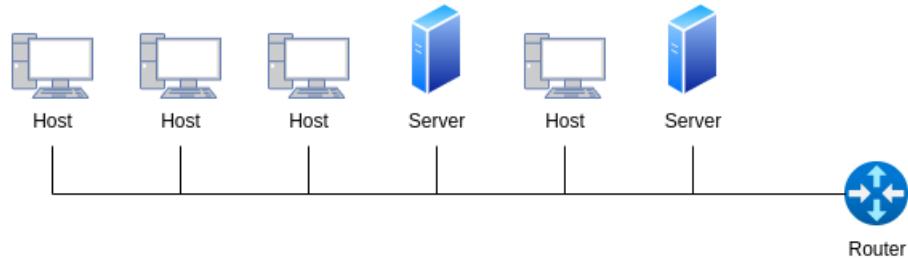
- **Local Area Network (LAN)**, solitamente corrispondente ad una rete privata che collega i sistemi terminali di un appartamento (es: una rete wi-fi o Ethernet). Ogni sistema terminale possiede un indirizzo che lo identifica univocamente all'interno della LAN.

Si differenziano in **LAN con cavo condiviso**, ossia dove tutti i dispositivi sono connessi al router tramite un cavo comune, e **LAN con switch**, ossia dove tutti i dispositivi sono connessi ad uno o più switch, i quali a loro volta sono connessi al router

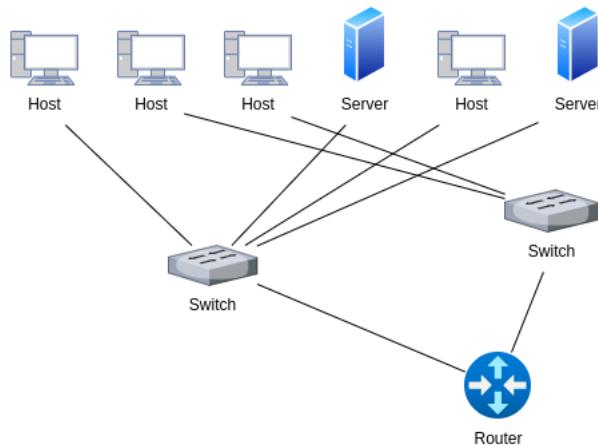
- **Metropolitan Area Network (MAN)**, avente scala pari ad una città
  - **Wide Area Network (WAN)**, avente scala pari ad un paese o una nazione, solitamente gestita da un **Internet Service Provider (ISP)**.
- Si differenziano in **WAN punto-punto**, ossia collegante due reti tramite un singolo mezzo mezzo di trasmissione, e **WAN a commutazione**, ossia collegante più reti tramite più mezzi e dispositivi di collegamento
- **L'Internet**, avente scala globale

Esempi:

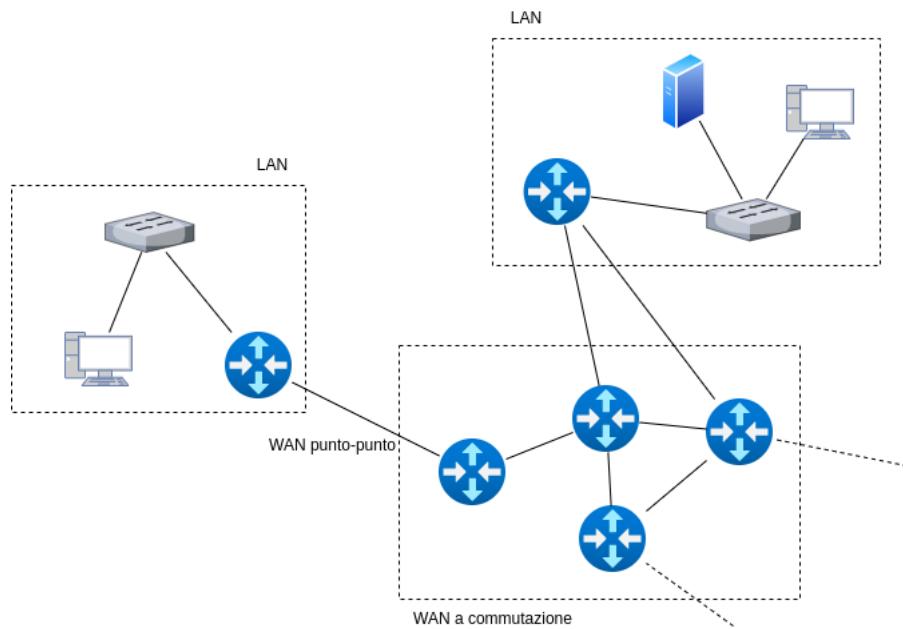
- **LAN a cavo condiviso**



- **LAN con switch**



- **Rete composta**



I supporti fisici utilizzabili per una trasmissione si differenziano in:

- **Doppino intrecciato** (ad esempio un cavo Ethernet), composto da due fili di rame isolati, uno utilizzato per inviare i dati ed uno per riceverli
- **Cavo coassiale**, composto da due conduttori di rame concentrici, entrambi bidirezionali, avente una larghezza di banda maggiore
- **Cavo in fibra ottica**, composto da una fibra di vetro che trasporta impulsi luminosi (dunque alla velocità della luce) al suo interno, ognuno rappresentante un singolo bit
- **Trasmissione wireless**, realizzata tramite l'invio di un segnale radio propagato nell'aria (es: rete cellulare, satellitare o wi-fi)

## 1.2 Struttura di Internet

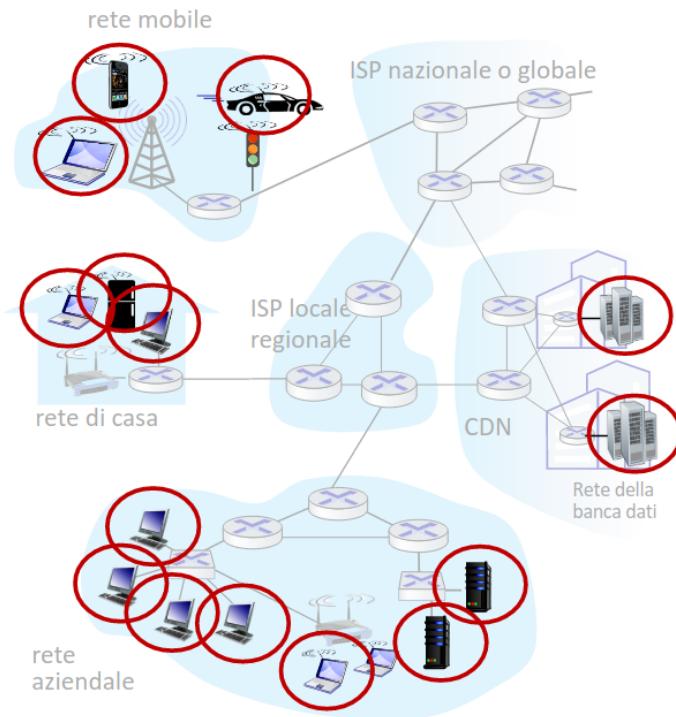
### Definition 3. Rete internet

Definiamo come **internet** (abbreviativo di internetwork) una **rete di reti**, ossia una rete che mette in comunicazione due o più reti tra di loro.

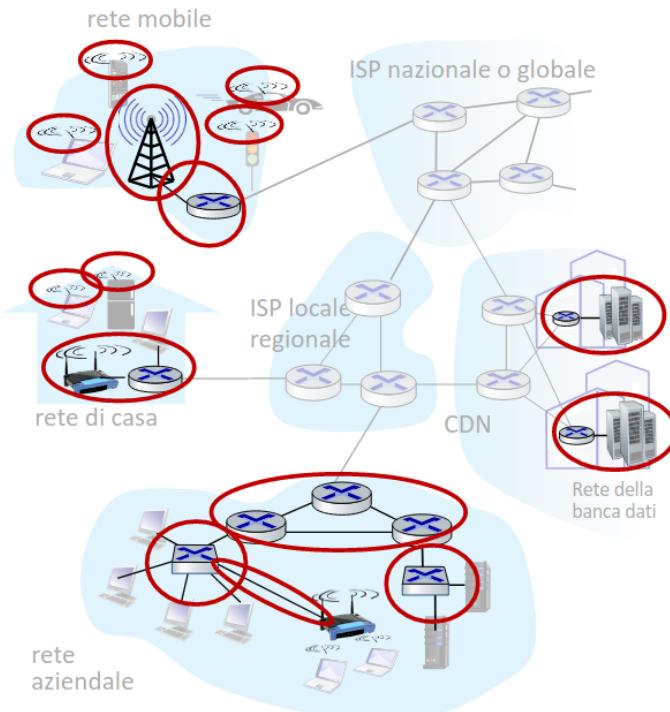
**Attenzione:** nonostante ciò che viene comunemente chiamato l'**Internet** sia una internet, è necessario puntualizzare che con tale termine comune viene indicata la **rete di tutte le reti**.

Al suo interno, la struttura di Internet risulta essere composta da:

- **Periferia della rete (network edge)**, corrispondente all'insieme di tutti i sistemi terminali connessi.

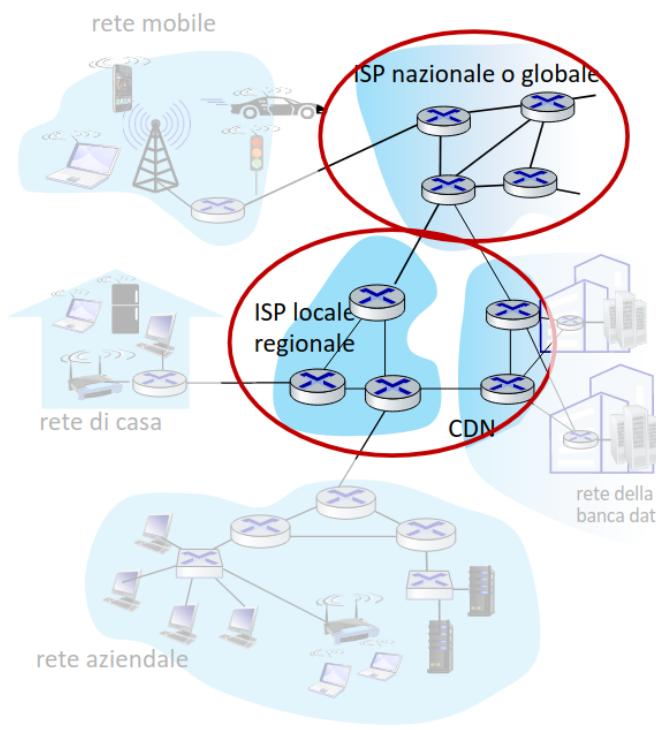


- **Reti di accesso (access network)**, corrispondente ai collegamenti fisici che connettono un sistema terminale al primo **edge router**, ossia il primo router presente nel percorso dal sistema terminale di origine ad un qualsiasi altro sistema terminale di destinazione.



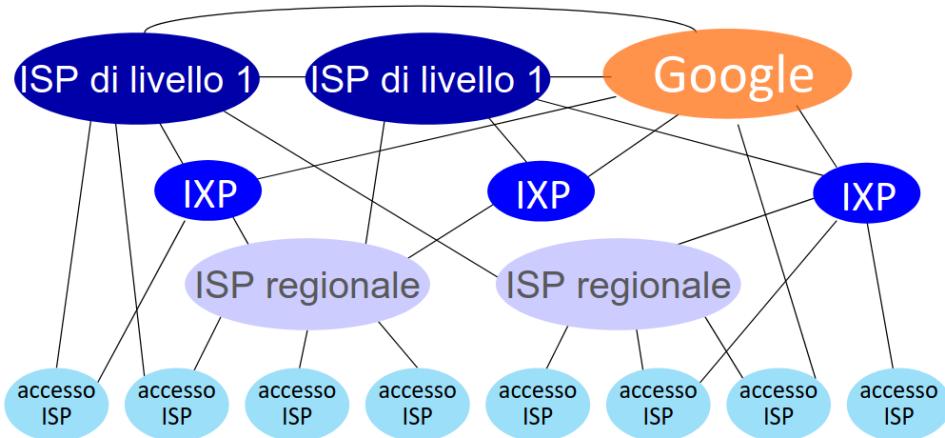
In particolare, l'accesso all'Internet può essere effettuato in più modi:

- **Accesso via cavo**, tramite supporti fisici connessi direttamente ad una rete di distribuzione, detta **cable headend** (es: il centralino di un ISP).
- **Accesso via Digital Subscriber Line (DSL)**, dove viene utilizzata la linea telefonica esistente per collegarsi alla rete dell'ISP
- **Accesso via Wireless LAN (WLAN)**, tramite un collegamento wireless ad una stazione base detta **access point** connessa con il router, a sua volta connesso con un cable headend
- **Accesso via rete cellulare**, dove viene utilizzata la rete cellulare esistente per collegarsi alla rete dell'ISP
- **Accesso via rete aziendale**, tramite una rete aziendale (o universitaria, privata, ...) direttamente connessa ad Internet
- **Nucleo di rete (core o backbone)**, ossia un sistema di router interconnessi tra di loro, corrispondente all'insieme di nodi cui viene realizzata la vera interconnessione tra tutte le reti.



In particolare, all'interno del backbone di Internet sono presenti **più livelli di reti ISP** (es: regionali, nazionali, aziendali, ...), le quali devono essere interconnesse tra di loro tramite degli **Internet Exchange Point (IXP)**.

Inoltre, nel recente periodo, nel backbone di Internet sono state integrate anche delle grandi reti private aziendali, ossia le **reti dei content provider** (es: Google, Netflix, ...), le quali, ormai, funzionano come vere e proprie ISP.



## 1.3 Pacchetti, Commutazione e Instradamento

### Definition 4. Pacchetto e Velocità di trasmissione

Dato un messaggio  $m$  da trasferire tra due terminali, definiamo come **pacchetti** l'insieme di blocchi di  $L$  bit tali che  $m = \{p_1, \dots, p_k\}$ .

Ogni pacchetto viene trasmesso nella rete ad una **velocità di trasmissione  $R$**  (anche detta larghezza di banda o capacità del collegamento).

### Definition 5. Forwarding e Routing

Le funzioni fondamentali di una rete si dividono in:

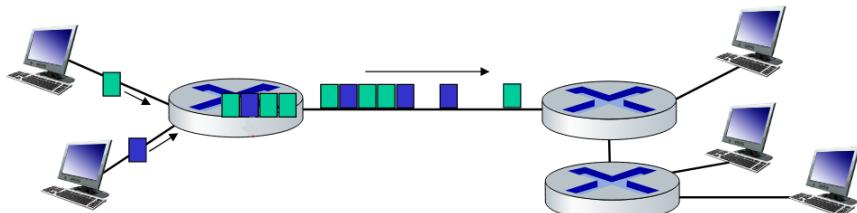
- **Forwarding o Switching (commutazione)**, ossia un'azione locale tramite cui vengono spostati i pacchetti in arrivo dal collegamento di ingresso del router al collegamento appropriato di uscita. Viene effettuato attraverso una **local forwarding table**, contenente gli indirizzi dei nodi locali
- **Routing (instradamento)**, ossia un'azione globale tramite cui vengono determinati i percorsi origine-destinazione seguiti dai pacchetti. Viene effettuato tramite **algoritmi di instradamento**

In particolare, la commutazione può avvenire in due modi:

- **Commutazione di pacchetto:**

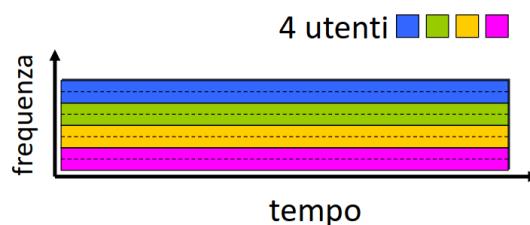
- La rete inoltra i pacchetti da un router all'altro attraverso i collegamenti presenti nell'instradamento dall'origine alla destinazione.
- Una volta inviato, un pacchetto deve completamente raggiungere il nodo a cui sta attualmente venendo inviato prima di poter essere trasmesso al collegamento successivo (**store & forward**)

- Se la velocità di trasmissione sul link di entrata supera la velocità di trasmissione di quello in uscita, i pacchetti verranno messi all'interno di una coda, in attesa di essere trasmessi sul link di uscita
- Se il buffer della coda raggiunge capienza massima, i pacchetti verranno scartati (**perdita di pacchetti**), per poi, se necessario, essere rinviati

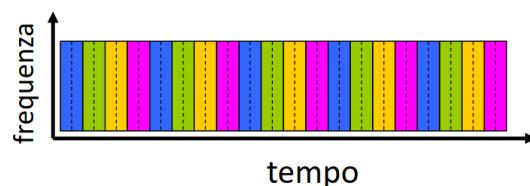


- **Commutazione di circuito:**

- La banda dei mezzi di trasmissione viene **suddivisa** in parti, riservando ognuna di essere ad una comunicazione tra un'origine ed una destinazione.
- Per via di tale suddivisione, il numero di utenti massimo della rete risulta essere **limitato dal numero di suddivisioni**
- La suddivisione può essere effettuata in due modalità:
  - \* **Frequency Division Multiplexing (FDM)**, dove le frequenze del mezzo di trasmissione vengono suddivise in bande di frequenza, ognuna di esse riservata ad una singola comunicazione, la quale può utilizzare al massimo la banda ad essa riservata



- \* **Time Division Multiplexing (TDM)**, dove il tempo viene suddiviso in slot, ognuno di essi riservato ad una singola comunicazione, la quale può utilizzare l'intera banda del mezzo per il breve lasso di tempo dedicato.



Nonostante la **commutazione di pacchetto** permetta l'accesso di un numero maggiore di utenti e non necessiti di stabilire una configurazione del collegamento, la presenza di una possibile perdita di pacchetti rende tale tipo di commutazione prettamente ottimo per **trasmissioni "bursty"**, ossia intermittenti e con lunghi periodi di inattività.

## 1.4 Misura delle prestazioni

### Definition 6. Larghezza di banda e Transmission rate

Con il termine **larghezza di banda (bandwidth)** indichiamo due concetti strettamente legati tra loro:

- La quantità (espressa in  $Hz$ ) rappresentante la **larghezza dell'intervallo di frequenze** utilizzato dal sistema trasmittivo, ossia l'intervallo di frequenze utilizzato dal sistema trasmittivo. Maggiore è tale quantità, maggiore è la quantità di informazioni veicolabili tramite il mezzo di trasmissione.
- La quantità (espressa in  $b/s$ ) detta anche **transmission rate (o bit rate)** rappresentante la **quantità di bit al secondo** che un link **garantisce di trasmettere**. Tale quantità è proporzionale alla larghezza di banda (in  $Hz$ )

### Definition 7. Throughput

Con il termine **throughput** indichiamo la **quantità di bit** al secondo che **passano attraverso un nodo** della rete.

### Observation 1

A differenza del **transmission rate**, il quale fornisce una misura della **potenziale velocità di un link**, il **throughput** fornisce una misura dell'**effettiva velocità di un link**.

In generale, dunque, si ha che

$$T < R$$

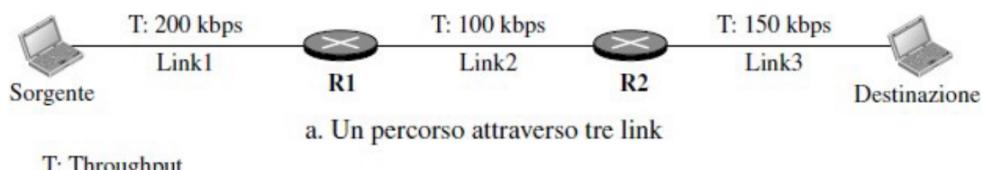
dove  $T$  è il throughput e  $R$  è il transmission rate

### Definition 8. Collo di bottiglia

Dato un percorso end-to-end, ossia tra un dispositivo e un altro, definiamo come **collo di bottiglia** il link limitante il throughput dei link presenti su tale percorso

Esempio:

- Consideriamo il seguente percorso



- Il link  $L_2$  risulta essere il collo di bottiglia di tale percorso, limitando il throughput del percorso a 100 kb/s

**Definition 9. Delay di trasmissione**

Definiamo come **delay (o latenza) di trasmissione** il tempo necessario ad un nodo per immettere un pacchetto su un link, corrispondente a:

$$D_t = \frac{L}{R}$$

dove  $L$  è la dimensione del pacchetto e  $R$  è il transmission rate del link

**Definition 10. Delay di propagazione**

Definiamo come **delay (o latenza) di propagazione** il tempo impiegato dall'**ultimo bit di blocco di dati** posto su un link ad essere propagato fino al nodo di destinazione, corrispondente a:

$$D_p = \frac{k}{v}$$

dove  $k$  è la lunghezza del link e  $v$  è la velocità di propagazione del link

**Definition 11. Delay di un pacchetto**

Definiamo come **delay (o latenza) di un pacchetto** il tempo totale necessario ad un pacchetto per essere inviato completamente da un nodo origine ad un nodo destinatario

$$D_n = D_e + D_q + D_t + D_p$$

dove:

- $D_e$  è il **delay di elaborazione del nodo**, dipendente dalle operazioni di controllo svolte dal nodo
- $D_q$  è il **delay di queueing**, ossia l'attesa del pacchetto all'interno della coda del nodo prima di essere trasmesso, dipendente dalla quantità di pacchetti presenti nella coda
- $D_t$  è il delay di trasmissione del link
- $D_p$  è il delay di propagazione del link

**Proposition 1. Prodotto rate per delay di propagazione**

Dato un link con transmission rate  $R$  e delay di propagazione  $D_p$ , il prodotto

$$B_{max} = R \cdot D_p = \frac{L \cdot k}{D_t \cdot v}$$

rappresenta il **massimo numero di bit distribuiti tutto sul cavo** contemporaneamente

**Esempi:**

- Si consideri un router A che trasmette pacchetti, ognuno di lunghezza  $L = 4000$  bit, su un canale di trasmissione con rate  $R = 10 \text{ Mb/s}$  verso un router B all'altro estremo del link. Si supponga che il delay di propagazione sia pari a 0.2 ms.

- Quanto impiega il router A a trasmettere un pacchetto al router B?

$$D_t = \frac{L}{R} = \frac{4 \cdot 10^3 \text{ b}}{10^7 \text{ b/s}} = 4 \cdot 10^{-4} \text{ s} = 0.4 \text{ ms}$$

- Quanto impiega il router A a trasmettere un bit al router B?

$$D_{1b} = \frac{1}{R} = \frac{1 \text{ b}}{10^7 \text{ b/s}} = 10^{-7} \text{ s} = 0.1 \mu\text{s}$$

- Qual è il massimo numero di pacchetti al secondo che possono essere trasmessi sul link?

$$\begin{aligned} 1 \text{ P} &= 4000 \text{ b} \implies 1 \text{ b} = \frac{1}{4000} \text{ P} \implies \\ &\implies R = 10^7 \text{ b/s} = \frac{10^7}{4000} \text{ P/s} = \frac{1}{4} \cdot 10^3 \text{ P/s} = 2500 \text{ P/s} \end{aligned}$$

- Supponendo che il router A invii i pacchetti uno dopo l'altro senza introdurre ritardi tra la trasmissione di un pacchetto e il successivo, quanto tempo impiega il router B a ricevere 4 pacchetti?

Poiché i pacchetti vengono inviati senza alcun delay tra di essi, possiamo considerare tali pacchetti come un unico grande pacchetto di dimensione  $4 \cdot L$ , implicando che

$$D_{4t} = \frac{4 \cdot L}{R} = \frac{16 \cdot 10^3 \text{ b}}{10^7 \text{ b/s}} = 16 \cdot 10^{-4} \text{ s} = 1.6 \text{ ms}$$

Inoltre, per lo stesso motivo, il tempo di propagazione rimarrà inalterato, poiché esso non dipende dalla lunghezza del pacchetto, ma solo dalla lunghezza e della velocità di propagazione del link. Di conseguenza, il tempo totale impiegato sarà  $1.6 \text{ ms} + 0.2 \text{ ms} = 1.8 \text{ ms}$

- Qual è il massimo numero di bit e il numero di pacchetti che possono essere presenti sul canale?

$$P_{max} = R \cdot D_p = 10^7 \text{ b/s} \cdot 0.2 \text{ ms} = 2000 \text{ b} = \frac{1}{2} \text{ P}$$

- Si consideri un host A che vuole inviare un file molto grande, 4 milioni di byte, a un host B. Il percorso tra A e B ha 3 link  $L_1, L_2, L_3$ , ognuno di lunghezza 300 km, ciascuno con rate rispettivo  $R_1 = 500 \text{ kb/s}$ ,  $R_2 = 2 \text{ Mb/s}$  e  $R_3 = 1 \text{ Mb/s}$ .

- Assumendo l'assenza di ulteriore traffico nella rete, qual è il throughput per il file transfer?

Poiché il link  $L_1$  risulta essere il collo di bottiglia del percorso, il throughput risulta essere  $R_1 = 500 \text{ kb/s}$

- Qual è il tempo totale impiegato per trasferire il file all'host B assumendo che i link siano cavi in fibra ottica?

Poiché non vi è specificata la lunghezza di ogni pacchetto, assumiamo che il file venga inviato come un unico grande pacchetto, implicando che  $L = 4 \cdot 10^6 \text{ b} = 32 \cdot 10^6 \text{ b}$ .

Di conseguenza, si ha che:

$$D_t(L_1) = \frac{32 \cdot 10^6 \text{ b}}{5 \cdot 10^5 \text{ b/s}} = 64 \text{ s}$$

$$D_t(L_2) = \frac{32 \cdot 10^6 \text{ b}}{2 \cdot 10^6 \text{ b/s}} = 16 \text{ s}$$

$$D_t(L_3) = \frac{32 \cdot 10^6 \text{ b}}{1 \cdot 10^6 \text{ b/s}} = 32 \text{ s}$$

Poiché  $L_1, L_2, L_3$  sono cavi in fibra ottica, la velocità di propagazione su di essi corrisponde alla velocità della luce, pari a  $\sim 3 \cdot 10^8 \text{ m/s}$ . Dunque, il delay di propagazione di ogni link corrisponderà a:

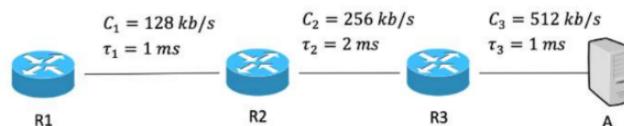
$$D_p = \frac{3 \cdot 10^5 \text{ m}}{3 \cdot 10^8 \text{ m/s}} = 1 \text{ ms}$$

Infine, concludiamo che il tempo totale impiegato corrisponda a:

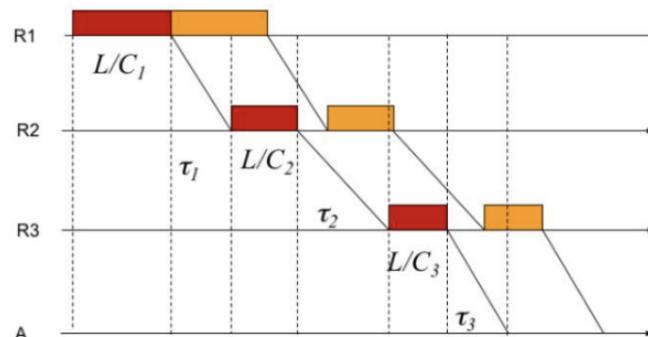
$$D_{tot} = D_t(L_1) + D_t(L_2) + D_t(L_3) + 3 \cdot D_p = 64 \text{ s} + 16 \text{ s} + 32 \text{ s} + 3 \cdot 1 \text{ ms} = 112,003 \text{ s}$$

3. Si consideri la rete nella seguente figura, dove  $C_1, C_2, C_3$  e  $\tau_1, \tau_2, \tau_3$  sono rispettivamente i transmission rate e i delay di propagazione dei tre link.

Al tempo  $t = 0$ , la coda di uscita di  $R_1$  contiene 2 pacchetti diretti ad  $A$ . Assumendo che la lunghezza dei pacchetti sia  $L = 512 \text{ b}$ , si indichi per ciascun pacchetto l'istante in cui esso viene completamente ricevuto da  $A$ .



Per aiutarci durante il calcolo, tracciamo il contenuto di ogni coda al passare del tempo:



Dunque, il tempo totale impiegato dal primo pacchetto corrisponderà a:

$$T_1 = \frac{L}{C_1} + \tau_1 + \frac{L}{C_2} + \tau_2 + \frac{L}{C_3} + \tau_3 = 4 \text{ ms} + 1 \text{ ms} + 2 \text{ ms} + 2 \text{ ms} + 1 \text{ ms} + 1 \text{ ms} = 11 \text{ ms}$$

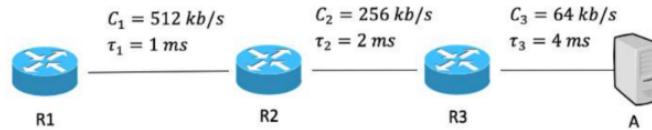
Analogamente, il tempo totale impiegato dal secondo pacchetto corrisponderà a:

$$T_2 = 2 \cdot \frac{L}{C_1} + \tau_1 + \frac{L}{C_2} + \tau_2 + \frac{L}{C_3} + \tau_3 = 8 \text{ ms} + 1 \text{ ms} + 2 \text{ ms} + 2 \text{ ms} + 1 \text{ ms} + 1 \text{ ms} = 15 \text{ ms}$$

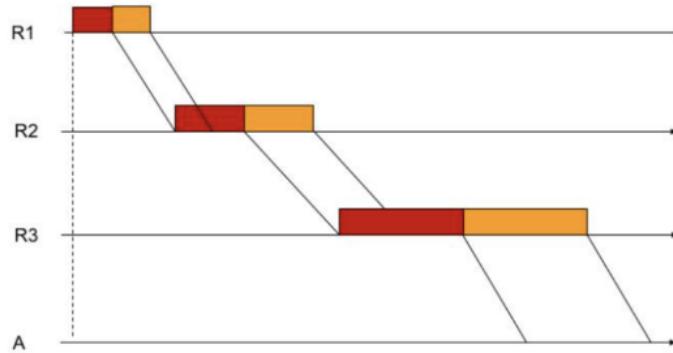
4. Si consideri la rete nella seguente figura, dove  $C_1, C_2, C_3$  e  $\tau_1, \tau_2, \tau_3$  sono rispettivamente i transmission rate e i delay di propagazione dei tre link.

Al tempo  $t = 0$ , la coda di uscita di  $R_1$  contiene 2 pacchetti diretti ad  $A$ . Assumendo che la lunghezza dei pacchetti sia  $L = 512 \text{ b}$ , si indichi per ciascun pacchetto l'istante in cui esso viene completamente ricevuto da  $A$ .

Inoltre, supponendo che vi siano  $n$  pacchetti, si indichi una formula generica descrivente per ciascun pacchetto l'istante in cui esso viene completamente ricevuto da  $A$



Come nel caso precedente, tracciamo il contenuto di ogni coda al passare del tempo:



Il tempo totale impiegato dal primo pacchetto corrisponderà a:

$$T_1 = \frac{L}{C_1} + \tau_1 + \frac{L}{C_2} + \tau_2 + \frac{L}{C_3} + \tau_3 = 1 \text{ ms} + 1 \text{ ms} + 2 \text{ ms} + 2 \text{ ms} + 8 \text{ ms} + 4 \text{ ms} = 18 \text{ ms}$$

Notiamo come, a differenza del caso precedente, il secondo pacchetto giunge nelle code successive mentre il primo pacchetto deve essere ancora completamente spedito, implicando che esso debba essere inserito nella coda di attesa.

Dunque, una volta raggiunta la coda finale, il secondo pacchetto potrà essere inviato solo una volta completato il primo pacchetto.

Di conseguenza, il suo tempo totale di ricezione corrisponde a:

$$T_2 = T_1 + \frac{L}{C_3} = 18 \text{ ms} + 8 \text{ ms} = 26 \text{ ms}$$

Applicando lo stesso ragionamento nel caso di  $n$  pacchetti, la formula generica descrivente l'istante di ricezione dell' $n$ -esimo pacchetto corrisponde a:

$$T_n = T_{n-1} + \frac{L}{C_3} = T_{n-2} + 2 \cdot \frac{L}{C_3} = \dots = T_1 + (n-1) \frac{L}{C_3} = 18 \text{ ms} + 8(n-1) \text{ ms}$$

## 1.5 Stack protocollare TCP/IP

### Definition 12. Protocollo

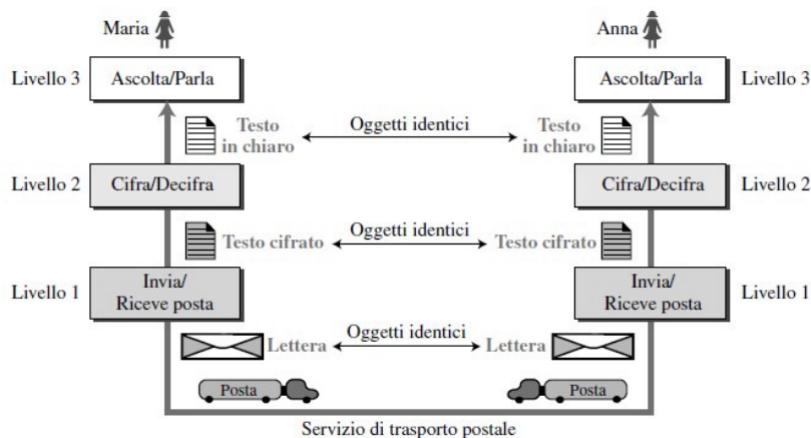
Un **protocollo** definisce l'insieme di **regole** che il dispositivo mittente e il dispositivo destinatario, così come tutti i sistemi intermedi coinvolti, devono rispettare per essere in grado di comunicare.

In situazioni più complesse, potrebbe essere opportuno suddividere i compiti necessari alla comunicazione fra **più livelli (layer)**, nel qual caso è richiesto **un protocollo per ciascun livello**. Tramite un layering dei protocolli, dunque, è possibile suddividere un compito complesso in compiti più semplici, ognuno gestibile da un singolo protocollo.

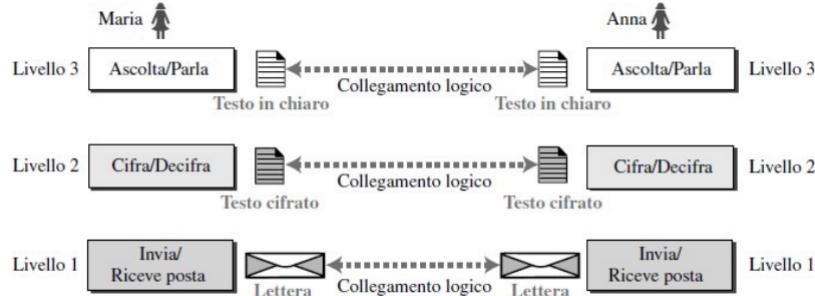
In particolare, ogni layer è **indipendente dagli altri** (modularizzazione), utilizzando i servizi forniti dal layer inferiore e offrendo servizi al layer superiore.

Ogni layer, dunque, può essere considerato come una **black box** con opportuni ingressi ed uscite, senza necessità di essere a conoscenza delle modalità con cui i dati in ingresso vengano trasformati in quelli di uscita.

Quando è richiesta una **comunicazione bidirezionale**, ciascun layer deve essere in grado di effettuare entrambi i compiti richiesti, ossia manipolare i dati in input per inviarli al livello superiore o manipolarli per inviarli al livello inferiore.



In particolare, l'effetto ottenuto tramite una suddivisione in uno stack di layer equivalenti permette l'instaurazione di un **collegamento logico** tra ogni livello dello stack: il protocollo implementato in ciascun livello specifica una comunicazione diretta tra i pari livelli delle due parti: il layer  $N$  di un dispositivo comunica solo ed esclusivamente con il layer  $N$  di tutti i dispositivi.



Inoltre, per via dell'estrema modularizzazione ottenuta, viene facilitata la manutenzione e l'aggiornamento del sistema, poiché il cambiando dell'implementazione del servizio di un layer rimane trasparente al resto del sistema.

### Definition 13. Stack protocollare TCP/IP

La principale forma di stack protocollare utilizzata corrisponde allo **stack protocolare TCP/IP**, la cui struttura a layer corrisponde a:

- **Livello di Applicazione**, il quale fornisce supporto alle applicazioni facente uso della rete (protocolli HTTP, SMTP, FTP, DNS, ...).
- **Livello di Trasporto**, il quale gestisce il trasferimento dei pacchetti dal processo del dispositivo mittente a quello del dispositivo destinatario (protocolli TCP, UDP, ...).
- **Livello di Rete**, il quale gestisce l'instradamento dei pacchetti dall'origine alla destinazione (protocolli IP, ...).
- **Livello di Collegamento (o Link)**, il quale gestisce la trasmissione dei pacchetti da un nodo a quello successivo sul percorso (protocolli Ethernet, Wi-fi, PPP, ...). Lungo il percorso, un pacchetto può essere gestito da protocolli diversi.
- **Livello Fisico**, dove avviene il vero e proprio trasferimento dei singoli bit

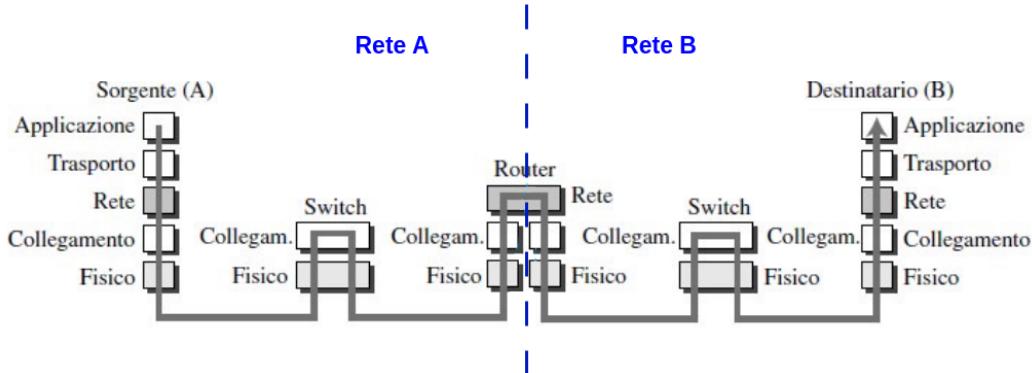


I livelli di Applicazione e di Trasporto sono gestiti tramite **software**, mentre i livelli di Collegamento e Fisico tramite **hardware**.

Durante l'invio di un pacchetto, quest'ultimo, partendo dal livello applicazione del dispositivo sorgente, **percorre tutti i layer dello stack protocollare**, fino a giungere al livello fisico, dove viene effettivamente inviato al nodo successivo.

Tutti i nodi intermedi presenti sul percorso lavoreranno utilizzando solo i livelli necessari. In particolare, ogni dispositivo utilizzerà il livello di collegamento, in modo da poter spedire il pacchetto stesso verso il nodo successivo del percorso.

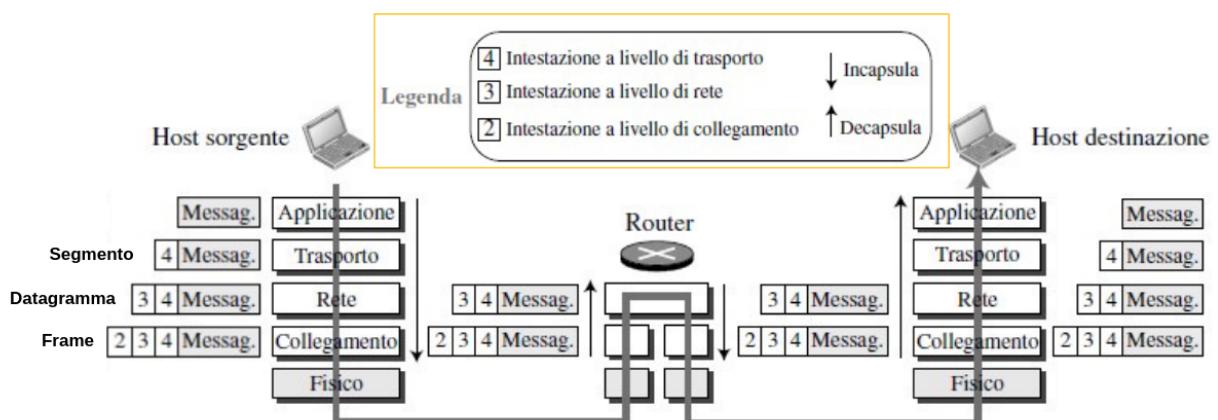
Nel caso in cui si raggiunga il **punto di scambio tra due reti**, solitamente un edge router, verrà utilizzato anche il livello di rete.



Prima di essere spedito al livello inferiore, ogni pacchetto viene **incapsulato**: una volta ricevuto il pacchetto dal layer superiore, il layer attuale applica un proprio **header (o intestazione)**, aggiungendo informazioni necessarie al layer del dispositivo di destinazione corrispondente a quello attuale.

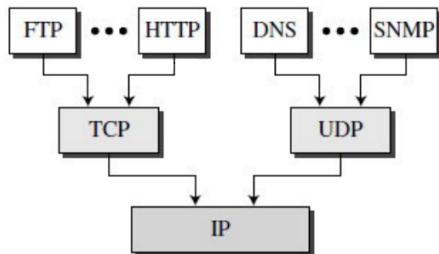
In particolare, ad ogni livello un pacchetto assume il nome di:

- **Messaggio (al livello di applicazione)**, corrispondente al pacchetto originale, senza alcuna intestazione
- **Segmento (al livello di trasporto)**, corrispondente al messaggio ricevuto dal layer superiore a cui viene aggiunto un header di trasporto
- **Datagramma (al livello di rete)**, corrispondente al segmento ricevuto dal layer superiore a cui viene aggiunto un header di rete
- **Frame (al livello di collegamento)**, corrispondente al datagramma ricevuto dal layer superiore a cui viene aggiunto un header di collegamento

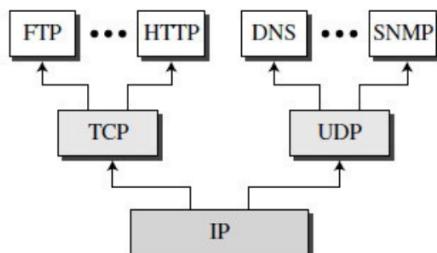


Poiché lo stack protocollare TCP/IP prevede la presenza di **più protocolli nello stesso livello**, ogni livello deve essere in grado di effettuare operazioni di:

- **Multiplexing**, dove ogni protocollo deve essere in grado di encapsulare (uno alla volta) i pacchetti ricevuti da più protocolli presenti al livello superiore



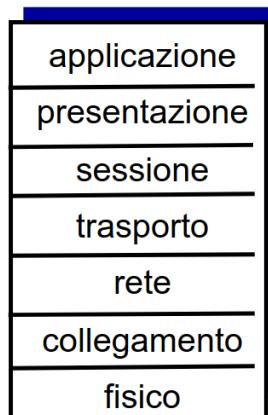
- **Demultiplexing**, dove ogni protocollo deve essere in grado di decapsulare i pacchetti ricevuti ed inviarli a più protocolli presenti nel livello superiore



Per realizzare ciò, nell'header di ogni layer viene inserito un **campo speciale** in grado di identificare quale sia il protocollo di appartenenza di tale pacchetto.

Un'evoluzione dello stack protocollare TCP/IP è il **modello Open Systems Interconnection (OSI)**, dove vengono interposti due livelli tra il livello di applicazione e il livello di trasporto:

- **Livello di Presentazione**, utilizzato per consentire alle applicazioni di interpretare i dati (es: crittografia, compressione, ...)
- **Livello di Sicurezza**, utilizzato per gestire servizi come la sincronizzazione o il ripristino dello scambio di dati



# Capitolo 2

## Livello di Applicazione

### 2.1 Principi delle applicazioni di rete

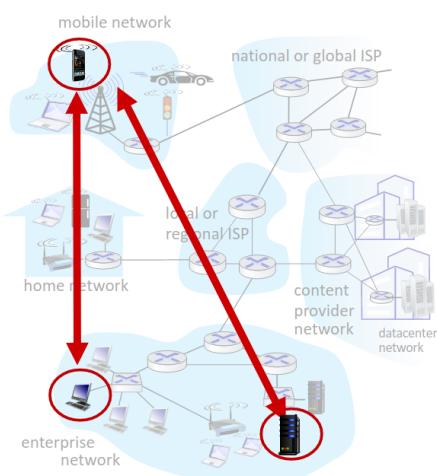
#### Definition 14. Paradigma di comunicazione

Un **paradigma di comunicazione** è una metodologia di scambio informazioni e gestione delle connessioni all'interno di una rete, principalmente all'interno di Internet.

In particolare, i due principali paradigmi utilizzati sono:

- **Paradigma Client-Server**, dove i sistemi terminali vengono divisi in due categorie:
  - **Client**, il quale comunica solo ed esclusivamente con un server, **richiedendo dei servizi** a quest'ultimo, e può rimanere anche inattivo se non necessario, implicando che esso possa avere indirizzi IP dinamici nel tempo. In particolare, per tali caratteristiche, non vi è una comunicazione client-client, ma solo una comunicazione client-server-client
  - **Server**, il quale possiede un indirizzo IP permanente, rimanendo sempre attivo in attesa di **fornire servizi** ai vari client richiedenti

Ad esempio, i protocolli HTTP, FTP e IMAP sono basati su tale paradigma

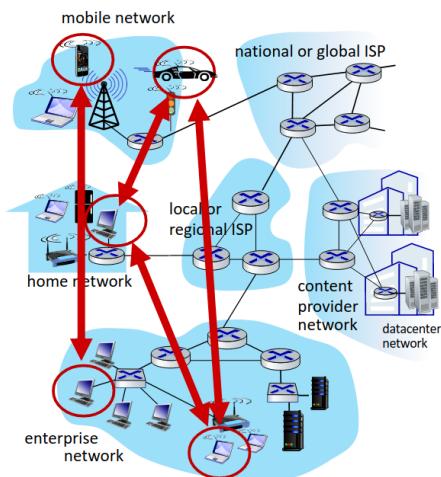


- **Paradigma Peer-to-Peer (P2P)**, dove i sistemi terminali vengono detti **peer** (tradotto: *pari, di equal importanza*) ed ognuno di essi è in grado di comunicare direttamente con ogni altro peer (assumendo quindi sia il compito di client che di server).

In particolare, ogni peer **richiede e fornisce servizi ad altri peer**, rendendo il sistema **estremamente scalabile**: ogni nuovo peer incrementa le capacità di servizio e le richieste di servizio.

Inoltre, come nel caso dei client, ogni peer può connettersi alla rete in modo intermittente utilizzando IP dinamici, diminuendo temporaneamente la quantità di servizi fornibili nella rete. Di conseguenza, la loro gestione risulta estremamente più complessa, ma anche più performante nel caso di un numero elevato di peer.

Un classico esempio di utilizzo del paradigma P2P risultano essere i vari protocolli legati al torrenting e alla condivisione di file di grandi dimensioni.



### Definition 15. Processo

Un **processo** è un programma in esecuzione all'interno di un sistema terminale.

In particolare, un **processo client** è un processo che avvia una comunicazione, mentre un **processo server** è un processo che attende di essere contattato da un processo client.

All'interno dello stesso sistema, due processi comunicano tra di loro utilizzando una comunicazione **inter-process**, definita dal sistema operativo. I processi situati su sistemi diversi, invece, comunicano tra di loro tramite **scambio di messaggi**

### Definition 16. Socket

Un **socket** è un'**astrazione software** tramite cui un processo può inviare e ricevere messaggi tramite il socket di un altro processo. Per poter comunicare, dunque, due processi devono connettersi tramite due socket (uno ciascuno), identificati da una coppia <Indirizzo\_IP, Numero\_Porta>

Ogni protocollo a livello di applicazione definisce:

- Le tipologie di messaggi scambiati (es: richiesta e risposta)
- La sintassi del messaggio
- La semantica del messaggio
- Le regole per come e quando i processi inviano e rispondono ai messaggi

In particolare, i protocolli a tale livello si differenziano in **protocolli aperti**, ossia definiti secondo uno standard pubblico ed adottato comunemente da ogni applicazione (es: HTTP, FTP, ...), e **protocolli proprietari**, ossia non pubblici e fini all'applicazione stessa (es: Skype, ...).

Per poter funzionare correttamente, ogni applicazione di rete necessita di alcuni **servizi di trasporto**. In particolare, esse possono necessitare di:

- **Integrità dei dati**, ossia un trasferimento dei dati affidabile al 100%, senza alcuna perdita di pacchetto o corruzione dei dati
- **Garanzie temporali**, ossia un basso ritardo per la ricezione dei dati
- **Garanzie di throughput**, ossia una quantità minima di throughput dati
- **Sicurezza**, ad esempio crittografia o integrità dei dati a seguito di manomissioni

### Definition 17. Transmission Control Protocol (TCP)

Il **Transmission Control Protocol (TCP)** è un protocollo risiedente sul **layer di trasporto** in grado di fornire **trasporto affidabile**, ossia senza perdita di alcun pacchetto, e controllo del flusso e della congestione, in cambio di un'assenza di garanzie temporali, di throughput e di sicurezza.

Inoltre, il protocollo TCP è **orientato alla connessione**, ossia richiedente una configurazione (**handshaking**) tra il processo client e il processo server

### Definition 18. User Datagram Protocol (UDP)

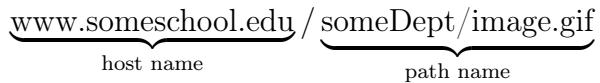
L'**User Datagram Protocol (UDP)** è un protocollo risiedente sul **layer di trasporto** in grado di fornire **trasporto veloce** poiché **non orientato alla connessione** ed **estremamente scarno**, ossia sprovvisto di: trasporto affidabile, controllo del flusso e della congestione e garanzie temporali, di throughput e di sicurezza

Poiché per loro natura i protocolli TCP ed UDP sono privi di garanzie di sicurezza, i messaggi scambiati tra socket TCP e UDP risultano sprovvisti di crittografia, attraversando il percorso instradato completamente in chiaro ed essendo quindi leggibili e manipolabili da qualsiasi dispositivo intermedio.

Per ovviare tale problema, viene implementato a livello di applicazione il protocollo **Transport Layer Security (TLS)** tramite socket realizzati con librerie software specifiche, fornendo connessioni crittografate, integrità dei dati ed autenticazione dell'end-point.

## 2.2 Web e Protocollo HTTP

Una pagina web è composta da **oggetti**, ognuno dei quali può essere archiviato su un diverso web server. In particolare, una pagina web consiste in un **file HTML** il quale include diversi oggetti referenziati tramite vari URL



### Definition 19. Protocollo HTTP

Il **protocollo HTTP (Hypertext Transfer Protocol)** è un protocollo a livello di applicazione utilizzato per la realizzazione di servizi web. La sua porta di riferimento comune all'interno dei socket è la **porta 80**.

Il protocollo HTTP è **stateless**, ossia non conservante alcuna informazione sulle richieste passate, e basato sul **paradigma client-server**, dove il client invia messaggi detti **richieste** e il server invia messaggi detti **risposte**.

Inoltre, il protocollo HTTP fa uso del **protocollo TCP**:

1. Il client avvia una connessione TCP con il server utilizzando la porta 80, rimanendo in attesa che il server accetti la connessione (TCP handshaking)
2. Vengono scambiati messaggi HTTP tra client e server
3. La connessione TCP viene chiusa

Le **connessioni HTTP** si differenziano in due tipologie:

- **Connessione non persistente**, dove viene aperta la connessione TCP e viene inviato massimo un oggetto prima di chiudere la connessione TCP
- **Connessione persistente (HTTP/1.1)**, dove viene aperta la connessione TCP e vengono inviati multipli oggetti in successione prima di chiudere la connessione TCP

**Esempio:**

1. Supponiamo che un utente inserisca l'URL dell'oggetto "www.someSchool.edu/ someDepartment/home.index", contenente del testo e 10 riferimenti ad immagini.
2. Il client HTTP dell'utente (browser, cURL, ...) avvia la connessione TCP con il server HTTP tramite la porta 80
3. Il server HTTP sull'host "www.someSchool.edu" riceve la richiesta di connessione, accettandola e notificando il client
4. Il client HTTP invia un messaggio di richiesta HTTP, contenente il path dell'oggetto desiderato, ossia "/someDept/index.html"
5. Il server HTTP riceve il messaggio di richiesta e invia il messaggio di risposta contenente l'oggetto desiderato, il quale a sua volta contiene i riferimenti alle 10 immagini.

6. A questo punto, si creano due scenari:

- Se la connessione non è persistente, il server chiude immediatamente la connessione TCP, implicando che l'intero processo debba essere ripetuto per tutti e 10 i riferimenti necessari
- Se la connessione è persistente, il client invierà in successione altre 10 richieste al server, richiedendo quindi solo la ripetizione dei passaggi 4 e 5 (per 10 volte), per poi chiudere la connessione TCP

### Definition 20. Round Trip Time (RTT)

Definiamo come **Round Trip Time (RTT)** il tempo impiegato da un pacchetto di piccole dimensioni per compiere il percorso client-server-client

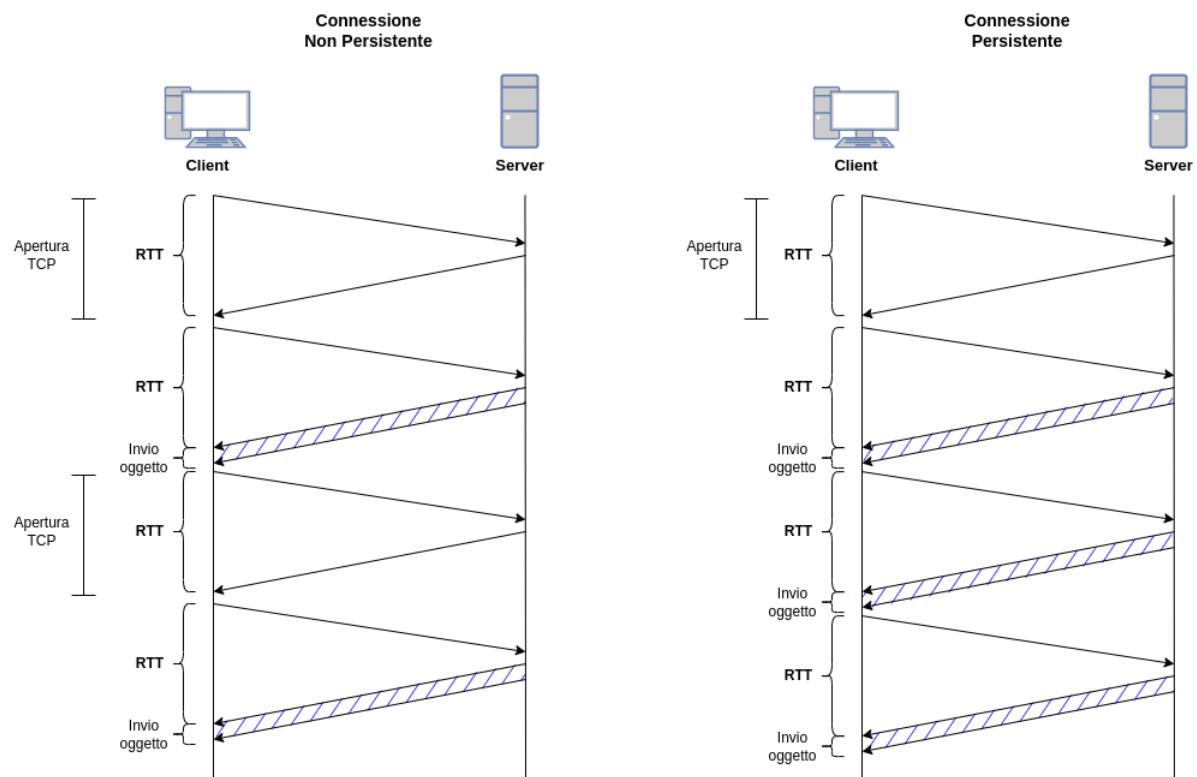
### Observation 2. Tempo di risposta HTTP

Se la **connessione non è persistente**, per ogni oggetto sono necessari due RTT, uno per avviare la connessione TCP ed uno per l'invio di richiesta e risposta, seguiti dal tempo necessario ad inviare l'oggetto

$$T_{tot} = (2 \text{ RTT} + \text{Tempo invio ogg.}) \cdot \text{Num. Oggetti}$$

Se la **connessione è persistente**, invece, saranno necessari un RTT per poter stabilire la connessione TCP, seguiti da un solo RTT per oggetto (con annesso tempo di invio)

$$T_{tot} = 1 \text{ RTT} + (1 \text{ RTT} + \text{Tempo invio ogg.}) \cdot \text{Num. Oggetti}$$



### 2.2.1 Messaggi di richiesta e risposta

I messaggi HTTP di **richiesta** e **risposta** vengono formattati un formato leggibile dall'uomo (in particolare, in codice ASCII).

Ogni messaggio di **richiesta HTTP** viene strutturato nel seguente modo:

- Una **riga di richiesta**, composta dal **metodo** utilizzato, il path richiesto e la versione di HTTP utilizzata, seguiti da un carattere di ritorno a capo, ossia \r, ed un carattere di avanzamento di riga, ossia \n

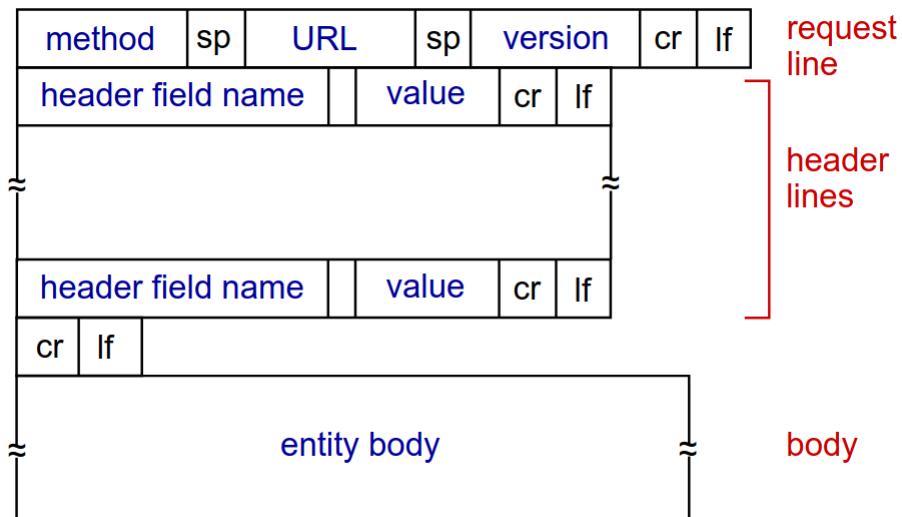
I **metodi** principali inseribili all'interno della riga di richiesta sono:

- **Metodo GET**, utilizzato per l'invio di dati al server, i quali vengono inseriti all'interno dell'URL a seguito di un carattere '?'  
(es: `www.mysite.com/search?user=myuser`)
- **Metodo POST**, utilizzato per l'invio di dati al server, i quali vengono aggiunti all'interno del body del messaggio (rimanendo quindi parzialmente offuscati all'utente)
- **Metodo HEAD**, utilizzato per richiedere solo l'header di risposta che verrebbe restituito dalla destinazione a seguito di una richiesta GET
- **Metodo PUT**, utilizzato per caricare un nuovo file o sostituirne uno esistente all'interno della destinazione (non più utilizzato poiché estremamente insicuro)
- Un **header (o intestazione)**, composto da varie linee contenenti informazioni utili alla connessione

Alcuni esempi di campi inseribili all'interno di un header di richiesta sono:

Campo Header	Descrizione
User-agent	Indica il programma client utilizzato
Accept	Indica il formato dei contenuti che il client è in grado di accettare
Accept-charset	Famiglia di caratteri che il client è in grado di gestire
Accept-encoding	Schema di codifica supportato dal client
Accept-language	Linguaggio preferito dal client
Authorization	Indica le credenziali possedute dal client
Host	Host e numero di porta del client
Date	Data e ora del messaggio
Upgrade	Specifica il protocollo di comunicazione preferito
Cookie	Comunica un cookie al server
If-Modified-Since	Invia il documento solo se è più recente della data specificata

- Un **body (o contenuto)**, ossia il vero contenuto del messaggio da inviare (solitamente vuoto a meno dell'uso del metodo POST)



Esempio:

```
GET /index.html HTTP/1.1\r\n
Host: www-net.cs.umass.edu\r\n
User-Agent: Firefox/3.6.10\r\n
Accept: text/html,application/xhtml+xml\r\n
Accept-Language: en-us,en;q=0.5\r\n
Accept-Encoding: gzip,deflate\r\n
Accept-Charset: ISO-8859-1,utf-8;q=0.7\r\n
Keep-Alive: 115\r\n
Connection: keep-alive\r\n
\r\n
```

Analogamente, ogni messaggio di **risposta HTTP** viene strutturato in modo simile, ma con alcune differenze:

- Una **riga di stato**, composta dalla versione di HTTP utilizzata, un **codice di status** e una **frase di status** descrivente in breve il codice di status

I **codici di status** si dividono in 5 categorie:

- **Codici 1xx**, indicanti che la risposta ricevuta contiene solamente informazioni  
(es: 100 Continue indica che il server è pronto a ricevere la richiesta del client)
- **Codici 2xx**, indicanti che la richiesta effettuata è andata a buon fine  
(es: 200 OK indica che la richiesta ha avuto successo e l'oggetto richiesto è stato trovato, 204 No Content indica che la richiesta ha avuto successo ma l'oggetto richiesto non contiene nulla al suo interno)
- **Codici 3xx**, indicanti che è stato effettuato un reindirizzamento a seguito della richiesta effettuata  
(es: 301 Moved Permanently indica che l'oggetto richiesto possiede un path diverso da quello richiesto, reindirizzando automaticamente tutte le richieste successive del client)

- **Codici 4xx**, indicanti un errore nella richiesta del client  
(es: **403 Forbidden** indica che il client non possiede i requisiti per accedere all'oggetto richiesto, **404 Not Found** indica che l'oggetto richiesto non esiste )
- **Codici 5xx**, indicanti un errore per cui il server non è riuscito a completare la richiesta  
(es: **500 Internal Server Error** indica un errore sconosciuto all'interno del server, **503 Service Unavailable** indica che il server è attualmente non disponibile)
- Un **header (o intestazione)**, composto da varie linee contenenti informazioni utili alla risposta

Alcuni esempi di campi inseribili all'interno di un header di risposta sono:

Campo Header	Descrizione
Date	Data e ora attuale
Upgrade	Specifica il protocollo di comunicazione preferito
Server	Indica il programma server utilizzato
Set-Cookie	Il server richiede al client di memorizzare un cookie
Content-Encoding	Specifica lo schema di codifica
Content-Language	Specifica la lingua utilizzata nel documento
Content-Length	Indica la lunghezza del documento
Content-Type	Specifica la tipologia del documento
Location	Chiede al client di inviare la richiesta ad un altro sito
Last-modified	Fornisce data e ora dell'ultima modifica del documento

- Un **body (o contenuto)**, ossia il vero contenuto del messaggio da restituire (in particolare, l'oggetto richiesto)

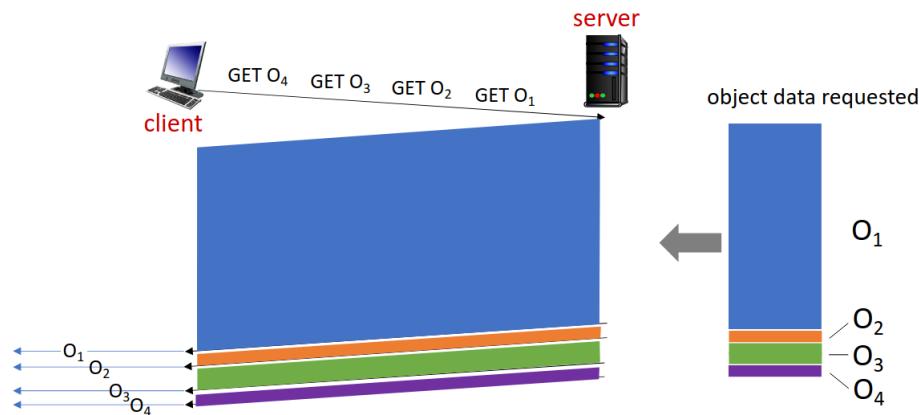
Esempio:

```
HTTP/1.1 200 OK\r\n
Date: Sun, 26 Sep 2010 20:09:20 GMT\r\n
Server: Apache/2.0.52 (CentOS)\r\n
Last-Modified: Tue, 30 Oct 2007 17:00:02 GMT\r\n
Accept-Ranges: bytes\r\n
Content-Length: 2652\r\n
Keep-Alive: timeout=10, max=100\r\n
Connection: Keep-Alive\r\n
Content-Type: text/html; charset=ISO-8859-1\r\n
\r\n
[document content...]
[...]
[document content...]
```

### 2.2.2 Versioni di HTTP

Come già discusso, il **protocollo HTTP/1.1** ha introdotto la possibilità di poter effettuare più richieste GET in successione tramite una singola connessione TCP. Tuttavia, tale modifica ha introdotto ulteriori problematiche:

- Il server risponde alle richieste GET nell'ordine in cui vengono effettuate (**First Come First Served (FCFS)**)
- Un oggetto di piccole dimensioni potrebbe dover attendere la trasmissione di oggetti di grandi dimensioni richiesti prima di esso (**blocco head-of-line (HOL)**)
- La perdita di un segmento TCP causa lo stallo del trasferimento di un oggetto



Per risolvere tali problematiche, il **protocollo HTTP/2** introduce una maggiore flessibilità al server nell'invio di oggetti al client:

- L'ordine di trasmissione degli oggetti richiesti viene stabilito in base alla priorità dell'oggetto specificata dal client
- Gli oggetti vengono **divisi in frame**, schedulati in modo da mitigare il blocco HOL
- Possono essere inviati più oggetti contemporaneamente (**multiplexing**)



Il **protocollo HTTP/3**, invece, risolve le ultime problematiche rimanenti all'interno del protocollo HTTP/2, tramite l'aggiunta di controlli sulla sicurezza, sugli errori e sulla congestione per oggetto, utilizzando il **protocollo QUIC** (basato su UDP) al posto del protocollo TCP.

### 2.2.3 Cookies e Web Caching

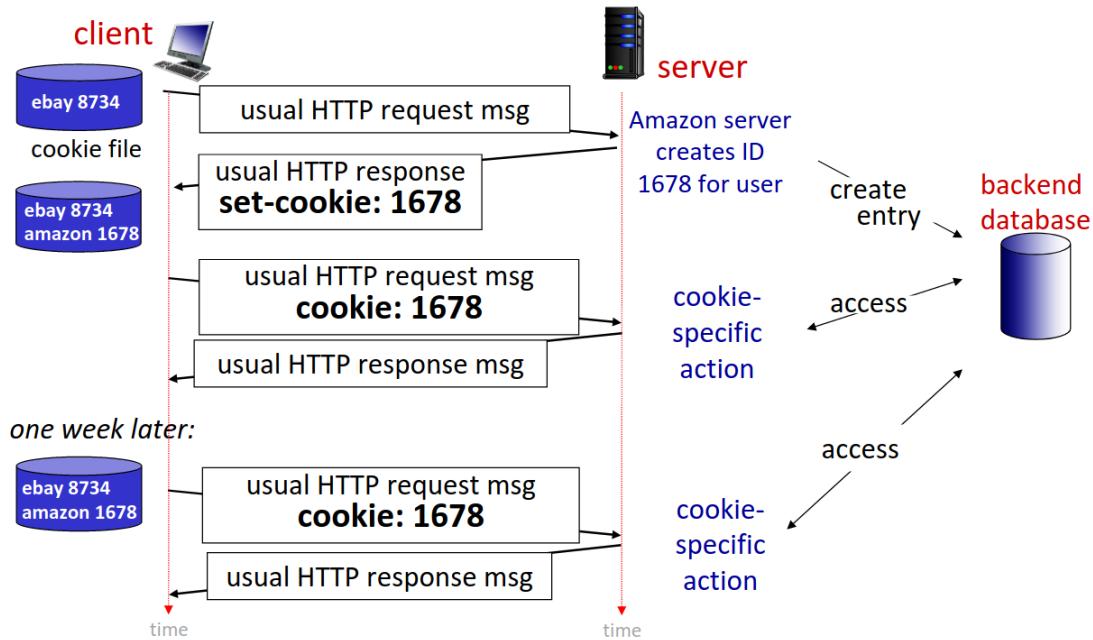
#### Definition 21. Cookie

Un **cookie** è un **piccolo file di testo** contenente brevi informazioni (preferenze sull'utilizzo, parametri preferiti, token di autorizzazione, ...) salvato all'interno di un client da parte di un server web

Poiché il protocollo HTTP è un protocollo **stateless**, i cookie vengono utilizzati all'interno delle applicazioni web per conservare indirettamente alcune informazioni sulle varie comunicazioni client-server effettuate, rendendo ogni richiesta HTTP indipendente dall'altra.

A seguito di un messaggio di risposta da un web server contenente il campo header **Set-Cookie**, il client salva il contenuto del cookie all'interno di un file. Durante le **succesive richieste** effettuate dallo stesso client allo stesso server, tutti i cookie impostati da tale server vengono **allegati ad ogni richiesta HTTP**.

Soltanmente, il cookie fornito dal server contiene un ID univoco, in modo da legare una voce nel suo database interno a quel client specifico.



La **durata di un cookie** inviato viene specificata tramite un campo header **Max-Age**, tramite il quale viene specificato il tempo di vita di tale cookie in **secondi**. Allo scadere di tali secondi, il client eliminerà automaticamente tale cookie. Inoltre, non c'è limite alla quantità di secondi specificabili, implicando che sia possibile specificare anche una quantità di secondi pari a mesi o anni

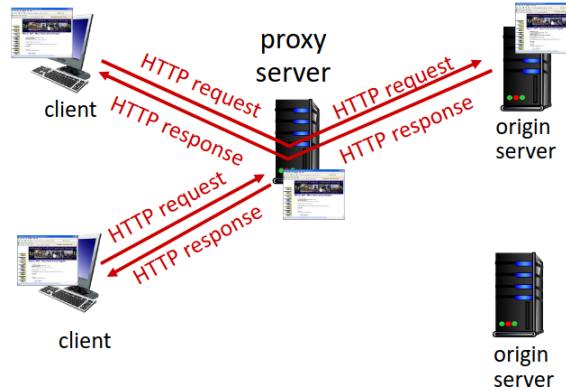
### Definition 22. Proxy Server

Un **proxy server** è un server utilizzato come **intermediario** tra un client e il vero server destinatario.

Solitamente, tale tipologia server viene utilizzato per il **web caching**:

- Se il documento richiesto **è presente** nella cache del proxy server, esso viene restituito al client senza dover raggiungere il server originale
- Se il documento richiesto **non è presente** nella cache, il proxy server inoltra la richiesta del client al server di origine, memorizzando nella sua cache il documento ricevuto nella risposta, restituendolo al client

Tramite il web caching è possibile ridurre notevolmente i tempi di risposta e il traffico nei link di accesso alla rete del server di origine, consentendo ai fornitori di contenuti di essere più efficienti.

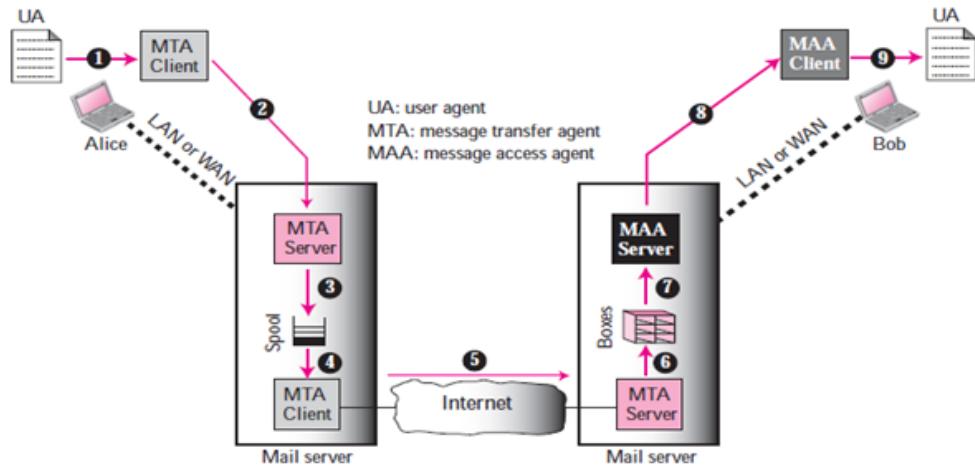


## 2.3 Posta elettronica

Il servizio di **posta elettronica** è costituito da tre entità fondamentali:

- Uno **User agent (UA)**, detto anche *mail reader*, è un processo attivo sul client utente attivato dall'utente stesso o da un timer. Si occupa di informare l'utente nel caso in cui sia disponibile una nuova email da leggere nella sua casella di posta.
- Inoltre, lo user agent permette la composizione, l'editing, l'invio e la lettura di messaggi di posta elettronica. Ogni messaggio di posta inviato da un UA viene passato ad un MTA
- **Mail Transfer Agent (MTA)**, è un processo attivo su un mail server utilizzato per il trasferimento Internet di un messaggio ricevuto da un UA o da un altro MTA
- **Mail Access Agent (MAA)**, è un processo attivo su un mail server utilizzato per leggere i messaggi di posta in arrivo

Ogni **mail server** è dotato di una **casella di posta (mailbox)**, contenente i messaggi in arrivo per l'utente, ed una **coda di messaggi**, contenente i messaggi dell'utente ancora da inviare.



### 2.3.1 Protocolli SMTP e MIME

#### Definition 23. Protocollo SMTP

Il **protocollo SMTP** (Simple Mail Transfer Protocol) è un protocollo a livello di applicazione utilizzato per l'invio di messaggi di posta elettronica in formato ASCII. La sua porta di riferimento comune all'interno dei socket è la **porta 25**.

Il protocollo SMTP effettua un **trasferimento diretto**, ossia dal mail server mittente a quello destinatario (dunque senza mail server intermedi), basato su un'**interazione comando/risposta**: viene inviato un comando in testo ASCII e viene ricevuta una risposta equivalente ad un codice di stato.

Inoltre, il protocollo SMTP fa uso del **protocollo TCP**:

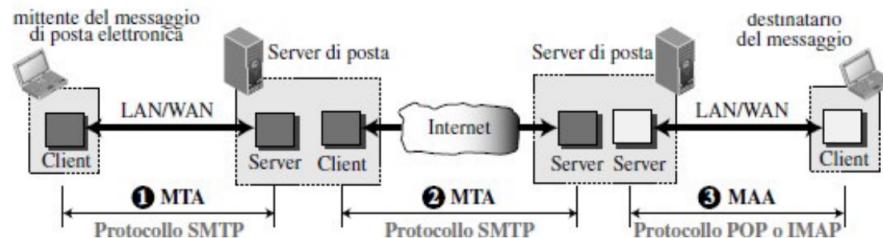
1. Il client avvia una connessione TCP con il server utilizzando la porta 25, rimanendo in attesa che il server accetti la connessione (TCP handshaking)
2. Vengono scambiati messaggi di posta tra client e server (**connessione persistente**)
3. La connessione TCP viene chiusa

#### Esempio:

1. Alice usa il suo UA per comporre il messaggio da inviare all'indirizzo di posta elettronica `bob@someschool.edu`
2. L'UA di Alice invia il messaggio al mail server di Alice, il quale porrà tale messaggio nella sua coda di messaggi. Successivamente, il client SMTP presente sul mail server di Alice apre una connessione TCP con il mail server di Bob
3. Il client SMTP invia il messaggio di Alice sulla connessione TCP tramite il suo MTA

4. Il mail server di Bob riceve il messaggio e lo pone nella casella di posta di Bob
5. Bob invoca il suo UA per leggere il messaggio, il quale preleverà il messaggio tramite l'MAA presente sul suo mail server

(NB: tale operazione *non* è svolta dal protocollo SMTP, bensì dal protocollo POP3 o dal protocollo IMAP che vedremo in seguito)



In particolare, lo scambio di messaggi viene gestito dal protocollo SMTP nel seguente modo:

1. Il client SMTP **tenta di stabilire** una connessione TCP sulla porta 25 con il server STMP. Se il server è attivo, la connessione TCP viene stabilita. Altrimenti, il client riproverà dopo un determinato lasso di tempo.
2. Una volta stabilita la connessione, il client e il server effettuano una **forma aggiuntiva di handshaking**, dove il client indica al server l'indirizzo email del mittente e del destinatario
3. Il client invia il messaggio sulla connessione TCP. Una volta ricevuto il messaggio, se ci sono altri messaggi da inviare viene utilizzata la stessa connessione TCP (**connessione persistente**). Altrimenti, il client invia al server una richiesta di chiusura della connessione.

#### Esempio:

- Di seguito, vediamo un esempio di interazione tra un server SMTP, indicato con S, e un client SMTP, indicato con C.

```

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection
  
```

Lo **standard RFC 822** definisce la struttura che ogni messaggio di posta elettronica deve assumere:

- Un **header** composto dai seguenti campi:

Campo Header	Descrizione
To	L'indirizzo del destinatario
From	L'indirizzo del mittente
CC	Indirizzi aggiuntivi di mittenti a cui far sapere dell'invio e il contenuto di tale email (abbreviativo di Carbon Copy)
BCC	Analogo al CC, ma non vengono mostrati al destinatario (abbreviativo di Blind CC)
Subject	L'argomento del messaggio
Sender	Il nome del mittente

- Un **body**, contenente il messaggio da inviare (**solo caratteri ASCII**)

Per poter inviare contenuti diversi dal semplice test ASCII, gli standard RFC 2045 e 2046 definiscono il **protocollo MIME (Multipurpose Internet Mail Extension)**, in grado di estendere i normali messaggi di posta elettronica in messaggi multimediali.

Vengono aggiunte alcune righe all'interno dell'header del messaggio inviato, in particolare una riga **Version**, indicante la **versione del protocollo** MIME utilizzata, e una riga **Type**, indicante il **tipo di dati multimediali** inviati, i quali, prima di essere spediti, vengono convertiti in una **codifica testuale** (solitamente base64), specificata da un campo aggiuntivo **Content-Transfer-Encoding**, in modo da poter essere trasmesso sottoforma di testo ASCII, per poi venir decodificati una volta che il messaggio è giunto al destinatario.

### 2.3.2 Protocolli POP3 e IMAP

#### Definition 24. Protocollo POP3

Il **protocollo POP3 (Post Office Protocol vers. 3)** è un protocollo **stateless** a livello di applicazione utilizzato per il download di messaggi di posta elettronica ricevuti. La sua porta di riferimento comune all'interno dei socket è la **porta 110**.

Per stabilire una connessione, il protocollo POP3 fa uso del **protocollo TCP**, effettuando quindi l'handshake TCP, per poi procedere nelle seguenti tre fasi:

1. **Autorizzazione**, dove lo UA invia nome utente e password per essere identificato dal mail server
2. **Transazione**, dove lo UA recupera i messaggi nella casella di posta dell'utente
3. **Aggiornamento**, dove, successivamente all'invio di un messaggio QUIT da parte dello UA, viene terminata la connessione e vengono rimossi dal mail server i messaggi contrassegnati durante la fase precedente

**Esempio:**

- Se la richiesta effettuata viene eseguita correttamente, il server risponderà con +OK, altrimenti con -ERR. Il comando retr permette di scaricare il messaggio, mentre il comando dele permette di marcare i messaggi da eliminare

```

S: +OK POP3 server ready
C: user rob
S: +OK
C: pass hungry
S: +OK user successfully logged on
C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off

```

- Successivamente, viene attivata la fase di aggiornamento, cancellando dal mail server i messaggi marcati tramite dele

Oltre ad essere un protocollo stateless, il protocollo POP3 non fornisce all'utente la possibilità di creare **cartelle remote** tra cui poter suddividere i messaggi, costringendo la creazione di tali cartelle solo a livello locale, implicando che esse non siano condivise tra i vari dispositivi dell'utente.

### Definition 25. Protocollo IMAP

Il **protocollo IMAP (Internet Message Access Protocol)** è un protocollo a livello di applicazione utilizzato per l'accesso ai messaggi di posta elettronica ricevuti. La sua porta di riferimento comune all'interno dei socket è la **porta 143**.

A differenza del protocollo POP3, tutti i messaggi vengono **conservati nel mail server**, permettendo all'utente di avere solo copie locali. Per via di ciò, il protocollo IMAP permette di:

- Associare ogni messaggio ricevuto ad una cartella, detta **inbox**
- Creare cartelle remote e spostare messaggi tra di esse
- Effettuare ricerche nelle cartelle remote
- Conservare lo stato tra le varie sessioni dell'utente (protocollo **stateful**)

## 2.4 Domain Name System (DNS)

### Definition 26. Domain Name System (DNS)

Il **Domain Name System (DNS)** è un sistema utilizzato per mappare singoli nodi di una rete ad un **nome** che li identifichi. Viene realizzato tramite un database distribuito, implementato come una gerarchia di **name server**.

Tra le funzioni fornite dal servizio DNS troviamo:

- **Traduzione** da nome host all'indirizzo IP relativo
- **Distribuzione del carico**, permettendo a più indirizzi IP, ognuno legato ad un server copia di quello originale, di corrispondere ad un unico nome. Quando un client effettua una richiesta, il servizio restituisce l'insieme di indirizzi legati a tale nome in un ordine casuale (rotazione DNS)
- **Host Aliasing**, ossia l'associazione di più sinonimi (alias) allo stesso indirizzo IP, permettendo l'associazione di un nome più semplice rispetto ad uno complesso  
(es: al nome `relay1.west-coast.enterprise.com` associamo l'alias `enterprise.com` e l'alias `www.enterprise.com`)

Per via delle sue funzioni, il servizio DNS risulta essere **fondamentale** per Internet.

In particolare, la **decentralizzazione** del servizio DNS risulta essere critica: se il servizio fosse centralizzato (ossia effettuato da un singolo nodo o rete) sarebbe sufficiente un singolo punto di fallimento affinché il servizio diventi inutilizzabili. Inoltre, se il servizio fosse centralizzato si avrebbe un volume di traffico troppo elevato dovuto alle miliardi di richieste effettuate giornalmente (es: il server DNS Comcast riceve 600 miliardi di richieste al giorno)

### 2.4.1 Gerarchia server DNS

Poiché il mapping DNS è **distribuito** su svariati server, dove in particolare nessuno di essi mantiene il mapping di tutti gli IP possibili (un IP corrisponde a 32 bit, dunque  $2^{32}$  IP possibili), il database tramite cui viene realizzato il servizio DNS è **gerarchico**, seguendo la struttura di un albero:

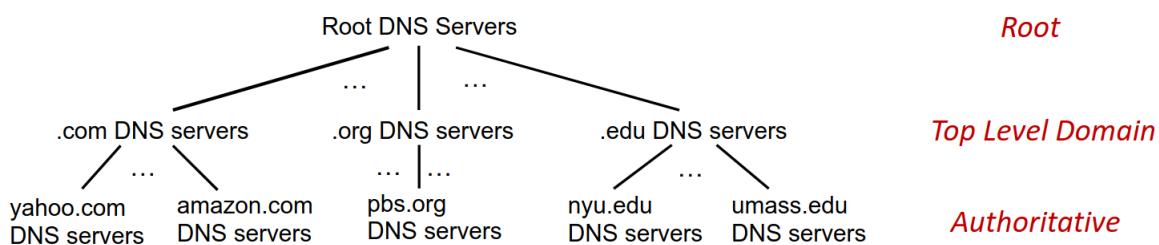
- **Root Server**:
  - Radice dell'albero
  - Viene interrogato da qualsiasi server DNS che non sia in grado di risolvere il nome di un server TLD (ossia restituire l'IP legato ad esso)
- **Server Top-Level Domain (TLD)**:
  - Viene interrogato per risolvere il nome di un server DNS autoritativo
  - Responsabili di domini come `.com`, `.org`, `.net`, ... e tutti i domini nazionali di primo livello, ossia `.it`, `.uk`, `.fr`, ...

- **Server autoritativi (o di competenza):**

- Viene interrogato per risolvere il nome di un host pubblicamente accessibile, solitamente all'interno di un'organizzazione
- Ogni organizzazione con host pubblicamente accessibili deve fornire i record DNS di pubblico dominio che mappano i nomi di tali host ai loro indirizzi IP

- **Server DNS locali (o default name server):**

- Non appartengono alla gerarchia. Ogni ISP ne è dotato
- Possiedono una cache locale delle recenti coppie di mappatura nome-indirizzo (potrebbero non essere aggiornate)
- Funge da proxy iniziale tra il client e il root server: se il nome non è nella cache del server locale, la richiesta viene inoltrata al root server



### Esempio:

1. Il client vuole ottenere l'indirizzo IP dell'host `www.amazon.com`
2. Viene contattato il server DNS locale dell'ISP di riferimento. Se il nome non viene risolto, si procede col passo successivo.
3. Viene contattato il root server per trovare l'indirizzo IP del server TLD `.com`
4. Viene contattato il server TLD `.com` per trovare l'indirizzo IP del server autoritativo `amazon.com`
5. Viene contattato il server autoritativo `amazon.com` per trovare l'indirizzo IP dell'host `www.amazon.com`

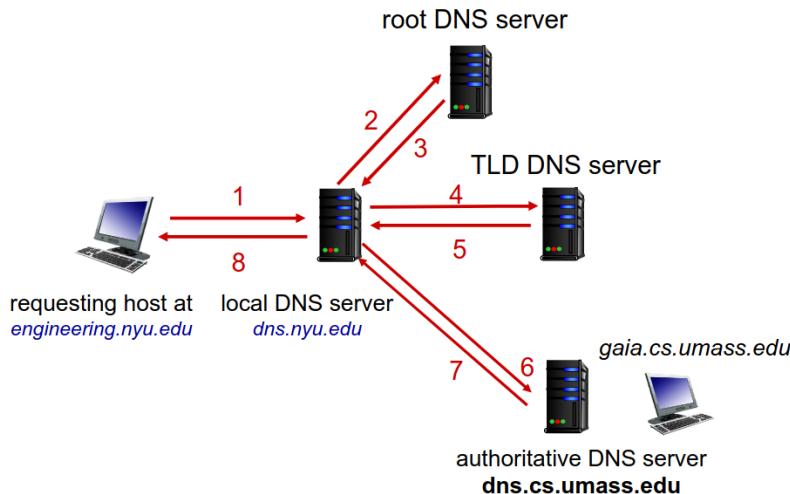
Ogni volta che un server DNS viene a conoscenza di una mappatura, essa viene **memorizzata** all'interno della cache, utilizzando tali record per rispondere a query future. I record presenti nella cache vengono cancellati allo scadere di un **TTL (Time-to-live)** o a seguito di un comando manuale.

Soltanente, all'interno della cache dei server DNS locali sono presenti i server TLD più comuni, implicando che il root server venga interrogato raramente.

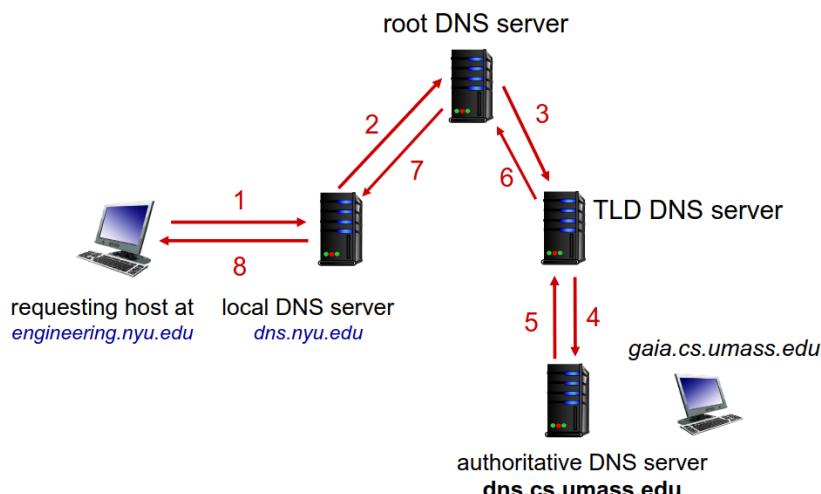
Tuttavia, è necessario notare che i record nella cache **potrebbero non essere aggiornati**: se viene cambiato l'indirizzo IP associato ad un nome presente nella cache, esso potrebbe non essere noto all'interno di Internet fino alla scadenza di tutti i TTL di tutti i server, poiché quest'ultimi risolverebbero la richiesta restituendo l'indirizzo IP precedente.

La **risoluzione dei nomi**, ossia la restituzione dell'indirizzo IP ad esso legato, può avvenire in due modalità:

- **Risoluzione a query iterativa**, dove il server contattato dal client risponde con il nome del prossimo server da contattare, il quale (probabilmente) sarà in grado di risolvere il nome



- **Risoluzione a query ricorsiva**, dove l'onere della risoluzione del nome viene affidato al server contattato, ricorsivamente



Ogni mappatura nome-indirizzo viene inserita all'interno di un **resource record (RR)**, il quale assume la struttura (name, value, type, ttl), dove a seconda del valore del campo **type** si ha che:

- **type = A**, indica che il campo **name** contiene il nome di un host interno ad un dominio (hostname) e il campo **value** contiene l'indirizzo IP di tale host  
(es: name = relay1.bar.foo.com, value = 45.37.93.126)
- **type = NS**, indica che il campo **name** contiene il nome di un dominio e il campo **value** contiene l'hostname del server autoritativo associato a tale dominio  
(es: name = foo.com, value = dns.foo.com)

- **type = CNAME**, indica che il campo **name** contiene un alias del nome canonico e il campo **value** contiene il nome canonico stesso

(es: name = www.ibm.com, value = servereast.backup2.ibm.com)

- **type = MX**, indica che il campo **name** contiene il nome di un mail server interno ad un dominio e il campo **value** contiene l'hostname di tale mail server

#### Esempio:

- Un server autoritativo per un hostname contiene un record di tipo A per l'hostname stesso, ad esempio

(corsi.di.uniroma1.it, 131.111.45.68, A)

- Un server non autoritativo per un dato hostname contiene un record di tipo NS per il dominio che include l'hostname e un record di tipo A che fornisce l'indirizzo IP del server DNS nel campo **value** del record NS.

Ad esempio, un server TLD .it che non è autoritativo per l'host corsi.di.uniroma1.it, contiene i due record

(uniroma1.it, dns.uniroma1.it, NS)

(dns.uniroma1.it, 128.119.40.111, A)

## 2.4.2 Protocollo DNS

### Definition 27. Protocollo DNS

Il **protocollo DNS** è un protocollo a livello di applicazione utilizzato per la risoluzione di hostname e nomi di dominio. La sua porta di riferimento comune all'interno dei socket è la **porta 53**.

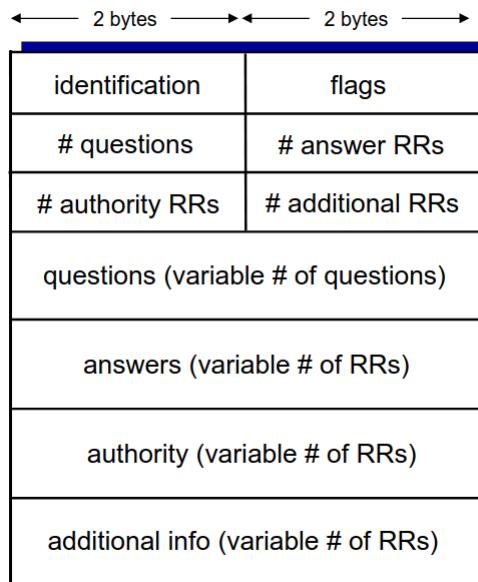
Al fine di rendere il trasferimento il più rapido possibile, il protocollo DNS utilizza il **protocollo UDP**, richiedendo l'invio di un singolo messaggio, evitando necessità della creazione del collegamento. Se un messaggio non giunge a destinazione dopo un determinato timeout, esso viene semplicemente rinvia.

Inoltre, il protocollo DNS è un protocollo **stateless**, (semplicemente poiché non è necessario salvare alcuno stato)

La **richieste** e le **risposte** DNS assumono la stessa struttura:

- Un **header** lungo 32 bit, composto da due campi da 16 bit:
  - **Identification**, contenente informazioni del richiedente
  - **Flags**, contenente flag di stato indicanti se il messaggio sia di richiesta o risposta, se la risoluzione ricorsiva sia preferita o disponibile e se la risposta sia di un server autoritativo

- Un campo **questions** di dimensione variabile contenente le informazioni per una richiesta
- Un campo **answers** di dimensione variabile contenente i RR da inviare come risposta
- Un campo **authority** di dimensione variabile contenente i RR autoritativi da inviare come risposta
- Un campo per le **informazioni aggiuntive**



## 2.5 Trasferimento di file

### 2.5.1 Protocollo FTP

#### Definition 28. Protocollo FTP

Il **protocollo FTP (File Transfer Protocol)** è un protocollo a livello di applicazione utilizzato per il trasferimento di file basato sul **paradigma client-server**.

Per gestire il trasferimento dei file, il protocollo FTP utilizza **due connessioni TCP**:

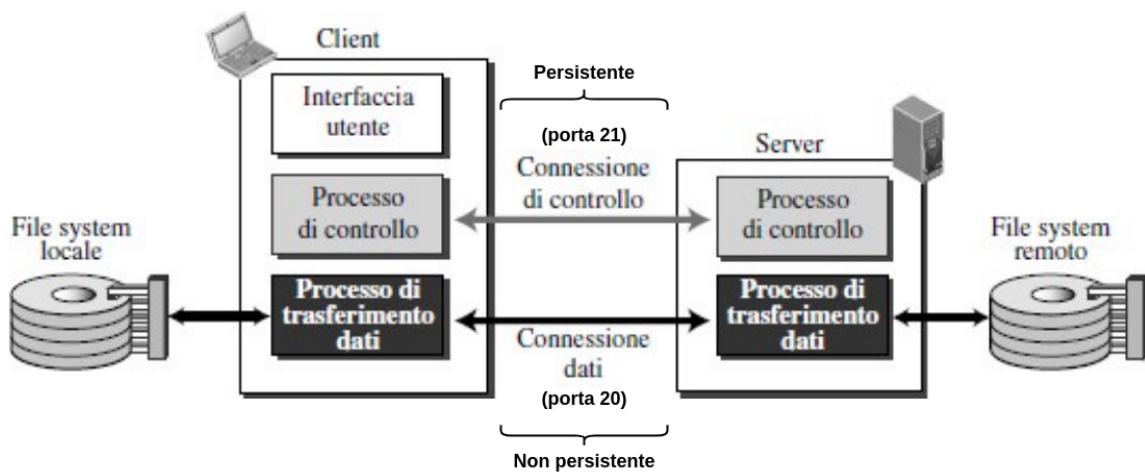
- Una **connessione di controllo** (porta 21), utilizzata per trasferire le informazioni per il controllo del trasferimento (es: nome utente, password, comandi per cambiare directory e per il trasferimento)
- Una **connessione dati** (porta 20), la quale viene aperta ogni qualvolta sia necessario trasferire un file, per poi chiuderla successivamente

Inoltre, il protocollo FTP è un protocollo **stateful**, conservando la directory corrente e l'autenticazione della sessione precedente

Nel protocollo FTP, il **client** corrisponde al dispositivo avviante il trasferimento verso un dispositivo remoto, mentre il **server** corrisponde al dispositivo remoto stesso.

Quando l'utente fornisce al proprio client il nome del server a cui connettersi tramite il comando `ftp <nome host>`, il processo client FTP stabilisce la **connessione di controllo** sulla porta 21.

Successivamente, il client trasferisce nome utente e password sulla porta 21, autenticandosi. Una volta ottenuta l'autorizzazione dal server, il client può **trasferire uno o più file** memorizzati nel file system locale verso quello remoto (o viceversa), aprendo e chiudendo la connessione dati sulla porta 20 ad ogni trasferimento.



I principali comandi del protocollo FTP sono:

Comando e Argomenti	Descrizione
ABOR	Interrompe il comando precedente
CDUP	Torna alla directory del livello precedente
CWD «nome directory»	Cambia directory corrente
DELE «nome file»	Elimina il file
LIST «nome directory»	Elenca i file nella directory
MDK «nome directory»	Crea una directory
PASS «password»	Invia la password dell'utente
PASV	Il server sceglie la porta della connessione
PORT «porta»	Il client sceglie la porta della connessione
PWD	Mostra nome directory corrente
QUIT	Termina la comunicazione
RETR «nomi dei file»	Trasferisce uno o più file dal server al client
RMD «nome directory»	Elimina la directory
RNTO «vecchio nome» «nuovo nome»	Rinomina il file specificato dal vecchio nome
STOR «nomi dei file»	Trasferisce uno o più file dal client al server
USER «nome utente»	Invia il nome dell'utente

## 2.5.2 Protocollo BitTorrent

### Definition 29. Protocollo BitTorrent

Il **protocollo BitTorrent** è un protocollo a livello di applicazione utilizzato per il trasferimento di file basato sul **paradigma peer-to-peer (P2P)**. Nonostante non abbia una porta standard, solitamente vengono utilizzate le **porte nel range 6881-6889** assieme al **protocollo TCP**.

Ogni peer entra a far parte di un **torrent**, ossia un gruppo di peer scambianti frammenti di file tra loro, registrandosi su un **tracker**, ossia un dispositivo che tiene traccia dei peer partecipanti al torrent, per poi connettersi ad un sottoinsieme di peer "vicini".

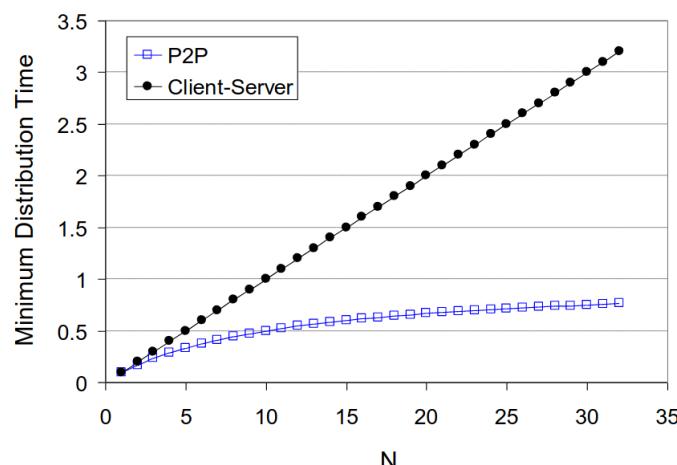
Durante il download di file, il peer svolge anche la funzione di uploader (**seeder**) di blocchi verso altri peer. Una volta ricevuto il file, il peer può scegliere se uscire dal torrent o rimanerne all'interno, continuando a svolgere la funzione di seeder.

In un dato momento, peer diversi possiedono diversi sottoinsiemi di blocchi componenti un file. Per richiedere tali blocchi, un peer chiede periodicamente agli altri l'**elenco dei blocchi** attualmente posseduti. Successivamente, il peer richiede i blocchi mancanti, dando precedenza ai più rari.

Per favorire l'altruismo tra i peer e sfavorire la presenza di **leecher**, ossia dispositivi che egoisticamente escono dal torrent una volta scaricato un file, il protocollo BitTorrent usa un approccio **tit-for-tat** (traduzione più vicina: *do ut des*, "io ti do e tu mi dai" ):

- Ogni peer seeder invia blocchi agli ulteriori quattro peer seeder che attualmente stanno uploadando i blocchi richiesti alla velocità maggiore
- Gli altri peer non appartenenti alla top 4 vengono "strozzati" (**choked**) dal peer seeder, bloccando l'invio dei blocchi ad essi.
- Ogni 10 secondi, tale top 4 viene rivalutata. Inoltre, ogni 30 secondi viene sbloccato casualmente un peer strozzato (**optimistic un-choking**), il quale può entrare o meno a far parte della top 4

Per via di tale approccio, il trasferimento di un file ad  $N$  dispositivi risulta più ottimale nel caso dell'applicazione del paradigma P2P



# Capitolo 3

## Livello di Trasporto

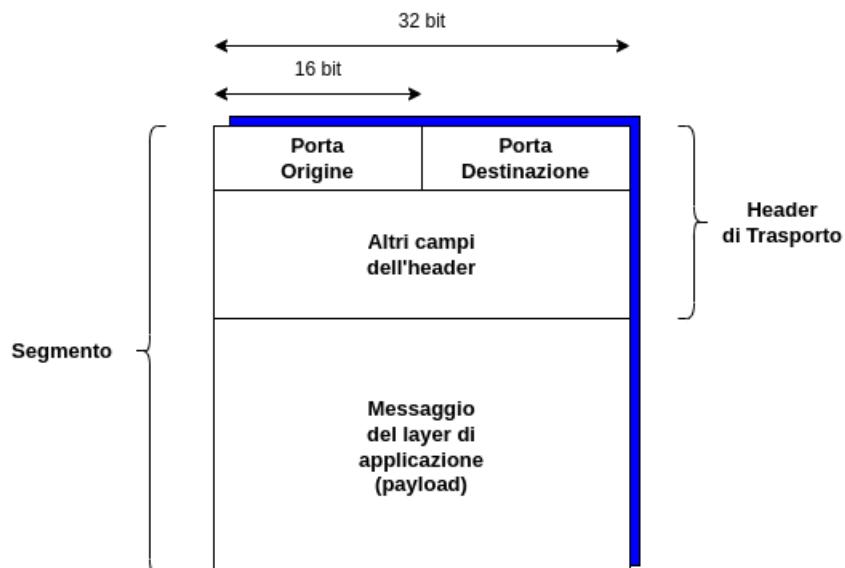
I servizi e protocolli situati nel **livello di trasporto** forniscono **comunicazione logica** tra processi applicativi in esecuzione su dispositivi diversi, a differenza del **livello di rete**, il quale si occupa della comunicazione logica direttamente tra i dispositivi stessi.

In particolare, il dispositivo mittente suddivide i messaggi dell'applicazione in segmenti, passandoli al livello di rete, mentre il dispositivo destinatario riassembra i segmenti in messaggi, passandoli al livello di applicazione.

### 3.1 Multiplexing e Demultiplexing

Per implementare le funzionalità di **multiplexing** e **demultiplexing** al livello di trasporto, ogni host utilizza **indirizzi IP** e **numeri di porta** per indirizzare correttamente un segmento al socket appropriato del destinatario:

- L'header di ogni segmento possiede un numero di porta per l'origine e la destinazione
- Ogni datagramma del livello di rete trasporta un segmento del livello di trasporto
- L'header di ogni datagramma possiede l'indirizzo IP dell'origine e l'indirizzo IP della destinazione

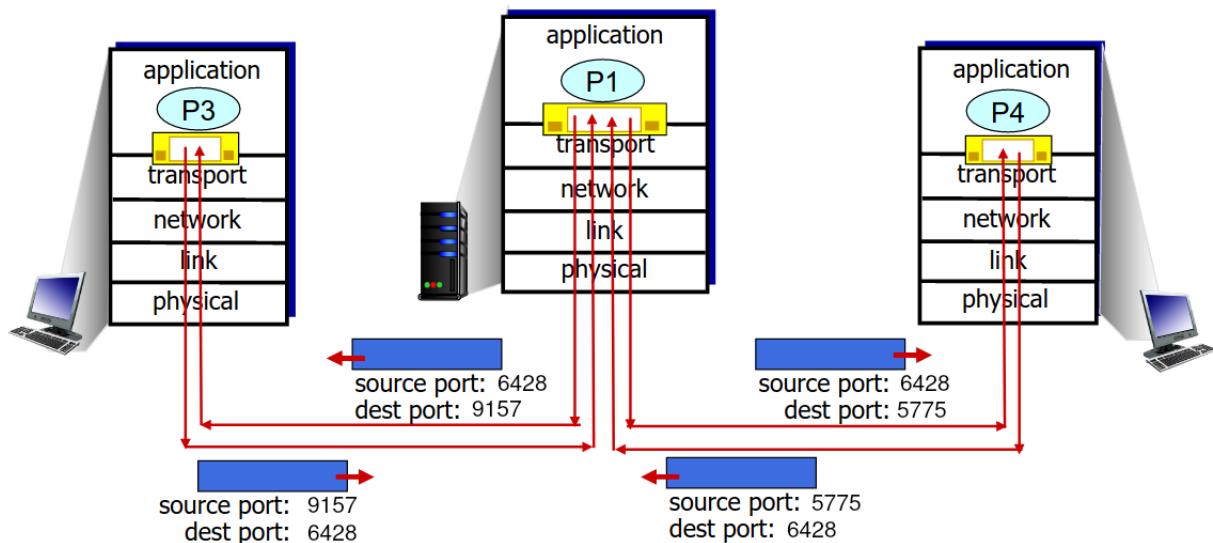


Nel caso di un **demultiplexing senza connessione** (es: protocollo UDP), durante la creazione di un socket all'interno di un processo è necessario specificare la porta locale dell'**host** con cui identificare tale socket.

(es: DatagramSocket d\_soc = new DatagramSocket(12534);)

Successivamente, qualsiasi dispositivo che voglia comunicare con tale host invierà un datagramma al cui interno sia specificata la coppia **indirizzo IP di destinazione** e la **porta di destinazione**. Una volta giunto alla destinazione, verrà letto il numero di porta di destinazione presente nell'header del segmento contenuto all'interno del datagramma ricevuto, indirizzando il segmento al **socket con tale numero di porta**.

In particolare, è necessario sottolineare che, in tal modo, il socket su tale host per la comunicazione con più mittenti sia **unico**. Di conseguenza, qualsiasi datagramma inviato su tale porta apparterrà allo **stesso stream dati**. Tuttavia, essi saranno comunque distinti univocamente dal numero di porta e l'indirizzo IP del mittente presenti nel datagramma.

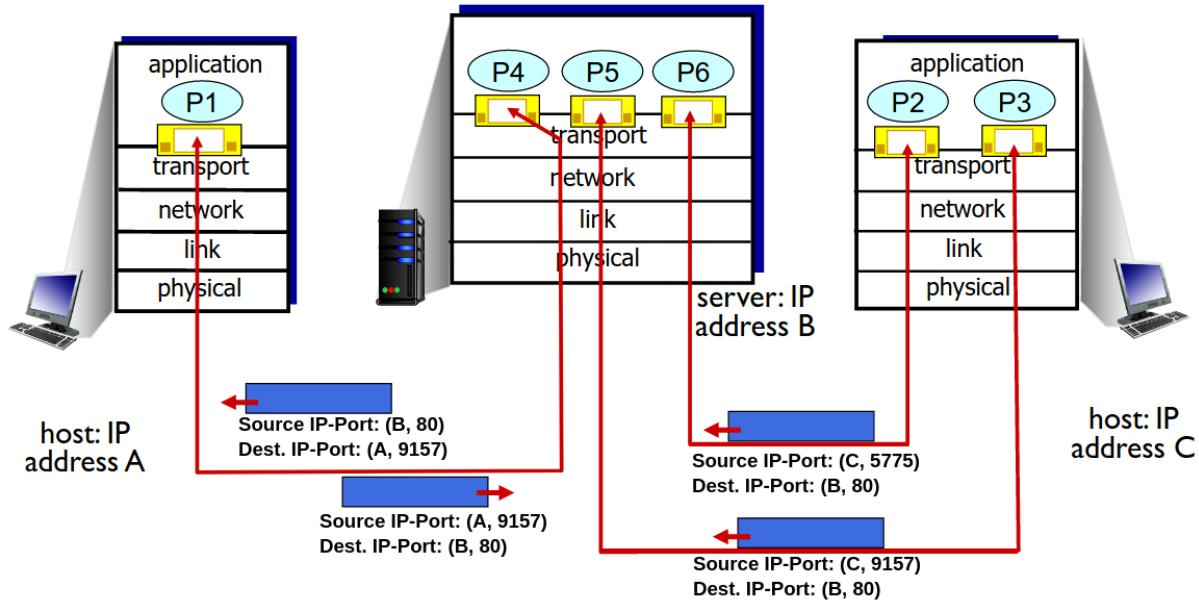


Nel caso del **demultiplexing con connessione** (es: protocollo TCP), invece, **ogni socket viene identificato univocamente come una quadrupla**

(IP\_Orig., Porta\_Orig., IP\_Dest., Porta\_Dest.)

Una volta ricevuto il datagramma, vengono utilizzati tutti e quattro i valori per indirizzare il segmento al socket appropriato. In tal modo, ogni **connessione** è identificata in modo univoco da una **coppia di socket** (una sul primo host ed una sul secondo host), permettendo di implementare le garanzie previste dai protocolli.

Inoltre, per via dell'identificazione univoca dei socket, un host può avere **più socket legati alla stessa porta** con una comunicazione diversa: se due host A e C avviano una connessione avente come destinazione la porta 80 dell'host B, su quest'ultimo verranno creati **due socket diversi**.



## 3.2 Trasporto senza connessione (protocollo UDP)

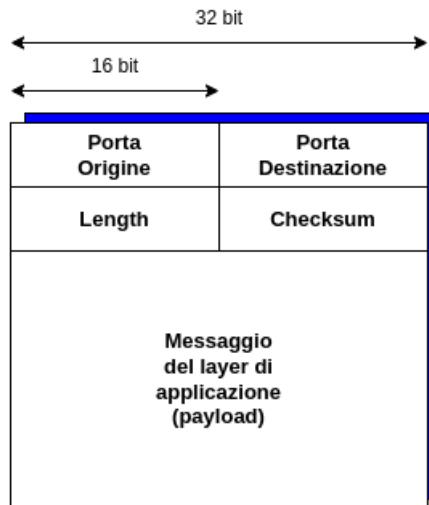
Come già accennato, il **protocollo UDP** è un protocollo di trasporto "senza fronzoli" (**bare bone**) e **senza connessione**. Per tanto, non avviene alcun handshake tra mittente e destinatario, implicando che ogni segmento UDP venga gestito indipendentemente dagli altri. Inoltre, il protocollo UDP svolge un servizio **best-effort**, dunque i segmenti UDP possono essere persi o consegnati in modo non ordinato.

Tuttavia, tali caratteristiche rendono UDP **vantaggioso** in alcune casistiche:

- Poiché non vi è alcuna connessione, il protocollo risulta semplice, oltre all'**assenza del ritardo RTT** necessario per l'handshake richiesto
- La dimensione dell'header è minima, rendendo il **pacchetto più leggero**
- L'**assenza di controllo della congestione** permette al protocollo UDP di tentare la trasmissione senza alcun limite di velocità e il funzionamento anche in casi di congestione dovuti ad un carico elevato sui nodi della rete
- Se si vuole rendere il trasferimento affidabile anche utilizzando UDP, basta implementare l'affidabilità necessaria al livello di applicazione (es: HTTP/3 tramite il protocollo QUIC), piuttosto che al livello di trasporto

In particolare, l'header utilizzato dal protocollo UDP, oltre a contenere le porte di origine e di destinazione, contiene solamente due campi aggiuntivi:

- Un campo **length** (16 bit), indicante la lunghezza del contenuto
- Un campo **checksum** (16 bit), utilizzato per rilevare errori nel segmento trasmesso



Il valore di **checksum** viene calcolato tramite una **somma in complemento ad uno con wrap-around**:

1. Il mittente considera il contenuto del segmento (compresi gli altri campi dell'header e gli indirizzi IP) come una sequenza di numeri interi a 16 bit
2. Il mittente calcola il checksum sommando in **complemento ad 1** (ossia sommando e poi invertendo tutti i bit) i numeri interi della sequenza.  
In particolare, se è presente un riporto finale generato dalla somma del bit più significativo, viene sommato anche tale riporto (**wrap-around**)
3. Il valore del checksum viene inserito nel campo dell'header e il pacchetto continua il processo di trasmissione
4. Una volta giunto a destinazione, l'host ricevente **calcola nuovamente il checksum**, verificando che sia uguale a quello inserito nell'header del segmento.

Se il checksum è differente, viene rilevato un **errore** nella trasmissione

Tuttavia, è necessario notare che se il **checksum è uguale non è detto** che la trasmissione non abbia generato alcun errore:

- Consideriamo il calcolo del checksum tra i seguenti numeri interi a 16 bit:

$$\begin{array}{r}
 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ + \\
 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 = \\
 \hline
 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1
 \end{array}$$

- Poiché è presente un riporto, viene effettuato il **wrap-around** sommando tale riporto ai restanti 16 bit:

$$\begin{array}{r}
 1 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \ 1 \ 0 \ + \\
 1 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \ 0 \ 1 = \\
 \hline
 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \\
 \quad \downarrow \quad \downarrow \quad \downarrow \\
 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0
 \end{array}$$

- Infine, vengono **invertiti i bit** del risultato per ottenere la somma in complemento ad 1

$$\begin{array}{cccccccccccccccccc}
 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 & & & & & & \downarrow & \downarrow & \downarrow & & & & & & & & \\
 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
 \end{array}$$

- Tuttavia, nel caso in cui i due bit meno significativi di entrambi i numeri fossero stati invertiti (per qualche motivo sconosciuto) durante la trasmissione, il **checksum calcolato sarebbe identico**

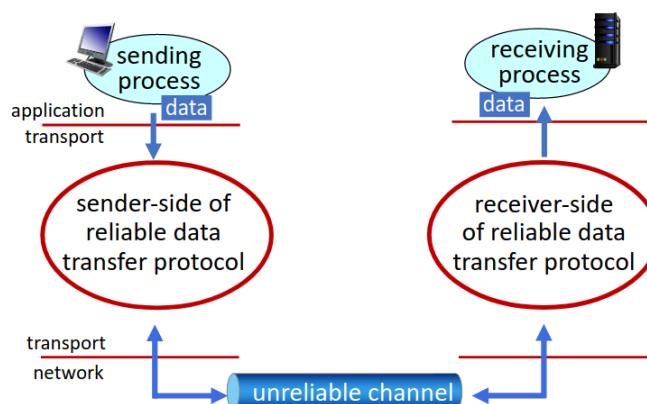
$$\begin{array}{cccccccccccccccccc}
 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & + \\
 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 = \\
 \hline
 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
 & & & & & & \downarrow & \downarrow & \downarrow & & & & & & & \\
 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\
 & & & & & & \downarrow & \downarrow & \downarrow & & & & & & & \\
 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1
 \end{array}$$

### 3.3 Trasferimento affidabile dei dati

Per realizzare un trasferimento affidabile dei dati, è necessario implementare un **canale sicuro** al cui interno non vengano perse o corrotte informazioni.



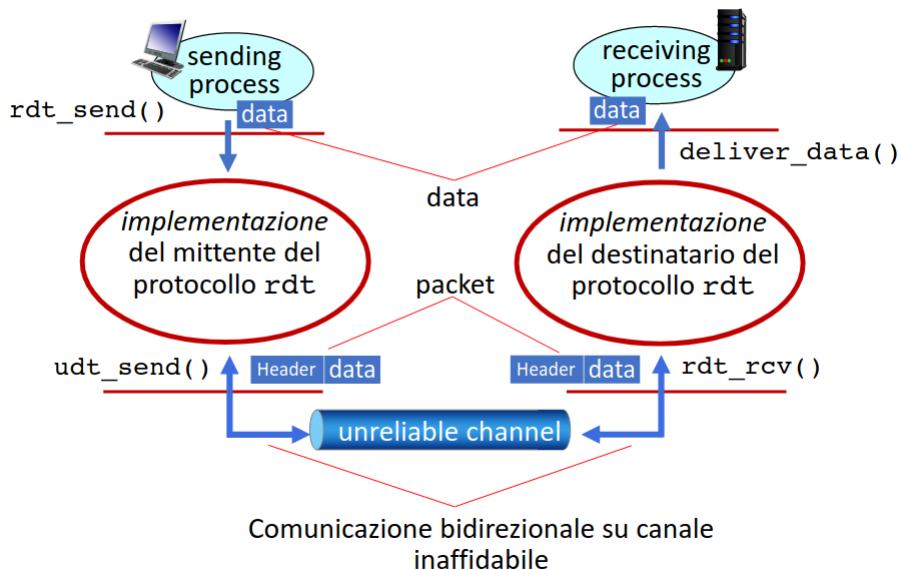
Tuttavia, un canale fisico che possa svolgere tale funzione risulta essere irrealizzabile. Per tale motivo, la complessità del **protocollo di trasferimento dati affidabile (RDT - Reliable Data Transfer)** dipende fortemente dalle caratteristiche del canale inaffidabile utilizzato.



Inoltre, è necessario puntualizzare che il mittente e il destinatario **non conoscono lo stato l'uno dell'altro** (es: se la ricezione sia andata a buon fine), a meno che non gli venga comunicato tramite un ulteriore messaggio.

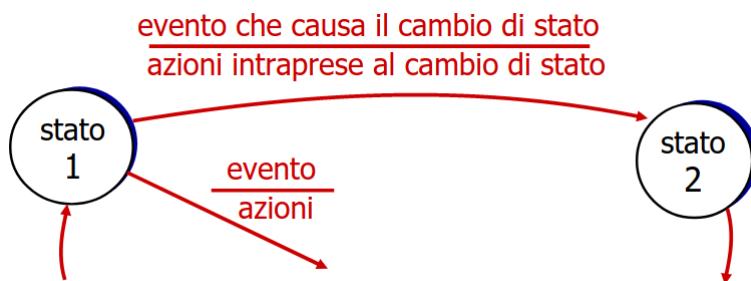
Il protocollo RDT presenta delle **interfacce** per il suo utilizzo:

- **rdt\_sent(data)**, il quale viene chiamato dal livello di applicazione e il cui argomento corrisponde ai dati da inoltrare al destinatario
- **udt\_send()**, dove UDT è acronimo di Unreliable Data Transfer, il quale viene chiamato dal protocollo RDT sul mittente per trasferire il pacchetto sul canale inaffidabile
- **rdt\_rcv()**, il quale viene chiamato alla ricezione del pacchetto dal destinatario
- **deliver\_data()**, il quale viene chiamato dal protocollo RDT sul destinatario per inoltrare al livello di applicazione i dati ricevuti



Poiché i dispositivi comunicanti non sono a conoscenza dello stato altrui, il protocollo RDT si basa su un **trasferimento dei dati unidirezionale**, dunque come se uno solo dei due sia il mittente ed uno solo sia il destinatario (sebbene in realtà sia bidirezionale).

Per rappresentare le operazioni e le decisioni effettuate dal protocollo, utilizzeremo le **macchine a stati finiti** (FSM - Finite State Machine) e in particolare la seguente notazione:



### 3.3.1 Protocollo RDT 1.0 e 2.0

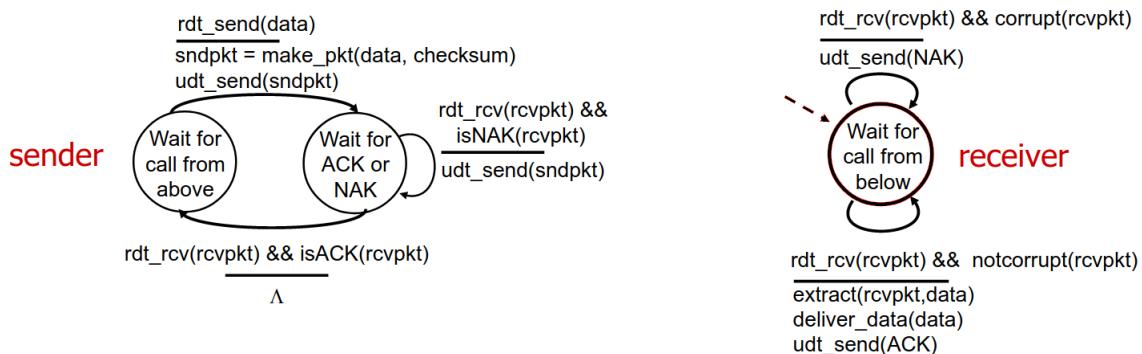
All'interno del **protocollo RDT 1.0**, viene assunto che il canale sottostante utilizzato per il trasferimento sia **perfettamente affidabile**, implicando che il mittente invii i dati nel canale e il ricevitore li legga direttamente, senza alcuna operazione aggiuntiva



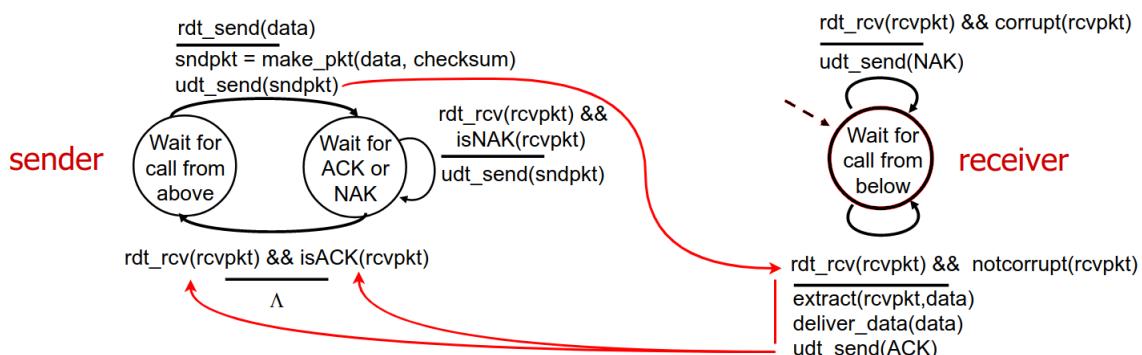
Nel **protocollo RDT 2.0**, invece, viene assunto che il canale sottostante possa invertire alcuni bit nel pacchetto inviato. Analogamente al protocollo UDP, viene utilizzato un **checksum** per rilevare la presenza di errori. Nel caso in cui venga rilevato uno di quest'ultimi, il destinatario comunicherà al mittente l'esito dell'operazione:

- **Acknowledgements (ACK)**, dove il destinatario dice esplicitamente al mittente che il pacchetto è stato ricevuto senza problemi
- **Negative acknowledgements (NAK)**, dove il destinatario dice esplicitamente al mittente che il pacchetto ricevuto presenta degli errori

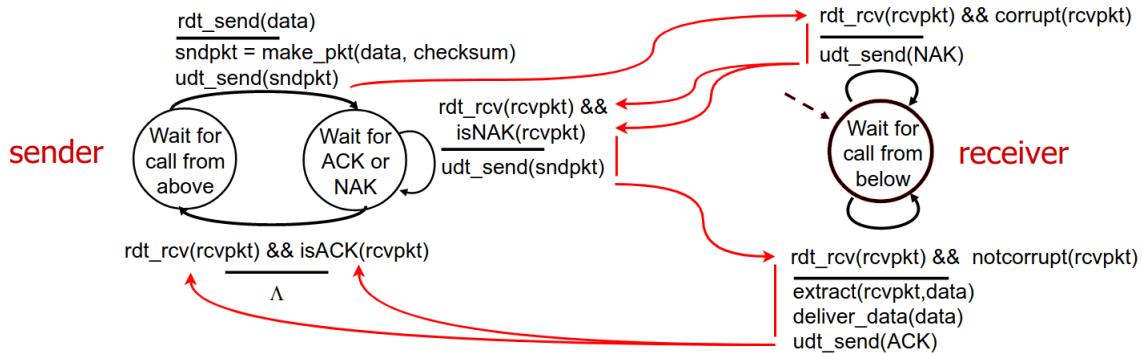
Successivamente all'invio di un pacchetto, il mittente rimane in attesa della risposta del destinatario (meccanismo **stop and wait**)



Se la risposta ricevuta è un **ACK**, il mittente torna il stato di attesa del prossimo pacchetto da parte del livello applicativo.



Se invece la risposta è un **NAK**, il mittente rinvia il pacchetto generante l'errore e rimane in attesa della risposta del destinatario, ripetendo nuovamente tale processo nel caso in cui si riceva nuovamente un NAK.

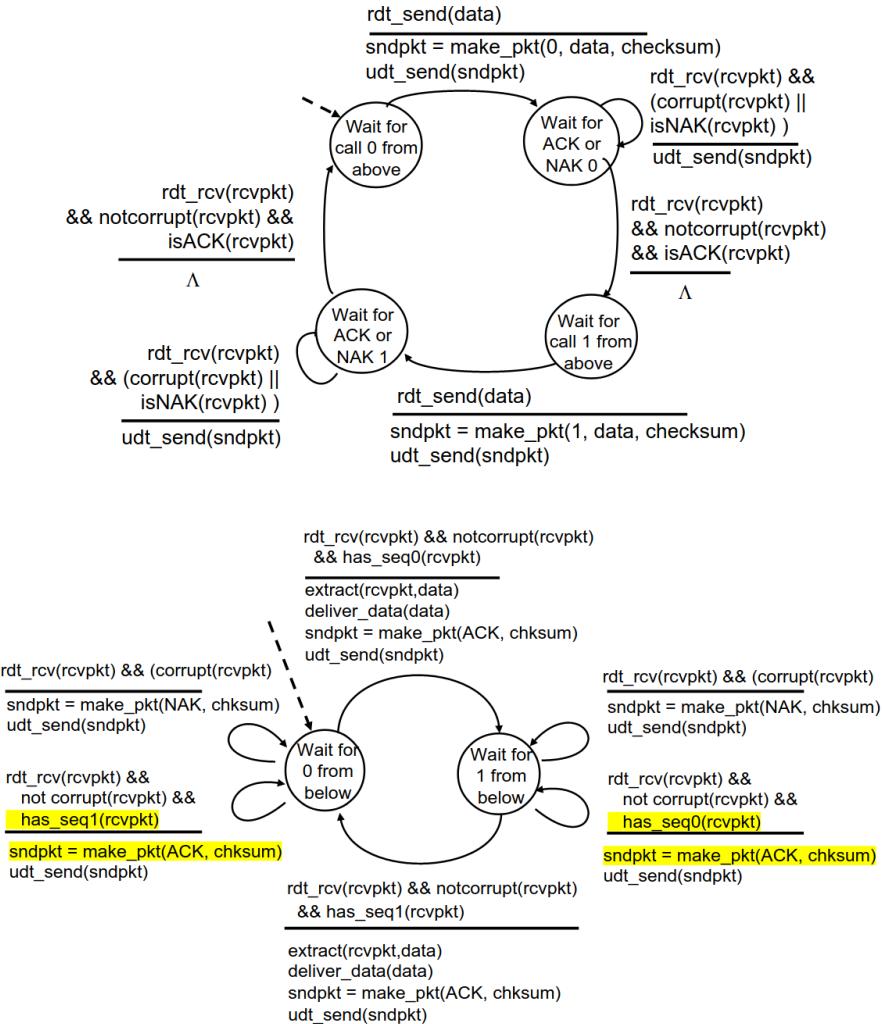


Tuttavia, la versione 2.0 del protocollo RDT presenta un **difetto fatale**: se la risposta ACK/NAK è corrotta, il mittente non è più a conoscenza di cosa sia accaduto al destinatario. Inoltre, non è sufficiente ritrasmettere il pacchetto per risolvere tale difetto, poiché il destinatario potrebbe ricevere due pacchetti duplicati ed inoltrarli al livello di applicazione.

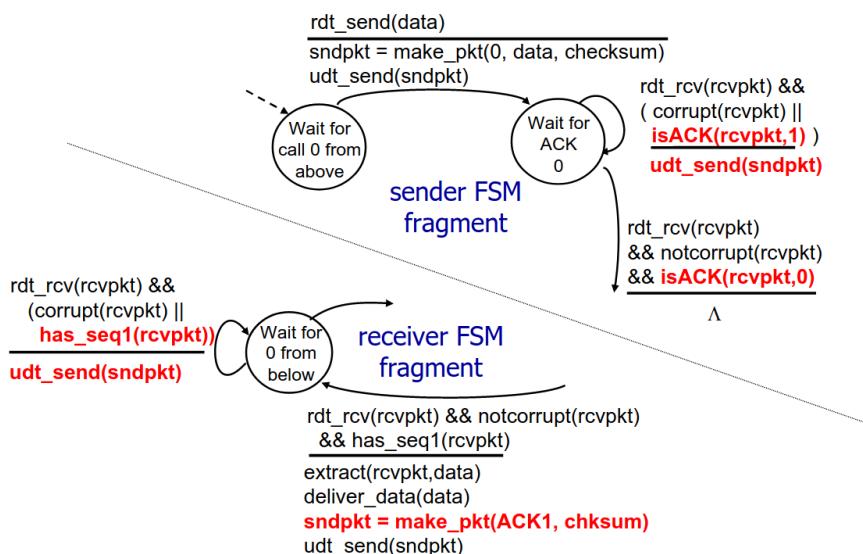
### 3.3.2 Protocollo RDT 2.1 e 2.2

Per risolvere il difetto fatale della versione 2.0, il **protocollo RDT 2.1**:

- Viene controllato se la risposta ACK/NAK sia **corrotta**. Nel caso in cui lo sia, il pacchetto viene rinvia.
- Il destinatario non è a conoscenza della possibile corruzione del pacchetto ACK/-NAK
- Viene aggiunto un **numero di sequenza** al pacchetto inviato. In particolare, sono necessari i numeri di sequenza 0 ed 1 affinché il protocollo stop and wait possa funzionare correttamente:
  - Assieme alla risposta di ACK, il destinatario invia un **numero di riscontro**, il quale, per convenzione, indica sempre il numero di sequenza del prossimo pacchetto atteso dal destinatario
  - Se il destinatario ha ricevuto correttamente il pacchetto 0, invia un riscontro con valore 1 (dunque il prossimo pacchetto atteso è il pacchetto 1)
  - Analogamente, se il destinatario ha ricevuto correttamente il pacchetto 1, invia un riscontro con valore 0 (dunque il prossimo pacchetto atteso è il pacchetto 0)
- Se il pacchetto ricevuto dal destinatario è un duplicato, esso viene automaticamente scartato senza essere inviato al livello di applicazione



In aggiunta alle modifiche della versione 2.1, il protocollo RDT 2.2 **elimina** la necessità di una risposta **NAK**: il ricevitore invia come numero di riscontro il numero di sequenza dell'**ultimo pacchetto correttamente**. In tal modo, un ACK duplicato al mittente comporta la stessa azione di un NAK, ossia la ritrasmissione del pacchetto corrente.

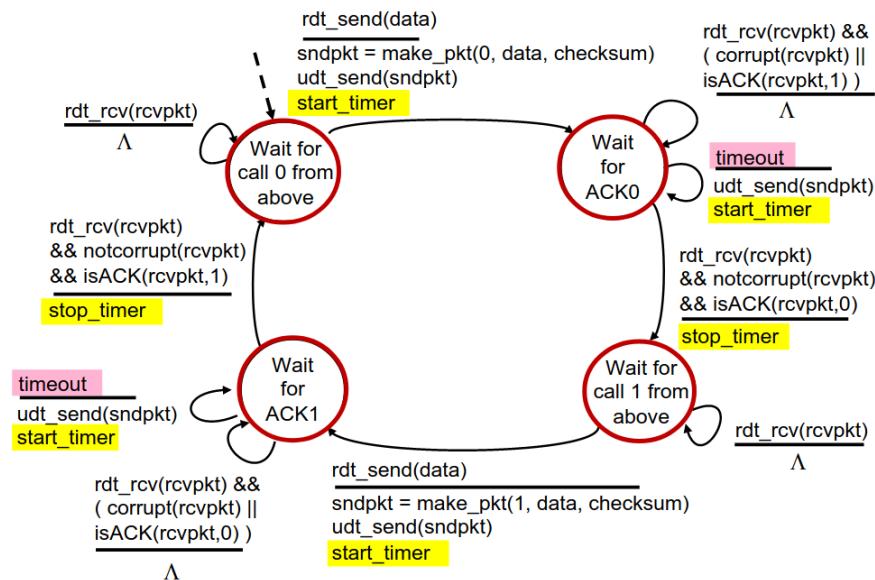


### 3.3.3 Protocollo RDT 3.0

Oltre all'assunzione di possibili bit invertiti, il **protocollo RDT 3.0** assume la possibilità di una **perdita di pacchetti**, sia dati che ACK. Per risolvere tale problematica, il mittente **attende un lasso di tempo**:

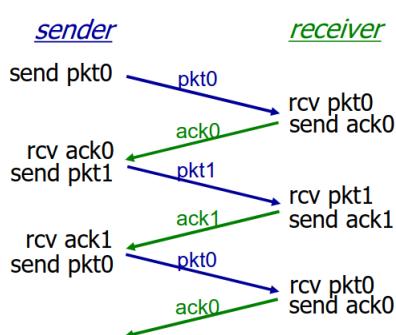
- Il destinatario deve specificare il **numero di sequenza del pacchetto** per il quale sta inviando un ACK
- Se non viene ricevuto alcun ACK allo scadere del lasso di tempo, il pacchetto dati (che indicheremo con pkt) viene ritrasmesso
- Se pkt o ACK arrivano successivamente allo scadere del tempo, il pacchetto verrà ritrasmesso, implicando che la trasmissione verrà duplicata (problema già gestito dai numeri di sequenza)

La FSM associata al mittente corrisponde a:

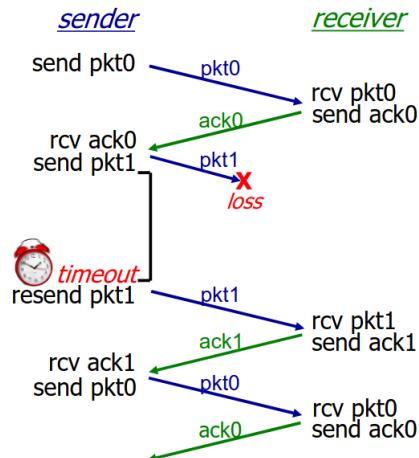


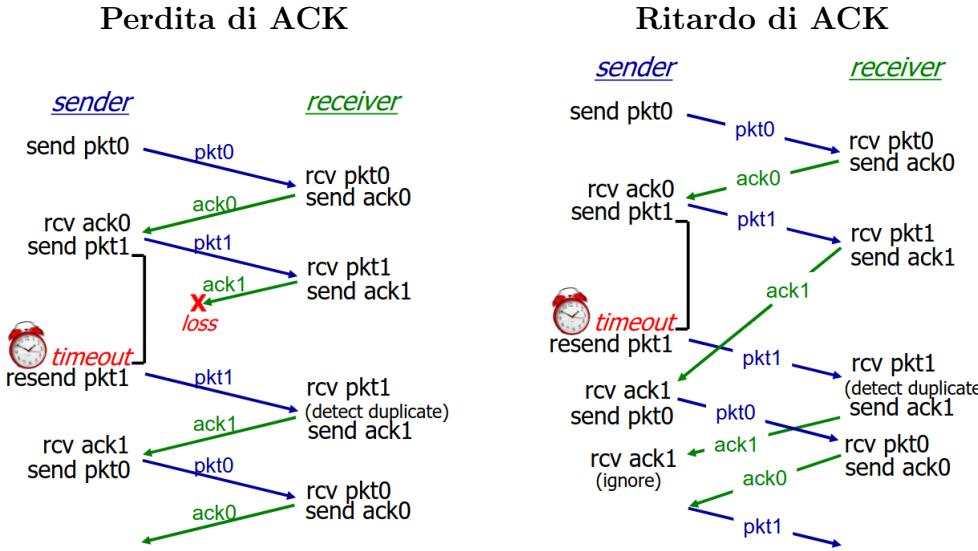
Di seguito, vengono mostrati alcuni esempi di gestione tramite protocollo RDT 3.0:

Nessuna perdita



Perdita di pkt



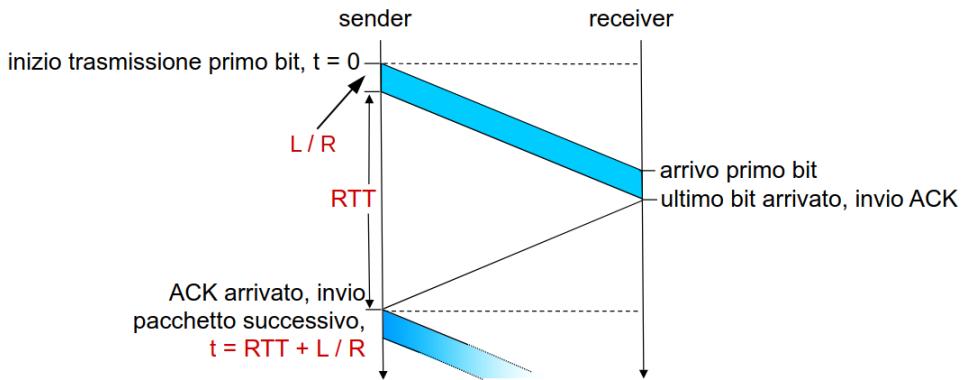


Tuttavia, in cambio dei notevoli benefici del meccanismo stop and wait, le **prestazioni** del protocollo RDT 3.0 risultano essere **infime**, limitando le prestazioni dell'infrastruttura sottostante, ossia il canale.

In particolare, la **percentuale di utilizzo**  $U_{mit}$  della comunicazione da parte del mittente, ossia la frazione di tempo in cui il mittente è impegnato nell'invio corrisponde a:

$$U_{mit} = \frac{D_t}{RTT + D_t}$$

dove  $D_t$  è il delay di trasmissione (dunque  $D_t = \frac{L}{R}$  con  $L$  la lunghezza del pacchetto e  $R$  il transmission rate del link)



### Esempio:

- Considerando un link avente un rate pari a  $R = 1 \text{ Gb/s}$ , una lunghezza di pacchetto pari a  $L = 8000 \text{ b}$  e un ritardo di propagazione sia pari a 15 ms, la percentuale di utilizzo del mittente corrisponde a:

$$D_t = \frac{8 \cdot 10^3 \text{ b}}{10^9 \text{ b/s}} = 8 \mu\text{s}$$

$$U_{mit} = \frac{8 \mu\text{s}}{30 \text{ ms} + 8 \mu\text{s}} = 27 \cdot 10^{-5} = 0.027\%$$

### 3.3.4 Go-back-N e Selective repeat

Per migliorare le prestazioni del protocollo RDT 3.0, viene utilizzato il **pipelining**, dove il mittente consente la presenza di molteplici trasferiti senza aver ricevuto un ACK precedente.

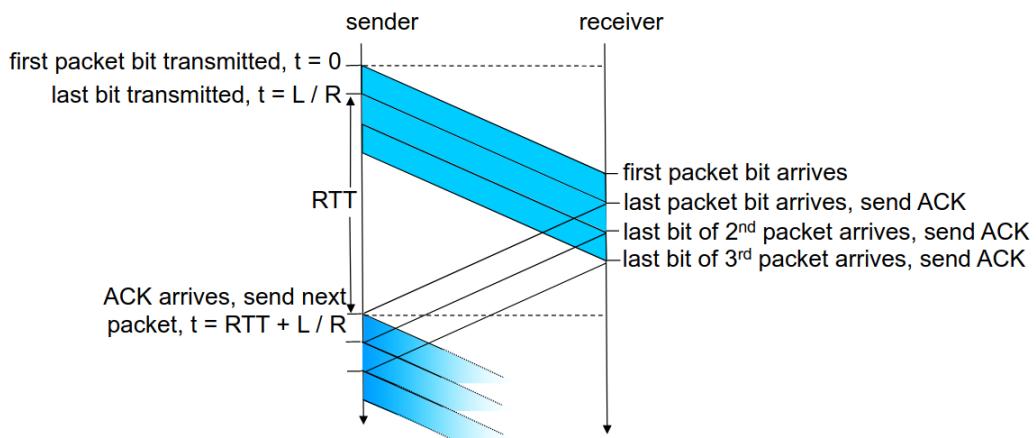
Per realizzare il pipelining, l'**intervallo di numeri di sequenza** deve essere **aumentato**, poiché è necessario tener traccia di più pacchetti simultaneamente, richiedendo inoltre la presenza di un **buffer** interno al mittente e al destinatario.

Poiche i pacchetti successivi al primo vengono inviati durante contemporaneamente al RTT del primo pacchetto, è sufficiente considerare un solo RTT, incrementando notevolmente la percentuale di utilizzo del mittente:

**Esempio:**

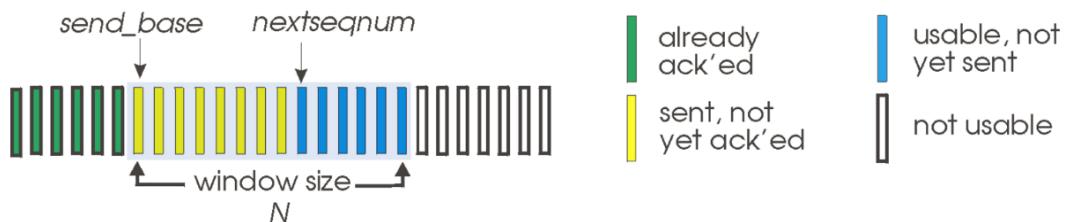
- Riprendendo i dati dell'esempio precedente, effettuando il pipelining con 3 pacchetti si ha che

$$U_{mit} = \frac{3 \cdot D_t}{RTT + D_t} = 3 \cdot \frac{8 \mu s}{30 ms + 8 \mu s} = 81 \cdot 10^{-5} = 0.081\%$$



Una delle metodologie con cui viene implementato il pipelining è il **Go-back-N**:

- Il mittente ha una "finestra" di  $N$  pacchetti consecutivi trasmessi senza ACK (**ACK cumulativo**). La ricezione del pacchetto **ACK(n)** viene interpretato dal mittente come un ACK per ognuno dei singoli  $N$  pacchetti, implicando che alla sua ricezione la finestra venga spostata in avanti in modo che essa abbia il pacchetto  $N + 1$  come primo pacchetto



- Viene mantenuto attivo un **timer** per il pacchetto della finestra inviato e senza ACK **più vecchio**. Una volta scaduto tale timeout, viene ritrasmesso il pacchetto e tutti i pacchetti con numero di sequenza maggiore presenti all'interno della finestra

- Il destinatario invia sempre l'**ACK con numero di sequenza maggiore** (in ordine) per i pacchetti attualmente ricevuti **correttamente**, implicando che non vi siano pacchetti con numero di sequenza minore mancanti.

Tale procedura potrebbe generare ACK duplicati e richiede di ricordare solamente un **valore `rcv_base`** (a differenza della finestra del mittente), corrispondente al numero di sequenza del pacchetto di cui si è in attesa

- Se il destinatario riceve un pacchetto fuori ordine, può, a seconda dell'implementazione, scartare tale pacchetto (**politica don't buffer**) o conservarlo (**politica buffer**), inviando in entrambi i casi un ACK con il più alto numero di sequenza che si trovi nell'ordine corretto, richiedendo quindi la trasmissione di tutti i pacchetti con numero di sequenza maggiore.

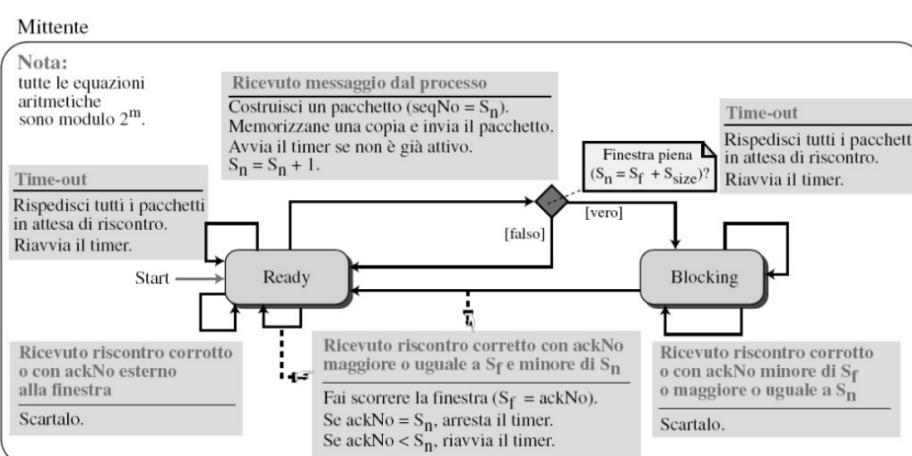
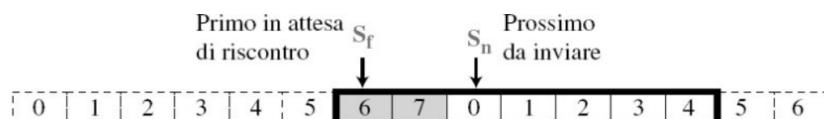


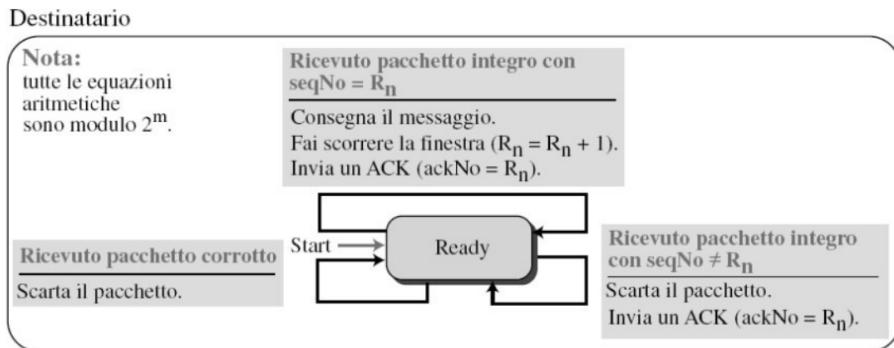
### Esempio:

- Consideriamo la seguente finestra di 7 pacchetti

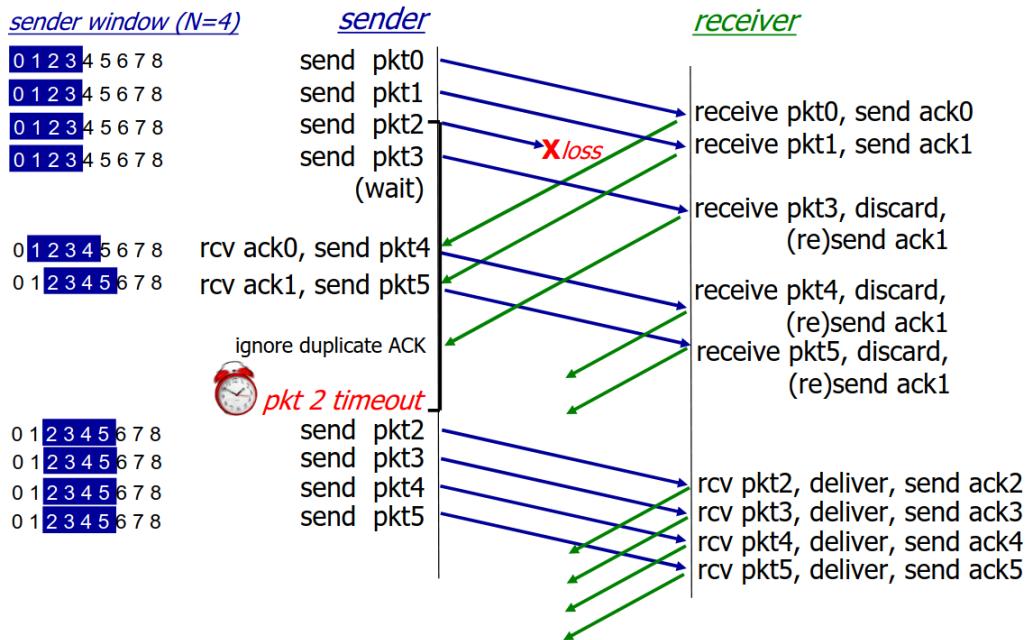


- Una volta ricevuto ACK(5), i pacchetti 4 e 5 vengono considerati come arrivati a destinazione, scorrendo la finestra in avanti





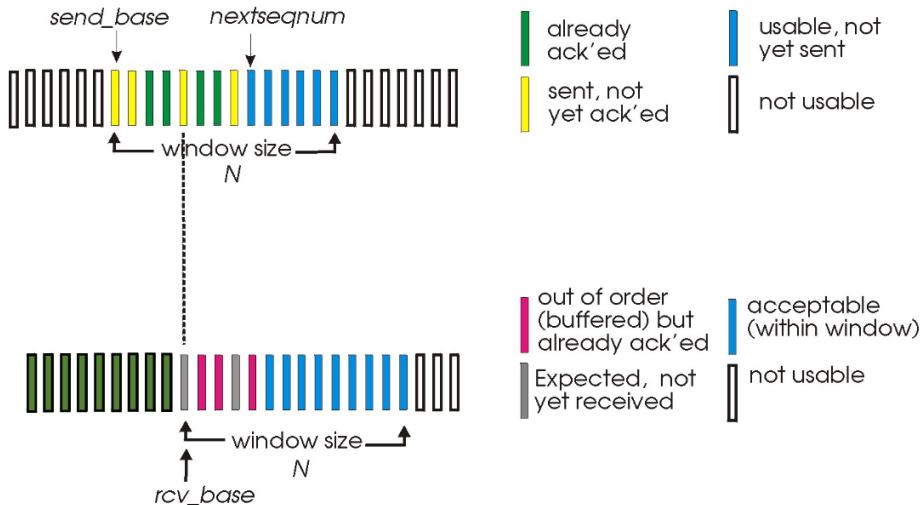
Esempio (protocollo Go-back-N con politica don't buffer):



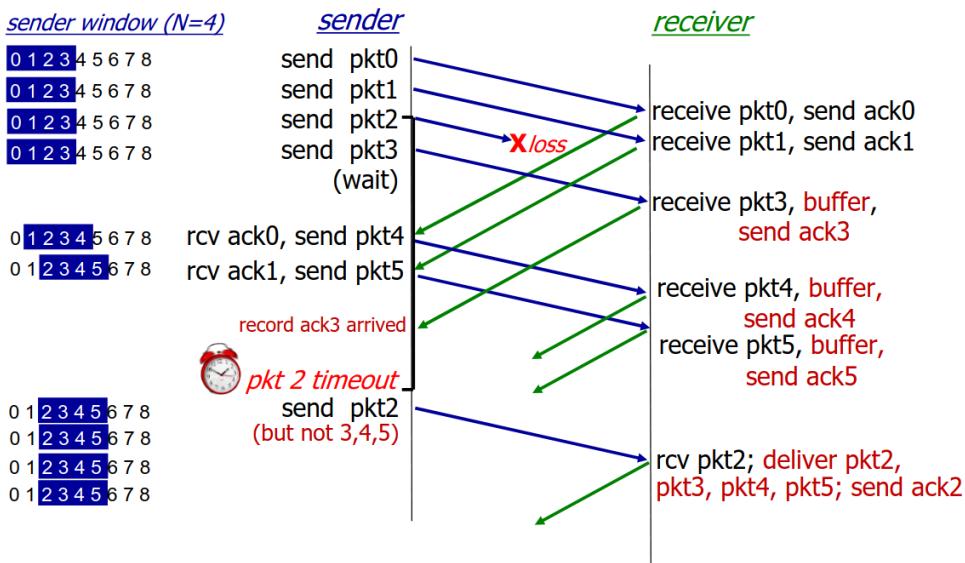
Poiché nel caso in cui un singolo pacchetto venga perso o corrotto è necessario rinviare tutti i pacchetti successivi già inviati nella pipeline, il protocollo Go-back-N può **peggiорare la congestione della rete**.

Contrariamente, il **protocollo Selective Repeat** è in grado di gestire tale problematica:

- Oltre al mittente, anche il destinatario è dotato di una **finestra di  $N$  pacchetti**
- Il destinatario conferma **individualmente** tutti i pacchetti ricevuti correttamente, anche nel caso in cui essi siano fuori sequenza, **bufferizzandoli** per l'eventuale consegna in ordine al livello superiore
- Il mittente mantiene un **timer per ogni pacchetto** inviato senza ACK, rinvia individualmente alla scadenza del suo timeout
- La finestra del mittente scorre a partire dal **pacchetto più alto confermato in ordine** (senza pacchetti non confermati prima di esso). Alla ricezione dell'ACK di un pacchetto, dunque, se tale pacchetto era il più piccolo pacchetto non ancora confermato, la finestra avanza fino al prossimo pacchetto non confermato

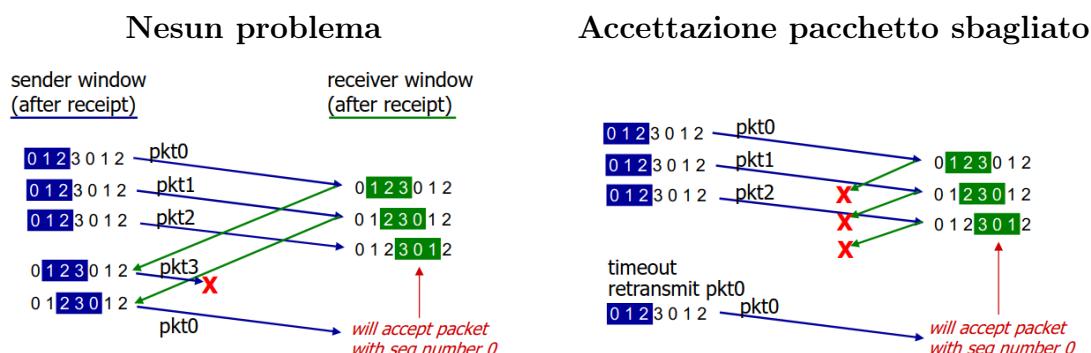


Esempio:



Tuttavia, anche il protocollo Selective Repeat non è privo di problematiche. In particolare, se la dimensione della finestra è troppo piccola, si può andare in contro a casi sfavorevoli (**dilemma della finestra**).

Ad esempio, con un range di numeri di sequenza pari a 0, 1, 2, 3 e una finestra di dimensione 3, si ha che:



**Esempio:**

- In una rete con un valore fisso  $m > 1$  (numero di bit della sequenza), è possibile utilizzare entrambi i meccanismi Go-Back-N e Selective Repeat, si indichino i vantaggi e gli svantaggi dell'impiego di ciascuno di essi. Quali altre considerazioni si devono fare per decidere quale meccanismo utilizzare?

- **Go-back-N**

- Ritrasmette tutti i frame inviati dopo il frame che si sospetta essere danneggiato o perso
- Se il tasso di errore è alto, spreca molta larghezza di banda
- Meno complicato
- Window size  $N - 1 = 2^m - 1$
- Riordinamento non è richiesto né lato mittente né lato destinatario
- Il destinatario non memorizza i frame ricevuti dopo il frame corrotto finché esso non viene ritrasmesso (dipende dall'implementazione)
- Non è richiesta alcuna ricerca di frame né lato mittente né destinatario

- **Selective Repeat**

- Ritrasmette solo i frame sospettati di essere persi o danneggiati
- Comparativamente meno larghezza di banda viene sprecata nella ritrasmissione
- Più complesso in quanto richiede l'applicazione di logica aggiuntiva, ordinamento e archiviazione, lato mittente e destinatario
- Window size  $\frac{N+1}{2} = 2^{m-1}$
- Il destinatario deve essere in grado di ordinare in quanto deve mantenere la sequenza dei frame
- Il destinatario memorizza i frame ricevuti dopo il frame danneggiato nel buffer finché il frame danneggiato non viene sostituito
- Il mittente deve essere in grado di cercare e selezionare il frame richiesto

Un'idea aggiuntiva implementata nei **trasferimenti bidirezionali** (dunque dove entrambi i dispositivi sono sia mittente sia destinatario, coincidenti con la vita reale) è il **piggybacking**, dove nel momento in cui un pacchetto stia trasportando dati dal dispositivo A al dispositivo B, vengono trasportati anche i riscontri ricevuti da A inerenti ai pacchetti ricevuti da B, in modo che entrambi i dispositivi ne siano a conoscenza, gestendo efficientemente il rinvio dei pacchetti.

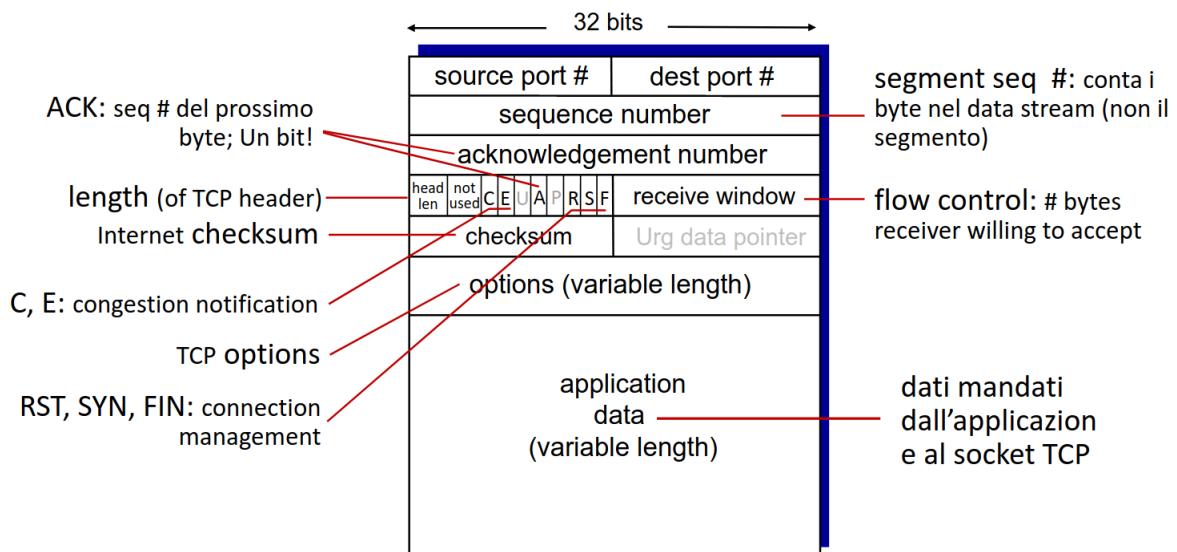
## 3.4 Trasporto orientato alla connessione (protocollo TCP)

Il **protocollo TCP** è un protocollo **end-to-end**, ossia con un solo mittente ed un solo destinatario, offerente un **byte stream affidabile e in ordine**, dove i messaggi del livello di applicazione vengono concatenati in un unico stream, a differenza dell'UDP, dove ogni messaggio è un segmento diverso.

Come già discusso, il protocollo TCP è **orientato alla connessione**, dove l'**handshaking** inizializza lo stato del mittente e del destinatario prima dello scambio dei dati.

Il flusso dati, inoltre è **full duplex**, ossia bidirezionale all'interno della stessa connessione (dati full duplex), limitati tuttavia da un **Maximum Segment Size (MSS)**. Tuttavia, è necessario sottolineare che si tratta di un paradigma diverso dalla commutazione di circuito, poiché la rete non è a conoscenza dello stabilimento della connessione.

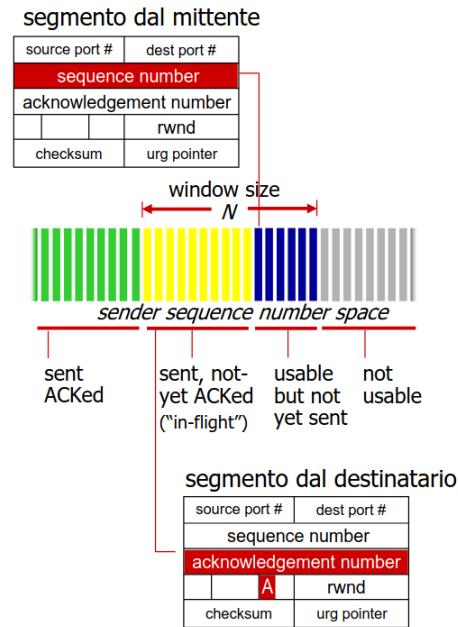
Per gestire il trasporto affidabile dei pacchetti, il protocollo TCP utilizza **ACK cumulativi** e **pipelining**, dove il window size dipende dal **flow control**, ossia una garanzia sul non sovraccarico del destinatario da parte del mittente, e dal **congestion control**, ossia una garanzia sol non sovraccarico della rete da parte del mittente.



Il **numero di sequenza** dei segmenti del protocollo TCP corrisponde al numero di sequenza del **primo byte del settore data** del segmento stesso. Per quanto riguarda l'**ACK**, viene utilizzato il numero di sequenza del **byte successivo aspettato**.

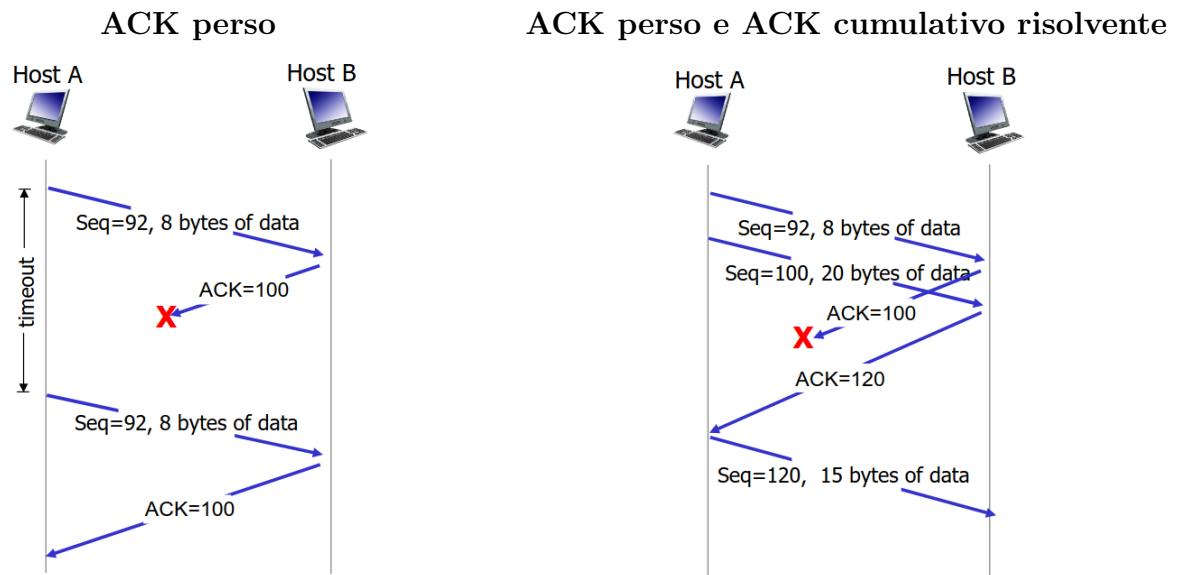
Nel momento in cui il mittente riceve dati dal livello di applicazione, viene creato il segmento e il suo numero di sequenza, avviando il timer a meno che esso non sia già in esecuzione. Inoltre, il **timer** utilizzato è **singolo** e collegato al **segmento non confermato più vecchio**. Allo scadere del **TimeoutInterval**, viene ritrasmesso il segmento che ha causato il timeout, riavviando il timer.

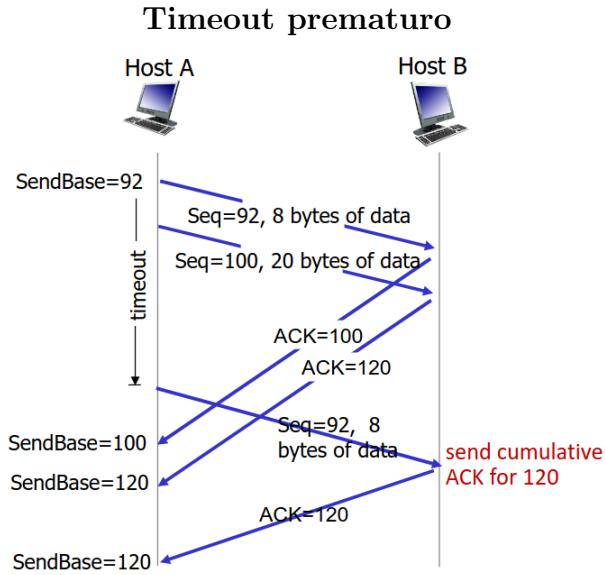
Alla **ricezione di un ACK**, invece, se quest'ultimo copre segmenti precedentemente non confermati (ricordiamo essere un **ACK cumulativo**), vengono aggiornate le informazioni di tali pacchetti, avviando il timer se vi sono ancora segmenti non confermati, il quale sarà collegato al nuovo segmento più vecchio (ibrido tra Go-back-N e Selective Repeat)



Per quanto riguarda il destinatario, invece, si hanno quattro scenari:

- All’arrivo di un **segmento in ordine**, con **numero di sequenza atteso** e tutti i segmenti **precedenti già confermati**, viene inviato un **delayed ACK**: dopo aver atteso 500 ms per il prossimo segmento, se quest’ultimo non è stato ricevuto viene inviato l’ACK
- All’arrivo di un **segmento in ordine**, con **numero di sequenza atteso** e ma con un segmento precedente **non ancora confermato**, viene immediatamente inviato un ACK cumulativo confermando entrambi i segmenti
- All’arrivo di un **segmento fuori ordine** e con numero di sequenza maggiore di quello atteso (dunque vi è un **gap**), viene inviato immediatamente un ACK duplicato
- All’arrivo di un segmento che in parte o totalmente **riempie in ordine un gap**, viene immediatamente inviato l’ACK





### 3.4.1 Gestione del timeout e stima del RTT

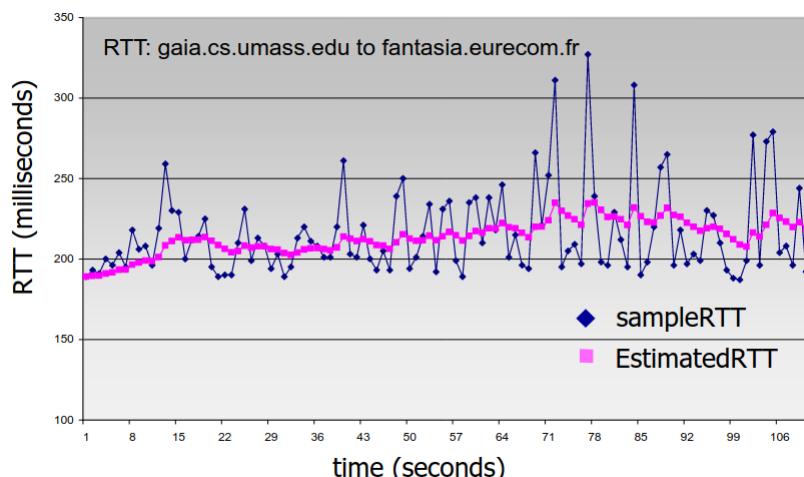
Il valore di timeout impostato deve essere **più lungo di un RTT**. Tuttavia, poiché il RTT è variabile, è necessario **stimarlo**.

Se il timeout scelto è **troppo corto**, si verificheranno troppi timeout prematuri, creando una serie di ritrasmissioni non necessarie. Se invece è **tropo lungo**, vi è una reazione troppo lenta a seguito della perdita di un pacchetto.

Per stimare il RTT, viene campionato un valore **SampleRTT**, ossia il tempo misurato dalla trasmissione del segmento fino alla ricezione dell'ACK (ignorando le ritrasmissioni). Poiché SampleRTT varia, viene utilizzata una **media delle misurazioni recenti** e non solo dell'ultimo SampleRTT (EWMA - Exponential Weighted Moving Average):

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

dove tipicamente si ha  $\alpha = 0.125$  e dove l'influenza del campione passato diminuisce in modo esponenziale



Il valore del **TimeoutInterval**, dunque, corrisponderà al valore attuale dell'EstimatedRTT sommato ad un **margine di sicurezza**

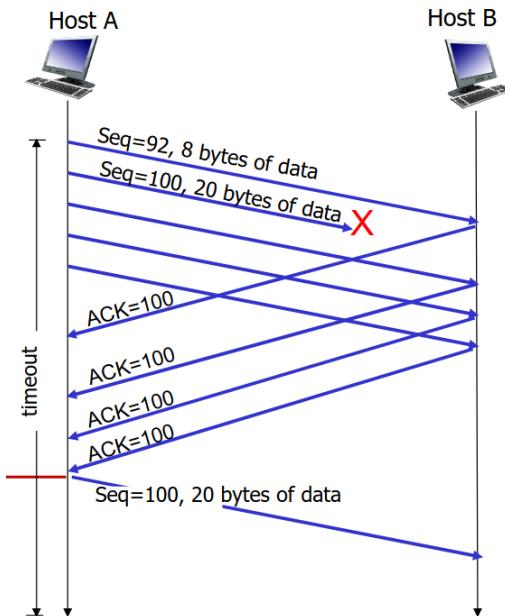
$$\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \cdot \text{DevRTT}$$

dove DevRTT è l'EWMA della deviazione di SampleRTT da EstimatedRTT

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

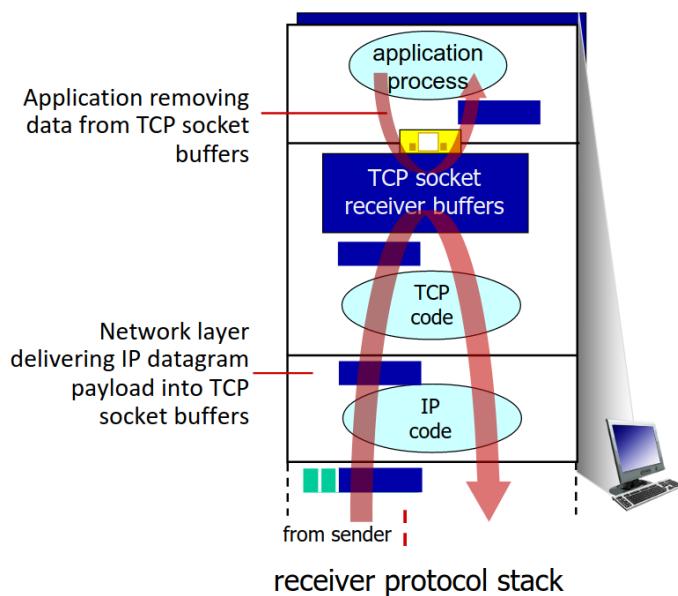
con un valore tipico  $\beta = 0.25$

Un'ottimizzazione ulteriore del protocollo TCP prevede l'implementazione del **fast retransmit**: se il mittente riceve 3 ACK aggiuntivi per gli stessi dati (dunque **tre ACK duplicati**), viene nuovamente inviato il segmento non confermato con numero di sequenza più piccolo poiché probabilmente tale segmento è andato perso, dunque non è necessario aspettare il timeout



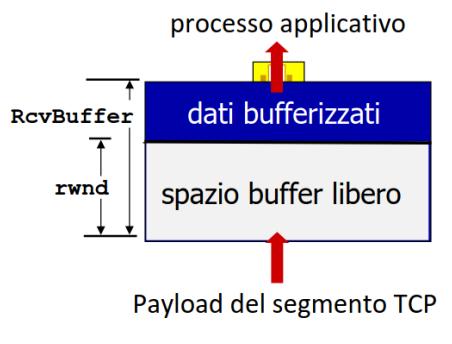
### 3.4.2 Controllo del flusso

Per poter funzionare correttamente, il protocollo TCP necessita di un **controllo del flusso**. Ad esempio, se la velocità con cui il livello di rete del destinatario fornisce i dati è maggiore rispetto a quella con cui il suo livello di applicazione rimuove i dati dal buffer del socket, il buffer andrà in **overflow**, implicando che i dati in eccesso vengano necessariamente **scartati**, risultando tuttavia come ricevuti correttamente dal destinatario.



Di conseguenza, è necessario che il **destinatario controlli il mittente**, impedendo che quest'ultimo possa riempire il buffer del destinatario trasmettendo troppi dati velocemente. Per gestire il flusso, quindi, viene utilizzata un campo **rwnd** (**receive window**) all'interno del segmento TCP:

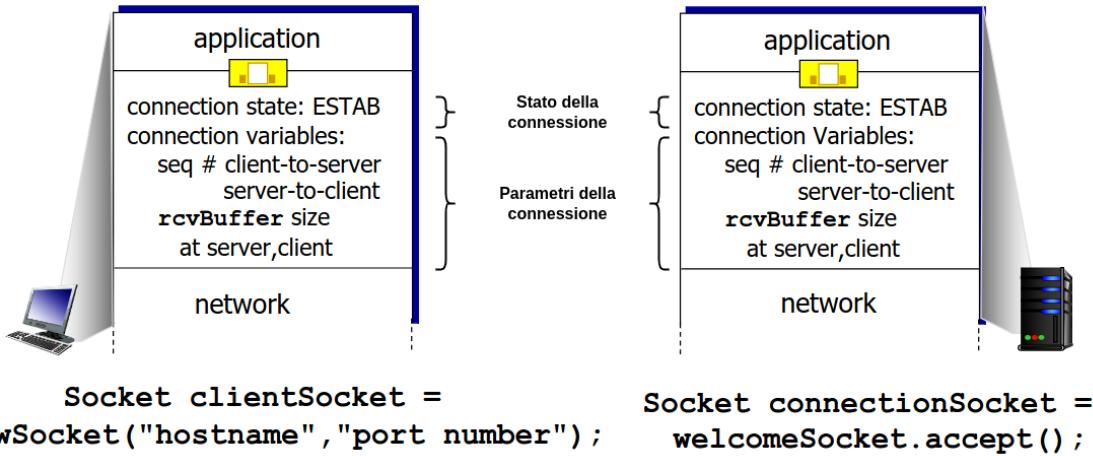
- Il destinatario inserisce in **rwnd** il numero di byte che è disposto ad accettare (dunque lo spazio rimanente nel buffer del socket)
- La dimensione del **RcvBuffer** impostata tramite le opzioni socket (predefinito a 4096 byte) o gestita automaticamente dal sistema operativo
- Il mittente limita la quantità di dati inviati senza ACK al valore di **rwnd**, garantendo che il buffer di ricezione non vada in overflow



Buffering lato destinatario TCP

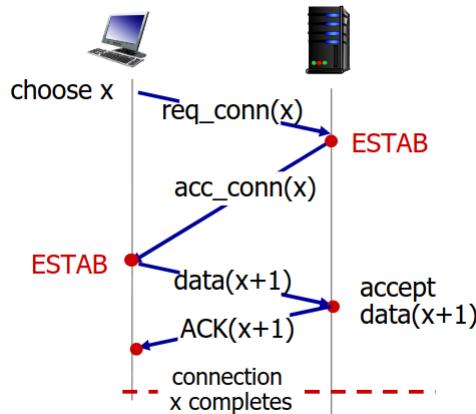
### 3.4.3 Gestione della connessione

Prima di effettuare lo scambio di dati, il mittente e il destinatario effettuano un **handshake**, dove viene determinata la disponibilità dell'un e dell'altro ad accettare di stabilire una connessione, concordando i parametri di quest'ultima (es: l'inizio del numero di sequenza)



Una prima implementazione dell'handshake è l'**handshake a 2 vie**, dove il mittente invia la richiesta di connessione al destinatario, il quale invia successivamente l'accettazione di tale richiesta, assumendo lo stato di connessione **ESTAB** (established).

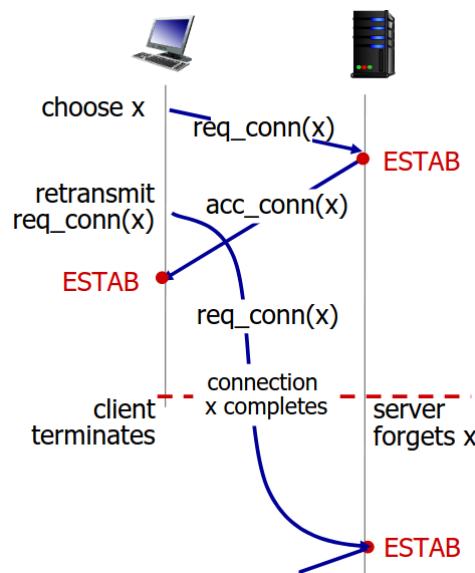
Una volta ricevuta l'accettazione, anche il mittente assumerà lo stato ESTAB, per poi procedere con l'invio effettivo dei dati.



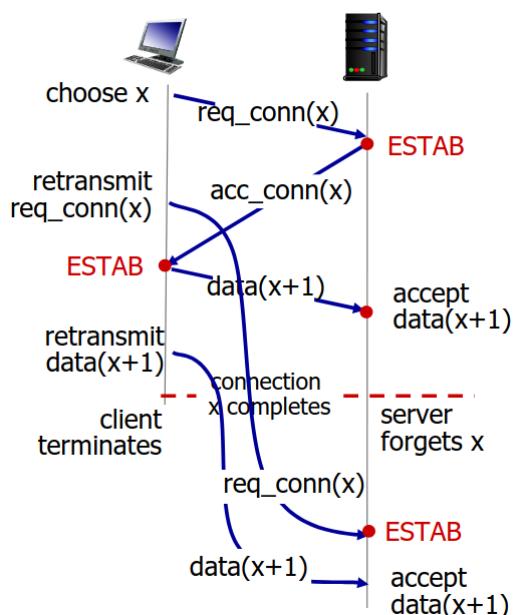
Tuttavia, in tale implementazione il destinatario **non è a conoscenza** della **ricezione** da parte del mittente del pacchetto di **accettazione** della connessione, presentando quindi **due problematiche fondamentali**:

- Nel caso in cui il mittente **rinvii la richiesta** di connessione allo scadere del timer TCP e l'accettazione del destinatario inerente alla prima richiesta giunge comunque dopo lo scadere del timer, il mittente suppone che la connessione sia andata a buon fine (nonostante il RTT sia estremamente basso), stabilendo quindi una **prima connessione**.

- Se tale connessione viene **terminata** prima che la seconda richiesta del mittente sia giunta al destinatario, quest'ultimo interpreterà la richiesta come una richiesta appartenente ad un'**seconda connessione**.
- Tuttavia, poiché tale richiesta era solo un rinvio della prima richiesta connessione, il client ignorerà la seconda richiesta di accettazione del server, creando quindi una **connessione fantasma senza client**



- Inoltre, nel caso in cui venga stabilita una connessione fantasma, si potrebbe andare incontro ad accettazioni di pacchetti dati duplicati

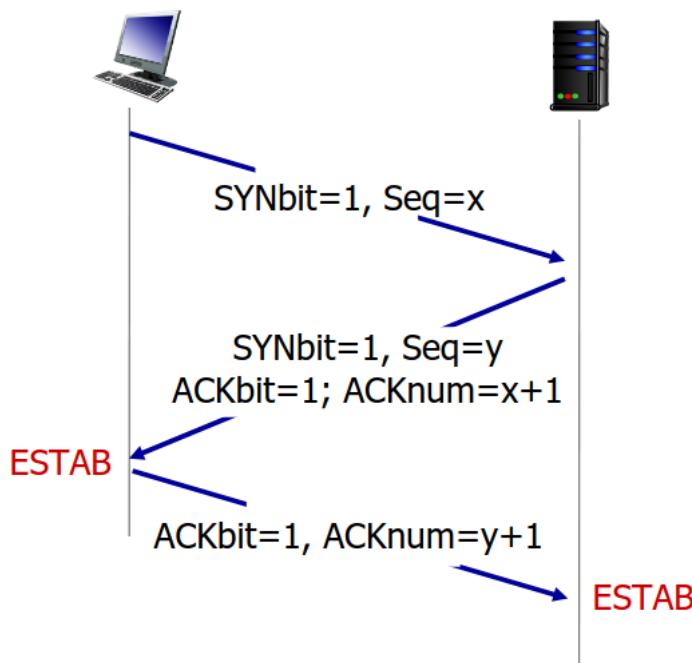


Di conseguenza, l'handshake TCP viene implementato attraverso uno scambio di 3 messaggi (**handshake a 3 vie**):

- Il mittente sceglie un numero di sequenza iniziale  $x$  e invia un pacchetto di tipo **SYN (synchronize)** al destinatario, richiedendo di stabilire una connessione.

Per inviare un pacchetto di tipo SYN, è sufficiente impostare il campo **SYN = 1** all'interno dell'header

- Una volta ricevuto il pacchetto SYN, il destinatario sceglie un numero di sequenza iniziale  $y$  e invia un pacchetto di tipo **SYN/ACK (synchronize and ACK)** al mittente, impostando i campi **SYN = 1** e **ACK = 1** nell'header
- Una volta ricevuto il pacchetto SYN/ACK, il mittente invia un pacchetto di tipo ACK (dunque con solo **ACK = 1**), passando in stato **ESTAB**
- Infine, una volta ricevuto il pacchetto ACK, anche il destinatario passerà in stato **ESTAB**



In questo modo, il destinatario sarà a conoscenza dello stato finale del mittente, risolvendo le due problematiche.

Per effettuare la **chiusura di una connessione**, il primo dispositivo (mittente o destinatario che sia) invia al secondo dispositivo un pacchetto di tipo **FIN (finished)**. Una volta ricevuto il pacchetto FIN, il secondo dispositivo risponderà con un pacchetto FIN/ACK, per poi inviare, dopo un breve lasso di tempo, un secondo pacchetto FIN. Analogamente, anche il primo dispositivo una volta ricevuto il pacchetto FIN invierà un pacchetto FIN/ACK.

Utilizzando tale **handshake a 4 vie**, entrambi i dispositivi riescono accertarsi il corretto termine della connessione. Inoltre, in tal modo entrambi i dispositivi possono terminare la connessione simultaneamente (creando una sorta di doppio handshake a 2 vie)

## 3.5 Controllo della congestione

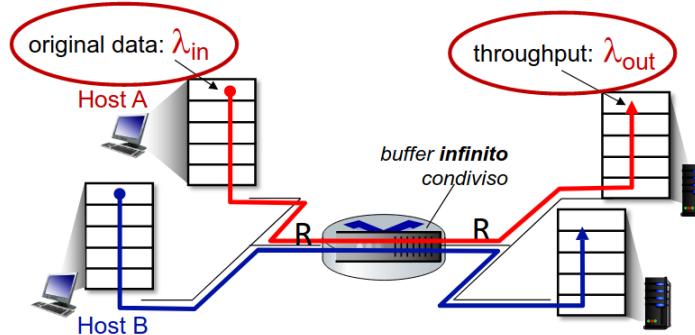
A differenza del **controllo del flusso**, il quale si occupa di gestire un mittente troppo veloce per un destinatario, il **controllo della congestione** si occupa di gestire situazioni in cui vi sono troppe fonti che inviano una grande quantità di dati troppo velocemente per poter essere gestiti correttamente dalla rete.

In presenza di congestione della rete, si manifestano **lunghi ritardi**, dovuti all'accodamento di troppi pacchetti nel buffer dei router, e **perdita di pacchetti**, dovuti agli overflow dei buffer dei vari router.

### 3.5.1 Cause e costi della congestione

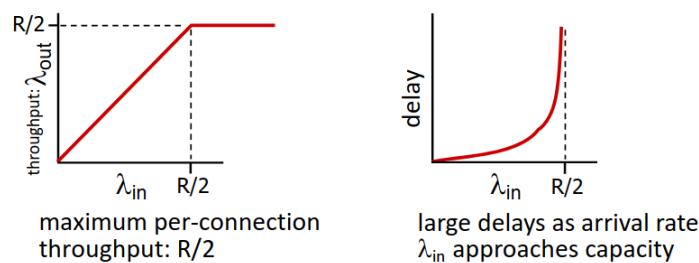
Consideriamo il seguente scenario:

- Vi sono due connessioni aperte passanti per un router con **buffer di dimensione infinita** e il transmission rate dei link è  $R$
- $\lambda_{in}$  è l'**arrival rate del router**, la quantità di dati inviati da un host della prima rete al router
- $\lambda_{out}$  è il **throughput del router**, la quantità di dati inviati dal router ad un host della seconda rete



In tal caso, poiché il buffer è infinito ci troviamo in una situazione in cui non sono necessarie ritrasmissioni dovute alla perdita del pacchetto. Di conseguenza, l'**arrival rate** riesce ad essere equivalente al **throughput**, corrispondente alla quantità di dati in uscita dal router.

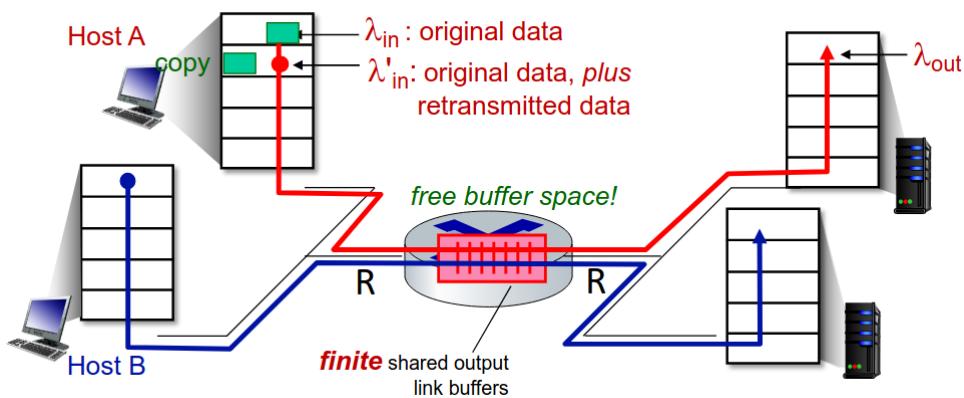
Tuttavia, poiché sono aperte due connessioni passanti per il router, il throughput massimo di ognuna di esse corrisponde a  $\frac{R}{2}$ . Inoltre, anche in tale scenario perfetto, man mano che  $\lambda_{in}$  si avvicina a  $\frac{R}{2}$ , il **delay cresce notevolmente**, per via carico eccessivo sui link stessi della rete.



Nella vita reale, ovviamente, la dimensione dei buffer è **finita**, implicando che alcuni pacchetti possano andar persi, venendo ritrasmessi dal mittente a seguito dello scadere del timeout.

Dato l'arrival rate  $\lambda'_{in}$  dei **dati originali sommati ai dati ritrasmessi**, è necessario sottolineare che:

- Al livello di applicazione, la quantità di dati inviati è equivalente a quella dei dati ricevuti, dunque si ha che  $\lambda_{in} = \lambda_{out}$
- Al livello di trasporto, tuttavia, l'input contiene anche i dati ritrasmessi, implicando che  $\lambda'_{in} \geq \lambda_{in}$



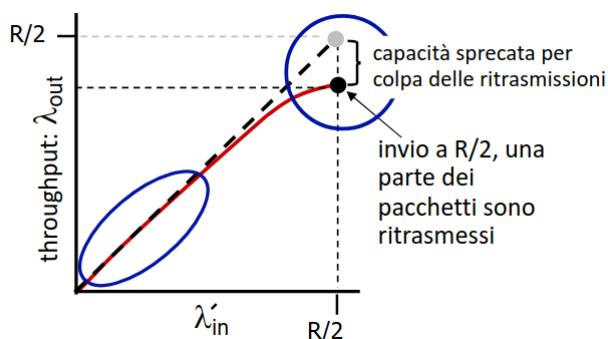
A questo punto, procediamo per **assunzioni** per studiare come la congestione influenzi l'infrastruttura:

- Idealmente, possiamo assumere che il mittente vada ad inviare i dati **soltamente** nel caso in cui esso sappia che i buffer dei router abbiano abbastanza spazio per ricevere il pacchetto (assunzione **perfect knowledge**)

In tal caso, ci troveremmo in una situazione identica allo scenario perfetto, poiché la rete sarebbe in grado di gestire il carico senza problemi, inviando i dati da un router all'altro al loro arrivo.

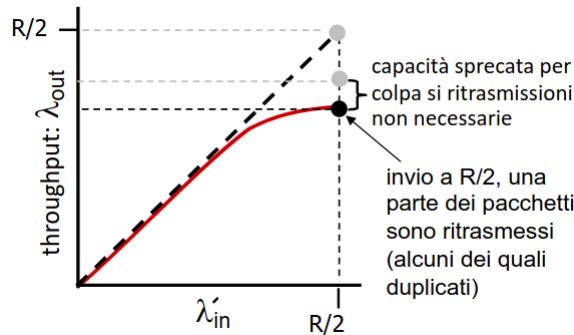
- In uno scenario più realistico, assumiamo che i pacchetti possano essere scartati a seguito di un **overflow** di un buffer e che il mittente sia a conoscenza perfetta di quali pacchetti siano andati persi, ritrasmettendoli (**perfect knowledge parziale**).

In tal caso, parte della capacità dei link verrebbe sprecata per via delle ritrasmissioni, **diminuendo il throughput**

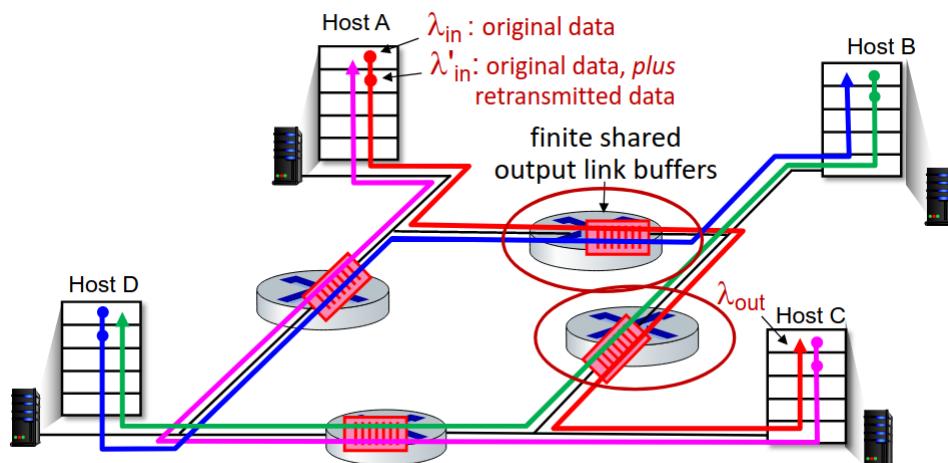


- In uno scenario reale, oltre alla perdita di pacchetti dovuta ad un overflow dei buffer, il **timer** del mittente può scadere prematuramente, inviando due copie dello stesso pacchetto ritrasmesso (**duplicati non necessari**)

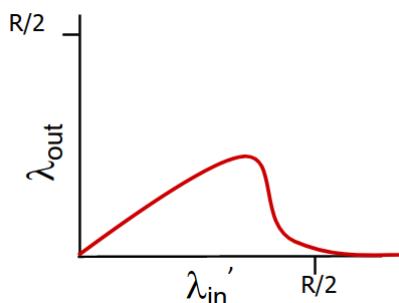
In tal caso, ulteriore parte della capacità dei link verrebbe sprecata per via delle ritrasmissioni non necessarie, **diminuendo il throughput ulteriormente**



Consideriamo invece ora una rete più realistica in cui tutti e quattro i dispositivi sono mittenti e vi siano più router colleganti le loro reti (rete multi-hop).



All'aumentare dei valori  $\lambda_{in}^{red}$  e  $\lambda_{in}^{red'}$  del collegamento rosso mostrato in figura, **tutti i pacchetti del collegamento blu vengono scartati**, poiché il buffer del router viene riempito dai pacchetti del collegamento rosso rinviati, portando il valore  $\lambda_{out}^{blue}$  a tendere a 0, diminuendo il throughput generale della rete



Possiamo quindi riassumere il comportamento della rete nei seguenti punti:

- Il throughput non può mai superare la capacità
- Il ritardo aumenta con l'avvicinarsi alla capacità
- La perdita e ritrasmissione riduce il throughput effettivo
- I duplicati non necessari riducono ulteriormente il throughput effettivo
- Viene sprecata capacità di trasmissione e buffering upstream per i pacchetti persi downstream

Infine, utilizziamo tali punti per definire i **costi della congestione**:

- È necessario un **lavoro** (numero di ritrasmissioni) **maggiori** per un dato throughput durante la congestione
- Il collegamento trasporta **più copie** dello stesso pacchetto **non necessarie**, riducendo il throughput massimo ottenibile
- Quando un pacchetto viene scartato, tutta la capacità di trasmissione upstream e la porzione di buffer utilizzata per esso viene **sprecata**

### 3.5.2 Controllo della congestione nel TCP

Per tentare di gestire la congestione, vengono principalmente utilizzati due approcci:

- **Controllo della congestione end-to-end**, dove non viene ricevuto alcun feedback esplicito dalla rete e la congestione viene **dedotta** dalle perdite e ritardi osservati dal mittente e il destinatario.

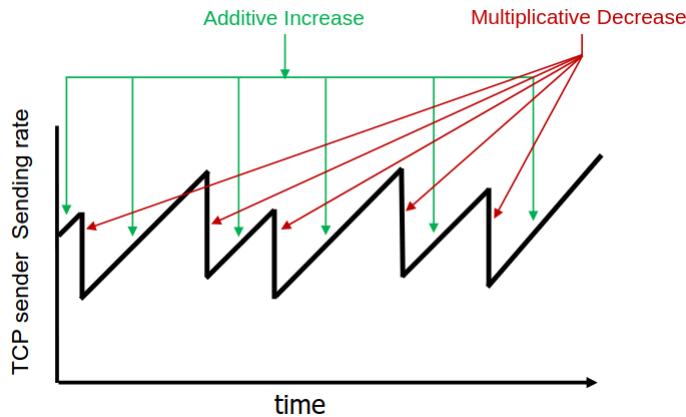
Tale approccio è adottato dal protocollo TCP

- **Controllo della congestione assistito dalla rete**, dove i router forniscono un feedback **diretto** agli host di invio/ricezione con flussi che passano attraverso router congestionati, indicando, in alcuni casi, direttamente il livello di congestione o la velocità di invio impostata

#### Definition 30. Algoritmo AIMD

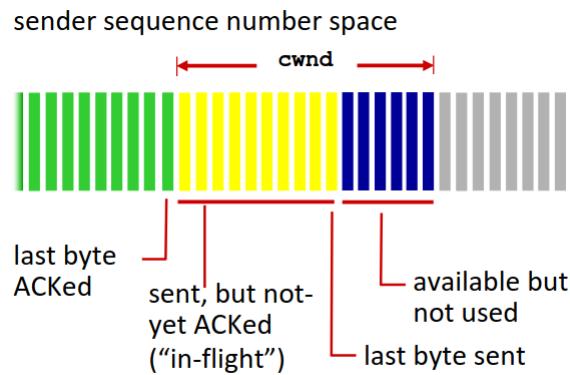
L'algoritmo **AIMD** (**Additive Increase, Multiplicative Decrease**) è un algoritmo utilizzato da alcune versioni del protocollo TCP per **prevenire la congestione**, dove i mittenti possono **aumentare la velocità di invio** fino a quando si verifica una **perdita di pacchetti**, per poi diminuirla:

- **Additive Increase**: il rate viene aumentato di 1 MSS (che ricordiamo essere il Maximum Segment Size) ad ogni RTT fino a quando una perdita non viene osservata
- **Multiplicative Decrease**: ad ogni perdita osservata, il rate di invio viene diviso per due



Approssimativamente, il protocollo TCP assume il seguente comportamento:

- Viene inviata una quantità di byte pari ad un valore **cwnd** (congestion window).



- Il mittente limita la trasmissione a

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

- Viene atteso un RTT per ogni ACK, per poi scegliere di inviare più byte

$$\text{TCP rate} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ B/s}$$

- Il valore **cwnd** varia dinamicamente reagendo alla congestione osservata. In particolare, utilizzando l'**algoritmo AIMD** ad ogni ACK ricevuto si ha che:

$$\text{cwnd} \approx \text{prev\_cwnd} + \left( \text{MSS} \cdot \frac{\text{MSS}}{\text{prev\_cwnd}} \right)$$

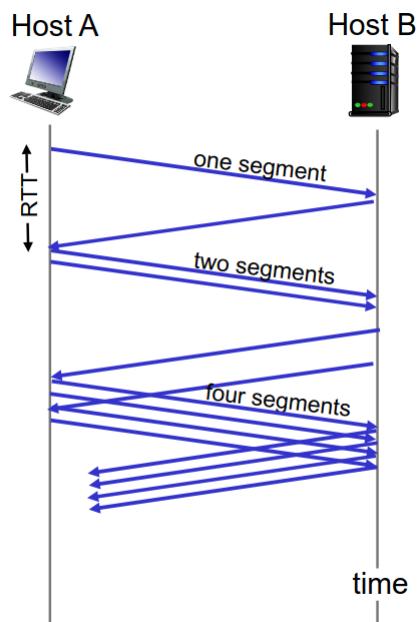
### Definition 31. Fast recovery

Il **fast recovery** è un criterio utilizzato da alcune versioni del protocollo TCP per prevenire la congestione, dove ad ogni perdita rilevata a seguito di un **triplo ACK** duplicato viene **dimezzato il rate di invio**

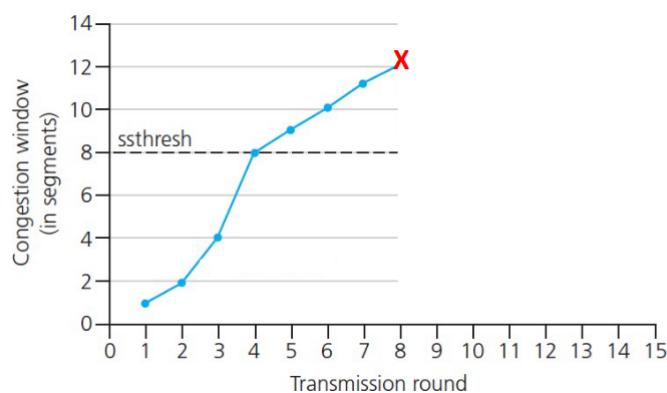
### Definition 32. Slow start

Lo **slow start** è un criterio utilizzato da alcune versioni del protocollo TCP **prevenire la congestione**, dove:

- Il valore cwnd viene **impostato ad 1 MSS** all'inizio della connessione o a seguito di un **timeout**
- Successivamente, il valore di cwnd viene **raddoppiato ad ogni RTT**, fino al rilevamento della prima perdita di pacchetto (**incremento esponenziale**)

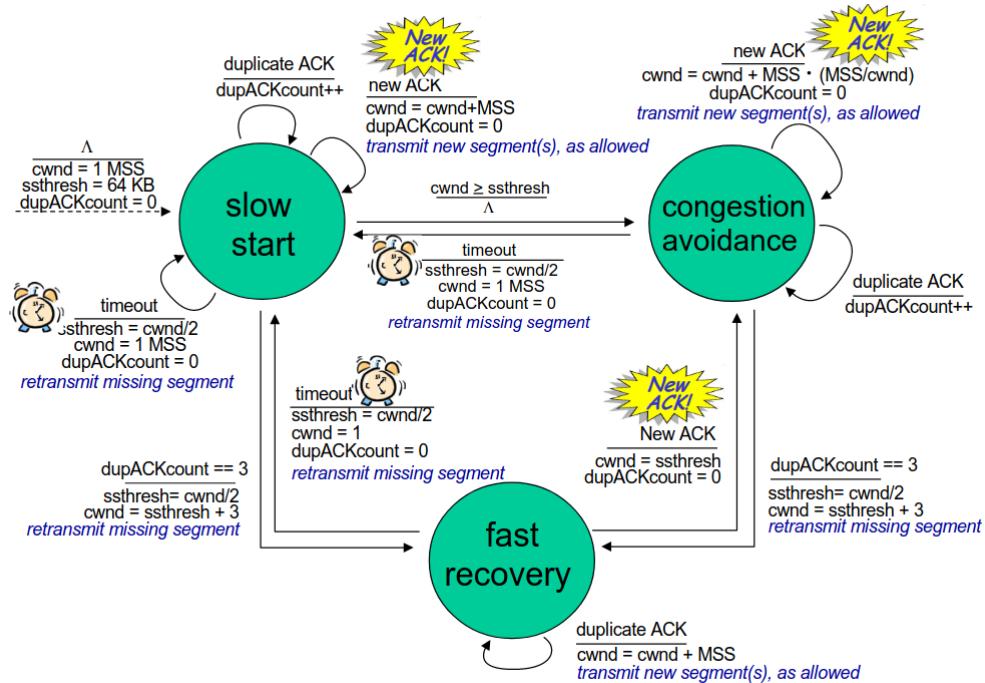


In particolare, per effettuare il **passaggio tra slow start e AIMD**, viene utilizzato un valore **ssthresh** (**slow start threshold**). A seguito del rilevamento di una perdita di pacchetto, il valore ssthresh viene impostato a  $\frac{\text{cwnd}}{2}$  (con il valore di cwnd precedente alla perdita)



Tali meccanismi di prevenzione della congestione di rete vengono utilizzati principalmente da due versioni del protocollo TCP:

- **TCP Tahoe**: viene utilizzato l'**algoritmo AIMD**, lo **slow start** e il fast retransmit
- **TCP Reno**: analogo al TCP Tahoe, ma con l'utilizzo aggiuntivo del **fast recovery**



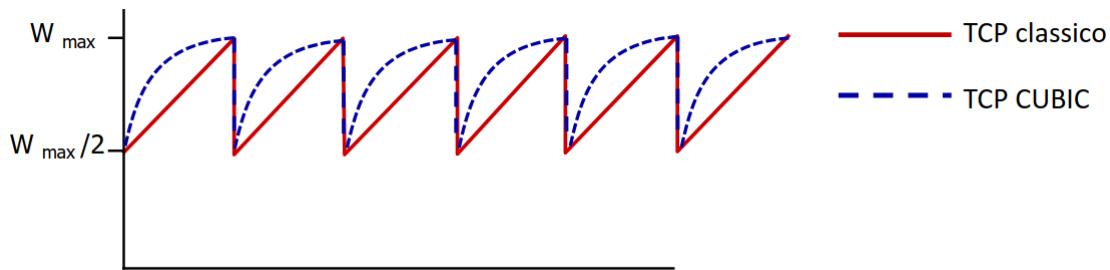
### Definition 33. TCP CUBIC

Il **TCP CUBIC** è una versione di TCP utilizzante l'algoritmo **CUBIC** per **prevenire la congestione**, il quale è definito dalla seguente logica:

- Viene utilizzato un valore  $W$ , corrispondente alla **velocità di invio** (o alla quantità di dati inviati dal mittente, ossia  $cwnd$ ). Ad ogni perdita rilevata, viene **dimezzato** tale valore.
- Il limite superiore  $W_{max}$  corrisponde alla **velocità di invio** nel momento in cui è stata rilevata una **perdita di pacchetto** (viene assunto che a seguito della perdita lo stato di congestione della rete non sia variato di molto)
- Viene utilizzato un valore  $K$  corrispondente al **momento di tempo stimato** in cui il valore  $W$  raggiungerà il limite superiore  $W_{max}$  (la modalità di stima viene definita dallo standard RFC 8312)
- Il valore  $W$  viene **incrementato in funzione del cubo della distanza tra il tempo corrente e  $K$** .

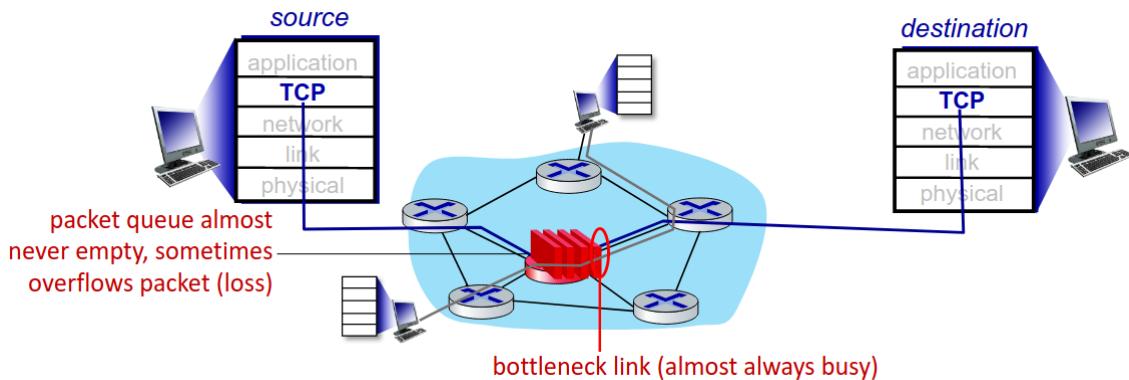
Di conseguenza, il valore  $W$  **crescerà molto rapidamente** quando il tempo corrente è lontano dal valore  $K$ , mentre **crescerà lentamente** al suo avvicinarsi.

Supponendo, ad esempio, che il valore di  $W_{max}$  rimanga sempre lo stesso nel tempo, il throughput del TCP CUBIC rispetto a quello standard risulta più elevato:

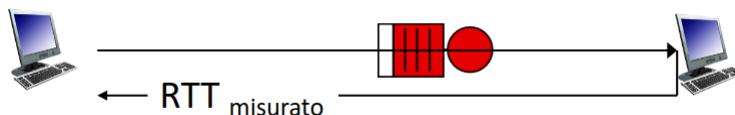


In particolare, essendo predefinito sul sistema operativo Linux, il TCP CUBIC è la versione del protocollo TCP più diffusa nei web server.

Una volta aver visto algoritmi e tecniche per prevenire la congestione, possiamo concentrarci sul link di uscita del stesso in cui si verifica la perdita, ossia il **bottleneck link**



In presenza di un bottleneck link, **aumentando il rate di invio** non aumenterà il throughput per via del bottleneck link ma **aumenterà il RTT misurato**.



Una soluzione ottimale, dunque, è quella di mantenere il percorso end-to-end **quasi pieno**, ma senza superare la soglia stabilità, utilizzando un approccio **delay-based**:

- Il valore  $RTT_{min}$  è il RTT minimo osservato dal mittente, corrispondente quindi al RTT osservato quando il percorso **non è congestionato**
- Approssimativamente, il **throughput misurato** ad ogni RTT corrisponde a:

$$\text{Throughput}_{\text{misurato}} = \frac{\text{Byte}_{\text{RTT}}}{\text{RTT}_{\text{misurato}}}$$

dove  $\text{Byte}_{\text{RTT}}$  è il numero di byte inviati nell'ultimo RTT, mentre il **throughput non congestionato** è pari a:

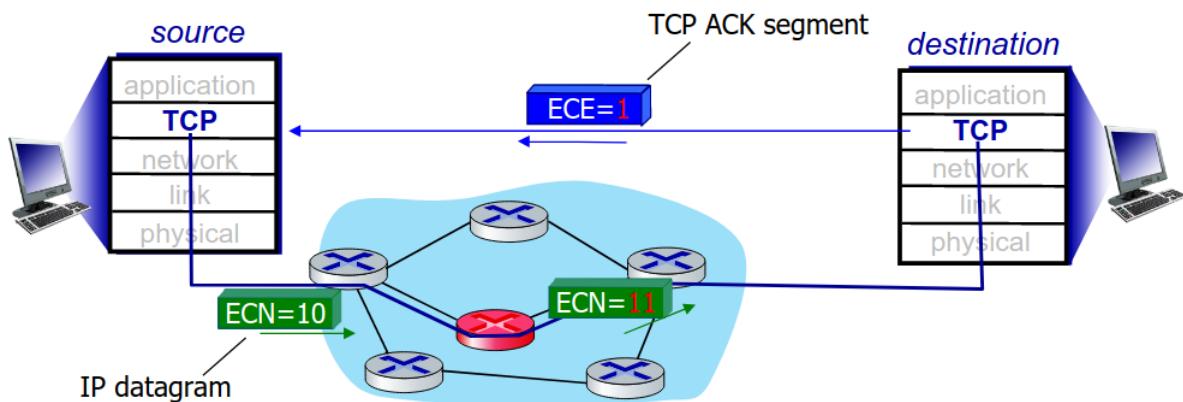
$$\text{Throughput}_{\text{non cong}} = \frac{\text{cwnd}}{\text{RTT}_{min}}$$

- Se il valore del throughput misurato è **molto vicino** a quello non congestionato, il valore di **cwnd** viene **incrementato** in modo lineare (dunque +1 MSS ad ogni RTT), mentre se è **molto inferiore** viene **decrementato** in modo lineare

Tramite tale approccio, alcune versioni di TCP (es: protocollo BBR) riescono ad indurre un controllo della congestione senza forzare delle perdite di pacchetto, massimizzando il throughput ma mantenendo basso il ritardo.

Altre versioni di TCP (es: TCP ECN), invece, implementano anche la seconda modalità di controllo della congestione, ossia il **controllo della congestione assistito dalla rete**, dove:

- Due bit dell'**header del livello di rete** vengono contrassegnati dal **router** per indicare lo stato della congestione
- Una volta raggiunto il destinatario, quest'ultimo imposterà il bit ECE sul segmento ACK per notificare al mittente lo **stato della congestione**



## 3.6 Equità nei protocolli di trasporto

### Proposition 2. Equità nelle connessioni

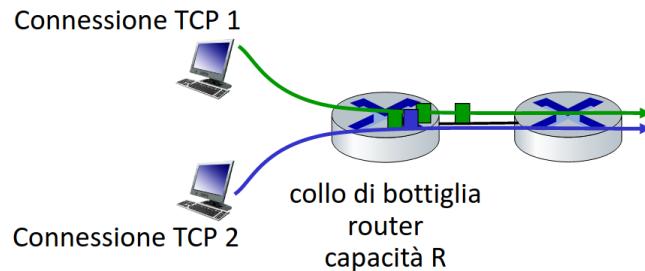
Affinché un protocollo di trasporto sia definibile **equo**, se  $K$  **sessioni** di tale protocollo condividono lo stesso **bottleneck link** con larghezza di banda  $R$ , ciascuna delle  $K$  sessioni deve avere una velocità media pari a  $\frac{R}{K}$

Notiamo con facilità che il **protocollo UDP** sia un protocollo **non equo** per via dell'assenza di un controllo della congestione e di limiti sulla banda utilizzabile.

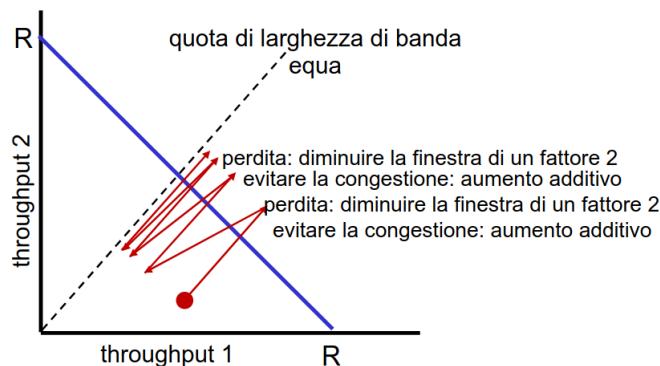
Per tale motivo, spesso applicazioni multimediali utilizzanti il protocollo UDP (es: i servizi streaming) "rubano" velocità di connessione ad altre applicazioni.

Per quanto riguarda il protocollo TCP, invece, è necessario effettuare uno studio:

- Consideriamo il seguente scenario con due sessioni TCP con algoritmo AIMD concorrenti sullo stesso bottleneck link



- Tramite l'**additive increase** viene generata una pendenza pari ad 1
- Tramite il **multiplicative decrease** viene ridotto proporzionalmente il throughput



Sotto **ipotesi idealizzate**, dunque, il **protocollo TCP** risulta essere **equo** (es: stesso RTT, numero fisso di sessioni, ...).

Tuttavia, molte applicazioni moderne utilizzano **più di una connessione TCP parallela** tra due host (es: un web browser). Di conseguenza, anche se la larghezza di banda fosse equamente distribuita tra tutte le connessioni possibili tra i due host, tale applicazione otterrebbe comunque una quantità di banda superiore alle altre applicazioni.

### Esempio:

- Consideriamo due host tra cui sono già aperte 9 connessioni TCP
- Se una nuova applicazione richiedesse 1 connessione TCP, tale applicazione otterrebbe una larghezza di banda pari a  $\frac{R}{10}$
- Se invece tale applicazione richiedesse 11 connessioni TCP, tale applicazione otterrebbe una larghezza di banda leggermente superiore a  $\frac{R}{2}$