

# Reti di Calcolatori a.a. 2019/2020

Francesco Iannelli

September 16, 2019

## Contents

<b>1</b>	<b>Introduzione al corso</b>	<b>4</b>
<b>2</b>	<b>Cenni di modelli stratificati</b>	<b>4</b>
2.1	Livello applicativo . . . . .	5
2.2	Livello di trasporto . . . . .	5
2.3	Livello rete . . . . .	5
2.4	Livello link . . . . .	5
<b>3</b>	<b>Introduzione alle reti</b>	<b>6</b>
3.1	LAN . . . . .	6
3.2	WAN . . . . .	7
3.3	Internetwork . . . . .	7
3.3.1	Reti a commutazione di circuito . . . . .	8
3.3.2	Reti a commutazione di pacchetto . . . . .	8
3.3.3	Packet Switch e Circuit Switch a confronto . . . . .	9
3.3.4	Circuiti virtuali . . . . .	9
3.3.5	Datagram Network . . . . .	9
3.4	Internet . . . . .	10
3.4.1	Servizi . . . . .	11
3.4.2	IETF/RFC/ICANN . . . . .	11
3.4.3	Rete di accesso . . . . .	11
<b>4</b>	<b>Metriche di riferimento</b>	<b>12</b>
4.1	Ritardi . . . . .	13
4.1.1	Ritardo di elaborazione del nodo . . . . .	13
4.1.2	Ritardo di accodamento . . . . .	13
4.1.3	Ritardo di trasmissione . . . . .	13
4.1.4	Ritardo di propagazione . . . . .	13
4.2	Esempio . . . . .	14
4.3	Volume di un link . . . . .	14
4.4	Esempio . . . . .	14
<b>5</b>	<b>Modelli Stratificati e Protocolli</b>	<b>15</b>
5.1	OSI: Open Systems Interconnection . . . . .	16
5.1.1	Modello ISO/OSI . . . . .	16
5.2	Protocollo . . . . .	17
5.3	Incapsulamento dell'informazione . . . . .	18
<b>6</b>	<b>Stack protocollare TCP/IP</b>	<b>19</b>
6.1	Lo strato applicativo . . . . .	20
6.1.1	Web . . . . .	23
6.1.2	HTTP Protocol . . . . .	24
6.1.3	TELNET . . . . .	31

6.1.4	SSH . . . . .	34
6.1.5	FTP . . . . .	37
6.1.6	DNS . . . . .	39
6.1.7	EMAIL - SMTP . . . . .	46
6.2	Lo strato di trasporto . . . . .	53
6.2.1	UDP . . . . .	62
6.2.2	TCP . . . . .	64
<b>7</b>	<b>Esercizi</b>	<b>84</b>

# 1 Introduzione al corso

email: federica.paganelli@unipi.it

modalità d'esame: prova scritta (compitini) e orale (facoltativo) più laboratorio che prevede progetto e orale.

**N.B. Si accede alla prova orale di laboratorio solo dopo aver passato lo scritto.**

## 2 Cenni di modelli stratificati

Sono architetture di comunicazione a strati.

Concetti generali:

- Stratificazione
- Information hiding
- Separation of concerns

Vantaggi della stratificazione:

- **Facilità di progettazione.**
- **Facilità di manutenzione.**
- **Possibilità di riciclo.**

Due modelli: **ISO/OSI** (approccio top-down) e **Stack TCP/IP** (approccio bottom-up), quest'ultimo vincente.

Idea chiave: suddivisione in sottoproblemi.



Figure 1: Modello stratificato

## 2.1 Livello applicativo

Fanno parte del livello applicativo:

- Identificativi delle risorse: URL, URI e URN.
- Il web: user agents, protocollo http.
- Protocollo FTP.
- TELNET: servizio di terminale virtuale.
- Posta elettronica.
- Sistema dei nomi DNS a dominio e la risoluzione dei nomi: iterativa e ricorsiva.
- molto altro ancora...

## 2.2 Livello di trasporto

Due tecnologie degne di nota:

1. Protocollo **TCP**: **connection-oriented**, *orientato alla connessione*.
2. Protocollo **UDP**: **connection-less**, molto più leggero, prende dati applicativi e li affida allo strato IP, NON da garanzie di consegna né di ordine.

## 2.3 Livello rete

Nel livello rete si ricava un percorso dall'host sorgente all'host destinatario usando le informazioni che si trovano nell'IP.

Verrà trattato il protocollo Ipv4 e introdotto il protocollo Ipv6.

## 2.4 Livello link

Si occupa di gestire il collegamento tra due nodi **adiacenti**. La tecnologia principale è l'**ethernet**.



Figure 2: Modello stratificato

### 3 Introduzione alle reti

*Cos'è una rete? Quante tipologie di reti ci sono? Cos'è internet?*

**Definizione 3.1.** Una **rete** è un'interconnessione di dispositivi in grado di scambiarsi e interpretare le informazioni. Comprende sistemi terminali e intermedi: *e.g. router, switch e modem*.

I sistemi terminali si possono dividere in due tipi: **host** e **server**, sull'host girano le applicazioni utente mentre il server esegue programmi che forniscono servizi applicativi ad applicazioni.

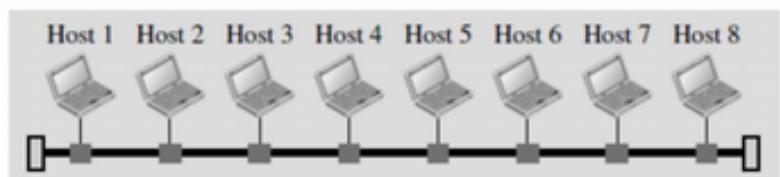
*N.B. il termine host può essere usato per indicare anche un server.*

*L'host, infatti, può essere sia un server sia il terminale di un utente che esegue un'applicazione client, più generalmente l'host è una macchina.*

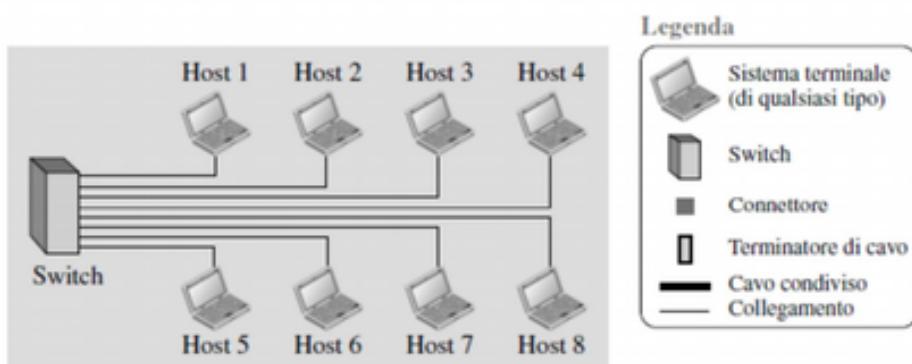
**Definizione 3.2.** Una rete è formata da **dispositivi** e da **tecnologie**.

#### 3.1 LAN

Acronimo di **Local Area Network**, è una rete di area geografica limitata collegata attraverso una tecnologia ethernet **bus** o ethernet **switch**: Ciascun



a. LAN con cavo condiviso (obsoleta)



b. LAN con switch (moderna)

host ha un cavo che lo collega allo switch e a ogni porta dello switch corrisponde un host. Lo **switch** possiede una tecnologia di autoapprendimento ed è una componente del **livello link**.

### 3.2 WAN

Acronimo di **Wide Area Network**, è una rete di area geografica estesa: è composta da due o più reti collegate tramite un mezzo di trasmissione. Le reti coinvolte potrebbero anche essere reti LAN. (*e il link potrebbe essere affittato a un'azienda da un operatore di telefonia*).

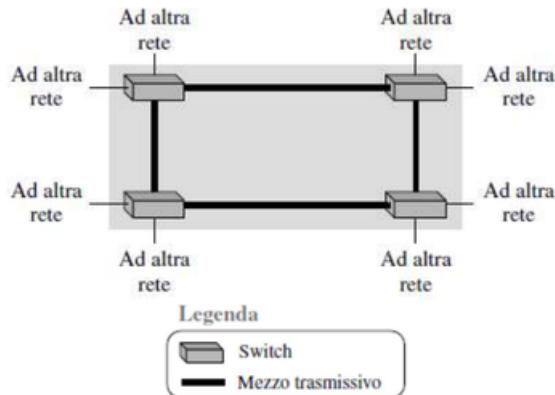


Figure 3: Un esempio di WAN a cavo condiviso e a commutazione.

Queste WAN permettono l'esistenza di **percorsi alternativi** e la **divisione del traffico**. Una **WAN punto a punto** invece ha solo 2 punti di terminazione.

### 3.3 Internetwork

L'internetwork è un sistema in cui ci sono più reti composte, capaci di scambiarsi informazioni e collegate. Concettualmente è una WAN ma è più complicata.

I dispositivi che la compongono si distinguono in **sistemi terminali** e dispositivi come gli **switch** e i **routers** che si trovano nel percorso tra i sistemi sorgente e i sistemi destinazione.

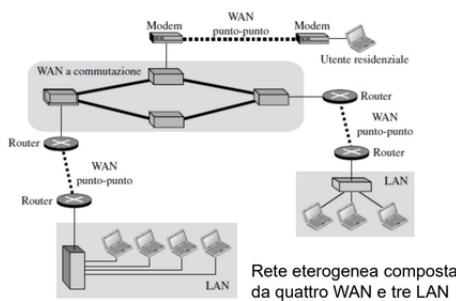


Figure 4: Una internetwork.

*Problema: come mandare informazioni da un host a un altro?*

### 3.3.1 Reti a commutazione di circuito

Nelle **reti a commutazione di circuito** le **risorse** sono **riservate end to end** per ogni connessione. Risulta quindi necessario il **setup della comunicazione** per instaurare la connessione ed elargire le risorse.

La risorsa non è tutto il link, bensì si considerano come risorse la capacità o la larghezza di banda porzionate per ogni connessione. Le risorse assegnate rimangono inattive se non utilizzate (*e.g. telefonata*). I dispositivi mantengono lo stato della connessione. **Le performance sono garantite**. La capacità delle linee (*link*) cambia a seconda della loro funzione all'interno della rete. Il **punto debole** delle reti a commutazione di circuito è la **poca flessibilità nel dispiegamento delle risorse**.

### 3.3.2 Reti a commutazione di pacchetto

Nelle **reti a commutazione di pacchetto** gli utenti inviano pacchetti che condividono le risorse del canale di comunicazione. **Non c'è** quindi **un canale dedicato** ai pacchetti di un singolo utente. La principale differenza rispetto alla commutazione di circuito risiede nell'implementazione della logica dei dispositivi di interconnessione, ovvero:

- **Commutazione di Circuito:** avviene il **setup** della connessione dove si prealloca l'utilizzo del collegamento trasmissivo con collegamenti garantiti.
- **Commutazione di Pacchetto:** non viene instaurata una connessione bensì le informazioni necessarie si trovano all'interno dei pacchetti stessi, non ci sono informazioni di connessione memorizzate nei dispositivi coinvolti.

Nelle reti a commutazione di pacchetto quindi le risorse vengono usate a seconda della necessità. Possono quindi verificarsi situazioni di contesa delle risorse e sussiste il pericolo di congestione o di perdita dei pacchetti nel caso in cui la dimensione della coda del router non fosse sufficiente a contenere il flusso dei pacchetti entranti: il commutatore (*router*) deve infatti ricevere l'intero pacchetto prima di poter cominciare a trasmetterlo sul collegamento in uscita (**store and forward**). **Non sono** quindi **garantite le prestazioni**.

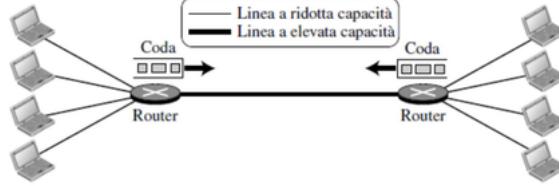


Figure 5: Rete a commutazione di pacchetto. Da notare le diverse capacità dei link.

### 3.3.3 Packet Switch e Circuit Switch a confronto

Vi sono 35 utenti su una rete con 100 Kbit/s di connessione e un link da 1 Mbit/s. Ogni utente è attivo solo il 10% del tempo.

Con una **rete a commutazione di circuito** si riescono a gestire **solo** 10 utenti.

Con una **rete a commutazione di pacchetto** si hanno i seguenti casi:

1. 10 o meno utenti attivi: nessun problema.
2. Più di 10 utenti attivi: ritardo.

Tuttavia che gli utenti siano tutti e 35 attivi contemporaneamente è poco probabile (infatti  $P(35) = 0.0004$ ). Se ne deduce che la rete a commutazione di pacchetto riesce a gestire tutti gli utenti contemporaneamente nella maggior parte dei casi.

Nonostante il risultato ottenuto non si deve pensare che la rete a commutazione di circuito sia obsoleta. Nel corso degli anni infatti le due tecnologie sono state **integrate** in vari modi.

La commutazione di circuito infatti è usata nella telefonia fissa (*PSTN: public switch telephone network*) per i servizi voce, la commutazione di pacchetto invece per i dati.

Nelle reti ottiche di prima e seconda generazione si usano entrambe le tecnologie.

### 3.3.4 Circuiti virtuali

I circuiti virtuali funzionano nel seguente modo: viene stabilito un path tra host sorgente e host destinazione e tutti i pacchetti di un certo flusso seguono lo **stesso** path.

### 3.3.5 Datagram Network

Con **datagram** si indica un'entità informativa autocontenuta che contiene le informazioni sufficienti per essere indirizzata alla destinazione senza comunicazioni aggiuntive tra sorgente e destinazione: **non è quindi detto** che pacchetti di uno stesso flusso seguano lo stesso path sulla rete.

### 3.4 Internet

Come interconnettere reti già esistenti?

**Definizione 3.3.** Una internet (con i minuscola) è una rete costituita da due o più reti interconnesse.

La internet più famosa è chiamata **Internet** (con i maiuscola) ed è composta da migliaia di reti interconnesse. Ogni rete connessa ad Internet deve usare il protocollo IP e rispettare certe convenzioni su come vengono assegnati nomi e indirizzi. Si possono facilmente aggiungere nuove reti.

Tuttavia è impensabile avere un link fisico tra ogni host, si hanno invece numerosi dispositivi di interconnessione che permettono la comunicazione da un host all'altro e da un router all'altro.

Uno scorcio delle **componenti di Internet**:

- Miliardi di dispositivi interconnessi (e.g. hosts, end systems).
- Link di comunicazione (e.g. fibre ottiche, doppini telefonici, cavi coaxiali, onde radio).
- Routers: instradano pacchetti (*sequenze*) di dati attraverso la rete.

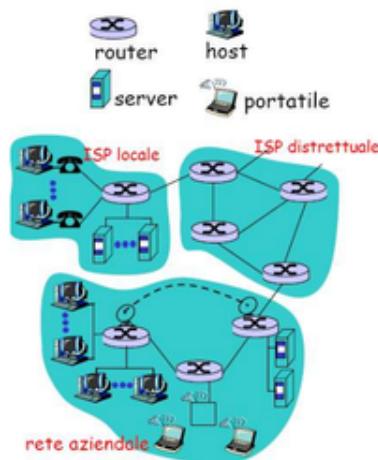


Figure 6: Una porzione di Internet

Uno scorcio delle **entità software** di Internet:

- Applicazioni e processi che elaborano le informazioni.
- **Protocolli** che regolamentano la trasmissione e la ricezione di informazioni e.g. TCP, IP, HTTP, FTP, PPP.
- Interfacce: verranno definite in seguito, sono le “membrane” che separano gli “strati”.

### 3.4.1 Servizi

L'**infrastruttura di comunicazione** consente il funzionamento delle applicazioni distribuite per scambio di informazioni (*e.g. WWW, email, giochi, e-commerce, database, controllo remoto, ecc.*).

Lo **stack protocolare** offre il servizio di connessione. Vi sono due approcci:

1. **Connection-less:** I dati vengono trasferiti **senza** stabilire una connessione, non c'è nessuna garanzia di ordine e consegna. *Ogni pacchetto ha una vita a sé.*
2. **Connection-oriented:** Prevede l'**instaurazione della connessione**, il trasferimento dei dati e, in seguito, la chiusura della connessione. Garantisce integrità, completezza e ordine.

### 3.4.2 IETF/RFC/ICANN

**Definizione 3.4.** L'IETF (Internet Engineering Task Force) è l'organismo che studia e sviluppa i protocolli in uso su Internet. Si basa su gruppi di lavoro a cui chiunque può accedere.

**Definizione 3.5.** RFC/STD (Request For Comments & STanDards) sono i documenti “ufficiali” che descrivono i protocolli usati su Internet. Sono pubblicamente accessibili in rete.

**Definizione 3.6.** ICANN (Internet Corporation for Assigned Names and Numbers) È un ente internazionale che coordina il sistema dei nomi di dominio (DNS), assegna i gruppi di indirizzi di rete, identificativi di protocollo e ha funzioni di controllo (blando) dello sviluppo di Internet.

### 3.4.3 Rete di accesso

Internet è una internetwork che consente a qualsiasi utente di farne parte. L'utente, tuttavia, deve essere fisicamente collegato a un ISP (*internet service provider*).

**Definizione 3.7.** Il collegamento che connette l'utente al primo router di internet è detto **rete di accesso**, sudetto collegamento può essere effettuato tramite rete telefonica, rete wireless o tramite accesso diretto.

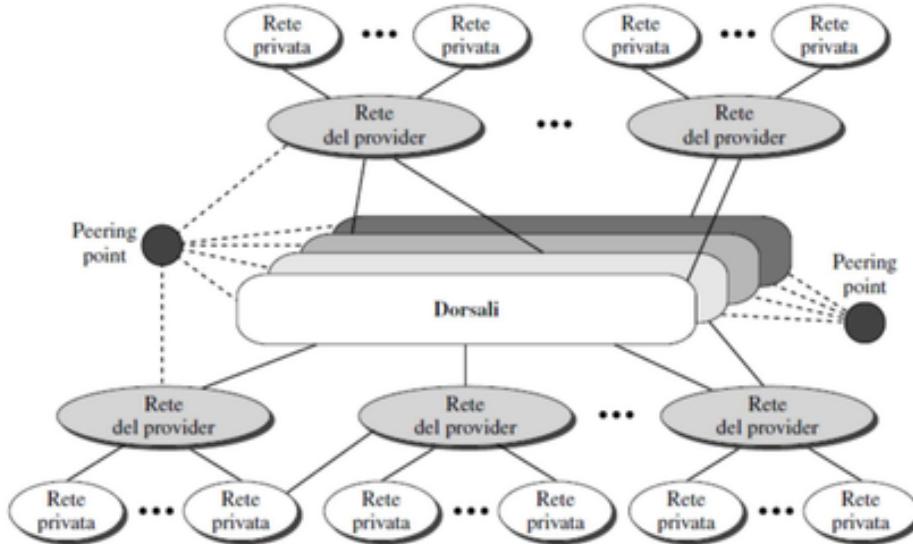


Figure 7: Un modello concettuale di Internet

## 4 Metriche di riferimento

*Come misurare le prestazioni di una rete?*

**Definizione 4.1.** La **larghezza di banda** o **bandwidth** è la larghezza dell'intervallo di frequenze utilizzato dal sistema trasmissivo.

**Definizione 4.2.** Il **bit rate** o **trasmission rate** è la quantità di dati che possono essere trasmessi o ricevuti nell'unità di tempo. [e.g. bps = bit/s]  
Il bitrate dipende dalla tecnica trasmissiva ed è proporzionale alla larghezza di banda.

**Definizione 4.3.** Il **throughput** è la quantità di traffico che arriva realmente a destinazione nell'unità di tempo, al netto di perdite sulla rete, del funzionamento dei protocolli etc.

**Definizione 4.4.** La **latenza** o **latency** è il tempo che passa dal momento in cui il primo bit parte dalla sorgente al momento in cui l'intero messaggio arriva a destinazione.

$$L = r_{\text{propagazione}} + r_{\text{trasmissione}} + r_{\text{accodamento}} + r_{\text{elaborazione}}$$

## 4.1 Ritardi

Il ritardo introdotto da un nodo è la somma di questi 4 ritardi:

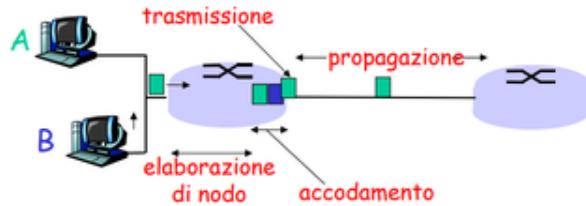


Figure 8: Una visione di contesto

### 4.1.1 Ritardo di elaborazione del nodo

Il ritardo di elaborazione è causato dall’elaborazione del percorso (ovvero dove inoltrare il pacchetto scegliendo il percorso *”migliore”*) e dal controllo di errori sui bit, è tipicamente piccolo e trascurabile.

### 4.1.2 Ritardo di accodamento

Il ritardo di accodamento è il tempo che un pacchetto passa nella coda del router, dipende dall’intensità e dal tipo di traffico. I pacchetti si accodano nei buffer dei router se il tasso di arrivo dei pacchetti eccede la capacità del collegamento di inoltrarli. Se non ci sono spazi liberi i pacchetti in arrivo vengono scartati.

### 4.1.3 Ritardo di trasmissione

Il ritardo di trasmissione è il tempo impiegato a trasmettere un pacchetto intero sul link.

$R$  = rate di trasmissione del collegamento.

$L$  = lunghezza del pacchetto.

$$r_{trasmissione} = \frac{L}{R}$$

### 4.1.4 Ritardo di propagazione

Il ritardo di propagazione è il tempo impiegato da un bit per essere propagato da un nodo (router) all’altro.

$d$  = lunghezza del collegamento fisico

$s$  = velocità di propagazione del collegamento fisico

$$r_{propagazione} = \frac{d}{s}$$

## 4.2 Esempio

Si consideri l'invio di un file di 1 MBit su un datalink di lunghezza 4800km:

$$d = 4800 \times 10^3 m$$

$$s = 3 \times 10^3 m/s$$

Si calcoli il ritardo di propagazione.

Soluzione:

$$r_{propagazione} = \frac{d}{s} = \frac{4800 \times 10^3 m}{3 \times 10^3 m/s} = 0.016 \text{ secondi.}$$

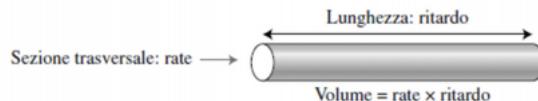
Sia il transmission rate pari a 64 kbps, si calcoli il ritardo di trasmissione.

$$r_{trasmissione} = \frac{L}{R} = \frac{10^6 bit}{64 \times 10^3 bps} = 15.625 \text{ secondi.}$$

Se il transmission rate fosse invece di 1 Gbps?

$$r_{trasmissione} = \frac{L}{R} = \frac{10^6 bit}{10^9 bps} = 0.001 \text{ secondi.}$$

## 4.3 Volume di un link



**Definizione 4.5.** Il volume di un link è il numero massimo di bit che il link può contenere.

$$\boxed{Volume = bitrate \times ritardo}$$

## 4.4 Esempio



Si calcoli il ritardo end-to-end di un pacchetto su un percorso con due router. Sia trascurabile il ritardo di congestione e si suppongano uguali su tutti link il propagation delay, il transmission delay e il processing delay.

**Soluzione:**

Essendoci due router intermedi bisogna attraversare tre link, quindi:

$$r_{totale} = 3 \times r_{propagation} + 3 \times r_{trasmissione} + 3 \times r_{processing}$$

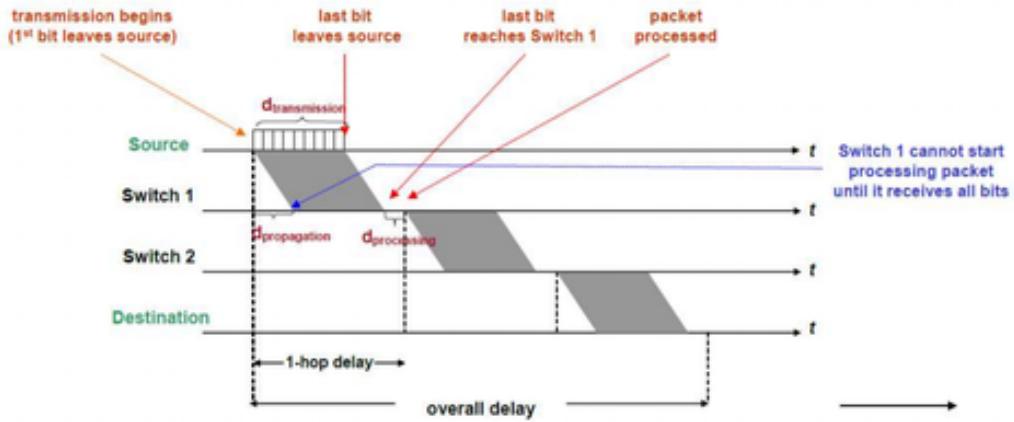


Figure 9: Quello che succede in dettaglio nell'esempio precedente.

## 5 Modelli Stratificati e Protocolli

*Cos'è un protocollo? Cos'è uno strato?*



Figure 10: Esempio di stratificazione. Si nota come i vari strati del modello interagiscono tra di loro, dal basso verso l'alto e viceversa, per consentire al sig. Rossi e al sig. Takamura di scambiarsi lettere.

Vademecum per la lettura del seguente contenuto: N.B.:

1. Gli strati comunicano (*di solito*) **solo** con gli altri strati a loro adiacenti.
2. Uno strato **fornisce** servizi allo strato superiore e **riceve** servizi da quello inferiore.
3. La comunicazione tra strati adiacenti avviene attraverso un'**interfaccia**.
4. Tra due entità diverse comunicano fra di loro solo gli strati dello **stesso livello** e secondo un **protocollo assegnato**, queste due entità sono dette **peer**.

## 5.1 OSI: Open Systems Interconnection

Le prime reti di calcolatori nacquero come **sistemi chiusi** in cui tutti i componenti dovevano essere dello stesso costruttore. Erano quindi tecnologie chiuse e **non interoperabili** l'una con l'altra a causa di drastiche differenze (*e.g. differenza di linguaggio, modelli di stratificazione diversi e impossibilità per i programmi applicativi di riuscire ad operare in ambiente distribuito*). Alla fine degli anni '60 esistevano: ARPANET, SNA (IBM), DNA (Digital).

I **Sistemi Aperti** nascono dall'obiettivo di alcune aziende di realizzare una rete di calcolatori in cui qualsiasi terminale potesse comunicare con un qualsiasi fornitore di servizi mediante qualsiasi rete.

Per realizzare un sistema aperto è necessario stabilire delle regole comuni: **gli standards**.

**Definizione 5.1.** Un sistema che implementa **protocolli aperti** è un **sistema aperto** (open system).

**Definizione 5.2.** Un set di protocolli è **aperto** se:

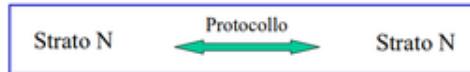
1. I dettagli (**specifiche**) dei protocolli sono disponibili pubblicamente.
2. I cambiamenti al set sono gestiti da un'organizzazione la cui partecipazione è aperta al pubblico

### 5.1.1 Modello ISO/OSI

L'International Organization for Standards (ISO) ha specificato uno standard per l'interconnessione di sistemi aperti: l' **Open System Interconnection Reference Model** (OSI-RM) poi diventato standard internazionale nel 1983 (ISO 7498). **Si basa sul concetto di architettura a strati i cui criteri sono:**

- **Divisione delle funzionalità:** il protocollo di telecomunicazione è diviso in strati o layers, ognuno dei quali svolge un compito piccolo e indipendente dagli altri.  
Si cerca quindi di mantenere un minor numero di strati possibile e di far svolgere a ognuno di essi il minor numero di compiti possibile.
- **Comunicazione mediante interfacce:** i livelli comunicano mediante **chiamate standard**. Ogni livello è tenuto a rispondere alle **sole** chiamate che gli competono e che verranno invocate dal singolo livello o dai due livelli ad esso adiacenti.
- **Information hiding:** le modalità con cui le funzioni competenti ad un livello vengono svolte non è visibile dall'esterno che ne è così svincolato.

## 5.2 Protocollo



I protocolli definiscono il **formato** e l'**ordine** dei messaggi inviati e ricevuti tra entità della rete al livello n-esimo e le **azioni** che vengono fatte per la loro **trasmissione** e **ricezione**.

**Definizione 5.3.** Un **protocollo** è un insieme di regole che permettono a due entità uno scambio **efficace** ed **efficiente** delle informazioni. **Definisce** il **formato** e il **significato** dei frame (campi del messaggio), dei pacchetti o dei messaggi che vengono scambiati tra gli **strati paritari** di due entità diverse.

Un protocollo specifica quindi:

- La **sintassi** di un messaggio (e.g. i campi).
- La **semantica**.
- **Le azioni da compiere** (e.g. per l'invio, alla ricezione, alla trasmissione etc...).

**Definizione 5.4.** Uno **strato** o livello è un modulo interamente definito attraverso i servizi, protocolli e le interfacce che lo caratterizzano.

**Definizione 5.5.** Un' **interfaccia** è il set di regole governanti sintassi e semantica della comunicazione tra due **strati successivi** della stessa **entità**.

**Definizione 5.6.** Un **servizio** è l'insieme di **primitive** (operazioni) che uno strato fornisce ad uno strato soprastante. (*vedi sez. 3.4.1*).

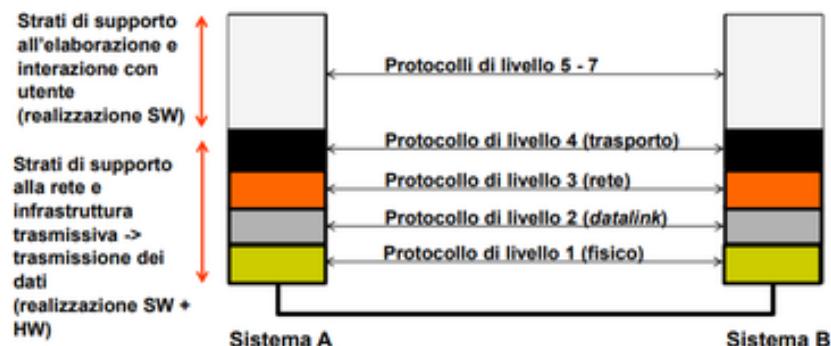


Figure 11: Esempio di stack protocollare.

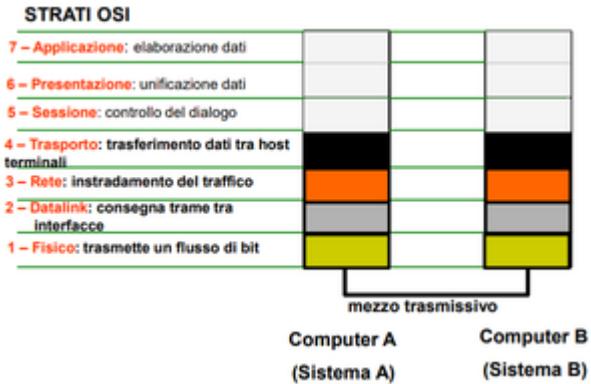


Figure 12: Gli strati di OSI.

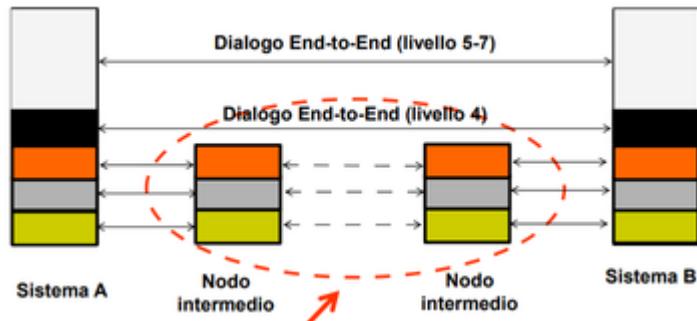


Figure 13: Esempio di collegamento tra end systems.

### 5.3 Incapsulamento dell'informazione

All'interno della rete l'informazione ha origine **al livello applicativo** (*livello 7 in figura*), discende quindi i vari livelli fino alla **trasmissione**, che avviene mediante il **canale fisico**. Da ogni livello attraversato viene aggiunta all'informazione una sezione informativa (o più di una) chiamata **header** che contiene informazioni pertinenti esclusivamente al livello stesso. Per i dati ricevuti invece si segue il cammino inverso. Si tratta infatti di un **processo di incapsulamento reversibile**.



- **Header:** è la qualificazione del pacchetto dati per questo livello.
- **DATA:** è il payload proveniente dal livello superiore.
- **Trailer:** è usato per rilevare e correggere gli errori.

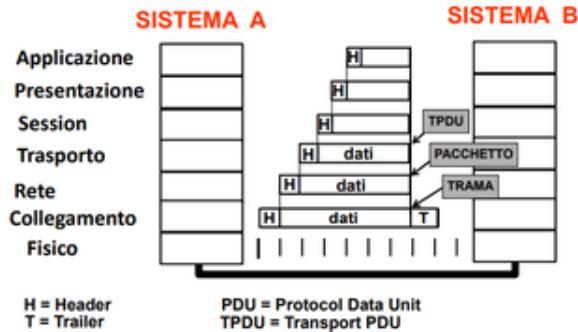


Figure 14: Il processo di incapsulamento. Da notare in particolare il payload.

## 6 Stack protocollare TCP/IP

**TCP/IP** è una **famiglia di protocolli** attualmente in uso su Internet. Si tratta di una **gerarchia di protocolli** costituita da **moduli interagenti**, ciascuno dei quali fornisce funzionalità specifiche. Il termine **gerarchia** significa che ciascun protocollo di **livello superiore** è supportato dai servizi **forniti** dai protocolli di **livello inferiore**. Definita in origine in termini di quattro livelli software soprastanti a un livello hardware, **la pila TCP/IP** è oggi intesa come **composta di cinque livelli**.

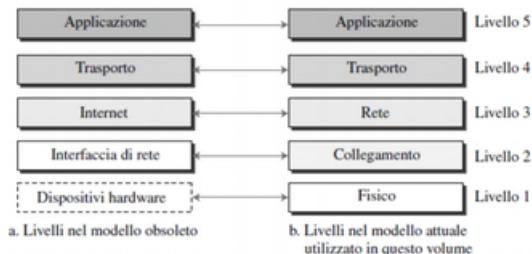


Figure 15: Livelli dello stack protocollare TCP/IP, si notino le differenze tra il modello originario e il modello attuale.

- Il **livello applicativo** supporta le applicazioni di rete.
- Il **livello di trasporto** supporta il trasferimento di dati da un host all'altro.
- Il **livello di rete** instrada i datagrammi dalla sorgente alla destinazione.
- Il **livello link** trasferisce dati tra elementi adiacenti della rete.
- Al **livello fisico** troviamo i bit sul link.

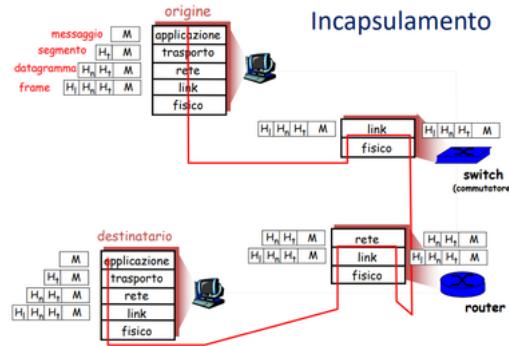


Figure 16: Esemplificazione del processo di incapsulamento/decapsulamento dell'informazione.

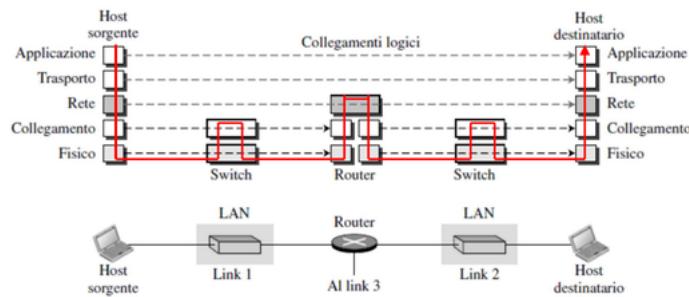


Figure 17: I vari collegamenti logici di comunicazione, in **rosso** invece, il **collegamento fisico**. La modularità del sistema fa in modo che gli strati paritari dei due host abbiano l'**illusione** di comunicare direttamente tra di loro. Si ricorda che queste entità situate a livelli corrispondenti su macchine (host) diverse sono dette **peer**.

## 6.1 Lo strato applicativo

Dello **strato applicativo** fanno parte le **applicazioni di rete**. Le applicazioni di rete sono composte da **processi distribuiti e comunicanti**, ovvero programmi eseguiti dai dispositivi terminali (host o end system) di una rete. Nella comunicazione **a livello applicativo** fra due dispositivi terminali interconnessi, due o più processi sono in esecuzione su ciascuno degli host comunicanti e si scambiano messaggi. Il protocollo dello strato applicativo definisce: (*repetita iuvant*)

- I **tipi** di messaggi scambiati al livello applicativo (e.g. richiesta e risposta).
- La **sintassi** e la **semantica** dei campi dei messaggi.
- Le **regole** di comunicazione **interprocessuale**.

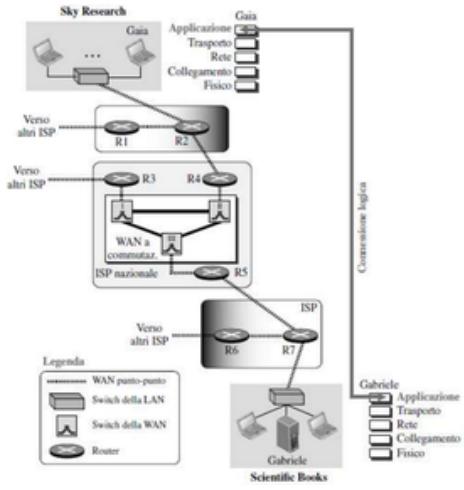


Figure 18: Esempio di comunicazione tra applicazioni di rete. Ancora una volta è bene ricordare che il protocollo crea l'*illusione* che i processi siano in comunicazione diretta.

Presentiamo ora i **paradigmi di comunicazione** dello strato applicativo:

- **Client-Server:** prevede un numero *limitato* di processi **server** che offrono servizi e sono **sempre** in esecuzione in attesa di ricevere richieste dai client. Un **client** è un programma che richiede un servizio. Tipicamente il client inizia il contatto con il server **invia**ndo una **richiesta** e il server risponde **offrendo** il **servizio** richiesto.
- **Peer-to-Peer:** comunicazione tra **peer (pari)** che possono sia **offrire** servizi che **inviare** richieste.
- **Misto.**

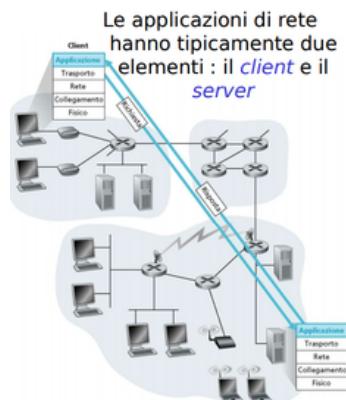


Figure 19: Esempio di paradigma client-server.

Abbiamo, seppur assai brevemente, esaminato la comunicazione a livello applicativo tra macchine diverse. Scendiamo ora più nel dettaglio iniziando a tracciare il percorso **reale** dei dati dallo **strato applicativo** allo **strato di trasporto**. Partiamo dalla seguente definizione:

**Definizione 6.1. API:** acronimo di Application Programming Interface, è un insieme di **procedure** e **regole** che un programmatore deve seguire per accedere a delle risorse. Facilita molto la programmazione del software client-side.

L' API che funge da **interfaccia** tra gli **strati di applicazione e di trasporto** è chiamata **socket** ed è usata dai processi dello strato applicativo per inviare e ricevere dati dallo strato di trasporto. Si tratta, ancora una volta, di una connessione **logica** poichè in realtà l'invio e la ricezione dei dati sono, nel **concreto**, responsabilità del sistema operativo e del protocollo TCP/IP.

Riportiamo un estratto dell'**API** di **TCP**:

```
connection TCPopen(IPaddress, int) //to open a conn.  
void TCPSend(connection, data) //to send data  
data TCPReceive(connection) //to receive data  
void TCPclose(connection) //to close a conn.
```

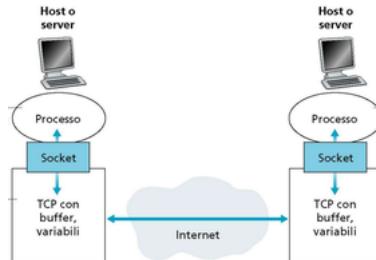


Figure 20: In figura due processi che comunicano tramite socket. Il processo è controllato dallo sviluppatore dell'applicazione, tutto ciò che è presente sotto la socket è controllato dal sistema operativo.

Sorge ora spontanea una domanda: **se ci fossero più processi su ogni host?**

Servirebbe un modo per **identificarli...** ci viene in aiuto il **socket address**.



Figure 21: Il socket address identifica sia il **processo** con il numero di porta che l'**host** con l'indirizzo IP, a ogni porta corrisponde un processo.

**Riassumendo:** dello strato applicativo fanno parte le **applicazioni di rete**, composte da **processi distribuiti** su macchine diverse che comunicano tra di loro mediante il protocollo proprio dello strato applicativo. Sebbene il protocollo fornisca alle applicazioni l'illusione di comunicare direttamente, la comunicazione è in realtà garantita da tutto lo stack protocolare sottostante. In particolare, lo strato applicativo dipende dai servizi offerti dallo **strato di trasporto** che gli è immediatamente sottostante. I due strati comunicano tramite una **API** che si chiama **socket**.

A seconda del **servizio di trasporto richiesto** dall'applicazione è possibile che si trovino in uso **protocolli di trasporto diversi** tra i quali: **TCP e UDP**. (*vedi sez. 2.2*)

Applicazione	Tolleranza alla perdita di dati	Throughput	Sensibilità al tempo
Trasferimento file	No	Variabile	No
Posta elettronica	No	Variabile	No
Documenti Web	No	Variabile	No
Audio/video in tempo reale	Si	Audio: da 5 Kbps a 1 Mbps Video: da 10 Kbps a 5 Mbps	Si, centinaia di ms
Audio/video memorizzati	Si	Come sopra	Si, pochi secondi
Giochi interattivi	Si	Fino a pochi Kbps	Si, centinaia di ms
Messaggistica istantanea	No	Variabile	Si e no

Figure 22: In figura le caratteristiche di alcune delle principali applicazioni di rete. Non tutte le applicazioni sono uguali. Tipicamente la telefonia di internet usa il protocollo UDP.

### 6.1.1 Web

Il **web** è formato da risorse indirizzate da **URL**, acronimo di **uniform resource locator**. Generalmente queste risorse sono **pagine web**, formate da altri oggetti referenziati (*e.g. altre pagine web, immagini ecc...*). Lo **user agent** o **client** per il web è chiamato **browser** ed il **server** è chiamato **web server**.

**Definizione 6.2.** Una **URI** o Uniform Resource Identifier è una stringa compatta di caratteri che identifica una risorsa fisica o astratta.

Le **URL** sono un **sottoinsieme** delle URI e identificano le risorse tramite la loro posizione all'interno della rete. Le **URN** acronimo di **uniform resource name** sono un altro sottoinsieme delle URI la cui funzione è di rimanere **globalmente uniche e persistenti** anche quando le risorse da loro puntate cessano di esistere o non sono più disponibili. La **sintassi** delle URI è organizzata **gerarchicamente** e i componenti sono disposti in ordine di importanza da sinistra verso destra.

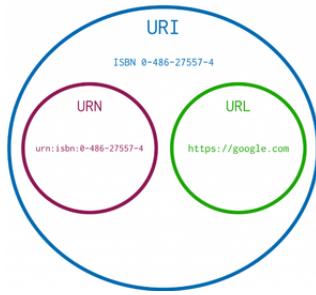


Figure 23: URN e URL non sono altro che specializzazioni delle URI.

Le **sintassi** della URI:

$< \text{scheme} > : // < \text{authority} > < \text{path} > ? < \text{query} >$

`http://maps.google.it/maps/place?q=lar...+p...+p...&hl=it`

- **scheme**: è **obbligatorio**, definisce lo **spazio dei nomi** della risorsa.
- **authority**: indica il **nome di dominio** di un host (*reg\_name*) o il suo **indirizzo IP** in notazione decimale puntata. Tipicamente identifica un computer sulla rete.
- **path**: contiene **dati** specifici per l'authority (o per lo scheme) e **identifica** la risorsa **nel contesto** di quel namespace e di quell'authority.

Le URI possono essere **absolute** o **relative**, una **URI assoluta** identifica una risorsa **indipendentemente** dal contesto in cui è usata. Una **URI relativa** identifica una risorsa in relazione ad un'altra URL, è **priva di schema e di authority**, non viaggia sulla rete ed è interpretata dal browser in relazione al documento di partenza.

(e.g. `http://www.w3.org/pub/WWW/TheProject.html` oppure  
`/pub/WWW/TheProject.html sotto l'host: www.w3.org`).

### 6.1.2 HTTP Protocol

Il protocollo **HTTP** è un protocollo di tipo **request/response**: il client inizia la connessione inviando un messaggio di **request** al server che a sua volta invia una **response**. Si tratta di un **protocollo generico**, poiché non dipende dal formato delle risorse, e **stateless** poiché le **coppie richiesta/risposta sono indipendenti l'una dall'altra**. In **HTTP 1.0** dopo la prima coppia di richiesta/risposta la connessione viene **terminata** mentre in **HTTP 1.1 si procede** con un'altra coppia.

Lo **schema http** è usato per accedere alle risorse attraverso il protocollo HTTP.

Si riporta di seguito la **sintassi** per le URL http:

*http : // < host > [:< port >][< path >]*

- **host:** è un **host domain di Internet** valido oppure un indirizzo IP in forma decimale puntata.
- **port:** è un **intero**, se la porta è vuota o non è indicata si usa automaticamente la porta 80.
- **path:** specifica la **request URI** (*vedi seguito*).

**N.B.** Il protocollo HTTP utilizza il protocollo TCP tramite la sua API.

```
//esempio client
c = TCPopen("131.115.7.24", 80);
TCPsend(c,"GET /index.html");
d = TCPreceive(c);

//esempio server
p = TCPbind(80); //where to wait for connections
d = TCPaccept(p); //waiting for connections
r = TCPreceive(d);
...
TCPsend(d,pag);
TCPclose(d);
```

**Definizione 6.3.** Una **connessione http** è un circuito logico di livello di trasporto stabilito tra due programmi applicativi per comunicare tra loro.

Una **connessione** può essere:

- **Non persistente**(*http1.0: RFC 1945*): si parla di connessione non persistente quando **per ogni richiesta** del **client** viene instaurata **una nuova connessione** con il **server**. Ciò aumenta il **carico** su quest ultimo e potrebbero verificarsi fenomeni di **congestione**. *Infatti per visualizzare le n immagini di un sito il client invia di seguito n richieste al server.*
- **Persistente**(*http1.1: RFC 2616*): la connessione è appunto **persistente**. Nello **standard** è specificato un meccanismo che consente al server di **chiudere** la connessione TCP su richiesta del client. (*CONNECTION = CLOSE, in GENERAL HEADER, vedi seguito.*)  
**N.B.** Una volta che la chiusura della connessione è stata segnalata, il client **non deve** inviare altre richieste.

**Esempio:**

Supponiamo che l'utente digitи la seguente URL:

**www.someSchool.edu/someDepartment/home.index**

Con **una connessione non persistente**, in ordine temporale succedono le seguenti cose:

1. Il **client http** invia una richiesta di connessione **TCP** verso il server http al **www.someSchool.edu** (*La porta 80 è usata di default per il server http.*)
2. Il **server http** dell'host **www.someSchool.edu**, che aspetta le richieste di connessione **TCP** alla **porta 80**, **accetta la richiesta di connessione** e notifica il client.
3. Il **client http** invia quindi un **messaggio di richiesta**, contenente la URL.
4. Il **server http** riceve il messaggio di richiesta, compila un **messaggio di risposta** con l'oggetto richiesto indicato dalla URL: **someDepartment/home.index**, invia il messaggio e in seguito **chiude la connessione**.
5. Il **client http** riceve il messaggio di risposta che contiene il file html e lo visualizza.

*Si ricorda che la ricezione e la trasmissione di tutti i messaggi elencati poco sopra avviene tramite socket.*

Supponiamo ora che la URL contenga dei riferimenti a 10 immagini, in tal caso per ogni riferimento si devono ripetere tutti i passaggi definiti sopra. Salta immediatamente all'occhio la **scarsa efficienza** della procedura.

Con **una connessione persistente** invece il server lascerebbe aperta la connessione TCP dopo aver spedito la prima risposta e vi riceverebbe quindi le richieste successive. La connessione verrebbe chiusa dal server quando specificato nell'header di un messaggio **inviatogli dal client**, oppure alla mancata ricezione di richieste per un certo intervallo di tempo (*time out*).

Un ulteriore miglioramento delle prestazioni, si otterrebbe con una tecnica di **pipelining**, consistente nell'invio da parte del client di molteplici richieste senza aspettarne la ricezione da parte del server.

Il **server deve** tuttavia inviare le risposte nello stesso ordine in cui sono state ricevute le richieste e il **client non** può inviare **in pipeline** richieste che usano metodi HTTP **non idempotenti**. Ma cos'è un **metodo idempotente?**

**Definizione 6.4.** Un **metodo idempotente** è un metodo tale che il suo effetto collaterale sulla risorsa è lo **stesso** per N o 1 richieste identiche che ne fanno uso. (e.g. *GET, HEAD, PUT, DELETE, OPTIONS, TRACE*)

**Definizione 6.5.** Un **metodo safe** è un metodo che non produce effetti collaterali sulle risorse. (e.g. *non le modificano: GET, HEAD, OPTIONS, TRACE...*)

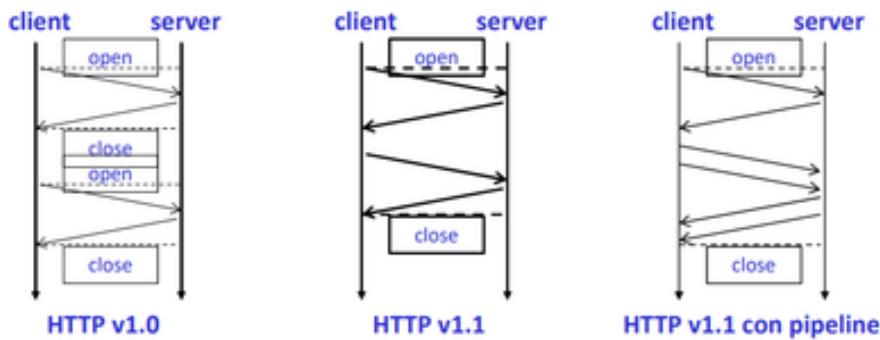


Figure 24: In figura sono rappresentate le differenze viste prima.

*Come sono strutturati i messaggi HTTP?*

Riportiamo di seguito la **struttura** di un generico messaggio http:

Request = Request-Line o Response = Status-Line per i messaggi di risposta.

```
*(` general-header
| request-header o response-header
| entity-header )
CRLF
[ message-body ]
```

La prima riga o **start line** distingue i messaggi di request dai messaggi di response.

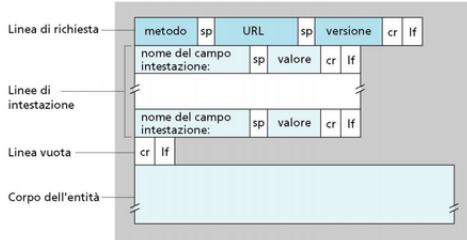


Figure 25: In figura un messaggio di richiesta.

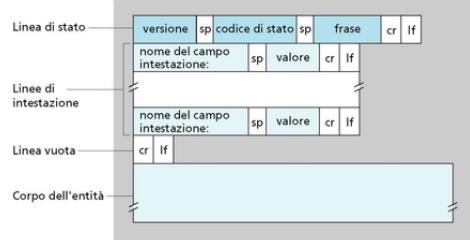


Figure 26: In figura un messaggio di risposta.

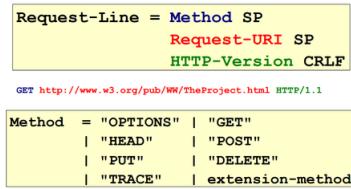


Figure 27: La struttura della **request line**. Il campo **method** è case sensitive ed indica l'operazione da eseguire sulla risorsa identificata dalla URI. Il metodo **POST** serve per inviare dal client al server le informazioni inserite nel body del messaggio, **PUT** è usato dal client per chiedere al server di creare o modificare una risorsa, **DELETE** per cancellarla.

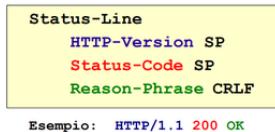


Figure 28: La **status-Line** è la prima riga del messaggio di risposta. Lo **status code** è un numero di 3 cifre, indica il risultato del tentativo di soddisfare la richiesta del client. La **reason-phrase** dà una breve descrizione testuale dello status-code. Lo status-code è rivolto alla macchina mentre la reason-phrase all'utente umano.

Gli **header** sono insiemi di coppie **nome : valore** che specificano alcuni parametri del messaggio trasmesso o ricevuto.

- **General Headers:** sono relativi alla **trasmissione** e si applicano a **tutto il messaggio**. (*e.g. Date, Connection: usato dal mittente per specificare delle opzioni desiderate per la connessione, ad esempio close. Transfer-encoding: specifica se e quali trasformazioni sono state applicate al corpo del messaggio ad esempio gzip, chunked ecc. Cache control: indica quale tipologia di cache può memorizzare il messaggio, può essere public, private o no-cache.*)
- **Entity Headers:** sono **metadati** relativi all'**entità trasmessa**. Ogni entity è costituita da un **entity body** e da una serie di **entity headers** che ne definiscono contenuto e proprietà. (*e.g. Content Length*)
- **Request Headers:** sono relativi alla **richiesta**. Supportano il **content negotiation**, il processo tramite cui il server sceglie la forma "giusta" del contenuto richiesto dal client. (*Specificano da chi è fatta la richiesta, a chi viene fatta, che tipo di risposte il client è in grado di accettare, l'autorizzazione, ecc.*)
- **Response Headers:** sono relativi al **messaggio di risposta**.

```

general-header = Cache-Control
               | Connection
               | Date
               | Pragma
               | Transfer-Encoding
               | Upgrade
               | Via

```

Figure 29: L'elenco dei general headers.

```

request-header = Accept           | Accept-Charset
                | Accept-Encoding | Accept-Language
                | Authorization
                | Proxy-Authorization
                | From            | Host
                | If-Modified-Since
                | If-Unmodified-Since
                | If-Match          | If-None-Match
                | If-Range
                | Max-Forwards     | Range
                | Referer          | User-Agent

```

Figure 30: L'elenco dei request headers. **Accept** specifica quali tipi di media sono accettati in risposta dal client, **Accept-Charset**, quali caratteri e **Accept-Encoding** quali formati.

"100"	- Continue	"101"	- Switching Protocols
"200"	- OK	"201"	- Created
"202"	- Accepted	"202"	- Non-Authoritative Information
"204"	- No Content	"205"	- Reset Content
"206"	- Partial Content		
"300"	- Multiple Choices	"301"	- Moved Permanently
"302"	- Moved Temporarily	"302"	- See Other
"304"	- Not Modified	"305"	- Use Proxy
"400"	- bad Request	"401"	- Unauthorized
"401"	- Authentication Required	"402"	- Payment Required
"404"	- Not Found	"405"	- Method Not Allowed
"406"	- Not Acceptable	"407"	- Proxy Authentication Required
"408"	- Request Time-out	"408"	- Conflict
"410"	- Gone	"411"	- Length Required
"412"	- Precondition Failed	"412"	- Request Entity Too Large
"414"	- Request-URI Too Large	"415"	- Unsupported Media Type
"500"	- Internal Server Error	"501"	- Not Implemented
"502"	- Bad Gateway	"503"	- Service Unavailable
"504"	- Gateway Time-out	"505"	- HTTP Version Not Supported

Figure 31: In figura l'elenco degli status-codes e il loro significato.

```

response-header = Age
                  | Location
                  | Proxy-Authenticate
                  | Public
                  | Retry-After
                  | Server
                  | Vary
                  | Warning
                  | WWW-Authenticate

```

Figure 32: I campi del **response-header** consentono al server di inviare informazioni aggiuntive che non possono essere poste nella status-line. Ad esempio: **Age**: indica quanto tempo è trascorso, in secondi, tra l'invio della risposta da parte del mittente e la generazione della stessa all'origine. **Location** è usato per reindirizzare il client verso un'altra URI per completare la sua richiesta o per fornire una nuova risorsa. Il campo **Server** contiene informazioni sul software usato dal server di origine per portare a compimento la richiesta del client.

Fin'ora abbiamo mostrato come il client accede alle risorse del web inviando richieste al server. Cosa succederebbe se, ad esempio, un client richiedesse continuamente la **stessa** risorsa? Potremmo evitare di inoltrare richieste tutte uguali al server?

Sì, ci viene in aiuto il **web caching**. L'obbiettivo del web caching è di soddisfare le richieste del client **senza** contattare il server. Funziona memorizzando **copie temporanee** di risorse web, servendole poi al client così da ridurre l'uso di **banda**, limitare il **workload** sul server e di conseguenza diminuire **tempo di risposta** verso gli altri clients.

Consideriamo due modelli possibili di web caching:

- **User Agent Cache:** lo **user agent** (il browser) mantiene una **copia** delle risorse visitate dall'utente.
- **Proxy Cache:** il **proxy intercetta il traffico** e mette in cache le risposte. Successive richieste alla stessa URI possono essere servite direttamente dal proxy senza inoltrare la richiesta al server. Sta poi all'utente configurare il browser per consentire gli accessi Web via proxy.

*Cos'è un proxy?*

**Definizione 6.6.** Un **proxy** è un **programma intermediario** che agisce sia da **server** che da **client**, **inviando e servendo richieste** per altri clients. Le richieste sono gestite internamente oppure sono girate a server terzi.

Resta ora da chiarire come è possibile che, sebbene abbiamo definito il protocollo HTTP come **stateless**, alcuni siti, *riconoscano* gli utenti.

Sebbene infatti il protocollo HTTP sia a tutti gli effetti stateless, le applicazioni web **non** lo sono. Come fare quindi a conciliare queste due realtà?

*Con i cookies!*

**Definizione 6.7.** I **cookies** sono stringhe di testo contenenti informazioni relative all'utente.

I **cookies** funzionano nel seguente modo:

1. Un client invia al server una richiesta HTTP.
2. Il server invia al client la risposta HTTP e in più una linea **set-cookie: 1678453. (esempio fittizio)**
3. Il client **memorizza** il cookie in un file e lo associa al server. Lo aggiungerà con la seguente linea: **cookie: 1678453** a tutte le sue successive richieste.
4. Alla successiva richiesta da parte del client, il server **risalirà** tramite il cookie alle informazioni ad esso **associate**.

### 6.1.3 TELNET

**TELNET**, acronimo per **T**ERminaL **N**ETwork, è un **protocollo client-server** che fornisce una comunicazione **interattiva ed orientata al testo** tra due macchine, è basato sul trasporto **connection-oriented** ed usa il protocollo **TCP**. Consente all'utente di effettuare una sessione di **login** in una macchina remota e quindi di utilizzarne il terminale. L'utente che si autentica tramite login remoto ha accesso infatti a tutti i **comandi** e ai **programmi** disponibili su di essa. **I comandi vengono eseguiti come se l'utente li digitasse dalla tastiera stessa della macchina.** Per estensione, **telnet** è anche il nome di un programma usato per avviare una sessione **TELNET** verso un host remoto. **TELNET** infatti include due programmi: un programma **TELNET client** e un programma **TELNET server**. **TELNET client** (**telnet**) interagisce con l'utente sulla macchina locale e scambia messaggi con **TELNET server**. Riportiamo di seguito una semplificazione del funzionamento:

1. L'utente, dalla **propria** macchina **locale** (**TELNET client**), stabilisce una connessione **TCP**, persistente per tutta la durata della sessione, con una **macchina remota** (**TELNET server**) alla **porta 23** e vi si **autentifica**.
2. Tutte le **battute dei tasti** della macchina locale vengono **trasmesse** dal client alla **macchina remota**.
3. La **macchina remota** accetta la connessione **TCP** e il **TELNET server** trasmette i dati al sistema operativo locale.
4. L'**output** della **macchina remota** viene quindi **ricevuto** e **trasmesso** sul terminale dell'utente.

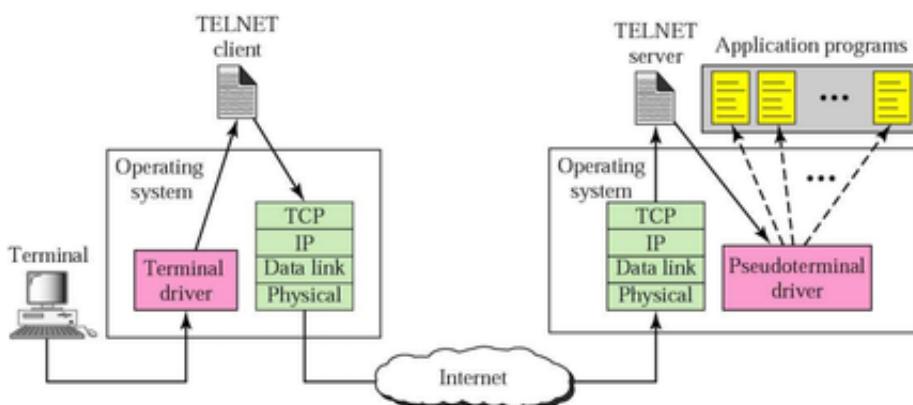


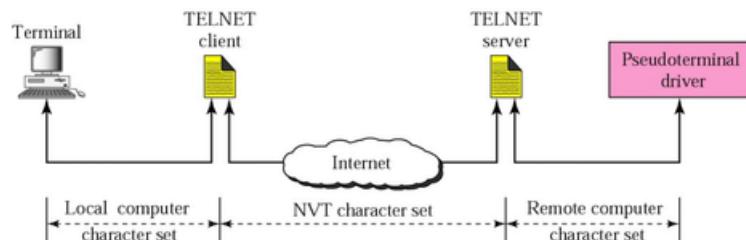
Figure 33: In figura il funzionamento di TELNET.

N.B. TELNET è un protocollo STATEFUL.

*Problema: come operare con sistemi operativi diversi?*

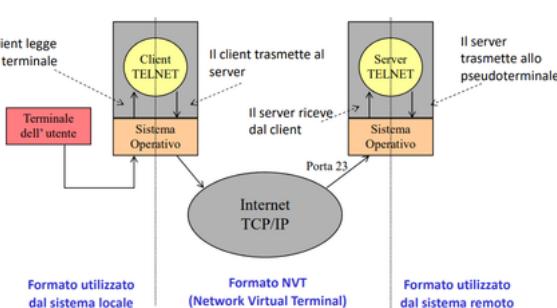
I terminali infatti non sono tutti uguali, possono differire gli uni dagli altri per il **set** e la **codifica** dei caratteri, per la **lunghezza** della **linea** e della **pagina** e per i **tasti funzione** individuati da diverse sequenze di caratteri.

Una **soluzione** è stata trovata mediante la definizione di un **Network Virtual Terminal (NVT)** che **definisce un set di caratteri universali**. L'NVT è quindi un dispositivo "immaginario" che fornisce una rappresentazione astratta di un terminale. Gli host, sia client che server, **traducono** le loro caratteristiche **locali tramite il set universale** così da apparire **in rete** come un NVT e assumono che l'host remoto sia anch'esso un NVT. Questa operazione di traduzione è ovviamente **reversibile**.



Il funzionamento di TELNET, con l'aggiunta delle trasformazioni intermedie sopra descritte diventa:

1. ...
2. Tutte le **battute dei tasti** della macchina locale vengono **trasformate in NVT** e successivamente **trasmesse dal TELNET client alla macchina remota**.
3. La **macchina remota** accetta la connessione TCP, il **TELNET server traduce** da NVT allo standard del sistema operativo remoto e infine gli trasmette i dati ricevuti.
4. ...



*N.B. NVT invia i caratteri di controllo prioritariamente con TCP URGENT.*

Concludiamo con qualche tecnicismo:

I terminali NVT si scambiano dati in formato **7-bit US-ASCII** e adottano inoltre un approccio **in-band signaling**, ovvero dati e comandi viaggiano sullo **stesso canale**. Per distinguere i due tipi di informazione si usa la seguente convenzione: ogni carattere è inviato come un **ottetto di bit** con **il primo bit settato a zero**. I caratteri, in notazione decimale, vanno dal numero 0 al 127. I comandi invece sono identificati tramite ottetti speciali di 1, in n. d. vanno dal numero 240 al numero 254, per distinguerli dai dati sono sempre preceduti da un carattere speciale: **IAC o Interpret As Command** identificato, sempre in n. d., dal numero 255. Essenzialmente quindi si usa un canale di 8 bit per scambiare dati di tipo 7 bit ASCII. I messaggi scambiati durante la fase iniziale della comunicazione, ovvero prima del login, sono **messaggi di controllo** e costituiscono la **Telnet Option Negotiation**, in sostanza sono usati per scambiare **informazioni sulle caratteristiche degli host**.

Comando	Codifica decimale	Significato
IAC	255	Interpret as command
EL	248	Erase line
EC	247	Erase character
IP	244	Interrupt process
EOR	239	End of record

*Ma quindi NVT conviene?*

Rispondiamo alla domanda mostrando dei semplici calcoli:

Supponiamo che  $N$  sia il numero di sistemi distinti che si vogliono far inter-operare:

- **Senza** l'uso di NVT, si necessita la scrittura di  **$N-1$**  client per ogni sistema ( $N-1$  TELNET-clients che traducano negli  $N-1$  standards dei sistemi), e 1 TELNET-server per ogni sistema:  
In totale si avranno  $N \times (N - 1) + N$  applicativi.
- **Usando** NVT, bisogna scrivere 1 TELNET-server per ogni sistema e  $N$  TELNET-client (1 per ogni sistema che traduca dallo standard di sistema allo standard di NVT):  
Avremo quindi  $N + N = 2N$  applicativi.

Quindi per  $N > 2$  **conviene** usare NVT!

#### 6.1.4 SSH

TELNET non possiede **alcuna** misura di sicurezza poichè è stato progettato per l'uso su **reti private**, trasmette tutto in chiaro, anche le **password!** Con l'avvento delle reti pubbliche però si è reso necessario prendere delle contromisure.

**SSH** o Secure SHell è un'applicazione nata per sostituire TELNET.

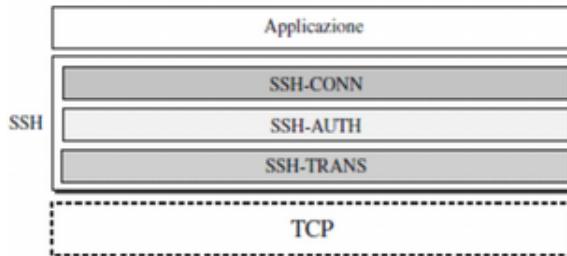


Figure 34: Le componenti di SSH

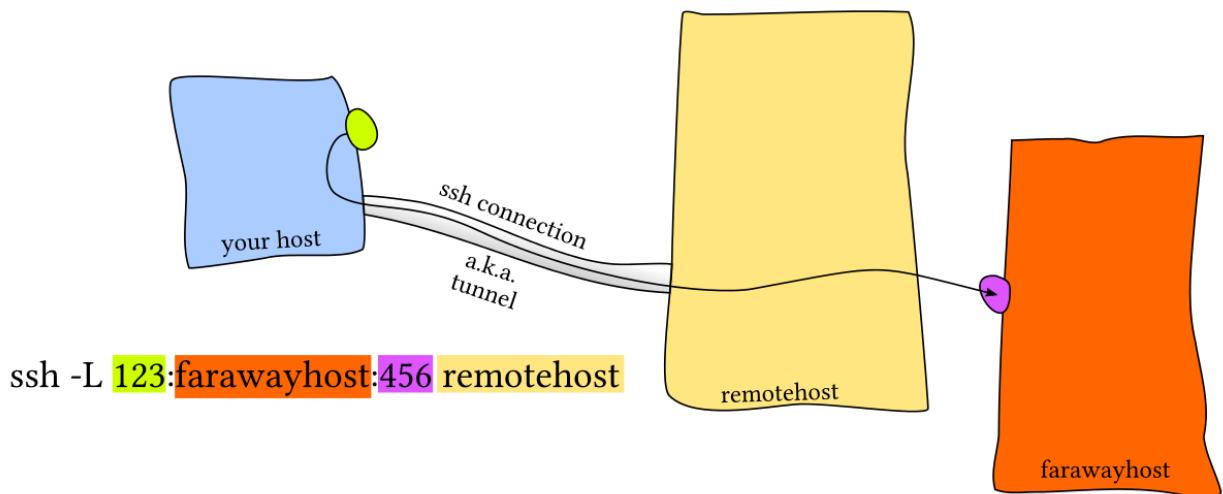
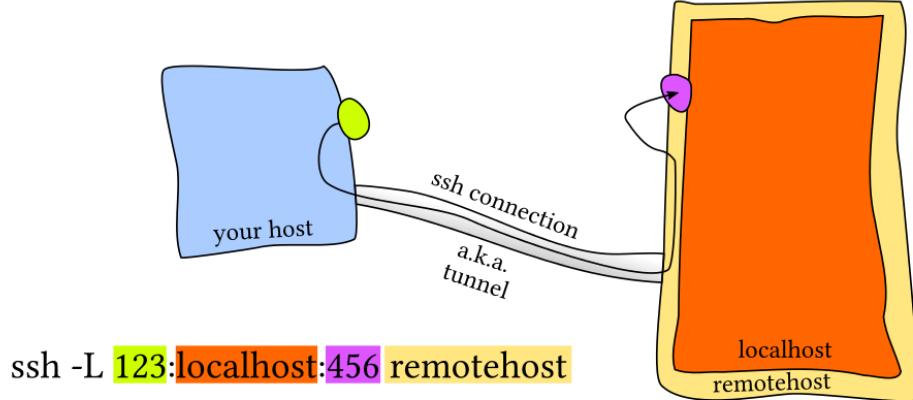
Il **protocollo** di livello applicazione **SSH** è composto da **tre** diverse componenti: **SSH-TRANS**, **SSH-AUTH** e **SSH-CONN**. La componente **SSH-TRANS** o **SSH-Transport Layer Protocol** costruisce un canale di comunicazione sicuro, tramite tecniche crittografiche, sfruttando la connessione offerta da TCP. Il **protocollo TCP** trasmette tutte le informazioni in chiaro, **non** è quindi in grado **da solo** di garantire **privacy** e **confidenza**. *SSH-TRANS è anche in grado di riconoscere se il server a cui ci stiamo connettendo è quello autentico o meno.* La componente **SSH-AUTH** si occupa di autenticare il client. *Sono disponibili altre tecniche di autenticazione oltre alla classica con username e password tra le quali l'accesso basato su coppie di chiavi crittografiche.* Infine la componente **SSH-CONN** sfruttando i servizi offerti dalle altre due componenti offre i servizi di **terminale, trasferimento file, creazione di tunnel** ecc...  
**SSH** offre quindi molti più servizi di TELNET.

*Cos'è il TCP Port Forwarding?*

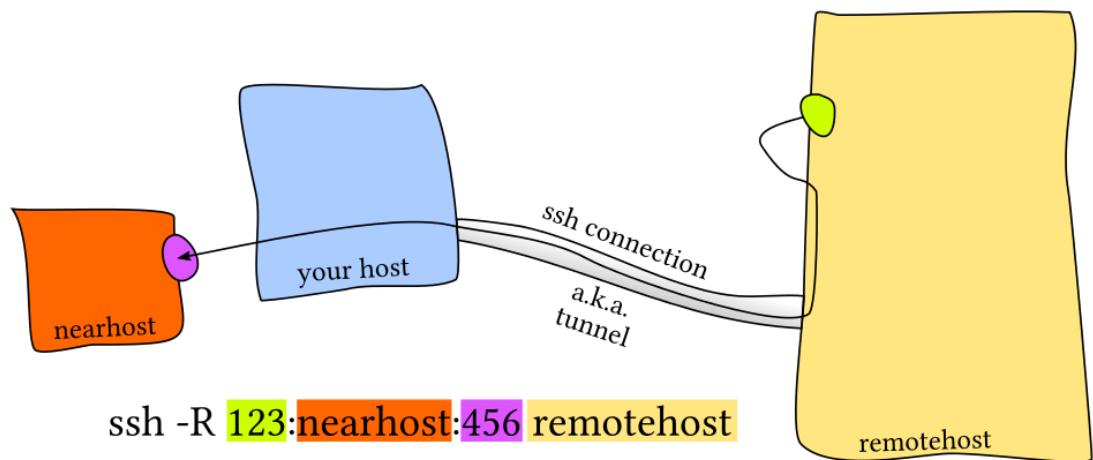
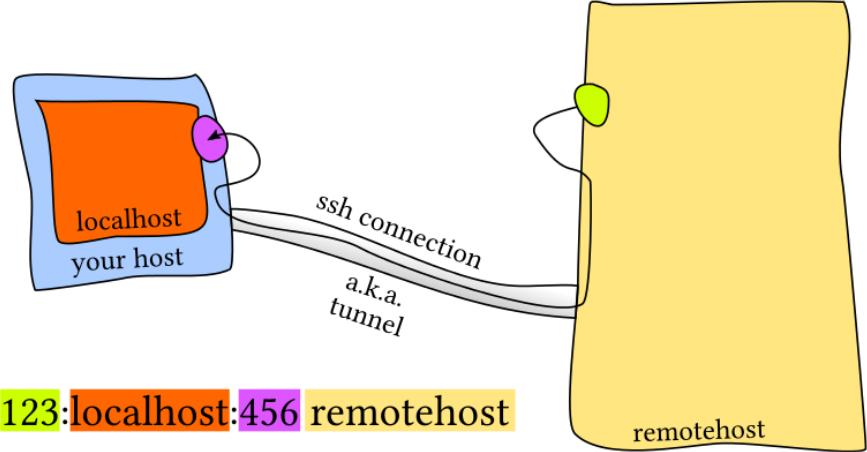
Il **TCP Port Forwarding** è un meccanismo che permette di **creare** un **canale di comunicazione sicuro** attraverso il quale **veicolare** qualsiasi tipo di **connessione TCP**. Opera creando un canale di comunicazione **cifrato** tra la **porta** all'**indirizzo remoto** a cui ci si vuole collegare e una **porta locale** libera. Le applicazioni punteranno il collegamento alla porta locale e la connessione verrà inoltrata automaticamente all'host remoto tramite un canale sicuro.

Segue una spiegazione delle differenze tra **local port forwarding** e **remote port forwarding**.

- **Local:** `ssh -L` specifica che il traffico sulla porta indicata della macchina locale deve essere reindirizzato verso la porta indicata della macchina remota.  
e.g. `ssh -L sourcePort:forwardToHost:onPort connectToHost` significa: connettiti via ssh a **connectToHost**, e inoltra tutti i tentativi di connessione che arrivano alla porta locale **sourcePort** verso la porta **onPort** della macchina chiamata **forwardToHost**, che si raggiunge tramite la macchina chiamata a sua volta **connectToHost**.



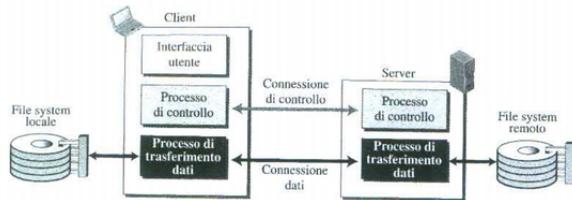
- **Remote:** `ssh -R` specifica che il traffico sulla porta indicata della macchina remota deve essere reindirizzato verso la porta indicata della macchina locale.  
e.g. `ssh -R sourcePort:forwardToHost:onPort connectToHost` significa: connettiti via ssh a **connectToHost**, e inoltra tutti i tentativi di connessione che arrivano alla porta remota **sourcePort** verso la porta **onPort** della macchina **forwardToHost**, che si raggiunge tramite la tua macchina locale.



N.B. SSH è un protocollo STATEFUL.

### 6.1.5 FTP

FTP acronimo per **File Transfer Protocol**, è un **protocollo** per il **trasferimento** di **dati** tra due host di una rete, è lo **standard** per il **trasferimento** di file **offerto da TCP/IP**. Adotta il modello **client-server**: il **client** richiede il trasferimento di un file che può consistere sia nell'**acquisizione** di una copia locale modificabile sia nell'eventuale **trasferimento** della copia modificata sull'host remoto (**server**).



Il **client** ha **tre** componenti: **interfaccia utente**, **processo di controllo** e **processo di trasferimento dati** mentre il **server** remoto ne ha solo **due**: **processo di controllo** e **processo di trasferimento dati**. La **separazione** del trasferimento dei dati da quello dei comandi rende il protocollo **FTP efficiente**. Infatti la **connessione di controllo** usa **regole semplici** così da ridurre lo scambio delle informazioni a **una riga di comando** e **una di risposta per ogni interazione**. Mentre la **connessione dati** usa **regole più complicate** a causa della varietà delle informazioni che vi transitano. **FTP offre funzionalità aggiuntive** oltre al semplice trasferimento di dati, infatti **mette a disposizione**:

- **Accesso interattivo:** l'utente può **navigare, cambiare e modificare** l'albero di directory nel file system dell'host remoto.
- **Specificazione del formato dei dati da trasferire** (e.g. file di testo o file binari)
- **Autenticazione:** il client può **autenticarsi** con username e password.

Poco sopra abbiamo accennato al fatto che **FTP** prevede l'instaurazione di **due tipi di connessione tra il client e il server** ovvero:

1. **Control connection:** prevede uno scambio di comandi e risposte tra client e server. Segue il protocollo TELNET e **rimane aperta per l'intera durata della sessione interattiva**. Si usa la **porta 21** del **server** e la codifica standard **NVT ASCII**.
2. **Data connection:** prevede il trasferimento di dati mediante procedure e la **specifica dei tipi**. I dati trasferiti possono essere parte di un file, un file o un set di file. **Viene aperta e chiusa per ogni singolo scambio**. Per lo scambio di dati il **server** usa la **porta 20**.

La **Data Connection** non segue il protocollo **TELNET** e la sua apertura avviene secondo uno schema completamente diverso, ovvero:

1. Il **client**, non il server, effettua un'**apertura passiva** usando una **porta effimera** e resta in attesa di connessione, viene fatto dal client poichè è tale processo che invierà i comandi, tramite la connessione di controllo, per il trasferimento dei file.
2. Il client **invia** questo **numero di porta** al **server** per mezzo del comando **PORT**.
3. Il **server**, ricevuto il numero di porta, **effettua un'apertura attiva**, ovvero apre la connessione, usando la propria **porta** nota **20** e quella effimera offerta dal client.

Per effettuare il trasferimento dei file inoltre, il client deve **definire il tipo** di file, la **struttura dati** e la **modalità di trasmissione** al fine di risolvere i problemi di eterogeneità tra client e server, va infatti ricordato che programma client e programma server sono su macchine **diverse**. Questo scambio di informazioni avviene mediante la **connessione di controllo**.

FTP è quindi un protocollo **STATEFUL** poichè il **server deve tener traccia dello stato dell'utente**: bisogna tenere conto infatti, tra le altre cose, della directory del file system remoto in cui si trova l'utente!

Concludiamo menzionando il fatto che esistono server che supportano connessioni FTP **senza autenticazione** (**Anonymous FTP**). Tipicamente consentono di accedere **solo** ad una parte del file system e permettono **solo** un subset di operazioni (e.g. la **PUT** **non** è permessa). Di solito **si usa** un **username comune** (solitamente "ftp" or "anonymous") e una **password qualsiasi** (e.g. l'indirizzo email dell'utente). *Era usate per distribuire a un pubblico dei file senza dover generare numerosi username e password.*

Comandi di controllo	Significato
USER	username
PASS	password
LIST	elenca i file della directory corrente
NLST	richiede elenco file e directory (ls)
RETR filename	recupera (get) un file dalla directory corrente
STOR filename	memorizza (put) un file nell'host remoto
ABOR	interrompe l'ultimo comando ed i trasferimenti in corso
PORT	indirizzo e numero di porta del client
SYST	il server restituisce il tipo di sistema
QUIT	(quit) chiude la connessione

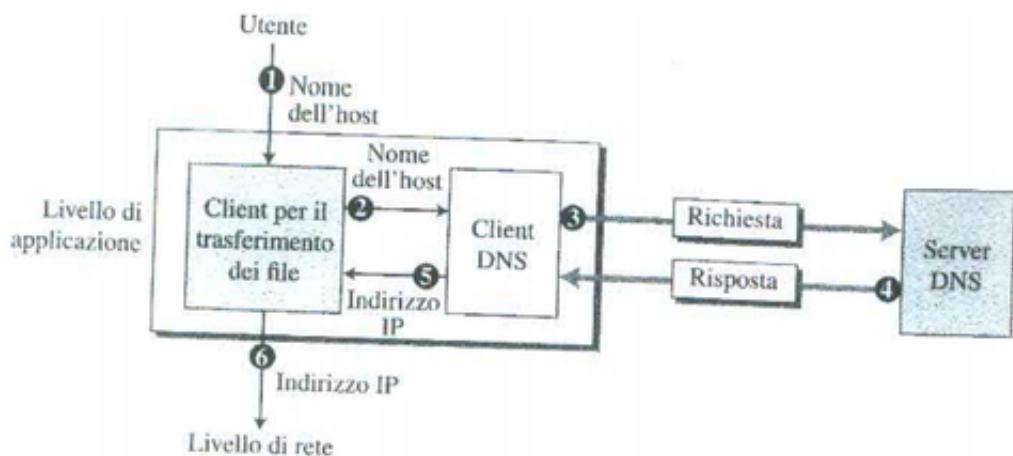
### 6.1.6 DNS

Sappiamo oramai che i dispositivi connessi in rete vengono individuati dai protocolli TCP/IP mediante i loro **indirizzi IP**, gli utenti, d'altro canto, preferiscono usare dei **nomi** invece che degli indirizzi numerici. Un **nome** identifica un **oggetto** mentre un **indirizzo** specifica **dove** l'oggetto è situato.

*Come fare per associare i nomi agli indirizzi?*

Agli albori di Internet l'associazione tra nomi logici e indirizzi IP era statica! Tutti i nomi *logici* e i relativi indirizzi IP erano contenuti in un file chiamato **host file** e periodicamente tutti gli host ne prelevavano una versione aggiornata, chiamata a sua volta **master host file**, da un server ufficiale. A noi utenti moderni però dovrebbe immediatamente saltare all'occhio la seguente **problematica**: le **dimensioni attuali di Internet** rendono questo approccio **impraticabile**. Non sarebbe infatti possibile che ogni host possa una **copia aggiornata** di un elenco del genere, in più la dimensione di questo sarebbe sterminata, per non parlare del **volume di traffico** sul server ufficiale, il pericolo dovuto alla presenza di un **unico punto di fallimento** e l'**impossibilità di scalare** di questo sistema.

Fu così che all'inizio degli anni '80 venne ideato il **DNS** o **Domain Name System**. L'**idea centrale** è di **suddividere** la sconfinata mole di associazioni nome-indirizzo e **distribuirne** le varie parti su **calcolatori sparsi per il mondo**. *Come funziona? Così:*



Supponiamo che un utente utilizzi un client di trasferimento file per accedere a un file su un server. L'utente conoscerà solo il **nome** del server e.g. *cheneso.com*, lo **stack TCP/IP** invece ha bisogno dell'**indirizzo IP** del server per stabilire una connessione. Alla pagina seguente troviamo i sei passi necessari per **associare l'indirizzo IP** al **nome** del server.

1. L'**utente** comunica il nome del server al client di trasferimento file.
2. Il **client di trasferimento file** trasmette il nome del server al **client DNS**.
3. Ogni computer una volta avviato conosce l'indirizzo IP di un **server DNS**, il **client DNS** invia dunque, usando la **porta 53**, al **server DNS** la richiesta di traduzione del nome simbolico del server.
4. Il **server DNS** risponde con l'indirizzo IP del server desiderato.
5. Il **client DNS** comunica al **client di trasferimento file** l'indirizzo IP del server.
6. Il **client di trasferimento file** utilizza così l'indirizzo IP ricevuto per accedere al server.

Per far sì che questo meccanismo funzioni bisogna **eliminare le ambiguità sui nomi** e per far ciò si è definito uno **spazio dei nomi**. Lo **spazio dei nomi** ha una **struttura gerarchica** per una ragione principale: se tutti i nomi fossero composti da una sola stringa alfanumerica servirebbe un'**autorità centrale** che controllasse l'unicità di ogni singolo nome. Una **struttura gerarchica** consente invece di avere nomi composti da diverse parti e di **delegare il controllo** su ciascuna parte a enti o società diverse, **decentralizzando** in tal modo il processo di controllo.

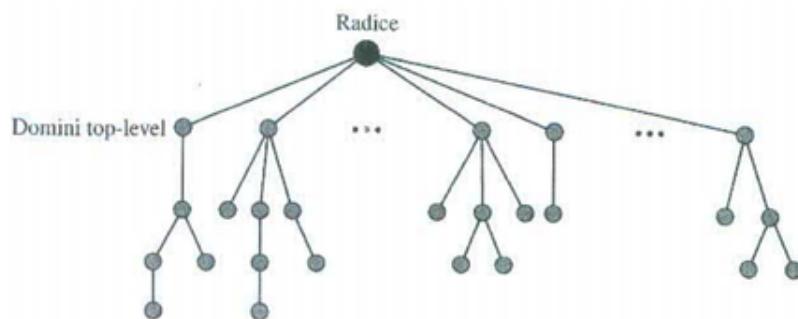


Figure 35: In figura lo spazio dei nomi di dominio.

Nello **spazio dei nomi di dominio** i nomi hanno una struttura ad **albero** con la radice in cima e un numero di livelli compreso tra 0 e 127. Ogni nodo è individuato da un'**etichetta** costituita da **massimo 63 caratteri** (*la radice ha l'etichetta vuota*), tutti i nodi collegati a uno stesso nodo da rami diversi hanno etichette **diverse**, ciò garantisce l'**univocità** dei nomi. Ogni nodo dell'albero ha inoltre un **nome di dominio**, che, se letto da sinistra verso destra, è costituito da tutte le etichette, separate da punti, di tutti i nodi a partire dal nodo stesso fino alla radice, la cui etichetta è la stringa **nulla**. Immediatamente sotto la radice si trovano i **domini top-level**.

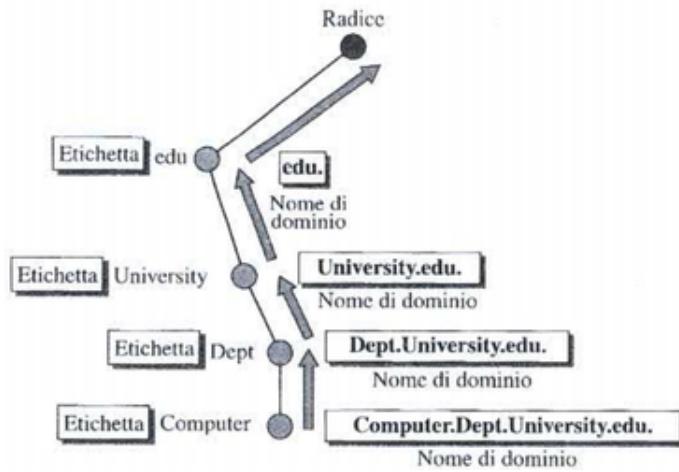


Figure 36: In figura le etichette e i nomi di dominio.

**Definizione 6.8.** Un **dominio** è un **sottoalbero** dello spazio dei nomi che viene identificato dal **nome di dominio** della sua radice.

*Ma dove sono contenute le informazioni relative allo spazio dei nomi di dominio?*

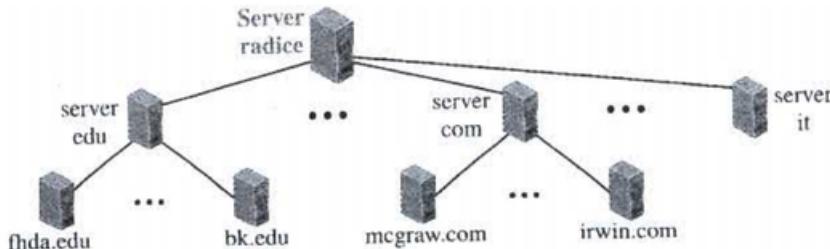


Figure 37: In figura la gerarchia dei name server.

All'interno dei **DNS-Servers** o **name servers**. L'intero spazio dei nomi è stato **diviso** in diversi domini, differenziati al **top-level**, e i domini così ottenuti sono stati divisi a loro volta ottenendo dei sotto domini. Ogni **name server** è responsabile di un dominio o di un sottodominio. Vi è quindi una **gerarchia di server**. Una **zona** è tutto ciò di cui è responsabile un **server**, se un server è responsabile di un dominio e non effettua suddivisioni in sottodomini, allora la sua zona e il suo dominio **coincidono**. Se invece il server suddivide il proprio dominio in sottodomini la sua zona e il suo dominio **differiscono**. Ogni name-server ha un database chiamato **file di zona** contenente le informazioni su tutti i nodi che ricadono nella **sua zona** di competenza.

Un **root server** è un server che ha per zona l'**intero albero dello spazio dei nomi**.

Soltanente i **root server** si limitano a immagazzinare riferimenti relativi ad altri server e **delegano** loro tutte le responsabilità.

Esempio di top-level domains:

Dominio	Uso
com	organizzazioni commerciali
edu	istituti di istruzione
mil	gruppi militari
gov	istituzioni governative americane
net	principali centri di supporto alla rete
org	organizzazioni diverse dalle precedenti
it, uk, us, fr, ecc.	codice geografico per nazioni

I server DNS possono essere **primari** o **secondari**.

- Un server **primario** possiede sul disco e aggiorna il **file di zona** relativo alla zona sotto la sua responsabilità.
- Un server **secondario** riceve le informazioni relative a una zona da un **server primario**.

I server **primari** e **secondari** hanno la medesima autorità sulla loro zona di competenza, è bene specificare che **un server può essere primario per una zona e secondario per un'altra**, **primario** e **secondario** sono quindi aggettivi **relativi alla zona**. Si noti inoltre come l'introduzione di un server secondario in una zona porti a una duplicazione del **file di zona** che può risultare utile per eventuali guasti al **server primario**.

*Come si associa l'indirizzo IP da un nome?*

Il processo con cui si associa l'indirizzo IP da un nome è detto **processo di risoluzione**. Il **protocollo DNS** è progettato come protocollo **client-server**. Un host che voglia ricavare un indirizzo IP da un nome si rivolge al **programma client** detto anche **resolver** che invierà un'opportuna richiesta al **server DNS più vicino** il quale, se dispone della risposta, invierà l'indirizzo o il nome (*è infatti possibile fare il processo al contrario*), oppure inoltrerà la richiesta a un altro server o comunicherà al resolver l'indirizzo di un altro server a cui fare riferimento. Il **resolver**, ricevuta la risposta, la esaminerà per verificarne la presenza di errori e la trasmetterà al processo che l'ha effettuata. La **risoluzione** può essere **ricorsiva** o **iterativa**.

- Con la **risoluzione ricorsiva**, la **query DNS** viaggia dall'host su cui è in esecuzione il processo applicativo che ne ha fatto richiesta fino a un host che conosce l'indirizzo IP richiesto, eventualmente scalando, poi discendendo e infine percorrendo a all'indietro l'**intera gerarchia dei server DNS**. Al server locale che ha richiesto la query viene inviato solo l'indirizzo IP risultante, ha richiesto quindi una **conversione completa**.

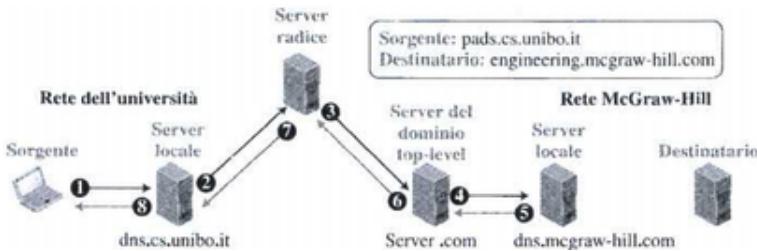


Figure 38: In figura un esempio di risoluzione ricorsiva.

- Con la **risoluzione iterativa** ogni **server-DNS** che non è in grado di risolvere la **query-DNS** dell'host risponde, **direttamente al server locale**, con l'indirizzo di un altro server in grado di risolverla. Al server locale che ha richiesto la query non viene quindi inviato solo l'indirizzo IP risultante. La **risoluzione iterativa** è supportata da **tutti i name server**, l'host può richiedere che venga usata la **risoluzione ricorsiva** ma potrebbe non essere disponibile.

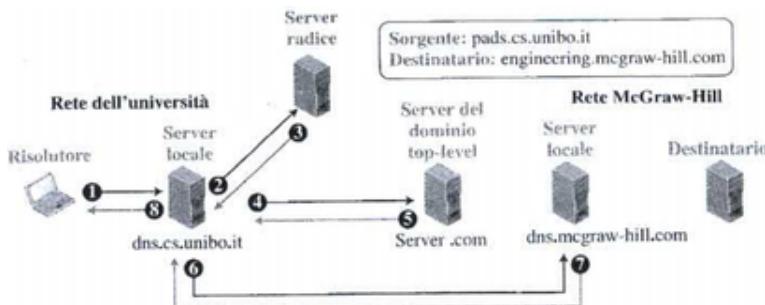


Figure 39: In figura un esempio di risoluzione iterativa.

Ogni volta che a un **server** arriva una richiesta di risoluzione di un nome, che non fa parte del suo dominio, deve cercare, all'interno del suo database, un altro server a cui inoltrare la richiesta. Ridurre questo tempo di ricerca significa ridurre il tempo di attesa della risposta e aumentare l'efficienza. Ancora una volta ci viene in aiuto il **caching**: una volta che un **server** ha appreso un' **associazione**, la inserisce in **cache**. Il server marca le risposte prese dalla propria cache come **unauthoritative** o non autorevole.

Dopo un certo lasso di tempo chiamato **TTL** o **time to live** il server cancella dalla cache l'associazione per evitare l'invio di risposte obsolete.

*Ma come è fatto quindi il database di un server?*

Il **database del server** non è altro che una collezione di **records** strutturati nel seguente modo:

(Nome di dominio, Tipo, Classe, TTL, Valore)

A ogni **nome di dominio**, quindi a ogni **nodo dell'albero dello spazio dei nomi di dominio**, è associato un record composto da 5 campi.

- **Nome di dominio** identifica il record della risorsa.
- **Tipo** definisce come interpretare il campo **Valore**.
- **Classe** definisce il tipo di rete, IN sta per Internet.
- **TTL** indica il numero di secondi per cui l'informazione deve essere ritenuta valida
- **Valore** contiene l'informazione memorizzata relativa al **nome di dominio**.

Di seguito i **tipi** dei **record**:

Tipo	Interpretazione
A	indirizzo Ipv4 a 32 bit
NS	identifica i server autoritativi di una zona
CNAME	il nome di dominio è un alias per quello ufficiale
SOA	informazioni autoritative riguardanti una zona
MX	server di posta del dominio corrente
AAAA	indirizzo Ipv6

*Che protocollo di livello trasporto è usato?*

Il sistema DNS può usare sia il **protocollo TCP** che l'**UDP**. Il **protocollo UDP** viene usato quando la dimensione del messaggio di risposta è **inferiore a 512 byte**, molto spesso infatti i **datagrammi utente UDP** non possono superare i 512 byte come dimensione massima. In caso contrario si usa il **protocollo TCP**.

*Come si aggiungono nuovi domini al DNS?*

Pagando i **registrar**, ovvero aziende commerciali accreditate dall'ICANN.

*Come sono fatti i messaggi DNS?*

I messaggi DNS posso essere di **due tipi**: di **interrogazione** o di **risposta** e i due tipi hanno lo **stesso** formato.

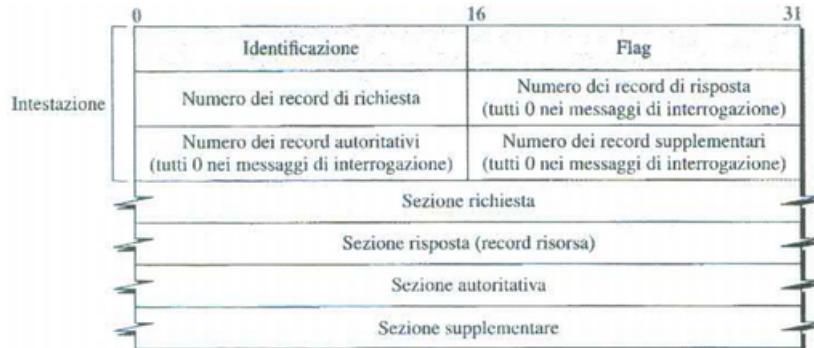


Figure 40: La struttura dei messaggi DNS, il messaggio interrogazione contiene solo la sezione richiesta mentre il messaggio di risposta contiene la sezione richiesta, la sezione risposta ed eventualmente le altre due.

Descriviamo ora brevemente i vari campi situati prima dell'**intestazione** lunga **12 bytes**:

- **Identificazione** è il campo usato dal **client** per associare la risposta all'interrogazione.
- Il campo **Flag** indica se si tratta di un messaggio di richiesta o di risposta e segnala inoltre la presenza di eventuali errori.
- I **quattro** campi successivi dell'**intestazione** specificano il numero di ciascun tipo di record presente nel messaggio.

Poi del **messaggio**:

- La **sezione di richiesta** che è inclusa nell'interrogazione e poi ripetuta anche nel messaggio di risposta è **formata da uno o più record di richiesta**.
- La **sezione di risposta** è presente **esclusivamente** nei messaggi di risposta consiste **in uno o più record di risorsa**
- La **sezione autoritativa** fornisce le informazioni di uno o più server autorevoli per l'interrogazione.
- La **sezione supplementare** contiene **informazioni aggiuntive** che potrebbero essere **utili** al **client DNS**.

*In UNIX si può utilizzare il comando nslookup per ottenere associazioni nome simbolico : indirizzo numerico.*

### 6.1.7 EMAIL - SMTP

La **posta elettronica** è uno dei primi servizi applicativi di Internet, la sua nascita risale infatti al 1971, quando un tale Ray Tomlinson installò su ARPANET un sistema in grado di scambiare messaggi fra le varie università. La **posta elettronica** consente agli utenti di scambiarsi **messaggi**, sebbene si siano trattate **altre applicazioni** fornenti questo servizio e.g. **HTTP** e **TCP**, il funzionamento della posta elettronica è tale da distinguersi da queste due applicazioni citate poco prima. Infatti in applicazioni come **FTP** e **HTTP** il programma **server** è **sempre attivo** e in attesa di richieste, che quando arrivano, vengono processate. Vi è quindi una richiesta e una risposta. Nella **posta elettronica** il **funzionamento** è **differente**: infatti l'invio di un messaggio è considerato una **transazione unidirezionale**, la risposta può anche non arrivare e se arriva è considerata un'altra transazione unidirezionale. Secondariamente **non avrebbe molto senso tenere in esecuzione continuata un programma server** in attesa che qualcuno ci invii un messaggio, potremmo ragionevolmente voler spegnere il computer nell'attesa. Ciò implica che l'idea di modello client-server debba essere realizzata in un altro modo, magari utilizzando dei **server intermedi** e **disaccoppiando** le funzionalità di **ricezione** da quelle di **invio**. Di seguito illustriamo concisamente l'architettura della posta elettronica.

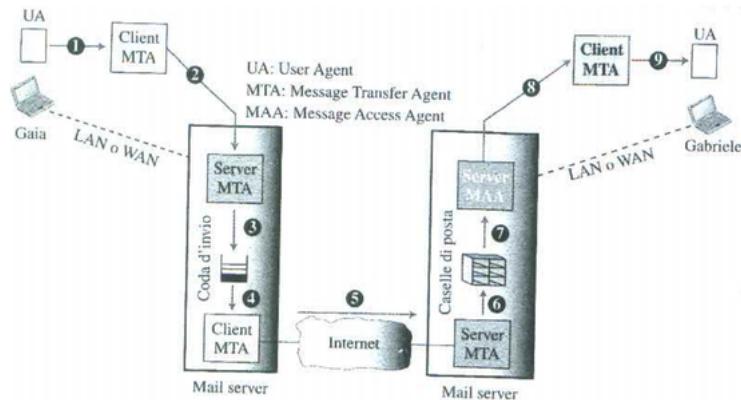


Figure 41: Un esempio di architettura di posta elettronica.

Tipicamente due utenti, mittente e destinatario, sono connessi a due **server di posta**. Ogni utente possiede una **mailbox** sul server, ovvero una porzione sulla memoria del server a cui solo lui può accedere e ogni server ha una **coda di invio**, o **spool**, dove memorizza i messaggi in attesa di essere inviati. Più:

- **UA o user agent**, prepara il messaggio e lo invia al server.
- **MTA o message trasfer agent**, è un'applicazione **push**.
- **MAA o message access agent**, è un'applicazione **pull**.

Illustriamo ora **come avviene l'invio di un messaggio** da parte di *Gaia* a *Gabriele*.

1. Gaia, utilizzando un programma **UA**, prepara il messaggio e lo invia al proprio server di posta.
2. Il **server di posta memorizza** il messaggio in una coda, e lo **invia** tramite un **programma MTA** al server di posta di Gabriele.  
**N.B. sono necessari due programmi MTA per ogni server di posta, un MTA server sempre attivo, in attesa di messaggi, e un MTA client, attivo all'evenienza, che contatta il server di posta a cui deve essere inviato un messaggio ed effettua l'invio.**
3. Il server di posta **riceverà** il messaggio e lo **memorizzerà** nella **casella postale di Gabriele** che, a sua volta usando un programma **MAA client**, contatterà il programma **MAA server** del **server** e riceverà infine il messaggio.

Lo **user agent** è il primo componente di un sistema di posta elettronica, **facilita all'utente l'invio e la ricezione dei messaggi**. Può avere o un'interfaccia a **riga di comando**, ma è abbastanza desueto, o di tipo **grafico**. Se l'utente decide di leggere i messaggi nella sua casella di posta elettronica lo **user agent** mostra un elenco dei messaggi ricevuti. Ogni riga dell'elenco offre un breve resoconto del messaggio, solitamente: **indirizzo del mittente, ora e oggetto del messaggio**.

*Come sono fatti gli indirizzi di posta elettronica?*

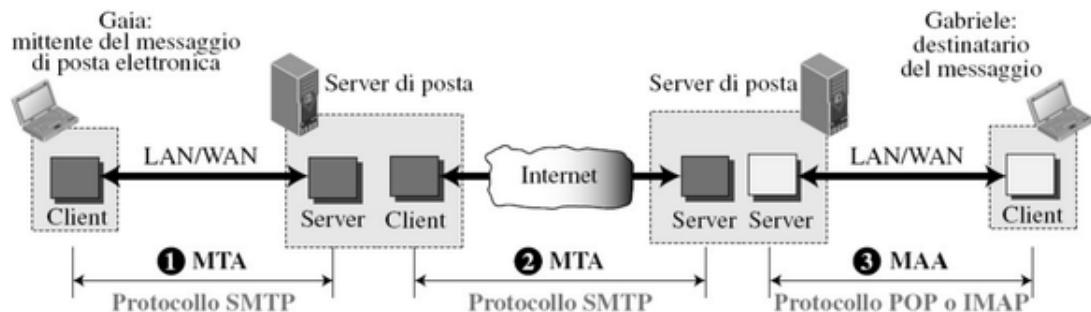
Gli indirizzi di posta elettronica individuano gli utenti in modo **univoco**. Su **Internet** consistono di **due parti**:



La **parte locale** identifica la **casella di posta del destinatario** sul server e il nome di dominio identifica il **server**.

*Come viaggiano i messaggi di posta elettronica?*

Con il protocollo **SMTP** o **simple mail transfer protocol**. Il protocollo **SMTP** definisce in maniera formale l'**interazione tra il client MTA e il server MTA**. Nell'operazione di invio di un messaggio SMTP è utilizzato **due volte**: tra il **mittente** e il **suo server di posta** e tra il **server di posta del mittente** e **quello del destinatario**. Di seguito un'immagine per schiarire le idee.



Il protocollo **SMTP** definisce come deve avvenire l'interazione, per mezzo di **comandi** e **risposte**, tra **client** e **server MTA**. I comandi sono inviati dal **client MTA** al **server MTA** e viceversa le risposte. I **comandi** e le **risposte** terminano tutti con la medesima **coppia** di caratteri: **ricono a capo e fine linea**.

La connessione tra **client MTA** e **server MTA** è una sola ed è **bidirezionale**.

- I **comandi** sono composti da una **keyword** e da **uno o più argomenti**.

Nome	Argomenti	Significato
HELO	nome host mittente	host mittente si identifica
MAIL FROM	mittente del messaggio	identifica mittente messaggio
RCPT TO	destinatario	identifica destinatario messaggio
DATA	corpo del messaggio	il messaggio
QUIT		termina sessione SMTP corrente
RSET		interrompe la transazione in atto
VRFY	nome destinatario	verifica validità nome destinatario

- Le **risposte** sono costituite da un **codice a tre cifre** seguite eventualmente da testo.

Codice	Descrizione
220	servizio pronto
221	servizio in chiusura canale trasmissione
250	comando richiesto completato
354	corpo del messaggio
421	servizio non disponibile
450	mailbox non disponibile
502	comando non disponibile

N.B. Il protocollo **SMTP** usa il protocollo di trasporto **TCP** per consegnare in maniera **affidabile** i messaggi.

La consegna di un messaggio tramite SMTP prevede **tre fasi**:

1. **Apertura della connessione o handshaking** con cui il client SMTP stabilisce una connessione TCP alla porta nota 25 con il server SMTP. Consiste di altre **tre sotto fasi**:
  - 1.1. Il **server** invia al **client** il codice **220** per indicare che è **pronto alla ricezione di messaggi** o il codice **421** in caso **contrario**.
  - 1.2. Il **client** si **identifica** con il comando **HELO** seguito dal suo **nome di dominio** in modo tale da informare il server del proprio nome di dominio.
  - 1.3. Il **server** invia il codice **250** o altri codici a seconda della situazione particolare.
2. **Invio del messaggio:** se l'apertura della connessione tra il client SMTP e il server SMTP è avvenuta con **successo**, il client può inviare **un singolo messaggio** a uno o più destinatari. Riportiamo di seguito gli **otto** passi necessari per portare a termine questa operazione:
  - 2.1. Il **client** invia al server il comando MAIL FROM con argomento l'indirizzo mail del mittente, in modo tale che, nel caso in cui si verifichino degli errori, il server **sappia a chi inviare i messaggi di errore**.
  - 2.2. Il **server** risponde con il codice 250.
  - 2.3. Il **client** invia al server il comando RCPT TO con argomento l'indirizzo mail del destinatario
  - 2.4. Il server risponde con il codice 250.
  - 2.5. Il **client** invia il comando DATA per iniziare il trasferimento del messaggio.
  - 2.6. Il **server** risponde con il codice 354.
  - 2.7. Il **client** invia il messaggio come sequenza di righe, ognuna terminante con la coppia di caratteri ritorno a capo e fine linea. Il messaggio termina con una riga **contenente solo un punto**.
  - 2.8. Il server risponde con il codice 250.
- Se ci sono **più destinatari** i passi 2.3 e 2.4 sono **ripetuti**.
3. **Chiusura della connessione:** Il **client**, trasferito il messaggio, **chiude la connessione**. Questa operazione avviene in **due fasi**:
  - 3.1 Il **client** invia al server il comando QUIT.
  - 3.2 Il **server** risponde con il codice 221.

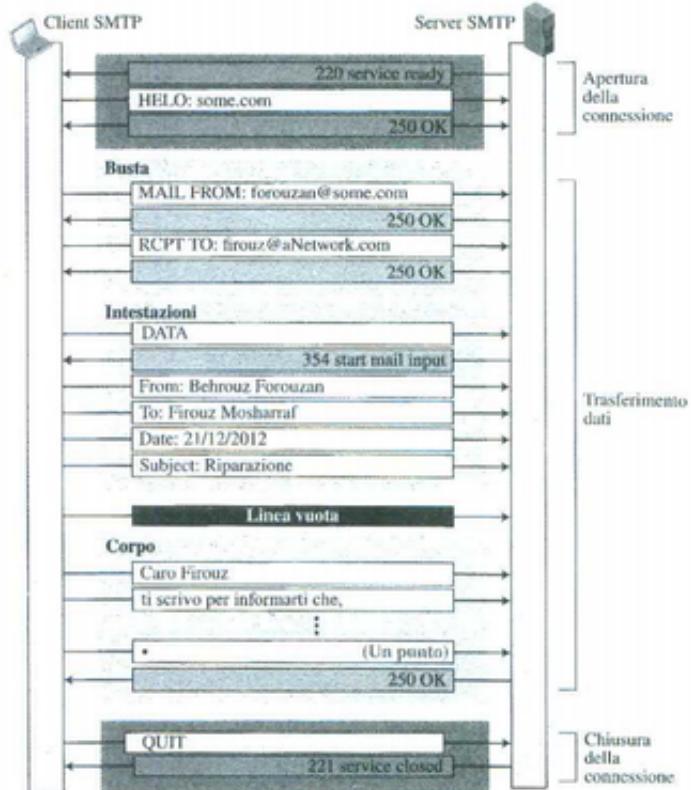


Figure 42: In figura un esempio di scambio di messaggi tramite SMTP. L'RFC 2822 definisce lo standard per il formato del messaggio. To: , From: , Subject: . **I campi sono diversi dai comandi SMTP. Il corpo del messaggio o body contiene solamente caratteri ASCII a 7 bit.**

*Come vengono ricevuti i messaggi?*

Poco sopra abbiamo descritto l'invio di un messaggio di posta elettronica come la **composizione di tre fasi**: una di **preparazione** del messaggio, una di **invio** e una di **ricezione**. Le prime due utilizzano i **protocollo SMTP**, che è un protocollo **push**, il messaggio viene *spinto* dal client mittente verso il server. L'ultima fase, quella di ricezione, usa invece un protocollo **pull**, il client destinatario *tira* i messaggi dal server. Attualmente sono in uso **due** protocolli di tipo **pull**: **POP3** o **Post Office Protocol v.3** e **IMAP4** o **Internet Mail Access Protocol v.4**. **POP3** è molto semplice ma ha funzionalità limitate. Il software **client POP3** è installato sul computer del destinatario mentre il software **server POP3** sul suo server di posta. Il **client POP3** apre una connessione TCP sulla porta **110** del **server POP3** e invia il proprio nome utente e la propria password per accedere alla casella postale. L'utente richiede poi la lista dei messaggi presenti e li preleva uno alla volta.

Il protocollo **POP3** prevede due modalità: **delete** e **keep**, con la modalità **delete** i messaggi vengono automaticamente eliminati dalla mailbox dopo il prelievo, con la modalità **keep** vengono tenuti per uso futuro.

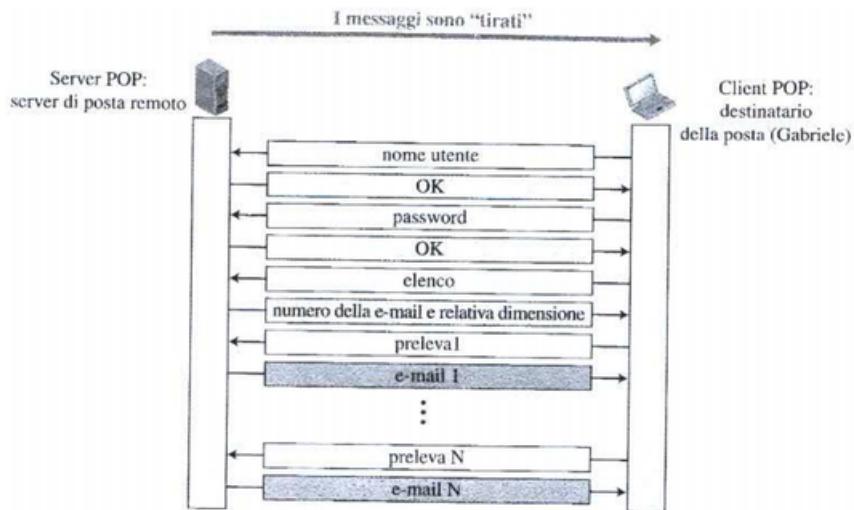


Figure 43: In figura il protocollo POP3.

Il protocollo **IMAP4** è simile al POP3 ma è più **potente** e **complesso**. POP3 non consente di **gestire più caselle** di posta sul server, di **organizzare** la posta e di **controllare una parte del messaggio** prima di **prelevarlo** nella sua interezza. IMAP4 invece fornisce varie funzionalità aggiuntive, tra le quali:

- **Controllare le intestazioni** dei messaggi prima di **prelevarli**.
- **Ricercare una stringa** specifica **nei messaggi** prima di **prelevarli**.
- **Prelevare i messaggi in modo parziale**. Utile per quando ci sono limitazioni di larghezza di banda.
- **Creare, cancellare e rinominare** le **mailbox** sul server di posta.
- **Creare una gerarchia di cartelle** all'interno della mailbox a scopo di archiviazione.

Abbiamo visto fin'ora come la posta elettronica abbia una struttura molto **elementare**, questa semplicità comporta però che i protocolli visti fin'ora supportino **soltanto** messaggi nel formato standard **NVT ASCII a 7 bit**.

*Come fare quindi a scambiarsi messaggi con caratteri non supportati dal formato NVT ASCII a 7 bit, come ad esempio il formato binario?*

Il **MIME** o **Multipurpose External Mail Extension** è un protocollo  **supplementare** che permette l'invio di messaggi in formato **diverso** dall'**ASCII**. Il **MIME** agisce **sia** dal lato **mittente** che dal lato **destinatario**. Traduce tutti i dati in formato **non ASCII** in **ASCII** **prima** di inviare il messaggio alla MTA e opera poi a ritroso quest'operazione di traduzione **presso** il destinatario dopo che ha ricevuto il messaggio tramite la MAA.

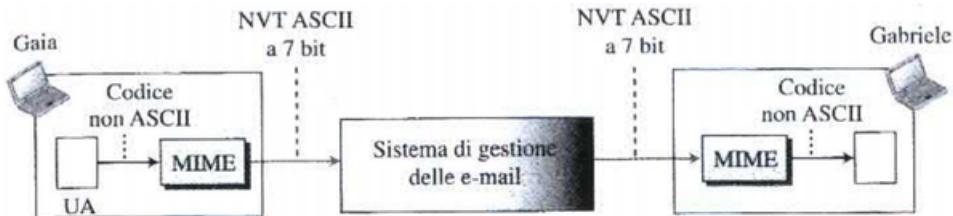


Figure 44: In figura il funzionamento del protocollo MIME.

Il **protocollo MIME** definisce **cinque** tipi di intestazioni specifiche che si aggiungono a quelle originali previste dal protocollo di posta elettronica:

- **MIME - version:** vi è dichiarata la versione di MIME usata.
- **Content Type:** vi sono dichiarati i tipi di dato contenuti nel corpo del messaggio. Il tipo del contenuto e il sottotipo sono separati da una barra. Il protocollo MIME usa sette tipi di dato diversi:
  - **Text:** **plain** o **html**, formato del testo.
  - **Multipart:** **mixed**, **parallel** o **alternative**, da informazioni circa le parti da cui è composto il messaggio.
  - **Message:** **RFC 882**, **partial**, **external-body**. Da informazioni circa il messaggio, rispettivamente: se è un messaggio incapsulato, se è una parte di un altro messaggio o se è un riferimento ad un altro messaggio.
  - **Image:** **jpeg** o **gif**.
  - **Video:** **mpeg**.
  - **Audio:** **basic**.
  - **Application:** **PostScript** o **octet-stream**.
- **Content-Transfer-Encoding:** definisce la codifica utilizzata in trasporto per il messaggio.
- **Content-ID:** individua univocamente una parte del messaggio nei messaggi che sono composti da più parti.
- **Content-Description:** indica se il corpo del messaggio contiene un'immagine, un file audio o video.

Concludiamo questa sezione offrendo una breve panoramica del servizio chiamato **web mail**. Data la larga diffusione della posta elettronica molti siti web ne offrono il servizio. I **webmail servers** a seconda del **client** interessato, che può essere **SMTP** o **HTTP**, ricevono ed evadono i messaggi tramite o il **protocollo HTTP** o il **protocollo SMTP** come illustrato nelle seguenti figure:

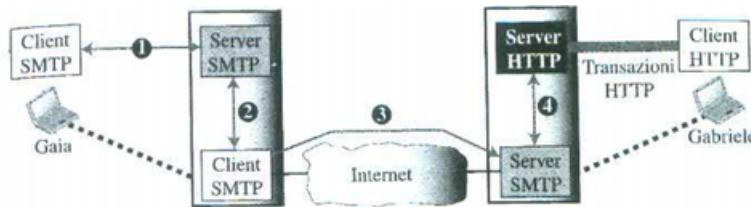


Figure 45: In figura uno scenario in cui **solo** il ricevente utilizza **HTTP**.

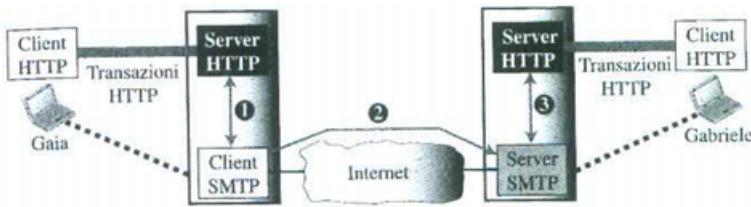


Figure 46: In figura uno scenario in cui **sia** che il ricevente che il destinatario utilizzano **HTTP**.

## 6.2 Lo strato di trasporto

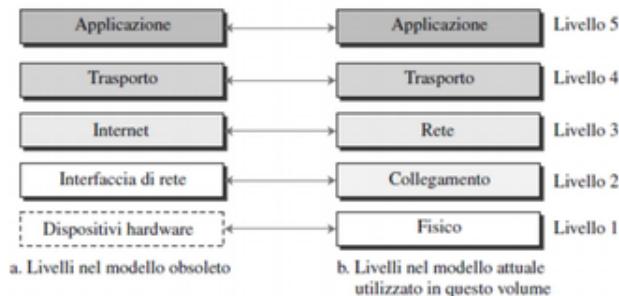


Figure 47: Livelli dello stack protocollare TCP/IP.

Nello **stack protocolare TCP/IP** il **livello di trasporto** è posizionato tra il **livello applicazione** e il **livello rete**, **fornisce** servizi al **livello applicazione** e ne **riceve** dal **livello rete**. Realizza una connessione **logica** fra **processi applicativi** in esecuzione su host system **diversi**.

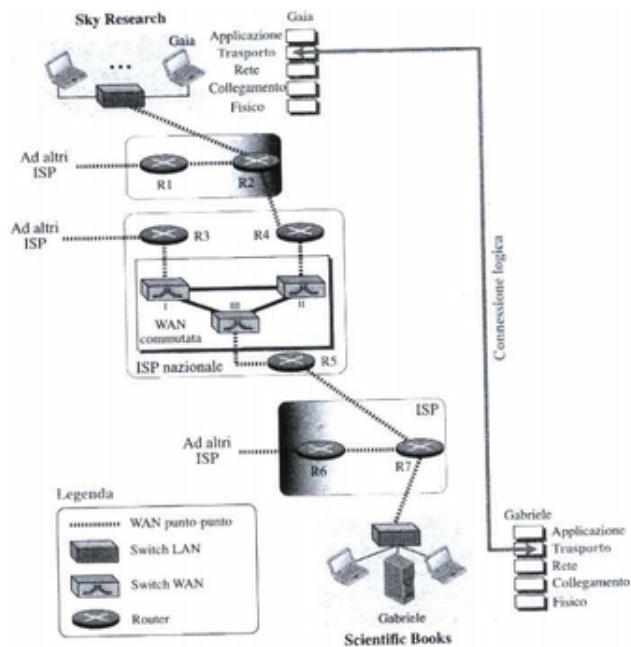


Figure 48: In figura una rappresentazione della connessione logica tra i livelli trasporto di due host.

Come anticipato poco prima, uno dei **servizi offerti** dal **livello di trasporto** è di **supportare la comunicazione** tra **processi applicativi**. Tuttavia le **informazioni, una volta inviate, prima** di arrivare ai singoli processi devono arrivare all'**host** su cui essi sono in esecuzione.

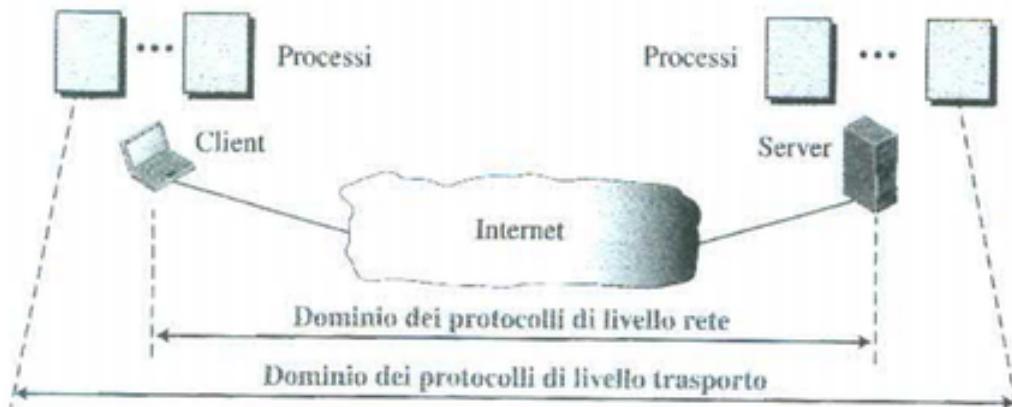


Figure 49: In figura un esempio di comunicazione tra dispositivi. Si notino i ruoli dei protocolli di rete e dei protocolli di trasporto.

I **protocolli di rete** si occupano di trasferire informazioni tra **macchine** e i **protocolli di trasporto** di indirizzarle ai **processi**.

## *Come indirizzare le informazioni verso i processi?*

Come osservato nella sezione precedente, uno dei principali modelli di comunicazione interprocessuale è il **client-server**. Per garantire quindi la **comunicazione** tra il **processo client** e il **processo server** c'è bisogno di:

1. **Identificare l'host locale e l'host remoto.** Spesso infatti, se non quasi sempre, il processo client e il processo server sono in esecuzione su host **diversi**. Gli host, in **rete**, vengono identificati mediante il loro **indirizzo IP**.
2. **Identificare il processo.** I sistemi operativi moderni sono **multiutente** e **multiprocesso**. I processi necessitano quindi di un identificatore detto **numero di porta**. I protocolli TCP/IP usano numeri di porta compresi tra **0** e **65535**, l'intervallo è codificabile con **16 bit**.

Al **client** viene **assegnato** un **numero di porta** detto **effimero**, ovvero di breve durata, dato il **breve** tempo di vita di un processo client. I numeri di porta **effimeri** sono superiori al **1023**.

Anche al **server** verrà **associato** un **numero di porta** che dovrà però essere **noto** al **client**. Nei **protocolli TCP/IP**, per i server, si usano **numeri di porta universali** detti anche **numeri di porta noti**. Ogni **client applicativo** conosce il numero di porta noto del **server applicativo** corrispondente.

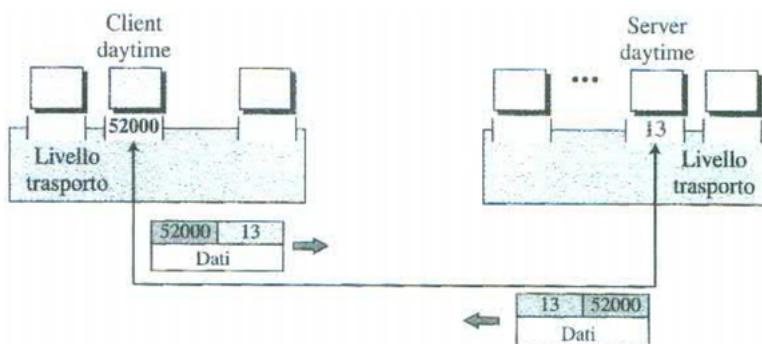


Figure 50: In figura i numeri di porta di un processo client e del corrispondente processo server.

**Ricapitolando:** l'**indirizzo IP** individua **un host** tra i miliardi di host dell'intera rete mondiale ed è un numero di 32 bit. Il **numero di porta** individua **un processo** in esecuzione su tale host.

L'**ICANN**, ovvero **Internet Corporation for Assigned Names and Numbers** ha suddiviso i numeri di porta in **tre** categorie:

1. **Numeri di porta noti**: sono compresi tra **0** e **1023** e sono **assegnati** dall'authority **ICANN**.
2. **Numeri di porta registrati**: sono compresi tra **1024** e **49151**, non sono **controllati né assegnati** dall'autority ICANN ma è **possibile registrarli** per evitare duplicazioni.
3. **Numeri di porta dinamici**: vanno da **49152** a **65535**, non sono **controllati né registrati** e possono essere usati come numeri di porta **privati o temporanei**.

*In ambiente UNIX nel file /etc/services sono memorizzate le associazioni nome server : porta nota.*

**Definizione 6.9.** La **combinazione** di **indirizzo IP** e **numero di porta** è detta **socket address**.

Quando un processo applicativo deve inviare un messaggio a un altro processo applicativo, lo passa al livello trasporto insieme a una **coppia** di **socket address**, uno che identifica il **processo stesso** e uno che identifica il **processo destinatario**. Il **protocollo di trasporto** riceve questi dati e vi aggiunge la propria **intestazione**, **incapsulando** il tutto in un **pacchetto**. Questi pacchetti sono chiamati **segmenti** nel **protocollo TCP** e **datagrammi utente** nel **protocollo UDP**. L'**incapsulamento** avviene dal lato del **mittente** e il **decapsulamento** avviene dal lato del **ricevente**.

Il metodo con cui un entità riceve informazioni da più di una sorgente è chiamato **multiplexing**, viceversa con **demultiplexing** si fa riferimento a un'entità che trasmette informazioni a più di un destinatario. Il **livello di trasporto** effettua **multiplexing** sul versante **mittente**, raccogliendo i messaggi da varie socket e incapsulandoli, e **demultiplexing** su quello del **destinatario**, consegnando i vari pacchetti in ingresso al socket appropriato.

*Come garantire un controllo del flusso?*

Il **livello di trasporto** supporta la comunicazione tra **processi applicativi**. Il **processo mittente produce informazioni** e le passa poi allo strato di trasporto **mittente** che le **consuma** e produce pacchetti. Lo strato di trasporto **destinatario** a sua volta **consuma pacchetti** e **produce informazioni** che verranno in seguito inviate al **processo destinatario**. Se i pacchetti vengono consumati con una velocità **inferiore** o **superiore** a quella con cui si è in grado di crearli o viceversa si va in contro a situazioni di **grave inefficienza** o addirittura di **perdita di dati**. Il controllo di flusso si occupa di **quest'ultima problematica**.

Una soluzione tipica per evitare la **perdita di dati** consiste nell'utilizzare **due buffer**. Uno dal **lato trasporto del destinatario** e uno dal **lato trasporto del mittente**. Se il **destinatario** ha il buffer **saturo** lo segnalerà al **mittente** che a sua volta **tratterrà** i messaggi nel **proprio** buffer e li rinvierà quando il **destinatario** segnalerà che il suo buffer non è più **saturo**.

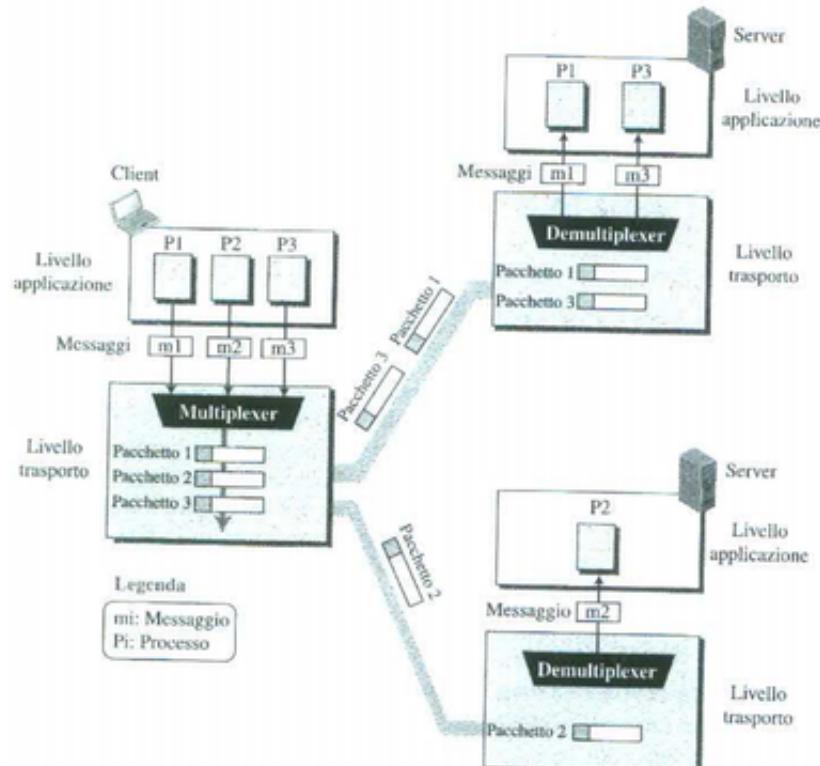


Figure 51: In figura multiplexing e demultiplexing.



Figure 52: In figura il controllo di flusso.

### *Come garantire l'affidabilità?*

Aggiungendo i **servizi di controllo degli errori** al livello di trasporto. Il **livello di rete** infatti, come si vedrà più avanti, è **fallace**, ciò comporta che si debba **garantire l'affidabilità**, se richiesta dalle applicazioni, **al livello di trasporto**. Nel **controllo degli errori** sono coinvolti solo i **livelli trasporto del mittente e del destinatario**, in particolare, è quest ultimo che gestisce il **controllo degli errori**:

- **Rileva e scarta i pacchetti corrotti**, notificando il problema al livello trasporto del mittente.
- **Tiene traccia dei pacchetti persi e scartati e ne gestisce la rispedizione**.
- **Riconosce i pacchetti duplicati e li elimina**.
- **Bufferizza i pacchetti fuori sequenza finché non arrivano quelli mancanti**.

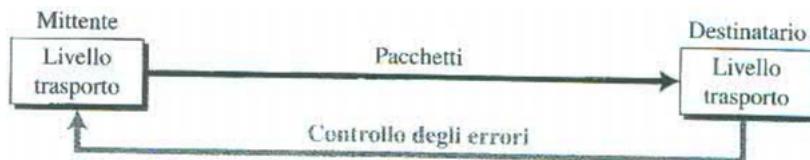


Figure 53: Il controllo degli errori.

Gli ultimi **tre** punti implicano che **il livello di trasporto del destinatario sappia riconoscere i pacchetti persi, duplicati o fuori sequenza**. Ciò è possibile grazie alla presenza di un campo, detto **numero di sequenza**, posto all'**interno** dei pacchetti. La **rispedizione** è gestita identificando il pacchetto mancante o corrotto mediante il suo numero di sequenza. I **pacchetti duplicati o mancanti** sono riconosciuti ordinando la sequenza di pacchetti entranti e verificando tutti i numeri di sequenza.

**N.B.** i numeri di sequenza vanno da 0 a  $2^n - 1$ , dove **n** è il numero di bit allocati per i numeri di sequenza specificato nell'intestazione del pacchetto. Se la sequenza di pacchetti è più lunga di  $2^n$  pacchetti si ricomincia ad enumerarli da 0, di fatto i **numeri di sequenza** sono in **modulo**  $2^n - 1$ .

e.g. per  $n = 3$  si ha la sequenza: 0 1 2 3 4 5 6 7 0 1 2 3 4 ...

Per **notificare** al mittente la corretta ricezione di uno o più pacchetti viene utilizzato il **numero di riscontro o ACK**. Il mittente identifica i pacchetti persi utilizzando un **timer** e **attivandolo dopo ogni invio**, se non riceve un **ACK prima della scadenza** allora rispedisce il pacchetto.

*È possibile combinare controllo degli errori e controllo del flusso?*

Sì, è **possibile**. Per farlo si usa una coppia di **buffer numerati**, combinando l'esigenza di avere due buffer a quella di numerare i pacchetti. Dal **lato mittente**, quando si **costruisce un pacchetto** per la spedizione, si usa come **numero di sequenza** la successiva posizione libera all'interno del buffer e lo si tiene in memoria finché non arriva l'**ACK** corrispondente. Dal **lato destinatario** invece, si tiene in memoria nel buffer il pacchetto ricevuto finché l'applicazione non è pronta a riceverlo, **solo** a quel punto viene inviato l'**ACK** e il pacchetto viene **eliminato**. L'intervallo dei numeri di sequenza: da 0 a  $2^n - 1$ , fa sì che possano essere rappresentati come un **cerchio** e il buffer, a seconda della sua **dimensione**, come una **sezione** di esso, chiamata anche **sliding window**. Quando all'interno del **buffer (sliding window)** si libera una sezione **contigua** di posizioni **a partire dall'inizio**, la **sliding window** scorre.

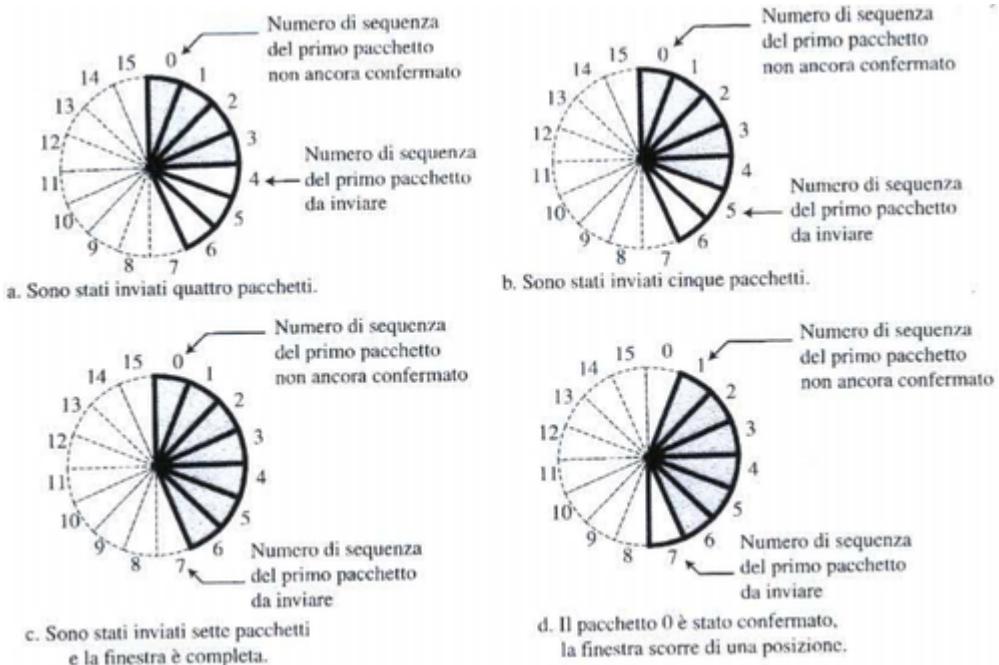


Figure 54: La sliding window. Si ricorda che è solo un'astrazione per far capire il concetto di fondo, in realtà si usano delle variabili per tenere traccia delle posizioni.

*Esiste un controllo della congestione?*

Sì, **esiste**, sebbene sia un **problema del livello rete**, il **protocollo TCP** implementa un **proprio** meccanismo di controllo della congestione.

*Che differenza c'è tra un servizio privo di connessione e un servizio orientato alla connessione?*

Un **protocollo di livello trasporto**, così come un protocollo di livello rete, può fornire **due differenti tipi di servizi**: **connection-oriented** o **connection-less**. A livello di rete un servizio **connection-less** può comportare che datagrammi facenti parte dello stesso messaggio transitino su percorsi fisici diversi. A **livello di trasporto** invece, il percorso dei pacchetti **non è rilevante**, la **presenza o l'assenza di connessione** implicano rispettivamente **la dipendenza o l'indipendenza fra i pacchetti**.

- In un servizio privo di connessione il processo applicativo mittente deve suddividere i suoi messaggi in porzioni accettabili dal **protocollo di livello trasporto connection-less**, che li tratterà come se fossero unità indipendenti e scorrelate. Le varie porzioni sono **consegnate** al livello trasporto **in ordine**, ma **non verranno ricevute** al processo applicativo destinatario **in tal modo**. Inoltre i pacchetti **non sono numerati**. Non è possibile implementare efficacemente **il controllo di flusso, il controllo degli errori e il controllo della congestione**.

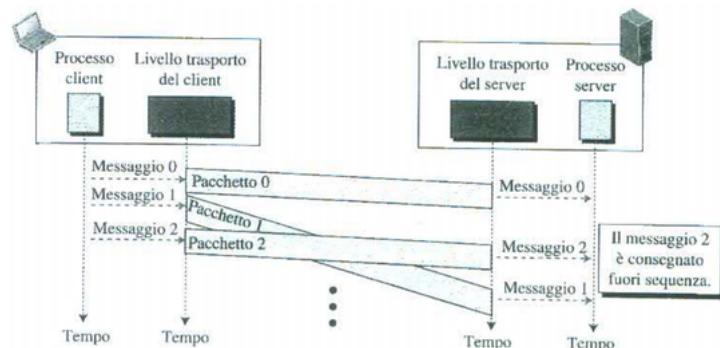


Figure 55: Un servizio privo di connessione.

- In un servizio **orientato alla connessione** i due processi comunicanti devono per prima cosa stabilire **una connessione logica** e poi possono iniziare a scambiarsi dati. Una volta **terminato lo scambio** la connessione viene **chiusa**. Come menzionato in precedenza ancora una volta l'approccio **connection-oriented** è **diverso** tra il livello di trasporto e il livello di rete. Nel livello di rete si presuppone una coordinazione totale fra gli host finali e tutti i router della rete, al livello di trasporto solo tra i due host finali. Ciò significa che **è possibile utilizzare un protocollo di trasporto connection-oriented indipendentemente dal protocollo di rete utilizzato**.

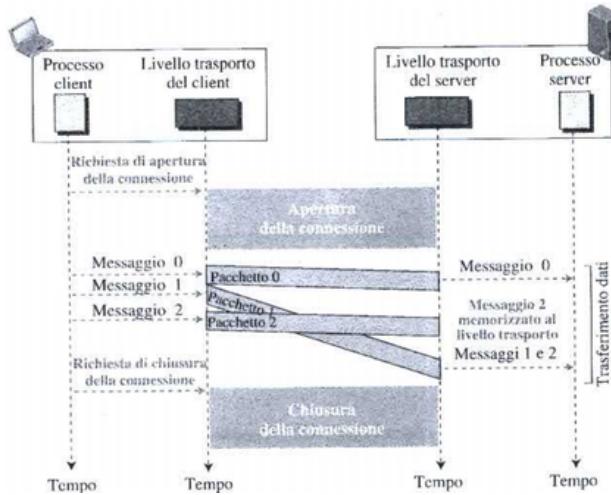


Figure 56: Un servizio orientato alla connessione.

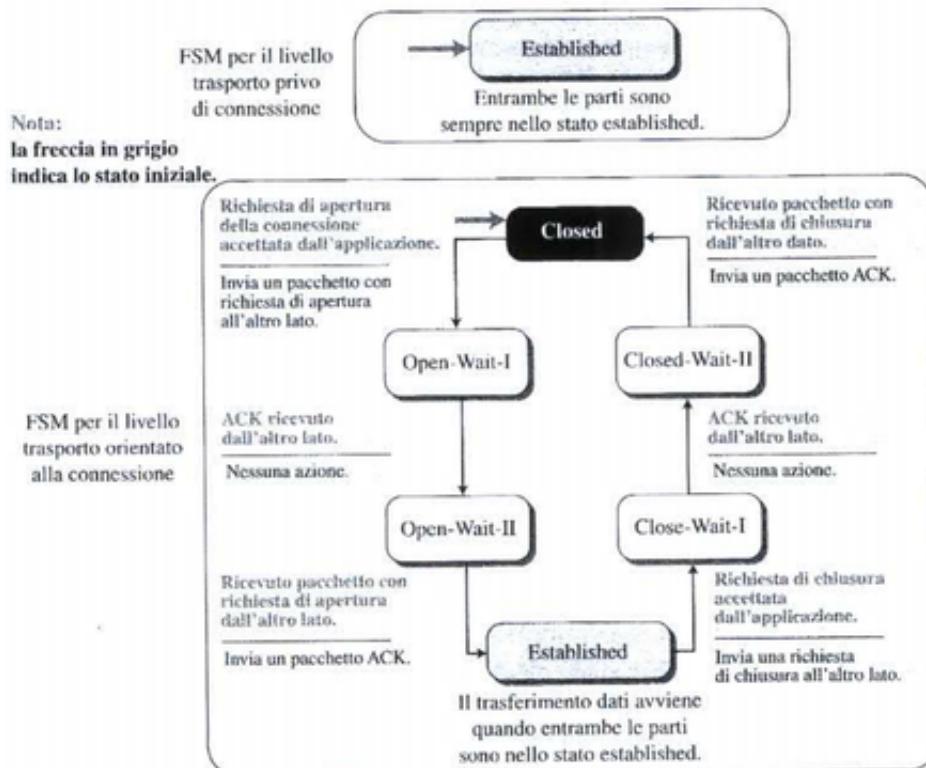


Figure 57: Differenze tra i due approcci descritti sopra rappresentati come automi a stati finiti. La rappresentazione dell'approccio connection-oriented in esempio prevede l'apertura di una connessione bidirezionale.

### 6.2.1 UDP

Il protocollo UDP o User Datagram Protocol è un protocollo di livello trasporto inaffidabile e privo di connessione. Non aggiunge niente ai servizi IP se non la comunicazione tra processi che avviene mediante l'utilizzo di due code, una in ingresso e una in uscita. Essendo molto semplice ha un overhead minimo ed è per ciò usato da processi che vogliono inviare messaggi contenuti senza preoccuparsi troppo dell'affidabilità che può comunque essere aggiunta al livello applicazione. Al livello trasporto non è effettuato nessun controllo di flusso né di congestione.

I pacchetti UDP sono chiamati datagrammi utente e data l'assenza di una connessione vengono trasmessi in modo indipendente gli uni dagli altri. La loro intestazione è costituita da 4 campi di 2 byte, per un totale di 8 byte.

N.B. Solo i processi che inviano messaggi di dimensione inferiore a 65507, i.e.  $65535 - 8$  byte di intestazione UDP =  $65527 - 20$  di intestazione IP = 65507, possono usare il protocollo UDP.

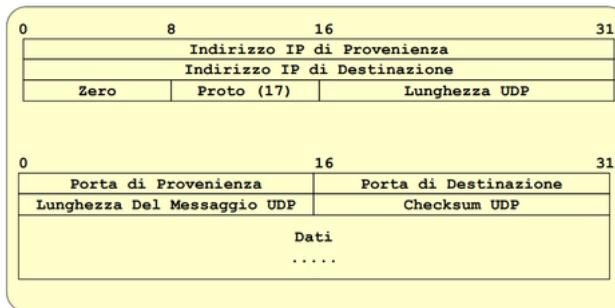


Figure 58: La struttura di un datagramma utente. La prima parte in figura è uno **pseudo header** e non viene trasmesso. Il campo proto serve a distinguere i pacchetti UDP (17) da quelli TCP.

I primi due campi definiscono i numeri di porta rispettivamente del mittente e del destinatario. Il terzo campo definisce la lunghezza totale del datagramma compresa l'intestazione. I 16 bit possono definire una lunghezza totale da 0 a 65535 ma in realtà la dimensione è inferiore, il datagramma utente viene infatti inserito in un datagramma IP di lunghezza totale di 65535. Lo **pseudoheader** è una parte dell'intestazione del **pacchetto IP** in cui viene incapsulato il datagramma utente. L'ultimo campo dell'intestazione può contenere la **checksum**, se il campo è settato con tutti 0, il mittente ha deciso di non calcolarla. La **checksum** è l'unico controllo degli errori che UDP mette a disposizione, usa tutti i campi del datagramma compreso lo **pseudoheader**, se quest ultimo non fosse incluso infatti, si rischierebbe che il datagramma, seppur privo di errori in invio, finisca a un host sbagliato per via di errori nel pacchetto IP che lo trasporta.

*Come viene calcolata la checksum?*

La **checksum** viene calcolata nel seguente modo:

- **Mittente**

1. Il messaggio viene diviso in parole da 16 bit.
2. Il valore della checksum viene inizialmente impostato a 0.
3. Tutte le parole del messaggio vengono sommate usando l'addizione complemento a 1.
4. Viene fatto il complemento a 1 della somma e il risultato è la **checksum**.
5. La checksum viene inviata insieme ai dati.

- **Destinatario**

1. Il messaggio che comprende la checksum viene ricevuto.
2. Vengono ripetuti i passaggi 3 e 4 descritti in precedenza.
3. Se il valore della checksum è 0 allora il messaggio viene accettato altrimenti viene scartato.

a capo	<table border="0"><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td colspan="16"><hr/></td></tr></table>	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	<hr/>															
1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0																																		
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1																																		
<hr/>																																																	
somma	<table border="0"><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td colspan="16"><hr/></td></tr></table>	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0	<hr/>															
1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1																																		
1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0																																		
<hr/>																																																	
checksum	<table border="0"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1																																
0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1																																		

*Come mai il DNS usa UDP?*

Il **DNS** usa **UDP** poiché le richieste che il **client DNS** invia al **server DNS** sono brevi e possono essere **contenute in un singolo datagramma UDP**. Inoltre le **risposte** devono arrivare **velocemente** e soprattutto viaggia un **solo messaggio** per volta: uno di richiesta e uno di risposta, quindi **non c'è un problema di ordine di sequenza**.

*Quali sono quindi i possibili campi di applicazione del protocollo UDP?*

- Processi che richiedono uno scambio di dati di volume limitato, con scarso interesse verso il controllo del flusso e degli errori. Non è quindi adatto ad applicazioni come **SMTP** e **FTP**.
- Processi con meccanismi interni di controllo di flusso e di errore.
- Processi di trasmissione multicast.
- Applicazioni interattive e in tempo reale che non tollerano latenza.

### 6.2.2 TCP

Il protocollo **TCP** è un protocollo **orientato alla connessione e affidabile**, è il protocollo di livello trasporto **più utilizzato** su Internet. Quando un processo sull'host A decide di inviare e ricevere dati da un processo sull'host B avvengono le seguenti **tre cose**:

1. I due processi stabiliscono una connessione **logica**.
2. Vengono scambiati dati in **entrambe** le direzioni.
3. La connessione viene terminata.

Oltre a essere **orientato alla connessione**, il protocollo TCP è anche **orientato al flusso dati**. Al contrario di UDP quindi due processi che comunicano via TCP hanno l'*illusione* di essere connessi con un **tubo**. Diremo che il processo in trasmissione **produce** il flusso mentre il processo in ricezione lo **consuma**.



Figure 59: Rappresentazione del flusso dati.

Poichè il processo produttore e il processo consumatore non scrivono e leggono dati necessariamente alla stessa velocità è necessario un **controllo del flusso** che consiste in **due buffer uno di trasmissione**, in cui si memorizzano i segmenti **inviai**, e **uno di ricezione** in cui si memorizzano quelli **ricevuti**.

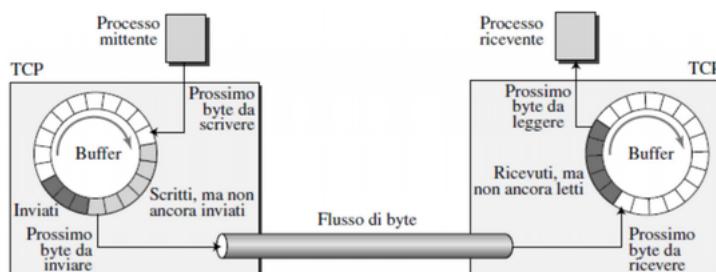


Figure 60: I due buffer.

Tipicamente viene usato un **buffer circolare**, poiché consente più agilmente di mettere a disposizione per la scrittura le celle che contengono byte inviati e la cui ricezione è stata confermata.

Sebbene la **bufferizzazione** gestisca il **controllo del flusso**, per inviare dati in uno **stream** bisogna ovviare a un altro problema. Il **livello IP infatti** invia i dati in **pacchetti** e non in un flusso di byte. Per far fronte a ciò **TCP raggruppa un certo numero di byte in unità chiamate segmenti, indipendenti** dal programma applicativo, a cui vi aggiunge un'intestazione prima di affidarli al livello rete. I **segmenti** vengono poi incapsulati in datagrammi IP e trasmessi. La **bufferizzazione** consente una **riduzione del traffico** sulla rete ottimizzando in un certo modo il numero di segmenti da trasmettere.

**N.B.** non è detto che tutti i segmenti abbiano la **stessa dimensione**.

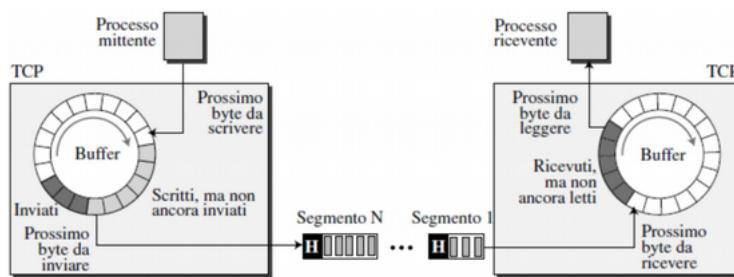


Figure 61: Il flusso di dati viene partizionato in segmenti, ognuno con il suo header.

Il **protocollo TCP** inoltre fornisce un servizio **full duplex**, nel quale, instaurata la connessione, le due entità possono **scambiarsi e ricevere** i dati **contemporaneamente**. **Ciascuna entità TCP** ha infatti i propri buffer di **invio e ricezione**.

*Come fa TCP a garantire l'affidabilità?*

Con un **meccanismo di numerazione** che prevede due campi: **il numero di sequenza** e **il numero di riscontro o ACK**. Questi due campi sono contenuti nell'**intestazione** dei segmenti TCP e fanno riferimento al **numero dei byte**, non al numero di segmento. **TCP infatti numera i byte che vengono trasmessi nella connessione**. Questa **numerazione** avviene all'interno del **buffer di trasmissione**, dopo che i byte sono stati generati dal processo applicativo mittente. I numeri di sequenza e di riscontro sono **diversi** nelle due **direzioni** della comunicazione. Il numero di sequenza del primo byte non necessariamente è 0 ma è **generato arbitrariamente** nell'intervallo tra 0 e  $2^{32} - 1$ .

Dopo aver numerato i byte, **TCP assegna a ogni segmento un numero di sequenza corrispondente al numero del primo byte del segmento**. Il **primo numero di sequenza o ISN** è un numero casuale tra 0 e  $2^{32} - 1$ . **Tutti gli altri numeri** sono il numero di sequenza del segmento precedente a cui viene sommato il numero di byte, **reali o fintizi**, da lui contenuti.

Oltre al **numero di sequenza** nell'intestazione di un **segmento TCP** è contenuto anche un **numero di riscontro**, più brevemente detto **ACK**. L' **ACK** è utilizzato dalle due entità comunicanti per **confermare i byte ricevuti**, indica infatti il **numero del prossimo byte** che l'entità si aspetta di ricevere. *e.g. se un'entità utilizza 5643 come ACK significa che ha ricevuto tutti i byte fino al 5642.* **N.B.:** non necessariamente vuol dire che l'entità ha ricevuto 5642 byte.

Riportiamo di seguito la **struttura** di un **segmento TCP**:

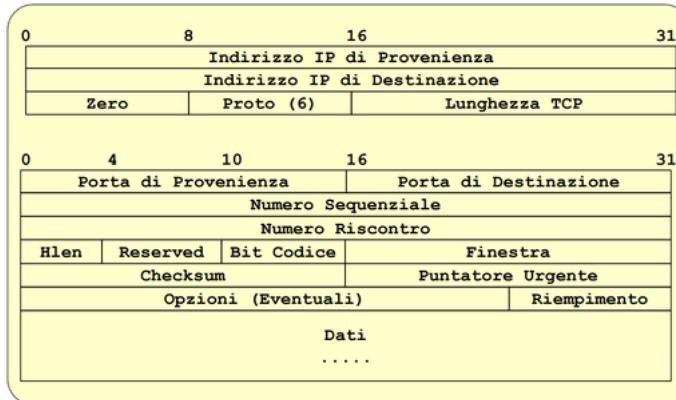


Figure 62: La struttura di un segmento TCP.

Il **segmento** consiste di un'**intestazione** di dimensione compresa tra 20 byte, in assenza di opzioni, e 60 byte altrimenti, seguita dai **dati** provenienti dal processo applicativo.

- **Numero di porta sorgente:** contiene il numero di porta del processo mittente sull'host che invia il segmento.
- **Numero di porta destinazione:** contiene il numero di porta del processo destinatario sull'host che riceve il segmento.
- **Numero di sequenza:** contiene il numero di sequenza associato al primo byte di dati contenuto nel segmento.
- **Numero di riscontro:** contiene il numero di sequenza del byte che l'entità si aspetta di ricevere.
- **HLEN o lunghezza dell'intestazione:** indica il numero di parole di 4 byte presenti nell'intestazione, varia da 5 ( $\frac{20}{4}$ ) a 15 ( $\frac{60}{4}$ ).
- **Flags di controllo:** contiene i 6 bit di controllo, più di un bit può essere attivo contemporaneamente. Questi bit intervengono nel **controllo del flusso, nell'apertura e nella chiusura della connessione e nella determinazione della modalità di trasferimento dei dati.**

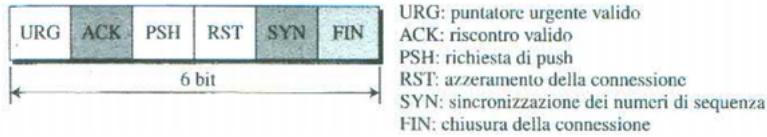


Figure 63: I flags di controllo e il loro significato.

- **Dimensione della finestra:** contiene la dimensione della finestra di ricezione di cui di cui dispone l'entità, la dimensione massima è di 65535 byte, legata ai 16 bit che compongono il campo. È solitamente indicato con RWND o receiving window ed è determinato dal ricevente, il mittente deve rispettare le sue indicazioni.
- **Checksum:** contiene il **checksum**, è calcolato da TCP con le stesse modalità descritte per UDP, tuttavia è **obbligatoriamente richiesto** per i datagrammi TCP. Lo pseudoheader ha lo stesso scopo descritto per UDP, però il **campo proto vale 6**.
- **Puntatore urgente:** è rilevante se il flag **URG** è attivato. Contiene la cifra da sommare al numero di sequenza del segmento per ottenere il numero dell'ultimo bit urgente della sezione dati.
- **Opzioni:** contiene da 0 a 40 byte di informazioni opzionali.

*Come funziona la connessione TCP?*

Il **protocollo TCP** è **connection-oriented** il che vuol dire che stabilisce una connessione logica tra il processo mittente e il processo destinatario e tutti i segmenti TCP vengono spediti lungo questo percorso. L'utilizzo di una connessione virtuale **semplifica i processi di conferma, di controllo degli errori e di rispedizione**. La trasmissione orientata alla connessione del protocollo TCP richiede **tre fasi: apertura della connessione, trasmissione dei dati e chiusura della connessione**.

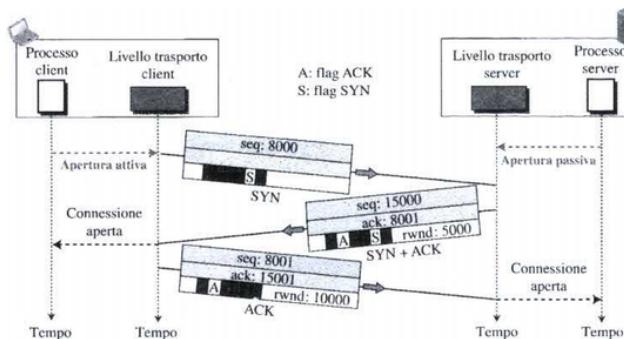


Figure 64: L'apertura della connessione TCP.

1. **Apertura della connessione:** nel TCP, essendo un protocollo **full duplex**, le due entità comunicanti devono inizializzare la connessione e ciascuna deve ottenerne l'*approvazione* da parte dell'altra. L'apertura della connessione TCP è detta **three-way handshake** e avviene nel seguente modo:
  - 1.1. Il **client** invia il primo segmento: si tratta di un **segmento SYN**, **senza dati utente** e **contenente solo il flag SYN** che specifica l' **ISN** della connessione client-server.
  - 1.2. Il **server** invia il secondo segmento, senza dati utente, con i flag **ACK** e **SYN**, l'ACK viene usato dal server per comunicare al client la corretta ricezione del primo segmento e il SYN per specificare l'ISN della connessione server-client. Dato che il segmento contiene un **riscontro**, conterrà anche la dimensione della finestra di ricezione a cui il client dovrà attenersi.
  - 1.3. A questo punto il **client** invia il terzo segmento e con il flag **ACK** comunica al server la corretta ricezione del secondo segmento. Il terzo segmento **può** contenere dati utente e se li contiene possiede anche un **numero di sequenza**.

Dopo aver instaurato la connessione è possibile trasferire i dati. Come detto sopra, il **protocollo TCP mittente memorizza i dati** generati dal processo mittente in un **buffer**, la quantità di dati che verranno poi inseriti nel segmento e inviati è a discrezione del TCP stesso. **Anche** il protocollo **TCP destinatario memorizza i dati in un buffer** e li invia al processo destinatario quando questo è pronto alla ricezione o quando il TCP destinatario lo ritiene opportuno.

*Cosa succede se non possono essere tollerati ritardi nella consegna e nella ricezione dei dati?*

Il protocollo TCP gestisce quest'eventualità con la **funzione push** che può essere **richiesta dal processo mittente**. In questo caso il **TCP mittente invia immediatamente i dati al TCP destinatario** e tramite il flag **PSH** specifica che i dati devono essere immediatamente consegnati al processo destinatario.

*Cosa succede se un processo necessita che il TCP elabori dei dati per primi indipendentemente dalla loro posizione nel segmento?*

Il processo invierà i dati urgenti in un segmento con il flag **URG** attivato, li posizionerà all'**inizio** del segmento stesso e indicherà poi al TCP mittente, mediante il **puntatore urgente**, il **numero dell'ultimo byte di dati urgenti** contenuti nel segmento. e.g. se il numero di sequenza del segmento vale 15000 e il puntatore urgente vale 200, l'ultimo byte urgente sarà il 15200.

## Come avviene la chiusura della connessione TCP?

Entrambe le entità coinvolte possono chiudere la connessione, solitamente però la chiusura della connessione è iniziata dal processo client. Sono offerte due vie per la chiusura della connessione: **handshake a tre vie** e **handshake a quattro vie con half close**.

- **Handshake a tre vie:** inizia con l'invio da parte del TCP client di un **segmento FIN** all'interno del quale viene settato il **flag FIN**. Questo segmento può contenere o meno dati utente, se non li contiene consuma solo un numero di sequenza. Il **server TCP**, dopo aver ricevuto il segmento FIN, notifica il client TCP della ricezione e annuncia la chiusura della connessione server-client inviando un **segmento FIN + ACK**. Questo segmento può contenere o meno dati da parte del server, se non li contiene consuma solo un numero di sequenza. Il **client TCP** finalizza la chiusura della connessione inviando un segmento ACK per notificare il server della ricezione del segmento precedente. Questo segmento **non contiene dati utente e non consuma numeri di sequenza**.

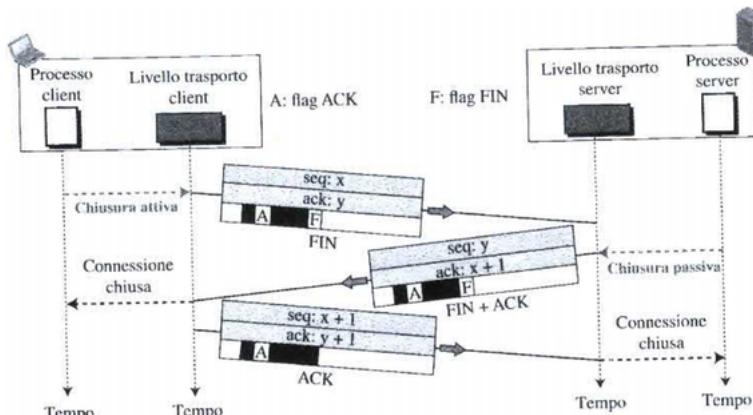


Figure 65: Chiusura della connessione tramite handshake a tre vie.

- Nella **chiusura handshake a quattro vie con halfclose** un **processo smette di inviare dati mentre ne sta ancora ricevendo dall'altro**. e.g. *l'ordinamento di una grossa quantità di dati inviati da un client a un server*. Il **client TCP** richiede la halfclose inviando un **segmento FIN**, il server accetta la richiesta di chiusura inviando un **segmento ACK**. Il trasferimento dei dati client-server **termina** ma il **server può ancora inviare dati al client**. Quando a sua volta il **server termina** l'invio dei dati, invia un **segmento FIN** al client che risponde con un **segmento ACK**.

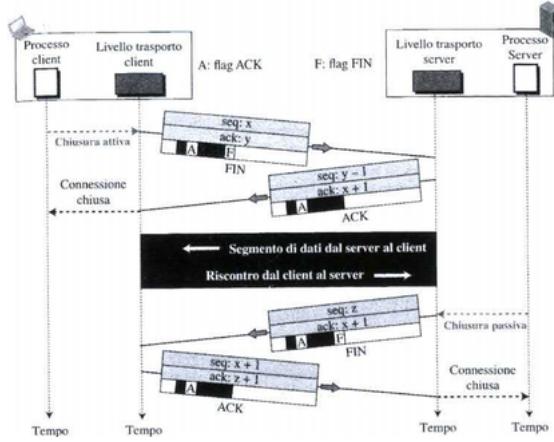


Figure 66: Chiusura della connessione tramite handshake a quattro vie con halfclose.

Una visione generale con un **diagramma delle transizioni di stato del protocollo TCP**:

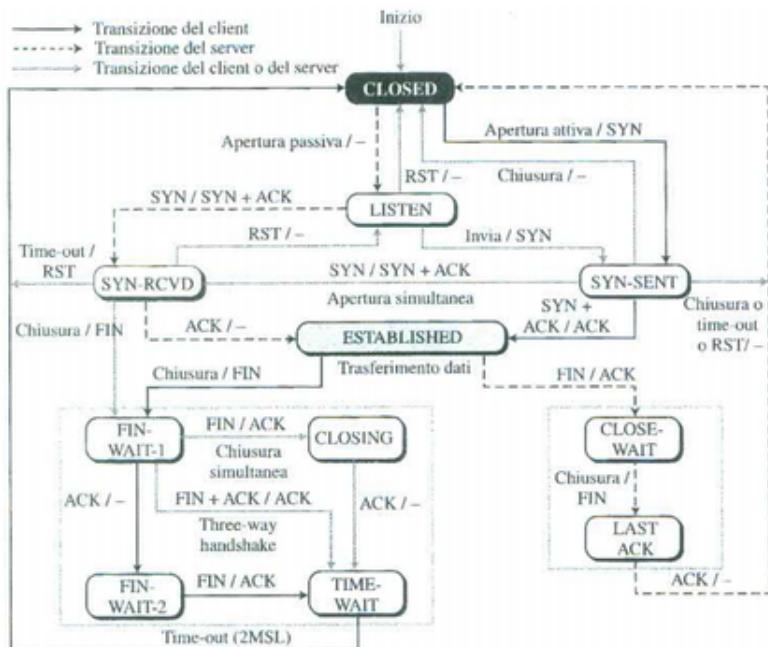


Figure 67: Il diagramma delle transizioni di stato.

**TIME\_WAIT** è lo stato finale in cui un'entità che esegue la chiusura attiva si trova prima della chiusura definitiva della connessione. Vi resta per due volte la MSL (Maximum Segment Lifetime). **Garantisce una terminazione affidabile della connessione in caso di perdita dell'ultimo ACK e consente l'eliminazione dalla rete dei segmenti duplicati.**

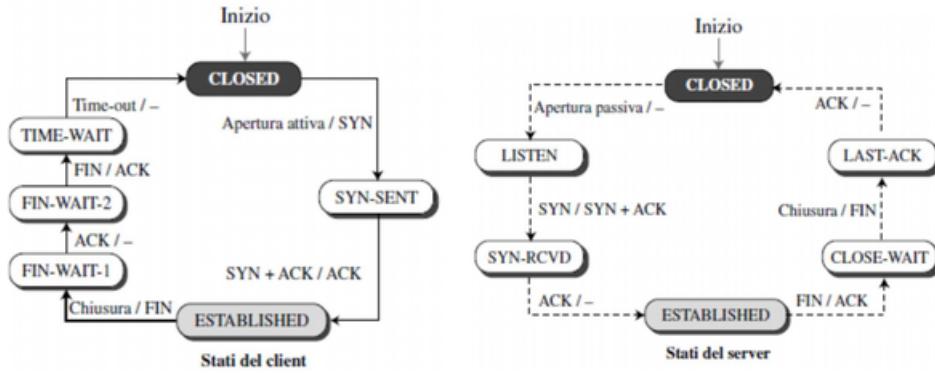


Figure 68: Il diagramma delle transizioni di stato con chiusura four-way halfclose.

Riportiamo di seguito i **significati degli stati**.

Stato	Significato
LISTEN	server in attesa di SYN
SYN-SENT	SYN inviato e attesa ACK
SYN-RECEIVED	SYN e ACK inviati e attesa di ACK
ESTABLISHED	connessione aperta
FIN-WAIT-1	primo FIN inviato attesa di ACK
FIN-WAIT-2	ricevuto ACK primo FIN attesa di secondo FIN
CLOSE-WAIT	ricevuto primo FIN e ACK inviato
TIME-WAIT	ricevuto secondo FIN e ACK inviato, attesa 2MSL
CLOSING	attesa di ACK chiusura
LAST-ACK	secondo FIN inviato attesa di ACK
CLOSED	assenza di connessione

*Come fa TCP a garantire il controllo del flusso?*

Descriviamo il **ciclo di vita dei dati** scambiati all'interno della comunicazione TCP. I dati sono:

1. **Generati** dal processo applicativo mittente.
2. Consegnati al **TCP mittente**.
3. Consegnati al **TCP destinatario**.
4. **Consumati** dal processo applicativo destinatario.

Il **controllo del flusso** è gestito con il seguente meccanismo: il **TCP destinatario** invia un feedback al **TCP mittente** che a sua volta invia un feedback al **processo applicativo mittente**. Quando la finestra del TCP mittente è **piena** i dati, provenienti dal processo, vengono **rifiutati**.

Come funzionano le finestre TCP?

Il **protocollo TCP** prevede due finestre, una di **ricezione** e una di **trasmissione**, per ognuna delle due direzioni di comunicazione, in totale sono quindi presenti **4 finestre**. Ricordiamo che TCP instaura **una sola connessione**. La **dimensione della finestra di trasmissione** è *inizialmente* determinata all'apertura della connessione. Si **apre** quando la dimensione della finestra di ricezione segnalata dal TCP destinatario lo consente e si **chiude** quando viene confermata la ricezione dei byte inviati. **Apertura, chiusura e ridimensionamento sono controllate dal TCP destinatario.**

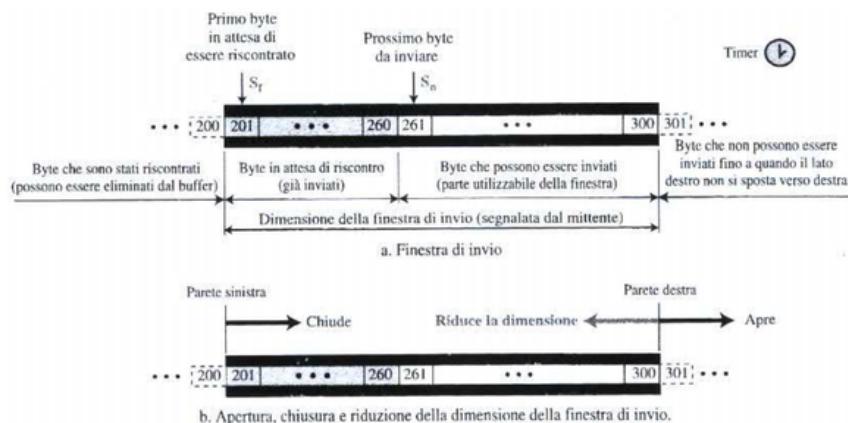


Figure 69: La finestra di trasmissione o di invio TCP.

Invece la **finestra di ricezione** si **chiude** quando giungono altri byte dal **TCP mittente** e si **apre** quando il processo applicativo destinatario richiede altri byte.

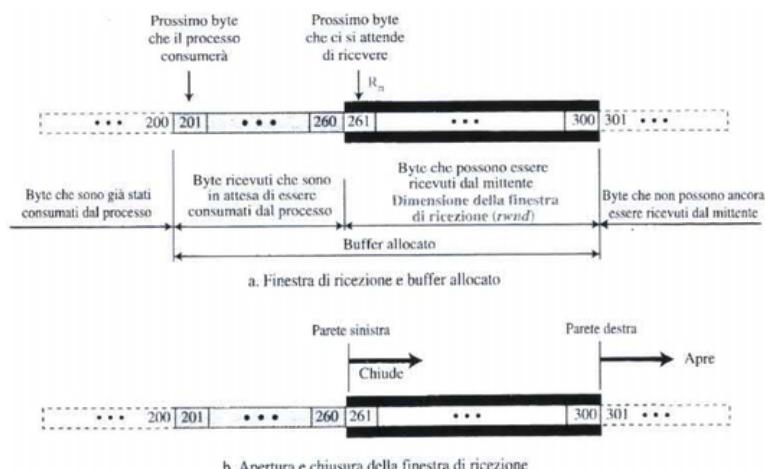


Figure 70: La finestra di ricezione TCP. Ipotizzeremo che non si riduca mai.

Alleghiamo di seguito un'immagine in grado di spiegare efficacemente ciò che succede in uno scenario **dinamico**.

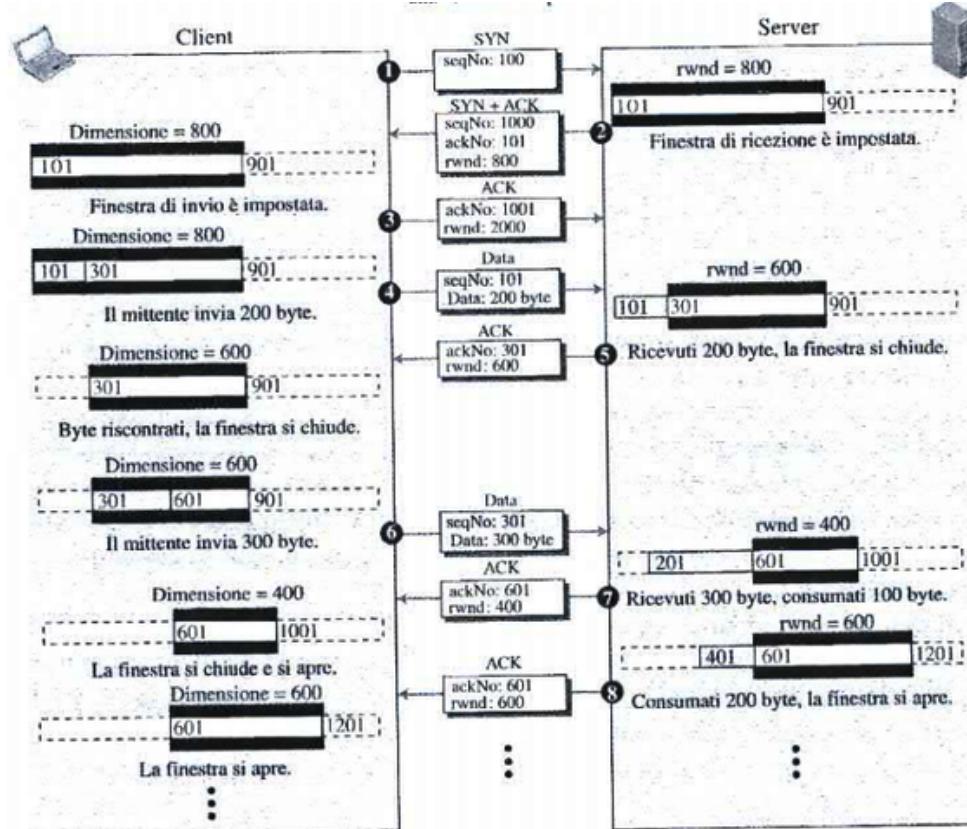


Figure 71: Il controllo di flusso in una ipotetica connessione TCP unidirezionale.

$$\text{rwindow} = \text{RecBuff} - (\text{LastByteReceived} - \text{LastByteRead})$$

Il **mittente** si assicura che:

$$\text{LastByteSent} - \text{LastByteAcked} < \text{rindow}$$

Ovvero che la quantità di dati trasmessi e non ancora riscontrati sia **minore** della **finestra di ricezione**.

**N.B.** **rwindow** può essere uguale a 0, qualora ad esempio il TCP destinatario non volesse ricevere dati per un certo periodo di tempo. Il mittente in questo caso **non** riduce realmente la dimensione della sua finestra ma continua a mandare **segmenti sonda** di 1 byte per ricevere l'**aggiornamento** sulla dimensione di **rwindow**. Questa tecnica si definisce **probing** ed è usata per **prevenire lo stallo**.

*Come fa TCP a garantire il controllo degli errori?*

Come è ormai più che noto **TCP è un protocollo di trasporto affidabile** e garantisce al processo applicativo che i dati vengano consegnati in **sequenza, senza errori, smarrimenti o duplicazioni**.

Gli strumenti utilizzati da TCP per individuare gli errori di trasmissione sono **tre**:

1. **La checksum:** ciascun **segmento TCP** contiene un campo **checksum** di 16 bit **obbligatorio** e utilizzato per identificare i **segmenti corrotti**. Se un segmento viene rilevato essere tale viene **scartato** e considerato **smarrito**.
2. **I messaggi di riscontro o ACK:** TCP usa gli ACK per riscontrare la **ricezione dei segmenti** che contengono un **numero di sequenza**, possono essere segmenti **dati** o di **controllo**. I segmenti ACK **non** usano numeri di sequenza e perciò **non** vengono **riscontrati**. Nella sua **implementazione originale** TCP è stato progettato per riscontrare i segmenti in modo **cumulativo**, ovvero notificando il numero del byte che si attende e ignorando i segmenti duplicati e fuori sequenza. Nelle **versioni più recenti** è implementato il **selective ack o SACK** il quale prevede che i **pacchetti** ricevuti **fuori sequenza** vengano ugualmente **memorizzati** e inoltre sia presente un riscontro per i pacchetti fuori sequenza e quelli duplicati contenuto nel campo **OPTIONS**. Un'entità TCP genera **riscontri** quando:
  - **Invia un segmento dati** a un'altra entità e **contemporaneamente invia l'ACK** contenente numero del prossimo byte che si aspetta di ricevere, così da **ridurre il traffico**. Se destinatario **non** ha dati da inviare e riceve un segmento in ordine ritarda l'invio dell'ACK di **500ms** a meno che non riceva un nuovo segmento sempre per una ragione di riduzione del traffico.
  - Vengono ricevuti **due segmenti nel giusto ordine e nessuno dei due è ancora stato riscontrato**. Viene inviato immediatamente un segmento ACK, dal destinatario al mittente, per **evitare la ritrasmissione inutile dei segmenti**.
  - Arriva un **segmento fuori sequenza** il destinatario invia immediatamente un ACK per consentire la **ritrasmissione rapida dei segmenti**.
  - **Arriva un segmento mancante o un segmento duplicato**.
3. **I timeout:** il **TCP mittente** inizializza un **timer di ritrasmissione**, o **RTO** per ogni segmento inviato. Il **primo** segmento inviato e in attesa di riscontro, che ricordiamo essere memorizzato nel buffer del TCP mittente, allo scadere del timer, è **ritrasmesso**.

**N.B.** Un segmento viene ritrasmesso o alla **scadenza del timer** o quando vengono ricevuti **tre ACK duplicati** per quello precedente, questa funzione è chiamata **ritrasmissione veloce**.

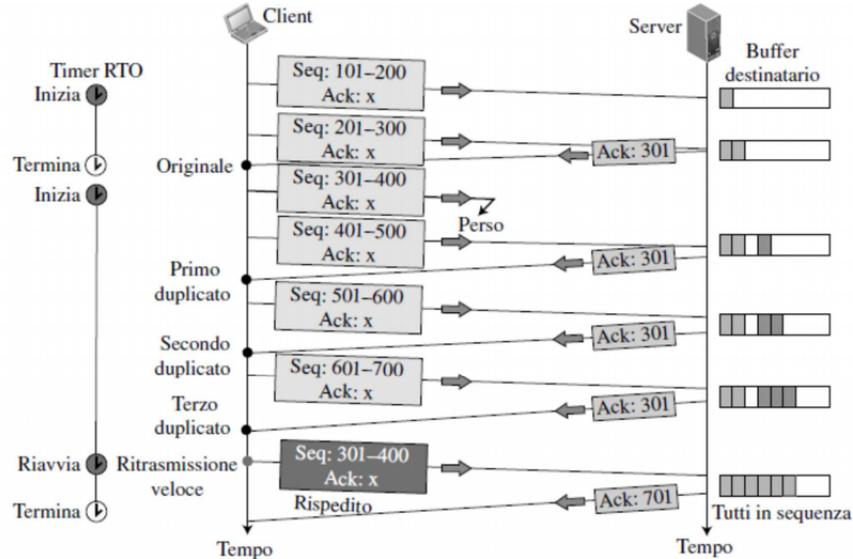


Figure 72: Un'esempio di scambio di dati tra due entità con un esempio di ritrasmissione veloce. Si supponga in uso una versione moderna di TCP dove i pacchetti fuori sequenza vengono memorizzati e non scartati.

L'RTO ha un **valore dinamico** calcolato in base al tempo di andata e di ritorno dei segmenti trasmessi detto anche **RTT** o **Round Trip Time**.

Presentiamo ora una collezione di vari scenari di operatività del protocollo TCP.

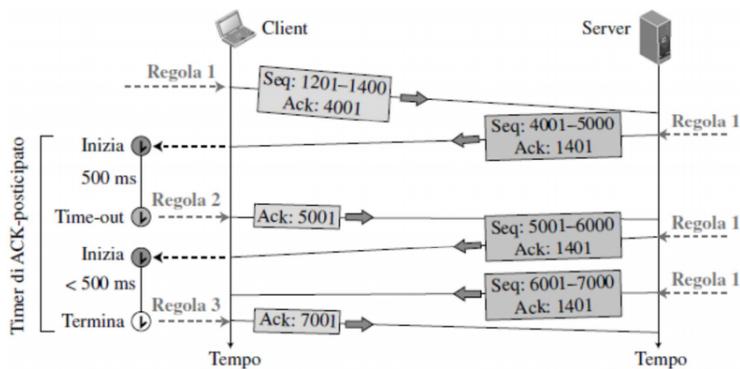


Figure 73: Uno scenario di operatività normale. Notare, come descritto in precedenza, il ritardo nella trasmissione dell'ACK dovuto alla ricezione di un solo segmento.

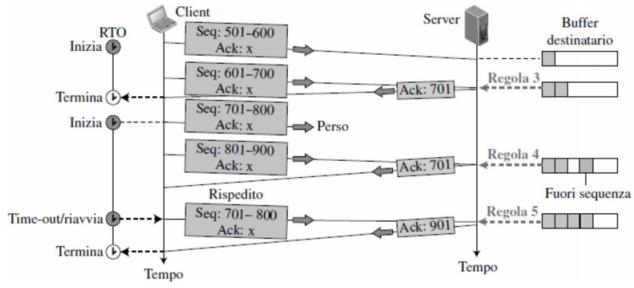


Figure 74: Uno scenario di perdita di un segmento. Notare, come descritto in precedenza, che alla ricezione del secondo segmento in ordine, alla ricezione del terzo segmento in *disordine* e alla ricezione del quarto segmento mancante viene immediatamente inviato un ACK.

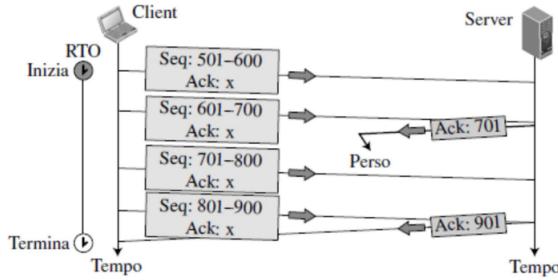


Figure 75: Uno scenario di perdita di un ACK. Notare come lo scenario venga ripristinato automaticamente in una condizione di normalità dalla spedizione dell'ACK successivo.

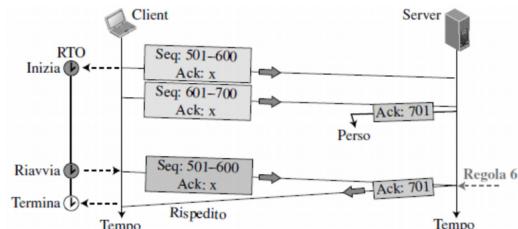


Figure 76: Uno scenario di perdita di un ACK. Stavolta si torna alla normalità tramite la ricezione di un segmento duplicato e il conseguente ed immediato invio dell'ACK da parte del server..

**N.B.** Lo smarrimento di un riscontro può provocare lo stallo ed è nel caso in cui, a seguito della ricezione da parte del TCP mittente di un segmento ACK con il campo **rwnd** settato a 0, si perda il successivo ACK con il campo rwnd non nullo. Il **TCP destinatario** supporrà che il TCP mittente lo abbia ricevuto e resterà in attesa dei dati.

Riassumiamo gli argomenti fin qui trattati presentando degli **ASF** per il **TCP mittente** e il **TCP destinatario**.

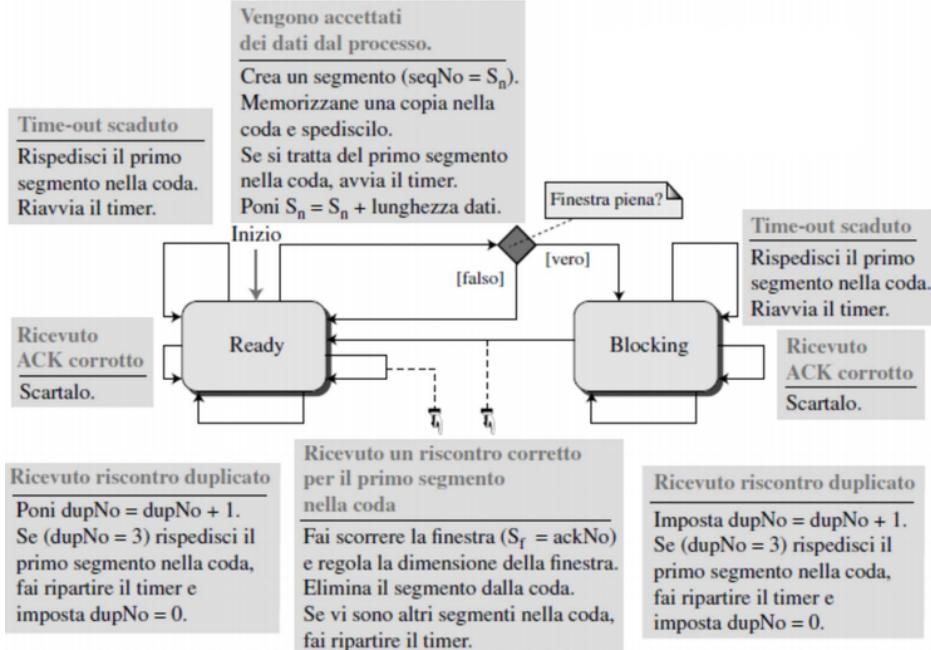


Figure 77: ASF del TCP mittente.

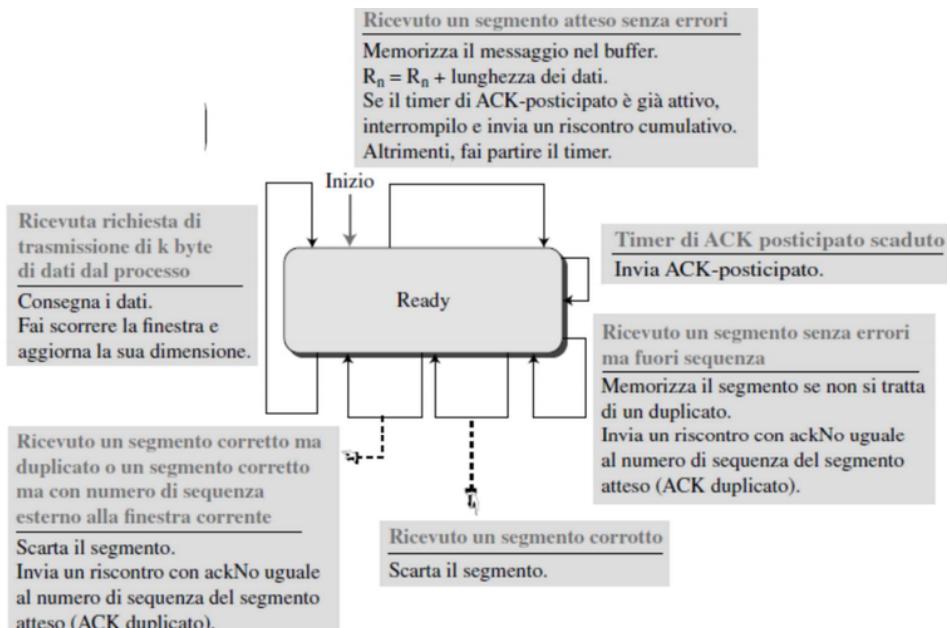


Figure 78: ASF del TCP destinatario.

*Come si calcola l'RTO o Retransmission Time-Out?*

**TCP** fa partire un **timer** quando invia un segmento all'**inizio** della coda di invio, quando il **timer scade** TCP **rispedisce il primo** segmento all'inizio della coda e fa ripartire il timer. Per definire l'**RTO** bisogna prima definire l'**RTT** o **Round Trip Timer**. Infatti l'**RTO** deve essere **maggior**e del'**RTT**. Ricordiamo che l'**RTT** è il **tempo trascorso da quando si invia un segmento a quando ne si riceve il riscontro**. Calcolare l'**RTT** è un procedimento abbastanza complicato a causa della sua **variabilità** nel percorso.

**Inizialmente** l'**RTT** vale 0 e l'**RTO** è **preimpostato**. Dopo la prima misurazione, ottenuta inviando un segmento e contando il tempo che passa tra l'invio e il riscontro, avremo un *RTT<sub>ESTIMATED</sub>*.

**N.B.** non si ha **nessuna** garanzia che il prossimo segmento impiegherà lo **stesso** tempo per essere inviato e riscontrato!

Perciò dalla **seconda misurazione** in avanti RTT viene calcolato utilizzando l'**RTT** dei segmenti precedenti con la seguente formula:

$$RTT_{ESTIMATED} = (1-\alpha) \times RTT_{ESTIMATED} + \alpha \times RTT_{SAMPLE} \quad (1)$$

L' **RTT<sub>SAMPLE</sub>** è riferito all'**ultimo** segmento inviato. **RTT<sub>ESTIMATED</sub>** è quindi la **combinazione dei suoi precedenti valori e il nuovo valore RTT<sub>SAMPLE</sub>**. Il valore di  $\alpha$  viene posto a 1/8 in modo da rendere via via meno importanti gli RTT dei pacchetti più vecchi. Si ha quindi:

$$RTT_{ESTIMATED} = 0,875 \times RTT_{ESTIMATED} + 0,125 \times RTT_{SAMPLE} \quad (2)$$

Oltre al valore RTT stimato è necessaria anche una stima della sua **variabilità** data dalla seguente formula:

$$RTT_{DEV} = (1-b) \times RTT_{DEV} + b \times | RTT_{SAMPLE} - RTT_{ESTIMATED} | \quad (3)$$

$$\text{Inizialmente } RTT_{DEV} = \frac{RTT_{sample}}{2}.$$

**RTT<sub>DEV</sub>** rappresenta una **stima di quanto RTT<sub>SAMPLE</sub> si discosta da RTT<sub>ESTIMATED</sub>**. Il valore di  $b$  viene posto a 1/4. Una volta ottenuti questi valori, il **timeout viene normalmente calcolato come**:

$$RTO = RTT_{ESTIMATED} + 4 \times RTT_{DEV} \quad (4)$$

In altre parole si prende il valore più recente di **RTT<sub>ESTIMATED</sub>** e gli si somma **quattro volte RTT<sub>DEV</sub>**, solitamente piccolo, per **equilibrarlo**.

*Come fa TCP a gestire la congestione?*

Ricordiamo che il fenomeno della **congestione** è originato dal tentativo delle sorgenti di **richiedere più banda di quella disponibile** sul percorso fino alle destinazione.

Il **traffico eccessivo** nella rete **può provocare**:

- **Lunghi ritardi** causati dall'accodamento dei pacchetti nei buffer dei router.
- **Perdita di pacchetti** causata dall'overflow nei buffer dei router.

Il **protocollo TCP** utilizza la **finestra di congestione** come principale strategia per evitare la congestione.

**TCP** usa le **finestre di ricezione** e le **finestre di trasmissione** per far sì che i due rispettivi buffer non vengano sovraccaricati. Fondamentalmente però la congestione è un **problema del livello di rete** e sebbene il protocollo IP non lo gestisca, la soluzione delle due finestre assicura l'assenza di congestione **agli estremi** della comunicazione ma **non nel mezzo**. La politica di rispedizione dei segmenti persi di TCP in combinazione con la congestione di rete potrebbe portare al **collasso dell'intera rete**. TCP non può quindi ignorare la congestione della rete ma non può nemmeno essere troppo conservativo più di quello che è già nell'inoltrare i segmenti. Si utilizza perciò una **seconda variabile** chiamata **cwnd** o **congestion window** il cui valore dipende dal **livello di congestione della rete**. Con l'aggiunta di questa seconda variabile il valore della finestra d'invio è determinato anche dalla congestione presente in rete.

$$\text{Dimensione della finestra} = \min(\text{rwnd}, \text{cwnd})$$

*Ma come si determina il valore di cwnd?*

Prima di rispondere a questa domanda è bene spiegare **come fa il protocollo TCP ad accorgersi della presenza di congestione sulla rete**. Il **TCP mittente** interpreta come sinonimi di congestione:

1. Il **timeout**.
2. La ricezione di **tre riscontri duplicati**.

Il secondo caso è **probabilmente meno critico del primo** poiché significa che un segmento è stato smarrito ma tre sono stati ricevuti. Può indicare o che la **rete è al limite della congestione o vi che è appena uscita**. L'**assenza** di una ricezione rapida e regolare dei riscontri è invece **evidenziata** dalla scadenza del **timeout**.

La strategia principale usata da TCP per controllare la congestione prevede **tre fasi**:

1. **Slow Start**.
2. **Congestion Avoidance**.
3. **Fast Recovery**.

A ogni fase corrisponde un algoritmo diverso.

- **Slow Start:** si basa sull'inizializzare la dimensione della finestra di congestione rendendola pari a quella della **MSS** o **Maximum Segment Size**. Aumentandola poi di MSS ogni volta che **un** segmento viene riscontrato. La MSS è stabilita all'inizio della connessione. Così facendo l'algoritmo parte lentamente ma procede con velocità via via crescente. La dimensione di cwnd nella fase di **slow start** può essere espressa come  $2^n$  dove n sono i **riscontri ricevuti**. Se **due segmenti vengono confermati con lo stesso riscontro cwnd aumenta solo di 1**. La crescita è comunque **esponenziale** nell'unità di tempo RTT, ma di esponente diverso. La fase continua **fino** al raggiungimento da parte di cwnd di una soglia chiamata **ssthresh** o **slow start threshold**. Al raggiungimento di questa soglia si entra nella fase di **congestion avoidance**.
- **Congestion Avoidance:** una volta che **cwnd** ha raggiunto una dimensione **relativamente grande**, per evitare che la crescita esponenziale finisca per causare problemi di congestione, il protocollo TCP usa un altro algoritmo che incrementa in modo **lineare** anziché **esponenziale** il valore di cwnd. La fase di **congestion avoidance** prevede che ogni volta che **l'intera finestra di ricezione viene riscontrata** la sua dimensione aumenti **di un'unità**.

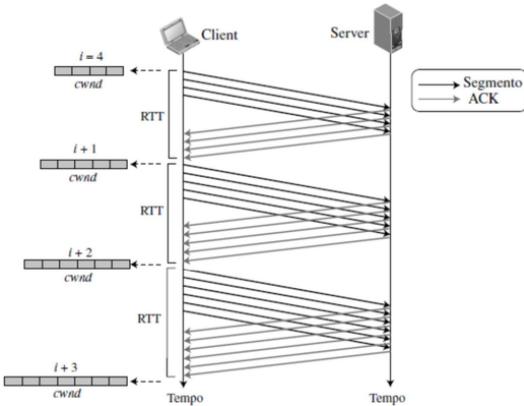


Figure 79: In figura la crescita della cwnd nella fase di congestion avoidance.

- La terza fase chiamata **Fast Recovery** è **opzionale** in TCP. La **versione originale non ne faceva uso**. Inizia quando arrivano **tre riscontri duplicati** interpretati come inizio di leggera congestione della rete. Aumenta la dimensione di cwnd in modo **lineare**, incrementandola di un'unità ogni volta che riceve **due riscontri duplicati**.

Analizziamo ora come, attraverso l'affermarsi di modelli sempre più recenti del protocollo TCP siano **cambiate le modalità di transizione da una fase all'altra**. La **prima versione** di TCP chiamata **TCP Tahoe** utilizzava **solo le prime due fasi**. Se durante la fase di **slow start** rileva la presenza di congestione allora **resetta** la dimensione di **cwnd** a 1 e **assegna** a **ssthresh** il vecchio valore di **cwnd dimezzato**. Se non la rileva entro il raggiungimento di **ssthresh** passa alla fase di **congestion avoidance**. In questa fase **non vi è un limite sulla dimensione di rwnd** e se non è rilevata congestione **aumenta linearmente fino alla chiusura della connessione**, se invece viene rilevata si riparte dalla fase di slow start con **cwnd pari a 1** e **ssthresh pari al vecchio valore di cwnd dimezzato**.

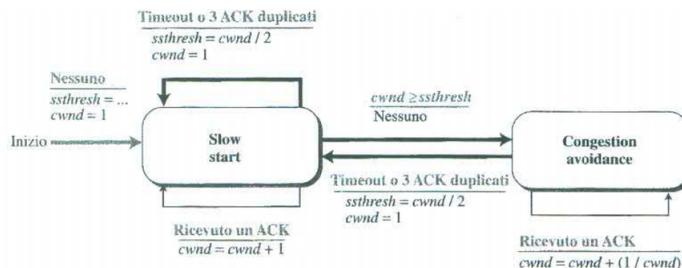


Figure 80: TCP Tahoe rappresentato come ASF.

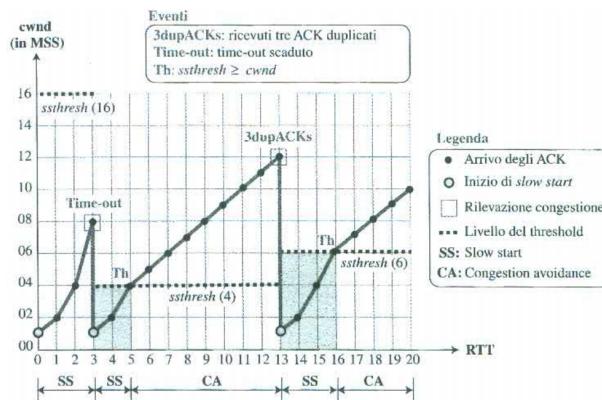


Figure 81: In figura l'evoluzione della dimensione di cwnd con TCP Tahoe.

In una versione più recente del protocollo TCP chiamata **TCP Reno**, è prevista anche la terza fase di **slow recovery**. TCP Reno inoltre **tratta i due sintomi della congestione in modo differente**. La mancata ricezione di un riscontro e il conseguente **timeout** portano TCP Reno a tornare nella fase di **slow start**, mentre la ricezione di **tre ACK duplicati**, sia che avvenga in **slowstart che in congestion avoidance**, portano TCP Reno ad **assegnare a ssthresh il valore di cwnd dimezzato**, a **cwnd il nuovo valore di sshtresh aumentato di tre unità** e a spostarsi nella fase di **fast recovery**, che può essere visto come uno **stato intermedio tra slow start e congestion avoidance**. In **fast recovery** **cwnd continua a crescere esponenzialmente**, se arriva un **ACK non duplicato** si giunge nello stato di **congestion avoidance** se invece giunge un altro **ACK duplicato** rimane in **fast recovery** e si **aumenta la cwnd di un'unità**, infine se scatta un timeout si torna in **slow start** con le modalità descritte per TCP Tahoe.

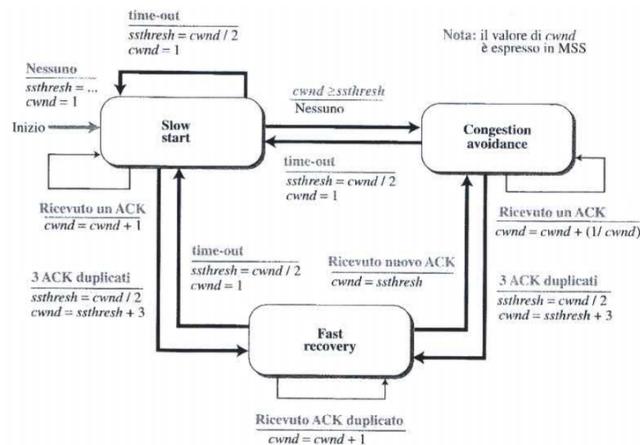


Figure 82: TCP Reno rappresentato come ASF.

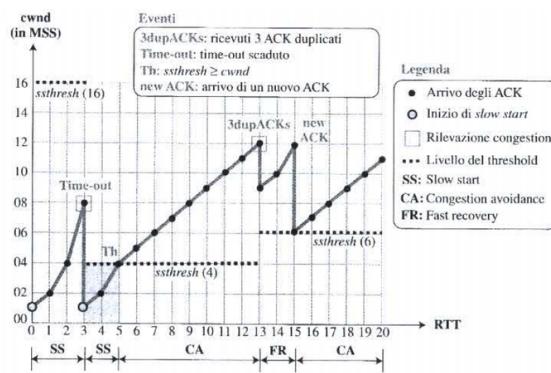


Figure 83: In figura l'evoluzione della dimensione di cwnd con TCP Reno.

Come calcolare il throughput di TCP?

Il throughput di TCP dipende dal comportamento della finestra di congestione e non è quindi costante, se riportato su un grafico avrebbe un profilo simile a quello dei denti di una sega detto anche **AIMD** o **addictive increase multiplicative decrease**. Ciò è dovuto a come è determinata, dopo la prima fase di **slow start**, la dimensione della finestra di congestione. Se i denti fossero tutti uguali il throughput sarebbe uguale a:

$$\frac{\frac{|max+min|}{2}}{RTT}$$

Sapendo che ogni rilevazione di congestione cwnd è impostato alla metà del suo valore precedente, **il throughput può essere calcolato come:**

$$\frac{0.75 \times W_{max}}{RTT}$$

Dove  $W_{MAX}$  è la dimensione media di cwnd in presenza di congestione.

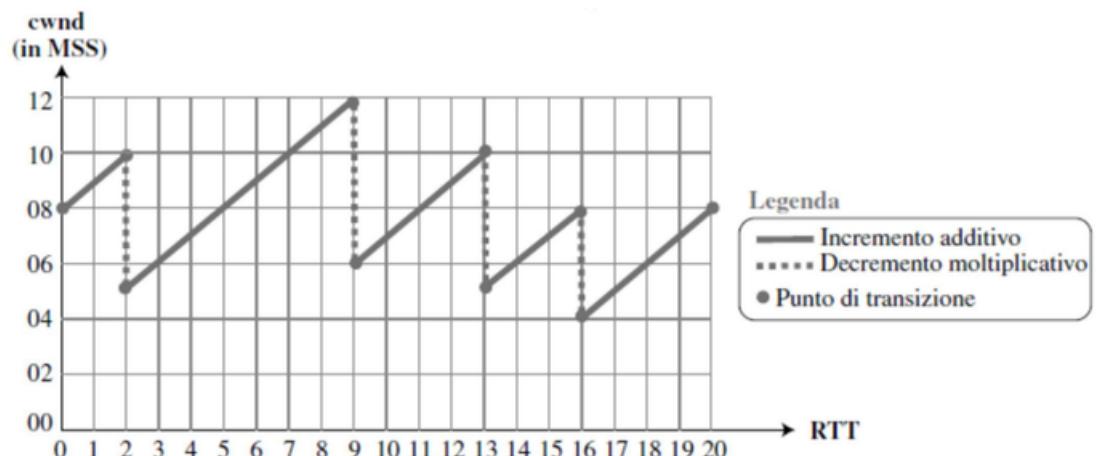


Figure 84: In figura il profilo AIMD.

Proviamo a calcolare il throughput avendo come dati il grafico, l' MSS pari a 10 KB (kilobyte) e un RTT di 100 ms. Otteniamo

$$W_{MAX} = \frac{10 + 12 + 10 + 8 + 8}{5} = 9.6 \text{ MSS}$$

da cui

$$throughput = \frac{0,75 \times W_{MAX}}{RTT} = \frac{0,75 \times (96 \times 8) Kbit}{100 ms} = 5,76 Mbit/s$$

## 7 Esercizi

1. L'utente

**mickey@disney.com**

invia dal suo PC una email a

**donald@disney.com**

. Indicare la sequenza di comandi SMTP inviati e ricevuti dal PC di **mickey@disney.com** se:

- (a) Il mailserver di disney.com non è raggiungibile.
- (b) Il mailserver di disney.com è raggiungibile.

**Soluzione:**

- (a) Il PC di mickey@disney.com non riesce a stabilire una connessione TCP con il mailserver di disney.com, quindi **nessun** messaggio SMTP viene inviato o ricevuto.
- (b) Il PC di mickey@disney.com stabilisce una connessione TCP con il mailserver di disney.com su cui scambia i seguenti comandi SMTP:  
R: 220 service ready  
I: HELO ...  
R: 250 OK  
I: MAIL FROM: mickey@disney.com  
R: 250 OK  
I: RCPT TO: donnald@disney.com  
R: 250 OK  
I: DATA  
I: ...  
I: ...  
R: 250 OK  
I: QUIT  
R: 221 service closed

È bene ricordarsi che il corpo del messaggio termina **sempre** con ritorno a capo e fine linea.

Per chiarimenti è consigliato il ripasso del capitolo sull'**SMTP**.

2. Un host deve risolvere il nome simbolico:

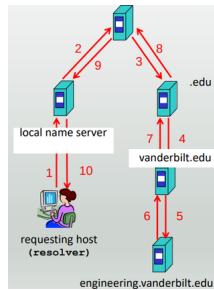
**host.engineering.vanderbilt.edu**

il cui indirizzo IP **non** è noto al suo resolver (i.e. servizio DNS dell'host). Supponendo che la gerarchia dei name server abbia **4** livelli, indicare, giustificando la risposta, il numero di messaggi DNS che, nel caso peggiore, circoleranno in Internet per risolvere tale nome simbolico, se:

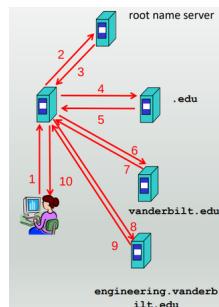
- (a) Si utilizza ad ogni livello una risoluzione ricorsiva.
- (b) Si utilizza ad ogni livello una risoluzione iterativa.

**Soluzione:**

- (a) Bisogna percorrere tutto l'albero dei name server, dal name server locale fino ad un root server, e poi da questo al name server autoritativo, che nel caso peggiore è il name server locale per engineering.vanderbilt.edu, per poi tornare indietro: in totale sono **4 + 4** messaggi a cui vanno aggiunti i 2 messaggi dal resolver al name server locale del client, e viceversa: quindi **10 messaggi**.



- (b) Anche ora i messaggi saranno **10**, perchè i name server coinvolti saranno 4, nel caso pessimo, più i 2 messaggi dal resolver al name server locale del client, e viceversa.



3. Un client C chiede la pagina web

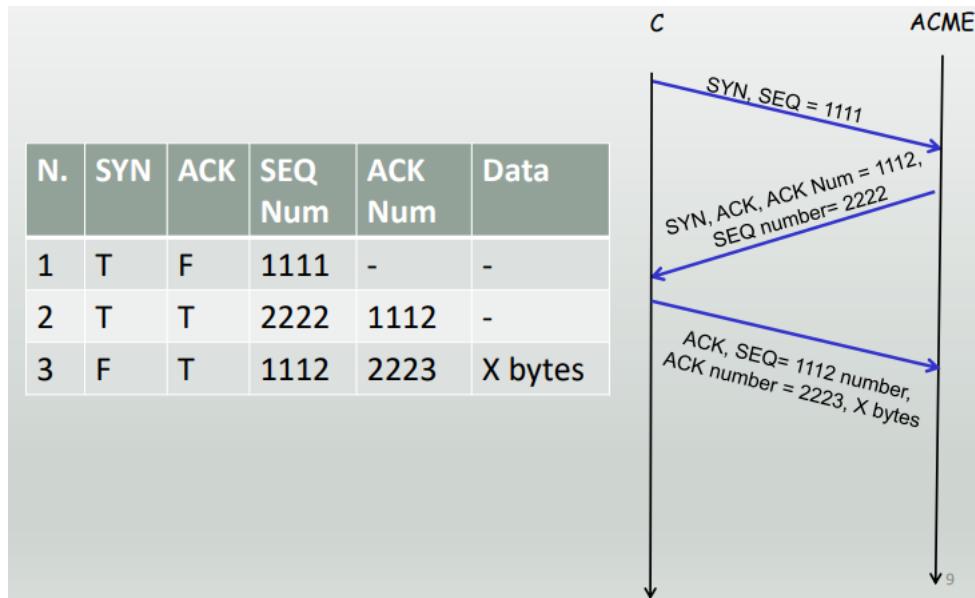
**www.acme.com/home/products.html**

al server B di

**www.acme.com**

con una GET che è contenuta in un segmento TCP il cui payload (campo Dati) è lungo X byte. Indicare, giustificando la risposta, i valori dei campi sequence number, ack number, e dei flags ACK ed SYN e lunghezza del campo DATA, in ciascuno dei segmenti che C e B si scambiano per aprire la connessione nell'ipotesi che l'ack finale dell'apertura della connessione sia inviato in piggybacking assieme alla GET. Si supponga che in B ed in C rwnd sia molto grande, che non scada alcun timeout, che non ci siano errori di trasmissione, che nessun segmento vada perduto, e che il numero di sequenza iniziale di C sia 1111 e quello di B sia 2222.

**Soluzione:**



Si ricorda che per **piggybacking** si indica la tecnica mediante la quale l'invio dell'ACK è unito all'invio dei dati. La tecnica migliora l'efficienza dei protocolli bidirezionali.

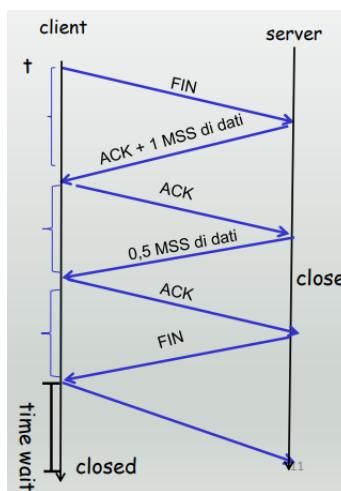
4. Un client C ha stabilito una connessione TCP con un server web S per scaricare una pagina web che consiste di tre oggetti. Al tempo  $t$ , subito dopo avere inviato la richiesta per il terzo oggetto, l'host di C invia a S un segmento con il flag FIN a true. Indicare, giustificando la risposta, il tempo minimo necessario al TCP di C per chiudere definitivamente la connessione supponendo che:

- La dimensione del terzo oggetto sia 1,5 MSS.
- RTT sia costantemente 700 msec e il maximum segment lifetime sia 1100 msec.
- Tutti i segmenti vengano ricevuti corretti ed in ordine, e che il valore di cwnd del TCP di S sia 1MSS quando esso riceve il FIN inviato dall'host di C. Trascurare i tempi di preparazione e di trasmissione dei segmenti e assumere che S abbia già a disposizione tutti i dati da inviare.

**Soluzione:**

$$3RTT + 2MSL = 4300\text{msec}$$

Il TCP di C riceve al tempo  $t + RTT$  il riscontro S1 del segmento FIN da lui inviato. Se S1 trasporta in piggybacking il primo MSS dei dati del terzo oggetto e il TCP di C invierà un riscontro C2 per tali dati, quindi S potrà inviare l'ultima porzione di dati e attenderne il riscontro, infine il TCP di S invierà un segmento con il flag FIN a true. Il TCP di C invierà un riscontro del FIN e considererà chiusa la connessione dopo avere atteso 2MSL, ovvero al tempo  $t + 3RTT + 2MSL = t + 4300$  msec.



5. Si descriva il meccanismo di controllo di flusso in TCP.

#### **Una possibile soluzione:**

Con controllo di flusso si intende la capacità del mittente di evitare la possibilità di saturare il buffer del ricevitore. Infatti, a livello TCP ogni host imposta un buffer di invio e uno di ricezione. Il processo applicativo destinatario legge i dati dal buffer di ricezione (non necessariamente nell'istante in cui arrivano). Il controllo di flusso ha lo scopo di regolare la frequenza di invio del mittente in base alla frequenza di lettura dell'applicazione ricevente allo scopo di non saturare il buffer del ricevente. TCP implementa questa funzione tramite una variabile detta **receive window** mantenuta nel mittente: questa variabile fornisce un'idea di quanto spazio è ancora a disposizione nel buffer del ricevitore. Tale valore è comunicato nel **campo window** dell'header TCP dall'host destinatario.

- Il valore di **receive window** è pari a:  
$$\text{rwnd} = \text{RcvBuffer} - (\text{LastByteReceived} - \text{LastByteRead})$$
- L'host destinatario comunica la dimensione di RcvWindow al mittente.
- Il mittente si assicura che  $\text{LastByteSent} - \text{LastByteAcked} < \text{rwnd}$  pari ovvero alla quantità di dati trasmessi e non ancora riscontrati.

Nelle situazioni in cui il buffer risulta pieno ( $\text{rwnd} = 0$ ), il mittente continua a mandare **segmenti sonda** di 1 byte per ricevere l'aggiornamento sulla dimensione di rwnd ed evitare lo stallo.

Per approfondire l'argomento recarsi al **capitolo sul TCP**.

6. Due host H1 e H2 comunicano tramite un canale che attraversa tre router R1, R2 e R3 e 4 link di capacità C1, C2, C3 e C4, rispettivamente come mostrato in figura. La comunicazione avviene tramite commutazione di pacchetto con trasmissione di tipo store and forward. Assumendo che il ritardo di propagazione sia trascurabile, che i ritardi di accodamento nei router R1, R2 e R3 sia rispettivamente a1, a2 e a3, e che il ritardo di elaborazione nei tre router sia uguale a 1 ms, dire quanto tempo è necessario per la trasmissione da H1 a H2 di un pacchetto di dimensione L nel seguente caso:

- $L = 10 \text{ KBytes}$ .
- $C1=C2=C3=C4 = 2 \text{ Mbps}$ .
- $a1=a2=a3 = 0,01 \text{ s}$ .



**Soluzione:** Il ritardo introdotto da ogni router è dato dalla somma tra: ritardo di trasmissione, ritardo di propagazione, ritardo di accodamento e tempo di elaborazione. In particolare:

- il tempo di trasmissione di H1 è pari a  $\frac{L}{C_1} = 40$  ms.
- il tempo di trasmissione di R1 è pari a  $\frac{L}{C_2} = 40$  ms.
- il tempo di trasmissione di R2 è pari a  $\frac{L}{C_3} = 40$  ms.
- il tempo di trasmissione di R3 è pari a  $\frac{L}{C_4} = 40$  ms.

Quindi il ritardo introdotto dai router è: ritardo introdotto dai router R1,R2,R3 =  $40 + 10 + 1 = 51$  ms.

Il ritardo dovuto alla trasmissione da H1 è pari a  $\frac{L}{R} = 40$  ms. Quindi il tempo complessivo per la trasmissione del pacchetto da H1 a H2 è:

$$51 \times 3 + 40 = 193 \text{ ms}$$

7. Tizio manda dal suo account di email

**tizio@libero.it**

un messaggio di posta elettronica con testo di 256 caratteri a Caio

**caio@occupato.it**

- Specificare il contenuto dei primi due messaggi TCP inviati dal mailserver di libero.it, ms.libero.it, per ricevere tale messaggio.
- Specificare, per ciascun segmento, payload, numero di sequenza, ack number, flags posti ad 1, porta origine e porta destinazione.

**Soluzione:**

- Il primo segmento avrà il payload vuoto e il payload del secondo conterrà 220 service ready.
- Il primo segmento sarà SYNACK, e quindi: come già detto, payload vuoto, flags SYN e ACK ad 1, numero sequenza Z, ack number Y, **porta mittente 25**, porta destinazione effimera. Il secondo segmento conterrà come payload 220 service ready, nessun flag ad 1, numero sequenza Z+1, ack number Y, porta mittente 25, porta destinazione effimera.

8. Dire in quali delle seguenti circostanze il TCP cambia la dimensione della finestra di congestione **cwnd**, e, nel caso, come viene ricalcolata.

Stato	Evento	Cambia la finestra?	Nuova dimensione della finestra
Slow Start	Ricezione di un ACK duplicato		
Slow Start	Scatta un timeout		
Congestion Avoidance	Scatta un timeout		

**Soluzione:**

Stato	Evento	Cambia la finestra?	Nuova dimensione della finestra
Slow Start	Ricezione di un ACK duplicato	NO	
Slow Start	Scatta un timeout	SI	CongWin = 1 MSS
Congestion Avoidance	Scatta un timeout	SI	CongWin = 1 MSS

9. Si consideri il seguente scenario TCP in cui, per semplicità, non sono indicati i segmenti inviati o re-inviati dal sender TCP al receiver TCP, che si suppone essere quelli necessari per avere i riscontri descritti di seguito.

- Al tempo  $t_0$  il TCP di un host A ha una connessione già stabilita, per la quale ha 4 segmenti full sized in volo (inviati ma non riscontrati) e nessun nuovo dato da spedire, il primo byte dei segmenti in volo è il byte Y,  $ssthresh = 6.5 \text{ MSS}$ ,  $cwnd = 5 \text{ MSS}$ . Inoltre, non ha ricevuto nessun riscontro duplicato.
- Tra il tempo  $t_0$  e il tempo  $t_1$  riceve 7 riscontri: i primi due non duplicati e con acknumber uguale a  $Y + 1 \text{ MSS}$  per il primo, e  $Y + 3 \text{ MSS}$  per il secondo. I seguenti 4 riscontri sono tutti duplicati, ed infine, al tempo  $t_1$ , riceve un settimo riscontro con acknumber uguale a  $Y + 4 \text{ MSS}$ .

Si supponga che non scatti alcun timeout tra  $t_0$  e  $t_1$ . Indicare, per ciascun riscontro ricevuto, lo stato del TCP e i valori di  $ssthresh$  e  $cwnd$ , giustificando la risposta.

**Soluzione:**

n.1	Riscontro	Cong Window	Threshold	Stato
1	Y+1 MSS	5+1 MSS	6,5 MSS	SS
2	Y+3 MSS	(5+1)+1 MSS*	6,5 MSS	CA
3-4	Y+3	7 MSS	6,5 MSS	CA
5	Y+3	3,5+3 MSS	3,5 MSS	FR
6	Y+3	7,5 MSS	3,5 MSS	FR
7	Y+4	3,5 MSS	3,5 MSS	CA

10. Descrivere in modo dettagliato e mediante uno **pseudocodice** le azioni svolte da un **destinatario TCP** per realizzare il controllo di flusso. Si assuma che ogni segmento ricevuto dal destinatario contenga anche lo pseudoheader (lunghezza segmento TCP). Inoltre, si supponga che la chiusura della connessione venga fatta dal mittente e che i segmenti di chiusura non contengano dati in piggybacking. Non occorre realizzare la parte iniziale delle azioni svolte (quindi le inizializzazioni delle variabili e l'apertura della connessione). Infine, si supponga che si invii un riscontro appropriato per ogni segmento ricevuto, e che il destinatario non debba inviare dati al mittente. Inoltre, per semplicità, si specifichino solamente i campi dell'header del riscontro relativi al controllo del flusso e al riscontro. Non occorre realizzare le interazioni con il livello applicativo (processo che legge dal buffer), ma solamente quelle con il mittente TCP. Si hanno a disposizione le seguenti procedure:

```

receive(segm) //per ricevere il segmento segm dal livello di rete
OK(segm) //restituisce true solo se segm è corretto
nuovo(segm.x) //vettore di booleani che vale true se dal campo x di
segm si deduce che i dati contenuti in segm non sono doppioni
(specificare x nella soluzione)
insert(segm.y,finestra) //per inserire segm.y nel buffer di
ricezione nella posizione corretta
calcolacknum(segm) //restituisce il numero di riscontro per il
riscontro associato a segm
send(risp) //per inviare risp al mittente

```

Descrivere il contenuto o la funzionalità delle variabili e delle altre funzioni o procedure eventualmente utilizzate.

**Soluzione:**

```
finito = false;
while (!finito) {
    receive(segm);
    finito = segm.FIN;
    #controllare inizio della chiusura della conn.
    if (!finito) {
        #se segmento non corrotto
        if (OK(segm)) {
            #se il segmento contiene nuovi dati
            if (nuovo(segm.seqnum)) {
                #inserisco i dati nel buffer
                insert(segm.dati, finestra);
                #nuovo valore di rwnd
                rwnd = rwnd - (segm.lungtotTCP - segm.HLEN);
                #flag ack settato a true
                risposta.ACK = true;
                risposta.ACKnum = calcolacknum(segm);
                risposta.rwnd = rwnd;
            }
            #invia o il nuovo ack o ultimo ack inviato
            send(risposta);
        }
    }
}
```

11. Discutere l'affermazione: *poiché FTP e HTTP sono protocolli adatti a trasferire file, possono essere usati indifferentemente.*

**Soluzione possibile:**

Esempio di risposta (schema per punti):

- Descrivere brevemente obiettivo HTTP e FTP.
- Entrambi usano TCP (trasferimento affidabile dei dati), elencare le differenze ai fini del trasferimento file.

**e.g. FTP**

- connessione controllo, persistente, inizializzata dal client, porta 21, comandi in formato ASCII a 7 bit
- connessione dati, inizializzata da server, porta 20, non persistente
- FTP è Stateful
- Offre funzionalità aggiuntive rispetto a HTTP per la gestione di file e directory (list, retr, put).

**e.g. HTTP**

- Un'unica connessione per dati e comandi.
- Interazione stateless.