



# Il socket e la comunicazione con i protocolli TCP/UDP



# 01

## Le applicazioni di rete

# Il modello ISO/OSI e le applicazioni

Nel modello ISO/OSI e TCP il livello delle applicazioni si occupa di implementare le applicazioni di rete che vengono utilizzate dall'utente finale, in cui il programmatore non si deve preoccupare dei livelli inferiori ma soltanto utilizzare le primitive di comunicazione messe a disposizione dai diversi protocolli degli altri strati: a tale livello possiamo individuare Internet come la struttura più complessa esistente.

Il modello a pila(stack) ISO/OSI può essere messo in relazione con altri stack protocollari, in modo da poterli comparare: la pila TCP/IP, formata da 4 livelli messa a confronto con la pila ISO/OSI.

Modello OSI		Modello TCP/IP
7 Applicazione	DHCP, DNS, FTP, HTTP, HTTPS, POP, SMTP, SSH, DNS, POP3, SNMP, etc...	Applicazione
6 Presentazione		
5 Sessione		
4 Trasporto	TCP UDP	Trasporto
3 Rete	IP Address: IPv4, IPv6	Internet
2 Collegamento	MAC Address	Rete fisica
1 Fisico	Ethernet cable, fibre, wireless, coax, etc...	

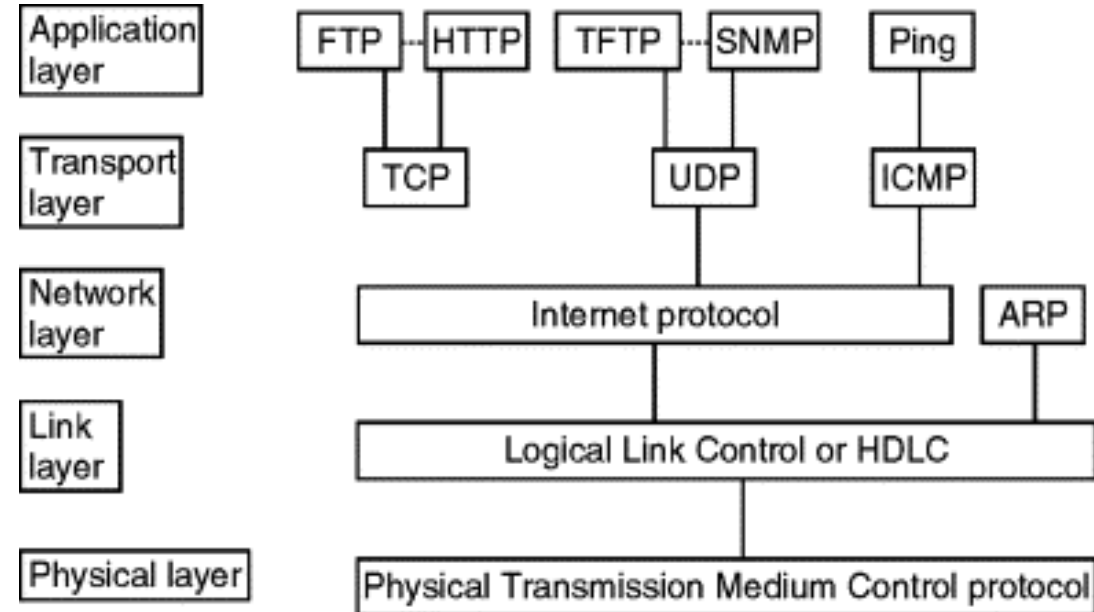
# Il modello ISO/OSI e le applicazioni

Il livello applicazione implementa i vari protocolli, tra cui:

- SNMP: Simple Network Management Protocol;
- SMTP: Simple Mail Transfer Protocol;
- POP3: Post Office Protocol;
- FTP: File Transfer Protocol;
- HTTP: HyperText Transfer Protocol;
- DNS: Domain Name System.

Questi protocolli vengono utilizzati da tutte le applicazioni di rete generali.

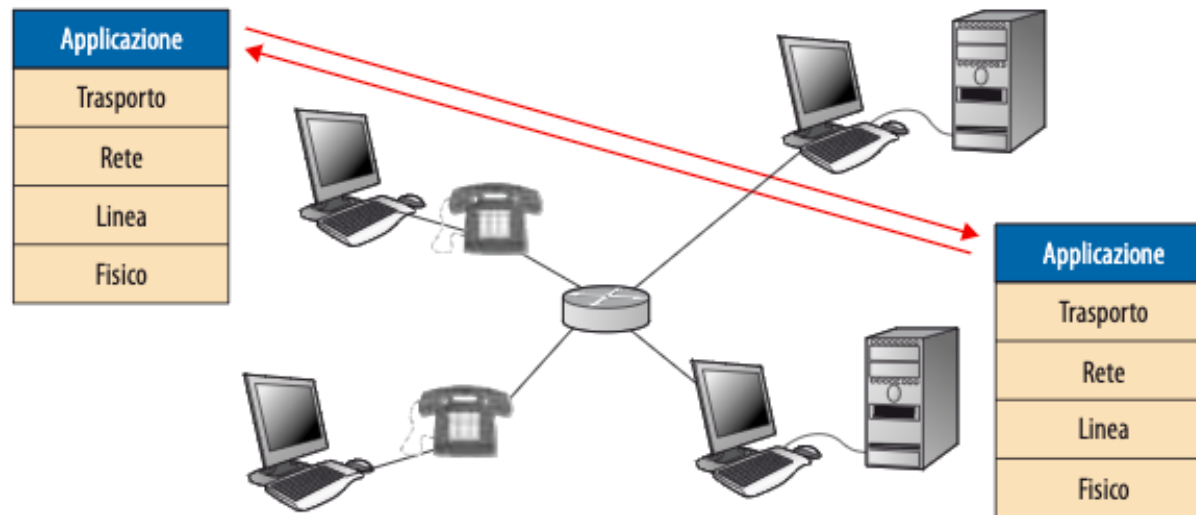
Sulla rete “girano” numerose **applicazioni proprietarie**, cioè sviluppate all'interno di un'organizzazione per la gestione privata, come per esempio il collegamento tra filiali remote di una società commerciale, la connessione in tempo reale degli agenti col magazzino centrale ecc.



# Applicazioni di rete

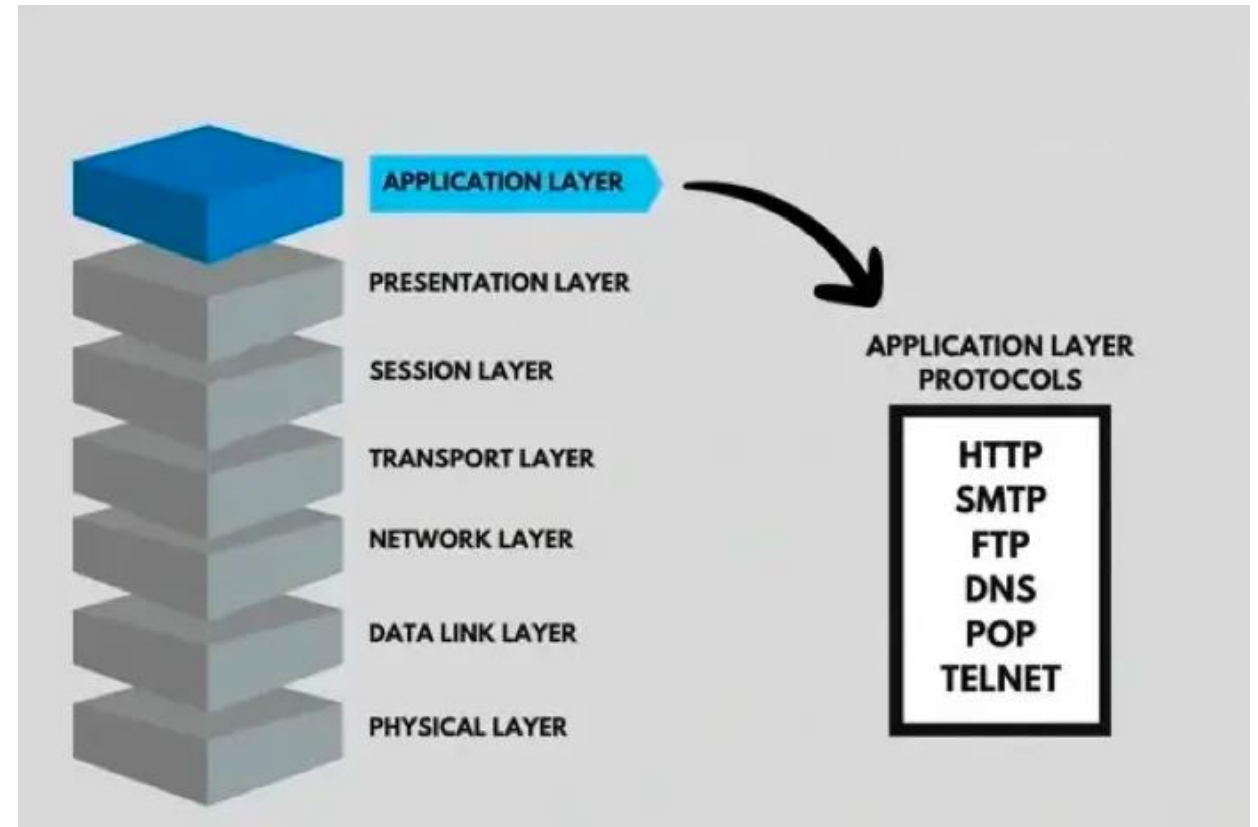
In generale un'applicazione di rete è costituita da **un insieme di programmi** che vengono **eseguiti su due o più computer contemporaneamente**: questi operano interagendo tra loro utilizzando delle risorse comuni, accedendo cioè concorrentemente agli archivi (database), mediante la rete di comunicazione che li connette.

L'applicazione di rete prende anche il nome di **applicazione distribuita** dato che non viene eseguita su di un solo elaboratore (concentrata).



# Applicazioni di rete

Il processi hanno la necessità di scambiare messaggi con gli altri processi della medesima applicazione, sia che essi appartengano alla stessa rete locale oppure che siano remoti e quindi dislocati dall'altra parte del globo: per comunicare tra loro questi processi devono mettersi in “contatto” tramite i loro indirizzi e utilizzando i servizi offerti dal livello di applicazione.

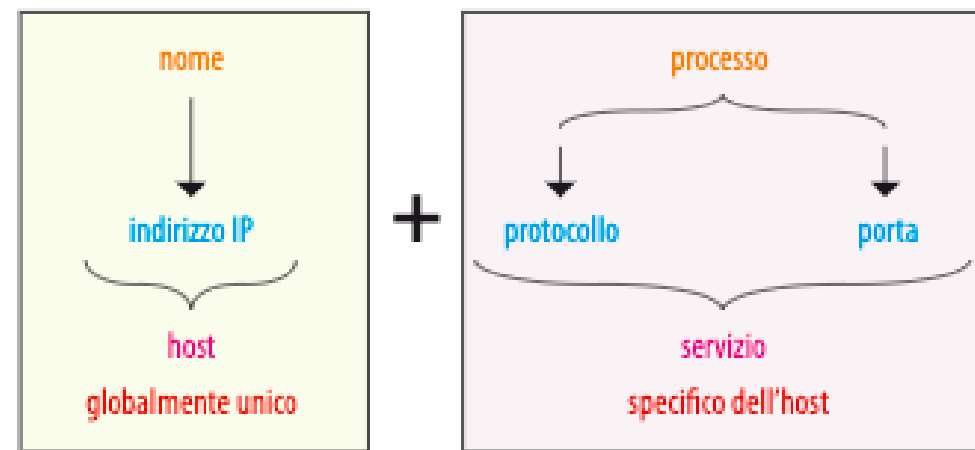


# Identificazione di un servizio mediante socket

Affinché un processo, presente su un determinato host, invii un messaggio a un qualsiasi altro host, il processo mittente deve identificare il processo destinatario in modo univoco.

L'identificazione non può avvenire soltanto mediante l'indirizzo IP del destinatario, in quanto quest'ultimo individua soltanto l'host ma non il processo specifico e su un host sono sicuramente in esecuzione più processi contemporaneamente.

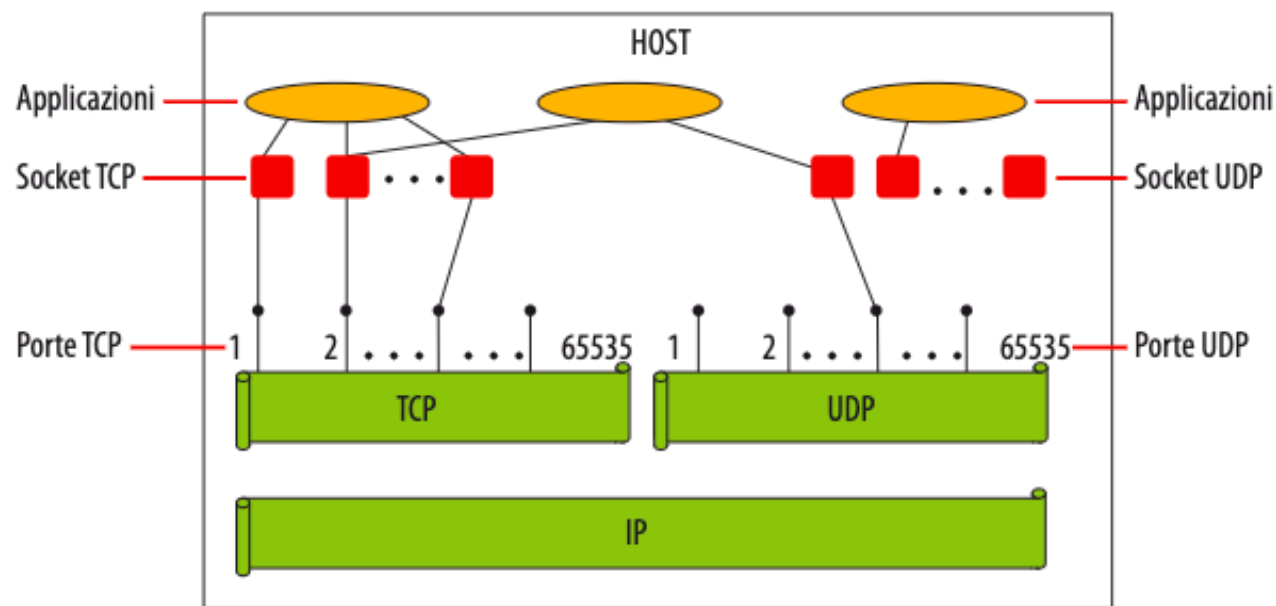
L'identificazione deve tenere conto di due informazioni, l'indirizzo IP e il processo appartenente a quel determinato host: per individuare il processo a ogni servizio offerto da un server è assegnato un identificatore unico per l'host, chiamato numero di porta, il server si registra con il software del protocollo di trasporto usando il numero di porta (bind) e il client lo contatta usando identificatore dell'host (indirizzo IP) + l'identificatore del servizio (numero di porta).



# Identificazione di un servizio mediante socket

Per richiedere il servizio a questo server è quindi necessario specificare l'indirizzo dell'host e il tipo di servizio desiderato, quindi: gli identificatori del server devono essere noti a priori al client per potersi connettere.

L'identificazione univoca avviene conoscendo sia l'indirizzo IP che il numero di porta associato al processo in esecuzione su un host: questo meccanismo prende il nome di meccanismo dei socket.

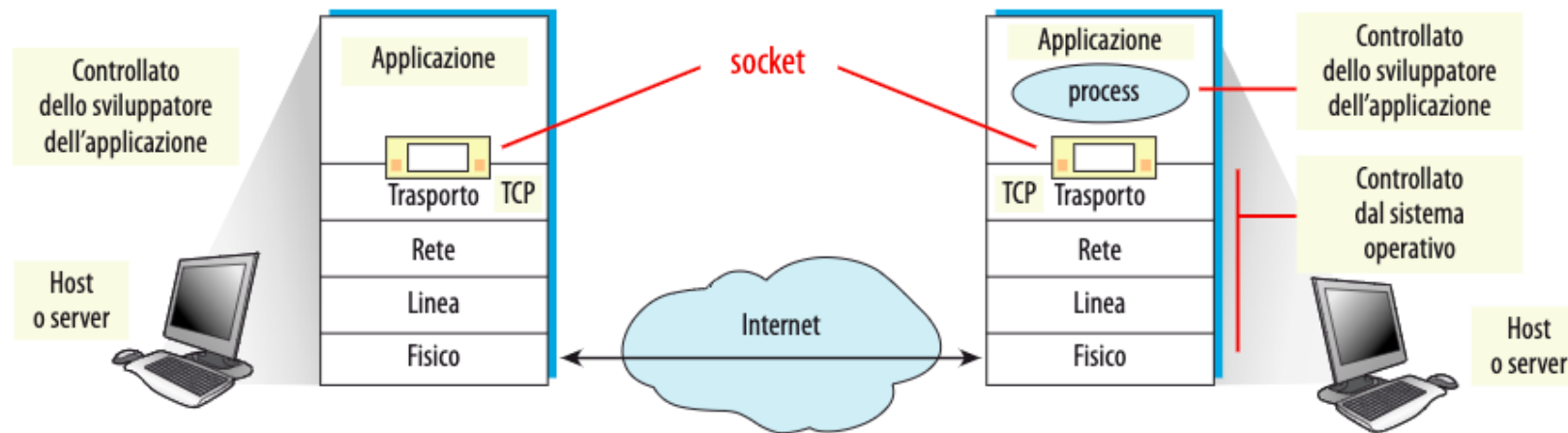




# Identificazione di un servizio mediante socket

Un socket è formato dalla coppia <indirizzo IP:numero della porta>; si tratta di un identificatore analogo a una porta, cioè a un punto di accesso/uscita; un processo che vuole inviare un messaggio lo fa uscire dalla propria “interfaccia” (socket del mittente) sapendo che un’infrastruttura esterna lo trasporterà attraverso la rete fino all’“interfaccia” del processo di destinazione (socket del destinatario).

Un socket consente di comunicare attraverso la rete utilizzando la pila TCP/IP ed è quindi parte integrante del protocollo: le API mettono a disposizione del programmatore gli strumenti necessari a codificare la connessione e l’utilizzo del protocollo di comunicazione.



# Identificazione di un servizio mediante socket

L'applicazione di rete può essere vista come composta da due parti:

1. una user agent, che funge da interfaccia tra l'utilizzatore dell'applicazione e gli aspetti comunicativi;
2. l'implementazione dei protocolli che permettono all'applicazione di integrarsi con la rete.

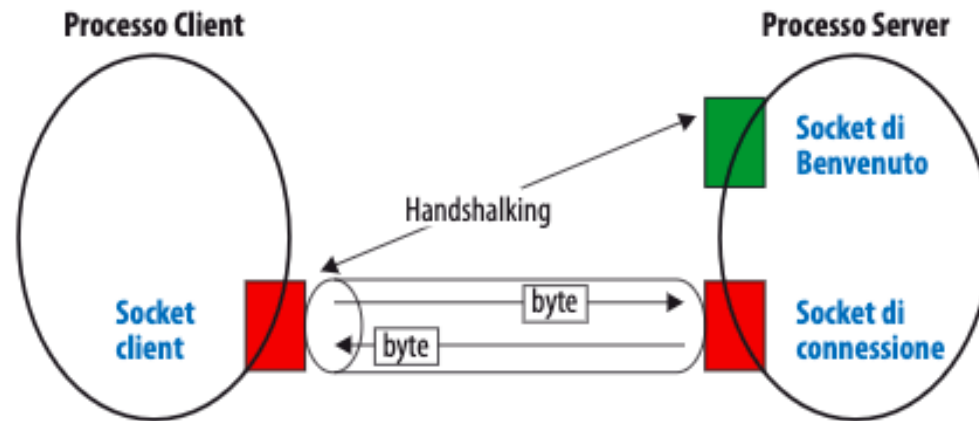


In un **browser Web** possiamo individuare queste due componenti:

1. l'interfaccia utente che serve a visualizzare i documenti ricevuti dai client, a permettere la loro navigazione e a richiedere nuovi documenti specificando la loro URL;
2. il motore del browser che è la parte che si preoccupa di inviare le richieste ai vari server e di ricevere le risposte.

# Identificazione di un servizio mediante socket

Il server offre il servizio a più client e, quindi, deve gestire la concorrenza: il socket di connessione è anche chiamato socket di benvenuto e alla richiesta accettata di un client il server crea dinamicamente un nuovo thread e gli assegna un socket di connessione.

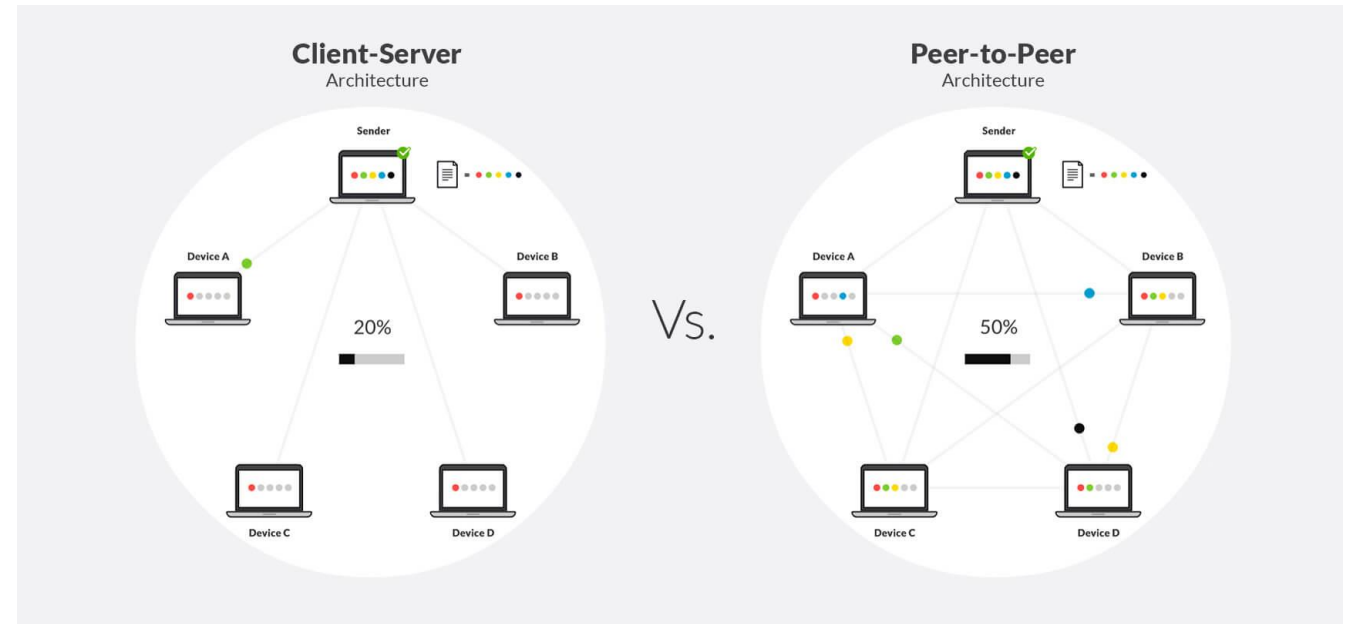


Il thread concorrente del server, con il suo flusso di controllo autonomo operante su memoria condivisa, soddisfa la richiesta del client attraverso la socket di connessione e, contemporaneamente, si rende disponibile ad attendere sul socket di benvenuto le successive richieste di altri client.

# Scelta dell'architettura per l'applicazione di rete

Il primo passo che il programmatore deve effettuare per progettare una applicazione di rete è **la scelta della architettura dell'applicazione**:

- client-server;
- peer-to-peer (P2P);
- architetture ibride (dove convivono client-server e P2P).



# Scelta dell'architettura per l'applicazione di rete

## Architettura client-server

Nella architettura client-server la caratteristica principale è che **deve** sempre **esserci un server attivo** che offre un servizio, restando in attesa che uno o più client si connettano a esso per poter rispondere alle richieste che gli vengono effettuate.

Il server deve sempre essere attivo e deve possedere un indirizzo IP fisso dove può essere raggiunto dagli host client: quindi l'indirizzo IP deve essere statico, contrariamente a quello dei client che generalmente è dinamico.

Un client non è in grado di comunicare con gli altri client ma solo con il server: più client possono invece comunicare contemporaneamente con lo stesso server.

Se un server viene consultato contemporaneamente da molti client potrebbe non essere in grado di soddisfare tutte le richieste e potrebbe entrare in stato di congestione: è necessario virtualizzare la risorsa realizzando una server farm. Questa non è altro che un server con un unico hostname ma con più indirizzi IP, trasparenti rispetto al client, sui quali vengono dirottate le richieste di connessione.



# Scelta dell'architettura per l'applicazione di rete

## Architettura peer-to-peer (P2P)

Nelle architetture peer-to-peer (P2P) abbiamo coppie di host chiamati peer (letteralmente “persona di pari grado, coetaneo”) che dialogano direttamente tra loro.

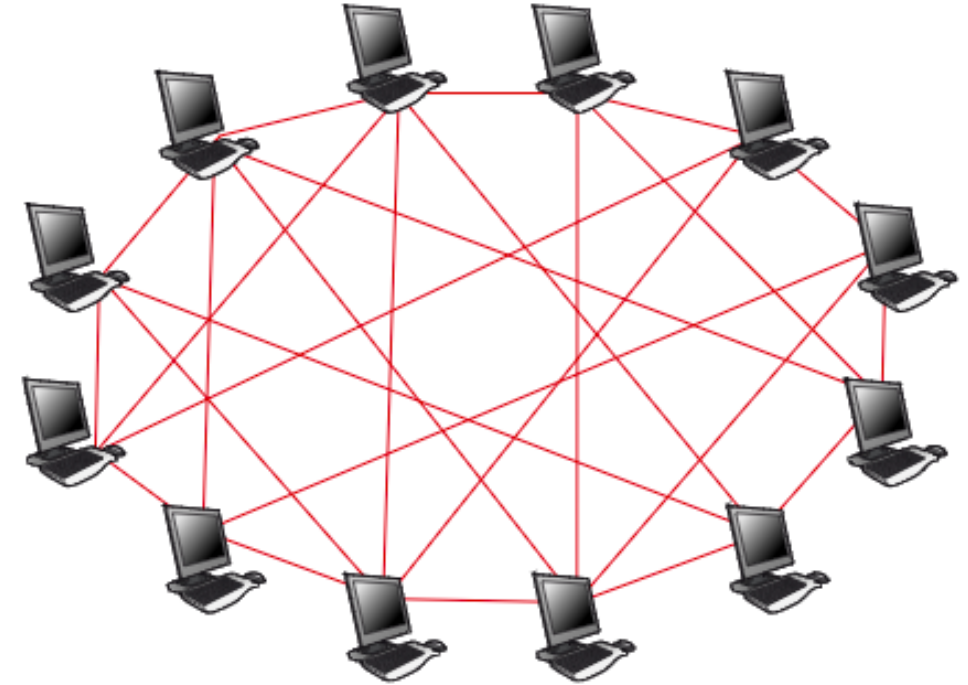
Un sistema P2P è formato da un insieme di entità autonome (peers), capaci di auto organizzarsi, che condividono un insieme di risorse distribuite presenti all'interno di una rete. Il sistema utilizza tali risorse per fornire una determinata funzionalità in modo completamente o parzialmente decentralizzato.

- P2P decentralizzato
- P2P centralizzato
- P2P ibrido (o parzialmente centralizzato)

# P2P decentralizzato

Nella architettura completamente decentralizzata un peer ha sia funzionalità di client che di server (hanno funzionalità simmetrica e sono anche chiamati servent), ed è impossibile localizzare una risorsa mediante un indirizzo IP statico: vengono effettuati nuovi meccanismi di indirizzamento, definiti a livello superiore rispetto al livello IP.

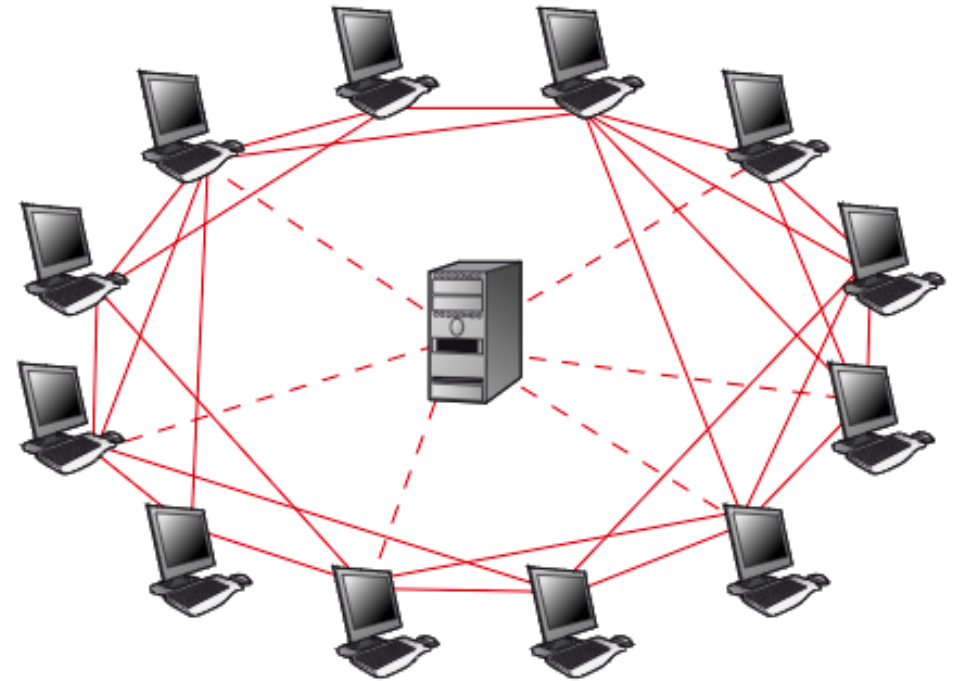
Le risorse che i peer condividono sono i dati, la memoria, la banda ecc.: il sistema P2P è capace di adattarsi a un continuo cambiamento dei nodi partecipanti (churn) mantenendo connettività e prestazioni accettabili senza richiedere l'intervento di alcuna entità centralizzata (come un server).



# P2P centralizzato

Il P2P centralizzato è un compromesso tra il determinismo del modello client-server e la scalabilità del sistema puro: ha un server centrale (directory server) che conserva informazioni sui peer (index, cioè il mapping risorse-peer) e risponde alle richieste su quelle informazioni effettuando quindi la ricerca in modalità centralizzata.

I peer sono responsabili di conservare i dati e le informazioni (il server centrale non memorizza file), di informare il server del contenuto dei file che intendono condividere e di permettere ai peer che lo richiedono di scaricare le risorse condivise.





# P2P ibrido (o parzialmente centralizzato)

Il P2P ibrido è un P2P parzialmente centralizzato dove sono presenti alcuni peer (detti supernodi o super-peer o ultra-peer) determinati dinamicamente (tramite un algoritmo di elezione) che hanno anche la funzione di indicizzazione: gli altri nodi sono anche chiamati leaf peer.



# Servizi offerti dallo strato di trasporto alle applicazioni

Le applicazioni richiedono allo strato di trasporto un insieme di servizi specifici oltre ai protocolli necessari per realizzarli, che possono essere standard o realizzati ad hoc. Tutti i protocolli, sia standard che specifici, hanno in comune una particolarità: trasferire dei messaggi da un punto a un altro della rete. Ogni applicazione deve scegliere tra i protocolli di trasporto quale deve adottare per realizzare un protocollo applicativo in base ai servizi che sono necessari alle specifiche esigenze della applicazione, che possono essere riassunte in:

- trasferimento dati affidabile;
- ampiezza di banda;
- temporizzazione;
- sicurezza

# Trasferimento dati affidabile

Con trasferimento di dati affidabile intendiamo un servizio che garantisce la consegna completa e corretta dei dati: da una parte sappiamo che alcune applicazioni, come per esempio quelle di audio/video possono tollerare qualche perdita di dati senza compromettere lo scopo dell'applicazione mentre altre, come per esempio il trasferimento di file, richiedono un trasferimento dati affidabile al 100%. A tale scopo il livello di trasporto mette a disposizione due protocolli:

1. **UDP (User Datagram Protocol):** il protocollo di trasporto senza connessione da utilizzarsi quando la perdita di dati è un fatto accettabile in quanto non è affidabile, non offre il controllo di flusso, il controllo della congestione, del ritardo e una banda minima;
2. **TCP (Transmission Control Protocol):** il protocollo orientato alla connessione da utilizzarsi quando la perdita di dati è un evento inaccettabile, ovvero quando il trasferimento deve essere affidabile; dà la garanzia di un trasporto senza errori o perdita di informazioni, effettua il controllo di flusso in quanto se il ricevente è più lento del mittente esso rallenterà per non sommergere il ricevente, esegue anche il controllo della congestione limitando il mittente se la rete è sovraccarica, ma non dà garanzie di banda minima.

# Trasferimento dati affidabile

Riportiamo una tabella dove sono indicati i protocolli utilizzati da alcune applicazioni:

APPLICAZIONE	PROTOCOLLO A LIVELLO APPLICAZIONE	PROTOCOLLO DI TRASPORTO SOTTOSTANTE
Posta elettronica	SMTP (RFC 2821)	TCP
Accesso a terminali remoti	Telnet (RFC 854)	TCP
Web	HTTP (RFC 2616)	TCP
Trasferimento file	FTP (RFC 959)	TCP
Multimedia in streaming	Proprietario (RealNetworks)	TCP o UDP
Telefonia Internet	Proprietario (Vonage, Dialpad)	Tipicamente UDP

# Ampiezza di banda (Bandwidth) o Throughput

Alcune applicazioni, come per esempio quelle multimediali, per poter “funzionare” hanno bisogno di avere una garanzia sulla larghezza di banda minima disponibile, cioè possono richiedere un throughput garantito di **R BSP**: si pensi alla trasmissione di un evento in diretta in una **Web-TV**. Altre applicazioni, invece, non hanno questo bisogno come prioritario, ma utilizzano in modo elastico l’ampiezza di banda che si rende disponibile: tipici esempi sono la posta elettronica o i sistemi **FTP**.

**Internet** ha una natura eterogenea e quindi i protocolli di trasporto non sono in grado di garantire la presenza di una certa quantità di banda per tutta la durata necessaria per trasmettere un messaggio: vedremo che mette a disposizione la possibilità di implementare un protocollo applicativo flessibile che realizza un **circuito virtuale** che si adatta se la banda è insufficiente.

# Temporizzazione

Alcune applicazioni, come la telefonia VoIP, i giochi interattivi, gli ambienti virtuali, per essere “realistiche” ammettono solo piccoli ritardi per essere efficaci: lo strato di trasporto non è in grado di garantire i tempi di risposta perché le temporizzazioni presenti assicurano un certo ritardo *end-to-end* tra le applicazioni. Il protocollo TCP garantisce la consegna del pacchetto, ma non il tempo che ci impiega e neppure il protocollo UDP, nonostante sia più veloce del TCP, è temporalmente affidabile.

La soluzione, come vedremo, è quella di utilizzare un protocollo di trasporto in tempo reale, come RTP (Real Time Protocol), che è in grado di studiare i ritardi di rete e calibrare gli apparati e i collegamenti per garantire di restare nei limiti di tempo prefissati, scegliendo alternativamente quando utilizzare UDP e quando TCP.

# Sicurezza

Un'applicazione può richiedere allo strato di trasporto la **cifratura** di tutti i dati trasmessi in modo tale che anche se questi venissero intercettati da malintenzionati non si perda la riservatezza. È quindi possibile che vengano richiesti dei **servizi di sicurezza** da applicare per garantire l'integrità dei dati e l'autenticazione **end-to-end**.

Il **problema della sicurezza** nelle reti riveste una grande importanza dato che le reti per loro natura non sono sicure: molteplici sono le minacce e i pericoli per i dati che sono presenti nei diversi **host** e che circolano sulla rete. Garantire la sicurezza di un sistema informativo significa impedire a potenziali soggetti attaccanti l'accesso o l'uso non autorizzato di informazioni e risorse.

# Conclusioni

Riportiamo in una tabella i requisiti richiesti al servizio di trasporto da parte di alcune applicazioni:

APPLICAZIONI	TOLLERANZA ALLA PERDITA DEI DATI	AMPIEZZA DI BANDA	SENSIBILITÀ DAL TEMPO
Trasferimento file	No	Variabile	No
Posta elettronica	No	Variabile	No
Documenti Web	No	Variabile	No
Audio/Video in tempo reale	Sì	Audio: da 5 Kbps a 1 Mbps	Sì, centinaia di ms
Audio/Video memorizzati	Sì	Video: da 10 Kbps a 5 Mbps	Sì, pochi secondi
Giochi interattivi	Sì	Fino a pochi K	Sì, centinaia di ms
Messaggistica istantanea	No	Variabile	Sì e no





# 02

## I socket e i protocolli per la comunicazione di rete

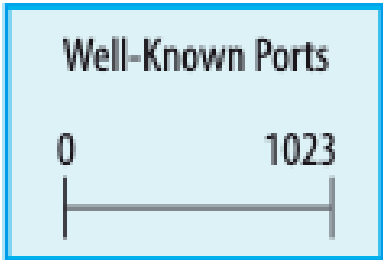
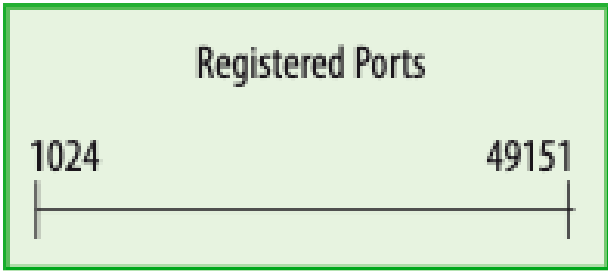

# Le porte di comunicazione e i socket

Affinché un processo, presente su un determinato host, invii un messaggio a un qualsiasi altro host, il processo mittente deve identificare il processo destinatario in modo univoco.

Generalmente ogni PC ha una sola porta fisica di connessione al network: se più applicazioni necessitano di utilizzare la rete, devono essere riconosciute in qualche modo, permettendo così di ricevere e spedire dati in modo corretto. Questo metodo di riconoscimento viene effettuato tramite le cosiddette porte logiche identificate da un numero detto port address oppure, sinteticamente, port; in questo modo è possibile instaurare simultaneamente più comunicazioni cosicché due applicazioni possono comunicare l'una con l'altra indipendentemente dal fatto che sulla rete stiano avvenendo altre comunicazioni: è sufficiente utilizzare porte logiche diverse. Il numero di port è specificato su 2 byte (da 0 a 65535) e identifica un particolare canale utilizzabile per la comunicazione.

Porta logica	Protocollo di rete	Porta logica	Protocollo di rete
7	ECHO	53	DNS (Domain Name Service)
21/tcp	FTP (file transfer protocol)	80/tcp	HTTP (Hyper Textual Transfer Protocol)
22/tcp	SSH (Secure SHell)	110/tcp	POP3 (Post Office Protocol, v3)
23/tcp	TELNET	143/tcp	IMAP (Internet Message Access Protocol)
25/tcp	SMTP (Simple Mail Transfer Protocol)	161	SNMP (Simple Network Management Protocol)
42	WINS (Windows Internet Naming Service)	443/tcp	HTTPS (Secure HTTP)

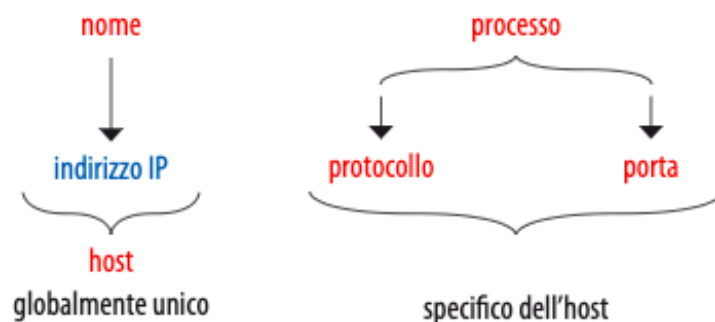
# Le porte di comunicazione e i socket

 <p>A light blue rectangular box with a blue border. Inside, the text 'Well-Known Ports' is centered at the top. Below it, the numbers '0' and '1023' are positioned at the left and right ends of a horizontal line, with vertical tick marks at each end.</p>	 <p>A light green rectangular box with a green border. Inside, the text 'Registered Ports' is centered at the top. Below it, the numbers '1024' and '49151' are positioned at the left and right ends of a horizontal line, with vertical tick marks at each end.</p>	 <p>A light orange rectangular box with an orange border. Inside, the text 'Dynamic and/or Private Ports' is centered at the top. Below it, the numbers '49152' and '65535' are positioned at the left and right ends of a horizontal line, with vertical tick marks at each end.</p>
<p>I port da 0 a 1023 sono riservati ad applicazioni particolari (quali HTTP, FTP, DNS, TelNet ecc.), costituiscono i cosiddetti <u>well-known port numbers</u> e possono essere usati solo dai server in apertura passiva.</p>	<p>I numeri di porta da 1024 a 49151 sono riservati a porte registrate e sono usati da alcuni servizi ma possono anche essere utilizzati dai client (<u>tutti i client usano normalmente le porte a partire dalla numero 1024 per collegarsi a un sistema remoto</u>).</p>	<p>I numeri di porta da 49152 a 65535 sono liberi per essere <u>assegnati</u> dinamicamente <u>dai processi applicativi</u>.</p>

# Le porte di comunicazione e i socket

L'identificazione di un servizio, quindi, avviene combinando l'indirizzo IP dell'host e della porta logica che viene utilizzata per renderlo disponibile in rete, quindi:

- indirizzo IP: identifica il nodo su cui opera il processo con cui si desidera comunicare;
- porta logica: identifica la porta utilizzata dal particolare processo all'interno di quel nodo.

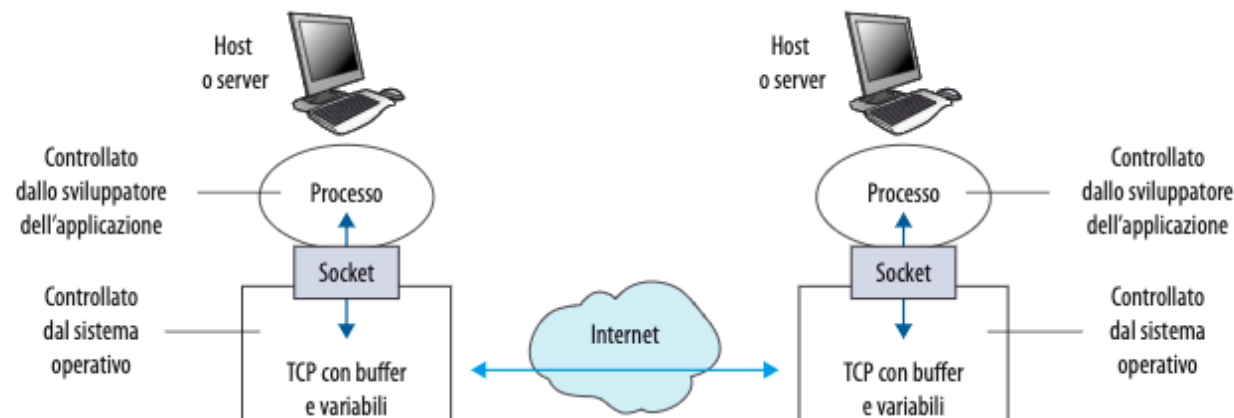


L'identificazione univoca avviene conoscendo sia l'indirizzo IP che il numero di porta associato al processo in esecuzione su un host: questo meccanismo prende il nome di meccanismo dei **socket**.

Un socket è formato dalla coppia <**indirizzo IP: numero della porta**>: è un identificatore analogo a una porta, cioè a un punto di accesso/uscita, unico per ogni rete.

# Le porte di comunicazione e i socket

Un processo che vuole inviare un messaggio lo “fa uscire” dalla propria “interfaccia” (**socket del mittente**) sapendo che un’infrastruttura esterna lo trasporterà attraverso la rete fino alla “interfaccia” del processo di destinazione (**socket del destinatario**).



Un **socket** consente quindi di comunicare attraverso la rete usando la pila **TCP/IP** ed è perciò parte integrante del protocollo: le **API** mettono a disposizione del programmatore gli strumenti atti a codificare la connessione e l’uso del protocollo di comunicazione.

# Le porte di comunicazione e i socket

## ESEMPIO

Nell'esempio schematizzato nella figura seguente, due applicazioni dell'host A e una dell'host B con i seguenti socket:

host A: <137.204.10.85:3300>

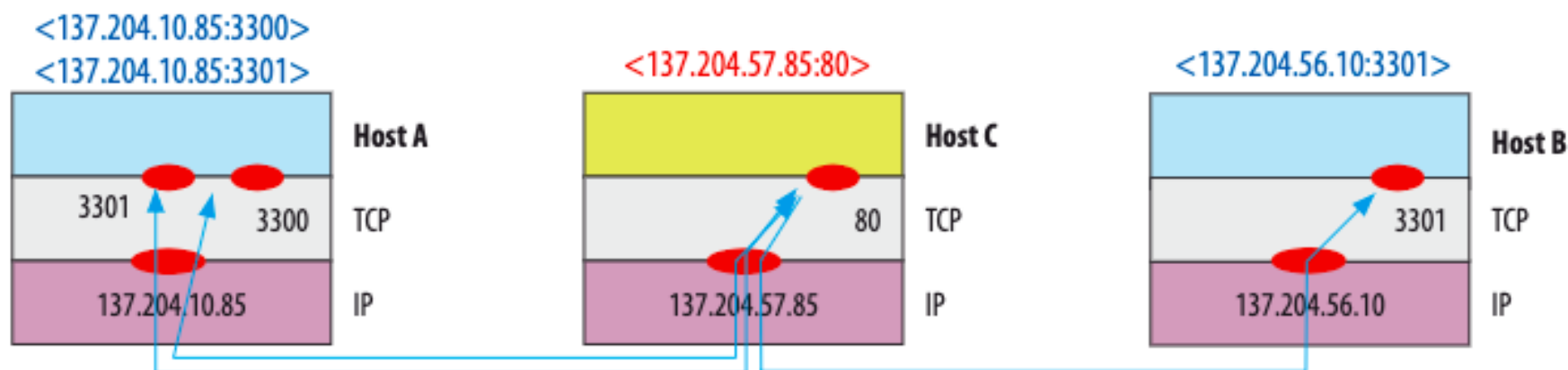
host A: <137.204.10.85:3301>

host B: <137.204.56.10:3301>

si connettono alla stessa porta 80 dell'host C richiedendo un servizio http al socket:

host C: <137.204.57.85:80>

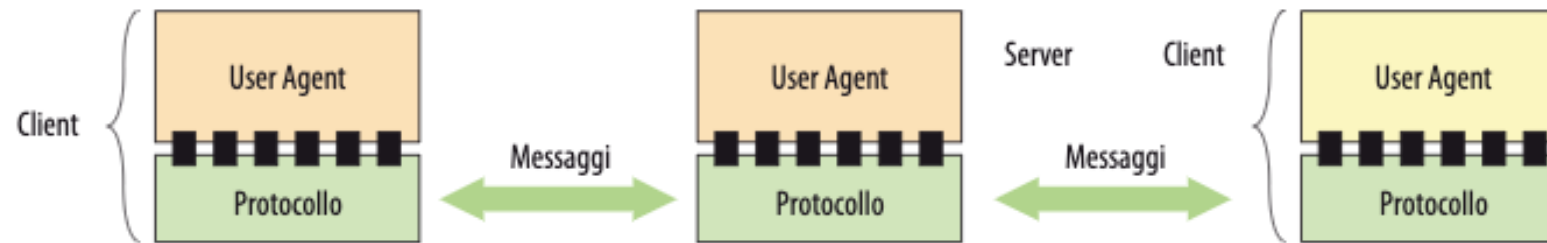
avendo due il medesimo indirizzo IP sono univocamente individuati mediante il valore del socket.



# Le porte di comunicazione e i socket

L'applicazione di rete può essere vista come composta da due parti:

1. una **user agent**, che funge da interfaccia tra l'utilizzatore dell'applicazione e gli aspetti comunicativi;
2. l'implementazione dei **protocolli**, che permettono all'applicazione di integrarsi con la rete.



Esiste una struttura chiamata **association** che consente d'identificare ogni singola connessione in modo univoco e che contiene le seguenti informazioni:

Protocollo (TCP, UDP...)	Indirizzo IP (locale)	Indirizzo IP (remoto)
	Porta (locale)	Porta (remota)

# Socket e i processi client-server

Il modello client-server è organizzato in due moduli, chiamati appunto server e client, operanti su macchine diverse:

- il server svolge le operazioni necessarie per realizzare un servizio;
- il client, generalmente tramite un'interfaccia utente, acquisisce i dati, li elabora e li invia al server richiedendo un servizio.

Quindi in rete sono presenti calcolatori su cui girano processi server che erogano servizi e sono in attesa di ricevere richieste di connessione da parte di processi client interessati a usufruire di tali servizi.





# 03

## La connessione tramite socket

# Generalità

Il concetto di **socket** è stato sviluppato come estensione diretta del paradigma **UNIX** di **I/O su file**, che si basa sulla sequenza di operazioni **open-read-write-close**:

- **open**: permette di accedere a un file;
- **read/write**: accedono ai contenuti del file;
- **close**: terminazione dell'utilizzo del file.

L'utilizzo dei **socket** avviene pressoché con la stessa modalità ma aggiungendo a questa struttura l'insieme dei parametri necessari a realizzare la connessione tra macchine remote, cioè richiedendo:

- gli indirizzi;
- il protocollo e numero di porta;
- il tipo del protocollo.

# Generalità

Ogni sistema operativo mette a disposizione nelle API i meccanismi per realizzare l'interfacciamento tra diversi protocolli: le **socket API** sono delle specifiche **API** di protocollo che hanno origine con Berkeley BSD UNIX e che oggi sono disponibili in Windows, Solaris e Linux.

Elenchiamo le funzioni presenti in Java:

- `socket()/server socket()`: crea un nuovo socket;
- `close()`: termina l'utilizzo di un socket;
- `bind()`: collega un indirizzo di rete a un socket;
- `listen()`: aspetta messaggi in ingresso;
- `accept()`: comincia a utilizzare una connessione in ingresso;
- `connect()`: crea una connessione con un host remoto;
- `send()/write()`: trasmette dati su una connessione attiva;
- `recv()/read()`: riceve dati da una connessione attiva.

# Generalità-plus

Le socket API sono un insieme di chiamate di sistema o funzioni fornite dai sistemi operativi per consentire la comunicazione tra processi attraverso una rete di computer. Le socket sono un concetto fondamentale nella programmazione di reti e forniscono un'interfaccia per la trasmissione di dati tra applicazioni su dispositivi separati. Di seguito sono presenti le principali funzionalità delle socket API:

## **Creazione di una Socket:**

`socket()`: Crea una nuova socket e restituisce un descrittore di file associato.

## **Associazione di una Socket a un Indirizzo e una Porta:**

`bind()`: Associa una socket a un indirizzo IP e a una porta locale.

## **Ascolto per Connessioni in Arrivo:**

`listen()`: Prepara una socket per accettare le connessioni in arrivo da altri endpoint.

## **Accettazione di una Connessione:**

`accept()`: Accetta una connessione in arrivo e restituisce una nuova socket associata a quella connessione.

## **Connessione a un Server:**

`connect()`: Inizia una connessione a un server remoto.

## **Invio e Ricezione di Dati:**

`send()`, `sendto()`: Invia dati attraverso una socket.

`recv()`, `recvfrom()`: Riceve dati attraverso una socket.

# Generalità-plus

## Chiusura di una Connessione:

`close()`: Chiude una socket.

## Gestione degli Errori:

`perror()`, `errno`: Forniscono informazioni sugli errori che si verificano durante l'uso delle socket.

## Configurazione delle Opzioni di Socket:

`setsockopt()`: Imposta opzioni specifiche della socket, come l'opzione di riutilizzo dell'indirizzo.

Le socket API sono implementate in modo simile su diverse piattaforme, ma possono variare leggermente in base al sistema operativo. I linguaggi di programmazione forniscono spesso wrapper intorno a queste chiamate di sistema per semplificare l'utilizzo delle socket. Ad esempio, in linguaggi come C, Python o Java, esistono librerie e classi che semplificano la programmazione di reti utilizzando le socket API sottostanti.

# Famiglie di socket

Esistono varie famiglie di socket (chiamate anche domini) dove ogni famiglia riunisce i socket che utilizzano gli stessi protocolli (Protocol Family) sottostanti: ogni famiglia supporta un sottoinsieme di stili di comunicazione e possiede un proprio formato di indirizzamento (Address Family). Tra le famiglie di socket ricordiamo:

- **Internet socket** (AF\_INET): è quella che utilizzeremo nelle nostre applicazioni e permette il trasferimento di dati tra processi posti su macchine remote connesse tramite una LAN o Internet;
- **Unix Domain socket** (AF\_UNIX): permette il trasferimento di dati tra processi sulla stessa macchina Unix.

# Tipi di socket

I socket sono fondamentalmente di tre tipi e per ciascuna tipologia abbiamo una diversa modalità di connessione:

- stream socket;
- datagram socket;
- raw socket.

Gli stream socket e datagram socket sono usati a livello applicativo mentre i raw sockets sono utilizzati nello sviluppo di protocolli e quindi esulano dagli obiettivi di questa trattazione

# Stream socket

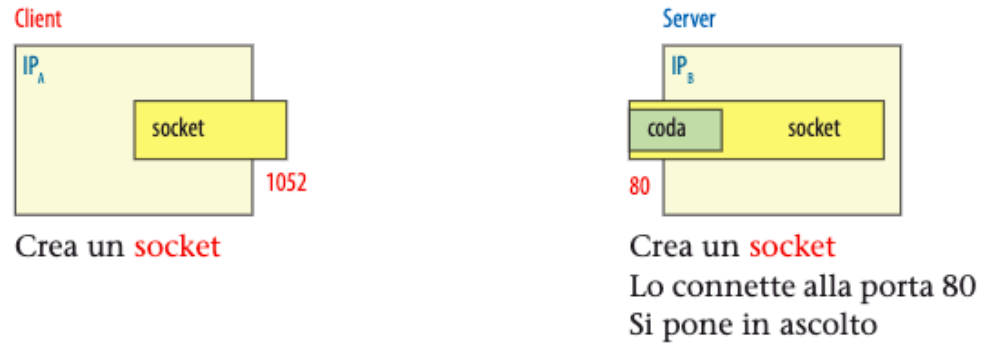
Con gli **stream socket** (**SOCK\_STREAM**) si realizza una connessione sequenziale tipicamente asimmetrica, affidabile e full-duplex basata su stream di byte di lunghezza variabile. Operativamente, ogni processo crea il proprio endpoint creando l'oggetto **socket** in **Java** e successivamente:

- il **server** si mette in ascolto, in attesa di un collegamento, e quando gli arriva una richiesta la esaudisce mediante la primitiva **accept()** che crea un nuovo **socket** dedicato alla connessione;
- il **client** si pone in coda sul **socket** del server e quando viene “accettato” dal **server** crea implicitamente il binding con la porta locale.



# Stream socket

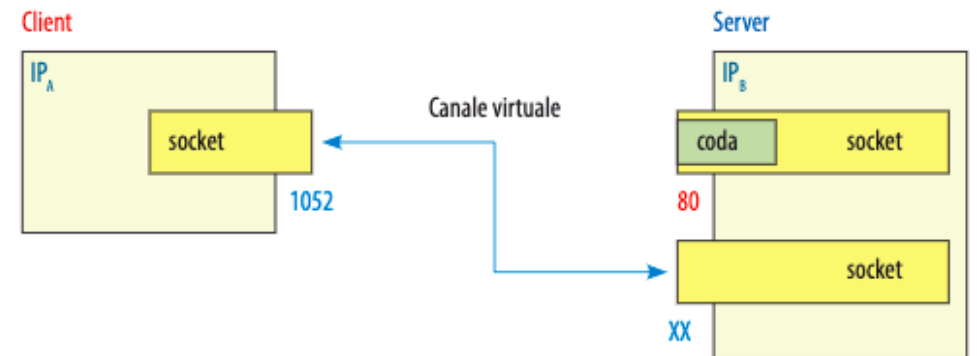
A) Inizializzazione dei processi.



B) Il client effettua la richiesta di collegamento.



C) Il server accetta la richiesta e stabilisce il collegamento.



# Stream socket - plus

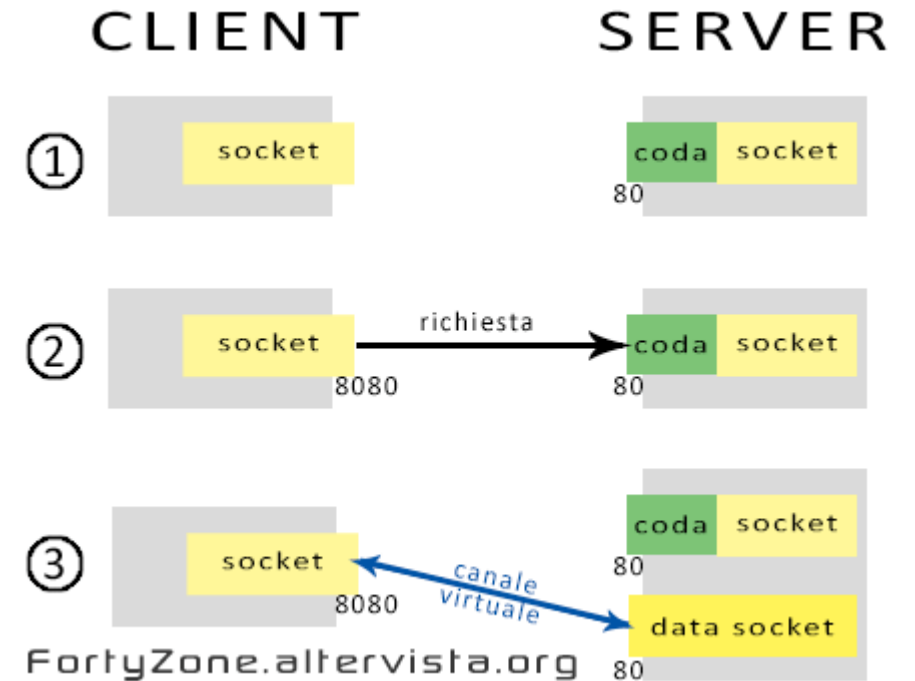
## Stream Socket

Essendo basati su protocolli a livello di trasporto come TCP, garantiscono una comunicazione affidabile, full-duplex, orientata alla connessione, e con un flusso di byte di lunghezza variabile.

La comunicazione mediante questo socket, si compone di queste fasi:

#1 -, Creazione dei socketClient e server creano i loro rispettivi socket il server lo pone in ascolto su una porta.

Dato che il server può creare più connessioni con client diversi (ma anche con lo stesso), ha bisogno di una coda per gestire le varie richieste.



# Stream socket - plus

## #2 – Richiesta di connessione

Il client effettua una richiesta di connessione verso il server. Da notare che possiamo avere due numeri di porta diversi, perchè una potrebbe essere dedicata solo al traffico in uscita, l'altra solo in entrata; questo dipende dalla configurazione dell'host. In sostanza, non è detto che la porta locale del client coincida con quella remota del server. Il server riceve la richiesta e, nel caso in cui sia accettata, viene creata una nuova connessione.

## #3 – Comunicazione

Ora client e server comunicano attraverso un canale virtuale, tra il socket del primo, ed uno nuovo del server, creato appositamente per il flusso dei dati di questa connessione: data socket. Coerentemente a quanto accennato nella prima fase, il server crea il data socket perchè il primo serve esclusivamente alla gestione delle richieste. È possibile, quindi, che ci siano molti client a comunicare con il server, ciascuno verso il data socket creato dal server per loro.

## #4 – Chiusura della connessione

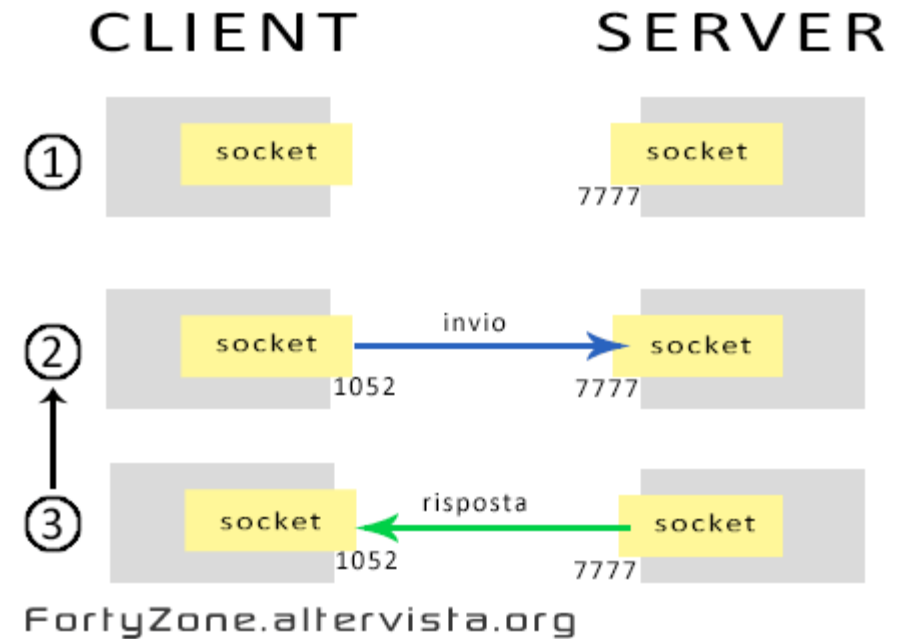
Essendo il TCP un protocollo orientato alla connessione, quando non si ha più la necessità di comunicare, il client lo comunica al server, che ne deinstanzia il data socket. La connessione viene così chiusa.

# Datagram socket - plus

## Datagram Socket

Sono basati su UDP, un protocollo a livello di trasporto che garantisce comunicazioni a bassa latenza (ideali per videochat, ad esempio), a discapito dell'affidabilità dei dati. Non esiste, infatti, controllo di flusso (ordinamento dei datagrammi e controllo degli errori).

Dato che l'UDP non è un protocollo orientato alla connessione, non esiste la fase di connessione vista poco fa, ma il client comunica direttamente con il server, quando vuole.



# Datagram socket - plus

## #1 – Creazione dei socket

Come nel tipo precedente, client e server creano i loro rispettivi socket, e il server lo pone in ascolto su una porta. Il socket del server, stavolta, non ha bisogno di una coda, in quanto i dati in entrata e in uscita non devono essere circoscritti all'interno di una connessione, quindi la comunicazione con diversi client si svolge sulla stessa interfaccia.

## #2 – Invio dei dati

Il client invia direttamente i datagrammi al server. Anche in questo caso vale quanto detto in precedenza riguardo i numeri di porta.

## #3 – Risposta del server

Il server manda al client un eventuale risposta. La comunicazione è quindi, costituita da un loop che dura finchè ci sono dati da inviare e, ovviamente, finchè gli host sono raggiungibili.

# Stream socket

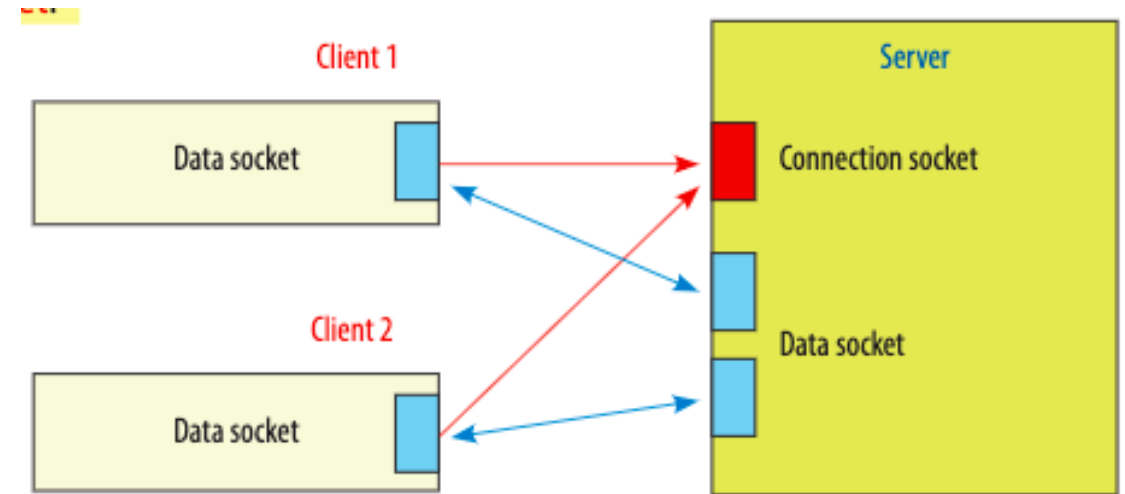
Il **server** accetta la richiesta e realizza “un canale virtuale” tra il **client** e un nuovo **socket** del **server** (indicato XX), in modo da lasciare il primo libero per le ulteriori richieste da parte di altri **client**.

I due processi si scambiano i dati (funzioni `read()` e `write()`) fino alla chiusura del canale, che viene effettuata rispettivamente mediante la chiamata della primitiva `close()`.

Il protocollo **TCP** è basato su questo tipo di **socket**.

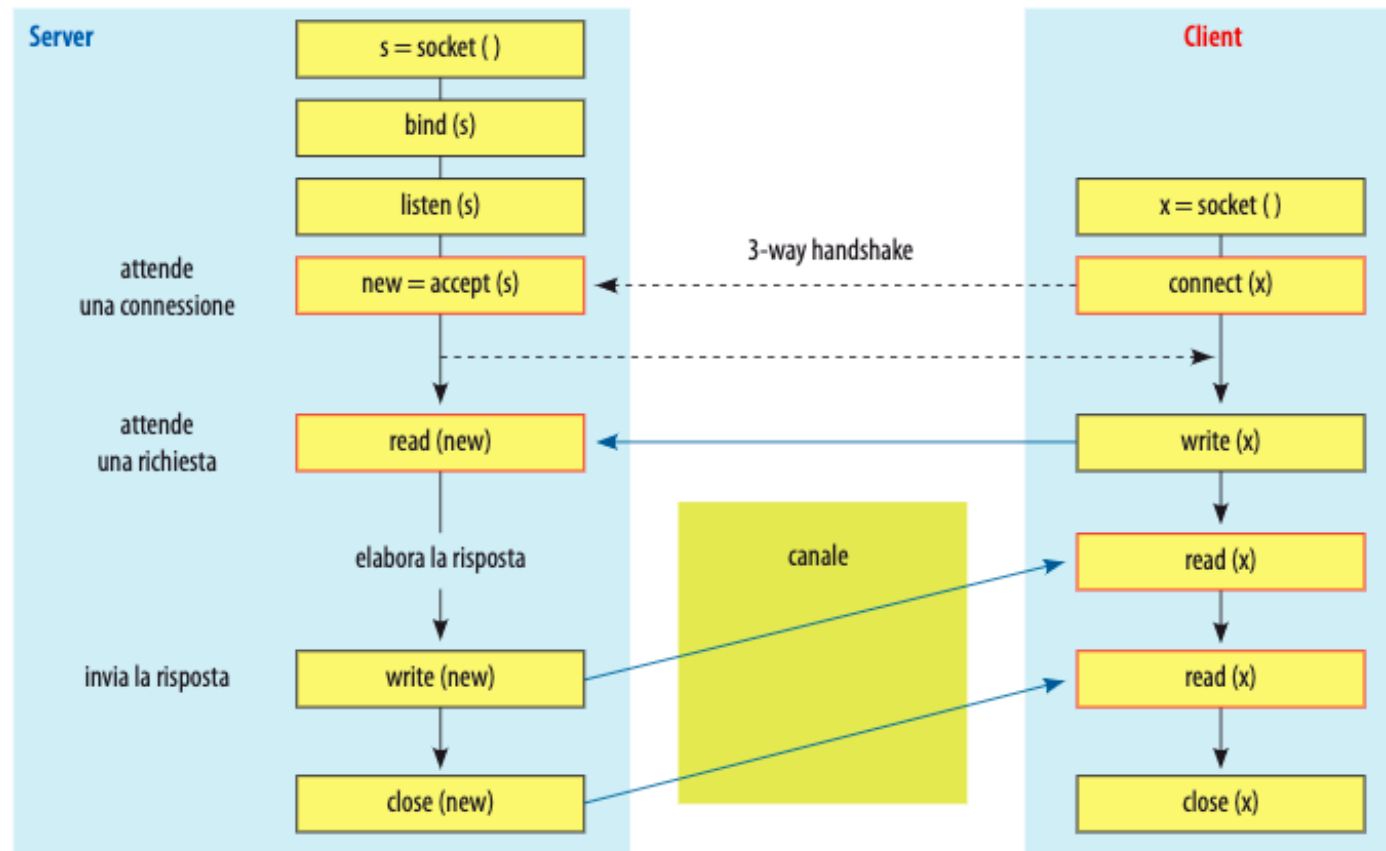
Nella figura possiamo osservare che in un **server TCP** abbiamo due tipi di socket:

- un tipo per accettare connessioni (condiviso), che chiamiamo **connection socket**;
- un tipo per le operazioni `send()` e `receive()` (non condiviso), che chiamiamo **data socket**.



# Stream socket

Lo schema logico completo della comunicazione TCP mediante socket nel linguaggio C è il seguente:



# Datagram socket

Con i **datagram socket (SOCK\_DGRAM)** viene realizzata la comunicazione che permette di scambiare dati senza connessione (i messaggi contengono l'indirizzo di destinazione e provenienza) mediante il trasferimento di datagrammi che inoltrano messaggi di dimensione variabile, senza garantire ordine o arrivo dei pacchetti, quindi si ha una comunicazione inaffidabile.

Permettono quindi di inviare da un **socket** a più destinazioni e ricevere su un **socket** da più sorgenti: in generale realizzano quindi il modello “molti a molti” e sono supportate nel dominio Internet dal protocollo **UDP**. Operativamente, ogni processo crea il proprio **endpoint** richiamando la primitiva che crea il **socket** e successivamente:

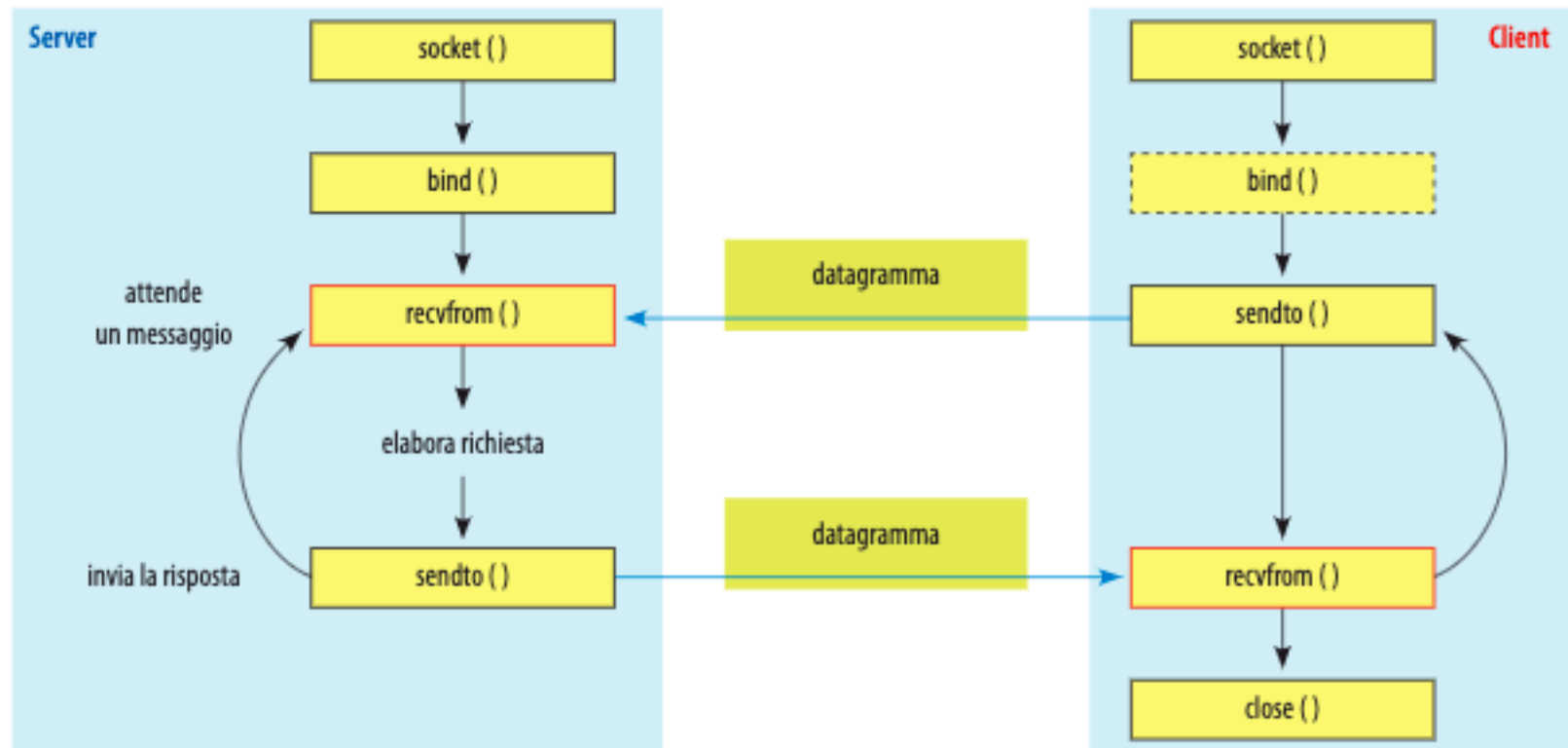
- il **server** si mette in attesa di ricevere i dati (mediante la primitiva **receive()** in **Java**) e alla loro ricezione può inviare una risposta (mediante la primitiva **send()** in **Java**);
- il **client** invia il pacchetto di dati al **server** (mediante la primitiva **send()** in **Java**) e può mettersi in attesa di una risposta con le stesse primitive che ha utilizzato il server (cioè mediante la primitiva **receive()** in **Java**).

Al termine della comunicazione il **socket** viene chiuso.



# Datagram socket

Lo schema logico completo della comunicazione UDP in linguaggio C mediante socket è il seguente:



# Trasmissione unicast e multicast

L'**unicast** è la “normale” situazione in cui un mittente invia e un destinatario riceve con eventualmente l'inversione dei ruoli (relazione **one-to-one**), mentre il multicast è utilizzato per trasmettere informazioni a più host contemporaneamente (relazione **one-to-many**).

Nella comunicazione di tipo **multicast** un insieme di processi formano un gruppo di **multicast** e un messaggio spedito da un processo a quel gruppo viene recapitato a tutti gli altri partecipanti appartenenti al gruppo.

Tipiche applicazioni **multicast** sono per esempio:

- **usenet news**: pubblicazione e diffusione di nuove notizie in tempo reale;
- **videoconferenze**: ogni host genera un segnale audio video che viene ricevuto dagli host associati ai partecipanti alla videoconferenza;
- **massive multiplayer games**: giochi di ruolo e Internet game dove un elevato numero di giocatori interagiscono in un mondo virtuale;
- **DNS (Domain Name System)**: aggiornamenti delle tabelle di naming inviati a gruppi di **DNS**;
- altre applicazioni quali chat, instant messaging, applicazioni P2P ecc

# Trasmissione unicast e multicast

Per implementare un sistema **multicast** è necessario poter definire uno schema di indirizzamento dei gruppi e un supporto che registri la corrispondenza tra un gruppo e i partecipanti oltre alla possibilità di ottimizzare l'uso della rete nel caso di invio di pacchetti a un gruppo (tramite multicast router).

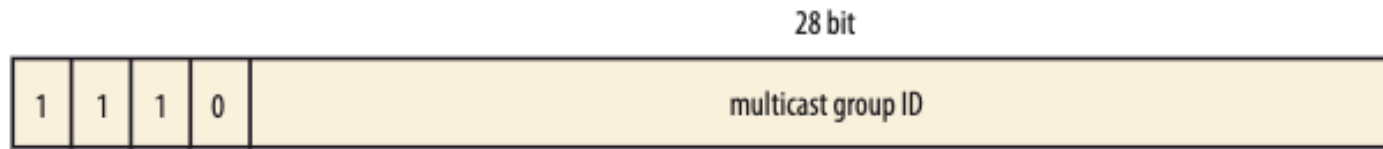
Il protocollo più utilizzato per il **multicast** in Internet è l'**IGMP** (**I**nternet **G**roup **M**anagement **P**rotocol) che serve a garantire la trasmissione, tra **host** e **multicast router** a essi direttamente collegati, dei messaggi relativi alla costituzione dei gruppi: per esempio fornisce a un **host** i mezzi per informare il **multicast router** a esso più vicino che un'applicazione vuole unirsi a un determinato gruppo multicast utilizzando normali **datagrammi IP**.

Le **API multicast** devono quindi contenere primitive per:

- **unirsi** a un gruppo di multicast (**join**): identificare e indirizzare univocamente un gruppo;
- **lasciare** un gruppo di multicast (**leave**);
- **spedire** messaggi a un gruppo, cioè a tutti i processi che in quel momento fanno parte del gruppo;
- **ricevere** messaggi indirizzati a un gruppo del quale l'host fa parte.

# Trasmissione unicast e multicast

Come indirizzo di multicast viene utilizzato un indirizzo IP di classe D, del tipo:



cioè con intervallo tra 224.0.0.0 – 239.255.255.255.

Gli indirizzi possono essere:

- **permanenti:** l'indirizzo di **multicast** viene assegnato dalla **IANA** (**Internet Assigned Numbers Authority**) e rimane assegnato a quel gruppo anche se in un certo momento non ci sono più partecipanti connessi; questi indirizzi sono detti **well-known**;
- **temporanei:** richiedono la definizione di un opportuno protocollo per evitare conflitti nell'attribuzione degli indirizzi ai gruppi ed esistono solo fino al momento in cui esiste almeno un partecipante.

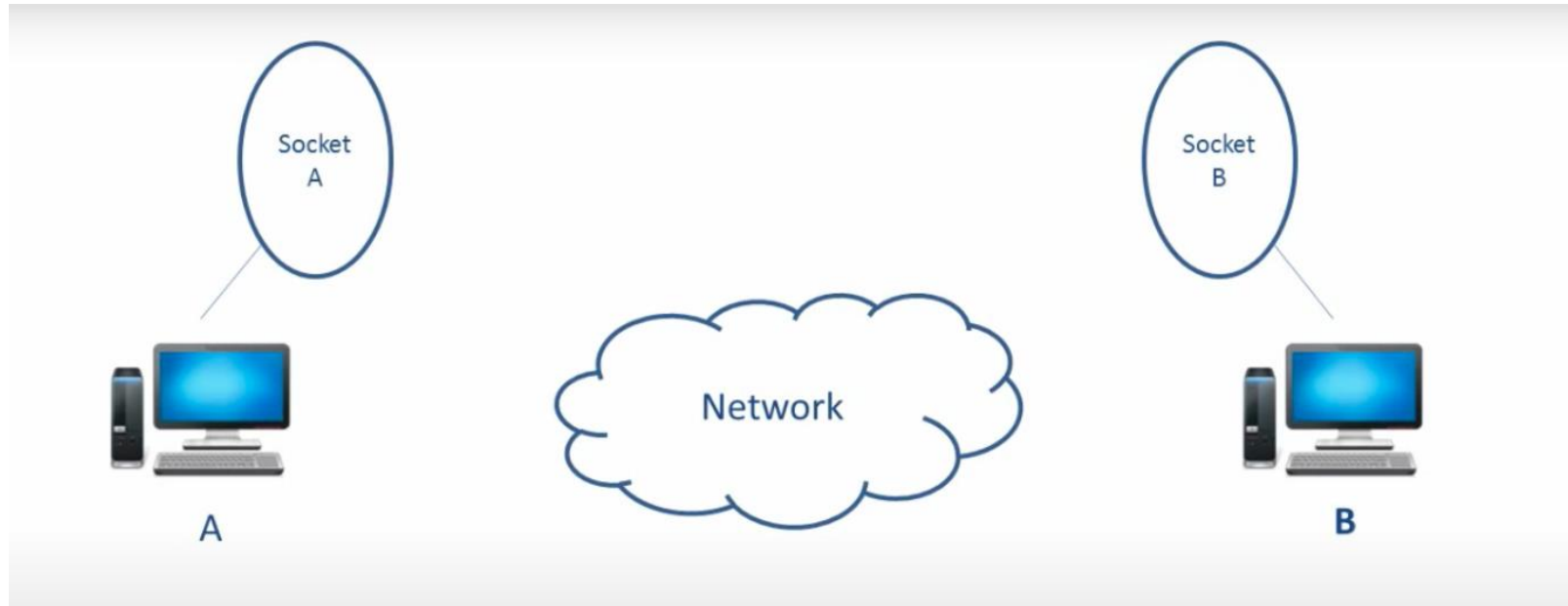
# Trasmissione unicast e multicast

La comunicazione **multicast** utilizza il paradigma **connectionless** dato che devono essere gestite contemporaneamente un alto numero di connessioni.

È anche possibile associare un nome simbolico a un gruppo di **multicast**: nella schermata a fianco, ottenuta all'indirizzo <http://ip-lookup.net> digitando come IP address **240.0.0.1**, è possibile vedere il nome di una parte di questi gruppi.

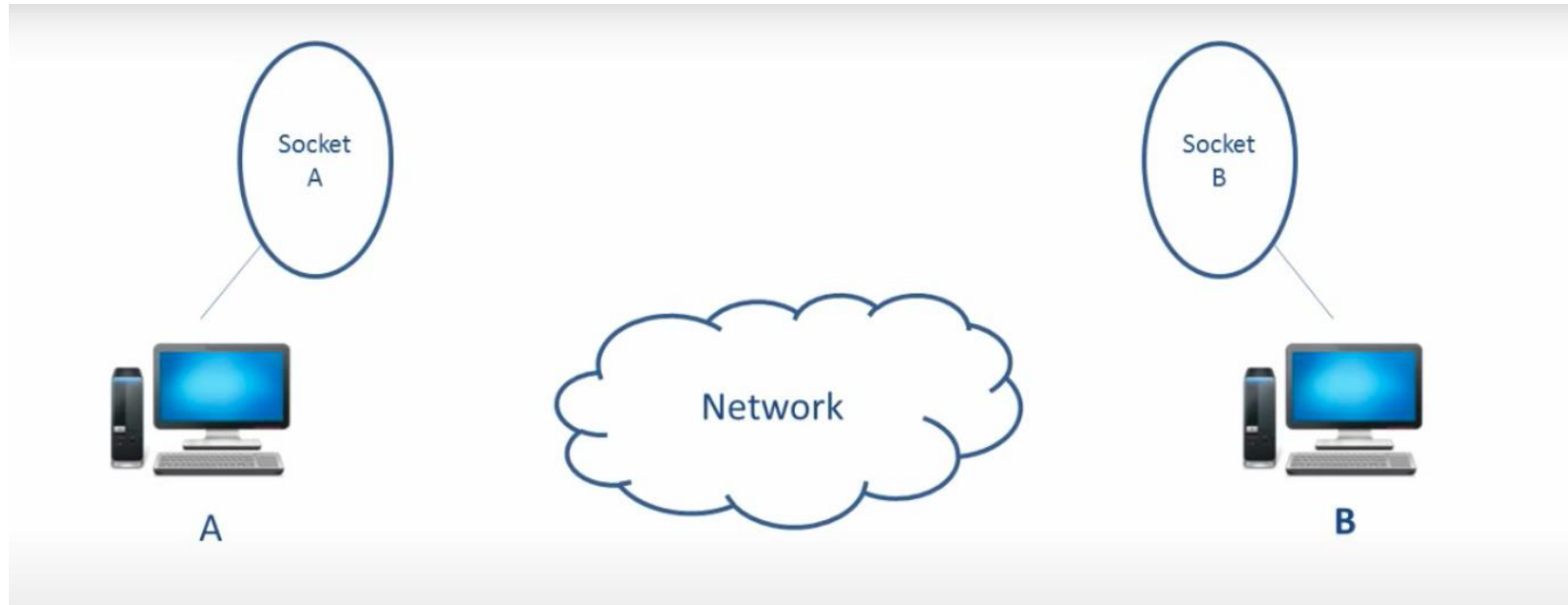
	IP address	Host name
1	224.0.0.0	base-address.mcast.net
	<b>224.0.0.1</b>	<b>all-systems.mcast.net</b>
1	224.0.0.2	all-routers.mcast.net
2	224.0.0.3	?
3	224.0.0.4	dmrp.mcast.net
4	224.0.0.5	ospf-all.mcast.net
5	224.0.0.6	ospf-dsmp.mcast.net
6	224.0.0.7	st-routers.mcast.net
7	224.0.0.8	st-hosts.mcast.net
8	224.0.0.9	rip2-routers.mcast.net
9	224.0.0.10	igrp-routers.mcast.net
10	224.0.0.11	mobile-agents.mcast.net
11	224.0.0.12	dhcp-agents.mcast.net
12	224.0.0.13	pim-routers.mcast.net
13	224.0.0.14	rsvp-encapsulation.mcast.net

# Socket



Socket A e Socket B sono due **oggetti in JAVA**. Socket input stream sono utilizzati per leggere i dati, i socket output stream sono utilizzati per scrivere o inviare dati.

# Server socket



Socket A e Socket B sono due **oggetti in JAVA**. Socket input stream sono utilizzati per leggere i dati, i socket output stream sono utilizzati per scrivere o inviare dati.