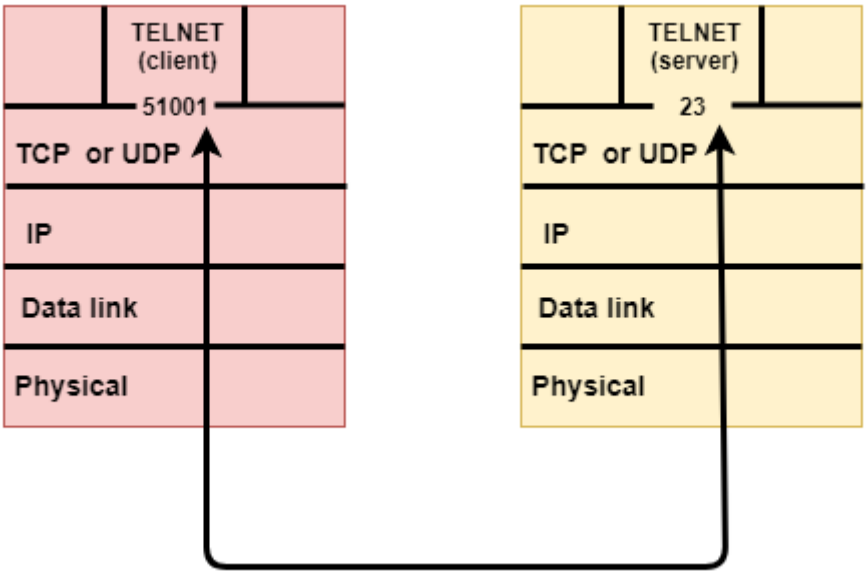
Livello di trasporto

Il compito del **livello di trasporto** è di fornire servizi al soprastante livello di applicazioni e per raggiungere tale scopo sfrutta i servizi offerti dal sottostante livello 3 (livello rete).

Lo scopo del livello di trasporto è fornire un canale logico di comunicazione end-to-end per pacchetti. Specificatamente si occupa di supplire alle mancanze delle funzionalità del trasferimento in termini di affidabilità implementando le suddette funzioni come garanzie sul trasporto stesso.

Il livello di trasporto si trova solo negli end-systems (hosts) e consente il collegamento logico tra processi applicativi.



Nell'immagine si vedono i due livelli di trasporto di due host che dialogano. Per dialogare devono passare nei livelli sottostanti

Protocolli di trasporto

Il livello di trasporto fornisce al livello applicativo delle connessioni con specifiche qualità, in particolare sfruttando i protocolli *UDP, TCP e QUIC*.

TCP

TCP (Transmission Control Protocol) è uno dei principali protocolli di comunicazione utilizzato nel livello di trasporto.
Il TCP consente di avere un servizio orientato alla connessione e affidabile.
Tramite TCP, tra mittente e destinatario si stabilisce una connessione logica affidabile e bidirezionale.

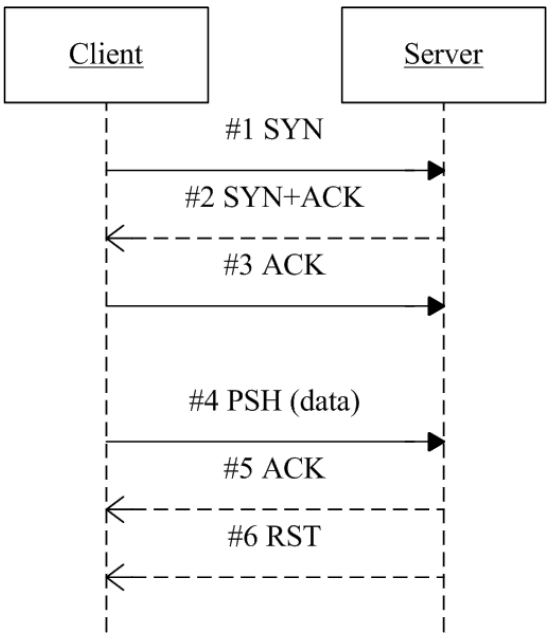
Le caratteristiche principali del *TCP* includono:

- *Affidabilità*: *TCP* garantisce una consegna affidabile dei dati. Utilizza meccanismi di riconoscimento, acknowledgment (abbreviato *ack*, conferma e controllo errori) e ritrasmissione per assicurarsi che i pacchetti dati siano consegnati correttamente e nell'ordine corretto. Se un pacchetto viene perso o corrotto durante la trasmissione, il mittente lo ri-trasmette finché il destinatario non conferma la sua ricezione.
- *Controllo di flusso*: *TCP* gestisce il controllo del flusso che serve per evitare che il mittente invii dati più velocemente di quanto il destinatario riesca a riceverli ed elaborarli.
- *Controllo della congestione*: *TCP* regola la velocità di invio in base alle condizioni della rete, per evitare di sovraccaricare i collegamenti e i router intermedi.
- *Orientamento alla connessione*: *TCP* stabilisce una connessione tra il mittente e il destinatario prima di iniziare a trasmettere i dati.

Per inviare dati tramite *TCP*, il protocollo suddivide il messaggio dato dal livello applicativo in pacchetti più piccoli chiamati **segmenti**. Questi segmenti contengono:
- I dati originali (payload).
- l'header *TCP* che contiene informazioni come il numero di sequenza, il numero di conferma, le informazioni di controllo e altri parametri necessari per garantire la corretta consegna e il controllo del flusso.

I segmenti TCP vengono inseriti in pacchetti IP (pacchetti creati nel livello di rete) e trasmessi attraverso la rete. I pacchetti IP possono anche seguire percorsi diversi prima di raggiungere la destinazione.
Una volta arrivati alla destinazione, il ricevitore assembla i segmenti in ordine di sequenza e utilizza le informazioni di controllo per confermare la ricezione dei segmenti .
Se un pacchetto non viene ricevuto correttamente, il *TCP* richiede la sua trasmissione al mittente per garantire l'integrità e l'affidabilità dei dati.
TCP è stato ed è ancora molto usato in Internet, soprattutto per servizi che richiedono affidabilità, come il trasferimento di file, molte comunicazioni web, email e accesso remoto.

Una connessione TCP passa attraverso almeno a queste fasi:
- **SYN**: Il client chiede di aprire una connessione (*SYN*) = "Inizio connessione".
- **SYN-ACK / ACK**: Il server risponde (*SYN-ACK*), e il client conferma (*ACK*) = "Connessione stabilita".
- **Data Transfer / PSH**: I due scambiano dati (segmenti con conferma, cioè *ACK*) = "Connessione attiva".
- **FIN / RST**: Uno dei due chiude la connessione (*FIN* o *RST*) = "Connessione terminata".



UDP

UDP, come TCP, è un protocollo del livello di trasporto, ma offre un servizio molto diverso.

A differenza del *TCP*, che offre una comunicazione affidabile e orientata alla connessione, l'*UDP* è un protocollo senza connessione e non garantisce l'affidabilità della consegna dei dati.
Questo significa che quando un'applicazione utilizza l'*UDP* per inviare dati a un'altra applicazione, i dati vengono semplicemente inviati senza alcuna garanzia che siano ricevuti o consegnati correttamente.
L'UDP consente quindi di avere un servizio non orientato alla connessione e non affidabile.

Caratteristiche principali di *UDP*:

- *Senza connessione*: Non viene stabilita alcuna connessione tra mittente e destinatario prima dell'invio dei dati. I dati vengono inviati senza nessuna negoziazione preliminare.
- *Comunicazione veloce e leggera*: Poiché *UDP* non ha la complessità della gestione delle connessioni o del controllo degli errori come *TCP*, è più veloce e leggero in termini di overhead.
- *Nessun controllo del flusso e della congestione*: *UDP* non gestisce meccanismi complessi di affidabilità come conferme, ritrasmissioni e riordinamento dei dati.
- *Utilizzo comune per applicazioni che richiedono bassa latenza*: *UDP* è spesso utilizzato in applicazioni in cui la velocità e la bassa latenza (il tempo di arrivo di un pacchetto) sono più importanti dell'affidabilità, come streaming multimediale, giochi online e trasmissioni in tempo reale.

Poiché l'*UDP* non garantisce la consegna affidabile dei dati, le applicazioni che utilizzano questo protocollo devono gestire eventuali perdite o errori di dati a livello dell'applicazione stessa, se necessario.

L'*UDP* è utilizzato in situazioni in cui la velocità e la semplicità sono più importanti dell'affidabilità nella consegna dei dati. Alcuni dei principali casi d'uso in cui l'*UDP* è preferito includono:

- *Flussi audio/video*: *UDP* è usato spesso per flussi audio/video in tempo reale, come videoconferenze, *VoIP* o trasmissioni live a bassa latenza.
- *Giochi online*: Nei giochi online, l'*UDP* è spesso preferito per trasmettere informazioni sulle azioni dei giocatori in tempo reale. La bassa latenza è critica per garantire un'esperienza di gioco reattiva e fluida.
- *Trasmissioni in tempo reale*: Nelle applicazioni di trasmissione in diretta, come trasmissioni sportive o eventi in tempo reale, l'*UDP* è preferito per fornire una visione in tempo reale senza buffering.
- *Servizi di ricerca di dispositivi locali*: Nelle reti locali, l'*UDP* è utilizzato in alcuni protocolli di ricerca di dispositivi, dove è importante una rapida scoperta dei dispositivi nella rete.
- *Applicazioni IoT (Internet of Things)*: In alcuni scenari IoT (i sensori della temperatura, ecc…), dove i dispositivi devono trasmettere piccole quantità di dati in modo rapido e senza molta complessità, l'UDP può essere adottato.

L'unità di dati inviata da UDP si chiama **datagramma**. Un datagramma *UDP* contiene un header, con informazioni di controllo essenziali, e un payload, cioè i dati ricevuti dall'applicazione.

QUIC

Dopo aver visto *TCP* e *UDP*, possiamo introdurre un protocollo più recente: ***QUIC***.
QUIC è un protocollo di trasporto moderno progettato per rendere le comunicazioni su Internet più veloci, sicure ed efficienti.
È particolarmente importante perché viene usato da *HTTP/3*, una delle versioni più recenti del protocollo *HTTP* utilizzato per il Web.

La caratteristica interessante di *QUIC* è che funziona sopra *UDP*.
Questo può sembrare strano, perché *UDP* è un protocollo non orientato alla connessione e non affidabile.
Tuttavia *QUIC* utilizza *UDP* come base e aggiunge sopra di esso molte funzionalità che normalmente associamo a *TCP*, come l'affidabilità, il controllo della congestione e la gestione della connessione.

Uno dei motivi per cui *QUIC* è stato sviluppato è ridurre il tempo necessario per iniziare una comunicazione. Con *TCP*, prima bisogna stabilire una connessione; se poi si usa *HTTPS* (HTTP più la sicurezza), bisogna aggiungere anche la negoziazione crittografica.
Questo può richiedere più passaggi prima che i dati veri e propri inizino a essere scambiati.

QUIC integra invece la sicurezza direttamente nel protocollo e usa meccanismi basati su *TLS*. Questo permette di ridurre il numero di passaggi necessari per stabilire una connessione sicura. Il risultato è che, in molti casi, una comunicazione può iniziare più rapidamente.

Un’altra caratteristica importante di *QUIC* è la gestione migliore delle interruzioni e dei cambi di rete. Per esempio, uno smartphone può iniziare a navigare usando il *Wi-Fi* e poi passare alla rete mobile.
Con una connessione tradizionale basata su *TCP*, questo cambio può interrompere la connessione, perché cambia l’indirizzo di rete (l’indirizzo IP) usato dal dispositivo.
QUIC, invece, è progettato per gestire meglio questi cambiamenti e mantenere la comunicazione attiva quando possibile.

QUIC migliora anche la gestione di più flussi di dati all’interno della stessa connessione. Nel Web moderno, una pagina non è formata da un solo file: contiene testo, immagini, fogli di stile, script, video e molte altre risorse. *QUIC* permette di gestire più flussi in parallelo, riducendo il rischio che il ritardo o la perdita di una parte dei dati blocchi inutilmente anche le altre.
Questa caratteristica è importante perché nelle comunicazioni di rete alcuni pacchetti possono arrivare in ritardo o andare persi. Con alcuni sistemi tradizionali, la perdita di un pacchetto può rallentare anche dati che sarebbero già disponibili. *QUIC* cerca di ridurre questo problema separando meglio i diversi flussi di comunicazione.

Possiamo quindi vedere QUIC come un protocollo che unisce alcune caratteristiche positive di *TCP* e *UDP*:

- Come *UDP*, è costruito su datagrammi e non richiede la stessa struttura rigida di *TCP*.
- Come *TCP*, può offrire affidabilità, controllo della congestione e gestione ordinata dei dati; però a differenza di TCP tradizionale, integra la sicurezza e può stabilire comunicazioni protette più rapidamente.

QUIC è adatto al Web moderno, dove servono velocità, sicurezza e capacità di gestire molti flussi di dati contemporaneamente.
QUIC è particolarmente importante perché è alla base di HTTP/3, la versione più recente di *HTTP*.

Porta

In un sistema di rete, **una porta** è il numero logico utilizzato per identificare un canale specifico di comunicazione all'interno di un host.
È uno degli elementi fondamentali che permette a più applicazioni nello stesso host di usare contemporaneamente la rete senza interferire tra loro.

Immagina un centralino telefonico di una grande azienda.
Tutti i telefoni sono collegati a un unico numero (l’indirizzo reale), ma quando chiami quel numero può essere messo in contatto con diversi interni: vendite, assistenza, amministrazione… Ecco: il numero di telefono è come l’indirizzo reale del computer, mentre gli interni sono le porte.

Quindi ricapitolando:

Le porte sono lo strumento utilizzato per permettere ad un host di effettuare più connessioni diversificate contemporanee verso altri host, facendo in modo che i dati contenuti da un servizio specifico vengano indirizzati al processo che li sta aspettando.

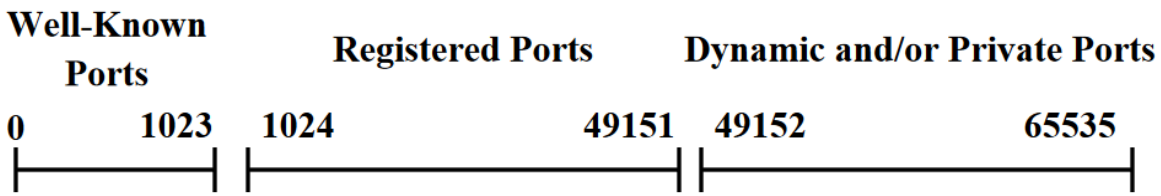
Nelle comunicazioni basate su *TCP* o *UDP*, le porte sono usate per associare i dati al processo applicativo corretto.
In una comunicazione di rete non esiste solo una porta: di solito ci sono una porta sorgente e una porta di destinazione.

- La porta di destinazione indica il servizio a cui vogliamo collegarci. Per esempio, un server web HTTPS ascolta normalmente sulla porta 443.
- La porta sorgente, invece, è scelta dal dispositivo che avvia la comunicazione. Spesso è una porta temporanea assegnata automaticamente dal sistema operativo.

In questo modo, il computer può distinguere più comunicazioni attive nello stesso momento. Per esempio, se un browser apre più connessioni verso siti web, il sistema operativo può usare porte sorgente diverse per riconoscere ciascuna comunicazione.

Le porte sono $2^{16} = 65536$, classificabili in tre gruppi:

- **le porte conosciute** sono quelle inferiori a 1024, e sono generalmente utilizzate a livello di sistema operativo o di processi di sistema. In genere rimangono in ascolto su queste porte applicazioni con funzioni di server (del livello applicativo).
  Alcuni esempi possono essere le applicazioni che utilizzino protocolli:
  - *FTP* (protocollo di trasferimento file, FileZilla usa questo protocollo) => porta 21.
  - *SSH* (un protocollo che permette di stabilire una sessione remota cifrata) => porta 22.
  - *SMTP* (un protocollo standard per la trasmissione di email) => porta 25.
  - *HTTP* (è un protocollo a livello applicativo usato come principale sistema per la trasmissione d'informazioni sul web) => porta 80.
  - *HTTPS* (è l'*HTTP* con l'aggiunta della sicurezza) → porta 443
- **Le porte registrate** Queste porte sono assegnate a specifici protocolli, applicazioni o servizi, ma non sono riservate in modo rigido come le porte conosciute.
  Le porte registrate sono spesso utilizzate da applicazioni e servizi meno comuni o da nuovi protocolli che richiedono un numero di porta standardizzato ma che non necessitano dell'ampia riconoscibilità associata alle porte ben note. Ad esempio, alcuni protocolli di comunicazione personalizzati o applicazioni meno diffuse potrebbero utilizzare porte registrate.
- **Le porte dinamiche/private** invece sono tutte le altre, liberamente utilizzabili da tutte le applicazioni utente, salvo l'occupazione contemporanea da parte di qualche altro processo. Le porte dinamiche sono generalmente assegnate dinamicamente dal sistema operativo e non sono riservate o assegnate in modo permanente a protocolli specifici. Ciò consente una flessibilità maggiore nell'allocazione delle risorse di comunicazione all'interno di un sistema, evitando conflitti di porte quando più applicazioni comunicano contemporaneamente.



Esempio
Vediamo un esempio di porta conosciuta
Quando visiti un sito web come "www.example.com" nel tuo browser, stai implicitamente comunicando con il server attraverso il protocollo HTTP (Hypertext Transfer Protocol) sulla porta predefinita 80. Il protocollo HTTP è utilizzato per richiedere e trasferire pagine web e altri contenuti attraverso il web.

Vediamo un esempio di porta registrata.
Un'azienda vuole creare un servizio web che consente di sommare due numeri.
Tale servizio deve ricevere in input due numeri e come output restituisce un numero che sarà la somma dell'input.
Questo programma calcolatrice viene posto nella porta 3000.
Tutti i client che vogliono usare il servizio devono collegarsi all'indirizzo del server specificando la porta 3000 e devono mettere nel loro messaggio due numeri e devono gestire come risultato un numero che sarà la somma dei due numeri precedenti (protocollo).

Vediamo un esempio di porta dinamica.
Quando apriamo un sito HTTPS, il browser si collega alla porta 443 del server web. Il computer dell'utente, però, usa anche una propria porta temporanea, scelta automaticamente dal sistema operativo, per identificare quella specifica comunicazione.
Per esempio, il browser potrebbe comunicare così: client: porta 52341 → server: porta 443
Se l'utente apre più schede o più servizi contemporaneamente, il sistema operativo può assegnare porte dinamiche diverse, in modo da distinguere le varie comunicazioni attive.

Socket

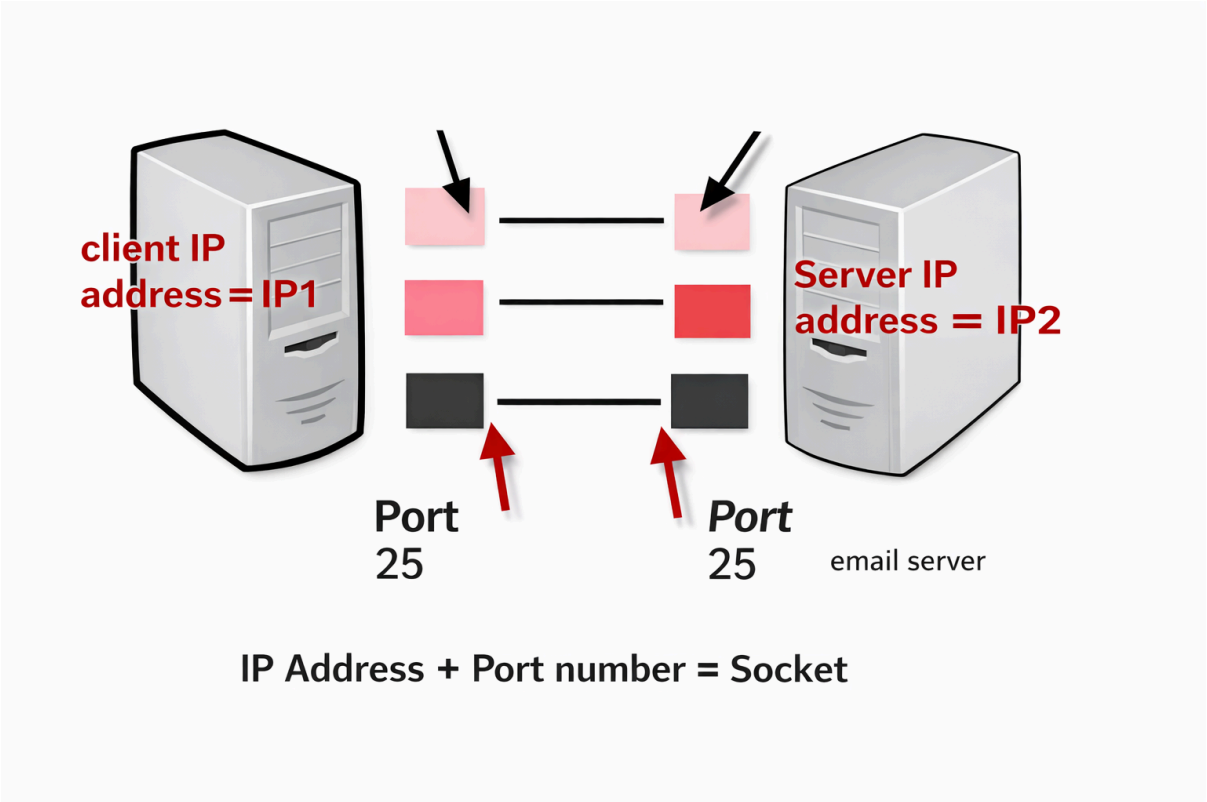
Quando due dispositivi comunicano attraverso una rete, non basta conoscere il loro indirizzo e la porta: serve anche un meccanismo che gestisca la connessione vera e propria tra i due programmi.
Questo meccanismo è la **socket**.

la socket rappresenta il punto software attraverso cui il programma comunica in rete.

- Nel caso *TCP* può rappresentare una connessione.
- Nel caso *UDP* rappresenta un punto di invio e ricezione di datagrammi.

La socket è l'oggetto software che permette a un programma di inviare e ricevere dati su una rete, sfruttando un indirizzo e una porta:

- L'indirizzo identifica il computer.
- La porta identifica il programma.
- La socket rappresenta la connessione tra i due.



Quando un'applicazione vuole comunicare con un'altra applicazione remota, crea una socket:

- Se usa *TCP*, stabilisce una connessione con il server.
- Se usa *UDP*, invia datagrammi verso un indirizzo e una porta di destinazione.

La comunicazione tra le due applicazioni avviene scambiando dati attraverso le loro rispettive socket.

Una socket locale è identificata almeno da un indirizzo, una porta e un protocollo di trasporto, per esempio TCP o UDP.
Nel caso di una connessione *TCP*, la comunicazione viene identificata in modo più preciso dalla combinazione di indirizzo e porta del mittente, indirizzo e porta del destinatario, e protocollo utilizzato.

Con questi elementi una socket identifica in maniera puntuale una specifica connessione tra due host.

Le fasi di una connessione con la socket sono:

- *Creazione*: il programma crea una socket e la lega a un indirizzo e una porta.
- *Connessione (per TCP)*:
  - Se l'host è un client, cerca il server e stabilisce una connessione.
  - Se l'host è un server, esso rimane in ascolto. Quando un client si connette, il server crea una nuova socket per quella connessione, con indirizzo e porta del client e del server.
- *Trasmissione*: i dati possono essere inviati e ricevuti attraverso la socket.
- *Chiusura*: una volta finita la comunicazione, la socket viene chiusa.

Le socket si classificano in due categorie, ciascuna caratterizzata da una propria modalità di comunicazione.

- **Datagram Socket (Socket *UDP*)**: questa tipologia di socket utilizza una connessione basata sul protocollo *UDP*. Ciò significa che l'invio dei dati avviene mediante il trasferimento di piccoli datagrammi, senza garantire il corretto ordine d'arrivo e la correttezza dell'informazione. Il client e il server non instaurano una vera e propria connessione, ma il client comunica direttamente con il server, quando vuole.
- **Stream Socket (Socket TCP)**: utilizzano una connessione basata sul protocollo *TCP*, quindi, connection-oriented, più controlli e affidabilità, ma introduce più overhead rispetto a una socket UDP.

Esempio
Sotto i passaggi quando un client comunica con uun browser l'apertura della pagina www.wikipedia.org:

- L'utente scrive www.wikipedia.org nel browser.
- Il sistema usa il DNS per ottenere l'indirizzo del server associato a quel nome.
- Il browser apre una socket *TCP* sulla porta 443 per comunicare con il server *HTTPS* (il protocollo HTTP ma sicuro) dell'indirizzo appena trovato.
- Il server risponde, i dati passano attraverso la socket.
- Il browser mostra la pagina web.
- Finito lo scambio, la socket viene chiusa.

Esempio di socket

Si esamina un esempio molto semplice di socket.
In questo esempio il client manda un numero al server e il server calcola il quadrato del numero e restituisce il risultato al client.
Sotto il codice del livello applicativo del server e dei client che sfruttano chiamate del livello di trasporto per mandare effettivamente i messaggi.
Le chiamate al livello di trasporto sono:

Lato client

- *CONNETTI (indirizzo, destPort):* Richiede una connessione con un host specificato (solo per protocolli orientati alla connessione, es. *TCP*). La funzione restituisce la socket specifica.
- *INVIA (socket, dati)*: Invia dati sulla connessione o sulla porta (*TCP* o *UDP*).
- R*ICEVI(socket)*: Riceve dati dal socket (sia *TCP* che *UDP*).
- *CHIUDI (socket)*: Termina la connessione (*TCP*).

Lato server

- *ATTENDI(localPort)*: Attende richieste di connessione sulla porta indicata (solo *TCP*). La funzione restituisce la socket specifica. Questa funzione rappresenta in modo semplificato l'insieme delle operazioni con cui il server si mette in ascolto e accetta una connessione.
- *INVIA (socket, dati)*: Invia dati sulla connessione o sulla porta (*TCP* o *UDP*).
- R*ICEVI(socket)*: Riceve dati dal socket (sia *TCP* che *UDP*).

- *CHIUDI(socket)*: Chiude la connessione o la porta.

Sotto si vedono le socket rispettivamente del client e del server.

Socket client

```
indirizzoServer = "127.0.0.1"
portaServer = 65432
numero = 5

socket = CONNETTI(indirizzoServer, portaServer)
INVIA(socket, numero)
risultato = RICEVI(socket)
CHIUDI(socket)
```

Socket server

```
portaServer = 65432

ITERA: #il server quando finisce un'operazione si rimette in ascolto
     socket = ATTENDI(portaServer)
     dato = RICEVI(socket)

     risultato = dato * dato
     INVIA(socket, risultato)
     CHIUDI(socket)
```