Laboratorio di Reti e Sistemi Distribuiti

2. Stack ISO/OSI ed Introduzione ai Socket in Linux

Roberto Marino, PhD[1]
roberto.marino@unime.it

[1]Dipartimento di Matematica, Informatica, Fisica e Scienze della Terra
Future Computing Research Laboratory
Università di Messina

Last Update: 26th February 2025

Indice

Introduzione

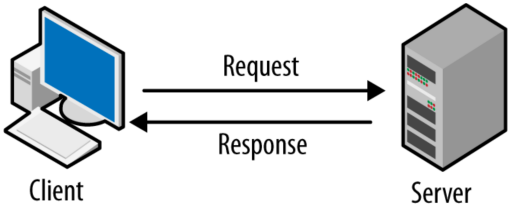
La differenza principale fra un'applicazione di rete e un programma normale è che quest'ultima per definizione concerne la comunicazione fra processi diversi, che in generale non girano neanche sulla stessa macchina. Questo già prefigura un cambiamento completo rispetto all'ottica del programma monolitico all'interno del quale vengono eseguite tutte le istruzioni, e chiaramente presuppone un sistema operativo multitasking in grado di eseguire più processi contemporaneamente.

Warning!

E' sempre possibile far comunicare due task che girano sulla stessa macchina attraverso lo stack TCP/IP!

Modello Client-Server

L'architettura fondamentale su cui si basa gran parte della programmazione di rete sotto Linux (e sotto Unix in generale) è il modello client-server caratterizzato dalla presenza di due categorie di soggetti, i programmi di servizio, chiamati server, che ricevono le richieste e forniscono le risposte, ed i programmi di utilizzo, detti client.

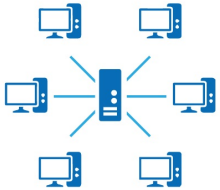


Server concorrenti ed iterativi

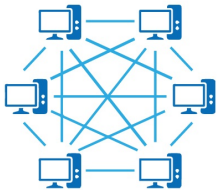
Normalmente si dividono i server in due categorie principali, e vengono detti **concorrenti o iterativi**, sulla base del loro comportamento. Un server iterativo risponde alla richiesta inviando i dati e resta occupato e non rispondendo ad ulteriori richieste fintanto che non ha fornito una risposta alla richiesta. Una volta completata la risposta il server diventa di nuovo disponibile. Un server concorrente al momento di trattare la richiesta crea un processo figlio (o un thread) incaricato di fornire i servizi richiesti, per porsi immediatamente in attesa di ulteriori richieste. In questo modo, con sistemi multitasking, più richieste possono essere soddisfatte contemporaneamente. Una volta che il processo figlio ha concluso il suo lavoro esso di norma viene terminato, mentre il server originale resta sempre attivo.

Modello peer-to-peer

Come abbiamo visto il tratto saliente dell'architettura client-server è quello della **preminenza del server rispetto ai client**, le architetture peer-to-peer si basano su un approccio completamente opposto che è quello di non avere nessun programma che svolga un ruolo preminente. Questo vuol dire che in generale ciascun programma viene ad agire come un nodo in una rete potenzialmente paritetica; ciascun programma si trova pertanto a ricevere ed inviare richieste ed a ricevere ed inviare risposte, e non c'è più la separazione netta dei compiti che si ritrova nelle architetture client-server.



A **Server** based Network

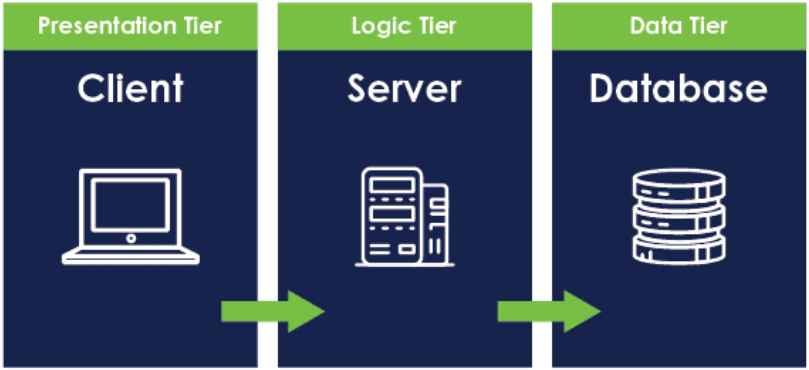


A **Peer-to-Peer** based Network

Modello Three-tier

Il modello **three-tier** si struttura, come dice il nome, su tre livelli. Il primo livello, quello dei client che eseguono le richieste e gestiscono l'interfaccia con l'utente, resta sostanzialmente lo stesso del modello client-server, ma la parte server viene suddivisa in due livelli, introducendo un middle-tier, su cui deve appoggiarsi tutta la logica di analisi delle richieste dei client per ottimizzare l'accesso al terzo livello, che è quello che si limita a fornire i dati dinamici che verranno usati dalla logica implementata nel middle-tier per eseguire le operazioni richieste dai client. In questo modo si può disaccoppiare la logica dai dati, replicando la prima, che è molto meno soggetta a cambiamenti ed evoluzione, e non soffre di problemi di sincronizzazione, e centralizzando opportunamente i secondi. In questo modo si può distribuire il carico ed accedere in maniera efficiente i dati.

Modell Three-tier



Modelli Broadcast e Multicast

In genere si parla di broadcast quando la trasmissione uno a molti è possibile fra qualunque nodo di una rete e gli altri, ed è supportata direttamente dalla tecnologia di collegamento utilizzata. L'utilizzo di questa forma di comunicazione da uno a molti però può risultare molto utile anche quando questo tipo di supporto non è disponibile (come ad esempio su Internet, dove non si possono contattare tutti i nodi presenti). In tal caso alcuni protocolli di rete (e quelli usati per Internet sono fra questi) supportano una variante del broadcast, detta multicast, in cui resta possibile fare una comunicazione uno a molti, in cui una applicazione invia i pacchetti a molte altre, in genere passando attraverso un opportuno supporto degli apparati ed una qualche forma di registrazione che consente la distribuzione della comunicazione ai nodi interessati.

Stack ISO/OSI

Una caratteristica comune dei protocolli di rete è il loro **essere strutturati in livelli sovrapposti**; in questo modo ogni protocollo di un certo livello realizza le sue funzionalità basandosi su un protocollo del livello sottostante. Questo modello di funzionamento è stato standardizzato dalla International Standards Organization (ISO) che ha preparato fin dal 1984 il Modello di Riferimento Open Systems Interconnection (OSI), strutturato in sette livelli.
`Il modello ISO/OSI` mira ad effettuare una classificazione completamente generale di ogni tipo di protocollo di rete; nel frattempo però era stato sviluppato anche un altro modello, relativo al protocollo TCP/IP, che è quello su cui è basata Internet, che è diventato uno standard de facto. Questo modello viene talvolta chiamato anche modello DoD (sigla che sta per Department of Defense), dato che fu sviluppato dall'agenzia ARPA per il Dipartimento della Difesa Americano.

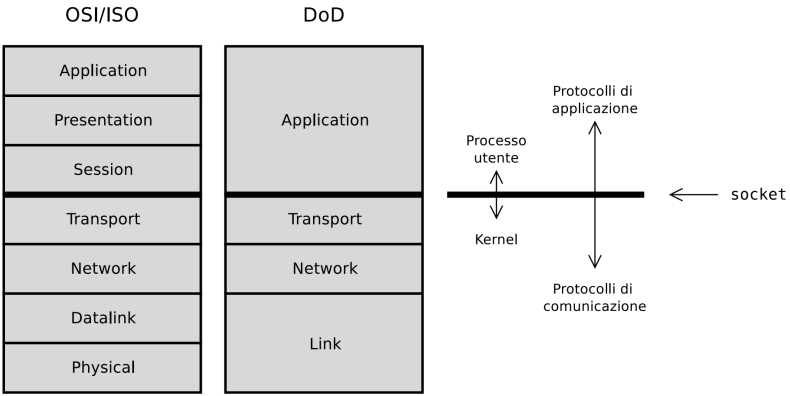


Figure: Struttura a livelli dei protocolli OSI e TCP/IP, con la relative corrispondenze e la divisione fra kernel space e user space

Livelli TCP/IP

Livello1: Applicazione (Es. HTTP, FTP, SMTP, DNS

É relativo all'interazione dell'utente con la rete, al formato dei dati. In genere questi vengono realizzati secondo il modello client-server, realizzando una comunicazione secondo un protocollo che è specifico di ciascuna applicazione.

Trasporto (Es. TCP, UDP)

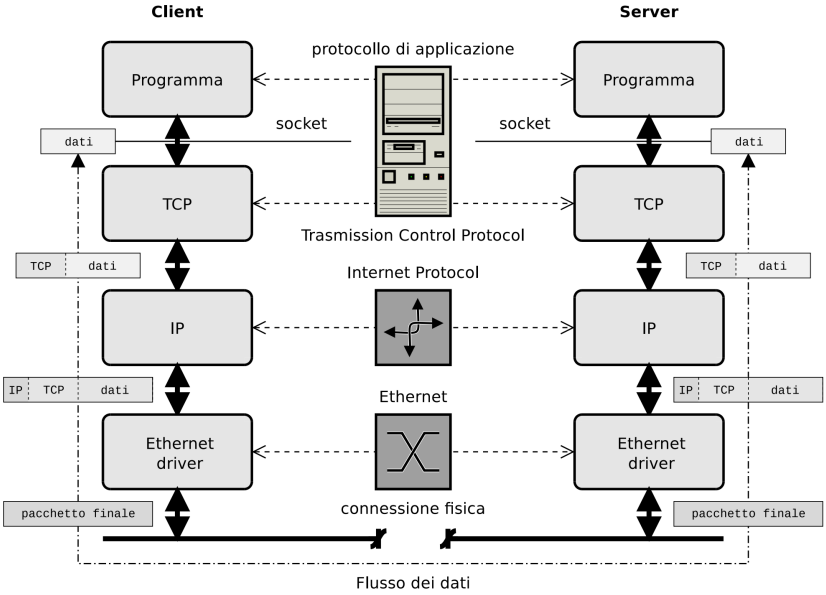
Fornisce la comunicazione **end-to-end** tra le due stazioni terminali su cui girano gli applicativi, regola il flusso delle informazioni, può fornire un trasporto affidabile, cioè con recupero degli errori o inaffidabile. I protocolli principali di questo livello sono il TCP e l'UDP.

Rete (Es. IPv4, IPv6, ICMP)

Si occupa dello smistamento dei singoli pacchetti su una rete complessa e interconnessa, a questo stesso livello operano i protocolli per il reperimento delle informazioni necessarie allo smistamento, per lo scambio di messaggi di controllo e per il monitoraggio della rete. Il protocollo su cui si basa questo livello è IP (sia nella attuale versione, IPv4, che nella nuova versione, IPv6).

Collegamento (ES. Ethernet, Ethercat, PPP)

È responsabile per l'interfacciamento al dispositivo elettronico che effettua la comunicazione fisica, gestendo l'invio e la ricezione dei pacchetti da e verso l'hardware.


Client
Server
protocollo di applicazione
Programma
Programma
socket
socket
dati
dati
TCP
TCP
Trasmission Control Protocol
TCP
dati
TCP
dati
Internet Protocol
IP
IP
IP
TCP
dati
IP
TCP
dati
Ethernet
Ethernet driver
Ethernet driver
pacchetto finale
connessione fisica
pacchetto finale
Flusso dei dati

TCP/IP Protocol Suite

Il TCP/IP è un insieme di protocolli diversi, che operano su 4 livelli diversi. Per gli interessi della programmazione di rete però **sono importanti principalmente i due livelli centrali, e soprattutto quello di trasporto**. La principale interfaccia usata nella programmazione di rete, quella dei socket, è infatti un'interfaccia nei confronti di quest'ultimi.

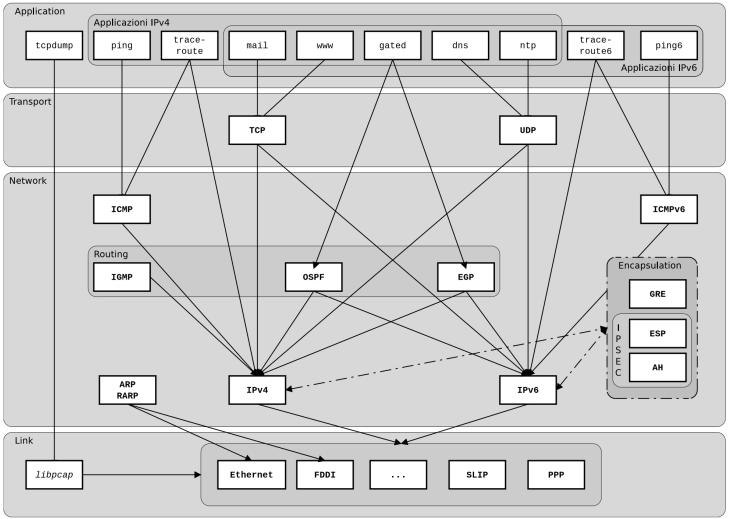


Figure: Panoramica sui vari protocolli che compongono la suite TCP/IP

Protocolli principali

IPv4

Internet Protocol version 4. È quello che comunemente si chiama IP. Ha origine negli anni '80 e da allora è la base su cui è costruita Internet. Usa indirizzi a 32 bit, e **formalizza tutte le informazioni di instradamento** per la trasmissione dei pacchetti sulla rete; tutti gli altri protocolli della suite (ecceto ARP e RARP e quelli specifici IPv6) vengono trasmessi attraverso di esso.

TCP

Trasmission Control Protocol. È un protocollo **orientato alla connessione** che provvede un trasporto affidabile per un flusso di dati bidirezionale fra due stazioni remote. Il protocollo ha cura di tutti gli aspetti del trasporto dei dati, come l'acknowledgment (il ricevuto), i timeout, la ritrasmissione, ecc. È usato dalla maggior parte delle applicazioni.

Protocolli principali

UDP

User Datagram Protocol. È un **protocollo senza connessione**, per l'invio di dati a pacchetti. Contrariamente al TCP il protocollo non è affidabile e non c'è garanzia che i pacchetti raggiungano la loro destinazione, si perdano, vengano duplicati, o abbiano un particolare ordine di arrivo.

ICMP

Internet Control Message Protocol. È il protocollo usato a livello 2 per **gestire gli errori e trasportare le informazioni di controllo** fra stazioni remote e instradatori (cioè fra host e router). I messaggi sono normalmente generati dal software del kernel che gestisce la comunicazione TCP/IP, anche se ICMP può venire usato direttamente da alcuni programmi come ping. A volte ci si riferisce ad esso come ICPMv4 per distinguerlo da ICMPv6.

Protocolli principali

ARP

Address Resolution Protocol. È il protocollo che **mappa un indirizzo IP in un indirizzo hardware sulla rete locale**. È usato in reti di tipo broadcast come Ethernet, Token Ring o FDDI che hanno associato un indirizzo fisico (il MAC address) alla interfaccia, ma non serve in connessioni punto-punto.

OSPF

Open Shortest Path First. È un protocollo di routing che permette ai router di **scambiarsi informazioni sullo stato delle connessioni** e dei legami che ciascuno ha con gli altri. Viene implementato direttamente sopra IP.

Header TCP

TCP Header				
Bits	0-15			16-31
0	Source port			Destination port
32	Sequence number			
64	Acknowledgment number			
96	Offset	Reserved	Flags	Window size
128	Checksum			Urgent pointer
160	Options			

Struttura del pacchetto TCP

L'intestazione TCP è composta dai seguenti campi:

- **Porta di origine (Source Port)** - 2 byte: Identifica la porta del mittente
- **Porta di destinazione (Destination Port)** - 2 byte: Identifica la porta del destinatario
- **Numero di sequenza (Sequence Number)** - 4 byte: Identifica il primo byte di dati nel segmento corrente. Viene utilizzato per riassemblare i dati in ordine corretto
- **Numero di acknowledgment (Acknowledgment Number)** - 4 byte: Contiene il prossimo numero di sequenza che il mittente si aspetta di ricevere. Viene utilizzato per confermare la ricezione dei dati
- **Lunghezza dell'intestazione (Data Offset)** - 4 bit: Indica la lunghezza dell'intestazione TCP in parole da 32 bit (4 byte). Questo campo è necessario perché l'intestazione può variare in lunghezza a causa delle opzioni

Struttura del pacchetto TCP

- **Flag (Control Bits)** - 9 bit: Include vari flag di controllo: URG (Urgent): Indica che il campo Urgent Pointer è significativo. ACK (Acknowledgment): Indica che il campo Acknowledgment Number è significativo. PSH (Push): Richiede che i dati vengano inviati immediatamente all'applicazione. RST (Reset): Resetta la connessione. SYN (Synchronize): Inizializza una connessione. FIN (Finish): Termina una connessione.
- **Finestra (Window Size)** - 2 byte: Indica la dimensione della finestra di ricezione, ovvero quanti byte il ricevitore è disposto a ricevere.
- **Checksum** - 2 byte: Utilizzato per il controllo degli errori nell'intestazione e nei dati.
- **Puntatore urgente (Urgent Pointer)** - 2 byte: Se il flag URG è impostato, questo campo indica la posizione dei dati urgenti nel segmento
- **Opzioni (Options)** - Variabile
- **Padding** - Variabile

Header IP

Bits	0–3	4–7	8–11	12–15	16–18	19–23	24–27	28–31
0	Version	Head Length	Type of Service		Total length (Packet)			
32	Identification				Flags	Fragment Spacing		
64	Lifespan (TTL)		Protocol		Header checksum			
96	Source Address							
128	Destination Address							
160	Options							

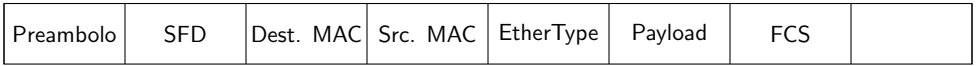
Struttura del pacchetto IP

- **Versione (Version)** - 4 bit: Indica la versione del protocollo IP (ad esempio, IPv4 o IPv6). Per IPv4, il valore è 4
- **Lunghezza dell'intestazione (IHL)** - 4 bit: Indica la lunghezza dell'intestazione IP in parole da 32 bit (4 byte). Il valore minimo è 5 (20 byte).
- **Tipo di servizio (Type of Service)** - 8 bit: Utilizzato per specificare la priorità e il tipo di servizio richiesto per il pacchetto (ad esempio, bassa latenza, alta affidabilità).
- **Lunghezza totale (Total Length)** - 16 bit: Indica la lunghezza totale del pacchetto IP, inclusi intestazione e dati, in byte. Il valore massimo è 65.535 byte.
- **Identificazione (Identification)** - 16 bit: Utilizzato per identificare i frammenti di un pacchetto IP. Tutti i frammenti di un pacchetto hanno lo stesso valore di identificazione.
- **Flag (Flags)** - 3 bit: Controlla la frammentazione del pacchetto: Bit 0: Riservato (deve essere 0). Bit 1 (DF - Don't Fragment): Se impostato, il pacchetto non deve essere frammentato. Bit 2 (MF - More Fragments): Se impostato, indica che ci sono ulteriori frammenti

Struttura del Pacchetto IP

- **Offset del frammento (Fragment Offset)** - 13 bit: Indica la posizione del frammento nel pacchetto originale, in unità di 8 byte.
- **Time to Live (TTL)** - 8 bit: Limita la durata del pacchetto nella rete. Viene decrementato di 1 ogni volta che il pacchetto passa attraverso un router. Se raggiunge 0, il pacchetto viene scartato.
- **Protocollo (Protocol)** - 8 bit: Indica il protocollo di livello superiore utilizzato nei dati (ad esempio, TCP = 6, UDP = 17, ICMP = 1)
- **Checksum dell'intestazione (Header Checksum)** - 16 bit: Utilizzato per il controllo degli errori nell'intestazione IP.
- **Indirizzo IP di origine (Source IP Address)** - 32 bit: Contiene l'indirizzo IP del mittente
- **Indirizzo IP di destinazione (Destination IP Address)** - 32 bit: Contiene l'indirizzo IP del destinatario.
- **Opzioni (Options)** - Variabile: Campo opzionale
- **Padding** - Variabile: Utilizzato per riempire l'intestazione IP fino a un multiplo di 32 bit.

Header Ethernet



Struttura della trama Ethernet

- **Preambolo (Preamble)** - 7 byte: Una sequenza di 7 byte (56 bit) utilizzata per sincronizzare i dispositivi di rete. Ogni byte ha il valore 0xAA (10101010 in binario).
- **SFD (Start Frame Delimiter)** - 1 byte: Un byte (0xAB) che indica l'inizio del frame.
- **Indirizzo MAC di destinazione (Destination MAC Address)** - 6 byte
- **Indirizzo MAC di origine (Source MAC Address)** - 6 byte
- **EtherType / Lunghezza (EtherType/Length)** - 2 byte: Se il valore è inferiore o uguale a 1500, indica la lunghezza del payload. Se il valore è superiore a 1500, indica il tipo di protocollo di livello superiore (ad esempio, IPv4 = 0x0800, IPv6 = 0x86DD).
- **Payload (Dati)** - 46-1500 byte: Contiene i dati effettivi trasmessi. Questo campo può includere un pacchetto IP, un segmento TCP, un datagramma UDP,
- **FCS (Frame Check Sequence)** - 4 byte: Checksum per il controllo di errore.

Incapsulamento

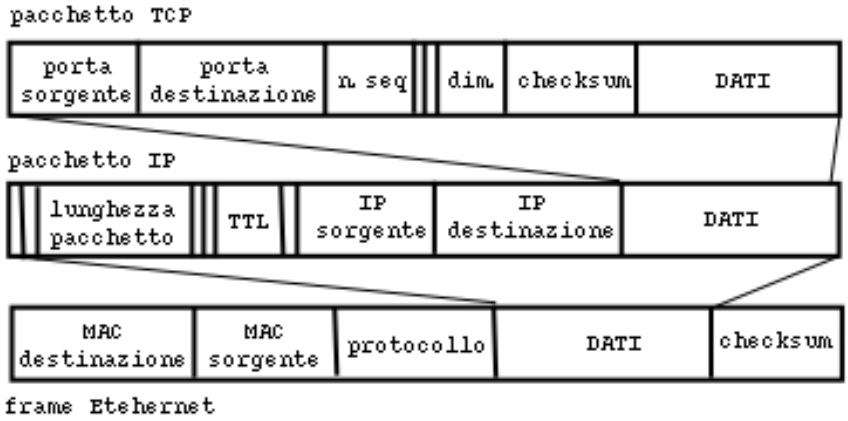


Figure: Incapsulamento TCP/IP/ETHERNET

Confronto Tcp/Ip/Ethernet

Table: Confronto tra TCP, IP ed Ethernet (RFC/IEEE)

Protocollo	RFC/IEEE	Tipo di Servizio	Affidabilità	Primitive Principali
TCP	RFC 793 *"Transmission Control Protocol"*	Orientato alla connessione	Affidabile	OPEN, SEND, RECEIVE, CLOSE
IP	RFC 791 *"Internet Protocol"*	Senza connessione	Non affidabile	SEND, DELIVER, ERROR (ICMP)
Ethernet	IEEE 802.3 RFC 894 *"IP over Ethernet"*	Frame-based	Non affidabile	SEND_FRAME, RECEIVE_FRAME

Primitive di Servizio TCP

Primitiva	Parametri	Descrizione
OPEN	Local Port Remote IP/Port	Inizializza una connessione (attiva/passiva).
SEND	Connection ID Data Push Flag Urgent Flag	Invia dati su una connessione esistente.
RECEIVE	Connection ID Buffer	Riceve dati dalla connessione.
CLOSE	Connection ID	Termina la connessione in modo ordinato.
ABORT	Connection ID	Termina bruscamente la connessione.
STATUS	Connection ID	Restituisce informazioni sullo stato della connessione.

Primitive di Servizio IP

Primitiva	Parametri	Descrizione
SEND	Source IP Destination IP Protocol TTL Data	Invia un datagramma IP.
DELIVER	Source IP Destination IP Protocol Data	Consegna i dati al protocollo di livello superiore.
ERROR	Type (ICMP) Code Source IP	Notifica errori (es. "Destination Unreachable").

Primitive di servizio Ethernet

Primitiva	Parametri	Descrizione
SEND_FRAME	Dest MAC Source MAC EtherType Data	Invia un frame Ethernet.
RECEIVE_FRAME	Buffer	Riceve un frame dal mezzo fisico.
COLLISION_DETECT	Nessuno	Gestisce le collisioni tramite CSMA/CD.

Concetto di Socket

Def: Socket

Un **socket** - descritto nel RFC 147 - è un oggetto software che permette l'invio e la ricezione di dati tra due endpoint, questi possono essere due host remoti (collegati tramite una rete) o due processi/threads locali (Inter-Process Communication)

Def: Network Socket

Un socket network è un meccanismo usato per scambiare informazione tra entità computazionali differenti **che sfrutta lo stack TCP/IP** attraverso un'interfaccia di rete fisica o virtuale.

Storia del socket

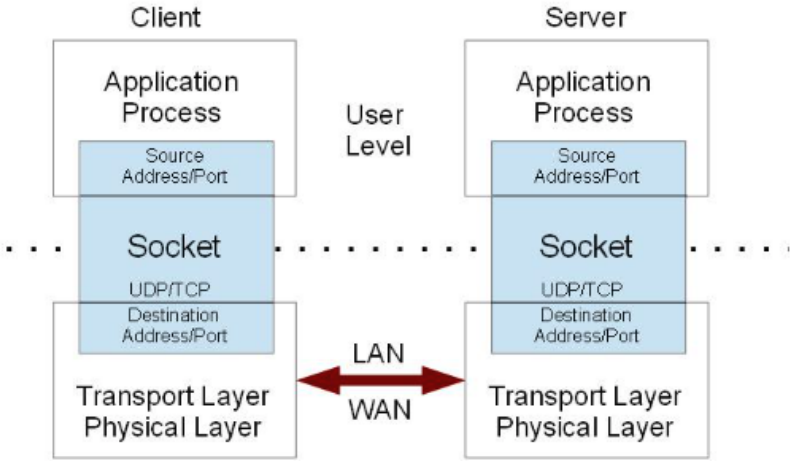
History

1971: Nasce il concetto di socket in ARPANET

1983: I socket vengono integrati come API nel sistema operativo BSD (Berkley Software Distribution), l'antesignano di UNIX

1990: Nasce il WWW ed i socket diventano di largo utilizzo

Socket: diagramma



Cosa succede dopo una chiamata a socket()?

Chiamata a socket

```
sockfd = socket(AF_INET,SOCK_STREAM,0);
```

In generale, quando un programma esegue una chiamata di sistema, **interrompe il kernel** (tramite un'interrupt software) e gli passa il controllo. Il kernel quindi esegue varie operazioni, come il salvataggio del contenuto dei registri e la verifica di eventuali errori nei parametri della chiamata di sistema. Infine, la funzione `sys_socket()` del kernel è responsabile della creazione del socket con l'indirizzo e il tipo di famiglia specificati, della ricerca di un descrittore di file inutilizzato e `della restituzione di questo numero allo spazio utente`.

Cosa succede dopo una chiamata a socket()?

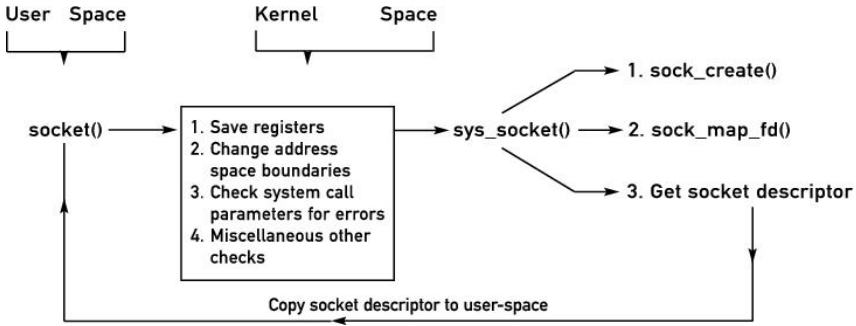


Figure: Chiamata a socket(): tra kernel space e user space

Creazione di un socket

Nello specifico quando un'applicazione chiama la funzione socket() il kernel esegue tre passaggi:

- Allocazione della struttura `struct socket`: l kernel crea una nuova istanza della struttura struct socket, che rappresenta il socket a livello di sistema operativo. Questa struttura contiene un puntatore a una struttura struct sock, che rappresenta il socket a livello di protocollo.
- Assegnazione di un file descriptor: Il kernel assegna un file descriptor al socket, che viene restituito all'applicazione. Questo file descriptor è collegato a un file object (`struct file`) nel kernel, che rappresenta il socket come un file.
- Inizializzazione del protocollo: Il kernel inizializza il protocollo specifico (ad esempio, TCP o UDP) associando il socket a una struttura `struct proto_ops`, che contiene le funzioni di callback per le operazioni sul socket (ad esempio, bind(), connect(), sendmsg()).

Everything is a file!

In Linux tutto è un file

I socket sono trattati come file in Unix e Linux perché il sistema operativo utilizza il concetto di "tutto è un file" (Everything is a file). Questo approccio unificato semplifica l'interazione con diversi tipi di risorse, come dispositivi hardware, file su disco, pipe, e, appunto, socket.

- **Uniformità'**: In Unix e Linux, l'accesso a molte risorse avviene attraverso operazioni su file (ad esempio, open(), read(), write(), close()). Trattare i socket come file permette di utilizzare le stesse funzioni di sistema per operare su file, pipe, dispositivi e socket, semplificando l'API per gli sviluppatori.
- **Astrazione**: Un file è un'astrazione che rappresenta una sorgente o destinazione di dati. I socket, essendo un meccanismo di comunicazione, si adattano perfettamente a questa astrazione.

Everything is a file!

- **Gestione tramite file descriptor**: Quando un socket viene creato, il kernel assegna un file descriptor (un intero non negativo) che rappresenta il socket. Questo file descriptor può essere utilizzato con funzioni come read(), write(), close(), proprio come un file tradizionale.
- In Linux, i socket sono integrati nel **Virtual File System (VFS)**, che fornisce un'interfaccia comune per gestire file, directory e altre risorse.

Integrazione del socket con il Virtual File System

Il **Virtual File System (VFS)** è uno strato di astrazione che permette al kernel di gestire file, directory e altre risorse in modo uniforme.

- Creazione del file descriptor: Quando un socket viene creato, il kernel assegna un file descriptor e crea un'istanza di struct file associata al socket.
- Binding delle operazioni di I/O: Le operazioni su file (ad esempio, read(), write(), close()) sono reindirizzate alle funzioni di callback definite in `struct file` per i socket.
- Interazione con lo stack di rete: Quando un'applicazione chiama una funzione come read() o write() su un socket, il kernel utilizza le funzioni di callback per interagire con lo stack di rete e gestire la comunicazione.

Kernel: Rappresentazione di un pacchetto

- **File**: `include/linux/skbuff.h`
- **Descrizione**: Rappresenta un pacchetto di rete (buffer di socket). Contiene i dati del pacchetto e i metadati necessari per la sua elaborazione attraverso i vari strati dello stack.
- **Campi principali**:

```
1  struct sk_buff {
2      struct sk_buff       *next;
3      struct sk_buff       *prev;
4      ktime_t              tstamp;
5      struct sock          *sk;
6      struct net_device    *dev;
7      unsigned int         len;
8      unsigned int         data_len;
9      __u16                protocol;
10     // ... molti altri campi ...
11 };
12
```

Kernel: Rappresentazione di una interfaccia di rete

- **File**: `include/linux/netdevice.h`
- **Descrizione**: Rappresenta un'interfaccia di rete (ad esempio, una scheda Ethernet o Wi-Fi). Contiene informazioni sull'interfaccia, come l'indirizzo MAC, lo stato, le funzioni di callback per la trasmissione e la ricezione dei pacchetti.
- **Campi principali**:

```
1 struct net_device {
2     char                    name[IFNAMSIZ];
3     unsigned long           state;
4     struct net_device_ops *netdev_ops;
5     unsigned int            mtu;
6     unsigned char           *dev_addr; // Indirizzo MAC
7     // ... molti altri campi ...
8 };
9
```

Rappresentazione di un socket in kernel space

- **File**: `include/net/sock.h`
- **Descrizione**: Rappresenta una socket a livello di kernel. Contiene informazioni sullo stato della connessione, gli indirizzi di origine e destinazione, le code di trasmissione e ricezione, e altre informazioni relative al protocollo di trasporto (TCP, UDP, ecc.).
- **Campi principali**:

```
1 struct sock {
2     struct sock_common  __sk_common;
3     struct sk_buff     *sk_receive_queue;
4     struct sk_buff     *sk_write_queue;
5     struct proto       *sk_prot;
6     unsigned int        sk_state;
7     // ... molti altri campi ...
8 };
9
```

Rappresentazione di un socket in user space

- **File**: `include/linux/net.h`
- **Descrizione**: Rappresenta una socket a livello di sistema operativo. Contiene un puntatore a una struttura `sock` e metadati aggiuntivi per la gestione delle operazioni di I/O.
- **Campi principali**:

```
1 struct socket {
2     socket_state        state;
3     struct sock         *sk;
4     const struct proto_ops *ops;
5     // ... altri campi ...
6 };
7
```

Rappresentazione di un indirizzo IPV4

- **File**: `include/linux/in.h`
- **Descrizione**: Rappresenta un indirizzo IPv4. Viene utilizzata per memorizzare indirizzi IP e porte nel formato richiesto dalle socket.
- **Campi principali**:

```
1 struct sockaddr_in {
2     __kernel_sa_family_t sin_family; // Famiglia di indirizzi (AF_INET)
3     __be16               sin_port;   // Porta
4     struct in_addr       sin_addr;   // Indirizzo IP
5     // ... altri campi ...
6 };
7
```

Rappresentazione delle operazioni su socket

- **File**: `include/linux/net.h`
- **Descrizione**: Contiene le operazioni (funzioni di callback) che possono essere eseguite su una socket, come `bind()`, `connect()`, `send()`, `recv()`, ecc.
- **Campi principali**:

```
1 struct proto_ops {
2     int (*bind)(struct socket *sock, struct sockaddr *myaddr, int
      sockaddr_len);
3     int (*connect)(struct socket *sock, struct sockaddr *vaddr,
      int sockaddr_len, int flags);
4     int (*sendmsg)(struct socket *sock, struct msghdr *m, size_t
      total_len);
5     // ... altre operazioni ...
6 };
7
```

Rappresentazione per Ipv4 (routing, cache indirizzi)

- **File**: `include/net/netns/ipv4.h`
- **Descrizione**: Contiene la configurazione specifica per IPv4 all'interno di uno spazio dei nomi di rete.
- **Campi principali**:

```
1 struct netns_ipv4 {
2     struct fib_table *fib_main; // Tabella di routing principale
3     struct inet_peer *peers;    // Cache per gli indirizzi IP
4     // ... altri campi ...
5 };
6
```

Rappresentazione di un protocollo

- **File**: `include/net/sock.h`
- **Descrizione**: Rappresenta un protocollo di trasporto (ad esempio, TCP o UDP). Contiene le funzioni di callback specifiche per il protocollo.
- **Campi principali**:

```
1 struct proto {
2     const char *name; // Nome del protocollo (ad esempio, "TCP")
3     int (*connect)(struct sock *sk, struct sockaddr *uaddr, int
      addr_len);
4     int (*sendmsg)(struct sock *sk, struct msghdr *msg, size_t len
      );
5     // ... altre operazioni ...
6 };
7
```