Laboratorio di Reti e Sistemi Distribuiti

3: Socket Programming 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: 4th March 2025

Man Pages

Le **man pages** (pagine di manuale) sono la documentazione ufficiale integrata nei sistemi Unix e Linux. Forniscono informazioni dettagliate su:

- Comandi da terminale (es. `ls`, `grep`)
- Chiamate di sistema (es. `socket`, `fork`)
- File di configurazione (es. `/etc/passwd`)
- Librerie e funzioni di programmazione

Sezioni delle Man Pages

Le man pages **sono organizzate in 8 sezioni numerate**:

Sezione	**Contenuto**
1	Comandi utente (es. `man 1 ls`)
2	Chiamate di sistema (es. `man 2 open`)
3	Funzioni di libreria (es. `man 3 printf`)
4	File speciali (dispositivi)
5	Formati file e convenzioni
6	Giochi
7	Varie
8	Comandi di amministrazione

Man Pages: Esempi

```
man <nome>
```

Esempio:

```
man ls       # Documentazione del comando ls
man 2 fork   # Documentazione della system call fork
```

```
man -k <keyword>
```

Esempio:

```
man -k "socket"
```

Origini: BSD

L'interfaccia socket fu introdotta con extbfBSD 4.2 Unix, sviluppato dall'Università della California, Berkeley, nel 1983. Questo modello di comunicazione fu pensato per supportare la nascente rete **ARPANET** e i protocolli **TCP/IP**, che sarebbero poi diventati la base di Internet.
L'API BSD Sockets forniva funzioni chiave come:

- `socket()` - creazione di un socket
- `bind()` - associazione a un indirizzo e una porta
- `listen()` - impostazione in ascolto per connessioni in ingresso
- `accept()` - accettazione di connessioni in arrivo
- `connect()` - connessione a un altro socket
- `send()`, `recv()` - invio e ricezione di dati

L'avvento di Linux

Quando il progetto **Linux** nacque nel 1991, la compatibilità con BSD era una priorità, quindi il sottosistema di rete fu progettato attorno al modello BSD Sockets. Con il tempo, il kernel Linux introdusse miglioramenti significativi:

- **Supporto a protocolli diversi da TCP/IP**: oltre a `AF_INET` (IPv4) e `AF_INET6` (IPv6), Linux supporta `AF_UNIX` (comunicazione locale) e `AF_NETLINK` (comunicazione kernel-user space).
- **Socket non bloccanti e asincroni**: con l'introduzione di `epoll` nel kernel 2.6 (2003), la gestione di molte connessioni simultanee è diventata più efficiente rispetto a `select()` e `poll()`.
- **Zero-Copy Networking**: miglioramenti come `sendfile()`, `splice()`, e **io_uring** (kernel 5.1, 2019) hanno ridotto l'overhead della copia di dati tra kernel e spazio utente.
- **Sicurezza**: con **seccomp**, **AppArmor** e **SELinux**, sono stati introdotti controlli più rigorosi sulle connessioni di rete.

Modernizzazione dei Socket

L'evoluzione continua dell'interfaccia socket ha portato a nuovi miglioramenti:

- **eBPF (Extended Berkeley Packet Filter)**: consente il filtraggio e il monitoraggio dei pacchetti a livello di kernel con un overhead minimo.
- **TLS nel kernel** (dal 2017): supporto alla crittografia a livello di kernel per migliorare le prestazioni.
- **Supporto per QUIC e HTTP/3**: il kernel Linux ha introdotto il supporto sperimentale per QUIC, il nuovo protocollo di trasporto sviluppato da Google.

... in evoluzione continua

L'interfaccia socket in Linux è nata come un'implementazione del modello BSD, ma si è evoluta significativamente in termini di efficienza e sicurezza. Oggi continua a essere un pilastro della programmazione di rete, con ottimizzazioni per applicazioni moderne come microservizi e cloud computing.

La funzione socket()

La funzione `socket` è utilizzata per creare un nuovo socket. Il suo prototipo è il seguente:

```
1 int socket(int domain, int type, int protocol);
```

- `domain`: Specifica il dominio di comunicazione (es. `AF_INET` per IPv4).
- `type`: Specifica il tipo di socket (es. `SOCK_STREAM` per TCP).
- `protocol`: Specifica il protocollo da utilizzare (di solito 0 per il protocollo predefinito).
- **Valore di ritorno**: Restituisce un file descriptor del socket in caso di successo, -1 in caso di errore.

Domain: Domini di comunicazione

Dominio	Descrizione
AF_UNIX	Comunicazione tra processi sulla stessa macchina (file system path).
AF_INET	Comunicazione IPv4.
AF_INET6	Comunicazione IPv6.
AF_NETLINK	Comunicazione tra kernel e spazio utente.
AF_PACKET	Accesso diretto ai pacchetti di rete (link layer).
AF_BLUETOOTH	Comunicazione con dispositivi Bluetooth.

Packet Family vs Address Family

A volte si trova PF_INET (Packet Family) invece di AF_INET (Address Family). Si tratta di diciture praticamente equivalenti introdotte dallo standard Posix per specificare casi di non interesse per il corso!

Type: Tipi di socket

Tipo	**Descrizione**
`SOCK_STREAM`	Socket orientato ai byte, affidabile (es. TCP).
`SOCK_DGRAM`	Socket basato su messaggi, non affidabile (es. UDP).
`SOCK_RAW`	Accesso diretto ai protocolli di rete di basso livello.
`SOCK_SEQPACKET`	Socket orientato ai pacchetti, affidabile.
`SOCK_RDM`	Socket affidabile ma non ordinato.
`SOCK_NONBLOCK`	Le funzioni di lettura e scrittura sul f.d. non bloccano il processo
`SOCK_CLOEXEC`	In caso di `execve()` il f.d. non sarà ereditato dal figlio

Domani/Type combinations

Famiglia	Tipo				
	SOCK_STREAM	SOCK_DGRAM	SOCK_RAW	SOCK_RDM	SOCK_SEQPACKET
AF_UNIX	si	si	–	–	si[3]
AF_LOCAL	sinonimo di AF_UNIX				
AF_INET	TCP	UDP	IPv4	–	–
AF_INET6	TCP	UDP	IPv6	–	–
AF_IPX	–	si	–	–	–
AF_NETLINK	–	si	si	–	–
AF_X25	–	–	–	–	si
AF_AX25	–	si	si	–	si
AF_APPLETALK	–	si	si	–	–
AF_PACKET	–	si	si	–	–
AF_KEY	–	–	si	–	–
AF_IRDA	si	si	si	–	si
AF_NETROM	–	–	–	–	si
AF_ROSE	–	–	–	–	si
AF_RDS	–	–	–	–	si
AF_ECONET	–	si	–	–	–

Tabella 14.2: Combinazioni valide di dominio e tipo di protocollo per la funzione socket.

Funzioni base: bind()

Associa un socket a un indirizzo e una porta specifica.

```
1 int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

- **sockfd**: File descriptor del socket creato con socket()
- **addr**: Puntatore a struttura sockaddr contenente:
  - `sa_family`: Famiglia di indirizzi (deve matchare domain del socket)
  - `sa_data`: Indirizzo e porta specifici
- **addrlen**: Lunghezza in byte della struttura indirizzo

Funzioni base: listen()

Mette il socket in ascolto per connessioni in entrata.

```
1 int listen(int sockfd, int backlog);
```

- **sockfd**: File descriptor del socket già associato con bind()
- **backlog**: Numero massimo di connessioni pendenti nella coda
  - Valori tipici: 5-10
  - Valori massimi dipendenti dal sistema (usare SOMAXCONN)

Funzioni base: accept()

Accetta una connessione in entrata.

```
1 int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

- **sockfd**: Socket in ascolto
- **addr**: Puntatore a struttura che riceverà l'indirizzo del client
- **addrlen**: Puntatore a variabile che specifica la dimensione della struttura
- **Return**: Nuovo file descriptor per la connessione accettata

Funzioni base: connect()

Connette un socket a un server remoto.

```
1 int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen
    );
```

- **addr**: Struttura contenente indirizzo IP e porta del server
- **addrlen**: Dimensione della struttura indirizzo

Funzioni di trasferimento dati: send()/recv()

Per socket orientati alla connessione (TCP).

```
1 ssize_t send(int sockfd, const void *buf, size_t len, int flags);
2 ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

- **buf**: Buffer contenente i dati da inviare/ricevere
- **len**: Dimensione del buffer
- **flags**: Flag di controllo (0 per default)
  - `MSG_OOB`: Dati urgenti
  - `MSG_PEEK`: Legge dati senza rimuoverli dal buffer

Funzioni di trasferimento dati: sendto()/recvfrom()

Per socket senza connessione (UDP).

```
1 ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
2                const struct sockaddr *dest_addr, socklen_t addrlen);
3
4 ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
5                  struct sockaddr *src_addr, socklen_t *addrlen);
```

- **dest_addr**: Indirizzo del destinatario (sendto)
- **src_addr**: Indirizzo del mittente (recvfrom)

Funzioni di supporto: ntohs()/htons()

Conversione byte order per porte.

```
1 uint16_t htons(uint16_t hostshort);  // Host to Network Short
2 uint16_t ntohs(uint16_t netshort);   // Network to Host Short
```

- Necessarie per garantire l'ordine dei byte corretto nella rete
- Esempio:

```
1 address.sin_port = htons(8080);  // Converte 8080 in network byte
      order
```

inet_pton()

Converte indirizzi IP da testo a formato binario.

```
1 int inet_pton(int af, const char *src, void *dst);
```

- **af**: Address family (AF_INET o AF_INET6)
- **src**: Stringa contenente l'indirizzo IP (es. "192.168.1.1")
- **dst**: Puntatore a struttura che riceve l'indirizzo binario
- Return: 1 (successo), 0 (formato non valido), -1 (errore)

Funzioni di supporto: perror()

Stampa un messaggio di errore descrittivo.

```
1 void perror(const char *s);
```

- **s**: Stringa descrittiva da anteporre al messaggio di errore
- Esempio:

```
1 if (listen(sockfd, 5) == -1) {
2     perror("listen failed");
3     exit(EXIT_FAILURE);
4 }
```

Funzioni di supporto: close()

Chiude un file descriptor.

```
1 int close(int sockfd);
```

- Importante per liberare risorse di sistema
- Deve essere chiamato per tutti i socket aperti

Funzioni di supporto: memset()

Inizializza una zona di memoria.

```
1 void *memset(void *s, int c, size_t n);
```

- Utilizzo tipico per pulire le strutture:

```
1 struct sockaddr_in serv_addr;
2 memset(&serv_addr, 0, sizeof(serv_addr));  // Azzera tutta la
      struttura
```

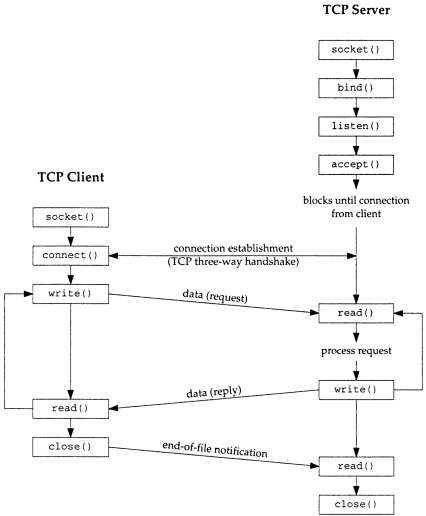
Memorizzare indirizzi IPv4

La struttura `sockaddr_in` è utilizzata per memorizzare un indirizzo IPv4.

```
struct sockaddr_in {
    sa_family_t    sin_family; // Famiglia di indirizzi (es. AF_INET)
    in_port_t      sin_port;   // Porta (in formato di rete)
    struct in_addr sin_addr;   // Indirizzo IP (in formato di rete)
    char           sin_zero[8]; // Padding per allineamento
};
```

- `sin_family`: Famiglia di indirizzi (es. `AF_INET`).
- `sin_port`: Numero di porta in formato di rete (usa `htons` per convertire).
- `sin_addr`: Struttura che contiene l'indirizzo IP.
- `sin_zero`: Padding per allineare la struttura a 16 byte.

TCP Client/Server Architecture



Inclusione delle librerie

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
```

Queste sono le librerie necessarie per il funzionamento del server:

- `stdio.h`: Fornisce funzioni di input/output.
- `stdlib.h`: Fornisce funzioni di utilità generale come `exit`.
- `string.h`: Fornisce funzioni per la manipolazione delle stringhe.
- `unistd.h`: Fornisce funzioni di sistema come `read` e `write`.
- `arpa/inet.h`: Fornisce funzioni per la gestione degli indirizzi di rete.

Definizione della porta e main()

```
1 #define PORT 8080
```

La porta su cui il server ascolta le connessioni in entrata è definita come 8080.

```
1 int main() {
2     int server_fd, new_socket;
3     struct sockaddr_in address;
4     int addrlen = sizeof(address);
5     char buffer[1024] = {0};
6     char *hello = "Hello from server";
```

La funzione principale del server. Qui vengono dichiarate le variabili:

- `server_fd`: File descriptor del socket del server.
- `new_socket`: File descriptor del socket per la connessione accettata.
- `address`: Struttura che contiene l'indirizzo del server.
- `addrlen`: Lunghezza dell'indirizzo.
- `buffer`: Buffer per memorizzare i dati ricevuti.
- `hello`: Messaggio da inviare al client.

Creazione del Socket e configurazione indirizzo

```
1    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
2        perror("socket failed");
3        exit(EXIT_FAILURE);
4    }
```

Crea un socket TCP (`SOCK_STREAM`) utilizzando il dominio IPv4 (`AF_INET`). Se la creazione fallisce, viene stampato un messaggio di errore e il programma termina.

```
1    address.sin_family = AF_INET;
2    address.sin_addr.s_addr = INADDR_ANY;
3    address.sin_port = htons(PORT);
```

Configura l'indirizzo del server:

- `sin_family`: Imposta il dominio di indirizzamento a IPv4.
- `sin_addr.s_addr`: Imposta l'indirizzo IP a `INADDR_ANY`, che significa che il server accetterà connessioni su qualsiasi interfaccia di rete.
- `sin_port`: Imposta la porta su cui il server ascolta, convertendola in formato di rete con `htons`.

Binding

```
1     if (bind(server_fd, (struct sockaddr *)&address, sizeof(address))
      < 0) {
2         perror("bind failed");
3         close(server_fd);
4         exit(EXIT_FAILURE);
5     }
```

Associa il socket all'indirizzo e alla porta specificati. Se il binding fallisce, viene stampato un messaggio di errore, il socket viene chiuso e il programma termina.

Listening

```
1     if (listen(server_fd, 3) < 0) {
2         perror("listen");
3         close(server_fd);
4         exit(EXIT_FAILURE);
5     }
```

Mette il socket in ascolto per connessioni in entrata, con una coda massima di 3 connessioni pendenti. Se fallisce, viene stampato un messaggio di errore, il socket viene chiuso e il programma termina.

Accept

```
1     if ((new_socket = accept(server_fd, (struct sockaddr *)&address, (
      socklen_t*)&addrlen)) < 0) {
2         perror("accept");
3         close(server_fd);
4         exit(EXIT_FAILURE);
5     }
```

Accetta una connessione in entrata. Se l'accettazione fallisce, viene stampato un messaggio di errore, il socket viene chiuso e il programma termina.

Read, Send and Close

```
1     read(new_socket, buffer, 1024);
2     printf("%s\n", buffer);
```

Legge il messaggio inviato dal client e lo stampa sulla console.

```
1     send(new_socket, hello, strlen(hello), 0);
2     printf("Hello message sent\n");
```

Invia un messaggio di risposta al client.

```
1     close(new_socket);
2     close(server_fd);
3     return 0;
4 }
```

Chiude i socket e termina il programma.

inet_pton()

```
1    if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
2        printf("\nInvalid address/ Address not supported \n");
3        return -1;
4    }
```

Converte l'indirizzo IP da formato stringa a formato binario. Se la conversione fallisce, viene stampato un messaggio di errore e il programma termina.

Significato di 0.0.0.0

Definizione

L'indirizzo IP 0.0.0.0 è un indirizzo IPv4 speciale con significati specifici a seconda del contesto in cui viene utilizzato.

Binding: "Tutte le interfacce" o "Qualsiasi indirizzo di rete"

- Quando un server si associa (binding) a 0.0.0.0, significa che è in ascolto su tutte le interfacce di rete disponibili (es. Ethernet, Wi-Fi, loopback/localhost 127.0.0.1, ecc.).
- Esempio: Se un server web si associa a 0.0.0.0:80, accetterà connessioni da dispositivi esterni (tramite l'IP pubblico della macchina), da altri programmi sulla stessa macchina (tramite 127.0.0.1 o localhost) o da qualsiasi altra interfaccia di rete collegata alla macchina.
- Al contrario: Se un server si associa a 127.0.0.1:80, accetterà solo connessioni dalla macchina locale.

Significato di 0.0.0.0

Rotta Predefinita nelle Tabelle di Routing

- Nelle tabelle di routing, 0.0.0.0 rappresenta spesso la rotta predefinita (il percorso "catch-all" per traffico che non corrisponde a nessun'altra rotta specifica).
- Esempio: Una voce in una tabella di routing come 0.0.0.0/0 indica il gateway predefinito, dirigendo il traffico verso il prossimo hop quando non esiste una rotta specifica (provate a digitare il comando `route -n` oppure `ip route`)

Attenzione!!!

0.0.0.0 $\neq$ "nessun IP"

Significa esplicitamente "tutti gli IP/interfacce"

Qualche esempio di utilizzo di 0.0.0.0

```
// Il server    in ascolto su tutte le interfacce (0.0.0.0:8080)
serv_addr.sin_addr.s_addr = INADDR_ANY; // INADDR_ANY corrisponde a
    0.0.0.0
```

```
$ netstat -tuln
Proto Recv-Q Send-Q Indirizzo Locale   Indirizzo Remoto   Stato
tcp        0      0 0.0.0.0:22        0.0.0.0:*         LISTEN   # SSH
    in ascolto su tutte le interfacce
tcp        0      0 127.0.0.1:5432    0.0.0.0:*         LISTEN   #
    PostgreSQL in ascolto solo localmente
```

```
$ ip route
default via 192.168.1.1 dev eth0  # 0.0.0.0/0 -> gateway predefinito
192.168.1.0/24 dev eth0 proto kernel scope link src 192.168.1.100
```

Quale differenza tra listen() ed accept()?

La funzione `listen()` prepara un socket TCP ad accettare connessioni in entrata dopo la creazione con `socket()` e il binding con `bind()`.

`int sockfd` File descriptor del socket

`int backlog` Dimensione massima della coda di connessioni pendenti

```
1 listen(sockfd, 5);  // Coda fino a 5 connessioni
```

Funzionamento

- Mette il socket in **stato passivo** (ascolto, stato LISTEN).
- Definisce una coda di connessioni pendenti per gestire richieste non ancora accettate.
- **Non blocca** l'esecuzione del programma (è una semplice configurazione del socket).

Quale differenza tra listen() ed accept()?

La funzione `accept()` accetta una connessione dalla coda creata da `listen()`, completando l'handshake TCP.

`int sockfd` File descriptor del socket in ascolto

`struct sockaddr *addr` Struttura per l'indirizzo del client (opzionale)

`socklen_t *addrlen` Dimensione della struttura (opzionale)

```
1 struct sockaddr_in client_addr;
2 socklen_t client_len = sizeof(client_addr);
3 int client_fd = accept(sockfd, (struct sockaddr*)&client_addr, &
    client_len);
```

Funzionamento

- **Blocca** l'esecuzione finché non arriva una connessione (a meno che il socket non sia impostato come non bloccante).
- **Crea un nuovo socket** dedicato alla comunicazione con il client specifico
- Il socket originale resta in ascolto per nuove connessioni.

Datagramma vs Pacchetto

Pacchetto (Packet), RFC 791

- Livello OSI: Strato di rete (Livello 3, es: IP).
- Unità fondamentale di dati nello strato di rete. (Level 3 PDU)
- Payload: Dati trasportati (es: segmento TCP o datagramma UDP).

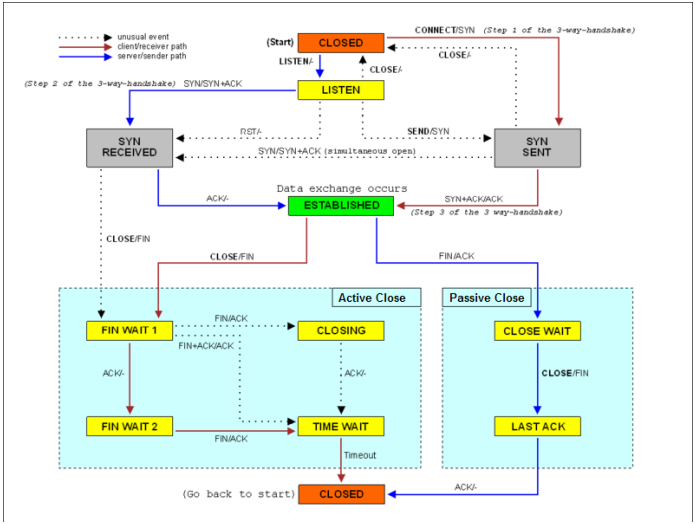
Datagram (Datagramma), RFC 768

- Livello OSI: Strato di trasporto (Livello 4, UDP)
- Unità di dati **indipendente e autocontenuta**, tipica dei protocolli senza connessione (Level 4 PDU)
- **I messaggi UDP sono chiamati datagrammi**

Segment (Segmento)

Per omogeneità di trattazione aggiungiamo che per TCP si parla di **segmento**

TCP/UDP State Machine



TCP State Machine

Stato	Descrizione
LISTEN	Socket in ascolto per connessioni in entrata
SYN_SENT	Client ha inviato SYN in attesa di risposta
SYN_RCVD	Server ha ricevuto SYN e inviato SYN-ACK
ESTABLISHED	Connessione attiva e operativa
FIN_WAIT1	Iniziata chiusura, in attesa di ACK
FIN_WAIT2	Ricevuto ACK, in attesa di FIN finale
TIME_WAIT	Connessione chiusa correttamente
CLOSE_WAIT	Attesa di chiusura corretta

UDP State Machine (pseudostati)

Stato	**Descrizione**
`UNCONN`	Socket UDP non connesso
`ESTAB`	Socket UDP connesso
`CLOSED`	Socket chiuso

Netstat e visualizzazione stati

```
# Per TCP
netstat -atnp | grep ':8080'
ss -atnp 'sport = :8080' # simile a netsta

# Per UDP
netstat -aunp | grep ':8080'
ss -aunp 'sport = :8080'

# Socket monitoring
watch "netstat -tan | grep -E 'TIME_WAIT|CLOSE_WAIT'"
```