

# Reti informatiche cod. 545II [9 CFU]

Corso di Laurea in Ingegneria Informatica

---

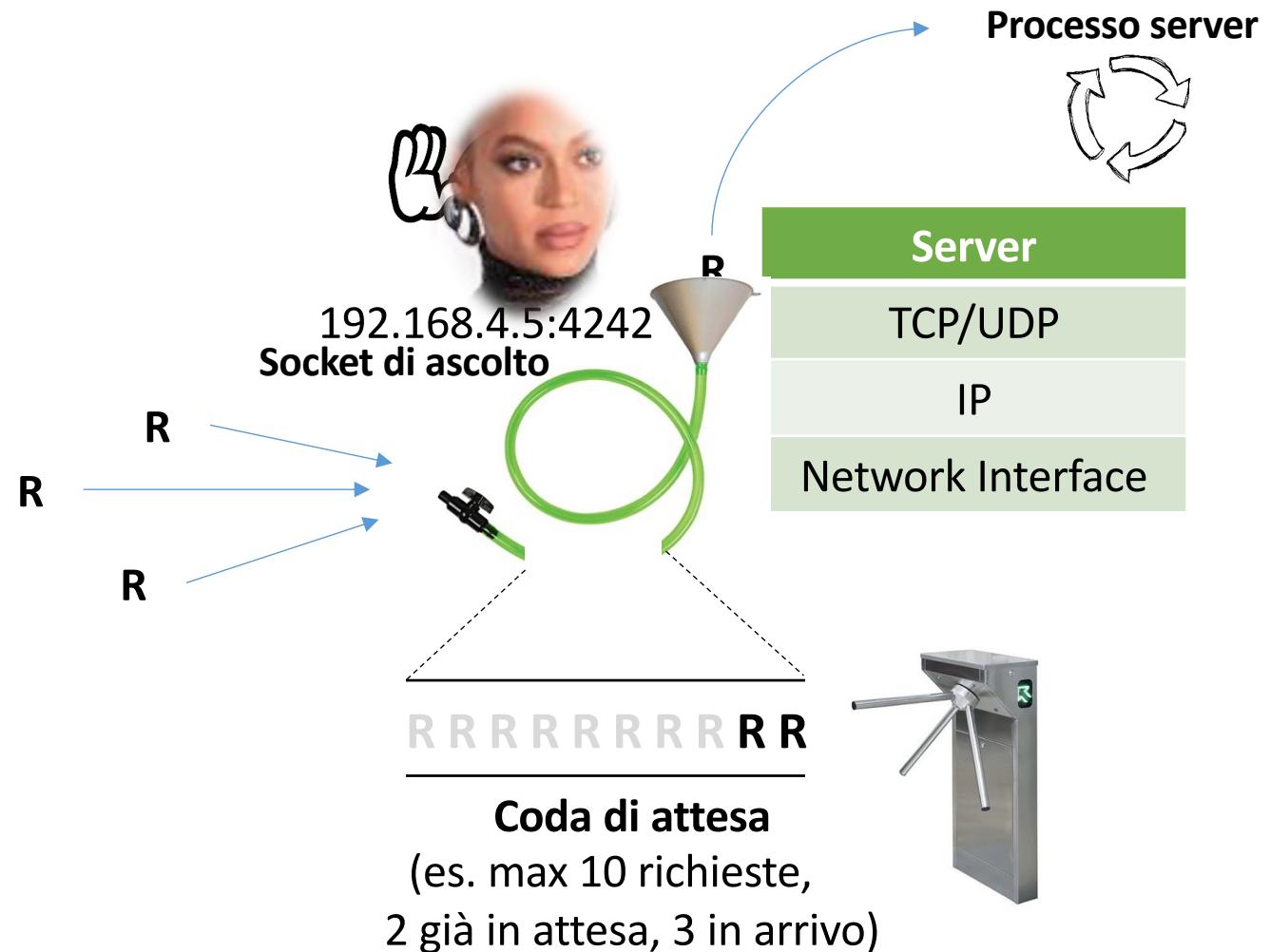
**Laboratorio e Programmazione di Rete**  
A. A. 2021/2022

Francesco Pistolesi, PhD  
Dipartimento di Ingegneria dell'Informazione  
[francesco.pistolesi@unipi.it](mailto:francesco.pistolesi@unipi.it)

# Lezione 7

Programmazione distribuita in C  
Parte 2

# Dopo `listen()`



# Primitiva **accept()**

- **Accetta una** richiesta di connessione pervenuta sul socket
  - Anche in questo caso, ha senso solo sui socket **SOCK\_STREAM**

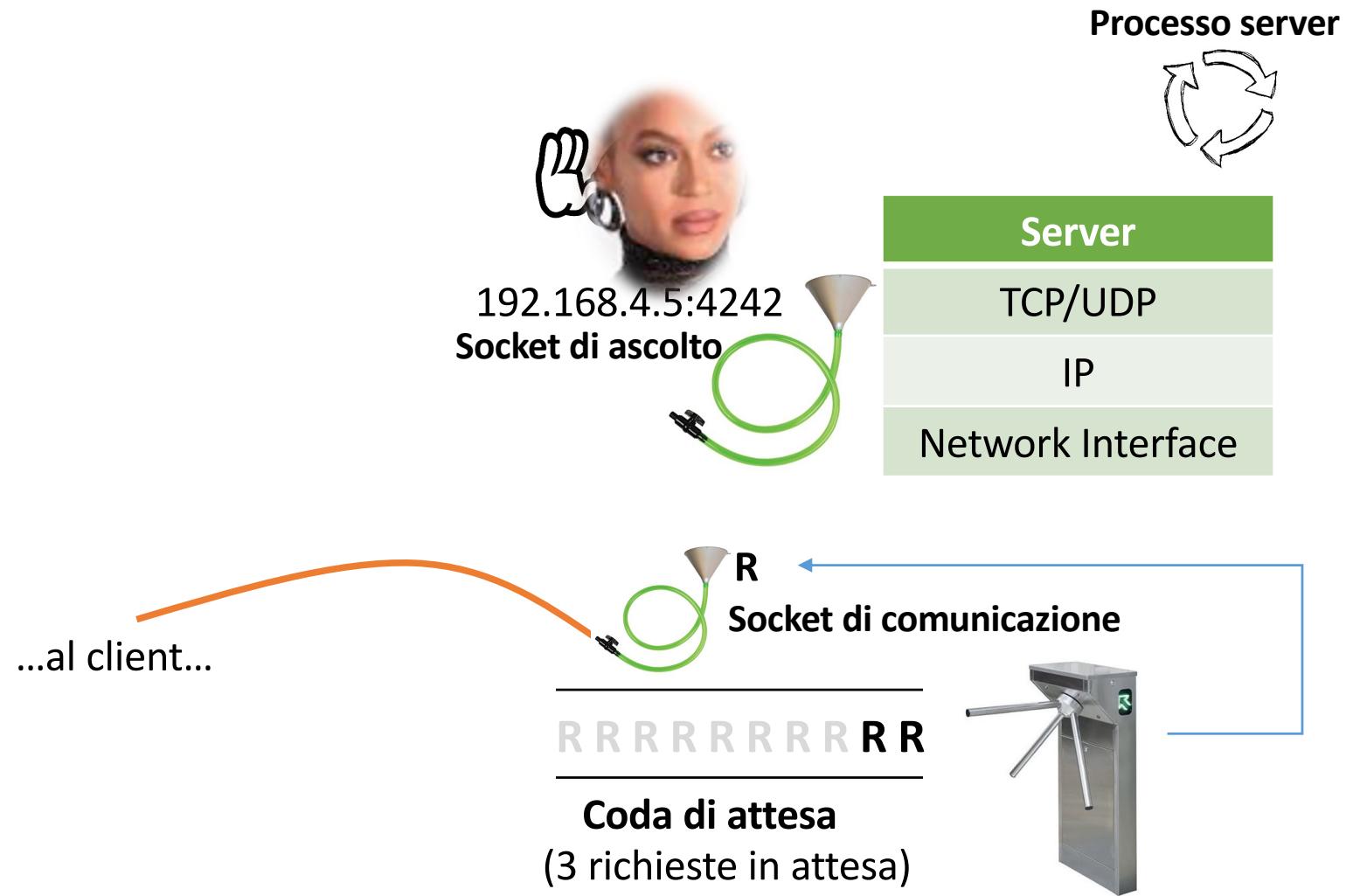
```
int accept(int sockfd, struct sockaddr *addr,  
          socklen_t *addrlen);
```

man 2 accept

- **sockfd**: descrittore del socket
- **addr**: puntatore a una struttura (vuota) di tipo **struct sockaddr** dove viene salvato l'indirizzo del client
- **addrlen**: dimensione di **addr**
- **Primitiva bloccante**: il programma si ferma in attesa di una richiesta di connessione
- All'arrivo di una richiesta di connessione, restituisce il **descrittore di un nuovo socket** (**detto socket di comunicazione**) usato per lo scambio di dati (-1 in caso di errore)

```
struct sockaddr_in cl_addr;  
int len = sizeof(cl_addr);  
new_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
```

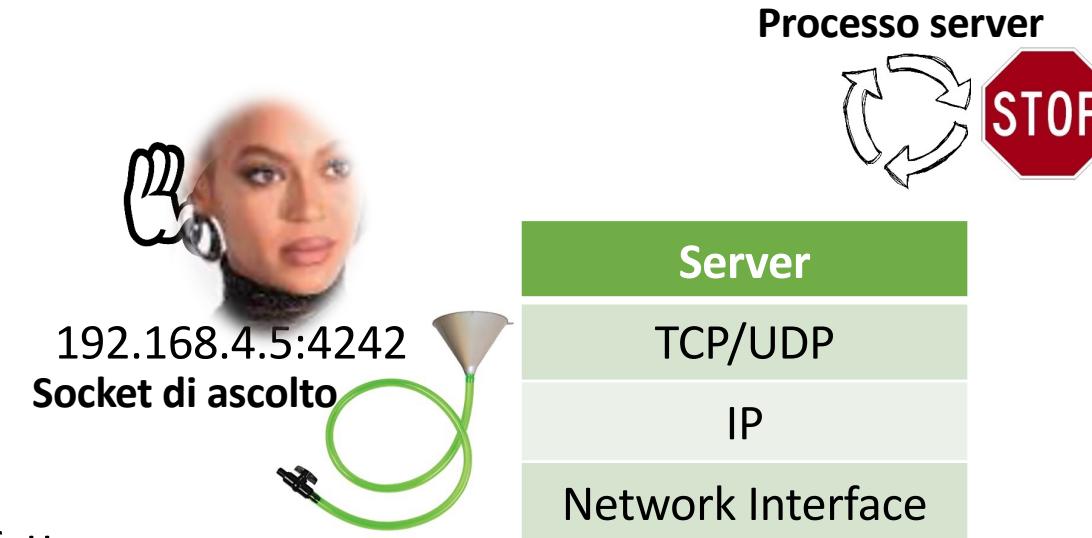
# Primitiva **accept()** [coda non vuota]



# Primitiva **accept()** [coda vuota]



Quando il processo server effettua la system call ad `accept()`, se la coda di attesa è vuota, **si blocca** in attesa che arrivi una richiesta.



---

R R R R R R R R R R

---

**Coda di attesa**  
(0 richieste in attesa)



# Processo server

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

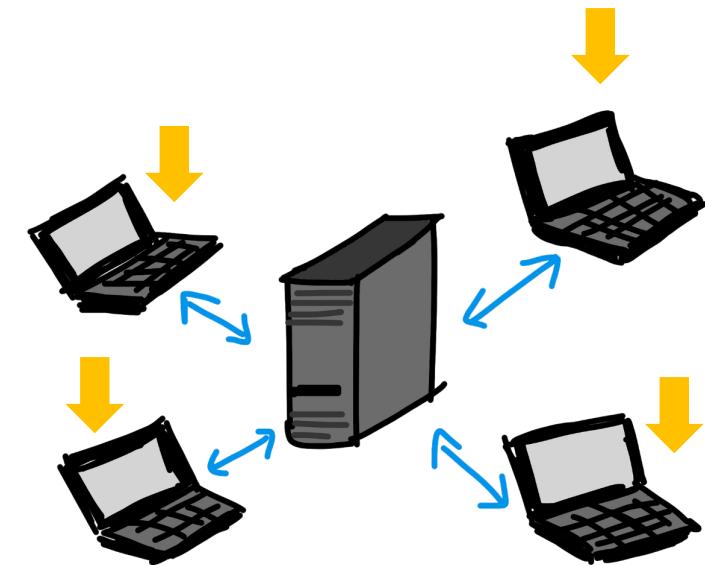
int main () {
    int ret, sd, new_sd, len;
    struct sockaddr_in my_addr, client_addr; // Strutture per gli IP
    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo socket*/
    memset(&my_addr, 0, sizeof(my_addr)); // Pulizia
    my_addr.sin_family = AF_INET ;
    my_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &my_addr.sin_addr);
    // In alternativa: my_addr.sin_addr.s_addr = INADDR_ANY;

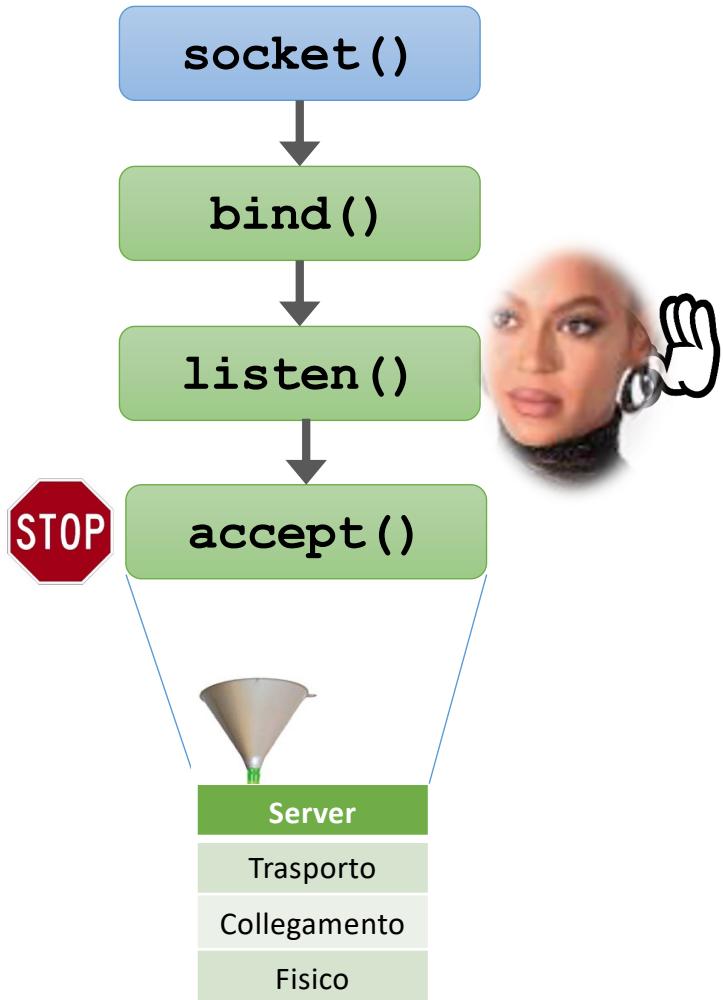
    ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
    ret = listen(sd, 10);
    len = sizeof(client_addr);
    new_sd = accept(sd, (struct sockaddr*)&client_addr, &len);
    //...
}
```

# Programmazione distribuita

Lato client



# Creazione del socket



Processo client



Il processo client crea il socket tramite la system call `socket()`

# Primitiva **connect()**

- Permette a un socket *locale* di inviare una **richiesta di connessione** a un socket *remoto*

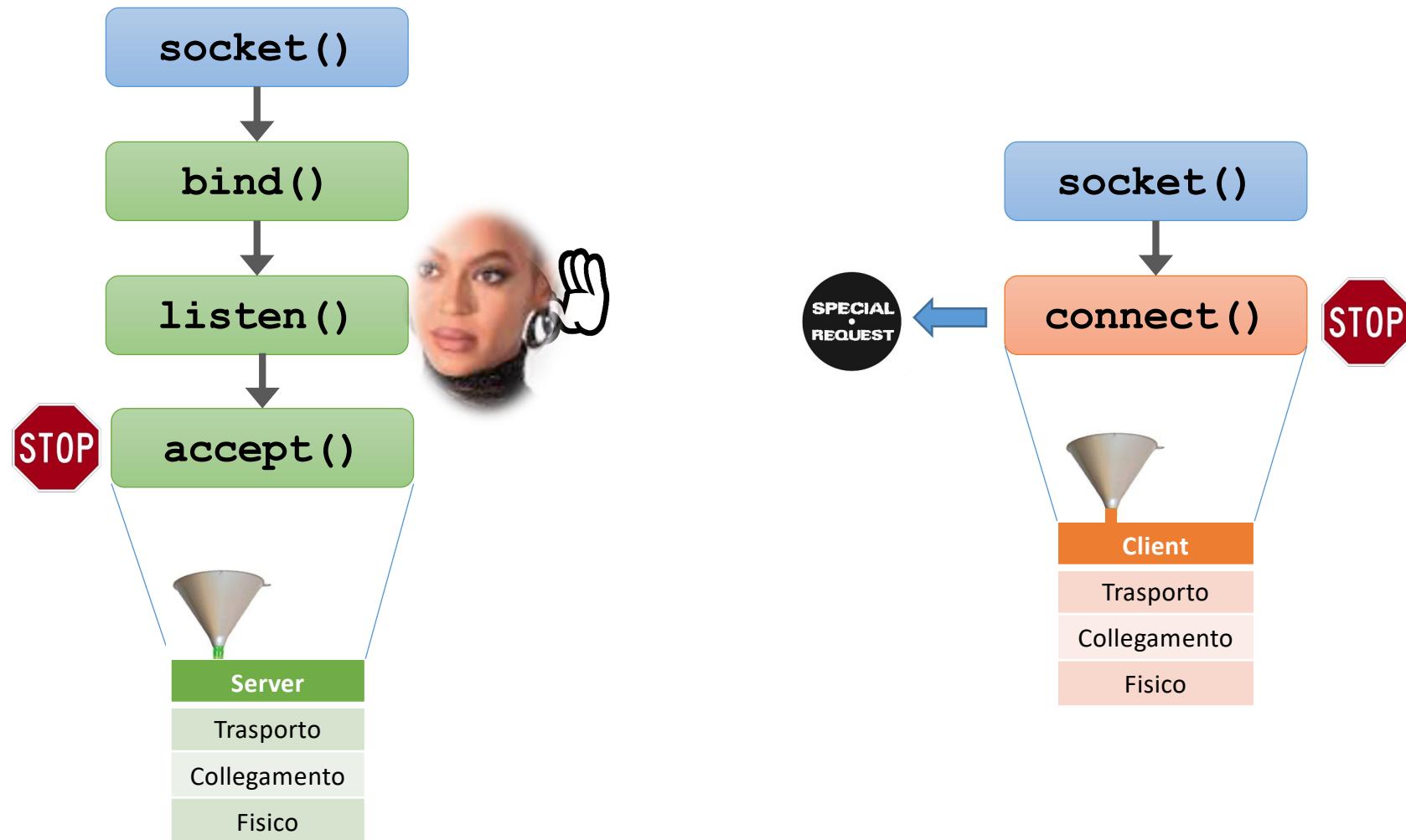
```
int connect(int sockfd, const struct sockaddr* addr,  
           socklen_t addrlen);
```

man 2 connect

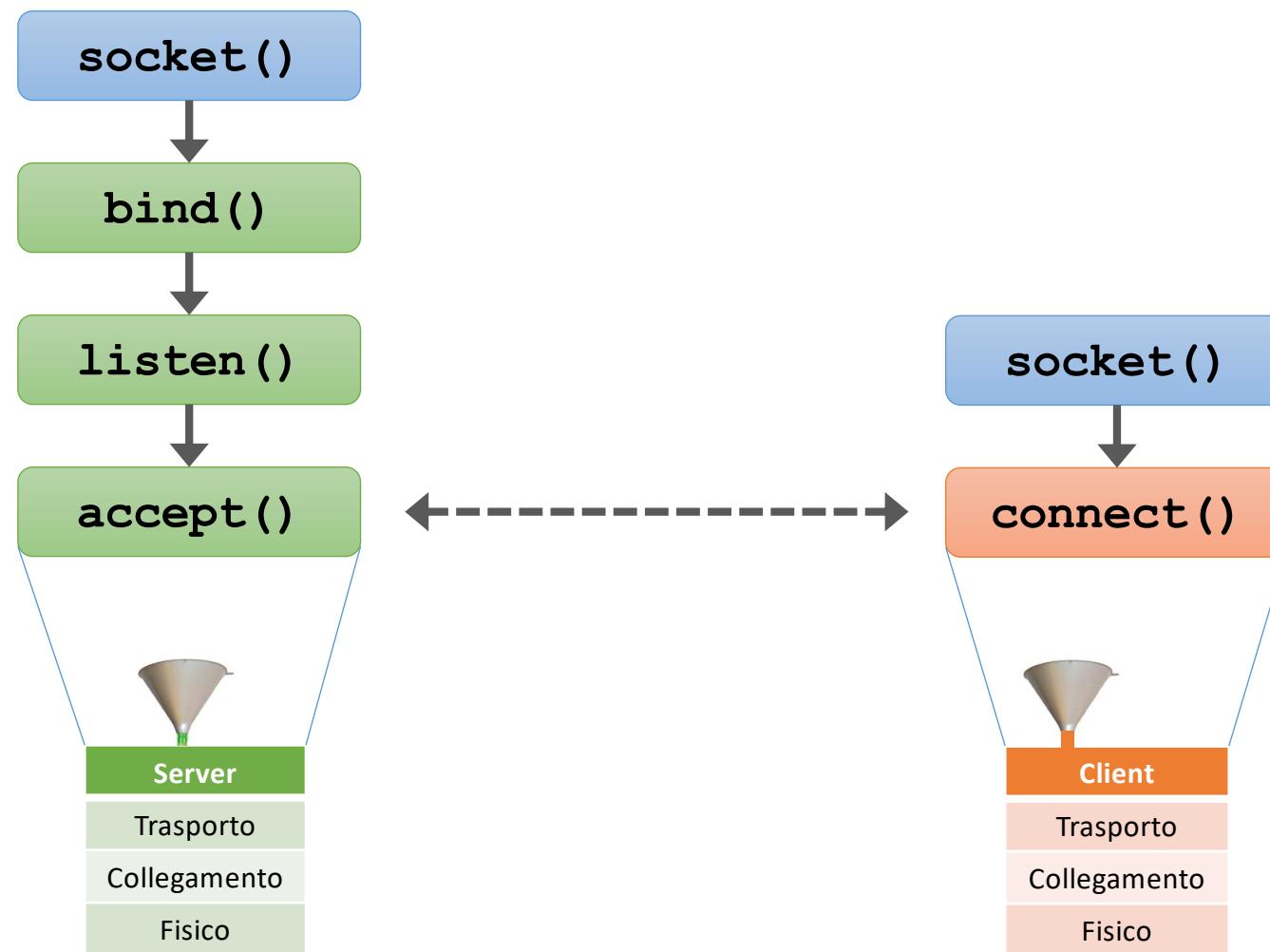
- **sockfd**: descrittore del socket locale
- **addr**: puntatore alla struttura contenente l'indirizzo del socket remoto
- **addrlen**: dimensione di **addr**
- La primitiva restituisce 0 se ha successo, -1 se si verifica un errore
- La primitiva è **bloccante**: il programma si ferma in attesa che la richiesta di connessione sia accettata

```
ret = connect(sd, (struct sockaddr*)&sv_addr, sizeof(sv_addr));
```

# Connessione



# Accettazione di connessione



# Processo client

```
#include <arpa/inet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main () {

    int ret, sd;
    struct sockaddr_in server_addr; // per il server

    /* Creazione socket */
    sd = socket(AF_INET, SOCK_STREAM, 0);

    /* Creazione indirizzo del server */
    memset(&server_addr, 0, sizeof(server_addr)); // Pulizia
    server_addr.sin_family = AF_INET ;
    server_addr.sin_port = htons(4242);
    inet_pton(AF_INET, "192.168.4.5", &server_addr.sin_addr);

    ret = connect(sd, (struct sockaddr*)&server_addr, sizeof(server_addr));
    // ...
}
```

# Programmazione distribuita

Scambio di dati

# Primitiva **send()**

- **Invia un messaggio** attraverso un socket connesso

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

man 2 send

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer contenente il messaggio da inviare
- **len**: dimensione del messaggio (in byte)
- **flags**: per settare le opzioni (per adesso mettiamolo a 0)
- La funzione restituisce il **numero di byte inviati**, -1 su errore
- **La funzione è bloccante**: il programma si ferma finché non ha scritto tutto il messaggio

# Primitiva **send()**

```
int ret, sd, len;
char buffer[1024];

//...

strcpy(buffer, "Hello Server!");
len = strlen(buffer);

// invio
ret = send(sd, (void*)buffer, len, 0);

if (ret < len) {
    // Gestione errore
}
```

# Primitiva **recv()**

- **Riceve un messaggio** da un socket connesso

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

man 2 recv

- **sockfd**: descrittore del socket
- **buf**: puntatore al buffer in cui salvare il messaggio
- **len**: dimensione in byte del messaggio desiderato
- **flags**: per settare le opzioni
- La funzione restituisce il **numero di byte ricevuti**, -1 su errore, 0 se il socket remoto si è chiuso (vedi più avanti)
- **La funzione è bloccante**: il programma si ferma finché non ha letto *qualcosa*

# Primitiva **recv()**

```
int ret, sd, bytes_needed;
char buffer[1024];

//...

bytes_needed = 20;

// ricezione
ret = recv(sd, (void*)buffer, bytes_needed, 0);
// Adesso 0 < ret <= bytes_needed
if (ret < bytes_needed) {
    // Gestione errore
}

ret = recv(sd, (void*)buffer, bytes_needed, MSG_WAITALL);
// Adesso ret == bytes_needed
```

# Primitiva `close()`

- **Chiude** un socket
  - Non può più essere usato per inviare o ricevere dati

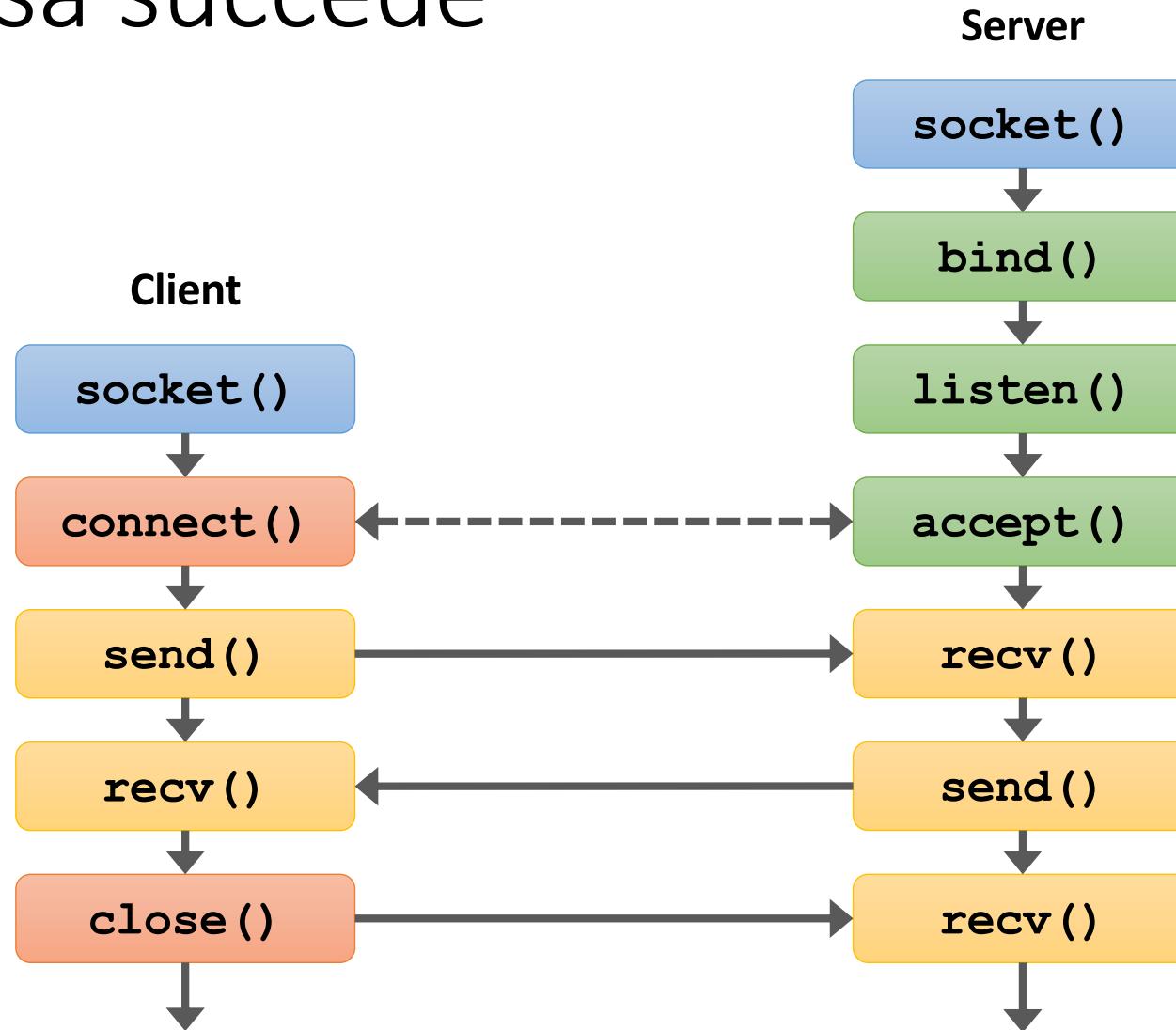
```
#include <unistd.h>

int close(int fd);
```

man 2 close

- **fd**: descrittore del socket
- La funzione restituisce 0 se ha successo, -1 su errore
- L'host remoto riceverà 0 dalla `recv()`

# Cosa succede



# Gestione degli errori

# Gestione degli errori

- Le primitive **restituiscono -1 in caso di errore**
  - In più settano una variabile, `errno`, che può essere letta per scoprire il motivo dell'errore
  - Nel manuale di ogni funzione c'è l'elenco degli errori possibili

```
#include <errno.h>
//...
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) {
    if (errno == EADDRINUSE) /* Gestisci errore */
    if (errno == EINVAL) /* Gestisci errore */
//...
}
```

man 3 errno

# Gestione degli errori

- A volte vogliamo solo sapere l'errore e uscire
  - `perror()` legge `errno` e stampa l'errore su schermo in forma leggibile

```
#include <stdio.h>
//...
ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
if (ret == -1) {
    perror("Error: ");
    exit(1);
}
//...
```

man 3 perror

# Esercizio 1

- Implementare un **server TCP mono-processo** che fornisce un messaggio "Hello!" ai client che si collegano
- Implementare il relativo **client**
  - Il client si connette, riceve il messaggio, lo stampa ed esce
- Note:
  - Gestire gli errori
  - Server e client conoscono la dimensione del messaggio

# Esercizio 2

- Implementare un **server TCP mono-processo** che re-invia al mittente un messaggio ricevuto
- Implementare un **client** che, di continuo:
  - Legge una stringa da tastiera
  - Se la stringa è "Bye" interrompe la connessione ed esce
  - Altrimenti invia la stringa, riceve la risposta e la stampa
- Il server continua a fare echo al client finché la connessione non viene interrotta
- Server e client leggono/scrivono sempre 20 byte
  - Attenzione al terminatore di stringa!