Laboratorio di
Reti Informatiche

Corso di Laurea Triennale in Ingegneria Informatica

A.A. 2016/2017

Ing. Niccolò Iardella

niccolo.iardella@unifi.it

Esercitazione 5

Programmazione con i socket – Parte 2

Programma di oggi

- Server concorrente
- Socket bloccanti e non bloccanti
- I/O multiplexing
- Socket UDP

Server concorrente

Tipi di server

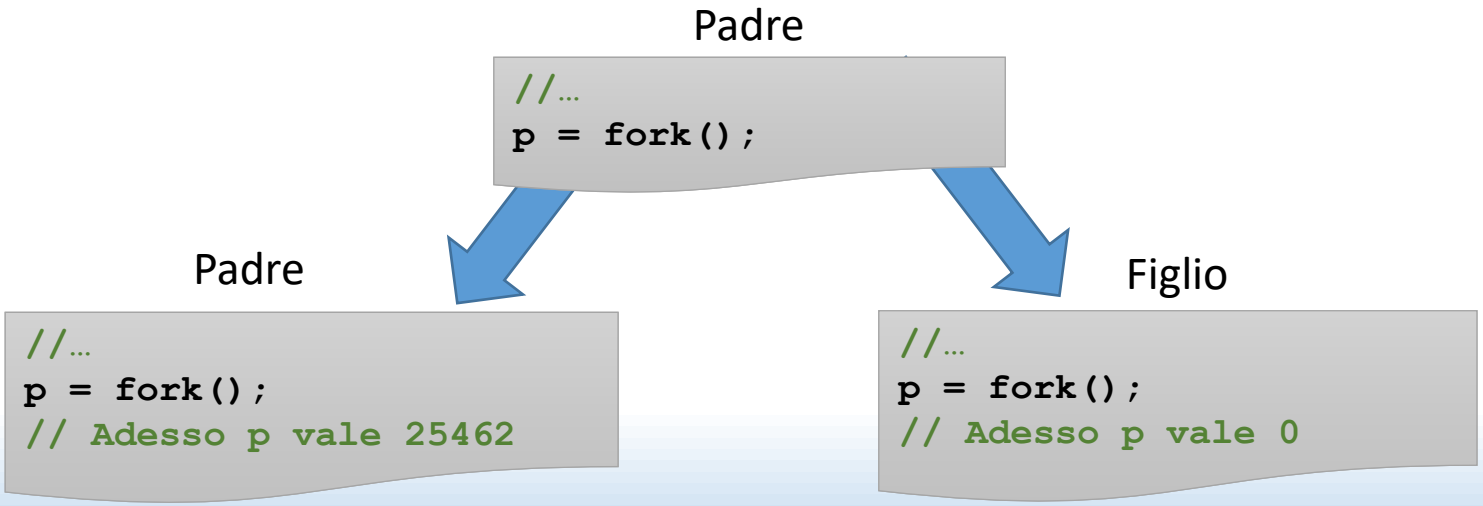
- Server *iterativo*
  - Viene servita una richiesta alla volta
- Server *concorrente*
  - Serve più richieste «contemporaneamente»
  - Per ogni richiesta accettata (`accept()`) il server crea un nuovo processo figlio

Creazione di un processo figlio


```
#include <unistd.h>
pid_t fork(void);
```

man 2 fork

- La primitiva **`fork()`** duplica il processo
  - Nel processo padre restituisce il PID del figlio
  - Nel processo figlio restituisce 0



Uso di **`fork()`**

```
pid_t pid;
//…
while (1) {
  new_sd = accept(sd, …);
  pid = fork();
  if (pid == 0) {
    // Qui sono nel processo figlio
    close(sd);
    // Servo la richiesta con new_sd
    //…
    close(new_sd);
    exit(0); // Il figlio termina
  }
  // Qui sono nel processo padre
  close(new_sd);
}
```

- Quando il processo viene duplicato, il padre e il figlio si ritrovano gli stessi descrittori, duplicati
- **Ognuno deve chiudere il descrittore che non usa**
  - Il padre chiude il descrittore del socket connesso al client
  - Il figlio chiude il descrittore del socket in ascolto

Server multi-processo


```
#include …

int main () {
  int ret, sd, new_sd, len;
  struct sockaddr_in my_addr, cl_addr;
  pid_t pid;

  sd = socket(AF_INET, SOCK_STREAM, 0);
  /* Creazione indirizzo */
  ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));
  ret = listen(sd, 10);
  int len = sizeof(cl_addr);
  while(1) {
    cl_sd = accept(sd, (struct sockaddr*)&cl_addr, &len);
    pid = fork();
    if (pid == -1) {
      /* Gestione errore */
    };
    if (pid == 0) {
      // Sono nel processo figlio
      close(sd);
      /* Gestione richiesta (send, recv, …) */
      close(cl_sd);
      exit(0);
    }
    // Sono nel processo padre
    close(cl_sd);
  }
```

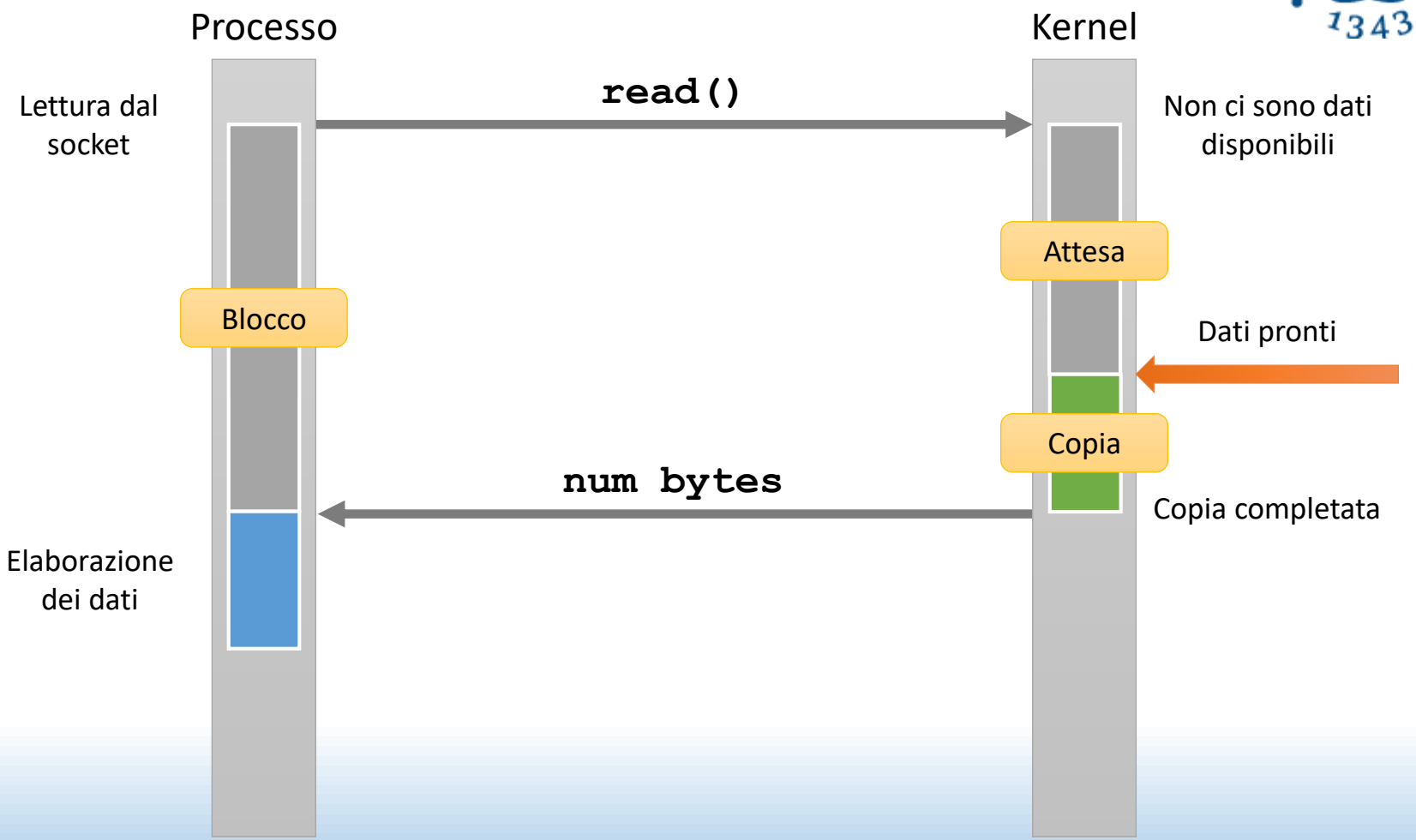
Modelli di I/O

Socket bloccanti e non bloccanti

Socket bloccante

- Di default, un socket è **bloccante**
  - Tutte le operazioni su di esso fermano l'esecuzione del processo in attesa del risultato
  - **`connect()`** si blocca finché il socket non è connesso
  - **`accept()`** si blocca finché non c'è una richiesta di connessione
  - **`send()`** si blocca finché tutto il messaggio non è stato inviato (il buffer di invio potrebbe essere pieno)
  - **`recv()`** si blocca finché non c'è qualche dato disponibile
    - O finché tutto il messaggio richiesto non è disponibile, con il flag **`MSG_WAITALL`**

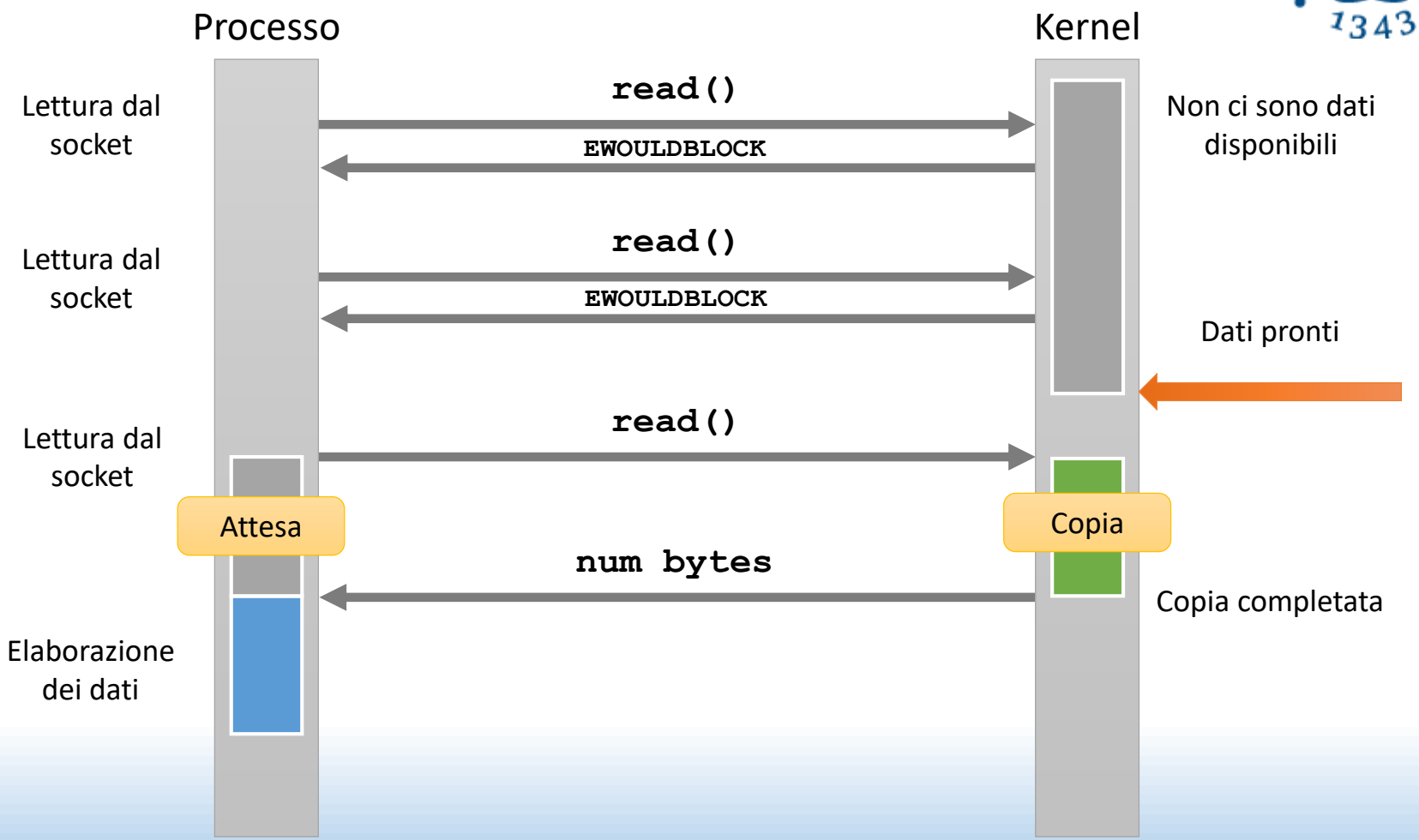
Socket bloccante



Socket non bloccante

- Un socket può essere settato come **non bloccante**
  - Le operazioni non attendono i risultati
  - **`connect()`** se non può connettersi subito restituisce -1 e setta **`errno`** a **`EINPROGRESS`**
  - **`accept()`** se non ci sono richieste restituisce -1 e setta **`errno`** a **`EWOULDBLOCK`**
  - **`send()`** se non riesce a inviare tutto il messaggio subito (il buffer è pieno), restituisce -1 e setta **`errno`** a **`EWOULDBLOCK`**
  - **`recv()`** se non ci sono messaggi restituisce -1 e setta **`errno`** a **`EWOULDBLOCK`**

Socket non bloccante

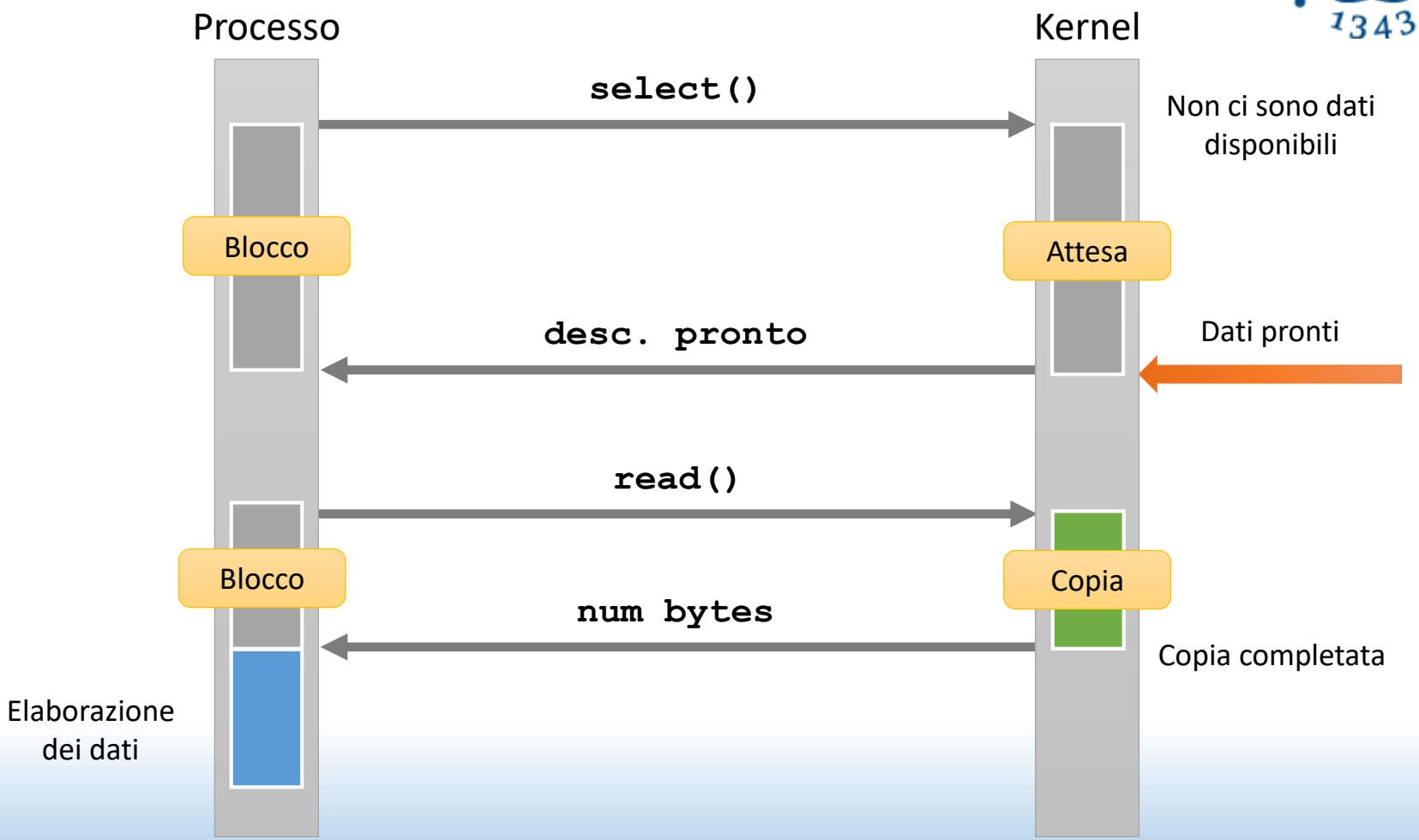


I/O multiplexing

Multiplexing I/O sincrono

- Problema:
  - Voglio controllare più descrittori/socket nello stesso momento
  - Se faccio operazioni su un socket bloccante, non posso controllarne altri
- Soluzione:
  - Multiplexing con la primitiva select()
  - Esamina più socket contemporaneamente, il primo che è pronto viene usato

Multiplexing I/O sincrono



Primitiva **`select()`**

- Controlla più socket contemporaneamente

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

`man 2 select`

- **`nfds`**: numero del descrittore più alto tra quelli da controllare, +1
- **`readfds`**: lista di descrittori da controllare per la lettura
- **`writefds`**: lista di descrittori da controllare per la scrittura
- **`exceptfds`**: lista di descrittori da controllare per le eccezioni (non ci interessa)
- **`timeout`**: intervallo di timeout
- La funzione restituisce il **numero di descrittori pronti**, -1 su errore
- **La funzione è bloccante**: si blocca finché un descrittore tra quelli controllati diventa pronti oppure finché il timeout non scade

Descrittori pronti

- **`select()`** rileva i socket pronti
- Un socket è pronto *in lettura* se:
  - C'è almeno un byte da leggere
  - Il socket è stato chiuso (**`read()`** restituirà **`0`**)
  - È un socket in ascolto e ci sono connessioni effettuate
  - C'è un errore (**`read()`** restituirà **`-1`**)
- Un socket è pronto *in scrittura* se:
  - C'è spazio nel buffer per scrivere
  - C'è un errore (**`write()`** restituirà **`-1`**)
    - Se il socket è chiuso, **`errno = EPIPE`**

Struttura per il timeout


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

struct timeval {
  long tv_sec;     /* seconds */
  long tv_usec;    /* microseconds */
};
```

- **`timeout = NULL`**
  - Attesa indefinita, fino a quando un descrittore è pronto
- **`timeout = { 10; 5; }`**
  - Attesa massima di 10 secondi e 5 microsecondi
- **`timeout = { 0; 0; }`**
  - Attesa nulla, controlla i descrittori ed esce immediatamente (*polling*)

Insieme di descrittori

- Un descrittore è un **`int`** che va da 0 a **`FD_SETSIZE`** (di solito 1024)
- Un insieme di descrittori si rappresenta con una variabile di tipo **`fdset`**
  - Non ci interessano i dettagli implementativi
  - Si manipola con delle *macro* simili a funzioni:

```
/* Rimuovere un descrittore dal set */
void FD_CLR(int fd, fd_set *set);
/* Controllare se un descrittore è nel set */
int  FD_ISSET(int fd, fd_set *set);
/* Aggiungere un descrittore al set */
void FD_SET(int fd, fd_set *set);
/* Svuotare il set */
void FD_ZERO(fd_set *set);
```

Insieme di descrittori

- **`select()`** modifica i set di descrittori:
  - Prima di chiamare **`select()`** inserisco nei set di lettura e di scrittura i descrittori che voglio monitorare
  - Dopo **`select()`** trovo nei set di lettura e scrittura i descrittori pronti

Utilizzo di **select()**


```
int main(int argc, char *argv[]){
  fd_set master;                    // Set principale
  fd_set read_fds;                  // Set di lettura
  int fdmax;                        // Numero max di descrittori

  struct sockaddr_in sv_addr;       // Indirizzo server
  struct sockaddr_in cl_addr;       // Indirizzo client
  int listener;                     // Socket per l'ascolto
  int newfd;
  char buf[1024];                   // Buffer
  int nbytes;
  int addrlen;
  int i;
  /* Azzero i set */
  FD_ZERO(&master);
  FD_ZERO(&read_fds);

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

Utilizzo di **select()**


```
sv_addr.sin_family = AF_INET;
sv_addr.sin_addr.s_addr = INADDR_ANY;
sv_addr.sin_port = htons(20000);
bind(listener, (struct sockaddr*)& sv_addr, sizeof(sv_addr));
listen(sd, 10);
FD_SET(listener, &master); // Aggiungo il listener al set
fdmax = listener;          // Tengo traccia del maggiore
for(;;) {
  read_fds = master;       // Copia
  select(fdmax + 1, &read_fds, NULL, NULL, NULL);
  for(i = 0; i <= fdmax; i++) {  // Scorro tutto il set
    if(FD_ISSET(i, &read_fds)) { // Trovato un desc. pronto
      if(i == listener) {        // È il listener
        addrlen = sizeof(cl_addr);
        newfd = accept(listener,
                       (struct sockaddr *)&cl_addr, &addrlen)
        FD_SET(newfd, &master); // Aggiungo il nuovo socket
        if(newfd > fdmax){ fdmax = newfd; } // Aggiorno max
```

Utilizzo di **`select()`**

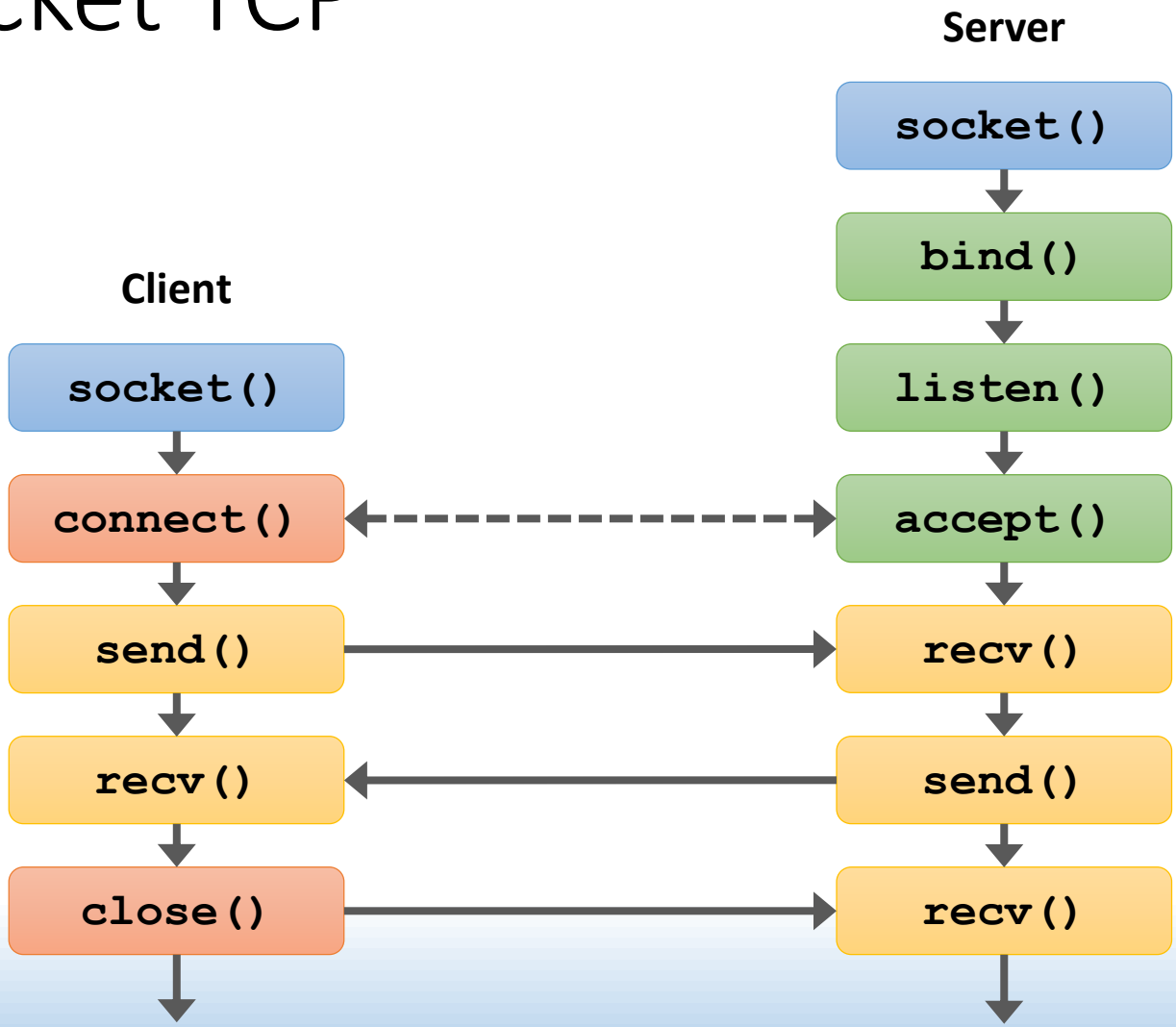

```
        } else {  // È un altro socket
          nbytes = recv(i, buf, sizeof(buf);
          //… Uso dati
          close(i);  // Chiudo socket
          FD_CLR(i, &master); // Rimuovo il socket dal set
        }
      }
    }
  }
  return 0;
}
```

Socket UDP

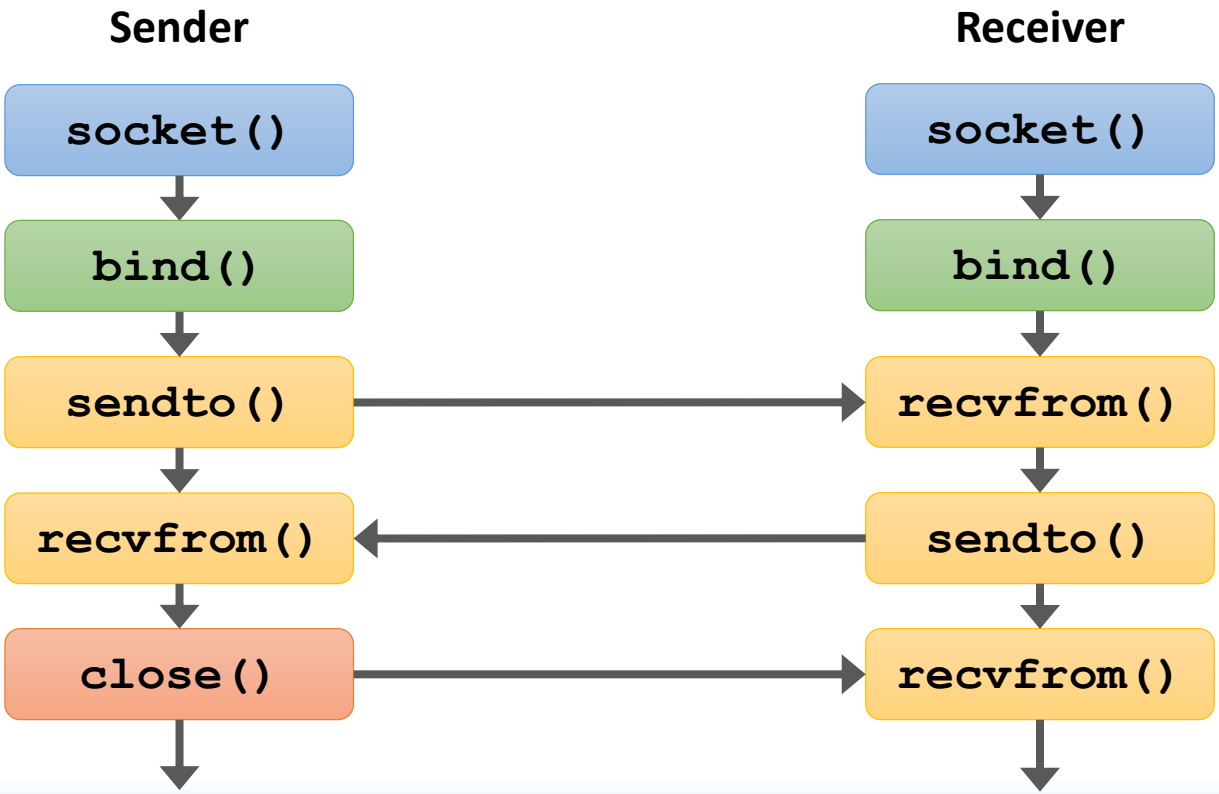
TCP vs UDP

- TCP instaura una **connessione**
  - Cioè prevede operazioni preliminari per instaurare un canale virtuale
  - Affidabile: i pacchetti inviati arrivano tutti, nell'ordine in cui sono stati inviati
  - Comporta latenza maggiore per il riordino e eventuali ritrasmissioni
- UDP è *connection-less*
  - Nessuna operazione preliminare
  - Rapido: nessun recupero e nessun riordino

Socket TCP



Socket UDP



Primitiva **`sendto()`**

- Invia un messaggio attraverso un socket all'indirizzo specificato

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

- **`sockfd`**: descrittore del socket
- **`buf`**: puntatore al buffer contenente il messaggio da inviare
- **`len`**: dimensione in byte del messaggio
- **`flags`**: per settare delle opzioni, lasciamolo a 0
- **`dest_addr`**: puntatore alla struttura in cui ho salvato l'indirizzo del destinatario
- **`addrlen`**: lunghezza di **`dest_addr`**
- 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 **`recvfrom()`**

- Riceve un messaggio attraverso un socket

```
ssize_t recvfrom(int sockfd, const void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t addrlen);
```

- **`sockfd`**: descrittore del socket
- **`buf`**: puntatore al buffer contenente il messaggio da inviare
- **`len`**: dimensione in byte del messaggio
- **`flags`**: per settare delle opzioni
- **`dest_addr`**: puntatore a una struttura vuota in cui salvare l'indirizzo del mittente
- **`addrlen`**: lunghezza di **`dest_addr`**
- 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*

Codice del server


```
int main () {
  int ret, sd, len;
  char buf[BUFLEN];
  struct sockaddr_in my_addr, cl_addr;
  int addrlen = sizeof(cl_addr);
  /* Creazione socket */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  /* Creazione indirizzo */
  memset(&my_addr, 0, sizeof(my_addr); // Pulizia
  my_addr.sin_family = AF_INET ;
  my_addr.sin_port = htons(4242);
  my_addr.sin_addr.s_addr = INADDR_ANY;
  ret = bind(sd, (struct sockaddr*)&my_addr, sizeof(my_addr));

  while(1) {
    len = recvfrom(sd, buf, BUFLEN, 0,
                   (struct sockaddr*)&cl_addr, &addrlen);
    //…
  }
```

Codice del client


```
int main () {
  int ret, sd, len;
  char buf[BUFLEN];
  struct sockaddr_in sv_addr; // Struttura per il server
  /* Creazione socket */
  sd = socket(AF_INET, SOCK_DGRAM, 0);
  /* Creazione indirizzo del server */
  memset(&sv_addr, 0, sizeof(sv_addr); // Pulizia
  sv_addr.sin_family = AF_INET ;
  sv_addr.sin_port = htons(4242);
  inet_pton(AF_INET, "192.168.4.5", &sv_addr.sin_addr);

 while(1) {
    len = sendto(sd, buf, BUFLEN, 0,
                 (struct sockaddr*)&sv_addr, sizeof(sv_addr));
    //…
  }
```

Socket UDP «connesso»

- Usando **`connect()`** su un socket UDP gli si può associare un indirizzo remoto
- Il socket riceverà/invierà pacchetti solo da/a quel indirizzo
  - **Non** è una connessione!
- Con un socket connesso si possono usare **`send()`** e **`recv()`**, evitando di specificare ogni volta l'indirizzo