Reti Informatiche

Socket (Seconda Parte)

Acknowledgment: Prof Vincenzo Auletta, Università di Salerno

Sommario

- Tipologie di server
- Modelli di I/O:
  - I/O bloccante
  - I/O non bloccante
  - I/O Multiplexing (primitiva select)
- Esempio server con select
- Socket UDP

Tipologie di server

- Server iterativo
  - viene servita una richiesta alla volta
- Server concorrente:
  - Per ogni richiesta da parte di un client (accettata dalla *accept*) il server
    - Crea un processo figlio (primitiva fork())
    - Crea un thread
    - Attiva un thread da un pool creato in anticipo
  - Il processo/thread si incarica di gestire il client in questione

Server concorrente multiprocesso

```
For(;;) {
        consd = accept(listensd, …);          /* probably blocks */
        if((pid = fork()) = = 0) {
                close(listensd);              /* child closes listening socket */
                doit(consd);                  /* process the request */
                close(consd);                 /* done with this client */
                Exit(0);                      /* child terminates */
        }
        close(consd);                         /* parent closes conn. socket */
}
```

- La close(consd) è obbligatoria
  - decrementa il numero di riferimenti al socket descriptor.
  - La sequenza di chiusura non viene innescata fintanto che il numero di riferimenti non si annulla

Server concorrente multiprocesso

```
#include <sys/types.h>
#include <unistd.h>
…
int sock, cl_sock, ret;
struct sockaddr_in srv_addr, cl_addr;
pid_t child_pid;
sock = socket(AF_INET, SOCK_STREAM,0);
if(sock==-1){ /*errore*/}
/*inizializzazione srv_addr*/
bind(sock, &srv_addr, sizeof(srv_addr));
listen(sock,QUEUE_SIZE);
while(1){
    cl_sock = accept(sock, &cl_addr, sizeof(cl_addr));
    if(cl_sock==-1){ /*errore*/}
    /* gestione cl_addr */
    child_pid = fork();
    if(child_pid==0) /* sono nel processo figlio*/
        gestisci_richiesta(cl_sock, sock, …); /*funzione per la gestione delle
    richieste per il det servizio */
    else /* sono nel processo padre*/
        close(cl_sock);
}
```

Server concorrente multi-threaded

```
For(;;) {
        consd = accept(listensd, …);            /* probably blocks */
        if ( pthread_create( &tid, NULL, doit, (void*)consd ) ) {
                        exit(0);            /* error on thread creation */
        }
}
```

- Chiusura della connessione
  - in questo caso la close(consd) viene fatta dal thread gestore

Server concorrente multi-threaded

```
#include <sys/types.h>
#include <unistd.h>
...
void* gestisci_richiesta( void* socket ) { /*funzione per la gestione delle richieste */ }
...
int sock, cl_sock, ret;
struct sockaddr_in srv_addr, cl_addr;
pthread_t tid;
sock = socket(AF_INET, SOCK_STREAM,0);
if(sock==-1){ /*errore*/}
/*inizializzazione srv_addr*/
bind(sock, &srv_addr, sizeof(srv_addr));
listen(sock,QUEUE_SIZE);
while(1){
      cl_sock = accept(sock, &cl_addr, sizeof(cl_addr));
      if(cl_sock==-1){ /*errore*/}
      /* gestione cl_addr */
      if ( pthread_create( &tid, NULL, gestisci_richiesta, (void*)cl_sock ) ) {
            exit(0);                  /* errore */
      }
}
```

Modelli di I/O

Modelli di I/O

- In Unix sono disponibili diverse modalità di I/O:
  - I/O bloccante
  - I/O non bloccante
  - I/O multiplexing

Primitive per invio dati su socket

- `ssize_t send(int sd, const void* buf, size_t len, int flags);`
- `ssize_t sendto(int sd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen);`
- `ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);`
- `ssize_t write(int fd, const void *buf, size_t count);`

Send bloccante(2)

- La send si blocca solo quando il buffer in trasmissione associato al socket è pieno.
- Se il socket è impostato come non bloccante ovviamente non ci sarà nessun blocco ma ritornerà un -1 settando la variabile di errore EAGAIN o EWOULDBLOCK a indicare che tale istruzione si sarebbe dovuta bloccare

Primitive per ricezione dati su socket

- `ssize_t recv(int sd, void* buf, size_t len, int flags);`
- `ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);`
- `ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);`
- `ssize_t read(int fd, void *buf, size_t count);`

Recv bloccante(2)

- La recv si blocca solo quando il buffer in ricezione associato al socket è vuoto.
- Se il socket è impostato come non bloccante ovviamente non ci sarà nessun blocco ma ritornerà un -1 settando la variabile di errore EAGAIN o EWOULDBLOCK a indicare che tale istruzione si sarebbe dovuta bloccare

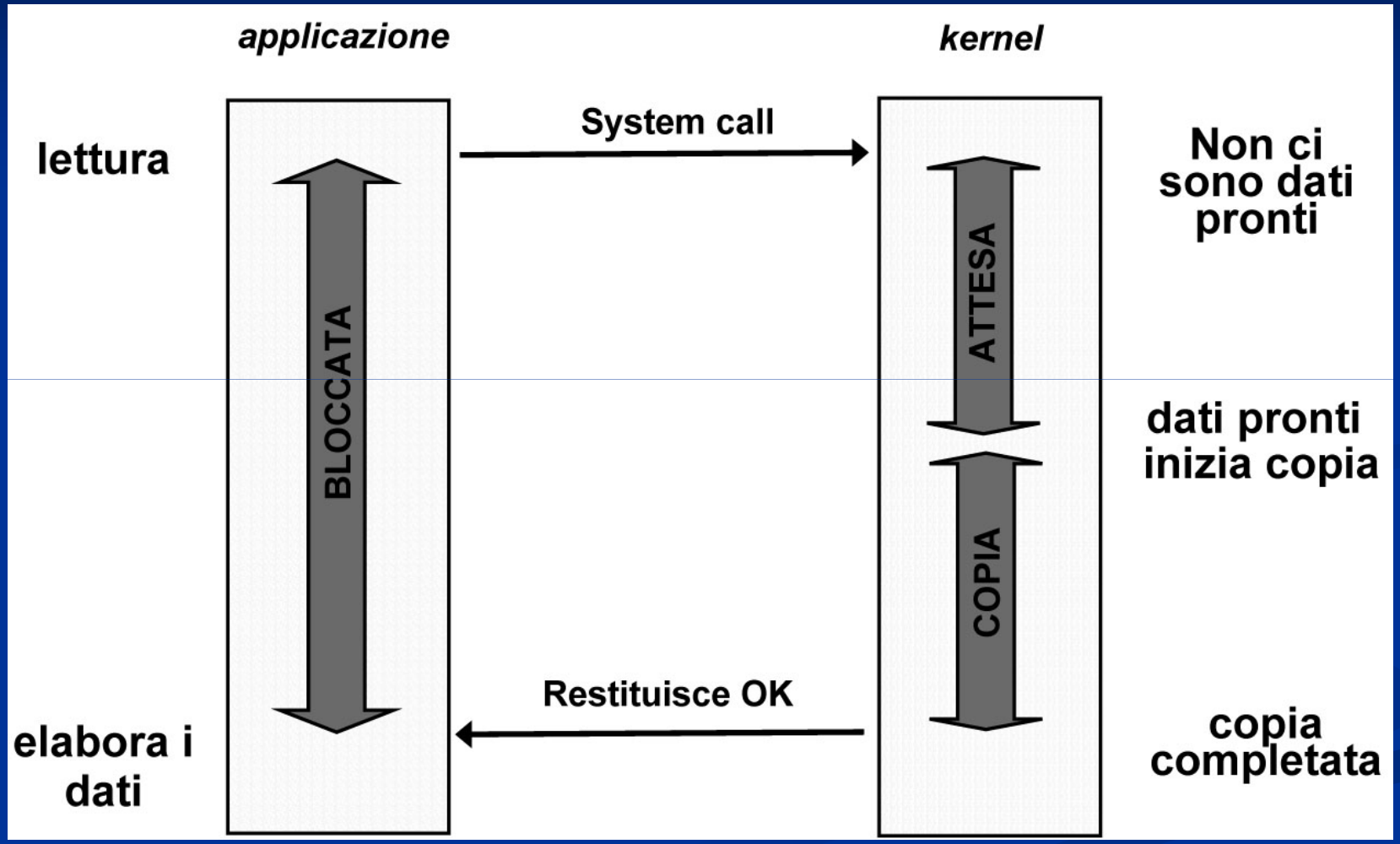
Buffer di ricezione e invio

- Ogni socket ha un buffer di ricezione ed uno di trasmissione
- Il buffer di ricezione è usato per mantenere i dati nel kernel prima di passarli all'applicazione
- Con TCP, lo spazio disponibile è quello pubblicizzato nella finestra di TCP
- Con UDP, eventuali pacchetti in overflow vengono cancellati

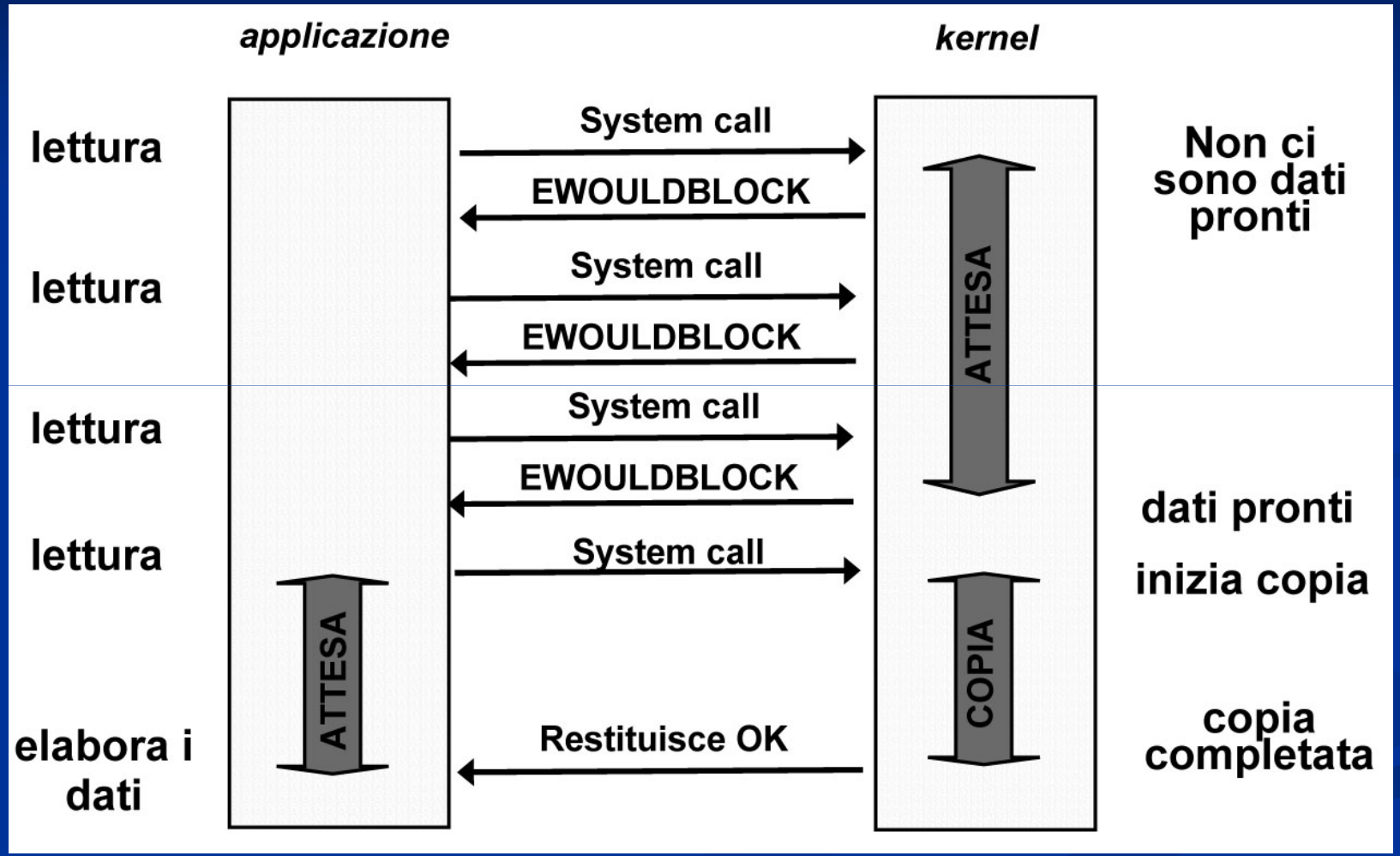
Buffer di ricezione e invio (2)

- Il buffer in spedizione è usato dall'applicazione per passare i dati al kernel per spedirli
- SO_RCVBUF e SO_SNDBUF permettono di cambiare la grandezza dei buffer

I/O Bloccante



I/O Non-Bloccante

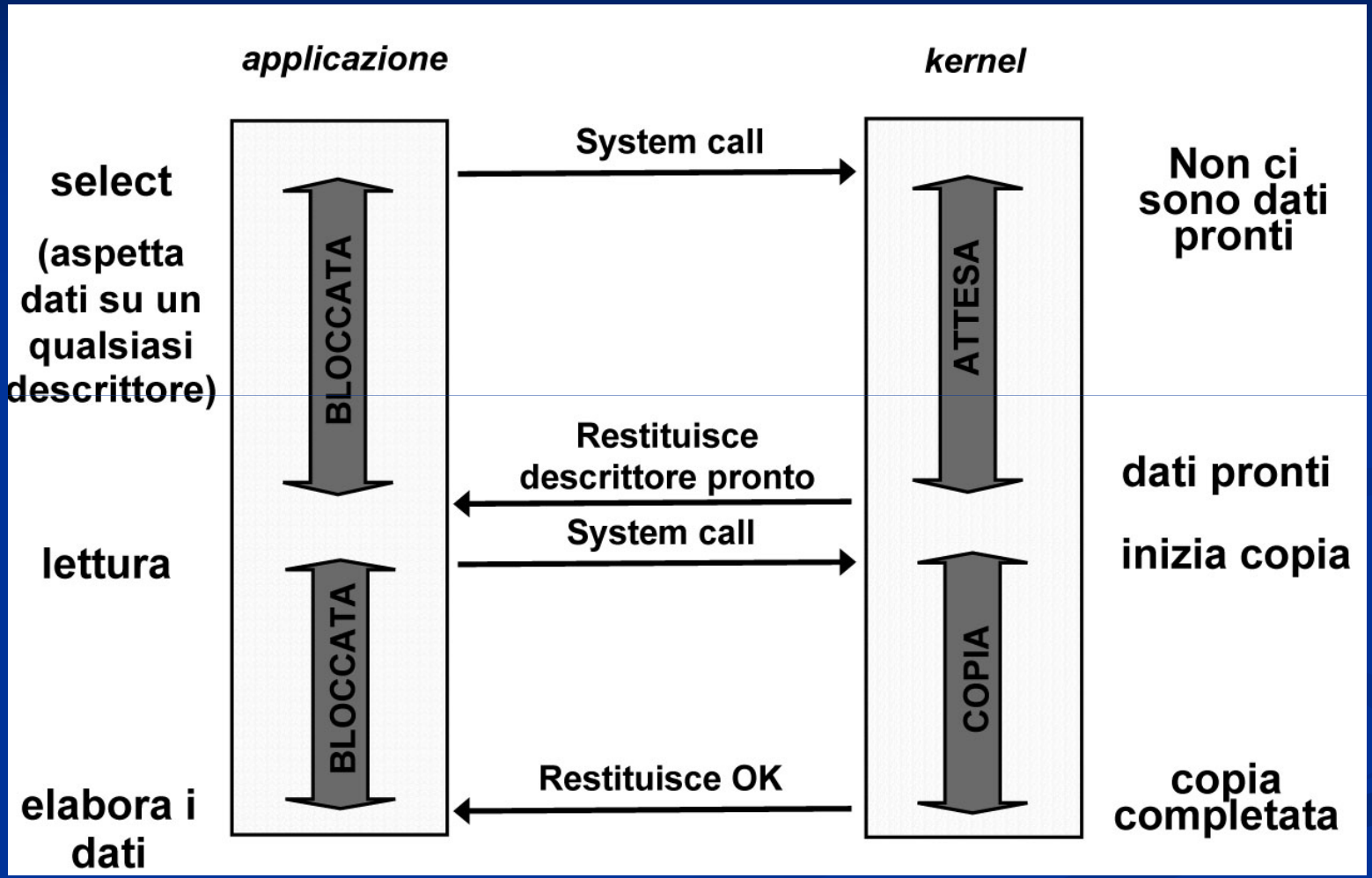


I/O Multiplexing

Problema

- In genere una funzione di input si blocca se non ci sono dati da leggere
  - può rimanere bloccata per molto tempo
  - l'altro descrittore non può essere controllato
- Serve un meccanismo per poter esaminare più canali di input contemporaneamente
  - Il primo canale che produce dati viene letto

I/O Multiplexing



Select

```
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfd, fd_set* readset,
fd_set* writeset, fd_set* exceptionset,
struct timeval *timeout);
```

- Restituisce
  - -1 se errore
  - 0 se scaduto il timeout
  - numero di descrittori pronti
- Permette di controllare contemporaneamente uno o più descrittori per lettura, scrittura o gestione errori

Timeout select

- timeout è il tempo massimo che la system call attende per individuare un descrittore pronto

```
struct timeval {
long tv_sec; /* numero di secondi */
long tv_usec; /* numero di microsecondi */
};
```

- timeout = 0
  - aspetta fino a quando un descrittore è pronto
- timeout = { 3; 5; }
  - aspetta fino al timeout e poi esce anche se non ci sono descrittori pronti
  - alcuni S.O. arrotondano a multipli di 10 millisecondi
- timeout = { 0; 0; }
  - controlla i descrittori ed esce immediatamente (polling)

Insiemi di Descrittori

- Insiemi di descrittori da controllare
  - readset: pronti per la lettura
  - writeset: pronti per la scrittura
  - exceptionset: condizioni di eccezione
    - Arrivo di dati fuori banda su un socket
    - Informazioni di controllo da uno pseudo terminale
- readset, writeset e exceptionset sono variabili di tipo fd_set
  - in genere è un array di interi in cui ogni bit rappresenta un descrittore
  - primo elemento dell' array rappresenta descrittori da 0 a 31
  - secondo elemento dell' array rappresenta descrittori da 32 a 63
- dettagli implementativi nascosti nella definizione

Operazioni su insiemi di descrittori

```
void FD_ZERO(fd_set *fdset)Azzera la struttura fdset
void FD_SET(int fd, fd_set *fdset) Mette a 1 il bit relativo a fd
void FD_CLR(int fd, fd_set *fdset) Mette a 0 il bit relativo a fd
int FD_ISSET(int fd, fd_set *fdset) Controlla se il bit relativo a fd è a 1
```

- Macro utilizzate per operare sugli insiemi di descrittori
- La costante FD_SETSIZE è il numero di descrittori in fd_set
  - definita in <sys/select.h> (solitamente 1024)
- in genere si usano meno descrittori
  - [0, maxd] è l'intervallo di descrittori effettivamente utilizzati
- Es. se siamo interessati ai descrittori 1,4,7,9 maxd = 10
  - i descrittori iniziano da 0

Descrittori pronti

- La select rileva i descrittori pronti
  - significato diverso per ciascuno dei tre gruppi
- Un socket è pronto in lettura se
  - ci sono almeno LWM (low-water mark) bytes da leggere
    - LWM selezionabile tramite opzioni del socket
      - per default è 1
  - il socket è stato chiuso in lettura (è stato ricevuto il FIN)
    - l'operazione di lettura ritorna EOF
  - Il socket è un socket di ascolto e ci sono delle connessioni completate
  - c'è un errore pendente sul socket
    - L'operazione di lettura ritornerà -1

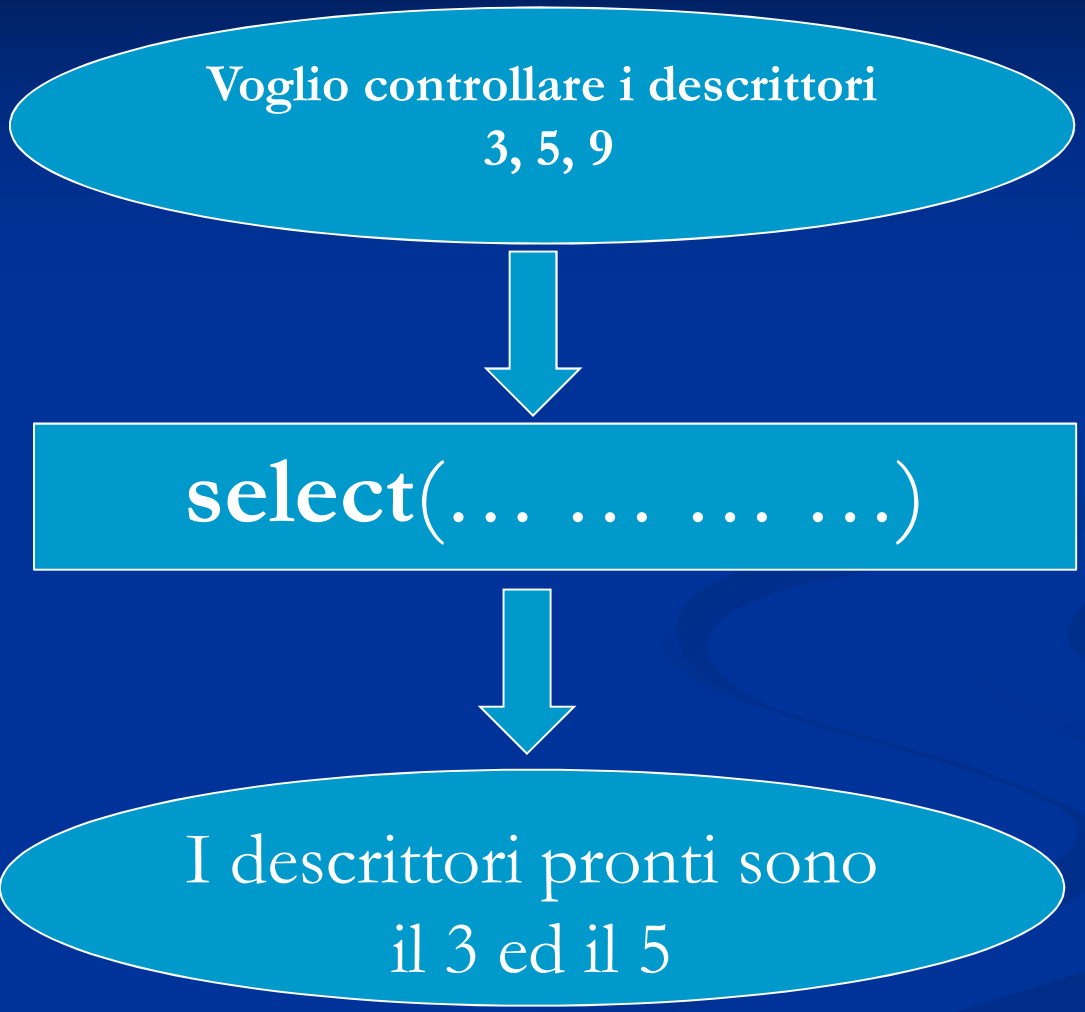
Descrittori pronti (2)

- Un socket è pronto in scrittura se
  - Il numero di byte liberi nel buffer di spedizione del socket è maggiore di LWM
    - LWM selezionabile tramite opzioni del socket
    - per default è 2048
    - L'operazione di scrittura restituisce il numero di byte effettivamente passati al livello di trasporto
  - Il socket è stato chiuso in scrittura
    - Un'operazione di scrittura genera SIGPIPE
  - C'è un errore
    - L'operazione di scrittura ritornerà -1 e errno specificherà l'errore

select (Input-Output)

- La select() modifica gli insiemi di descrittori puntati da *readset*, *writeset* e *exceptionset*.
- Quando chiamiamo la select() indichiamo, attraverso questi insiemi, i descrittori cui siamo interessati.
- Quando la funzione ritorna, negli stessi insiemi, troviamo i descrittori che sono pronti.
- I descrittori pronti hanno il relativo bit ad 1 nei set di uscita (usiamo FD_ISSET() per controllare se il bit di un descrittore è settato).

select (Input-Output)



Esempio select

```
#define PORT 2020
int main(int argc, char *argv[]){
/* master file descriptor list */
fd_set master;
/* temp file descriptor list for select() */
fd_set read_fds;
/* server address */
struct sockaddr_in serveraddr;
/* client address */
struct sockaddr_in clientaddr;
/* maximum file descriptor number */
int fdmax;
/* listening socket descriptor */
int listener;
/* newly accept()ed socket descriptor */
int newfd;
/* buffer for client data */
char buf[1024];
int nbytes;
/* for setsockopt() SO_REUSEADDR, below */
int yes = 1;
int addrlen;
int i, j;
/* clear the master and temp sets */
FD_ZERO(&master);
FD_ZERO(&read_fds);
/* get the listener */
if((listener = socket(AF_INET, SOCK_STREAM, 0)) == -1){
perror("Server-socket() error!");
exit(1);}
```

Creazione fd_set

Inizializzazione dei set

Esempio select (2)

```
/*"address already in use" error message */
if(setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int)) == -1){
perror("Server-setsockopt() error!");
exit(1);}
/* bind */
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = INADDR_ANY;
serveraddr.sin_port = htons(PORT);
if(bind(listener, (struct sockaddr *)&serveraddr, sizeof(serveraddr)) == -1){
    perror("Server-bind() error!");
    exit(1);}
/* listen */
if(listen(listener, 10) == -1)
{perror("Server-listen() error!");
     exit(1);}
/* add the listener to the master set */
FD_SET(listener, &master);
/* keep track of the biggest file descriptor */
fdmax = listener; /* so far, it's this one*/
for(;;){
/* copy it */
read_fds = master;
if(select(fdmax+1, &read_fds, NULL, NULL, NULL) == -1)
{    perror("Server-select() error!");
    exit(1);}
printf("Server-select() is OK...\n");
```

Voglio controllare il socket listener, quindi setto il relativo bit.

Faccio una copia del set dei descrittori da controllare perché la select modifica i set in input.

Esempio select (3)

```
/*run through the existing connections looking for data to be read*/
for(i = 0; i <= fdmax; i++)
{   if(FD_ISSET(i, &read_fds))
   { /* we got one... */
    if(i == listener){
      /* handle new connections */
      addrlen = sizeof(clientaddr);
       if((newfd = accept(listener, (struct sockaddr *)&clientaddr,
&addrlen)) == -1){
          perror("Server-accept() error!");
      }else{
        printf("Server-accept() is OK...\n");
         FD_SET(newfd, &master); /* add to master set */
        if(newfd > fdmax){
         /* keep track of the maximum */
         fdmax = newfd;}
         printf("%s: New connection from %s on socket %d\n", argv[0],
inet_ntoa(clientaddr.sin_addr), newfd);}
        }else{
            /* handle data from a client */
          if((nbytes = recv(i, buf, sizeof(buf), 0)) <= 0){
               /* got error or connection closed by client */
              if(nbytes == 0)
                   /* connection closed */
                   printf("%s: socket %d hung up\n", argv[0], i);
              else
                   perror("recv() error!");
              /* close it... */
              close(i);
```

Se il descittore pronto è quello di listener allora significa che ho una richiesta di connessione pendente e posso fare accept.

Aggiungo il nuovo descrittore tra quelli da controllare da ora in avanti.

Altrimenti gestisco la richiesta di un client.

Esempio select (4)

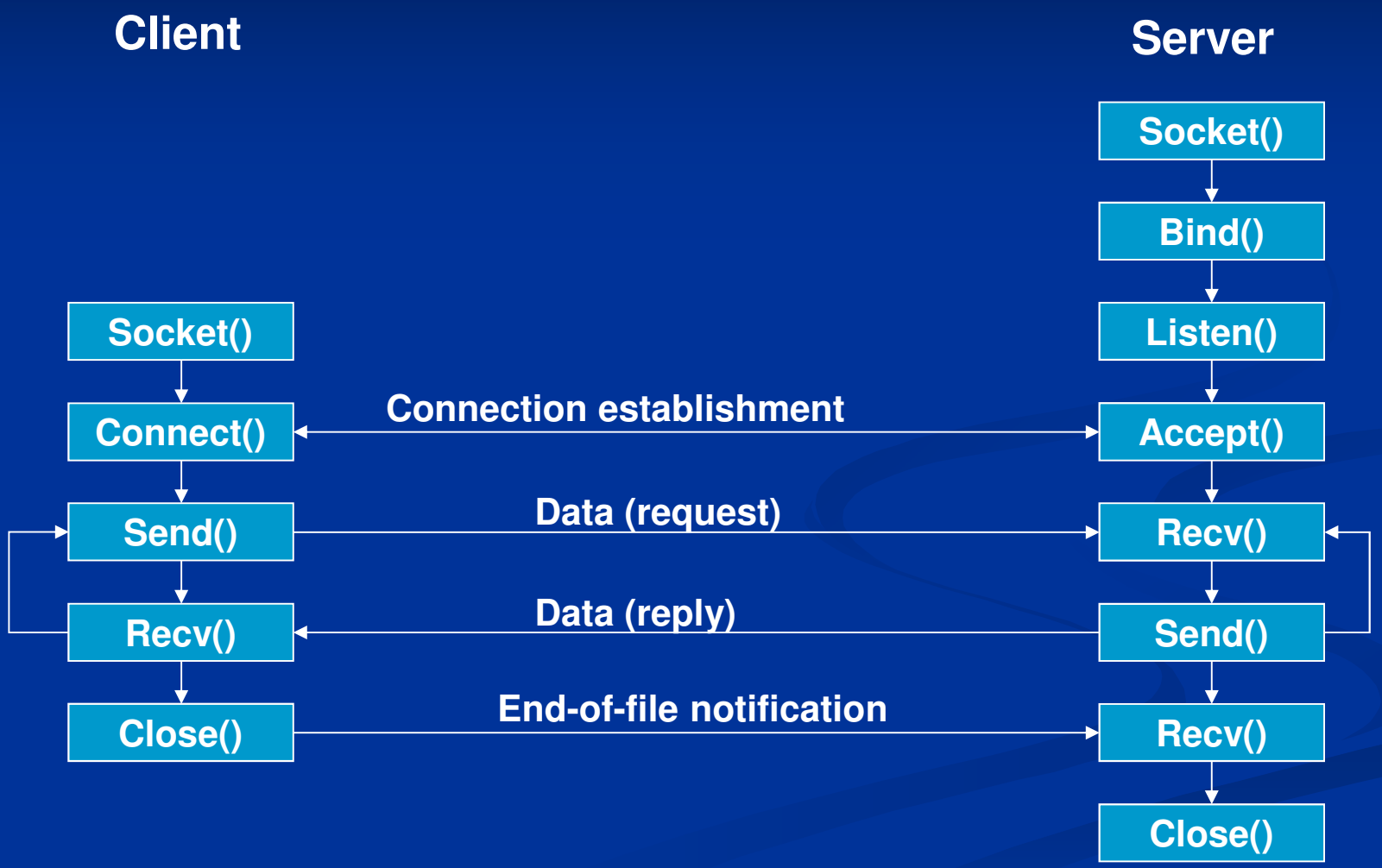
```
/* remove from master set */
                FD_CLR(i, &master);
            }else{
                /* we got some data from a client*/
                for(j = 0; j <= fdmax; j++){
                    /* send to everyone! */
                    if(FD_ISSET(j, &master)){
                        /* except the listener and ourselves */
                        if(j != listener && j != i){
                            if(send(j, buf, nbytes, 0) == -1)
                                perror("send() error lol!");}
                        }
                    }
                }
            }
        }
    }
}
return 0;
}
```

Levo il descrittore da quelli da controllare.

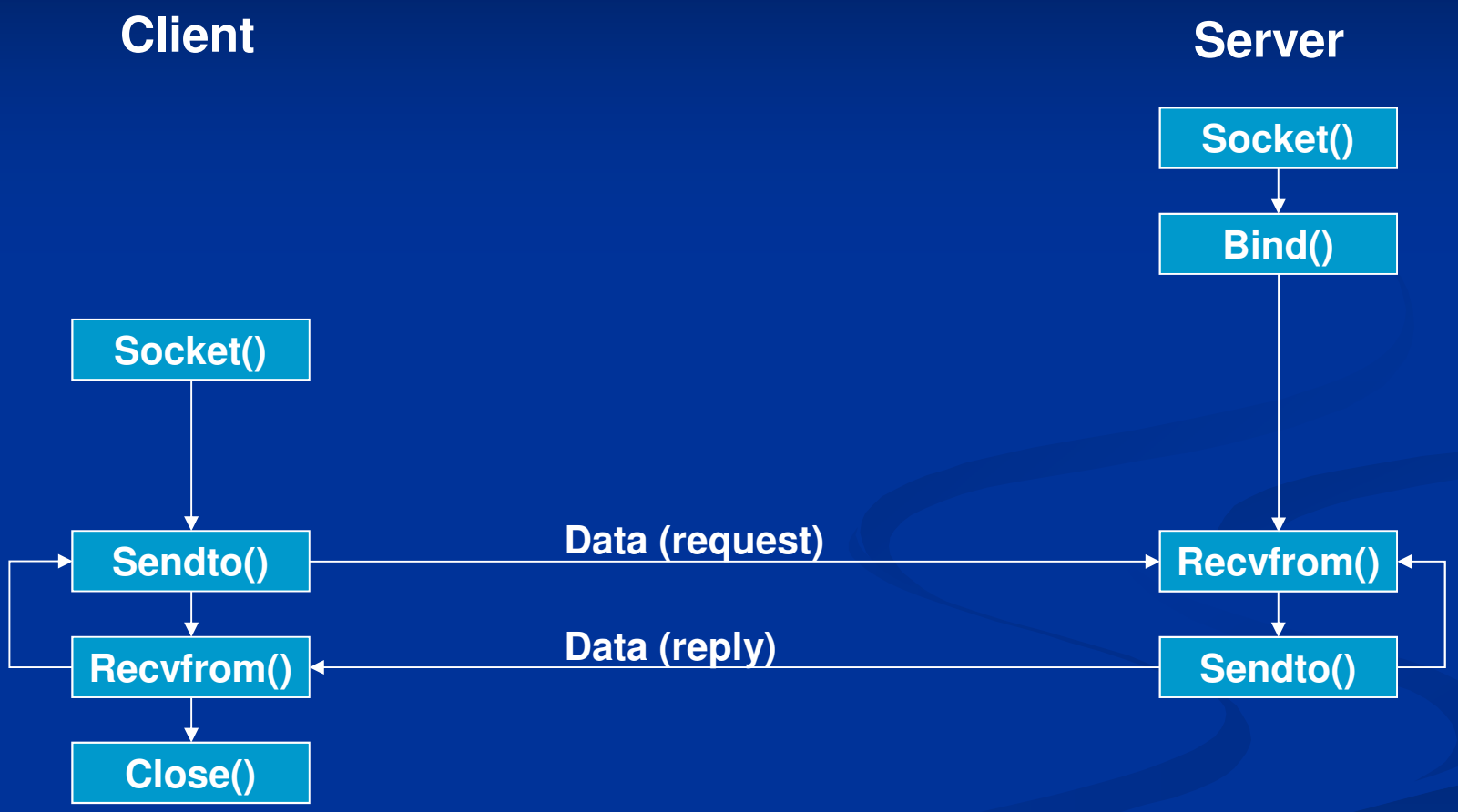
Mando il messaggio ricevuto a tutti gli altri client connessi.

Socket UDP

Interazioni Client-Server TCP



Interazioni Client-Server UDP



Funzioni di Input/Output

```
int recvfrom(int sd, void* buf, int n,
int flags,struct sockaddr* from,
socklen_t *len);
int sendto(int sd, const void* buf, int
n, int flags, const struct sockaddr*
to, socklen_t len);
```

- Per leggere o scrivere su un socket UDP si utilizzano funzioni di sistema differenti da TCP

- Restituiscono
  - numero di byte letti/scritti se OK (>= 0)
  - -1 se c'è un errore

Controllo sul mittente

- Un'applicazione in ascolto su una porta UDP accetta tutti i datagram ricevuti
  - Il client deve controllare se il datagram ricevuto è la risposta del server o proviene da un'altra applicazione
  - Tutti i datagram ricevuti che non provengono dal server devono essere scartati
- Possibile soluzione
  - Il client confronta l'indirizzo del socket da cui ha ricevuto il datagram con quello del socket a cui ha inviato la richiesta

Esempio Client UDP

```
#define BUFLEN 1024

int main(int argc, char **argv) {
    char buf[ BUFLEN] ;
    int s;
    struct sockaddr_in sa;
    int dport;

    if (argc != 3) {
        fprintf(stderr, "Usage: udp-send-client <host> <port>n");
        exit(1);
    }
    dport = atoi (argv[ 2] );

    /* open the udp socket */
     if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    /* fill the address structure */
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(dport);
```

Esempio Client UDP (2)

```
    /* resolve the name */
    if (inet_aton(argv[1], &sa.sin_addr) == 0) {
        struct hostent *he;

        /* argv[ 1] doesn't appear a valid IP address. try to resolve as name */
        he = gethostbyname(argv[ 1]);
        if (!he) {
            fprintf(stderr, "Can't resolve '%s'n", argv[ 1]);
            exit(1);
        }
        sa.sin_addr = * (struct in_addr*) he->h_addr;
    }

    /* loop: send a UDP packet to host/port for every input line */
    while(fgets(buf, BUFLEN, stdin)) {
        int wlen;

     wlen = sendto(s, buf, strlen(buf), 0, (struct sockaddr*)&sa, sizeof(sa));
    if (wlen == -1) {
        perror("sendto");
        exit(1);
        }
    }
    close(s);
    return 0;
}
```

Esempio Server UDP

```
#define BUFLEN 1024

int main(int argc, char **argv) {

    char buf[ BUFLEN];
    int s;
    struct sockaddr_in sa;
    int dport;

    if (argc != 2) {
        fprintf(stderr, "Usage: udp-print-server <port>\n");
        exit(1);
    }
    dport = atoi (argv[ 1] );

    /* open the udp socket */
     if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }

    /* fill the address structure */
    memset(&sa, 0, sizeof(sa));
    sa.sin_family = AF_INET;
    sa.sin_port = htons(dport);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);
```

Esempio Server UDP (2)

```
    /* bind the socket */
    if (bind(s, (struct sockaddr*) &sa, sizeof(sa)) == -1) {
        perror("bind");
        exit(1);
    }

    /* loop: read UDP data from <port> and print to standard output */
    while(1) {
        int rlen;
        struct sockaddr_in ca;
        socklen_t calen = sizeof(ca);

        rlen = recvfrom(s, buf, BUFLEN-1, 0, (struct sockaddr*)&ca, &calen);
        if (rlen == -1) {
            perror("sendto");
            exit(1);
    } else if (rlen == 0) {
        break;
    }

    printf("[ %s] %s", inet_ntoa(ca.sin_addr), buf);
    }
    close(s);
    return 0;
}
```

Datagrammi perduti

- Se un datagram si perde (es. un router lo butta via) l'applicazione che lo sta attendendo può rimanere bloccata in una recvfrom()
  - In alcuni casi è possibile porre un timeout sulla recvfrom()
  - Comunque non è possibile scoprire se il messaggio del client non è mai arrivato al server oppure se la risposta del server non è arrivata al client

Errori asincroni

- Una funzione provoca un errore asincrono se il segnale di errore arriva dopo il completamento della funzione
  - Es. la sendto restituisce OK perché il datagram è stato spedito. Successivamente ICMP restituisce un messaggio di errore di "hostunreachable"
- Per default gli errori asincroni non sono passati ad un socket UDP
  - Lo stesso socket può essere utilizzato per inviare datagram a più destinazioni
  - ICMP restituisce l'header del datagram che ha provocato l'errore
  - Il kernel non ha modo di passare queste informazioni all'applicazione

Client UDP Connessi

- E' possibile creare un socket UPD virtualmente connesso utilizzando la funzione connect()
  - Significato differente dalla connect() su socket TCP
- La connect() su un socket UDP implica che il kernel memorizza l'indirizzo IP e la porta con cui si vuole comunicare
  - Il client potrà inviare datagram solo all'indirizzo specificato dalla connect()

Caratteristiche di un socket UDP Connesso

- Può inviare datagram soltanto all'indirizzo specificato nella chiamata alla connect
  - non si usa sendto ma write o send
  - i datagram verranno automaticamente spediti all'indirizzo specificato nella chiamata a connect
- Può ricevere solo datagram inviati dall'indirizzo specificato nella chiamata alla connect
  - non si usa recvfrom, ma read o reacv
  - un server UDP può comunicare con un solo client per volta
- Errori asincroni possono essere controllati
  - Il kernel segnala l'errore in errno