IPC: InterProcess Communication

Uso di segnali → processo invia limitata quantità info (solo un tipo di segnale) e manca il mittente del segnale

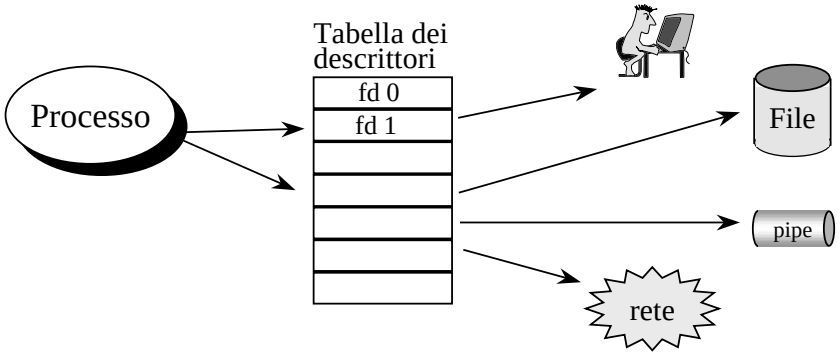
Uso di file → solo tra processi che condividono un file system

IPC
- → pipe (solo tra processi con un avo in comune)
- → shared memory
- caso remoto → **socket**

I Processi e l'I/O

I processi interagiscono con l'I/O secondo il paradigma *open-read-write-close*

Un processo vede il mondo esterno come un insieme di descrittori



Flessibilità (possibilità di pipe e ridirezione)

Anche le Socket sono identificate da un descrittore (socket descriptor), con stessa semantica read/write (es. read blocca in attesa dati)

Programmazione di rete e paradigma *open-read-write-close*

La programmazione di rete richiede delle funzionalità non gestibili in modo completamente omogeneo ai file e alle pipe

- un collegamento via rete, cioè l'associazione delle due parti comunicanti può essere
  - **con connessione** (simile o-r-w-c)
  - **senza connessione**
- fd: trasparenza dai nomi va bene nel caso file (flessibilità), ma nel caso di network programming può non essere adatto
- in programmazione di rete bisogna specificare più parametri per un collegamento:
  <protocollo;indirizzo locale;processo locale;indirizzo remoto;processo remoto>

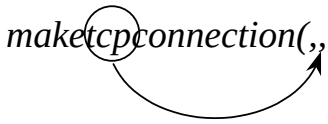
Programmazione di rete e paradigma *open-read-write-close*

- UNIX I/O è orientato allo stream, non a messaggi di dimensioni prefissate → Alcuni protocolli di rete fanno uso di messaggi di dimensioni prefissate
- è necessario che l'interfaccia socket possa gestire più protocolli
- schemi Client/Server sono asimmetrici

L’interfaccia Socket

UNIX + TCP/IP ⇨ BSD UNIX (Progetto Università Berkeley finanziato da ARPA)

Il BSD UNIX implementa un’interfaccia generale per la programmazione di rete, di cui l’interfaccia di programmazione al TCP/IP è un caso particolare



L’interfaccia Socket
Obiettivi

Meccanismo di comunicazione per processi residenti su nodi diversi

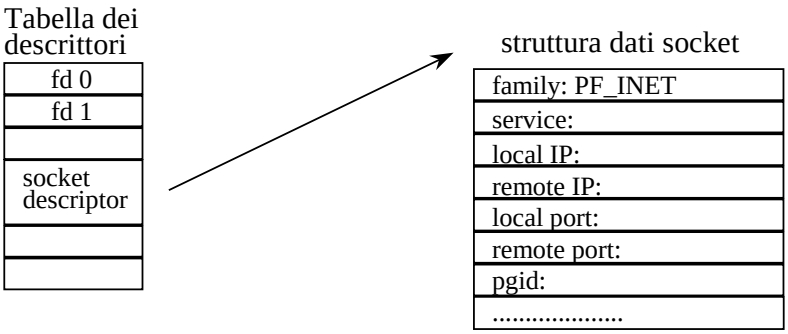
Interfaccia indipendente dall’architettura di rete sottostante

Supportare diversi insiemi di protocolli, diversi hardware, diverse convenzioni sulla specifica dei nomi, etc.

Socket

E' necessario fornire una nuova **astrazione** per definire un canale di comunicazione e in particolare i suoi terminali

socket: è un oggetto che rappresenta un terminale di un canale di comunicazione bidirezionale

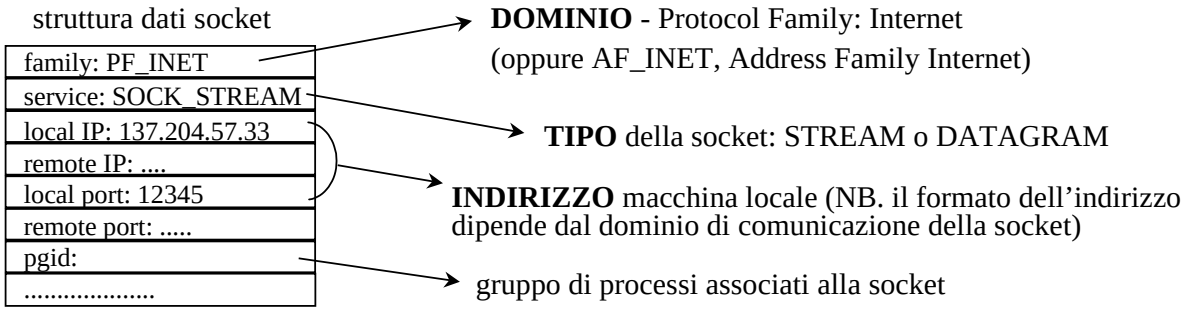


Socket

Una socket è creata all'interno di un dominio di comunicazione

Dominio di comunicazione:
semantica di comunicazione + standard di denominazione

Esempi di domini: UNIX, **Internet**, etc.



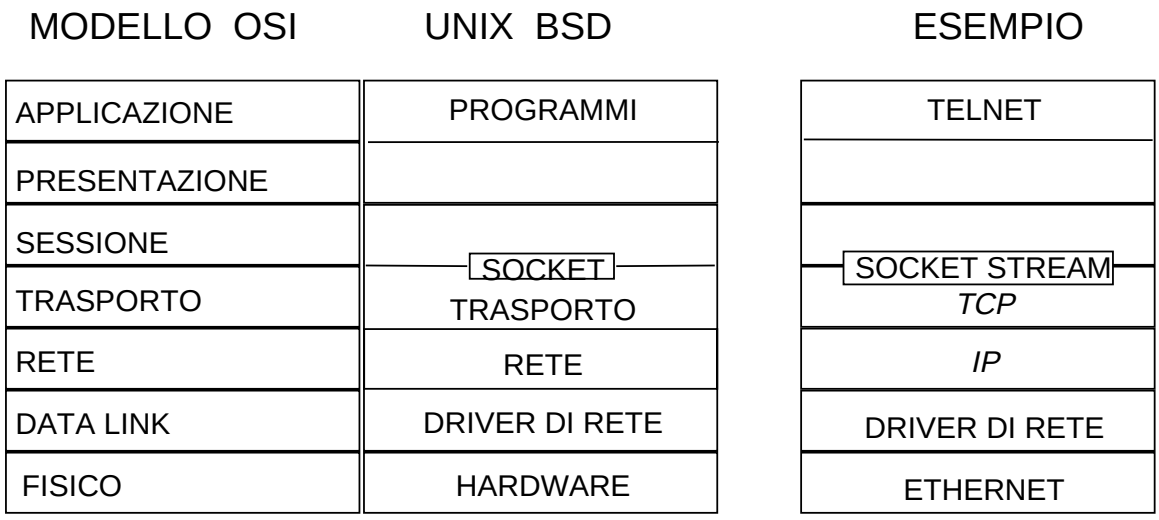
Domini di comunicazione di una Socket

DOMINI	descrizione
PF_UNIX	comunicazione locale tramite pipe
PF_INET	comunicazione mediante i protocolli ARPA internet (**TCP/IP**)
.....	.....

Il dominio di comunicazione è identificato dalla Protocol Family (es. PF_INET) oppure dall'Address Family (es. AF_INET)

In teoria sarebbe possibile definire più tipi di Address Family (sistemi di naming) per una specifica Protocol Family

DOMINIO Internet (PF_INET, AF_INET)

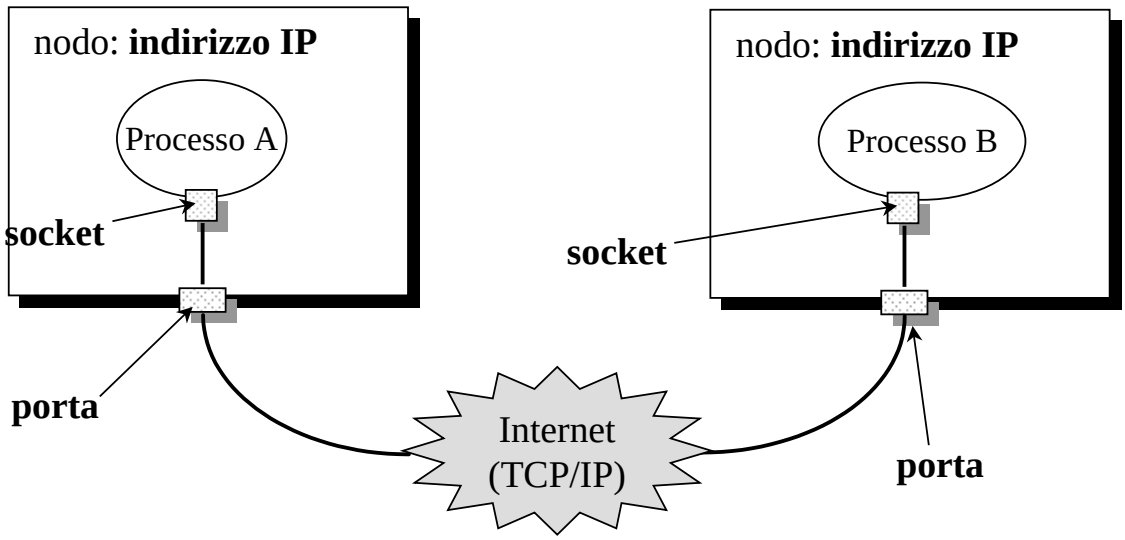


FORMATO INDIRIZZI

DOMINIO AF_ UNIX : L'indirizzo ha lo stesso formato del nome di un file (pathname), dimensione massima 108 bytes.

DOMINIO AF_INET **: indirizzo Internet composto da 32 bit per denotare il nodo della rete (indirizzo IP del host) e 16 bit per denotare una delle possibili 64K porte**

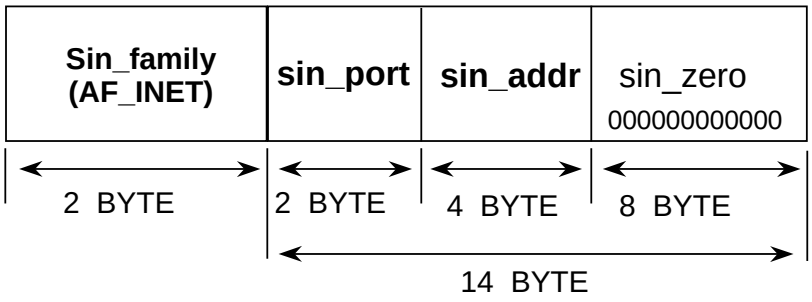
Socket



Formato indirizzi - Dominio AF_INET

struttura sockaddr_in

```
struct   sockaddr_in
         {
         short  sin_family ;
         u_short  sin_port ;
         struct  in_addr  sin_addr;
         char  sin_zero [8] ;
         }
```



Tipi di Socket

TIPO	descrizione
SOCK_STREAM	terminale di un canale di comunicazione **con connessione** (socket stream o TCP)
SOCK_SEQPACKET	trasferimento affidabile di sequenze di pacchetti
SOCK_DGRAM	terminale di un canale di comunicazione **senza connessione** (socket datagram o UDP)
SOCK_RAW	accesso diretto al livello di rete (IP)

Tipi di Socket

Una **socket STREAM** stabilisce una **connessione** (socket TCP, SOCK_STREAM).
La comunicazione è **affidabile**, **bidirezionale** e i byte sono consegnati in **sequenza**.
Non sono mantenuti i **confini dei messaggi.**
(presenza di meccanismi out-of-band)

Una **socket DATAGRAM** non stabilisce alcuna connessione (**connectionless**) (socket UDP, SOCK_DGRAM).
NON c'è garanzia di consegna del messaggio.
I **confini dei messaggi** sono mantenuti.
Non è garantito **l'ordine** di consegna dei messaggi.

Quale tipo di Socket utilizzare?

Fare molta attenzione alle differenze semantiche tra le socket STREAM e quelle DATAGRAM, che in genere guidano nella scelta:

servizi che richiedono una connessione ⟷ servizi connectionless

Prestazioni: le socket DATAGRAM hanno un costo inferiore (non si deve stabilire una connessione, etc.)

Esiste un numero massimo di connessioni TCP che si possono aprire.

Nel caso DATAGRAM può essere necessario portare a livello di applicazione i controlli sulla perdita dei messaggi o sul non ordinamento degli stessi.

Quale tipo di Socket? Quale livello di trasporto, UDP o TCP?

Utilizzare TCP (**con connessione**) quando:

- affidabilita' fondamentale
- l'ordine dei messaggi e' importante
- semantica exactly once
  (servizio del server non idempotente, server con stato)

Utilizzare UDP (**senza connessione**) quando:

- le prestazioni sono fondamentali
- troppe connessioni sarebbero richieste
- non ci sono problemi di ordinamento messaggi
  (es. ogni msg contenuto in un pacchetto UDP)
- semantica more than once

Creazione di una socket

sd = **socket** (dominio, tipo, protocollo);
int sd, dominio, tipo, protocollo;

Crea una SOCKET e ne restituisce il **descrittore** sd (socket descriptor).

dominio denota il particolare dominio di comunicazione (es. AF_INET)

tipo indica il tipo di comunicazione (es. SOCK_STREAM o SOCK_DGRAM)

protocollo specifica uno dei protocolli supportati dal dominio (se si indica zero viene scelto il protocollo di default)

Definisce il protocollo usato dalla socket:
<***protocollo****;indirizzo locale;porta locale;indirizzo remoto;porta remota*>

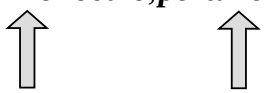
Associazione socket - indirizzo locale

```
error = bind (sd, ind, lun);
int error, sd;
struct sockaddr * ind;
int lun;
```

Associa alla socket di descrittore sd l'indirizzo codificato nella struttura puntata da ind e di lunghezza lun (la lunghezza e' necessaria, poiche' la funzione bind puo' essere impiegata con indirizzi di lunghezza diversa)

Collega la socket a un indirizzo locale:

<*protocollo;**indirizzo locale**;**porta locale**;indirizzo remoto;porta remota*>

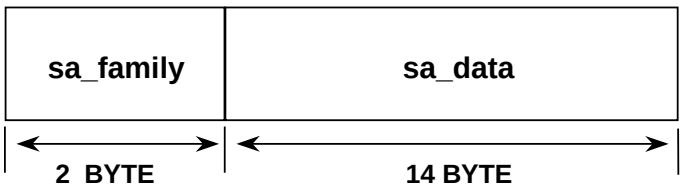


Formato indirizzi

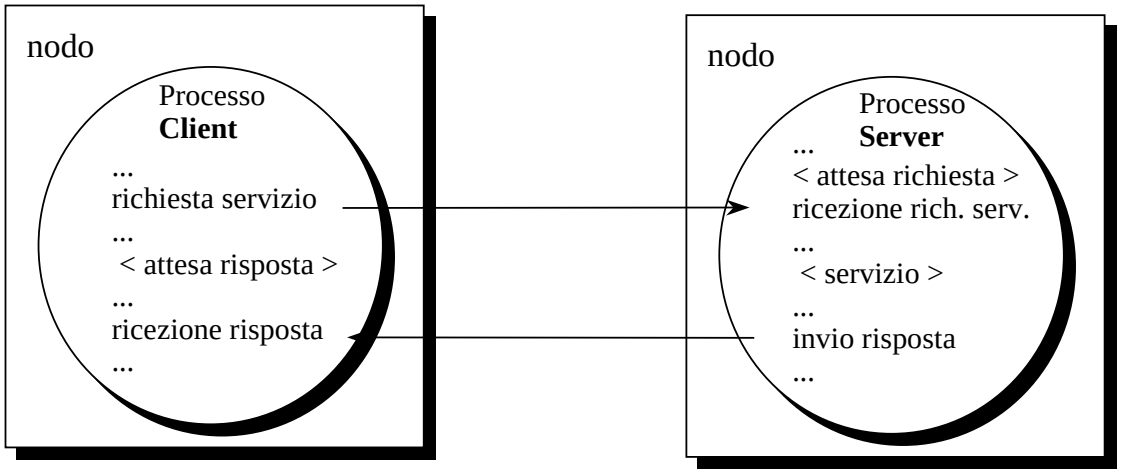
Le funzioni per la programmazione di rete (es. bind, connect) possono ricevere diversi tipi di indirizzo a seconda della famiglia di protocolli scelta (es. AF_INET)

il prototipo richiede un generico puntatore (void * in ANSI C)

definizione di una generica struttura `sockaddr` (e cast)



Il modello Client/Server



Comunicazione **asimmetrica**, molti:1

Il Cliente nomina esplicitamente il destinatario
Il Servitore non esprime il nome del processo con cui desidera comunicare

Comunicazione connection-oriented

(socket STREAM o TCP)

Collegamento (asimmetrico) tra processo Client e processo Server:

1) il server e il client devono **creare ciascuno una propria socket** e definirne l'indirizzo (primitive socket e bind)

2) deve essere creata la **connessione** tra le due socket

3) fase di **comunicazione**

4) **chiusura** delle socket

Comunicazione connection-oriented

2) Creazione della connessione tra il client e il server (schema **asimmetrico**):

LATO CLIENT
Il client richiede una connessione al server (primitiva **connect**)
Uso di una socket attiva

LATO SERVER
Il server definisce una coda di richieste di connessione (primitiva listen) e attende le richieste (primitiva **accept**)
Uso di una socket passiva (listening socket)

Quando arriva una richiesta, la richiesta viene accettata, stabilendo così la connessione. La comunicazione può quindi iniziare.

Comunicazione connection oriented
(lato **Client**)

```
error = connect (sd, ind, lun) ;
int error, sd;
struct sockaddr * ind;
int lun;
```

Richiede la connessione fra la socket locale il cui descrittore è sd e la socket remota il cui indirizzo è codificato nella struttura puntata da ind e la cui lunghezza è lun

Definisce l'indirizzo remoto a cui si collega la socket:
<*protocollo;indirizzo locale;porta locale;****indirizzo remoto****;****porta remota***>

Comunicazione connection oriented
(lato **Server**)

```
error = listen (sd , dim);
int error , sd , dim;
```

Trasforma la socket in passiva (listening), pronta per ricevere una richiesta di connessione

Crea una coda in cui vengono inserite le richieste di connessione con la socket di descrittore sd, provenienti dai client.

La coda può contenere al più dim elementi.

Le richieste di connessione vengono estratte dalla coda quando il server esegue la accept().

Comunicazione connection oriented
(lato **Server**)

```
nuovo_sd = accept (sd , ind, lun);
int nuovo_sd , sd;
struct sockaddr * ind;
int * lun;
```

Indirizzo e' parametro in/out

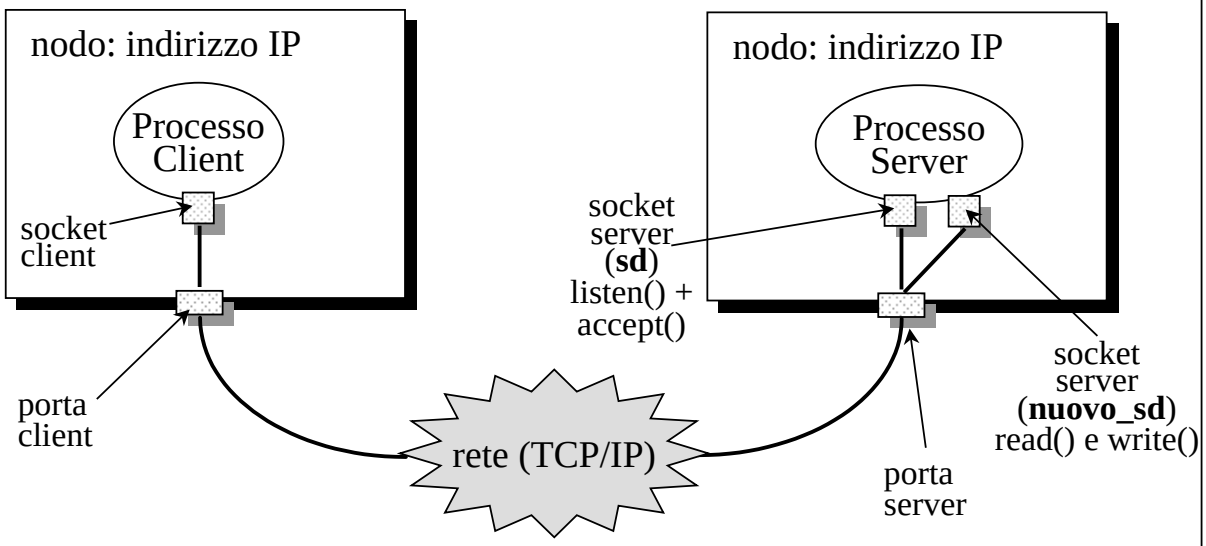
Estrae una richiesta di connessione dalla coda predisposta dalla listen().

Se **non** ci sono richieste di connessione in coda, **sospende** il server finchè non arriva una richiesta alla socket sd.

Quando la richiesta arriva, crea la socket di lavoro nuovo_sd e restituisce l'indirizzo della socket del client tramite ind e la sua lunghezza tramite lun.

La comunicazione (read/write) si svolge sulla nuova socket nuovo_sd

Comunicazione connection oriented



Comunicazione connection oriented

Una volta completata la connessione si possono utilizzare le normali primitive read e write

nread = **read** (sd, buf, lun);
nwrite = **write** (sd, buf, lun);

buf è il puntatore a un buffer di lunghezza lun dal quale prelevare o in cui inserire il messaggio.

sd è il socket descriptor.

uso di send e recv per dati out-of-band.

Comunicazione connection oriented (termine connessione)

Al termine della comunicazione, la connessione viene interrotta mediante la primitiva close che chiude la socket

```
error  =  close (sd);
int  sd;
```

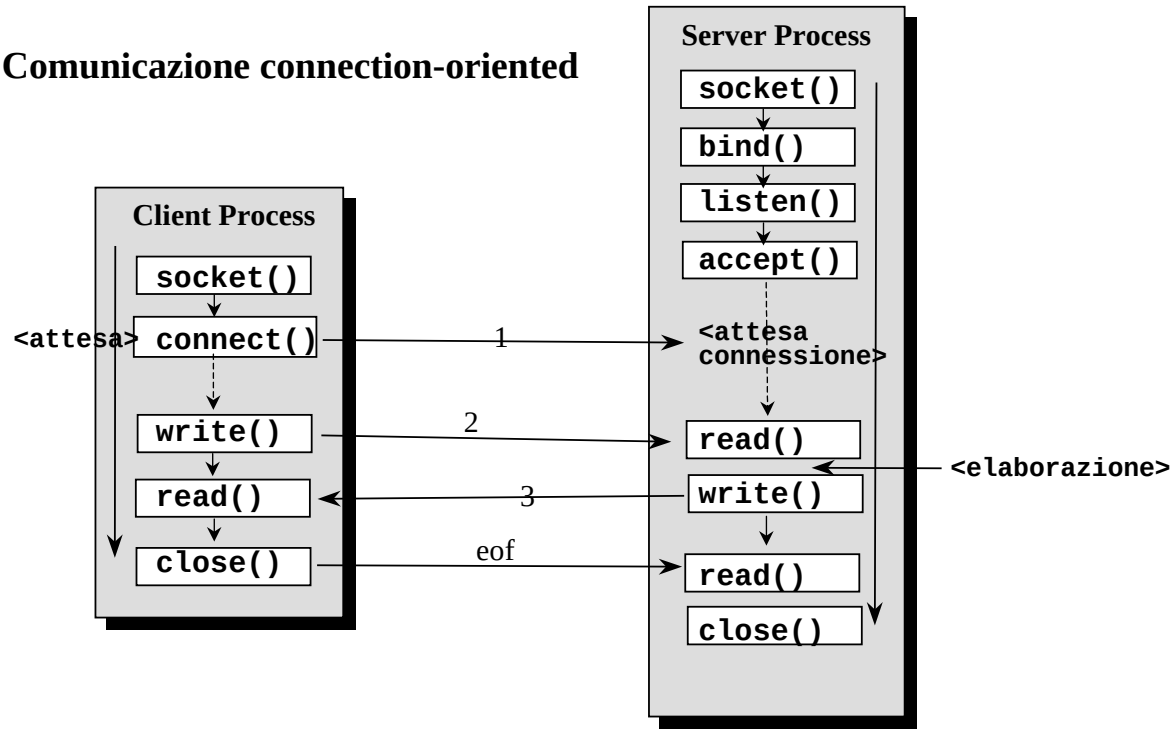
Problemi close():

- chiusura connessione avviene solo se sd e' l'**ultimo** descrittore aperto della socket;
- chiusura di entrambi gli estremi della connessione;

```
error  =  shutdown (sd, how);
int  sd, how;
```

Per terminare una connessione si può usare anche la shutdown, che fornisce più controllo:
chiusura solo dell'estremo di ricezione (how=0, SHUT_RD)
chiusura solo dell'estremo di trasmissione (invio eof) (how=1, SHUT_WR)
chiusura dei due estremi (how=2, SHUT_RDWR)

Comunicazione connection-oriented



Esempio: Dominio **Internet**, socket con connessione (tipo **stream**)

Struttura **Client** (creazione socket e connessione)

```
main ( )
{
  .......
  sd = socket (AF_INET, SOCK_STREAM , 0 );
  .......
  <la bind è opzionale per il client>
  .......
  <preparazione nella struttura rem_indirizzo dell'indirizzo del server>
   .......
  connect (sd , (struct sockaddr *) &rem_indirizzo, sizeof (rem_indirizzo));
   .....
```


Esempio: Client

Struttura Client (comunicazione)

```
  ......
  write (sd, buf, dim);
  read (sd, buf, dim);
  ......
  <chiusura, uso di close o shutdown>
  close (sd);
  .......
  exit(0);
}
```

Esempio: Server

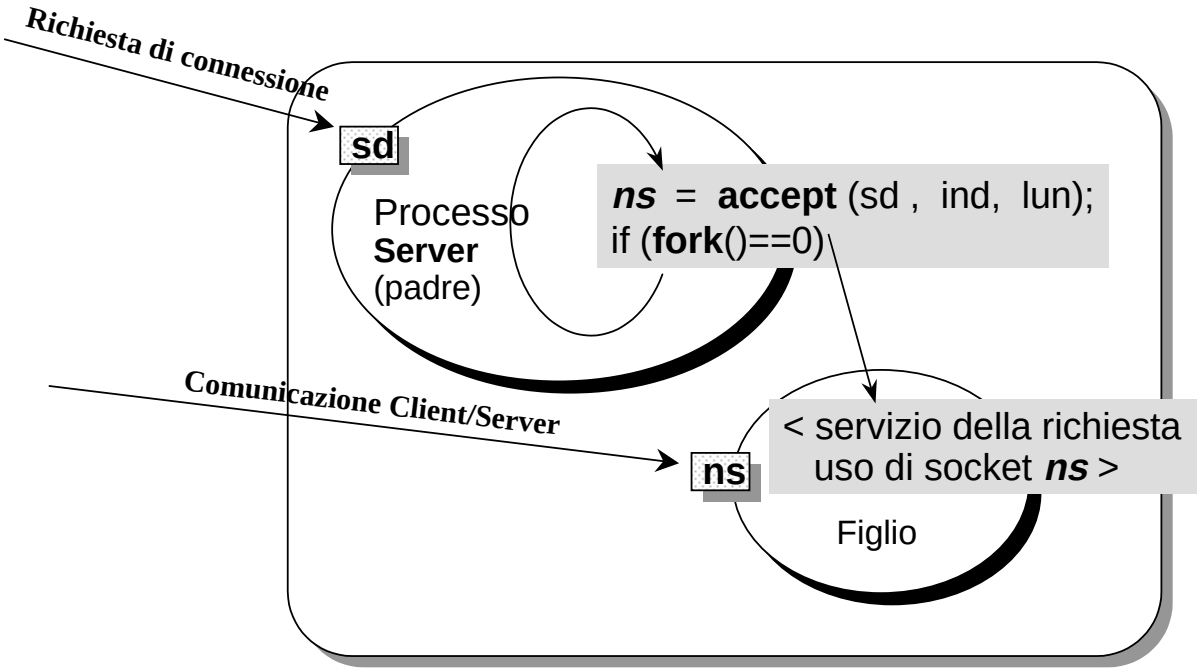
```
main ( )
{
  .........
  sd = socket (AF_INET, SOCK_STREAM , 0);
  .........
  <preparazione nella struttura mio_indirizzo dell'indirizzo del server>
  .........
  bind (sd, & mio_indirizzo, sizeof(mio_indirizzo));
  .........
  listen (sd, 5);
  .........
```

IP + porta

Esempio: Server

```
.........
for ( ; ; )
      {
      ns = accept (sd, 0 , 0 );
      .........
      nread = read (ns, buf, dim);
      .........

      close (ns);
      }
}
```

Server *concorrente* multi-processo connection-oriented



			Protocollo di rete	
			con connessione	senza connessione
Tipo di Server	iterativo			
	concorrente	singolo processo		
		multi processo		

Comunicazione connectionless
(socket DATAGRAM)

Non viene creata una connessione tra processo client e processo server.

1) il server e il client devono creare ciascuno una propria socket (primitiva **socket)**; il server deve legare la propria socket a una porta (primitiva **bind**)

2) i processi possono comunicare specificando, per ogni messaggio, la socket del destinatario

Attenzione che UDP e' un protocollo:
[connectionless
[unreliable
[datagram

Comunicazione connectionless
(socket DATAGRAM)

int **sendto**(int sd, const char *msg, int len, int flags,
const struct sockaddr *to, int tolen);

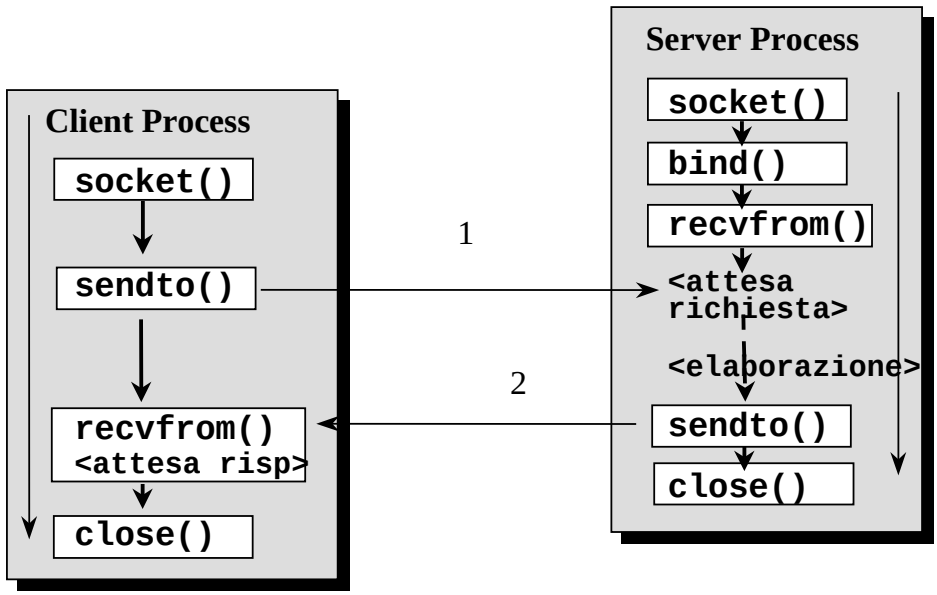
int **recvfrom**(int sd, char *buf, int len, int flags, struct sockaddr *from,
int *fromlen);

Le primitive recvfrom e sendto, oltre agli stessi parametri delle read e write, hanno anche due parametri aggiuntivi che denotano indirizzo e lunghezza di una struttura socket:
• nella sendto servono per specificare l'indirizzo del destinatario;
• nella recvfrom servono per restituire l'indirizzo del mittente.

La recvfrom sospende il processo in attesa di ricevere il messaggio (coda si messaggi associata alla socket).

La recvfrom puo' ritornare zero, se ha ricevuto un messaggio (un datagramma) di dimensione zero.

Comunicazione senza connessione
(socket DATAGRAM)



Nomi logici dei nodi e indirizzi fisici

struct hostent ***gethostbyname**(const char *name);
struct hostent ***gethostbyaddr**(const char *addr, int len, int type);

gethostbyname riceve in ingresso il nome logico di un host Internet e restituisce una struttura **hostent** (contenente l'indirizzo fisico IP del nodo).

gethostbyaddr riceve in ingresso l'indirizzo fisico di un host Internet e restituisce una struttura **hostent** (contenente in nome della macchina).

```
struct hostent {
        char   *h_name;        /* canonical name of host */
        char   **h_aliases;    /* alias list */
        int    h_addrtype;     /* host address type */
        int    h_length;       /* length of address */
        char   **h_addr_list;  /* list of addresses */
       #define h_addr  h_addr_list[0]  /* address, for backward compatiblity */
};
```

Preparazione di un indirizzo remoto

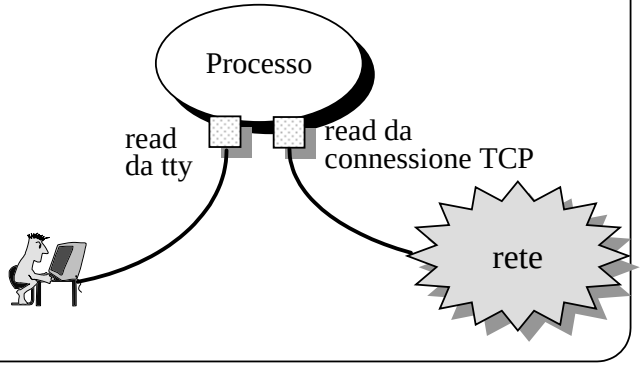
```
struct hostent *host;                   /*ptr a info per host remoto*/
struct sockaddr_in rem_indirizzo;       /*indirizzo processo remoto*/

.....
memset ((char *)&rem_indirizzo, 0, sizeof(struct sockaddr_in));

/* Preparazione indirizzo remoto a cui connettersi */
rem_indirizzo.sin_family = AF_INET;
host = gethostbyname("nome_host_remoto");  /* host rem, es.: deis33 */
if (host == NULL) {
        printf("nome_host_remoto not found in /etc/hosts\n"); exit(2);
}

rem_indirizzo.sin_addr.s_addr=((struct in_addr *) (host->h_addr))->s_addr;
rem_indirizzo.sin_port = 22375;       /*possibile uso htons()*/
```


Modelli di Input/Output

- I/O bloccante
- primitive non-bloccanti (scarsa efficienza del polling)
- I/O guidato dai segnali, in cui il kernel invia un segnale (SIGIO) quando riceve dei dati su un descrittore (problema: unico segnale per tutti descrittori)
- I/O multiplexing (select), per sapere quale descrittore e' pronto per I/O
- I/O asincrono (estensione real time di POSIX.1)

I/O guidato dai segnali: kernel avvisa quando I/O puo' essere **iniziato** senza che il processo si blocchi

I/O asincrono: kernel avvisa quando I/O puo' essere **completato** senza che il processo si blocchi



Multiplexing Sincrono di Input/Output

Si utilizza la primitiva **`select()`** che consente al processo utente di attendere un qualsiasi evento tra vari eventi attesi.

Gli argomenti che si passano alla **`select()`** specificano al kernel:

- a quali descrittori siamo interessati
- a quali condizioni siamo interessati per i descrittori
  (lettura, scrittura o condizioni eccezionali)
- quanto vogliamo aspettare

Quando la **`select()`** ritorna, il kernel ci dice:

- il numero totale di descrittori che sono pronti
- quali descrittori sono pronti per ognuna delle tre condizioni specificate

Primitiva **`select()`** per eseguire il multiplexing sincrono dell'I/O.

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

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

`numfds` è il numero max di descrittori che la select esamina (da `0` a `numfds-1`)

`timeout`

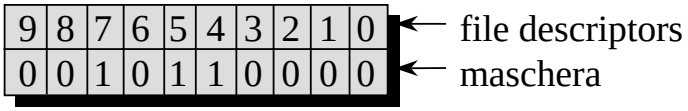
se `timeout`	punta a NULL, attesa indefinita di un evento
	punta a una struttura con un tempo 0, polling dei descrittori
	punta a una struttura con un tempo non 0, intervallo di timeout

`select()` per attesa contemporanea di eventi di I/O su più socket descriptor.

La `select()` esamina i descrittori per vedere se sono pronti per

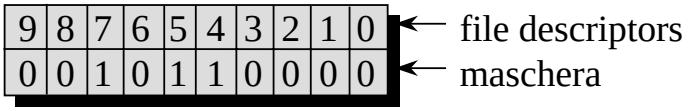
- la lettura (`readfds`)
- la scrittura (`writefds`)
- o se hanno qualche condizione eccezionale sospesa (`exceptfds`)

`readfds, writefds, exceptfds` sono delle maschere codificate in binario.



Le **maschere** `readfds, writefds, exceptfds` specificano alla select quale insieme di descrittori esaminare (NULL se non interessano).

Operazioni sulle maschere dei descrittori




```
void FD_SET(int fd, fd_set &fdset);
void FD_CLR(int fd, fd_set &fdset);
int  FD_ISSET(int fd, fd_set &fdset);
void FD_ZERO(fd_set &fdset);
```

`FD_SET` include il particolare fd in fdset
`FD_CLR` rimuove fd dal set fdset
`FD_ISSET` restituisce un valore diverso da zero se fd fa parte del set fdset, zero altrimenti
`FD_ZERO` inizializza l'insieme di file descriptor a zero

Primitiva select

Ci possono essere tre possibili valori di ritorno dalla `select()`:

- -1 in caso di errore
- il numero di descrittori pronti (nelle stesse maschere `readfds, writefds, exceptfds` restituisce il **sottoinsieme** di descrittori pronti)
- 0 se non ci sono descrittori pronti ed è finito il timeout

Cosa significa che un descrittore è pronto

condizione di lettura: (una read non bloccherebbe)
in una socket sono presenti dati da leggere
una socket può accettare una connessione (accept non blocca)
in una socket si è verificato un errore
un end of file è considerato una condizione di lettura

condizione di scrittura: (una write non bloccherebbe)
in una socket la connessione è completata
in una socket si possono spedire altri dati
in una socket si è verificato un errore

condizione eccezionale: (una condizione eccezionale è in attesa sul descrittore)
arrivo dati out-of-band

Il fatto che un descrittore sia **non bloccante** non influisce sul comportamento della select, che si blocca per il tempo del time out se il descrittore non è pronto

Non blocking I/O

Le primitive viste sono tipicamente bloccanti.

Si possono però rendere non bloccanti settando un opportuno flag associato al file descriptor o al socket descriptor. In questo modo le primitive che non possono essere completate per un qualunque motivo non vengono effettuate.

Utilizzo di `fcntl()` o `ioctl()`

```
fcntl(descrittore, F_SETFL, FNDELAY);
```

Con fcntl() possibile uso di O_NDELAY o O_NONBLOCK

```
int arg=1; /* valore 0 non bloccante, valore 1 bloccante */
ioctl(descrittore , FIONBIO, arg);
```


Non blocking I/O

chiamate modificate:

`accept``()`
ritorna un errore di tipo EWOULDBLOCK

`connect``()`
ritorna un errore di tipo EINPROGRESS

`read``()`
errore EWOULDBLOCK

`write``()`
errore EWOULDBLOCK

I/O guidato dai segnali

Nel caso di I/O guidato dai segnali, il kernel avvisa un processo quando un descrittore specificato è pronto per l'I/O.

La notifica avviene a mezzo segnale.

Per utilizzare l'I/O guidato dai segnali un processo deve:

- definire un gestore del segnale **SIGIO**
- stabilire il **pid** del processo o il **pgid** del gruppo di processi a cui deve essere inviato il SIGIO
- **abilitare** l'I/O signal driven

I/O guidato dai segnali : il SIGIO

Il segnale **SIGIO** indica che un descrittore (es. una socket) è pronto per eseguire dell'I/O.

Può essere inviato a un singolo processo oppure a un gruppo di processi.

Problema:
un processo può avere **un solo gestore** per un determinato segnale.
se più descrittori sono pronti per l'I/O signal driven, quando arriva un SIGIO non si sa quale sia il descrittore pronto (uso di **`select`**)

I/O guidato dai segnali : esempio

```
int    io_handler(); /* SIGIO interrupt handler */

/* associamo il gestore io_handler al segnale SIGIO */
signal (SIGIO, io_handler);


/* definiamo il processo o il gruppo di processi a cui il
                                    kernel manderà il SIGIO */
if (fcntl(s, F_SETOWN, getpid()) < 0) {
       perror(" errore di F_SETOWN "); exit(1);
} /* Oppure -getpgid() per gruppo */


/* abilitiamo la ricezione dei segnali di I/O */
if (fcntl(s, F_SETFL, FASYNC) < 0) {
       perror(" errore di F_SETFL FASYNC "); exit(1);
}
```


I/O guidato dai segnali: esempio di Server

```
main()
{
   ...
   signal (SIGIO, io_handler);
   ...
   <preparazione indirizzo della socket datagram del server>
   ds = socket(AF_INET, SOCK_DGRAM, 0); /*crea datagram socket*/
   bind(ds, &myaddr, sizeof(myaddr));  /* Bind datagram socket */
   set_up_async(ds);  /*  rende asincrona la socket */

   ...
   <preparazione indirizzo della socket stream del server>
   ss = socket(AF_INET, SOCK_STREAM, 0); /* stream socket */
   bind(ss, &myaddr, sizeof(myaddr));  /* Bind stream socket */
   set_up_async(ss);  /* rende asincrona la socket */

   listen(ss, 5);
```

Server con I/O guidato dai segnali

```
        ...

        FD_ZERO(&readmask);    /*  setup iniziali per le mask */
        FD_SET(ds, &readmask);
        FD_SET(ss, &readmask);
        ...
        for ( ; ; ) { pause(); }

} /* end main */


set_up_async(s)
{
        ...
        flag = getpid();    /* SIGIO al processo */
        fcntl(s, F_SETOWN, flag)
        ...
        fcntl(s, F_SETFL, FASYNC)
}
```


Server con I/O guidato dai segnali
gestore del SIGIO

```
void io_handler()
{
   /*  select dei socket descriptors */
   numfds = select(max + 1, &mask, (char *)0, (char *)0,&timeout);

   if (FD_ISSET(ds, &mask)) {
      ......
      count = recvfrom(ds, buf, BUFLEN, 0, &peeraddr, &lung);
      .....
   }

   if (FD_ISSET(ss, &mask)) {  /* richiesta di connessione */
      ....
      newsock = accept(ss, &peeraddr, &lung);
      .....
      FD_SET(newsock, &readmask);  /*aggiunta newsock a readmask*/
      set_up_async(newsock);
   }
   ...
```

Server con I/O guidato dai segnali
gestore del SIGIO

```
...
for (h = ss + 1; h < num_sock_connesse ; h++) {
   ...
   if (FD_ISSET(h, &mask)) {
      ...
      count = read(h, buf, BUFLEN, 0); /* rx dati stream */
      ...
      if(count==0){ /* la read ha ritornato 0, è stato chiuso
                                    l'estremo di scrittura */
           FD_CLR(h, &readmask);
           close(h);
      }
   }
}
```