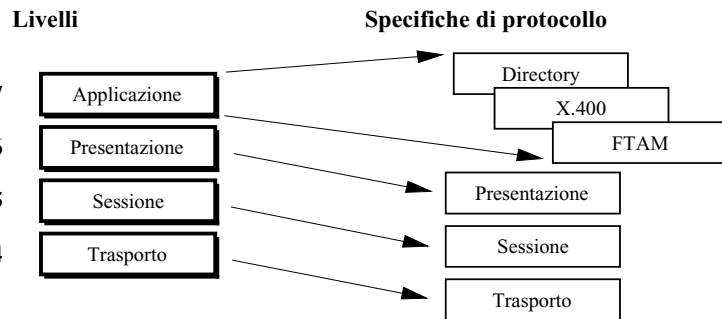
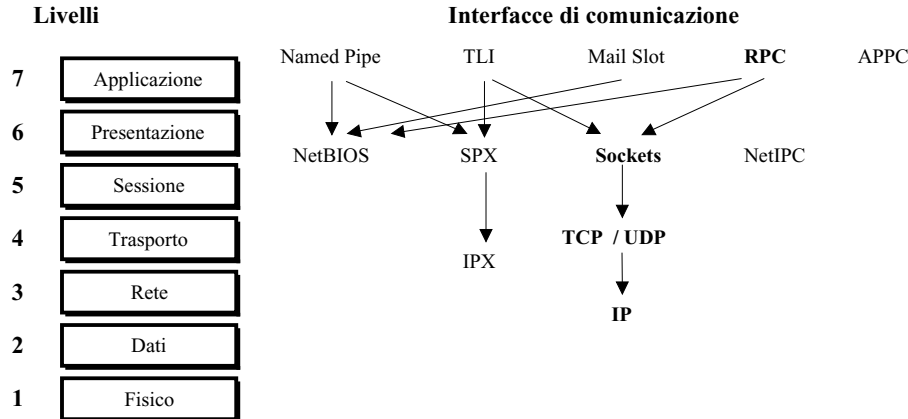


Strumenti di Comunicazione per lo scambio di messaggi

Livelli di strumenti diversi



Necessità di definire e di diffondere l'uso di strumenti standard di comunicazione

socket come endpoint per comunicare in modo flessibile, differenziato ed efficiente

UNIX per la comunicazione/sincronizzazione

Uso di segnali

*processo invia limitata quantità info
solo un **tipo di segnale** e
senza indicazione del mittente del segnale*

Uso di file

*solo tra processi che condividono lo stesso file system
processi locali allo stesso nodo*

Comunicazione locale =>

pipe (solo tra processi con un avo in comune)

pipe con nome (per processi su una stessa macchina)

shared memory (stessa macchina)

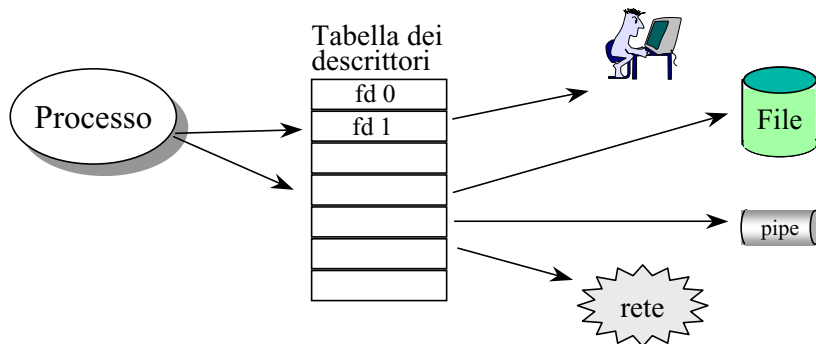
COMUNICAZIONE REMOTA => SOCKET

Interfaccia e definizione iniziale

Unix BSD (Berkeley Software Distribution)

UNIX e Input/Output

In UNIX ogni **sessione aperta** sui file viene mantenuta attraverso un **file descriptor** (fd) che ogni processo vede in modo privato tramite una tabella di kernel (tabella dei file aperti del processo)



paradigma *open-read-write-close*

apertura della sessione

operazioni della sessione (read / write)

chiusura della sessione

Le socket entrano in questo paradigma

La trasparenza tipica delle azioni locali sui file potrebbe diventare un problema nell'uso di connessioni remote e di azioni specifiche

nella comunicazione si devono specificare più parametri per definire un collegamento con connessione:

< **protocollo di trasporto;**

indirizzo locale; processo locale;

indirizzo remoto; processo remoto

>

UNIX primitive comunicazione/sincronizzazione

In UNIX Berkeley

Socket meccanismo

strumenti di *comunicazioni locali o remote*

con **politiche differenziate**

Alternativa anche locale superando i problemi degli strumenti concentrati (**pipe** solo tra *processi in gerarchia*)
ben integrata con processi e trasparente

SOCKET

- processi UNIX possono **scrivere/leggere** messaggi e stream, molte opzioni

Requisiti

- **eterogeneità:** la comunicazione fra processi su architetture diverse
- **trasparenza:** la comunicazione fra processi deve potere avvenire indipendentemente dalla localizzazione fisica
- **efficienza:** l'applicabilità di questi strumenti limitata dalla sola performance
- **compatibilità:** i *naive process* (leggono dallo standard input e scrivono sullo standard output e standard error) devono potere lavorare in ambienti distribuiti senza subire alcuna modifica
- **protocolli di comunicazione** diversi con convenzioni differenziate

Socket Dominio di comunicazione:

semantica di comunicazione +
standard di nomi

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

Semantica di comunicazione include

- affidabilità di una trasmissione
- possibilità di lavorare in *multicast*

Naming è il meccanismo usato per indicare i punti terminali di una comunicazione

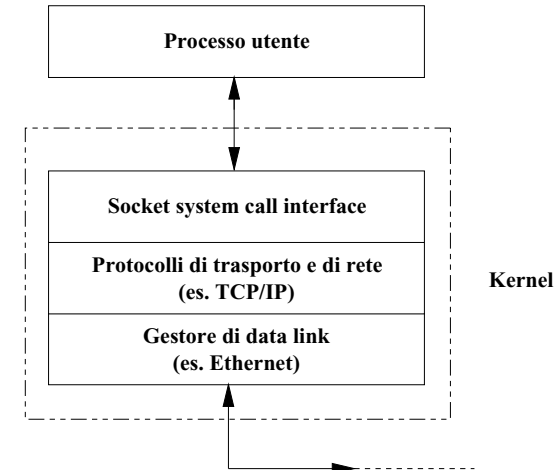
Si sceglie il **dominio** più appropriato mediante un'interfaccia standard

Astrazione unificante **SOCKET**

- comunicazione *in modo omogeneo*
- semantica *uniforme*
 - scelta **connection-oriented** e **connectionless** consistente nei domini

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

TRASPARENZA nella COMUNICAZIONE



Le socket come **interfaccia omogenea** per i servizi

Socket terminale della comunicazione
Socket descriptor integrato con i file descriptor

USO di protocolli di trasporto diversi
anche altri protocolli, diversi da quello che si considera
uso di suite TCP/IP (sia UDP sia TCP)

Chiamata	Significato
<i>open()</i>	Prepara un dispositivo o un file a operazioni di input/output
<i>close()</i>	Termina l'uso di un dispositivo o un file precedentemente aperto
<i>read()</i>	Ottiene i dati da un dispositivo di input o da un file, e li mette nella memoria del programma applicativo
<i>write()</i>	Trasmette i dati dalla memoria applicativa a un dispositivo di output o un file
<i>lseek()</i>	Muove I/O pointer ad una specifica posizione in file /dispositivo
<i>fcntl()</i>	Controlla le proprietà di un file descriptor e le funzioni di accesso
<i>ioctl()</i>	Controlla i dispositivi o il software usato per accedervi

SOCKET

SERVIZI vari tipi di socket

datagram scambio di messaggi senza garanzie
supporto **best effort**

stream (stream virtuale)
scambio messaggi *in ordine nei due sensi,*
senza errori, non duplicati, out-of-band
flusso nessun confine di messaggio

seqpacket (XNS)

raw (Internet debug)

PROTOCOLLI

anche più **domini di comunicazione**

UNIX (AF_UNIX)

Internet (AF_INET)

XEROX (AF_NS)

CCITT (AF_CCITT) X.25

In ogni dominio ci sono **protocolli** diversi

Tabella dei file
aperti del processo

Socket descriptor

dominio
servizio
protocollo
indirizzo locale
porta locale
connessione remota

Flessibilità

Combinazioni possibili fra **dominio** e **tipo** con
indicazione del protocollo

Tipo socket	AF_UNIX	AF_INET	AF_NS
Stream socket	Possibile	TCP	SPP
Datagram socket	Possibile	UDP	IDP
Raw socket	No	ICMP	Possibile
Seq-pack socket	No	No	SPP

Protocolli più probabili nello standard Berkeley

AF_INET	Stream	IPPROTO_TCP	TCP
AF_INET	Datagram	IPPROTO_UDP	UDP
AF_INET	Raw	IPPROTO_ICMP	ICMP
AF_INET	Raw	IPPROTO_RAW	(raw)
AF_NS	Stream	NSRPROTO_SPP	SPP
AF_NS	Seq-pack	NSRPROTO_SPP	SPP
AF_NS	Raw	NSRPROTO_ERROR	Error Protocol
AF_NS	Raw	NSRPROTO_RAW	(raw)
AF_UNIX	Datagram	IPPROTO_UDP	UDP
AF_UNIX	Stream	IPPROTO_TCP	TCP

prefisso AF ==> Address Family

simboli PF_UNIX, PF_INET, PF_NS, PF_CCITT

la sigla PF significa *Protocol Family* e

coincide completamente con la address family

simboli PF_SNA, PF_DECnet e PF_APPLETALK

NAMING PER LE SOCKET

Nomi logici delle socket (VALIDI LOCALMENTE)

un indirizzo di socket dipende dal dominio

Nomi fisici da associare (end-point VALIDI GLOBALMENTE)

una socket deve essere collegata al sistema fisico ==>

è necessario il **binding**, cioè

legare la socket ad una **porta del nodo**

si realizza una entità detta **half-association**

half-association

• dominio Internet:

{famiglia indirizzo, indirizzo Internet, numero di porta}

socket collegata ad un *numero di porta*

• dominio UNIX:

{famiglia indirizzo, path nel filesystem, file associato}

• dominio CCITT:

indirizzamento legato al protocollo di rete X.25

Internet

nodi **{identificatore_rete, identificatore_host}**

numeri di porta all'interno del nodo (port number)

alcuni riservati (*well-known addresses*) per *servizi noti*

FTP è associato il port number 21

In ambito Internet tutti i port number da 1 a 1023

sono riservati al super-user del sistema

UNIX - Variabili per socket address

Nomi collegati alle socket

socket address in due tipi di strutture

sockaddr (indirizzo generico)

sockaddr_in (famiglia AF_INET)

```
struct sockaddr {
```

```
    u_short    sa_family;
```

```
    char       sa_data[14];
```

```
};
```

```
struct sockaddr_in {
```

```
    u_short    sin_family;
```

```
    u_short    sin_port;
```

```
    struct in_addr sin_addr;
```

```
/* char       sin_zero [8]; non usata */
```

```
};
```

```
struct in_addr {u_long s_addr};
```

```
struct sockaddr_in myaddr;
```

sin_family == famiglia di indirizzamento AF_INET

sin_port == port address

da assegnare prima del binding o

prima della richiesta di connessione

il server deve conoscere l'indirizzo della socket

il client può ottenerlo dalla primitiva

sin_addr == una struttura Internet del nodo remoto

Si vedano i file di inclusione corrispondenti

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

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

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

RAPPRESENTAZIONE dei NOMI in C

Si usano strutture dati per i nomi fisici

Le funzioni usano tipicamente un puntatore generico ad una locazione

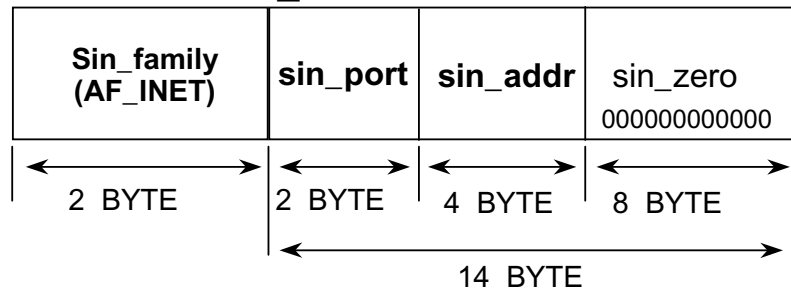
char * in C

void * in ANSI C

struct *sockaddr*



struct *sockaddr_in*



NOME in caso di dominio internet

Indirizzo Internet di un Host remoto

NOMI GLOBALI

corrispondenza tra **nome logico** di dominio del nodo e **nome fisico** Internet di nodo ==>

primitiva *gethostbyname()*

restituisce l'indirizzo Internet e lunghezza

```
#include <netdb.h>
```

```
struct hostent * gethostbyname (name)
```

```
char *name;
```

name dichiarato nel file /etc/hosts

la tabella riporta per ogni nome logico di nodo

indirizzo Internet

(per IP di classe B

2 byte per la rete ed 1 per la sottorete, 1 per host)

aliases

137.204.56.11 didahp1 hp1

137.204.56.12 didahp2 hp2

137.204.56.13 didahp3 hp3

gethostbyname restituisce

un puntatore alla struttura *hostent*

oppure NULL se fallisce

**La ricerca avviene in principio localmente
poi integrata anche con strumenti di supporto
dei sistemi di nome (DNS)**

Struttura di supporto

```
struct hostent {
    char * h_name; /* nome ufficiale dell'host */
    char ** h_aliases; /* lista degli aliases */
    int h_addrtype; /* tipo dell'indirizzo host */
    int h_length; /* lunghezza dell'indirizzo */
    char ** h_addr_list;
/* lista degli indirizzi presa dal server dei nomi host */
#define h_addr h_addr_list[0]
/* indirizzo dell'host in compatibilit  con altre versioni
La struttura si trova in /usr/include/netdb.h per
HP-UX v8.0. */
}
```

Esempio di utilizzo

```
#include <netdb.h>
struct hostent * hp;
struct sockaddr_in peeraddr;
peeraddr.sin_family= AF_INET;
peeraddr.sin_port= 22375;

if (hp=gethostbyname (argv[1]) )
peeraddr.sin_addr.s_addr=
((struct in_addr *) (hp->h_addr))
/* casting */
-> s_addr;
else /* errore o azione alternativa */
```

La ricerca avviene al **nodo cliente** fornendo un **identificativo logico** e ottenendo come **risultato** gli **attributi completi del nodo**

Port address per un servizio

getservbyname() restituisce il numero di porta relativo ad un servizio noto

tabella di corrispondenze fra **servizi e porte**
(e **protocollo**)

file *etc/services*

<i>echo</i>	7/tcp	# Echo
<i>systat</i>	11/tcp users	# Active Users
<i>daytime</i>	13/tcp	# Daytime
<i>daytime</i>	13/udp	#
<i>qotd</i>	17/tcp quote	# Quote of the Day
<i>chargen</i>	19/tcp ttytst source	# Character
<i>chargen</i>	19/udp ttytst source	# Generator
<i>ftp-data</i>	20/tcp	# File Transfer Protocol (Data)
<i>ftp</i>	21/tcp	# File Transfer Protocol (Control)

utilizzata *prima del bind*
se il client non lo compie,
prima della richiesta di connessione

```
#include <netdb.h>
struct servent *getservbyname (name,proto)
char *name,*proto;
(se TCP   l'unico registrato con quel servizio allora 0)
```

Esempio

```
#include <netdb.h>
struct servent *sp;
struct sockaddr_in peeraddr;
sp=getservbyname ("example", "tcp");
peeraddr.sin_port=sp->s_port;
```

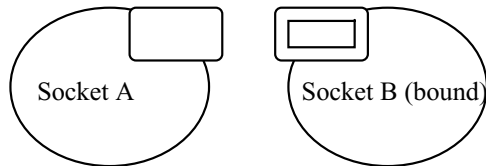
Primitive sulle socket

creazione **socket** in ogni processo

Nome logico LOCALE

s = **socket** (dominio, tipo, protocollo)

```
int s,          /* file descriptor associato alla socket */
    dominio, /* UNIX, Internet, etc. */
    tipo,       /* datagram, stream, etc. */
    protocollo; /* quale protocollo */
```



- il client ha creato la socket

- il server ha creato la socket

- il server ha collegato la socket ad un indirizzo

Client e Server in stato di pre-connessione

aggancio al sistema 'fisico': nodi e porte GLOBALI

rok = **bind** (s, nome, lungnome)

```
int rok, s;
```

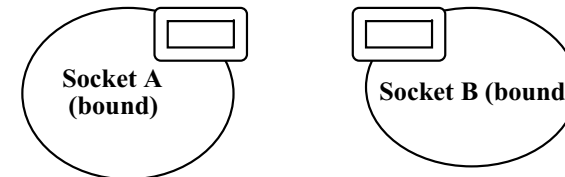
```
struct sockaddr *nome; /* porta locale per la socket */
```

```
int lungnome;
```

Socket datagram

puri end point per la comunicazione

Le **socket** con le **porte** formano **half-association** (interne ad un solo processo)



- il client ha creato la socket

- il server ha creato la socket

- il client ha collegato la socket ad un indirizzo

- il server ha collegato la socket ad un indirizzo

sufficiente per scambio messaggi scambio datagrammi

processo Client

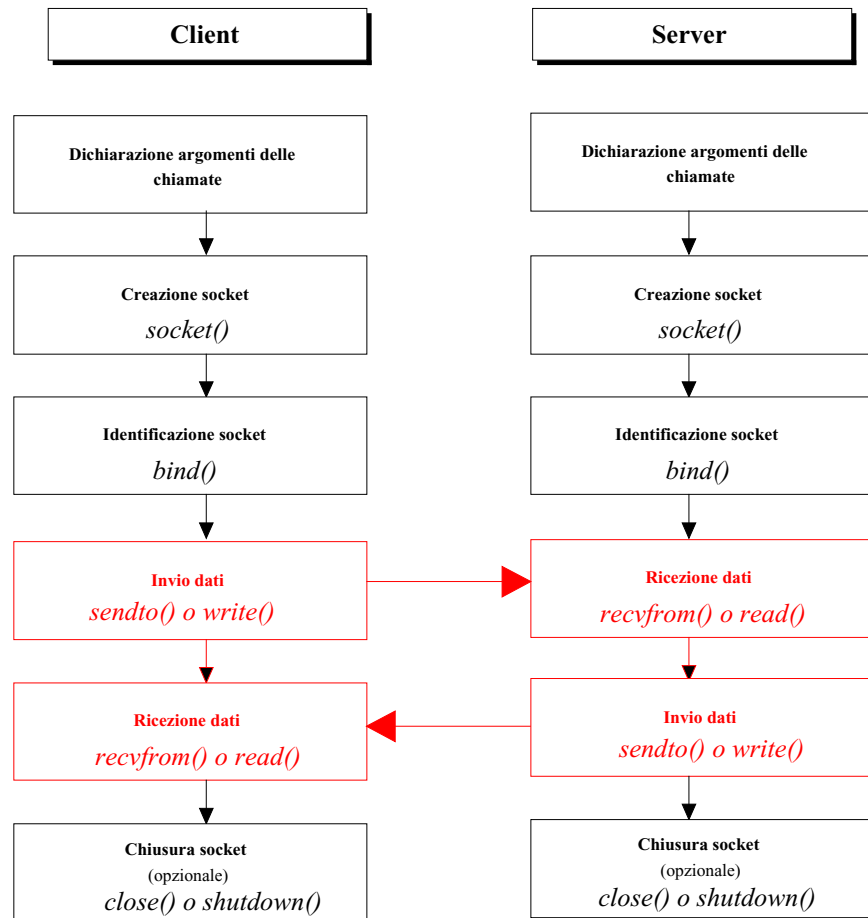
- dichiarazione delle variabili di riferimento a socket
- conoscenza dell'*indirizzo Internet* del nodo remoto
- conoscenza della *porta* del servizio da usare

processo Server

- dichiarazione delle variabili di riferimento a socket
- conoscenza della *porta* per il servizio da offrire
- assegnazione di *wildcard address* per l'indirizzo Internet locale (*server multiporta*)

Si può usare **connect** per specificare il partner di ogni invio successivo (e usare la **send** anzichè la **sendto**)

DATAGRAMMI



protocollo semplificato:
non ci sono fasi di gestione della connessione
(apertura e chiusura) ma endpoint aperti

Creazione di una socket datagramma ed uso

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
s = socket (AF_INET, SOCK_DGRAM, 0);
int s;
```

```
nbytes = sendto (s, msg, len, flags, to, tolen)
int s, nbytes; char *msg;
int len, flags;
struct sockaddr_in *to; int tolen;
```

```
nbytes = recvfrom (s, buf, len, flags, from, fromlen)
int s, nbytes; char *buf;
int len, flags;
struct sockaddr_in *from; int *fromlen;
Informazioni in uscita via *buf, *from, *fromlen
```

s == socket descriptor
buf == puntatore al messaggio da spedire
len == sua lunghezza
flags == flags attualmente non sono implementati
(a parte MSG_PEEK)
to/from == puntatore all'indirizzo della socket partner
tolen/
fromlen == sua lunghezza
nbytes == lunghezza messaggio inviato/ricevuto

Il risultato di ritorno il numero dei byte trasmessi/ricevuti
il flag **MSG_PEEK** non estrae dalla socket il messaggio

Datagrammi

Lunghezza massima del messaggio
(ad esempio 9K byte o 16K byte)

Uso del protocollo UDP e IP
non affidabile intrinsecamente

**A livello utente si può ottenere maggiore affidabilità
prevedendo una ritrasmissione dei messaggi**

recvfrom ritorna solo un messaggio per volta
per prevenire situazioni di perdita di byte di un messaggio,
len più alto possibile

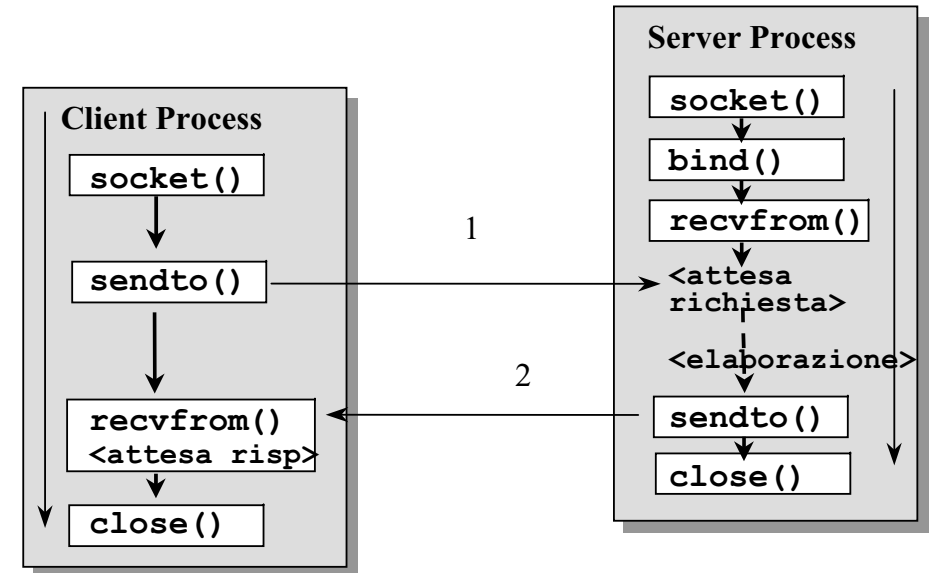
CLIENTE (MITTENTE)

```
struct sockaddr_in *servaddr;  
char msg[2000];  
int count;  
...  
count= sendto (s, msg, strlen(msg), 0,  
servaddr, sizeof(struct sockaddr_in));
```

SERVITORE (RICEVENTE)

```
struct sockaddr_in *clientaddr;  
char buffer[BUFSIZE];  
int count;  
...  
addrlen=sizeof(sockaddr_in);  
count= recvfrom (s, buffer, BUFSIZE,  
0, clientaddr, &addrlen);  
...  
close (s);
```

Protocollo di uso



per Client e Server realizzati con socket UDP, notiamo:

- UDP non è affidabile

in caso di perdita del messaggio del Client o della risposta del Server, il Client si blocca in attesa indefinita della risposta (utilizzo di timeout?)

- Blocco del Client

anche nel caso di invio di una richiesta a un Server non attivo (errori nel collegamento al server notificati solo sulle socket connesse)

- UDP non ha flow control

se il Server riceve troppi datagrammi per le sue capacità di elaborazione, questi vengono scartati, senza nessuna notifica ai Client
una coda per ogni socket modificabile in lunghezza con l'uso di opzione SO_RCVBUF

Esempio di Echo, parte server

Per migliore leggibilità si ipotizzi di avere creato delle semplici funzioni Bind, Socket, Sendto, etc. che incapsulano le rispettive system call bind, socket, sendto, etc. e gestiscano gli errori

```
int sockfd, n;
socklen_t len;
char mesg[MAXLINE];
struct sockaddr_in server_address,
client_address;

sockfd = Socket(AF_INET, SOCK_DGRAM, 0);
bzero(&server_address, sizeof(server_address));
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = INADDR_ANY;
server_address.sin_port = 7;
Bind(sockfd,
      (SA *) &server_address,
      sizeof(server_address));

for (;;)
{ /* ciclo infinito, non termina mai*/
  len = sizeof(client_address);
  n = Recvfrom(sockfd, mesg, MAXLINE, 0,
                &client_address, &len);
  m = Sendto(sockfd, mesg, n, 0,
              &client_address, len);
}
...
```

Socket con connessione STREAM socket per stabilire una CONNESSIONE

PROTOCOLLO DI INIZIALIZZAZIONE

Sincronizzazione CLIENTE/SERVITORE

ruolo **attivo/passivo** in relazione mutua

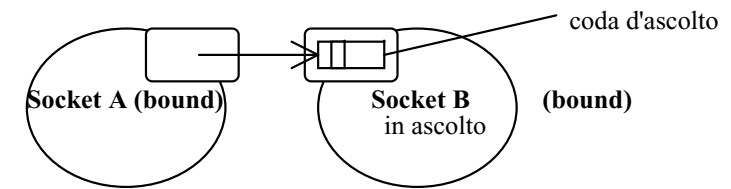
una entità (attiva) richiede la relazione di coppia

una entità (passiva) accetta la relazione

modello **cliente/servitore**

una entità (cliente) richiede il servizio

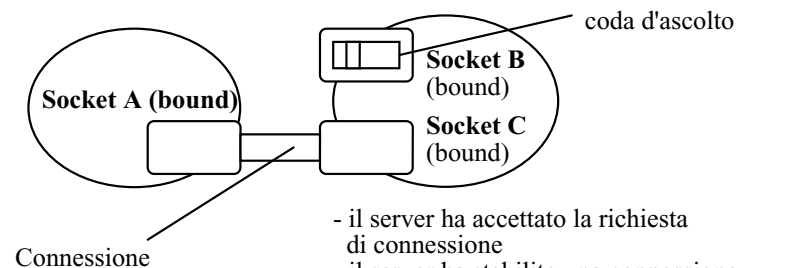
una entità (server) accetta il servizio e risponde



- il client ha fatto una richiesta di connessione

- il server ha ricevuto una richiesta nella coda d'ascolto

Client e Server al momento della richiesta di connessione



- il server ha accettato la richiesta di connessione

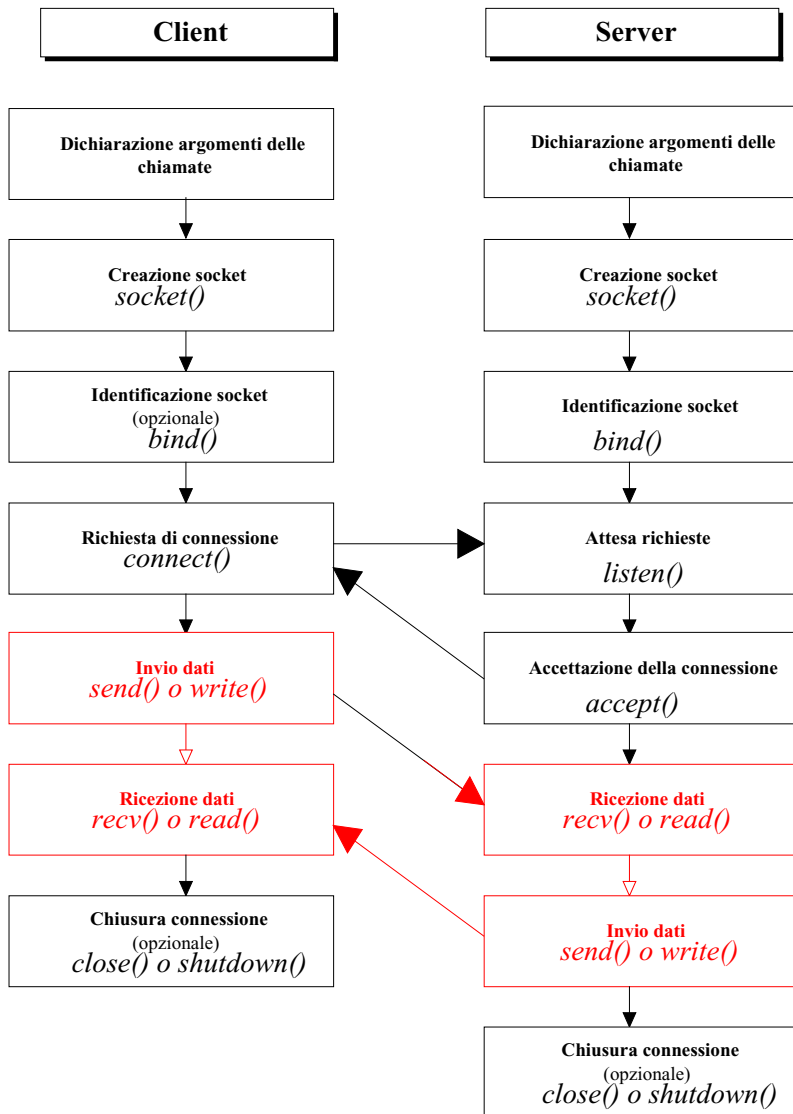
- il server ha stabilito una connessione con il client mediante una nuova socket con le stesse caratteristiche della originale

- la socket originale continua ad ascoltare le eventuali richieste di connessione

Client e Server nello stato di connessione avvenuta

Passi di realizzazione del protocollo

per la connessione



Connessione tra la attiva e la nuova socket

SOCKET CONNESSE

CONNESSIONE permane fino alla chiusura di una delle due half-association

CANALE VIRTUALE

I processi collegati si possono mandare messaggi

(**send, recv**) ma possono fare **read** e **write**

recnum = **read** (s, buff, length);

sentnum = **write** (s, buff, length);

SOCKET usate in modo differenziato

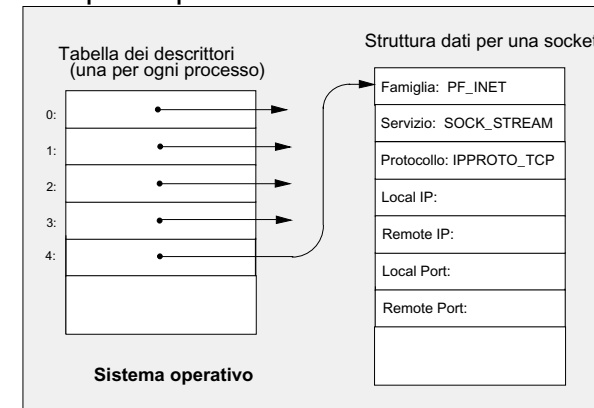
processi naive

Quindi un processo può avere **input/output**

ridiretto su una socket e lavorare in remoto

processi più intelligenti

sfruttano la piena potenzialità



Molte possibilità non di default (avanzate)

come ritrovare i dati **out-of-band**

come realizzare **attese multiple** o **operazioni asincrone**

Wildcard address per trovare indirizzo Internet locale

Indirizzo in socket address locale INADDR_ANY interpretato come qualunque indirizzo valido

In caso di workstation con porte multiple (e più indirizzi Internet)

Il server può accettare le connessioni da ogni indirizzo

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
struct sockaddr_in sin;
sin.sin_addr.s_addr= INADDR_ANY;
<<identificazione della socket>>
```

In questo modo, si possono attendere richieste da qualunque potenziale cliente anche per server con indirizzi di rete multipli

Creazione di una socket

Chiamata *socket()* crea un punto terminale di comunicazione fra processi

{famiglia d'indirizzamento, tipo, protocollo di trasporto}

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (af, type, protocol)
    int af,type,protocol;
```

La primitiva **socket()** restituisce *socket descriptor* o *-1* se la creazione fallisce

protocollo di trasporto anche 0 oppure uno specifico

Uso di *getprotobyname()*

uso del file /usr/include/netinet/in.h

```
int s;
s=socket (AF_INET,SOCK_STREAM,0);
```

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

aggancio della socket in half-association

La chiamata *bind()* collega alla **socket** creata la **porta**

Il server **deve** legare un port address
prima di accettare connessione

Il client può fare il bind o meno

*il sistema assegna un numero di porta fra le libere
il cliente tenta la connessione con la porta del server
Il server risponde ad una richiesta di porta corretta*

Half-association per una socket: **BIND**

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
int bind ( s, addr, addrlen)
    int s;
    struct sockaddr_in *addr;
    int addrlen;
```

s == socket descriptor che identifica la socket

addr == struttura con indirizzo di porta e indirizzo internet del nodo stesso

addrlen == la lunghezza di addr

Lega al nome locale

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

Socket address con sin_port a 0 in alcuni sistemi porta alla assegnazione automatica di un numero di porta libero (come fare nel caso questo non sia possibile?)

```
struct sockaddr_in myaddr;
<<assegnazione del socket address>>
bind (ls,&myaddr,sizeof(struct sockaddr_in));
```

a volte la bind non viene invocata esplicitamente, ma implicitamente dalla primitiva successiva: viene in ogni caso fatta!

CLIENT - Richiesta di connessione

Il processo **client** richiede una connessione con

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <errno.h>
int connect (s, addr, addrlen)
    int s, addrlen;
    struct sockaddr_in *addr;
```

s == socket descriptor

addr == puntatore al socket address remoto

addrlen == lunghezza di questa struttura

connect() sincrona

termina quando la richiesta è accodata o con errore

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

Casi di errore in errno:

- ETIMEDOUT: tentativo di connessione in time-out: la coda d'ascolto del server è piena o non creata
- EISCONN: una socket non può essere connessa più volte per non duplicare association
- ECONNREFUSED: impossibilità di connessione

anche modalità asincrona *nonblocking I/O*

Connect() può anche fare il **bind()** automaticamente

Il **client** al successo della connect() usa la connessione sperando che il tutto sia andato a buon fine (?)

```
struct sockaddr_in *peeraddr;
<< assegnazione del socket address remoto >>
s=socket(AF_INET,SOCK_STREAM,0);
if ( connect(s,&peeraddr,sizeof(struct sockaddr_in)) < 0) errore;
```

SERVER - Attesa di richieste di connessione

Dopo il bind, il server deve preparare una **coda** di accettazione per le **richieste** di connessione dei client in attesa di servizio

primitiva *listen()*

si crea la coda per il socket descriptor
la connessione viene stabilita solo dopo l'accodamento della richiesta di un client e l'azione del server

la socket **deve** essere stata collegata alla porta dal sistema via bind

<<nessun file incluso>>

```
int listen (s,backlog)
```

```
int s,backlog;
```

s == socket descriptor

backlog == intero fra 1 e 20

massimo numero di richieste in coda (consigliato <=5)

Evento *richiesta non accodata*

segnalato **solo** al **processo client** (ETIMEDOUT)

non segnalato al **server**

Variabile globale segnala il tipo di errore dalla chiamata

file /usr/include/errno.h

sistema *errno* ==> valore EINVAL

<< preparazione argomenti >>

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

```
... << identificazione della socket >>
```

```
bind (s,&myaddr,sizeof(struct sockaddr_in));
```

```
listen (s,5);
```

SERVER - Accettazione di una richiesta di connessione

Il **server** accetta le richieste di connessione

primitiva accept() ==>

si crea una nuova socket per la connessione e si fornisce il socket descriptor relativo

La **nuova socket**:

- ha una *semantica di comunicazione come la vecchia*
- ha lo stesso port address della vecchia socket
- è connessa alla socket del client

La primitiva **accept() sincrona**

(a meno di modalità di tipo nonblocking I/O)

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

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

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

```
int accept (ls, addr, addrlen)
```

```
int ls, *addrlen; struct sockaddr_in *addr;
```

si ottengono informazioni tramite **addr* e **addrlen*

ls == socket descriptor

altri parametri passati per indirizzo

addr == indirizzo del socket address connesso

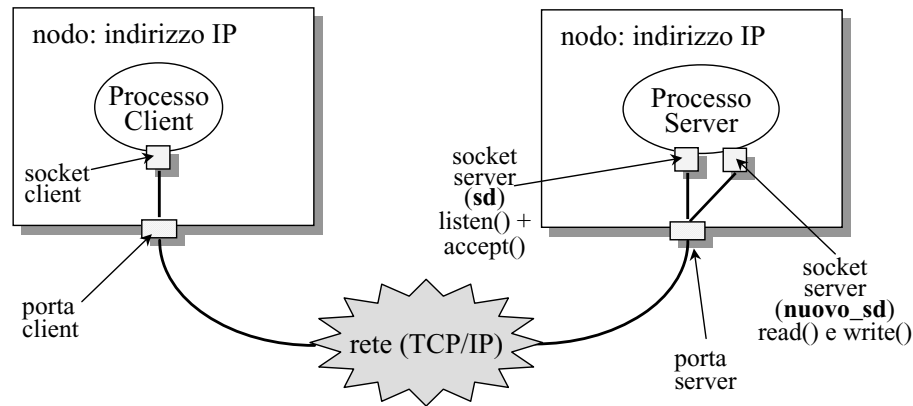
addrlen == la lunghezza espressa in byte

Area in spazio utente per la socket

La accept() non offre la possibilità di filtrare le richieste devono essere accettate tutte oppure nessuna

Sono possibili **errori** (vedi errno)

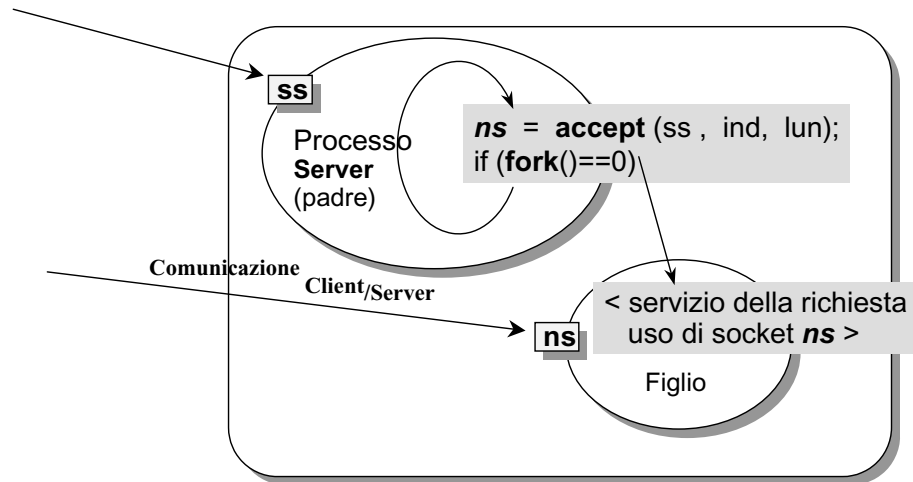
Dopo una accept:
visione complessiva



Per un server concorrente multiprocesso

Server concorrente multi-processo connection-oriented

Richiesta di connessione



Esempio

```
struct sockaddr_in peeraddr;
<< allocazione del socket address ed inizializzazione dell'area
di memoria associata: socket, bind, listen >>
addrlen=sizeof(sockaddr_in);
s=accept(ls, &peeraddr, &addrlen);
```

Invio e Ricezione di dati sulla connessione

Uso dei socket descriptor come file descriptor UNIX
read() / write() e send() / recv()

```
#include <sys/types.h>
#include <sys/socket.h>
int send (s, msg, len, flags)
    int s; char *msg; int len,flags;
```

```
int recv (s, buf, len, flags)
    int s; char *buf; int len,flags;
```

s == socket descriptor
buf /msg == puntatore al messaggio
len == lunghezza del messaggio
flags == opzioni di comunicazione

- **flags** send():
 - 0 per un funzionamento normale,
 - MSG_OOB per un messaggio out-of-band
- **flags** recv():
 - 0 per un funzionamento normale,
 - MSG_OOB per un messaggio out-of-band
 - MSG_PEEK per una lettura non distruttiva dallo stream

Il risultato della primitive rappresenta il numero di byte rispettivamente inviati o ricevuti (-1 se fallimento)

chiamate sincrone

(a meno delle modalità nonblocking I/O)

```
char buf[30]="Messaggio di prova";
int count;
count=send(s,buf,30,0);
...
count=recv(s,buf,5,0);
```

I messaggi sono comunicati ad ogni primitiva?

⇒ i dati sono bufferizzati dal protocollo di trasporto TCP: non è detto che siano inviati subito ma raggruppati solo alla prima comunicazione 'vera' possiamo ottenere indicazioni sulla reale trasmissione

Soluzione ⇒ messaggi di *lunghezza pari al buffer* o si comanda il flush del buffer

Come preservare i messaggi in ricezione?

⇒ ogni receive restituisce i dati pronti: *stream mode* TCP non implementa marcatori di fine messaggio

Soluzioni ⇒ **messaggi a lunghezza fissa**
messaggi a lunghezza variabile

si alternano un messaggio a lunghezza fissa e uno vero e proprio variabile in lunghezza

il primo contiene la lunghezza del secondo

messaggi con indicatore di lunghezza

il primo byte contiene la lunghezza del messaggio e si legge questo per la receive successiva

Chiusura di una socket di Connessione

Primitiva *close()*
int **close** (s)
int s;

La chiamata *close()* decrementa il contatore dei processi referenti al socket descriptor e *restituisce il controllo subito* il chiamante non lavora più con quel descrittore *dopo un intervallo di tempo controllato da un'opzione avanzata (SO_LINGER)*

```
int sd;
sd=socket(AF_INET,SOCK_STREAM,0);
...
close(sd);
```

C'è un ritardo tra la chiusura applicativa e la reale deallocazione della memoria di supporto

Alla chiusura, ogni messaggio nel buffer è spedito, mentre ogni dato non ricevuto viene perduto, poi si dealloca la memoria

La durata di questa fase è poco controllata con la *close*

Dopo la *close*, se si legge dalla socket, si ottiene *finfile*, se si scrive si ottiene un segnale di connessione non più esistente

shutdown() più elegante per abbattimento della connessione per una migliore gestione delle risorse

Le operazioni di chiusura sono garanzie di collegamento con il pari sul canale

Shutdown

Per chiudere una connessione in uno solo dei due versi
la trasmissione passa da full-duplex ad half-duplex

per tutti i processi che usano lo stesso socket descriptor

<<primitiva locale: nessun inclusione>>

int **shutdown** (s,how)

int s, how;

s == socket descriptor della socket

how == direzione

how=0, SHUT_RD

non si **ricevono** più dati, ma si può trasmettere
un send() del pari ritorna con -1 ed il processo riceve SIGPIPE
*(importante in caso **non bloccante**)*

how=1, SHUT_WR

si possono ricevere i dati dalla socket
senza **ulteriori trasmissioni** di dati
abbattimento graceful della connessione
Il processo collegato alla connessione alla lettura riceve end-of-file

how=2, SHUT_RDWR

entrambi gli effetti

Esempio di chiusura di un canale di trasmissione di una socket ed invia *end-of-file* al processo remoto

```
int s; ... /* socket descriptor aperto su stream*/  
shutdown (s,1); /* chiusura dello stream di output: fine file */
```

Considerando il protocollo sottostante In TCP

CLOSE

la **close** (fd) segnala la immediata intenzione di non fare più operazioni sullo stream ⇒ primitiva immediata

tutti i dati dell'altra direzione in attesa non vengono ricevuti e il ricevente ottiene una fine del file ⇒ **deve fare una close**

I segmenti non ancora trasmessi dalla parte di chi chiude possono essere buttati via ma sono inviati i dati in uscita
I segmenti non ancora trasmessi dalla parte del pari devono essere buttati via e non trasmessi

vedi **demon** che ha creato la connessione e passa al server il socket descriptor
il demon deve chiudere la socket

SHUTDOWN

la **shutdown** (fd, 1) segnala la intenzione di non fare **più invii** sullo stream ⇒

il ricevente è avvertito con una fine del file ⇒

ricevuta anche dopo molte ricezioni
può fare anche molti invii
decide quando fare la shutdown dal suo verso

si attua una chiusura dolce e i segmenti non ancora trasmessi dal pari possono essere inviati ed effettivamente ricevuti
In Java esistono **shutdownInput()** e **shutdownOutput()**

Trasparenza di comunicazioni fra processi

Uso di write() e read() al posto delle send() e recv()

```
#include <unistd.h>
ssize_t write (fileds,buf,nbytes)
ssize_t read (fileds,buf,nbytes)
```

```
int fileds;
char *buf;
size_t nbytes;
```

trasparenza della comunicazione

Prima **close()** dei file descriptors 0,1,2

Uso di **fork()** di un comando locale
dopo aver creato tre socket
con socket descriptor *stdin*, *stdout* e *stderr*

Poi **exec()**
Il processo locale naif è così
connesso ad un processo remoto

Funzioni ausiliarie

obiettivo portabilità

Manipolazione interi

Quattro funzioni di libreria per convertire da formato di rete in formato interno per interi (lunghi o corti)

```
/* trasforma un intero da formato esterno in interno */
shortlocale = ntohs(shortrete);
longlocale = ntohl(longrete);
```

```
/* trasforma un intero da formato interno in esterno */
shortrete = htons(shortlocale);
longrete = htonl(longlocale);
```

Manipolazione indirizzi IP

Conversione tra indirizzi IP a 32 bit e corrispondente notazione col punto decimale (ascii: "123.34.56.78")

inet_addr() converte l'indirizzo dalla forma con punto decimale
indirizzo = **inet_addr**(stringa);
stringa contiene l'indirizzo in formato punto decimale
risultato *indirizzo* IP a 32 bit

inet_ntoa() esegue la funzione inversa
stringa = **inet_ntoa**(indirizzo);
indirizzo indirizzo IP a 32 bit (cioè un *long integer*)
risultato *stringa* di caratteri con indirizzo in forma con punto

FUNZIONI che non richiedono fine stringa (ma assumono solo blocchi di byte senza terminatore)

bcmp (addr1, addr2, length)	memset (addr1, char, length)
bcopy (addr1, addr2, length)	memcpy (addr1, addr2, length)
bzero (addr1, length)	memcmp (addr1, addr2, length)

Trasformazione dei dati (usando le socket)

Gli interi sono composti da più byte

2 byte per gli interi a 16 bit, 4 byte per interi a 32 bit e possono essere rappresentati in memoria secondo **due modalità diverse** di ordinamento.

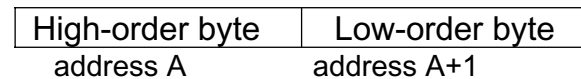
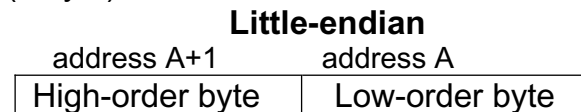
Little-endian Byte Order

byte di ordine più basso nell'indirizzo iniziale

Big-endian Byte Order

byte di ordine più alto nell'indirizzo iniziale

Es. 16 bit (2 byte)



Big-endian

Network Byte Order (NBO)

ordinamento di byte usata da un protocollo di rete

Protocolli Internet Big-endian

Host Byte Order (HBO)

non un unico ordinamento

Es. Intel Little-endian, Solaris Big-endian

Funzioni accessorie da usare con le socket

htons() e **htonl()** conversione da HBO a NBO valori (per **word short** 16 bit / e **double word long** 32 bit)

ntohs() e **ntohl()** convertono valori da NBO a HBO

Principali chiamate dell'interfaccia socket

di kernel dal BSD 4.2

Oltre alle funzioni viste sopra,

(si sperimenti la *gethostbyname* per il nodo locale)

si vedano le azioni

Chiamata	Significato
<i>socket()</i>	Crea un descrittore da usare nelle comunicazione di rete
<i>connect()</i>	Connette la socket a una remota
<i>write()</i>	Spedisce i dati attraverso la connessione
<i>read()</i>	Riceve i dati dalla connessione
<i>close()</i>	Termina la comunicazione e dealloca la socket
<i>bind()</i>	Lega la socket con l'endpoint locale
<i>listen()</i>	Socket in modo passivo e predispone la lunghezza della coda per le connessioni
<i>accept()</i>	Accetta le connessioni in arrivo
<i>recv()</i>	Riceve i dati in arrivo dalla connessione
<i>recvmsg()</i>	Riceve i messaggi in arrivo dalla connessione
<i>recvfrom()</i>	Riceve i datagrammi in arrivo da una destinazione specificata
<i>send()</i>	Spedisce i dati attraverso la connessione
<i>sendmsg()</i>	Spedisce messaggi attraverso la connessione
<i>sendto()</i>	Spedisce i datagrammi verso una destinazione specificata
<i>shutdown()</i>	Termina una connessione TCP in una o in entrambe le direzioni
<i>getsockname()</i>	Permette di ottenere la socket locale legata dal kernel (vedi parametri <i>socket, sockaddr, length</i>)
<i>getpeername()</i>	Permette di ottenere l'indirizzo del pari remoto una volta stabilita la connessione (vedi parametri <i>socket, sockaddr, length</i>)
<i>getsockopt()</i>	Ottiene le opzioni settate per la socket
<i>setsockopt()</i>	Cambia le opzioni per una socket
<i>errord()</i>	Invia un messaggio di errore in base a <i>errno</i> (stringa su <i>stderr</i>)
<i>syslog()</i>	Invia un messaggio di errore sul file di log (vedi parametri <i>priority, message, params</i>)

PRIMITIVE SINCRONE

Quali primitive possono avere una **elevata durata?** **bold**

Si sperimentino le primitive non note (e note)

Esempio

In un'applicazione distribuita Client/Server per una rete di workstation UNIX (BSD oppure System V) il Client presenta l'interfaccia:

rcp nodoserver nomefile

dove ***nodoserver*** specifica l'indirizzo logico del nodo contenente il processo Server e ***nomefile*** è il nome assoluto di un file presente nel file system della macchina Client

Il processo Client deve inviare il file ***nomefile*** al Server

Il processo Server deve copiare il file ***nomefile*** nel direttorio /ricevuti (si supponga di avere tutti i diritti necessari per eseguire tale operazione)

La scrittura del file nel direttorio specificato deve essere eseguita **solo se** in tale direttorio non è presente un file di nome ***nomefile***, per evitare di sovrascriverlo

Il Server risponde al Client il carattere 'S' per indicare che il file non è presente, il carattere 'N' altrimenti

Si supponga inoltre che il Server sia già legato alla porta 12345

Esercizio: rcp - lato Client

```
...
server_address.sin_family = AF_INET;
/* Prepara indirizzo remoto connessione */
host = gethostbyname(argv[1]);
if (host == NULL)
{printf("%s non trovato", argv[1]);
exit(2);} /* if ok, big endian */
server_address.sin_addr.s_addr=
    ((struct in_addr *)
     (host->h_addr))->s_addr;
server_address.sin_port = htons(12345);
sd=socket(AF_INET, SOCK_STREAM, 0);
if(connect(sd, (struct sockaddr *)
    &server_address,
    sizeof(struct sockaddr)<0)
{perror("Errore in connect"); exit(1);}
if (write(sd, argv[2],
    strlen(argv[2])+1)<0)
{ perror("write"); exit(1);}
if ((nread=read(sd, buff, DIM_BUFF))<0)
{ perror("read"); exit(1);}
if(buff[0]=='S')
{if((fd=open(argv[2], O_RDONLY))<0)
{perror("open"); exit(1);}
while((nread=read(fd, buff, DIM_BUFF))>0)
    write(sd,buff,nread);
close(sd);
/* ho spedito il file */
printf("File spedito\n");
}
/* non si sovrascrive il file */
else
{printf("File esiste, termino\n");
close(sd);
}
...

```

Esercizio: rcp - lato Server

Il server deve conoscere la propria porta (che deve essere nota ai clienti) e il proprio IP (noto ai clienti)

Si deve fare la bind

```
...
sd=socket(AF_INET, SOCK_STREAM, 0);
if(sd<0) {perror("apertura socket");
exit(1);}

mio_indirizzo.sin_family=AF_INET;
mio_indirizzo.sin_port=12345;

if(bind(sd, (struct sockaddr*)&mio_indirizzo,
sizeof(struct sockaddr_in))<0)
{perror("bind"); exit(1); }

listen(sd,5);
/*trasforma in socket passiva
d'ascolto*/

chdir("/ricevuti");
...
```

A questo punto sono possibili progetti di
server sequenziali o
server concorrenti

Server sequenziale o iterativo

Un servizio alla volta

```
...
for(;;)
{
ns=accept(sd, (struct sockaddr *)
&client_address, &fromlen);
read(ns, buff, DIM_BUFF);
printf("il server ha letto %s \n",
buff);

if((fd=open
(buff, O_WRONLY|O_CREAT|O_EXCL))<0)
{ printf("file esiste, non opero\n");
write(ns, "N", 1);
}
else
{
printf("file non esiste, copia\n");
write(ns, "S", 1);
while((nread=read(ns, buff,
DIM_BUFF))>0)
{
write(fd, buff, nread);
cont+=nread;
}
printf("Copia eseguita di %d byte\n",
cont);
}
close(ns);
} exit(0);
...
```

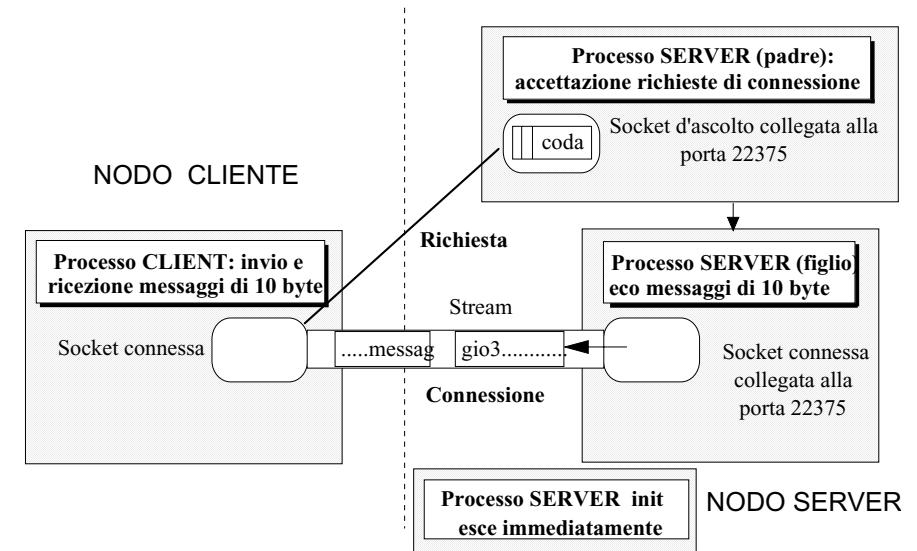
Server concorrente multiprocesso

```
for(;;) {
    ns=accept(sd,(struct sockaddr *)
        &client_address,&fromlen);
    if (fork()==0)
        /* figlio */
        close(sd);
    read(ns, buff, DIM_BUFF);
    printf("il server ha letto %s \n",
        buff);
    if((fd=open(buff,O_WRONLY|O_CREAT|O_EXCL))
        <0)
    {
        printf("file esiste, non opero\n");
        write(ns,"N", 1);
    } else
    {
        printf("file non esiste, copia\n");
        write(ns,"S", 1);
        while((nread=read(ns, buff,
            DIM_BUFF))>0)
            { write(fd,buff,nread); cont+=nread;}

        printf("Copia eseguita di %d byte\n",
            cont);
    }
    close(ns); exit(0);
}

/* padre */
close(ns);
wait(&status);
/* attenzione: sequenzializza ...
   Cosa fare? */
}
```

Esempio con CONNESSIONE e PARALLELISMO



```
/* CLIENT */
```

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

```
char *ctime(); /* routine di formattazione dell'orario */
int ricevi () /* routine di ricezione di un messaggio */
```

```
int s; /* socket descriptor */
struct hostent *hp; /* puntatore alle informazioni host remoto */
long timevar; /* contiene il ritorno dalla time() */
struct sockaddr_in myaddr_in; /* socket address locale */
struct sockaddr_in peeraddr_in; /* socket address peer */
```

```

main(argc, argv)
int argc; char *argv[];
{ int addrlen, i;
  char buf[10]; /* L'esempio scambia messaggi da 10 bytes */

if (argc != 3) {
  fprintf(stderr, "Uso: %s <host remoto> <nrice>\n", argv[0]);
  exit(1); }

/* azzera le strutture degli indirizzi */
memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));

/* assegnazioni per il peer address da connettere */
peeraddr_in.sin_family = AF_INET;
/* richiede informazioni a proposito del nome dell'host
   passato alla chiamata */
hp = gethostbyname (argv[1]);
/* il risultato è già big endian e pronto per la TX */
if (hp == NULL) {
  fprintf(stderr, "%s: %s non trovato in /etc/hosts\n",
          argv[0], argv[1]);
  exit(1); }

peeraddr_in.sin_addr.s_addr =
  ((struct in_addr *) (hp->h_addr))->s_addr;
/* non usiamo la htonl per problemi di implementazione
   si provi il tutto in diversi ambienti operativi */

```

```

/* definisce il numero di porta senza la chiamata
getservbyname(): non registrato in /etc/services
   Se fosse registrato nel nodo cliente avremmo:
   struct servent *sp;
/* puntatore alle informazioni del servizio */

sp = getservbyname ("example", "tcp");
if (sp == NULL) {
  fprintf(stderr, "%s: non trovato in /etc/services\n", argv[0]);
  exit(1); }
peeraddr_in.sin_port = htons (sp->s_port);
*/

/* il numero di porta deve essere trasformato */
peeraddr_in.sin_port = htons(22375);

/* creazione della socket */
s = socket (AF_INET, SOCK_STREAM, 0);
if (s == -1) {
  perror(argv[0]);
  fprintf(stderr, "%s: non posso creare la socket\n", argv[0]);
  exit(1); }

/* No bind(): il numero di porta del client assegnato dal sistema.
   Il server lo conosce alla richiesta di connessione; il
   processo client lo ricava con la primitiva getsocketname()
   Modificare nel caso non si abbia questa feature */
/* tentativo di connessione al server remoto */
if (connect (s, &peeraddr_in, sizeof(struct sockaddr_in)) == -1)
{ perror(argv[0]);
  fprintf(stderr, "%s: impossibile connettersi con il server \n",
          argv[0]);
  exit(1);
}
/* lo stream è stato ottenuto in modo completo (!?) */

```



```

addrrlen = sizeof(struct sockaddr_in);
if ( getsockname(s, &myaddr_in, &addrrlen) == -1)
    { perror(argv[0]);
      fprintf(stderr, "%s: impossibile leggere il socket address\n",
              argv[0]);
      exit(1); }

```

/ scrive un messaggio iniziale per l'utente */*

```

time(&timevar);
printf("Connessione a %s sulla porta %u alle %s",
       argv[1], ntohs(myaddr_in.sin_port), ctime(&timevar));

```

/ Il numero di porta espresso in byte in formato interno: conversione nel formato host mediante **ntohs()**. Per alcuni host i due formati coincidono (sistemi HP-UX), maggiore portabilità */*

```

sleep(5); /* attesa che simula un'elaborazione */

```

/ NON C'È PRESENTAZIONE DEI DATI*

*Invio di **messaggi** al processo server remoto*

Si mandano un insieme di interi successivi

****buf=i** pone i primi 4 byte del vettore buf uguali alla codifica dell'intero in memoria*

Il server rispedisce gli stessi messaggi al client (senza usarli)

Aumentando il numero e la dimensione dei messaggi, potremmo anche occupare troppa memoria dei gestori di trasporto =>

*sia il server che il client stabiliscono un limite alla memoria associata alla coda delle socket */*

/ invio di tutti i messaggi */*

```

for (i=1; i<=atoi(argv[2]); i++) {
    *buf = htonl( i );
    if ( send(s, buf, 10, 0) != 10)
        { fprintf(stderr, "%s: Connessione terminata per errore",
                  argv[0]);
          fprintf(stderr, "sul messaggio n. %d\n", i);
          exit(1);
        }
}

```

*/***Shutdown()** della connessione per successivi invii (modo 1): Il server riceve un end-of-file dopo le richieste e riconosce che non vi saranno altri invii di messaggi */*

```

if ( shutdown(s, 1) == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: Impossibile eseguire lo shutdown\
della socket\n", argv[0]);
    exit(1); }

```

*/*Ricezione delle risposte dal server*

Il loop termina quando la recv() fornisce zero, cioè la terminazione end-of-file. Il server la provoca quando chiude il suo lato della connessione/*

```

while ( ricevi(s, buf, 10) )
    printf("Ricevuta la risposta n. %d\n", ntohl( *buf) );
/* Per ogni messaggio ricevuto, diamo un'indicazione locale */

```

/ Messaggio per indicare il completamento del programma. */*

```

time(&timevar);
printf("Terminato alle %s", ctime(&timevar));
}

```

```

int ricevi (s, buf, n)
    int s; char * buf; int n;
{ /* ricezione di un messaggio di specificata lunghezza */
int i, j;
/* Il ciclo interno verifica che la recv() non ritorni un
messaggio più corto di quello atteso (n byte)
recv ritorna appena vi sono dati e non attende tutti i
dati richiesti
Il loop interno di recv garantisce la ricezione fino al byte
richiest e permette alla recv successiva di partire sempre
dall'inizio di una risposta

if ((i=recv (s, buf, n, 0)) != n )
{   if (i == -1) {
        perror(argv[0]);
        fprintf(stderr, "%s: errore nella lettura delle risposte\n",
                argv[0]);
        exit(1); }
    while (i < n) {
        j = recv (s, &buf[i], n-i, 0);
        if (j == -1) {
            perror(argv[0]);
            fprintf(stderr, "%s: errore nella lettura delle risposte\n",
                    argv[0]);
            exit(1); }
        i += j; }
    }
return n; }
/* Per messaggi di piccola dimensione la frammentazione
è improbabile, ma con dimensioni superiori (qualche
Kbyte) il pacchetto può venire suddiviso dai livelli
sottostanti, e una ricezione parziale diventa più probabile.
In ogni caso, in ricezione dobbiamo sempre aspettare
l'intero messaggio, se questo è significativo */

```

/* SERVER con parallelismo*/

```

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
long timevar; /* contiene il valore fornito da time() */
int s; /* socket descriptor */
int ls; /* socket per la listen()*/
struct hostent *hp; /* puntatore all' host remoto */
struct sockaddr_in myaddr_in; /* per il socket address locale */
struct sockaddr_in peeraddr_in; /* per il socket address peer */

main(argc, argv)
int argc; char *argv[];
{   int addrlen;

/* Azzera le strutture degli indirizzi */
memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));

/* Assegna la struttura d'indirizzo per la listen socket. */
myaddr_in.sin_family = AF_INET;
/*Il server ascolta su un qualunque suo indirizzo
(wildcard address), invece che sul suo indirizzo di
rete.
Convenzione per server connessi a più reti: consente di
attendere richieste su ogni rete
==> maggiore portabilità del codice */

/* assegna IP generico e numero di porta */
myaddr_in.sin_addr.s_addr = INADDR_ANY;
myaddr_in.sin_port = 22375;
/* Crea la socket d'ascolto */

```

```

ls = socket (AF_INET, SOCK_STREAM, 0);
if (ls == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile creare la socket.\n", argv[0]);
    exit(1); }

/* Collega la socket all'indirizzo fissato */
if ( bind (ls, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile eseguire il collegamento.\n",
        argv[0]);
    exit(1); }

/* Inizializza la coda d'ascolto richieste (massimo 5 pendenti)*/
if ( listen (ls, 5) == -1) {
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile l'ascolto sulla socket\n",
        argv[0]);
    exit(1); }

```

/*Inizializzazione completata.

Il programma crea un processo daemon. La chiamata setsid() sgancia il processo dal terminale di controllo e lo stacca dal gruppo del processo padre (il processo diventa leader di una nuova sessione non collegata a nessun terminale) */

```

setsid();          /* Unix System V */
switch (fork()) {
case -1:/* Impossibilità di creazione di un processo figlio. */
    perror(argv[0]);
    fprintf(stderr, "%s: impossibile creare un daemon.\n", argv[0]);
    exit(1);

```

```

case 0: /* FIGLIO e schema di processo DEMONE */
/*Il demone entra in un loop di attesa e, ad ogni richiesta, crea un processo figlio per servire la chiamata. Il daemon chiude lo stdin e lo stderr, mentre lo stdout è assunto come ridiretto ad un file di log per registrare gli eventi di esecuzione*/
    close(stdin);
    close(stderr);

/*Il segnale SIGCLD è ignorato (SIG_IGN) per evitare di mantenere processi zombi per ogni servizio eseguito
Il daemon non deve occuparsi degli zombies */
signal(SIGCLD, SIG_IGN);
for (;) {
    addrlen = sizeof(struct sockaddr_in);

/*accept() bloccante in attesa di richieste di connessione
Completa fornendo l'indirizzo del chiamante e la sua lunghezza, oltre che un socket descriptor per la connessione instaurata */
    s = accept (ls, &peeraddr_in, &addrlen);
    if ( s == -1) exit(1);

    switch (fork()) {
    case -1:
/* Non è possibile generare un figlio ed allora esce */
        exit(1);
    case 0:
/* Esecuzione del processo figlio che gestisce il servizio */
        server(); /* ulteriore figlio per il servizio */
        exit(0);

```

```

default:
/* Il processo daemon chiude il socket descriptor e
torna ad accettare ulteriori richieste. Questa
operazione consente al daemon di non superare il
massimo dei file descriptor ed al processo figlio fare una
close() effettiva sui file */
    close(s);      } /* for */
} /* demone */

```

```

default: /* processo INIZIALE */
/* Il processo iniziale esce per lasciare libera la console */
    exit(0); }
}

```

```

char *inet_ntoa(); /* routine formattazione indirizzo Internet */
char *ctime(); /* routine di formattazione dell'orario ottenuto
da una time () */

```

```
int ricevi ();
```

```

/* procedura SERVER: routine eseguita dal processo
figlio che il daemon crea per gestire il servizio
Si ricevono alcuni pacchetti dal processo client, si
elaborano e si ritornano i risultati al mittente; inoltre
alcuni dati sono scritti sullo stdout */

```

```

server()
{ int reqcnt = 0; /* conta il numero di messaggi */
char buf[10]; /* l'esempio usa messaggi di 10 bytes */
char *hostname; /* nome dell'host richiedente */
int len, len1;

```

```

/* Chiude la socket d'ascolto ereditata dal processo daemon */
close (ls);

```

```

/* Cerca le informazioni relative all'host connesso
mediante il suo indirizzo Internet
Primitiva gethostbyaddr() usando indirizzi corretti */
hp = gethostbyaddr ((char *) & ( ntohl( peeraddr_in.sin_addr ),
sizeof (struct in_addr), peeraddr_in.sin_family);

```

```

if (hp == NULL) hostname = inet_ntoa (peeraddr_in.sin_addr);
/* Non trova host ed allora assegna l'indirizzo formato Internet */
else
{ hostname = hp->h_name; /* punta al nome dell'host */ }
/* stampa un messaggio d'avvio */
time (&timevar);
printf("Inizio dal nodo %s porta %u alle %s",
hostname, ntohs(peeraddr_in.sin_port), ctime(&timevar));

```

```

/* Loop di ricezione messaggi del cliente
Uscita alla ricezione dell'evento di shutdown del
processo client, cioè all'evento fine del file
ricevuto dal server

```

```

Si possono anche assegnare opzioni avanzate alla socket,
come, ad esempio, la chiusura ritardata per offrire più
sicurezza ai dati (SO_LINGER) */

```

```

while (ricevi (s, buf, 10) )
{ reqcnt++; /* Incrementa il contatore di messaggi */
sleep(1); /* Attesa per simulare l'elaborazione dei dati */
/* Invio della risposta al cliente per ogni messaggio */
if ( send (s, buf, 10, 0) != 10)
{ printf("Connessione a %s abortita sulla send\n", hostname);
exit(1); }
/* sui dati mandati e ricevuti non facciamo nessuna
trasformazione */
}

```

*/*Il loop termina se non vi sono più richieste da servire
La close() può avere un effetto ritardato per attendere la
spedizione di tutte le risposte al processo client.*

*Questa modalità avanzata può permettere al processo
server di rilevare l'istante di cattura dell'ultima risposta
e stabilire così la durata della connessione */*

```
close (s);  
/* Stampa un messaggio di fine. */  
time (&timevar);  
printf("Terminato %s porta %u, con %d messaggi, alle %s\n",  
       hostname, ntohs(peeraddr_in.sin_port),  
       reqcnt, ctime(&timevar));  
}
```

Si modifichi il processo server per aggiungere indicazioni di tempo nei messaggi *dal servitore al cliente e viceversa* per temporizzare la durata delle operazioni

Primitive sospensive

Una primitiva sospensiva interrotta da un segnale deve essere riattivata dall'inizio

vedi schema seguente (errno == **EINTR** verifica tale evento)

```
for (;)  
{ int g, len = sizeof (from);  
  g=accept (f, (struct sockaddr *)&from, &len);  
  if (g < 0) { if (errno != EINTR) syslog(LOG_ERR, ..."p);  
              continue; } /* necessità di ripetizione primitiva */  
  ... /* altro codice in caso di successo */  
}
```

Ricezione eventi multipli (server/client) attesa contemporanea di eventi di I/O su più socket descriptor (o file)

legata alla gestione di **più eventi** all'interno di uno stesso processo e al blocco imposto dalle primitive sincrone

A causa dei problemi causati da primitive sincrone

Le operazioni bloccanti (lettura) o con attese pregiudicano un completamento di altre un processo server (con molti servizi da svolgere) potrebbe sospendersi su una primitiva e non potere servire altre richieste che arrivano su socket diverse

Risposta in C con PRIMITIVA select select()

blocca il processo in attesa di **almeno un** evento fra **più eventi attesi possibili** (range da **0** a **soglia** intera) **anche solo per un definito intervallo massimo**

Gli eventi sono legati alle operazioni che il processo potrebbe fare (e di cui si vole evitare la sospensione).

attesa multipla globale sincrona o con durata massima (time-out) bloccante o meno

Primitive di comunicazione potenzialmente sospensive

Primitive di lettura

accept, receive (in tutte le forme), eventi di chiusura

Primitive di scrittura

connect, send (in tutte le forme), eventi di chiusura

Primitive con eventi anomali

Dati out-of-band, eventi di chiusura

```
#include <time.h>
int select (nfd, readfd, writefd, exceptfd, timeout)
size_t nfd; /* numero massimo di eventi attesi
           è il limite superiore e dice quali bit considerare */
int * readfd, * writefd, * exceptfd; /* maschere */
const struct timeval * timeout;
```

condizioni del tipo

- **evento di lettura: operazione possibile**
 - . in una socket sono presenti dati da leggere *recv()*
 - . in una socket passiva c'è una richiesta (OK *accept()*)
 - . in una socket connessa si è verificato un end of file
 - . in una socket si è verificato un errore
- **evento di scrittura: operazione completata**
 - . in una socket la connessione è completata *connect()*
 - . in una socket si possono spedire altri dati con *send()*
 - . in una socket connessa il pari ha chiuso (SIGPIPE)
 - . in una socket si è verificato un errore
- **condizioni eccezionali: errore o urgenza**
 - . arrivo di dati *out-of-band*,
 - . inutilizzabilità della socket, *close()* o *shutdown()*

azione bloccante fino al timeout

Se *timeout* *NULL* (0) ⇒
 attesa indefinita di un evento (**sincrona** con il primo)
 se punta ad una struttura con un **tempo** ⇒
 intervallo di **time-out**
 alla scadenza, restituisce 0 e cancella le maschere
Massimo intervallo di attesa 31 giorni

azione non bloccante

polling dei canali se *timeout* a struttura con valore 0

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

Maschere di bit

La chiamata esamina gli **eventi** per i file descriptor **specificati** nelle maschere (valore ingresso bit ad 1)

9	8	7	6	5	4	3	2	1	0	<i>posizione file descriptor</i>
0	0	1	0	1	1	0	0	0	0	maschera ingresso
0	0	1	0	1	0	0	0	0	0	maschera uscita

I bit della maschera corrispondono ai file descriptor

si esaminano solo i file descriptor il cui bit è ad 1
`fds[(f / BITS_PER_INT)] & (1 << (f % BITS_PER_INT))`

Al ritorno della chiamata le tre maschere sono modificate
 in relazione agli eventi per i corrispondenti file descriptor
 1 se evento verificato, 0 altrimenti

Eventi di lettura/scrittura/anomali

terminano la primitiva *select*

si possono *o trattare tutti*
 o uno solo

si può selezionare un ordine del servizio

- la **scrittura successiva** potrebbe portare ad azioni di ritardo per questioni di bufferizzazione
- la **indicazione di lettura**, potrebbe, a volte, non portare a lettura immediata (*recv* con risultato -1)
 si suggerisce di farne una serie fino a risultato OK
 alla fine, si può ottenere una *lettura vuota* (0 byte):
 questa, in caso di socket connessa in stream, indica che il canale è stato chiuso dal partner
 (o un guasto ha chiuso il nodo)

Utilità per maschere di bit

Macro definite in *usr/include/stdio.h* o altri file

```
typedef long fd_mask;
#define NFDBITS (sizeof(fd_mask) * 8)
    /* 8 bit in un byte */

#define howmany(x,y) (((x)+((y)-1))/(y))

typedef struct fd_set {
    fd_mask
    fds_bits[howmany(FD_SETSIZE,NFDBITS)];
} fd_set; /* definizione della maschera */

/* macro per inserire un file descriptor nella maschera,
   per togliere, per verificare la presenza ed per azzerare
   la maschera */

#define FD_SET(n,p)
((p)->fds_bits[(n)/NFDBITS] |= (1<<((n)% NFDBITS)))

#define FD_CLR(n,p)
((p)->fds_bits[(n)/NFDBITS] &=~(1<<((n)% NFDBITS)))

#define FD_ISSET(n,p)
((p)->fds_bits[(n)/NFDBITS]&(1<<((n)% NFDBITS)))

#define FD_ZERO(p)
memset((char *) (p), (char) 0,sizeof(*(p)))
```

Ci sono altre funzioni ausiliarie utili. Si vedano le funzioni **bcopy** e **bzero** e **memset**, **memcpy**, **memcmp** per lavorare su blocchi di memoria utente

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);
```

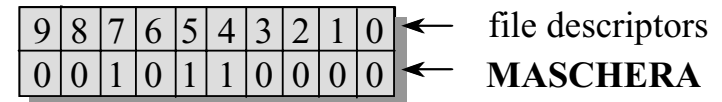
FD_SET include il particolare fd in fdset

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 descrittori a zero

le macro consentono di operare agilmente sugli array di bit che rappresentano gli eventi di interesse su cui la select deve fare la osservazione



La select realizza una possibilità di **ricezione di un insieme di eventi**, sincronizzandosi fino alla ricezione del primo evento

Esempio di uso della `select()` e macro di manipolazione delle **relative maschere**

```
#include <stdio.h>
do_select(s)
    int s; /* socket descriptor */
{struct fd_set read_mask, write_mask;
 /* dichiarazione delle maschere */
int nfds; int nfd;

/* ciclo infinito */
for (;;) {
/* azzera la maschera */
    FD_ZERO(&read_mask);
    FD_ZERO(&write_mask);
    FD_SET(s,&read_mask);
/* inserisce il file descriptor nella maschera */
    FD_SET(s,&write_mask);
    nfds=s+1;
    nfd=select (nfds,&read_mask,&write_mask,NULL,
                (struct timeval*)0);

    if (nfd==-1)
/* per select generali, potrebbe essere timeout*/
        {perror("select: condizione inattesa."); exit(1);}
    if (FD_ISSET(s,&read_mask)) do_read(s);
/*cerca il file descriptor nella maschera*/
    if (FD_ISSET(s,&write_mask)) do_write(s);
    }
}
```

Esempio di Server Concorrente con connessioni

```
#define LEN 100
typedef struct { long len; char name[LEN]; } Request ;
typedef struct { long value; long errno; /* 0 successo */
                } Response ;
typedef int HANDLE;

int recv_request (HANDLE h, Request *req)
/* funzione di ricezione di una intera richiesta */
{ int r_bytes, n;
  int len = sizeof *req;

  for (r_bytes = 0; r_bytes < len; r_bytes += n)
    { n = recv (h, ((char *) req) + r_bytes, len - r_bytes, 0);
      if (n <= 0) return n; }
/* Decodifica len */
  req->len = ntohl (req->len);
  return r_bytes;
}

int send_response (HANDLE h, long value)
/* funzione di invio al cliente di una risposta */
{ Response res; size_t w_bytes;
  size_t len = sizeof res;
  res.errno = value == -1 ? htonl (errno) : 0;
  res.value = htonl (value);
  for (w_bytes = 0; w_bytes < len; w_bytes += n)
    { n = send (h, ((char *) &res) + w_bytes, len - w_bytes, 0);
      if (n <= 0) return n; }
  return w_bytes;
}
```



```

HANDLE create_server_endpoint (u_short port)
/* funzione di attesa di richieste di canale */
{ struct sockaddr_in addr; HANDLE h;
  h = socket (PF_INET, SOCK_STREAM, 0);
  memset ((void *) &addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port = htons (port);
  addr.sin_addr.s_addr = INADDR_ANY;
  bind (h, (struct sockaddr *) &addr, sizeof addr);
  listen (h, 5);
  return h;
}

long action (Request *req);
/* azione qualunque di servizio */
{ ... }

long handle (HANDLE h) {
/* per ogni possibile evento da parte di un cliente
  connesso, si esegue la funzione handle
  in questa funzione si riceve la richiesta (letture pronta)
  si attua l'azione e
  si invia la risposta
*/
struct Request req; long value;
if (recv_request (h, &req) <= 0) return 0;
  /* azione */
value = action (&req);
return send_response (h, value);
}

```

```

int main (int argc, char *argv[])
/* servitore concorrente con connessione */
{ u_short port = argc > 1 ? atoi(argv[1]) : 10000;
  /* Porta di listen per connessioni */
/* attesa di richieste di connessione */
  HANDLE listener = create_server_endpoint(port);
  HANDLE maxhp1 = listener + 1;
  /* numero corrente di possibili socket da verificare */
  fd_set read_hs, temp_hs;
  FD_ZERO(&read_hs); FD_ZERO(&temp_hs);
  FD_SET(listener, &read_hs); FD_SET(listener, &temp_hs);
  temp_hs = read_hs;

  for (;;)
  { HANDLE h;
    select (maxhp1, &temp_hs, 0, 0, 0);
    /* verifica delle richieste presenti */
    for (h = listener + 1; h < maxhp1; h++)
    /* richieste sulle connessioni */
    { if (FD_ISSET(h, &temp_hs))
      if (handle (h) == 0)
        /* Chiusura della connessione da parte del cliente */
        { FD_CLR(h, &read_hs); close(h); }
    }
    /* richieste di nuove connessioni */
    if (FD_ISSET(listener, &temp_hs)) {
      h = accept (listener, 0, 0);
      FD_SET(h, &read_hs);
      if (maxhp1 <= h) maxhp1 = h + 1; }
    temp_hs = read_hs;
  }
}

```

Servitore Multiplo (multiservizio)

Possibilità di un unico servitore per più servizi

Vantaggi: un unico collettore attivo che si incarica di smistare le richieste

Il **servitore multiplo** può

- **portare a termine** completamente i servizi per richiesta
- **incaricare** altri processi del servizio (specie in caso di connessione e stato) e **tornare al servizio** di altre richieste

Unico processo master per molti servizi

riconosce i servizi e attiva il servizio stesso
Un master può ricevere più richieste

- servire direttamente la richiesta
- generare un processo a parte

Vedi BSD UNIX inetd (/etc/services)

inetd

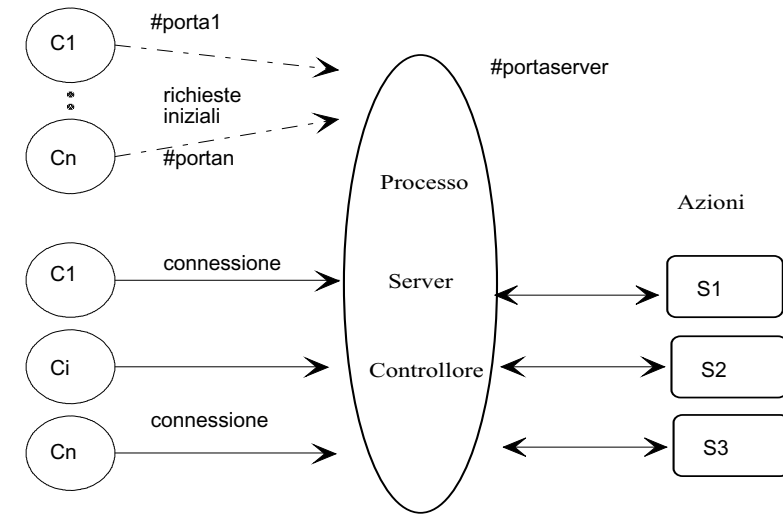
*alcuni servizi sono svolti in modo diretto (interno)
altri sono invece delegati a processi creati su richiesta*

Problemi:

Il server può diventare il **collo di bottiglia** del sistema

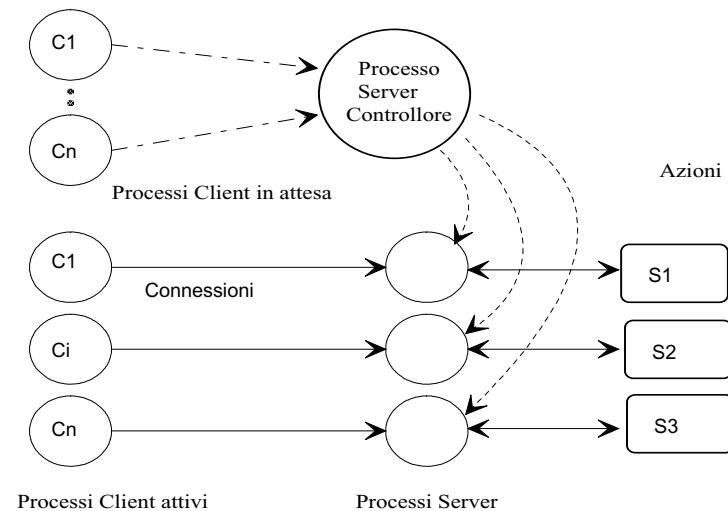
Necessità di decisioni rapide e leggere

Server multiplo singolo processo con connessione



Processi Client attivi

Server multiplo con processi multipli



Processi Client attivi

Processi Server

Esempio di file di configurazione del demone INETD

```
# @(#)inetd.conf 1.24 92/04/14 SMI
# Configuration file for inetd(8). See inetd.conf(5).
#
# To re-configure the running inetd process, edit this file, then
# send the inetd process a SIGHUP.

# Internet services syntax:
# <service_name> <socket_type> <proto> <flags> <user>
#           <server_pathname> <args>
#
# Ftp and telnet are standard Internet services.
ftp      stream tcp nowait root    /usr/etc/in.ftpd  in.ftpd
telnet    stream tcp nowait root    /usr/etc/in.telnetd in.telnetd
#
# Tnamed serves the obsolete IEN-116 name server protocol.
#
# Shell, login, exec, comsat and talk are BSD protocols.
shell     stream tcp nowait root    /usr/etc/in.rshd  in.rshd
login     stream tcp nowait root    /usr/etc/in.rlogind in.rlogind
exec      stream tcp nowait root    /usr/etc/in.rexecd in.rexecd
talk    dgram udp wait root    /usr/etc/in.talkd in.talkd
#
# Finger, systat and netstat give out user information which may be
# valuable to potential "system crackers." Many sites choose to disable
# some or all of these services to improve security.
finger  stream tcp nowait nobody /usr/etc/in.fingerd in.fingerd
#
#systat   stream tcp nowait root /usr/bin/ps      ps -auwwx
#netstat  stream tcp nowait root /usr/ucb/netstat netstat -f inet
#
# Time service is used for clock synchronization.
time      stream tcp nowait root    internal
time    dgram udp wait root    internal
#
```

```
# Echo, discard, daytime, and chargen are used primarily for testing.
echo      stream tcp nowait root internal
echo      dgram  udp wait    root internal
discard   stream tcp nowait root internal
discard   dgram  udp wait    root internal
daytime   stream tcp nowait root internal
daytime   dgram  udp wait    root internal
#
# RPC services syntax:
# <rpc_prog>/<vers> <socket_type> rpc/<proto> <flags> <user>
#           <pathname> <args>
#
# Rexecution usually obtained in other ways.
#rexrd/1    stream rpc/tcp wait root    /usr/etc/rpc.rexd rpc.rexd
#
# Rquotad serves UFS disk quotas to NFS clients.
rquotad/1  dgram rpc/udp wait root    /usr/etc/rpc.rquotad
rpc.rquotad
#
# Rstatd is used by programs such as perfmeter.
rstatd/2-4  dgram  rpc/udp wait root /usr/etc/rpc.rstatd  rpc.rstatd
#
# The rusers service gives out user information.
rusersd/1-2  dgram  rpc/udp wait root /usr/etc/rpc.rusersd  rpc.rusersd
#
# The spray server is used primarily for testing.
sprayd/1     dgram  rpc/udp wait root /usr/etc/rpc.sprayd  rpc.sprayd
#
# The rwall server lets anyone on the network bother everyone on your machine.
wallld/1     dgram  rpc/udp wait root /usr/etc/rpc.wallld  rpc.wallld
#
# rpc.cmsd is a data base daemon which manages calendar data backed
# by files in /usr/spool/calendar
100068/2-3   dgram  rpc/udp wait root /usr/etc/rpc.cmsd  rpc.cmsd
#
# Sun ToolTalk Database Server
100083/1     stream  rpc/tcp wait root /usr/etc/rpc.ttdbserverd
rpc.ttdbserverd
```

Dalla parte del cliente

anche concorrenza dalla parte del cliente
per ottenere i vantaggi di servizi multipli

a) soluzione concorrente

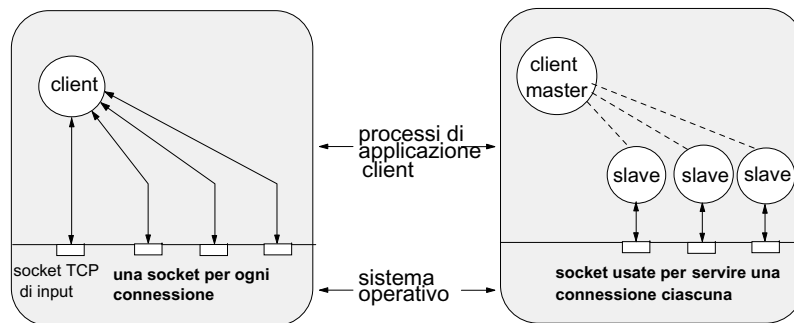
possibilità che il cliente unico gestisca più interazioni
necessità di asincronismo

uso di select e politiche di servizio opportune

b) soluzione parallela

possibilità di generare più processi (slave) che
gestiscono ciascuno una diversa interazione con un
server
possibilità di interagire con più server contemporaneamente (multicast)

*come interagire con il server in modo appropriato?
condivisione di socket?*



Gestione opzioni per Stream Socket

funzioni primitive per configurare socket

getsockopt() / setsockopt()

leggere e fissare le modalità di utilizzo delle socket
(tipicamente il valore del campo vale 0 o 1)

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

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

```
char optval = 1;
```

```
int getsockopt (s, level, optname, optval, optlen)
```

```
int s, level, optname, *optlen;
```

```
int setsockopt (s, level, optname, &optval, optlen)
```

```
int optlen;
```

s == socket descriptor legato alla socket

level == livello di protocollo per socket SOL_SOCKET

optname == nome dell'opzione

optval == puntatore ad un'area di memoria per valore

optlen == lunghezza (o puntatore) quarto argomento

Opzioni	Descrizione
SO_DEBUG	abilita il debugging (valore diverso da zero)
SO_REUSEADDR	riuso dell'indirizzo locale
SO_DONTROUTE	abilita il routing dei messaggi uscenti
SO_LINGER	ritarda la chiusura per messaggi pendenti
SO_BROADCAST	abilita la trasmissione broadcast
SO_OOBINLINE	messaggi prioritari pari a quelli ordinari
SO_SNDBUF	setta dimensioni dell'output buffer
SO_RCVBUF	setta dimensioni dell'input buffer
SO_SNDLOWAT	setta limite inferiore di controllo di flusso out
SO_PCVLOWAT	limite inferiore di controllo di flusso in input
SO_SNDTIMEO	setta il timeout dell'output
SO_RCVTIMEO	setta il timeout dell'input
SO_USELOOPBACK	abilita network bypass
SO_PROTOCOLTYPE	setta tipo di protocollo

Riutilizzo del socket address (STREAM)

Opzione **SO_REUSEADDR** modifica bind()

Il sistema tende a non ammettere più di un utilizzo di un indirizzo locale

con l'opzione, si convalida l'indirizzo di una socket **senza controllare la unicità di associazione**

Socket con wildcard address per una porta in uso
Controllo al momento della connessione

Protocollo	Coda di rx	Coda di tx	Indirizzo locale	Indirizzo remoto	Stato della connessione
TCP	0	0	*.2000	*.*	ASCOLTO

netstat segnala un demone sulla porta 2000
all'indirizzo generico Internet * (wildcard address)

Un processo client richiede una connessione

Protocollo	Coda di rx	Coda di tx	Indirizzo locale	Indirizzo remoto	Stato cns
TCP	0	0	137.204.57.33.2000	137.204.57.32.4000	STABILITA
TCP	0	0	*.2000	*.*	ASCOLTO

Il processo demone originale continua ad attendere sulla socket legata alla porta 2000 ed al wildcard address

Se termina, il riavvio necessita **SO_REUSEADDR**

Senza, il processo tenta il collegamento alla porta 2000 con wildcard address ==> bind() errore ==>

già presente una connessione per lo stesso socket address

```
int optval=1;
setsockopt(s, SOL_SOCKET, SO_REUSEADDR,
           &optval, sizeof(optval));
bind(s,&sin,sizeof(sin));
```

Controllo periodico della connessione

Il protocollo di trasporto può inviare messaggi di controllo periodici per analizzarne lo stato di una connessione (**SO_KEEPALIVE**)

Se problemi ==>

connessione è considerata abbattuta

i processi avvertiti da un **SIGPIPE** *chi invia dati*
da **end-of-file** *chi li riceve*

Verifica ogni 2 ore e dopo 6 minuti di tentativi
opzione in dominio Internet tipo boolean

Timeout per operazioni

Opzioni **SO_SNDTIMEO** e **SO_RCVTIMEO**

Tempo massimo di durata di una primitiva di send/ receive
==> dopo, il processo viene sbloccato

```
int result; int waittime=10;
result=setsockopt(s, SOL_SOCKET, SO_RCVTIMEO,
                 &waittime, sizeof(waittime));
```

Dimensioni buffer di trasmissione/ricezione

Opzioni **SO_SNDBUF** e **SO_RCVBUF**

Aumento della dimensione buffer di trasmissione ==>

invio messaggi più grandi senza attesa
massima dimensione possibile 65535 byte

```
int result; int buffersize=10000;
result=setsockopt(s, SOL_SOCKET, SO_RCVBUF,
                 &buffersize, sizeof(buffersize));
```

Azioni eseguite alla chiusura di socket

Con opzione **SO_LINGER** si influenzano i messaggi solo in caso di **uso di close**

struttura di dati *linger* in `/usr/include/sys/socket.h`:
`struct linger { int l_onoff; int l_linger; /* attesa in sec */ }`

l_onoff	l_linger	Graceful/Hard Close	Chiusura Con/ Senza attesa
0	don't care	G	Senza
1	0	H	Senza
1	valore > 0	G	Con

valore di default l_onoff 0
disconnessione graceful
NON abbiamo controllo della memoria

chiusura hard della connessione

l_linger a 0
ogni dato non inviato è perso

chiusura graceful della connessione

l_onoff ad 1 e l_linger valore positivo
la chiamata `close()` completa dopo il tempo in secondi specificato dal `linger` e (si spera) dopo la trasmissione di tutti i dati nel buffer

L'opzione di linger non influenza lo shutdown()

Socket con modalità asincrona

Socket che non blocchino il chiamante

a default => modello **sincrono** delle socket

Socket asincrone

Si rendono asincrone con primitiva `ioctl` o `fctl`

Socket asincrone che si gestiscono con **select** o con un **gestore del segnale** di completamento della operazione
SIGIO segnala un cambiamento di stato della socket
(ad esempio per l'arrivo di dati)

SIGIO ignorato dai processi che non hanno definito un gestore

gestore di SIGIO robusto per situazioni inaspettate

Segnale e consegna

SIGIO socket asincrona

Attributo `FIOASYNC` con primitiva `ioctl()`

`#include <sys/ioctl.h>`

`int ioctl (int filedesc, int request, ... /* args */)`

filedesc == file descriptor

request == tipo di attributo da assegnare

poi == valori da assegnare all'attributo

parametro *process group* del processo alla socket asincrona. Primitiva `ioctl()` attributo `SIOCSPGRP`

Consegna di `SIGIO`:

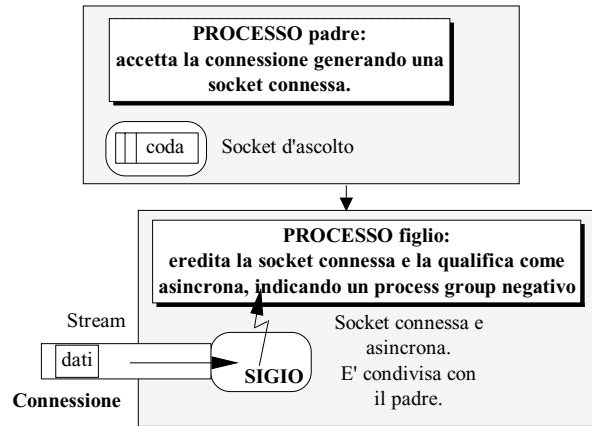
`int ioctl (filedesc, SIOCSPGRP, &flag) flag`

valore **negativo** => segnale per il processo con pid uguale al process group

valore **positivo** => segnale arriva a tutti i processi del process group

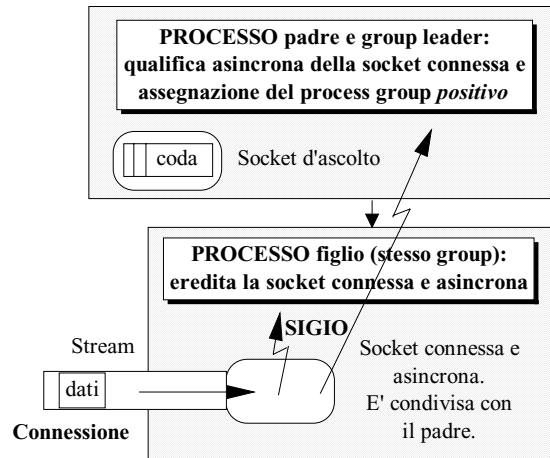
Segnale completamento

Segnale SIGIO e consegna al processo



Valore negativo

Consegna SIGIO al solo processo del gruppo



Valore positivo

Consegna di SIGIO a tutti i processi del gruppo

ESEMPIO

Due esempi di socket asincrone con consegna del segnale al singolo processo ed al gruppo di processi:

```
int ls; /* socket d'ascolto */
int flag=1;
/* valore per FIOASYNC per socket asincrona */
int handler; /* gestore delle I/O sulle socket */
```

```
signal(SIGIO,handler);
if (ioctl (ls,FIOASYNC,&flag) == -1)
{ perror("non posso rendere asincrona la socket");
  exit(1); }
flag= -getpid(); /* identificatore di processo negativo */
if (ioctl (ls,SIOCSPGRP,&flag) == -1)
{ perror("non si assegna il process group alla socket");
  exit(1); }
```

Le opzioni sono poco standard.

Per Linux:

```
int flag, ret;

#ifdef __linux__
flag = fcntl (s, F_GETFL);
if (fcntl (s, F_SETFL, flag | FASYNC) == -1)
{ perror("fcntl failed");
  exit(-1);}
flag = -getpid();
if (fcntl (s, F_SETOWN, flag) == -1)
{ perror("fcntl F_SETOWN"); exit (-2);}
#endif
```

Nonblocking I/O (per la stessa azione)

Primitive bloccanti

anche ottenute con primitiva `ioctl()`

parametro FIONBIO

valore 0 / 1
modalità **bloccante** / **non bloccante**

chiamate modificate

- **`accept()`**
restituisce errore di tipo `EWOULDBLOCK`
- **`connect()`**
condizione d'errore di tipo `EINPROGRESS`
- **`recv()` e `read()`**
condizione d'errore di tipo `EWOULDBLOCK`
- **`send()` e `write()`**
condizione d'errore `EWOULDBLOCK`

Esempio di assegnazione dell'attributo non bloccante

```
#include <sys/ioctl.h>
int s; /* socket descriptor */
int arg=1; /* valore per la qualifica non blocking */
ioctl(s,FIONBIO,&arg);
ioctl(s,FIOASYNC,&arg);
```

Funzionamento **non bloccante** delle primitive anche mediante chiamata `fcntl()`

`FCNTL ()` dedicata al controllo dei file aperti

FCNTL

```
#include <fcntl.h>
int fcntl (fileds, cmd, .../* argomenti */)
    int fileds; /* descrittore del file */
    int cmd;
    /* argomenti */
```

```
if (fcntl (descr, F_SETFL, FNDELAY) < 0)
{ perror("non si riesce a rendere asincrona la socket");
  exit(1); }
```

anche `O_NDELAY`

I comandi hanno significati diversi in

System V

- **`O_NDELAY`**

le chiamate `read()`, `recv()`, `send()`, `write()` senza successo immediato ritornano un valore 0.

Possibile problema: equivale alla condizione di end-of-file sulle socket

- **`O_NONBLOCK`** (POSIX.1 standard, System V vers.4)

Le chiamate senza successo immediato ritornano -1 e la condizione d'errore `EAGAIN`

BSD

- **`FNDELAY`, `O_NONBLOCK`**

le chiamate `read()`, `recv()`, `send()`, `write()` senza successo immediato ritornano con valore -1 e la condizione d'errore `EWOULDBLOCK`

Progetto

Unix e TCP/IP portano a considerare la concorrenza tra diverse connessioni, più che sulle singole richieste

		Tipo di comunicazione	
		con connessione	senza connessione
S E R V E R	sequenziale iterativo	servizi poco pesanti e affidabili	Molto diffusi: per servizi poco pesanti nonaffidabili solitamente stateless
	concorrente singolo processo	Il singolo processo facilita condivisione dati tra richieste diverse Servizi poco pesanti e affidabili	Poco usati
	concorrente multi processo	Molto diffusi: servizi pesanti e affidabili es. Server Web	Costo fork non rende questa classe conveniente (a meno di pesantissime operazioni di I/O)

In Unix un Server concorrente si ottiene generando un processo nuovo per ogni richiesta di servizio (server concorrente multi processo)

Si possono realizzare anche server concorrenti singolo processo, sfruttando multiplexing dell'I/O e di I/O signal driven