Università degli Studi di Bologna
Facoltà di Ingegneria

Corso di
Reti di Calcolatori L-A

Progetto C/S con Socket in C

Antonio Corradi

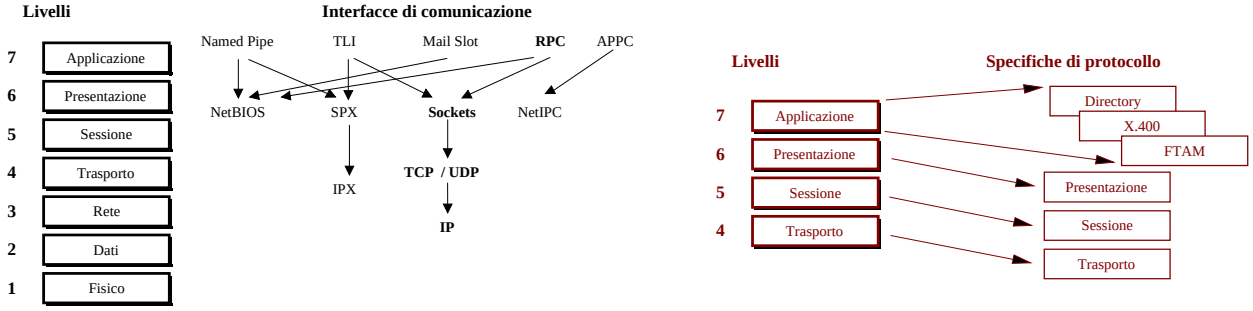
Anno accademico 2009/2010

COMUNICAZIONE e SOCKET

Necessità di **Strumenti di Comunicazione per supportare scambio di messaggi**

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

Scena con molti strumenti diversi, applicativi e di livello diverso



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

UNIX: STRUMENTI di COMUNICAZIONE

UNIX ⇨ **modello** e strumenti per **comunicazione/sincronizzazione**

Si definisce e si regola la comunicazione/sincronizzazione locale

Uso di **segnali** ⇨

processo invia un evento senza indicazione del mittente

Uso di **file** ⇨

solo tra processi che condividono il file system sullo stesso nodo

Poi, solo tra processi coresidenti sullo stesso nodo

- **pipe** (solo tra processi con un avo in comune)
- **pipe con nome** (per processi su una stessa macchina)
- **shared memory** (stessa macchina)

Comunicazione e sincronizzazione remota ⇨

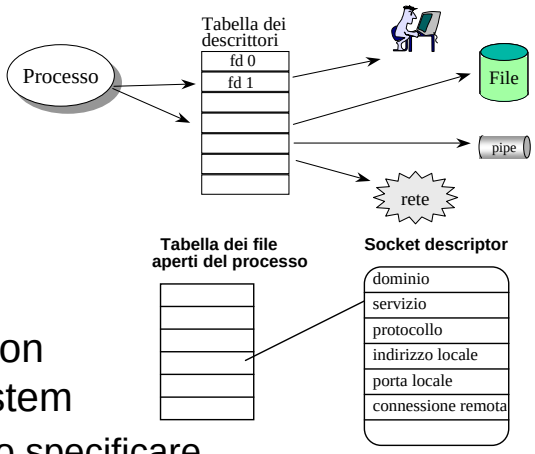
SOCKET Unix BSD (Berkeley Software Distribution)

UNIX: MODELLO di USO

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

Paradigma di uso: **open-read-write-close**

- apertura della sessione
- operazioni della sessione (read / write)
- chiusura della sessione



Le socket conformi a questo paradigma con trasparenza rispetto alle azioni sul file system

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

protocollo di trasporto; e **quadrupla**

< indirizzo locale; processo locale; indirizzo remoto; processo remoto>

UNIX: PRIMITIVE

UNIX deve fornire funzioni primitive di comunicazione

UNIX Berkeley introduce il meccanismo di socket,

strumenti di comunicazione locali o remote con politiche differenziate, in alternativa ai problemi degli strumenti concentrati, trasparente e ben integrata con processi e file

SOCKET su cui i processi possono scrivere/leggere messaggi e stream, con molte opzioni e requisiti

- **eterogeneità**: comunicazione fra processi su architetture diverse
- **trasparenza**: la comunicazione fra processi indipendentemente dalla localizzazione fisica
- **efficienza**: l'applicabilità delle socket limitata dalla sola **performance**
- **compatibilità**: i naive process (filtri) devono potere lavorare in ambienti distribuiti senza subire alcuna modifica
- **completezza**: protocolli di comunicazione diversi e differenziati

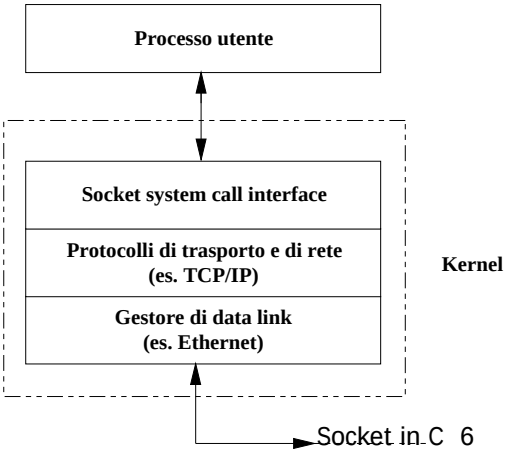
UNIX: TRASPARENZA

Le socket come strumento con interfaccia omogenea a quella usuale per i servizi da invocare in modo trasparente

- Socket endpoint della comunicazione
- Socket descriptor integrato con i file descriptor

con protocolli di trasporto diversi e default TCP/IP (sia UDP sia TCP)

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



SOCKET: DOMINIO di COMUNICAZIONE

Dominio di comunicazione per socket come specifica del modello: semantica di comunicazione + standard di nomi relativi

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

Semantica di comunicazione include

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

Naming modo per indicare i punti terminali di comunicazione

Il dominio più appropriato scelto tramite un'interfaccia standard

La prima scelta di esempio è tra comunicazione con connessione e senza connessione tipica di ogni dominio

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

SCELTE di COMUNICAZIONE

Tipi di servizio e socket

- datagram: scambio di messaggi senza garanzie (best effort)
- stream: scambio bidirezionale di messaggi in ordine, senza errori, non duplicati, nessun confine di messaggio, out-of-band flusso (stream virtuale e non reale)
- seqpacket: messaggi con numero di ordine (XNS)
- raw: messaggi non trattati ma solo scambiati (per debug protocolli)

Protocolli diversi in ogni dominio di comunicazione

- UNIX (AF_UNIX)
- Internet (AF_INET)
- XEROX (AF_NS)
- CCITT (AF_CCITT) X.25

Ancora SCELTE di COMUNICAZIONE

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

prefisso AF ⇨ Address Family
PF_UNIX, PF_INET, PF_NS,
prefisso PF ⇨ Protocol Family
cioè address family
PF_SNA, PF_DECnet e
PF_APPLETALK

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

SISTEMA di NOMI per le SOCKET

Nomi logici delle socket (nomi LOCALI) ⇨ indirizzo di socket nel dominio

Nomi fisici da associare (nomi GLOBALI) ⇨ una porta sul nodo

una socket deve essere collegata al sistema fisico e richiede binding, cioè il legame tra socket logica ed entità fisica corrispondente

Half-association come coppia di nomi logica e fisica

- dominio **Internet**: socket collegata a porta locale al nodo
  { famiglia indirizzo, indirizzo Internet, numero di porta}
- dominio **UNIX**: socket legata al file system locale
  {famiglia indirizzo, path nel filesystem, file associato}
- dominio CCITT: indirizzamento legato al protocollo di rete X.25

In Internet

Nodi **nomi IP** **{identificatore_rete, identificatore_host}**

Porta **numeri** distintivi sul nodo (1-1023 di sistema, 1024-65535 liberi)

TIPI per INDIRIZZI e NOMI SOCKET

Per le variabili che rappresentano i nomi delle socket si deve considerare la necessità di flessibilità degli indirizzi

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 mioindirizzosocket; /* variabile per il nome*/
```

sin_family ⇨ famiglia di indirizzamento sempre AF_INET

sin_port ⇨ numero di porta

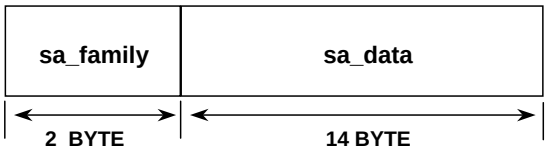
sin_addr ⇨ indirizzo Internet del nodo remoto (numero IP)

INDIRIZZI e NOMI SOCKET

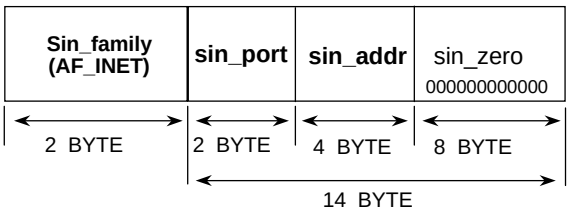
Si usano strutture dati per i nomi fisici che servono alla applicazione

RAPPRESENTAZIONE dei NOMI in C

struct sockaddr



struct sockaddr_in



I programmi usano di solito un puntatore generico ad una locazione di memoria del tipo necessario

char * in C **void *** in ANSI C

Si vedano i file di inclusione tipici …

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

FUNZIONI di SUPPORTO ai NOMI

Un utente conosce il nome logico Internet di un Host remoto come stringa e non conosce il nome fisico corrispondente

corrispondenza tra nome logico e nome fisico per le primitive ⇨

primitiva **gethostbyname()** restituisce l'indirizzo Internet e dettagli

```
#include <netdb.h>
struct hostent * gethostbyname (name)
    char * name;
```

gethostbyname restituisce un puntatore alla struttura **hostent** oppure NULL se fallisce; il parametro name ricercato nel file /etc/hosts che si comporta come una tabella di corrispondenze, ad esempio…

```
137.204.56.11    didahp1    hp1
137.204.56.12    didahp2    hp2
137.204.56.13    didahp3    hp3
```

La ricerca avviene localmente, poi integrata anche con strumenti come sistemi di nomi (DNS)

FUNZIONE GETHOSTBYNAME

Struttura hostent (intesa come descrizione completa di host)

```
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 del'indirizzo */
 char ** h_addr_list;   /* lista indirizzi dai nomi host */
#define  h_addr  h_addr_list[0]  /* indirizzo nome host */
}
```

La struttura hostent permette di avere informazioni del tutto complete di un nodo di cui abbiamo un nome logico

Le informazioni più rilevanti sono il nome fisico primario (primo nella lista, cioè **h_addr**) e la sua lunghezza (in Internet è fissa), ma anche lista di nomi logici e fisici

Ogni indirizzo caratterizzato da contenuto e lunghezza (variabile)

USO GETHOSTBYNAME

Esempio di utilizzo della **gehostbyname** per risolvere l'indirizzo logico: si usa una variabile di appoggio riferita tramite puntatore che ha valore in caso di successo per dare valore a **peeraddr**

```
#include <netdb.h>
struct hostent *  hp;
struct sockaddr_in      peeraddr;
peeraddr.sin_family  =  AF_INET;
peeraddr.sin_port    =  22375;
if (hp = gethostbyname (argv[1]) ) /* in caso di successo */
 peeraddr.sin_addr.s_addr = /* assegnamento dei 4 byte IP */
  ((struct in_addr *) /* casting */
   (hp->h_addr)) -> s_addr;
else /* errore o azione alternativa */
```

In ogni caso, si cerca di ottenere il valore di IP nel campo corretto

FUNZIONE GETSERVBYNAME

In modo simile, per consentire ad un utente di usare dei *nomi logici di servizio senza ricordare la porta*, la funzione **getservbyname**( ) di utilità restituisce il numero di porta relativo ad un servizio

Anche se non ci sono corrispondenze obbligatorie, la pratica di uso ha portato ad una serie di porte note (*well-known port*) associate stabilmente a servizi noti, per consentire una più facile richiesta

file /etc/services come tabella di corrispondenze fra servizi e porte su cui si cerca la corrspondenza {nome servizio, protocollo, ⇨ porta}

```
echo        7/tcp               # Echo
systat      11/tcp  users       # Active Users
daytime     13/tcp              # Daytime
daytime     13/udp              #
qotd        17/tcp  quote       # Quote of the Day
ftp-data    20/tcp              # File Transfer Protocol (Data)
ftp         21/tcp              # File Transfer Protocol (Control)
```

USO GETSERVBYNAME

Esempio di utilizzo della **getservbyname** per trovare numero di porta usando una variabile di appoggio riferita tramite puntatore che permette di ritrovare il numero di porta nel campo **s_port**

```
#include <netdb.h>
struct servent * getservbyname (name, proto)
          char *name, *proto;
```

/* se TCP è l'unico servizio registrato con quel nome, allora 0 */

Si utilizza prima di usare l'indirizzo del servizio (tipo `sockaddress_in`), per dare valore alla parte di numero di porta…

```
#include <netdb.h>    /* vedi formato del record struct servent */
struct servent *sp;
struct sockaddr_in peeraddr;
sp = getservbyname("echo","tcp");
peeraddr.sin_port = sp->s_port; /* assegnamento della porta */
```


PRIMITIVE PRELIMINARI

Per lavorare sulle socket sono preliminari due **primitive di nome**

Per il nome logico LOCALE, si deve creare socket in ogni processo

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

Si è introdotta una nuova azione per l'impegno dei nomi fisici GLOBALI, attuando l'aggancio al sistema 'fisico' di nodi e porte

```
rok = bind (s, nome, lungnome)
int rok, s;          /* le primitive restituiscono valore positivo se ok */
struct  sockaddr *nome;  /* indirizzo locale per la socket */
int  lungnome;           /* lunghezza indirizzo locale */
```

Le primitive di nome sono **significative** ed **essenziali** entrambe

SOCKET DATAGRAM

Le **socket datagram** sono dei veri **end-point di comunicazione e** permettono di formare **half-association** (relative ad un solo processo), ma usabili per **comunicare con chiunque del dominio**

Si possono **scambiare messaggi** (datagrammi) sapendo/avendo:

processo Mittente o Cliente

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

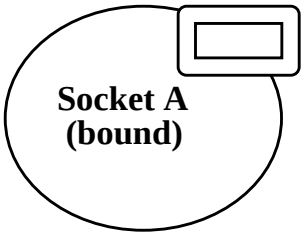
processo Ricevente o Server

* dichiarazione delle variabili di riferimento a una **socket**
* conoscenza della porta per il **servizio da offrire**
* ricezione su qualunque indirizzo IP locale (wildcard address) utile per server con più connessioni, detti **multiporta**

SOCKET DATAGRAM

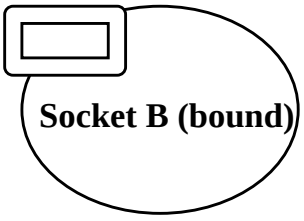
Le **socket datagram** sono degli **end-point per comunicazione con chiunque del dominio**

processo Mittente o Cliente



- il client ha creato la socket
- il client ha collegato la socket ad un indirizzo

processo Ricevente o Server

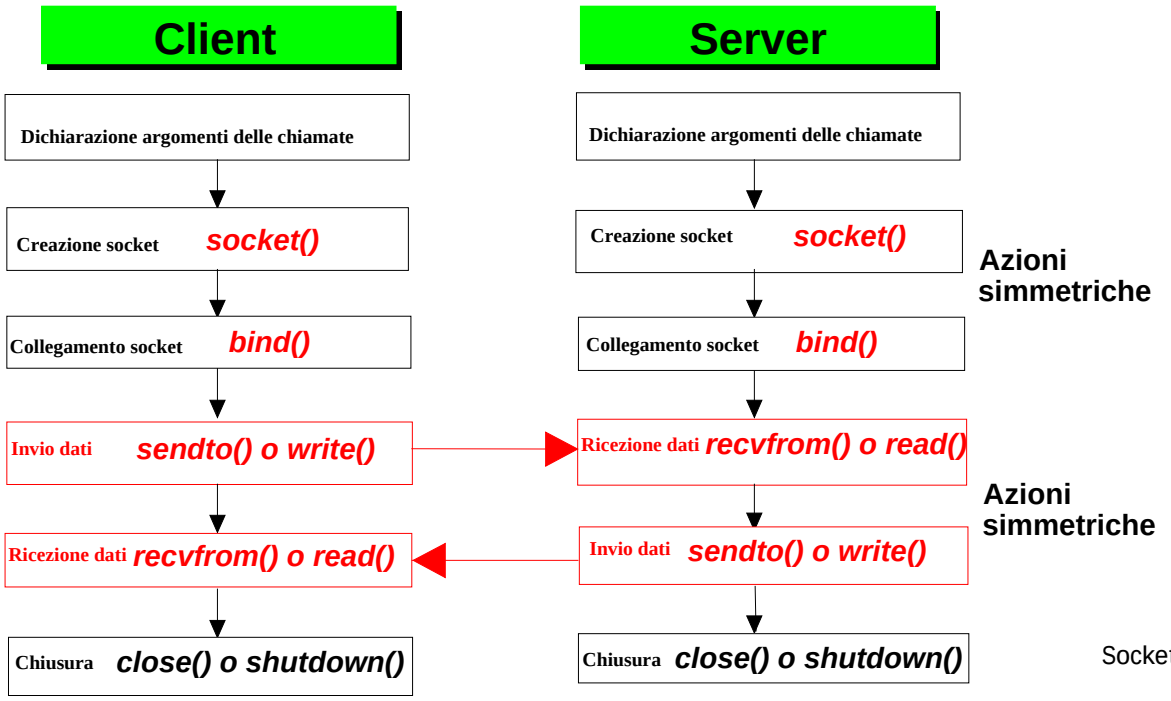


- il server ha creato la socket
- il server ha collegato la socket ad un indirizzo

Le **socket datagram** permettono direttamente di fare delle **azioni di invio/ricezione** a chiunque rispetti il protocollo

PROTOCOLLO DATAGRAM

Le **socket datagram** sono usate con un **protocollo che si basa sulla sequenza di primitive** qui sotto (alcune opzionali, quali?)



PRIMITIVE di COMUNICAZIONE

Per comunicare ci sono due primitive, di *invio e ricezione* datagrammi

```
nbytes = sendto (s, msg, len, flags, to, tolen)
int s, nbytes; char *msg; /* area che contiene i dati */
int len, flags; /* indirizzo, lunghezza e flag di operazione */
struct sockaddr_in *to; int tolen; /* indirizzo e lunghezza*/
nbytes = recvfrom (s, buf, len, flags, from, fromlen)
int s, nbytes; char *buf; /* area per contenere dati */
int len,flags; /* indirizzo, lunghezza e flag di operazione */
struct sockaddr_in *from; int * fromlen; /* ind e lung. ind.*/
```

restituiscono il numero dei byte trasmessi/ricevuti

nbytes ⇨ lunghezza messaggio inviato/ricevuto **s** ⇨ socket descriptor
buf, len ⇨ puntatore al messaggio o area e sua lunghezza
flags ⇨ flag (MSG_PEEK lascia il messaggio sulla socket)
to/from/tolen/fromlen ⇨ puntatore alla socket partner e sua lunghezza

USO delle SOCKET DATAGRAM

I mittenti/riceventi preparano sia le socket, sia le aree di memoria da scambiare, tramite messaggi (**`sendto`** e **`recvfrom`**)

I **datagrammi** scambiati sono **messaggi di lunghezza limitata** su cui si opera con una unica azione, in invio e ricezione (in modo unico) senza affidabilità alcuna

- Lunghezza massima del messaggio (spesso 9K byte o 16K byte)
- Uso del protocollo UDP e IP, non affidabili intrinsecamente

NON tutti i datagrammi inviati arrivano effettivamente al ricevente

`recvfrom` restituisce solo un datagramma per volta

- per prevenire situazioni di perdita di parti di messaggio massima area possibile

A livello utente si può ottenere maggiore affidabilità prevedendo

- invio di molti dati insieme **(mai mandare 2 datagrammi, se basta 1)**
- ritrasmissione dei messaggi e richiesta di datagramma di conferma

ESEMPIO di C/S con DATAGRAM

CLIENTE (MITTENTE) /* assumiamo che siano state invocate le socket e bind corrette e che in msg ci siano valori da trasmettere */

```
struct sockaddr_in * servaddr; char msg[2000]; int count; …
count = sendto (s, msg, strlen(msg), 0,
          servaddr, sizeof(struct sockaddr_in));
… close (s); /* se non si usa la socket, si deve chiuderla */
```

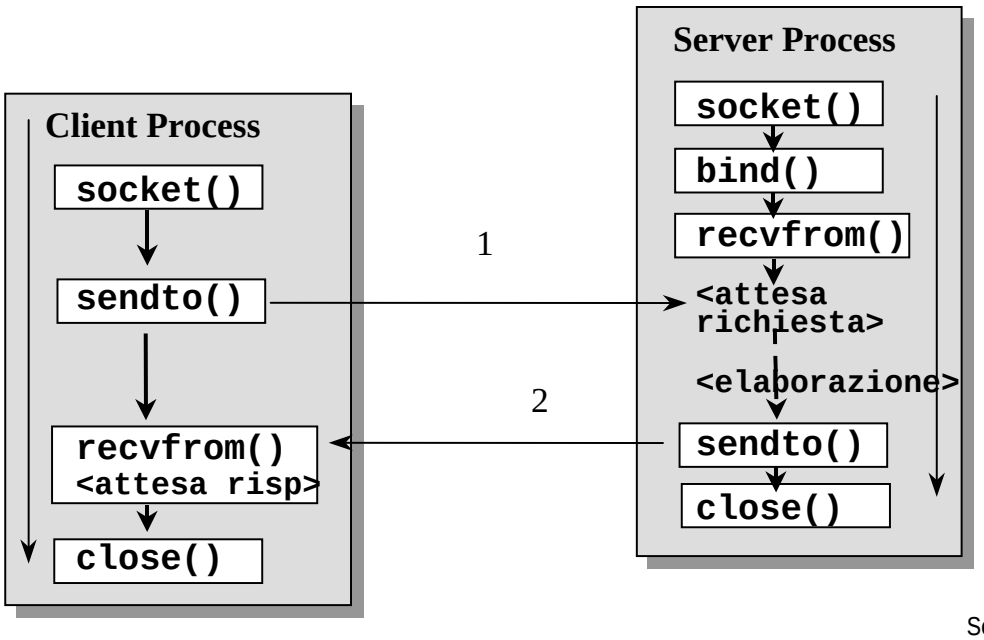
SERVITORE (RICEVENTE) /* anche qui sono state fatte le socket e bind, e si è definito il buffer per contenere le informazioni ricevute */

```
struct sockaddr_in * clientaddr;
char buffer[BUFFERSIZE]; int count, addrlen; …
addrlen = sizeof(sockaddr_in); /* valore di ritorno */
count = recvfrom(s, buffer,BUFFERSIZE,0, clientaddr,&addrlen);
… close (s);
```

/* la ricezione legge il datagramma sullo spazio riservato:
e se non ci sta? */

PROTOCOLLO C/S con DATAGRAM

I datagrammi sono **semplici messaggi** che spesso permettono di realizzare **interazioni Cliente/Servitore**



PROPRIETÀ dei DATAGRAM

In caso di scambi con datagrammi - e socket relative, ossia Client e Server realizzati con socket UDP

- **UDP non 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?)

- possibile **blocco del Client** in attesa di **risposta** che non arriva

  anche nel caso di invio di una richiesta a un Server non attivo non segnalati errori (errori notificati solo su socket connesse)

- UDP non ha alcun **controllo di flusso** (flow control)

  se il Server riceve troppi datagrammi per le sue capacità di elaborazione, questi vengono scartati, senza nessuna notifica ai Client

  la coda (area di memoria per accodare messaggi in IN/OUT per ogni socket) si può modificare in dimensione con l'uso di opzioni SO_RCVBUF/ SO_SENDBUF

PRIMITIVE SOCKET PASSO PASSO

Uso di alcune costanti molto utili e comode

Per esempio, per trovare gli indirizzi Internet locali

Uso di wildcard address

Viene riconosciuto INADDR_ANY un indirizzo di socket locale interpretato come qualunque indirizzo valido per il nodo corrente

Particolarmente utile per server in caso di residenza su workstation con più indirizzi Internet per 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;  /* qualunque indirizzo IP */
<<identificazione della socket>>
```


ESEMPIO di C/S con SOCKET

Usiamo come esempio il caso di **Echo, parte server (porta 7)**, ossia un servizio che rimanda al mittente ogni datagramma che arriva…

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

Parte dichiarativa iniziale e uso di funzioni per azzerare aree (bzero)

```
int sockfd, n, len;
char mesg[MAXLINE];     /* due socket, una locale e una remota */
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;
```

ESEMPIO di SERVER DATAGRAM

Abbiamo considerato incapsulate le primitive in una libreria C (altro file) che racchiude il tutto con una funzione omonima ma che tratta il caso di errore …

```
Bind(sockfd, &server_address, sizeof(server_address));
/* il server si sviluppa in un ciclo infinito, non termina mai */
for (;;)
{
 len = sizeof (client_address);
 n = Recvfrom(sockfd, mesg, MAXLINE, 0,
         &client_address, &len);
 m = Sendto(sockfd, mesg, n, 0,
         &client_address, len);
} …
```

Come sono le nuove funzioni che incapsulano le primitive?

ESEMPIO di LIBRERIA in C (repetita …)

Una libreria C si sviluppa sempre su due file,

- uno dichiarativo (*file.h*) che contiene solo dichiarazioni di procedure/funzioni (solo firme),
- uno operativo che contiene il reale codice ed esprime le reali operazioni (*file.c*)

Nel nostro caso i due file sono **`socketDatagram.h`**, che contiene

```
int Bind(int sockfd,
   struct sockaddr_in sock, int socklen);
```

Il primo file va incluso in ogni programma che ne abbia bisogno

Nel file **`socketDatagram.c`** abbiamo il vero e proprio codice (con eventuali variabili, ecc.) *da linkare con i programmi che lo usano*

```
int Bind(int sockfd, struct sockaddr_in sock, int socklen)
{ int res = bind(sockfd, sock, socklen);
  if (res>=0)return res;
  else /* gestisci uscita con exit */ ; }
```

SOCKET STREAM

Le **socket stream** prevedono una **risorsa** che rappresenta **la connessione virtuale** tra le entità interagenti

PROTOCOLLO e RUOLI differenziati CLIENTE/SERVITORE

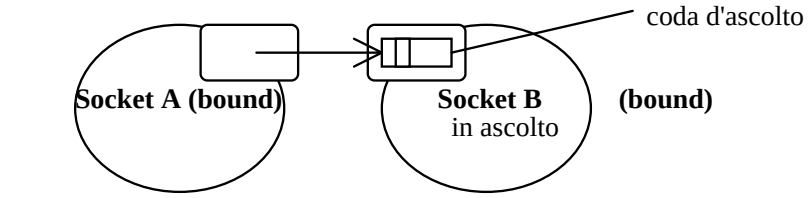
una entità (cliente) richiede il servizio

una entità (server) accetta il servizio e risponde

ruolo attivo/passivo nello stabilire la connessione

entità attiva richiede la connessione, la entità passiva accetta

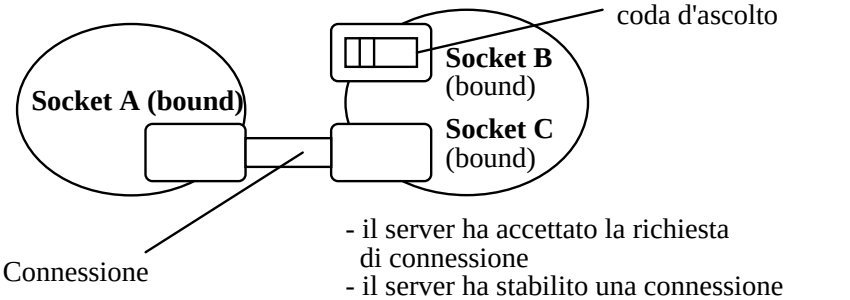
primitive diverse e comportamento differenziato all'inizio



- il client ha fatto una richiesta di connessione
- il server ha ricevuto una richiesta nella coda d'ascolto

SOCKET STREAM: CONNESSIONE

Una volta stabilita la **connessione la comunicazione** tra le entità interagenti **è del tutto simmetrica**



- 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

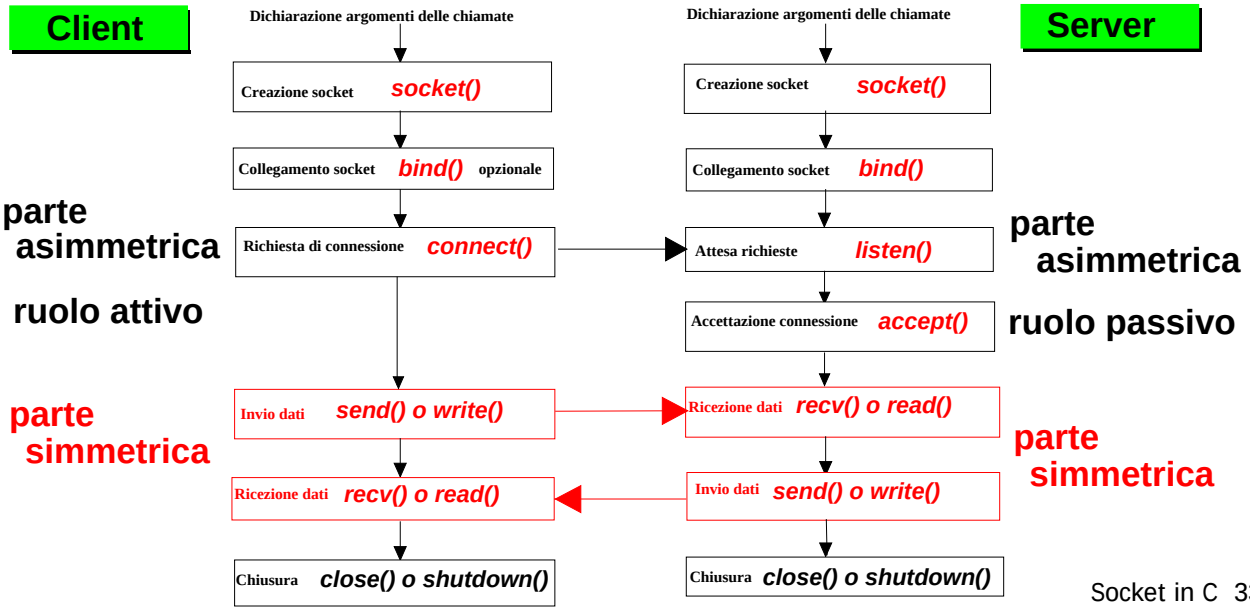
ruolo **attivo/passivo** di entrambi che possono inviare/ricevere informazioni

naturale ruolo **attivo del client** che comincia la comunicazione

primitive diverse e comportamento differenziato all'inizio

PROTOCOLLO SOCKET STREAM

Le **socket stream** sono usate con un **protocollo a sequenza differenziata di primitive e con ruoli diversi, per poi arrivare alla omogeneità dopo avere stabilito la connessione**



SOCKET STREAM CONNESSE

La CONNESSIONE, una volta stabilita, permane fino alla **chiusura di una delle due half-association**, ossia alla **decisione** di uno due entità interagenti (ancora scelta omogenea)

Sulla connessione i due interagenti possono sia mandare/ricevere byte (come messaggi utente) **`send` / `recv`,** ma anche possono fare azioni semplificate, e uniformi alle semplici sui file, **`read`** / **`write`**

```
recnum  =  read (s, buff, length);
sentnum =  write(s, buff, length);
```

Processi naive possono sfruttare le socket stream, una volta stabilita la connessione, per lavorare in modo trasparente in remoto

come fanno i filtri (o) che leggono da input e scrivono su output (vedi ridirezione, piping, ecc.) però nel distribuito

Processi più intelligenti possono sfruttano la piena potenzialità delle primitive delle socket

PRIMITIVE SOCKET: SOCKET()

Sia il cliente, sia il servitore devono per prima cosa dichiarare la risorsa di comunicazione a livello locale

Creazione di una socket specificando: {famiglia d'indirizzamento, tipo, protocollo di trasporto} ossia il nome logico locale

socket() fa solo creazione locale

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (af, type, protocol)
    int af, type, protocol;   /* parametri tutti costanti intere*/
```

protocollo di trasporto default 0, oppure uno specifico

socket() restituisce un socket descriptor o -1 se la creazione fallisce

```
int s;  /* il valore intero >= 0 corretto, negativo errore*/
s = socket (AF_INET,SOCK_STREAM,0);
```

socket deve poi legarsi al nome globale e visibile…

PRIMITIVE SOCKET: BIND()

Una socket deve essere legata al livello di nomi globali ossia visibili

La bind() collega la socket creata localmente alla porta e nodo globali

```
#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 a livello locale

addr ⇨ struttura con indirizzo di porta nome nodo stesso

addrlen ⇨ *la lunghezza di addr*

La bind crea half-association protocollo: < IP locale; porta locale;> per protocollo: <IP locale; porta locale; IP remoto; porta remota>

Ogni nodo che vede essere contattato e visibile deve fare la bind …

USO PRIMITIVE SOCKET: BIND()

Ogni server che deve essere raggiunto dai clienti deve fare la bind

I clienti possono fare il bind o meno, perché non hanno necessità di essere visibili in modo esterno, ma solo con meccanismo di risposta

il cliente può anche farsi assegnare una porta dal sistema …

Spesso i clienti non fanno bind ma viene invocata solo in modo implicito

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
int s, addrlen = sizeof(struc sockaddr_in);
struct sockaddr_in *addr; /* impostazione dei valori locali, con possibili assegnamenti di nodo e porta per l'indirizzo locale */
/* addr con sin_port a 0 richiede una assegnazione di un numero di porta libero in alcuni sistemi (come fare se non disponibile?)*/
s = socket (AF_INET,SOCK_STREAM,0);
res = bind (s, addr, addrlen);
if (res<0) /* errore e exit */  else /* procedi */ …
```


SOCKET CLIENT: CONNECT()

Il client deve creare una connessione prima di comunicare

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

s ⇨ socket descriptor

addr ⇨ puntatore al socket address remoto

addrlen ⇨ lunghezza di questa struttura

risultato ⇨ se negativo errore, se positivo restituisce il file descriptor

La primitiva `connect()` è una primitiva di comunicazione, sincrona, e termina quando la richiesta è accodata o in caso di errore rilevato

Al termine della connect la connessione è creata (almeno lato cliente)

protocollo: <IP locale; porta locale; IP remoto; porta remota>

SOCKET CLIENT: CONNECT()

La **connessione** è il veicolo per ogni comunicazione fatta attraverso il canale virtuale di comunicazione

La primitiva **`connect()`** è la controparte per il coordinamento iniziale del cliente che **ha la iniziativa** e si **attiva** per preparare le risorse

- La primitiva può avere tempi di completamento anche elevati
- La primitiva è una reale primitiva di **comunicazione remota**

Al completamento, in caso di errore (risultato <0), la motivazione del problema nel valore nella variabile **errno** (file /usr/include/errno.h)

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

In caso di **successo**, il client **considera** la **connessione stabilita** (?)

USO SOCKET CLIENT: CONNECT()

La primitiva **`connect()`** è anche capace di fare eseguire la bind

il sistema assegna al cliente la prima porta libera facendo una bind

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
int s, addrlen = sizeof(struc sockaddr_in);
struct sockaddr_in *peeraddr; /* impostazione dei valori del server, con uso di gethostbyname e gestservbyname eventualmente …*/
s = socket (AF_INET,SOCK_STREAM,0);
res = connect (s, addr, addrlen);
if (res<0) /* errore e exit */
else /* procedi avendo realizzato la connessione */ …
```

La **`connect()`** deposita la richiesta di connessione nella coda del servitore e non attua la **connessione con il server (sopra TX)** …

Azioni successive potrebbero fallire a causa di questa potenziale dissincronicità

SOCKET SERVER: LISTEN()

Il server deve creare una **coda per possibili richieste di servizio**

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

s ⇨ socket descriptor

backlog ⇨ numero di posizioni sulla coda di richieste (1-10, tipicamente **5**)

risultato ⇨ se negativo errore, se positivo restituisce il file descriptor

La primitiva `listen()` è una primitiva locale, senza attesa e fallisce solo se attuata su socket non adatte (no `socket()`, no `bind()`, ...)

Al termine della listen, la coda è disponibile per accettare richieste di connessione (`connect()`) nel numero specificato

- Una richiesta accodata fa terminare con successo la `connect()`
- Le richieste oltre la coda sono semplicemente scartate, la `connect()` fallisce dalla parte cliente; nessuna indicazione di nessun tipo al server

SOCKET SERVER: ACCEPT()

Il server deve **trattare ogni singola richiesta accodata con**

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

ls ⇨ socket descriptor

si ottengono informazioni sulla connessione tramite *addr e *addrlen

addr ⇨ indirizzo del socket address connesso

addrlen ⇨ la lunghezza espressa in byte

risultato ⇨ se negativo errore, se positivo restituisce una nuova socket connessa al cliente

La primitiva `accept()` è una primitiva locale, con attesa, e correlata alla comunicazione con il cliente: se ha successo produce la vera connessione, se fallisce, in caso di socket non adatte (no `socket()`, no `bind()`, no `listen()`, ...), non consente di proseguire

SOCKET SERVER: ACCEPT()

La **connessione** è il veicolo per ogni comunicazione

La primitiva **`accept()`** è la controparte per il coordinamento iniziale del server che **non ha la iniziativa** ma deve decidere autonomamente quando e se **attivare la reale** connessione

- La primitiva può avere tempi di completamento anche elevati
- La primitiva lavora in locale recuperando dati 'da **comunicazione remota**'

Al completamento, in caso di successo, la nuova socket:

- **ha una semantica di comunicazione come la vecchia**
- **ha la stessa porta della vecchia socket**
- **è connessa alla socket del client**

La vecchia socket di `listen()` per ricevere richieste è inalterata …

La **`accept()`** non offre la possibilità di filtrare le richieste che devono essere accettate tutte in ordine, una da ogni invocazione di **`accept()`**

`accept()` e **`connect()`** realizzano una sorta di **rendez-vous**

USO SOCKET CLIENT: ACCEPT()

La primitiva **`accept()`** è usata dal server per ogni nuova connessione

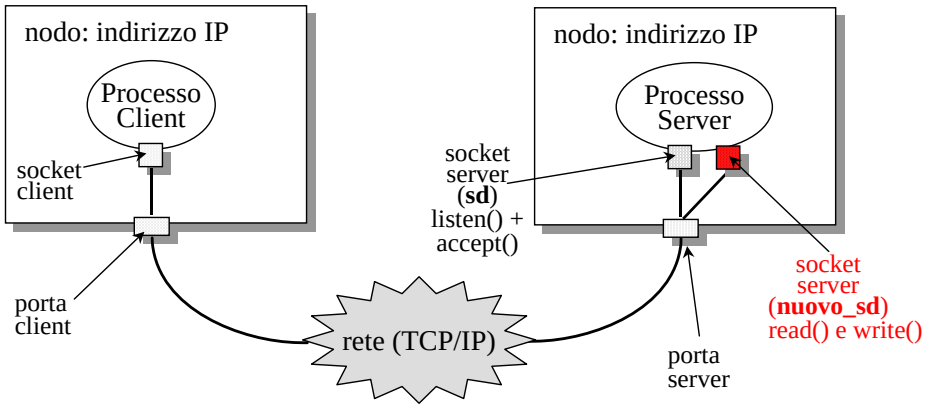
```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
int s, res, addrlen = sizeof(struc sockaddr_in);
struct sockaddr_in *myaddr; /* impostazione dei valori del server,
con uso di gethostbyname e gestservbyname eventualmente …*/
struct sockaddr_in *peeraddr; int peeraddrlen;
s = socket (AF_INET,SOCK_STREAM,0); …
res = bind (s, myaddr, addrlen);…
res = listen (s, backlog);…
ns = accept (s, peeraddr, &peeraddrlen); /* per riferimento */
if (ns<0) /* errore e exit */
else /* procedi avendo la nuova socket ns */ …
```

La **`accept()`** ottiene la nuova connessione visibile tramite ns

I parametri permettono di conoscere il cliente collegato

ACCEPT() e nuova SOCKET

La **`accept()`** attua la reale connessione dalla parte server e crea la nuova socket connessa



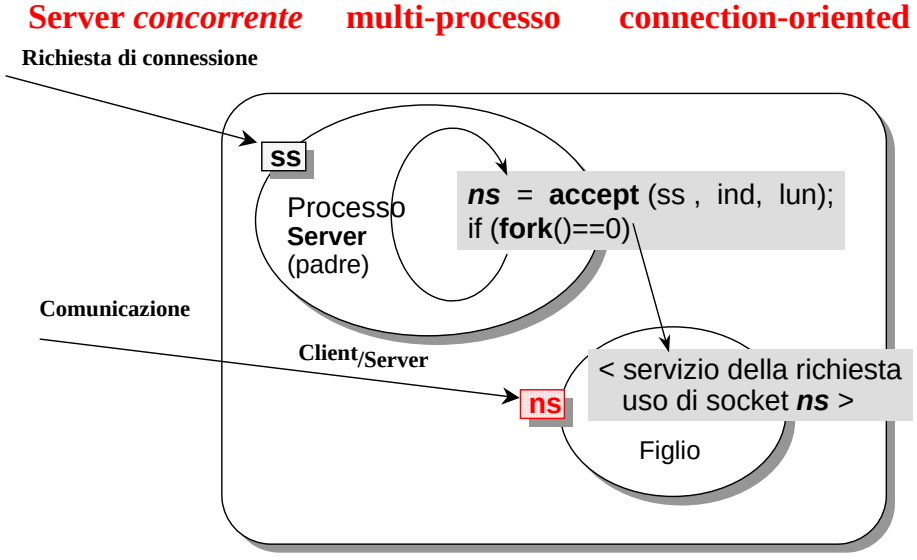
La nuova socket insiste sulla stessa porta della socket di **`bind()`** e si differenzia da questa per la funzione (non di **`listen()`** e **`accept()`**) e per il collegamento ad un cliente specifico

Lato server**,** la connessione è rappresentata da questa socket

SERVER SOCKET e PROCESSI

La nuova socket connessa permette di disaccoppiare le funzionalità del server, tra **accettazione** dei servizi e **connessioni** in atto

Ogni nuova socket rappresenta un **servizio separato e separabile**: in caso di server parallelo multiprocesso, la decomposizione è aiutata



COMUNICAZIONE sulla CONNESSIONE

La connessione può essere usata con **`send()`** e **`recv()`** per inviare e ricevere dati (byte) da ognuno dei due pari connessi

```
#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 all'area che contiene il messaggio (IN/OUT)

`len` ⇨ lunghezza del messaggio

`flags` ⇨ opzioni di comunicazione

risultato ⇨ **numero di byte realmente inviato/ricevuto**

flag send() / recv(): 0 normale / MSG_OOB per un messaggio out-of-band

flag recv(): MSG_PEEK per una lettura non distruttiva dallo stream

COMUNICAZIONE sulla CONNESSIONE

La connessione può essere usata con **`read()`** e **`write()`** per inviare e ricevere dati (byte) da ognuno dei due pari connessi: le primitive usuali sono del tutto usabili per socket connesse e la semantica è quella delle precedenti omologhe

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

`s` ⇨ socket descriptor

`msg` ⇨ puntatore all'area che contiene il messaggio (IN/OUT)

`len` ⇨ lunghezza del messaggio

risultato ⇨ **numero di byte realmente inviato/ricevuto**

La scrittura/send tende a consegnare alla driver i byte della primitiva

La lettura/read attende e legge almeno 1 byte disponibile, cercando di trasferire in spazio utente i byte arrivati sulla connessione

COMUNICAZIONE A STREAM

I messaggi sono comunicati ad ogni primitiva? **NO**

i dati sono bufferizzati dal protocollo TCP: non è detto che siano inviati subito ma raggruppati e inviati poi alla prima comunicazione 'vera' decisa dalla driver TCP

Soluzione messaggi di **lunghezza pari** al buffer

o **comandi espliciti** di flush del buffer

Come preservare i messaggi in ricezione?

ogni receive restituisce i dati preparati dalla driver locali: TCP a stream di byte non implementa marcatori di fine messaggio

Soluzioni **messaggi a lunghezza fissa**

Per messaggi a lunghezza variabile, si alternano un messaggio a lunghezza fissa e uno variabile in lunghezza, il primo contiene la lunghezza del secondo, ossia **uso di messaggi con indicatori espliciti di lunghezza** letti con due receive in successione

USO di STREAM

Ogni stream viene creato con successo tra due endpoint (e solo due per non avere interferenze) tipicamente su nodi diversi

i dati viaggiano nelle due direzioni e ogni endpoint li invia e li riceve dopo che hanno impegnato la comunicazione (banda e risorse)

Ogni endpoint ha una sorgente di input e un sink di output

Come sfruttare al meglio la rete?

ogni data che sia stato inviato deve essere ricevuto evitando di dovere scartare dati che sono arrivati ad un endpoint e non vengono accettati

Soluzione **segnali di fine flusso**

Su ogni flusso viaggiano dati fino alla **fine del file che segnala che non ci si devono aspettare più dati**

Ogni endpoint deve osservare un protocollo: deve leggere tutto l'input fino alla fine del flusso, e dalla sua direzione di flusso, **deve inviare una fine del flusso quando vuole terminare i dati in uscita**

CHIUSURA SOCKET: CLOSE()

Per non impegnare risorse non necessarie, si deve rilasciare ogni risorsa non usata con la **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

Il processo segnala al sistema operativo di rilasciare le risorse locali

alla driver di rilasciare le risorse remote

dopo un intervallo di tempo controllato da un'opzione avanzata (SO_LINGER)

```
int sd;
sd=socket(AF_INET,SOCK_STREAM,0);
...
close(sd); /* spesso non si controlla la eccezione  … ? */
```


EFFETTI della CLOSE()

La close è una primitiva a durata limitata (locale) per il chiamante e con un impatto sulla comunicazione e sul pari

Si può registrare un ritardo tra la chiusura applicativa e la reale deallocazione della memoria di supporto

Ogni socket a stream è associata ad un buffer di memoria sia per contenere i dati in uscita sia per i dati di ingresso

Alla chiusura, ogni messaggio nel buffer associato alla socket **in uscita** sulla connessione **deve essere spedito**, mentre ogni **dato in ingresso** ancora non ricevuto **viene perduto**

Solo dopo si può deallocare la memoria del buffer

Dopo la close (e la notifica), se il pari connesso alla socket chiusa legge dalla socket, ottiene **finefile**, se scrive ottiene un **segnale di connessione** non più esistente

La durata della close per la driver è poco controllata e può impegnare le risorse anche per molto tempo (minuti)

ALTERNATIVA alla CLOSE()

La primitiva **shutdown**() permette di terminare la connessione con una migliore gestione delle risorse

La primitiva **shutdown** produce una chiusura 'dolce' direzionale e tipicamente si chiude uno solo dei due versi di una connessione

```
int shutdown (s,how)  int s, how;
```

s ⇨ socket descriptor della socket

how ⇨ verso di chiusura

how = 0, SHUT_RD

non si ricevono più dati, ma si può trasmettere

send() del pari ritorna con -1 ed il processo riceve SIGPIPE

how = 1, SHUT_WR

si possono ricevere i dati dalla socket senza ulteriori trasmissioni di dati

L'altro pari collegato alla connessione alla lettura riceve end-of-file

how = 2, SHUT_RDWR entrambi gli effetti

USO SHUTDOWN()

La primitiva **shutdown**() viene tipicamente usata per una buona gestione delle azioni tra i due pari

Ognuno dei due gestisce il proprio verso di uscita e lo controlla

Se decide di finire usa una shutdown dell'output e segnala di non volere più trasmettere

Il pari legge fino alla fine del file e poi sa che non deve più occuparsi di input

Un pari con **shutdown(fd,1)** segnala la intenzione di non fare più invii nel suo verso

il pari, avvertito con una fine del file, può fare ancora uso del suo verso per molti invii, fino alla sua decisione di shutdown si attua una chiusura dolce

In modo del tutto simile, in Java esistono **shutdownInput()** e **shutdownOutput**() per le chiusure dolci direzionali

PROCESSI NAIVE

I processi molto semplici, **processi naive**, si comportano da **filtri** e leggono dallo standard input e scrivono sullo standard output…

possono essere **utilizzati in modo quasi trasparente nella comunicazione** (comandi UNIX sort, find, ecc)

Se il processo naif, usa **`write()`** e **`read()`** al posto delle `send()` e `recv()`allora può leggere da socket e scrivere su socket

Si deve preparare la attivazione secondo il protocollo usuale:

- Uso di fork() di un comando locale

  dopo aver creato tre socket che devono avere file descriptor stdin, stdout e stderr (0, 1, 2)

- Uso di exec() mantenendo i file descriptor

Il **processo locale naif** è così connesso automaticamente a dei **canali di comunicazione** e tramite questi a **processi remoti**

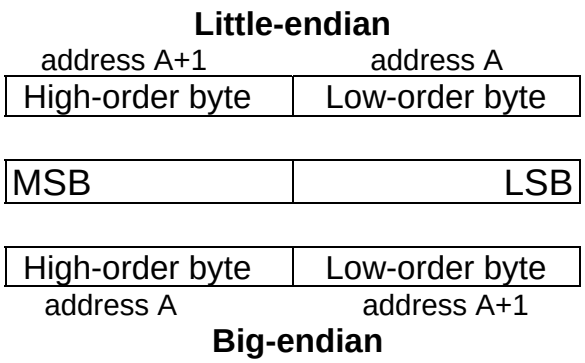
PRESENTAZIONE dei DATI

Gli interi sono composti da più byte e possono essere rappresentati in memoria secondo due modalità diverse di ordinamento.

2 byte per gli interi a 16 bit, 4 byte per interi a 32 bit

Little-endian Byte Order byte di ordine più basso nell'indirizzo iniziale

Big-endian Byte Order byte di ordine più alto nell'indirizzo iniziale



Network Byte Order (NBO)

ordinamento di byte per la rete

Protocolli Internet Big-endian

Host Byte Order (HBO)

non un unico ordinamento

Es. Intel Little-endian, Solaris Big-endian

FUNZIONI ACCESSORIE SOCKET

Funzioni accessorie da usare con socket con obiettivo di portabilità

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

Funzioni ausiliarie di 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 – net to host */

```
shortlocale =   ntohs(shortrete);
longlocale  =   ntohl(longrete);
```

/* trasforma un intero da formato interno in esterno – host to net */

```
shortrete   =   htons(shortlocale);
longrete    =   htonl(longlocale);
```


ALTRE FUNZIONI ACCESSORIE SOCKET

Manipolazione indirizzi IP per comodità

Funzioni per traslare da IP binario a 32 bit a stringa decimale a byte separato da punti (ascii: "123.34.56.78")

Conversione da notazione col punto a indirizzi IP a 32 bit

inet_addr() converte l'indirizzo dalla forma con punto decimale

```
indirizzo = inet_addr(stringa);
```

Prende una stringa con l'indirizzo in formato punto decimale e dà risultato l'indirizzo IP a 32 bit da utilizzare nelle primitive

Conversione da indirizzi IP a 32 bit a da notazione col punto

inet_ntoa() esegue la funzione inversa

```
stringa = inet_ntoa(indirizzo);
```

Prende un indirizzo indirizzo IP a 32 bit (cioè un long integer)

E fornisce come risultato una stringa di caratteri con indirizzo in forma con punto

ALTRE FUNZIONI ACCESSORIE

In C tendiamo a lavorare con stringhe, ossia con funzioni che assumono aree di memoria (stringhe) con il terminatore zero binario (tutti 0 in un byte) o fine stringa

Per fare operazioni di confronto, di copia e di set

Gli indirizzi internet non sono stringhe (non hanno terminatore), ma le driver spesso assumono di avere zeri binari

FUNZIONI che non richiedono fine stringa (ma assumono solo blocchi di byte senza terminatore) per lavorare su indirizzi e fare operazioni di set, copia, confronto

```
bcmp   (addr1, addr2, length)     /* funzioni  BSD */
bcopy  (addr1, addr2, length)
bzero  (addr1, length)
memset  (addr1,  char,  length)  /* funzioni  System V */
memcpy  (addr1, addr2, length)
memcmp  (addr1, addr2, length)
```


API SOCKET

Molte sono le primitive per le socket, e sono tutte SINCRONE

Chiamata	Significato
socket( )	Crea un descrittore da usare nelle comunicazione di rete
connect( )	Connette la socket a una remota
write( )	Spedisce i dati attraverso la connessione
read( )	Riceve i dati dalla connessione
close( )	Termina la comunicazione e dealloca la socket
bind( )	Lega la socket con l'endpoint locale
listen( )	Socket in modo passivo e predispone la lunghezza della coda per le connessioni
accept( )	Accetta le connessioni in arrivo
recv( )	Riceve i dati in arrivo dalla connessione
recvmes( )	Riceve i messaggi in arrivo dalla connessione
recvfrom( )	Riceve i datagrammi in arrivo da una destinazione specificata
send( )	Spedisce i dati attraverso la connessione
sendmsg( )	Spedisce messaggi attraverso la connessione
sendto( )	Spedisce i datagrammi verso una destinazione specificata

Quali primitive possono avere una elevata durata? In bold

API SOCKET

Seguono ancora primitive per le socket

Chiamata	Significato
`shutdown( )`	Termina una connessione TCP in una o in entrambe le direzioni
`getsockname( )`	Permette di ottenere la socket locale legata dal kernel (vedi parametri *socket*, *sockaddr*, *length*)
`getpeername( )`	Permette di ottenere l'indirizzo del pari remoto una volta stabilita la connessione (vedi parametri *socket*, *sockaddr*, *length*)
`getsockopt( )`	Ottiene le opzioni settate per la socket
`setsockopt( )`	Cambia le opzioni per una socket
`perror()`	Invia un messaggio di errore in base a errno (stringa su stderr)
`syslog()`	Invia un messaggio di errore sul file di log (vedi parametri *priority*, *message*, *params*)

Si sperimentino le primitive non note (e note)

ESEMPIO SOCKET STREAM

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 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

RCP: LATO CLIENT - INIZIO e SET

```
…
/* Prepara indirizzo remoto connessione */
server_address.sin_family = AF_INET;
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; /* copia indirizzo server */
server_address.sin_port = htons(12345);
sd=socket(AF_INET, SOCK_STREAM, 0);
/* non facciamo la bind, ma la richiediamo tramite la connect */
if(connect(sd, (struct sockaddr *)&server_address,
   sizeof(struct sockaddr))<0)
 {perror("Errore in connect"); exit(1);}
/* abbiamo il collegamento con il server tramite la connessione  (?) */
```


RCP: LATO CLIENT - PROTOCOLLO

```
if (write(sd, argv[2], strlen(argv[2])+1)<0)
 { perror("write"); exit(1);} /* invio nome del file */
if ((nread=read(sd, buff, 1))<0)
 { perror("read"); exit(1);} /* ricezione risposta server */
if(buff[0]=='S') /* se il file non esiste, si copia */
{if((fd=open(argv[2],O_RDONLY))<0){perror("open");exit(1);
}
 while((nread=read(fd, buff, DIM_BUFF))>0)
          write(sd,buff,nread);
   /* ciclo letture e scritture su socket */
 close(sd); /* ho spedito il file */  printf("File spedito\n");
}
else  /* non si sovrascrive il file */
{printf("File esiste, termino\n"); close(sd);}
…
```

RCP: LATO SERVER - INIZIO e SET

Nel server osserviamo la sequenza regolare di primitive

…

```
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 la socket in socket passiva d'ascolto e di servizio */

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

…

A questo punto sono possibili progetti differenziati, non visibili al cliente

server sequenziali o server concorrenti

RCP: SERVER SEQUENZIALE

… **/* per la read vedi anche esempi successivi */**

```
for(;;) {   /* ciclo di servizio */
ns=accept(sd,(struct sockaddr *)&client_address, &fromlen);
read(ns, buff, DIM_BUFF); printf("server legge %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  /* ciclo di lettura dalla socket e scrittura su file */
{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); close (fd);  /* chiusura di file e connessione */
} exit(0);  /* il server è di nuovo disponibile ad un servizio */
```

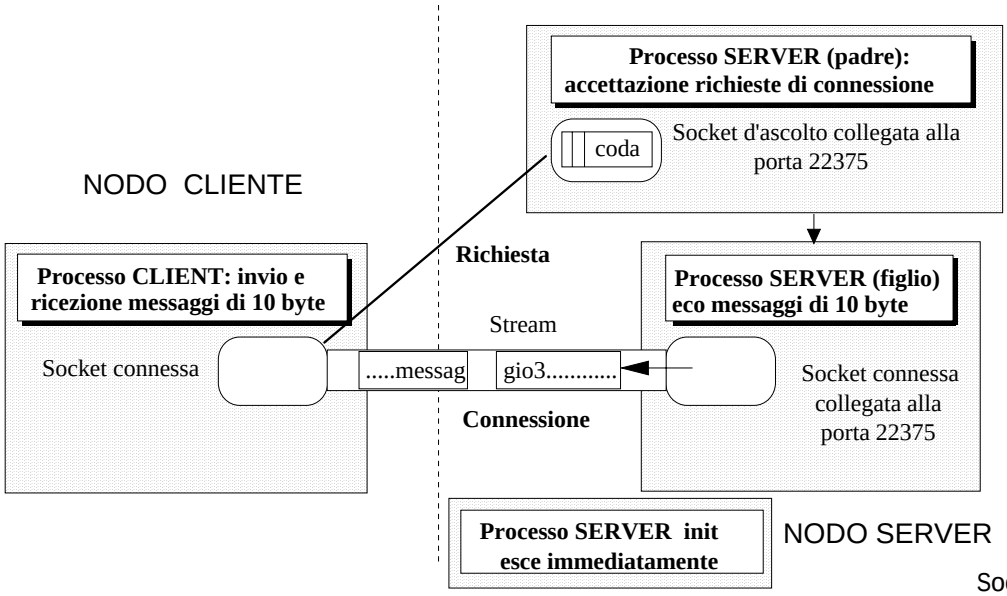
…

RCP: SERVER MULTIPROCESSO

```
for(;;) {
ns=accept(sd,(struct sockaddr *)&client_address,&fromlen);
if (fork()==0) /* figlio */
{close(sd);read(ns, buff, DIM_BUFF); /* chiude socket servizio */
 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    /* facciamo la copia del file, leggendo dalla connessione */
{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); close(fd); exit(0); }
close(ns); wait(&status); } /* padre */
/* attenzione: si sequenzializza ... Cosa bisognerebbe fare? */
```


ESEMPIO SERVIZIO REALE

Si vuole realizzare un **server parallelo** che sia attivato solo col suo nome e produca un **demone di servizio** che attiva una connessione per ogni cliente e che risponde ai messaggi del cliente sulla connessione (echo) fino alla fine delle richieste del cliente



ECHO: LATO CLIENT - INIZIO

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdio.h>
#include <netdb.h>
char *ctime();   /* dichiarazione routine di formattazione dell'orario */
int ricevi ();   /* dichiarazione routine di ricezione di un messaggio*/
int s;                  /* socket descriptor del cliente */
struct hostent *hp;     /* puntatore alle informazioni host remoto */
long timevar;           /* contiene il risultato 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]; /* messaggi di 10 bytes */
if (argc != 3)
{ fprintf(stderr, "Uso: %s <host remoto> <nric>\n", argv[0]); exit(1)}
```


ECHO: LATO CLIENT - SET

```
/* 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  */
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); }   /* trovato il nome IP fisico */
peeraddr_in.sin_addr.s_addr =
   ((struct in_addr *)(hp->h_addr))->s_addr;
/* non si usa la htonl dopo la gethostbyname: la si provi in diversi ambienti */
```

LATO CLIENT - ANCORA SET

```
/* definisce il numero di porta senza la chiamata getservbyname()
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);*/ /* invece */
peeraddr_in.sin_port = htons(22375);
/* numero di porta trasformato nel formato network via htons(). In
architetture in cui i due formati coincidono si ottiene maggiore efficienza */
s = socket (AF_INET,SOCK_STREAM,0); /* creazione della socket */
if (s == -1) { perror(argv[0]); /* controllo errore */
fprintf(stderr, "%s: non posso creare la  socket\n", argv[0]); exit(1); }
```


LATO CLIENT - PRIMITIVE

```
/* No bind: la porta del client assegnato dal sistema. Il server lo vede alla
richiesta di connessione; il processo client lo ricava con getsocketname() */
if(connect (s, &peeraddr_in, sizeof(struct sockaddr_in))== -1)
{ perror(argv[0]); /* tentativo di connessione al server remoto */
  fprintf(stderr,"%s: impossibile connettersi con server\n", argv[0]);
  exit(1); }   /* altrimenti lo stream è stato ottenuto (!?) */
addrlen = sizeof(struct sockaddr_in); /* dati connessione locale */
if (getsockname (s, &myaddr_in, &addrlen) == -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], myaddr_in.sin_port, ctime(&timevar));
/* Il numero di porta espresso in byte senza bisogno di conversione */
sleep(5); /* attesa che simula un'elaborazione al client */
```

CLIENT - INVIO DATI CONNESSIONE

/* **NON C'È PRESENTAZIONE DEI DATI**

Invio di messaggi al processo server mandando un insieme di interi successivi

`*buf=i` pone i primi 4 byte di 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 */

```
for (i=1; i<= atoi(argv[2]); i++)
{/* invio di tutti i messaggi nel numero specificato dal secondo argomento */
 *buf = htonl(i);  /* i messaggi sono solo gli interi successivi */
 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);
 }
}
```

/* i messaggi sono mandati senza aspettare alcuna risposta !!!! */

CLIENT - RICEZIONE DATI

/* **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*/

```
/* Per ogni messaggio ricevuto, diamo un'indicazione locale */
while (ricevi (s, buf, 10))
 printf("Ricevuta la risposta n. %d\n", ntohl( *buf));
/* Messaggio per indicare il completamento del programma */
time(&timevar); printf("Terminato alle %s", ctime(&timevar));
}
```

FUNZIONE DI RICEZIONE MESSAGGI

```
int ricevi (s, buf, n)  int s; char * buf; int n;
{int i, j; /* ricezione di un messaggio di specificata lunghezza */
if ((i = recv (s, buf, n, 0)) != n && != 0) {
if (i == -1) { perror(argv[0]);
 fprintf(stderr,"%s: errore in lettura\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 in lettura\n", argv[0]); exit(1); }
 i += j; if (j == 0) break; }
} /* si assume che tutti i byte arrivino … se si verifica il fine file si esce */
return n; }
```

/* Il ciclo interno verifica che la recv() non ritorni un messaggio più corto di quello atteso (n byte)

La recv ritorna appena vi sono dati e non attende tutti i dati richiesti

Il loop interno di recv garantisce la ricezione fino al byte richiesto e permette alla recv successiva di partire sempre dall'inizio di una risposta */

RICEZIONE MESSAGGI

Nelle applicazioni Internet è molto comune trovare funzioni come quella appena vista di ricezione

Dobbiamo ricordare che la **ricezione considera un successo un qualunque numero di byte ricevuto** (anche 1) e ne segnala il numero nel risultato (a meno di opzioni: vedi low watermark)

Per evitare tutti i problemi, dobbiamo **fare ricezioni / letture ripetute**

Esiste il modo di cambiare il comportamento della receive intervenendo sui low-watermark (vedi opzioni socket)

In ogni caso, in ricezione dobbiamo sempre verificare la dimensione del messaggio, se nota, o attuare un protocollo per conoscerla durante la esecuzione, per aspettare l'intero messaggio significativo

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**

LATO SERVER - INIZIO

```
#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 ricevere richieste e  la listen()*/
struct hostent *hp;       /* puntatore all' host remoto */
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; /* Azzera le strutture degli indirizzi */
memset ((char *)&myaddr_in, 0, sizeof(structsockaddr_in));
memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));
```


LATO SERVER - SET

```
/* 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 s uno specifico indirizzo di rete ⇨ maggiore portabilità del codice
Convenzione per considerare server su nodi connessi a più reti,
consentendo di attendere richieste da ogni rete (e indirizzo relativo) */
/* assegna IP generico e  numero di porta */
myaddr_in.sin_addr.s_addr = INADDR_ANY;
myaddr_in.sin_port = 22375;
```

Il server deve sempre garantirsi un assegnamento di porta e IP che siano quelli con cui è conosciuto ai clienti

Deve quindi seguire la sequenza intera delle primitive

SERVER - PRIMITIVE di CONNESSIONE

```
/* Crea la socket d'ascolto delle richieste */
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 (tipicamente al 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 della socket principale completata
Il programma deve creare un processo daemon ed uscire dopo
avere lasciato il processo a garantire il servizio */
```


SERVER - DAEMON di SERVIZIO

Il processo prima di uscire deve preparare le condizioni per il figlio 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)

Poi si genera un figlio che deve lavorare (il daemon) e si termina …

Il daemon genera un figlio per ogni richiesta di connessione

```
setsid();          /* Unix System V: nuovo gruppo processi */
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: dopo */
default:   exit(0);   /* processo INIZIALE */
} /* Il processo iniziale esce e lascia libera la console utente */
}
```

DAEMON di SERVIZIO

```
case 0:  /* FIGLIO e schema di processo DEMONE: qui */
```

/* 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);
```

/* si ignora il segnale SIGCLD (SIG_IGN) per non mantenere processi zombi per ogni servizio eseguito */

```
signal(SIGCLD, SIG_IGN);
```

/* Il demone entra in un loop e, ad ogni richiesta, crea un processo figlio per servire la chiamata */

```
for (;;) { addrlen = sizeof(struct sockaddr_in);
```

/* **accept()** bloccante in attesa di richieste di connessione

Dopo la accept, il daemon ottiene dalla accept l'indirizzo del chiamante e la sua lunghezza, oltre che un nuovo socket descriptor per la connessione */

```
s = accept (ls, &peeraddr_in, &addrlen);
if ( s == -1) exit(1);
```


DAEMON - SERVIZIO

```
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: /* successo in generazione demone  */
        close(s);
```

/* 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 */

```
                        } /* fine switch */
} /* for  fine ciclo del daemon*/
```

Resta il comportamento del figlio del daemon in **server()**

PROCESSO di SERVIZIO

procedura SERVER: routine eseguita dal processo figlio del daemon è qui che si gestisce la connessione: si ricevono i pacchetti dal processo client, si elaborano, e si ritornano i risultati al mittente; inoltre si scrivono alcuni dati sullo stdout locale

```
char *inet_ntoa(); /* routine formato indirizzo Internet */
char *ctime();     /* routine di formato dell'orario ottenuto da time () */
int ricevi ();
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;
  close (ls); /* Chiude la socket d'ascolto ereditata dal daemon */
```

Il server 'vero' deve leggere tutti i dati secondo il formato predefinito e rispondere dopo averli elaborati

PROCESSO di SERVIZIO

```
/* Cerca le informazioni relative all'host connesso mediante il suo indirizzo
Internet  usando la gethostbyaddr() per rendere leggibili gli indirizzi */
hp = gethostbyaddr ((char *)&(ntohl(peeraddr_in.sin_addr) ,
     sizeof (struct in_addr), peeraddr_in.sin_family);
if (hp == NULL)  hostname =
     inet_ntoa(ntohl(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));
```

CICLO di SERVIZIO

```
/* Loop  di ricezione messaggi del cliente
Uscita alla ricezione dell'evento di shutdown, cioè alla fine del file */
while (ricevi (s, buf, 10) )
{reqcnt++; /* Incrementa il contatore di messaggi  */
 sleep(1); /* Attesa per simulare l'elaborazione dei dati */
 if(send(s,buf,10,0)!=10) /* Invio risposta per ogni messaggio*/
 {printf("Connessione a %s abortita in 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 */
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)); }
```


ADDENDUM PRIMITIVE

In alcuni kernel, le primitive sospensive hanno un qualche problema in caso di interruzione con segnali, dovuto a interferenza tra spazio kernel e utente

Una primitiva sospensiva interrotta da un segnale deve essere riattivata dall'inizio usando uno schema come il seguente

(**errno == EINTR** verifica che si sia stata una interruzione da segnale)

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

 }
```

ATTESE MULTIPLE

Le primitive su socket bloccano il processo che le esegue…

In caso di possibile ricezione da sorgenti multiple (eventi multipli che possono sbloccare un processo) è necessaria la possibilità di attendere contemporaneamente su più eventi di I/O legati a più socket (o file)

- La gestione delle socket ha portato ad una primitiva legata alla gestione di più eventi all'interno di uno stesso processo e al blocco imposto dalle primitive sincrone
- Le operazioni bloccanti (lettura) o con attese pregiudicano il servizio di altre ad un processo server (con molti servizi da svolgere) che potrebbe sospendersi su una primitiva e non potere servire altre richieste su socket diverse

Risposta in C con PRIMITIVA **`select()`**

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

SELECT()

Primitiva per attesa multipla globale sincrona o con durata massima
`select()` primitiva con time-out intrinseco

- Azioni di comunicazione potenzialmente sospensive
- Lettura *accept, receive (in tutte le forme), eventi di chiusura*
- Scrittura *connect, send (in tutte le forme), eventi di chiusura*
- Eventi anomali *dati out-of-band, eventi di chiusura*

La select permette di accorgersi di eventi relativi a socket rilevanti

```
#include <time.h>
int select (nfds, readfds, writefds, exceptfds, timeout)
size_t nfds ;   /* numero massimo di eventi attesi, inteso come limite
      superiore al numero delle socket e per indicare quali bit considerare*/
int *readfds,  *writefds, *exceptfds;
/* tre maschere per gli eventi  o anche null per indicare non interesse */
const struct timeval * timeout;
/* time out  massimo o anche null se attesa indefinita */
```

EVENTI per la SELECT()

La `select()` invocata sospende il processo fino al primo evento o al timeout (se si specifica il timeout) o attende il primo evento (sincrona con il primo)

Eventi di lettura: rendono possibile e non bloccante un'operazione

- 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 o errore

Eventi di scrittura: segnalano un'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) o errore

Eventi e condizioni eccezionali, segnalano errore o urgenza

- arrivo di dati out-of-band,
- inutilizzabilità della socket, close() o shutdown()

INVOCAZIONE della SELECT()

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

La **`select()`** invocata richiede al sistema operativo di passare delle informazioni sullo stato interno di comunicazione

All'invocazione segnala nelle maschere gli eventi di interesse

Al completamento, restituisce il numero di eventi occorsi e indica quali con le maschere (parametri di ingresso/uscita) maschere

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

Con azione sospensiva bloccante sincrona, il massimo intervallo di attesa 31 giorni

Con timeout quanto specificato nel parametro

Con azione non bloccante, si specifica zero come valore nel timeout e si lavora a polling dei canali

MASCHERE per la SELECT()

La chiamata esamina gli eventi per i file descriptor specificati nelle tre maschere (valore ingresso bit ad 1) relative ai tre tipi

I bit della maschera corrispondono ai file descriptor a partire dal fd 0 fino al fd dato come primo parametro

Prendendo come maschera la seguente

9	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*posizione file descriptor*
1	0	**1**	0	**1**	**1**	0	0	0	0	maschera ingresso
0	0	**1**	0	**1**	**0**	0	0	0	0	maschera uscita

si esaminano solo i file descriptor il cui bit è ad 1, ossia per socket 4,5,7,9

```
fds[(f / BITS_PER_INT)] & (1<<(f % BITS_PER_INT))
```

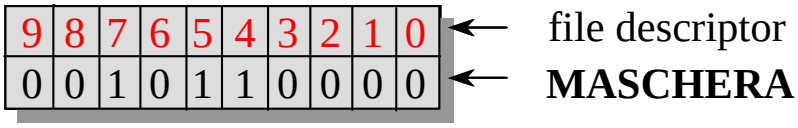
Al ritorno della chiamata le tre maschere sono modificate in relazione agli eventi per i corrispondenti file descriptor

1 se evento verificato, 0 altrimenti, ossia gli eventi 4,5,7 si sono verificati

Anche **un solo evento** di lettura/scrittura/anomalo **termina la primitiva select**, dopo cui si possono o trattare tutti o uno solo, anche selezionando un qualunque ordine del servizio

OPERAZIONI sulle MASCHERE

Per facilitare la usabilità si introducono operazioni sulle maschere, che sono array di bit, di dimensione diversa per le diverse architetture




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

FD_SET include la posizione particolare fd in fdset ad 1

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

FD_CLR rimuove fd dal set fdset (reset della posizione)

```
int  FD_ISSET(int fd, fd_set &fdset);
```

FD_ISSET restituisce un predicato che determina se la posizione di fd fa parte del set fdset, o non ne fa parte (0 ed 1)

```
void FD_ZERO(fd_set &fdset);
```

FD_ZERO inizializza l'insieme di descrittori a zero

MACRO sulle MASCHERE

Le operazioni sono macro C definite in /usr/include/stdio.h

```
typedef long fd_mask;    /* un certo numero di 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    /* definizione della maschera */
{fd_mask fds_bits[howmany(FD_SETSIZE,NFDBITS)];}fd_set;
/* la dimensione dell'array dipende dal sistema operativo: da cosa? */
#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)))
```


ESEMPIO di SELECT()

Gestione di socket e select (maschere) …

```
#include <stdio.h>
do_select(s) int s;   /* socket descriptor di interesse */
{struct fd_set read_mask, write_mask; int nfds, nfd;
for (;;) {/* ciclo infinito */
/* azzera le maschere  e set posizione*/
FD_ZERO(&read_mask); FD_SET(s,&read_mask);
FD_ZERO(&write_mask); FD_SET(s,&write_mask); nfds=s+1;
nfd=select(nfds,&read_mask,&write_mask, NULL,
                          (struct timeval*)0);
if (nfd==-1) /* -1 per errore, anche 0 per  timeout*/
{perror("select: condizione inattesa"); exit(1);}
/* ricerca successiva del file descriptor nella maschera e trattamento */
if (FD_ISSET(s,&read_mask))  do_read(s);
if (FD_ISSET(s,&write_mask)) do_write(s);
}}
```

SERVER CONCORRENTE a CONNESSIONE

Si vuole progettare un server concorrent monoprocesso che possa accettare servizi molteplici tutti con necessità di una connessione con il cliente

senza incorrere nei ritardi di un server sequenziale

Il server deve potere portare avanti le attività disponibili senza attese non necessarie dovute agli strumenti o primitive

È necessario considerare che tutte le socket sono legate alla stessa porta, ed avendo un unico processo, i numeri delle socket sono facilmente prevedibili

- Tipicamente 0,1,2 sono impegnati per standard poi si sale
- Alla chiusura, i file descriptor liberati sono occupati in sequenza dal basso

Il server comincia ad occupare il **primo file descriptor per la socket di listen** (ricezione richieste connessione), poi cresce con le altre ad ogni richiesta …

Toglie un file descriptor per ogni chiusura e rioccupa dai valori bassi

SERVER CONCORRENTE - supporto

```
#define LEN 100
typedef struct { long len; char name[LEN]; } Request ;

typedef int HANDLE;
/* funzione  di ricezione di una intera richiesta: ciclo letture fino ad ottenere l'intera richiesta */
int recv_request (HANDLE h, Request *req)
{ 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; }
  req->len = ntohl (req->len); /* Decodifica len */
  return r_bytes;
}
```

SERVER CONCORRENTE - supporto

```
typedef struct { long value, errno; /* 0 successo */ } Response ;
/* funzione  di invio di una intera richiesta: ciclo di scritture fino ad ottenere l'invio dell'intera richiesta */
int send_response (HANDLE h, long value)
{Response res; size_t w_bytes;
 size_t len = sizeof res;
/* il risultato prevede i campi errore e valore */
  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;
}
```


SERVER - PREPARAZIONE

```
int main   (int argc, char *argv[])
{
/* porta di listen per connessioni */
 u_short port  = argc > 1 ? atoi(argv[1]) : 10000;
/* trattamento iniziale della socket */
 HANDLE listener = create_server_endpoint(port);
/* numero corrente di possibili socket da verificare */
 HANDLE maxhp1 = listener + 1;
fd_set read_hs, temp_hs;
/* due maschere, 1 di stato stabile e 1 di supporto che si possa usare */
FD_ZERO(&read_hs);
FD_SET(listener, &read_hs);
temp_hs = read_hs;
/* ciclo di select per il processo servitore…*/
}
```

SERVER - CICLO

```
for (;;) /* ciclo di select per il processo servitore…*/
{ HANDLE h; /* verifica delle richieste presenti */
  select (maxhp1, &temp_hs, 0, 0, 0);
  for (h = listener + 1; h < maxhp1; h++)
   { if (FD_ISSET(h, &temp_hs)) /* richieste sulle connessioni */
      /* per ogni dato su connessione, trattamento in handle() */
      if (handle (h) == 0)
      /* se è il caso di chiusura della connessione da parte del cliente */
        { FD_CLR(h, &read_hs);  close(h); }
   }
  if (FD_ISSET(listener, &temp_hs)){ /* nuova connessione */
    h = accept (listener, 0, 0); FD_SET(h, &read_hs);
    if (maxhp1 < = h) maxhp1 = h + 1;   }
  temp_hs=read_hs; }
}
```


SERVER - INIT

```
/* funzione di preparazione della socket di listen sulla porta */
HANDLE create_server_endpoint (u_short port)
{struct sockaddr_in addr;
 HANDLE h; /* file desriptor della socket iniziale */
  h = socket (PF_INET, SOCK_STREAM, 0);
  /* set di indirizzo per  il server  */
  memset ((void *) &addr, 0, sizeof addr);
  addr.sin_family = AF_INET;
  addr.sin_port =  ntohs(port);
  addr.sin_addr.s_addr = INADDR_ANY;
  /* usuali primitive da parte server  */
  bind (h, (struct sockaddr *) &addr, sizeof addr);
  listen (h, 5);
  return h;
}
```

SERVER – AZIONI SPECIFICHE

/* funzione di preparazione della socket di listen sulla porta */

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

/* per ogni possibile evento da parte di un cliente connesso, si esegue la funzione handle

questa funzione riceve la richiesta (letture senza sospensione)

attua l'azione e invia la risposta

a richiesta nulla, il cliente ha chiuso la connessione ⇨ si chiude*/

```
long handle (HANDLE h)
{ struct Request req; long value;
 if (recv_request (h, &req) <= 0)   return 0;
 value = action (&req); /* azione */
 return send_response (h, value);
}
```


SERVER MULTIFUNZIONE

Spesso è significativo avere un **unico servitore per più servizi** come 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 deve riconoscere le richieste ed anche attivare il servizio stesso

Problemi:

Il server può diventare il collo di bottiglia del sistema

Necessità di decisioni rapide e leggere

Vedi **BSD UNIX inetd Internet daemon (/etc/services)**

inetd svolge alcuni servizi sono svolti in modo diretto (servizi interni) ed altri li delega a processi creati su richiesta

definito un linguaggio di configurazione per specificare comportamento

CONFIGURAZIONE INETD

```
# @(#)inetd.conf 1.24 SMI Configuration file for inetd(8).
# 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
# Shell, login, exec, comsat and talk are BSD protocols.
Shell stream tcp  nowait   root /usr/etc/in.rshd  in.rshd  …
talk  dgram  udp  wait    root  /usr/etc/in.talkd in.talkd
# Finger, systat and netstat are usually disabled for security
# Time service is used for clock syncronization.
time  stream tcp nowait    root   internal
time  dgram  udp  wait     root   internal
```


CONFIGURAZIONE INETD

```
# Echo, discard, daytime, and chargen are used 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>
# 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
```

CLIENTE sequenziale o parallelo

Si può gestire la concorrenza anche dalla parte del cliente

a) soluzione concorrente

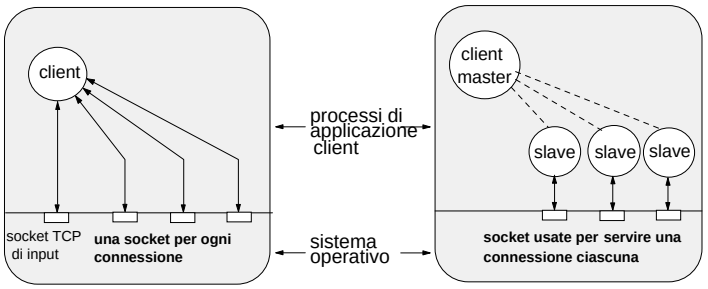
possibilità che il cliente unico gestisca più interazioni con necessità di gestione dell'asincronismo

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

Questo permette anche di interagire con più server contemporaneamente ad esempio con multicast



OPZIONI per le SOCKET

`getsockopt()` `setsockopt()`

funzioni di utilità per configurare socket, cioè leggere e variare le modalità di utilizzo delle socket

```
#include <sys/types.h>
#include <sys/socket.h>
char  optval = 1; /* tipicamente il valore del campo vale o 1 o 0 */
int getsockopt (s, level, optname, &optval, optlen)
   int s, level, optname, optval, *optlen;
int setsockopt (s, level, optname, &optval, optlen)
   int s, level, optname, optval, 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 del quarto argomento

POSSIBILI OPZIONI per le SOCKET

Si riescono a cambiare molti comportamenti anche in modo molto granulare e con molto controllo

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_RCVLOWAT	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_PROTOTYPE	setta tipo di protocollo

POSSIBILI OPZIONI per le SOCKET

Timeout per operazioni ⇨ Opzioni SO_SNDTIMEO e SO_RCVTIMEO

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

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

Dimensioni buffer di trasmissione/ricezione ⇨ SO_SNDBUF e SO_RCVBUF

Intervento sulla **dimensione del buffer di trasmissione o ricezione** di socket Si interviene sulle comunicazioni, eliminando attese anche per messaggi di dimensioni elevate (massima dimensione possibile 65535 byte)

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

Controllo periodico della connessione

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

RIUTILIZZO INDIRIZZI SOCKET

Opzione SO_REUSEADDR modifica comportamento della bind()

Il sistema tende a non ammettere più di un utilizzo di un indirizzo locale e uno successivo viene bloccato (con fallimento)

con l'opzione, si convalida l'indirizzo di una socket senza controllo della unicità di associazione

In particolare è utile in caso di server che devono potere essere riavviati e che devono essere operativi immediatamente

Si ricordi che la porta rimane impegnata dopo una close per anche molte decine di secondi ce senza l'opzione dovremmo aspettare per il riavvio

Il processo che deve eventualmente agganciarsi alla porta per il riavvio, senza la opzione potrebbe incorrere in un fallimento fino alla liberazione della porta (e della memoria corrispondente)

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


DURATA CLOSE su SOCKET

Opzione SO_LINGER modifica comportamento della close()

Il sistema tende a mantenere la memoria in uscita dopo la close anche per un lungo intervallo

Con l'opzione, si prevede la struttura linger da /usr/include/sys/socket.h:

```
struct linger { int l_onoff; int l_linger;  /* attesa in sec */ }
```

A default, disconnessione impegnativa in risorse, l_onoff == 0

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

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 close() completa dopo il tempo in secondi specificato dal linger e (si spera) dopo la trasmissione di tutti i dati nel buffer

MODALITÀ PRIMITIVE SOCKET

Sono di molto interesse modi che non siano sincroni ma lavorino senza nessuna attesa correlata alla comunicazione

Socket asincrone con uso di primitive **`ioctl`** o **`fcntl`** e opzioni

Le socket asincrone permettono operazioni senza attesa, ma al completamento tipicamente l'utente viene avvisato con un segnale ad hoc

- SIGIO segnala un cambiamento di stato della socket (per l'arrivo di dati)
- SIGIO ignorato dai processi che non hanno definito un gestore

Gestione della socket e del segnale, ad esempio per la consegna dei dati

SIGIO socket asincrona con attributo **FIOASYNC** con primitiva ioctl()

```
#include <sys/ioctl.h>
int ioctl  (int filedesc, int request, ... /* args */)
```

`filedescr` ⇨ file descriptor

`request` ⇨ tipo di attributo da assegnare

`poi` ⇨ valori da assegnare all'attributo

A chi si deve consegnare il segnale, in un gruppo di processi???

MODALITÀ CONSEGNA SEGNALI

Per dare strumenti con la necessaria visibilità a volte si devono tenere in conto altre caratteristiche

SIGIO a chi deve essere consegnato in un gruppo di processi?

Per la consegna di SIGIO, primitiva **`ioctl()`** con attributo **SIOCSPGRP**

parametro process group del processo alla socket asincrona.

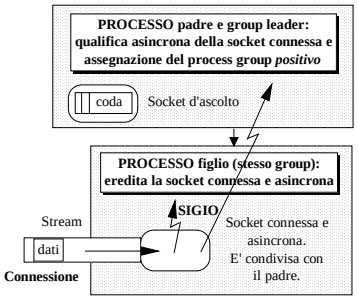
```
int ioctl (filedescr,  SIOCSPGRP, &flag)
```

`flag` valore negativo ⇨

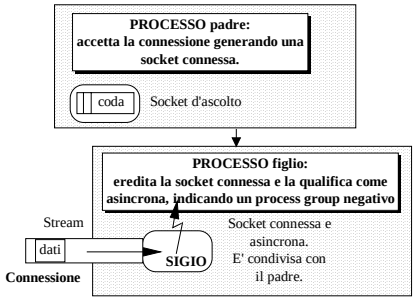
segnale solo al processo con pid uguale al valore negato

`flag` valore positivo ⇨ segnale arriva a tutti i processi del process group

valore positivo



valore negativo



ESEMPIO ASINCRONO

```
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); /* aggancio del gestore segnale */
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);     }
```

In questo caso si consegna il segnale al solo processo che ne ha fatto richiesta e si attiva l'handler solo per lui alla occorrenza del segnale di SIGIO

ALTRI ESEMPI ASINCRONO

Le azioni sono fortemente dipendenti dal kernel e poco standard ☹

Per la maggior parte dei sistemi Linux (ma si consulti la doc in linea)

```
...
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
```

Potreste trovare molte proposte con delle macro condizionali (ma cosa sono?)

SOCKET NON BLOCCANTI

In uno stesso sistema, modi simili si ottengono con procedimenti e primitive diverse ☹

Il non blocco si ottiene anche con primitiva ioctl() e parametro FIONBIO

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

Si modificano le primitive in caso non bloccante

- 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);
```


ANCORA SOCKET NON BLOCCANTI

Si può usare anche la fcntl ☹ dedicata al controllo dei file aperti

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

Ad esempio:

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

Ma anche con attributi diversi O_NDELAY e comandi con significati diversi in sistemi diversi

- System V: O_NDELAY
  read(), recv(), send(), write() senza successo valore 0 immediato
- POSIX.1 standard, System V vers.4: O_NONBLOCK
  Le chiamate senza successo valore -1 e la condizione d'errore EAGAIN
- BSD: FNDELAY, O_NONBLOCK
  Le chiamate senza successo valore -1 e la condizione d'errore EWOULDBLOCK

PROGETTO SERVER

In UNIX il progetto del server può essere molto flessibile, sia concorrente multiprocesso, sia monoprocesso, …

e anche con eventuali processi leggeri

e anche il client

		Tipo di comunicazione	
		con connessione	senza connessione
SERVER	**sequenziale** iterativo	servizi poco pesanti e affidabili	**Molto diffusi: per servizi poco pesanti non affidabili 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)