



**Università degli Studi di Bologna  
Scuola di Ingegneria**

# **Corso di Reti di Calcolatori T**

***Progetto C/S con Socket in C***

**Antonio Corradi**

**Anno accademico 2023/2024**

# SOCKET PER COMUNICAZIONE

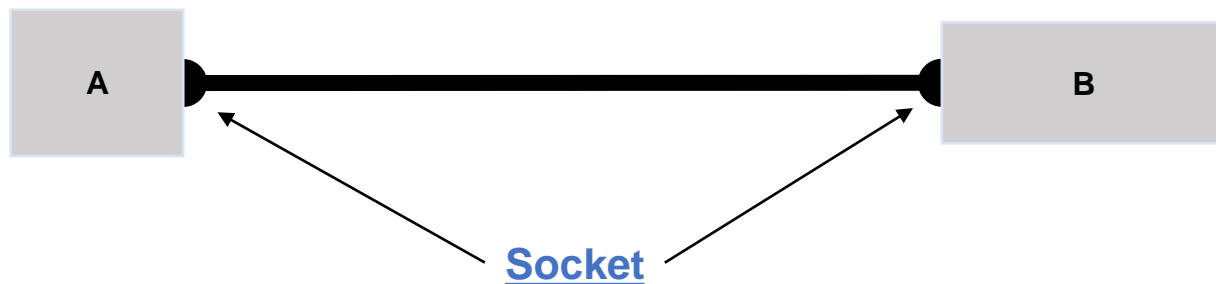
---

**Problema:** come comunicano tra loro **macchine distinte, diverse, fortemente eterogenee?**

Un **Client (A)** e un **Server (B)** su macchine diverse **possono comunicare** sfruttando **diversi tipi** di modalità di comunicazione che permettono una **qualità** e un **costo diverso** associato

Come in **Java, in C** è possibile programmare la rete attraverso **meccanismi di comunicazione** (*sul sistema operativo*) che qui sono disponibili attraverso **le API di UNIX**

**Le API socket sono lo strumento comune a tutti gli ambienti di linguaggio**



# TUTTE LE SOCKET IN C

---

## Argomenti

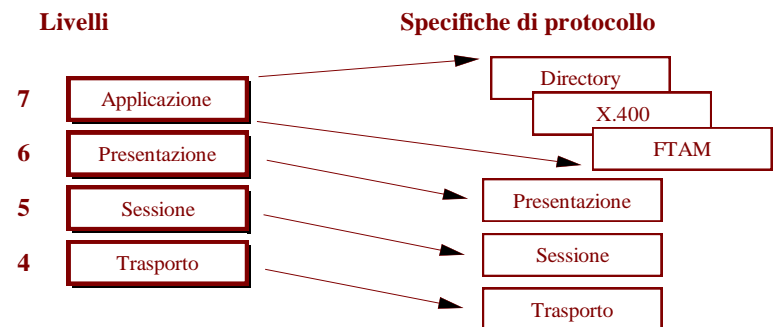
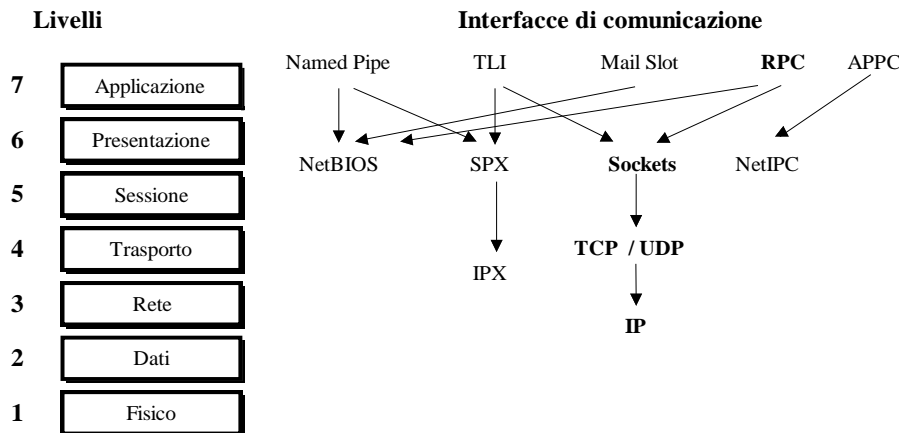
- 1) **Socket** e loro sistema di nomi
- 2) **Socket datagram** senza connessione
- 3) **Socket stream** con connessione
- 4) Funzioni **ausiliarie e primitive**
- 5) Esempi dei **due tipi di processi Client Servitore** usando socket diverse con implementazioni sequenziali o parallele multiprocesso
- 6) **Opzioni sulle socket** per ottenere comportamenti **non a default**
- 7) **Comportamenti non sincroni bloccanti** della socket via **primitive di kernel**
- 8) **Primitiva Select** per l'attesa multipla con servitore mono-processo

# 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**

**Scenario anni 80 con strumenti diversi, di livello applicativo e con servizi diversificati e poco integrabili attraverso modi standard**



**Socket come endpoint** per comunicare in modo **flessibile**, **differenziato ed efficiente e standard**

# UNIX: STRUMENTI DI COMUNICAZIONE

---

## **modello socket e strumenti standard per comunicazione e sincronizzazione**

UNIX definisce e 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 coresidenti 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

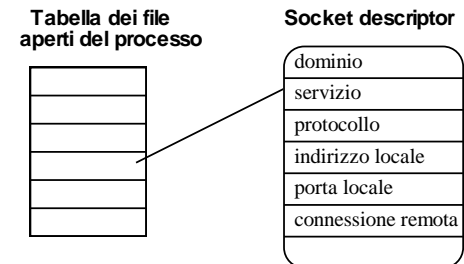
In UNIX ogni processo mantiene una **tabella di kernel** (**tabella dei file aperti del processo**) in cui ogni sessione aperta sui file viene mantenuta attraverso uno specifico file descriptor (fd intero)

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

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

Le socket sono conformi a questo paradigma in modo omogeneo rispetto alle azioni sul file system

Ovviamente, nella comunicazione internamente si specificano **più parametri** per definire un collegamento con connessione:

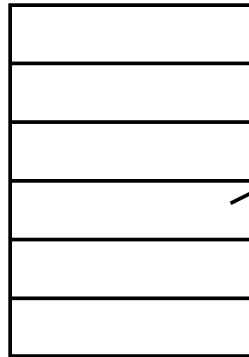


**protocollo di trasporto;** e **NOME a quadrupla**

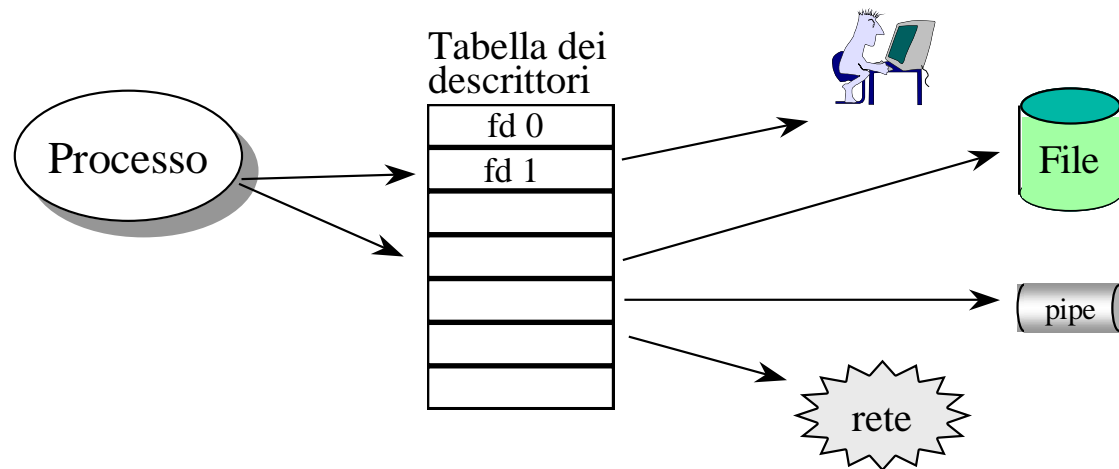
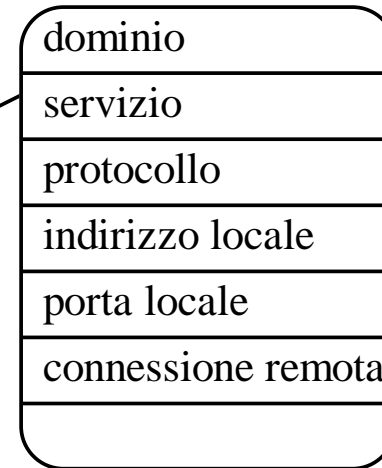
**< indirizzo locale; processo locale; indirizzo remoto; processo remoto >**

# UNIX: MODELLO DI USO

**Tabella dei file  
aperti del processo**



**Socket descriptor**



# UNIX: PRIMITIVE

---

UNIX **deve fornire funzioni primitive di comunicazione**

**(API sincrone e bloccanti in Unix a livello locale)**

UNIX Berkeley introduce il meccanismo di **socket standard**, come strumenti di **comunicazione locale o remota** con politiche differenziate, in alternativa ai problemi degli strumenti concentrati, **trasparente e ben integrata con processi e file**

**i processi possono scrivere/leggere messaggi e stream su socket, 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



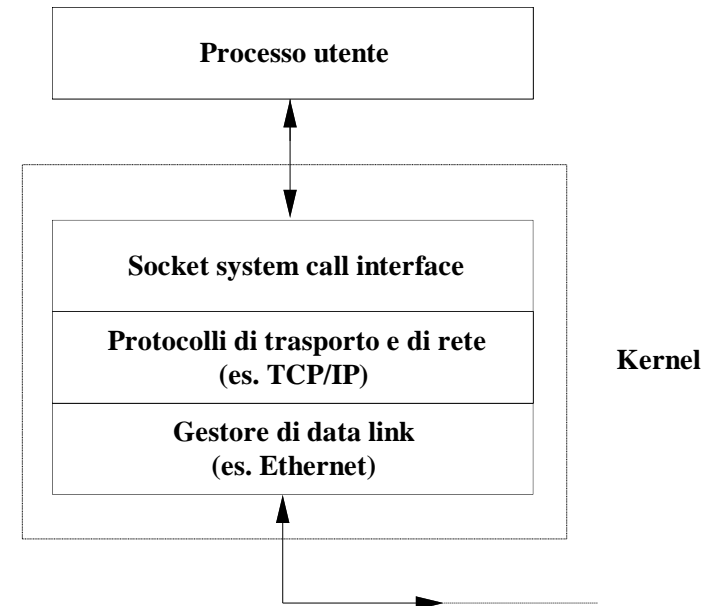
# UNIX: TRASPARENZA

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

- **Socket** endpoint della comunicazione (**nuovo tipo**)
- **Socket descriptor** integrato con i file descriptor (**intero**)

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

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



# SOCKET: DOMINIO DI COMUNICAZIONE

---

Il modello specifica la socket il **Dominio di comunicazione** per:

**Semantica di comunicazione + relativo standard di Nomi**

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 ( <b>TCP/IP</b> )
.....	.....

# TIPO DI COMUNICAZIONE

---

## Tipo 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 scambiati senza azioni aggiuntiva (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 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 C 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 vari tipi di strutture dati

**sockaddr** (indirizzo generico)      **sockaddr\_in** (famiglia AF\_INET)

```
struct sockaddr { u_short sa_family; char sa_data[14]; }
```

```
struct sockaddr_in /* famiglia internet */
{ 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};
```

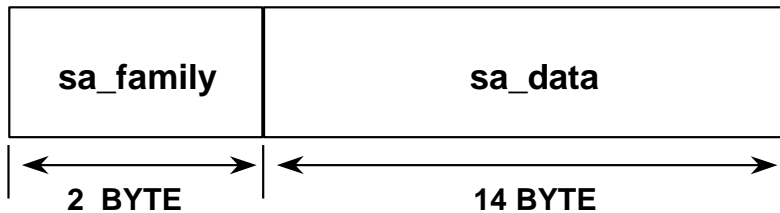
```
struct sockaddr_in mioindirizzosocket; /* variabile per il nome*/
```

<b>sin_family</b>	⇒ famiglia di indirizzamento	sempre AF_INET
<b>sin_port</b>	⇒ numero di porta	
<b>sin_addr</b>	⇒ indirizzo Internet del nodo remoto (numero IP)	

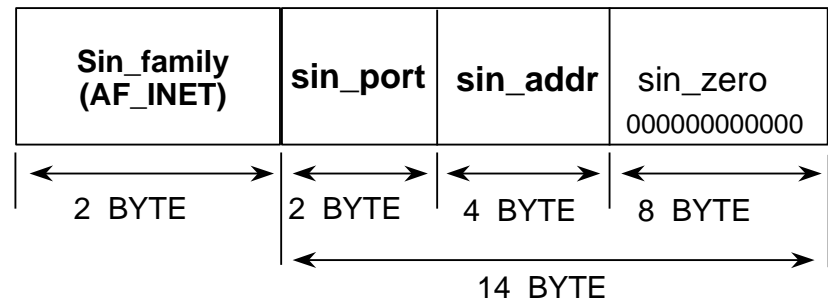
# INDIRIZZI E NOMI SOCKET IN C

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 (o aree **a caratteri** o aree **non definite – void**)

`char *` in C

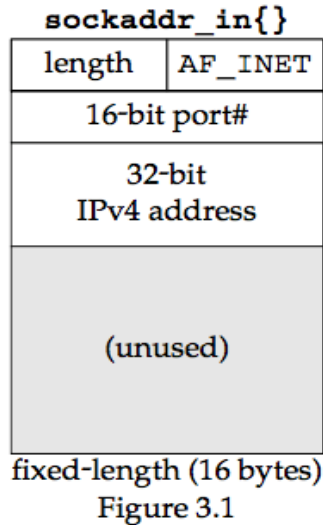
Si vedano i file di inclusione tipici ...

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

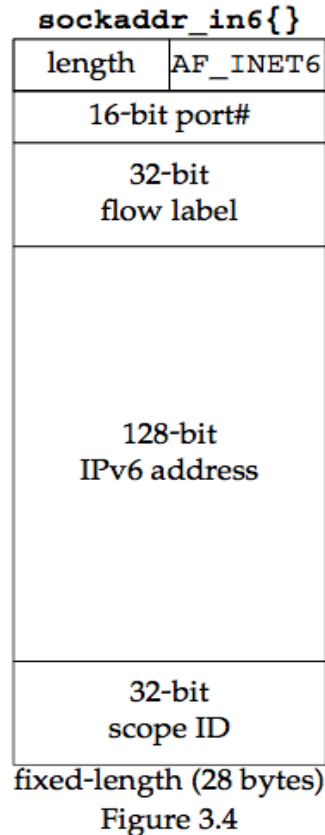
`void *` in ANSI C

# INDIRIZZI SOCKET SOCKADDR

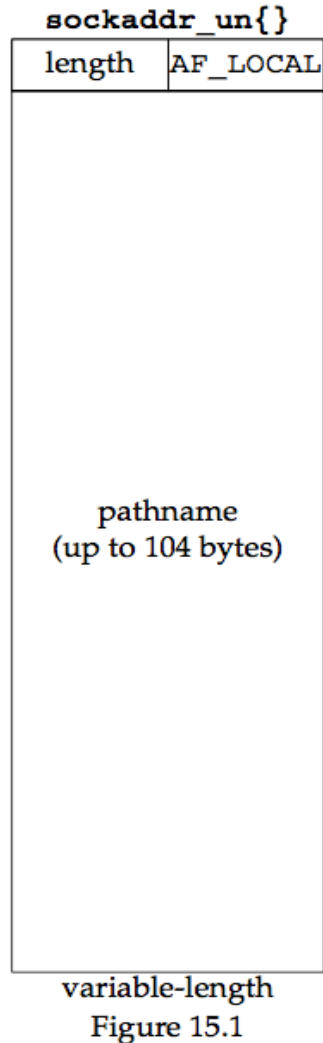
## IPv4



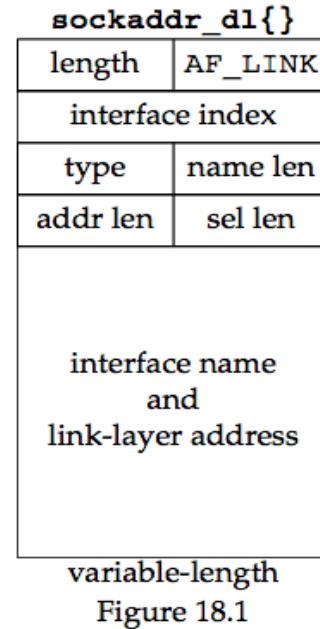
## IPv6



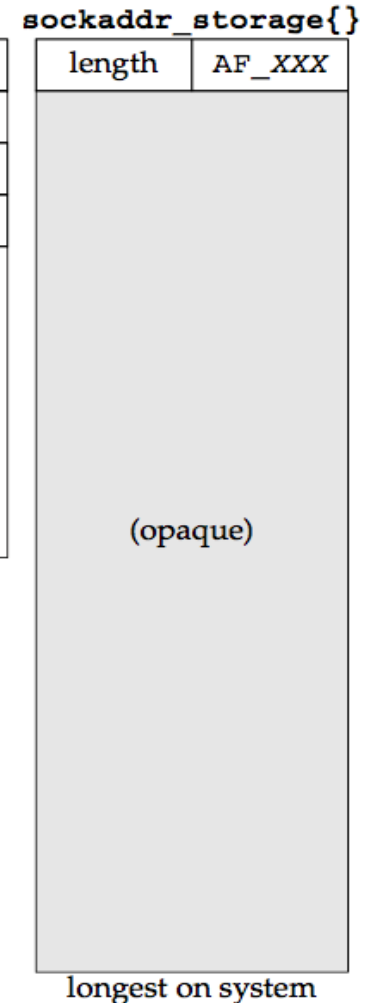
## Unix



## Datalink



## Storage





# 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 avveniva 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 dell'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 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** (della variabile)

# USO GETHOSTBYNAME

---

Esempio di utilizzo della `gethostbyname` 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])) /* caso di successo */
    peeraddr.sin_addr.s_addr = /* assegnamento 4 byte IP */
    ((struct in_addr *) (hp->h_addr))
    /* casting di puntatore */
    -> 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 corrispondenza

{nome servizio, protocollo,  $\Rightarrow$  porta}

```
echo      7/tcp      # Echo
systat    11/tcp      users # Active Users
daytime   13/tcp      # Daytime
daytime   13/udp      #
gotd      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 di nomi fisici per agganciarsi ai nodi e porte locali

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

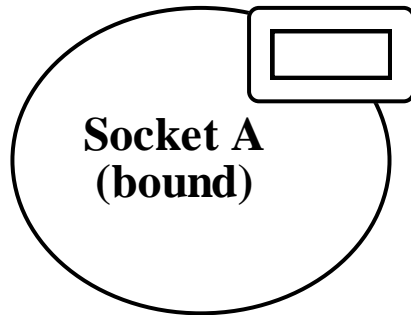
- dichiarazione delle variabili di riferimento a una **socket**
- conoscenza della porta per il **servizio da offrire**
- ricezione su qualunque indirizzo IP locale (wildcard address), utile per server con più connessioni, detti **multiporta**

# MODELLO SOCKET DATAGRAM

---

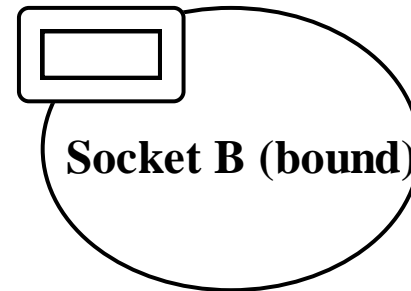
Le **socket datagram** sono **end-point** per comunicare con chiunque dello stesso dominio

**processo Mittente o Cliente**



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

**processo Ricevente o Server**



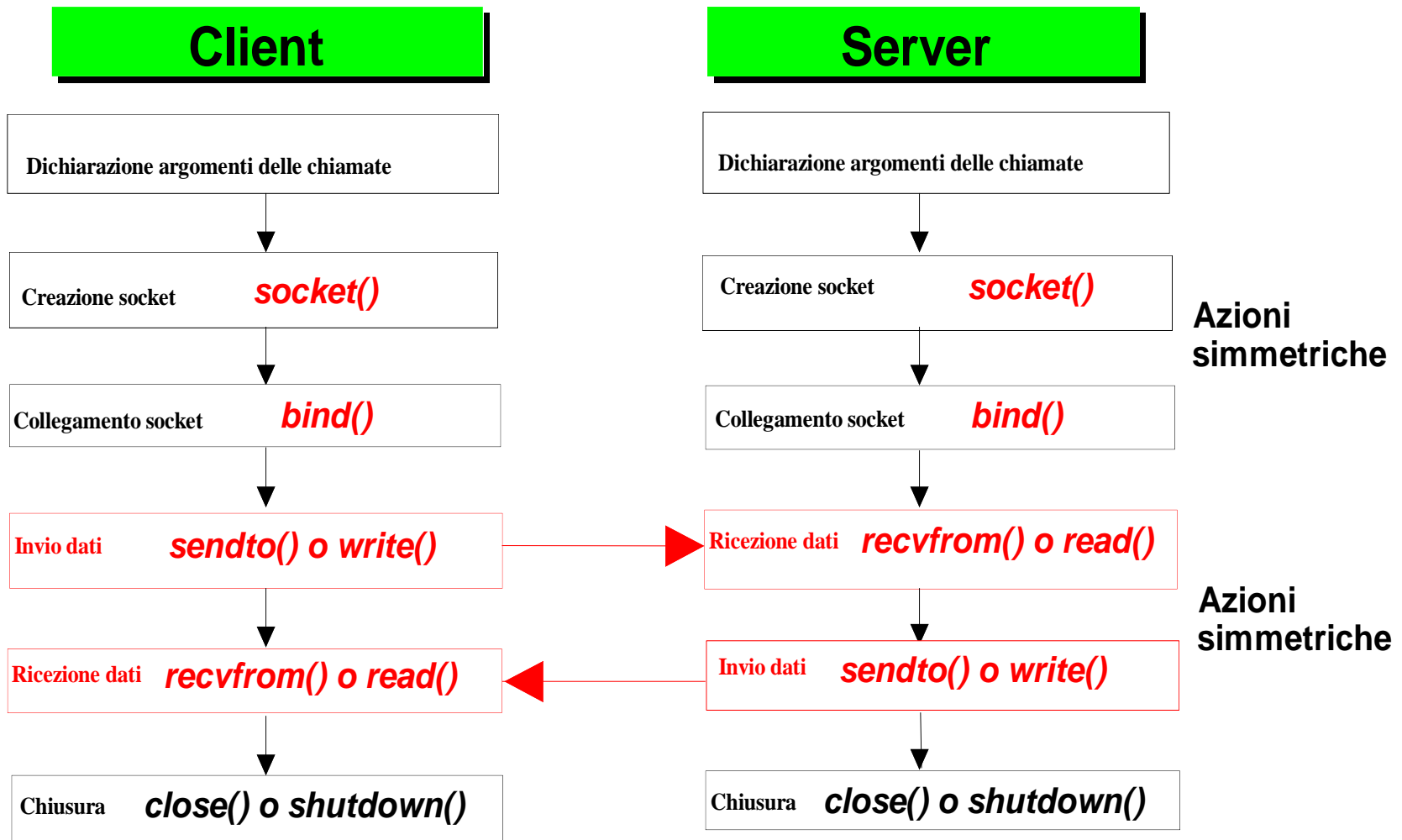
- il server ha creato la socket
- il server ha collegato la socket ad un indirizzo

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



# PROTOCOLLO SOCKET DATAGRAM

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

<b>nbytes</b>	⇒ lunghezza messaggio inviato/ricevuto
<b>s</b>	⇒ socket descriptor
<b>buf, len</b>	⇒ puntatore al messaggio o area e sua lunghezza
<b>flags</b>	⇒ flag (MSG_PEEK lascia il messaggio sulla socket)
<b>to/from/ fromlen/tolen</b>	⇒ 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 nella comunicazione (best effort).

- Lunghezza massima del messaggio (**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 si riceve con la massima area possibile

A livello utente si può ottenere **maggiore affidabilità prevedendo**

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

# 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, sizeof(msg), 0,  
               servaddr, sizeof(struct sockaddr_in));  
... close (s); /* se non si usa la socket, si deve chiuderla */
```

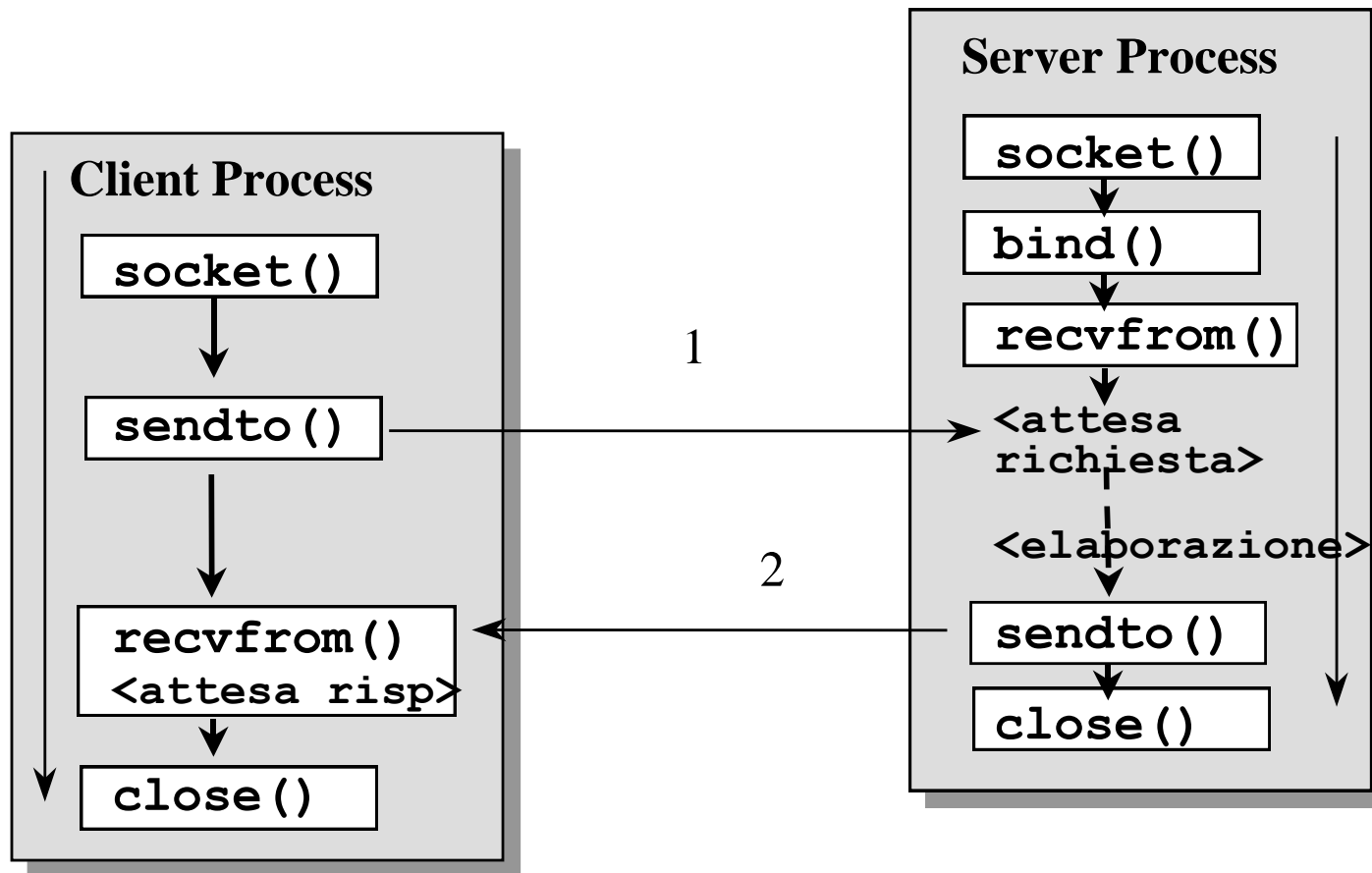
**SERVITORE (RICEVENTE)** /\* anche qui sono state fatte le socket e bind, e si è definito il buffer per contenere le informazioni ricevute \*/

```
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/Server**



# PROPRIETÀ DEI DATAGRAMMI

---

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 infinita 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 vengono 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 C PASSO PASSO

---

*Uso di alcune costanti molto utili e comode*

Per esempio, per denotare tutti 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 relativo al nodo di residenza */
<<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...

Il **server di echo** è un demone che deve svilupparsi su una **unica socket datagram**

**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 da cui si riceve il messaggio */
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

---

Un server datagram è un demone che deve leggere sempre da una socket trattare il messaggio datagram e poi scrivere sulla stessa socket per rispondere al mittente...

```
/* il server si sviluppa in un ciclo infinito di lettura e
risposta, e 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);
}
```

# 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

Per due byte **Little-endian**

address A+1	address A
High-order byte	Low-order byte

MSB	LSB
-----	-----

High-order byte	Low-order byte
address A	address A+1

**Big-endian**

# FUNZIONI ACCESSORIE SOCKET

---

**Funzioni accessorie da usare con socket con obiettivo di portabilità (li useremo solo per parte di controllo e non per i dati):**

- **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 per la parte di controllo solamente**

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 stringa 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 come risultato l'indirizzo IP a 32 bit da utilizzare nelle primitive

### Conversione da indirizzi IP a 32 bit a stringa, 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

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

# API SOCKET IN C

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

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

Quali primitive possono avere una elevata durata? *In bold (anche dopo)*

# API SOCKET IN C

---

## Seguono ancora primitive per le socket

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

Si sperimentino le primitive non note (e note)  
(sono tipicamente locali e non costose)

# SOCKET STREAM IN C

---

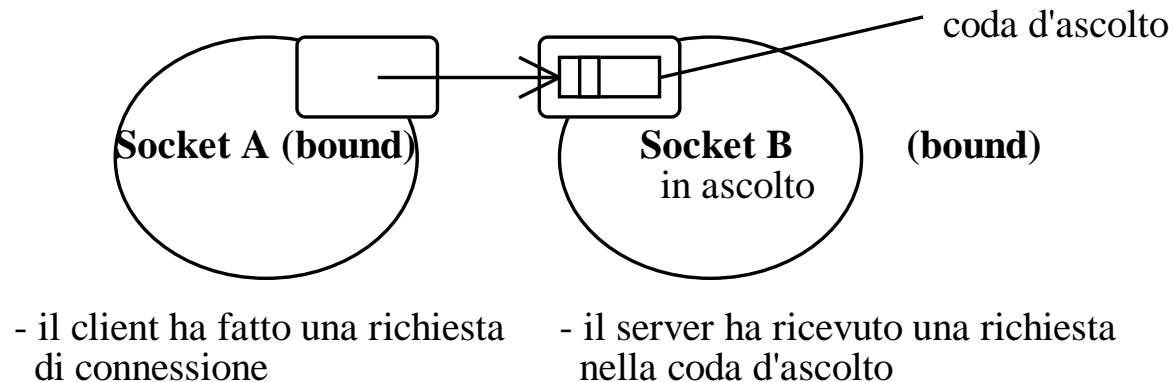
Le **socket stream** prevedono una **risorsa** che rappresenta la **connessione virtuale** tra le entità interagenti

## 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

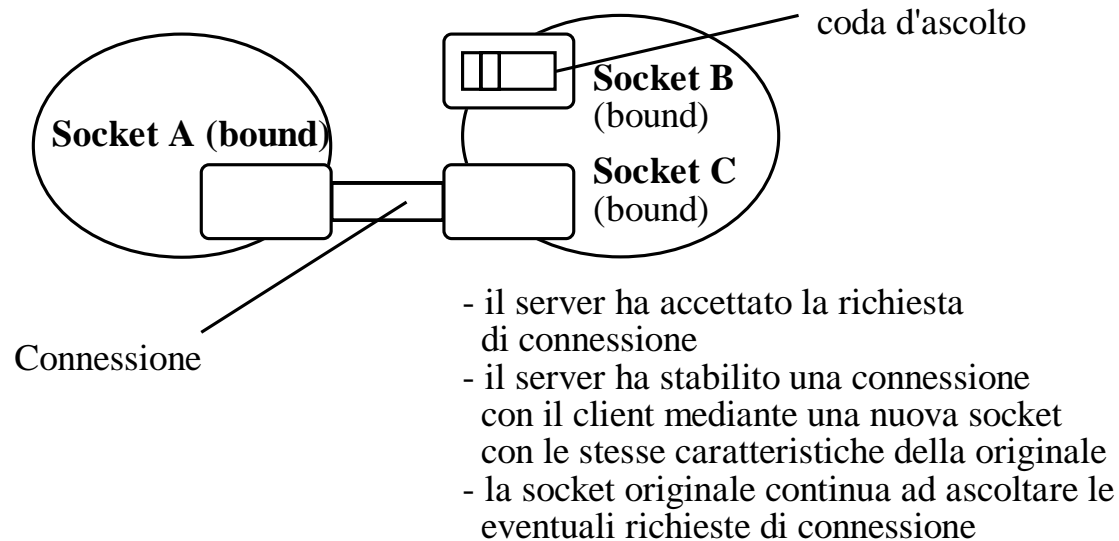
- entità attiva richiede la connessione, la entità passiva accetta
- primitive diverse e comportamento differenziato all'inizio





# SOCKET STREAM: CONNESSIONE

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



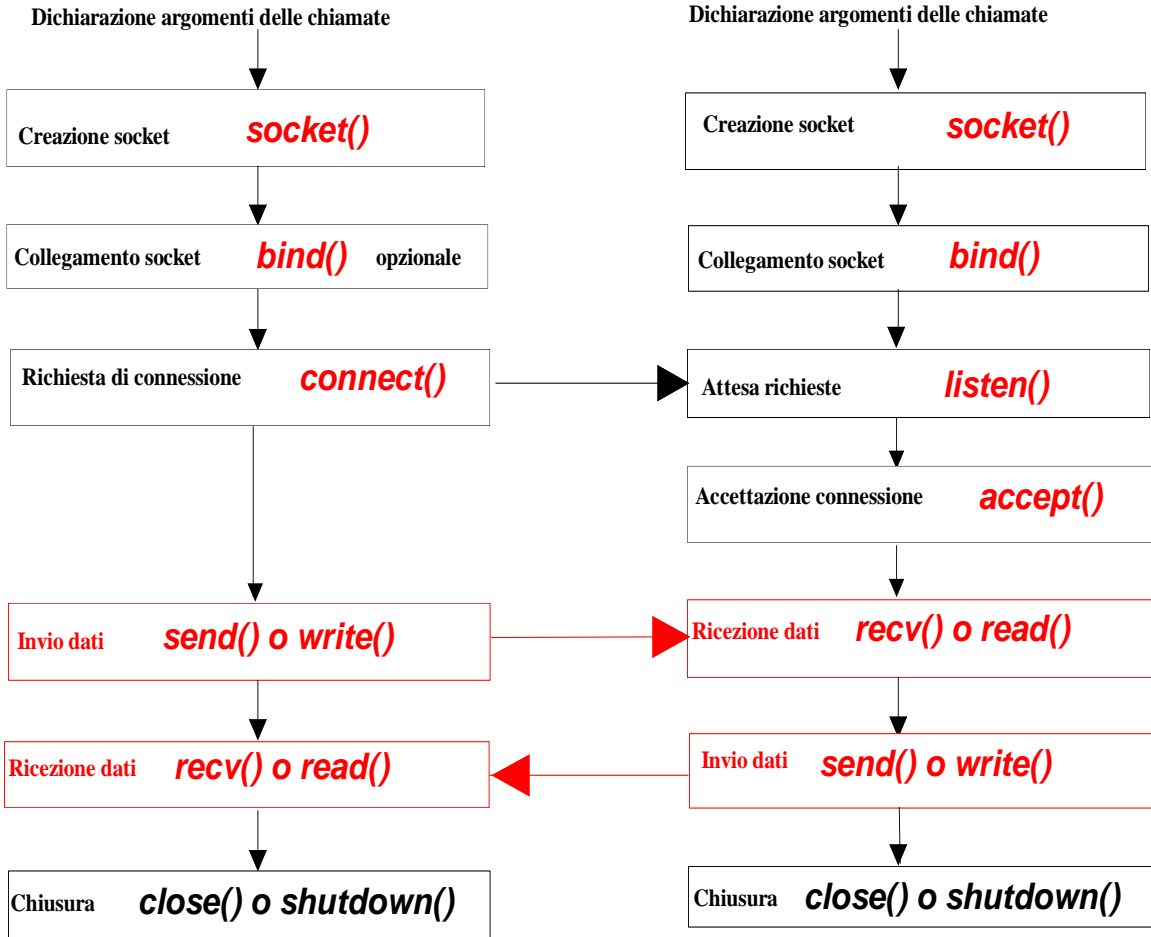
- ruolo **attivo/passivo** di entrambi che possono inviare/ricevere informazioni
- naturale ruolo **attivo del client** che comincia la comunicazione
- primitive diverse e comportamento differenziato all'inizio

# PROTOCOLLO SOCKET STREAM IN C

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

**Client**

**Server**



parte  
asimmetrica  
ruolo attivo

parte  
asimmetrica  
ruolo passivo

parte  
simmetrica

parte  
simmetrica

# SOCKET C STREAM CONNESSE

---

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

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 naif** 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 sfruttare la **piena potenzialità** delle primitive delle socket.

# PRIMITIVE SOCKET C: 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 C: BIND()

---

Una socket deve essere legata al livello di nomi globali ossia visibili (qui legame **con nomi globali del nodo corrente**)

La **bind()** collega la socket creata localmente a porta e nodo globali visibili:

```
#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;
```

<b>s</b>	⇒ socket descriptor che identifica la socket (liv. locale)
<b>addr</b>	⇒ struttura con indirizzo di porta nome nodo stesso
<b>addrlen</b>	⇒ <i>la lunghezza di addr</i>

La bind crea half-association **protocollo: < IP locale; porta locale;>**

Con l'obiettivo di ottenere l'intera connessione

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

Ogni nodo che vuole **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 la 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(struct 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 C 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;
```

<b>s</b>	⇒ socket descriptor
<b>addr</b>	⇒ puntatore al socket address remoto
<b>addrlen</b>	⇒ lunghezza di questa struttura
<b>risultato</b>	⇒ 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 dopo avere fatto delle rismissioni**

Al termine della connect la connessione è creata (**almeno lato cliente e a livello di driver TCP**)

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

# SOCKET C 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 l'iniziativa** e si **attiva** per preparare le risorse

- La primitiva può avere tempi di completamento anche **elevati perché esegue anche ritrasmissioni**
- 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`)

- **ECOMM** - **Communication error on send**
- **ECONNABORTED** - **Connection aborted** (non ritentare)
- **ECONNREFUSED** - impossibilità di connettersi (non ritentare)
- **ETIMEDOUT** - tentativo di connessione in time-out: la coda d'ascolto del server è piena o non creata. Non si è depositata la richiesta

In caso di **successo**, il client **considera immediatamente la connessione stabilita** (anche se il server non ha accettato il tutto)



# USO SOCKET CLIENT: CONNECT()

---

La primitiva **connect()** è anche capace di invocare la bind e 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(struct sockaddr_in);
struct sockaddr_in *peeraddr; /* impostazione dei valori del
server, con uso di gethostbyname e getservbyname
eventualmente ...*/
s = socket (AF_INET, SOCK_STREAM, 0);
res = connect (s, addr, addrlen);
if (res<0) /* errore e exit */      else /* procedi con la
connessione */ ...
```

La **connect ()** ha successo e restituisce il controllo quando ha depositato la richiesta nella coda del servitore (vedi **listen ()**)

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 non sincronicità**

# SOCKET C SERVER: LISTEN()

---

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

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

<b>s</b>	⇒ socket descriptor
<b>backlog</b>	⇒ numero di posizioni sulla coda di richieste (1-10, tipicamente <b>5</b> )
<b>risultato</b>	⇒ se negativo errore, se positivo restituisce il file descriptor

La primitiva **listen()** è una primitiva **locale**, passante (istantanea) e **senza attesa**; 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

- L'accodamento della richiesta 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 C 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 C 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 creata:**

- 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 lasco**: il livello applicativo lavora in **modo non simmetrico cliente/servitore** mentre il trasporto è molto coordinato)

# USO SOCKET CLIENT: ACCEPT()

---

La primitiva **accept()** è usata dal server per ottenere un nuova connessione

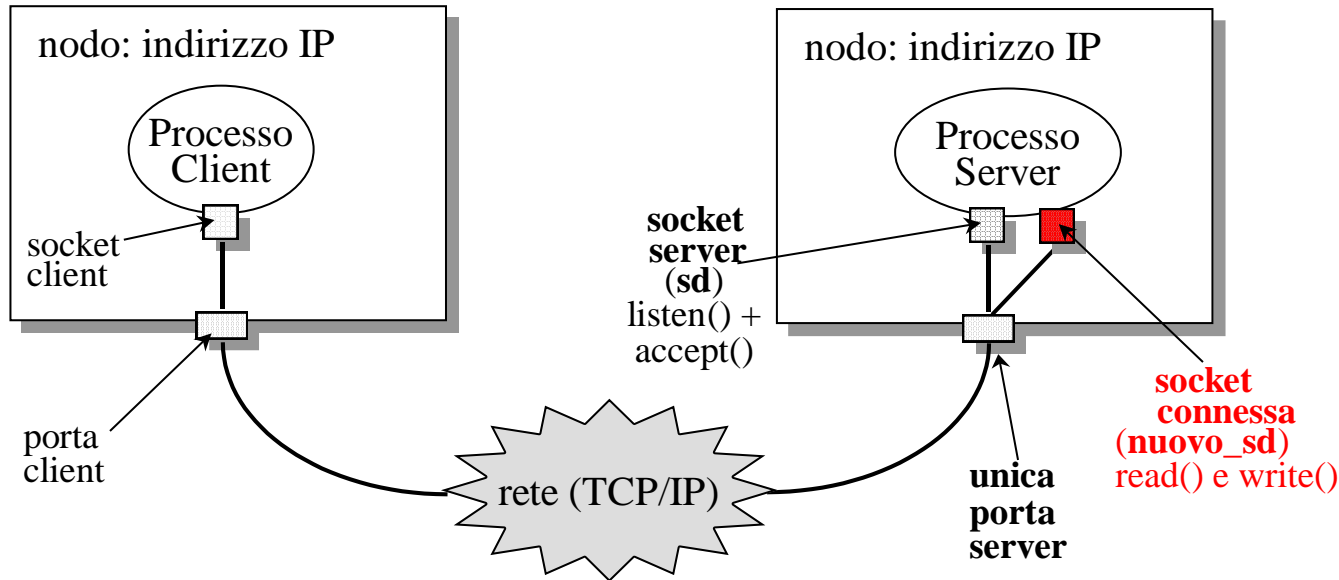
```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
int s, res, addrlen = sizeof(struct sockaddr_in);
struct sockaddr_in *myaddr; /* impostazione dei valori del
server, con uso di gethostbyname e gestservbyname
eventualmente ...*/
struct sockaddr_in *peeraddr; int peeraddrlen;
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

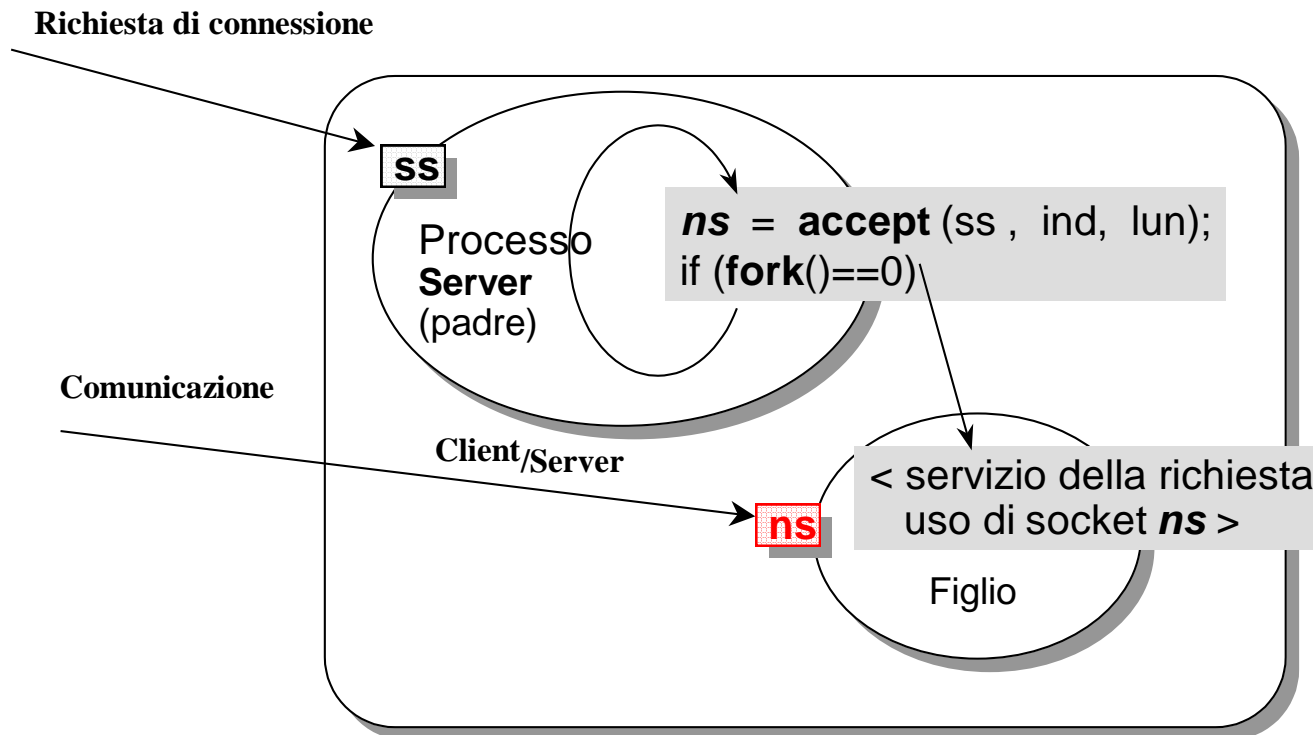
Lato server, la connessione è rappresentata da questa socket

# 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

**Server concorrente    multi-processo    connection-oriented**



# 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;
```

<b>s</b>	⇒ socket descriptor
<b>buf /msg</b>	⇒ puntatore all'area che contiene il messaggio (IN/OUT)
<b>len</b>	⇒ lunghezza del messaggio
<b>flags</b>	⇒ opzioni di comunicazione
<b>risultato</b>	⇒ <b>numero di byte realmente inviato/ricevuto</b>

*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: le primitive read/write sono usabili per socket connesse con la semantica di send/receive

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

<code>s</code>	⇒ socket descriptor
<code>msg</code>	⇒ puntatore all'area che contiene il messaggio (IN/OUT)
<code>len</code>	⇒ lunghezza del messaggio
<b>risultato</b>	⇒ <b>numero di byte realmente inviato/ricevuto</b>

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

La **lettura/read** attende e legge almeno **1 byte disponibile (vedi la opzione watermark)**, cercando di trasferire in spazio utente i byte arrivati sulla connessione

# COMUNICAZIONE E CLIENTE/SERVITORE

---

Le operazioni del **cliente servitore** che usano una socket connessa hanno un obiettivo di uso della connessione ma usano per realizzarlo le singole primitive

La **scrittura non è sincrona con la lettura a livello applicativo**

La connessione garantisce un disaccoppiamento

Posso fare **molte scritture (di un byte)** e consumare **tutti i byte con una unica lettura** e viceversa

**Le primitive sono sincrone con che cosa?**

Ognuno lavora **con il proprio endpoint** e deve fare i conti con la propria **visione verticale**:

- La **send/write** può essere **sospesa dalla driver che non ha memoria e messa in attesa**
- La **receive /read** può essere **messa in attesa e sospesa dalla driver che non ha almeno un byte da consegnare**

La **lettura** e la **scrittura a livello applicativo sono sempre sincrone con la driver TCP che le serve**

# USO DI STREAM

---

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

- 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 lavora con una sorgente di input e un sink di output

**Come sfruttare al meglio la rete?**

- ogni dato che sia stato inviato deve essere ricevuto evitando di dovere **scartare dati che sono arrivati ad un endpoint** e non vengono ricevuti

**Esiste una corrispondenza tra numero di scritture e letture?**

- Qui il livello di trasporto TCP fa da facilitatore (non come nei datagrammi): le azioni delle driver di trasporto sono **ottimizzate e lavorano in 'modo indipendente' dalla applicazione**. Una lettura può anche **consumare molti byte** di scritture diverse se possibile; ovviamente potrebbe anche consumare solo in parte i dati provenienti da una scrittura applicativa

# 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 opzione watermark

**Come preservare i messaggi in ricezione?**

- ogni receive restituisce i dati della driver locali: TCP (essendo a stream di byte) non implementa marcatori di fine messaggio

**Soluzione:** **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 **usiamo messaggi con descrittori espliciti di lunghezza**, letti con due letture in successione

# USO DI STREAM E FINE

---

**Disciplina di uso connessione a stream:**

**ogni nodo ha un source (in) e un sink (out)**

Ciascuno controlla la propria **parte di autorità**, il proprio sink, su cui scrive, e deve essere collaborativo con il pari per le azioni **sul source** da cui legge

**Come si coopera? segnali di fine flusso**

- Su ogni flusso viaggiano **dati** in stream fino alla **fine del file del pari di autorità** che, **dopo avere operato completamente segnala con una chiusura che non ci si devono aspettare più dati** (via fine file) e si aspetta che i dati siano consumati
- **La chiusura causa una indicazione di fine file all'altro**

**Ogni endpoint deve osservare un protocollo:**

- **deve leggere tutto l'input dal source fino alla fine del flusso**
- **deve inviare una fine del flusso** quando vuole terminare i dati in uscita sulla sua direzione del flusso

# 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 primitiva è **locale, passante e senza attesa** (istantanea per il processo invocante)
- La primitiva 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 (con tempi non prevedibili)**
- Con un ritardo anche **molto lungo a default** (e limitandone in tempo la consegna dei dati ad un intervallo di tempo controllato da un'opzione socket avanzate (vedi opzione 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 minima locale** che termina **immediatamente** per il chiamante (a default) e con **un forte impatto sulla comunicazione e sul pari (e sulla driver TCP)**

- Si registra un (forte) ritardo tra la chiusura applicativa e la **reale deallocazione della memoria** di supporto nella driver TCP

Ogni **socket a stream** è associata ad un **buffer di memoria** per mantenere i **dati in uscita** e i **dati di ingresso**

- **Alla chiusura**, ogni messaggio nel buffer associato alla socket **in uscita** sulla connessione **deve essere spedito (mettendoci tutto il tempo necessario)**, mentre ogni **dato in ingresso** ancora non ricevuto **viene buttato via (occupazione della memoria porta)**
- **Solo dopo**, il sistema può **deallocare la memoria del buffer di trasporto (porta) e la si libera**

**Dopo la close** (e la notifica), il pari connesso alla socket chiusa:

- se legge dalla socket, alla fine ottiene **finfile**
- se scrive, dopo alcune richieste, ottiene un **segnale di connessione** non più esistente (pipe broken)
- La durata **della chiusura memoria per la driver TCP** è poco predicibile 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 sui due versi

La primitiva `shutdown` produce una chiusura 'dolce' direzionale (tipicamente si chiude **solo il verso di uscita** della connessione)

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

`s`       ⇒     socket descriptor della socket

`how`    ⇒     verso di chiusura

`how = 0, SHUT_RD`

- INPUT - non si ricevono più dati, ma si può trasmettere
- `send()` del pari ritorna con -1 ed il processo riceve SIGPIPE

`how = 1, SHUT_WR`

- OUTPUT - si può solo ricevere dati dalla socket e non trasmettere
- L'altro pari collegato alla connessione alla lettura riceve end-of-file

`how = 2, SHUT_RDWR` entrambi gli effetti



# USO DELLA SHUTDOWN()

---

La primitiva `shutdown(1) out` 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 quell'input e lo può chiudere

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

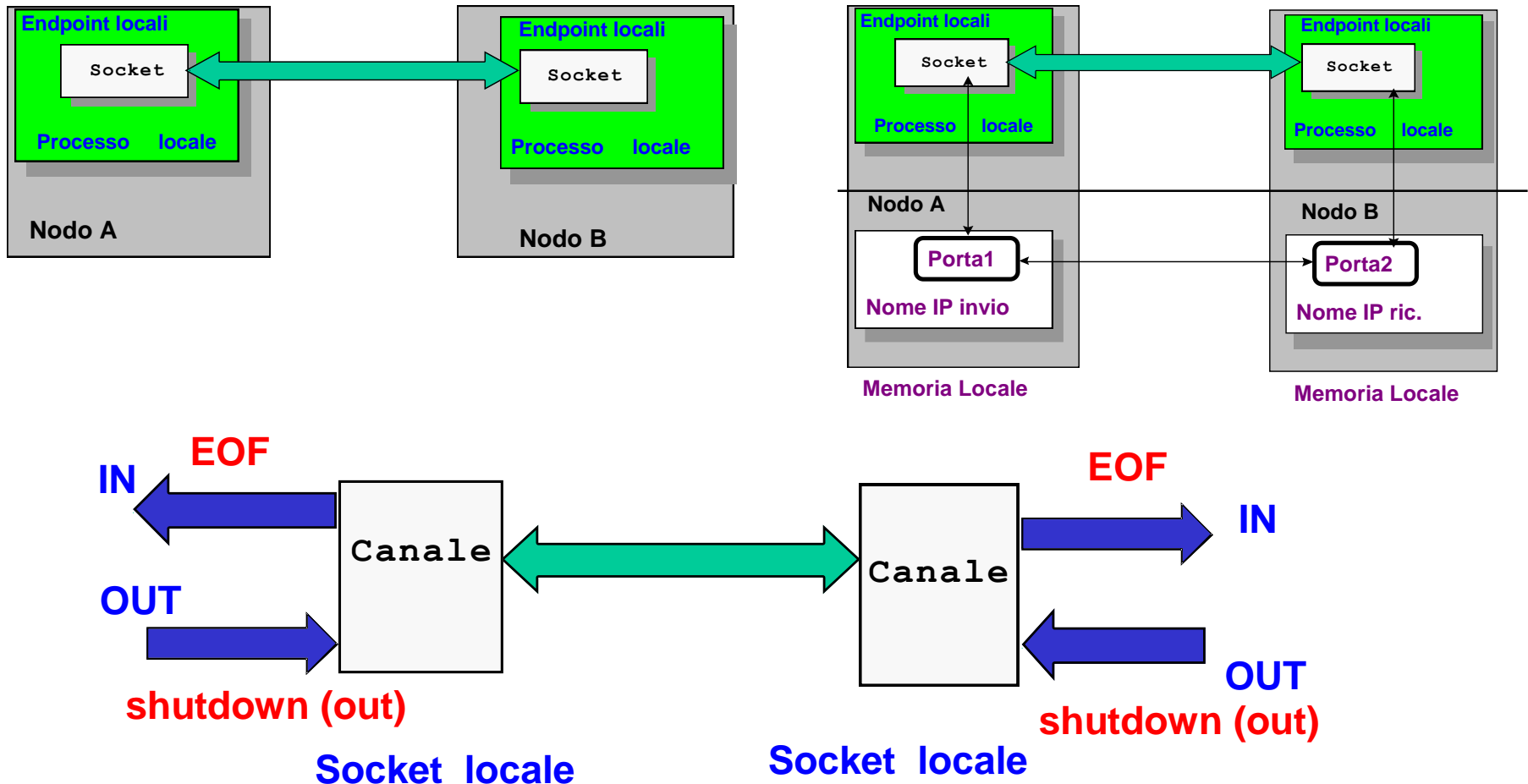
- il pari, avvertito con una fine del file, può fare ancora uso del suo verso per molti invii, fino alla sua decisione di shutdown si attua una chiusura dolce

In modo del tutto simile, in Java esistono

`shutdownInput()` e `shutdownOutput()` per le chiusure dolci direzionali

# CANALI IN IN/OUT E GESTIONE

La comunicazione stream comporta una **buona gestione dei canali a diversi livelli** (network, trasporto, e applicativo)

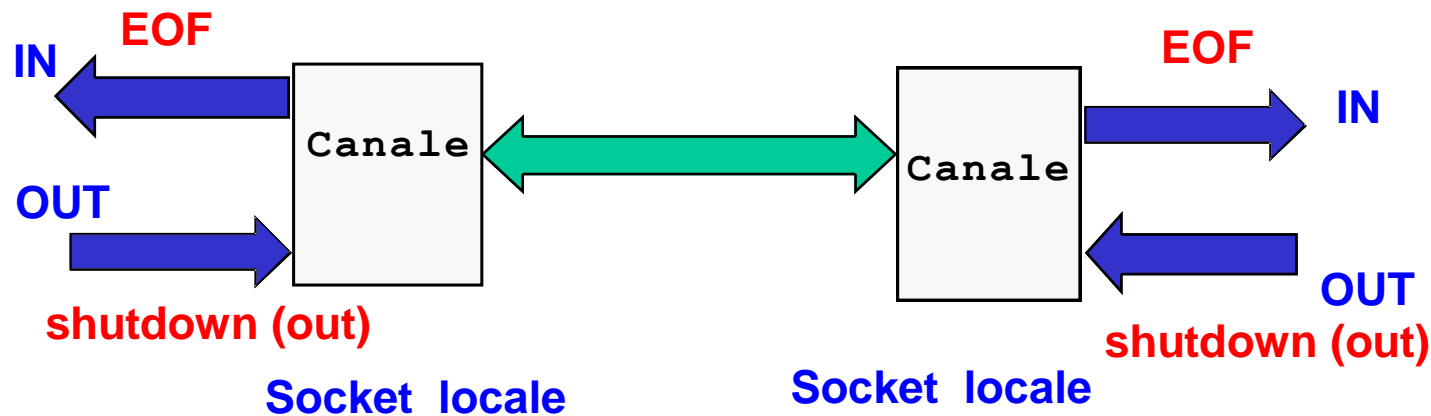


# BUONA GESTIONE DEI CANALI

La comunicazione stream comporta per ciascuno di avere un verso di **autorità (output)** su cui siamo padroni di decidere la **chiusura** e un verso in cui **siamo slave (input)** e su cui dipendiamo da chi ha l'autorità

La buona gestione di uno stream prescrive di lavorare per verso:

- **sull'output** posso fare la chiusura quando voglio - `shutdown (s, 1)`; l'altro riceve l'EOF
- l'input dipende dall'altro, quando si riceve il fine file, si può chiudere quella parte - `shutdown (s, 0)`



# CHIUSURA DELLA SOCKET STREAM

---

In caso di **chiusura** di una socket stream, chiudiamo la parte applicative non faremo più azioni al livello applicativo

**La close è una primitiva molto veloce e passante per l'utente (a default)**

**Libera il file descriptor del processo**

Se consideriamo il livello trasporto, la driver **TCP mantiene traccia della connessione** e ha memoria associata alla socket, che può contenere sia dati nel **buffer di input** (arrivati dal pari, ma non consumati), sia **dati in uscita** (mandati dal processo che ha fatto la close, ma non ancora inviati)

Il contratto TCP prescrive che:

- **I dati di input vengano buttati via;**
- **I dati di output vengano consegnati al pari che è tenuto ad aspettarli e riceverli fino a che non riceve la fine del file;**  
la consegna è garantita dalla driver mittente e potrebbe durare anche molto tempo, ma viene onorata fino alla fine;  
questo potrebbe impedire l'aggancio alla stessa porta da parte un nuovo processo locale (per questo la opzione *reuseaddress*)

# PROCESSI NAIVE

---

I processi molto semplici (**processi naif**) 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()` 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**

# 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 relativo 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 contenuto del 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

---

... **File descriptor 0 - 1 - 2 già aperti**

**/\* 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; /* set indirizzo server */
server_address.sin_port = htons(12345);
sd=socket(AF_INET, SOCK_STREAM, 0); sd=3
```

**/\* 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) // fd=4
    {perror("open");close(sd) 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

```
// File descriptor 0 - 1 - 2 già aperti
sd=socket(AF_INET, SOCK_STREAM, 0); // sd=3
if(sd<0) {perror("apertura socket"); exit(1);}
mio_indirizzo.sin_family=AF_INET;
mio_indirizzo.sin_port=htons(12345);
mio_indirizzo.sin_addr=INADDR_ANY;
/* non htonl: simmetrico*/
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); // sd=4
printf("server legge %s \n", buff);
if((fd=open(buff, O_WRONLY|O_CREAT|O_EXCL))<0) // sd=5
{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); // libero 4 e 5
/* il server è di nuovo disponibile ad un servizio */
```

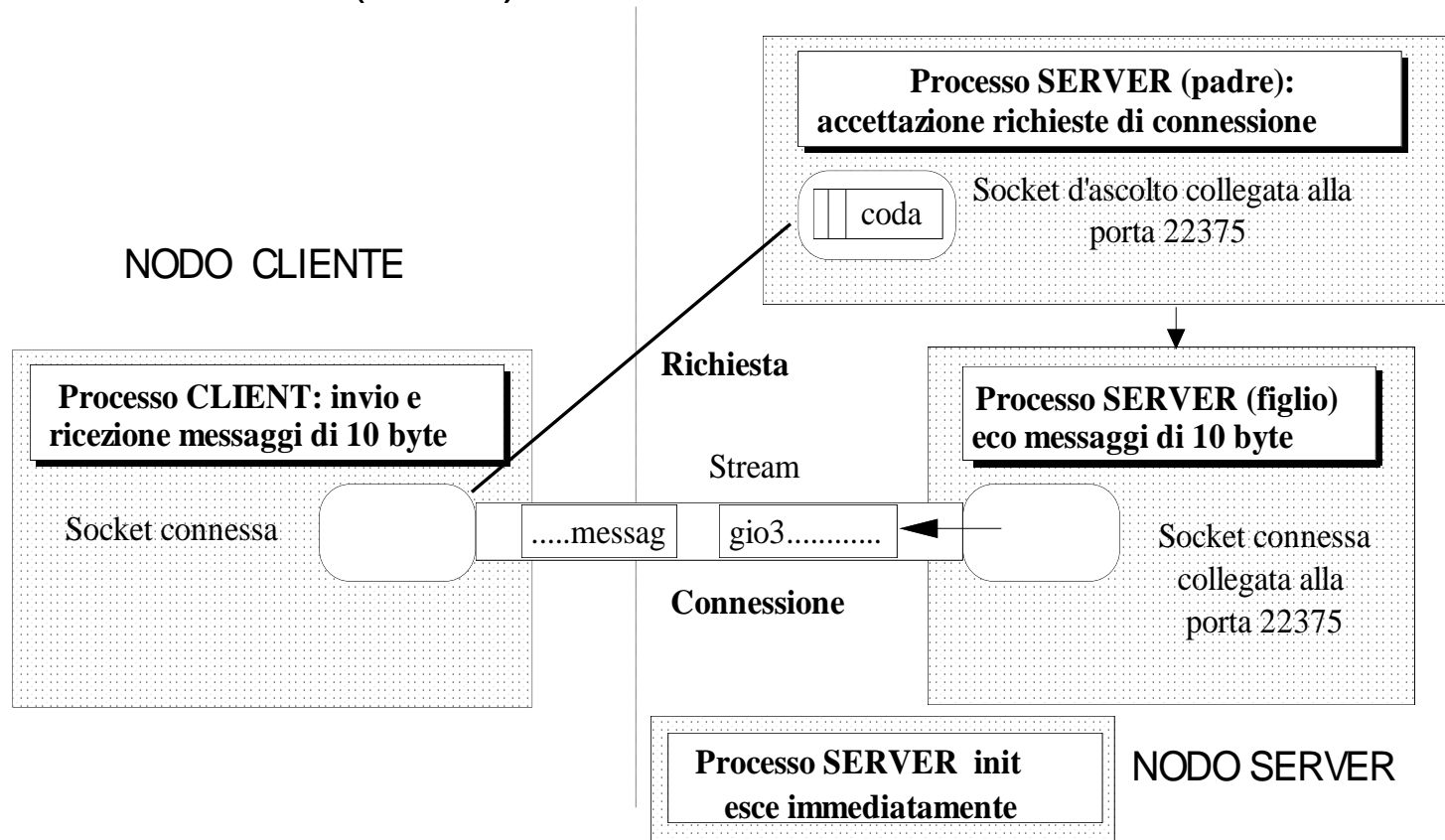
# RCP: SERVER MULTIPROCESSO

---

```
for(;;) { // sd=4
ns=accept(sd, (struct sockaddr *)&client_address, &fromlen);
if (fork()==0) /* figlio */
{close(sd);read(ns, buff, DIM_BUFF); // figlio libera 3
/* chiude socket servizio */
printf("il server ha letto %s \n", buff);
if((fd=open(buff, O_WRONLY|O_CREAT|O_EXCL))<0) // fd = 3
{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); } // libera 3 e 4
/* padre */
close(ns); // padre libera 4
wait(&status); }
/* 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 per formattazione dell'orario */
long int time (); /* tempo in secondi da inizio */
int ricevi ();
/* dichiarazione routine di ricezione di un messaggio*/
int s; /* socket descriptor del cliente */;
char * nomeprog;
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);
}
```

```
nomeprog = argv[0];
```

```
/* per gestire condizioni d'errore in procedure */
```

# 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 */  
  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 (getsocketname (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], ntohs(myaddr_in.sin_port), ctime(&timevar));  
/* Il numero di porta espresso in byte senza convertire */  
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  $\Rightarrow$  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 = 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", *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(nomeprog);
            fprintf(stderr, "%s: errore in lettura\n", nomeprog);
            exit(1); }
        while (i < n) j = recv (s, &buf[i], n-i, 0);
        { if (j == -1) {
            perror(nomeprog);
            fprintf(stderr, "%s: errore in lettura\n", nomeprog);
            exit(1); }
            i += j; if (j == 0) break; }
        } /* si assume che tutti i byte arrivino ... se si verifica il fine file si esce */
    return i; }
}
```

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

- **la recv ritorna appena ci sono dati e non attende tutti i dati richiesti**

Il loop di recv interno 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

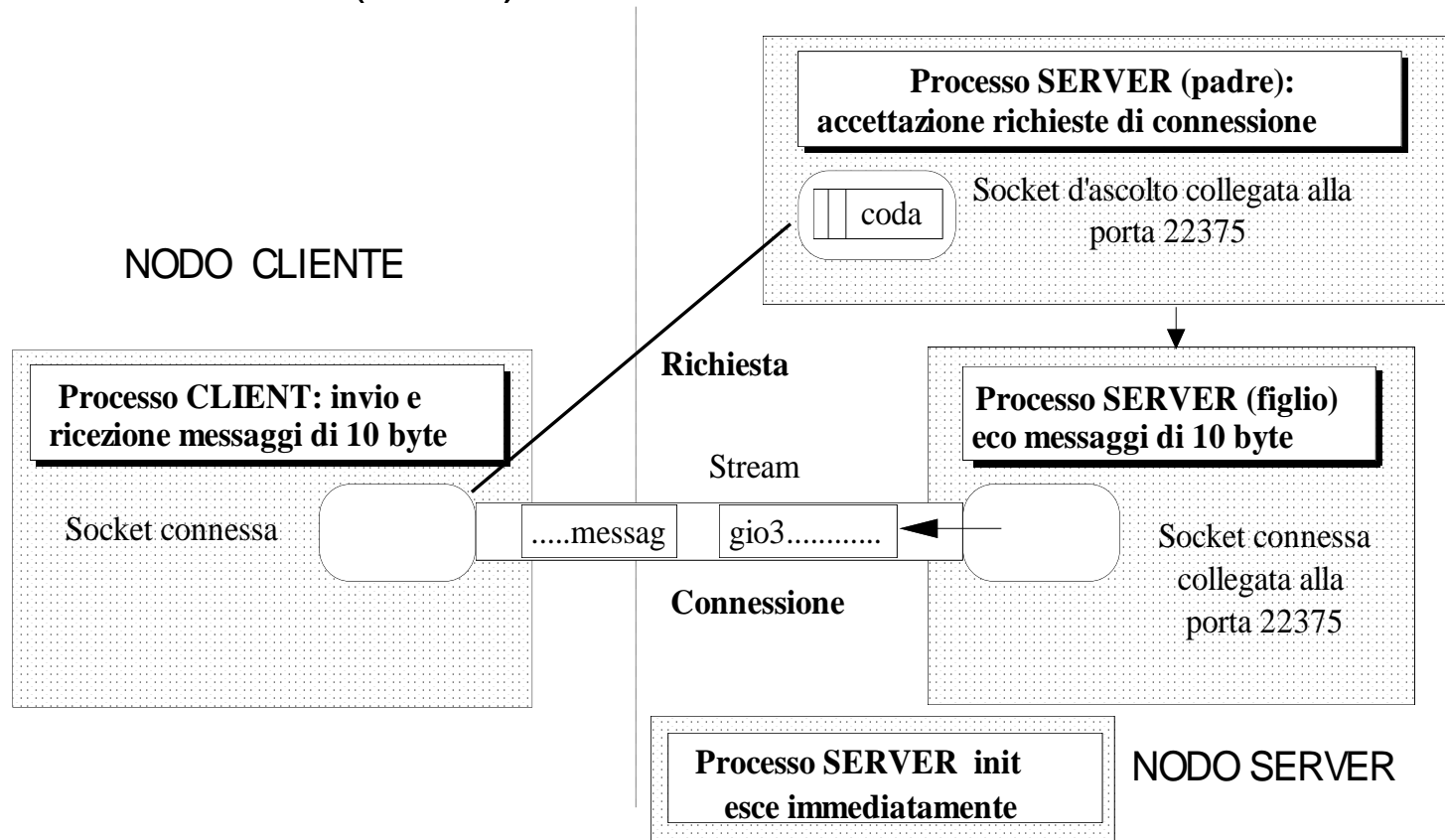
- Dobbiamo ricordare che la **ricezione** considera un successo un qualunque numero di byte ricevuto (anche 1) e ne segnala il numero nel risultato (a meno di opzioni: vedi **low watermark**)
- Per evitare tutti i problemi, dobbiamo **fare ricezioni / letture ripetute**
- *Esiste il modo di cambiare 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 l'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**

# 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



# 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 */
char * nomeprog;

main(argc, argv) int argc; char *argv[];
{int addrlen; nomeprog=argv[0]; /* si azzerano gli indirizzi
*/;

memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_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 su 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 = htons(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 */
long int 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 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 ci sia stata una interruzione da segnale di interruzione)

...

`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*/`

`}`

# MODALITÀ DELLE SOCKET

---

Le primitive di UNIX sono **sincrone bloccanti e prevedono sempre attesa del completamento da parte della driver di autorità**

**Opzioni delle socket di cui parliamo ...**

Abbiamo le opzioni per ottenere da una **singola socket dei comportamenti diversi dal default** e secondo qualche piccola alterazione del comportamento del kernel su alcuni aspetti

**Il progetto dei kernel richiede di avere invece di avere comportamenti fortemente diversi per ogni aspetto di una socket.**

Ci sono **operazioni di kernel**, come **ioctl** o **fctl** (I/O control e File control) che sono state definite per questo e permettono di **ottenere comportamenti asincroni o non bloccanti**

**L'uso di queste permette di avere progetti molto flessibili e di basso livello**



# OPZIONI PER LE SOCKET

---

## getsockopt() setsockopt()

funzioni di utilità per configurare socket, cioè leggere e variare le **modalità di utilizzo delle socket attraverso valori di attributi**

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

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

```
char optval = 1;
```

```
/* tipicamente il valore del campo vale 0 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;
```

<b>s</b>	⇒	socket descriptor legato alla socket
<b>level</b>	⇒	livello di protocollo per socket: <b>SOL_SOCKET</b>
<b>optname</b>	⇒	nome dell'opzione
<b>optval</b>	⇒	puntatore ad un'area di memoria per valore
<b>optlen</b>	⇒	lunghezza del quarto argomento

# OPZIONI PER LE SOCKET

---

Attraverso le opzioni sulla singola socket si possono 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 utilizzatore alla volta di un indirizzo locale (porta): ogni ulteriore viene bloccato (con fallimento)

con l'opzione, si richiede che la socket sia **senza controllo della unicità di associazione**

In particolare è utile in caso di **server** da riavviare in **caso di crash** e che devono essere operativi immediatamente (e senza ritardo)

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

Il processo che tenti di agganciarsi alla stessa porta per il riavvio, senza la opzione potrebbe incorrere in un fallimento fino a che la memoria della porta non fosse libera (e i dati tutti regolarmente inviati)

```
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 sola `close()`

- A default, il sistema **dopo la close** tende a **mantenere** la **memoria in uscita** anche per un lungo intervallo (per la consegna dei dati)

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** per TCP in risorse, `l_onoff == 0`

<code>l_onoff</code>	<code>l_linger</code>	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  $\Rightarrow$  ogni dato non inviato è buttato via

### chiusura graceful della connessione

- `l_onoff` ad 1 e `l_linger` valore positivo  $\Rightarrow$  la `close()` completa al massimo dopo il tempo in secondi specificato dal `linger` e (si spera) dopo la trasmissione di tutti i dati nel buffer (aggancio applicazione e trasporto)

# MODALITÀ DELLE SOCKET

---

Il **progetto dei kernel** richiede di avere invece di avere comportamenti fortemente **non standard per le socket**

Questa capacità potrebbe essere il progetto di *lettura o scrittura efficiente di driver di dispositivo che richiede comportamenti e gestioni molto asincroni* una differenza di comportamento rispetto ai normali modi di lavoro che **sono sincroni**

Quando faccio una primitiva aspetto il risultato (dal kernel)

Esistono **operazioni di kernel** come I/O control e File control che **permettono di ottenere comportamenti asincroni o non bloccanti su file** (e quindi anche per socket)

Questi strumenti sono stati definiti **per le driver di kernel che non possono lavorare in modo sincrono**

Ma possiamo usarli oltre che per le driver, per i file (descriptor) e le socket

# MODALITÀ DELLE SOCKET

---

Sono molto significativi modi **non sincroni bloccanti che non prevedano attesa del completamento**

Socket diventano 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 */)

```

**filedesc**     ⇒ 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)
```

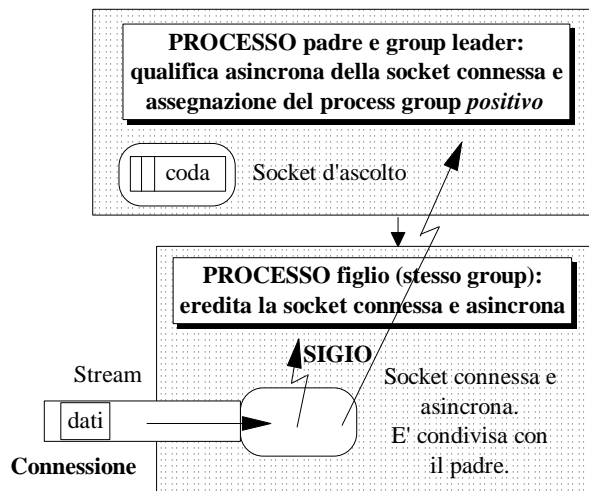
flag valore negativo  $\Rightarrow$

il segnale **solo ad un processo** con pid uguale al valore negativo

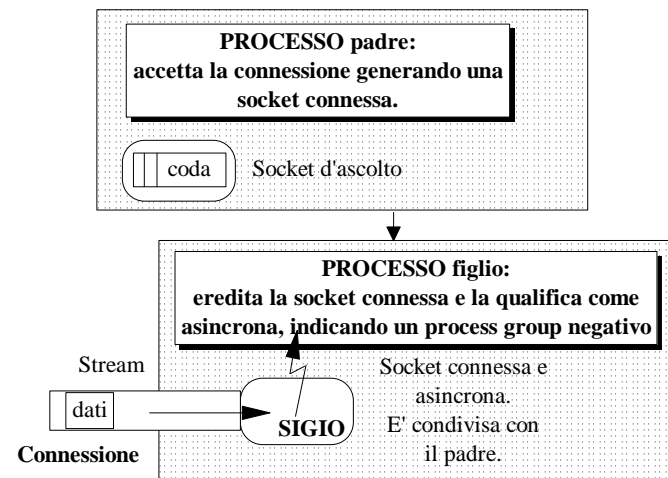
flag valore positivo  $\Rightarrow$

il segnale arriva **a tutti i processi** del process group

**valore positivo**



**valore negativo**





# ESEMPIO DI SOCKET RESA ASINCRONA

---

```
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,SIOCSGRP,&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

**Le primitive non fanno aspettare il processo che le invoca e va oltre**  
**Quando arriva il segnale, il solo processo può gestire il completamento**

# ALTRI ESEMPI ASINCRONI

---

**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`

# SOCKET NON BLOCCANTI

---

Se rendiamo una **socket** **asincrona o sincrona non bloccante** (meglio), ci troviamo in una situazione di uso in cui le operazioni vengono richieste e attuate, ma il **processo non si sospende in attesa della terminazione**

Nel caso di **azioni di lettura** ci serve a poco, poiché siamo interessati ad aspettare il dato che arriva da altri (e dobbiamo recuperare il risultato al **SIGIO**)

Per il caso di **azioni di scrittura**, noi possiamo comandarle, e non aspettare il termine (uso per select)

- In caso di **scrittura non bloccante**, possiamo confidare che la **operazione sia passata al livello sottostante della driver e confidiamo che abbia successo, anche se non sappiamo quando**
- Se siamo interessati a sapere se una operazione precedente è completata prima di **fare una ulteriore azione di scrittura** (senza ingolfare le driver): la informazione è il completamento della azione precedente (**SIGIO**) catturato anche dalla **select** (maschera output, vedi dopo)

# PRIMITIVE SU SOCKET

---

Le primitive su socket **bloccano il processo che le esegue**

## **Pensiamo alle send e receive**

Sono di comunicazione e hanno una **implementazione sincrona bloccante del processo nei confronti del kernel** su cui sono fatte

**Send** (write): aspetto fino a che non ho consegnato i dati al kernel

**Receive** (read): aspetto fino a che il kernel non ha alcuni dati

**Non sono sincrone bloccanti per il livello di processo**

**applicativo (NON succede MAI che il processo che fa send aspetti fino alla ricezione del processo ricevente)**

**Eseguire una primitiva può bloccare il processo che l'ha fatta**

**Se un processo deve fare molte azioni e non sa in che ordine farle, è difficile scegliere quale azione comandare tra azioni multiple**, visto non sappiamo se ci bloccheranno (blocco come azione locale)

# ATTESE MULTIPLE (MULTI RECEIVE)

---

Le primitive su socket possono **bloccare il processo** che le esegue  
**È impossibile esprimere di volere fare più di una azione, senza scegliere a priori quale sia, ma tenendole tutte aperte...**

In caso di possibile **ricezione da sorgenti multiple** (eventi multipli o primitive, ciascuno dei quali può bloccare il processo) è necessaria la **possibilità di attendere contemporaneamente su più eventi legati a socket diverse** (o file descriptor) **su cui le azioni da fare non siano bloccanti**

- Una **operazione bloccante** (lettura) o con attesa **pregiudica ad un processo server il servizio di altre operazioni** (con servizi pronti da svolgere)  
il server potrebbe sospendersi su una primitiva e non potere servire altre richieste su socket diverse
- La gestione delle socket usa una **nuova primitiva legata alla gestione di più eventi all'interno di uno stesso processo** e al blocco imposto dalle primitive sincrone

**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)**, anche con un **intervallo di timeout** definito dall'utente. **Gli eventi corrispondono alla possibilità di fare azioni non sospensive**

# EVENTI PER LA SELECT()

---

La `select()` invocata sospende il processo fino al primo evento o al timeout (se si specifica il timeout), e 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()` o `read()`
- 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()` o `write()`
- in una socket connessa il pari ha chiuso (`SIGPIPE`) o errore

Eventi **anomali ed eccezionali**: segnalano errore o urgenza

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

**Al termine della select, il processo può servire gli eventi (ossia fare una delle azioni conseguenti) senza bloccarsi**



# SELECT()

---

Primitiva di **attesa multipla sincrona** o con **durata massima**

**select()** primitiva con time-out intrinseco

Azioni di **comunicazione su socket** 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 che permettano di non sospendere il processo che le fa**

```
#include <time.h>
```

```
int select (nfd, readfds, writefds, exceptfds, timeout)
    size_t nfd; /* numero massimo di eventi attesi, inteso come
limite superiore al range delle socket e indicare quali bit
considerare*/
```

I parametri successivi sono passati per riferimento in/out

```
int *readfds, *writefds, *exceptfds;
/* tre puntatori a maschere di eventi (un bit ogni
filedescriptor) o anche null per indicare non interesse */
const struct timeval * timeout;
/* puntatore a time out massimo o null se attesa indefinita */
```

# INVOCAZIONE DELLA SELECT()

---

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

(i quattro ultimi parametri passati per indirizzo rif.)

- Si notino i parametri di input / output

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

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

All'invocazione, segnala nelle **maschere** gli **eventi di interesse e il tempo**

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

- Con azione **sospensiva bloccante sincrona** (**NULL nel timeout**) si attende **per sempre** (massimo reale 31 giorni)
- Con **timeout si attende al massimo** quanto specificato nel parametro `timeval`
- Con **azione non bloccante e passante**, si specifica **zero** come valore nel campo di 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 alle tre tipologie

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

**Prendendo come maschera in Input ed output la seguente**

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

- La select **esamina solo i file descriptor** il cui bit è ad **1**, qui per socket **4,5,7,9 (nfd è 10)**  
qui si sono verificati gli eventi **4,5,7**

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

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

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

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 a secondo delle diverse architetture

9	8	7	6	5	4	3	2	1	0	← file descriptor
0	0	1	0	1	1	0	0	0	0	← MASCHERA

```
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 tutti a zero

# MACRO SULLE MASCHERE

---

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

Sequenza di parole in memoria per la maschera

```
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 nfd, 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);
nfd=s+1;
nfd=select(nfd, &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 con CONNESSIONE

---

Si vuole progettare un server concorrente 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 lo rioccupa a partire dal valore libero più basso**

# USO DELLE SOCKET E MASK SELECT

---

**TABELLA dei file aperti del server dopo listen**

0	1	2	3	4	5	6	7
X	X	X	X	-	-	-	-

**Maschera temp\_hs per select dopo listen (maxhp 4)**

0	1	2	3	4	5	6	7
0	0	0	1	0	0	0	0

**Lo stato dei file aperti deve essere rispecchiato nella maschera della select che deve chiedere al kernel solo i sd corretti**



# Uso SOCKET E MASK SELECT

**TABELLA dei file aperti del server dopo 4 accept**

0 1 2 3 4 5 6 7

X	X	X	X	X	X	X	X
---	---	---	---	---	---	---	---

**Maschera temp\_hs per select congrua (maxhp 8)**

0 1 2 3 4 5 6 7

0	0	0	1	1	1	1	1
---	---	---	---	---	---	---	---



**Ogni accept apre il sd libero successivo (4, 5, 6, 7)**

**Ma deve organizzare le stesse richieste alla select con maschera consistente (set delle posizioni sulla maschera)**


# Uso SOCKET E MASK SELECT

TABELLA dei file aperti del server dopo la chiusura della connessione 5 (5 libero)

0	1	2	3	4	5	6	7
X	X	X	X	X	-	X	X

Maschera temp\_hs per select deve essere azzerata (maxhp 8)

0	1	2	3	4	5	6	7
0	0	0	1	1	0	1	1



Ogni chiusura produce un fd libero e disponibile per la prossima accept, che alla prossima connessione userà per la successiva il fd 5

# 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 name e len pronti per risposta */
  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 ? errno : 0;  
    /* value ha prodotto anche errno */  
    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); 3
    /* 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 temporaneo
    che si possa usare, cambiare valore, e ripristinare*/
    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);
  /* ciclo su socket aperte */
  for (h = listener + 1; h < maxhp1; h++) 4 - per ogni accept nuovi sd
  { if (FD_ISSET(h, &temp_hs))
    /* per ogni richiesta sulle connessioni,
       trattamento in handle() */
    if (handle (h) == 0)
    /* in caso di chiusura di connessione da parte del cliente */
    { FD_CLR(h, &read_hs); close(h); }
    } ad ogni close libera un sd
    if (FD_ISSET(listener, &temp_hs)) {
  /* nuova connessione */ ad ogni accept sd 4, 5, 6, ...
  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 descriptor della socket iniziale */
    h = socket (PF_INET, SOCK_STREAM, 0); sd 3 di listen
    /* 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**

- la funzione riceve la richiesta (letture senza sospensione)
- attua l'azione e invia la risposta
- a richiesta nulla, il cliente ha chiuso la connessione  $\Rightarrow$  si chiude \*/

```
long handle (HANDLE h)
```

```
{ struct Request req; long value;
```

```
  if (recv_request (h, &req) <= 0)    return 0;
```

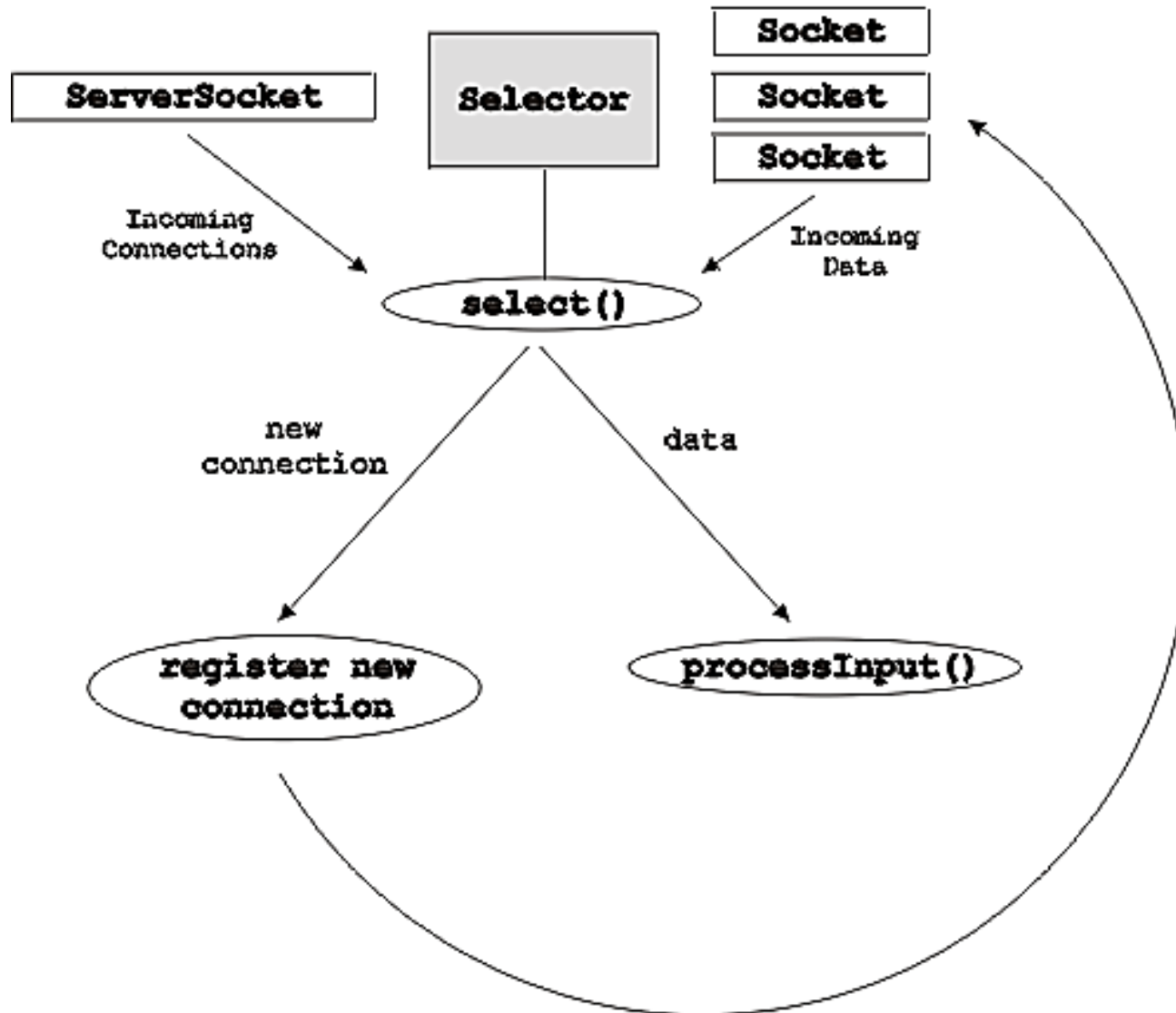
```
  value = action (&req); /* azione */
```

```
  return send_response (h, value);
```

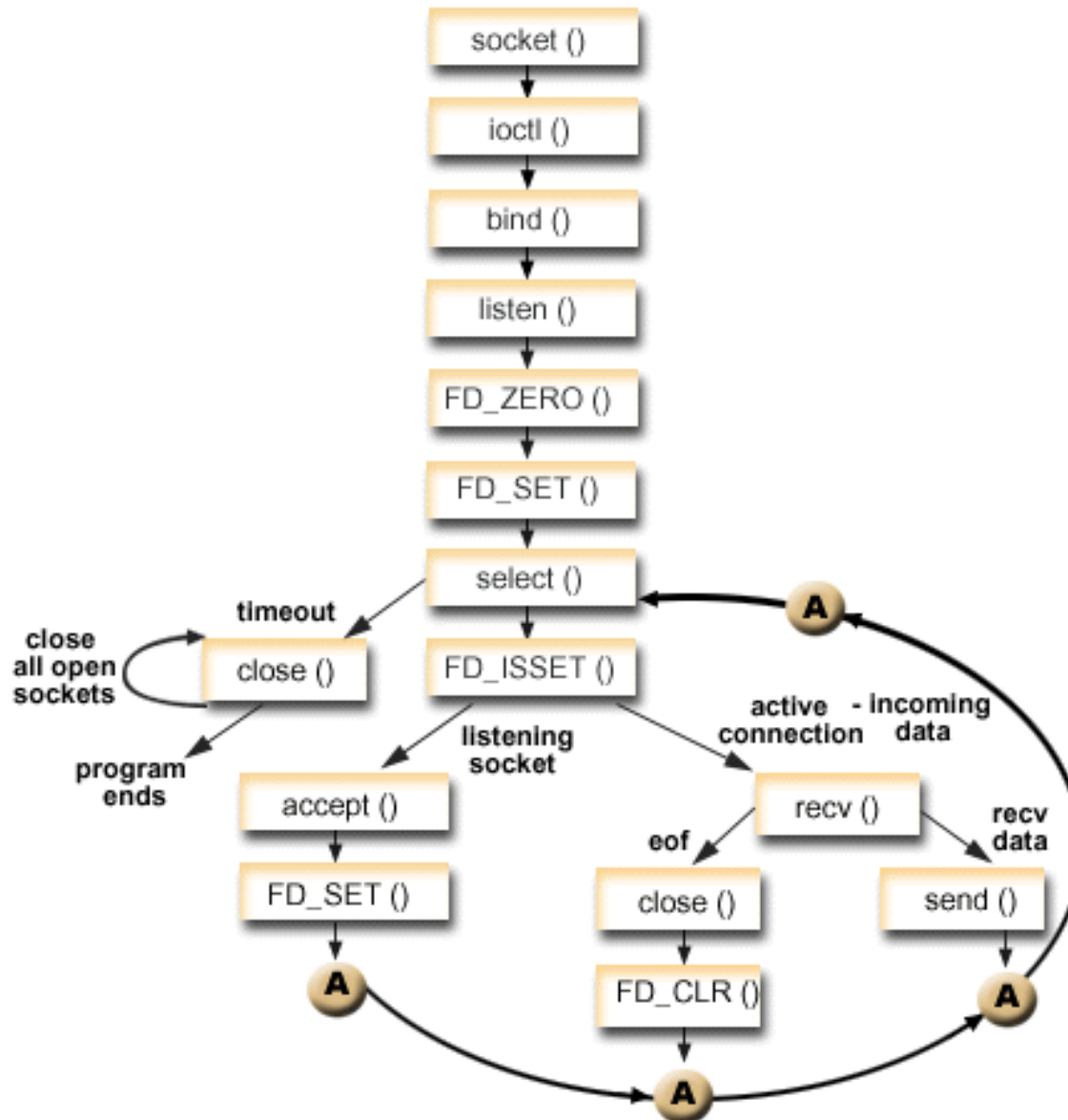
```
}
```



# ESEMPIO CICLO SELECT



# ESEMPIO CICLO SELECT



# SELECT IN SCRITTURA

---

La **select** prevede eventi di **lettura** (e anomali) e **scrittura**

La **lettura** (e anomali) prevede eventi di scrittura causati da altre entità con cui si può interagire (datagrammi inviati, dati inviati, richiesta inviate...)

Per la **scrittura** invece abbiamo il fatto che la scrittura dipende in toto da **chi fa l'azione di scrittura** (a default con azione **sincrona bloccante**): quindi l'evento potrebbe solo avvenire dopo l'azione che eventualmente lo blocca

- Per cui la **maschera di scrittura** serve **solo in caso di socket in modalità asincrona o non bloccante**
- In questo caso, la select segnala l'evento **di completamento della azione di scrittura precedente** (e permette a chi fa la gestione con la select **di fare la azione successiva** alla select e di comandare altre azioni)

# 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

Un solo 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 in modo diretto (servizi interni) ed altri li delega a processi creati su richiesta
- **inetd** definisce un linguaggio di configurazione per specificare il proprio 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 synchronization.
time  stream tcp    nowait    root  internal
time  dgram  udp     wait      root  internal
```

# CONFIGURAZIONE INETD

```
# echo, discard, daytime, ... chargen are mainly 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/1-2dgram  rpc/udp  wait  root  /usr/etc/rpc.rusersd
rpc.rusersd

```

```
# The spray server is used primarily for testing.
```

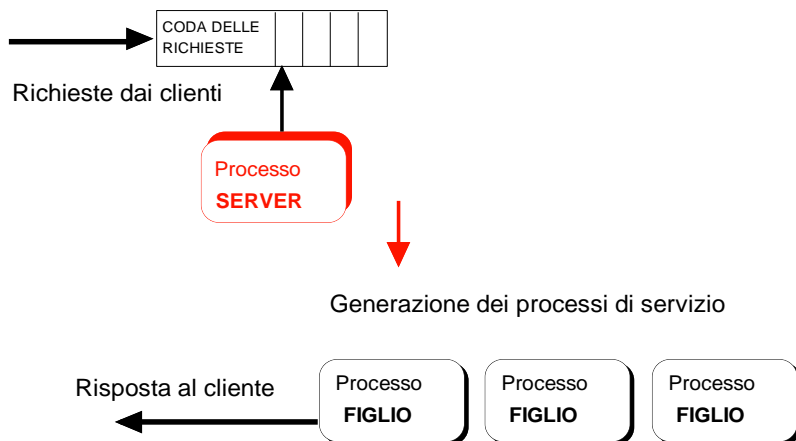
```
sprayd/1    dgram    rpc/udp    wait    root    /usr/etc/rpc.sprayd
            rpc.sprayd
```

# SERVITORE CONCORRENTE

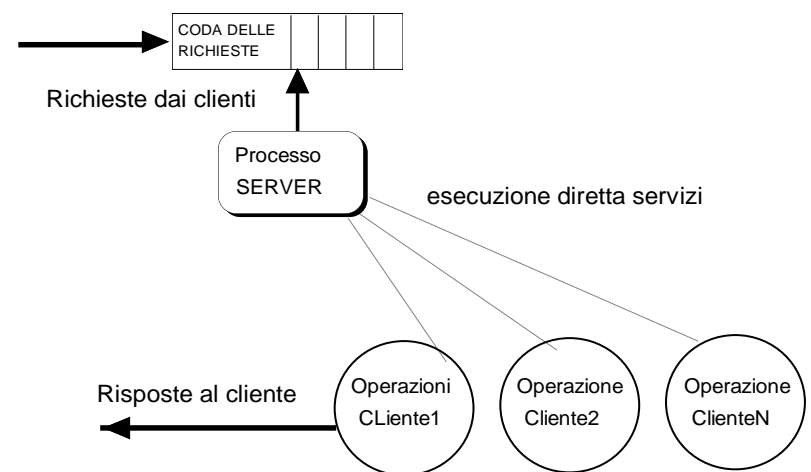
Il progetto di un **servitore concorrente** può seguire due schemi

**Servitore concorrente multiprocesso:** un processo server si occupa della coda delle richieste e genera processi figli, uno per ogni servizio ( $T_G$ )

**Servitore concorrente monoprocesso:** un unico processo server usa la select e si divide tra il servizio della coda delle richieste dei clienti e le operazioni vere e proprie



**servitore concorrente  
multiprocesso**



**servitore concorrente  
singolo processo**

# 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 (**invio senza attesa delle richieste e select per ricezione risposte**)

- 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

