SC per Inter Process Comminication

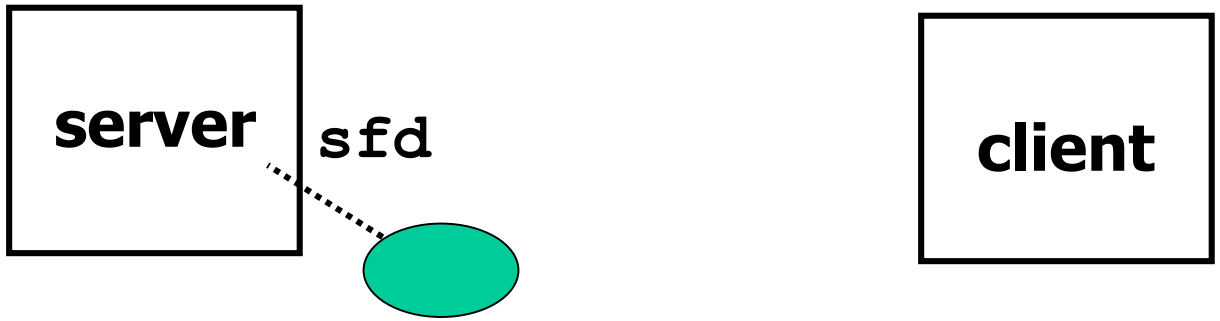
Comunicazione fra macchine diverse: socket

Sockets

- File speciali utilizzati per connettere due o più processi con un canale di comunicazione
  - i processi possono risiedere su macchine diverse
- le SC che lavorano sui socket sono più complesse di quelle che lavorano sulle pipe
  - ci sono diversi tipi di socket
    - ci concentreremo su quelli con connessione
  - c'è asimmetria fra processi client e server
  - le diverse operazioni realizzate dalla **`mkfifo()`** e dalle **`open()`** sono divise fra più system call

Sockets (2)

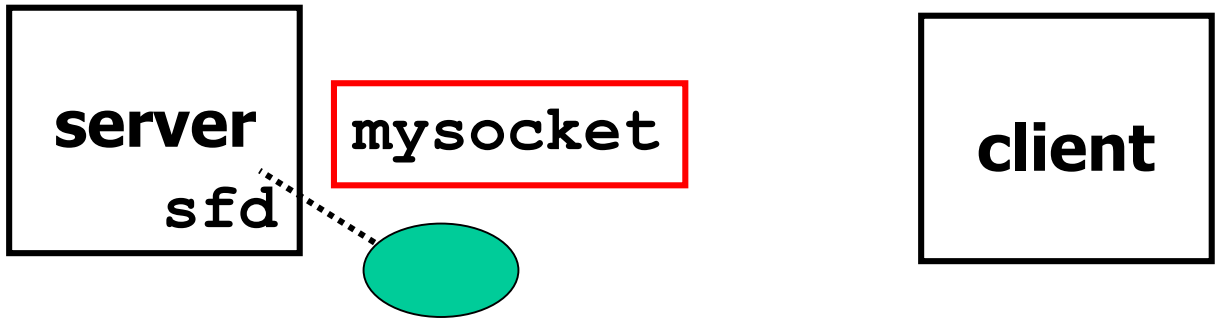
- Come collegare due processi con un socket:
  - Server: passo 1: creare un file 'ricevitore' (*socket*) e allocare un file descriptor (SC socket)




```
sfd=socket(...)
```

Sockets (3)

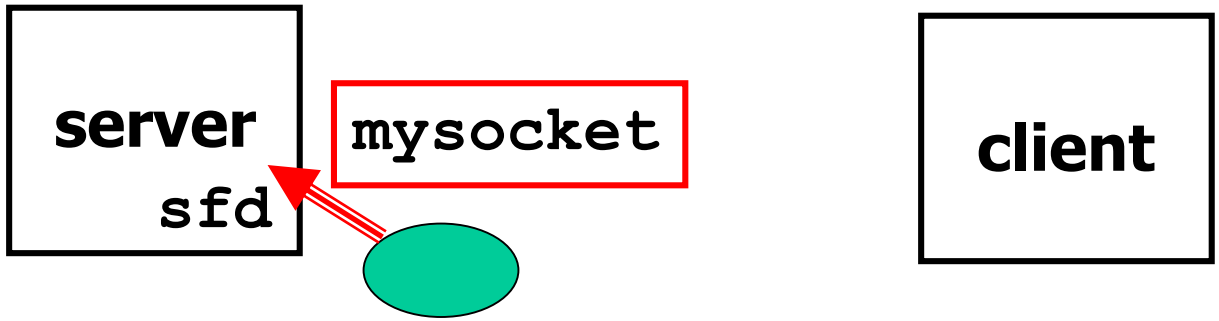
- Come collegare due processi con un socket:
  - Server: passo 2: associare il socket `sfd` con un ***indirizzo*** (**`mysocket`**), (SC bind)
    - ci sono diversi tipi di indirizzi, AF_UNIX, AF_INET ...




```
bind(sfd,"mysocket",...)
```

Sockets (4)

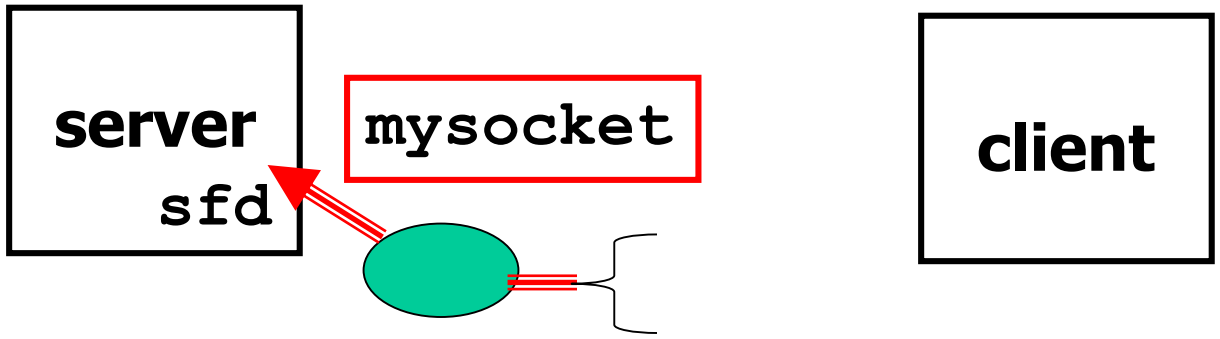
- Come collegare due processi con un socket:
  - Server: passo 3: specificare che sul socket siamo disposti ad accettare connessioni da parte di altri processi (SC listen)




```
listen(sfd,...)
```

Sockets (5)

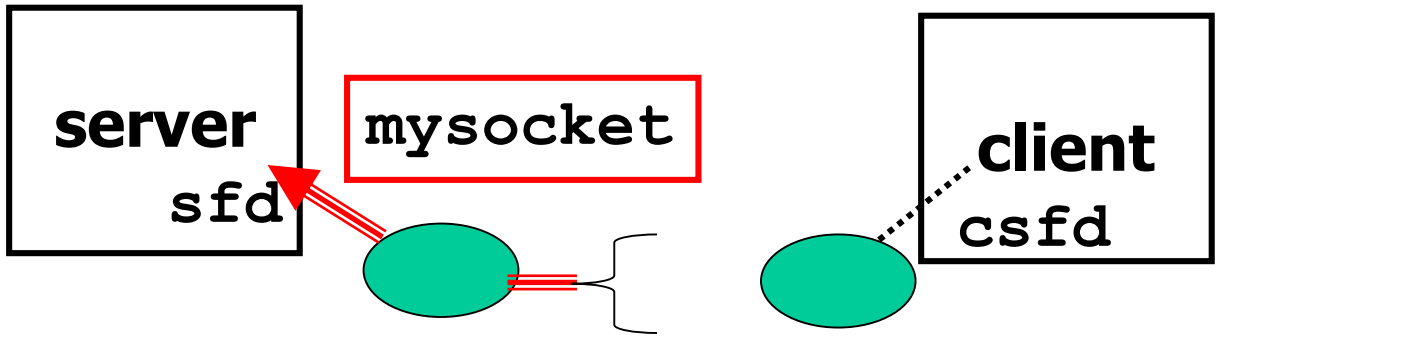
- Come collegare due processi con un socket:
  - Server: passo 4: bloccarsi in attesa di richieste di connessioni da parte di altri processi (SC accept)




```
accept(sfd,...)
```

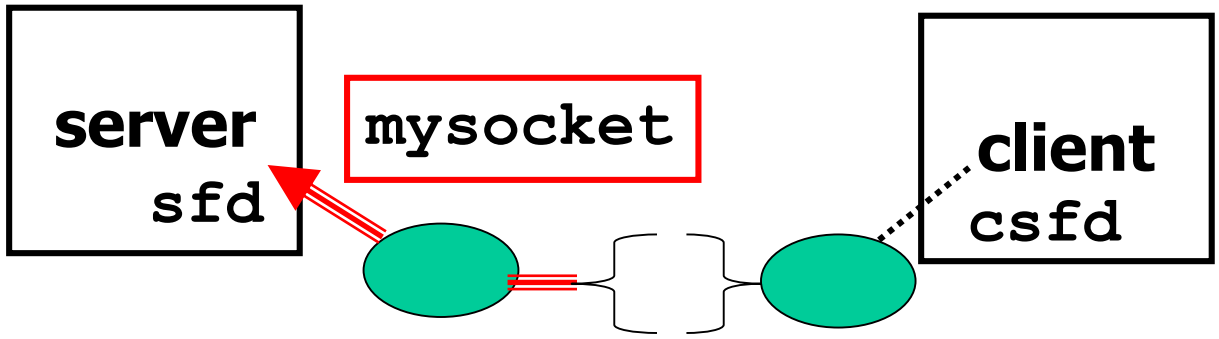
Sockets (6)

- Come collegare due processi con un socket:
  - Client: passo 1: creare un socket ed il relativo file descriptor (SC socket)



Sockets (7)

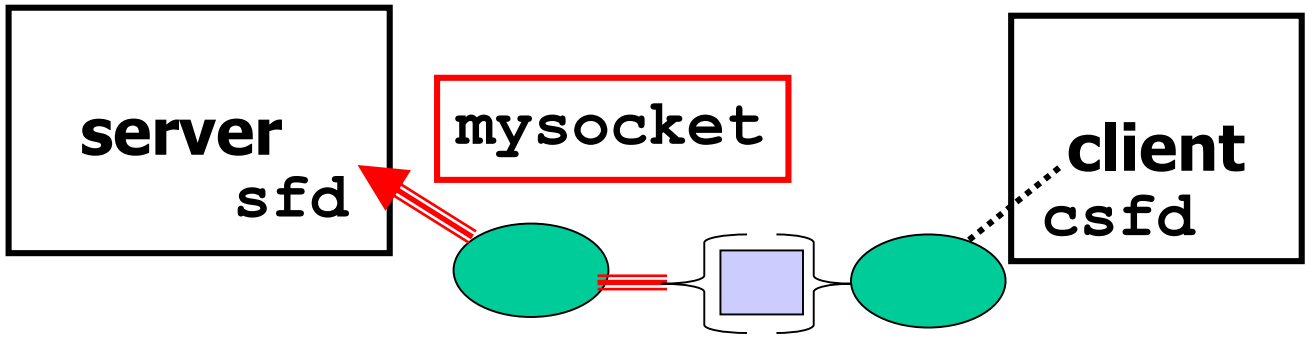
- Come collegare due processi con un socket:
  - Client: passo 2: collegarsi con il socket del server usando il nome esportato con la bind (SC connect)



`accept(sfd...)` `connect(csfd,"mysocket"...)`

Sockets (8)

- Come collegare due processi con un socket:
  - Client: passo 2: c'è un match fra accept e connect, viene creato un nuovo socket

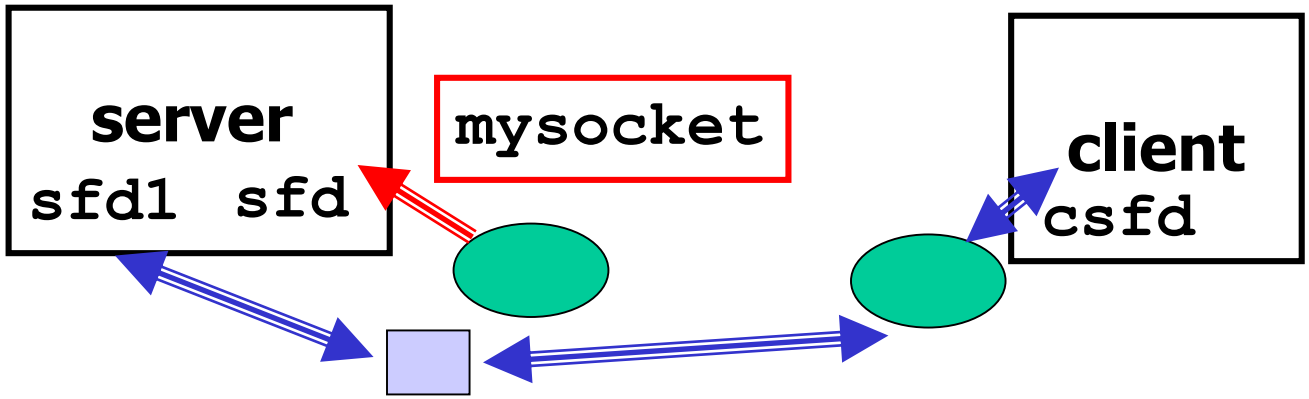



```
sfd1=accept(sfd...)
```

```
connect(csfd,"mysocket"...)
```

Sockets (9)

- Come collegare due processi con un socket:
  - la accept ritorna al server il descrittore del nuovo socket usabile per il resto della comunicazione con il client

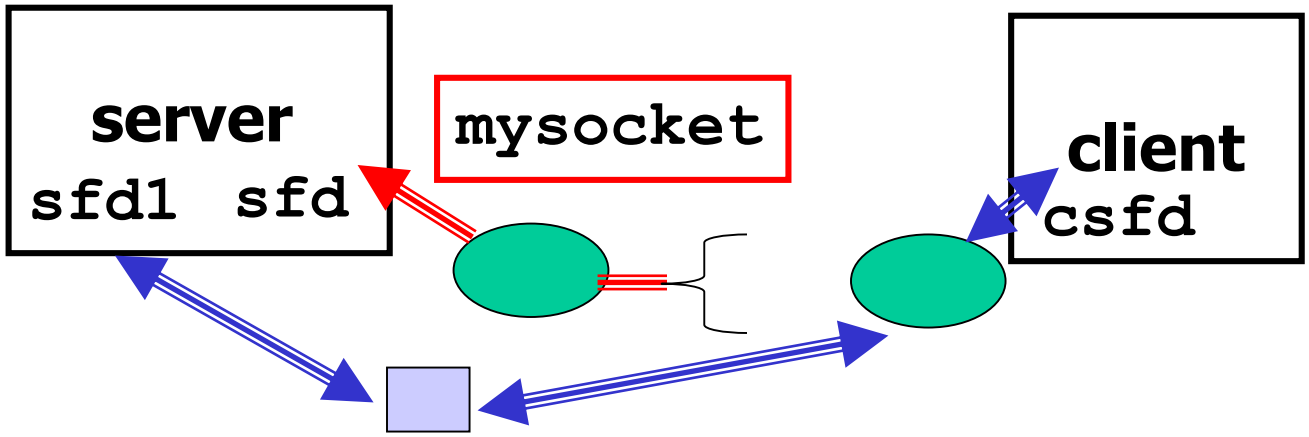



```
write(sfd1...)
read(sfd1 . . .)
close(sfd1)
```

```
write(csfd...)
read(csfd . . .)
close(csfd)
```

Sockets (10)

- Come collegare due processi con un socket:
  - il server si può rimettere in attesa di connessioni da altri client con un'altra accept




```
sfd2=accept(sfd...)
```

Creazione: socket()

```
#include <sys/socket.h>
int socket(
 int domain, /*dominio: AF_UNIX, AF_INET … */
 int type,   /* SOCK_STREAM, SOCK_DGRAM … */
 int protocol
 /* sem dipende dal tipo, di solito 0 … */
);
/* (sfd) success (-1) error, sets errno */
```

- crea un socket, il file descriptor ritornato può essere usato dal server per accettare nuove connessioni e dal client per inviare i dati

Creazione: socket() (2)

```
fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
```

- dominio:
  - deve essere lo stesso nella **`bind()`** successiva
  - ce ne sono molti, per ora consideriamo **`AF_UNIX`** , file Unix usabili sulla stessa macchina
  - a volte **`PF_UNIX`** invece di **`AF_UNIX`** : noi usiamo i nomi definiti dallo standard
  - accenneremo poi ad altri tipi di domini

Creazione: socket() (3)

```
fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
```

- tipo di connessione
  - ce ne sono diversi, connected, connectionless, ci concentriamo sui *connected*
  - dipendono dal dominio
- protocollo
  - se si può scegliere (dipende dal dominio)
  - 0 default

Socket: esempio

```
/* un client ed un server che comunicano via
   socket AF_UNIX : per semplicità di lettura
  la gestione degli errori non è mostrata*/
#include <sys/types.h>
#include <sys/socket.h>
…
int main (void){
   int fd_skt, fd_c;

  /* creazione indirizzo */
      ……
  /* segue ……  */
```

Socket: esempio (2)

```
  if (fork()!=0) { /* padre, server */
      /* creazione server socket ……  */
      fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
  …… /* segue ……  */
  }
  else { /* figlio, client */
    /* creazione client socket ……  */
    fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
   …… /* segue ……  */ }
}
```

Dare il nome: bind()

```
#include <sys/socket.h>
int bind(
 int sock_fd,          /*file descr socket … */
 const struct sockaddr * sa, /* indirizzo */
 socklen_t sa_len,     /* lung indirizzo */
);
/* (0) success (-1) error, sets errno */
```

- assegna un indirizzo a un socket
- tipicamente in AF_UNIX serve un nome nuovo, non si può riusare un file esistente

Indirizzi AF_UNIX

```
#include <sys/un.h>
/* struttura che rappresenta un indirizzo */
/* man 7 unix */

#define UNIX_PATH_MAX 108

struct sockaddr_un {
    sa_family_t sun_family; /* AF_UNIX */
    /* pathname socket */
    char sun_path[UNIX_PATH_MAX];
};
```

Socket: esempio (3)

```
  /* ind AF_UNIX : ogni famiglia di indirizzi
     ha i suoi include e le sue strutture*/
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>          /* ind AF_UNIX */
#define UNIX_PATH_MAX 108
#define SOCKNAME "./mysock"
int main (void) {
 int fd_skt, fd_c;
 struct sockaddr_un sa;
 strncpy(sa.sun_path, SOCKNAME,UNIX_PATH_MAX);
 sa.sun_family=AF_UNIX;
 /* segue */
```

Socket: esempio (4)

```
   /* il server esegue bind */


if (fork()!=0) { /* padre, server */
      fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
      bind(fd_skt,(struct sockaddr *)&sa,
            sizeof(sa));
      …… }
}
 /* segue */
```

Socket: esempio (5)

```
if (fork()!=0) { /* padre, server */
   fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
   bind(fd_skt,(struct sockaddr *)&sa,
         sizeof(sa));
   ……

}

}
```

Cast necessario:
struct sockaddr* è il 'tipo generico'

Accettare connessioni: listen()

```
#include <sys/socket.h>
int listen(
 int sock_fd,  /*file descr socket … */
 int backlog, /* n max connessioni in coda */
);
/* (0) success (-1) error, sets errno */
```

- segnala che il socket accetta connessioni
- limita il massimo di richieste accodabili
  - **SOMAXCONN** è il massimo consentito
- Va chiamata solo una volta prima di iniziare!

Socket: esempio (6)

```
.....
    if (fork()!=0) { /* padre, server */
      fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
      bind(fd_skt,(struct sockaddr *)&sa,
            sizeof(sa));
      listen(fd_skt,SOMAXCONN);
      ......

    }

}
```

Accettare connessioni: accept()

```
#include <sys/socket.h>
int accept(
    int sock_fd,  /*file descr socket … */
    const struct sockaddr * sa,
       /* indirizzo o NULL */
    socklen_t sa_len, /* lung indirizzo */
);
/* (fdc) success (-1) error, sets errno */
```

- se non ci sono connessioni in attesa si blocca
- altrimenti accetta una della connessioni in coda, crea un nuovo socket per la comunicazione (e ritorna **`fdc`**, nuovo file descriptor)

Accettare connessioni: accept() (2)

- `fdc = accept(sock_fd,psa,lung);`

  - se **`psa!=NULL`**, al ritorno contiene l'indirizzo del socket che ha accettato la connessione (e **`lung`** è la lunghezza della struttura -- inout param)
  - altrimenti si può specificare **`NULL`**, e 0 come lunghezza in questo caso non viene restituito niente.
  - **`fdc`** è il file descriptor del nuovo socket che sarà usato nella comunicazione

Socket: esempio (6)

```
.....
    if (fork()!=0) { /* padre, server */
      fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
      bind(fd_skt,(struct sockaddr *)&sa,
            sizeof(sa));
      listen(fd_skt,SOMAXCONN);
      fd_c = accept(fd_skt,NULL,0);
      .....

    }

}
```

Connettere un client: connect()

```
#include <sys/socket.h>
int connect(
    int sock_fd,  /*file descr socket … */
    const struct sockaddr * sa, /* indirizzo */
    socklen_t sa_len, /* lung indirizzo */
);
/* (0) success (-1) error, sets errno */
```

- è lo stesso indirizzo usato nella **`bind()`**
- quando si sblocca **`sock_fd`** può essere utilizzato per eseguire I/O direttamente

Socket: esempio completo

```
/* altri include omessi */
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <sys/un.h>          /* ind AF_UNIX */
 #define UNIX_PATH_MAX 108    /* man 7 unix */
 #define SOCKNAME "./mysock"
 #define N 100
 int main (void) {
  int fd_skt, fd_c; char buf[N];
  struct sockaddr_un sa;
  strncpy(sa.sun_path, SOCKNAME,UNIX_PATH_MAX);
  sa.sun_family=AF_UNIX;
  /* segue */
```

Socket: esempio completo (2)

```
if (fork()!=0) { /* padre, server */ }
else { /* figlio, client */
    fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
    while (connect(fd_skt,(struct sockaddr*)&sa,
            sizeof(sa)) == -1 ) {
        if ( errno == ENOENT )
            sleep(1); /* sock non esiste */
        else  exit(EXIT_FAILURE); }
    write(fd_skt,"Hallo!",7);
    read(fd_skt,buf,N);
    printf("Client got: %s\n",buf) ;
    close(fd_skt);
    exit(EXIT_SUCCESS); } /* figlio terminato */
```

Socket: esempio completo (3)

```
if (fork()!=0) { /* padre, server */
    fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
    bind(fd_skt,(struct sockaddr *)&sa,
          sizeof(sa));
    listen(fd_skt,SOMAXCONN);
    fd_c=accept(fd_skt,NULL,0);
    read(fd_c,buf,N);
    printf("Server got: %s\n",buf) ;
    write(fd_c,"Bye!",5);
    close(fd_skt);
    close(fd_c);
    exit(EXIT_SUCCESS);  }
}
```

Socket: esempio completo (4)

```
bash:~$ gcc -Wall -pedantic clserv.c -o clserv

bash:~$ ./clserv
-- un po'di attesa
Server got: Hallo!
Client got: Bye!
bash:~$
```

Socket: commenti

- Nell'esempio visto manca completamente la gestione dell'errore, che deve essere effettuata al solito modo
- Una volta creata la connessione si possono leggere e scrivere i dati come con una pipe
- Si può fare molto di più dell'esempio:
  - gestire più client contemporaneamente
  - indirizzi AF_INET: comunicazione con macchine diverse
  - usare comunicazioni connectionless (datagram) senza creare una connessione permanente
  - gestire macchine con diversi *endians*
  - gestire dettagli del comportamento dei socket, accedere al database dei nomi (eg. *www.unipi.it*)

Socket: gestire più client

- Il problema:
  - Dopo aver creato la prima connessione il server deve contemporaneamente:
    - mettersi in attesa sulla **`accept()`** per una nuova connessione
    - mettersi in attesa con **`read()`** dei messaggi in arrivo dai clienti già connessi
  - se scelgo l'attesa sbagliata posso avere poca efficienza o deadlock
  - posso decidere di usare I/O asincrono + sleep() ma diventa molto difficile da programmare e lo stesso non efficiente

Socket: gestire più client (2)

- Due tipiche soluzioni:
  - usare un server multithreaded,
    - un thread *dispatcher* che si mette in attesa ripetutamente su **`accept()`** per una nuova connessione
    - ogni volta che la connessione è stata stabilita viene creato un nuovo thread *worker* che si mette in attesa con **`read()`** dei messaggi in arrivo solo dal cliente relativo ad una particolare connessione
  - usare un server sequenziale e la SC **`select()`** che permette di capire quale file descriptor è 'pronto'
    - discuteremo in dettaglio questa soluzione

Aspettare I/O da più fd: select()

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

int select(
 int nfds,   /*numero descr. di file*/
 fd_set * rdset,  /*insieme lettura o NULL*/
 fd_set * wrset,  /*ins. scrittura o NULL*/
 fd_set * errset,  /*ins. errore o NULL*/
 struct timeval* timeout /* timeout o NULL */
);
/* (numero di bit settati) on success (-1) on
   error, sets errno */
```

Aspettare I/O da più fd: select() (2)

- semantica
  - si blocca finchè uno dei descrittori specificati è pronto per essere usato in una SC senza bloccare il processo:
    - un descrittore **`fd`** atteso per la lettura (**`rdset`**) è considerato ‘pronto’ quando una **`read`** su **`fd`** non si blocca, quindi se ci sono dati o se è stato chiuso (con EOF la **`read`** non si blocca)
    - un descrittore **`fd`** atteso per la scrittura (**`wrset`**) è considerato ‘pronto’ quando una **`write`** su **`fd`** non si blocca
    - un descrittore **`fd`** atteso per eccezioni (**`errset`**) è considerato ‘pronto’ quando viene notificata una eccezione
  - all’uscita della **`select()`** i set sono modificati per indicare chi è pronto.

Aspettare I/O da più fd: select() (3)

- Come specificare i set
  - **`rdset`**, **`wrset`**, **`errset`** sono maschere di bit e vengono manipolate usando delle macro:
    - **`FD_ZERO(&fdset)`** azzera la maschera **`fdset`**
    - **`FD_SET(fd,&fdset)`** mette a 1 il bit corrispondente al file descriptor **`fd`** nella maschera **`fdset`**
    - **`FD_CLR(fd,&fdset)`** mette a 0 il bit corrispondente al file descriptor **`fd`** nella maschera **`fdset`**
    - **`FD_ISSET(fd,&fdset)`** vale 1 se il bit corrispondente al file descriptor **`fd`** nella maschera **`fdset`** era a 1 e vale 0 altrimenti

Aspettare I/O da più fd: select() (4)

- Come specificare i set: esempio
  - tipicamente si azzera un set e poi si mettono a 1 i bit che interessano:

```
-- creare la maschera di attesa per il
-- primo fra fd1 e fd2 pronto in lettura
fd_set rdset;

FD_ZERO(&rdset);
FD_SET(fd1,&rdset);
FD_SET(fd2,&rdset);
```

Aspettare I/O da più fd: select() (5)

- Come controllare quale è ‘pronto’
  - non c’è modo di avere la lista, si devono testare tutti separatamente con **FD_ISSET**

```
-- quale fra fd1 e fd2 è pronto in lettura

if (FD_ISSET(fd1,&rdset)) {
   read(fd1,buf,N);
}
else if (FD_ISSET(fd2,&rdset)) {
   read(fd2,buf,N);
}
.....
```

Select(): esempio

- Modifichiamo l'esempio client server in modo da gestire 4 client

  - discutiamo prima il codice del server
  - poi il client
  - infine il main di attivazione …

Select: esempio (2)

```
static void run_server(struct sockaddr * psa) {
  int fd_sk, /* socket di connessione */
     fd_c, /* socket di I/O con un client */
     fd_num=0, /* max fd attivo */
     fd; /* indice per verificare risultati
            select */
  char buf[N]; /* buffer messaggio */
  fd_set set, /* l'insieme
                 dei file descriptor attivi */
     rdset; /* insieme fd attesi in lettura */
  int nread; /* numero caratteri letti */
```

Select : esempio (3)

```
static void run_server(struct sockaddr * psa) {
 int fd_sk, fd_c, fd_num=0, fd;
 char buf[N]; fd_set set, rdset; int nread;
 fd_sk=socket(AF_UNIX,SOCK_STREAM,0);
 bind(fd_sk,(struct sockaddr *)psa,sizeof(*psa);
 listen(fd_sk,SOMAXCONN);
 /* mantengo il massimo indice di descrittore
  attivo in fd_num */
 if (fd_sk > fd_num) fd_num = fd_sk;
 FD_ZERO(&set);
 FD_SET(fd_sk,&set);
```

Select : esempio (4)

```
static void run_server(struct sockaddr * psa) {
 int fd_sk, fd_c, fd_num=0, fd;
 char buf[N]; fd_set set, rdset; int nread;
 fd_sk=socket(AF_UNIX,SOCK_STREAM,0);
 bind(fd_sk,(struct sockaddr *)psa,sizeof(*psa);
 listen(fd_sk,SOMAXCONN);
 if (fd_sk > fd_num) fd_num = fd_sk;
 FD_ZERO(&set);
 FD_SET(fd_sk,&set);
 while (1) {
  rdset=set; /* preparo maschera per select */
  if (select(fd_num+1,&rdset,NULL,NULL,NULL)==-
  1)) {/* gest errore */ }
```

Select : esempio (4.1)

```
static void run_server(struct sockaddr * psa) {
 int fd_sk, fd_c, fd_num=0, fd;
 char buf[N]; fd_set set, rdset; int nread;
 fd_sk=socket(AF_UNIX,SOCK_STREAM,0);
 bind(fd_sk,(struct sockaddr *)psa,sizeof(*psa);
 listen(fd_sk,SOMAXCONN);
 if (fd_sk > fd_num) fd_num = fd_sk;
 FD_ZERO(&set);
 FD_SET(fd_sk,&set);
 while (1) {
  rdset=set;
  if (select(fd_num+1,&rdset,NULL,NULL,NULL)==-
  1)) {/* gest errore */ }
```

Attenzione al '+ 1', vogliamo il numero dei descrittori attivi, non l'indice massimo

Select : esempio (4.2)

```
static void run_server(struct sockaddr * psa) {
 int fd_sk, fd_c, fd_num=0, fd;
 char buf[N]; fd_set set, rdset; int nread;
 fd_sk=socket(AF_UNIX,SOCK_STREAM,0);
 bind(fd_sk,(struct sockaddr *)psa,sizeof(*psa));
 listen(fd_sk,SOMAXCONN);
 if (fd_sk > fd_num) fd_num = fd_sk;
 FD_ZERO(&set);
 FD_SET(fd_sk,&set);
 while (true) {
  rdset=set;
  if (select(fd_num+1,&rdset,NULL,NULL,NULL)==-
  1)) {/* gest errore */ }
```

Bisogna inizializzare ogni volta rdset perchè la select lo modifica

Select : esempio (5)

```
else { /* select OK */
  for (fd = 0; fd<=fd_num;fd++) {
    if (FD_ISSET(fd,&rdset)) {
      if (fd== fd_sk) {/* sock connect pronto */
         fd_c=accept(fd_sk,NULL,0);
         FD_SET(fd_c, &set);
         if (fd_c > fd_num) fd_num = fd_c; }
      else {/* sock I/0 pronto */
        nread=read(fd,buf,N);
        if (nread==0) {/* EOF client finito */
          FD_CLR(fd,&set);
          fd_num=aggiorna(&set);
          close(fd); }
```

Select : esempio (6)

```
        else { /* nread !=0 */
            printf("Server got: %s\n",buf) ;
            write(fd,"Bye!",5);
        }
    } } } } /* chiude while(1) */

} /* chiude run_server */
```

Select: esempio (7)

```
static void run_client(struct sockaddr * psa) {
 if (fork()==0) { /* figlio, client */
     int fd_skt; char buf[N];
     fd_skt=socket(AF_UNIX,SOCK_STREAM,0);
     while (connect(fd_skt,(struct sockaddr*)psa,
             sizeof(*psa)) == -1 ) {
        if ( errno == ENOENT )  sleep(1);
        else exit(EXIT_FAILURE); }
       write(fd_skt,"Hallo!",7);
       read(fd_skt,buf,N);
       printf("Client got: %s\n",buf) ;
       close(fd_skt);
       exit(EXIT_SUCCESS);
}} /* figlio terminato */
```

Select: esempio (8)

```
...
#include <sys/select.h>
#include <sys/un.h> /* ind AF_UNIX */
#define SOCKNAME "./mysock"
#define N 100
int main (void){
  int i; struct sockaddr_un sa; /* ind AF_UNIX */
  strcpy(sa.sun_path, SOCKNAME);
  sa.sun_family=AF_UNIX;
  for(i=0; i< 4; i++)
    run_client(&sa); /* attiv client*/
  run_server (&sa); /* attiv server */
  return 0;
}
```

Select: esempio (9)

```
bash:~$ gcc -Wall -pedantic clserv.c -o clserv

bash:~$ ./clserv
Server got: Hallo!
Server got: Hallo!
Client got: Bye!
Client got: Bye!
Server got: Hallo!
Server got: Hallo!
Client got: Bye!
Client got: Bye!
-- il server rimane attivo
```

Select: esempio (10)

- Ancora qualcosa:
  - un modo per terminare il server è utilizzando i segnali (es. SIGINT -- CTRL-C) è possibile personalizzare la gestione del segnale in modo da farlo terminare gentilmente
    - ne parleremo nella lezione sui segnali ...
  - Sono disponibili altre SC per il controllo dei descrittori pronti. Es. **`pselect, poll`**
    - non le vedremo

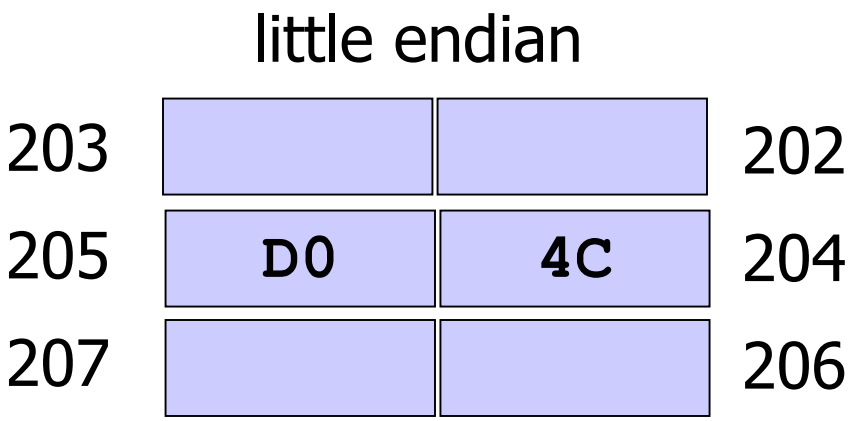
Socket AF_INET

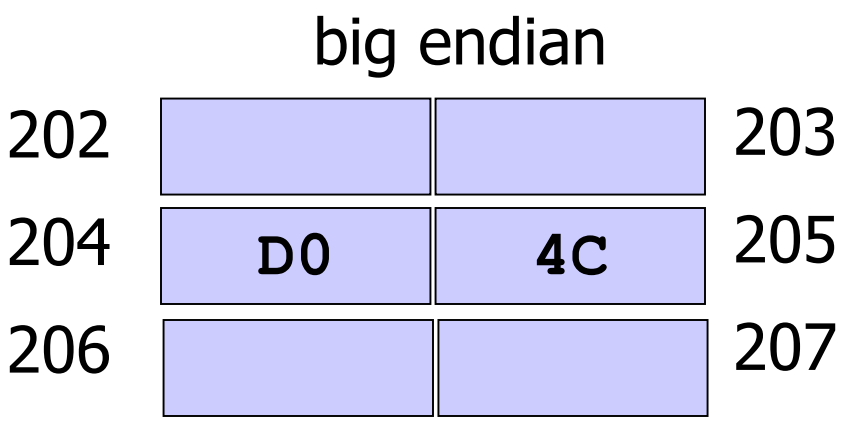
- I socket con indirizzi AF_UNIX possono essere usati solo sulla stessa macchina
- Ci sono molte altre famiglie di indirizzi
- Accenneremo ad AF_INET
  - famiglia di indirizzi che permette di far comunicare i socket su Internet
  - vedremo come usarli, dettagli su come funziona Internet al corso di reti

Big and little endian

Se comunichiamo una sequenza di byte fra macchine diverse:

- la comunicazione preserva l'ordine dei byte (quindi non ci sono problemi con le stringhe)
- Ci sono però due modi di rappresentare i numeri binari su più byte: es **`D04C`** esadecimale, indirizzo 204 (2 byte)

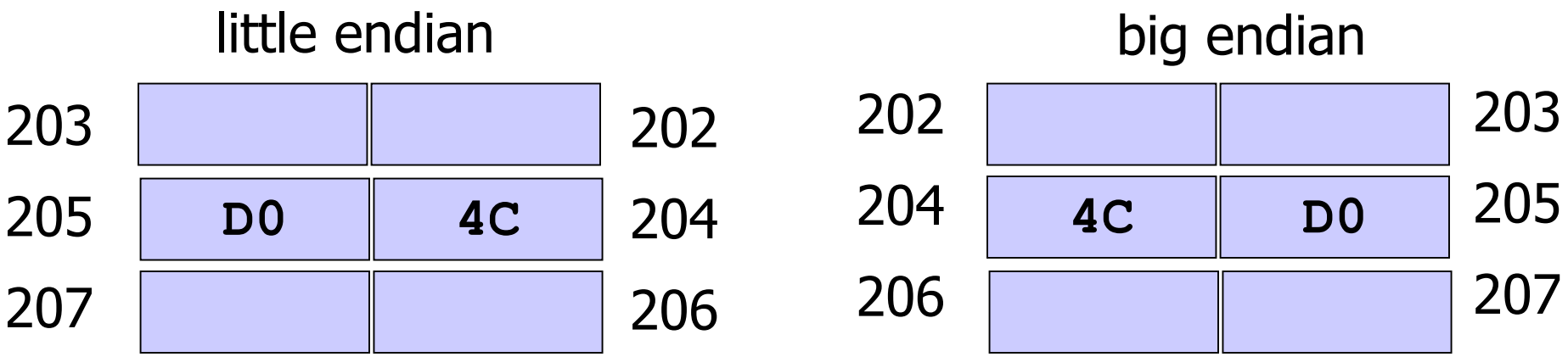




Big and little endian (2)

Quindi se comunico **D04C** esadecimale da little endian a big endian

- verrà inviato prima **4C** e poi **D0**
- la macchina ricevente interpreta il primo come byte più significativo e il secondo come byte meno significativo
- quindi il numero ricevuto sarò **4CD0**



Big and little endian (3)

E’ necessario avere un ordine dei byte indipendente dalla macchina e usare sempre quello per gli scambi

- è il *network byte order*
- la macchina mittente converte dalla sua rappresentazione (*host byte order*) al network byte order
- il ricevente fa l’inverso
- è necessario farlo solo per dati binari ‘grezzi’ non per dati già in formato standard (es. JPEG)

Conversione: network byte order

```
#include <arpa/inet.h>
/*(man 7 ip)*/
uint16_t htons (uint16_t hostnum);
uint32_t htonl (uint32_t hostnuml);
/* ritornano il numero convertito in net byte
  order (no error return) */

uint16_t ntohs (uint16_t netnum);
uint32_t ntohl (uint32_t netnuml);
/* ritornano il numero convertito in host byte
  order (no error return) */
```

Conversione: esempio

Vogliamo:

- stampare i due byte che rappresentano D04C in ordine di indirizzo crescente
- in *host byte order*
- ed in *network byte order*

Conversione: esempio (2)

```
#include <arpa/inet.h>
#include <stdio.h>

int main (void) {
   uint16_t n=0xD04C,nnet;
   unsigned char* p;
   p = (unsigned char *) &n;
   printf("%x %x \n",*p,*(p+1)); /* host ord*/
   nnet = htons(n);
   p = (unsigned char *) &nnet;
   printf("%x %x \n ",*p,*(p+1)); /* net ord*/
   return 0;
}
```

Conversione: esempio (3)

```
-- compilo su AMD athlon XP (/proc/cpuinfo)
bash:~$ gcc -Wall -pedantic endian.c -o endian
bash:~$ ./endian
4c d0
d0 4c
bash:~$
-- AMD, x86 sono little endian
-- network byte order è big endian (dai vecchi
   VAX)
```

Indirizzi AF_INET

- Sono indirizzi fomati da:
  - un identificativo della macchina (indirizzo IPv4)
    - numero a 32 bit assegnato ad un'interfaccia di rete sulla macchina
    - vengono scritti in *dotted notation* ovvero i 4 byte convertiti in decimale e separati da punti es. 216.109.125.70
    - ognuno ha anche un nome simbolico es. *www.yahoo.com*
  - un identificativo della porta all'interno della macchina
    - è un numero a 16 bit
    - identifica un servizio es. 80 (porta server HTTP), 21 (porta server ftp), etc...

Indirizzi AF_INET (2)

```
#include <netinet/in.h>
/* strutture che rappresentano un indirizzo */
struct sockaddr_in {
    sa_family_t sin_family; /* AF_INET */
    /* num di porta (uint16_t) net byte order */
    in_port_t sin_port;
    /* indirizzo IPv4 */
    struct in_addr sin_addr;
};

struct in_addr {
 in_addr_t s_addr; /* ind. (uint32_t) net bo */
};
```

Indirizzi AF_INET: esempio

```
/* vogliamo rappresentare 216.119.125.70 porta
   80*/


struct sockaddr_in sa;
sa.sin_family = AF_INET;
/* numero di porta (uint16_t) deve essere in
   network byte order */
sa.sin_port = htons(80);
/* indirizzo IPv4 (uint32_t) deve essere in
   network byte order 216.119.125.70 */
sa.sin_addr.s_addr = htonl(216<<24) + \
   htonl(119<<16) + htonl(125<<8) + htonl(70);
```

Indirizzi AF_INET: esempio (3)

```
/* vogliamo rappresentare 216.119.125.70 porta
   80*/
struct sockaddr_in sa;
sa.sin_family = AF_INET;
/* numero di porta (uint16_t) deve essere in
   network byte order */
sa.sin_port = htons(80);
/* indirizzo IPv4 (uint32_t) deve essere in
   network byte order 216.119.125.70 */
sa.sin_addr.s_addr = \
   inet_addr(”216.109.125.70");
```

Indirizzi AF_INET: conversione

- funzioni di conversione IPv4:
  - **`inet_addr`** (da dotted notation a numero a 32 bit formato rete) **`inet_ntoa`** (inverso)
  - **`inet_ntop`**, **`inet_pton`** sono la loro generalizzazione e funzionano anche con IPv6

Esempio: get HTTP page

```
/* vogliamo ottenere una pagina HTTP
   chiedendola direttamente al web server di
   131.114.3.24 porta 80*/
#define REQUEST ”GET / HTTP/1.0\r\n\r\n"
#define N 1024
int main (void) {
   struct sockaddr_in sa;
   int fd_skt, nread;
   char buf[N];
   sa.sin_family = AF_INET;
   sa.sin_port = htons(80);
 sa.sin_addr.s_addr=inet_addr(”131.114.3.24");
```

Esempio: get HTTP page (2)

```
...
  fd_skt =socket(AF_INET,SOCK_STREAM,0);
  connect(fd_skt,(struct sockaddr*)&sa, \
          sizeof(sa));
  write(fd_skt,REQUEST,strlen(REQUEST));
  nread = read(fd_skt,buf,sizeof(buf));
  write(1,buf,nread);
  close(fd_skt);
  return 0;
}
/* devono essere gestite tutte le situazioni di
  errore omesse qua per leggibilità */
```

Esempio: get HTTP page (3)

```
bash:~$ gcc -Wall -pedantic getpage.c -o getp
bash:~$ ./getp

????? Lo vedremo in laboratorio ????

bash:~$
```

AF_INET: nomi mnemonici

- Di solito non si usano i numeri ma dei nomi mnemonici:
  - **`www.di.unipi.it www.yahoo.com`** etc
  - è possibile convertire i nomi mnemonici nel numero corrispondente in dotted notation usando un database distribuito (DNS) accessibile da C con **`getaddrinfo`**
    - attenzione **`getaddrinfo`** ritorna errori suoi trasformabili in stringa con **`gai_strerror`**
  - è possibile ottenere il nome mnemonico ‘pubblico’ della macchina su cui stiamo girando con **`gethostname`**