# Introduzione ai socket

## Socket locali

Contiene lucidi tratti da: 2006-2007 Marco Faella, Clemente Galdi, Giovanni Schmid (Università di Napoli Federico II), 2004-2005 Walter Crescenzi(Università di Roma 3).

# Generalità

- Introdotti con la release 4.2 di BSD nel 1983, i socket rappresentano lo "standard de facto" per la programmazione di rete;

- Sono disponibili su tutte le piattaforme Unix e Mac OS e, sebbene con sintassi di programmazione lievemente differenti, su altri tipi di SO come Microsoft Windows (WinSock);

- Il loro successo è dovuto al fatto che costituiscono una interfaccia estremamente potente e flessibile:

    - consentono la comunicazione interprocesso sia localmente, ossia nell'ambito di uno stesso sistema, sia tra sistemi differenti interconnessi tramite rete;
    - sebbene il loro impiego più diffuso sia con la famiglia di protocolli di Internet (TCP/IP), possono essere utilizzati con molte altre famiglie di protocolli di rete (Appletalk, X.25,...)

# API per IPC

- Astrazione alla base di una API per IPC
- Permettono la comunicazione tra due processi:
  - in locale, cioè sulla stessa macchina
  - in remoto, ovvero attraverso un collegamento in rete
- Indipendenti dal Linguaggio e dal Protocollo
  - spesso denominati Berkeley (BSD) socket
  - rappresentano un estremo della comunicazione

# Socket e Descrittori

- Nascondono al programmatore tutti i dettagli della comunicazione
  - per es. basata su TCP/IP
- Sono referenziabili tramite descrittori, ovvero interi non negativi, esattamente come i file
- Diversi tipi ottenibili tramite la chiamata di sistema `socket()`

# Panoramica

- Definisce un canale di comunicazione bidirezionale
- Progettato per client-server
- Fornisce dei file descriptor
  - si possono usare read, write, close, etc.
- Vari tipi di socket
  - socket locali
  - socket TCP

# Domini e Stili di Comunicazione

- La logica alla base della programmazione con i socket è semplice: per realizzare i vari tipi di comunicazione viene utilizzato sempre lo stesso insieme di API generiche, precisandone la semantica (ossia il funzionamento) attraverso opportuni argomenti;

- I socket permettono di specificare il tipo di comunicazione attraverso le nozioni di dominio e di stile;

- Il dominio di un socket equivale alla scelta di una famiglia di protocolli. Le correnti release del kernel di molti Unix prevedono ventisei diverse famiglie;

- Ogni dominio è individuato univocamente da un intero non negativo, cui corrispondono una o più costanti simboliche del tipo PF_*nomefamiglia*, definite nell'header <sys/socket.h>;

# Domini e Stili di Comunicazione

- Tra i domini più importanti si ricordano:
  - PF_LOCAL (o PF_UNIX, o PF_FILE), per le comunicazioni in locale tramite filesystem (reale o virtuale);
  - PF_INET, la famiglia TCP/IP con Ipv4;
  - PF_INET6, la famiglia TCP/IP con Ipv6;
  - PF_IPX, famiglia di protocolli per reti Novell;
  - PF_APPLETALK, famiglia di protocolli per reti Appletalk;

- A ciascun dominio possono corrispondere in teoria uno o più schemi di indirizzamento, ossia tipi di indirizzi, per cui i socket prevedono la nozione di famiglia di indirizzi, ciascuna individuata univocamente attraverso un numero non negativo, cui corrisponde (sempre tramite l'header socket.h) una costante simbolica del tipo AF_*nomefamiglia*;

- I domini finora introdotti dispongono di un unico schema di indirizzi, per cui le attuali implementazioni prevedono l'equivalenza tra nomi di dominio e nomi di indizzi.

# Stili di Comunicazione

- Lo stile di comunicazione definisce il tipo di canale logico instaurato per la comunicazione, ossia le caratteristiche della trasmissione;

- Le trasmissioni possono ad es. avvenire a flusso o a pacchetti, essere affidabili o non affidabili, richiedere o meno la negoziazione di una connessione, etc.;

- Lo stile di comunicazione è individuato da costanti simboliche del tipo SOCK_*nomestile*, definite anch'esse nell'header <sys/socket.h>;

- Gli stili principali sono:
  - SOCK_STREAM, corrispondente ad un canale di trasmissione bidirezionale a flusso, con connessione, sequenziale ed affidabile;

# Stili di Comunicazione

- SOCK_DGRAM, che consiste in una trasmissione a pacchetti (datagram) di lunghezza max. prefissata, senza connessione e non affidabile;
- SOCK_RAW, per l'accesso a basso livello ai protocolli di rete ed alle varie interfacce;

- Assegnato un dominio, la scelta dello stile di comunicazione corrisponde in pratica ad individuare uno specifico protocollo tra quelli appartenenti al dominio;

- Non tutte le combinazioni "dominio-stile di comunicazione" sono valide, in quanto non è detto che in una famiglia di protocolli esista un elemento per ciascuno dei possibili stili;

# Indirizzamento

• Una ulteriore caratteristica di una trasmissione è l'indirizzamento: per individuare le parti in comunicazione è necessario specificarne gli indirizzi;

• Ogni famiglia di protocolli ha una sua forma di indirizzamento ed in corrispondenza a questa una particolare struttura di indirizzi;

• Gli indirizzi vengono specificati alle socket API tramite puntatori ad opportune strutture dati, i cui nomi incominciano con sockaddr_ ed il cui suffisso richiama il nome del relativo dominio;

• Le funzioni devono poter gestire molteplici strutture e il problema di come passare i puntatori è stato risolto all'epoca della definizione dell'interfaccia dei socket con l'introduzione di una struttura di indirizzi generica;

# Indirizzamento

• La struttura generica degli indirizzi è definita nell'header <sys/socket.h> come segue:

```
struct sockaddr {
sa_family_t sa_family; /* address family: AF_xxx */
char sa_data[14]; /* address (protocol-specific) */
};
```

• I prototipi delle funzioni che gestiscono indirizzi fanno uso di puntatori al tipo struct sockaddr, ed è pertanto necessario effettuare una conversione al tipo di struttura di indirizzi effettivamente utilizzata nella chiamata;

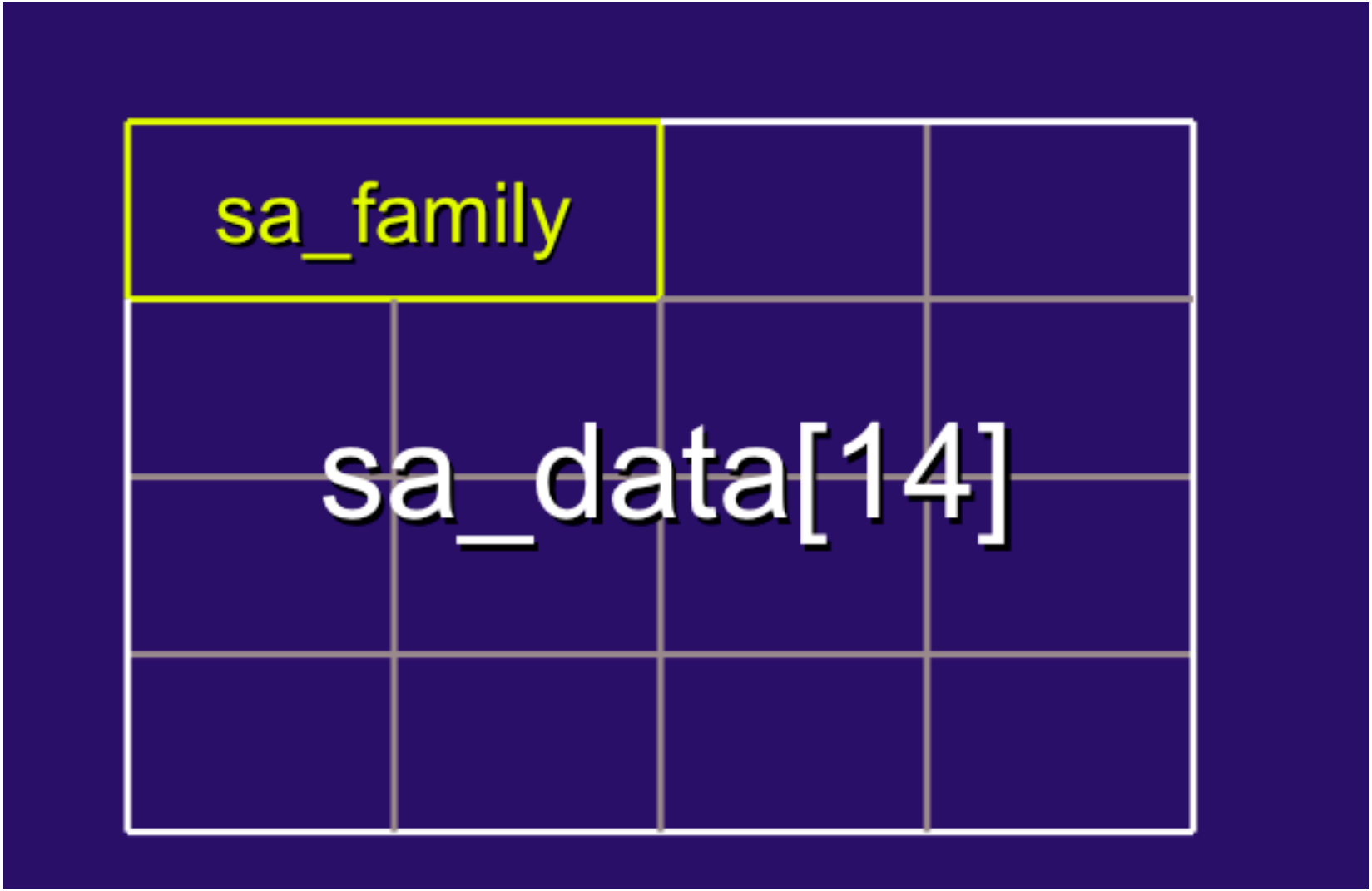
• Per il programmatore la struttura sockaddr non ha altra rilevanza che quella di imporre la conversione di tipo, ma il kernel la utilizza per recuperare il campo sa_family e determinare il tipo di indirizzo;

# Indirizzamento

- In UNIX le strutture dati per la gestione degli indirizzi sono progettate per essere adatte a diversi domini di comunicazione e protocolli. Ad es.
  - `sa_family = AF_INET`
  - `sa_family = PF_LOCAL,PF_UNIX`
- Generico indirizzo:

```
#include <sys/socket.h>
struct sockaddr {
    unsigned short sa_family;    // address family, AF_xxx
    char           sa_data[14]; // 14 bytes of protocol
};                                // address
```

# Indirizzamento



# Indirizzi

- Siamo interessati a `sa_family = AF_INET`, cioè al dominio “internet”
- In tal caso servono 2 byte per un numero di porta e 4 byte per un indirizzo IP

```
#include <netinet/in.h>
struct sockaddr_in {
    short int sin_family;          // Address family
    unsigned short int sin_port;   // Port number
    struct in_addr sin_addr;       // IP Internet Address
    unsigned char sin_zero[8];     // Stessi byte di…
};                                 // …struct sockaddr
```

14 bytes

# Comunicazione Client-Server

- crea un socket tramite la funzione omonima ed assegna ad esso un nome (indirizzo) affinchè sia condivisibile da altri processi (funzione bind);

- attende per le connessioni dei client sul socket predisposto tramite la funzione listen, che associa al socket una coda di lunghezza opportuna per la gestione di connessioni simultanee da parte dei client;

- accetta le connessioni da parte dei client tramite la funzione accept, il cui effetto è quello di creare un nuovo socket per ogni nuova connessione con un client ***C***. Tale socket è quello che realizza effettivamente il canale di comunicazione con ***C***;

- se il server è iterativo, un client è posto in attesa sulla coda generata da listen fintanto che il server ha terminato di servire il client precedente;

# Comunicazione Client-Server

- se il server è concorrente, per ogni client viene generato un processo ad hoc per offrire il servizio, in modo che più client possano essere serviti in modalità concorrente. In tal caso l'attesa sulla coda è limitata al tempo necessario alla gestione della richiesta del client da parte del server ed allo start-up del nuovo processo;

Dal lato client la procedura è molto più semplice:

- il client crea un socket tramite la funzione socket, ma a tale chiamata *non* fa seguire una bind, visto che il socket non deve essere indirizzabile;

- per stabilire una connessione con il server, richiama la funzione connect utilizzando il nome del socket predisposto dal server come indirizzo.

# Panoramica: il server

- Crea il socket — socket
- Gli assegna un indirizzo — bind
- Si mette in ascolto — listen
- Accetta nuove connessioni — accept
- ...
- Chiude il socket — close
- Se il socket è locale: cancella il file corrispondente — unlink

# Funzione Socket

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

int socket(int family, int type, int protocol);
```

- apre un socket, allocando una voce nella tabella dei file del kernel e permettendo la specifica del dominio, dello stile e del protocollo di comunicazione;
- *family* è la famiglia cui appartiene il protocollo (dominio);
- *type* definisce il tipo di comunicazione,
- *protocol* specifica il protocollo della famiglia;
- restituisce -1 in caso di insuccesso, un numero positivo (il socket descriptor) altrimenti.

# Creare un socket

int **socket**(int **famiglia**, int **tipo**, int **protocollo**);

- Crea un socket di una determinata categoria
- Nel nostro caso:
  - famiglia = PF_LOCAL oppure PF_INET
    - (PF = *Protocol Family*)
  - tipo = SOCK_STREAM
  - protocollo = 0
- Restituisce il descrittore del socket, oppure -1

# Funzione Socket

| **Valori per *family*** | **Tipo di protocolli** |
|---|---|
| (AF_) PF_UNIX | Protocolli per comunicazioni locali su sistemi UNIX |
| (AF_) PF_INET | Protocolli per Internet (vers. 4) |
| (AF_) PF_INET6 | Protocolli per Internet (vers. 6) |
| (AF_) PF_APPLETALK | Protocolli per LAN Appletalk |
| (AF_) PF_X25 | Protocolli dello standard ITU X.25 per reti a commutazione di pacchetti |
| ... | |

| **Valori per *type*** | **Tipo di trasmissione** |
|---|---|
| SOCK_STREAM | a flusso, sequenziale ed affidabile |
| SOCK_DGRAM | a pacchetti di lungh. max. fissata, non sequenziale e non affidabile |
| SOCK_RAW | accesso a basso livello (costruzione “manuale” dei pacchetti) |
| SOCK_SEQPACKET | a pacchetti di lungh. max. fissata, sequenziale ed affidabile |
| ... | |

| **Valori per *protocol*** | **Protocollo della famiglia INET** (definiti in <netinet/in.h>) |
|---|---|
| IPPROTO_TCP | TCP |
| IPPROTO_UDP | UDP |
| IPPROTO_RAW | IP |
| IPPROTO_ICMP | ICMP |
| ... | |

# Creare un socket locale

```
int fd = socket(PF_LOCAL, SOCK_STREAM, 0);

if (fd<0) perror("socket"), exit(1);
```

```
int fd = socket(PF_INET, SOCK_STREAM, 0);

if (fd<0) perror("socket"), exit(1);
```

```
int fd = socket(PF_INET, SOCK_DGRAM, 0);

if (fd<0) perror("socket"), exit(1);
```

# La funzione bind

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

int bind(int sd, const struct sockaddr *my_addr, int addrlen);
```

- assegna un indirizzo locale al (l'insieme di) socket individuato dal socket descriptor *sd*;
- *sd* è un socket descriptor ottenuto da una precedente chiamata a socket;
- *my_addr* è l'indirizzo locale, specificato secondo il formato caratteristico della famiglia di protocolli cui *sd* è riferito;
- *addrlen* è la lunghezza del suddetto indirizzo;
- restituisce -1 in caso di errore, 0 altrimenti.

# Assegnare un indirizzo a un Socket

```
int bind(int sockfd,
        const struct sockaddr *my_addr,
        socklen_t addrlen);
```

- Assegna l'indirizzo my_addr al socket sockfd
- Il tipo effettivo del secondo argomento dipende dalla famiglia del socket
  - socket locali: un indirizzo è sostanzialmente il nome di un file; bind fallisce se il file esiste già
- Il terzo argomento deve essere pari a sizeof del secondo argomento
- Restituisce: 0 se OK, -1 altrimenti

# Indirizzi locali

in sys/un.h:

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
      sa_family_t  sun_family;
      char         sun_path[UNIX_PATH_MAX];
};
```

*vedere: man 7 unix*

come si usa:

```
struct sockaddr_un mio_indirizzo;

mio_indirizzo.sun_family = AF_LOCAL;
strcpy(mio_indirizzo.sun_path, "/tmp/mio_socket");

bind(fd, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo));
```

# Mettersi in ascolto

int **listen**(int **sockfd**, int **lunghezza_coda**);

- Mette il socket in modalità di ascolto
  - cioè in attesa di nuove connessioni
- Il secondo argomento specifica quante connessioni possono essere in attesa di essere accettate
- Restituisce: 0 se Ok, -1 altrimenti

# Funzione listen

```
#include <sys/socket.h>
int listen (int sd,  int backlog);
```

•utilizzata da un server in caso di comunicazione orientata alla connessione;
•pone il socket specificato da ***sd*** in modalità passiva, ossia in ascolto di eventuali connessioni, predisponendo per esso una coda per le connessioni in arrivo di lunghezza pari a ***backlog;***
• ***backlog*** rappresenta il numero massimo di connessioni pendenti accettate. Se tale numero è superato, il client riceverà un errore, oppure – nel caso di protocolli come TCP che prevedono la ritrasmissione – la richiesta del client verrà ignorata in modo da poter essere ritentata;
•restituisce 0 in caso di successo, -1 altrimenti.

# Accettare una nuova connessione

int **accept**(int **sockfd**,
struct sockaddr ***indirizzo_client**,
socklen_t ***dimensione_indirizzo**);

- Il secondo e terzo argomento servono ad identificare il client
  - possono essere NULL
- Restituisce un nuovo descrittore! (oppure -1)
  - crea un nuovo socket, dedicato a questa nuova connessione
  - il vecchio socket resta in ascolto

# Accettare nuove connessioni

```
#include <sys/socket.h>
int accept (int sd, const struct sockaddr *addr, int addrlen);
```

- utilizzata da un server in caso di comunicazione orientata alla connessione, restituisce un nuovo socket descriptor su cui si potrà operare per effettuare la comunicazione con un client;
- estrae la prima connessione relativa al socket descriptor ***sd*** in attesa sulla coda delle connessioni, definita grazie ad una precedente chiamata a listen per ***sd*** ;
- nella struttura ***addr*** e nella variabile ***addrlen*** vengono restituiti, rispettivamente, l'indirizzo e la lunghezza di tale indirizzo per il client che si è connesso;
- restituisce un nuovo socket descriptor in caso di successo, -1 altrimenti. Il nuovo socket eredita le caratteristiche di ***sd***;

# Struttura di un server

```
int fd1, fd2;
struct sockaddr_un mio_indirizzo;

mio_indirizzo.sun_family = AF_LOCAL;
strcpy(mio_indirizzo.sun_path, "/tmp/mio_socket");

fd1 = socket(PF_LOCAL, SOCK_STREAM, 0);
bind(fd1, (struct sockaddr *) &mio_indirizzo, sizeof(mio_indirizzo));

listen(fd1, 5);
fd2 = accept(fd1, NULL, NULL);
...
close(fd2);
close(fd1);
unlink("/tmp/mio_socket");
```

# Ricapitolazione: il server

- Creare un socket
  - **fd socket**(famiglia, tipo, protocollo)
- Assegnare un indirizzo al socket
  - **ok bind**(fd, indirizzo, dimensione_indirizzo)
- Mettersi in ascolto sul socket
  - **ok listen**(fd, lunghezza_coda)
- Accettare una nuova connessione
  - **fd accept**(fd, indirizzo_client, dimensione_ind)
- Chiudere e cancellare
  - **close** (ed **unlink** per i socket locali)

# Panoramica: il client

- Crea il socket — socket
- Si connette ad un server — connect
- ...
- Chiude il socket — close

# Connettersi ad un server

```
int connect(int sockfd,
        const struct sockaddr *serv_addr,
        socklen_t addrlen);
```

- Connette il socket sockfd all'indirizzo serv_addr
- Il client deve conoscere l'indirizzo del server
- Il terzo argomento deve essere pari a sizeof del secondo argomento
- Restituisce 0 oppure -1

# Funzione connect

```
#include <sys/socket.h>
int connect (int sd, const struct sockaddr *serv_addr, int addrlen);
```

- utilizzata dal lato client, ha due funzionalità differenti a seconda che la comunicazione sia orientata o meno connessione;
- In entrambi i casi collega il socket locale di identificativo *sd* al socket remoto di indirizzo *serv_addr*;
- nel caso di comunicazioni con connessione attiva la procedura di avvio della connessione (il *three-way handshake* per il TCP) e ritorna solo quando la connessione è stabilita o si è verificato un errore;
- restituisce 0 in caso di successo, -1 altrimenti.

# Funzione close

```
#include <unistd.h>
int close(int sock_fd);
```

- restituisce zero in caso di successo, -1 in caso di errore
- serve per dichiarare che non si vuole più utilizzare il socket
- più processi dello stesso host possono condividere la stessa socket
  - solo se tutti avranno eseguito una `close()` il sistema operativo provvederà a chiudere la connessione (necessariamente di tipo SOCK_STREAM) concludendo il protocollo TCP
- la chiusura è simmetrica: la connessione sarà effettivamente chiusa quando sarà stata chiusa sia sul server che sul client

# Struttura di un client

```
int fd;
struct sockaddr_un indirizzo;

indirizzo.sun_family = AF_LOCAL;
strcpy(indirizzo.sun_path, “/tmp/mio_socket”);

fd = socket(PF_LOCAL, SOCK_STREAM, 0);
connect(fd, (struct sockaddr *) &indirizzo, sizeof(indirizzo));
...
close(fd);
```

# Ricapitolazione: il client

- Creare un socket
  - **fd socket**(famiglia, tipo, protocollo)
- Connettersi ad un dato indirizzo
  - **ok connect**(fd, indirizzo, dimensione_indirizzo)
- Chiudere la connessione
  - **ok close(fd)**

# Leggere da un socket

- Si può usare read
- Se non ci sono dati da leggere, read **blocca** il processo in attesa di dati (come per una pipe)
- E' normale ottenere meno bytes di quelli richiesti (come per una pipe)
- Ottenere 0 bytes significa che il socket è vuoto ed inoltre è stato chiuso

# Scrivere su un Socket

- Si può usare write
- E' normale riuscire a scrivere meno bytes di quelli richiesti (bisogna riprovare con il resto!)
- Se il socket è stato chiuso, il processo riceve il segnale SIGPIPE
  - di default, questo segnale termina il processo
  - se si ignora questo segnale (signal(SIGPIPE, SIG_IGN)), write restituisce -1 e imposta errno=EPIPE
  - oppure, si può catturare il segnale

# Esercizio 1

- Implementare un server che fornisce ai client l'ora esatta, usando un socket locale
  - sugg.: una stringa che rappresenta l'ora esatta si può ottenere così:

```
#include <time.h>
...
char buffer[26];
time_t ora;
time(&ora);
printf(" Ora esatta : %s\n",
                    ctime_r(&ora, buffer));
```

  - la funzione time restituisce l'ora in un formato interno (time_t)
  - la funzione ctime_r trasforma il formato interno in stringa; ha bisogno di un buffer di (almeno) 26 caratteri

# Esercizio 1

- Ad ogni nuova connessione, il server scrive sul socket l'ora corrente, poi chiude la connessione e si rimette in attesa di nuove connessioni
- Implementare anche un client che riceve l'ora dal server e la stampa sul terminale
- Provare a lanciare diversi client in rapida successione

# Esercizio: Server

```
#define SOCKET_NAME "/tmp/my_first_socket"

static int gestisci(int);

int main(int argc, char **argv)
{
  int listen_sd, connect_sd; // Socket descriptor

  struct sockaddr_un my_addr, client_addr;
  socklen_t client_len;

  my_addr.sun_family = AF_LOCAL;
  strcpy(my_addr.sun_path, SOCKET_NAME);

  // Server section: create a local socket
  if ( (listen_sd = socket(PF_LOCAL, SOCK_STREAM, 0)) < 0)
    perror("socket"), exit(1);
```

# Esercizio

```
// remove socket file if present
  unlink(SOCKET_NAME);

  // bind socket to pathname
  if ( bind(listen_sd, (struct sockaddr *) &my_addr, sizeof(my_addr)) < 0)
    perror("bind"), exit(1);

  // put socket in listen state
  if ( listen(listen_sd, 1) < 0)
    perror("listen"), exit(1);

  while (1) {
    client_len = sizeof(client_addr);
     fprintf(stderr, " sizeof(client_addr)=%d \n", client_len);

    if ( (connect_sd = accept(listen_sd, (struct sockaddr *) &client_addr, &client_len)) < 0)
      perror("accept"), exit(1);

    fprintf(stderr, " new connection \n");
    fprintf(stderr, " client address: %100s\n ", client_addr.sun_path);

    // handle the connection
    gestisci(connect_sd);
    close(connect_sd);
  }

  return 0;
```

# Esercizio

```
int gestisci(int sd)
{
  char buf[100];
  int n;

  time_t ora;
  time(&ora);
  printf(" Ora: %s\n", ctime_r(&ora, buf));
  printf(" Ora: %d\n", strlen(buf));
  write(sd, buf, 26);
  return 0;
}
```

# Esercizio: client

```
#define SOCKET_NAME "/tmp/my_first_socket"

static int client(void);

int main(int argc, char **argv)
{
  return client();
}

// Client section: Sends integers from 0 to 4, at 1 second intervals, and then terminates
int client(void)
{
  char msg[100];
  struct sockaddr_un srv_addr;
  int sd, i;
  ssize_t temp; /* signed size_t */

  signal(SIGPIPE, SIG_IGN);

  // crea il socket
  sd = socket(PF_LOCAL, SOCK_STREAM, 0);

  if (sd < 0) perror("socket"), exit(1);

  srv_addr.sun_family = AF_LOCAL; // unsigned short int
  strcpy(srv_addr.sun_path, SOCKET_NAME);
```

```
 // si connette all'indirizzo predefinito
  if ( connect(sd,(struct sockaddr *) &srv_addr, sizeof(srv_addr) ) < 0)
    perror("connect"), exit(1);

  sleep(1);
  read(sd, msg, 26);
  printf(" client riceve: *%s*\n", msg);
  printf(" client riceve: %d\n", strlen(msg));
  return 0;

}
```

# Esercizio