

Introduzione ai socket

Socket locali

Panoramica

- 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

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`

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

Creare un socket locale

```
int fd = socket(PF_LOCAL, SOCK_STREAM, 0);  
if (fd<0) perror("socket"), exit(1);
```

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

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

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

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
 - E' un problema se il server chiude il socket prima che il client abbia letto l'ora?
- Implementare anche un client che riceve l'ora dal server e la stampa sul terminale
- Provare a lanciare diversi client in rapida successione