Programmazione di Sistema – 6

Lucidi per il corso di Laboratorio di Sistemi Operativi tenuto da Paolo Baldan presso l'Università Ca' Foscari di Venezia, anno accademico 2004/2005. Parte di questo materiale è rielaborato dai lucidi del Corso di Laboratorio di Sistemi Operativi tenuto da Rosario Pugliese presso l'Università di Firenze, anno accademico 2001/02.

IPC: Inter Process Communication

Meccanismi che permettono a processi distinti di comunicare e scambiare informazioni.

I processi che comunicano possono risiedere

- *sulla stessa macchina (segnali, pipe, fifo, socket)*
- *su macchine diverse (socket).*

La comunicazione può essere finalizzata a

- *cooperazione: i processi scambiano dati per l'ottenimento di un fine comune*
- *sincronizzazione: lo scambio di informazioni permette a processi indipendenti, ma correlati, di schedulare correttamente la propria attività (es. di non accedere contemporaneamente ad una risorsa condivisa).*

Socket

Socket

Potente meccanismo di comunicazione tra processi in ambiente Unix.

- *Comunicazione bidirezionale tra processi.*
- *I processi comunicanti possono risiedere*
  - *nello stesso host*
  - *in host diversi.*

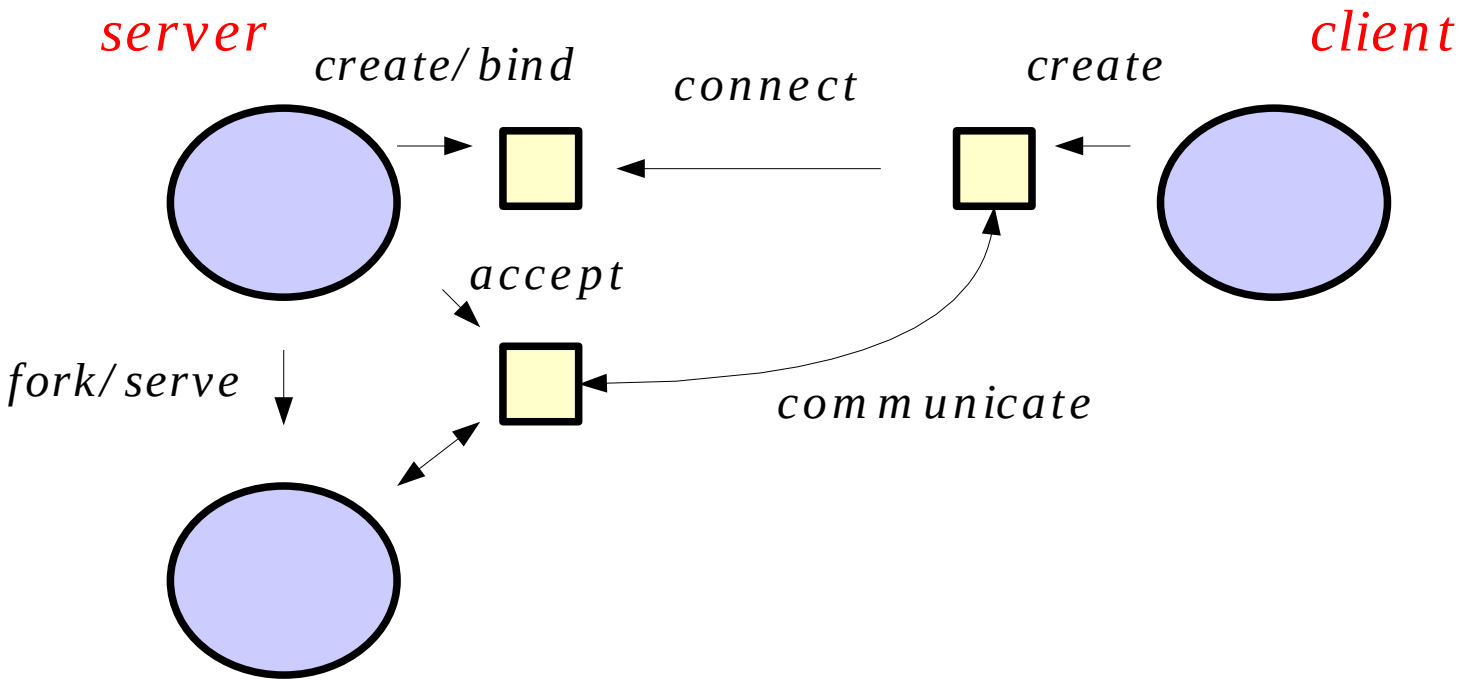
Utilizza l'interfaccia di accesso ai file.

Es.

- *collegamento remoto di un utente (rlogin)*
- *trasferimento di file da una macchina all'altra.*

Socket

Astrattamente, un socket rappresenta un endpoint di comunicazione al quale è possibile associare un nome.



Socket

La comunicazione tramite i socket si basa sul modello client-server.

- *Un processo, il server, crea un socket il cui nome è noto anche ai processi client.*
- *I client possono comunicare col server tramite una connessione al suo socket.*
  - *Il client crea un socket anonimo e quindi chiede che sia connesso al socket del server.*
  - *Una connessione che ha successo restituisce un descrittore di file al client ed uno al server, che permettono di leggere e scrivere sul canale.*
  - *Una volta creata la connessione col client, solitamente il server crea un processo figlio che si occupa della gestione della connessione, mentre il processo originario continua ad accettare altre connessioni da client.*

Creazione di un socket: socket()

int socket (int domain, int type, int protocol)

domain *specifica il dominio in cui si vuole utilizzare il socket. Useremo solo i valori*

- **AF_UNIX**
- **AF_INET**.

type *specifica il tipo di comunicazione. Useremo solo il valore* **SOCK_STREAM**.

protocol *specifica il protocollo da utilizzare per la trasmissione dei dati. Se vale 0 indica il protocollo di default per la coppia domain/type in questione.*

Restituisce un descrittore di file associato al socket creato, se ha successo; -1, altrimenti.

Socket: Attributi principali

Dominio: indica dove risiedono server e client. Alcuni domini sono

- **AF_UNIX**: *client e server sono sulla stessa macchina*
- **AF_INET**: *client e server sono ovunque in Internet.*

Tipo: tipo di comunicazione tra server e client. I tipi principali sono

- **SOCK_STREAM**: *affidabile, preserva l'ordine di invio dei dati, basata su stream di byte di lunghezza variabile.*
  - *Un socket viene creato ed esiste durante una sessione in cui coppie di processi comunicano tra loro. Alla fine della sessione il socket viene chiuso e, di fatto, rimosso dal sistema.*
- **SOCK_DGRAM**: *inaffidabile, senza una connessione fissa, basata su messaggi di lunghezza variabile.*
  - *Il socket viene creato per una singola comunicazione tra processi e non esiste nè prima nè dopo. Una comunicazione ulteriore tra gli stessi processi richiede un nuovo socket.*

Socket: Attributi principali

Protocollo: specifica i mezzi di basso livello con cui il tipo di socket è implementato.

- *Ogni socket può adottare protocolli diversi per la comunicazione*
  - *protocolli stream, come TCP/IP, che assicurano la ricezione dei dati spediti preservando anche l'ordine di spedizione.*
  - *protocolli datagram, come UDP, che non assicurano l'affidabilità della comunicazione*
- *Tipicamente, le system call che richiedono un parametro per la specifica del protocollo accettano 0 per indicare "il protocollo più opportuno".*

Noi vedremo solo i socket **SOCK_STREAM**

Socket SOCK_STREAM

Il server

- *crea un socket anonimo con* **`socket()`**
- *gli assegna un indirizzo con* **`bind()`**
- *dichiara il numero di richieste di connessioni che possono essere accodate su quel socket con* **`listen()`**
- *accetta una connessione con* **`accept()`**
- *opera sul socket con* **`read()`** *e* **`write()`**

Il client

- *crea un socket anonimo con* **`socket()`**
- *lo collega al socket del server con* **`connect()`**
- *opera sul socket con* **`read()`** *e* **`write()`**.

Sia il client che il server, una volta terminate le operazioni col socket, lo chiudono con **`close()`**.

File di intestazione

Un programma che usa socket deve includere

sys/types.h

sys/socket.h

Inoltre, deve includere

sys/un.h, *se usa socket del dominio* **AF_UNIX**

netinet/in.h, **arpa/inet.h** *e* **netdb.h**, *se usa socket del dominio* **AF_INET**.

Assegnazione di un indirizzo: bind()

int bind (int fd, const struct sockaddr *addr, int addrlen)

Associa al socket anonimo riferito dal descrittore **fd** *l'indirizzo (locale o di rete, a seconda del dominio) memorizzato nella struttura puntata da* **addr**.

Tipo e valore dell'indirizzo dipendono dal dominio del socket. Possono essere

- *nome di un file nel file system locale (per* **AF_UNIX**)
- *indirizzo IP e numero di porta (per* **AF_INET**).

addrlen *è la lunghezza della struttura dati che memorizza l'indirizzo.*

Restituisce 0, se ha successo; -1, altrimenti.

Assegnazione di un indirizzo: bind()

*Il tipo (**struct sockaddr ***) è*

```
struct sockaddr {
 unsigned short sa_family; /* address family,
                              AF_xxxx */
 char sa_data[14];         /* 14 byte protocol
                              address */
}
```

*Le strutture dati indirizzo relative ai diversi domini devono essere convertite in (**struct sockaddr ***) all'invocazione di **bind()**.*

Assegnazione di un indirizzo: AF_UNIX

In **AF_UNIX** *gli indirizzi sono nomi di file.*

Qualsiasi file può essere utilizzato, purché si abbiano i diritti di scrittura sulla directory che lo contiene.

Per connettersi ad un socket è sufficiente avere diritti di lettura sul file relativo.

La struttura dati che contiene l'indirizzo (definita in **sys/un.h***) è*

```
struct sockaddr_un {
   unsigned short sun_family;    /* AF_UNIX */
   char sun_path[108];           /* pathname */
}
```

I campi dovrebbero assumere i seguenti valori

sun_family*:* **AF_UNIX**

sun_path*: il pathname completo o relativo del socket.*

Assegnazione di un indirizzo: AF_UNIX

Se si tenta un **`bind()`** *con un nome di file esistente si verifica un errore.*

Quindi è opportuno

- *invocare prima un unlink() di quel nome oppure*
- *generare un nome di file nuovo (*`tempnam()`*, ...)*

La lunghezza dell'indirizzo, ovvero il terzo parametro di **`bind()`**, *è calcolata con* **`sizeof()`**.

Esempio: Creazione e bind in AF_UNIX

```
#include <stddef.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>  /* For AF_UNIX sockets */

int make_named_socket (const char *filename) {
   struct sockaddr_un name;
   int sock;

   sock = socket (AF_UNIX, SOCK_STREAM, 0);
   if (sock < 0) {
      perror ("socket");
      exit (EXIT_FAILURE);
   }

   name.sun_family = AF_UNIX;
   strcpy (name.sun_path, filename);

   if (bind (sock, (struct sockaddr *) &name, sizeof(name)) < 0) {
      perror ("bind");
      exit (EXIT_FAILURE);
   }
   return sock;
}
```

Connessione ad un indirizzo: connect()

```
int connect (int fd, const struct sockaddr *addr,
             int addrlen)
```

Tenta di connettere un socket anonimo creato da un client, riferito dal descrittore **`fd`**, *ad un socket il cui indirizzo è nella struttura puntata da* **`addr`**.

Dominio, tipo e protocollo del socket del client devono concordare con quelli del socket del server.

`addrlen` *contiene la lunghezza della struttura dati che memorizza l'indirizzo.*

Connessione ad un indirizzo: connect()

Se **`connect()`** *ha successo,* **`fd`** *può essere usato per comunicare con il socket del server.*

La struttura a cui punta **`addr`** *deve seguire le stesse regole viste nel caso di* **`bind()`**.

`connect()` *restituisce valore*

- *0, se ha successo (la connessione è instaurata);*
- *-1 altrimenti (es. il socket del server non esiste, la sua coda è piena, ecc.)*

Quando ha successo, **`connect()`** *si sincronizza con* **`accept()`**.

Numero di connessioni: listen()

`int listen (int fd, int queuelength)`

Specifica il numero massimo **`queuelength`** *di richieste di connessioni pendenti che possono essere accodate sul socket corrispondente al descrittore* **`fd`**.

Se un client tenta una connessione ad un socket la cui coda è piena

- *la connessione gli viene negata (*`connect()` *fallisce)*
- *gli viene notificato l'errore* **`ECONNREFUSED`**.

Accettazione di connessione: accept()

```
int accept (int fd, struct sockaddr *addr,
            int *adLen)
```

Estrae la prima richiesta di connessione dalla coda delle connessioni pendenti del socket riferito da **`fd.`**

Crea un nuovo socket con gli stessi attributi di quello originario.

Lo connette al socket del client e restituisce un descrittore di file corrispondente, che permette di comunicare con il client.

Quindi il socket originario può essere usato per accettare altre connessioni.

Accettazione di connessione: accept()

Normalmente **`accept()`** *blocca l'invocante fino alla connessione.*

La struttura puntata da **`addr`** *è riempita con l'indirizzo del client (usata con connessioni Internet).*

`adLen` *punta ad un intero, il quale, dopo che la connessione è stabilita, darà la lunghezza effettiva, in byte, della struttura puntata da* **`addr`**.

`accept()` *restituisce un descrittore di file se ha successo; -1 altrimenti.*

Operazioni su socket

Una volta stabilita una connessione, il socket del client e quello del server possono essere utilizzati in tutto e per tutto come descrittori di file sui quali si opera mediante **`read()`** *(bloccante) e* **`write()`**.

Vedi anche **`send()`** *e* **`recv()`**.

Una volta concluse le operazioni su di un socket, il relativo descrittore può essere chiuso tramite una **`close()`**.

Esempio di socket AF_UNIX: chef e cook

Consiste di due programmi

`chef.c`, *il server*, **`cook.c`**, *il client*

Il server crea un socket **`recipe`** *e attende la richiesta di una ricetta da parte di un client*

Il client si connette al socket **`recipe`** *e invia la richiesta (nella forma di un numero)*

Il server fornisce la ricetta richiesta (letta da un file) sul socket **`recipe`**

Il client legge la ricetta fornita dal server (sequenza di stringhe di lunghezza variabile terminate dal carattere '\0') e man mano che la legge la mostra sullo standard output, quindi termina.