Esercitazione [06]

Client/Server con Socket

Riccardo Lazzeretti - lazzeretti@diag.uniroma1.it
Serena Ferracci - ferracci@diag.uniroma1.it

Sistemi di Calcolo 2
Programmazione dei Sistemi di Calcolo Multi-Nodo

Corso di Laurea in Ingegneria Informatica e Automatica

A.A. 2019-2020

Sommario

- Soluzione esercizio su CopiaFile e TimeServer
- Obiettivi dell’esercitazione
- Server socket
- Client socket
- Esercizio: EchoServer

Esercizio copiare un file in C

- Sorgente da completare: `copy.c`
- Argomenti
  - File sorgente S
  - File destinazione D
  - Dimensione B del batch di lettura/scrittura (opzionale, default 128 byte)
- Semantica

  Effettuare una copia di S in D tramite una sequenza di letture da S e scritture in D a blocchi di B byte per volta
- Esercizio: completare il codice dove indicato
  - Per testare la propria soluzione è disponibile lo script `test.sh`

```c
while (1) {
int read_bytes = 0; // index for
        //writing into the buffer
int bytes_left = block_size;
        //number of bytes to
        //(possibly) read

while (bytes_left > 0) {
  int ret = read(src_fd, buf +
    read_bytes, bytes_left);

  // no more bytes left to read!
  if (ret == 0) break;

  if (ret == -1){
    if(errno == EINTR) // read()
        //interrupted by a signal
    continue;
    // handle generic errors
    handle_error("Cannot read from
        source file");
  }

  bytes_left -= ret;
  read_bytes += ret;
}
```

```c
// no more bytes left to write!
if (read_bytes == 0) break;

int written_bytes = 0; // index for
        //reading from the buffer
bytes_left = read_bytes; // number
        //of bytes to write

while (bytes_left > 0) {
  int ret = write(dest_fd, buf +
        written_bytes, bytes_left);


  if (ret == -1){
    if(errno == EINTR) // write()
        //interrupted by a signal
    continue;
    // handle generic errors
    handle_error("Cannot write to
        destination file");
  }

  bytes_left -= ret;
  written_bytes += ret;
}
}
```

Esercizio su TimeServer

- Scenario
  - Il server in ascolto su una porta nota
  - Il client si connette ed invia il messaggio «TIME»
  - Se il server riceve il messaggio atteso, risponde con ora e data correnti, altrimenti con un messaggio d'errore
- Sorgenti: `server.c` e `client.c`
- Esercizio: completare le parti relative all'invio/ricezione su socket
  - Per il momento, senza gestire letture/scritture parziali
- Soluzione
  - Sia per il client che per il server, bisogna verificare che il valore di ritorno delle chiamate `send`/`recv` non sia negativo
  - Se è negativo ed `errno` corrisponde ad un interrupt, la chiamata va ripetuta

```
/** For the time being we don't deal with
 *  partially received replies!
 */

     // receive command from client
    while ( (recv_bytes = recv(socket_desc,
           recv_buf, recv_buf_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        handle_error("Cannot read
           from socket");
    }

    // send reply
    if (recv_bytes == allowed_command_len
           && !memcmp(recv_buf, allowed_command,
                      allowed_command_len)) {
        time_t curr_time;
        time(&curr_time);
        sprintf(send_buf, "%s", ctime(&curr_time));
    } else {
        sprintf(send_buf, "INVALID REQUEST");
    }


    while ( (ret = send(socket_desc, send_buf,
           server_message_len, 0)) < 0 ) {
        if (errno == EINTR) continue;
        handle_error("Cannot write to the
           socket");
    }

    // close socket
    ret = close(socket_desc);
    if (ret<0) handle_error("Cannot close
           socket for incoming connection");
```

```
/** For the time being we don't deal with
 *  partially received replies!
 */

     // send command to server
    while ( (ret = send(socket_desc, command,
           command_len, 0)) < 0) {
        if (errno == EINTR) continue;
        handle_error("Cannot write to socket");
    }

    // read message from the server
    while ( (recv_bytes = recv(socket_desc,
           recv_buf, recv_buf_len-1, 0)) < 0 ){
        if (errno == EINTR) continue;
        handle_error("Cannot read from
           socket");
    }
    // add string terminator manually!
    recv_buf[recv_bytes] = '\0';

    // close socket
    ret = close(socket_desc);
    if (ret<0) handle_error("Cannot close
           socket for incoming connection");
```

Obiettivi Esercitazione [07]

- Imparare ad impostare un'applicazione client/server che preveda:
  - Server single-thread
    - Come mettersi in ascolto su una porta nota?
    - Come accettare una connessione da client?
  - Client
    - Come connettersi ad un server in ascolto?
  - Semplice protocollo basato su messaggi testuali

Server Socket

- Come mettersi in ascolto su una porta nota?
  - Creazione socket - funzione `socket()`
  - Binding della socket su un indirizzo locale - funzione `bind()`
  - Infine, mettersi in ascolto - funzione `listen()`
- Come accettare una connessione da client?
  - Attesa di una connessione - funzione `accept()`
    - Una volta accettata una connessione, si ha a disposizione un descrittore di socket da usare per scambiare messaggi (tramite `send()`/`recv()`)
  - Una volta terminato lo scambio di messaggi, la connessione col client va chiusa - funzione `close()`

Strutture dati per le socket

- `struct in_addr`: rappresenta un indirizzo IP a 32 bit
- `struct sockaddr_in`: descrizione di una socket; al suo interno le informazioni principali sono:
  - Famiglia dell'indirizzo (`sin_family`)
    - Per i nostri scopi, `AF_INET`: protocollo IPv4
    - Ne esistono altre, es: `AF_UNIX`, `AF_BLUETOOTH`
  - Indirizzo IP (`sin_addr.s_addr`), per i nostri scopi:
    - Lato server, `INADDR_ANY`: in ascolto su tutte le interfacce
    - Lato client, specifica l'indirizzo IP del server
  - Numero porta (`sin_port`)
    - Bisogna rispettare l'ordine di trasmissione dei byte per la rete
    - `sin_port = htons(port)` per invertire l'ordine dei bytes

Funzione `socket()`

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

- Crea una socket, ossia un endpoint di comunicazione
- Argomenti
  - `family`: per i nostri scopi, `AF_INET`
    (vedi struttura dati `struct sockaddr_in`)
  - `type`: per i nostri scopi, `SOCK_STREAM` (protocollo TCP)
    - Ne esistono altre, es: `SOCK_DGRAM` (protocollo UDP)
  - `protocol`: per i nostri scopi, `0`
- Valore di ritorno
  - In caso di successo, il descrittore della socket
  - In caso di errore, `-1`, `errno` è settato

Funzione **`bind()`**

```
int bind(int fd, const struct sockaddr *addr, socklen_t len);
```

- Assegna un indirizzo ad una socket
- Argomenti
  - `fd`: descrittore della socket (restituito da `socket()`)
  - `addr`: puntatore ad una struttura dati che specifica l'indirizzo
    - Per i nostri scopi: la struttura `struct sockaddr_in` va castata a `struct sockaddr`
  - `len`: dimensione della struttura dati puntata da `addr`
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

Funzione `listen()`

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

- Marca la socket come passiva, i.e., specifica che può essere usata per accettare connessioni tramite la funzione `accept()`
- Argomenti
  - `sockfd`: descrittore della socket (restituito da `socket()`)
  - `backlog`: lunghezza massima della coda per le connessioni
    - Se una connessione arriva quando la coda è piena, la connessione viene rifiutata
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

Funzione `accept()`

```
int accept(int fd, struct sockaddr *addr, socklen_t *len);
```

- Accetta una connessione su una socket in ascolto
  - È una chiamata bloccante: rimane in attesa di connessioni
- Argomenti
  - `fd`: descrittore della socket (restituito da `socket()`)
  - `addr`: puntatore ad una struttura dati `struct sockaddr` che verrà riempita con le info della socket del client
  - `len`: puntatore ad un intero che verrà settato con la dimensione della struttura dati `addr`
- Valore di ritorno
  - In caso di successo, un descrittore per comunicare col client
  - In caso di errore, `-1`, `errno` è settato

Funzione `close()`

```
int close(int fd);
```

- Nel caso `fd` sia un descrittore di socket, chiude la socket stessa
  - `read()` *successive dall'altro endpoint restituiranno* `0` *!!!*
- Argomenti
  - `fd`: descrittore della socket (ritornato da `socket()`)
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

Client Socket

- Come connettersi ad un server in ascolto?
  - Creazione socket - funzione `socket()`
  - Connessione al server - funzione `connect()`
  - Una volta terminato lo scambio di messaggi, la connessione col client va chiusa - funzione `close()`

Funzione `connect()`

```
int connect(int fd, const struct sockaddr *addr, socklen_t l);
```

- Tenta una connessione su una socket in ascolto
- Argomenti
  - `fd`: descrittore della socket (ritornato da `socket()`)
  - `addr`: puntatore ad una struttura dati `struct sockaddr` che descrive la socket alla quale connettersi (quella del server)
  - `l`: dimensione della struttura dati puntata da `addr`
- Valore di ritorno
  - In caso di successo, `0`
  - In caso di errore, `-1`, `errno` è settato

Protocollo con messaggi testuali

- Implementazione un protocollo client-server basato su messaggi di testo
  - Il server è in ascolto su una porta nota
  - Il client si connette al server
  - Inizia uno scambio di messaggi di testo secondo uno schema predefinito («protocollo»)
  - Protocollo di base
    - Il client invia una richiesta al server
    - Il server riceve la richiesta, la elabora, produce una risposta
    - Il server invia la risposta al client

Esercizio proposto: EchoServer

- Server single-thread in ascolto su una porta nota
- Il client si connette al server:
  1. L'utente inserisce da terminale un messaggio
  2. Il client invia il messaggio inserito al server
  3. Se il messaggio inviato dal client è «QUIT», entrambi terminano la connessione.
  4. In caso contrario, il server risponde con lo stesso messaggio ricevuto. Entrambi ripartono dal punto 1.
- Sorgenti: `client.c` e `server.c`

Esercizio proposto: EchoServer

- Esercizio (lato client)
  - Completare le parti mancanti, relative a:
    - Creazione e distruzione socket
    - Instaurare una connessione con il server
    - Invio/ricezione di messaggi via socket (gestire letture/scritture parziali)
      - Attenzione: non conosciamo la dimensione del messaggio
  - Per l'esecuzione, lanciare `prof_server` e `client` su terminali diversi

Esercizio proposto: EchoServer

- Esercizio (lato server)
  - Completare le parti mancanti, relative a:
    - Creazione, apertura e distruzione socket
    - Accettare una connessione in ingresso
    - Invio/ricezione di messaggi via socket (gestire letture/scritture parziali)
      - Attenzione: non conosciamo la dimensione del messaggio
  - Per l'esecuzione, lanciare `server` e `prof_client` su terminali diversi
  - Se tutto funziona, lanciare `server` e `client` su terminali diversi