ChatFe: Un sistema di chat multi-utente

Progetto del corso di Laboratorio Sistemi Operativi a.a. 2014/2015

April 19, 2015

1 Introduzione

Questo documento descrive il progetto del corso di *Laboratorio di Sistemi Operativi* a.a. 2014/2015.
Il progetto consiste nella realizzazione di un sistema di *chat* client-server per lo scambio di messaggi testuali tra un insieme di utenti.

1.1 Regole

- gli studenti possono organizzarsi in gruppi di 1,2 persone al massimo. Il numero ideale è **2**.
- il progetto deve essere sviluppato in una directory chatFe-name, dove name è la concatenazione dei cognomi dei componenti del gruppo.

  La directory di progetto deve essere strutturata nel seguente modo:

  - il file Makefile per la compilazione. Devono essere utilizzati le opzioni di compilazione '-Wall". Il make deve definire 3 target:
    - chat, default target, per compilare i sorgenti
    - install per copiare gli eseguibili chat-server e chat-client nella directory bin
    - clean per cancellare i file prodotti da una precedente compilazione
  - la directory bin che contiene gli eseguibili
  - la directory doc che contiene la documentazione
  - la directory src che contiene i sorgenti, cioè i file .c e .h
- il codice deve essere **commentato** e i nomi delle variabili devono essere **significative**. In particolare si richiede:

  - un commento all'inizio di ogni funzione che specifichi in modo sintetico l'uso della funzione e il significato delle variabili
  - un commento per ogni variabile o struttura dati che ne descrive il significato e il ruolo.
- **NON** possono essere utilizzate le istruzioni di sleep o alarm per risolvere problemi di race-condition o mutua esclusione tra processi e/o threads.

1.2 Relazione e Tempi di consegna

Gli studenti devono allegare al progetto una relazione di max 10 pagine (foglio A4, font 12pt, interlinea singolo). La relazione **NON** deve contenere il listato di tutto il codice, ma può contenere spezzoni di codice o di strutture dati al fine di spiegare il loro significato ed utilizzo. La relazione deve contenere:

- le scelte di progetto, le principali strutture dati e la descrizione degli algoritmi fondamentali
- la descrizione della struttura dei programmi
- le difficoltà incontrate e le soluzione adottate.

Il progetto e la relazione in formato PDF devono essere consegnati via email inviandoli all'indirizzo `schsst@unife.it` una settimana prima della prova di esame.
Gli studenti devono essere in grado di discutere in modo **critico** le soluzione adottate nel progetto in sede di prova orale, ed essere in grado di **modificare** su richiesta il codice.
È assolutamente vietato copiare codice sviluppato da altre persone e/o dal web.

2 Il Progetto *ChatFe*

Lo scopo del progetto è la realizzazione di un sistema client-server per lo scambio di messaggi di testo tra un insieme di utenti.
Il sistema è costituito da due applicativi:

- *chat-server*: programma server che verifica se un utente è abilitato ad accedere al servizio, mantiene traccia degli utenti connessi, e riceve e spedisce messaggi da e verso i programmi client. Il programma server è unico per tutti gli utenti.
- *chat-client*: programma client che si occupa di interagire con il server. Ogni utente che vuole dialogare esegue una istanza del programma client.

Il progetto deve essere sviluppato in ambiente Linux utilizzando esclusivamente la libreria **Pthread** per la gestione e sincronizzazione dei threads. Quindi, **non** è ammesso l'utilizzo dei semafori, ad esempio la libreria `semaphores.h`

2.1 Il programma chat-server

Il programma server viene eseguito in singola istanza su una macchina mediante il comando:

```
$ chat-server user-file log-file
```

dove:

- user-file: è un file di testo che contiene tutti i nomi degli utenti registrati al aservizio ed autorizzati ad accedere al servizio di chat. Ogni utente è specificato da:
  - un user-name, una stringa alfanumerica di max. 256 caratteri,
  - nome e cognome, stringa alfanumerica di max. 256 caratteri
  - indirizzo email, stringa alfanumerica di max. 256 caratteri

Il file descrive un utente per riga, le cui informazioni sono separate dal carattere :, ed le righe sono separate dal carattere di newline (\n).

Esempio:

```
pippo:Mario Rossi:rossi@gmail.com
pluto:Giulia Verde:verde@yahoo.com
paperino:Barbara Giallo:giallo@alice.it
```

- log-file: è un file di testo in cui il server tiene traccia del login/logout degli utenti e dei messaggi inviati. Ogni riga di informazione ha un *timestamp* e le righe sono separate dal carattere di newline (\n).

  Il formato dei messaggi è il seguente:

  ```
  timestamp:login:user-name
  timestamp:mittente:destinatario:testo
  timestamp:logout:user-name
  ```

  Esempio:

  ```
  Sun Apr 15 17:08:36 2012:login:pippo
  Sun Apr 15 17:08:42 2012:pippo:paperina:sono contento di sentirti
  Sun Apr 15 17:09:35 2012:timestamp:pluto:pluto:Ciao !
  Sun Apr 15 17:09:43 2012:logout:pluto
  ```

  I messaggi inviati in *broadcast* a due o più utenti appariranno più volte su linee diverse. Ad esempio, il messaggio inviato in broadcast dall'utente `pluto` mentre sono connessi gli utenti `pluto, qui, quo, qua` apparirà come:

  ```
  Sun Apr 15 17:08:43 2012:pluto:pluto:Ciao !
  Sun Apr 15 17:08:43 2012:pluto:qui:Ciao !
  Sun Apr 15 17:08:43 2012:pluto:quo:Ciao !
  Sun Apr 15 17:08:43 2012:pluto:qua:Ciao !
  ```

  L'ordine in cui compaiono le linee non è specificato.

Il funzionamento del server è il seguente:

- Se il file user-file esiste viene letto e caricato in una tabella **hash** gestita internamente dal server, e realizzata mediante liste di trabocco. La chiave della tabella è il **nome dell'utente** e per ogni utente memorizza il descrittore del *socket* su cui l'utente è connesso o l'indicazione di utente disconnesso.

- ad ogni richiesta di *connessione*, se l'utente è presente in tabella accetta la connessione altrimenti la rifiuta.

- ad ogni richiesta di *registrazione e connessione*, aggiunge l'utente in tabella e accetta la connessione.

- Quando il server terminerà scriverà sul file le chiavi della tabella, inclusi gli eventuali nuovi utenti che si saranno registrati durante l'attivazione del servizio.

- Dopo la creazione della tabella hash viene creato il file di log log-file, se esiste già viene troncato. Il file conterrà una intestazione del tipo:

```
******************************************************
** Chat Server started @ Sun Mar  6 15:55:39 CET 2011 **
******************************************************
```

- I messaggi di errore devono essere scritti su `STDERR` e sul file di log.
- Il server deve essere eseguito in modalià demone (cioè il processo char-server esegue una istruzione di `fork` lancia un figlio e termina.

2.2 Organizzazione del server

Il server è un processo Linux multi-threaded composto da:

- un thread main,
- un thread dispatcher
- uno o più threads workers,

Il thread main provedde alle inizializzazione della tabella degli utenti e successivamente apre una *socket* master (`AF_UNIX, AF_INET`) su i cui programmi client si possono connettere per accedere al servizio.
Il thread main esegue un loop controllato dalla variabile globale `go`, e ad ogni ciclo aspetta una richiesta di connessione da parte di un utente. Ad ogni richiesta di connessione il thread main esegue un thread `worker` in modalità detached, ovvero quando il thread termina le risorse da esso possedute sono rilasciate al sistema operativo senza la necessità che il thread main abbia eseguito una operazionedi di `pthread_join`.

Il thread worker esegue un loop controllato dalla variabile globale `go`, e ad ogni ciclo:

- attende un richiesta dal client
- registra il comando nel log-file; l'accesso al log file deve essere fatto in mutua esclusione per evitare di mischiare le stampe con altri threads;
- se il client ha inviato un comando di registrazione, o di listing degli utenti connessi, allora lo esegue;
- se il client ha inviato una richiesta di spedire un messaggio ad un utente singolo o in broadcast, allora lo passa al thread dispatcher; l'interazione con il thread dispatcher avviene secondo il modello **produttore-consumatore**, mediante un buffer circolare di dimensione `K`;

Il thread dispatcher esegue un loop controllato dalla variabile globale `go`. Ad ogni ciclo estrae una richiesta dal buffer circoalre; estrae dalla richiesta il nome del destinatario, e mediante questa stringa accede alla tabella hash degli utenti registrati, e se l'utente è connesso estrae l'identificatore del socket, e invia il messaggio al destinatario o ai destinatari in caso di broadcast.
L'accesso agli elementi della tabella hash deve essere fatto in mutua esclusione, per evitare l'accesso concorrente con i thread worker a elementi che condividono la medesima chiave hash.

Il server termina se gli viene inviato da shell il segnale di `SIGTERM` o `SIGINT`. In questo caso la variabile globale `go` viene messa a zero, e tutti i thread escono dal ciclo loop. Il thread main attende la terminazione del thread dispatcher e dei thread worker attivi. In numero di thread worker attivi è mantenuto in una variabile globale `numThreadAttivi`; questa variabile è incrementata dal thread main ogni volta che viene creato un thread, e decrementata dai thread worker che terminano a seguito di una richiesta di disconnessione inviata dal proprio client. Successivamente chiude la connessione verso i client e termina.

3 Il programma chat-client

Il programma client viene eseguito mediante il comando:

```
$ chat-client [-h] [-r "Name Surname email"] username
```

Il funzionamento del client è il seguente:

- effettua il parsing delle opzioni, e se il parsing è corretto, cerca di collegarsi con il server. In caso di insuccesso il client reitera la connessione per un massimo di 10 volte. Ogni tentativo di connessione è effettuata a distanza di 1 secondo dal precedente.
- Se il collegamento ha successo invia un messaggio di `LOGIN`. Se il client è stato eseguito con l'opzione `-r` allora invia il messaggio di `REGISTER_AND_LOGIN`.
- Se il collegamento è accettato dal server il client crea due thread:
  - un thread che si occupa di leggere i messaggi da `STDIN` ed inviarli al server
  - un thread che si occupa di ascoltare i messaggi del server e scriverli su `STDOUT`
- I messaggi di errore devono essere scritti su `STDERR`.
- l'opzione -h stampa un messaggio di uso che riassume il funzionamento, i comandi ed il loro significato.

Le operazioni implementati dal client sono:

- invio di un messaggio ad un singolo utente, basta scrivere il messaggio nel seguente formato:

  #dest destinatario:testo\n

- invio di un messaggio in broadcast, basta scrivere il messaggio nel seguente formato: (\n):

  #dest :testo\n

- I messaggi possono contenere qualsiasi carattere eccetto il "#" con cui si identificano i comandi. I comandi supportati sono:
  - #ls: per conoscere la lista dei degli utenti connessi
  - #logout: per sconnettersi dal server e terminare il client
  - #dest: per specificare un destinatario. Se il destinatario è vuoto il messaggio è inviato in broadcast.

I messaggi in arrivo dal server sono visualizzati dal client nel seguente modo:

- i messaggi di broadcast sono visualizzati nel seguente formato

  mittente:*:testo\n

- i messaggi inviati ad un solo utente sono visualizzati nel seguente formato

  mittente:destinatario:testo\n

- Il server può terminare mediante il segnale di `SIGTERM` o `SIGINT`.

4 Protocollo di Comunicazione

Client e server si scambiamo messaggi mediante una socket. Il formato dei messaggi è definito dalla seguente struttura:

```
typedef struct {
  char          type;      // message type
  char        * sender;    // message sender
  char        * receiver;  // message receiver
  unsigned int msglen;     // message length
  char        * msg;       // message buffer
} msg_t;
```

- il membro `type` definisce il tipo del messaggio. I messaggi possono avere i seguenti tipi:

```
#define MSG_LOGIN   'L'  // login
#define MSG_REGLOG  'R'  // register and login
#define MSG_OK      'O'  // OK
#define MSG_ERROR   'E'  // error
#define MSG_SINGLE  'S'  // message to single user
#define MSG_BRDCAST 'B'  // message broadcasted
#defien MSG_LIST    'I'  // list
#define MSG_LOGOUT  'X'  // logout
```

- il membro `length` definisce la lunghezza del campo `buffer`, incluso il terminatore di stringa `'\0'`.
- il membro `buffer` è un puntatore ad un array di caratteri.

4.1 Messaggi da Client a Server

I messaggi da client a server sono i seguenti:

- `MSG_LOGIN`: messaggio di login spedito dal client quando tenta di connettersi:
  - `sender`: vuoto
  - `receiver`: vuoto
  - `msglen`: lunghezza dell'array message
  - `msg`: contiene la stringa user-name

  Il server risponderà con un messaggio di `MSG_OK` o `MSG_ERROR`.
- `MSG_REGLOG`: messaggio di registrazione e login.
  - `sender`: vuoto
  - `receiver`: vuoto
  - `msglen`: lunghezza dell'array message
  - `msg`: contiene la stringa user-name:Name Surname:email

  Il server risponderà con un messaggio di `MSG_OK` o `MSG_ERROR`.
- `MSG_LOGOUT`: messaggio di disconnessione.

  - `sender`: vuoto
  - `receiver`: vuoto
  - `msglen`: 0
  - `msg`: vuoto

- `MSG_LIST`:
  - `sender`: vuoto
  - `receiver`: vuoto
  - `msglen`: 0
  - `msg`: vuoto

- `MSG_SINGLE`: messaggio per un singolo utente.
  - `sender`: vuoto
  - `receiver`: user-name
  - `msglen`: lunghezza del messaggio
  - `msg`: testo del messaggio

- `MSG_BRDCAST`: messaggio di broadcast per tutti gli utenti connessi.
  - `sender`: vuoto
  - `receiver`: vuoto
  - `msglen`: lunghezza del messaggio
  - `msg`: testo del messaggio

4.2 Messaggi da Server a Client

I messaggi da server a client sono i seguenti:

- `MSG_OK`: messaggio di OK.
- `MSG_ERROR`: messaggio di errore. Il campo `buffer` contiene il messaggio di errore.
- `MSG_LIST`: messaggio di risposta ad una richiesta di conoscere l'insieme degli utenti connessi. Il campo `buffer` contiene la lista degli utenti nel seguente formato:

  pluto:paperino:qui:quo:qua

- `MSG_SINGLE`: messaggio spedito da un utente ad un'altro utente.
  - `sender`: user-name
  - `receiver`: vuoto
  - `msglen`: lunghezza del messaggio
  - `msg`: testo del messaggio

- `MSG_BRDCAST`: messaggio di broadcast spedito da un utente a tutti.

- `sender`: user-name
- `receiver`: vuoto
- `msglen`: lunghezza del messaggio
- `msg`: testo del messaggio

A Funzione per creare un timestamp

Esempio di funzione per creare il timestamp

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <time.h>
#include <string.h>

void timestamp ( char * ts ) {
  time_t t;
  t = time(NULL);
  ctime_r(&t, ts);
  ts[strlen(ts)-1] = '\0'; // remove newline
}

int main () {

  int i ;
  char ts[64];

  for (i=0; i<10; i++) {
    timestamp(ts);
    printf("%s\n", ts);
    sleep(1);
  }

  return 0;
}
```

B Esempio di processo daemon

```
int main (int argc, char * argv[]) {
  ....
  pid = fork();

  if ( pid == 0 ) {
    chat-server();
  } else if ( pid < 0 ) {
    perror("ERROR: fork failed");
    exit(-1);
  }

  // bye bye
  exit(0);
}
```

C Tabella degli utenti

Esempio di funzione hash da utilizzare per accedere alla tabella

```c
/* numero primo sexy (991,997): 997-991=sex=6 */
#define HL 997

int hashfunc(char * k) {
  int i = 0;
  int hashval = 0;
  for ( i=0; i < strlen(k); i++ ) {
    hashval = ((hashval*256) + k[i]) % HL;
  }
  return hashval;
}
```