Relazione progetto ChatFe

Franco Masotti

October 9, 2014

Contents

Part I
Il progetto

1 Scelte di progetto

Ho deciso di suddividere ogni problema significativo ed utilizzato spesso in funzioni. Se la funzione é utilizzata all'interno di piú file sorgenti allora questa si trova all'interno di un file sorgente a sé stante. Client e server affrontano alcuni problemi comuni (come la gestione degli errori, lettura/scrittura sui socket, ecc...) e per questo utilizzano funzioni condivise. Funzioni utilizzate solo all'interno di un determinato file sorgente rimangono in quel file sorgente.

Il server esegue tre tipi di thread: thread main, worker e dispatcher. Per lavorare piú facilmente e per chiarezza ho creato tre file sorgenti (uno per ogni tipo di thread). Ho ritenuto che per il client non fosse necessario creare piú di un sorgente dato che si tratta si un applicativo piú semplice rispetto al server.

Per testare il server (prima di aver scritto il client) ho utilizzato una connessione `telnet` allinterno di una shell:

```
$ telnet 127.0.0.1 1234
```

attraverso il quale ci si collega direttamente al server. Per questo motivo, non potendo prevedere l'input di un utente attraverso questo tipo di connessione, il server fa rigorosi controlli di input. Per utilizzare `telnet` é neccessario conoscere il protocollo di comunicazione della chat.

Il protocollo di comunicazione é basato su cinque campi, di lunhezza massima prefissata e ognuno con un significato ben preciso, separati dal carattere ':'. Quesi campi sono definiti nella struttura `msg_t`. In questo modo é piú semplice sia per il server sia per il client la lettura di ogni campo. É sufficiente infatti tokenizzare ogni campo basandosi sui ':', e non, ad esempio, su un metodo che utilizza dei numeri per rappresentare la lunghezza di un dato campo.

Ogni file sorgente (`*.c`) ha un corrispondente file header (`*.h`). Il compilatore esegue la compilazione codizionale basandosi sul file `main_includes.h` evitando cosí che un file sia incluso piú volte del necessario, il che genererebbe errori. In questo modo ho potuto separare alcune funzioni chiave dalle altre, in modo semplice.

Tutte le funzioni di allocazione di memoria (come `malloc`, `strdup`) vengono controllate in caso di errore. Ho anche utilizzato funzioni con protezione contro buffer overflow (come `strncat`, `snprintf`, `fgets`) quando possibile, poiché non si puó sempre prevedere l'input dell'utente.

2 Principali strutture dati

La gestione degli utenti viene fatta inizialmente da una lista riempita all'avvio del server. La lista é definita da una struttura di tipo `users`. Questa é usata insieme alla tabella hash in modo da agevolare le operazioni di ricerca, aggiunta e invio di messaggi tra utenti.

Per passare argomenti ai thread vengono usati dei puntatori (di tipo `void *`) a delle strutture dati. Queste strutture contegono informazioni necessarie al funzionamento dei thread. Ad esempio al `thread worker`, che si occupa della comnicazione tra client e server, viene passato il `client socket descriptor, csd_socket`, del client appena connesso[1]:

```
// vedi do_tworker.h
struct thread_worker_struct // dati thread worker
{
  int csd_socket; // client socket descriptor
  users *head_usrs_list;
  td_c_p *td_mux_struct; // serve per comunicazione con thread dispatcher
  tw_log_c *tw_mux_struct; // scrittura su log file
};
```

I mutex e le informazioni associati ai thread sono presenti negli header del `thread worker` e `thread dispatcher`. Qui sono definiti nelle rispettive strutture:

```
// vedi do_tworker.h
struct _tw_log_c // thread worker log control
{
  pthread_mutex_t mux_log; // semaforo per log file
  pthread_cond_t cond_log; // coda di attesa per log file
  int w_logfile;
};

// vedi do_tdispatcher.h
#define K SOMAXCONN / 2
#define REQ_NUM K / 2 // numero max di richieste

struct _td_c_p
{
  char *buff_r[K]; // buffer delle richieste
  int pos_r;
  int pos_w;
  int count; // numero totale di elementi
  pthread_mutex_t mux_tdispatcher;
  pthread_cond_t full;
  pthread_cond_t empty;
```

[1] Per questioni di spazio riferirsi al codice sorgente per i commenti

```
};
```

Nel thread worker é definito il mutex, la condition e l'intero che si riferiscono alla scrittura sul file di log in modo che un solo `thread worker` alla volta scriva sul log. Il `thread dspatcher` invece ha bisogno dei mutex perché deve entrare in funzione quando il buffer delle richieste non é vuoto (cioé `count != 0`) altrimenti rimane in attesa di una `pthread_cond_signal` dal `thread worker`. Le variabili `pos_r` e `pos_w` definiscono rispettivamente la posizione corrente di lettura e scrittura sul buffer circolare `buff_r`, la prima incrementata dal `thread dispatcher` la seconda incrementata dal `thread worker`.

Assieme alla tabella hash ho creato una lista singolarmente concatenata per gestire alcune informazioni riguardanti gli utenti. Queste informazioni comprendono il nome utente (`username`) e il numero corrente della richiesta associata a tale utente (`req`). Questa struttura é utilizzata nelle operazioni di rierca, in particolare la stampa degli utenti collegati, nell'invio dei messaggi singoli e broadcast (nei quali viene effettivamente usato `req`) e durante la disconnessione del client:

```
// vedi hash.h
struct _users
{
  char *username;
  int req; /* numero della richiesta
  * da usare per la comunicazione
  * tra tworker e tdispatcher */
  struct _users *next; // puntatore al successivo
};
typedef struct _users users; // ridefinizione struttura
users usrs; // dichiarazione instanza globale struttura
```

3 Descrizione degli algoritmi fondamentali

Nel `thread main` viene aperto il file degli utenti `user file` e viene letto riga per riga con fgets:

```
while (fgets (tmp_f_l_buf, sizeof (tmp_f_l_buf), tmp_usr_fd) != NULL)
{
  insert = (hdata_t *) malloc (sizeof (hdata_t));
  ret_str_token_read = str_token_read (tmp_f_l_buf, insert);
  if (ret_str_token_read == 0)
  {
      insert -> sockid   = -1;
      search = CERCAHASH (insert -> uname, H);
      if (search == NULL)
      {
          INSERISCIHASH (insert -> uname, insert, H);
          head_do_tmain = ADD_USR_TO_LIST (insert -> uname, head_do_tmain);
      }
      else
          err_handler_argv (E_USR_ALREADY_EXISTS, insert -> uname, 0, 0);
  }
}
```

Ogni riga del file corrisponde alle informazioni di un utente singolo. La riga viene tokenizzata e salvata nella tabella hash grazie alla funzione `str_token_read`. Successivamente viene controllata la validitá dei dati della riga, in particolare se esiste una riga con lo stesso nome utente. Questo viene fatto accedendo alla tabella hash con `CRECAHASH`. Se i campi non corrispondono al tipo di dato previsto allora vengono o segnalati o ignorati.

Alla chiusura del programma, nel `thread_main` vengono salvati tutti gli utenti dalla hash table allo `user-file`. Viene ispezionata la lista, elemento per elemento, con il putatore tremporaneo tmp e per ogni utente trovato nella tabella hash (`search != NULL`) vengono stampate le informazioni nello `user-file` (con `snprintf`):

```
tmp = head_do_tmain;
while (tmp != NULL)
{
  search = CERCAHASH (tmp -> username, H);
  if (search != NULL)
  {
    len_usr_record = strlen (search -> uname)
    + strlen (search -> fullname)
    + strlen (search -> email) + 4;
    usr_record = (char *) malloc (sizeof (char) * len_usr_record);
    if (usr_record == NULL)
      err_handler (E_MALLOC, 'x', 'w');
```

```
    bzero ((void *) usr_record, len_usr_record);

    snprintf (usr_record, len_usr_record, "%s:%s:%s\n", search -> uname,
    search -> fullname, search -> email);

    ret_io_bytes = write (usr_fd, usr_record, len_usr_record - 1);
    if (ret_io_bytes != (ssize_t) len_usr_record - 1)
      err_handler_argv (E_WRITE_USR_RECORD_FILE, search -> uname, 0, 'w');

    free (usr_record);
  }
  tmp = tmp -> next;
}
```

Dopo l'accettazione di un client il `thread_main` avvia un nuovo `thread_worker` il quale si mette in attesa di comandi con la funzione `get_and_parse_cmd` presente nel file sorgente `parse.c`. La funzione si blocca subito su una read di 1 Byte. Questo viene fatto per avere un "blocco" affinche la systemcall `ioctl` (con il flag `FIONREAD`) possa calcolare il numero di Byte in attesa nel socket cosícché le funzioni di allocazione della memoria (es: `malloc`) possano funzionare correttamente. Poiché ogni campo del messaggio in entrata é separato dal carattere ':', all'interno di `get_and_parse_cmd` viene fatto un controllo preliminare del formato. Se il controllo é superato, viene chiamata la funzione demar che fa il demareshalling vero e proprio. É da notare che `get_and_parse_cmd` é una funzione condivisa, quindi utilizzata dal `thread_receiver` nel client:

```
[...]

ret_io_bytes = read (csd_local, dummy_byte, 1);
if (ret_io_bytes == 0 || errno == EINVAL)
  return 1;

ret_io_bytes = ioctl (csd_local, FIONREAD, &total_msglen);

[...]

if (total_msglen < 5)
{
  read (csd_local, dummy_buff, (size_t) total_msglen); // flush socket
  free (dummy_buff);

  return 2; // bad cmd
}

[...]
```

```
buff_msg = (char *) malloc (sizeof (char) * total_msglen);
if (buff_msg == NULL)
  err_handler (E_MALLOC, 'x', 'w');
bzero ((void *) buff_msg, total_msglen);

ret_io_bytes = read (csd_local, buff_msg, total_msglen);
if (ret_io_bytes != (ssize_t) total_msglen)
  err_handler (E_READ_SOCK, 'x', 'w');

[...]

ret_demar = demar (final_buff_msg, msg_t_local);

[...]

free (final_buff_msg);

if (ret_demar == 2) // comando sconosciuto
  return 2; // ritorna comando sconosciuto
if (ret_demar == 3) // formato messaggo non valido
  return 3; // ritorna formato non valido

return 0;
```

Part II
Descrizione della struttura dei programmi

Tutti i file sorgenti (`*.c`) si riferiscono all'header `main_includes.h`. Ogni file sorgente ha il proprio file header.

I programmi sono strutturati in modo che venga controllato il valore di ritorno delle funzioni fondamentali e che venga fatto il casting esplicito quando necessario, per evitare problemi e warning del compilatore o durante l'esecuzione dei programmi. Ad esempio nella funzione `write_logfile_cmd`, presente nel sorgente `do_tworker.c` viene controllato il valore di ritorno della `write` attraverso il casting esplicito da un tipo `size_t` a `ssize_t`.

```
[...]

ret_io_bytes = write (log_fd, "\n", strlen ("\n"));
if (ret_io_bytes != (ssize_t) strlen ("\n"))
  err_handler (E_LOG_FILE_WRITE, 0, 'w');

[...]
```

Anche se potrebbe sembrare inutile, questo garantisce la correttezza dei programmi.

Per semplificare la gestione degli errori ho definito delle macro in `err_handler.h` ognuna corrisponente ad una stringa. In questo modo il codice risulta piú pulito ed é sufficiente cambiare una volta la stringa per vederla cambiare in tutto il programma.

Part III
Difficoltá e soluzioni adottate

La maggior parte dei problemi si sono verificati con la gestione delle stringhe e dei socket.

Per risolvere il primo problema ho fatto riferimento allo starndard C che prevede che ogni stringa abbia il proprio terminatore, cioé il carattere '\0'. Per questo motivo prima di ogni `malloc` c'é una chiamata alla funzione `bzero` che consente di azzerare tutti i bit di una zona di memoria. Quando non é stato possibile usare `bzero` ho settato come ultimo carattere dell'array, '\0'.

Per quanto riguarda la gestione dei socket il problema sta nell'utilizzo di funzioni bloccanti, come `recv` o `read` che creano molti problemi quando si lavora con piú thread. Ad esempio alla terminazione del server, se un client é collegato, il server é bloccato sulla `recv`, viene chiuso il socket sul quale l'utente é collegato in modo da generare un errore (`EINVAL`) che viene gestito come errore paricolare. Questo puó essere verificato nella funzione `get_and_parse_cmd`:

```
[...]

ret_io_bytes = read (csd_local, dummy_byte, 1);
if (ret_io_bytes == 0 || errno == EINVAL)
  return 1;

[...]
```

Un metodo simile é stato applicato nel client quando si effettua il logout:

```
[...]

case 1: // fatal err or closed socket
{
  go = 0;
  fprintf (stderr, "Connessione rifiutata o disconnessione\n");
  fclose (stdin);
  break;
}

[...]
```

In questo caso il thread receiver, quando rileva che il socket é stato chiuso (cioé `case 1:`) chiude `stdin` con `fclose(stdin)` in modo che `fgets` nel thread sender si sblocchi e ritorni `NULL`, cosí da uscire in sicurezza:

```
[...]

ret_fgets = fgets (msg, sizeof (msg), stdin); // get msg from stdin
```

```
if (ret_fgets != NULL) // se stdin  ancora aperto

[...]

else // riferito all'if sopra
{
  go = 0;
  fprintf (stdout, "\n"); // vai a capo dopo disconnessione
}

[...]
```