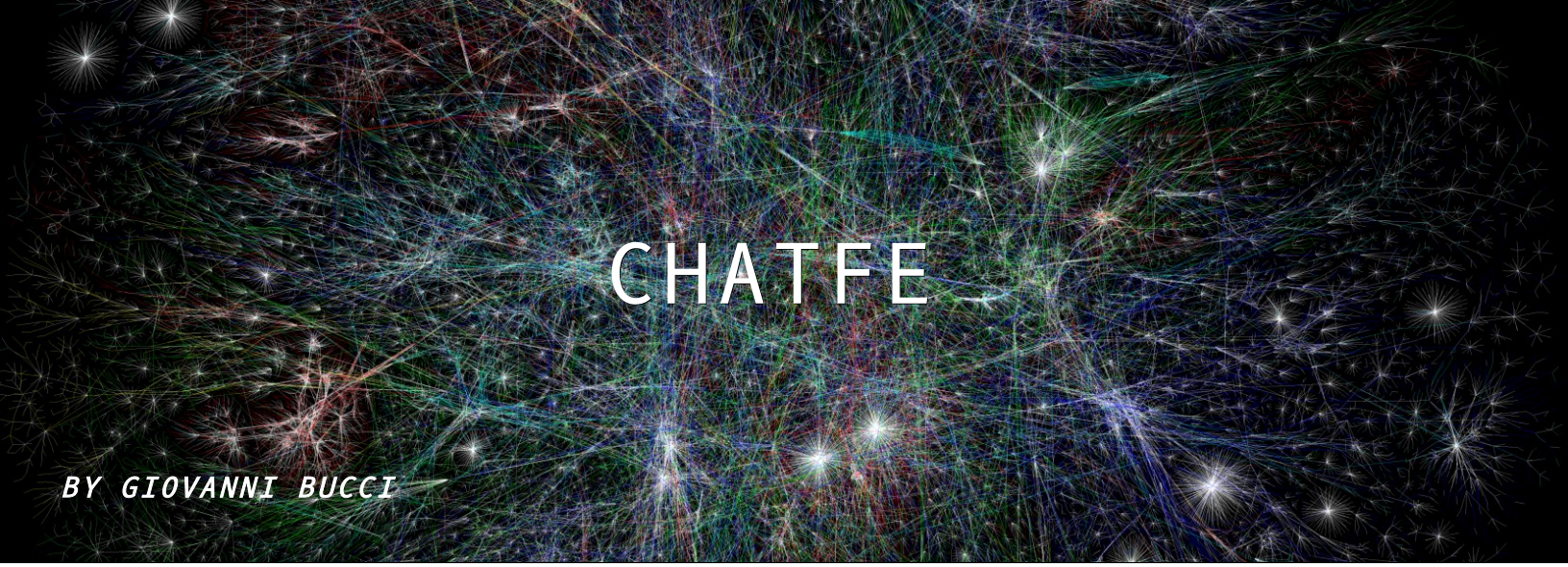

Il Progetto

Durante il corso di Sistemi Operativi ci è stato chiesto di realizzare una chat in C avente una struttura Client-Server e che utilizzasse la parallelizzazione delle risorse e delle funzioni attraverso la libreria Pthread. Il progetto sviluppato è interamente FOSS e risiede in un repository pubblico su Github (link a fine documento).

La Struttura

Il Server ha il compito di gestire tutte le richieste provenienti dai Client che ne fanno richiesta. I componenti (thread) che agiscono sono differenti in base al comando che devono eseguire. Al momento dell'avvio del Server infatti vengono creati due thread: il *ThreadMain ed il ThreadDispatcher.* Il primo ha il compito di restare in attesa delle possibili connessioni da parte di diversi Client. Ad ogni Client connesso viene associato un *ThreadWorker* personale che ha il compito di riempire la struttura *msg_t* con i dati formattati provenienti dal socket. Se il messaggio è di tipo *“login”, “logout”, “registrazione e login” o “listing”* lo esegue facendo uso delle funzioni omonime. Il restante set di comandi eseguibili dal Client viene gestito dal *ThreadDispatcher* che agendo in mutua esclusione sulla struttura *BufferPC* per evitare race conditions esegue l'attività principale della chat: lo scambio di messaggi tra utenti.

Il Client è logicamente strutturato in modo diverso. Esso è composto da 3 importanti blocchi di codice. Il primo che viene richiamato è il Main del programma. In base ai parametri che gli vengono passati da console è in grado di riconoscere quale servizio richiedere al server (login o registrazione). Dopo aver effettuato la connessione con il server invia una stringa formattata in un modo particolare che indica il tipo di servizio da ricevere (Marshalling); se la precedente richiesta ha avuto successo, vengono creati due Thread che hanno il compito di leggere da tastiera le richieste dell'utente e di restare in ascolto delle risposte provenienti dal Server sulla socket creata con lo stesso. A questo punto inizia il

vero e proprio scambio di messaggi Client-Server-Client. La connessione rimane attiva finché il client non invia un messaggio di *“#logout”* o finché il server non viene chiuso attraverso l'invio di segnali di tipo SIGINT o SIGTERM.

Marshalling?

Il marshalling è una tecnica che, attraverso una specifica formattazione di una stringa, consente di inviare dati tra due processi che non condividono gli stessi dati. Questo è ovviamente necessario perché è impossibile inviare per esempio una struttura dati considerato che verrebbe inviata la locazione alla struttura invece che il suo contenuto. Pertanto, viene letta la struttura da elaborare ed il suo contenuto viene copiato dentro ad una stringa che verrà successivamente inviata attraverso un canale di comunicazione.

Come è stato implementato

Il Marshalling implementato in questo progetto ha ricevuto innumerevoli cambiamenti (come possono dimostrare i Commit effettuati sul repository online). Ho cercato sin da subito di creare il miglior codice possibile che potesse soddisfare due principi secondo me fondamentali: il minor invio di Byte sulla socket (sia da parte del client che da parte del server) per rendere la conversazione meno pesante ,e lo snellimento del lavoro del Server per gestire i vari messaggi.
All'inizio la soluzione che avevo adottato e che mi sembrava migliore comportava l'utilizzo di messaggi aventi un formato simile al seguente:

S005pippo005pluto00012ciao a tutti

Questo formato ha il pregio di soddisfare efficacemente il primo principio ritenuto da me essenziale (inviare il minor numero di Byte sulla socket) ma non il secondo (il server è obbligato ad effettuare 7 read dalla Socket per recuperare le 5 componenti del messaggio: *type, sender, receiver, msgLen, msg*).
Dopo numerosi commit, tentativi e Segmentation Fault dovuti ad una serie di errate letture dalla socket (probelmatiche dovute alla lettura o meno dei caratteri di terminazione di stringhe '\0'), ho ritenuto soddifacente il compromesso ottenuto dalla soluzione che poi si è rivelata quella finale: la stringa che ora viene inviata dal Client al Server è ora così composta:

000034S005pippo005pluto00012ciao a tutti

Il messaggio ora è preceduto da 6 Byte che indicano la lunghezza dell'intera stringa; ritengo che sia un equo compromesso perché, utilizzando questa specifica formattazione, il server per svuotare la socket ora deve impiegare solamente 2 *read()* (una per ricevere la lunghezza e l'altra per leggere l'intero messaggio) alleggerendo quindi il carico del Server.

Principali Componenti

La maggior parte del codice del progetto è stato implementato nel server. Il client infatti ha solamente il compito di leggere i comandi dalla tastiera dell'utente, formattarli in modo che il server rieca a comprenderli, inviarli ed attendere una risposta. Il server invece ha una struttura molto complessa.
L'intera chat si basa su una Hash Table che ha il compito di mantenere in memoria i dati relativi a tutti gli utenti che usufruiscono del servizio; è quindi necessario che tutte le funzioni che ne fanno utilizzo debbano agire nel “rispetto” delle altre, ovvero in mutua esclusione. Le due funzioni che sono il cuore pulsante dell'intero progetto sono suddivise in due files separati: sono il threadWorker ed il threadDispatcher. Essi sono strettamente correlati in quanto il primo carica i dati che gli vengono passati dal client dentro una struttura dati e successivamente comunica attraverso un Buffer Produttore-Consumatore i dati che devono essere inviati attraverso la socket. Il Dispatcher è composto da due principali algoritmi:

```
void writeOnBufferPC(char *msg);
int readFromBufferPC(char **sender, char **receiver, char **msg);
```

questi agiscono a stretto contatto con una struttura dati avente tutti i campi necessari a contenere i principali dati dei messaggi e i componenti che assicurano una corretta mutua esclusione:

```
typedef struct {
        char *message[K];
        char *receiver[K];
        char *sender[K];
        int readpos, writepos;
        int count;
        pthread_mutex_t buffMux;
        pthread_cond_t FULL;
        pthread_cond_t EMPTY;
} buffStruct;
```

**msg* viene popolato dentro al threadWorker attraverso una specifica funzione e contiene tutti i dati necessari che, analizzati e suddivisi, faranno parte della struttura precedentemente descritta.
Il compito di *readFromBufferPC()* è quello di riempire i buffer che ha ricevuto come parametri con il sender, receiver ed il messaggio che serviranno in fase di send verso i client.

Signal Handling

La consegna richiede che alla ricezione di un segnale di tipo SIGINT o SIGTERM, il server esegua una manovra di uscita che consenta di interrompre lo scambio di messaggi, salvare i nuovi utenti nello user-file e terminare. Tutte queste operazioni sono effettuate attraverso la funzione

void sighand(int sig);

In primo luogo ha il compito di settare la variabile booleana *go* a *false*, facendo terminare tutti i loop controllati da questa condizione. Al momento dello sviluppo però si è presentato un problema: tutti i threadWorker attivi erano bloccati in attesa di una nuova lettura dalla socket sulla funzione *read()*. La soluzione adottata (dopo molti tentativi superflui) è stata quella di “saziare” la richiesta sulla quale il thread era bloccato. Dentro la funzione *sighand()* è stata infatti inserita una connessione “fasulla” verso il server stesso, con lo scopo di sbloccare il threadWorker e farlo terminare correttamente.
Si è posto inoltre un'altro problema dovuto alla terminazione del server: come far disconnettere tutti i client connessi?
E' stata riutilizzata quindi la funzione *writeOnBufferPC()* che, attraverso un messaggio speciale, carica nella struttura ed invia in BROADCAST a tutti i client connessi una richiesta di disconnessione che il client è in grado di decifrare.
Il threadListener del Client infatti contiene una condizione simile:

```
if (strncmp(buffer, P_LOGOUT, 6) == 0) {
        loggedOut = true;
}
```

Il loop per la lettura, essendo controllato dalle variabili *go && !loggedOut,* esce, portando alla terminazione del processo.

Filosofia di Programmazione

Le soluzioni implementative utilizzate in questo progetto hanno voluto rispecchiare due principali filosofie che caratterizzano (in genere) qualsiasi metodo di programmazione che riguardi in qualche maniera il mondo dell' OpenSource: KISS e “Do one thing, and do it well”.
Gli algoritmi che reggono questo progetto (quasi tutti) sono strutturati in modo tale che siano efficienti (per esempio non esistono array di dimensioni finite (tranne che per definire la dimensione massima dello username, nome, cognome ed email dell'utente) ma tutti i buffer utilizzati sono allocati dinamicamente in base alle esigenze delle stringhe che devono contenere).
Ho provato a strutturare le funzioni il più chiaramente possibile, rendendole snelle e capaci di eseguire un solo compito, garantendo quindi di essere portatili ed espandibili.
Alcune regole per me essenziali al fine di una corretta lettura del codice del

programma sono state decise prima della reale implementazione:
L'utilizzo di una nominazione delle funzioni di tipo *CamelCase* e il preferire l'uso di variabili booleane al posto di variabili intere (dove ovviamente è consentito / utile) rende il codice più facilmente interpretabile e maggiormente ligio alle “buone regole” del C11.

Un po' di numeri

Il repository su Github conta 54 commit, il primo è stato effettuato il 22 Aprile.
Dentro la cartella *src/* della root del progetto sono presenti 24 Files (tra .c e librerie).
Il server è stato ospitato/testato su due diverse macchine: un MacBook del 2008 sul quale è installato Manjaro 8.12 e su una VPS dalle caratteristiche scadenti sulla quale gira Ubuntu 13.4.
Analizzando il progetto con uno strumento chiamato Cloc si ottengono le seguenti informazioni riguardanti il numero di righe totali, il numero di commenti ed il numero di righe di codice effettivamente scritte:

```
chatFe-Bucci|master⚡ ⇒ cloc src
      26 text files.
      26 unique files.
       2 files ignored.

http://cloc.sourceforge.net v 1.62  T=0.13 s (190.6 files/s, 15393.2 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               12            406            236           1089
C/C++ Header                    12             83             10            114
-------------------------------------------------------------------------------
SUM:                            24            489            246           1203
-------------------------------------------------------------------------------
```

https://github.com/puskin94/chatFe-Bucci