# Progetto ChatFe

Luca Pajola, Marco Tieghi

# Introduzione

Per cercare di capire al meglio il progetto e la teoria del corso che avremo applicato successivamente abbiamo cercato di capire il funzionamento dei meccanismi di threads, processi, mutex, sockets… in singolo, prima di iniziare il progetto ChatFe vero e proprio. Dopo aver preso padronanza con i meccanismi messi a disposizione dalle librerie dei sistemi Unix, abbiamo iniziato a ragionare sulla struttura dell'intera chat. Come anche definito dalla descrizione del progetto fornita, era chiaro che avremmo dovuto scrivere due programmi principali, il *server* e il *client*. Tuttavia, per avere una struttura più organica e meglio organizzata ci siamo posti anche l'obiettivo di creare alcune librerie personali di supporto: prima fra tutte una libreria che ci permettesse di gestire i messaggi, dandoci la possibilità di formattarli secondo le specifiche richieste e di eseguire marshalling e unmarshalling. A questo scopo abbiamo scritto la libreria *CMTP* (ChatFe Message Transfer Protocol). Altra libreria importante che abbiamo implementato è quella che ci avrebbe permesso di gestire in maniera più semplice il file degli utenti (*gestoreUtenti*). Librerie secondarie sono invece quelle relative alla gestione di un buffer circolare (*circBufferUtil*) e della scrittura di log nel log file (*gestoreLogFile*).

Dopo aver quindi definito il funzionamento di base di server e client, e aver testato il corretto collegamento tramite socket dei due programmi, abbiamo cominciato ad estendere le diverse funzioni degli stessi per soddisfare tutte le specifiche richieste.

# I programmi

## Il gestore dei messaggi

Abbiamo capito fin da subito che per permettere un corretto scambio di messaggi fra client e server ci serviva un protocollo di formattazione dei messaggi in uscita e in ingresso che avrebbe dovuto essere adottato uniformemente sia dal server sia dal client. A questo scopo, per non aggiungere difficoltà nella scrittura del codice dei due programmi principali abbiamo definito il *CMTP*, una serie di funzioni che possano gestire tutti gli aspetti della gestione dei messaggi della chat. La prima funzione che abbiamo definito è stata **messageCreate()**, la quale dinamicamente (i campi del messaggio variano da server a client) doveva adattare il formato di un messaggio (definito tramite struttura *msgt*): infatti definendo un numero come primo parametro della funzione si è in grado di memorizzare correttamente il nome del mittente, del destinatario e il messaggio nella struttura, a seconda che questa debba essere spedita dal client o dal server (secondo le specifiche di progetto).

Altro problema incontrato è stato scegliere un modo per poter inviare una struttura dati: chiaramente la soluzione adottata è stata di ricorrere al meccanismo di marshalling. È a questo proposito che abbiamo definito la funzione **marshalling()**

la quale ricevendo in input una struttura di tipo msg_t è in grado di formattare i campi in una stringa. La stringa risultante da questa operazione è scritta nel formato:

```
type#sender#receiver#message#
```

Si è deciso di usare come delimitatore il '#' poiché sarebbe stato un simbolo che l'utente non avrebbe potuto usare e quindi sicuro.

Viceversa, una volta ricevuta la stringa si sarebbe dovuto ricomporre il messaggio di tipo msg_t, quindi abbiamo scritto un **unmarshalling** che riceve in input la stringa e tramite la suddivisione in token (delimitati da #) ricompone il messaggio e restituisce il tipo msg_t

## Il server

Ci siamo resi conto fin dall'inizio che il server avrebbe richiesto il lavoro maggiore in quanto sarebbe stato l'elemento principale.

Il server si compone di quattro funzioni principali:

- main(): il main è piuttosto semplice, in quanto serve solo per lanciare un processo demone. Sono comunque ricavati i nomi del file utente e del log file passati dall'esecuzione e l'abilitazione della cattura di segnali (nello specifico SIGINT e SIGTERM)
- mainServer(): questa è la funzione richiamata alla creazione di un processo demone. È in questa funzione che eseguiamo la creazione e il collegamento di un socket per attendere nuove connessioni da parte di nuovi clients. Il server rimane in ascolto (tramite *listen()*) e solo quando accetta una nuova richiesta viene creato il thread worker per gestire la comunicazione con quello specifico client. Viene creato un solo thread dispatcher per gestire l'invio di tutti i messaggi ricevuti. È in questa funzione che inizializziamo il log file e carichiamo gli utenti già registrati nell'apposita struttura

- workerFunc(): questa è la funzione richiamata dal mainServer ogni qualvolta accetta una nuova connessione da parte di un client. Ogni client ha il suo worker, che comunica solo con lui. Innanzitutto, per gestire il numero totale di utenti connessi (tramite la variabile globale *numThreadAttivi*) abbiamo definito un mutex per accedere in mutua esclusione alla varibile (per incrementarla al login di un utente o per decrementarla al suo logout). In seguito, dopo aver creato il buffer che riceverà le stringhe (i messaggi) dal socket e aver memorizzato il socket di quel client (passato come argomento di funzione), eseguiamo un ciclo regolato sia dalla lettura da socket sia dalla variabile di controllo *go*. Ogni volta che riceviamo un messaggio, ne eseguiamo l'unmarshalling e a seconda del tipo procediamo diversamente: nel caso di login, verifichiamo che l'utente esista già e in caso di un qualche errore decrementiamo la variabile di threads attivi; nel caso di registrazione, memorizziamo l'utente nella tabella utenti; nel caso di un messaggio di listing, generiamo la lista degli utenti attualmente connessi e poi inviamo il messaggio al mittente; nel caso di messaggi a singolo o in broadcast inseriamo il messaggio in un buffer circolare per poterlo inviare al thread dispatcher
- dispatcherFunc(): questa è la funzione richiamata alla creazione dell'unico thread dispatcher, il quale si occupa di prelevare i messaggi singolo o broadcast dal buffer circolare. Se è un messaggio singolo, dal messaggio ricaviamo il nome del destinatario, determiniamo il suo numero di socket accedendo alla tabella utenti e gli spediamo il messaggio dopo averne fatto il marshalling. Nel caso

invece di messaggio broadcast, ricaviamo il nome di tutti gli utenti connessi e spediamo ad ognuno, ricavandone il socket number dall'usernamen, il messaggio. Il dispatcher è anch'esso gestito dalla varibile di controllo go.

- signal_handler(): questa funzione gestisce la ricezione dei segnali SIGTERM e SIGINT per il server. Ponendo go a 0, si esce da ogni ciclo e tutti i threads worker e il thread dispatcher terminano la loro esecuzione.

## Il client

Il client ha richiesto meno tempo del server in quanto ha un funzionamento più semplice. È strutturato in tre funzioni principali:

- main(): è la funzione richiamata all'avvio. Dopo aver eseguito il fetching dei parametri di input e la corretta interpretazione delle diverse possibili opzioni (-h e -r), si esegue la connessione con il server e si invia il messaggio di login o di registrazione+login (la funzione **colonTokenizer** serve in quest'ultimo caso a formattare i dati nel formato `username:name surname:mail`). Se il server risponde con esito positivo (l'utente è registrato e il server ha accettato la connessione creando il relativo worker), allora vengono creati i threads writer e listener. Al termine del main eseguiamo una join sui due thread, per terminare il client solo alla morte dei due threads.
- Writer(): è la funzione che gestisce l'invio di messaggi. Infatti una volta creato, il thread resta in attesa di ogni ogni input da riga di comando, attendendo che l'utente inserisca dei comandi validi. Se

l'utente inserisce qualcosa, il writer prima verifica se l'utente inserisce una sequenza valida (è per questo motivo che abbiamo pensato di creare una funzione apposita che verifichi i caratteri di input, **tipoMessaggioUtente**), e se lo è crea il messaggio apposito (ricorrendo al CMTP) e lo invia al server.

- Listener(): questa funzione, richiamata alla creazione del thread listener, gestisce la ricezione di messaggi. Senza troppe complicazioni, quando il client riceve un messaggio dal socket, ne viene eseguito l'unmarshalling e a seconda del tipo (singolo, broadcast, listing o errore) visualizza correttamente il messaggio ricevuto.

## Librerie di supporto

Oltre alla già definita libreria di gestione messaggi (CMTP) abbiamo deciso di dare una miglior organizzazione al progetto raccogliendo la gestione del buffer circolare, della tabella utenti e del log file in librerie separate.

- circBufferUtil.c: questa libreria si occupa di gestire interamente il buffer circolare usato dal server per permettere lo scambio di messaggi fra thread worker e thread dispatcher. Il buffer è organizzato in una struttura mesBufStr. In questa struttura sono definiti un mutex (per l'accesso in mutua esclusione al buffer), due conditions, una nel caso in cui il buffer non abbia più messaggi (e quindi non possa più estrarne) e una nel caso in cui sia pieno (e non possa più memorizzarne). Troviamo poi un array di tipo msg_t, per memorizzare i messaggi veri e propri, i due indici che definiscono la posizione delle testine di lettura e di scrittura, un contatore per il mes-

saggi pendenti e un intero che indica la dimensione massima del buffer. Sono inoltre definite tre funzioni: initStruct() per inizializzare il buffer (l'intera struttura) non appena creiamo il socket del server, extract() per estrarre un messaggio, insert() per inserire un messaggio nel buffer.

- gestoreUtenti.c: questa libreria è usata nella gestione della tabella utenti, caricata all'avvio del server e utilizzata per memorizzare le informazioni degli utenti connessi e/o registrati a ChatFe. Si basa sull'hash table fornita per memorizzare i dati, e fornisce alcune funzioni che abbiamo scritto per poter accedere alla tabella per: leggere il file degli utenti, salvarlo con le informaizoni aggiornate di nuovi utenti registrati, ricavare il socket number di un utente, creare la lista degli utenti connessi, effettuare login/registrazione/logout (aggiornando il socketID dell'utente). Abbiamo deciso di gestire gli utenti connessi in una lista separata e di tenere traccia del numero di utenti connessi e registrati. L'accesso alle bash table avviene tramite controllo su mutex.
- gestoreLogFile.c: semplice libreria per includere la funzione di stampa sul log file. Le sue funzioni sono richiamate dal server qualora si debba salvare un log nel file. Le funzioni permettono di: accedere al log file passato come parametro di input nel server, salvare un messaggio appositamente formattato per il logout/login di un utente e per i messaggi singoli e broadcast ricevuti dal server.

## Difficoltà incontrate

Le principali difficoltà incontrate durante il progetto non sono state relative alla gestione di threads, o delle connessioni fra server e client, ma hanno principalmente riguardo errori di formattazione dei messaggi, di una non corretta lettura delle stringhe ricevute e problemi di memoria.

Per l'ultimo caso abbiamo risolto cercando di verificare dove avvenissero i problemi di memoria, e conseguentemente controllare l'uso corretto di puntatori e della pulizia delle variabili usate al termine del ciclo di utilizzo così da ricevere i nuovi valori senza contaminazione di memoria.

Per i primi casi invece abbiamo dovuto lavorare per capire dove avvenissero i problemi di scorretta formattazione e verificare l'uso corretto delle funzioni di lettura dalla memoria e delle funzioni da noi scritte di marshalling e unmarhsalling (che sono state riscritte varie volte, a volte semplificandole altre complicandole, per ottenere il risultato desiderato).

Un altro problema trovato era relativo alla non chiusura corretta del server, che è stata riscontrata nel buffer circolare ed è stata risolta inserendo un messaggio vuoto prima di terminare tutti i threads (`insert(&msgBuff,messageCreate(2,NULL,NULL,NULL));`). Inoltre, per terminarlo correttamente, si è inserita una falsa connessione così da chiudere il socket.