

Relazione Laboratorio di sistemi operativi 2017/2018

Chatterbox

Stefano Spadola - Corso B - 534919

Indice

1 Introduzione.....	1
2 Struttura dei file principali.....	2
2.1 chatty.c.....	2
2.1.1 Main.....	2
2.1.2 Thread Worker.....	2
2.1.3 Terminazione.....	3
2.2 Connections.h.....	3
2.3 Threadlib.h.....	3
2.4 QueueLib.h.....	3
2.5 Userlib.h.....	4
3 Progettazione, debug e testing.....	4
4 Bug e miglioramenti del codice.....	5

1 Introduzione

Chatterbox implementa un server capace di far comunicare dei client fra di loro, consentendogli di scambiare sia messaggi testuali che file. Il progetto è strutturato in modo tale da gestire i vari client in maniera concorrente, utilizzando apposite strutture dati accedute in modo sincrono, per evitare situazioni di inconsistenza che potrebbero comportare dei malfunzionamenti durante lo scambio dei messaggi da parte dei client.

Il progetto gira attorno al file `chatty.c` che è il server vero e proprio, e servendosi di alcune librerie importate, riesce a portare a termine il proprio lavoro. Nello

specifico le librerie principali sono: parser.h queuelib.h userlib.h msgqueue.h threadlib.h e connections.h.

La struttura completa del progetto può essere osservata dall'immagine seguente:

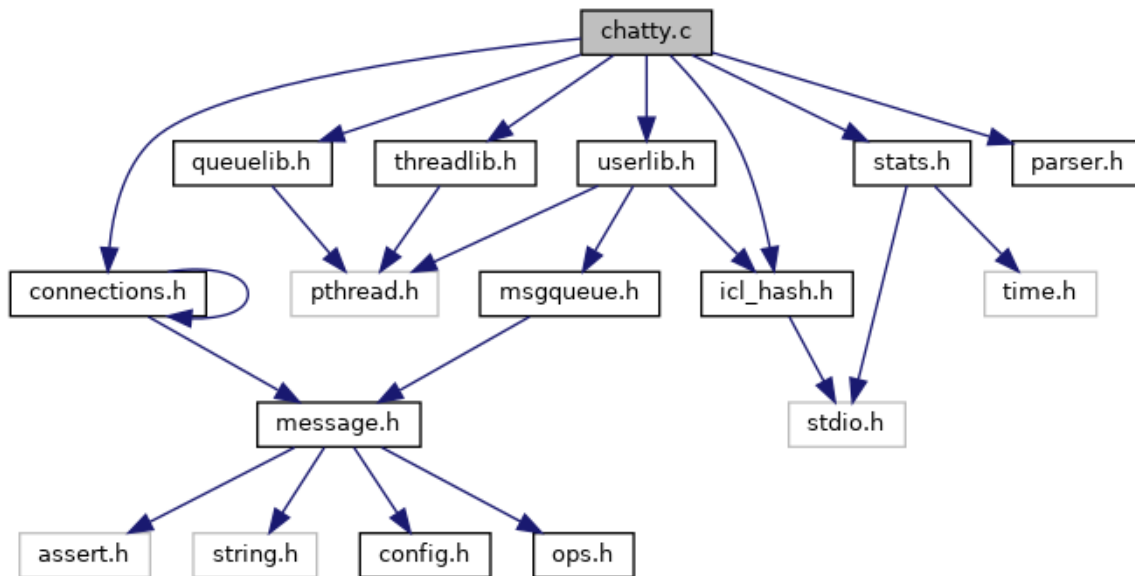


Figura 1: Diagramma dei file inclusi per `chatty.c`

*Si rimanda alla documentazione (`documentazione.pdf`) per una visione ancora più completa.

2 Struttura dei file principali

2.1 `chatty.c`

2.1.1 Main

La struttura di `chatty.c` è di per se semplice, è composta da un **main** che viene lanciato all'avvio del server (`./chatty -f <fileconfig>`), il quale provvederà a:

1. Effettuare il parsing di `<fileconfig>` passato come argomento (*attraverso `parser.h`*), e associarlo a una sua struttura dati.
2. Registrare i segnali da catturare.
3. Creare una coda per i file descriptor dei client da servire (*attraverso `queuelib.h`*).
4. Creare la struttura dati per la memorizzazione degli utenti e dei loro storici dei messaggi (*attraverso `userlib.h`*).
5. Ed infine dopo aver creato un socket per la comunicazione con i client, lancerà i **thread workers** che si occuperanno di servire i client (*attraverso `threadlib.h`*).

Il main resterà successivamente in ascolto sul canale di comunicazione da lui creato, per registrare i file descriptor degli utenti che richiedono di effettuare operazioni.

2.1.2 Thread Worker

Il carico di lavoro effettivo, è effettuato dai thread worker, che una volta estratti i filedescriptor dei client dalla coda (queuelib.h), li processano per portare a termine le operazioni richieste. Il flusso di esecuzione è semplice, una volta ottenuto il filedescriptor un worker leggerà dal canale di comunicazione il messaggio inviato dal client, e lo processerà individuando l'operazione richiesta attraverso uno switch. La funzione che viene passata ai thread worker è **"worker"**.

2.1.3 Terminazione

La terminazione del server è *attuata mediante segnali*. I segnali gestiti dal server per terminare correttamente sono: SIGINT/SIGQUIT/SIGTERM. Essi sono registrati alla creazione del main e sono strutturati in modo tale che una volta inviati e catturati dal processo, venga lanciata una funzione di terminazione che:

1. Cambia il contenuto della variabile **alive** messa a guardia dei due while responsabili della vita del **thread main** e dei **thread worker**.
2. Scrive sulla coda un "messaggio speciale di terminazione".
3. Lancia una signal ai thread che si erano sospesi sulla coda, per svegliarli.

Una volta terminati i thread, il main provvederà a effettuare una pulizia di tutte le strutture dati a cui si è appoggiato, liberando la memoria.

2.2 Connections.h

È il file utilizzato come protocollo di connessione fra il server e i client. Si appoggia su un socket AF_UNIX, e implementa lo scambio vero e proprio dei messaggi in un formato descritto dalla libreria message.h. Le operazioni sono di apertura della connessione, di scrittura e lettura dei messaggi. La comunicazione avviene in modo bidirezionale.

2.3 Threadlib.h

Questa libreria è cioè che chatty utilizza per implementare un pool di n thread worker, la struttura è semplice, implementa la creazione, l'inizializzazione e la terminazione dei thread.

2.4 Queuelib.h

È la coda utilizzata tra il **thread main** per inserire i file descriptor dei client che richiedono operazioni e i **thread worker** che lavorano per soddisfare le richieste. La dimensione è finita e settata dall'utente durante la configurazione iniziale del server, tramite il file di configurazione. La coda è strutturata per essere acceduta in muta esclusione, questo per gestire la concorrenza tra i thread (fra di loro) e tra i thread e il thread main, ed è bloccante per quanto riguarda le operazioni di estrazione da parte dei thread (se la coda è vuota).

2.5 Userlib.h

È la struttura dati principale utilizzata dal server per immagazzinare gli utenti e gli storici dei messaggi scambiati. Utilizza **due tabelle hash**, una per la memorizzazione vera e propria degli utenti (**users**), una per i relativi filedescriptor (**fdusr**).

Il motivo di tale implementazione è che un utente potrebbe effettuare una disconnessione implicita (ovvero senza inviare un corretto messaggio di disconnessione) rendendo quindi il logout possibile solo per mezzo del file descriptor. Quindi per ragioni di prestazione, è più ragionevole avere una associazione diretta per file descriptor, piuttosto che cercare per ogni utente il relativo file descriptor.

*A destra un diagramma relativo alla struttura principale

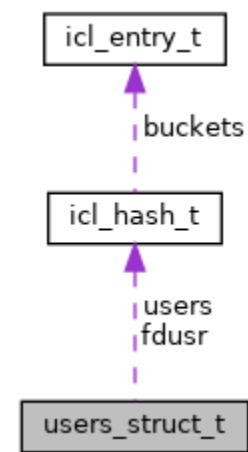


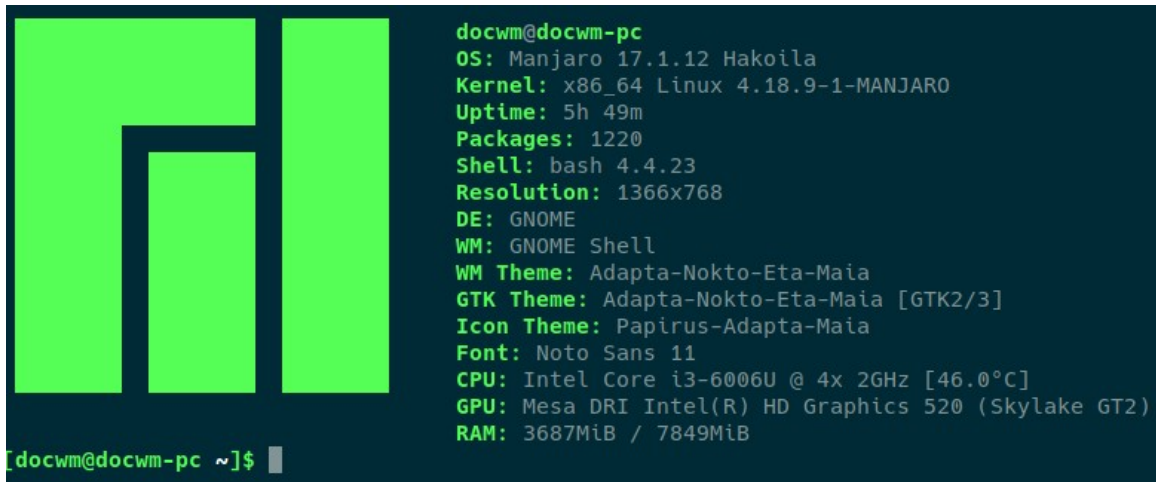
Figure 2: Struttura dati per la memorizzazione degli utenti

3 Progettazione, debug e testing

La progettazione è iniziata prima studiando per bene il codice fornito dal docente, ovvero principalmente *client.c* e il formato dei messaggi in *message.h*, per poi incominciare a implementare le funzioni di comunicazione fra client e server fornite dagli header nel file *connections.h*. Successivamente si sono implementate e testate (con dei driver scritti appositamente) singolarmente le librerie *parser.h*, *queuelib.h*, *msgqueue.h* e *threadlib.h*. Infine si sono messi assieme i pezzi, creato il gestore utenti, ridefinito alcune librerie, e testati uno per uno i testcase forniti dal docente mentre si continuava passo passo lo sviluppo del software.

Per il debug si è utilizzato **gdb**, che si è dimostrato utilissimo per scovare diversi errori di segmentazione e **valgrind** per riuscire a trovare i memory leaks.

L'ambiente di sviluppo utilizzato:

A screenshot of a terminal window with a dark blue background and light green text. On the left side, there is a large, stylized light green logo consisting of three vertical bars of varying heights. The terminal text displays system information for a Manjaro Linux environment. The prompt is 'docwm@docwm-pc'. The output includes OS, Kernel, Uptime, Packages, Shell, Resolution, DE, WM, WM Theme, GTK Theme, Icon Theme, Font, CPU, GPU, and RAM details. The prompt at the bottom is '[docwm@docwm-pc ~]\$' followed by a cursor.

```
docwm@docwm-pc
OS: Manjaro 17.1.12 Hakoila
Kernel: x86_64 Linux 4.18.9-1-MANJARO
Uptime: 5h 49m
Packages: 1220
Shell: bash 4.4.23
Resolution: 1366x768
DE: GNOME
WM: GNOME Shell
WM Theme: Adapta-Nokto-Eta-Maia
GTK Theme: Adapta-Nokto-Eta-Maia [GTK2/3]
Icon Theme: Papyrus-Adapta-Maia
Font: Noto Sans 11
CPU: Intel Core i3-6006U @ 4x 2GHz [46.0°C]
GPU: Mesa DRI Intel(R) HD Graphics 520 (Skylake GT2)
RAM: 3687MiB / 7849MiB

[docwm@docwm-pc ~]$
```

Successivamente, a progetto ultimato, sulla stessa macchina si è testato con il comando "**make consegna**" sia sul SO dove si è sviluppato, sia sulla MV fornita dal docente (xubuntu.vmdk).

4 Bug e miglioramenti del codice

Il codice così strutturato è funzionante, e passa tutti i testcase forniti. Potrebbero esserci (e anzi ci sono) dei bug non legati ai testcase, ma dai test extra che ho fatto non è emerso nulla, è anche stato fatto largo uso di valgrind per risolvere tutti i memoryleaks incontrati durante la programmazione.

Il codice sicuramente è migliorabile, una delle migliore è sicuramente il fatto di poter gestire la tabella hash degli utenti a blocchi, in quanto per un numero elevato di utenti registrati, si abbassa il livello di parallelizzazione, poiché l'accesso così come è stato implementato blocca l'intera tabella degli utenti, dando un calo delle prestazioni legate al numero degli utenti registrati.