Chatterbox

Progetto per il modulo di
laboratorio SOL a.a. 2017-2018

Valerio Besozzi 543685

Capitolo 1

Introduzione

1.1 Chatterbox

Quest'anno per il progetto di laboratorio di SOL e' stato assegnato chatterbox, un server concorrente che implementa una chat.

In questa relazione cerchero' di illustrare le varie tappe da me affrontate durante la realizzazione del progetto, concentrandomi su:

- Progettazione
- Implementazione (o Sviluppo)
- Debugging e testing
- Conclusione

In ogni tappa approfondiro' diversi aspetti, considerati da me importanti, che hanno caratterizzato la realizzazione di chatterbox.

Capitolo 2

Progettazione

2.1 Introduzione

All'inizio della fase di progettazione ho deciso di suddividere il progetto in vari moduli principali:

- Parser
- Threadpool
- Socket
- Task Manager
- Hash map

Per ogni modulo ho realizzato dei mock per verificarne il funzionamento.

2.2 Threadpool

Per sviluppare un server concorrente dovevo realizzare un sistema per dividere le richieste dei client su piu' thread. Per ottenere cio' ho deciso di utilizzare un thread pool, un design pattern per la gestione della concorrenza.

Il suo funzionamento e' semplice: il thread pool e' dotato di una coda interna di task, ogni volta che ricevo una richiesta da

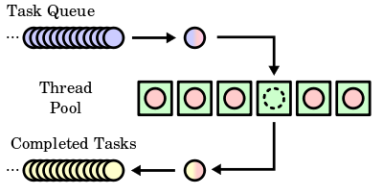

Figura 2.1: Esempio funzionamento thread pool.

un client la aggiungo alla coda, sara' compito del thread pool assegnare il task al primo thread disponibile. La Figura 2.1 illustra in modo semplice il funzionamento di un thread pool.

2.3 Hash map

Una delle parti fondamentali della progettazione di chatterbox e' stata la scelta della struttura dati per memorizzare gli utenti. Ho scelto quindi di utilizzare un hash map.

Utilizzando un hash map, operazioni come l'inserimento, la ricerca e l'eliminazione sono poco costose come complessita' temporale.

2.4 Chatterbox 2.0

Dopo aver completato il progetto utilizzando *select* per monitorare i client mi sono imbattuto in un *articolo*[1] che illustrava le differenze tra *select* ed *epoll*. Tra le principali differenze c'e' che il costo di *select* e' $O(n)$, dove n e' il numero di descrittori controllati, mentre per *epoll* e' $O(m)$, dove m e' il numero degli eventi accaduti.

Incuriosito dai vari vantaggi offerti ho deciso di sostituire *select* con *epoll*. Grazie alla suddivisione del progetto in vari moduli, in parte indipendenti fra loro, questa scelta ha portato solo alla modifica di *chatty.c* e parzialmente di *task.c*, lasciando il resto intatto.

[1] https://medium.com/@copyconstruct/the-method-to-epolls-madness-d9d2d6378642

Capitolo 3

Implementazione

3.1 Introduzione

Dopo la fase di progettazione c'e' la parte di sviluppo vero e proprio. In questo capitolo non andro' ad analizzare ogni aspetto implementativo di chatterbox, invece mi concentrero' su quelli che considero piu' importanti.

3.2 Gestione utenti

La gestione degli utenti e' essenziale in chatterbox, per questo ho creato *userdata*.

Userdata contiene: il numero di utenti connessi, le statistiche relative al server e le strutture dati contenenti gruppi e utenti registrati.

Per memorizzare gli utenti ed i gruppi ho utilizzato come struttura dati hash map.

3.3 Gestione connessioni

Come accennato in precedenza la gestione dei client connessi e' implementata usando *epoll*, ora ne parlero' in dettaglio.

Quando un client si connette viene segnato il relativo descrittore come pronto per essere letto (EPOLLIN), successivamente, dopo aver letto la richiesta del client, il descrittore viene segnato come pronto alla scrittura (EPOLLOUT). Al successivo ciclo di epoll, se un client e' marcato come pronto alla scrittura, viene recuperata la richiesta letta in precedenza, viene eseguita ed inviata la risposta.

3.4 Gestione segnali

Per la gestione dei segnali, ad esempio SIGPIPE, SIGQUIT e SIGUSR1, ho utilizzato le funzioni *signals*(), *get_stats*() e *print_stats*().

Per garantire che l'operazione di stampa delle statistiche sia signal safe ho deciso di creare un thread, chiamato *t_stats*, questo esegue la funzione *print_stats*(). Quando chatty riceve il segnale SIGUSR1, il signal handler corrispondente invoca la funzione *get_stats*(), quest'ultima avverte, tramite la variabile globale *alert*, il thread che esegue *print_stats*() il quale chiama la funzione $printStats(FILE * fout)$ che scrive su un file le statistiche del server.

3.5 Hash map

Per l'implementazione del hash map ho utilizzato *uthash.h*, un header che offre diverse macro per operare su hash table. Ho scelto uthash in quanto efficente e di facile implementazione.

Capitolo 4

Debugging e testing

> Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.
>
> *Brian Wilson Kernighan*

4.1 Introduzione

Il debugging e' una delle fasi piu' ostiche durante la realizzazione di un software, sopratutto se nelle fasi precedenti si e' creato un codice poco chiaro, progettato male e/o troppo complicato. Una delle soluzioni e' fin da subito, durante lo sviluppo, prestare attenzione alla gestione della memoria dinamica, gestione dei thread e mutex per evitare successivamente problemi difficili da individuare tipo segmentation fault, data race e deadlock.

4.2 Strumenti utilizzati

Oltre a metodi classici per il debugging ho utilizzato diversi tool per il debug di problemi di memoria e di data races. Ho utilizzato strumenti per il debug come *gdb*, *valgrind* e *Google sanitizer*[1].

[1] https://github.com/google/sanitizers

4.3 Testing

Sono stati utilizzati tutti i test case forniti oltre ad alcuni creati da me.
Il progetto e' stato testato su diverse distribuzioni linux:

- Fedora 28
- Debian Sid
- Solus
- *Gentoo*[2]
- Ubuntu 14.10

[2]Non ha superato il test 4 a causa di un bug di valgrind-3.13.0

Capitolo 5

Conclusione

5.1 Considerazioni finali

Questo progetto mi ha permesso di applicare le nozioni viste a lezione e di approfondire diversi aspetti base dello sviluppo di un software. Ho imparato ad utilizzare strumenti nuovi, ad esempio *git*, *Google sanitizer*, *make* e *doxygen*, ed a conoscere meglio strumenti gia' utilizzati in precedenza, come *valgrind*, *gdb* e *vim*.