Chatterbox 2017-2018

Giulio Purgatorio, 516292 A

g.purgatorio@studenti.unipi.it

Questa è una breve relazione che risponde ai punti richiesti nel PDF del progetto di Sistemi Operativi Laboratorio 2017-2018, **Chatterbox**.

Scelte progettuali

Come **struttura principale** troviamo una **tabella Hash** (*chatty_users*) in cui tutti i dati utili di un qualsiasi utente registrato vengono salvati, quali per esempio i messaggi pendenti, il nickname di un utente e così via.

Questa tabella Hash è composta da:

- Un array di puntatori alla struttura User, contenente il nickname, i messaggi pendenti, il suo stato (*online/offline*) e indici per cercare di ottimizzare determinate operazioni (*last_msg, oldest_msg*);
- Un array di Locks, una per ogni lista di trabocco della tabella Hash per l'accesso in mutua esclusione a tale lista;
- Un intero per la dimensione. La dimensione segue i consigli appresi durante il percorso di laurea: un numero primo, lontano dalla potenza di 2 la quale rappresentazione binaria è non 0 nei bit di ordine più alto (vedasi come esempio: 127: 0b1111111 e 128: 0b10000000). 367 è la size scelta considerando la mole dei dati nei vari test finali.

Questo implica che non ho gestito la parte opzionale del progetto, i gruppi.

Di supporto a questa tabella troviamo un **array di struct** (*onUsers*) dove è possibile reperire facilmente chi è online in un dato momento. Questa scelta è stata fatta in quanto il parametro che gestisce il numero di utenti online è relativamente piccolo nei vari test (*MaxConnections*) e quindi è preferibile avere una lista di supporto piuttosto che dover controllare tutta la tabella Hash ogni singola volta.

La struttura di base di questo array contiene l'username ed il suo File Descriptor associato (che d'ora in poi chiamerò per semplicità FD).

Infine, per quanto riguarda proprio i FD, ho implementato una **lista classica** (con normalissime operazioni di *Pop()* e *Push()*) da poter usare per i vari lavori gestiti dai Workers.

I **Workers** sono dei Threads creati dal MainFile (*chatty.c*) in modo da avere il server sempre pronto a delegare per gestire future richieste e mai a gestire esplicitamente la richiesta stessa.

Essendoci quindi dei Threads concorrenti troviamo la possibilità di Race Conditions: per questa problematica ho deciso di utilizzare varie Locks con scopo diverso, mostrando quindi un approccio rigido e conservativo per evitare di poter cadere in Deadlock. Esiste una lock come già specificato per ogni lista di trabocco, esiste una lock per la lista degli utenti online, esiste una lock per la select e così via: tutto ciò comporta Overhead, ma confrontandomi con colleghi ho trovato che i miei tempi di esecuzione sono nella norma.

Riguardo la gestione dei segnali, vengono gestiti nel MainFile come richiesto SIGPIPE, SIGTERM, SIGQUIT, SIGINT e SIGUSR1, un segnale speciale specifico di questo progetto.

I Worker hanno un ciclo di vita interrotto solo dal segnale SIGQUIT che li fa terminare seguendo la logica del Graceful Shutdown. Questi stanno in attesa su una condizione in modo da non fare attesa attiva, aspettando una richiesta da un FD da gestire, gestiscono la richiesta e ripetono il ciclo.

In questo senso è quindi intuitivo paragonare l'interazione tra client e server ad un classico Produttore-Consumatore, dove in particolare il server è l'unico Produttore ed i Threads sono i consumatori.

Testing

Per quanto riguarda il testare il progetto, prima della fase finale ho creato un piccolo client ed un piccolo server con cui testare manualmente gli esiti delle varie operazioni: una volta controllato che le operazioni singole andassero a buon fine, è bastato unirle per poter passare ai test finali inclusi nello zip del progetto. Mi sono quindi ispirato all'Unit Testing, creando dei sottoproblemi da risolvere piuttosto che risolvere per intero il problema sin dall'inizio. In particolare, oltre al testare l'esito di ogni operazione che non mostra i flussi interni d'esecuzione, ho usato un Debugger, quello di Visual Studio Code, in modo da poter controllare lo stato delle variabili e simili in ogni momento.

Per testare cosa accadesse durante l'esecuzione più velocemente rispetto allo Step del Debugger mi sono anche avvalso della compilazione condizionale, stampando varie informazioni inerenti all'Accept e simili: tale questione può essere osservata compilando e definendo DEBUG o semplicemente de-commentando i due #define DEBUG presenti nel codice.

Il codice è stato testato sulla MV del corso (Xubuntu) e su una MV dell'ultima distribuzione di Ubuntu.

Difficoltà incontrate

Come ho già specificato ho creato e testato le operazioni una alla volta, aggiungendole man mano che venivano completate al codice finale. Una volta terminata la prima scrittura del codice ho provato a testare con i test del progetto vero e proprio e, con un po' di lavoro, sono riuscito a passare il primo (testconf.sh). Il secondo ed il terzo hanno richiesto un po' più d'attenzione in termini funzionali per casi che non avevo gestito come il client si aspettava, o magari per statistiche errate in determinate situazioni, ma anche lì dopo qualche giorno sono riuscito a passarli. Il test 5 (teststress.sh) invece non ha creato troppi problemi in quanto ogni operazione era fondamentalmente circondata da Locks e la peculiarità di questo test era solo il tentativo di "stressare il progetto" con molte operazioni. Per quanto riguarda il test4 (testleaks.sh) invece ho avuto diverse difficoltà perché i miei stessi test iniziali non hanno tenuto conto dei possibili leaks, trovandomi quindi con una buona mole di errori tutti insieme: ovviamente Valgrind ha dato il suo contributo specificando dove fossero questi leaks, ma devo ammettere che sia stato il test che mi ha dato più filo da torcere. Questo dimostra quanto la completezza di un test sia necessaria a tempo di organizzazione del test stesso, piuttosto che a cercare di rimediare agli errori commessi.

Giulio Purgatorio, 516292, corso A.