Relazione Progetto di Sistemi Operativi

2017/2018

Monica Amico , Corso B

N. matricola: 516801

1. Introduzione

Il progetto chatterbox ha come scopo lo sviluppo di un server concorrente che implementa una chat e gestisce le operazioni richieste dagli utenti.

2. Struttura del server e flusso di lavoro

Il server è organizzato in 3 componenti principali, per un totale di ThreadsInPool threads. Al suo avvio il server, nel main, inizializza le principali strutture dati e avvia le sue 3 componenti.

Le componenti sono:

- thread Listener: questo thread svolge un doppio lavoro, infatti è sia listener per le nuove connessioni in ingresso che dispatcher per le richieste provenienti da client già connessi. Come listener questo thread accetta nuove connessioni, qualora sia possibile, registrando il file descriptor della connessione in una maschera utilizzata nella fase di dispatching. In quest'ultima fase il thread sta in attesa, per un certo periodo di tempo sulla funzione select e si sblocca in 3 situazioni differenti, escluse situazioni anomale:

    1.1. Nuova connessione, ovvero quando la socket del server è pronta per una accept(). In questo caso il FD viene inserito nella maschera della select;

    1.2. Nuova richiesta, quando un client il cui FD è registrato nella maschera invia una nuova richiesta: il FD viene inserito in una coda FIFO (descritta nella sezione successiva);

    1.3. Chiusura di una connessione: il FD viene inserito in una coda FIFO.

- threads Worker: questi threads si occupano di estrarre i FD che hanno inviato una richiesta dalla coda FIFO riempita dal listener ed effettuano l'operazione richiesta. Se l'esito dell'operazione richiesta è positivo, il FD viene re-inserito nella maschera della select altrimenti viene chiuso e il client associato disconnesso.
- thread Handler dei segnali: questo thread sta in attesa della ricezione di uno dei segnali previsti. In caso di segnale di terminazione i threads listener e worker

vengono avvisati tramite una variabile booleana e terminano la loro esecuzione ma non prima di concludere un eventuale operazione in corso.

3. Strutture dati

Per la gestione degli utenti e delle varie operazioni da effettuare sono state utilizzate le seguenti strutture.

3.1 Strutture per la gestione delle connessioni:

Con il termine gestione delle connessioni ci si riferisce a 3 situazioni differenti:

- Gestione delle connessioni attive: quando un client si connette al server, se non è stato raggiunto il numero massimo di connessioni, il file descriptor del client viene inserito in una lista di attesa. Tale lista, gestita con politica FIFO, è dichiarata e definita nei file list.c,h, un puntatore alla testa di questa lista è memorizzato nella variabile ***fd_list_head***.
- Gestione della disconnessione: per mantenere una corrispondenza tra nome utente e file descriptor viene utilizzata una lista unidirezionale definita in listfduser.h. Quando il server realizza che un client precedentemente connesso non è più attivo utilizza questa lista per estrarre il nome dell'utente a partire dal FD e disconnetterlo, un puntatore alla testa di questa lista è memorizzato nella variabile ***fd_user_head***.
- Gestione della mutua esclusione sulle socket: per gestire la concorrenza nelle scritture e letture sulle socket dei vari client connessi viene usata la struttura dati definita in circular.h, il puntatore a questa struttura è memorizzato nella variabile ***fd_locks***. Questo aspetto viene approfondito nella sezione sulla gestione della concorrenza.

3.2 Strutture per la gestione degli utenti e dei gruppi:

- Gestione degli utenti: Le informazioni sugli utenti registrati alla chat sono contenute in una tabella hash, implementata nel file hash.h. Le collisioni all'interno della tabella hash vengono gestite con liste di trabocco unidirezionali. Il numero totale di liste per tale tabella è uguale a ***BUCKETS (200)***. Ogni elemento della tabella hash rappresenta un utente e contiene: il nome dell'utente, il file descriptor (-1 se è offline), un intero che indica se l'utente è online, un array di messaggi pendenti allocato dinamicamente di dimensione ***MaxHistMsgs*** e un puntatore all'eventuale prossimo utente nella lista di trabocco. Ogni utente verrà inserito nella lista di trabocco[i]. L'indice i viene calcolato con opportuna formula hash, che prende come parametro il nome dell'utente da inserire.

- Gestione dei gruppi: viene utilizzata una lista contenente elementi che rappresentano gruppi, implementata nel file group.h, i puntatori a questa struttura è memorizzato nelle variabili ***g_head e g_tail.*** Ogni elemento della lista contiene il nome del gruppo, nome del creatore del gruppo e un puntatore ad una tabella hash, che ha la stessa struttura generale della tabella che viene utilizzata per la gestione di tutti gli utenti ma di dimensioni ridotte, adatta a contenere solo i membri del gruppo. I messaggi pendenti degli utenti dei gruppi verranno comunque gestiti all'interno della tabella hash "generale" di tutti gli utenti.

4. Gestione concorrenza

La sincronizzazione tra thread listener e worker viene effettuata secondo il paradigma produttore-consumatore in cui il listener agisce da produttore inserendo i FDs degli utenti con nuove richieste nella lista fd_list_head mentre il worker consuma i FDs o sta in attesa se la coda è vuota. La sincronizzazione tra questi due thread ed il thread handler avviene tramite una variabile booleana: alla ricezione di un segnale di terminazione l'handler sveglia eventuali threads in attesa su variabili di condizione ed imposta un valore boolean (stop) che indica ai threads di dover terminare.

La concorrenza per l'accesso agli elementi tabelle hash prevede l'utilizzo di una variabile di mutua esclusione ogni K liste di trabocco dove K è un decimo del numero di liste. Per realizzare la mutua esclusione sui FDs degli utenti connessi si è scelto di utilizzare una mutex per ogni FD per garantire un alto grado di parallelismo, nell'assunzione che MaxConnections non sia un valore eccessivamente grande. Un alternativa a questa soluzione, che effettivamente potrebbe portare ad un overhead di risorse, potrebbe essere quella di utilizzare lo stesso meccanismo usato per la tabella hash.

Insiemi di variabili globali utilizzate all'interno della stessa sezione critica condividono la stessa mutex, un esempio sono le variabili utilizzate per la select: un intero ed una maschera che condividono una mutex.

Per prevenire situazioni di deadlock si è scelto di utilizzare un'acquisizione ordinata delle risorse, quando necessario.

5. Note sulla suddivisione dei files

- chatty.c : contiene il main e il codice dei threads;
- list.c,h : definizione ed implementazione della lista dei FDs con nuove richieste, utilizzata per il prod-cons tra listener e worker;
- hash.c,h: definizione ed implementazione della tabella hash per la gestione degli utenti;
- group.c,h: definizione ed implementazione delle strutture per la gestione dei gruppi;
- config.h: macro e tipi ausiliari (bool_t ecc…);
- circular.c,h: definizione ed implementazione della struttura per l associazione fd-mutex;
- listuserfd.c,h: definizione ed implementazione della struttura per l'associazione fd-nome utente;
- parsing.c,h: definizione ed implementazione degli algoritmi per la lettura dei file di configurazione.

Nei files presenti nel kit sono state implementate le funzioni richieste. In connections.c sono presenti dei "wrappers" per eseguire le funzioni di comunicazione in mutua esclusione mentre in message.h sono state aggiunte due brevi funzioni per clonare un messaggio, si sono rese necessarie per correggere errori di "aliasing" tra puntatori che portavano a doppie free o ad accessi in zone di memoria non allocate.

6. Note sullo sviluppo del codice

Il progetto è stato sviluppato su macchina virtuale con ubuntu 16 su macBook air.

Per la stesura e il debug del codice è stato utilizzato VisualStudio Code.

Per il debug di errori riguardanti la memoria è stato utilizzato valgrind con il tool helgrind per il controllo di eventuali race condition.