Sistemi Operativi e Laboratorio

Relazione finale

*di Priscilla Raucci, matricola ******, corso **

Chatty è un server concorrente che implementa una chat attraverso cui gli utenti possono scambiarsi messaggi testuali e/o files collegandosi attraverso un programma client.
Il server si basa sull'utilizzo di un socket di tecnologia AF_UNIX, e tutto il codice è stato implementato e testato su Ubuntu17.10 .
Di seguito si discutono le principali scelte implementative.

1. Organizzazione del Codice

Il codice del server è suddiviso in più files, i principali sono i seguenti:

- chatty.c: è il file principale, contiene il main del server, le invocazioni per la messa in piedi di tutte le strutture dati necessarie, la funzione di ricezione e invio delle richieste ai thread(request_handler) e le funzioni per gestiscono il comportamento dei thread della threadpool.
- myicl_hash: il file che contiene il codice per la costruzione e gestione e l'utilizzo della hashtable in cui son raccolte tutte le informazioni degli utenti registrati in chatty. La hashtable ha anche la responsabilità di gestire la mutua esclusione per quanto riguarda la scrittura in direzione server-clients. Il codice della tabella hash è stato riadattato dal codice messo a disposizione dai docenti durante il corso
- groups: ile che contiene il codice per la costruzione e gestione delle strutture dati dei gruppi
- queue : file che contiene il codice per la costruzione e gestione della coda all'interno della threadpool che si occupa di memorizzare le richieste dei client arrivate al thread master per essere poi prelevati dai thread che le gestiranno.
- utils : file che contiene codice per la lettura e scrittura sicura su socket.
- parser : file che contiene la funzione di parser che si occupa di leggere i valori all'interno del file di configurazione e inserirli in una apposita struttura dati per il loro utilizzo.

2. Funzionamento del Server

Alla base del funzionamento del server sta un threadpool con modello master-slave. Il thread master è il main del file chatty.c, che ha il compito di settare il socket e rimanere in ascolto, per mezzo della funzione select, su tutti i file descriptor. Quando un file descriptor è pronto, il master, invoca la funzione request_handler all'interno della quale riceve sulla connessione tutti gli elementi della richiesta (con tipo message_t) da parte del client, inserisce i dati in una opportuna strutture dati e la mette a disposizione dei thread slave.

Il task dei thread slave prevede che essi, in ciclo continuo, attendano le singole richieste dal thread master(ogni thread gestisce una singola richiesta, non tutta la connes-

sione di un client) e la gestiscano, rispondendo al client in modo opportuno. I thread slave sono quelli che si occupano anche di inviare una eventuale risposta al client ricevente o a ulteriori client. Il thread master, nel momento in cui mette a disposizione dei thread slave una data richiesta, non ne monitora più il successo o meno.
Si noti che nella gestione delle richieste da parte dei thread slave la struttura dati myicl_hash.c svolge un compito di primaria importanza, anche in quanto si occupa direttamente della scrittura sul socket e gestisce la mutua esclusione. Nel paragrafo 4 verrà approfondito questo punto.

Le richieste dei singoli client vengono messe a disposizione di thread slave dal thread master mediante l'utilizzo di una coda circolare concorrente(queue.c). Ogni elemento inserito all'interno della coda ha tipo request, una struttura dati appositamente creata all'interno della quale troviamo l'indirizzo di connessione del client richiedente e la richiesta(tipo message_t) fatta al server.

3. Strutture dati e mutua esclusione

Per la gestione dei dati dei client è stata utilizzata una unica grande tabella hash(anagrafica) per le cui chiavi sono state utilizzati i nickname degli utenti(che da consegna devono essere univoci). All'interno della hashtable troviamo delle strutture dati che contengono le informazione degli utenti registrati: il nickname, un bit di connessione(settato a -1 in caso di utente non connesso), l'indirizzo di connessione(che viene settato ogni volta che l'utente si riconnette al server), un puntatore alla lista di messaggi pendenti e una coppia mutex e una variabile di connessione per la scrittura sulla connessione del dato utente.
La scelta della tabella hash come struttura di base per la gestione dei dati è stata obbligata per soddisfare le esigenze di rapidità espresse nella consegna, tuttavia un punto critico nella gestione della tabella è nel caso della deconnessione degli utenti. Quando il thread master del server legge la chiusura di connessione, ne conosce solamente l'indirizzo e non il nickname(dunque la funzione hash è inutilizzabile). L'unica soluzione è quella di scandire tutti gli utenti connessi, questa soluzione, pur essendo costosa non è stata ritenuta sufficiente per giustificare l'utilizzo di una ulteriore struttura dati(dizionario o tabella hash).

L'accesso alla tabella hash è sempre fatto in mutua esclusione. Come politica si è scelto di utilizzare una unica mutex per le modifiche all'intera tabella, riducendo il più possibile la regione critica da essa controllata e che venga utilizzata solo per il controllo dell'accesso dei dati della tabella. Per ogni utente registrato viene allocata una mutex, che verrà utilizzata non solo per l'accesso ai dati dell'utente particolare, ma che servirà anche per regolare la scrittura su quella data connessione(per evitare che due thread differenti scrivano contemporaneamente ad uno stesso utente).

Le alternative a questa politica poteva essere l'utilizzo di una mutex per ogni bucket della tabella hash, questo però comportava un duplice problema:

1. lo spreco di spazio, Idealmente una tabella hash non dovrebbe mai essere completamente satura, quindi avremmo dovuto allocare mutex che idealmente non sarebbero mai state utilizzate.
2. l'overhead per la gestione(lock e unlock continui) di molteplici mutex per la gestione della tabella hash.

Considerando questi due fattori è stato scelto di utilizzare un unica mutex, coscienti del fatto che l'utilizzo di una singola comporta comunque un overhead.
Questa scelta è stata anche influenzata dalla volontà di inglobare tutta la gestione della mutua esclusione(di tutta la tabella e anche dei singoli utenti) all'interno della struttura dati *icl_hash_t* (in modo che i thread del threadpool non le dovessero gestire) e che le mutex fossero allocate a prescindere dal numero di buckets richiesti al momento della creazione della struttura dati.

Si noti che all'interno della struttura della tabella vengono utilizzate altre due chiavi , per la modifica dei valori del numero dei registrati(*nentries*) e del numero dei connessi(*nconnected*), lock che vengono sempre prese solamente quando un thread è già in possesso del lock su tutta la tabella.
Anche la lock sulle mutex dei singoli utenti viene presa solo nel momento in cui il thread ha già in mano la lock su tutta la struttura, subito dopo aver preso il mutex sul singolo utente la mutex su tutta la struttura viene rilasciato.
Questa convenzione è stata stabilita fin dall'origine per evitare problemi di deadlock.

Si sottolinea che myicl_hash.c/myicl_hash.h ha una responsabilità che va ben oltre la gestione dei dati degli utenti, ma che gestisce oltre che la mutua esclusione, tutta la scrittura dei dati in risposta dal server ai singoli client.
Al codice da cui prende spunto sono state aggiunte numerose funzioni ad hoc per l'utilizzo in questo particolare utilizzo della tabella hash.

3. Gruppi

Per l'implementazione e la gestione dei gruppi è stata costruita una struttura dati(grouplist_t), che sostanzialmente è una lista (linked list) di tutti i gruppi. Ogni gruppo(con tipo group_t) al suo interno memorizza il ome del gruppo, il numero dei componenti e una lista dei nickname dei componenti del gruppo.
All'interno di un gruppo basta memorizzare questi dati in quanto il nickname è anche la chiave della hash table, una volta inviata la richiesta di unviare un dato messaggio a tutti gli utenti di un gruppo il server si occuperà di inviare il dato messaggio a tutti i componenti, scandendo la lista di utenti della struct con il nome del nome del gruppo corrispondente e servendosi delle funzioni implementate in myicl_hash.c per l'invio vero e proprio del messaggio ai singoli utenti.

La scelta du una linked list comporta un overhead all'interno delle operazioni di ricerca, operazione che viene effettuata per l'invio dei messaggi. Alternativa a questa struttura poteva essere l'utilizzo di una ulteriore tabella hash, tuttavia sembrava eccessivo l'utilizzo di una struttura così complessa, considerando che il numero dei gruppi sia in genere drasticamente minore di quello degli utenti. L'utilizzo della tabella hash non avrebbe inoltre risolto il problema di ricerca di un utente all'interno di un gruppo, quindi in caso di gruppi molto numerosi sarebbe sorto un problema analogo.

Tutta le gestione della struttura è in mutua esclusione, con una politica analoga a quella della hashtable. la struttura che raccoglie tutti i gruppi ha una unica mutex, tuttavia non sembrava opportuno dotare ogni gruppo di una mutex particolare, sempre nella visione che il numero di gruppi è nettamente inferiore a quello degli utenti e che le operazioni sui gruppi sono limitate rispetto a quelle di invio di messaggi a utenti singoli.

4. Gestione dei segnali e chiusura

Il server, in apertura (nel momento di instanziare tutte le strutture dati) maschera tutti i segnali, in un secondo momento applica una maschera, sensibile a SIGINT SIGTERM e SIGQUIT. La ragione principale di questa scelta è dovuta al fatto nella prima fase di costruzione delle strutture dati il server potrebbe essere interrotto in uno stato non ancora consistente(creando potenziali problemi alla funzione di cleanup), mentre in seguito i tre segnali di chiusura vengono abilitati contemporaneamente poichè gestiscono la chiusura del server allo stesso modo(utilizzano tutti e tre la stessa funzione). In questa fase non viene ancora abilitato il segnale SIGUSR1 per la stampa delle statistiche.

Un punto critico nella gestione dei segnali è la funzine select; prima della select tutti i segnali vengono nuovamente mascherati come descritto prima(la maschera permette la ricezione di SIGINT SIGTERM e SIGQUIT), mentre subito dopo viene abilitata la ricezione del segnale SIGUSR1 per la stampa delle statistiche.
Nella fase di attesa della select dunque il sever è insensibile al segnale per la stampa delle statistiche, questa fase di attesa è legata esclusivamente alla prontezza dei client a collegarsi, e quindi non è predicibile la durata. Per ovviare a questo problema all'interno della select è presente un timeout che, anche in caso in cui non ci fossero notifiche sulla connessione, permette di uscire dalla select, e testare il segnale SIGUSR1.

La gestione dei segnali è reasponsabilità del thread master, all'interno della funzione handle_signal gestisce la chiusura(invocando anche la funzione della cleanup) o la stampa delle statistiche

Al momento della ricezione di uno dei segnali di chiusura viene fatta una doppia operazione, con la funzione di killQueue, vengono eliminati tutti gli elementi della coda che manda le richieste dal thread master ai thread slave e attraverso un parametro viene impedito l'accodamento di ulteriori richieste. I thread slave quando testeranno all'interno del ciclo(in particolare all'interno della funzione pop della coda) usciranno.