Relazione chatterbox

Progetto del modulo di laboratorio di Sistemi Operativi

Lorenzo Gazzella
546890 - Corso A

2017/2018

Indice

1 Principali scelte di progetto **2**
1.1 Strutture dati principali utilizzate . . . 2
1.1.1 icl_hash_s (fornita durante il corso) . . . 2
1.1.2 hash_s . . . 2
1.1.3 coda_circolare_s e coda_circolare_iteratore_s . . . 2
1.1.4 utente_connesso_s . . . 2
1.1.5 array_s . . . 3
1.1.6 Queue_t (fornita durante il corso) . . . 3
1.1.7 config . . . 3
1.1.8 connessi_s . . . 3
1.1.9 stat_s . . . 3
1.2 Scelte sulle size delle strutture . . . 3
1.3 Gestione della memoria . . . 3
1.4 Accesso ai file . . . 4

2 Struttura del codice **4**
2.1 Logica della divisione su più file . . . 4
2.1.1 chatty.h e chatty.c . . . 4
2.1.2 gestioneRichieste.h e gestioneRichieste.c . . . 4
2.1.3 parser.h e parser.c . . . 5
2.1.4 utility.h . . . 5
2.1.5 struttureCondivise.h . . . 5
2.1.6 script.sh . . . 5
2.2 Librerie utilizzate . . . 5

3 Struttura del programma **5**
3.1 Funzionamento generale del server . . . 6
3.2 Interazione tra thread e i relativi protocolli di comunicazione . . . 6
3.3 Gestione della terminazione . . . 7

4 Struttura dei programmi di test **8**

5 Difficoltà incontrate e soluzione adottate **8**

6 Note sulla compilazione **10**

1 Principali scelte di progetto

1.1 Strutture dati principali utilizzate

Di seguito viene riportata la lista di tutte le strutture dati utilizzate con spiegazioni e commenti.

1.1.1 icl_hash_s (fornita durante il corso)

Rappresenta una tabella hash con lo scopo di memorizzare tutti gli utenti registrati nel server. Gli utenti registrati sono coloro che hanno inviato una richiesta di `REGISTER_OP`.

1. `Key`: Come chiave della tabella hash è stato usato il nickname del client registrato.
2. `Value`: Come valore della tabella hash è stato usato un coda_circolare_s.

1.1.2 hash_s

Struttura dati utilizzata per associare ad una tabella hash, un insieme di lock. Ogni lock si occuperà di gestire la mutua esclusione di un determinato sottoinsieme della tabella hash. Questo perché è una delle strutture dati più utilizzate e avere soltanto una lock per l'intera hash avrebbe rallentato l'esecuzione del server.

1.1.3 coda_circolare_s e coda_circolare_iteratore_s

Rappresenta un array circolare utilizzato per memorizzare la history di un utente registrato. Viene fornito anche l'iteratore sugli elementi dell'array circolare.

1.1.4 utente_connesso_s

Rappresenta un utente connesso:

1. `nickname`: Nickname dell'utente.
2. `fd`: File descriptor con la quale si può comunicare con l'utente.
3. `fd_m`: Lock relativa all'`fd`. Ogni volta che si usa l'`fd` bisogna prima prendere la lock per non incorrere in stati inconsistenti.

Un utente connesso è un client che ha inviato una richiesta di `CONNECT_OP`.

1.1.5 array_s

Struttura dati utilizzata per associare una lock ad un array. Nel progetto è stata usata per memorizzare un insieme di utenti connessi (utente_connesso_s). Ogni volta che si accedeva all'insieme si doveva prendere la lock.

1.1.6 Queue_t (fornita durante il corso)

Rappresenta una coda in mutua esclusione. Nel progetto è stata usata per inserire le richieste inviate dai client ma non ancora gestite.

1.1.7 config

Struttura dati utilizzata per salvare tutte le configurazioni del server. Viene inizializzata all'inizio facendo il parsing del file di configurazione.

1.1.8 connessi_s

Struttura dati utilizzata per associare una lock ad un contatore. Nel progetto è stata usata per tenere traccia del numero di socket attive.

1.1.9 stat_s

Struttura dati utilizzata per associare una lock alla struttura dati `struct statistics` fornita nel kit del progetto.

1.2 Scelte sulle size delle strutture

1. Per quanto riguarda la tabella hash utilizzata per memorizzare gli utenti registrati, è stato deciso di settare la sua grandezza attraverso una macro (, file config.h) per essere più scalabile e modulare. E' stata inizializzata con un valore pari a 1500. Questo perché il server è stato progettato per contenere decine di migliaia di utenti registrati. Inoltre è stata inserita la macro `HASHGROUPSIZE` per la gestione delle lock relative alla tabella hash. In particolare `HASHSIZE\HASHGROUPSIZE` rappresenta il numero di lock predisposte alla gestione della mutua esclusione.

2. Per quanto riguarda l'array degli utenti connessi, è stato deciso di inizializzarlo a un valore pari a `MaxConnections` poiché rappresenta il limite massimo di socket attive.

1.3 Gestione della memoria

Dato che il server è stato pensato per gestire decine di migliaia di utenti e per restare in esecuzione per un periodo di tempo relativamente lungo, è stata posta particolare attenzione all'utilizzo della memoria dinamica. Per la gestione della memoria è stato deciso di utilizzare le funzioni di cleanup registrate tramite la funzione `atexit()` dopo ogni operazione di allocazione. E' stato utilizzato

questo approccio in modo tale da non avere alcun memory leak al termine del programma.

1.4 Accesso ai file

Per la gestione dell'accesso ai file (sia in lettura che in scrittura) sono state utilizzate due funzioni fornite durante il corso:

1. `readn()`: Funzione che si occupa della lettura di un file e la memorizzazione in un buffer
2. `writen()`: Funzione che si occupa della scrittura del contenuti di un buffer in un file

2 Struttura del codice

2.1 Logica della divisione su più file

E' stato deciso di suddividere il progetto in più cartelle per una migliore modularità, mantenimento e chiarezza.

1. `libs`: Contiene tutte le librerie utilizzate dal server
2. `tests`: Contiene tutti i test che sono stati fatti per verificare la correttezza di alcuni sottoinsiemi del server.
3. `DATA`: Fornita nel kit del progetto
4. `docs`: Cartella di destinazione del doxygen

I file che riguardano la gestione diretta del server sono stati inseriti nella directory principale. Di seguito viene riportata una lista con una breve descrizione.

2.1.1 chatty.h e chatty.c

Contengono rispettivamente la dichiarazione e il corpo dei metodi utilizzate per la gestione del server. In particolare contengono tutte le funzioni di inizializzazione del server, le funzioni di cleanup e le funzioni eseguite dai thread:

1. `pool()`: Funzione eseguita dai thread appartenenti al pool.
2. `listener()`: Funzione eseguita dal thread listener.
3. `segnali()`: Funzione eseguita dal thread usato per la gestione dei segnali.

2.1.2 gestioneRichieste.h e gestioneRichieste.c

Contengono rispettivamente la dichiarazione e il corpo dei metodi utilizzati per la gestione delle singole richieste che il server deve gestire. Ogni richiesta corrisponde a una funzione.

2.1.3 parser.h e parser.c

Contengono rispettivamente la dichiarazione e il corpo dei metodi utilizzati per il parsing di un file di configurazione.

2.1.4 utility.h

Contiene alcune funzioni di utilità per l'accesso ai file e per il controllo del valore di ritorno di una esecuzione di funzione.

2.1.5 struttureCondivise.h

Contiene tutte le strutture dati utilizzate dal server e alcune funzioni che permettono di prendere le relative lock più facilmente, rendendo il codice più pulito e comprensibile.

2.1.6 script.sh

File contenente lo script richiesto nel testo del progetto.

2.2 Librerie utilizzate

Come detto precedentemente tutte le librerie utilizzate sono state inserite nella cartella `lib`. Di seguito viene riportata una breve descrizione

1. **GestioneHashTable**: Directory contenente il `.c` e il `.h` della libreria fornita durante le lezioni che implementa una tabella hash.
2. **GestioneQueue**: Directory contenente il `.c` e il `.h` della libreria fornita durante le lezioni che implementa una coda in mutua esclusione.
3. **GestioneHistory**: Directory contenente il `.c` e il `.h` della libreria scritta dall'autore del progetto che implementa una coda circolare usata per salvare la history degli utenti registrati.

3 Struttura del programma

Come da specifiche, l'architettura del server è stata pensata in modo da gestire decine di migliaia di utenti e si è adottato quindi per avere una programma multithread:

1. **Thread Listener**: Singolo thread utilizzato per la gestione delle connessioni con i client, per ricevere una serie di richieste da parte di essi e per la comunicazione con ogni thread del pool. Il funzionamento principale consiste nel registrare una serie di file descriptor su cui si resta in attesa. Quando uno di questi risulta pronto, si gestisce e ci si rimette in attesa. I file descriptor registrati possono essere di tre tipi:

(a) **Pipe**: Ci saranno tante pipe quanti sono i thread appartenenti al pool. Saranno usate per la comunicazione con essi.

(b) **Socket server**: Utilizzata per restare in ascolto di connessione da parte dei client.

(c) **Scoket client**: Ci saranno tanti fd quanti sono i client connessi. Verranno utilizzati per ricevere le richieste da parte di essi.

Ogni volta che il thread listener riceve una richiesta da parte di un client, toglie il relativo fd dall'insieme dei file descriptor in cui è in ascolto e lo aggiunge in una coda condivisa con i thread appartenenti al pool.

2. **Thread Pool**: Insieme di thread. Ogni thread preleva dalla coda un fd (relativo a un particolare client), legge la richiesta, la gestisce e risponde con un messaggio. Infine, attraverso la pipe con il listener, gli si comunica di riaggiungere il file descriptor del client all'insieme degli fd su cui deve rimanere in ascolto.

3. **Thread Segnali**: Singolo thread utilizzato per la gestione di tutti i segnali inviati al server. Il thread listener e i thread del pool ignorano tutti i segnali. Il thread segnali rimane in ascolto su:

   (a) `SIGUSR1`

   (b) `SIGINT`

   (c) `SIGTERM`

   (d) `SIGQUIT`

   La gestione di questi segnali segue le specifiche date.

3.1 Funzionamento generale del server

Il server inizia subito facendo il parsing del file di configurazione per poi inizializzare tutte le strutture dati utilizzate durante l'esecuzione. Inizia ora un ciclo che si interrompe solo quando arriva un segnale di tipo `SIGINT`, `SIGTERM` o `SIGQUIT` oppure quando si verifica un errore. Le azioni svolte nel ciclo sono descritte nel prossimo paragrafo. Infine, prima di terminare il programma, si distruggono tutte le strutture dati che si stanno utilizzando e si libera tutta la memoria.

3.2 Interazione tra thread e i relativi protocolli di comunicazione

Il programma principale si può suddividere in altri due grandi programmi:

1. **Esecuzione normale** (Figura 1): Come riportato nel sequence diagram, l'esecuzione normale del server prevede l'attesa da parte del thread listener di una richiesta da parte di un client. Il thread listener aggiunge l'fd del

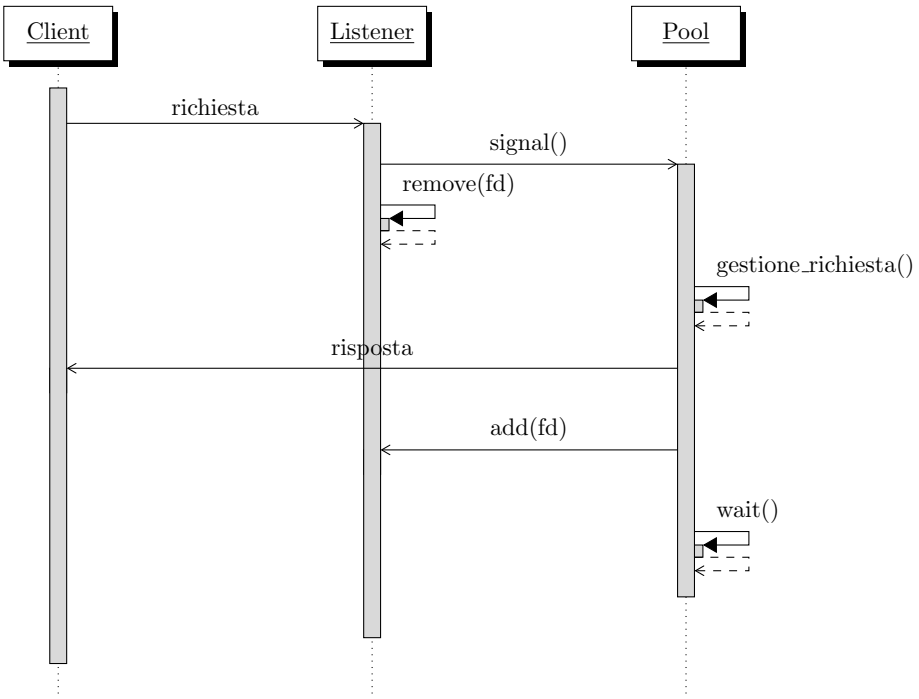

Figura 1: Sequence diagram esecuzione normale

client nella coda condivisa, lo elimina dall'insieme dei file descriptor sui cui deve restare in ascolto e risveglia un thread del pool in stato di wait. Questo passa alla gestione della richiesta, all'invio di un messaggio di risposta al client e infine comunica al listener che l'fd del client appena gestito deve essere reinserito nell'insieme.

2. **Arrivo di un segnale**: Il thread dei segnali ha due comportamenti differenti in base al segnale che arriva:
   (a) **SIGUSR1**: Aggiornamento del file delle statistiche.
   (b) **SIGINT, SIGTERM, SIGQUIT**: Viene liberata tutta la memoria in uso dal server e viene interrotta l'esecuzione del programma come descritto nel prossimo paragrafo.

3.3 Gestione della terminazione

All'arrivo di SIGINT, SIGQUIT o SIGTERM, il thread che gestisce i segnali deve far si che il server interrompa l'esecuzione. Innanzitutto vengono interrotti i thread:

1. **Listener**: Il thread listener viene interrotto tramite una chiamata `pthread_cancel()`. Per far si che non ci sia nulla di allocato alla terminazione di questo thread,

è stata utilizzata la funzione `pthread_setcancelstate()` per abilitare e disabilitare la possibilità di terminazione.

2. **Pool**: Per far terminare i thread appartenenti al pool è stato deciso di inserire tanti messaggi nella coda delle richieste pari al numero di thread nel pool. I messaggi inseriti hanno un valore speciale (`-1`) che il pool riconosce e termina l'esecuzione.

3. **Segnali**: Poiché è il thread che riceve il segnale di terminazione del server, è bastato chiamare la funzione `pthread_exit()`.

Una volta terminati tutti i thread, viene risvegliato il main che era in attesa sulla `pthread_join()`, quindi chiude la socket e ritorna 0. Dato che all'inizializzazione del server sono state registrate alcune funzioni di cleanup, all'uscita della funzione main, tutta la memoria verrà liberata in modo tale da non avere memory leak al termine dell'esecuzione.

4 Struttura dei programmi di test

I programmi di test sono tutti contenuti all'interno della cartella **tests**. E' stato deciso di fare alcuni test per controllare il corretto funzionamento di alcuni sottoinsiemi dell'intero programma. In particolare è stata posta particolare attenzione alle librerie sviluppate in modo da rendere lo sviluppo del server più semplice e senza doversi preoccupare in seguito di memory leak non gestiti correttamente dalle librerie. Nella cartella **tests** sono contenuti i seguenti test:

1. **Connections**: Test per il file `connection.c`. Per controllare il corretto funzionamento è stata fatta una serie di chiamate alla libreria e testate con valgrind, sia per vedere che l'esecuzione fosse quella che ci si aspettasse, sia per testare che non ci fossero memory leak.

2. **History**: Test per la libreria che implementa una coda circolare. Anch'esso è stato sviluppato in modo da eseguire diverse chiamate alle funzioni fornite dalla libreria, per poi controllare la corretta esecuzione.

3. **Listener**: Semplice test che invia una serie di richieste al thread listener del server.

4. **Parser**: Semplice test per il file `parser.c`.

Per il debugging finale e intermedio è stato sempre utilizzato lo strumento **valgrind**. I test forniti con il kit del progetto sono stati eseguiti con successo su una macchina virtuale Ubuntu 18.04.1 LTS.

5 Difficoltà incontrate e soluzione adottate

Ponendo particolare attenzione all'affidabilità e correttezza che un server deve avere, senza tralasciare l'efficienza, una delle parti più difficili sia da progettare che da sviluppare è stata quella della gestione delle lock nel caso in cui un

utente si deregistrava. Il problema principale è che gli utenti si potevano scambiare messaggi e prima di farlo dovevano prendere la lock sull'fd del ricevente. Questo però avrebbe portato ad un problema se il ricevente avesse fatto una richiesta di `UNREGISTER_OP` che avrebbe comportato la cancellazione dell'utente dalla hash e dall'array degli utenti connessi. Il problema è che cancellando l'utente, il sender sarebbe restato in attesa su una lock eliminata che come da manuale (`pthread_mutex_destroy()`) il comportamento sarebbe indefinito. La soluzione adottata è stata quella di inizializzare l'array degli utenti connessi fin dall'inizio con un numero di elementi pari a `MaxConnections`. Ogni elemento aveva la lock inizializzata, il `nickname` a `NULL` e l'`fd` a `-1`. Ogni volta che un client faceva una richiesta di CONNECT_OP, si cercava una posizione nell'array degli utenti connessi con `nickname` uguale a `NULL` e `fd` uguale a `-1` e si valorizzavano i campi con gli opportuni valori. Inoltre si è implementata una funzione che dato un nickname o un fd, restituiva la relativa posizione dell'array dei connessi (`-1` se non era connesso). In questo modo, ogni volta che serviva prendere la lock sull'`fd`, si svolgeva sempre la seguente lista di azioni:

1. Si prende la lock sui connessi
2. Si chiama la funzione che restituisce la posizione `pos` di un certo `nickname` all'interno dell'array dei connessi.
3. Se la `pos` è diversa da `-1`
   (a) Si chiama la funzione per prendere la lock sull'fd in posizone `pos`
   (b) Una volta presa la lock si controlla che in pozione `pos` ci sia ancora l'utente su cui vogliamo fare l'operazione.
      i. Se in posizione `pos` c'è ancora quell'utente vuol dire che è ancora connesso e si può procedere con la gestione della richiesta.
      ii. Se in posizione `pos` non c'è più quell'utente vuol dire che è stato disconnesso mentre si era in attesa sulla lock, allora si ritorna errore.

Inizializzando in questo modo l'array degli utenti connessi, è stato facile implementare la funzione `disconnect_op()`, in quanto doveva soltanto settare il campo `nickname` dell'utente che si vuole disconnettere a `NULL` e il campo `fd` a `-1`.
Un altro aspetto su cui si è prestata particolare attenzione, come detto precedentemente, è stato quello di avere un server ragionevolmente affidabile e sicuro. In particolare è stato evitato il rischio di deadlock usando la tecnica delle lock ordinate. In tutto il progetto è stato seguito il seguente ordinamento:

1. Lock sulla hash table degli gli utenti registrati
2. Lock sull'array degli utenti connessi
3. Lock sull'fd di un utente connesso

6 Note sulla compilazione

E' stato deciso di non controllare se l'operazione di una richiesta fosse di tipo `DISCONNECT_OP` visto che non si conosceva il protocollo esatto dato che il client utilizzava la chiusura della socket come messaggio di disconnessione. Per questo motivo, quando si esegue il comando `make`, si ha un warning poiché uno switch non controlla tutte le possibili operazioni (`op_t`).