Relazione Progetto LSO

Chiara Caglieri - Mat 516555

A.A 2016/2017

Contents

1 Strutturazione del codice

Il codice del server che si occupa di gestire le richieste provenienti dai vari client è strutturato in tre parti principali, corrispondenti a tre singoli file:

- **chatty.c** : è il file contenente il codice relativo alla configurazione del server e all'inizializzazione delle strutture dati necessarie al funzionamento dello stesso.
- **tools.c** : è il file contenente le funzioni di utilità invocate dai vari thread del server. Tali funzioni gestiscono: la coda delle richieste alla quale accedono i worker per servire i client, la terminazione dovuta all'arrivo dei

segnali SIGQUIT, SIGTERM, SIGINT e SIGUSR1 con annessa scrittura delle statistiche, la liberazione della memoria allocata per le tabelle hash relative agli utenti e ai gruppi e la distruzione di un pool di mutex.

- **threads.c** : è il file principale del server che contiene il codice eseguito dai thread worker e dal thread dispatcher allo scopo di servire le richieste dei client.

Inoltre, tutte le dichiarazioni di variabile e i prototipi delle struct sono contenuti nell'header file **tools.h**.

2 Strutture dati e motivazione

2.1 Coda delle richieste

Allo scopo di realizzare il multiplexing dei workers rispetto alle richieste dei client si è presentata la necessità di implementare una coda da cui questi potessero prelevare le richieste in ordine FIFO.
Una volta riconosciuta la presenza di una nuova richiesta, il thread dispatcher si occupa di inserire il fd della connessione (attraverso la *insQueue()*)all'interno della coda e di risvegliare eventuali thread in attesa di un client da servire.
Allo stesso modo, i thread worker prelevano un file descriptor dalla testa della coda (attraverso la *extractFd()*), sospendendosi sulla variabile di condizione *queueEmpty* quando questa è vuota.
La coda è stata di conseguenza implementata come una lista bidirezionale contenente come elementi i file descriptor corrispondenti ai vari client richiedenti.

2.2 Memorizzazione degli utenti e dei gruppi

Per quanto riguarda la memorizzazione degli utenti è stata utilizzata una tabella hash con funzione di hashing basata su stringhe. L'hash value generato è infatti associato al nickname dell'utente che si vuole inserire. Il numero di buckets utilizzato per la creazione della tabella hash può essere impostato cambiando il valore della macro (file *tools.h*). Il valore impostato di default è 24, il quale si è rivelato essere un buon compromesso per far sì che la tabella hash sia bilanciata ma comunque non sparsa.
Al momento della registrazione, si è scelto di utilizzare il nickname utente come chiave. per la parte dati dell'oggetto da inserire nella tabella hash è stata invece definita una struct contenente:

- **Nickname** dell'utente
- **History** dei messaggi (ricevuti e non) destinati all'utente
- **Numero di messaggi** correntemente nella history dell'utente
- **History index**, usato come puntatore alla posizione della history in cui inserire un nuovo elemento

La History dei messaggi è stata implementata come un array, assumendo che il numero di messaggi memorizzabili sia esiguo (considerando anche che i messaggi dalla history vengono, come da specifica, eliminati solo quando questa è piena).

Lo stesso tipo di struttura dati è stata utilizzata per la memorizzazione dei gruppi di utenti. In particolare, la tabella hash relativa ai gruppi contiene struct con campi:

- **Groupname** relativo al gruppo registrato
- **Tabella hash dei membri**, contenente elementi che indicano il nickname di ogni singolo membro del gruppo in questione.

E' stato memorizzato soltanto il nickname relativo ai client all'interno della tabella hash, ma questa si presta ovviamente a memorizzare anche altri potenziali campi (ad esempio se si volesse distinguere fra utente amministratore o ordinario del gruppo).

2.3 Memorizzazione dei file descriptor

E' stata inoltre implementata una collezione di file descriptor associati ai client, allo scopo di poter facilmente rintracciare i client online e individuare il fd associato a un dato nickname.
L'implementazione di tale collezione è stata realizzata attraverso un array di dimensione pari al numero massimo delle connessioni concorrenti previsto dal server, sotto l'assunzione che tale numero si mantenga ragionevolmente contenuto. L'array in questione contiene oggetti con campi:

- **Nickname**
- **File descriptor**

dove inizialmente tutti i valori relativi al fd sono impostati a -1. Questo valore limite permette di poter denotare le posizioni libere nell'array, grazie al fatto che il file descriptor relativo ad una connessione verso un client non può mai essere un valore negativo. inoltre, al momento della connessione, il server va a cercare il nickname del client nell'array e, se lo trova, si limita semplicemente a cambiare il fd aggiornandolo al nuovo.

3 Struttura dei programmi

3.1 Thread dispatcher

Nell'ambito del programma è presente soltanto un thread dispatcher che in principio si occupa di inizializzare i set di file descriptor utilizzati dalla *select()*. Successivamente esegue un ciclo che ha termine soltanto quando viene intercettato un segnale di terminazione dal server. All'interno di questo ciclo, esegue le seguenti operazioni:

- **Re-inizializza il set di lavoro**: ciò è necessario al fine di azzerare le eventuali modifiche effettuate al set durante l'iterazione precedente (escluso il caso in cui un file descriptor viene chiuso).
- **Effettua la select()**, con la quale rileva attività sui file descriptor nel set
- **Analizza tali file descriptor**, distinguendo fra attività sul file descriptor relativo al socket, che identifica una nuova connessione, e attività su file descriptor relativi ai client, che denotano quindi nuove richieste su connessioni già stabilite in precedenza.

Nel momento in cui viene rilevata attività sul file descriptor relativo al socket viene effettuata la *accept()* e il nuovo file descriptor associato al client viene inserito nel set. Nel caso in cui l'accettazione della connessione faccia sì che il numero di connessioni concorrenti superi il limite prestabilito, quest'ultima non viene accettata e viene inviato un messaggio di errore al client interessato.
Se invece è stata rilevata attività su un fd relativo ad una connessione già stabilita, questo viene temporaneamente rimosso dal set allo scopo di permettere a uno dei worker di gestire la richiesta prima che ne venga inserita un'altra proveniente dallo stesso client in coda. Successivamente tale fd viene inserito nella coda da cui prelevano i thread worker.
Sarà poi il worker stesso ad occuparsi di re-inserire il fd del client servito all'interno del set.

3.2 Thread worker

Nell'ambito del server, vengono creati uno o più thread worker che si occupano di servire le richieste provenienti dai client. Tali thread eseguono il medesimo codice, che è, come nel caso precedente, racchiuso all'interno di un ciclo il quale si esaurisce alla ricezione di un segnale di terminazione. Uno schema delle operazioni eseguite da un worker è il seguente:

- **Estrazione di un fd dalla coda**. Tale operazione avviene all'interno della *extractFd()* e rappresenta anche il momento in cui il thread worker si mette in attesa della disponibilità di richieste.
- **Lettura della richiesta**. Una volta ottenuto il fd, il worker procede a leggere l'header della richiesta (l'eventuale body sarà letto successivamente, se sarà necessario). Attraverso tale lettura, il worker stabilisce se la connessione è stata chiusa da parte del client (e in tal caso torna ad eseguire la *extractFd()*), oppure se è possibile procedere e servire la richiesta.
- **Esecuzione dello switch() sull'operazione richiesta** allo scopo di individuare il codice da eseguire per servirla.
- **Reinserimento del fd nel set usato dalla select()**. Una volta gestita la richiesta, il worker esegue una *FD_SET* del file descriptor corrispondente al client appena servito all'interno del set.

4 Interazioni

Le interazioni che avvengono fra i vari client e il server seguono tutte lo stesso protocollo.
Innanzitutto il client invia una richiesta al server, la quale può essere composta dal solo header oppure corredata da un body.
Una volta ricevuta la richiesta, il server (in questo caso il thread worker) si occupa di verificare che quest'ultima sia legittima e che non violi alcun vincolo (ad esempio viene controllato che la lunghezza del nickname da registrare non superi un dato limite):

- Se la richiesta **non è legittima**, il server invia al client un messaggio di errore e termina la gestone della richiesta.
- Se la richiesta **è legittima**, il server invia al client il messaggio di Ok e prosegue la gestione della richiesta.

Nell'ambito della gestione della richiesta e dello scambio di informazioni tra client e server, entrambe le parti fanno uso delle stesse funzioni di trasmissione e ricezione presenti in *connections.c*. Tali funzioni fanno uso delle primitive *read()* e *write()* e sono state realizzate in modo da rilevare quando la connessione risulta chiusa da parte del client (lettura di 0 bytes), quando si verifica un errore (valore di ritorno di una read/write negativo), oppure quando è necessario leggere ulteriori bytes poichè con una sola read non si è riusciti a leggere l'intera porzione di dati.
In particolare, l'ultimo controllo è stato implementato attraverso l'uso di un puntatore che, prendendo come esempio quello della scrittura di un buffer, punta all'inizio della porzione dei dati ancora da inviare. La funzione quindi cicla fino a che tutti i byte previsti non sono stati inviati.

4.1 Gestione della concorrenza

Come descritto nei paragrafi precedenti, le strutture dati presenti nell'ambito del programma sono per la maggior parte condivise e accessibili da tutti i thread del server. Per ottenere e garantire la mutua esclusione nell'accesso a tali strutture dati sono state utlizzate le seguenti mutex:

- **setMtx**: mutex atta a proteggere l'accesso del set di lavoro della select(), sul quale agiscono sia il thread dispatcher, sia tutti i thread worker presenti.
- **queueMtx**: mutex atta a proteggere l'accesso alla coda concorrente in cui vengono inseriti i file descriptor delle connessioni dei vari client che hanno presentato una richiesta. Serve a garantire che la stessa richiesta non venga prelevata da più worker, o che non venga eliminata prima di essere presa in carico da un worker. Come nel caso precedente, hanno accesso alla coda sia il thread dispatcher, sia tutti i thread worker.

- **statsMtx**: mutex atta a proteggere l'accesso e la modifica dei parametri delle statistiche, ai quali hanno concorrentemente accesso il thread dispatcher e tutti i thread worker.
- **fdArrayMtx**: mutex atta a proteggere l'accesso all'array contenente i file descriptor delle connessioni concorrenti.
- **mutexPool**[]: un array di mutex volto a garantire l'accesso concorrente alle tabelle hash e la mutua esclusione sulle scritture verso i client. La funzione hash utilizzata garantisce una buona distribuzione di mutex rispetto ai nickname in gioco, rendendo accettabile l'uso di tale pool di mutex anzichè una mutex per ogni client.
  E' stato ritenuto necessario proteggere le scritture verso i client con delle mutex per evitare che due worker tentino di scrivere al medesimo client contemporaneamente causando l'arrivo di un messaggio non significativo. Un esempio di questo è il seguente:
  - Il client A richiede la ricezione dei messaggi nella history al worker 1.
  - Il client B richiede al worker 2 l'invio di un messaggio destinato al client A.
  - Il worker 2 rileva che il client A è online (sta ricevendo i messaggi nella history) e quindi procede con l'invio immediato, il quale si interpone però con l'invio dei messaggi della history effettuato dal worker 1.

Oltre alle mutex, è stata anche utilizzata una variabile di condizione per permettere l'attesa dei thread worker nel caso la coda delle richieste sia vuota: **queueEmpty**. Un thread viene risvegliato dalla wait su tale variabile di condizione sia quando la coda si riempie, sia quando arriva un segnale di terminazione (procedura descritta nel paragrafo seguente).

5 Gestione dei segnali

Per quanto riguarda la gestione dei segnali, sono state realizzate due funzioni gestore da invocare al momento della ricezione del segnale. L'installazione di queste funzioni gestore viene effettuata all'interno del main() del server (file *chatty.c*) nel seguente modo:

- Creazione di un oggetto di tipo **struct sigaction**
- Assegnamento della funzione gestore al campo *sa_handler* dell'oggetto creato
- Installazione del gestore relativamente a un dato segnale attraverso la funzione *sigaction()*.

5.1 Terminazione immediata

La funzione **termHandler()** si occupa di gestire la terminazione immediata del server e per fare ciò setta la variabile "trigger" *terminate* a 1 (è una variabile globale che viene settata a 0 al momento dell'avvio del server) ed esegue una broadcast sulla variabile di condizione **queueEmpty** per risvegliare eventuali thread in attesa di nuove richieste. Una volta risvegliati, i thread testeranno la variabile globale *terminate* e procederanno con la *pthread_exit()* .

5.2 Generazione delle statistiche

La funzione **usr1Handler()** gestisce invece la scrittura delle statistiche nel file designato e non causa difatti la terminazione di alcun thread. La scrittura nel file avviene attraverso la funzione *printStats()*.

6 Gestione della memoria

Durante lo sviluppo del progetto, si è cercato di limitare lo spreco di memoria tentando di allocare ogni volta solo e soltanto lo spazio necessario. Questo non è sempre stato possibile, poichè non sempre il numero di byte necessari è noto a priori. In questi casi si è adottata una strategia di tipo "upper bound", definendo cioè limiti superiori alle dimensioni di oggetti allocati che garantissero però la corretta esecuzione del programma. Tali parametri sono facilmente modificabili poichè definiti attraverso *#define*.
In ogni caso, sono state liberate tutte le locazioni di memoria allocate durante l'esecuzione del programma, il quale non presenta memory leak.

7 Esecuzione

Il programma è stato sviluppato e testato su Ubuntu Artful 17.10 e su XUbuntu, entrambi eseguiti su Macchina Virtuale. Il test relativo ai memory leaks è stato eseguito su sistema operativo XUbuntu e la versione di Valgrind utilizzata è la 3.10.

Oltre al controllo di leaks eseguito dal test4, è stato effettuato un controllo attraverso Valgrind sul resto dei test, il quale non ha riportato alcun errore. Il comando utilizzato per eseguire tali test è il seguente:

```
valgrind --leak-check=full make test*
```