Relazione sull' implementazione del server concorrente multithread MEMBOX.

Gabriele Barreca

Luca Murgia

Indice:

Introduzione

Il lavoro è stato svolto da Gabriele Barreca e Luca Murgia.

Il programma è stato strutturato in diverse librerie, per avere un livello di astrazione che fosse il più alto possibile e di conseguenza ottenere una lettura/comprensione del codice molto più semplice e selettiva.
Nel programma si trovano le seguenti librerie:

- Queue.h : si occupa della coda delle connessioni. E' strutturata come un array circolare con inserimenti e prelievi con metodo FIFO.

- Operation.h : qui sono presenti tutte le funzioni che operano sulla Repository

- Connection.h : è la libreria che si occupa di attivare la connessione (Lato Client) e di far comunicare Client e Server.

- Icl_hash.h : è la libreria che si occupa delle funzioni sulla tabella hash. Si è scelto di usare quella di default messa a disposizione dai docenti.

- Erch.h : una mini-libreria , per il controllo sugli errori.

- Message.h , Stats.h, ops.h : sono le librerie messe a disposizione dai docenti che contengono le strutture dati del programma.

1. Il Server Membox

1.1 Strutture Dati

Thread array

Un array di thread che, una volta in esecuzione, estrarranno dalla coda una connessione, qualora fosse presente, ed eseguiranno le operazioni richieste dal client.

Repository

La Repository è una Tabella hash implementata tramite un array di liste di trabocco.
Questa tabella è gestita dalle funzioni contenute nella libreria icl_hash.h. Ogni elemento presente nella Repository ha i seguenti campi:
- chiave del messaggio, univoca per ogni elemento. Necessaria per la ricerca dell'elemento stesso in caso di rimozione, per esempio.
- lunghezza del buffer del messaggio.
- buffer del messaggio stesso.

Lock array

Per evitare il caso in cui più thread volessero fare operazioni diverse o anche uguali sugli stessi elementi, abbiamo introdotto un array di Lock la cui dimensione è uguale al numero di liste di trabocco.
L'idea è che quando un thread effettua un'operazione sulla Repository viene bloccata la lista di trabocco corrispondente (in cui si trova l'elemento), in modo che altri thread debbano aspettare il loro turno per accedervi.
Ovviamente nell'eventualità di molti conflitti questo rallenterebbe il programma, in quanto si perderebbe la concorrenzialità dello stesso. Per ovviare a questo problema è stata creata una Repository con 10000 liste di trabocco (o semplicemente molto grande) per rendere più bassa possibile la probabilità che questo possa accadere.

Coda di connessioni

E' una struttura dati condivisa, alla quale possono accedere diverse funzioni. All'interno della struttura troviamo:

- un array circolare nel quale accodiamo i file descriptors delle connessioni;

- un puntatore all'indice della testa della coda, corrispondente all'elemento prossimo all'estrazione;

- un intero *len*, che rappresenta il numero di elementi attualmente inseriti in coda. Serve inoltre per il calcolo del modulo durante l'inserimento e l'estrazione.

Abbiamo deciso di implementare la coda come array circolare e non come una lista, perché la dimensione di quest'ultima è nota grazie al file di configurazione; l'accesso diretto agli elementi della struttura inoltre riduce il numero di operazioni necessarie all'inserimento di un nuovo elemento.

Struct Args

Struttura degli argomenti da passare alla funzione di ogni thread, contiene:
- un puntatore alla coda delle connessioni
- un puntatore alla repository
- un puntatore all' array di lock

1.2 Main Thread

Il Main Thread è il cuore del server. E' la parte principale, il suo compito più importante è quello di eseguire il lavoro di Dispatcher, ossia resta in ascolto per accettare le richieste di connessione mandategli dal Client fin tanto che non arriva un segnale di chiusura.

Prima di adempiere al suo lavoro abbiamo le seguenti fasi preliminari:
- inizializzazione di tutti i dati che caratterizzano il server e la Repository, prelevati dal file di configurazione.
- mascherati i segnali SIGQUIT, SIGUSR1 e SIGUSR2 , fatti ereditare a tutti i thread figli che verranno successivamente creati.
- creazione di coda, Repository, array dei lock, struttura degli argomenti, thread per la gestione dei segnali e, infine, i thread lavoratori.

Una volta finito il compito di Dispatcher, si aspetta che tutti i threads finiscano il loro lavoro per poi essere chiusi (compreso il thread che gestisce i segnali).
Infine viene ripulita la Repository e deallocata tutta la memoria in uso.

1.3 Dispatcher

E' trattato in un punto a parte nella relazione per motivi di chiarezza ma è il Main Thread ha fare da Dispatcher.
Una volta accettata la connessione, quest'ultima viene messa in coda per essere prelevata dai threads lavoratori.
Ogni qual volta arriva una connessione, prima di essere inserita in coda, viene controllato che essa non sia piena.

Erroneamente a quanto si possa pensare, per noi la coda non è piena quando ha raggiunto la massima dimensione passatagli durante la creazione (indicato dalla variabile MaxConnection) ma solo quando il numero di elementi presenti nella coda è MaxConnection meno il numero di threads attivi in quel momento.
Infatti, potendo accodare solo un certo numero di connessioni (Il massimo numero delle connessioni che posso avere meno il massimo numero dei thread che posso creare) e dato che a volte questo numero può essere zero, abbiamo deciso di sfruttare a nostro vantaggio la statistica che ci indica quanti Threads stanno lavorando in quel momento. Tutto questo per non creare code vuote e allo stesso tempo per non avere un numero di connessioni totali, fra accodate ed in uso, che sia maggiore del numero totale di connessioni che posso accodare.
Nel caso in cui la coda fosse piena, viene invece inviato un messaggio al client.

1.4 Worker

Il worker è la funzione che viene passata ad ogni thread presente nel threadpool.
Questa funzione è composta da due cicli while innestati:

- Il ciclo esterno rimuove dalla coda il socket descriptor della connessione e va avanti finché non arriva una richiesta di chiusura (tramite segnale e controllata con una variabile globale).

- Il ciclo interno, invece, continua a leggere dal buffer finché il client manda messaggi o finché non venga fatta una richiesta di chiusura (tramite segnale e controllata con una variabile globale).
I dati letti vengono salvati in un messaggio ed eseguite le operazioni ad esso associate. Una volta eseguita l'operazione, prima di passare alla successiva, viene creato un messaggio di riposta da tornare al Client tramite il canale di scrittura della socket.

Infine quando il buffer è stato svuotato, si considerano terminate tutte le richieste di quel client e la connessione viene chiusa così da liberare la memoria occupata e rendere il suo file descriptor nuovamente utilizzabile.

1.5 Signal Handler

Abbiamo deciso di far gestire i segnali ad un thread apposito anziché personalizzarne il loro sig_handler.
Questo per 2 motivi :
1) per una più semplice lettura
2) per far si che ogni qualvolta avvenga la cattura di un segnale, si possa eseguire del codice c senza limitazioni. Se avessimo usato un semplice signal_handler non avremmo potuto usare tutte le funzioni di libreria in quanto non (tutte) safe.

L'idea dietro al thread handler è semplice. Rimane in ascolto in attesa di un qualsiasi segnale, ricordo che sono stati mascherati quelli che mi interessa intercettare ed esegue le corrispettive funzioni.

In caso di :
- SIGUSR1: viene effettuata la stampa delle statistiche sul file apposito (opportunamente aperto e la cui destinazione è stata recuperata dal file di configurazione).

- SIGUSR2: viene effettuata la stampa delle statistiche. Inoltre vengono informati il dispatcher e il worker al quale viene comunicato che non devono più accettare connessioni. Il worker dovrà finire di lavorare con l'ultima connessione disponibile.
Infine viene mandato un segnale di broadcast a tutti i threads che erano rimasti in attesa aspettando che la coda si riempisse e chiuso i socket descriptor del server (per evitare che la funzione che accetta connessioni ne aspetti all'infinito una nuova).

- SIGQUIT: analogo a SIGUSR2. La differenza consiste nelle tempistiche di chiusura. Mentre il primo aspetta di finire eseguire tutti le operazioni dell'ultima connessione attiva, SIGQUIT notifica la chiusura appena il server ha eseguito l'ultima operazione ignorando così le successive.

2. Librerie delle funzioni

2.1 Connections

Open connection

La funzione open connection viene utilizzata dal client per aprire una connessione con il server.
In caso in cui la connessione non vada a buon fine verrà riprovata per un numero di volte arbitrario scelto dal Client. Fra un tentativo e l'altro

passeranno un numero x di secondi, anche questi scelti dal Client.

ReadHeader

La funzione ReadHeader legge dalla socket la parte che andrà a comporre l'header del nostro messaggio, ossia l'operazione e la chiave.
Non essendo le funzioni interne (read) atomiche, mi assicuro con un ciclo che venga scritto tutto in contenuto del messaggio. (Un ciclo per ogni parte dell'header).
In caso contrario torno un messaggio di errore per individuare la read fallita. Mentre nel worker di Membox.c stampo l'errore generato dalla funzione per capire il motivo del fallimento. Ogni volta che client non avrà più messaggi da inviare verrà stampato a schermo il messaggio : “Fallita la prima read dentro la ReadHeader” , che indicherà l'effettiva conclusione del lavoro del Client.

ReadData

Analoga a ReadHeader. Con la differenza che vengono lette le informazioni che comporranno il dato del messaggio, ossia la lunghezza e il buffer.

SendRequest

E' la funzione opposta alle ReadReply e ReadData. Quindi scrive anziché leggere, con lo stesso meccanismo sopra spiegato.

SendHeader

Come la SendRequest ma scrive sulla socket solo l'header del nostro messaggio, questo perché al Client il Dato di risposta interessa solo nel caso in cui ha fatto una richiesta di Get. Altrimenti interessa solo l'esito dell'operazione.

ReadReply

Vedi funzioni ReadHeader e ReadData.

2.2 Operation

In questa libreria abbiamo come variabili globali:
- Un flag che se uguale a 0 mi indica che la Repository è libera, occupata altrimenti. Con rispettiva mutex.
- La struttura delle Stats con la sua mutex.
- Variabili per l'inizializzazione del server.

Per implementare le operazione di Lock e Unlock abbiamo deciso di implementare un flag che ci indicasse lo stato della Repository.
A questo flag, settato a 0 di default, viene assegnato come valore il Thread_id del thread chiamante, in quanto è un valore univoco. Ogni volta che dovrà essere fatta un'operazione il thread che la dovrà eseguirà controllerà lo stato del flag. Se 0 o ha il flag uguale al suo Thread_id allora può lavorare sulla Repository. Altrimenti dovrà aspettare che essa venga “Unlockata” .

Funzione conf

Questa funzione serve a salvare in delle variabili globali tutte le caratteristiche del server e della Repository.
Viene fatto un ciclo in cui leggo tutto il file di configurazione. Se riconosco che è un commento allora ignoro la riga e vado avanti. In caso non lo fosse leggo il valore e lo salvo nella variabile corrispondente.

Funzione putop

La putop è la funzione che viene eseguita in seguito alla ricezione di un messaggio avente come campo operazione PUT_OP.
Questa funzione implementa al suo interno la funzione di inserimento presa dalla libreria icl_hash.h.
La putop prende come argomenti il puntatore alla repository, il puntatore al messaggio ricevuto e restituisce il valore dell'operazione del messaggio di

risposta.
Questo valore è uguale ad OP_OK se l'operazione è andata a buon fine, OP_PUT_SIZE se la dimensione del messaggio superasse la dimensione contenibile nella repository, OP_PUT_TOOMANY nel caso in cui la repository fosse satura di oggetti, OP_PUT_REPOSIZE nel caso in cui la repository avesse raggiunto il limite massimo di byte contenibili, infine OP_FAIL nel caso in cui si verificasse un generico errore.

Funzione Updateop

La Updateop è la funzione che viene eseguita alla ricezione di un messaggio avente come campo di operazione UPDATE_OP.

La Updateop implementa un metodo per controllare se l' elemento da aggiornare non esiste o se la sua dimensione non corrisponde alla dimensione del messaggio da sostituire. In entrambi i casi restituisce un errore, rispettivamente OP_UPDATE_NONE e OP_UPDATE_SIZE.
Questa funzione prende come argomenti il puntatore al messaggio da sostituire e il puntatore alla repository di oggetti.
Restituisce OP_OK nel caso in cui l' operazione HA un esito corretto, OP_FAIL nel caso di un generico errore.

Funzione Getop

La Getop è una funzione mandata in esecuzione alla ricezione dell'operazione GET_OP. Implementa una funzione di ricerca al fine di trovare e restituire la parte dati del messaggio corrispondente alla chiave.
La Getop prende come parametri il puntatore al messaggio chiamante e il puntatore alla repository, restituisce OP_OK in caso di successo, OP_FAIL in caso di fallimento e OP_GET_NONE in caso di assenza dell'elemento cercato.

Funzione Removeop

La Removeop viene eseguita alla ricezione di REMOVE_OP. Prende come parametri il puntatore al messaggio inviato e il puntatore alla repository.
Essa elimina l'elemento dentro la repository attraverso le funzioni FreeData e FreeKeyche rispettivamente si occupano del dato e della chiave del messaggio.

Viene restituito OP_REMOVE_NONE nel caso in cui l'elemento non fosse presente, OP_OK in caso di successo e OP_FAIL in caso di fallimento generico.

Funzione Lockop

La funzione Lockop assegna al flag globale, che mi dice se l'intera Repository è in stato di Lock o meno, il Thread_id del thread chiamante.

Funzione Unlockop

La funzione controlla il valore del flag globale. In caso quest'ultimo fosse uguale al Thread_id del chiamante, lo resetta a 0. Per indicare che la Repository è di nuovo libera.

Funzione Execute_OP

La funzione Execute_op, utilizzata dal server, prende come parametri il puntatore al messaggio inviato dal client, il puntatore alla repository ed il puntatore all' array di lock delle repository.
Una volta analizzato l'operazione del messaggio ricevuto chiama la funzione corrispondente, con il relativo aggiornamento delle statistiche.
Prima di fare ogni operazione controlla lo stato della Repository (se occupato o meno).
Una volta eseguita l'operazione salverà l'esito di quest'ultima in un messaggio di risposta che verrà inviato al Client.

2.3 Queue

La libreria di funzioni Queue gestisce tutte le operazioni sull'array circolare di connessioni, assieme a due variabili, una mutex e una variabile di condizione utilizzate per la mutua esclusione.

Funzione Newqueue

La funzione Newqueue inizializza la coda.

Funzione Empty_queue

Controlla che la coda sia vuota, accedendo al campo 'len' della struttura e assicurandosi che sia uguale a 0.

Funzione Full_queue

Controlla se la coda è piena, controllando il valore della lunghezza.

Funzione Push_queue

La funzione Push_queue inserisce un socket descriptor nella coda delle connessioni.
La funzione blocca la coda per assicurarsi che nessun vi possa accedere. Inserisce l'elemento nell'indice corrispondente al resto della somma tra il numero di elementi già inseriti e la posizione della testa, diviso il numero degli elementi massimi nella coda.
Incrementa il valore che indica il numero di elementi presenti nella coda, manda un segnale per svegliare gli eventuali thread che stanno aspettando una connessione.

Funzione Pop_queue

Essa restituisce l'elemento che è stato inserito in coda da più tempo.
La funzione blocca la coda per evitare che altri thread prelevino lo stesso elemento. Controlla che la coda sia vuota e, in caso affermativo, aspetta che si riempia. Mentre aspetta controlla inoltre il valore della variabile globale che traccia lo stato del server (per capire se deve essere spento o meno), in caso questa variabile sia settata a 0 esce dal ciclo e dalla funzione senza restituire nessuna connessione. Nel caso in cui la coda non fosse vuota viene prelevato l'elemento che deve essere restituito, decrementata la lunghezza e

ricalcolata la posizione della testa.

2.4 Erch

E' una semplice libreria in cui sono definite solo 4 macro.
Ognuna di esse controlla l'esito di una malloc, setta Errno e restituisce il tipo valore adatto in base a dove sia stata effettuata la malloc.

3. Script Bash

Nello script bash l'idea usata è molto semplice.
Ho un array di flag. Ogni elemento corrisponde a una coppia, terna o singolo valore da stampare a schermo.

Viene controllato se gli argomenti passati sono opzioni o il nome del file. In base al tipo di opzione letto si settano i corrispondenti flag a 1. Se non ci sono opzioni vengono settati tutti.

Si effettua un primo controllo sui i flag per la creazione dei titoli .
Il secondo controllo, quello più importante, avviene all'interno del ciclo che leggere il file testo riga per riga.
All'interno di questo ciclo vengono inoltre aggiornati i massimi valori richiesti.
Considerando che le opzioni non ricoprono tutti i valori presenti nelle statistiche abbiamo deciso di usare il flag corrispondete alle lock per capire il tipo di stampa da effettuare. Se questo flag è settato a 1 allora dobbiamo stampare solo l'ultimo Timestamp, se 0 allora stampare tutti le righe presenti nel file.