Membox

Gabriele Tenucci, Stefan Daniel Motoc

Giugno 2016

Indice

1 Overview

Il processo Membox rappresenta l'implementazione di un server concorrente che deve gestire le richieste di uno o più client, che possono connettersi ad esso tramite una socket AF_UNIX per richiedere di effettuare operazioni su una repository di oggetti.

2 Implementazione

Abbiamo deciso di suddividere il progetto in più moduli per semplificare il debugging e permettere una miglior comprensione della logica del programma.

2.1 Struttura dei file

Per funzionare, il processo Membox si serve, oltre alle librerie fornite, di alcune strutture dati, implementate nei seguenti file:

- threadpool.h

  Contiene le strutture dati del threadpool e dei parametri necessari per il corretto funzionamento dei thread worker

- queue.h

  Contiene la struttura della coda delle connessioni

- icl_hash.h

  Contiene l'implementazione della tabella hash utilizzata per creare la repository

- parser.h

  Contiene la struttura dati nella quale vengono salvati i parametri letti dal file di configurazione

2.2 Threads

Il processo principale è suddiviso in più thread.

2.2.1 Main Thread

Si occupa di inizializzare le strutture dati, di creare la socket tramite la quale il server comunicherà con i client e di generare il thread Signal Handler e i thread Worker, a cui farà da Dispatcher, inserendo in una coda le connessioni in entrata. Una volta ricevuto il segnale di terminazione, ha il compito di chiudere le connessioni, deallocare le strutture dati, e stampare le statistiche sulle operazioni svolte dai worker.

2.2.2 Signal Handler Thread

Cattura i segnali arrivati al processo, settando una varibile globale e stampando le statistiche del server su file nel caso di un segnale adeguato. Gli altri thread avranno il compito di dover controllare periodicamente il valore di handled_signal e agire di conseguenza. Qualsiasi segnale diverso dai seguenti verrà ignorato dal thread. I segnali gestiti sono:

- SIGINT, SIGTERM, SIGQUIT: in seguito alla loro cattura, il server deve terminare subito la sua esecuzione. Alla variabile handled_signal viene quindi assegnato il valore del segnale ricevuto, e viene inviato un segnale di risveglio a tutti i thread worker in attesa di ricevere nuove connessioni.

- SIGUSR1: questo segnale fa sì che il thread Signal Handler invochi la funzione che permette di stampare su file le statistiche del server relative al timestamp corrente.

- SIGUSR2 : una volta ricevuto tale segnale, il Signal Handler comunica al server, tramite la variabile handled_signal, che non dovrà più accettare nuove connessioni, ma che dovrà aspettare che quelle attualmente esistenti terminino, prima di liberare la memoria allocata e chiudersi.

Inoltre, il thread Signal Handler risveglia i thread worker in attesa di nuove connessioni, permettendogli di terminare correttamente la loro esecuzione.

2.2.3 Worker Threads

Formano il pool di thread, cioè l'insieme di thread che si occupano di gestire le connessioni presenti nella coda. Se la coda della connessioni è vuota, si mettono in attesa sulla variabile di condizione varCond, aspettando di essere risvegliati dal main in seguito all'arivo di una nuova richiesta di conessione, oppure dal thread Signal Handler in caso di arrivo di un segnale di terminazione.

Abbiamo deciso di implementare un thread specifico per la gestione dei segnali perchè questo ci ha consentito di evitare di associare una funzione ad ogni tipo di segnale catturabile. Questo ci dà una maggiore flessibilità nel momento in cui la gestione richiede l'utilizzo di parametri provenienti dal server, in quanto questi possono essere passati direttamente come argomenti alla funzione eseguita dal thread. Altrimenti, infatti, saremmo stati costretti a memorizzare i parametri necessari in variabili globali.

3 Queue

Il Main Thread, dopo aver accettato una conessione, ne inserisce il File Descriptor nella coda delle connessioni ricevute. Da qui, i worker preleveranno le connessioni in attesa di essere gestite.
Per implementare tale coda, ci siamo serviti di un array circolare, cioè di una struttura dati contenente un array e due indici: head e tail, che indicano rispettivamente il primo e l'ultimo elemento inserito. Questo ci ha permesso di allocare una struttura dati di dimensione fissa, uguale a quella specificata all'interno del file di configurazione, e di non dover gestire l'allocazione di nuovi elementi all'interno della coda stessa.
Abbiamo inoltre deciso, per semplificare la gestione della sincronizzazione tra thread, di utilizzare l'acquisizione e il rilascio della mutua esclusione direttamente all'interno delle singole funzioni della coda.

3.1 Dequeue

La funzione dequeue acquisisce la mutua esclusione sulla coda e preleva un elemento dalla posizione di indice head.
Nel caso in cui la coda fosse vuota, questa funzione mette in attesa il thread worker che l'ha invocata, fino a che non verrà risvegliato in seguito ad un inserimento o ad un segnale. Se il risveglio avviene grazie ad un inserimento, la funzione restituisce l'elemento appena messo in coda, altrimenti restituisce un valore di errore, che il richiamante dovrà occuparsi di gestire.

3.2 Enqueue

Questa funzione si occupa di inserire una nuova connessione nella coda, alla posizione indicata dall'indice tail.
Nel caso in cui nella coda non fossero presenti elementi, la funzione enqueue inserisce la nuova connessione ricevuta ed invia un segnale di risveglio ai thread messi in attesa dalla funzione dequeue.
La funzione restituisce un errore solo nel caso in cui la coda fosse piena.

4 Repository

Per salvare i dati della repository ci siamo serviti della tabella hash fornita, che ha permesso un migliore utilizzo delle risorse. La tabella hash, infatti, è stata particolarmente utile per abbassare i tempi di esecuzione delle operazioni, dato che ci ha consentito di gestire la mutua esclusione in maniera efficiente, tramite un array di lock.
E' stato inoltre necessario trovare un numero di righe adeguato per la tabella, così da aver il minor numero di collisioni possibile, evitando cioè che troppi elementi finissero nello stesso indice di tabella. Considerando che le specifiche richiedono che il server sia in grado di gestire decine di migliaia di connessioni, abbiamo deciso di utilizzare il numero primo 2111, un compromesso accettabile tra prestazioni e spreco di memoria.

4.1 Gestione Lock

Essendo Membox un processo multithreaded, più worker si ritroveranno ad accedere alla struttura dati. Per far sì che questo avvenga correttamente, è necessario utilizzare la mutua esclusione, che impedisce ai thread di lavorare contemporaneamente sullo stesso dato. Per implementarla, abbiamo deciso di non usare una singola lock in grado di bloccare l'utilizzo dell'intera repository, in quanto questo sarebbe stato poco efficiente, ma di utilizzare un array di lock. Tale array, definito all'interno della struttura dati threadPool, ha una lunghezza pari al numero di liste di trabocco contenute nella tabella hash. In questo modo, quando un thread ha bisogno di effettuare un'operazione, non è costretto a bloccare tutta la tabella fino al termine della sua esecuzione, ma solo la lista di trabocco associata alla chiave ricevuta, permettendo così agli altri worker di lavorare sulla parte restante della tabella.
Per calcolare la giusta lock da acquisire, viene invocata la funzione ulong_hash_function, definita all'interno del file threadpool.h.

5 Thread Pool

Il Thread Pool rappresenta l'insieme dei thread Worker generati dal Main per gestire le connessioni accettate dal server. Quest'ultimo, infatti, dopo aver accettato una nuova connessione, ne inserisce il file descriptor nella coda delle connessioni, dove i Worker andranno a ritirarlo, occupandosi di gestire le operazioni richieste.

5.1 makePool

Il pool viene creato attraverso l'invocazione della funzione makePool che, usando la funzione di libreria pthread_create dentro un ciclo, genera un numero di thread pari al valore della variabile ThreadsInPool, letta dal file di configurazione, impostando poi threadFun come funzione iniziale di ogni thread.

5.2 threadFun

Il thread inizia la propria esecuzione tentando di prelevare un elemento dalla coda, richiamando la funzione dequeue. Se tale funzione restituisce un File Descriptor non valido, significa che è stato catturato un segnale. In tal caso il thread deve terminare immediatamente, attraverso l'invocazione della funzione pthread_exit. Altrimenti, può iniziare a prelevare operazioni dalla connessione, controllando il valore della variabile globale lockedRepository, così da assicurarsi che la repository non sia in stato di lock da parte di altri thread:

- se lockedRepository è settata a 0, la repository non è bloccata, quindi il thread può procedere;
- altrimenti, il worker deve controllare che il valore della variabile sia uguale al proprio ThreadID:
  - Se lo è, significa che è il thread stesso ad aver richiesto la lock sulla repository, quindi può effettuare nuove operazioni.

– altrimenti un altro thread sta bloccando la tabella, quindi l'operazione non può essere completata e, di conseguenza, il thread dovrà rispondere con un header il cui campo op sarà OP_LOCKED, preparandosi a ricevere un altro messaggio.

5.3 Operazioni

Per scegliere l'operazione da eseguire, il thread worker controlla il campo op contenuto nell'header del messaggio ricevuto.
Le operazioni consentite sono:

- PUT_OP: inserimento di un elemento all'interno della repository
  Prima di inserire un elemento, si controlla che:

  – non sia stato raggiunto il massimo numero di oggetti consentiti per la repository, rappresentato dalla variabile StorageSize, letta dal file di configurazione. Se la dimensione attuale è maggiore di quella consentita, l'operazione di Put fallirà e il worker risponderà alla richiesta con un header il cui campo op conterrà OP_PUT_TOOMANY.

  – non sia stata raggiunta la dimensione massima della repository, in byte, identificata con StorageByteSize. Se la somma tra l'attuale dimensione della tabella e la lunghezza in byte del file da inserire è maggiore di quella consentita, il worker risponderà con un header contentente OP_PUT_REPOSIZE.

  – il dato da inserire non abbia una lunghezza maggiore di MaxObjSize. In tal caso, il thread setterà il campo op del proprio header a OP_PUT_SIZE.

  A questo punto il thread può procedere con l'inserimento, acquisendo la mutua esclusione sulla lista di trabocco associata alla chiave del dato da inserire. L'operazione restituisce un header contenente OP_OK se l'operazione è andata a buon fine, OP_PUT_ALREADY altrimenti.
  Viene infine rilasciata la mutua esclusione e il thread può eseguire un'altra operazione.

- UPDATE_OP: aggiornamento del dato di una chiave presente nella repository
  Il worker acquisisce la mutua esclusione sulla lista di trabocco associata alla chiave ricevuta. Successivamente, controlla che l'elemento sia già presente all'interno della repository:
  - se non lo è, viene restituito un header contenente OP_FAIL
  - altrimenti, si verifica che la lunghezza del dato ricevuto sia uguale a quella del dato già presente nella repository:
    * se hanno lunghezza diversa, viene restituito un header che contiene OP_UPDATE_SIZE
    * altrimenti si procede con la sostituzione
  - Il worker quindi elimina il dato precedente e prova ad inserire quello appena ricevuto.
    * Se l'operazione ha successo, viene restituito un header con OP_OK
    * OP_FAIL altrimenti

    A questo punto, viene rilasciata la mutua esclusione.

- GET_OP: restituisce il dato associato ad una chiave nella repository
  Il thread acquisisce la mutua esclusione e cerca all'interno della repository l'elemento associato alla chiave ricevuta.
  - Se l'esito è positivo, restituisce un messaggio con OP_OK nell'header e il dato trovato nel campo data.
  - Altrimenti restituisce un messaggio con OP_GET_NONE nell'header e campo data vuoto.

- REMOVE_OP: rimozione di una chiave e del relativo dato
  Dopo aver acquisito la mutua esclusione, il worker controlla che la chiave ricevuta sia presente nella repository
  Se lo è, tenta di rimuovere il dato associato alla chiave.
  Se ha successo restituisce un header con OP_OK.

Se non ha successo, o se la chiave non è contenuta nella repository, viene restituito un header con OP_REMOVE_NONE

- LOCK_OP: permette di bloccare l'intera repository, in modo tale da evitare che altri thread siano in grado di accedervi concorrentemente
  Controlla che la repository non sia già bloccata da un altro thread worker. Se non lo è, la blocca e restituisce un header contenente OP_OK. Altrimenti, restituisce OP_LOCKED.

- UNLOCK_OP: sblocca la tabella hash in seguito ad una operazione di lock
  Verifica di avere la lock sulla repository. Se ce l'ha, la rilascia e restituisce un header contenente OP_OK. Nel caso in cui non abbia la lock, restituirebbe OP_LOCKED. Se la lock non è settata, restituisce OP_LOCK_NONE.

Alla fine di ogni operazione vengono aggiornate le statistiche relative al server, in accordo al loro esito, controllando infine l'eventuale presenza di segnali di terminazione pendenti.

6 Gestione Connessioni

Il client si connette al server utilizzando le funzioni contenute nel file connections.h. Tutte le funzioni presenti settano errno in caso di errore, come specificato nella documentazione doxygen.
Il server Membox apre una socket che utilizza per comunicare con i client che, per connettersi, usano la funzione openConnection. Per l'invio dei messaggi sulla socket, si utilizzano le funzioni sendHeader e sendRequest, mentre per la ricezione readHeader, readData e readReply (o readRequest se a leggere è il server).

Tutte le funzioni che effettuano una lettura controllano che la lunghezza del dato letto sia effettivamente uguale a quella del tipo di dato, ad eccezione del campo buf, dove, se vengono letti meno caratteri di

quelli dichiarati nel campo len, si continua a ciclare fino a che non viene raggiunta la lunghezza dichiarata, oppure si riceve un messaggio di errore in seguito alla read. Se durante la lettura si verifica un errore, infatti, la funzione utilizzata restituisce -1 e setta errno.
Lo stesso principio viene applicato alle funzioni di scrittura sulla socket.

7 Bash Script

Lo script bash è progettato per riuscire ad interpretare le statistiche generate dal server Membox, che le salva su un file di testo.
Lo script può essere eseguito con l'aggiunta di alcuni flag, che stampano tutte e sole le informazioni ad essi relative. Il primo parametro passato deve contenere il nome del file delle statistiche, mentre i flag non presenti nelle specifiche vengono ignorati.

I flag passati come parametro vengono letti e salvati in un array di bit (0 se non presenti, 1 altrimenti). Essi possono essere passati in qualsiasi ordine, ma vengono sempre associati alla stessa posizione all'interno dell'array.
Lo script scorre il file riga per riga, selezionando solo i parametri richiesti dai flag. Nel caso in cui sia presente il flag -m, che richiede di calcolare i massimi valori relativi a MaxConn, MaxSize, MaxObj, si utilizzano variabili locali per salvare i valori, che verranno stampati solo dopo aver completato la lettura del file. I valori relativi agli altri flag vengono, invece, stampati riga per riga.

Il controllo della formattazione del file viene fatto progressivamente, durante ogni ciclo di lettura della riga.

8 File Parser

Per leggere i parametri di configurazione abbiamo utilizzato parser.c, che scorre il file riga per riga, ignorando quelle il cui primo carattere era '#', '\n' e '\0'. Il parser è pensato per funzionare sui file di configurazione forniti. Pertanto, è necessario che ogni dato sia introdotto dalla stringa "=<spazio>".
I parametri devono obbligatoriamente essere ordinati nello stesso modo in cui sono presentati all'interno dei file di configurazione forniti, ossia:
UnixPath, MaxConnections, ThreadsInPool, StorageSize, StorageByteSize, MaxObjSize, StatFileName.
Affinchè il parser legga la configurazione in modo corretto, è inoltre necessario che all'interno del file siano presenti tutti i sette parametri, divisi al più da una serie di commenti.

9 Problemi Noti

In fase di testing, ci siamo accorti di un problema riguardante l'invio e la ricezione di messaggi. Infatti, eseguendo test7, è possibile accorgersi di alcuni messaggi di errore relativi a dati non corretti. Dato che ciò si verificava di rado, abbiamo deciso di creare un test che non facesse altro che inviare e ricevere dati tramite socket, controllando poi che il messaggio ricevuto corrispondesse a quello inviato. Per far questo, abbiamo prima inizializzato il dato del messaggio utilizzando la funzione init_data, e successivamente controllato lo stesso usando la funzione check_data, entrambe presenti nel client.c fornitoci.
Eseguendo questo test, abbiamo riscontrato che i due messaggi corrispondevano quasi sempre, tranne in alcuni rari casi.
Il test utilizzato è contenuto nel file test_connections.c.

10 Difficoltà Incontrate

Inizialmente abbiamo riscontato un problema relativo alla scrittura e alla lettura tramite socket di messaggi di grandi dimensioni. Infatti, essendo il buffer della socket limitato, per messaggi molto lunghi è stato necessario effettuare più operazioni di read o write per garantire la completa lettura o scrittura del dato. Per fare questo, abbiamo inserito le operazioni all'interno di un ciclo, contando il numero di caratteri letti e assicurandoci di aver letto il numero corretto di byte.

Un'ulteriore problema è stato quello relativo al modo con cui implementare l'attesa dei thread worker, in caso di coda delle connessioni vuota. Abbiamo deciso di fare ciò direttamente all'interno della funzione dequeue. Essa, infatti, controlla se non ci sono elementi in coda: in tal caso fa una wait su una variabile di condizione, così da mettere il thread in attesa. Il risveglio può avvenire tramite una signal effettuata durante un'operazione di enqueue, oppure in seguito alla cattura di un segnale da parte del thread Signal Handler.

Abbiamo inoltre dovuto affrontare problemi di deadlock all'interno della coda, contenente un meccanismo di mutua esclusione tra thread. Infatti, in caso di coda vuota e ricezione di un segnale, non avevamo correttamente rilasciato la mutua esclusione.

Infine, abbiamo avuto numerosi problemi relativi a memory leaks, dovuti al mancato rilascio di alcune strutture allocate. Per risolvere, abbiamo dovuto modificare alcune strutture dati per permetterci di salvare i puntatori a tali strutture e poter quindi liberare correttamente la memoria.

11 Documentazione Doxygen

Per mostrare meglio la struttura del progetto, abbiamo configurato un generatore di documentazione, utilizzando doxygen. Per generare l'intera documentazione è sufficiente entrare nella directory del progetto e lanciare il comando doxygen. Tale documentazione sarà inserita all'interno della dirctory doc, in vari formati.