SPARSE - MEMBOX

Implementazione di una libreria per la gestione di matrici sparse e di un server concorrente per la gestione di un repository

Autori:
Emanuele Aurora
Vlad Alexandru Pandelea

Progetto Sistemi Operativi e Laboratorio A.A. 2015/2016

Indice

1 - Introduzione

La seguente relazione ha lo scopo di presentare l'implementazione dei due progetti per l'esame di Sistemi Operativi e Laboratorio, Sparse e Membox.

1.1 – Breve sintesi dei progetti

Sparse è una libreria atta a gestire matrici di numeri in floating point con precisione doppia, che sono “sparse”, cioè con una bassa percentuale di elementi diversi da 0 rispetto alla numero totale di elementi. La sua implementazione richiede che vengano memorizzati solo i numeri diversi da 0.
Il codice è commentato seguendo il formato Doxygen.

Membox è un server concorrente il cui scopo è gestire un repository di oggetti e le connessioni dei Client, i quali possono eseguire una serie di comandi sul repository. La concorrenza è gestita tramite un Pool di Thread e le connessioni con il meccanismo dei Socket.

2 - Discussione scelte SPARSE

Non sono state prese particolari scelte progettuali nell'implementare SPARSE.

Riguardo alla formattazione del file binario, è sorto il problema dell'endianess, cioè che due macchine diverse potrebbero formattare i file binari in modo diverso, dando vita ad errori gravi che renderebbero i file formattati illeggibili. Non sono state prese particolari contromisure per questo effetto, dato che riteniamo che vada oltre lo scopo del progetto, quindi si raccomanda di utilizzare le funzioni di salvataggio e caricamento da file binario sulla stessa macchina.

Una breve spiegazione del funzionamento dell'algoritmo di moltiplicazione è commentata sopra la funzione, ed è possibile ottimizzarlo ulteriormente, per esempio aggiungendo strutture di controllo per l'aggiunta di un elemento in O(1) oppure sfruttando il multithreading (questo particolare problema è facilmente parallelizzabile).

3 – struttura MEMBOX

Il progetto MEMBOX è costituito dai seguenti file:

- membox.c, membox.h
- connections.c, connections.h
- thread_pool.c, thread_pool.h
- icl_hash.c, icl_hash.h
- message.h
- ops.h
- stats.h
- membox.conf3
- testsave.sh testload.sh makefile

- più i file forniti dal docente non modificati (client.c, conf1, conf2 ecc)

3.1 – membox.c

Il file membox.c contiene la maggior parte del codice del progetto; al suo interno è gestita:
- l'apertura del file di configurazione e la sua lettura per creare la struct contente le informazioni base del server,
- la funzione che gestisce il comportamento dei singoli thread dopo essersi connessi ad un dato Client,
- l'implementazione di tutte le operazioni che un Client può richiedere al Server,
- tutte le chiamate di funzione che il Server può fare sulla tabella hash,
- la creazione e il funzionamento del thread che gestisce i segnali,
- la gestione del file tableFile (per il caricamento ed il salvataggio della tabella del server da file) e del file statsFile (per il salvataggio delle statistiche da file),
- il Main del server.

All'interno del Main viene inizializzato il server MEMBOX, e vengono creati:
struct di configurazione, struct delle statistiche, connessione sui socket, hash table, thread pool, coda delle connessioni Client, la mask sulle Signal poi tramandata a tutti i thread standard e il thread gestore delle Signal.

Il passo successivo riguarda il cominciare ad accettare Client e a gestire le connessioni; questo è implementato tramite un ciclo infinito che viene fermato solo quando si verifica un'errore nell'accettazione di un Client oppure dopo che è stata richiesta una chiusura del server tramite Signal (viene gestita con la variabile isClosed settata dal thread che gestisce le Signal). Se tutto va a buon fine, il Server inserisce il file descriptor nella coda della connessioni Client, e questo viene recuperato da uno dei thread, che gestirà le richieste.

Dopo essere uscito dal ciclo il Main gestisce una chiusura pulita di tutte le strutture dati aperte e l'aggiornamento e salvataggio del file delle statistiche; se l'uscita è stata causata da un Signal USR2 ed è previsto dal file di configurazione, il server salva lo stato attuale della tabella hash nel file indicato (dato che il formato è binario, si ripetono le considerazioni del progetto SPARSE).
La chiusura delle struct è preceduta da una join di tutti i thread in esecuzione, e verrà spiegato durante la trattazione dei segnali come è assicurata l'uscita di tutti i thread.

3.2 – connections.c

Il file connections.c gestisce tutte le connessioni tra i Client e il Server; le principali funzioni contenute sono openConnection, sendRequest e readMessage. Le funzioni sono usate in comune da Server e Client, cioè i messaggi sono inviati tra le parti con gli stessi meccanismi (sendRequest si occupa di scrivere un messaggio sul socket sia per una richiesta del Client sia per una risposta del Server), e readMessage può autonomamente comprendere se deve leggere il Data di un messaggio dopo averne letto l'Header e aver verificato il valore di operation.

3.3 – thread_pool.c

Il file thread_pool.c gestisce tutto quello che riguarda il thread pool e la coda dei Client. Al suo interno c'è il codice per:
- inizializzare la struttura thread pool, con tutti i suoi elementi,
- inizializzare i singoli thread della thread pool, assegnandogli la funzione opExecute (di membox.c) per poter gestire i messaggi dei Client,
- inizializzare la coda di Client, e tutte le funzioni associate per la sua gestione (pop e push di un elemento),
- la funzione threadLifeCycle che gestisce un thread dalla nascita alla chiusura,
- la chiusura pulita della thread pool e della coda di Client.

Di particolare importanza è la funzione threadLifeCycle; in questa è definito il seguente comportamento per i thread:
- comincia il ciclo, controlla se il Server sta chiudendo (isClosing),
- il Server non sta chiudendo, quindi prendi il primo file descriptor dalla coda dei Client libero (se non ce ne sono mettiti in attesa su una variabile di condizione, col tipico modello produttore consumatore, in modo da poter essere risvegliato dal Server quando questo mette un elemento nella coda),
- dopo aver preso il FD inseriscilo nell'array currentFD; questo viene usato dal Server per poter sapere in ogni momento quali FD sono in utilizzo,
- esegui la funzione opExecute, che gestirà tutti i messaggi del Client finché la connessione non sarà terminata,
- infine togli il FD da currentFD (è stato chiuso dentro opExecute).

3.4 – icl_hash.c

Il file icl_hash.c contiene il codice per la gestione della tabella hash necessaria per contenere gli oggetti del repository. E' stata usata la base fornita dai docenti, poi pesantemente modificata per adattarla al progetto e alle sue necessità.

Le principali modifiche riguardano:
- l'aggiunta alla struct dell'elemento length per conoscere a priori la dimensione dell'oggetto,
- la modifica delle funzioni insert, insert_update, delete,
- la creazione delle funzioni print_table, icl_hash_update, icl_hash_save e icl_hash_load.

Le modifiche ad insert e insert_update sono mirate soprattutto a far capire quale errore ha fatto fallire la funzione (usando op_err); infatti una put può fallire sia perché ci sono stati errori imprevedibili (fallimento di una malloc), sia perché l'oggetto con la chiave in questione esiste già nella tabella.

La creazione di icl_hash_save e icl_hash_load è stata necessaria per implementare il salvataggio della tabella nel suo stato corrente in formato binario all'interno di un file dato. Questa funzione verrà trattata più approfonditamente nel capitolo delle funzioni aggiuntive.

3.5 – file .sh e membox.conf3

I file testsave.sh e testload.sh sono stati appositamente creati per fare dei test della funzione di salvataggio su file della tabella; vengono implementati nei test 8 e 9 del makefile ed il loro funzionamento è il seguente:

testsave.sh fa la put di 500 elementi della tabella, salva il file delle statistiche e poi verifica che nelle statistiche siano presenti 500 put totali e 0 put fallite.

testload.sh fa la put di 500 elementi della tabella, salva il file delle statistiche e poi verifica che nelle statistiche siano presenti 500 put totali e 500 put fallite.

L'utilizzo di questi file è definito nel makefile:
test 8:

crea un Server membox, e prova testsave.sh; se il test passa killa membox con un Signal USR2, in modo da salvare la tabella nel file definito in membox.conf3. Dopo 3 secondi (quando la kill ha avuto effetto) crea un altro Server membox con le stesse configurazioni, ed esegue testload.sh.
In questo caso dato che deve aver caricato la tabella della precedente sessione tutte e 500 le put devono fallire. Se questo accade il test è superato.

Test 9:

crea un Server membox, e prova testsave.sh; se il test passa killa membox con un Signal QUIT, in modo da NON salvare la tabella nel file definito in membox.conf3. Dopo 3 secondi (quando la kill ha avuto effetto) crea un altro server membox con le stesse configurazioni, ed esegue testsave.sh.
In questo caso dato che deve aver fallito a caricare la vecchia tabella (la QUIT chiude la sessione di membox prima di fare il salvataggio) tutte le put devono avere successo. Se questo accade il test è superato.

3.6 – file .h

Tutti i file .h hanno lo scopo di definire le struct utilizzate nei file .c

4 – progettazione MEMBOX

4.1 – File di configurazione

Il file di configurazione è trattato nella funzione loadConf di membox.c. Essa legge le righe del file con la seguente logica:
- se la riga comincia con # la ignora (commento)
- se non è così divide la riga in 3 parti, separando su spazi, tab e invio
- se sono più di tre parti ritorna con errore, il file non è formattato correttamente
- altrimenti:
    - se la seconda parte è diversa da “=” da errore
    - altrimenti legge la prima e la confronta con tutte le variabili da inizializzare; se c'è riscontro legge la terza parte, se necessario la trasforma in intero, e da il valore trovato alla corrispondente variabile della struct conf

Questa implementazione implica due cose sul file di configurazione, cioè che l'uguale (“=”) deve essere separato dalle altri parti del messaggio almeno da uno spazio, e che se c'è una stringa PATH da leggere questa non può contenere spazi al suo interno.

La struct conf ha una configurazione standard, ed è stata aggiunta la possibilità di utilizzare quella invece di caricarne una propria; infatti è possibile avviare il Server membox in due modi:
- ./membox
- ./membox -f /path_conf

dove il primo caso indica di utilizzare la configurazione standard (definita in CREATE_STD_CONFIG dentro membox.c) e il secondo di aprirne una personalizzata nel PATH indicato.

Quando si apre una configurazione diversa, la struct viene prima inizializzata con CREATE_STD_CONFIG, così che si possa scegliere di modificare solo alcuni valori (viene considerata solo l'ultima corrispondenza trovata di una variabile) invece di doverli inserire tutti nel file.

4.2 – Protocollo di comunicazione

Il protocollo di comunicazione si basa sui Socket di tipo AF_UNIX.
I Client si devono connettere al Socket indicato nel file di configurazione e poi possono cominciare ad inviare messaggi al Server con le relative richieste.
I messaggi verranno trattati quando un thread libero avrà preso in carico la gestione del Client. Questi messaggi devono essere divisi in due parti, cioè header e data, definiti nelle rispettive struct.

Nel caso di messaggi che non richiedono la parte data (esempio GET_OP) questa deve essere comunque creata, e data → buf deve essere settato a NULL. Infatti il file connections.c capisce se deve inviare anche data oppure no basandosi sul fatto che questo sia NULL o meno.

Il messaggio viene spacchettato e inviato con una serie di write (2 se è solo l'header e 4 se è un messaggio completo), e quando il thread ha finito di leggere e trattare il messaggio, ne invierà uno di risposta con lo stesso meccanismo (se la risposta è composta solo dall'header verrà scritto nel buffer solo l'header).

La connessione può cadere da parte del Server per 2 motivi.

- Il Server riceve un segnale di QUIT, INT o TERM, quindi il server deve chiudere il più velocemente la connessione, e chiude i Socket dei Client aperti; in questo caso il Client si accorge del fallimento dalla chiusura del Socket.
- Il numero di Client nella coda di connessioni è eccessivo; per rispettare i limiti quando un Client si presenta in queste condizioni si deve chiudere la connessione. Il Server gestisce questa chiusura impacchettando un messaggio OP_FAIL e inviandolo a Client senza leggere sue possibili richieste. Quindi chiude subito il Socket, per cominciare a gestire nuove connessioni.

4.3 – Client Queue

La coda di Client ha lo scopo di gestire il flusso di Client che arrivano oltre alla capacità dei thread. Le dimensioni della coda sono prese dal file di configurazione (il limite di connessioni). La gestione dei Client è la seguente:

il Server, dopo aver ricevuto il FD di un Client, controlla se la coda lo permette (c'è abbastanza spazio) e inserisce il FD nella coda; la gestione del Client è poi completamente affidata alla coda stessa e a i thread che la utilizzano.

I thread da quando vengono creati provano a prelevare un elemento dalla coda, e se non ce ne sono si mettono in attesa su una variabile condizionale. Questo meccanismo permette ai thread di “staccarsi” da un Client al termine di una comunicazione e di potersi collegare col successivo.

4.4 – opExecute

La funzione opExecute è il cardine della trattazione dei messaggi inviati tra Client e Server. Questa richiama la readMessage interna a connections.c in modo da poter leggere un messaggio, e chiama la funzione corretta per gestirlo dopo averne verificato l'op.

E' stata definita una funzione per la gestione di ogni operazione, e queste settano sempre il messaggio di ritorno a seconda degli effetti che ha avuto la richiesta.

OpExecute aggiorna sempre le statistiche dopo aver chiamato una delle funzioni per gestire le operazioni, tranne in alcuni casi in cui questo è fatto dentro alle singole funzioni per motivi di praticità (per esempio nella PUT_OP prima di fare l'insert l'elemento deve essere prenotato, dato che c'è un limite al numero massimo di elementi della tabella e che questo può generare errori di concorrenza se non trattato nel modo corretto).

OpExecute esce solo dopo aver fallito la scrittura o la lettura di un messaggio; questo può avvenire perché il Client ha chiuso la sua connessione oppure perché c'è stato un Signal che ha artificialmente provocato disconnessione, ed è tramite questo meccanismo che ci si può assicurare la chiusura dei thread.

Alla chiusura opExecute verifica sempre che non ci siano lock globali attive del Client attualmente gestito, e se ci sono vengono chiuse prima di uscire.

4.5 – lock

Nel progetto sono state utilizzate molte lock per gestire le variabili usate in comune; vengono qui elencate le principali strutture dati che ne fanno uso:

La struct degli delle statistiche (una lock per elemento), la coda delle connessioni, una lock per accedere alla variabile global_lock_id (indica chi ha la lock globale e vale -1 se nessuno ce l'ha), una per accedere all'array currentFD.

Di particolare interesse sono le lock utilizzate per accedere alla tabella hash;

per gestire questa è stato deciso di usare un numero elevato di mutex per non rallentare l'accesso agli elementi, ma non eccessivamente così da non avere uno spreco di memoria.

Il calcolo del numero di lock usate per la tabella dipende dal numero di righe per la stessa, dove:
- se la tabella ha più di 1000 righe si prende lock = num_righe / 100,
- se la tabella ha un numero compreso tra 10 e 1000 righe vengono sempre usate 10 lock,
- altrimenti il numero di righe equivale al numero di lock.

Ad ogni lock quindi corrisponde una serie di righe consecutive della tabella, e quando un thread deve capire su quale mutex applicare la lock per una data key della tabella hash, prima applica la funzione hash per capire di quale riga si tratta e dopo calcola quale lock gestisce il gruppo di righe della tabella dove è contenuta la riga necessaria.

5 – Gestione dei segnali

La gestione dei segnali è completamente affidata al thread sigThread (membox.c), infatti nel main e in tutti gli altri thread vengono di fatto ignorati. SigThread ripete un ciclo infinito dove sfrutta la funzione sigWait per gestire un segnale alla volta, e ignora tutti i segnali non inerenti alle specifiche.

Mentre la Signal USR1 è un semplice richiamo ad una funzione, le implementazioni delle Signal di chiusura sono di particolare rilevanza; vengono quindi divisi i due tipi di chiusura:
- Chiusura veloce (SIGINT, SIGQUIT, SIGTERM),
- Chiusura pulita (SIGUSR2)

La chiusura pulita viene aziona tramite un meccanismo di variabili globali; la variabile isClosing (contenuta nella struct della coda di connessioni) permette a tutti i thread (e al Server stesso) di individuare uno stato di chiusura.

I thread considerano questo stato solo quando non stanno gestendo un Client, e si interrompono quando lo trovano.

Il server lo considera solo se non sta accettando un Client, quindi viene utilizzata la funzione shutdown per risvegliarlo e permettergli di cominciare con la procedura di chiusura.

Durante la procedura di chiusura il Server aspetta che tutti i thread abbiano completato il loro lavoro (tramite le join), e questo accade solo dopo che i Client hanno chiuso le connessioni già attive; infatti la chiusura pulita impone che le connessioni già attive siano terminate naturalmente.

Questo segnale permette di salvare la tabella corrente nel file indicato dal file di configurazione, se questo è settato, mentre non viene fatto se è una chiusura veloce.

La chiusura veloce impone invece che ogni thread si fermi il prima possibile (anche senza completare le operazioni in corso); per farlo è stato creato un array (currentFD) che indica tutti i FD aperti in un certo momento.

Il thread dei Signal chiude forzatamente tutti i Socket aperti, costringendo i singoli thread ad uscire dallo stato corrente.

Dopo aver sbloccato i thread si sblocca il main con lo shutdown, e poi vengono ripetute le istruzioni della chiusura pulita: si salvano le statistiche e si libera la memoria.

6 – Statistiche

Come sono gestite le statistiche è già stato discusso in precedenza; rimane comunque da sottolineare che sono state aggiunte statistiche aggiuntive:

- upd_ins e upd_ins_failed servono a descrivere l'operazione aggiuntiva UPDATE_INSERT, descritta successivamente,
- fail_lock serve a definire il numero di operazioni fallite a causa di una lock globale; opExecute da precedenza agli errori di tipo lock globale, quindi se uno di questi si presenta verrà segnalato anche se erano presenti altri errori (per esempio nel formato del messaggio). Quindi può capitare che le richieste non vengano eseguite e che appunto non rimanga traccia del loro passaggio, da cui è nata la necessità di aggiungere questa variabile.

7 – Extra e problemi riscontrati

Oltre ad i due test su testload.sh e testsave.sh è stato aggiunto un particolare test per la verifica del funzionamento del multithread su membox. Questo test si basa sul principio che più thread possano fare un lavoro sequenziale più velocemente di un singolo thread, anche se questo può non essere vero in un processore singol core.

Quindi il funzionamento del test è proprio di far ripetere due volte in sequenza la stessa operazione ad due Server che hanno configurazioni leggermente diverse: uno è limitato ad un thread e l'altro a 20 thread.

Il test è superato se il tempo per il processo a singolo thread è maggiore di quello per il processo multithread.

Un'altra funzione extra è quella di salvataggio della tabella hash in un file definito nel file di configurazione. Questo salvataggio avviene solo se è presente l'opzione LoadTablePath nel file di configurazione; se questo esiste, il programma prova ad aprirlo per caricare la tabella presente nel file (se presente). Se non esiste semplicemente lo ignora fino al momento della chiusura del server.

Una volta arrivata la SIGUSR2 (non funziona con gli altri segnali di terminazione) il Server apre in scrittura il file e riscrive la tabella al momento della chisura. Se ci sono stati altri errori che hanno portato la terminazione del server, il vecchio file (se esisteva) che conteneva la tabella non viene modificato.

Un'operazione extra implementata nel Server è UPDATE_INSERT, che agisce in modo simile alla UPDATE, però inserendo l'elemento se questo non esisteva prima. Non è stata aggiunta però al file delle statistiche, in quanto questo poteva interferire con i test e lo script bash. Non è possibile testare questa operazione senza modificare il Client, dato che nel codice l'operazione non è riconosciuta e non viene inviato anche la parte DATA necessaria alla sua esecuzione.

Uno dei problemi riscontrati è stato quello della lock globale; per risolvere tutte le richieste vengono effettuati i seguenti passaggi:

il thread che intende prendere la lock globale deve attendere che tutti gli altri thread abbiano terminato le operazioni locali. Per farlo prima modifica una variabile comune a tutti i thread che indica la sua intenzione a effettuare una lock globale;
con questa operazione tutti i thread che successivamente vogliono collegarsi alla tabella, si accorgono che c'è uno stato di lock e riportano un errore al Client.

Successivamente il Server si mette in attesa per ottenere tutte le lock locali della tabella, così da avere “quasi” la certezza che nessun thread sia in attesa di prendere una lock locale per fare un'operazione nella tabella; infatti, tutti i thread che avevano già acquisito una lock locale hanno terminato la loro operazione e gli è stato sicuramente vietato di rimettersi in attesa su una lock locale.

Esiste in realtà una trascurabile possibilità che questo meccanismo non funzioni, per esempio con le seguenti azioni:

- un thread si appresta a fare un'operazione sulla tabella; per farlo come prima cosa verifica la variabile comune che dice se c'è qualche altro thread in procinto di fare una lock globale. Questa variabile è falsa, quindi si appresta a fare la lock locale, ma viene interrotto prima di cominciarla,
- il thread che lo interrompe è intenzionato a fare una lock globale, e modifica il valore della variabile che lo indica; si noti come questo non impedisce al thread originale di continuare il suo cammino,
- il thread originale viene riesumato, e si mette in attesa sulla lock locale.

Per ovviare a questa possibilità, è stato deciso che anche il thread che ha controllo globale deve fare le lock locali quando opera sulla tabella, infatti dopo aver preso tutte le lock locali le rilascia immediatamente per poterle riprendere in future operazioni.

Come principale interpretazione della lock globale abbiamo inteso che chi effettua la lock globale non voglia che nessun altro thread interferisca con i valori che il primo prova a modificare-leggere nella tabella. Questa specifica è stata violata con la precedente implementazione, però la possibilità che questo crei problemi è infinitesima: non solo si deve presentare il caso limite descritto precedentemente, ma il primo thread (che potrà fare solo un'operazione nella tabella prima di essere bloccato) dovrà interferire con la stessa key del secondo e dovrà farlo dopo di lui.