File Storage Server

Andrea Simone Costa
597287

July,13 2021

Contents

1 Introduzione

Il progetto è stato suddiviso in una decina di moduli, descritti nel seguito della relazione, ognuno con un proprio compito ben definito. Ogni modulo possiede un proprio Makefile, alcuni Makefile hanno delle dipendenze verso altri Makefile, ma ovviamente è presente il Makefile top-level che permette di compilare l'intero progetto e di eseguire i test.

2 File-System

Thread safety Il file-system utilizza un doppio livello di mutex per garantire la correttezza e l'efficienza delle operazioni. Vi è una mutex globale (`FileSystem.overallMutex`), che blocca qualunque operazione sull'intero file-sytstem, la quale è utile per eseguire operazioni delicate come la rimozione di un file e la rimozione dei dati collegati ad un particolare client, oltre che per avere la certezza che non possano esservi thread in determinate parti del codice richiedenti una maggiore attenzione. Il secondo livello di mutex è presente su ogni singolo file, ovvero le note `mutex` e `ordering` unite alla variabile di condizione `go` e ai counter `activeReaders, activeWriters`, livello che implementa il classico pattern lettori/scrittori senza priorità.

Flags e lock Ogni file possiede un field `currentlyLockedBy`, indicante il client che ne possiede la lock, una queue `openedBy` che contiene un riferimento ai client che hanno aperto il file, una queue `waitingLockers` che contiene un riferimento ai client che sono in attesa di ottenere la lock sul file e un field `ownerCanWrite` indicante il client che ha l'autorizzazione di effettuare una write sul file, in seguito all'apertura dello stesso con entrambi i flag. Tale flag viene opportunamente resettato quando necessario.
La queue `waitingLockers` evita di dover fermare un thread worker in attesa di far ottenere la lock sul file ad un client; il client verrà inserito nella coda che è stata opportunamente gestita affinché, prima o poi, riceva un risposta, sia essa posiitva o negativa, ad esempio a causa della eviction del file di interesse.

Invarianti Le seguenti invarianti vengono rispettate dalle operazioni eseguibili sul file-system:

- non è possibile aprire un file lockato da un altro client
- è possibile aprire e chiudere un file non lockato né dal client richedente né da un altro client
- è possibile lockare e unlockare un file non aperto dal client richedente
- non è possibile lockare immediatamente un file lockato da un altro client, il client richiedente finisce in attesa
- non è possibile leggere un file lockato da un altro client
- non è possibile rimuovere un file non lockato dal client richiedente
- è possibile chiudere un file lockato da un altro client
- non è possibile aprire un file precedentemente aperto dal client richiedente e mai chiuso

Implementazione Internamente il file-system utilizza sia una lista che un dizionario per la memorizzazione dei file, due strutture dati che vengono tenute sincronizzate. Il dizionario velocizza la ricerca di un file tramite path, mentre la lista agevola l'implementazione delle politiche di rimpiazzamento.

LRU Se la politica di rimpiazzamento scelta è la `LRU`, allora le varie operazioni sui file hanno come conseguenza, a basso livello, una estrazione ed un reinserimento in testa nella lista dei file, lista che di base fornisce una politica `FIFO` out-of-the-box.

Moduli Il modulo client utilizza estensivamente i seguenti moduli: list, icl_hash.

3 Server

Thread Sono presenti tre diverse tipologie di thread: il thread principale gestore dell'intero server, il thread worker adibito alla gestione della richiesta di un client e un thread adibito alla ricezione e gestione dei segnali.
Il thread principale comunica con i worker tramite una queue unbounded thread-safe, inviando tramite essa gli `fd` dei client aventi una nuova richiesta da soddisfare, mentre i worker restituiscono al server gli `fd` dei client la cui i-esima richiesta è stata gestita tramite una pipe dedicata. Il thread principale utilizza la funzione `select` per la gestione dei vari `fd` in gioco. Ricevuto un segnale, il thread dedicato comunica con il thread principale attraverso un'altra apposita pipe.

Segnali Il server ignora completamente `SIGPIPE`, mentre i segnali `SIGINT`, `SIGQUIT` o `SIGHUP` sono stati mascherati come richiesto. La loro ricezione comporta l'invio di un messaggio specifico al main thread, il quale si comporta di conseguenza come richiesto nelle specifiche.
Il thread dedicato maschera anche i segnali `SIGTSTP` e `SIGTERM`, ma solo per poter stampare un piccolo messaggio; la loro ricezione comporta l'immediata terminazione, senza se e senza ma, del server.

Moduli Il modulo server utilizza estensivamente i seguenti moduli: file-system, communication, logger, config-parser, simple-queue e list.

4 Client

Semantica delle option La semantica di alcune option non era ben definita, perciò sono state prese le seguenti decisioni:

- option `-D`: è valida solo se il parametro `dirname` è presente e se essa segue immediatamente una option `-w`, `-W` o `-a`. Il suo campo di azione riguarda infatti solo l'option, se è valida, precedente. La folder `dirname` è considerata la folder root per i file restituiti dal server, i quali possiedono un path assoluto che ben si presta a questa interpretazione.
- option `-d`: è valida solo se il parametro `dirname` è presente e se essa segue immediatamente una option `-r` o `-R`. Il suo campo di azione riguarda infatti solo l'option, se è valida, precedente. La folder `dirname` è considerata la folder root per i file restituiti dal server, i quali possiedono un path assoluto che ben si presta a questa interpretazione.
- option `-w`: questa opzione tenta una `openFile(..., O_CREATE | O_LOCK)` su ogni file passatole come parametro e, per i file dove ha successo, scrive il contenuto tramite una `writeFile`.

Option aggiuntive Ulteriori option per il client sono disponibili:

- `-O dirname` imposta la directory dove memorizzare i file eventualmente evictati da una `openFile`.
- `-o file1[,file2]` apre una lista di file passati come parametro senza impostare alcun flag.
- `-e file1[,file2]` apre una lista di file passati come parametro impostando il flag O_CREATE.
- `-p file1[,file2]` apre una lista di file passati come parametro impostando il flag O_LOCK.
- `-s file1[,file2]` chiude una lista di file passati come parametro.
- `-a fileSource,fileDest1[,fileDest2]` legge dal disco il file *fileSource* e ne appende il contenuto nei file destinazione seguenti che devono essere già presenti sul file-system remoto. I file eventualmente evictati dal server possono essere salvati sul disco utlizzando l'option `-d`.

Moduli Il modulo client utilizza estensivamente i seguenti moduli: api, command-line-parser e list.

5 API

Client API Questo modulo, contenuto all'interno del modulo `Client`, si occupa di implementare l'API al quale ogni client deve fare riferimento per poter comunicare con il server, oltre a provvedere funzioni come `writeLocalFile` e `readLocalFile` per leggere e scrivere file da/sul disco.

Moduli Il modulo API utilizza estensivamente i seguenti moduli: communication.

6 Communication

Protocollo di comunicazione La comunicazione fra client e server è basata sullo scambio di molteplici messaggi, ognuno dei quali è composto da un numero di byte fissi per indicare la lunghezza dei dati scambiati e poi i dati a seguire.
Il client, per ogni operazione che desidera eseguire, invia innanzitutto un codice che identifica, univocamente, tale operazione, per poi inoltrare i dati necessari. Ogni operazione richiesta dal client riceve come prima risposta un codice contenente il risultato della stessa, `0` in caso di successo, `1` in caso di insuccesso. Se l'operazione remota è fallita seguirà una stringa contenente un messaggio di errore, mentre se ha avuto successso seguiranno uno o più messaggi contenenti il risultato.

7 Parser del file di configurazione

Formato Il file di configurazione è un file testuale contenente una lista di coppie chiave-valore, dove la chiave è separata dal valore da una virgola, e le varie coppie sono separate da delle newline. Il parser, utilizzato dal modulo server, richiede in ingresso un file da analizzare e permette di leggere, data una chiave nota, il corrispondente valore.

8 Parser degli argomenti

Per l'analisi degli argomenti passati da linea di comando è stato preferito l'uso di un parser creato ad hoc, il quale fornisce in una lista le coppie opzione-parametro.

9 Thread-safe queue

Il modulo server necessita di una coda thread-safe, e unbounded, per poter comunicare ai worker i file descriptor dei client in attesa di essere gestiti. Questa coda ha la caratteristica di poter essere eliminata in qualunque momento in totale sicurezza, eliminazione che segnala ai worker la necessità di terminare immediatamente.

10 List

Il modulo list è un ADT generico che fornisce tutte le principali operazioni effettuabili su una lista, ed è stato estensivamente utilizzato in tutto il progetto.

11 Test

Il modulo test contiene i tre differenti test richiesti nelle specifiche e una folder, `local-file-system`, contenente alcuni file che i vari client scrivono sul server. Ogni sub-folder di test contiene lo script bash per avviare il test e il file di configurazione richiesto dal server, il quale ne attende il path come argomento da riga di comando.

Nota bene La maggior parte degli output sono stati redirezionati.
Dopo aver eseguito un determinato test, all'interno della corrispondente folder sarà presente una sub-folder `output` contenente:

- un file `log.txt` nel quale è inserito l'output del modulo logger
- un file `server.txt` nel quale sono inserite le stampe del modulo server (solo stdout)
- più file `client[...].txt` nei quali sono inserite le stampe dei client (sia stdout che stderr)

I file letti/evictati saranno invece disponibili in un'altra sub-folder, ovvero `clients`.

Nota bene Il test numero uno, con i suoi `100+ Megabyte` allocati, necessita di un tempo paragonabile, se non superiore, a quello del test numero tre per essere completato, sia perché valgrind è chiamato in causa, sia perché la folder `local-file-system`, da `30+ Megabyte`, viene interamente caricata nel file-system. Nella macchina sulla quale è stato svilluppato il progetto sono solitamente necessari `30+` secondi per ottenere il risultato sperato.

12 Note

Gestione degli errori La variabile `ERRNO` è stata utilizzata sia in lettura che in scrittura in più occasioni, anche se non è stata la scelta principale per il report degli errori dei vari moduli. Ogni modulo dichiara infatti una propria lista di errori, con messaggi corrispondenti, e praticamente tutte le funzioni accettano un parametro aggiuntivo `int* error` nel quale inoltrare un eventuale errore.
In certi casi la gestione degli errori avrebbe potuto/dovuto essere più fine, ma per ragioni di tempo alcune eventualità sono state volutamente ignorate.

Problemi noti Il client non si comporta sempre bene se riceve in ingresso path di file/cartelle non esistenti sul disco locale, a volte terminando in modo a dir poco brutale. Sono stati riscontrati problemi anche con path inusuali, contenenti ad esempio spazi o caratteri speciali. In taluni casi anche il server è entrato in difficoltà, suppongo a causa di problemi durante lo scambio di tali path con il client.
Ritengo doveroso anche segnalare la mancanza di controlli, lato server, per evitare lock circolari sui file nel file-system. Poniamo ad esempio che il client `a` possieda la lock sul file `A`, mentre il client `b` possiede la lock sul file `B`. Se il client `a` richiedesse ora la lock sul file `B`, mentre il client `b` richiedesse contemporaneamente la lock sul file `A`, questi due client si troverebbero in deadlock. Il problema sarebbe risolubile mantenendo un grafo delle dipendenze, impedendo tutte le operazioni che genererebbero cicli in tale grafo.

Test dei moduli Ogni modulo possiede una propria cartella di test nella quale sono presenti alcuni test specifici per il modulo, la maggior parte non aggiornati alle ultime modifiche.

Codice di terze parti Nel progetto è stata utilizzata la funzione `iap_getparents` realizzata da *The Apache Software Foundation* sotto licenza *Apache Software Foundation (ASF)*, la libreria `icl_hash` realizzata da *Jakub Kurzak* e le funzioni `readn` e `writen` realizzate da *W. Richard Stevens and Stephen A. Rago*.

Ambiente di sviluppo Per un problema hardware del pc sono stato costretto ad utilizzare una macchina a 32 bit (Raspberry Pi), sulla quale ho sviluppato la maggior parte del progetto. Grazie all'aiuto di due colleghi che ringrazio infinitamente, ovvero Francesco Lorenzoni e Giorgio Dell'Immagine, ho potuto testare il progetto sulla macchina virtuale fornitaci, nella quale per fortuna non abbiamo riscontrato problemi rilevanti.

Requirement aggiuntivi In aggiunta ai requirement base sono stati implementati: i file di log tramite un modulo *logger* dedicato, il target *test3*, le primitive `lockFile`/`unlockFile`, il supporto all'option `-D` e l'eviction policy `LRU`. Inoltre, ulteriori option per il client sono disponibili.

Repository GitHub È possibile trovare il codice sorgente del progetto seguendo il link https://github.com/jfet97/file-storage-server.