Relazione Progetto SOL 2020-2021

Sofia Pisani
Matricola: 646301

07/09/2022

Contents

1 Istruzioni per l'uso

1.1 Compilazione

Il progetto non include eseguibili precompilati. Prima di utilizzarlo bisogna utilizzare uno dei seguenti comandi make:

make all : Genera gli eseguibili server.out e client.out nella cartella out.

make server : Genera l'eseguibile server.out nella cartella out.

make client : Genera l'eseguibile client.out nella cartella out.

make clean : Ripulisce la cartella out e la cartella obj dai moduli oggetto compilati.

make test[1,2,3] : Genera gli eseguibili necessari ed esegue automaticamente il test richiesto.

1.2 Client

1.2.1 Avvio

Per lanciare il client, dopo averlo compilato, eseguire il file client.out, dandogli delle opzioni valide.

1.2.2 Comandi

Di seguito una lista dei comandi disponibili e i loro effetti:

-h : Stampa una lista dei comandi disponibili e i loro effetti.

-f filename : Connette il client al socket di nome filename.

-w dirname[,n=0] : Scrive sul server fino a n file dalla cartella dirname, visitando ricorsivamente le subdirectory. Se n=0 o non è specificato scrive tutti i file trovati.

-W file1[,file2] : Scrive sul server tutti i file specificati.

-D dirname : Imposta la cartella in cui il client salvera i file espulsi dal server a seguito di un capacity miss. Se non impostata i file espulsi dal server verrano ignorati.

-r file1[,file2] : Legge dal server tutti i file specificati.

-R [N=0] : Legge n file qualsiasi dal server. Se n=0 o non è specificato scrive tutti i file trovati.

-d dirname : Imposta la cartella in cui il client salvera i file letti. Se non impostata i file letti verranno ignorati.

-t time : Imposta il tempo in millisecondi che intercorrerà tra una richiesta al server e la prossima (0 di default).

-l file1[,file2] : Ottiene una lock sui file specificati.

-u file1[,file2] : Rilascia la lock sui file specificati.

-c file1[,file2] : Rimuove tutti i file specificati dal server.

-p : Abilita le stampe sullo stdout.

1.3 Server

1.3.1 Avvio

Per lanciare il server, dopo averlo compilato, eseguire il file server.out, passandogli opzionalmente da linea di comando un path al file di configurazione da usare. Se non specificato cerchera un file config.txt nella directory corrente, e se non trovato usera delle impostazioni di default.

1.3.2 File di Configurazione

Un file di configurazione valido è una serie di coppie CHIAVE=VALORE, ognuna su una linea diversa. Il file di configurazione può anche includere commenti, cioè linee che iniziano con // verranno ignorate. Le chiavi disponibili, insieme a una breve descrizione dei loro effetti e ai loro valori di default, possono essere trovati nel file config.txt generato durante la compilazione del server.

2 Architettura del Server

Il client ha un architettura estremamente semplice, limitandosi a parsare le opzioni da linea di comando e lanciando le richieste necessarie al server tramite l'API. Per questo non è particolarmente interessante, e ci limiteremo a parlare dell' architettura del server.
Il server ha infatti una struttura ben più complessa, dovendo gestire i file memorizzati al proprio interno, e connessioni simultanee da più client.

2.1 Layout dei Thread

Il server funziona su multipli thread:

2.1.1 Main Thread

Il main si occupa dello startup del server, facendo il parsing del file di config, inizializzando appropriatamente le strutture dati necessarie, e inizializzando la ThreadPool. Una volta fatto questo andra ad accettare connessioni, e passarle alla ThreadPool perchè le gestisca.
Infine, gestisce la terminazione intercettando i segnali SIGINT SIGHUP e SIGQUIT alla cui ricezione provvedera a liberare le risorse e a stampare il sunto delle statistiche.

2.1.2 ThreadPool Manager

Il manager della ThreadPool è un thread che gestisce tale struttura. Si occupa principalmente di generare e uccidere thread a seconda delle necessità, oltre a svolgere un ruolo nella terminazione della ThreadPool. I thread generati al suo interno si occupano poi indipendentemente di consumare la queue di task submittate alla ThreadPool.
È stato scelto di implementare una ThreadPool dinamica, quindi che si ridimensioni a seconda del carico di lavoro, per permettere al server di essere più reattivo di fronte ad alti numeri di connessione. Sopratutto, dato che ogni thread si occupa di una singola connessione, operazioni bloccanti quali una lock andrebbero a bloccare una grande parte delle risorse del server, senza un effettivo uso di cpu. Il server può comunque essere eseguito con una ThreadPool statica semplicemente impostando la CORE_POOL_SIZE uguale alla MAX_POOL_SIZE nell file di configurazione. Test1 e Test2 sono eseguiti così di default, mentre il Test3 ha di default una MAX_POOL_SIZE illimitata, in modo da poter testare anche questa funzionalità. Se questo fosse un problema vi invito gentilmente a cambiare il file di configurazioni src/TESTS/test3.txt prima di eseguire il make test3.

2.1.3 Worker Thread

I worker thread si occupano di consumare le task submittate alla ThreadPool. Il server genererà una task per ogni connessione, e quindi i worker thread andranno a occuparsi della gestione di un intera connessione alla volta, secondo il seguente loop:

- Ricezione della richiesta.
- Processing della richiesta, facendo le adeguate chiamate al FileSystem e generando una risposta adeguata.
- Invio della risposta.

Questo loop tende chiaramente a fare molte chiamate bloccanti, in quanto sia la ricezione che l'invio dipendono dalla disponibilità del client, oltre ad eventuali richieste di lock. Non abbiamo considerato che questo fosse un grande problema in vista della ThreadPool dinamica, che ci permette di generare un numero adeguato di worker thread, sfruttando quindi il multi-threading al suo massimo per ottimizzare chiamate bloccanti o I/O bound.
I worker thread vanno anche a tenere traccia dello stato della loro connessione corrente, tenendo aggiornata una struttura ConnState.

2.2 Stato del server

Il server deve tenere traccia di:

Informazioni relative ai file memorizzati. Vengono conservate nella struttura FileSystem, e gestite tramite le funzioni che usano questa struttura.

Informazioni relative alle singole connessioni. Ogni thread gestisce una connessione singola, e ne ricorda lo stato in una struttura ConnState. Informazioni rilevanti sono ad esempio i file aperti al momento e il file lockato se presente.

Informazioni relative ai thread in esecuzione. Queste vengono gestite autonomamente dalla ThreadPool.

Queste tre strutture dati sono le più complesse e interessanti, e sono quindi quelle di cui parleremo nel dettaglio.

2.2.1 FileSystem

Il FileSystem deve tenere traccia di:

I File presenti nella memoria. Vengono conservati in un HashTable per permettere accessi efficienti.

La capacità corrente del server. Il numero corrente di File nel FileSystem e la dimensione totale di questi vengono conservati in due AtomicInt.

I metadati dei File. Questi servono ad implementare delle policy informate per la gestione dei capacity miss. Vengono conservati in una List.

Per la gestione della concorrenza, abbiamo un R/W lock su tutto il FileSystem, e una mutex per gli accessi alla lista dei metadati. L'implementazione dell HashTable è inoltre thread-safe, permettendo accessi concorrenti.
Un File è visto come un buffer di una certa dimensione, identificato univocamente da un nome. Non essendo nel FileSystem presente un astrazione delle directory, il nome è l'intero path al File. I File supportano una compressione seamless, opzionalmente attivabile dal file di config. Se questi vengono impostati come compressi, infatti, sulle write/append comprimeranno automaticamente i propri contenuti. Nelle read, invece, li decomprimeranno nel buffer del risultato. La compressione è implementata dalla libreria zlib.
Se un operazione farebbe superare al FileSystem i limiti della capacità stabiliti all inizializzazione, non la esegue, restituendo EOVERFLOW in errno. Quando accade ciò il chiamante dovrà occuparsi di chiamare freeSpace()

specificando quanto spazio liberare, causando una capacity miss. Le capacity miss sono gestite quindi all interno di freeSpace, che farà una chiamata alla policy per la scelta di un bersaglio da eliminare. Questo prenderà sempre il primo File (non locked) dalla List dei metadati, che verrà però prima ordinata secondo un euristica appropriata a seconda della policy scelta in configurazione. Tramite questo meccanismo sono supportate policy arbitrarie, di cui ne è implementata una certa varieta, dalle più alle meno ottimali (o sensate).

2.2.2 ThreadPool

La struttura ThreadPool deve tenere traccia di:

Le task da eseguire. Vengono conservate in una SyncQueue, una coda sincrona che permette ai thread di venire deschedulati mentre attendono nuovi elementi.

I thread vivi. Tiene il numero dei thread vivi in un AtomicInt, e i loro pid in una List. Queste sono aggiornate automaticamente dai worker thread alla loro nascita e morte.

La terminazione dei worker thread è gestita prima di tutto con la flag terminate. Questa viene controllata da ogni worker thread prima e dopo l'aver estratto una task dalla SyncQueue. In quanto thread in attesa di nuovi lavori nella SyncQueue sono però deschedulati, vengono anche inseriti nella pool dei lavori fittizi die(), che causano la terminazione del thread chiamante, svegliando quindi e terminando tutti i thread in attesa. Si supporta inoltre la cancellazione dei worker thread, per quanto questa sia sconsigliata in quanto rischia di portare alla perdita di risorse. Questa viene chiamata nella FastExit solo come ultima risorsa, se i thread non terminano in modo pulito in un tempo ragionevole.

2.2.3 ConnState

La struttura ConnState deve tenere traccia di:

I file aperti. I FileDescriptor restituiti dal FileSystem vengono conservati in un HashTable con chiave il nome del File.

Il file locked. Il FileDescriptor di un eventuale File in stato locked viene anche conservato separatamente, in modo da poter implementare un limite ad un singolo File locked per connessione.

All apertura di un File, il FileSystem genera un FileDescriptor. Questo associa al nome del File delle flag, che segnano quali permessi l'utilizzatore ha su di questo (lettura, scrittura, se il File è lockato o meno). Per successive operazioni sul File dovremo poi usare questo FileDescriptor.
È stato scelto di limitare il numero di File locked in un determinato momento a uno per connessione. Quando quindi una connessione va a chiedere la lock di un File, implicitamente richiede la unlock di eventuali File che tenesse locked. Questa scelta implementativa ci è sembrata ragionevole e giustificata per una serie di motivazioni. Prima di tutto, nell API come dato nel testo del progetto, è implicito che due client possano mandarsi in deadlock a vicenda. Se infatti il ClientA locka prima file1 e poi file2, e ClientB locka prima file2 e poi file1, per le specifiche dell API si deve andare in deadlock, almeno che non si possano rilasciare File già lockati. Questo è quello che facciamo, rilasciando sempre il file già locked, limitando quindi i client a un solo file locked alla volta, e risolvendo qualsiasi situazione di deadlock tra client. Inoltre, sono pochi i workflow in cui sarebbe ragionevole ottenere la lock su più file in contemporanea, mentre qualsiasi workflow rischierebbe di andare in deadlock senza questa limitazione. Ci sembra una buona idea dare agli utenti una limitazione chiara se gli evitiamo così i rischi di malfunzionamenti ben più opachi.

3 Comunicazione Client/Server

La comunicazione client/server accade tramite socket AF_UNIX. Sopra questi socket vengono serializzati e deserializzati una serie di strutture, prima di tutte il Message. I Message rappresentano alternativamente una

richiesta dal client al server, o una risposta dal server al client. Contengono due valori numerici: Un tipo, che contiene il tipo della richiesta (ad esempio di read o di write),e uno status, che contiene l'esito della risposta (OK o un qualche tipo di errore). Contengono inoltre una stringa null terminata info, in cui il server metterà messaggi informativi sul esito della richiesta, e che il client userà a seconda della richiesta, spesso per specificare il nome del File bersaglio. Infine possiedono un contenuto, un array di bytes che può essere usato a seconda della richiesta, o in risposta per mandare contenuti di file in seguito a una read o a un capacity miss.
Per mandare più File in un solo Message, come a seguito di una capacity miss o di una readN, esistono i FileContainer. Questi associano semplicemente a un contenuto opaco un nome. Sono però presenti metodi per serializzare e deserializzare array di FileContainer in un singolo buffer, permettendo di mandare un numero arbitrario di File in un unico Message, senza perderne il nome.

4 Migliorie Possibili

Per quanto il sistema nella forma corrente sia perfettamente funzionale, e capace di gestire un traffico moderatamente alto, sono possibili notevoli migliorie nella sua usabilità.
Un client interattivo, che aspetti quindi comandi da linea di comando, permetterebbe workflow molto più interattivi e complessi.
Inoltre, al posto di limitare i client a tenere una lock per volta, preferiremmo implementare metodi diversi di prevenzione dei deadlock. Primo tra questi sarebbe il sostituire la lockFile dell API con una lockFiles, che andrebbe come ora a unlockare automaticamente file precedentemente lockati, ma permetterebbe comunque di ottenere una lock su più file in sicurezza. Infatti, sapendo l'intera richiesta delle risorse future di un client, si può implementare un algoritmo del banchiere, o, più semplicemente, ordinare le operazioni di lock in base al nome del File bersaglio.
Inoltre, per quanto l'API e il server supportino un operazione di append, questa non viene mai chiamata dal client. Si potrebbe certamente fare uso di più comandi lato client, tra cui un modo di stampare una lista dei file presenti, e la capacità di fare un append a un file già esistente.

5 Github e codice di terze parti

5.1 Github

Tutto il codice dell progetto, e la relazione, sono presenti sul github pubblico `https://github.com/Sofnya/Progetto-SOL-2021`. Notare che molti commit appaiono erroneamente dall account "wit00", non mi è chiaro a chi appartenga. Sembra un errore nel mio client di git, in quanto questi commit con autore errato appaiono da quando sono passata a GitKraken.

5.2 Codice di terze parti

Per l'implementazione dell HashTable utilizziamo la funzione hash Murmur3 sviluppata da Austin Appleby e hostata su github al link `https://github.com/aappleby/smhasher`. Questa è rilasciata nel dominio pubblico senza copyright.
Utilizziamo inoltre la libreria zlib per la compressione dei File. Questa è copyright di Jean-loup Gailly e Mark Adler, che ne permettono l'utilizzo libero. Il progetto è trovabile all indirizzo `https://www.zlib.net/`.
Infine, per generare i nostri UUID, utilizziamo un frammento di codice preso da stackoverflow cortesia dell utente themoondotshine all indirizzo `https://stackoverflow.com/a/2182269`.