Relazione progetto FileStorage

Sara Grecu - mat 545509

Link to Github repository

Indice

1 Introduzione

FileStorage e' un servizio di storage che lavora principalmente in memoria RAM. Infatti e' costituito da un server che memorizza in memoria principale i file inviati da uno o piu' client, e li gestisce in base alle loro richieste.

2 Architettura generale

L'architettura generale del sistema e' basata sul paradigma richiesta-risposta: i client interagiscono con il server tramite richieste inviate su una socket di tipo AF_UNIX nota. Il **protocollo di comunicazione** prevede:

- che il client invii al server, per ciascuna richiesta:
  - La size della stringa contenente la richiesta effettiva
  - La richiesta effettiva in formato *"tiporichiesta;pathdelfilesucuioperare"*
  - La size del buffer da inviare
  - Il buffer non vuoto nel caso di operazioni di scrittura (writeFile, appendToFile), o vuoto se non necessario
- che il server invii in risposta al client:
  - Il codice d'errore che indica l'esito della richiesta
  - Nel caso dell'operazione readFile: size del file letto, e l'effettivo file
  - Nel caso dell'operazione readNFiles: size del path del file letto, path effettivo, size del file, e il file effettivo
  - Nel caso di operazioni di write:
    - Se la scrittura causa l'espulsione di altri file: size del path del file eliminato, path effettivo, size del file, e il file effettivo

3 Client

Per prima cosa, vengono inseriti tramite riga di comando tutti i comandi che l'utente vuole mandare al server (e.g. .cl w ./text_directory/prova1.txt). Se necessario, per conoscere la sintassi dei comandi da usare e' sufficiente includere *-h* nella riga di comando, e verranno stampati a video tutti comandi e i relativi parametri da inserire. Quando il client viene avviato chiama la funzione dispatcher(int argc, char* argv[]), che tramite getopt() effettua il parsing della riga di comando, e in base ai comandi inseriti chiama le funzioni specificate nell'API.

3.1 Comunicazione tra client e server

L'API per la comunicazione tra client e server comprende i file specificati nella libreria dinamica libclient.so. Questa include l'implementazione di operazioni ausiliari (contenute nei file client_utils.c, utils.c e socketIO.c) e delle operazioni richieste, dove ciascuna prepara la richiesta da inviare al server, la invia e attende la risposta. Vediamo le piu' **particolari**:

3.1.1 Funzione openFile

In base ai flag passati come parametro, prepara una richiesta diversa:

- Crea il file tramite il flag O_CREATE (O_CREATE == 2)
- Apri il file (flag == 0)
- Crea il file, aprilo e acquisisci la lock, tramite i flag O_CREATE | O_LOCK (O_CREATE | O_LOCK == 6)
- Apri il file e acquisisci la lock, tramite il flag O_LOCK (O_LOCK == 4)

3.1.2 Le funzioni writeFile e appendToFile

Dopo aver inviato la richiesta, attendono il codice d'errore: se e' 909 e' stato liberato dello spazio per poter inserire il buffer specificato, per cui la funzione dovra' reinviare la richiesta e il rispettivo buffer al server finche' questo non gli comunichera' il successo dell'operazione.
Se e' arrivato il codice 909, e il parametro dirname non e' nullo, il file espulso dal server viene memorizzato all'interno della directory specificata.
Nel server, viene controllato tramite una maschera di bit, che l'operazione immediatamente precedente alla writeFile, per ciascun client, sia stata la openFile(O_CREATE | O_LOCK); se non lo e' stata, viene restituito un codice di errore.

3.1.3 La funzione readNFiles

Sappiamo che la readNFiles prende come parametro un intero N, questo indica quanti file leggere nel caso in cui sia maggiore di 0, altrimenti che li deve leggere tutti. Nel caso in cui N<=0 vi e' un while che verra' interrotto solo alla ricezione di un codice di errore (111), altrimenti se N>0 e' stato usato un for, che si interrompe se riceve il codice 111. Questa funzione invia N richieste separate di lettura, una dopo l'altra, con la sintassi *readN;i*, con i che varia da 0 ad N o semplicemente maggiore o uguale a 0.

3.1.4 Note

Al termine di ogni operazione, se il flag -p e' pari ad 1 vengono stampati eventuali byte scritti, byte letti e se la richiesta e' stata eseguita con successo; se non fosse cosi', viene stampato un messaggio d'errore (i dettagli sono riportati nell'ultima sezione).

4 Server

Il server si occupa di memorizzare in memoria principale i file inviati dai client e li gestisce in base alle loro richieste. Per prima cosa, effettua il parsing del file di configurazione, di modo tale da inizializzare le variabili relative al suo funzonamento: il **numero di worker** che devono essere presenti nel server durante la sua esecuzione, il **numero massimo di file** ammissibili nel server, la **massima size** che il file storage puo' avere, e il **nome della socket** a cui collegarsi. Quindi, inizializza le strutture dati necessarie al suo funzionamento,

crea e avvia il thread gestore dei segnali, e poi avvia il selector non solo per ascoltare e accettare connessioni dai client, ma anche per ascoltare eventuali messaggi da due **pipe** create in precedenza: **una** utilizzata dal thread gestore dei segnali per inviare l'avvenuta ricezione di un segnale al thread main, e l' **altra** usata dagli worker per inviare al thread main l'elaborazione della richiesta inviata precedentemente.

4.1 Le strutture dati

Il Server utilizza piu' strutture dati per creare il file storage:

- Una tabella hash per memorizzare i file
- Una lista FIFO condivisa tra il thread main e gli worker, da cui questi prelevano le richieste da svolgere
- Una lista FIFO condivisa in tutto il server che memorizza l'ordine di memorizzazione di ciascun file, cosi' che in fase di esecuzione dell'algoritmo di sostituzione, venga prelevato l'elemento in testa (per semplicita', in seguito verra' chiamata coda cache)
- Una lista FIFO associata ad ogni nodo, dove sono memorizzati i file descriptor in attesa di effettuare un'operazione su di esso
- Due pipe, entrambe usate per comunicare con il main: la prima e' usata del gestore dei segnali per comunicargli la ricezione di un segnale, la seconda per comunicargli la risposta da mandare ad un client.

4.2 Funzionamento dell'algoritmo di rimpiazzamento

Nel caso in cui l'inserimento di un file, oppure la scrittura/append su un file va a superare le soglie imposte dalla specifica, e' previsto che l'algoritmo di rimpiazzamento prelevi il pathname del file in testa alla coda cache, e chiami la deletes. Quest'ultima non elimina davvero il nodo, ma inserisce quello prelevato in uno passato come parametro, che verra' poi eliminato dallo worker dopo aver inviato la risposta al client. E' stata presa questa decisione di modo tale da riutilizzare il codice della deletes sia nell'algoritmo di rimpiazzamento, che per implementare la rimozione di un nodo dalla tabella hash.
Nel caso specifico in cui l'aggiunta di un nodo, causi un sovrannumero dei file nella struttura, viene chiamato l'algoritmo di rimpiazzamento, ma non e' prevista nessuna directory in cui memorizzare i file eliminati.

4.3 Sincronizzazione con le pthread_mutex

La sincronizzazione con le pthread_mutex viene gestita a piu' livelli di granularita':

- Sul singolo nodo, nel caso di operazioni di lettura e scrittura
- Sulla lista di trabocco a cui appartiene il nodo nel caso di operazioni di aggiunta e/o di eliminazione del nodo
- Sulla coda cache
- Sulla lista FIFO delle richieste

4.3.1 La coda richieste

La coda richieste condivisa tra il thread main e gli worker prevede una mutex e una variabile di condizione, su cui gli worker si mettono in attesa nel caso questa sia vuota. Non e' previsto che si riempia, perche' viene trattata come se fosse di grandezza infinita.

4.3.2 I nodi

Su ciascun nodo viene utilizzata una pthread_mutex nel caso di operazioni di aggiunta e scrittura, ma se il client non detiene la **lock logica** sul nodo, allora viene rilasciata la mutex e restituito un errore al client.

4.4 Sincronizzazione logica

Su ciascun nodo, oltre che una pthread_mutex viene utilizzata una **lock logica**, acquisita dai client se l'operazione di lockFile termina con successo, altrimenti vengono messi in attesa (e cioe', inseriti in coda alla lista di attesa del nodo). Nelle altre operazioni (di read, di write, ecc), se il client non detiene la lock sul nodo, gli viene restituito un codice di errore (202).
I thread vengono risvegliati, e gli viene data la rispettiva lock, in ordine FIFO quando viene chiamata la unlock.
Quando un nodo viene eliminato, a ciascun client in attesa viene restituito un codice di errore che gli indica che non possono piu' operare su quel file.

4.5 Script bash

Per la creazione del file di configurazione e' stato implementato uno script bash: create_config.sh, e per la creazione del file statistiche.sh e' stato usato uno script awk: rules.awk, le cui regole sono definite in base alle stampe che vengono fatte sul file log_file.txt ad ogni operazione.

5 Gestione Errori

Per ogni errore che si verifica sono stati creati degli errori ad hoc, che quando arrivano al client vengono stampati a video tramite la macro presente nel file check_errors.h. Ecco l'elenco degli errori:

- 202 nel caso in cui la lock sia stata acquisita da un altro client
- 303 nel caso in cui si tenti di fare un' operazione se il file non e' aperto
- 404 nel caso in cui si tenti di fare un'operazione se il file non esiste e il flag O_CREATE non e' stato specificato
- 505 nel caso in cui il file non esista
- 707 nel caso in cui si tenti di fare la appendToFile prima della writeFile
- 111 nel caso in cui non ci siano piu' file da leggere
- 444 nel caso in cui il file che si vuole scrivere/di cui si vuole fare la append nel server sia troppo grande

- 555 nel caso in cui la lock non fosse stata acquisita
- 909 nel caso in cui fosse stato liberato dello spazio per effettuare una write o una append
- 999 nel caso in cui il nodo su cui il client voleva effettuare delle operazioni fosse stato eliminato a causa di una delete() o perche' il server e' terminato
- 606 nel caso in cui il client abbia cercato di fare la writeFile senza aver fatto prima la openFile(path, O_CREATE | O_LOCK)
- -1 nel caso di errore generico

6 File di test

I file di test a cui si fa riferimento in test1, test2, test3 sono contenuti nelle directory test_dir1 e test_dir2; queste includono file binari, immagini e file testuali. I file binari sono stati ricavati dal comando:
head -c num_byte </dev/urandom > myfile.txt
Per l'esecuzione della make clean viene usato lo script ./script/cleaner.sh.