UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea in Informatica

File Storage Server

Sistemi Operativi e Laboratorio

Prof. Alessio Conte
Prof. Maurizio A. Bonuccelli

Luca Cirillo
545480 (A)

Anno Accademico 2020/2021

1. Introduzione

FSS

File Storage Server, abbreviato in *FSS*, è uno strumento che offre uno spazio di memorizzazione condiviso, authentication-free e collaborativo, ispirato ai moderni servizi di cloud storage, ma più semplice. Un server centrale si occupa della gestione dello spazio, mentre più client possono sfruttare le APIs messe a disposizione per interagire con esso, per caricare nuovi file, leggere e modificare quelli esistenti, e così via. Lo storage non prevede un meccanismo di autenticazione e i file caricati al suo interno, identificati univocamente dal loro *path*, sono disponibili a tutti i client che desiderano accedervi. Lo spazio destinato alla memorizzazione dei file è gestito come una *cache*: il caricamento di un file di dimensioni superiori rispetto allo spazio disponibile in un certo istante comporta l'espulsione di uno o più file, al fine di liberare lo spazio necessario per memorizzarlo. Naturalmente, la dimensione del file non può essere maggiore della capienza massima dello storage. L'*algoritmo di rimpiazzo* si occuperà di selezionare questi file *vittima* secondo la politica scelta, come descritto in seguito. Il codice, i test e i file di configurazione si trovano all'interno di una repository pubblica sul servizio di hosting *GitHub*, raggiungibile a questo indirizzo.

Dettagli sul metodo di sviluppo

Il progetto è stato sviluppato con il supporto dell'IDE *Visual Studio Code*, testato con l'ausilio di strumenti quali *GDB* e *Valgrind*, e ottimizzato per la compilazione e l'esecuzione sulla macchina virtuale *XUbuntu* fornita durante il corso. L'utilizzo del version control *Git* ha concesso il beneficio di poter sviluppare con tranquillità e maggiore controllo sul codice, grazie alla possibilità di spostarsi in branches separati in cui portare avanti le singole features, e tornare facilmente indietro in caso di problemi. Il codice è stato **ampiamente commentato** per garantirne una comprensione a 360°, ma comunque scritto per essere quanto più possibile "*self explanatory*", con nomi di variabili, metodi e statements chiari e significativi. Questi ultimi sono stati scritti in lingua Inglese, mentre i commenti interamente in Italiano. I commenti, inoltre, sono stati scritti utilizzando lo stile fornito dall'estensione Better Comments, per Visual Studio Code, che permette di categorizzarli e fornire loro un meccanismo di "*highlighting*".

1 Dalla compilazione all'esecuzione

Per compilare ed eseguire correttamente il sistema FSS è necessario creare due eseguibili separati, uno per il **client** ed uno per il **server**. Questa operazione è resa immediata grazie all'utilizzo di un Makefile, con regole specifiche per ogni componente del progetto. La cartella *build* ospiterà gli eseguibili, i file oggetto generati durante la compilazione, e tutto il necessario per eseguire i test, contenuti nell'omonima sottocartella. Si riportano alcuni utilizzi del Makefile fornito.

```
# Supponendo di essere gia' nella directory del progetto, si creano gli eseguibili
# La regola di default per il comando make compila sia client che server
make
```

```
# Oppure, si possono compilare i sorgenti separatamente
make client
make server
# Il comando clean elimina i residui del processo di compilazione
make clean
# Il comando cleanall elimina anche i residui dei test
make cleanall
# Infine, e' possibile lanciare i tre test forniti
make test1
make test2
make test3
```

Dettagli implementativi

FSS è implementato seguendo il paradigma **client-server**, con comunicazione di tipo **richiesta-risposta**. E' permessa quindi una sola istanza del server ma molteplici istanze dei client.

1 Server

1.1 Configurazione

La configurazione del server viene fatta attraverso un file di testo, che viene opportunamente parsato, e tramite il quale è possibile specificare alcuni parametri, quali la capienza massima dello storage, il numero di threads worker, eccetera. Nella cartella *config* è possibile trovare un template del file di configurazione, con la spiegazione di ogni possibile parametro, ed un file di esempio pronto per essere utilizzato. La struttura del file è molto semplice e flessibile: ogni riga indica un parametro, specificato da una coppia *chiave=valore*, e sono ammesse righe vuote e commenti, preceduti dal simbolo '#'.

```
# Configurazione File Storage Server

# Numero di threads worker
THREADS_WORKER=<int>

# Dimensione massima dello Storage, in Mb
STORAGE_MAX_CAPACITY=<int>
# Numero massimo di file consentiti
STORAGE_MAX_FILES=<int>
# Politica di rimpiazzamento
REPLACEMENT_POLICY=<fifo|lru|lfu>

# Path al Socket file
SOCKET_PATH=<path>
# Path al Log file
LOG_PATH=<path>
```

1.2 Threads

Come previsto dalla specifica, il server è un **singolo processo multi-threaded**, che segue lo schema **master-worker**. Il thread *main* funge anche da **dispatcher** delle richieste in entrata, memorizzate in una coda thread-safe e condivisa tra tutti i workers. Il numero di questi threads è fissato all'avvio tramite il file di configurazione, e si occupano di processare le richieste inviate dai client. Un ulteriore thread di supporto gestisce invece l'arrivo dei segnali, in particolare di *SIGINT, SIGQUIT* e *SIGHUP*, che permettono l'arresto in sicurezza del server.

1.3 File

Lo storage memorizza e indicizza i file grazie ad una **hashmap**, utilizzando come chiave il path del file, che lo identifica univocamente all'interno del sistema. Ogni file possiede tre informazioni principali:

- **Name**: indica il nome del file, e include il suo path;
- **Contents**: lo storage memorizza qui il contenuto del file;
- **Size**: intuitivamente, la dimensione del contenuto del file.

Oltre a queste informazioni, un file presenta ulteriori *metadati*, utilizzati dall'algoritmo di rimpiazzo in base alla politica scelta:

- **Creation time**: l'istante di tempo in cui il file è stato creato;
- **Last use time**: il tempo dell'ultimo utilizzo del file;
- **Frequency**: il numero di accessi unici al file.

I file vengono creati, scritti, letti, aggiornati e rimossi solamente dai client, sfruttando le API esposte dal server. Tuttavia, prima di effettuare una qualsiasi operazione su un file, è necessario **aprirlo**. Un file può essere aperto in due modalità: **lettura** e **scrittura**. Lato server, l'accesso in lettura è gestita con una *linked list* di interi, che identificano i client che ne hanno richiesto l'apertura, mentre l'accesso in scrittura, essendo esclusivo, è memorizzato tramite un semplice intero.

La preferenza sulla modalità di apertura di un file è espressa dal client attraverso l'utilizzo dei seguenti *flags*:

- **O_CREATE**: crea un nuovo file;
- **O_READ**: apre un file in sola lettura;
- **O_LOCK**: apre un file in scrittura (e in lettura).

I flags possono essere combinati in OR bit-a-bit, per specificare ad esempio la volontà di creare un nuovo file e contestualmente di aprirlo in scrittura, con *O_CREATE | O_LOCK*. Inoltre, devono essere rispettati i seguenti vincoli: un file non ancora presente sul server richiede la presenza del flag *O_CREATE*, che creerà un nuovo file e lo aprirà automaticamente in lettura per il client (oppure in scrittura, se anche *O_LOCK* è stato specificato); se invece il file esiste già, questo può essere aperto in lettura con il flag *O_READ* o in scrittura con il flag *O_LOCK*, ricordando che la modalità in scrittura apre implicitamente il file anche in lettura. Un file già aperto in lettura deve essere aperto in scrittura tramite la API *lockFile*.

1.4 Algoritmo di rimpiazzo

Come già accennato, lo spazio di memorizzazione è gestito come una cache di file: se viene richiesto il caricamento di un file la cui dimensione supera lo spazio disponibile in quel momento, uno o più file vengono selezionati, espulsi e inviati al client. L'algoritmo di rimpiazzo quindi entra in gioco unicamente in contesti in cui lo spazio disponibile è limitato. La politica con cui i file, chiamati *vittime*, vengono selezionati per la rimozione viene specificata tramite il file di configurazione, e può influenzare drasticamente le performance del sistema in base al workload da elaborare. Sono state implementate tre politiche differenti:

- **FIFO**: vengono espulsi i file presenti nello storage da più tempo, in base al *creation time*;
- **LRU**: vengono espulsi i file utilizzati meno recentemente, in base al *last use time*;
- **LFU**: vengono espulsi i file utilizzati meno frequentemente, in base a *frequency*.

I file espulsi dal server vengono inviati al client la cui richiesta ha reso necessaria l'esecuzione dell'algoritmo.

1.5 Logging

Il server effettua il log di ogni operazione, sia essa di gestione interna o di comunicazione con i client, in un apposito file, il cui percorso è specificato nel file di configurazione. Il formato del file di log è stato scelto per essere quanto più descrittivo possibile, includendo **data e ora**, ***livello*** dell'informazione riportata, che può essere uno tra *INFO*, *DEBUG*, *WARN* ed *ERROR*, **identificativo** del thread che ha gestito l'operazione, e una **descrizione** di quest'ultima, che include anche eventuali parametri, codici di errore ed esito della richiesta.

```
25-10-2021 15:07:39 | INFO | Server bootstrap
25-10-2021 15:07:40 | INFO | [000000000] CLIENT: 7 connected
25-10-2021 15:07:40 | INFO | [109164288] OPEN: ./build/dummy-1 3 => 0
25-10-2021 15:07:40 | INFO | [109164288] WRITE: ./build/dummy-1 798720 bytes => 0
25-10-2021 15:07:40 | INFO | [109164288] CLOSE: ./build/dummy-1 => 0
25-10-2021 15:07:40 | INFO | [109164288] OPEN: ./build/dummy-2 3 => 0
25-10-2021 15:07:40 | INFO | [109164288] WRITE: ./build/dummy-2 1761280 bytes => 0
25-10-2021 15:07:40 | INFO | [109164288] CLOSE: ./build/dummy-2 => 0
25-10-2021 15:07:40 | INFO | [109164288] OPEN: ./build/dummy-3 3 => 0
25-10-2021 15:07:40 | INFO | [109164288] WRITE: ./build/dummy-3 901120 bytes => 0
25-10-2021 15:07:40 | INFO | [109164288] CLOSE: ./build/dummy-3 => 0
25-10-2021 15:07:42 | INFO | [109164288] CLIENT: 7 has left
25-10-2021 15:07:43 | WARN | SIGHUP received
25-10-2021 15:07:43 | INFO | Server shutdown
```

Inoltre, allo spegnimento del server, vengono visualizzare una serie di informazioni e statistiche sullo storage: tempo di attività, numero massimo di file memorizzato, dimensione massima raggiunta, numero di esecuzioni dell'algoritmo di rimpiazzo, file ancora presenti all'interno dello storage.

2 Client

2.1 Parametri

L'interfaccia del client offre una serie di parametri da riga di comando che permettono di interagire con il server. Innanzitutto, il comando *-h* permette di visualizzare un elenco dei parametri accettati e una loro breve descrizione, e viene qui riportato:

- **-f**: specifica il nome del socket utilizzato dal server;
- **-p**: abilita la modalità *verbose*;
- **-w**: scrive sul server tutti i file contenuti in una cartella, o fino al raggiungimento di un limite superiore al numero di file da inviare;
- **-W**: scrive sul server una lista di file, eventualmente sovrascrivendone il contenuto;
- **-D**: indica la cartella in cui salvare eventuali file espulsi durante le operazioni di scrittura (opzionale);
- **-r**: legge dal server una lista di file;
- **-R**: legge dal server un numero specifico di file, oppure tutti quelli presenti;
- **-d**: indica la cartella in cui salvare i file letti durante le operazioni di lettura (opzionale);
- **-l**: richiede l'accesso in scrittura di una lista di file;
- **-u**: rilascia l'accesso in scrittura di una lista di file;
- **-c**: cancella dal server una lista di file;
- **-t**: indica il tempo di attesa tra una richiesta e l'altra.

I parametri vengono inizialmente parsati per controllarne la correttezza; superata questa fase, le richieste vengono aggiunte in una coda con politica FIFO, dalla quale verranno poi estratte per essere elaborate.

3 API

Le APIs esposte dal server e implementate dal client seguono tutte la stessa struttura: ritornano un intero maggiore o uguale a 0 in caso di successo, un intero negativo in caso di fallimento e settano la variabile *errno*. Ogni API effettua i controlli necessari sulla validità dei parametri forniti, sia lato client che lato server.

La coppia **openConnection** e **closeConnection** permette al client di instaurare una connessione al server. L'utilizzo di queste due API è implicito, ovvero il client apre una connessione verso il socket specificato una volta terminato il parsing dei parametri, e la chiude dopo aver elaborato tutte le richieste in coda. Le operazioni di lettura e scrittura richiedono l'utilizzo delle API **openFile** e **closeFile**, rispettivamente per aprire e chiudere i file. L'apertura di un file può avvenire secondo diverse modalità: operazioni quali *-r* e *-R*, ovvero le API **readFile** e **readNFiles**, aprono il file in sola lettura, permettendo l'accesso concorrente ad altri client che desiderano leggere lo stesso file. Operazioni di scrittura quali *-w*, *-W* e *-c*, ovvero **writeFile** e **removeFile** aprono il file in scrittura, richiedendo così un accesso mutualmente esclusivo allo stesso, per portare a termine le suddette operazioni. Un file può essere aperto in lettura, e successivamente richiedere l'estensione alla modalità in scrittura attraverso la API **lockFile**, da revocare successivamente con **unlockFile** (oppure, implicitamente, chiudendo il file).

4 Tests

All'interno della cartella *build/tests/* sono presenti tre script Bash, che testano singolarmente diverse componenti del sistema. Ai client verrà un fornito un set di *dummy files* da utilizzare per interagire con il server, generati on-demand dallo script in dimensione e numero fissati per il tipo di test che si vuole

eseguire. Infine, un ulteriore script Bash permette di analizzare il file di log del server per generare alcune statistiche rilevanti.

4.1 Test n.1

L'obiettivo del primo test è verificare che non vi siano *memory leaks* da parte del server al momento della sua terminazione, e avvia quest'ultimo utilizzando *Valgrind* con il parametro *–leak-check=full*. Le istanze dei client avviati dallo script testano tutte le API disponibili. Il test si intende superato se, al termine, il numero di *malloc* e il numero di *free* coincidono.

4.2 Test n.2

L'obiettivo del secondo test è verificare il funzionamento dell'algoritmo di rimpiazzo. Il server viene avviato, questa volta senza Valgrind, con parametri molto stretti: una dimensione dello storage di appena 1 MBytes e un massimo di 10 file, proprio per mandare in esecuzione molte volte l'algoritmo. Una serie di client vengono eseguiti in parallelo ed eseguono solamente scritture sul server. Il test si intende superato se il numero di volte in cui l'algoritmo di rimpiazzo è stato eseguito risulta ragionevole.

4.3 Test n.3

L'obiettivo del terzo test è verificare la resilienza del server. Si configura di fatto come uno *stress-test*: vengono eseguiti ininterrottamente e per un tempo totale di trenta secondi, molteplici istanze di client che scrivono, leggono ed effettuano altre operazioni sul server, avendo a disposizione un set di 100 dummy files, di dimensione che varia da 20 Kbytes a 2Mbytes. Il test si intende superato se il server non si chiude inaspettatamente e se, al termine, le informazioni e statistiche visualizzate risultano ragionevoli.

4.4 Statistiche

Lo script Bash denominato *statistiche.sh* accetta in input il path di un file di log generato dal server. Questo verrà opportunamente parsato per analizzare il comportamente del server durante la sua esecuzione e riportare alcune statistiche, quali il numero di operazioni di lettura e scrittura, il picco di file e capacità raggiunto dal server, il massimo numero di connessioni simultanee e così via. Si riporta qui un esempio di output:

```
Number of reads: 70, average bytes read: 1213001 bytes
Number of writes: 101, average bytes written: 2682068 bytes
Number of locks: 0
Number of unlocks: 0
Number of open-locks: 101
Number of closes: 101
Maximum number of files reached by the storage: 70
Maximum size reached by the storage: 127 MBytes
Number of runs of the replacement algorithm: 16
Maximum number of concurrent connections: 1
```