File Storage Server

Progetto di Laboratorio di Sistemi Operativi - a.a. 2020/2021

Giannini Sara

Corso B

Mat. 561119

Indice

Repository Github : https://github.com/SaraGiannini/FileStorageServer

1. Server

Il Server viene definito come un processo multithreaded.

Il thread main si occupa della configurazione iniziale, dell'avvio degli altri thread e dell' ascolto con la select dei file descrittori pronti.
Da tale thread vengono quindi avviati un thread Signal per la gestione dei segnali sincrona per la terminazione del Server e i thread Worker per la gestione delle richieste da Client connessi, la cui descrizione è rimandata al paragrafo 5 sulla comunicazione.

All'avvio del Server si configura il sistema effettuando il parsing di un file, passato da linea di comando con l'opzione *-f*, per ottenere le informazioni relative al File Storage Server da inizializzare, il numero di thread Worker da avviare e il socket su cui accettare i Client.
Il file di configurazione passato è un file testuale formattato in modo che vengano ignorate le righe vuote e le righe che iniziano con '#', che sono commenti utili per la comprensione del file. Le righe considerate vengono viste come coppie '*chiave : valore*' e da queste si crea una struttura sul server utilizzata per inizializzare lo stato del sistema.
Il file per la configurazione è nominato "*config.txt*" e si trova nella directory corrente.
I file di configurazione per i test sono nominati "*configt*.txt*" e si trovano nella directory test.
La configurazione viene implementata con 'configserver.c'.

Durante la sua esecuzione, il server effettua il logging su un file con estensione .log contenente testo formattato in modo che ogni riga inizia con il timestamp dell'evento seguito da una descrizione dell'evento registrato.
Per gli eventi con cui si deve effettuare la statistica finale, si aggiunge una riga che inizia con il timestamp seguito da una parola chiave tra parentesi [ ] eventualmente seguita da alcune informazioni per i calcoli di statistica.
Sul file di log può scrivere sia il thread main del Server che i thread Worker, perciò la scrittura sul file è regolata con una mutex.
Il file creato in cui viene registrato il log di esecuzione viene nominato "*logging.log*" o "*loggingt*.log*" per i test con indicazione al test effettuato.
Il file, in ogni caso, viene creato nella directory corrente del progetto.
Il logging di esecuzione viene implementato con 'loggingserver.c'.
Le statistiche finali vengono elaborate con lo script 'script/statistiche.sh' in cui viene parsato il file di log, passatogli come argomento, e con l'utilizzo di grep si prelevano le righe in cui è presente la parola chiave racchiusa da parentesi [ ].

La gestione dei segnali per la terminazione del Server, come accennato sopra, è stata implementata con l'utilizzo di un thread Singal dedicato alla gestione sincrona con la sigwait.
Per la gestione dei segnali viene creata una struttura da passare al thread Signal in cui sono presenti due flag, oltre alla maschera dei segnali e alla entry di scrittura della pipe di comunicazione con il thread main.
Il flag relativo a SIGINT e SIGQUIT è per la terminazione immediata, e quello relativo a SIGHUP è per fermare le richieste in arrivo. Poichè sia il thread Signal che il thread main accedono a tali flag, in lettura e scrittura, nella struttura sopra citata è presente anche una mutex.
La funzione del thread Signal e le funzioni ausiliarie per settare/resettare i flag e per saperne il valore sono implementate in 'signalhserver.c'.

2. Strutture lato Server per definire il File Storage Server

Per rappresentare il FileStorage Server viene definita una struttura filestorage_t che, oltre ai campi relativi alla capacità, in termini di dimensione e di numero file, e alla politica di rimpiazzamento, ingloba due strutture per memorizzare i file:

- una tavola hash dei file implementata con 'icl_hash.c', fornita durante il corso, per effettuare in modo più efficiente le ricerche del file oggetto di una richiesta.
  La ricerca avviene utilizzando come chiave il path assoluto del file.

Si assume che il fattore di carico α della tavola hash, dato dal rapporto tra il numero massimo di file del FS e il numero di buckets della tavola, sia inferiore a 0.90 e per ottenere un buon compromesso tra costo in tempo e costo in spazio si prende α = 0.75.

- una lista dei file di cui si ha il riferimento alla testa e il riferimento alla coda, per effettuare in modo più efficiente l'espulsione di un file con politica FIFO, poichè l'inserimento del file avviene in coda alla lista seguendo l' ordinamento FIFO. Per le altre politiche di rimpiazzamento è invece necessario scorrere tutta la lista facendo riferimento ai cami opportuni.

L'accesso a tutto il FileStorage Server in mutua esclusione da parte dei thread Worker del server viene regolato con una mutex.

I file a cui si fa riferimento nelle strutture interne al FileStorage Server seguono la struttura file_t che, oltre alle informazioni del file quali path assoluto, contenuto e dimensione, contiene i campi necessari alla gestione del file definiti di seguito.
Sono presenti due campi, referencetime e referencecount, per effettuare il rimpiazzamento con le politiche LRU e LFU utilizzando rispettivamente un clock_t per il tempo in numero di clock trascorsi dall'avvio del programma e un intero come contatore, questi vengono inizializzati con l'allocazione del file e aggiornati ad ogni riferimento.
Sono presenti altri due campi, ocreate e olock, che memorizzano il Client che ha effettuato la creazione del file e il Client che ha richiesto la Mutua Esclusione sul file.

1. ocreate viene settato su richiesta del Client alla *openFile*-creazione, con valore fd (file descriptor) del richiedente per indicare che è colui che può effettuare subito la prima scrittura sul file con *writeFile*, altrimenti -1 quando si effettua un'altra operazione sul file e non si può più fare la *writeFile.*
2. olock viene settato su richiesta del Client alla *openFile(O_LOCK)* o con *lockFile*, con valore fd del richiedente dell'operazione per indicare che è colui che detiene la Mutua Esclusione sul file, altrimenti -1 quando è "uncloked". Viene modificato con *unlockFile, closeFile e closeConnection* con cui si va a prendere il primo Client in attesa di fare la *lockFile*, e se non presente ha valore -1.
   Di questo campo si tiene conto ad ogni operazione, esclusa *lockFile*, poichè se un Client effettua una richiesta su un file con olock settato da un altro Client, tale richiesta viene terminata con errore, perciò per poter effettuare l'operazione il file deve essere "unlocked" oppure olock settato dal Client stesso.
   Quando un Client effettua una richiesta di *lockFile* su un file con olock settato da un altro Client, tale richiesta viene sospesa inserendo il Client richiedente nella lista di attesa. Da tale lista il Client verrà rimosso come descritto sopra per poter concludere la richiesta che era stata sospesa.
   La richiesta, che era stata sospesa e poi ripresa come appena descritto, può concludersi con esito positivo, acquisendo la Mutua Esclusione, oppure con esito negativo quando il file per cui si stava aspettando è stato eliminato, per capacity misses del FileStorage Server o per volontà di un altro Client.

Per contenere i Client in attesa di fare la *lockFile*, viene implementata una lista di fd che rappresenta i Client che hanno fatto la richiesta e sono stati sospesi.
Inoltre viene implementata un'altra lista di fd che contiene i Client che hanno aperto il file.
L'accesso al file in mutua esclusione da parte dei thread Worker del server viene regolato da una mutex e da una variabile di condizione con l'utilizzo di un contatore dei lettori.

I Client a cui si fa riferimento nelle due liste sopra citate sono rappresentati dalla struttura fd_t costituita dall'intero per il fd relativo alla connessione con il Client e dal puntatore al prossimo.

L'implementazione della gestione del FileStorage Server è contenuta in 'filestorage.c'.

3. Client

Il Client è definito come un processo costituito da un unico thread.

All'avvio si effettua il parsing da linea di comando andando a creare due liste di '*opzione - argomento*'.
Una prima lista utilizzata per la configurazione iniziale del Client da cui si ottiene il socket di connessione con il Server, l'abilitazione alla stampa, il tempo di attesa per l'invio di due richieste successive, e la richiesta

di aiuto per la stampa delle richieste accettabili.
La seconda lista è quella dedicata alle richieste da inviare al Server una alla volta tramite l'utilizzo della libreria di API.
L'implementazione del parsing della linea di comando è definita con 'parsingclient.c'.

Prima di passare alla libreria API, le richieste vengono elaborate. Ovvero per quelle che presentano più argomenti separati da ',', si dividono tali argomenti definendo un array; se gli argomenti sono tutti file verrà effettuata una richiesta separata per ogni file dell'array.
Per queste richieste gli argomenti non devono presentare spazi prima e dopo la virgola *(-W file1,file2)*.
Per le richieste il cui argomento è opzionale, '*-R*' e '*-t*', l'opzione può essere seguita dall'argomento con o senza lo spazio *(-R1 oppure -R 1)*.

I file, come indicato precedentemente, vengono identificati con il loro path assoluto. Per le operazioni di scrittura con '*-w*' e '*-W*' il path assoluto viene definito con la funzione 'realpath' prima di passare la richiesta alla funzione della API, per tutte le altre richieste il file viene passato da linea di comando direttamente con il path assoluto.

4. API

Nella libreria API, implementata con 'api.c', sono presenti le funzioni per gestire le richieste lato Client da mandare al Server e da cui ricevere le risposte.
Inoltre, sono presenti le funzioni necessarie alla ricezione e al salvataggio dei file, letti o espulsi, nella directory, se indicata da linea di comando.
La ricezione dei file utilizza la struttura 'sreturnfile_t' contenente le lunghezze del path e del contenuto del file ritornato dal server.

Nelle funzioni per interagire con il FileStorage Server, le richieste vengono prodotte utilizzando la struttura 'crequest_t' contenente l'operazione richiesta, i possibili flag, la lunghezza del path e la lunghezza del contenuto del file da inviare al server.

5. Comunicazione Server - Client

Il Server viene implementato come richiesto seguendo lo schema Master - Worker in cui i thread Worker sono implementati con un array di thread la cui funzione da eseguire gestisce una richiesta letta dal Client.
La comunicazione da Master a Worker è gestita tramite una coda condivisa a capacità limitata implementata thread safe con 'boundedqueue.c', per contenere i fd dei Client connessi sulla socket del Server che dovranno essere gestiti dai thread Worker.
La comunicazione da Worker a Master avviene su una pipe la cui entry di scrittura viene passata al thread Worker dal thread Manager all'avvio del Worker. Con tale pipe il thread Worker comunica al Master il fd del Client la cui richiesta è stata gestita e può essere reinserito nella coda condivisa per effettuare nuove richieste.

Il protocollo di comunicazione richiesta-risposta utilizza le due strutture sopra citate, 'crequest_t' per le richieste del Client e 'sreturnfile_t' per i file ritornati dal Server, definite in 'conn.h'.

Il Client invia una richiesta dalla funzione di API seguendo questo schema:

- invia un messaggio con la struttura 'crequest_t' contenente: il codice dell'operazione, la lunghezza del path (non per *-R*), la lunghezza del contenuto del file per le operazioni di scrittura (*-W, -w, -a*), i flags per le operazioni *openFile* (O_CREATE | O_LOCK) e *readNFiles*(N).
- invia un messaggio contenente il path assoluto del file oggetto di richiesta, per tutte tranne *-R.*
- invia un messaggio contenente il contenuto del file oggetto di richiesta, per *-W, -w, -a*.

Simmetricamente, il thread Worker del Server riceve i messaggi nello stesso ordine: legge la richiesta con la stessa struttura del client, poi per tutte le operazioni esclusa la *readNFiles,* legge il path assoluto del file e se

è una scrittura *writeFile* o *appendToFile* legge il contenuto del file.

A questo punto il thread Worker con la richiesta ottenuta, e i relativi dati, effettua la gestione di tale richiesta.
In caso di successo, la risposta che il thread Worker invia al Client è diversa per ogni tipo di operazione gestita:

- *openFile* invia l'esito positivo, se è uguale a 1 indica che nell'effettuare l'operazione è stato espulso un file a causa di capacity misses e il Client stamperà un messaggio indicante il verificarsi di tale evento.
- *readFile* invia l'esito positivo (OK = 0), la lunghezza del contenuto del file letto e infine il contenuto del file.
- *readNFiles* invia il numero di file che verrano spediti e per ogni file invia: un messaggio con la struttura 'sreturnfile_t' contenente la lunghezza del path assoluto del file e la lunghezza del contenuto del file, poi invia il path assoluto e il contenuto del file.
- *writeFile* e *appendToFile* inviano l'esito positivo, se maggiore di 0 indica il numero di file espulsi a seguito di capacity misses, e per ogni file invia i messaggi come per *readNFiles,* la struttura con le lunghezze, poi il path e il contenuto del file.
- per tutte le altre operazioni invia semplicemente l'esito postivo (OK = 0).

In caso di errore durante la gestione viene inviato un codice negativo significativo dell'errore che si è verificato.
Il Client quando riceve il primo messaggio dal Server verifica se tale valore sia negativo (< OK) e in tal caso utilizzerà tale codice per stampare un messaggio di errore.
Altrimenti per esito positivo, il Client riceve, come descritto sopra, i messaggi nello stesso ordine e per ogni caso gestisce la risposta del server nel modo opportuno.

I codici delle operazioni e di errore sono definiti nel file 'conn.h'.

La funzione del thread Worker è implementata in 'workerserver.c' insieme alle funzioni per inviare i file, letti o espulsi, e per inviare il messaggio di errore.

Istruzioni Compilazione ed esecuzione

Compilare il codice con '*make*'.

Per effettuare i test eseguire '*make test**', in modo implicito verranno rimosse tutte le directory e i file prodotti da test precedenti così da avere la directory di lavoro pulita per il nuovo test.