



UNIVERSITÀ DI PISA

Progetto SOL 2020/2021

Studente: Paul Maximilian Magos

Matricola: 588669

Professore Massimo Torquati

Professore Gabriele Mencagli

27 Marzo 2022

1 Introduzione

Progetto di tipo Client-Server, Server che si comporta come file storage, interagendo con i client per soddisfare richieste quali, caricare nuovi file, leggere dei file, ottenere l'esclusiva o rilasciare l'esclusiva su di un file e cancellare file.

2 Server

Il server è suddiviso principalmente in 5 parti ognuna delle quali si occupa di implementare una certa funzionalità. La funzionalità implementata dal server stesso però è quella di verificare quali file descriptor sono pronti per essere letti, ciclando così tra gli fd da 0 al massimo fd. Tra questi oltre ai client stessi ovviamente sono presenti anche la ReadEnd della pipe del SignalHandler e le ReadEnd delle pipe dei worker. Quando un client richiede un servizio il server aggiunge il suo fd alla queue della ThreadPool e va a rimuoverla dal set di fd (nel caso in cui si deve riascoltare il client la funzione eseguita dai worker si occuperà della scrittura sulla pipe dell'fd da riascoltare, che verrà letta da parte del processo principale e verrà riaggiunta al set).

2.1 Storage Implementation

La memorizzazione dei file è implementata da un hash table, attraverso la libreria icl_hash consigliata nel corso di laboratorio, che ha una dimensione di file totale massima pari a tre volte il numero massimo di file definito dal file di configurazione (vedi 2.2).

Scelta fatta anche a favore dell'algoritmo di espulsione dei file (oltre ovviamente al fatto che le hashtable hanno bisogno di $2n$, con n numero di file, spazio per poter lavorare), che espelle i file solo successivamente all'inserimento del nuovo file creando così la possibilità di avere un overflow nel caso in cui lo storage sia esausto di posizioni. La concorrenza viene "gestita" attraverso una variabile di mutua esclusione per lo storage generale, e per i file due mutex e una variabile di condizione per regolarne l'accesso.

Procedure di accesso ai file:

1. WRITE

- a. Quando un client richiede la scrittura di un nuovo file questo viene creato se non esiste e il suo stato rimarrà fino a che non vengono ricevuti i dati relativi al file in Creato ed accessibile solo all'utente che lo ha creato. Successivamente dopo la creazione si ricevono i dati del file e si salvano nella tabella, modificando lo stato del file in scritto e da chiudere, questo per evitare che un altro client che va a scrivere sul server vada ad espellere il file. Successivamente si ha la chiusura vera e propria.

2. READ

- a. La prassi è la stessa della scrittura, con la differenza che la open si occuperà di verificare che il file esista nel server (nel caso della read normale, la readN apre i file quando seleziona il file mentre itera per definire la lista dei file leggibili), e successivamente il file viene inviato al client, e chiuso.

Queste sono le due macro categorie di comportamenti eleggibili per un worker thread, ogni operazione descritta a partire dalla open alla close utilizzano due funzioni per modificare il numero di lettori e scrittori del file. Per le scritture ovviamente verrà modificato il numero di scrittori, che non può mai essere più d'uno, e per le letture verrà modificato il numero di lettori. Il numero di lettori contemporanei non è vincolante, a differenza del numero di lettori e scrittori che devono necessariamente escludersi a vicenda:

- Readers > 0 \Rightarrow Writers = 0
- Writers > 0 \Rightarrow Readers = 0 (in realtà Writers == 1 descrive al meglio la condizione)

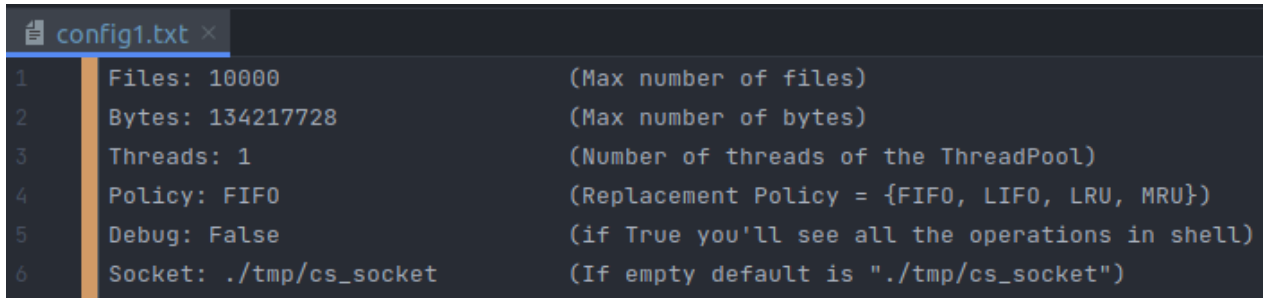
2.2 Config Parser

All'avvio tra le varie funzionalità il server come primo step inizializza le sue strutture dati di stato e di configurazione:

- La struttura di configurazione denominata *ServerConfig* contiene le configurazioni iniziali definite da un file di config, se presente in fase avvio).
- La struttura di stato denominata *ServerStorage* contiene anche la tabella dei file e tutte le informazioni relative agli accessi avvenuti

La configurazione viene opportunamente letta se passata come argomento al momento di avvio del server, in caso contrario prenderà dei valori di default

DEFAULT \Rightarrow {100 File, 15MB, 15 Thread, Policy FIFO. Debug False, Socket `“./tmp/cs_socket”`}



```
config1.txt x
1 Files: 10000 (Max number of files)
2 Bytes: 134217728 (Max number of bytes)
3 Threads: 1 (Number of threads of the ThreadPool)
4 Policy: FIFO (Replacement Policy = {FIFO, LIFO, LRU, MRU})
5 Debug: False (if True you'll see all the operations in shell)
6 Socket: ./tmp/cs_socket (If empty default is "./tmp/cs_socket")
```

Possono essere ordinati secondo il proprio volere, gli argomenti devono necessariamente essere presenti se non per la socket che se non indicata prenderà un valore di default.

2.3 Log

Come secondo step dell'inizializzazione viene inizializzata la struttura del log, che altro non è che un puntatore ad un file di testo vero e proprio ed una mutex per accedervi (Il log verrà modificato da più thread). Se non indicato assieme ad un file di config in fase di avvio del server da shell, il path del log sarà standard ovvero `./log/logs`. Per ogni avvio differente verrà creato un nuovo file di testo caratterizzato dalla data e l'ora odierna come nome.

Oltre alla funzione di creazione e distruzione del log, abbiamo la funzione denominata *appendOnLog()* che altro non fa che aggiungere una nuova riga al file con l'ora e la stringa a parametri variabili passata dal chiamante.

2.4 ThreadPool

Per la gestione di tutte le richieste da parte dei client vi è una struttura denominata *ThreadPool*, con N thread di tipo pthread, N definito dalla configurazione, una queue di task definita da due nodi (testa e coda) e un relativo contatore, una mutex e una variabile di condizione per regolare l'accesso alla queue. Inoltre è presente anche un array di array di `[N][2]`, ovvero 2 "posizioni" per N che è il numero di thread. Questo infatti rappresenta le pipe per ogni thread che vengono controllate dal ciclo principale del server. Quando un client si collega al server viene aggiunto al set di fd "da ascoltare", successivamente quando la sua richiesta viene aggiunta alla queue della *ThreadPool* il suo fd viene "eliminato" dal set.

Una volta conclusa la task per un client, verrà scritto il suo fd sulla WriteEnd della pipe del worker thread che lo ha servito, ad indicare al server che deve riascoltare quel client.

Il worker thread si occupa di prendere la task dalla queue (che non è altro che l'fd del client) e successivamente esegue delle operazioni che in questo caso, sono caratterizzate dalla *TaskExecute* una funzione che si occupa di leggere la richiesta da parte di un client e soddisfarla.

Le Task della Queue sono un'astrazione dato che non sono propriamente delle vere e proprie attività da svolgere, ma sono semplicemente gli fd dei client che vogliono effettuare un'operazione.

2.5 SignalHandler

Il server come parte dell'inizializzazione impone di ignorare i segnali diretti al processo, per poi delegare la loro gestione ad un thread dedicato definito da una funzione SignalHandler che non fa altro che rimanere in sigwait fino all'arrivo di un segnale di tipo SIGINT, SIGHUP o SIGQUIT.

In tutti i casi il thread andrà a chiudere la sua pipe, (per evitare che rimanga aperta), e setterà lo stato del server in base al segnale ricevuto.

Se il segnale è SIGINT, l'handler setterà il ServerStatus a S (che sta per ShutDown), o meglio dirà al processo principale di spegnersi senza più servire i client, chiudendo anche tutte le connessioni.

Per gli altri due segnali l'handler setterà lo status a Q (che sta per Quitting), o meglio stiamo chiudendo serviamo solo gli ultimi client già "entrati".

3 Client

Il client non è altro che un processo singolo avviato da linea di comando che segue alla lettera le richieste della consegna. Ogni problema che può presentarsi viene "riferito alla shell" (stampato) se avviato in modalità -p.

Composto principalmente da CommandParser e CommandHandler.

- CommandParser
 - Crea una lista dei comandi passati da linea di comando e controlla che tutti i comandi abbiano gli argomenti che gli spettano, e le combinazioni corrette. Le funzioni -d e -D per esempio devono essere utilizzate rispettivamente in concomitanza le funzioni -R e -W o -w,
 - Apre inoltre la connessione al server se possibile, e "Imposta" il flag di standard output.
- CommandHandler
 - Questo è quello che che si occupa di fare le richieste vere e proprie, per ogni comando trovato nella lista generata dal Parser, verranno eseguite le dovute chiamate all'API

3.1 "Third-Party" functions

Per verificare l'esistenza dei path dati alle funzioni -d e -D, vengono utilizzate due librerie "ausiliarie", perchè non sono standard c99, che sono *realpath* e *stat* con relativa *struct stat* (secondo man7.org sono entrambe conformi POSIX-2001) per permette di ottenere il path assoluto dei path e di poter stabilire se un elemento trovato è un file normale o una directory (o anche altro).

Per parsare gli argomenti dei vari comandi si fa uso della funzione *strtok_r* ovvero la versione reentrant della strtok, non propriamente necessaria, ma permette una scalabilità più ampia. (Anch'essa definita POSIX da man7.org)

4 API

L'api segue le richieste dettate. Implementa le funzioni di comunicazione tra client e server. Ogni funzione va a definire il messaggio che vuole mandare al server in base alla funzione chiamata, se si presenta un errore di qualsiasi tipo questo viene gestito.

Per la comunicazione si utilizza una struttura messaggio che contiene alcuni parametri quali:

- Contenuto di tipo void*
- Dimensione in size_t del contenuto
- Tipo di Richiesta (es. WRITE, OPEN, READ, CLOSE)
- Feedback (Successo, ERRORE)
- Additional (Se tutto va a buon fine non dovrebbe contenere nulla se non nel caso della ReadNFiles o della expelled in cui invia il numero di file effettivamente letti, se Feedback è su errore conterrà

l'errore dato dal server in fase di esecuzione della richiesta; Se per esempio la richiesta è di accedere ad un file lockato da un altro client, additional avrà come codice EBUSY) PS. Nel caso della open additional conterrà il flag di modalità di apertura del file

Ogni funzione utilizza uno "standard" ovvero crea il messaggio con il tipo di richiesta e i dati relativi al file su cui vuole fare un'operazione ovvero il path e la len del path, successivamente attende dal server conferma del fatto che l'operazione sia effettivamente possibile, o in caso contrario il motivo per cui non è possibile, come ultimo step viene nel caso di scritture inviato un messaggio con il contenuto vero e proprio del file e la dimensione, o viene letto nel caso di letture, il file richiesto.

La ReadNFiles e la Write condividono (come anche la append) il gestore dei file che vengono espulsi o letti, ovvero la funzione che gestisce questi file è la stessa, perché a tutti gli effetti le azioni da svolgere sono le stesse ovvero leggo il path del file che mi viene inviato, confermo di volerlo ricevere, lo ricevo e nel caso in cui la directory rispettivamente -d o -D sia settata lo salvo creando un path di cartelle dettato proprio dal path del file in origine. Es.

- File sul server : home/file.txt
- Cartella Client : ./expelled
- Nuovo path nella memoria del client : ./expelled/home/file.txt

5 Make

Il makefile è strutturato in diverse opzioni dedicate ognuna a soddisfare una richiesta del progetto.

- ☐ **make**: Compila il target di default andando a generare tutte le cartelle necessarie al funzionamento del progetto per come sono stati impostati i test, compila tutti i target per bin/client e bin/server che sono rispettivamente gli "eseguibili" finali, e successivamente pulisce da tutti i file oggetto creati dalla compilazione.
- ☐ **make clean**: pulisce andando ad eliminare tutti i file oggetto creati dalla compilazione
- ☐ **make cleanall**: oltre ad avere il comportamento della clean, elimina anche gli eseguibili, ed esegue lo script delTexts per pulire completamente la cartella.
- ☐ **make makeMan**: stampa le opzioni del make
- ☐ **make test1**: Va ad eseguire il test1, dando i permessi di esecuzione e poi eseguendo lo script test.sh con parametro 1 (./scripts/test.sh 1). Questo va a testare il server con valgrind e 4 client su 4 nuovi terminali che effettuano diverse richieste. (in questo test il server ha una capacità significativa che non va ad intaccare i sistemi di espulsione)
- ☐ **make test2**: Va ad eseguire il test2, simile al primo test senza valgrind e con molte meno risorse da parte del server per poter sfruttare proprio il meccanismo di espulsione dei file. Script test.sh con parametro 2. (./script/test.sh 2)
- ☐ **make test3**: Va ad eseguire il test3 che come il 2 avvia il server senza valgrind, successivamente dopo poco tempo (do tempo al server di avviarsi completamente, millesimi di secondo comunque) avvio 14 client all'unisono che vanno ad effettuare per i primi 2 richieste "infinite" (essendo la -w una funzione che invia tutti i file di una directory, è come se fossero N write perché la sua implementazione è propriamente quella) di scrittura di file e per il primo dei due anche una directory per i file espulsi, i successivi 8 si occupano di inviare un file, richiederlo in lettura, ottenere la mutua esclusione, rilasciarla e cancellare il file appena inviato. Gli ultimi 4 si occupano di richiedere 10 file in lettura ognuno e salvarli in una directory. Script test3.sh (./scripts/test3.sh)
- ☐ **make stats**: Stats permette di ottenere le statistiche andando a parsare il file di log, che può essere passato come argomento, nel caso in questo non avvenga andrà a cercare nella cartella ./log, gli ultimi log di ogni test. (Es. ./log/test1/22/03/2022_13:20:12.txt) ed andrà a parsarne il contenuto.

- ☐ **make genTexts**: Oltre ad essere lo script eseguito dal target di default, può anche essere eseguito manualmente nel caso in cui per esempio si voglia aumentare la dimensione dei file per avere test più eterogenei. Script genTexts.sh. Oltre alle cartelle e i file crea anche un altro script bash che permette di mantenere finestre di shell aperte attraverso uno script aperte.
- ☐ **make delTexts**: Elimina tutti i file e le cartelle generate da genTexts.

NOTA Se il test1 e il test2 danno problemi dovuti ad un certo “x-terminal-emulator” si possono eseguire senza l’ausilio di terminali nuovi, eseguendo rispettivamente **make test1X** e **make test2X** che andranno ad eseguire tutto lo script sullo stesso terminale.

Nota Finale

Il progetto è stato pubblicato da me su github, ed è accessibile al link:
<https://github.com/PaulMagos/StorageServer>

Viene fatto uso di alcune implementazioni viste su stack overflow:

Per la sleep tra le varie operazioni, al link
<https://stackoverflow.com/a/1157217>

Per la creazione di cartelle al link
<https://stackoverflow.com/a/9210960>

Per l’accesso alle cartelle a cascata, è presente un qualcosa di simile al link
<https://stackoverflow.com/a/8438663>