UNIX FILE STORAGE SERVER

CORSO DI SISTEMI OPERATIVI @Unipi

Marco Marinò - Mat. 596440

GitHub Repository
https://github.com/marcomarinodev/file-storage-server-SOL

COME ESEGUIRE IL PROGRAMMA

1) make cleanall
2) make all
3) make test1 oppure make test2

INTRODUZIONE

Il progetto consiste nella creazione di un file storage server il quale mantiene file di qualsiasi tipo all'interno della memoria principale. Il protocollo attraverso cui vengono inseriti, rimossi e letti i file dal server è il protocollo client server. Il server riceve una richiesta dal client (formattazione della richiesta specificata in seguito) e attraverso un dispatcher, assegna il compito di eseguire la richiesta ad uno dei thread worker specificati in un file di configurazione (se ne parlerà più tardi) e saranno direttamente i worker a rispondere ai client in attesa di risposta. Il motivo per cui il server salva i dati in memoria principale è il seguente: il server è agnostico rispetto al disco del client (in generale, al client stesso). Il client manda richieste tramite API.

SERVER CORE

Il core del server è presente nel metodo run_server che prende in input una struttura Setup, composta da:

1. Path del socket
2. Numero dei workers
3. Capienza massima in bytes
4. Numero massimo di file
5. Path del file di log

Questa struttura viene ricavata dal metodo parse_config il quale legge un file *.txt in cui ogni linea rappresenta in ordine le informazioni citate prima (es: linea 1 path del socket). Vengono inizializzate due pipe; mwpipe ha il compito di fare da tramite tra il thread worker e il master (quando il worker termina l'esecuzione di una richiesta, scrive in mwpipe in modo da far sbloccare la select e rimettere ad 1 il bit relativo al descrittore del client di cui la richiesta è stata esaudita); pfd scritta da un thread che si occupa del la gestione dei segnali (sighandler_thread in attesa sigwait dei segnali indicati dalla maschera passata come input), infatti, quando pfd viene scritto, si sblocca la select e si va in un branch specifico (fd == pfd[0]) che gestisce il segnale specifico. Spendo delle parole per giustificare la scelta di gestione dei segnali:

- sig == SIGINT || sig == SIGQUIT
  - entro in SIGQUIT/SIGINT mode (is_sigquit = TRUE)
  - indico che si tratta dell'ultimo ciclo di select
  - sblocco ogni worker
- sig == SIGHUP
  - entro in SIGHUP mode (is_sighup = TRUE)
    - quando un client chiude la connessione e non ci sono più connessioni attive e siamo in SIGHUP mode, allora trattiamo questo caso come se fosse SIGQUIT/SIGINT. Dunque sblocchiamo i worker in attesa sulla pending_requests (coda in cui vi sono le richieste da soddisfare) usando la variabile di condizione pending_requests_cond (e la chiamata safe_cbroadcast) e indichiamo che si tratta dell'ultimo ciclo di select

- chiudo il server socket in modo da evitare che altri client si possano connettere nel frattempo
- metto a 0 il bit relativo al descrittore del socket
- verifico la dimensione delle connessioni attive
  - se ci sono connessioni attive allora aspetto che terminino (verifico che la dimensione dell'active_connections sia pari a 0)
  - altrimenti, si attiva la SIGQUIT/SIGINT mode (is_sigquit = TRUE)

Una volta terminati i cicli select, stampo le informazioni riguardo la situazione dello storage (quanti file ci sono, a quanto ammonta la dimensione occupata), libero la memoria allocata dinamicamente (implica anche la liberazione dei tcb con safe_pjoin). Da notare che se occorre SIGHUP la dll active_connections ha come invariante il fatto che essa sia vuota, come richiesto da specifica. Il server memorizza i file nella memoria principale utilizzando una struttura chiamata FRecord così composta:

- char pathname[MAX_PATHNAME] id univoco per un record nello storage
- size_t size
- time_t last_edit ultimo utilizzo necessaria per l'implementazione della politica di rimpiazzo LRU
- is_new boolean per discriminare un file che si vuole inserire durante LRU; banalmente se l'unico file che è presente nello storage è un file tale per cui is_new = TRUE allora vuol dire che questo file è troppo grande per lo storage. Questo controllo viene fatto prima di poter applicare la LRU, in modo da evitare che vengano eliminati dei file inutilmente
- last_op usato soprattutto quando si tenta di fare una operazione del tipo writeFile in cui bisogna controllare che l'ultima operazione fatta sul file sia una openFile con flag O_CREATE
- is_open flag
- pid_t last_client
- char *content contenuto del record salvato sullo heap

PROTOCOLLO ServerRequest / Response

Quando il server va ad ascoltare il descrittore, ottiene una struttura che rappresenta una richiesta ServerRequest così composta:

- pid_t calling_client (process id del client)
- int cmd_type (tipo di richiesta)
- char pathname[MAX_PATHNAME]
- long int size (possibile dimensione del contenuto inviato, se esiste)
- char content[MAX_CHARACTERS], di proposito ho scelto di dare una dimensione massima del contenuto della richiesta, pari a 0.8 MB

Maggiori informazioni sui valori possibili che può assumere una richiesta, sono scritte nel codice. La risposta ha un response code diviso in failure codes e success codes (es: FAILED_FILE_SEARCH oppure O_CREATE_SUCCESS). Come spiegato più in avanti, questi codici permettono alle API di capire se la richiesta è andata a buon fine oppure no. Struttura Response:

- char pathname[MAX_PATHNAME]
- char content[MAX_CHARACTERS]
- size_t content_size
- int code (the response code)

CACHE

Come politica di rimpiazziamento, ho deciso di implementare una LRU (Least Recently Used), in quanto a questo scopo mi sembrava quella più adatta (un'alternativa sarebbe potuta essere la LFU). Ho pensato ad un metodo FRecord *select_lru_victims(size_t incoming_req_size, char *incoming_path, int *n_removed_files) che mi ritorna un array di record eliminati dallo storage in modo da far spazio al nuovo record, ecco come funziona: server_stat è una struttura accessibile in mutua esclusione e contiene info sul server, come actual_capacity che rappresenta la dimensione corrente dello storage; la capacità viene ampliata alla dimensione del file da ospitare (rompendo momentaneamente l'invariante (questa è una operazione controllata,

in quanto non inseriamo sin da subito il record, ma cambia semplicemente il valore dell'intero)); ciclo ogni volta che viene sforata la capacity del server (sempre presente in server_stat) oppure se viene sforata la capacità in termini di numero di file max_files (presente in server_stat); all'interno del ciclo chiamo char *lru(HashTable ht, char *incoming_path) che ritorna oldest_path, ovvero il pathname del record che ha last_edit più vecchio. Per poter scansionare ogni record esamino per ogni riga di HashTable storage_ht, ogni record nella sua lista di trabocco. Quando lru restituisce il path del record meno usato recentemente, allora il record selezionato viene prima impilato in FRecord *victims (valore di ritorno di select_lru_victims) e poi viene definitivamente eliminato dallo storage. Inoltre viene passato per riferimento un intero che rappresenta il numero di file espulsi. Questa infomrazione non possiamo saperla a priori, di fatto FRecord *victims viene allocata massimizzando la sua dimensione rispetto al numero di file presenti nello storage prima di chiamare lru. Una volta che si ottiene l'array di record che sono stati espulsi dalla cache, occorre spedirli al client tramite API (il quale saprà se è stata specificata una cartella con -D in cui mettere i file), dunque semplicemente prima si notifica quanti file verranno spediti, in modo che l'API possa già sapere quante readn deve fare.

SERVER API

Come scelta di implementazione ho seguito questo schema: in base a ciò che voglio inviare, inizializzo la struttura ServerRequest con le informazioni che servono (es: cmd_type = WRITE_FILE_REQ) e dichiaro una Response response pronta ad ospitare response dal server. La specifica per le API si trova nel file s_api.h.

CLIENT

Il client distingue comandi in comandi di tipo configurazione e i comandi di tipo richiesta (sostanzialmente quelli che chiameranno le API). La prima cosa che fa il client è allocare due liste che conterranno comandi e richieste rispettivamente. Queste liste verranno man mano riempite tramite int _getopt(LList *configs, LList *reqs, int argcount, char **_argv) che fa il parsing delle opzioni verificando anche che ad ogni operazione sia stato assegnato il giusto parametro. Tramite la funzione int validate(LList configs, LList requests) è possibile poi verificare se la concatenazione dei comandi insieme hanno senso, per esempio non è ammesso: -d senza -R o -r, oppure -D senza -W o -w. La configurazione del client avviene tramite la funzione Client_setup apply_setup(LList config_commands) . Client_setup è una struttura composta da:

- char *socket_pathname nome del socket ("/tmp/server_sock")
- char *dirname_buffer nome cartella file letti
- char *ejected_buffer nome cartella file espulsi
- int req_time_interval tempo che intercorre tra una richiesta al server ed un'altra
- int op_log -p attivo
-

Il metodo int perform(Client_setup setup, LList *request_commands) legge dalla Linked List delle richieste ed manda una richiesta alla volta usando le API, in base ovviamente al comando preso dalla lista (semplice switch case del comando digitato dall'utente). Per poter ricavare gli argomenti, la funzione perform tokenizza gli argomenti, usando la ',' come separator. Ogni volta che una richiesta viene eseguita, essa viene eliminata dalla lista e si passa al nodo successivo della Linked List. Da notare l'utilizzo di realpath per poter ottenere l'absolute path dei file, dunque l'utente può tranquillamente inserire il path relativo rispetto alla posizione dell'eseguibile del client. Il metodo void manage_config_option(char **opt_id, char **opt_arg_value, int opt, LList *configs, char *_optarg) mette la configurazione nella lista delle operazioni di configurazione, stessa cosa per le operazioni di interazione col server attraverso il metodo void manage_request_option(char **opt_id, int opt, Request **_req, LList *reqs, char *_optarg). Il metodo void stub_perform() è uno stub usato per testare principalmente il metodo appendToFile in quanto non vi è presente una operazione specificata nel testo del progetto che permetta all'utente di fare esplicitamente la append di qualcosa in un file pre-esistente nello storage.

STRUTTURE DATI USATE

- queue.h coda generica usata nel server per implementare la coda di richieste da eseguire
- ht.h tabella hash generica usata dal server per poter memorizzare FRecord
- linked_list.h coda generica usata sia dal client per ospitare operazioni di configurazione che operazioni di chiamate al Server, ed usata anche dal server, in particolare nella tabella hash per percorrere le liste di trabocco

- doubly_ll.h lista doppiamente linkata (non c'è un motivo particolare per cui ho scelto questa struttura dati) per poter mantenere le connessioni attive

PARTI OPZIONALI SVOLTE

1. Prodotto File di Log (mi ha aiutato in fase di testing)
2. Realizzata l'opzione -D in modo da avere un riscontro visivo dei file espulsi dalla cache di file