Relazione SOL

Francesco Benocci 602495

Appello straordinario marzo 2022

Contents

1 GitHub

Link GitHub `https://github.com/Benocci/SOL-20-21`

2 Server

Config Il file di config passato come argomento deve essere un .txt con la coppia chiave=valore su righe diverse. Esso viene scansionato opportunamente ed inserito in una linked list che viene poi tradotta in una struttura che presenta i campi utili all'esecuzione del programma.

Gestione del lavoro Il server utilizza una socket per ricevere le richieste da parte dei client, nello specifico viene utilizzata una `select` che legge i file descriptor in lettura accettando nuove connessioni, chiudendo quelle da terminare oppure inserendo nella coda dei task i fd dei client da servire. I task vengono poi rimossi dalla coda dai thread worker che svolgono effettivamente la richiesta.

Mutua esclusione La concorrenza è resa sicura attraverso l'utilizzo di tre mutex, nello specifico:

- `mtx_status`: viene utilizzata per la modifica delle statistiche del server e ne garantisce l'atomicità;
- `mtx_file_log`: viene utilizzata per la scrittura nel file di log in modo tale che ogni scrittura su di esso sia atomica;
- `mtx_storage_access`: viene utilizzata per garantire l'accesso in mutua esclusione allo storage.

Inoltre la sopracitata coda dei task è una struttura che al suo interno utilizza una mutex chiamata `mtx` e due condition variables (`full` e `emty`) che ne garantiscono le insersioni e le rimozioni in maniera sicura.

Terminazione Vengono gestiti tre tipi di segnali per la terminazione del server come richiesto, nel caso di `SIGHUP` si imposta una variabile softExit che permette di concludere le richieste già inoltrate senza accettarne di nuove, terminando il server una volta completate quelle inoltrate. Nel caso di `SIGINT` e `SIGQUIT` invece viene impostata una variabile hardExit che porta alla chiusura istantanea del server senza permettere al server di concludere le richieste già inoltrate.

3 Storage

I File Sono rappresentati da una struct che mantiene tra i suoi campi:

- `pathname`: pathname del file;
- `data`: contenuto del file;
- `data_lengt`: dimensione del file;
- `lockedBy`: 0 se il file non è in stato di lock altrimenti mantiene il PID del client che ha la lock;
- `openBy`: -1 se il file non è aperto altrimenti mantiene il PID del client che tiene aperto il file.

Strutture dati Lo storage è rappresentato dall'unione di due strutture dati, una coda che permette di salvare i file in ordine così da poterli eliminare con politica **FIFO** e una hash table che permette di leggere il file senza dover scorrere la coda. Tutte le operazioni fatte sullo storage, quindi sono fatte in maniera coerente su entrambe le strutture mantenendo il loro contenuto identico.

Rimpiazzamento Il rimpiazzamento può avvenire per numero massimo di file raggiunti o per capacità raggiunta, il primo caso viene trattato nella richiesta del client writeFile che porta al superamento dei file massimi e viene risolto con l'eliminazione del primo file nella coda che viene spedito al client nel messaggio di risposta dal server. Il caso di superamento della capacità del server invece può avvenire nei casi in cui si stia facendo una writeFile o una appendFile, in entrambi i casi viene trattata allo stesso modo, i file rimossi (sempre con politica **FIFO**) dalla coda vengono inseriti in una coda temporanea per poi essere inviati uno per volta al client con messaggi di risposta. L'utilizzo di questa coda ausiliaria permette di non bloccare lo storage per troppo tempo in attesa dell'invio dei file.

4 Client

Il client si riassume in due cicli while:

1. un getopt che fa il parsing dei comandi passati da command line inserendoli in una struttura dati (`optList`) che si comporta come una coda;
2. un processo inverso in cui la struttura dati viene svuotata ed ogni comando viene processato comunicando opportunamente con il server.

5 Comunicazione

La comunicazione tra client e server avviene attraverso l'invio e la ricezione di "messaggi" che si incarnano nella struct `Msg` che al suo interno mantiene i seguenti campi:

- `opt_code`: intero che contiene codice dell'opzione che il client intende fare;
- `flags`: intero che contiene il flag nel caso di openFile e il numero di file nel caso di readNfile;
- `response_code`: stringa contenente il codice di risposta dal server;
- `path`: pathname del file su cui si sta operando;
- `pid`: process id del client che invia il messaggio;
- `data`: contenuto del file su cui si sta operando;
- `data_lenght`: dimensione del file su cui si sta operando.

Protocollo di comunicazione Lato server, acquisita la connessione, si invia un messaggio di benvenuto che notifica al client la corretta connessione, il client si collega al server utilizzando il metodo dell'API openConnection che sarà conclusa correttamente in caso di ricezione di un messaggio di benvenuto. Stabilita la connessione lato server si procede a ricevere i messaggi inviati dal client fino a quando non si riceverà un messaggio di closeConnection che interromperà la connessione.

6 Client API

Di seguito si elencano i metodi presenti nell'api e se ne descrive in breve il funzionamento:

- **openConnection/closeConnection**: permettono iniziare e terminare una comunicazione con il server. La prima cerca di connettersi alla socket per un tempo massimo di tentativi come richiesto, in caso di connessione avvenuta viene letto il messaggio di benvenuto dal server. La seconda si occupa semplicemente di mandare un messaggio di fine connessione attendendone uno di conferma che ne attesta la correttezza;
- **openFile/closeFile**: lato client sia openFile che closeFile si traducono in un messaggio spedito in cui viene specificata la richiesta e il file su cui viene fatta (e il flag nel caso di openFile) e un messaggio di risposta;
- **readFile**: si traduce anch'essa in uno scambio di messaggi, il primo mandato dal client specificando il tipo di operazione e il file e il secondo ricevuto dal client con all'interno il contenuto del file richiesto;

- **readNFile**: il messaggio spedito dal client, oltre al tipo di operazione, conterrà nel campo flags il numero N di file da leggere. Lato server quindi verranno salvati in una coda temporanea N (o meno in caso nel server non ce ne siano abbastanza) file che poi verranno inviati al client richiedente uno per volta. Terminato l'invio dei file si andrà a ricevere un ulteriore messaggio di conferma della corretta lettura di tutti i file richiesti (o possibili);
- **writeFile**: prima di inviare il messaggio, nel metodo viene aperto il file e ne viene letto il contenuto così da poterlo inviare. Lato server ricevuto il messaggio si potrà di capire se c'è stato bisogno di espellere un file per motivi di capacità e quindi di andare a ricevere ulteriori messaggi. In caso di nessuna espulsione oppure di un messaggio che indica che non bisognerà più aspettarsi altri file, il client riceverà un ultimo messaggio che notifica l'avvenuta scrittura nel server e che conterrà, in caso di espulsione di un file per numero di file massimi, il file espulso;
- **appendFile**: in maniera del tutto analoga alla writeFile, dopo aver spedito il messaggio con il contenuto da aggiungere al file, avverrà la comunicazione per ricevere i file espulsi per capacità insufficiente. Esattamente come detto prima, finita la comunicazione per i file espulsi, verrà ricevuto un ultimo messaggio che conferma lo stato della scrittura;
- **lockFile/unlockFile**: entrambi i metodi consistono in un messaggio inviato al server con tipo di operazione e pathname del file; lato server l'operazione utilizza il campo `pid` del messaggio per settare `lockedBy` del file nello storage. Al termine dell'operazione viene inviato un file con lo stato dell'operazione;
- **removeFile**: come nei casi precedenti si ha un messaggio inviato al server con il tipo dell'operazione e pathname del file ed un messaggio ricevuto con lo stato dell'operazione.

In tutti i metodi elencati si fa un controllo sui messaggi ricevuti, nello specifico sul campo `response_code` attraverso la funzione `responseCheck`, così da capire se le operazioni lato client sono avvenute correttamente (codice "ok") potendo così ritornare 0 oppure ritornando 1 settando errno in base al tipo di codice ricevuto.

7 Informazioni aggiuntive

Librerie di terze parti: hash table di Jakub Kurzak presente sul didawiki.

Nota su appendFile: dato che non era specificata un'operazione del client che richiedesse l'utilizzo di appendFile, quest'ultima viene utilizzata nel caso in cui si proceda a fare una scrittura su un file già esistente andando quindi a riscrivere al suo interno il contenuto in append. Ovviamente per accedere ad una appendFile si deve ricevere dal server un messaggio di errore di scrittura su file già esistente.