Relazione Progetto
Sistemi Operativi e Laboratorio

Samuele Calugi
Corso A
579086

Luglio 2021

Contents

1 Introduzione

Come previsto dalle indicazioni dell'esame, il progetto di seguito è stato sviluppato nella versione ridotta e semplificata poiché la parte facoltativa non è prevista nell'appello di Luglio 2021. Il codice sorgente e i vari file necessari per il corretto funzionamento del programma sono stati caricati in una repository pubblica su Github. È possibile accedere alla repository del progetto attraverso il seguente link: `https://github.com/Walrus98/Progetto-SOL`, così da poter visualizzare il codice sorgente e tutti i commit effettuati.
Le librerie utilizzate nel progetto sono le seguenti:

- **icl_hash.h**, libreria vista a lezione utilizzata per la creazione di mappe.

- **list_utils.h**, libreria realizzata dal sottoscritto utilizzata per la creazione di liste monodirezionali.

1.1 Makefile

L'intero progetto può essere compilato ed eseguito attraverso le seguenti regole:

- **all**: viene chiamato per default dal comando **make**, compila il client e il server attraverso le regole **build-client** e **build-server**.
- **build-client**: compila tutti i file del client e genera l'eseguile.
- **build-server**: compila tutti i file del server e genera l'eseguile.
- **build-server-test1**: compila tutti i file del server con i config richiesti da test1 e genera l'eseguile.
- **build-server-test2**: compila tutti i file del server con i config richiesti da test2 e genera l'eseguile.
- **client**: esegue un clear della console e successivamente avvia il client con valgrind.
- **server**: esegue un clear della console e successivamente avvia il server con valgrind.
- **server-test1**: esegue un clear della console e successivamente avvia il server con i config richiesti da test1 con valgrind.
- **server-test2**: esegue un clear della console e successivamente avvia il server con i config richiesti da test2 con valgrind.
- **clean**: rimuove tutti i file object generati dalle regole elencate precedentemente.

1.2 Test case

Nel progetto è presente una cartella di nome *"tests"* che al suo interno contiene i due file bash richiesti per testare il corretto funzionamento del programma. I due script una volta avviati compilano in maniera autonoma client e server tramite le regole del Makefile sopra citate. Successivamente, in base al tipo di file bash avviato, viene eseguito il server con il propio file config opportuno e un insieme di client che eseguono differenti richieste sul server.

2 Funzionamento del programma

2.1 API

Per potersi interfacciare con il server, il client include all'interno del suo codice un file header chiamato **client_network.h**. Quest'ultimo possiede l'elenco di tutti i metodi richiesti dal testo dell'esame e permette quindi di per poter inviare e ricevere messaggi con il server attraverso l'utilizzo di un protocollo. Tutti i metodi del file header sono implementati nella classe **client_network.c**.

2.2 Protocollo

I messaggi trasmessi dal client al server sono formati da due parti:

- **Header**, contiene un id che identifica il tipo di pacchetto trasmesso e la dimensione del payload.
- **Payload**, il contenuto effettivo del pacchetto che si vuole trasmettere.

Di conseguenza, ogni pacchetto è formato da due buffer: il **BufferHeader** e il **BufferPayload**.

Il **BufferHeader** ha dimensione fissa di **8 Byte** di cui:

- i primi 4 byte, (quindi il primo intero) contengono l'id del pacchetto che si vuole trasmettere.
- i 4 byte successivi, (quindi il secondo intero) contengono la dimensione del payload.

Il **BufferPayload** ha una dimensione variabile e contiene tutti i dati che si vogliono trasmettere al server

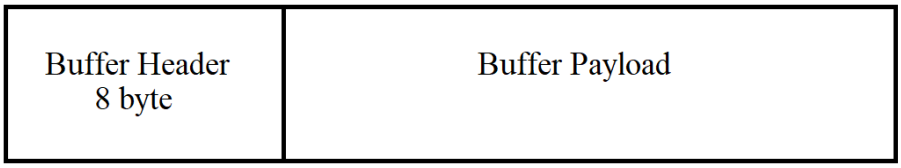

Figure 1: Stuttura del buffer di invio di messaggi al server

Poichè il client ogni volta che invia un messaggio al server si mette in attesa di ricevere un messaggio di risposta da quest'ultimo (quindi i pacchetti da parte del client vengono inviati in maniera sequenziale), il server non ha bisogno di inviare al client l'id del pacchetto perché il client conosce già il tipo di pacchetto che sta per ricevere. Di conseguenza, il server invia prima la dimensione del pacchetto, poi il contenuto del buffer che vuole inviare.
In alcuni casi, se il server non deve inviare nessun dato se non l'esito della richiesta che il client ha inviato, il server invia semplicemente un intero come messaggio di risposta: nel caso di successo, il server invia 0, in caso di errore un numero negativo.

3 Server

3.1 Config

Come richiesto dall'esame, il server all'avvio legge un file di configurazione con il seguente formato:

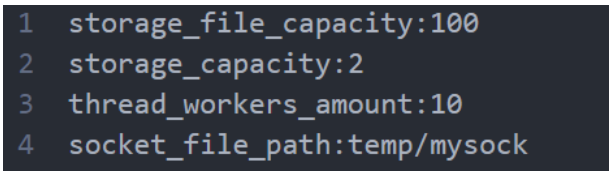

Figure 2: Config.txt del server

Nel quale:

- **storage_file_capacity**: definisce il numero massimo di file che possono essere inseriti nello storage.
- **storage_capacity**: definisce la dimensione massima in MB dello storage.
- **thread_workers_amount**: definisce il numero di thread workers.
- **socket_file_path**: definisce il percorso in cui creare il file socket.

3.2 Struttura del Server

Il server adotta come politica di rimpiazzamento dei file la politica **FIFO**, in cui il primo file ad entrare, è anche il primo file ad uscire dallo storage. All'avvio, il server legge il file config sopra citato e successivamente avvia i seguenti thread:

- **Thread Signal Handler**: si occupa di gestire la ricezione dei segnali attraverso l'utilizzo del metodo **sigwait()**, in caso di ricezione di un segnale di *SIGINT*, *SIGQUIT* o *SIGHUP* il server termina correttamente l'esecuzione del programma come richiesto dall'esame.
- **Thread Dispatcher**: si occupa di accettare le nuove connessioni da parte dei client e delega le loro richieste ad una pool di *Thread Worker*.
- **Thread Workers**: un insieme di thread che si occupano di eseguire le richieste dei client in maniera concorrente sullo storage del server.

Nel progetto sono state utilizzate due pipe:

- **pipeHandleClient**, utilizzata per inserire nuovamente il file descriptor all'interno del Thread Dispatcher dopo che il Thread Worker ha terminato l'esecuzione del task richiesto.
- **pipeHandleConnection**, è usata per terminare l'esecuzione del Thread Dispatcher quando viene inviato uno dei segnali di terminazione. Il Thread Signal Handler riceve il segnale, tramite la pipe comunica con il Thread Dispatcher (che è fermo in attesa di ricevere nuove richieste sul metodo **select()**) e notifica al Thread Dispatcher di terminare la propria esecuzione. Nel caso di un *SIGINT* o *SIGQUIT*, il Thread Dispatcher termina istantaneamente. Nel caso di un *SIGHUP* il Thread Dispatcher tiene traccia di tutte le connessioni aperte tramite un contatore, quando il numero delle connessioni aperte diventa 0, termina l'esecuzione.

3.3 Server Storage

Lo Storage del server è stato implementato come una mappa attraverso l'utilizzo della libreria **icl_hash.h**, ed ha:

- come chiave: il file, definito da una struct.
- come valore: la lista di tutti gli utenti che hanno richiesto l'operazione di open su quel file.

La struct del file è formata da:

- **filePath**: il percoso assoluto del file.
- **fileContent**: il contenuto del file.
- **fileSize**: la dimensione del file in byte.
- **fifo**: utilizzato dalla politica di rimpiazzamento FIFO. Ogni volta che un nuovo file viene creato, il server gli assegna un un valore univoco ed incrementale (per esempio il primo file avrà come valore 0, il secondo 1, etc.). Durante la politica di rimpiazzamento, il server sceglie come vittima il file che ha il valore fifo più basso e poi lo rimuove dallo storage.
- **fileLock**: lock utilizzata dai Thread Workers per eseguire le operazioni sul server in mutua esclusione. Ogni volta che un thread deve effettuare delle operazioni su un file già inserito nella mappa (per esempio aggiungere un nuovo utente alla lista o la modifica del contentuo del file), il Thread Worker deve prima acquisire la lock sul file, eseguire il task e poi rilasciarla a termine dell'operazione.

```
4  typedef struct File {
5      char *filePath;
6      char *fileContent;
7      size_t *fileSize;
8      int *fifo;
9      pthread_mutex_t *fileLock;
10 } File;
```

Figure 3: Struct del file

3.4 Lock

Per rispettare la concorrenza e l'atomicità nel Server Storage, sono state utilizzate le seguenti lock:

- **StorageLock**: lock globale su tutta la mappa, viene utilizzata per inserire un nuovo file all'interno della struttura dati.
- **CapacityLock**: lock utilizzata per modificare in mutua esclusione il numero di file presenti nel server e la dimensione totale dello storage.

- **ReplacementFrequencyLock**: lock utilizzata per contare in mutua esclusione quante volte è stata chiamata la politica di rimpiazzamento (per stampare il resoconto del server a fine esecuzione).
- **fileLock**: lock del file, utilizzata per modificare in mutua esclusione un file già presente all'interno della mappa.

4 Client

Come già spiegato, il client per comunicare con il server utilizza la libreria **client_network.h** che si occupa di aprire la connessione sul server tramite socket e di comunicare con il server utilizzando il protocollo spiegato precedentemente.
Il client per eseguire una qualsiasi operazione su un file del server deve prima eseguire la richiesta di open su quel file, eccetto per la richiesta di **readN()** dato che non è possibile conoscere il numero preciso di file che l'utente andrà a leggere. Di conseguenza, un utente può eseguire la remove o la close di un file soltanto se prima ha richiesto la open su di esso.
Quando un utente si disconnette, il server rimuove in mutua esclusione (tramite **fileLock**) il file descriptor dell'utente disconnesso da ogni lista della mappa, se presente. In questo modo, se un nuovo utente si connette al server e ottiene lo stesso file descriptor dell'utente disconnesso precedentemente, il nuovo utente non avrà nessuna richiesta di open già eseguita.

Come richiesto dal testo dell'esame, il client può inviare differenti richieste al server passando i comandi per argomento. Per poter ricevere i comandi in qualsiasi ordine (quindi non è necessario che il primo comando da passare al client è *-f*), quest'ultimi vengono inseriti all'interno di una lista monodirezionale. Successivamente, il client prima controlla la presenza degli argomenti: *-h*, *-p*, *-t*, *-d* e *-f*, se questi vengono trovati allora il client li esegue, li rimuove dalla lista e poi esegue tutti i comandi rimasti nello stesso ordine di inserimento.