Relazione sul Progetto di Sistemi Operativi e Laboratorio

A.A. 2020/2021

Luca De Paulis

1 giugno 2021

1 Introduzione e informazioni generali

Il progetto di Sistemi Operativi e Laboratorio dell'A.A. 2020/2021 consiste nella realizzazione di un sistema di *filestorage* gestito da un server multithreaded che comunica con i vari potenziali clienti tramite un API. Durante tutto il corso dello sviluppo, il codice sorgente e i vari file necessari al funzionamento del progettexttttto sono stati caricati su Github e sono accessibili alla repository pubblica con indirizzo `https://github.com/lucadp19/SOL-Project-2021`.

2 Makefile

Per compilare il progetto è messo a disposizione un Makefile con diverse opzioni.

`make` L'opzione base è la regola `all`, che può essere invocata dando il comando `make`. Questa regola compila l'intero progetto, creando inizialmente le cartelle necessarie se esse non sono ancora presenti.

`make clean` Per ripulire la directory dai vari file oggetto, eseguibili e librerie viene anche messa a disposizione una regola `make clean`.

`make test1` La regola `make test1`, se lanciata dopo la compilazione, si occupa di eseguire il primo test, che mostra le varie opzioni del client.

`make test2` La regola `make test2` esegue il secondo test, che mostra l'algoritmo di rimpiazzamento del server. Di default questa regola fa in modo che il server scelga l'algoritmo LRU, ma con le regole `make test2FIFO` e `make test2LRU` possiamo rendere esplicita questa scelta.

`make cleanTests` Vengono inoltre fornite tre regole ulteriori, ovvero `make cleanTest1`, `make cleanTest2` e `make cleanTests` che si occupano solo di ripulire i file generati dai vari test in modo da poterli rieseguire partendo da uno stato pulito.

`make stats` Infine il target `make stats` lancia lo script `scripts/statistiche.sh` che tenta di analizzare l'ultimo file di log presente nella directory `logs/`.

3 Libreria di utilities

Il client, il server e la libreria di API sfruttano tutte i tipi e le funzioni di base definite nella libreria `libutil.so`. Essa è composta da diversi include file, ognuno associato ad una specifica funzionalità.

Il file `util/util.h` contiene gli include file più comuni, più alcune funzioni e macro che vengono usate frequentemente in ogni parte del progetto.

I file `util/node.h`, `util/list.h`, `util/hash.h`, `util/hash/hashtable.h` e `util/hash/hashmap.h` contengono in particolare le strutture dati più usate nel corso del progetto, ovvero liste doppiamente linkate e tabelle hash.

La distinzione tra *tabelle* e *mappe* è dettata dal fatto che le due strutture vengono usate in modo diverso: la prima per gestire collezioni di interi, mentre la seconda collezioni di coppie chiave-valore, dove la chiave è una stringa e il valore è un tipo generico `void*`; unificarle in una sola struttura avrebbe incrementato di molto la complessità del codice.

Infine il file `util/files.h` contiene alcune funzionalità per gestire la scrittura o lettura di file dal disco, in modo da evitare la duplicazione del codice tra client e API.

4 API

Le API permettono la comunicazione tra client e server tramite un insieme di funzioni definite in `api.h`. Tra queste, `api_perror` è degna di nota in quanto consente di scrivere su stderr gli errori generati dalle funzioni delle API con un messaggio che rispecchia meglio il loro significato (descritto comunque nella documentazione delle funzioni in `api.h`) rispetto alla funzione `perror` standard.

L'API è realizzata in modo da essere completamente indipendente dal client che la usa, e pertanto viene compilata in una libreria dinamica di nome `libapi.so`. Per poterla usare è tuttavia necessario linkare anche la libreria di utilities `libutil.so`, che contiene funzionalità sfruttate dalle API.

La comunicazione tra server e API sfrutta un preciso protocollo che dipende dal tipo di richiesta fatta dalle API: i dati necessari per implementare questo protocollo sono descritti nell'include file `server-api-protocol.h`. In particolare le API innanzitutto inviano al server un codice che rappresenta il tipo di operazione richiesta dal client, e poi inviano tutti i dati necessari. A questo punto:

- se l'operazione può comportare l'espulsione di file dal server e il loro invio al client attraverso le API, il server invia innanzitutto un messaggio intermedio che indica la buona riuscita dell'operazione fino a quel punto oppure un errore, poi invia i file espulsi e infine invia un codice di successo/errore finale;
- invece se l'operazione non comporta espulsione di file il server invia semplicemente un messaggio finale alle API.

Alla fine della comunicazione con il server le API inviano al client il codice restituito dal server opportunamente trasformato in un valore di `errno`: il client può leggerlo tramite `api_perror` e, nel caso sia un codice di errore, decidere se terminare o ritentare. In generale gli errori inviati dal server sono *non-fatali*: l'unico caso di errore fatale è quello descritto da `ENOTRECOVERABLE` che indica un fallimento da parte del server, oppure delle API, oppure nella comunicazione tra i due.

Infine, le funzioni `lockFile` e `unlockFile` non sono state realizzate, ma sono comunque invocabili da un client: al momento entrambe le funzioni restituiscono immediatamente `-1` con un codice di errore `errno = ENOSYS`, per indicare che la funzione non è implementata.

5 Il client

Il client sfrutta le API fornite per inoltrare delle richieste al server. Le richieste vengono passate al client tramite linea di comando ed esso si occupa di farne il parsing e controllare che rispettino i vincoli dati dalla specifica. Questo controllo tuttavia viene ignorato se viene passata al client l'opzione `-h`, che stampa un messaggio che spiega le varie opzioni e fa terminare il client immediatamente.

Per fare il parsing il client sfrutta l'estensione GNU del comando `getopt` che consente di dichiarare opzioni con valori opzionali: in particolare ciò viene sfruttato per realizzare l'opzione `-R`. Per chiamare questa opzione con un valore esplicito è quindi importante non lasciare spazi tra `-R` e il valore scelto.

Rispetto a quanto descritto nella specifica, il client ha inoltre un'opzione `-a` che setta il tempo (in millisecondi) da aspettare dall'inizio di un tentativo di connessione verso il server fino all'eventuale timeout (segnalato dalla `openConnection` tramite l'errore `ETIMEDOUT`).

Inoltre il client si occupa di trasformare ogni path relativo dato dall'utente in un path assoluto, in modo che il server identifichi i file solamente attraverso il loro path assoluto. Questo non vale per le directory passate alle opzioni `-d` e `-D`: esse potrebbero non esistere e quindi sarebbe impossibile determinare il loro path assoluto tramite la funzione `realpath`. In questo caso le API si occupano di creare le directory che non esiste, anche ricorsivamente, usando alcune delle funzioni dell'include file `util/files.h`.

Come nel caso delle API, non avendo implementato le funzionalità di lock/unlock dei file se le opzioni `-l` o `-u` vengono invocate dall'utente la funzione che esegue le varie opzioni stamperà un messaggio di errore spiegando che quell'opzione non è stata implementata.

Infine, il client non termina al primo codice di errore restituito dal server (tramite le API): infatti è possibile che il file richiesto sia stato eliminato dall'algoritmo di rimpiazzamento, oppure che sia stato aperto da un altro client in modalità locked, eccetera. Dunque il client (se l'opzione `-p` è stata selezionata) scrive su schermo l'eventuale errore e, se l'errore non è fatale, prova ad eseguire il resto delle operazioni.

6 Il server

Il server è l'applicazione multithreaded che gestisce lo storage di file. All'apertura legge un file di configurazione con il seguente formato:

```
no_worker = <number of worker threads>
max_space = <max space in MBytes>
max_files = <max number of files>
cache_pol = <cache replacement policy, may be FIFO or LRU>
sock_path = <path to socket file>
path_dlog = <path to the directory containing log files>
```

Possono esserci un numero arbitrario di spazi tra la parola chiave e il simbolo `=`, oppure tra esso e il valore; inoltre possono esserci un numero arbitrario di righe vuote tra una coppia *`chiave=valore`* e un'altra, ma ogni tale coppia deve essere su una riga separata.

Di default il server assume che il file di configurazione si trovi in `./config/config.txt`; per cambiare path basta passare il path desiderato come unico argomento da linea di comando all'eseguibile del server.

All'avvio il server quindi crea le varie strutture dati necessarie al suo funzionamento, come la hashmap `files`, che conterrà i vari file memorizzati nel server, e lancia diversi thread. Uno di questi è il thread *signal handler*: il suo scopo è gestire i segnali `SIGHUP`, `SIGINT` e `SIGQUIT` inviati al server per richiederne la terminazione. In tutti e tre i casi il server aspetta che i thread terminino il loro lavoro (se non sono in attesa) e libera la memoria allocata, in modo da ottenere una chiusura pulita.

Le richieste dei client sono gestite interamente dai thread workers: il thread master (cioè il main) si occupa di inserirle in una coda FIFO, mentre i worker prelevano queste richieste, le eseguono e restituiscono il risultato al thread main. In caso di successo o errore non-fatale il thread main continua la sua connessione con il cliente, mentre in caso contrario chiude la connessione.

Nel gestire le varie richieste i worker devono agire concorrentemente sulla hashmap dei file. Il protocollo di accesso scelto è quello delle *lock reader-writer*: ogni file può essere aperto da un thread in modalità lettore (se deve solo leggere le informazioni contenute nel file) oppure in modalità scrittore (se deve anche modificarle). In particolare più thread possono leggere lo stesso file contemporaneamente, ma se un thread è in modalità scrittore nessun altro thread può né leggere né modificare la hashmap.

Per gestire la memoria limitata dai parametri definiti dal file di configurazione, il server implementa degli algoritmi di rimpiazzamento, che consentono di eliminare file in modo da liberare spazio. Gli algoritmi implementati

sono l'algoritmo FIFO (che elimina il file creato più lontanamente nel tempo) e l'algoritmo LRU (che elimina il file usato più lontanamente nel tempo).

Quando il server raggiunge i limiti prefissati sceglie dunque dei file da espellere e, se possibile, li invia alle API che avranno il compito di scriverli su disco. Tuttavia nel caso della `openFile` con flag `O_CREATE` ciò non è possibile, in quanto la funzione della API non contempla la possibilità di scrivere file su disco. In questo caso il file viene semplicemente distrutto e questo fatto viene riportato nel file di log.

Questo file (che verrà creato nella directory specificata nel file di configurazione) conterrà ogni operazione svolta dal server durante il suo utilizzo, il successo o fallimento e l'eventuale quantità di dati scritti o letti. Per ottenere alcuni dati significativi da un file di log è possibile sfruttare lo script `scripts/statistiche.sh` invocandolo direttamente oppure lanciando il target `make stats` del Makefile.

7 Test e scripts

Per testare le varie funzionalità del server e del client sono stati creati due file di test, `scripts/test1.sh` e `scripts/test2.sh`.

Il primo serve a mostrare le varie opzioni del client e il suo funzionamento, mentre il secondo serve a mostrare il funzionamento dell'algoritmo di rimpiazzamento dei file. In particolare per mostrare la differenza tra diverse politiche di rimpiazzamento lo script `scripts/test2.sh` può opzionalmente prendere un parametro da linea di comando, che corrisponde al tipo di algoritmo da usare e può essere LRU oppure FIFO.

I due script di test possono anche essere lanciati attraverso i target `make test1` e `make test2` del Makefile: in questo caso il test2 userà di default l'algoritmo LRU, ma si può specificare esplicitamente l'algoritmo FIFO tramite il target `make test2FIFO`.

Infine lo script `scripts/statistiche.sh` può essere usato per ottenere alcune statistiche significative dal file di log generato dal server. Se si passa un parametro da linea di comando lo script tenterà di analizzare il file al path dato; altrimenti di default cercherà il file più recente contenuto nella directory `./logs/`.