Relazione progetto "Biblio" Laboratorio II

Angelo Quartarone - M. 598396

AA 2022-2023 corso A

1 Introduzione

Il progetto consiste sullo sviluppo di Biblio, un sistema bibliotecario distribuito. Spacchettando l'archivio compresso troveremo diverse directory e diversi file:

- bibData: contiene tutti i file che ogni server elaborerà all'inizio della propria esecuzione
- bibData_old e bibData_to_test: sono directory che contengono i file (originali) che vengono preparati all'esecuzione da uno script in bash che ne sistema la formattazione, ed i file aggiornati che vengono modificati al termine dell'esecuzione di ogni server. Questa scelta è stata fatta per motivi di "comodità", date le varie esecuzioni di test che ho dovuto effettuare
- conf: è una directory che contiene il file di configurazione che i vari client utilizzano per ottenere le informazioni utili alle connessioni via socket ai vari server
- lib: contiene tutte le librerie utilizzate dai server e dai client per funzionare (di seguito saranno descritte più nel dettaglio)
- log: contiene 2 tipi di file .log: i primi sono i file di log generati dall'esecuzione dei server, i secondi contengono i log di Valgrind
- bibaccess.sh: è uno script in bash utilizzato per mostrare in output tutte le richieste processate da ogni singolo server
- bibclient.c: contiene il codice del client
- bibserver.c: contiene il codice del server
- fileCleaner.sh: è uno script in bash che standardizza i file con cui ogni server costruisce il proprio inventario; nello specifico, data l'inconsistenza nei file originali ho deciso di adottare questa formattazione: "etichetta: valore; etichetta: valore; . . . ", quindi con uno spazio dopo ogni ":" e ";"
- README.md

2 Librerie

In questo progetto è stata utilizzata la libreria queue.c fornita dalla prof.sa Alina Sirbu, con l'aggiunta di funzioni per la gestione errori e con una nuova funzione che risolve un problema che ho riscontrato nella scrittura del codice bibserver.c; nello specifico, avevo notato che la chiamata pop() bloccava il chiamante in un loop in cui si attendevano dei dati; avendo la necessità di avere il pieno controllo sui cicli per una gestione ottimale dei segnali e quindi la corretta terminazione dei thread, ho reso la chiamata alla funzione non bloccante, e poi ho inserito questa funzione in un while loop all'interno del behavior dei thread worker in modo da poter fare una break alla ricezione di un segnale.
È presente una libreria linkedList.c, che implementa una mia versione di lista concatenata: questa ha le classiche funzioni di aggiunta e rimozione dei nodi, in più l'ho dotata di una funzione che ritorna l'i-esimo elemento della lista, funzione che ho trovato utile per recuperare le informazioni da un nodo senza doverlo estrarre esplicitamente.
Book.c contiene tutte le funzioni che entrambi server e client utilizzano per lavorare con i la struttura dati del singolo libro. Infine, util.c contiene delle funzioni per: handling degli errori, stampa di messaggi thread safe in stderr e una mia implementazione sicure della free che imposta il puntatore passato in input a NULL una volta de-allocata la memoria puntata da esso.

3 Struttura e scelte implementative

Server

Il server inizia le sua esecuzioni con delle operazioni preliminari quali:

- la creazione di mutex utili alla gestione delle zone critiche presenti nella scrittura dei vari log (sia su file che su standard error)
- la creazione e il detach del thread che si occuperà soltanto della gestione dei segnali apertura dei file di log
- parsing del file record
  - Il mio server utilizza una struttura dati personalizzata chiamata single_book (vedi book.h) formata da array di caratteri (di lunghezza fissata a 1024 bytes), uno per ogni campo che un libro può presentare; successivamente, queste strutture "riempite" vengono aggiunte ad una linked list (se e solo se un libro non è stato già aggiunto; due libri si definiscono diversi se hanno almeno un campo con contenuto differente). Alcuni libri possono presentare il campo "prestito" vuoto, in questo caso, la funzione loan_check modifica il campo prestito inserendo la stringa "AVAILABLE"
- creazione delle socket e scrittura delle informazioni per il collegamento dei client sul file "addr.conf"

- creazione di W thread worker
- main loop: in questo while loop vengono iterate tutte quelle azioni utili al funzionamento dell'accept, dell'aggiornamento della struttura fd_set e dell'aggiunta dei file descriptor pronti su una coda di lunghezza illimitata condivisa dai vari thread

infine troviamo tutte le azioni "di terminazione" come:

- join dei thread worker
- scrittura sul file di tutti i record aggiornati
- chiusure di file descriptor e free di memoria allocata.

behaviour del thread worker

ogni thread riceve per argomento la struttura contenente:

- la coda condivisa di file descriptor
- la lista condivisa di libri
- mutex
- fd_set

Ogni thread esegue un while true loop in cui è codificato il mio algoritmo di gestione richieste: controllo: se è stato registrato un SIGINT o un SIGTERM e se la coda di file descriptor è vuota allora termina. Lettura su socket: se il numero di bytes letti è maggiore di 0 e se non è stata immessa una data corretta dal client viene mandato a quest'ultimo un messaggio di errore; altrimenti un for cycle itera su tutti i libri contenuti nella lista ed esegue dei controlli: se il libro che che il client ha richiesto è presente nella libreria, il tipo del messaggio ricevuto è prestito e il prestito può essere effettuato, viene inviato al client un record contenente le informazioni dei libri presi in prestito. se il tipo del messaggio era invece una query, il server invia tutti i record che soddisfano la ricerca. In qualunque caso non vengono trovate informazioni il server invia una risposta negativa. Dopo controlla se il numero di bytes letti è uguale a zero, se lo è, allora rimuove il file descriptor dal fd_set e aggiorna il numero massimo di client presenti. Ogniqualvolta viene mandato un messaggio ad un client viene anche aggiornato il file .log.

Client

Il client inizia la sua esecuzione con un parsing degli argomenti, affidato alla funzione arguments_parser; essa legge tutti gli argomenti passati per parametro e salva le informazioni all'interno di una struttura dati personalizzata chiamata variable_for_request dividendole in etichetta e valore. Successivamente queste strutture vengono inserite in una linked list che verrà quindi utilizzata per la

creazione della richiesta da inviare al server. A questo punto il client legge dal file di configurazione le informazioni per il collegamento via socket. Utilizzando ora un ciclo il client conta il numero di server alla quale collegarsi e, tramite un altro ciclo, legge effettivamente le informazioni salvandole in un array di file descriptor creato con il numero esatto di server presenti nel file calcolato dal primo ciclo. Quindi avviene il collegamento vero e proprio ai vari socket e la costruzione del messaggio da inviare.

Troviamo, infine, il main loop in cui vengono ripetute l'invio della richiesta e la lettura della risposta. Il programma termina (dopo aver eseguito tutte le free necessarie) solo dopo aver interrogato tutti i server disponibili.

Principali strutture dati utilizzate

```c
typedef struct
{
    char autore[BUF_SIZE];
    char titolo[BUF_SIZE];
    char editore[BUF_SIZE];
    char anno[BUF_SIZE];
    char volume[BUF_SIZE];
    char scaffale[BUF_SIZE];
    char collocazione[BUF_SIZE];
    char luogo_pubblicazione[BUF_SIZE];
    char descrizione_fisica[BUF_SIZE];
    char nota[BUF_SIZE];
    char prestito[BUF_SIZE];
} single_book;
```

(a) single_book

```c
typedef struct
{
    char *etichetta;
    char *valore;
} variable_for_request;
```

(b) var_for_request

Figure 1: Strutture utilizzate in bibserver e bibclient

```c
typedef struct node
{
    void *data;
    struct node *next;
} node_t;

typedef struct linked_list
{
    node_t *head;
    pthread_mutex_t lock;
    int size;
} linked_list_t;
```

Figure 2: Struttura linked list

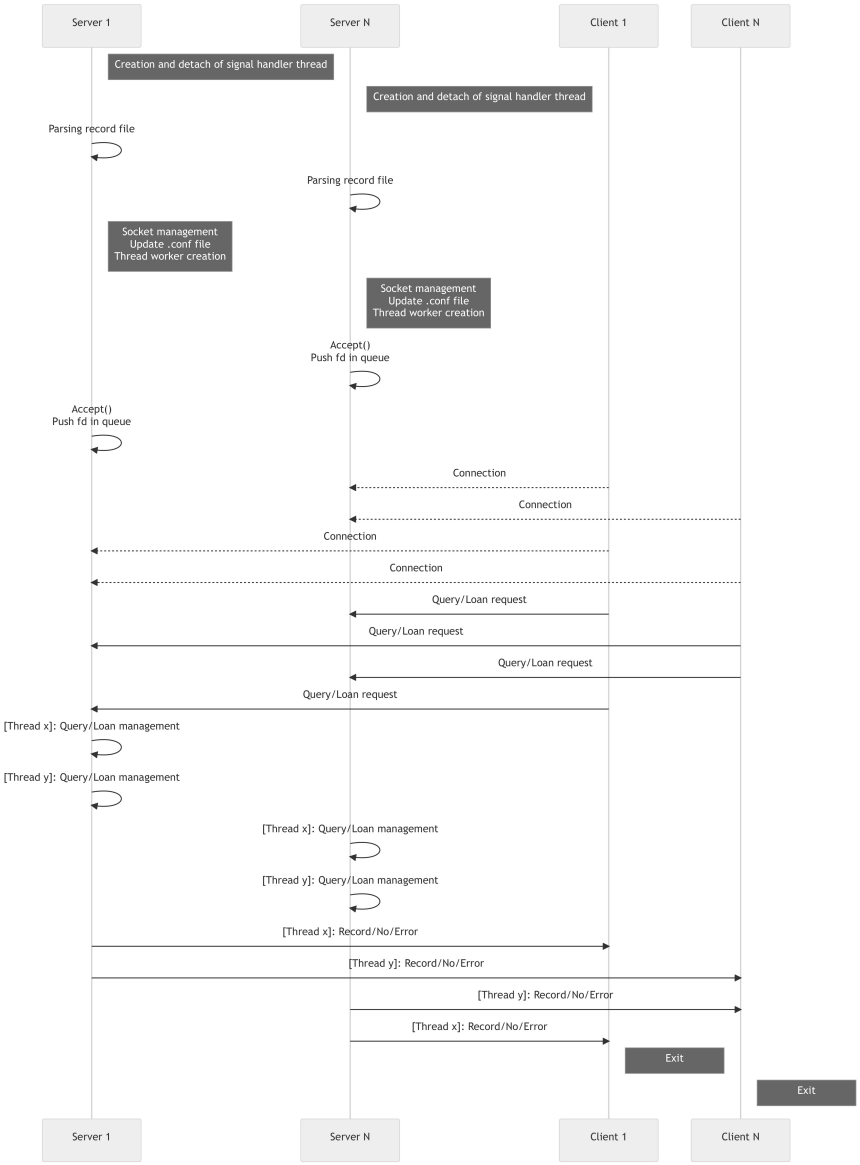

Figure 3: Schema esempio esecuzione