Progetto Laboratorio 2

Giulio Rossi

Introduzione

Questo progetto è sviluppato a partire dai file di intestazione i quali vengono usati per la dichiarazione di funzioni e di strutture. Sono implementati successivamente dai loro rispettivi file.c che poi verranno usati all'interno del codice del server e del client.
I file e le cartelle sono organizzati in base alla loro funzionalità o al loro tipo. Iniziando della cartella **config** troviamo il file bib.conf che contiene le informazioni che servono al client per connettersi a tutti i server attivi.In **data** possiamo trovare i file txt che contengono le informazioni relative ai record che andranno a far parte della biblioteca. Nella cartella **include** ci sono tutti i file di intestazione dove vengono dichiarate le funzioni e le strutture dati che verranno usate dai rispettivi file sorgente nella cartella **src** e anche dai file del server e del client. Dentro **logs** ci sono i file di log che verranno generati dall'esecuzione del programma.Poi nella cartella principale troviamo bibaccess (per il controllo dei file di log), il Makefile (per la compilazione e l'esecuzione) e il file README.md.
Le librerie usate servono rispettivamente per:

- commonutils.h: gestione errori e operazioni di base(allocazione, chiusura file descriptor ecc..)
- mythreads.h: gestione operazioni su thread, mutex e condition variables
- mysocket.h: gestione operazioni su socket e comunicazione
- recordutils.h: funzioni principali per l'esecuzione delle funzioni della biblioteca
- timeutils.h: funzioni per gestire i prestiti della biblioteca
- unboundedqueue.h: gestione della coda condivisa

Le strutture dati principali si trovano nel file *recordutils.h* e in *unboundedqueue.h* e contengono rispettivamente:
recordutils.h:

- *message* : struttura dati per i messaggi tra client e server
- *Record* : struttura dati che contiene i libri della biblioteca
- *argt* : struttura dati condivisa tra i thread

unboundedqueue.h:

- *Nodet* : Nodo della coda
- *Queuet* : Coda condivisa

Uso una struttura apposita per il messaggio in modo da poter comunicare tutti i campi che sono richiesti dal server per elaborare correttamente la richiesta. Inoltre creo anche una struttura per contenere i record che vengono elaborati dalla funzione *ReadRecordFromFile* all'interno del file **recordutils.c.** Per riuscire ad elaborare i correttamente i file bib.txt forniti come esempio ho usato il pattern con cui venivano scritti i record

nomecampo:valore;

Una volta fatto ciò ho potuto usare la funzione *strtok* mentre eseguivo la lettura del file per poter riconoscere un campo e il suo valore associato in modo da inserirlo nella struttura dati. Uso come delimitatori i caratteri **";"** e **":"** posso così distinguere ogni campo da ogni valore. Facendo in questo modo posso avere il completo controllo dei dati all'interno del file bib.txt nel mio programma e questo mi consente di fare tutte le operazioni richieste.
Per poter poi cercare un determinato record quando arriva un messaggio dal client uso la funzione *SearchRecord* che cerca se un record è presente nella struttura e vede se il prestito è scaduto o non è presente.In questa funzione scorro tutti i record della struttura e se viene richiesto un prestito controllo se il libro è disponibile (cioè ha passato la data di inizio prestito da almeno 30 giorni o 30 secondi per testarlo oppure se il campo prestito non esiste). Il valore di ritorno di questa funzione è un array dove ogni posizione corrisponde alla posizione di un record nella biblioteca e metto 1 in

posizione i se il record i deve essere poi inviato al client perchè verifica le condizioni della richiesta fatta al server. Le funzioni *RebuildRecord* e *ProcessRequestmsg* servono rispettivamente ,la prima per costruire una stringa composta dai campi della struttura che ci consentirà poi dopo di scriverla all'interno del file txt per apportare le modifiche durante l'esecuzione, mentre la seconda serve per processare il messaggio ricevuto dal client per poi poter esaminare la struttura per la ricerca di quella determinata richiesta.
Passando invece alle librerie le funzioni più importanti sono *sainit* e *Socket* all'interno del file **mysocket.c**. La prima viene usata per inizializzare la socket con porta, tipo di socket e indirizzo. Soffermandoci sull'indirizzo questa funzione fa in modo che, nel caso la utilizzi il server e quindi quando crea la socket deve scegliere un qualsiasi indirizzo libero sulla macchina(INADDRANY), invece se la usa il client puo' associare l'indirizzo del server letto direttamente dal file di configurazione. La funzione *Socket* oltre a creare la socket fa anche un controllo sul bind usando *setsockopt* che imposta due opzioni del socket (SOREUSEADDR e SOREUSEPORT) che consentono il riutilizzo di un indirizzo e di una porta.
Nel file **timeutils.c** controllo con le sue funzioni che la data attuale quando avvio il programma sia maggiore della data del prestito della biblioteca in modo da decidere se il prestito andrà a buon fine o meno. Faccio uso del metodo *mktime* per ottenere il formato usato nei file bib.txt e compararlo con la data.

All'interno di **bibserver.c** il *worker* usa un ciclo infinito e fa si che i thread continuino ad eseguire il loro compito finchè non vengono terminati (SIGINT o SIGTERM). Il thread riceve un messaggio tramite la funzione *Recv* dal client e dopo controlla se ha ricevuto un numero di byte maggiore di 0 cosi da poter poi fare le sue operazioni, oppure se i byte sono uguali a 0 aggiorno il massimo tra i client(con la funzione $UpdateMax$) e libero il file descriptor.
Nel caso maggiore di 0 acquisisco il lock perchè deve processare il messaggio del client, cercarlo all'interno della struttura e poi scrivere all'interno del file di log quello che ha trovato (se ha trovato qualcosa), poi manda il messaggio indietro al client con *Send*. Per scrivere sul file di log serve l'accesso esclusivo sennò puo succedere che più thread voglino contemporaneamente scrivere sullo stesso file e si verificherebbero problemi di concorrenza con le operazioni di scrittura che possono sovrapporsi ed essere in-

consistenti. Finito l'invio dei dati mando un ultimo messaggio al client *MSGFINISHED* che permette di capire al client quando fermare l'ascolto dei messaggi dal server. Passando al main invece dopo avere inizializzato tutto e creato i thread passo al *while* dove mi metto ad ascoltare sulla socket con la *Select* e quando arriva un client viene settato il suo file descriptor e aggiornato l'$fdmax$, poi viene messo in coda in modo che poi possa essere utilizzato. Il while poi termina l'accettazione dei client quando arriva un segnale tra quelli settati nel *SignalHandler*.Finito il ciclo arrivo alla terminazione del programma dove metto in coda tanti -1 quanti sono i thread che uso come terminazione dei thread, andando poi a controllare nella funzione il valore estratto e se corrisponde a -1 finisco la sua esecuzione. Dopo viene chiamata la $join$ per aspettare che tutti i thread terminino e poi libero le risorse.

Nel file del client **bibclient.c** uso uno *structoptionlongoptions*[] per poter elencare le opzioni accettate dal programma, insieme ai loro requisiti (se richiedono o meno un argomento) e.g

"p", noargument, NULL, 'p'

vuol dire che il prestito non richiede un argomento ma solo - -p). Viene utilizzato - *-p* al posto di -p per via di getoptlong che accetta solo la doppia linea. All'interno del ciclo, utilizzo la funzione *getoptlong* per estrarre l'opzione corrente e il suo argomento (se presente) dall'array argv. Utilizzo un costrutto switch-case per gestire ciascuna opzione in base al suo identificativo (ritornato da getoptlong). Per ciascuna opzione, eseguo le azioni corrispondenti (e.g –autore, aggiunge la stringa "autore:" seguita dall'argomento alla stringa buff) Una volta processate tutte le opzioni avrò costruito una stringa dello stesso formato di un record da poter inserire in *message.data* per poi poterlo inviare al server. Poi leggo il file di configurazione e faccio un for per inviare e ricevere messaggi da ogni server attivo (mi fermo e passo al server successivo quando arriva un messaggio di tipo $MSGFINISHED$). Il file bash *bibaccess* gestisce l'analisi di file di log verificando passo passo che i record scritti all'interno siano del formato giusto(e.g nomecampo:valore; oppure che iniziano tutti con autore) e inoltre controlla anche che il numero effettivo dei record sia uguale al numero scritto in fondo al file. Alla fine se la variabile $validformat$ è true vuol dire che ha passato tutti i controlli e posso stampare il risultato della verifica.

Problemi incontrati e soluzioni adottate

Per gestire i segnali all'interno del server, ho utilizzato *sigaction* insieme a una variabile di tipo *volatilesigatomict*. Quando viene ricevuto un segnale, l'handler associato imposta questa variabile a 1. Nel ciclo di accettazione dei client nel main, se la variabile è stata impostata a 1, il ciclo termina. Questo meccanismo consente al server di reagire ai segnali in modo efficace e di interrompere le operazioni in corso quando necessario.

Un'altro problema che ho incontrato è condividere le informazioni di connessione dai server ai client e settare le porte. Essendo che l'indirizzo usato dai server sarà lo stesso per ogni server, la porta dovrà essere diversa tra ognuno di essi così da permettere a tutti di ricevere i dati. Prima generavo randomicamente una porta da 1111 a 9999 ma essendo che i server venivano avviati tutti nello stesso momento la porta diventava sempre la stessa per tutti e cercavano di creare il file. Inoltre anche sistemando questo problema un server non conosceva le porte degli altri quindi quella porta poteva essere gia stata scelta. Per risolvere questi problemi ho innanzi tutto impostato una **sleep** di 1 secondo fra un server e l'altro all'avvio (nei test del Makefile). Questo mi consente la corretta esecuzione della funzione $ReadConfFile$ all'interno del server dove genero una porta random e controllo se nel file di configurazione esiste gia un server con quella porta e in tal caso la genero ancora finchè non diventa unica.
Inoltre si è presentato un altro problema con la *Recv* che dava l'errore *Resource temporarily unavailable* (che corrisponde a EAGAIN), e quindi l'operazione di ricezione dati si bloccava. Per risolvere il problema nella funzione Recv controllo se la richiesta non va a buon fine per un errore *EAGAIN o EWOULDBLOCK* e nel client faccio un ciclo dove nel caso questo accade aspetto un po e poi riprovo a fare la Recv. Inoltre uso anche questo codice nel main del server:

```
1 int flags = fcntl(serverSocket, FGETFL, 0);
2 fcntl(serverSocket, FSETFL, flags | ONONBLOCK);
```

L'impostazione $ONONBLOCK$ con la funzione fcntl rende la socket non bloccante sia sul lato del server che su quello del client. Questo significa che sia send che recv su entrambi i lati (server e client) non bloccheranno il programma se non ci sono dati pronti per la lettura o se il buffer di scrittura è pieno.