Relazione progetto SOL

Tommaso Macchioni

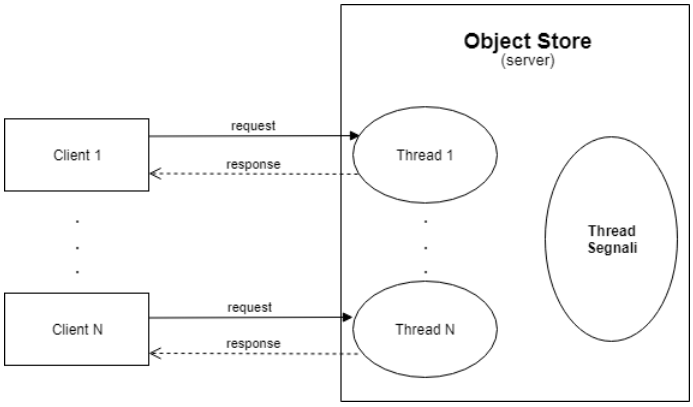
2019

Indice

1 Introduzione

Per realizzare l'*object store* implementato come sistema client-server, si è scelto di adottare un server multi-threaded, il quale, per ogni client, creerà un thread che si prenderà carico delle sue richieste.

Affinché il thread principale possa soddisfare le richieste di connessione da parte di numerosi client, la ricezione e quindi la gestione dei segnali, è stata in parte delegata ad un thread specifico.



2 Lato Server

2.1 Strutture dati e variabili globali

Sono state utilizzate due *strutture dati*:

- **Hash Table.** Una tabella hash per tenere traccia dei client attualmente connessi. La *chiave* è il nome con cui il client si è registrato mentre come *valore* il file descriptor ad esso associato.
- **Statistiche.** Una struttura che contiene la sopra citata *Hash Table* e altre variabili per la memorizzazione di dati statistici quali il numero massimo di client accettati, e il numero di client connessi e di errori riscontrati.

La necessità di condividere queste strutture tra i diversi thread, ha reso obbligatorio l'utilizzo di meccanismi per garantire la mutua esclusione.

Infine si è utilizzato una *variabile globale* (`quit`) come flag per fare in modo che, qualora ci sia necessità di terminare l'intero processo, i thread possano "saperlo", completare eventuali operazioni in corso e avviare la procedure di chiusura.

2.2 Thread principale (main)

Il *thread principale* ha il mero compito di attendere nuove richieste di connessione da parte di clients affidando loro un thread e di attendere eventuali messaggi da parte del thread dei segnali tramite una pipe senza nome.

Quest'ultimo non farà altro che scrivere nella pipe il tipo di segnale ricevuto e sarà il thread principale ad agire di conseguenza.

Prima di mettersi in ascolto infinito, il thread principale effettua, dunque, delle operazioni preliminari:

- Registrazione della funzione `cleanup` tramite `atexit()` per effettuare la deallocazione delle strutture dati;
- Mascheramento dei segnali di `SIGINT`, `SIGQUIT`, `SIGTERM` e `SIGUSR1`;
- Ignorazione del segnale `SIGPIPE`;
- Creazione di una pipe senza nome per comunicare col thread dei segnali;
- Creazione del thread dei segnali;
- Inizializzazione delle strutture dati.

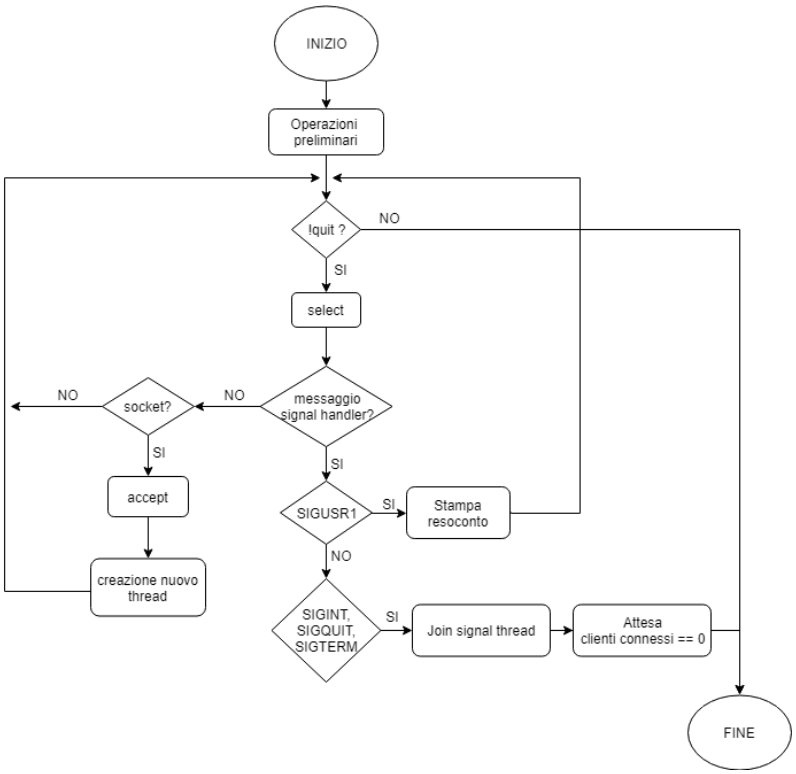
Al fine di evitare un'attesa infinita sulla funzione di `accept` e quindi di poter ricevere eventuali messaggi da parte del thread dei segnali, si è scelto di utilizzare le funzionalità fornitaci dalla `listen`, impostando preventivamente un timeout arbitrario.

Essenzialmente, il `main`, nel suo ciclo infinito, si mette in attesa sulla `listen` per un certo periodo di tempo, dopodiché controlla su quale *file descriptor* (fd) è avvenuta un'attività (scrittura o lettura). Se il fd è quello relativo alla socket si crea un nuovo thread (*detached*), se invece il fd è quello relativo alla pipe, si gestisce immediatamente il segnale.

La tipologia del segnale eventualmente ricevuto determina le operazioni svolte:

- `SIGINT`, `SIGQUIT`, `SIGTERM`: si setta `quit` ad 1, si attende la terminazione del thread dei segnali e si esce dal ciclo in modo da terminare correttamente il processo.
- `SIGUSR1`: si chiamata la funzione non di sistema `lsR` che stampa su standard output alcune informazioni di stato del server e poi si continua il ciclo.

Per terminare correttamente il processo, precedentemente alla attivazione della funzione di `cleanup`, il server rimane in attesa su una *variabile condizione* (interna alla struttura dati per le statistiche) fintanto che il numero di client connessi non è pari a zero.



2.3 Thread dei segnali

Il thread incaricato di gestire i segnali per l'intero processo, come precedentemente detto, viene creato dal thread principale. Per intercettare i segnali si è fatto uso della funzione `sigwait`. Se il segnale ricevuto corrisponde a `SIGINT`, `SIGQUIT`, `SIGTERM` o `SIGUSR1`, lo si comunica al server tramite la pipe ed eventualmente si avvia la procedura di chiusura.

2.4 Thread dei client e gestione degli errori

Ogni thread è connesso a un solo client e insieme, rappresentano il fulcro dell'intera comunicazione client-server.

Dopo aver opportunamente mascherato i segnali, anche in questo caso, all'interno del ciclo infinito, si fa uso della `select` con timeout, per evitare un'attesa infinita sulla funzione di `read` utilizzata per la lettura della *request* da parte del client.

Per ogni richiesta e quindi tipologia di *header* ricevuto, il thread opera di conseguenza:

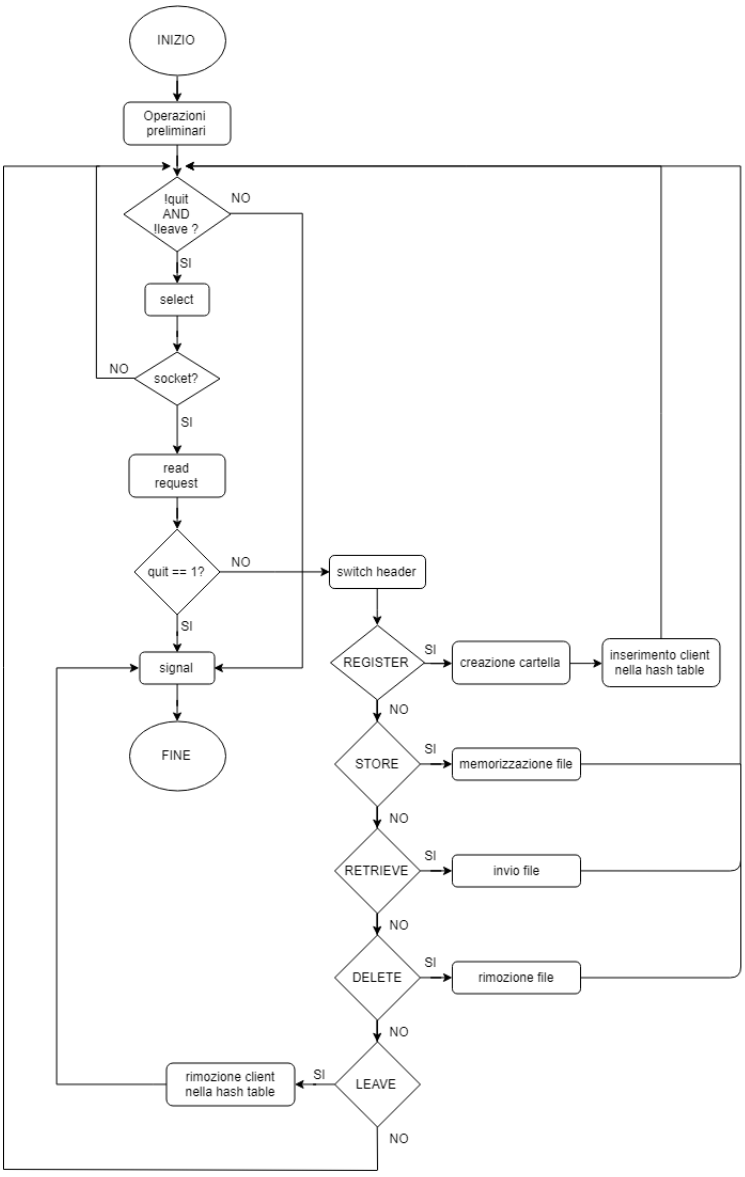
- `REGISTER`. Si chiama la funzione `register_fnct` nella quale si verifica che il nome fornitoci non sia già presente come chiave all'interno della *hash table* (che non sia già connesso) ed eventualmente si crea una cartella a suo nome e lo si inserisce nella tabella tramite la funzione `icl_hash_insert`.
- `STORE`. Si chiama la funzione `store_fnct` nella quale si estrapola il nome del file da salvare e la sua lunghezza in bytes. Se il dato rientra nella prima `read` effettuata, si salva immediatamente, altrimenti si itera fintanto che non siano stati ricevuti il numero di bytes dichiarati nel messaggio.
- `RETRIEVE`. Si chiama la funzione `retrieve_fnct` nella quale si verifica che il file richiesto esista ed eventualmente, dopo aver estrapolato la grandezza del file tramite la struttura di sistema `stat`, lo si spedisce come messaggio al client.
- `DELETE`. Si chiama la funzione `delete_fnct` nella quale si verifica che il file richiesto esista ed eventualmente lo si elimina.
- `LEAVE`. Si elimina il client dalla *hash table* tramite la funzione `icl_hash_delete` e si esce dal ciclo.

Precedentemente la chiusura, ci assicuriamo che il client sia stato eliminato dalla tabella hash, chiudiamo il fd della socket e infine inviamo una `signal` sulla variabile condizione utilizzata dal thread principale in modo da farlo terminare e lasciare l'object store in uno stato consistente.

In ogni funzione e all'interno del ciclo del thread, tutti gli eventuali errori, a parte quelli che richiedono la terminazione tempestiva del processo, sono inviati al client e non terminano l'esecuzione del thread. Un messaggio di errore è della forma: `KO errno`. Dove `errno` rappresenta la variabile intera definita nell'header file `errno.h`.

Inoltre vanno sottolineate alcune sottili scelte implementative:

- In ogni funzione sopracitata viene controllato che il messaggio ricevuto da parte del client sia conforme al protocollo;
- La disconnessione da parte di un client non comporta la cancellazione della propria cartella;
- Non possono esser connessi più utenti col solito nome;
- Le operazioni di `STORE`, `RETRIEVE`, `DELETE` e `LEAVE` non possono essere effettuate se il client non ha prima eseguito la connessione tramite la `REGISTER`.



3 Lato client

Un client, per poter comunicare con l'*objectstore* (server), utilizzerà le funzioni fornite dalla libreria implementata.

Ogni funzione di libreria, prima di svolgere le operazioni richieste, valuta che i campi passati come argomenti siano validi, dopodiché comunica col server.

Nel caso in cui si riceva un messaggio di errore da parte del server, viene settata una variabile speciale, interna alla libreria, denominata `myerrno`. L'introduzione di questa variabile si è resa necessaria al fine di poter tenere traccia dell'ultimo errore riscontrato lato server evitando di compromettere la variabile di sistema `errno`.

Al fine di testare la correttezza dell'intero codice è stato realizzato un client di test che, dopo essersi connesso tramite la `os_connect` e prima di effettuare la disconnessione tramite la `os_disconnect`, effettua una delle seguenti operazioni, sulla base della tipologia di test richiesta:

- **Test 1.** Memorizzazione di 20 files distinti:
  - 10 contenenti array di interi di grandezza crescente,
  - 10 contenenti una frase ripetuta in ordine crescente.
- **Test 2.** Richiesta di recupero di 3 files distinti, di cui 1 non esistente al fine di testare anche i casi di errore.
- **Test 3.** Eliminazione di 4 files distinti, di cui 2 non esistenti al fine di testare, nuovamente, anche i casi di errore.

Per stampare su standard output un breve rapporto sull'esito dei test, è stata implementata una struttura dati che tiene conto del nome del client ed il numero di operazioni totale e di ogni tipologia e inoltre, è stata realizzata una funzione `print_log` che stampa, per ogni operazione effettuata, una stringa in stile *Common Log Format*:

```
nome_client data richiesta esito
```

4 Makefile, file di log e script

Il Makefile realizzato, oltre ad avere i target `all` per generare l'eseguibile e `clean` e `cleanall` per ripulire la directory, presenta il target `test`. Questo, dopo aver aver avviato l'eseguibile relativo all'objectstore, esegue lo script `testsum.sh`.

Il file di script `testsum.sh` esegue in ordine le seguenti operazioni:

1. Esecuzione contemporanea di 50 istanze di client che effettuano il *Test 1*,
2. Invio del segnale `SIGUSR1` al server, interposto all'esecuzione dei client precedenti,
3. Attesa terminazione clients,
4. Esecuzione contemporanea di 50 istanze di client precedentemente registrati, di cui 30 effettuano il `Test 2` e 20 il `Test 3`,
5. Attesa terminazione clients,
6. Invio di un secondo segnale `SIGUSR1` al server,
7. Invio di un segnale di terminazione `SIGQUIT` al server,
8. Interpretazione e conseguente stampa su standard output.

5 Struttura del programma

I principali sorgenti che compongono l'intera implementazione dell'objectstore comprendono:

- `objectstore.c`

  Intero server.

- `icl_hash.c` e `icl_hash.h`

  Implementazione ed interfaccia della Hash Table.

- `client.c`

  Client di test.

- `client_conn.c` e `client_conn.h`

  Implementazione ed interfaccia della libreria lato client.

- `conn.h`

  Definizioni di macro utilizzate sia lato server che client.

- `utils.c` e `utils.h`

  Funzioni di utilità utilizzate sia lato server che client.

Infine:

- `Makefile`
- `testsum.sh`

6 Test Aggiuntivo

Per verificare ulteriormente la correttezza del programma è stato realizzato un secondo test. All'interno della cartella `Test Aggiuntivo` è possibile eseguire un secondo `Makefile`. E' necessario lanciarlo prima col target `all` e successivamente col target `test`.

Il test consiste nella registrazione di un client denominato `Musicista` (quindi aggiungere una cartella a suo nome), nella lettura dell'immagine `bbking.jpg` e successivamente nella memorizzazione di questa, nello spazio a lui riservato.

7 Sistemi operativi testati

I S.O. utilizzati per testare la correttezza del programma sono stati i seguenti:

- Ubuntu 18.04 LTS
- Xubuntu 14.10
- Peppermint 9