Relazione Progetto SOL

Alessandro Puccia

a.a 2018/2019

1 Introduzione

Lo scopo dell'Object Store è quello di ricevere dai client delle richieste di memorizzazione, recupero e cancellazione di blocchi di dati dotati di nome, detti "oggetti". L'Object Store memorizza, all'atto della registrazione, una cartella con lo stesso nome del client in cui verranno conservati tutti i suoi file.
I client interagisce con il server che implementa l'Object Store attraverso l'utilizzo di una libreria che fornisce tutte le funzionalità necessarie per la creazione, scrittura e lettura degli header di comunicazione.
Tutto il codice è stato compilato, linkato e testato con successo su Ubuntu 18.04 e sulla macchina virtuale del corso.

2 Librerie

Vengono implementate le seguenti librerie:

- `libosclient.a`: utilizzata dai client, permette di inviare le richieste al server quali: registrazione, memorizzazione, cancellazione, recupero e disconnessione.
- `libserverfuns.a`: utilizzata dal server per poter interagire con il file system. E' stata implementata come libreria statica perchè l'utilizzatore è soltanto il server.

2.1 `libosclient.a`

La libreria lato client è composta da due file di implementazione, il primo chiamato `osclient.c` fornisce le seguenti funzioni:

1. `char* os_register(char *name)`
   Permette al client di connettersi e registrarsi all'object store con il nome `name`.
   Ritorna 1 se la connessione e la comunicazione hanno avuto successo; 0 in caso di errore dovuto alla comunicazione, argomento non valido o se viene riutilizzata senza una precedente `os_disconnect`. Vengono effettuati al massimo 10 tentativi di connessione
2. `int os_store(char *name, void *block, size_t len)`
   Permette al client di memorizzare un oggetto con nome `name`.
   Ritorna 1 se la comunicazione e la memorizzazione hanno avuto successo; 0 in caso di errore dovuto alla comunicazione, argomenti non validi, problemi nella memorizzazione o se il client non risultava registrato.

3. `void *os_retrieve(char *name)`
   Permette al client di recuperare un oggetto precedentemente memorizzato con nome `name`.
   Ritorna 1 se la comunicazione e il recupero hanno successo; 0 in caso di errore dovuto alla comunicazione, problemi nella cancellazione, argomenti non validi o se il client non risultava registrato.

4. `int os_delete(char *name)`
   Permette al client di cancellare un oggetto precedentemente memorizzato con nome `name`.
   Ritorna 1 se la comunicazione e la cancellazione hanno successo; 0 in caso di errore dovuto alla comunicazione, problemi nella cancellazione, argomenti non validi o se il client non risultava registrato.

5. `int os_disconnect()`
   Permette al client di disconnettersi dall'object store.
   Ritorna 1 se la comunicazione ha successo, 0 in caso di errore dovuto alla comunicazione o se il client non risultava registrato.

Per la gestione degli errori viene utilizzata una variabile globale che, oltre a inglobare gli errori dovuti alle chiamate di libreria/sistema, viene utilizzata per gli eventuali messaggi di errore spediti dal server.
E' stata introdotta, un'ulteriore funzione `char *check_response(char *buffer, ...)` che si occupa semplicemente di effettuare il parsing della risposta ricevuta dal server.
Il secondo file chiamato `sktcomm.c` contiene le funzioni che permettono la creazione, scrittura e lettura degli header del protocollo (lo stesso file verrà compilato e linkato insieme al file `server.c`):

1. `ssize_t readn(int fd, void *block, size_t size)`
   Permette la lettura dal socket identificato da `fd` dei dati che saranno conservati in un buffer puntato da `block`.
   Ritorna la dimensione dei dati letti se la lettura ha avuto successo, -1 in caso di errori nella lettura, 0 in caso di una disconnessione dall'altro lato della comunicazione.

2. `ssize_t writen(int fd, void *block, size_t size)`
   Permette la scrittura nel socket identificato da `fd` dei dati puntati da `block`.
   Ritorna la dimensione dei dati scritti se la scrittura ha avuto successo, -1 in caso di errori nella scrittura, 0 in caso di una disconnessione dall'altro lato della comunicazione.

3. `char *init_header(char *h_type, size_t *nbytes, ...)`
   Permette la creazione di tutti gli header specificati dal protocollo e identificati da `h_type`.
   Ritorna un puntatore all'header appena creato assegnando a `*nbytes` il numero di byte scritti se la creazione ha avuto successo; NULL altrimenti.

4. `int read_header(int fd, char **buffer)`
   Permette di leggere dal socket identificato da `fd` un header del protocollo fino al carattere di newline che verrà scritto in un blocco puntato da `*buffer`; la lettura avviene al singolo byte, ciò comporta un notevole overhead ma permette sia ai client che al server di poter effettuare le scritture/letture dei byte effettivamente occupati dall'header.
   Ritorna 1 in caso di successo, -1 in caso di errori nella lettura, 0 in caso di disconnessione dall'altro lato della comunicazione.

2.2 `libserverfuns.a`

1. `char *create_client_dir(char *client_name, size_t *dir_pathdim)`
   Permette al server di creare, se non esiste, la directory dedicata al client.
   Se ha avuto successo ritorna il pathname della directory, assegnando a `*dir_pathdim` la dimensione del path , NULL altrimenti.

2. `int create_client_file(char *file_name, size_t file_size, char *data, char *dir_pathname, size_t dir_pathdim)`
   Permette al server di creare un nuovo file su cui verrà scritto l'oggetto puntato da `data`, nel caso in cui già esisteva un file con nome `file_name` allora lo stesso viene azzerato e riscritto.
   Se la creazione ha avuto successo e il file precedentemente non esisteva ritorna 0, altrimenti ritorna la dimensione del vecchio file; se la creazione non ha avuto successo ritorna -1.

3. `int delete_client_file(char *file_name, char *dir_pathname, size_t dir_pathdim)`
   Permette al server di cancellare il file con nome `file_name`.
   Ritorna la dimensione del file se la cancellazione ha avuto successo, -1 altrimenti.

4. `char *retrieve_client_file(char *file_name, size_t *file_size, char *dir_pathname, size_t dir_pathdim)`
   Permette al server di recuperare il file con nome `file_name`.
   Ritorna un puntatore ai dati letti assegnando a `file_size` la loro dimensione se il recupero ha avuto successo, NULL altrimenti.

3 Object Store

3.1 `server.c`: Inizializzazione

Al momento dell'avvio del server vengono bloccati con una maschera i segnali SIGQUIT, SIGTERM, SIGINT, e SIGUSR1, il segnale SIGPIPE viene ignorato.
Viene inizializzata una struttura dedicata alle statistiche del server le cui principali componenti sono:

- il numero di client connessi.
- il massimo numero di client connessi.
- il numero di oggetti contenuti nell'Object Store.
- la dimensione totale dell'Object Store.

Le statistiche sono accompagnate da una lock per poterne garantire l'aggiornamento in mutua esclusione e da un condition variable usata durante la fase di terminazione. Nel caso in cui la cartella `data` fosse già presente all'avvio, viene utilizzata la chiamata di sistema `ftw` per poter aggiornare le statistiche in base ai file già presenti in precedenza.
La gestione dei segnali viene effettuata da un thread apposito che si metterà in attesa attraverso `sigwait` sui segnali specificati dalla maschera passata come argomento, si occuperà della stampa delle statistiche in seguito alla ricezione del segnale SIGUSR1 e della comunicazione al thread main e ai thread worker se si è ricevuto uno degli altri segnali; per far ciò viene utilizzata una pipe il cui fd lato scrittura verrà chiuso dal thread gestore se si riceve uno dei segnali specificati prima. Si è preferito utilizzare un thread dedicato in modo da poter svegliare in maniera più efficiente gli altri thread senza utilizzare un timer sulla `select` che provocherebbe dei risvegli inutili sia del thread

main che dei worker.
Successivamente viene creato il socket "passivo" su cui il main attenderà le nuove connessioni.

3.2 `server.c`: Gestione delle connessioni

Il thread main rimane in attesa di lettura con una `select` sui due principali fd: quello della pipe e quello "passivo". In caso di una nuova richiesta di connessione, viene lanciato un thread worker con argomento il fd dedicato al client; esso si occuperà interamente delle sue richieste. Se il bit relativo al fd della pipe lato lettura, specificato nella maschera al momento dell'inizializzazione, risulta settato si procederà alla terminazione.

3.3 `worker.c`: Gestione delle richieste

Ogni thread worker rimane in attesa di lettura con una `select` sul fd della pipe e sul fd dedicato al client. A seguito di una richiesta da parte del client il worker si interfaccerà con il file system attraverso la libreria `serverfuns.a` e provvederà all'aggiornamento delle statistiche del server. In caso di fallimento viene spedito un messaggio che indica brevemente l'errore.
Notare inoltre che, essendo garantita l'univocità dei nomi dei client, non è stato necessario mantenere eventuali strutture condivise che permettono la memorizzazione dei nomi dei client e degli id dei thread a loro dedicati.

3.4 Terminazione

In caso di errori gravi (allocazioni di memoria, scrittura/lettura su socket) il thread che ha ricevuto l'errore termina l'esecuzione provvedendo ad effettuare un cleanup della memoria allocata, la terminazione dei thread worker è prevista anche nel caso di fallimento dovuto a una richiesta di connessione. Se si richiede la terminazione del server a seguito di un segnale, mentre uno o più thread worker stanno seguendo una richiesta, essa verrà completata e successivamente i thread provvederanno alla terminazione aggiornando i contatori dei client.
La terminazione del thread main avviene solo quando tutti i thread worker e il gestore dei segnali terminano l'esecuzione in modo da evitare memory leak. Per garantire questo, viene utilizzata una condition variable su cui il main si mette in attesa fino a quando il numero di client ancora connessi è maggiore di zero. Non è stata utilizzata `pthread_exit` per poter garantire il cleanup della memoria allocata per la lock e la condition variable in maniera thread-safe.

4 `client.c`

Il client che verrà lanciato per effettuare i tre test specificati nel testo provvederà, oltre alla connessione e alla disconnessione dal server, alla memorizzazione di venti stringhe che verranno ripetute più volte, in modo da effettuare delle `os_store` di dati di grandezza crescente, partendo da 100 byte fino ad arrivare a 100000 byte.
I risultati vengono stampati nel file `testout.log` che includerà, oltre alla data del test, il nome del client, il tipo di test che è stato effettuato e l'esito: successo se e solo se tutte le venti operazioni sono andate a buon fine; fallimento altrimenti, specificando il numero di operazioni fallite.

5 `testsum.sh`

Con questo script bash si analizza il contenuto di `testout.log`; viene stampato sullo `stdout` il numero totale di client che si sono connessi e, per ogni batteria di test, il numero di client che hanno superato/fallito il test.
Nel caso di fallimento viene stampato il nome del client e il numero di operazioni fallite.