Progetto Object Store
Relazione

Giuseppe Bisicchia

559033

Sommario

In questa relazione si andranno ad analizzare le principlali scelte progetturali e l'architettura generale dell'*Object Store*. Si analizzerà dapprima la suddivisione in moduli del codice, dandone così una visione generale. Successivamente, si descriverà l'architettura e le principali caratteristiche del *server*. Infine si tratterà la gestione della fase di testing.

Indice

1 Package

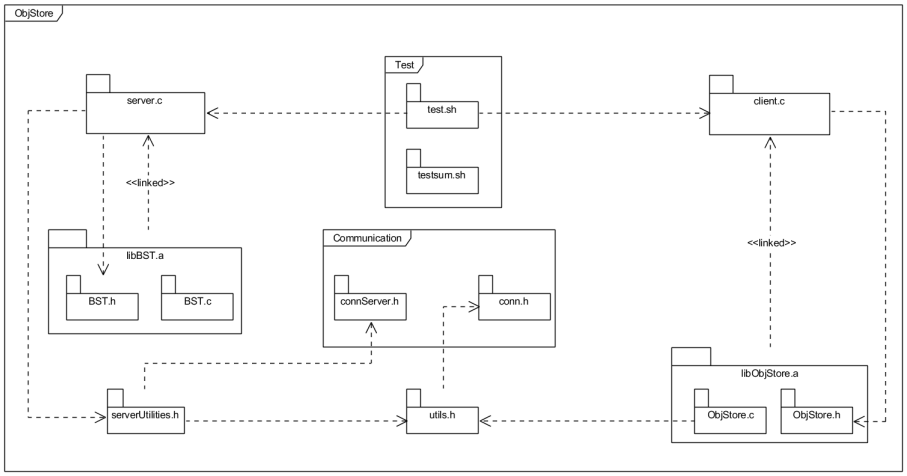

Figura 1: Object Store: UML Package Diagram

Segue una descrizione dei vari moduli che compongono l'*Object Store*:

- `utils.h`
  Contiene procedure e macro per la gestione degli errori generabili da chiamate di sistema e alcune macro di base utilizzate in più moduli.
- `serverUtilities.h`
  Contiene le procedure che permettono al server l'interazione con il *File-System* per l'implementazione dei servizi di *store*, *delete* e *retrieve* offerti dal server.
- `conn.h`
  Implementa le funzioni per la lettura e scrittura sulla socket.
- `connServer.h`
  Specializza le funzioni per la lettura sulla socket da parte del server.
- `server.c`
  Contiene le funzioni per la realizzazione del server, tra le più importanti:
  - `sigHandler`: Thread delegato alla gestione dei segnali.
  - `worker`: Thread che interagisce con un client e ne realizza le richieste.
  - `endWorker`: Termina nel modo corretto un Worker.
- `client.c`
  Realizza il client di test. Vi è stata aggiunta la possibilità di lanciare il client anche con il parametro 0 che esegue in successione sia le store che le retrieve e le delete.

- `ObjStore.h & ObjStore.c`
  Realizza la libreria `libObjStore.a` che permette a un client di interagire con il server.
- `BST.h & BST.c`
  Implementa un *Binary Search Tree*.
- `testsum.sh`
  Elabora le informazioni contenute in *testout.log* e stampa a video l'esito del test.
- `test.sh`
  Genera i client per l'esecuzione del test.

2 Server

Il server è l'elemento centrale di tutto l'*Object Store* e per questo gli viene dedicata una sezione a parte. Per il suo corretto funzionamento ha bisogno che gli venga agganciato in fase di linking la libreria *libBST.a*: questa implementa un `Binary Search Tree` che viene utilizzato per controllare quali utenti sono on-line in un dato istante. Ogni utente infatti ha una cartella dedicata, cartella che è identificata da un *i-node*, quando un utente è quindi on-line il valore dell' *i-node* della sua cartella viene inserito nel `BST`, in modo che in caso un altro client si voglia registare con lo stesso nome utente, l'accesso gli venga impedito. Quando un utente si disconnette l'i-node relativo viene eliminato dal `BST`. Il server viene implementato mediante tre tipologie di thread, chiamate main, sigHandler e worker, nella prossima sezione verranno analizzati singolarmente.

2.1 Architettura

Si andrà ora ad analizzare l'architettura del server. In figura 2 un diagramma UML descrive come i vari thread interagiscono tra di loro. È importante notare come sia il `sigHandler` che i vari `worker` venagono creati in modalità *detached*. Inoltre ereditano dal main una maschera che copre tutti i segnali.

2.1.1 `sigHandler`

Questo thread di occupa ricevere i segnali e gestirli opportunamente, per farlo si mette in attesa attraverso una `sigwait`. Quando arriva un segnale diverso da `SIGUSR_1` viene settata a 1 la variabile `sig_flag`, variabile che segnala a tutti i thread di concludere le operazioni in corso e terminare. In caso del segnale `SIGUSR_1` viene chiamata la procedura `debugStatistics()` che si occupa di fornire un report sullo stato attuale del server, inoltre riporta varie informazioni sullo storico del server: queste vengono memorizzate attraverso delle variabili che contengono valori riguardanti il numero di operazioni per tipo, quante sono fallite, quanti dati sono stati memorizzati, quanti inviati...

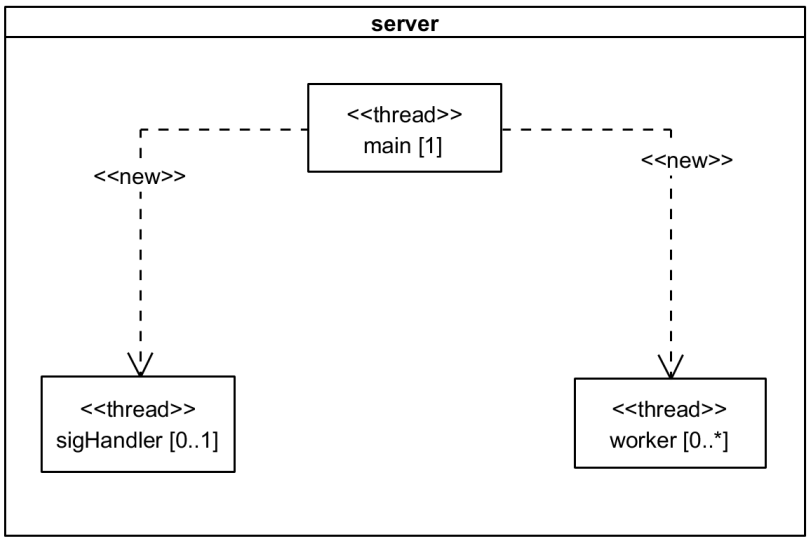

Figura 2: Server: UML Composite Structure Diagram

2.1.2 `main`

Il `main` è il primo thread ad essere creato e l'ultimo a terminare. Si occupa di creare tutti gli altri thread e di chiudere il server. Il suo compito è infatti quello di generare il `sigHandler`, inizializzare il `BST` e aprire la comunicazione sulla `socket`. Ogni qual volta poi un nuovo client cerca di connettersi, il `main` crea un nuovo thread worker che si occuperà di tutte le interazioni. Quando `sig_flag` viene settata, il thread smette di restare in attesa di un nuovo client e attende che tutti gli altri thread terminino, a quel punto termina anche la sua esecuzione. Per sapere quando tutti gli altri thread sono terminati inizia a ciclare in attesa che la variabile `tasks` diventi non positiva. Infatti ogni thread, ad eccezione del `main`, quando viene creato incrementa la variabile `tasks` e quando termina la decrementa.

2.1.3 `worker`

Questo thread si occupa di interagire con un determinato client su un particolare *file descriptor*. In caso di errori non inerenti alla comunicazione il thread invia un messaggio di KO e passa ad attendere la prossima istruzione, solo in caso di errori in lettura o scrittura sul *file descriptor,* viene terminato prematuramente. La terminazione normale del thread avviene quando `sig_flag` viene settata. La terminazione, in ogni caso, viene gestatita dalla procedura `endWorker`.

`endWorker` La funzione riceve in input un *file descriptor*, l'indice di un *i-node*, un puntatore ad una zona di memoria allocata sullo heap, una stringa contenente un messaggio di errore il numero della riga in cui si è verificata la chiamata alla

funzione e il file in cui si è verificata. Questa funzione si occupa di decrementare la variabile `tasks` chiudere il *file descriptor*, rimuovere l'indice dell' *i-node* dal `BST`, deallocare la memoria e in caso stampare un messaggio di errore.

3 `connServer.h`

Per permettere al server una corretta comunicazione con i client e al contempo garantire la terminazione dei workers secondo le specifiche, è stato necessario creare delle funzioni ad hoc.

3.1 `int waitFd(int fd)`

Questa funzione è fondamentale per l'implementazione del server: si mette in attesa che ci sia qualcosa da leggere sul *file descriptor* passato come argomento, l'attesa però non è bloccante: vi è, infatti, un timeout periodico che permette di verificare se la variabile `sig_flag` sia stata settata e in caso uscire dall'attesa. Questo permette al server di non sopsendersi indeterminatamente in lettura su un *file descriptor* non uscendone anche se il flag è stato settato.

3.2 `char* readnServer(int fd, void* buf, size_t size)`

Può essere intesa come una specializzazione della funzione `readn` contenuta in `conn.h`. Si comporta esattamente come quella, ad eccezione che non si mette in attesa direttamente sulla `read` ma chiama `waitFd` e in caso `sig_flag` venga settata esce immediatamente segnalandolo.

3.3 `char* readMsgServer(int fd)`

Legge un carattere alla volta fino a quando non trova un '\n' a quel punto restituisce la stringa letta. Se `sig_flag` è stata settata termina immediatamente restituendo `NULL`.

4 Testing

Si andrà ora a discutere di come è stato realizzata l'architettura per il testing del progetto.

4.1 `make test`

`make test` prepara l'ambiente di esecuzione ripulendolo, eventualmente, dalla cartella `data` e dal `objstore.sock`, lancia un server in background e avvia test.sh che lancerà prima 50 client con l'opzione 1, terminati questi ne lancia 30 con l'opzione 2 e 20 con l'opzione 3. L'output dei client verrà rediretto sul file `testout.log`. il formato dell'output di ciascun client è il seguente:
REPORT numeroOperazioniEffettuate numeroOperazioniFallite

CONNECT esitoOperazione

STORE numeroRichiesteEffettuate numeroRichiesteFallite sommaByteInviatiConSuccesso

RETRIEVE numeroRichiesteEffettuate numeroRichiesteFallite sommaByteRicevutiConSuccesso

DELETE numeroRichiesteEffettuate numeroRichiesteFallite

LEAVE esitoOperazione

Ad eccezione di REPORT e CONNECT le altre voci sono opzionali e dipendono dall'esito della CONNECT e dall'opzione di test richiesta.

4.2 `testsum.sh`

Lo script elaborerà il file `testout.log` e stamperà a video una serie di informazioni sul test tra le quali: client lanciati, quante operazioni in totale e per tipo sono state effettuate, quante operazioni in totale e per tipo sono fallite e quante hanno avuto successo, somma byte ricevuti e inviati con successo, % di fallimento per client, % di fallimento per batteria di test...

Appendice A `Makefile`

Oltre ai phony richiesti nelle specifiche ne sono stati aggiunti altri per rendere più agevole il testing:

- Server: lancia un server ripulendone prima, eventualmente, l'ambiente di esecuzione dalla cartella `data` e dal `objstore.sock`.
- Client: lancia un client di test con il parametro 0.
- Client1: lancia un client di test con il parametro 1.
- Client2: lancia un client di test con il parametro 2.
- Client3: lancia un client di test con il parametro 3.
- valServer: lancia, con valgrind, un server ripulendone prima, eventualmente, l'ambiente di esecuzione dalla cartella `data` e dal `objstore.sock`.
- valClient: lancia, con valgrind, un client di test con il parametro 0.