Sistemi Operativi e Laboratorio

Gioacchino Mirko Matonti

Settembre 2019

1 Scelte progettuali

Di seguito sono mostrate le scelte effettute durante lo sviluppo di questo progetto.

1.1 Struttura dati di appoggio

Come struttura dati di appoggio per memorizzare le informazioni dei client attivi ho scelto di utilizzare una struct worker:

```
typedef struct worker {
    struct worker *next;
    struct worker *previus;
    int fd_worker;
    int signed_up;
    int online;
    char name[MAX_NAME_LENGHT + 1];
    pthread_t thread_id;
}worker;
```

I worker sono collegati tra di loro attraverso la lista $*workerlist$ gestita in mutua esclusione. Non possono coesistere client omonimi ed inoltre l'username del client può avere dimensione massima 100b. Il membro *int online* segnala che il thread è in attesa di nuovi messaggi.

L'objectStore memorizza alcune informazioni in un analoga struct:

```
typedef struct server_info {
    int connected_clients;
    long double store_size;
    int n_object;
    struct sockaddr_un sa;
    struct pollfd poll_fd;
    int is_running;
    pthread_t signal_thread;
}server_info;
```

I membri *connected_clients* e *store_size* sono contatori rispettivamente dei client connessi e della dimensione dello store. Di notevole importanza è la variabile *int is_running* che segnala ai thread worker lo stato di funzionamente dell'ObjectStore.

1.2 Header

Il protocollo di scambio messaggi prevede l'utilizzo di un header: $OPERAZIONE\ NAME\ LEN\ DATI$. Nel corso di questo progetto ho assunto come accennato poc'anzi che il nickname assegnato ad ogni client e nel caso della Store/Retrieve il nome del file abbiano come dimensione massima 100b.

2 Struttura del codice

Il progetto è composto da un server di seguito nominato *ObjectStore*, una libreria *libobjectstore* per il suo interfacciamento ed un client che attraverso quest'ultima mostra le principali operazioni eseguibili.

- **server.c** – implementazione dell' ObjectStore, si occupa di accettare le connessioni dei client e di lanciare il relativo thread worker.
- **worker.c** – thread worker eseguito per ogni client connesso, si occupa di gestire tutte le richieste di quest'ultimo.
- **handler.c** – si occupare del parsing dei messaggi ricevuti dai client, lavora in simbiosi con il worker per espletare le varie richieste ricevute.
- **libobjectstore.c** – implementazione della libreria di accesso.
- **utils.c** – modulo di supporto contentenente vari metodi condivisi, inoltre il relativo header utils.h contiene le principali macro che regolano il funzionamente interno dell ObjectStore.
- **signal.c** – thread che si occupa di gestire i segnali indirizzati all'ObjectStore.
- **client.c** – client demo realizzato secondo le specifiche fornite, si occupa di testare le varie funzionalità messa a disposizione dall' ObjectStore.

3 Gestione segnali

La gestione dei segnali è affidata ad un thread *signal_thread* il quale gestisce SIGINT, SIGTERM e SIGUSR1.

SIGINT e SIGTERM avviano la procedura di chiusura del server settando la variabile condivisa $server- > is\_running$ a 0, notificando a tutti i thread attualmente attivi di avviare anch'essi le procedure di terminazione.

SIGUSR1 come indicato nel testo del progetto, richiama la funzione $print\_server\_info()$ che stampa sullo stdout dell' ObjectStore le seguenti informazioni:

- Numero di client connessi
- Size dello store Kb/Mb
- Numero di oggetti presenti nello store

4 Libreria utente

La libreria mette a disposizione i seguenti metodi:

- **int os_connect(char *name)** – inizia la connessione all'object store, registrando il cliente con il name dato. Restituisce true se la connessione ha avuto successo, false altrimenti.
- **int os_store(char *name, void *block, size_t len)** – richiede all'object store la memorizzazione dell'oggetto puntato da block, per una lunghezza len, con il nome name. Restituisce true se la memorizzazione ha avuto successo, false altrimenti.
- **void *os_retrieve(char *name)** – recupera dall'object store l'oggetto precedentemente memorizzatato sotto il nome name. Se il recupero ha avuto successo, restituisce un puntatore a un blocco di memoria, allocato dalla funzione, contenente i dati precedentemente memorizzati. In caso di errore, restituisce NULL.
- **int os_delete(char *name)** – cancella l'oggetto di nome name precedentemente memorizzato. Restituisce true se la cancellazione ha avuto successo, false altrimenti.
- **int os_disconnect()** – chiude la connessione all'object store. Restituisce true se la disconnessione ha avuto successo, false in caso contrario.

5 Test e note d'uso

Il progetto è stato testato sui seguenti sistemi operativi:

- Ubuntu 18.04
- Xubuntu 14.10 (VM fornita dal docente)
- Debian 9.9

5.1 Esecuzione

Eseguire il comando *make* per generare gli eseguibili. Lanciare l'*ObjectStore* con *./server.o* mentre in un altra shell eseguire *make test* per lanciare la batteria di test.

5.2 Analisi dei test

Il progetto è corredato di uno script *testsum.sh* il quale fornisce un resoconto delle operazioni eseguite dal comando *maketest*.

5.3 Extra: client demo interattivo

Eseguendo *make testclient* è possibile lanciare il client demo interattivo utilizzato nella fase di sviluppo. (richiede che un instanza di *./server.o* sia attiva).

Disclaimer : essendo un extra utilizzato solo a scopi di sviluppo interni, testclient.o potrebbe contenere alcuni bug.