Progetto del Corso di Sistemi Operativi e Laboratorio 2018/19
Object Store

Federico Matteoni
Mat. 530257

Introduzione

Il progetto richiedeva la realizzazione di un sistema **client** - **server** implementante un *object store* con l'obiettivo di supportare le richieste di memorizzazione di un gran numero di applicazioni.
Le connessioni avvengono su **socket in dominio locale** e il server deve poter **supportare un numero elevato di connessioni in ogni dato momento**.
Si richiedeva l'implementazione, separata, di un **server** come eseguibile autonomo e di una **libreria** sfruttabile dai client che implementa il protocollo specificato. Inoltre, si richiedeva la realizzazione di un **client di esempio** usato per eseguire i vari test richiesti.

Il Server

Il server è stato implementato seguendo la filosofia di un thread-dispatcher che, una volta accettata una connessione, istanzia un thread e la assegna ad esso.
Ogni thread quindi è dedicato ad un singolo client e alla relativa connessione.
Le comunicazioni avvengono su socket di tipo AF_UNIX locale, di nome *objectstore.sock*. Per consistenza, il nome è stato inserito come macro all'interno sia del server che della libreria client, insieme alla dimensione del buffer di lettura/scrittura attraverso la socket.
Il server mantiene un **numero massimo di thread** attivi contemporaneamente, e di conseguenza un numero **massimo di client connessi in un dato momento**. Questa scelta è nata dal desiderio di voler realizzare un server che non fosse utilizzabile per rallentare il sistema su cui viene eseguito, ad esempio con fini malevoli generando un numero infinito di client connessi. Tale numero è mantenuto in una macro definita nel codice del server.

Una volta accettata una connessione, il server prepara ed esegue un thread ad essa dedicato. Il thread si occuperà, immediatamente dopo l'inizializzazione, di controllare il messaggio sul socket. In caso sia il messaggio di registrazione, estrapolerà il nome utente e controllerà che esso non sia già connesso, per garantire l'univocità di utenti connessi in un dato momento. Questo controllo è realizzato con un array di clienti connessi che viene mantenuto aggiornato dai thread stessi in mutua esclusione. Quando un nuovo utente si registra, il thread si preoccuperà di controllare che esso non sia già connesso esaminando il thread e, in caso sia un cliente non connesso, lo inserirà nella prima posizione libera del thread. Se il client è già registrato, o se il numero massimo di client connessi contemporaneamente è stato raggiunto, il thread notificherà l'errore e chiuderà la connessione. Se invece il client è autorizzato a procedere, il thread notificherà l'avvenuta connessione e si preparerà a soddisfare le richieste che arriveranno dal client a cui è assegnato.

Store
Quando viene richiesta una STORE, il thread servente controlla che non esista già un file con il nome richiesto all'interno dello spazio dell'utente che sta servendo.
In caso esiste già un file, il thread risponde con un errore, notificando che il file è già esistente. Questo per evitare sovrascritture accidentali (in caso si voglia modificare un file si può prima eseguire la DELETE dello stesso) e mantenere univoco il nome all'interno dello spazio.
In caso il file non esista, salvo errori di comunicazione o allocazione il file viene memorizzato leggendo il contenuto della lunghezza indicata dal socket. Il file viene scritto mettendo prima la lunghezza dello stesso e subito dopo il suo contenuto. Questo per rendere più immediata la lettura in fase di RETRIEVE. Se la STORE va a buon fine, il thread aggiorna i dati dell'object store incrementando il numero di oggetti memorizzati e la dimensione totale dello store.

Retrieve
Su richiesta di una RETRIEVE, il thread andrà subito a controllare che il file richiesto esista nello spazio dell'utente e che sia possibile leggerlo. In caso positivo, lo leggerà memorizzando la lunghezza del contenuto e il contenuto stesso. Dopodiché, costruirà il messaggio di risposta e lo spedirà attraverso il socket.
In caso di errore, se ad esempio il file non esiste oppure si sono verificati errori nell'allocazione, esso viene segnalato.

Delete
Se viene ricevuta una REMOVE, analogamente alla gestione della RETRIEVE il thread controllerà che il file esista e che sia rimovibile. Se la situazione lo permette, il thread elimina il file e risponde positivamente al client, altrimenti segnalerà il problema avvenuto.
In caso di successo della DELETE, il thread aggiornerà le statistiche dell'object store, decrementando appositamente il numero di oggetti memorizzati e la dimensione totale dello store.

Leave
In caso di LEAVE, il thread si preoccuperà, dopo aver risposto, di avviare la propria chiusura, segnalare la disconnessione del client e la chiusura del thread per fini statistici (vedi SIGUSR1) e deallocare le risorse utilizzate.

SIGUSR1 e segnali
Il server gestisce autonomamente i seguenti segnali:

- SIGUSR1, ricevuto il quale stampa a video le seguenti statistiche relative al dato istante prima di interrompere il server:
    - Il numero di thread attivi
    - Il numero di client connessi
    - Quanti oggetti sono memorizzati nell'object store

    - La dimensione totale dell'object store

  Questo è stato realizzato attivando una flag alla ricezione del segnale. Il thread dispatcher controllerà tale flag e, se è stata settata, la pulirà per poi stampare a video le informazioni di cui sopra. Dopodiché avvierà la procedura di chiusura del programma.
- SIGPIPE, ignorandolo
- SIGINT e SIGTERM, con i quali termina in sicurezza, deallocando le risorse e chiudendo i socket

La Libreria Client

La libreria implementa le funzioni richieste, generando le richieste nel formato specificato dal protocollo indicato.
Le varie richieste sono **costruite dinamicamente concatenando i vari pezzi dell'header** e spedendo il messaggio così creato sul socket. Ogni richiesta è seguita da una risposta dal server che viene opportunamente gestita: in caso di risposta positiva (ad esempio del tipo "OK \n") la libreria segnala l'avvenuta operazione, in caso contrario la libreria stampa a schermo il messaggio d'errore contenuto nella risposta, prima di segnalare che l'operazione ha riscontrato errori.
Questo messaggio di errore ha sempre il prefisso "Errore", utile nello script di analisi dei risultati per il conteggio e la visualizzazione degli errori avvenuti.

La Comunicazione

Le comunicazioni sul socket avvengono tramite messaggi di lunghezza fissa di 100B. L'unico caso in cui questo non avviene è durante le operazioni di STORE e RETRIEVE, dove la seconda parte del messaggio, contenente i dati binari, è scritta e letta dal socket a seconda della lunghezza specificata nell'header. Si sono rese quindi necessarie due write/read consecutive, la prima con l'header di 100B e la seconda con i dati di lunghezza variabile contenuta nell'header.

Il Client

Il client di esempio richiesto genera i contenuti da memorizzare e verificare a partire da una stringa di 100 caratteri casuali.
I 20 file da memorizzare sono generati a partire da tale stringa, concatenandola a sé stessa per poter generare stringhe con lunghezze di volta in volta incrementate di 5258 caratteri, così che in 20 iterazioni si arrivi a circa 100'000 caratteri.

I tre tipi di test che il client può eseguire sono divisi in tre metodi separati, alla fine dei quali, dopo la disconnessione, si prepara un report da mandare in output contenente il tipo di test eseguito, il numero di test eseguiti e il numero di successi e fallimenti.
Questo output formattato è poi letto dallo script di analisi per poter estrapolare varie

statistiche, come il numero di client lanciati e la percentuale di successi e fallimenti sia totali sia di ogni tipologia di test.

Test

Insieme al test richiesto, durante la stesura del progetto ho ritenuto comodi altri test da eseguire per garantire la qualità del codice:

- Per poter verificare l'effettiva univocità di client connessi ho scritto un semplice script che esegue in contemporanea 10 client con lo stesso nome, così da verificare in output l'effettiva presenza di 9 messaggi d'errore riguardanti il nome del client duplicato
- Analogamente, per verificare l'univocità degli oggetti memorizzati dal singolo client, uno script esegue prima un client con un determinato nome che esegue un test di tipo 1 (20 STORE), appena ha concluso riesegue un client col medesimo nome e test, che non potrà memorizzare nessuno dei 20 oggetti poiché tutti già esistenti
- Un ultimo test progettato è molto simile al test richiesto, con la differenza di lanciare un totale di 1000 client in contemporanea che eseguono test di tipo 1, dopodiché in contemporanea altri 500 client con test di tipo 2 e 500 client con test di tipo 3. Questo test ha il duplice scopo di testare il limite di thread in esecuzione in un dato momento e di esercitare una sorta di stress test sull'Object Store

Tutti i test progettati danno risultati positivi senza anomalie sui seguenti sistemi dov'è stato testato il progetto:

- Ubuntu 18.04.3
- Xubuntu 14.10