Università di Pisa

PROGETTO “File Storage Server”

Simone Tassotti – 583282

Introduzione

La consegna del progetto richiede la realizzazione di un server per la memorizzazione di file, la cui particolarità stava nel dover memorizzare/operare su di essi **esclusivamente** in memoria principale (rappresentati tramite il loro **path assoluto)**, senza mai andare a utilizzare il disco; il server deve simulare il comportamento tipico di una memoria cache; la dimensione del server non muta durante l’esecuzione. Il server, all’avvio, legge i vari settaggi di funzionamento da un file di config dal quale ricava: quanti thread avviare; il numero massimo di file nel server; la capacità massima in MB; il numero di utenti massimo che può aprire un file; il canale socket che specifica dove avviene la connessione ecc…
Il server lavora in modalità “Single Process – Multi Threaded”, con la realizzazione di API specifiche per la connessione dei client; inoltre, il server doveva “comportarsi” come una memoria cache, andando quindi a gestire il caso di “Capacity Misses”.

Implementazione Server

Il server è stato realizzato in più parti: libreria per la gestione di un file, del pool di Thread, di un logFile, della memoria cache e della comunicazione Client – Server.

La prima parte di sviluppo è stata focalizzata sulla libreria “File”: rappresenta la struttura del file: pathname, dimensione, buffer dei dati, lista degli utenti connessi, massimo numero di utenti che possono aprire il file, utenti che richiedono la lock sul file e il tempo di ultimo accesso. La libreria non gestisce i casi di accesso concorrente al file. Il compito aspetterà alla libreria di gestione della cache “FileStorageServer” (tra poco descritta); le funzioni permettono la creazione della struttura, la scrittura in “append”, apertura/chiusura/lock/unlock da parte di un utente del file, aggiornamento del tempo di ultimo accesso e deallocazione della struttura stessa. Inoltre, è stata implementata una seconda libreria per la gestione degli utenti che sono in attesa di lock tramite la libreria “Queue”, che gestisce una semplice coda.

Successivamente, è iniziata la realizzazione della libreria di gestione della memoria cache “FileStorageServer”. La struttura della cache è formata da due tabelle hash (contenute in “icl_hash”, fornitaci a lezione durante le esercitazioni): una dove vengono memorizzati i file aperti ma **non ancora scritti** nella memoria (salvo solo il pathname e l’ID del client che lo ha aperto), l’altra contiene i file **effettivamente scritti** nella memoria e salva tutta la struttura del file. In aggiunta a queste due strutture ci sono una terza e quarta struttura dati che gestiscono la politica di rimpiazzo LRU dei file memorizzati e la mappa dei client connessi e dei file che sono stati aperti durante il collegamento. Le funzioni di gestione della memoria cache garantiscono l’apertura/chiusura di un file, la creazione, aggiunta nel server, la scrittura in “append”, la rimozione e la lock/unlock da parte di uno o più client e la lettura del contenuto di ogni singolo file in mutua esclusione attraverso la gestione degli accessi alle strutture di: “Pre-Inserimento”; “Memoria cache”; “accesso al singolo file” che viene regolamentata a seconda del posizionamento in tabella hash. All’inizio di ogni operazione viene aggiornata di volta in volta la “LRU” in modo tale che, in caso di scrittura/aggiunta del file in memoria si venga a creare una situazione di “Capacity Misses” e si vadano a espellere quei file che sono stati utilizzati meno recentemente. La politica di rimpiazzamento dei file non fa distinzione tra file aperti/locked, ovvero, se un file è aperto ma comunque non viene utilizzato da tempo viene cancellato. Se il client effettua un’operazione

su di esso, riceverà un errore di file mancante o espulso; stessa cosa vale per gli utenti locked, i quali vengono segnalati per essere sbloccati. Altra funzione è quella della lettura del file config per il caricamento iniziale del server: la funzione va leggere il file di config e, andando a scartare le porzioni in cui sono presenti commenti, va ad analizzare i parametri passati, tra cui: numero massimo di file, capacità in MB del server, numero di Thread che devono gestire le richieste, numero di utenti che possono aprire un singolo file e il numero massimo di utenti connessi al server contemporaneamente.

Dopo aver effettuato il parsing del file di configurazione, il server crea sia la memoria cache da utilizzare e avvia il pool di thread gestito dalla libreria "threadPool" che regola il flusso di task che ogni thread deve ricevere e l'eventuale arresto (forzato o meno) del pool. Il thread manager spedisce i compiti al pool di thread worker che poi, attraverso un task specifico denominato "Server_API", andranno a gestire le singole richieste inviate dai client.

Ogni operazione viene salvata e controllata su un file di log, gestito dalla libreria "logFile", che si occupa della sua formattazione e dell'accesso concorrente di più thread che vogliono scriverci.

Il programma generale di gestione del server si occupa di ascoltare le richieste, mandare i task al pool di thread, riabilitare gli utenti che vogliono continuare la conversazione e gestire l'arrivo di un segnale di shutdown, attraverso la gestione dei tre segnali **SIGINT, SIGQUIT, SIGHUP**. L'handler dei segnali è gestito da un thread supplementare che, in caso di richiesta di shutdown, blocca completamente il server e manda un segnale al thread pool di arresto immediato (in caso di ricezione di **SIGINT o SIGQUIT**), altrimenti aspetta che tutti i client ancora connessi abbiamo terminato per poi far chiudere il pool (nel caso di ricezione di **SIGHUP**). Infine, al pool viene comunicato un task speciale che consegue lo spegnimento di ogni thread e, in caso di hard shutdown, cancellando ogni task pendente e lasciando solo quello speciale da eseguire.

Gestione Client

Il client, che riceve le informazioni da linea di comando, effettua un ordinamento delle opzioni in modo tale che, per es., un'operazione di lock posta prima dell'operazione di connessione al server non fallisca. Le operazioni che devono essere gestite per prime sono: -h, -p, -t, -f, (analizzate nel seguente ordine). Le richieste al server vengono gestite direttamente dalle API realizzate secondo la specifica presentata. Nella libreria contenente le API, sono state aggiunte altre funzioni che operano a livello locale del client che vanno a leggere i file da una cartella e, in modo speculare, li vanno a scrivere in una directory specificata. Nelle funzioni di "openConnection" e "readNFiles" sono presenti delle particolarità: nel primo caso, la gestione del timeout di connessione al server viene effettuato da un thread di supporto che notifica lo scadere del tempo a disposizione per tentare la connessione; nel secondo caso le readNFiles legge unicamente quei file che non sono locked da altri client in quell'istante.

Funzioni condivise

La libreria "utils" implementa, sia lato server che lato client, funzioni per lo scambio reciproco di dati garantendo che tutti i byte spediti/letti vengano recapitati grazie alle funzioni "readn" e "writen" forniteci a lezione per la risoluzione del problema delle write/read parziali su socket.

LINK GITHUB

N.B.: Per eseguire i file lanciare "chmod 777"