Chatty Server

Autore: Mattia Angelini

Matricola: 502688

Premessa sui commenti:

Per abitudine non riempio il codice di commenti, poiché credo sia di gran lunga una soluzione migliore, utilizzare il codice stesso per spiegare cosa il codice faccia attraverso l'uso di nomi opportuni.

Ciò ha 2 vantaggi : * Basta usare un qualsiasi autocompletamento per sapere cosa la funzione faccia, invece che allestire un doxygen. * Non esiste il problema di avere commenti che si riferiscono a versioni obsolete del codice.

Detto ciò, i commenti sono presenti ma utilizzati solo dove necessario.

Convenzioni:

Malloc e Free:

Unica nota importante qui è che, le **Variabili Globali** vengono liberate solamente all'uscita dell'applicazione, più in dettaglio: il thread principale si assicura che tutti i sotto-thread siano terminati, poi effettua la free delle suddette variabili, in quanto, una loro rimozione prematura ,avrebbe causato delle situazioni imprevedibili.

Nomi:

- **Funzioni:** le funzioni che effettuano lock e relative unlock sui dati utilizzati finiscono sempre con il suffisso: **"_S"**, e le funzioni che allocano le strutture sono state provviste di voce **new**, in ultimo mi sono assicurato (se non in casi eccezionali), che tutte le funzioni abbiano come prefisso il nome del modulo in cui sono state definite.
- **Variabili:** le variabili globali definite nel file *ServerGlobalData.h* hanno tutte il prefisso: **"GD_"**.

Descrizione dei moduli

Inizio con il dire che in questa sezione, verranno spiegati solo i moduli con le funzionalità meno intuitive, o che implementano funzionalità non banali.

Channel:

Il channel è la parte del progetto che svolge il compito di mettere in comunicazione i thread tra di loro.

Il suo funzionamento è basato su un array nel quale vengono messi i messaggi, che i thread possono quindi inserire e/o leggere, più nello specifico si parla di operazioni di POP e di PUSH

(dunque gli elemeni vengono rimossi dopo una POP). Il channel assicura che le succitate operazioni di POP e/o PUSH siano sincronizzate e, che quindi, non ci siano race condition. In secondo luogo, il channel mette in attesa passiva i thread, in caso non sia possibile soddisfare la loro richiesta.

Il numero di thread in *POP* o *PUSH* non è limitato in nessun modo, ma per lo scopo del progetto esiste un unico thread che crea messaggi (un nuovo fd è disponibile per la lettura) e una serie di thread che eseguono operazioni di POP sul canale per gestire il nuovo fd.

Strutture generiche (HashTable & List):

Nel progetto sono state implementate 2 tipologie di strutture generiche, come da titolo Hash e Liste. Per strutture generiche, mi riferisco ad un sistema, che permette di avere un implementazione unica della gestione delle strutture, poiché permette di contenere qualsiasi tipo di dato, sfruttando puntatori generici (void*).

Ma, sebbene questa soluzione era molto buona per Applicativi *single-thread*, si è rilevato di difficile implementazione nel caso di sistemi *multi-thread*.

Nel progetto dunque ho dovuto scegliere come gestire i problemi sollevati da questa implementazione, sia per le liste che per le hash.

- Se provavo a rendere il sistema sicuro allora avrei dovuto effettuare la Deep-copy(copiare tutti i valori interni del valore, cosa che nel caso di liste, array o simili erà lenta, e dunque non fattibile), in secondo luogo con questa soluzione avrei introdotto problemi di sincronizzazione dei dati.
- Se invece provavo a effettuare copie per puntatore, avrei velocizzato le operazioni ma d'altro lato avrei introdotto tanti problemi di race condition.

Alla fine dunque ho scelto la soluzione intermedia, ovvero effettuare shallow-copy (copio solo la struttura puntata dai nodi della struttura e non eventuali valori puntati), assicurandomi, laddove neccessario, che le dovute lock e unlock venivano effettuate. Per velocizzare alcune parti ho invece usato i puntatori hai dati contenuti, nelle HashTable o Liste, effettuando ovviamente le dovute operazioni di sincronizzazione.

Altro problema relativo ad entrabe le strutture, era che erano generiche, ciò non mi permetteva senza informazioni extra, di capire come effettuare operazioni di Free, copia o comparazione tra i valori contenuti nelle strutture. Per ovviare al problema ho utilizzato puntatori a funzione, in modo da specificare come eseguire tali operazioni, in base alla tipologia di elementi che mettevo nella lista.

Parliamo ora più nello specifico come ho implementato le liste e HashTable.

Liste:

Le liste sono state implementate con nodi **Bidirezionali**, in quanto, per il costo di un puntatore extra, ottengo la abilità di rimuovere con semplicità l'elemento in coda. Ma di fatto la lista vera e propria, è contenuta dentro una struttura List, che tiene traccia dei nodi in prima ed ultima posizione della lista più qualche informazione utile : come il numero di elementi presenti.

HashTable:

Come per le liste ho usato una struttura che contiene poi la vera *HashTable*. Tale struttura oltre alla *HashTable* vera e propria ha anche delle informazioni relative alla HashTable come ad esempio la sua dimensione.

La hashTable, gestisce le collisioni con liste di trabocco ottenute dalle liste generiche sopra citate, che contengono gli elementi della hash(HashElement) in cui vengono salvati i puntatori ai dati assieme alla loro relativa chiave di salvataggio.

Syncronized HashTable & List:

Sono di fatto *"Wrapper"* per i relativi moduli, che assicurano che non vi siano race condition attraverso delle **mutex**.

Va però fatto notare che mentre per le liste la mutex è unica, per le HashTable è presente un array di mutex partizionate, sul quale vengono effettuate le operazioni di lock in base al indice della chiave.

Syncronized Socket:

Questo modulo va inizializzato prima di poter essere utilizzato, ma una volta creato permette di effettuare operazioni sugli FD, senza preoccuparsi di race condition in scrittura in quanto utilizza un array di mutex per evitare il problema.

SettingManager:

Si occupa del caricamento delle impostazioni del server, caricando le informazioni contenute in un file, nel formato specificato caricandole in una apposita struttura.

Nello specifico il modulo e stato reso il più possibile **Fault tollerable**, facendogli ignorare problemi di case (nel nome dei settaggi), spazi (tra i settaggi e i loro valori) ed eventuali caratteri speciali.

Soluzioni implementative:

Sincronia:

Come detto sopra viene fatto uso di **"Syncronized Socket"** per evitare race condition nella scrittura delle socket, Mentre in lettura, la sincronia e' assicurata dal fatto che, un Fd viene preso in carico da solamente da un worker alla volta.

Faccio invece uso delle **Syncronized Hash** (alla fine non ho usato le **Syncronized List**) per le race condition sulle hash, sempre introdotte sopra.

Per far comunicare il main thread con i suoi sotto-thread, ho usato il **Channel**, che automaticamente mette in attesa i thread dei quali non può, soddisfare le richieste.

Infine mi sono provvisto di necessarie mutex, la dove necessario per le variabili globali, come ad

esempio le statistiche.

Gestione richieste:

I **Worker** (sotto-thread), hanno il compito di leggere le richieste effettuate dal fd preso in carico. La richiesta viene letta e fatta passare per un router che decide che operazione il client voglia effettuare, se la richiesta viene eseguita correttamente, la funzione che gestisce la richiesta si assicurerà di inviare tutti i messaggi ed Ack necessari, ma non esegue riposte con messaggi di errore. Se l'operazione va a buon fine, il worker si assicura di notificare il main-thread che ha rimesso lo Fd preso in carico nel set globale degli Fd, attraverso un segnale di tipo **SIGUSR2**, che sveglierà la **pselect** del main-thread facendogli aggiornare il suo set degli Fd. Ma nel caso di errori sarà annullata la richiesta e deciso dal worker che fare con lo fd : * Rispondere al client con il relativo messaggio d'errore, e rimettere lo Fd nel Set. * Distruggere la socket (errori gravi quali Broken Pipe), in questo caso lo Fd non sarà reinserito nel set.

Main-Thread (Stampa settaggi e ascolto nuove connessioni):

Il main-thread ha il compito di caricare i settaggi oltre a quello di iniziallizzare tutte le variabili globali, e le necessarie strutture per la sincronizzazione. Esegue anche la creazione (ovviamente) dei suoi sotto-thread con il compito di eseguire le richieste, che si metto in attesa passiva sul Channel in attesa di nuovi Fd da gestire. A questo punto, viene creata la socket dove il thread si mette in ascolto di nuove connessioni, che al momento della ricezione provvederà a inviare nel canale ai sotto-thread lo fd relativo. Questo fino al ricevimento di un segnale di stampa statistiche (SIGUSR1), che blocca l'arrivo di nuove connessioni per il tempo di stampare le statistiche, o di un segnale di terminazione che fa di fatto terminare l'applicativo dopo aver atteso la chiusura dei sotto-thread ed eseguito le Free necessarie.

Strutture dati:

Le strutture dati sono per lo più HashTable, usate sia per gli utenti sia per i gruppi. Mentre le liste sono state usate per la *History* dei messaggi utente (anche se in secondo luogo sarebbero sati più performanti degli array) e come supporto delle Hashtable.

Altro:

- La struttura **User** contiene tutte le informazioni riguardo ad un utente (nome,stato,History,etc..), dunque ,per recuperare la lista degli utenti online, vado ad eplorare la hash delgi utenti(non una grande soluzione ma per lo "scope" del progetto va più che bene).
- Il codice interno a **Log()** oppure **ON_DEBUG()**, viene compilato solo se la flag -DDEBUg è specificata.
- I segnali "Gestibili" sono gestiti dal main-thread, che analizza un flag settato sulla ricevimento del segnale, per capire cosa fare e notificando dove necessario i sotto-thread.

Assunzioni sulle specifiche:

1. I gruppi devono iniziare con la parolachiave **"gruppo"**.

Possibili miglioramenti:

- La funzione di **"hashing"** non è ben distribuita, una sostituzione con una più equiprobabile potrebbe migliorare le performance.
- Ottimizzare la dimensione delle hash.
- Ottimizzare la dimiensione delle Mutex partizionate, in modo da evitare per quanto possibile le collisioni, cosa che in questo caso blocca un thread.
- Alcune strutture possono essere sopostate sullo Stack, rendendo più veloce la loro rimozione*
- Implementare la soft-close(il programma a meno di SIGTERM si assicura di scodare tutto dal Channel).
- Ottimizzare il ciclo di ascolto del main-thread, in modo da evitare problemi di signal race.
- Ottimizzare gli import.

Problemi conosciuti:

- **Con alcune versioni di valgrind è possibile che sia rilevato un errore nella pselect per la maschera vuota.**
- **Non compila con nuove versioni di gcc, per un cambio d'uso di alcuni flag.**

Sistema e strumenti :

- **OS:** Ubuntu 16.04 (Linux 3547af374672 4.9.60-linuxkit-aufs #1 SMP Mon Nov 6 16:00:12 UTC 2017 x86_64 x86_64 x86_64 GNU/Linux)
- **GCC:** gcc version 5.4.0 20160609 (Ubuntu 5.4.0-6ubuntu1~16.04.9)
- **VALGRIND:** valgrind-3.11.0