

# RETI Lab - Progetto

Francesco Lorenzoni - 599427

January 27, 2022

## Contents

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Server</b>	<b>1</b>
2.1	Server.java . . . . .	2
2.2	ServerInternal.java . . . . .	3
<b>3</b>	<b>Client</b>	<b>6</b>
<b>4</b>	<b>Test</b>	<b>7</b>

## 1 Introduzione

**JDK** Il progetto è stato realizzato utilizzando il Java SE Development Kit 14.0.2

**Librerie di terze parti** Sono state usate le tre librerie per serializzazione e deserializzazione di `jackson` e la libreria `lombok`.

**GitHub repo** [github.com/frenzis01/RETI-progetto](https://github.com/frenzis01/RETI-progetto)

**Struttura Directory** Nella directory principale sono presenti la specifica, la relazione e la cartella *prj* che contiene i file del progetto. Di seguito una breve descrizione del contenuto delle subdir di *prj*:

- *bkp* file .json di backup del server
- *config* file .json di configurazione del server e manifest utilizzati nella creazione dei .jar
- *jar* .jar eseguibili per Server e Client; è possibile passare a entrambi il percorso di un eventuale file di configurazione, altrimenti lo cercheranno nella posizione di default o, se non ci fosse, useranno dei valori di default. Per testare il funzionamento quindi è sufficiente eseguire `java -jar (Server|Client).jar <configPath>`
- *lib* librerie jackson e lombok
- *out* file .class generati dalla compilazione dei file sorgente
- *src* file sorgente .java
- *tests* file .sh utilizzati in fase di testing

## 2 Server

**Scelte** Alcuni brevi cenni sulle scelte e le assunzioni fatte:

- Il server esegue il multiplexing dei canali con un `Selector` `NIO` e utilizza un pool di thread per l'elaborazione delle richieste.
- In un secondo momento è stata aggiunta la seguente limitazione: un utente può essere risultare loggato da un solo terminale alla volta. Tuttavia, nonostante sia attiva di default, è un parametro configurabile.
- È presente un file .json di configurazione sia per il Server che per il Client. In caso tali file non siano presenti, o non siano specificati tutti i campi, verranno assegnati dei valori di default.

- Il server comunica al client l'indirizzo e la porta per ricevere le notifiche di reward in risposta a un'operazione di login con successo.

**File sorgente del server** Sono presenti tre file sorgente relativi al Server:

- **ServerMain** Lancia il processo server.
- **ServerInternal** Mantiene lo stato interno del server. Espone tutte le funzionalità necessarie per implementare le richieste effettuate tramite TCP.
- **Server** Si occupa di eseguire il binding in un **Registry** di uno stub che espone le funzioni del server implementate tramite RMI, quali la registrazione di nuovi utenti e l'iscrizione/disiscrizione al servizio di aggiornamento dei follower.

## 2.1 Server.java

### Struttura Thread

- **tcpThread** Gestisce le connessioni TCP con i client tramite un **Selector NIO**
- **loggerThread** Esegue periodicamente ogni  $x$  secondi il backup dello stato interno del server su dei file **.json**
- **rewardThread** Esegue periodicamente ogni  $y$  secondi il calcolo delle ricompense e invia la notifica di avvenuto calcolo in un datagramma UDP tramite un indirizzo multicast
- **btcRateThread** Dato che il calcolo del tasso di cambio fra wincoin e bitcoin tramite [random.org](http://random.org) risulta particolarmente oneroso, per non rallentare eccessivamente la risposta a una richiesta **wallet btc** è stato aggiunto questo thread dedicato che si occupa di ricalcolare tale tasso allo scadere di un timeout.
- **workerPool** fixed thread pool di worker che elaborano le stringhe di risposta da inviare ai client. Ai worker viene assegnato un **Runnable** restituito da **requestHandler()**.

### Strutture Dati

- **ConcurrentHashMap<String, String> loggedUsers** Associa a ogni indirizzo remoto dei client collegati tramite TCP l'username dell'utente con cui hanno effettuato il login. Più thread possono elaborare richieste di login e logout *concorrentemente*.
- **ConcurrentLinkedQueue<RegisterParams> toRegister** Quando un worker ha elaborato la risposta a una richiesta inserisce in questa coda la risposta stessa il **SocketChannel** del client a cui bisogna scrivere.
- **Pipe pipe** Per segnalare al **Selector sel** la presenza di una risposta pronta nella coda sopra descritta, i worker scrivono un byte (*token*) nella **Pipe**. C'è dunque una corrispondenza 1:1 fra le risposte nella coda e i *token* presenti nella pipe, i quali verranno consumati da **tcpThread**.
- **boolean** quit flag globale settato dallo **shutdownHook** quando viene ricevuto **SIGINT** o **SIGTERM**.

### Multiplexing

Operazioni di **cancel()** e assegnamento di interest sulle **SelectionKey** vengono eseguite esclusivamente da **tcpThread**, così come tutte le operazioni di lettura e scrittura sui **SocketChannel** associati ai client. È bene sottolineare che essendo i canali non bloccanti è possibile che **read()** e **write()** vengano eseguite in modo parziale.

- **Scrittura** I thread worker **Server.requestHandler()** preparano un **ByteBuffer** contenente la lunghezza della risposta da inviare al client e la risposta stessa, il quale verrà inserito come attachment della key quando **tcpThread** eseguirà **register()**. **sendResult()**, chiamata quando **key.isWritable()**, utilizzerà l'attachment della key per scrivere sul **SocketChannel** associato e solo se l'operazione avrà "esaurito" il **ByteBuffer** registrerà l'interesse alla lettura da quel canale, altrimenti, quando il client sarà nuovamente disponibile per ricevere byte verrà eseguita nuovamente una **write** (**attachment**), sapendo che la **position** dell'attachment indica il primo byte che non è stato ancora inviato.

- **Lettura** Si può osservare che quando viene eseguita `register(selector, OP_READ, null)` è sempre specificato `null` come attachment. Questo torna utile perché permettere di distinguere un canale da cui bisogna iniziare a leggere da un canale su cui è stata eseguita una lettura parziale. In `getClientRequest()` se `key.attachment() == null` allora viene letto un `int` (si assume che venga letto tutto) rappresentante la lunghezza della richiesta effettiva, dopodiché viene allocato un `ByteBuffer` della dimensione appena letta e viene inserito come attachment. Vengono letti byte dal `SocketChannel` scrivendoli sull'attachment e solo se il `ByteBuffer` viene riempito, si elimina l'interesse a `OP_READ` e se la richiesta risulta essere un *logout*, si rimuove l'entry da `loggedUsers` e si ripristina l'interesse alla lettura, altrimenti viene affidata l'elaborazione al `workerPool`.

## RMI

L'interfaccia esposta dal server tramite RMI è `ROSint.java` ed è implementata da `ROSimp.java`. Il server permette la registrazione di nuovi utenti e l'invio di una notifica ogni volta che un altro utente esegue *follow* o *unfollow*. In `ROSimp` è presente `loggedUsers` che serve per tenere traccia dei client che si sono registrati al servizio di notifica dei follower. Quando un client esegue con successo *login*, viene aggiornata (ma non stampata su stdout) la lista locale di follower tramite il metodo `newFollowers()` esposto dall'interfaccia client `ROCint`.

Ogni volta che un utente *A* inizia o smette di seguire *B*, viene eseguito il metodo `ROSimp.update(B)`, il quale cerca tutte le interfacce su cui è loggato *B* (ne verrà trovata al più una) e su quelle che trova chiama il metodo `newFollowers()`, aggiornando la lista locale di followers del client, il quale, se ha abilitato la stampa delle notifiche, vedrà l'aggiornamento su stdout.

## Richieste e risposte

Le richieste devono rispettare il formato presente nella specifica del progetto per essere considerate valide; tale controllo viene eseguito sfruttando delle RegEx nella funzione `Server.parseRequest(s,k)`.

La medesima funzione si occupa anche di chiamare i metodi di `ServerInternal` per il calcolo dell'eventuale risposta, i quali lanceranno le eccezioni `NotExistingUser` e `NotExistingPost` in caso di riferimenti a post o utenti non presenti in *winsome*, mentre ritorneranno valori  $\neq 0$  per indicare errori di altro genere. La risposta sarà incapsulata in una stringa, che il client si limiterà a stampare.

## JSON Backup

All'avvio il server esegue il metodo `ServerInternal.restoreBackup()` che cerca di recuperare i file di backup dalla cartella specificata nel file di configurazione e, se li trova tutti, ripristina lo stato interno del server; se sono presenti solo alcuni file .json, ma non tutti, il server non esegue il ripristino, per evitare di ritrovarsi ad avere uno stato interno inconsistente.

Si assume che lo stato salvato nei file json sia consistente.

In base al timeout stabilito, verrà eseguito il backup periodico delle tre `ConcurrentHashMap` e dei due counter chiamando `ServerInternal.write2json()`.

## Terminazione

All'avvio viene settato uno `ShutdownHook` che coglie `SIGINT` o `SIGTERM` e setta il flag di chiusura `Server.quit` e sveglia il `Selector`. una volta terminato `tcpThread`, che si occupa di ricevere nuove richieste, vengono eseguiti un'ultima volta il calcolo delle ricompense e il backup dello stato interno, dopodiché il server termina.

## 2.2 ServerInternal.java

### Strutture Dati

Lo stato interno del server è mantenuto da alcune `ConcurrentHashMap`, in particolare:

- `users` gli utenti registrati su winsome
- `posts` i post pubblicati su winsome
- `tagsUsers` informazione "ridondante" che associa ad ogni tag esistente gli utenti registrati con esso.

Inoltre sono presenti due contatori che vengono sempre e solo incrementati, per identificare i post e per tenere traccia di quante volte è stato eseguito l'algoritmo di calcolo delle ricompense.

## Classi

- **User** Classe utilizzata da **ServerInternal** per rappresentare un utente di winsome.
- **Transaction** Classe "*Pair*" rappresentante un update del wallet di un utente.
- **Post** Classe utilizzata da **ServerInternal** per rappresentare un post. In essa è presente il campo **rewiners** per poter risalire rapidamente agli utenti che hanno eseguito il rewin di tale post.
- **ServerConfig** Classe contenente i campi configurabili del server. Sono presenti dei valori di default.
- **ServerInternal.UserWrap** / **PostWrap** Classi utilizzate per salvare lo stato di uno **User** o di un **Post** in un determinato momento. Contengono solo i campi utilizzati nelle risposte da inviare ai client.

## Reward

Per ottimizzare il calcolo delle ricompense e non dover controllare tutti i post, anche quelli su cui nessuno ha interagito dall'ultima esecuzione dell'algoritmo (e che dunque non portano alcuna nuova modifica ai wallet), ogni volta che un client interagisce con un post, quest'ultimo viene inserito in una delle tre **Map newUpvotes**, **newDownvotes** e **newComments**. L'algoritmo esamina un post alla volta prendendolo da queste tre **Map** e quando ha terminato la valutazione dello stesso, lo rimuove. Il metodo **rewardAlgorithm()** implementa l'algoritmo.

Per risalire all'*età* di un post, intesa come il numero di volte che le ricompense sono state calcolate dalla creazione del post, in ogni post è presente il campo **rewardIterationsOnCreation**, contenente il numero di run dell'algoritmo al momento della creazione del post. Essendo presente anche **ServerInternal.rewardPerformedIterations** che indica l'attuale numero di run dell'algoritmo, diventa immediato determinare l'età di un post.

## Concorrenza

La gestione della concorrenza è banale per le **ConcurrentHashMap** e per i campi

```
1 private static int idPostCounter;  
2 private static int rewardPerformedIterations;  
3 private static volatile double authorPercentage;  
4 private static volatile double btcRate;
```

Mentre per l'accesso alle istanze di **User** e **Post** è stata utilizzata una **ReentrantReadWriteLock** per ogni istanza.

Per scongiurare il rischio di eventuali deadlock, non vengono mai acquisite contemporaneamente lock su due istanze distinte. Il problema dell'attesa circolare si potrebbe evitare con acquisizione ordinata delle lock, magari utilizzando **LinkedHashMap** e/o sfruttando eventuali invarianti di utenti e post, tuttavia è stato preferito l'approccio sopra nominato.

Il pattern seguito è circa il medesimo in tutte le funzioni interessate, possiamo utilizzare **listFollowing()** come esempio:

```
1 public static HashSet<UserWrap> listFollowing(String username) throws NotExistingUser {  
2     User user = checkUsername(username);  
3     HashSet<UserWrap> toRet = new HashSet<>();  
4     user.readl.lock();  
5     HashSet<String> following = new HashSet<String>(user.following);  
6     user.readl.unlock();  
7     for (String followed : following)  
8         toRet.add(new ServerInternal().new UserWrap(users.get(followed)));  
9     return toRet;  
10 }
```

Per prima cosa otteniamo una reference all'oggetto **User**, dopodiché acquisiamo la lock in lettura per poi sbloccarla subito dopo aver copiato la struttura dati su cui dovremo iterare (in questo caso **user.following**).

Abbiamo ottenuto una *shallow copy*, ma essendo il contenuto dei Set utilizzati sempre **String**, dunque immutabile, o **Integer** mai modificati rappresentanti gli ID dei post, non c'è bisogno di eseguire una *deep copy*. A questo punto

possiamo iterare sulla copia locale del Set in questione, acquisendo e rilasciando la lock in lettura su ciascun utente ottenuto tramite `users.get(followed)` (riga 9).

L'acquisizione e il rilascio della lock avvengono nel costruttore di `UserWrap`.

**Osservazioni** Innanzitutto possiamo notare, assumendo che un determinato utente possa essere loggato da un solo terminale alla volta, che l'acquisizione della lock in lettura su `user` non è necessaria, dato che non è possibile eseguire più operazioni per volta richieste da uno stesso utente e non esiste alcuna possibilità per altri utenti loggati di modificare il campo `following` non loro.

Tuttavia il server inizialmente era stato realizzato per funzionare considerando anche la possibilità che uno stesso utente fosse loggato da più di un terminale e l'implementazione è rimasta; in seguito a chiarimenti sulla specifica è stata aggiunta la limitazione sopra citata, che si può rimuovere impostando a `false` il campo `exclusiveLogin` nel file di configurazione `.json` del server.

Riassumendo, abbiamo stabilito che è safe leggere la struttura dati interessata, in questo caso `user.following`, ma in generale, questo potrebbe non essere sempre vero.

Per esempio, in `showFeed()` i blog degli utenti seguiti potrebbero subire variazioni fra il momento della copia degli stessi e l'invio della risposta al client, tuttavia garantire che ciò non succeda porterebbe ad una eccessiva serializzazione delle richieste, indebolendo o eliminando i vantaggi di un server multithreaded; inoltre il peggio che può succedere è che nel Set di post restituito ci sia un post in più o in meno a causa di una recentissima aggiunta o rimozione, dunque "poco male".

L'eventualità sopra descritta è la situazione che si presenta pressoché sempre eliminando il limite *"massimo un client per ogni utente loggato"*. Una gestione della concorrenza di questo genere ci porta ad ottenere una forma di consistenza detta *eventual consistency*.

**Delete e Rewin** `deletePost()` oltre ad eliminare l'entry richiesta dalla Map `posts`, si occupa anche di eliminare l'id del post rimosso dai blog dell'owner e dei rewiner.

Nel caso in cui `rewinPost()` e `deletePost()` operino contemporaneamente su uno stesso post vogliamo evitare che (in ordine) il thread "rewiner" ottenga una reference all'istanza `Post`, venga schedulata ed eseguita `deletePost()` e infine venga completata l'operazione di rewin, ovvero l'aggiunta dell'id, che ormai si riferisce a un'istanza di `Post` non più esistente/valida, al blog del rewiner; non sarebbe grave, il risultato di `posts.get(id)` viene sempre controllato, ma risulterebbe certamente poco elegante.

```
1 public static int deletePost(int postID, String username) throws NotExistingUser,
   NotExistingPost {
2     ...
3     posts.computeIfPresent(idPost, (id, post) -> {
4         if (post.owner.equals(username)) {
5             wrapper.ps = post;
6             wrapper.ps.readl.lock();
7             return null;
8         }
9         return post;
10    });
11    if (wrapper.ps == null) // client isn't the owner
12        return -1;
13    HashSet<String> rewiners = new HashSet<String>(p.rewiners);
14    p.readl.unlock();
15    ...
16 }
17
18 public static int rewinPost(int idPost, String username) throws NotExistingUser,
   NotExistingPost {
19     ...
20    p = posts.computeIfPresent(idPost, (id, post) -> {
21        post.writel.lock(); // if present acquire lock
22        return post; // re-assign the same value
23    });
24    if (p == null) // another thread called deletePost() in the meantime
```

```

25         throw new NotExistingPost();
26     user.writel.lock();
27     if (checkFeed(user, p)){ // we can acquire readlock while holding writelock
28         p.rewiners.add(username);
29         user.blog.add(p.idPost);
30     }
31     else
32         toRet = 2;
33     user.writel.unlock();
34     p.writel.unlock();
35     ...
36 }

```

Per ovviare a questo problema viene sfruttata l'atomicità del metodo `computeIfPresent()` e la lock sull'istanza `Post`. Il risultato è che i casi si riducono a due sole possibilità:

1. Viene eseguita prima `computeIfPresent()` in `deletePost()` e questo porterà al verificarsi della condizione a riga 24;
2. Viene eseguita prima `computeIfPresent()` in `rewinPost()`, il thread "deleter" si bloccherà in attesa alla riga 6, finché l'utente non sarà aggiunto ai rewiner del post e l'id del post al blog del rewiner. Quando viene ripresa l'esecuzione del thread bloccato, `rewinPost()` sarà già terminata.

**NB:** `rewinPost()` è l'unico caso in cui vengono eseguite due `lock()` annidate.

### 3 Client

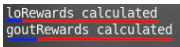
Per prima cosa il client cerca il file di configurazione, se non lo trova (o non sono specificati tutti i campi) vengono utilizzati dei valori di default.

Dopodiché cerca di localizzare il `Registry` per ottenere uno stub delle funzioni esposte dal Server.

A questo punto apre un `SocketChannel` e lo collega all'indirizzo remoto del server.

### Richieste

Le richieste inserite da CLI vengono tutte inviate al server e si attende la risposta, che viene stampata su `stdout`, con alcune eccezioni che richiedono una gestione differente:

- **register** invoca l'operazione di registrazione sullo stub del server, comunicando esclusivamente tramite RMI.
- **login** innanzitutto controlla che non ci sia già un utente loggato sul client, dopodiché inoltra la richiesta di login al server tramite il `SocketChannel` e, in caso di successo, si registra al servizio di aggiornamento della lista dei follower invocando il metodo `ROSint.registerForCallback` sullo stub del server e avvia uno `snifferThread` che si mette in ascolto su un `MulticastSocket` e stampa i datagrammi UDP ricevuti. Nella risposta del server sono inclusi l'indirizzo multicast e la porta a cui collegarsi.
- **notify** comando aggiunto per abilitare/disabilitare la stampa della notifica di avvenuto calcolo delle ricompense e delle modifiche ai follower che portava a scomodi artefatti di questo genere. 
- **list followers** viene stampato il contenuto del Set di follower aggiornato periodicamente tramite RMI. Non viene inoltrata alcuna richiesta tramite TCP al server, nonostante sia in grado di gestirla.

### Testing

Per poter eseguire uno stress test del server è stato aggiunto il campo `"cli"` nella configurazione del client. Se settato a `true`, il processo client non leggerà le richieste da tastiera, ma utilizzerà come input dei comandi passati come argomento all'eseguibile separati da `'+'`.

Non è un sistema robusto, eventuali `'+'` nei comandi (esempio: `post "+title" "content"`) rendono il parsing inefficace, è stato realizzato ed utilizzato esclusivamente per scopi di testing.

Inoltre, se il flag `cli` è settato, tutte le stampe saranno disabilitate.

In questo modo, è possibile lanciare dei processi client fornendo già un set di richieste da inviare come *cmdline-arguments*; Questa possibilità è sfruttata dallo script .sh ausiliario **spawnclients.sh**

## 4 Test

### Script presenti

Nella cartella *tests* sono presenti i seguenti script .sh:

- **(client|server)CompAndRun.sh** compila i file .java e lancia *ClientMain|ServerMain*
- **(client|server)Build.sh** compila i file .java e genera il rispettivo file .jar
- **stressTest.sh** Compila i file sorgente e genera Server.jar e Client.jar. esegue il server in background (lasciando le sue stampe abilitate), dopodiché fa partire un numero N di processi *spawnclients.sh* in background e attende 40 secondi; infine termina gli N processi lanciati e invia **SIGTERM** al processo server.  
Il server viene avviato impostando *"exclusiveLogin" : "false"*, dunque permettendo allo stesso utente di essere loggato da più processi client contemporaneamente.
- **spawnclients.sh** script ausiliario che sceglie da un array ed esegue un client con alcune richieste passate come *cmdline-arguments*, quando il processo client lanciato termina ne sceglie un altro e ricomincia, finché non viene terminato da un segnale.

### Osservazioni Stress Test

L'output generato dal test mostra che i Thread del pool si alternano nell'elaborazione delle richieste in modo piuttosto omogeneo.

Nonostante l'esecuzione si blocchi per breve tempo di tanto in tanto, sembra che ciò sia dovuto al tempo necessario alla JVM per avviarsi ed eseguire i processi Client; una rapida analisi dello stato dei Thread del processo Server con `jcmd <PID> Thread.print` durante questi momenti di apparente "stallo" non ha mostrato evidenza di deadlock.