
UNIVERSITÀ DI PISA

# Relazione Progetto WINSOME

## Corso di Laboratorio di Reti

## A.A. 2021/22

Satta Marcello

Corso B - 580495

## Content

# 1 Introduzione

WINSOME è un social network basato su architettura client-server che, oltre a garantire agli utenti registrati la possibilità di condividere le proprie idee, ricompensa questi ultimi se i post pubblicati appaiono interessanti agli occhi dei followers, che possono esprimere le proprie opinioni in merito, tramite commenti e voti (positivi o negativi) e che ricevono essi stessi una parte di ricompensa in qualità di curatori dello specifico post.

L'applicazione è fortemente ispirata a STEEMIT, social media basato su blockchain con un sistema di ricompense simile ma più complesso.

Gli utenti si possono registrare sulla piattaforma specificando username (univoco per ogni utente), password e da uno a cinque tags, ovvero categorie di interesse che determinano le preferenze dell'utente per quanto riguarda i tipi di post che vorrebbero visualizzare sul proprio feed e pubblicare sul proprio blog.

Gli utenti possono "cercarsi" sulla piattaforma per interessi comuni e "seguirsi" vicendevolmente o meno, grazie alle operazioni di "follow" e "unfollow", condividere post degli utenti appartenenti alla lista dei seguiti (operazione di "rewin") e, come detto in precedenza, diventare curatori dei post condivisi sul proprio feed.

Le ricompense vengono calcolate ad intervalli ciclici e aggiornano il portafoglio degli utenti interessati in caso la ricompensa associata al proprio contributo (sia da autore, sia da curatore) sia maggiore di 0.

Gli utenti possono controllare il proprio conto e l'insieme delle transazioni a loro favore tramite l'operazione "wallet" e ottenere le stesse informazioni convertite al tasso di cambio della criptovaluta Bitcoin, anziché nella moneta proprietaria Wincoin.

# 2 Architettura generale del progetto

Il progetto, come è suggerito all'interno dell'introduzione, si basa sul paradigma client-server. Gli utenti possono accedere all'applicazione tramite il client che fornisce un'interfaccia testuale da terminale e possono così interagire con il server, che si occupa della maggior parte delle operazioni, mantiene in memoria le strutture dati e si fa carico del salvataggio delle informazioni riguardanti gli utenti e i post su file JSON.

Le tecnologie di rete utilizzate per la comunicazione tra entrambe le parti sono diverse.

Innanzitutto, un utente può effettuare la registrazione alla piattaforma sfruttando il servizio RMI tramite cui il client crea un riferimento agli oggetti remoti resi disponibili dal server, utilizzando un Registry conosciuto da entrambi i lati. Dato che la specifica richiede di mantenere lato client la lista dei followers di un utente, il Registry è stato sfruttato anche per reperire suddetta lista dal server all'avvio del client.

La tecnologia RMI Callback è invece utilizzata per ricevere due tipi di notifiche, una per avvisare l'utente che ha ricevuto un nuovo follow da parte di un altro utente e una per avvisare del contrario, ovvero che un utente ha abbandonato la lista di followers dell'utente di partenza.

Una volta effettuata la registrazione, durante la probabile successiva operazione di login, viene instaurata una connessione TCP tra server e client che occorre alle due parti per comunicare sulla richiesta e l'esito della maggior parte delle operazioni. Il server, una volta ricevuta una richiesta di connessione attiva un thread per ogni client (il server è stato realizzato con Java I/O e threadpool).

Il client, il cui scheletro portante è costituito da uno switch che indirizza alle varie operazioni disponibili, manda una richiesta alla volta al server. Il codice del task lato server che si occupa del singolo client è stato costruito in modo chirale rispetto al client e possiede al suo interno uno switch che, in base alla richiesta dell'utente, si occupa di incanalare il flusso d'esecuzione verso l'operazione corretta.

I messaggi tra client e server sono stringhe che vengono riconosciute ambo i lati per garantire la corretta esecuzione dell'operazione. Ogni operazione richiede un numero di parametri obbligatori e diverse restrizioni sul tipo e sulla qualità, condizioni che vengono controllate da entrambi i lati per maggior robustezza.

Parallelamente al processo server e ai processi client, ogni entità è provvista di un thread parallelo che si occupa, lato server, di calcolare le ricompense e mandare su un gruppo Multicast un datagramma UDP di avvenuto aggiornamento dei conti e delle transazioni e, lato client, di collegarsi al gruppo Multicast e di ricevere il messaggio del suddetto aggiornamento. Il server condivide con ogni client, durante l'invio del messaggio di conferma

dell'operazione di login, l'indirizzo IP e la porta del gruppo Multicast su cui stare in ascolto.

La terminazione lato client è completamente delegata all'utente che può effettuare l'operazione "quit" per terminare l'applicazione. Se l'utente non effettua l'operazione "logout" ma decide di terminare immediatamente con la "quit" viene effettuato implicitamente il logout dal sistema.

In caso di crash del client (per esempio se viene lanciato un segnale di interrupt da terminale, control + C), l'operazione di logout non avviene. L'utente resta così intrappolato online pur non potendo interagire con il server. Per ovviare a questo problema viene chiesto all'utente di accedere al sistema utilizzando la "login_after_crash" che corregge questo malfunzionamento.

La terminazione del server avviene tramite interrupt da terminale. Idealmente il server dovrebbe mantenersi sempre attivo in attesa di nuove richieste di connessione.

Il file di configurazione con cui sia server che client recuperano tutte le informazioni necessarie al corretto funzionamento del sistema è stato creato grazie a Java.properties.

## 2.1 Organizzazione del codice

Il codice è stato organizzato secondo il seguente schema:

- src
    - server
        - User.java
        - Wallet.java
        - Transaction.java
        - Post.java
        - Comment.java
        - ClientTask.java
        - ServerMulticast.java
        - Server.java
        - ServerMain.java
    - client
        - Client.java
        - ClientMain.java

        - ClientMulticast.java
    - common
        - RMIInterface.java
        - ClientCallbackNotify.java
- lib
    - jackson-annotations-2.9.7
    - jackson-core-2.9.7
    - jackson-databind-2.9.7
- configFile
    - src
        - ConfigFileMain.java
    - configServer
        - configFileServer.properties
    - configClient
        - configFileClient.properties
- backup (cartella e contenuto al suo interno creati al primo avvio del Server)
    - backup.json

# 3 Server

## 3.1 ServerMain

Il Server viene lanciato tramite la funzione `main` presente all'interno del file ServerMain.java. Come prima operazione vengono letti dal file "configFileServer.properties" tutti i parametri necessari al corretto funzionamento del sistema, che comprendono:

- L'intervallo di tempo tra cui vengono calcolate le ricompense relative ai singoli post, da assegnare all'utente autore e ai curatori;
- La percentuale di ricompensa da assegnare all'autore del singolo post (la differenza verrà divisa tra i curatori del post);
- Il numero di porta su cui il Server attende le richieste di connessione dei client;

- Il numero di porta su cui verrà inviato il datagramma UDP che avvisa i client del corretto calcolo e aggiornamento delle ricompense. Il numero di porta verrà inviato al login del singolo client per potergli permettere di attendere il datagramma.
- Il numero di porta su cui il Registry RMI rimane in ascolto;
- Il nome con cui il riferimento all'oggetto remoto è registrato sul Registry. Tramite questo nome i client potranno effettuare l'operazione di `lookup` e invocare i metodi dell'oggetto remoto (da ricontrollare);
- Indirizzo IP su cui il Server invia e riceva i messaggi scambiati con il client;
- Indirizzo IP del gruppo Multicast su cui i client ascoltano e aspettano la conferma di avvenuto calcolo delle ricompense e aggiornamento degli wallet;
- Il numero di backlog che indica la lunghezza massima della coda delle richieste di connessione;

Successivamente, dopo la creazione dell'oggetto `Server`, di cui si discuterà più avanti, in cui sono implementati tutti i metodi di gestione delle richieste, viene lanciato il thread che si occupa del calcolo delle ricompense (il metodo di calcolo (`updateRewards`) è presente all'interno dell'oggetto Server, passato come parametro) e dell'invio, ai clienti iscritti al gruppo Multicast, del messaggio di avvenuto calcolo e aggiornamento dei portafogli virtuali.

In seguito, viene creato l'oggetto remoto (implementato dall'oggetto Server tramite l'interfaccia `RMIInterface`) e il Registry su cui esso è iscritto, viene creato un threadpool che gestirà i thread (uno per client) che si occupano della ricezione delle richieste di operazione da parte dei client sulla connessione TCP e, dopo la creazione del `ServerSocket` sull'indirizzo IP e sulla porta specificati sul file di configurazione (localhost, 1501), il `ServerMain` si mette in attesa, all'interno di un ciclo `while` di nuove richieste di connessione da parte dei client.

## 3.2 Server

La classe `Server` contiene le strutture dati che mantengono le informazioni degli utenti e dei post creati da essi ed i metodi che agiscono su queste strutture.

Inizialmente è stato deciso di mantenere due strutture separate, una `ConcurrentHashMap<String, User>` utile a mantenere le informazioni degli utenti (la classe `User` verrà discussa in seguito) e una `ConcurrentHashMap<Integer, Post>` contenente le informazioni relative ai post (anche la classe `Post` verrà discussa in seguito). Con questo tipo di impostazione la collezione di `Post` avrebbe mantenuto tutti i post (non eliminati dall'autore) di tutti gli utenti e sebbene l'accesso alla `ConcurrentHashMap` risultasse agevole per alcune operazioni, come quella di calcolo delle ricompense, è stato deciso di cambiare la struttura dei dati di modo che ogni oggetto utente contenesse, in qualità di autore, una `ConcurrentHashMap` di `Post`, cosa che risulta più logica e più comoda per diverse operazioni (si pensi alle operazioni di "blog" e "show_feed" che richiedono rispettivamente di mostrare l'elenco di post creati dall'utente e l'elenco di post creati dagli utenti seguiti).

La scelta del tipo di collezione è ricaduta sulle `ConcurrentHashMap` per due motivi principali: innanzitutto, sia gli utenti, sia i post devono mantenere un identificatore univoco come richiesto dal committente, nel primo caso si tratta di una stringa username che modella il nickname dell'utente, nel secondo si tratta di un numero intero che identifica l'id del post. Entrambi queste variabili, date la loro univocità, sono state sfruttate come chiavi all'interno della `ConcurrentHashMap`; in secondo luogo, la struttura in questione permette accesso in lettura e scrittura thread-safe e lo fa sfruttando la propria implementazione divisa in 16 (di default) segmenti o bucket provvisti di altrettante lock, che permette l'accesso da parte di un numero arbitrario di thread in lettura e di un numero fisso di thread in scrittura (sempre 16 di default). Inoltre, gli inserimenti e le rimozioni, operazioni utilizzate all'interno del progetto sono state rese atomiche e il valore di ritorno è sempre controllate per garantire che non vi siano state sovrapposizioni nell'aggiunta o nella cancellazione di qualche elemento.

È presente inoltre una collezione, sempre `ConcurrentHashMap<String, ClientCallbackNotify>` che modella gli utenti, o meglio, i client registrati al servizio di Callback. Quanto un utente decide di seguirne un altro, quest'ultimo riceverà una notifica che verrà stampata su schermo se l'utente è online. L'interfaccia su cui sono dichiarati i metodi che permettono il

meccanismo di notifica è implementata all'interno della classe Client, che verrà descritta in seguito.

La classe `Server` inoltre contiene come variabili private la directory e il file (sia come oggetto `File`, sia come `String`, con cui ne è specificato il nome) su cui verrà eseguito il backup dei dati in formato Json (è stata utilizzata la libreria Jackson), l'oggetto `objectMapper` per serializzare e deserializzare gli oggetti verso e dal file di backup, una variabile `change` che contiene il valore di cambio alla criptovaluta Bitcoin, e la suddetta percentuale di retribuzione nei confronti dell'autore, passata come parametro dopo essere stata letta dal file di configurazione dal `ServerMain`.

Come già specificato l'oggetto server viene creato all'avvio del `ServerMain`. All'interno del costruttore è presente la chiamata al metodo `loadBackup()` che carica in memoria i dati salvati sul file Json. Se la cartella di backup non esiste, viene creata e i file vengono salvati ad ogni operazione di modifica della struttura e alla logout dell'utente tramite il metodo `backup()`.

## 3.3 ClientTask

Passiamo ora alla descrizione della classe `ClientTask`, descrizione che si intersecherà con quella riguardante la classe Server, dato che molti metodi che quest'ultima implementa vengono chiamati all'interno della classe `ClientTask`.

Questa classe modella un task che verrà mandato in esecuzione da un thread creato all'interno del threadpool. Il task si occupa di ricevere richieste da parte di un singolo client che riguardano le operazioni da svolgere lato server e di inviare le relative risposte.

Infatti, possiede come variabili private:

- un oggetto server, che modella appunto il server;
- il socket su cui il client si è collegato;
- un `BufferedReader` per poter leggere le richieste inviate dal client;
- un `BufferedWriter` per poter inviare le risposte verso il client;
- il numero di porta del gruppo Multicast, parametro che verrà inviato al client durante l'operazione di risposta al login;
- l'indirizzo IP del gruppo Multicast, anch'esso inviato al client con la stessa modalità del numero di porta;

Il `ClientTask` implementa l'interfaccia `Runnable` e di conseguenza deve implementare il metodo `run()`.

All'avvio del task, il programma prepara uno `StringTokenizer` per poter effettuare il parsing del messaggio di richiesta da parte del client. Dopo di che, entra in ciclo `do while` la cui prima operazione, bloccante, è una `readline`, all'interno del metodo `get` (si occupa della lettura dal `BufferedReader`) che attende un nuovo messaggio da parte del client. Quando il messaggio arriva, la stringa viene "tokenizzata" e il primo token, salvato nella variabile `operation`, è stato inserito come espressione all'interno dello `switch case` che reindirizza ai vari case che si occupano di preparare i parametri dell'operazione richiesta attraverso lo `StringTokenizer`, e chiamare il metodo corretto definito all'interno della classe Server. Di seguito ecco le etichette che indirizzano ai vari case all'interno dello `switch`:

- "quit": richiede la terminazione del programma lato client. La "quit" effettua il logout dell'utente e chiude il socket di comunicazione;
- "help": il client richiede al server le informazioni sulle operazioni disponibili. Il server risponde con una lista di queste di cui per ognuna una breve descrizione;
- "login": il client richiede di poter effettuare l'operazione di login digitando "login username password". Se l'utente non è già loggato, è reduce da un fallimento lato client e non ha inserito credenziali errate, l'operazione va a buon fine;
- "login_after_crash": il client viene avvisato della possibilità di questa operazione in caso tenti di fare il login, le credenziali sono corrette, ma il sistema non gli permette di accedere all'applicazione. Questa operazione può essere richiesta in caso di crash dell'applicazione lato client. Infatti, in questo caso, il server non ha la possibilità di effettuare correttamente il logout dell'utente. L'operazione è molto simile alla "login" ("login_after_crash username password"), ma prima viene effettuata l'operazione di logout non effettuata prima del crash. Una volta loggato, l'utente può riprendere l'utilizzo corretto dell'applicazione;

- “logout”: il client richiede l’operazione di logout, viene cambiato il valore della variabile che indica lo stato dell’utente (da online a offline, `boolean logged`). L’operazione è richiesta tramite la stringa “logout username”;
- “follow”: il client richiede di seguire un utente (“follow username”). Tramite oggetto remoto, se l’altro utente si trova online riceverà una notifica di avviso;
- “unfollow”: il client richiede di non seguire più un utente (“unfollow username”). Tramite oggetto remoto, se l’altro utente si trova online riceverà una notifica di avviso;
- “rate”: il client richiede di votare un post presente all’interno del suo feed. Una richiesta di voto di un post non presente all’interno del feed (ovvero il cui autore non è seguito dall’utente che effettua la richiesta) riceverà una risposta negativa. Si può esprimere un solo voto per post e non può essere revocato o modificato. L’operazione va a buon fine se il post è presente e se il voto è valido, ovvero se è uguale a 1 o -1. La richiesta dell’operazione è della forma “rate idPost vote”;
- “list_users”: il client richiede la lista degli utenti con cui possiede almeno un tag in comune. Operazione “list_users”;
- “list_following”: il client richiede la lista di utenti che segue. Operazione “list_following”;
- “blog”: il client richiede la lista di post di cui è autore. Operazione “blog”;
- “show_feed”: il client richiede la lista di post condivisi dagli utenti che segue. Operazione “show_feed”;
- “show_post”: il client richiede di visionare un singolo post. Il post deve appartenere al feed dell’utente, oppure al proprio blog. Operazione “show_post idPost”;
- “post”: il client richiede la pubblicazione di un nuovo post. Operazione “post “title” “content”. Le stringe “title” e “content” non possono essere nulle e la loro lunghezza non deve superare rispettivamente i 20 e i 500 caratteri;

- “delete”: il client richiede la cancellazione di un post di cui è l’autore. Le richieste di cancellazione che riguardano post il cui autore è un altro utente non verranno accettate;
- “rewin”: il client richiede di condividere un post creato da un altro utente. La richiesta di rewin relativa a post non presenti sul proprio feed, ovvero il cui autore è seguito dall’utente, non verranno accettate;
- “comment”: il client richiede la pubblicazione di un commento relativo ad un post presente sul proprio feed o sul proprio blog. Operazione “comment idPost “comment””;
- “wallet”: il client richiede di visionare il proprio portafoglio virtuale. Operazione “wallet”;
- “wallet_btc”: il client richiede di visionare il proprio portafoglio virtuale cambiato di valuta in Bitcoin. Operazione “wallet_btc”;

Le operazioni “register” e “list_followers”, dato che la prima è implementata tramite servizio RMI e la seconda recupera la risposta localmente (la lista di followers è aggiornata tramite il Callback), non vengono gestite tramite richiesta-risposta su connessione TCP.

Affinché le richieste siano corrette devono rispettare alcune condizioni riguardanti il numero di token (per esempio, per il login sono necessari tre token, “login” “username” “password”) e alcune condizioni sul contenuto del messaggio (per esempio un voto riguardante un post può essere o 1 o -1, ma nient’altro). Queste condizioni sono controllate sia lato client che lato server per maggiore robustezza.

Come già riportato in precedenza, la comunicazione sul socket TCP avviene tramite i metodi `get` e `send`, che leggono e scrivono rispettivamente sul `BufferedReader` e `BufferedWriter`.

## 3.4 ServerMulticast

La classe `ServerMulticast` modella il thread parallelo necessario al calcolo delle ricompense di cui si è parlato nel capitolo sull’architettura generale del progetto. La classe possiede le seguenti variabili private:

- l’oggetto server passato come parametro al costruttore all’interno del `main` (`ServerMain`);

- un oggetto `Timestamp` che modella l'ultimo istante di tempo in cui si è svolto il calcolo delle ricompense;
- un intero che rappresenta l'intervallo di tempo in millisecondi tra un calcolo delle ricompense e il successivo;
- il numero di porta del gruppo Multicast su cui verrà inviato il messaggio di avvenuto calcolo e aggiornamento portafogli;
- l'indirizzo IP del gruppo Multicast;

Queste ultime tre variabili corrispondono a quelle lette dal file di configurazione del Server.

La classe `ServerMulticast` implementa l'interfaccia `Runnable` e conseguentemente il metodo `run()`. Al suo avvio predispone un `DatagramSocket` sull'indirizzo specificato dal file di configurazione e il datagramma da inviare al gruppo Multicast. Dopo di che entra in un ciclo `while(true)` in cui per prima cosa chiama il web service random.org per ottenere un numero intero generato casualmente (del servizio web si parlerà più avanti) che fungerà da tasso di cambio per la moneta virtuale Bitcoin, successivamente il thread rimane in stato di sleep per tutta la durata dell'intervallo di tempo tra il calcolo delle ricompense e il successivo, infine calcola le ricompense e aggiorna i portafogli grazie al metodo presente nella classe Server (`updateRewards()`) ed invia il messaggio al gruppo Multicast di avvenuta operazione.

## 3.5 Le classi del Server

Come si potrà intuire dai capitoli precedenti, all'interno della directory Server sono presenti delle classi che modellano il sistema. In particolare, possiamo notare come le classi presenti si compongono tra loro in un gioco di scatole cinesi.

Da una parte abbiamo la classe `User` che modella un singolo utente, che conterrà al suo interno una collezione di `Post`, classe che modella un singolo post. All'interno della classe post abbiamo una collezione di oggetti di tipo `Comment`, ovvero i commenti che gli utenti possono scrivere sotto ad ogni post.

All'interno della classe `User` vi è anche un oggetto `Wallet` che rappresenta il portafoglio virtuale dell'utente ed al suo interno una collezione di

`Transaction`, ovvero le transazioni accreditate sul portafoglio ad ogni calcolo di ricompense dell'autore dei post.

Segue la descrizione delle classi nominate.

### 3.5.1 La classe User

La classe è provvista delle seguenti variabili private:

- `String username` : nome univoco dell'utente;
- `String password` : password di accesso all'applicazione;
- `ConcurrentLinkedQueue<String> tags` : collezione di tags scelti dall'utente alla registrazione;
- `ConcurrentLinkedQueue<String> followed` : collezione di utenti followers, ovvero che seguono l'utente;
- `ConcurrentLinkedQueue<String> following` : collezione di utenti seguiti dall'utente;
- `ConcurrentHashMap<Integer, Post> blog` : collezione di post creati dall'utente;
- `boolean logged` : flag vero o falso che indica se l'utente è online o meno;
- `ReentrantLock userLock` : lock utilizzata al login e al logout dell'utente dal sistema;
- `Wallet wallet` : oggetto di tipo Wallet che modella il portafoglio dell'utente.

La classe si compone inoltre di due costruttori (uno utilizzato da Jackson per creare l'oggetto a partire dal file Json), dei metodi set e get che riguardano tutte le variabili private ed alcuni metodi aggiuntivi per l'inserimento e la rimozione dei followers e dei following;

### 3.5.2 La classe Post

La classe contiene al suo interno le seguenti variabili private:

- `int id` : intero univoco che indica l'id del singolo post;
- `int nIterations` : numero di iterazioni di calcolo ricompense a cui il post è stato sottoposto;
- `String title` : titolo del post;
- `String content` : contenuto del post;

- `int positiveVotes` : numero di voti positivi che il post ha ricevuto;
- `int lastPositiveVotes` : numero di voti positivi che il post ha ricevuto al momento dell'ultimo calcolo ricompense, è necessario per calcolare la ricompensa solamente sui nuovi voti ricevuti;
- `int negativeVotes` : numero di voti negativi che il post ha ricevuto;
- `int lastNegativeVotes` : numero di voti negativi che il post ha ricevuto al momento dell'ultimo calcolo ricompense, è necessario per calcolare la ricompensa solamente sui nuovi voti ricevuti;
- `ConcurrentHashMap<String, Integer> votes` : collezione contenente i voti (positivi o negativi) espressi dai relativi utenti, il cui username è utilizzato come chiave;
- `ConcurrentHashMap<Integer, Comment> comments` : collezione contenente i commenti al singolo post;
- `int nComment` : numero di commenti inseriti.

Oltre ai metodi costruttori, getters e setters sono presenti due metodi relativi all'aggiunta di un voto. Il primo, `addVote`(), incrementa i voti positivi o negativi a seconda del parametro passato ed aggiunge un elemento alla collezione `votes`, inserendo come chiave il nome dell'utente che ha espresso il voto e come valore il voto stesso (1 o -1). Il secondo metodo `checkConsistency()` controlla che la dimensione della collezione `votes` sia consistente con il numero di voti inseriti, sia positivi che negativi.

### 3.5.3 La classe Comment

La classe `Comment` è composta dalle seguenti variabili private:

- `String author` : autore del commento;
- `String comment` : contenuto del commento;
- `Timestamp timestamp` : istante di tempo in cui il commento è stato inserito.

I metodi presenti all'interno della classe sono due costruttori e i metodi get e set delle variabili private.

La variabile `timestamp` è utile per discriminare i nuovi commenti da prendere in considerazione nel calcolo delle ricompense (all'interno del metodo

di calcolo un commento viene preso in considerazione se e solo se il suo valore è maggiore dell'istante di tempo in cui è stato fatto il calcolo precedente).

### 3.5.4 La classe Wallet

All'interno della classe `Wallet` sono presenti le seguenti variabili private:

- `float balance` : indica il conto totale presente all'interno del portafoglio;
- `int nTransaction` : numero di transazioni aggiunte;
- `ConcurrentHashMap<Integer, Transaction> transactions`: collezione che tiene traccia di tutte le transazioni (la classe `Transaction` verrà discussa a seguire).

Oltre a costruttore, getters e setters, la classe contiene i seguenti metodi: `addBalance()`, che aggiunge una cifra passata come parametro al conto totale e `addTransaction()` che aggiunge una transazione alla collezione di `Transaction`.

### 3.5.5 La classe Transaction

La classe `Transaction` possiede le seguenti variabili private:

- `Timestamp timestamp` : indica l'istante di tempo in cui la transazione è stata creata;
- `double increment` : indica l'ammontare della transazione.

La classe dispone inoltre di due metodi costruttore, getters e setters per entrambe le variabili private.

## 3.6 Concorrenza

Come già annunciato in precedenza, la funzione `main` all'interno della classe `ServerMain` crea un threadpool che si occupa della gestione dei thread a cui verranno dati in pasto i `ClientTask`. Il `ThreadPoolExecutor`, creato attraverso il metodo statico `Exectuors.newCachedThreaPool()`, è di tipo `cachedThreadPool`, ciò significa che, all'avvio, vi sono 0 thread attivi e, se un client si connette sul socket del `Server`, un `ClientTask` che si occuperà delle richieste del client verrà mandato in esecuzione su un nuovo thread, e così per ogni altro client che richiede di connettersi, a patto che non vi siano thread

creati in precedenza ma non impegnati in alcun task. Infatti, se un client richiede l'operazione di "quit", il task termina e il thread rimane attivo per 60 secondi. L'attivazione di un thread è un'operazione costosa e, se arriva una nuova richiesta di connessione entro l'intervallo di tempo stabilito pocanzi, il thread "disoccupato" viene impiegato per ospitare l'esecuzione del nuovo `ClientTask`. Se il thread, passato il quanto di tempo, non riceve alcun lavoro da svolgere viene disattivato definitivamente.

L'accesso alle risorse condivise è garantito essere mutualmente esclusivo in modo implicito grazie alle strutture dati scelte per l'implementazione. Come già riportato infatti, le `ConcurrentHashMap` permettono la lettura simultanea di un numero arbitrario di thread e la scrittura simultanea di un numero fisso di thread, 16 di default.

Di conseguenza, sulla lettura di un particolare elemento della collezione non ci poniamo grossi problemi. Tuttavia, potrebbero sorgere degli accessi concorrenziali in fase di scrittura. Avrebbe senso aumentare il numero fisso di writer che possono scrivere simultaneamente sulla struttura? In parte si, garantirebbe una maggiore libertà di scrittura, tuttavia questo approccio non garantirebbe da solo la tanto cercata thread safety. Infatti, è possibile che vi sia comunque overlapping su uno stesso bucket o segmento. Per ovviare a questo problema è stata prestata particolare attenzione a tutte le operazioni di scrittura che vengono effettuate lungo il codice. I metodi utilizzati in scrittura sono i seguenti:

- `V putIfAbsent(K key, V value)`

Se la chiave specificata non è ancora stata associate ad alcun valore, viene associata con il valore dato. Restituisce il valore precedente associato alla chiave, o `null` se non c'era alcun valore associato alla chiave.

- `V remove(Object key)`

Rimuove la chiave ed il suo corrispondente valore dalla `HashMap`.

Restituisce il valore precedente associato alla chiave, o null se non c'era alcun valore associato alla chiave.

Grazie al valore di ritorno è possibile controllare se l'elemento è stato già mappato (nel caso della `putIfAbsent`) o non è presente all'interno della

collezione (nel caso della `remove`). In questo modo riusciamo a rendere atomiche e quindi sicure le operazioni di inserimento e rimozione.

Può sembrare un controsenso bloccare l'accesso dell'intera collezione con il monitor `synchronized` andando a perdere la sua proprietà concorrente intrinseca. Tuttavia, è stato ritenuto necessario, per lo stesso motivo per cui gli iteratori di questo tipo di collezioni sono definiti "weakly consistent", definire alcuni metodi come `synchronized`, metodi nel quale viene letta l'intera `HashMap` (si pensi al metodo di `backup()` o al calcolo delle ricompense).

Due altre operazioni delicate sono il login e il logout. Solo un client può accedere a o abbandonare un profilo utente, per questo utilizziamo delle `Lock` sulla variabile `User` che sta tentando di accedere o lasciare la piattaforma.

Le collezioni interne alle classi descritte in precedenza sono state definite come `ConcurrentLinkedQueue` per garantire simultaneità di accesso per le operazioni che le riguardano (per esempio la lista di followers e following).

La scelta di optare per questa implementazione, ovvero Java IO e threadpool, anziché di Java NIO, è stata dettata dalla semplicità con cui è stato possibile realizzare il Server. A discapito di ciò, c'è da dire che con Java NIO avremmo ottenuto migliori prestazioni sia in fatto di agilità del sistema, sia in fatto di scalabilità.

## 3.7 Il servizio web offerto da Random.org

Il tasso di cambio valuta dalla moneta virtuale WINCOIN a BITCOIN viene generato casualmente dal servizio web raggiungibile presso il sito internet random.org. Il metodo, che richiede un intero ed una frazione decimale casuale, è contenuto all'interno della classe `ServerMulticast` e viene chiamato ad ogni iterazione, dopo il calcolo delle ricompense e prima della `sleep()` che simula gli intervalli ciclici di attesa prima di un nuovo calcolo ricompense.

Il metodo in questione, a partire dall'URL del sito che calcola i numeri casuali apre un `InputStream` verso il servizio web con una richiesta HTTP GET. La risposta, contenuta nell'`InputStream` viene prima letta da un `InputStreamReader` ed infine ottenuta come variabile `int` (nel caso del numero intero) e double (nel caso della frazione decimale) tramite un `BufferedReader.readline()` ed il successivo parsing (`Integer.parseInt()`, `Double.parseDouble()`).

Il valore random del numero intero oscilla tra 1 e 20394 (tasso reale di cambio Euro-Bitcoin arrotondato), il valore frazionario decimale invece oscilla tra 0 e 1.

# 4 Client

Il Client è composto da tre file: il ClientMain.java, il Client.java e il ClientMulticast.java.

## 4.1 ClientMain

L'applicazione viene lanciata dalla funzione `main` presente nel primo file che esegue alcune operazioni preliminari prima di chiamare la funzione start presente all'interno del secondo file.

Queste operazioni comprendono innanzitutto la lettura del file di configurazione (.properties) dove sono presenti i seguenti dati necessari al corretto funzionamento del Client:

- il numero di porta su cui il Server è in ascolto;
- Il numero di porta su cui il Registry RMI rimane in ascolto;
- Il nome con cui il riferimento all'oggetto remoto è registrato sul Registry;
- L'indirizzo IP del client.

Successivamente, attraverso il metodo di `lookup`, il client chiede e ottiene il riferimento all'oggetto remoto reso noto dal Server sul Registry, utilizzando i parametri letti dal file di configurazione.

Dopo di che viene creato un nuovo oggetto `Client` e chiamata la funzione start che prende come parametri l'oggetto remoto grazie al quale un utente può registrarsi, indirizzo IP e porta su cui inviare le richieste di operazione e che da avvio alla vera e propria interfaccia utente.

## 4.2 La classe Client

La classe Client contiene al suo interno le seguenti variabili private:

- `String username` : stringa che identifica il nome dell'utente che effettua il login sulla piattaforma;

- `ConcurrentLinkedQueue<String> followers` : collezione che modella i seguaci dell'utente attivo sul sistema;
- `RMIInterface serverRMI` : oggetto remoto passato dal `ClientMain` alla creazione dell'oggetto `Client`. È necessario per l'operazione di registrazione dell'utente e per la registrazione al sistema di notifica;
- `ClientCallbackNotify stub` : stub del Client da passare al momento della registrazione al sistema di notifica;
- `Socket socket` : socket su cui l'utente invia le richieste al Server;
- `BufferedReader reader` : `BufferedReader` su cui leggere le risposte del Server;
- `BufferedWriter writer` : `BufferedWriter` su cui scrivere le richieste da mandare al Server;
- `String ClientSocketAddress` : indirizzo IP su cui il Client si connette;
- `int TCPServerPort` : numero di porta su cui il Client si connette.

Come detto in precedenza, come il `ClientTask`, lo scheletro portante del `Client` è uno switch (all'interno di un `do while`, interrotto quando il flag `quit == true`, modifica scatenata dall'operazione "quit" richiesta dall'utente) la cui espressione è ottenuta dal primo token della stringa letta in input da tastiera prima di entrare all'interno dello switch.

Prima che la richiesta venga inviata al Server, i parametri contenuti all'interno di quest'ultima sono oggetto di controllo di correttezza.

Le operazioni disponibili sono le stesse descritte all'interno del paragrafo riguardante il `ClientTask`, ad eccezione della "register", implementata a parte sfruttando il servizio RMI (il Client può invocare i metodi dell'oggetto remoto reso disponibile sul Registry dal server) e la "list_followers", la cui collezione è mantenuta localmente dal Client (in caso di accessi successivi al primo la lista viene richiesta al Server tramite servizio RMI, lo stesso usato per la registrazione utente, e quando arriva una nuova notifica di un nuovo o "vecchio" follower, che corrisponde ad una chiamata del Server all'oggetto remoto implementato dal Client attraverso l'interfaccia `ClientCallbackNotify`, l'username in questione viene rispettivamente aggiunto o rimosso dalla lista).

La classe, oltre ai già citati metodi di notifica Callback definiti all'interno dell'interfaccia `ClientCallbackNotify`, contiene inoltre le stesse funzioni di invio e ricezione sul socket TCP (`get` e `send`) presenti all'interno della classe `ClientTask`.

### 4.3 ClientMulticast

La classe `ClientMulticast` contiene le seguenti variabili private:

- `String multicastAddress`: indirizzo IP del gruppo Multicast;
- `int multicastPort`: numero di porta su cui rimanere in attesa del datagramma;

La classe `ClientMulticast` implementa l'interfaccia `Runnable` e modella un task mandato in esecuzione su un nuovo thread durante l'operazione di login all'interno della classe Client, task il cui compito è collegarsi al gruppo Multicast ed attendere il messaggio di notifica di avvenuto calcolo delle ricompense e aggiornamento portafogli.

Una volta in esecuzione, il task crea un `MulticastSocket` e si aggiunge al gruppo di ricezione della notifica. Dopo di che entra in un ciclo `while` (interrotto dal Client tramite una interrupt lanciata alla operazione di "logout" o a quella di "quit)" e si mette in attesa sulla `receive()` del datagramma UDP che, una volta arrivato verrà stampato su schermo. Una volta fuori dal ciclo `while` il socket abbandona il gruppo Multicast.

## 5 Le interfacce comuni

### 5.1 RMIInterface

In questa interfaccia, implementata dalla classe Server e resa disponibile come oggetto remoto (servizio RMI), si trovano i seguenti metodi:

- `register` : metodo di registrazione sulla piattaforma;
- `registeraForCallback` : registrazione al servizio di notifica Callback;
- `unregisteraForCallback` : metodo di cancellazione della registrazione al servizio di notifica Callback;
- `callBackFriends` : metodo con cui un Client, al nuovo avvio, richiede al Server la lista dei suoi followers, al fine di mantenerla in una struttura locale.

### 5.2 ClientCallbackNotify

L'interfaccia, implementata dal Client e resa disponibile come oggetto remoto che permette il funzionamento corretto del servizio di Callback, contiene al suo interno i seguenti metodi:

- `notifyNewFriend` : metodo di notifica che avvisa il Client di un nuovo follower;
- `notifyOldFriend` : metodo di notifica che avvisa il Client di un follower che ha smesso di seguire l'utente.

## 6 Istruzioni per la compilazione e l'esecuzione

Per la compilazione e l'esecuzione, posizionarsi da terminale UNIX all'interno della cartella WINSOME e inserire i seguenti comandi:

"bash compile.sh" // compilazione dei file .java;

"bash create_jar.sh" // creazione file .jar eseguibili.

Dopo la compilazione e la creazione dei .jar:

"bash run_server_jar.sh" // avvio del server;

"bash run_client_jar.sh" // avvio del client (da chiamare su un'altra finestra del terminale).

I file .class compilati verranno prodotti all'interno della cartella "out" che riprende la struttura della cartella "src" contenente i file sorgente.
All'interno della cartella "jar", che contiene due sottocartelle "client" e "server" al cui interno è presente un file MANIFEST.MF che specifica la Main-Class (ed inoltre nella cartella "server" sono anche presenti le librerie all'interno della cartella lib, specificate nel MANIFEST.MF relativo tramite Class-Path), verranno creati i file .jar.
Se si volesse eseguire anche il programma di creazione dei file di configurazione (compilato tramite il primo comando "bash compile.sh"), ecco il comando:

"bash run_clientConfig_jar.sh" // avvio programma di creazione file di configurazione.

Questa operazione, a meno di modifiche dei dati di configurazione che si trovano sul codice sorgente, è ridondante in quanto nella cartella relativa sono stati già inclusi i file di configurazione, sia lato Server sia lato Client.

# 7 Esempio di esecuzione

Viene ora illustrato un esempio di esecuzione. L'esempio è stato eseguito leggendo i seguenti parametri dai file di configurazione:

- Client:
  - RMIBindingName=ServerRMI
  - ClientSocketAddress=127.0.0.1
  - RMIRegistryPort=4002
  - TCPServerPort=1501
- Server:
  - RMIBindingName=ServerRMI
  - backLog=50
  - ServerSocketAddress=localhost
  - authorRewardPercentage=79
  - MultiCastAddress=239.255.1.3
  - RMIRegistryPort=4002
  - TCPServerPort=1501
  - UDPMultiCastPort=6789
  - timeIntervalRewards=30000

Il Server scrive un messaggio di ready e poi stampa ad intervalli regolari il messaggio di avvenuto calcolo ricompense.

```
marcellosatta@MBP-di-Marcello winsome % bash run_server_jar.sh
WINSOME Server ready.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
```

Utente “Marcello” si registra, effettua il login, crea un post di saluto (altre operazioni effettuate: “list_followers”, “follow Luigi”, “list_following”, “wallet”, “wallet_btc”, “logout”, “quit”):

```
marcellosatta@MBP-di-Marcello winsome % bash run_client_jar.sh
< -----Welcome to WINSOME!-----
> register Marcello Satta Chitarra Libri
< User registered successfully.
> login Marcello Satta
< User logged in.
> post "ciao!" "ciao a tutti, popolo di Winsome!"
< Post created successfully.
> Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
Server updated rewards and wallets.
< New friend: Luigi
Server updated rewards and wallets.
Server updated rewards and wallets.
list_followers
< List of followers:
Luigi
> follow Luigi
< Marcello started to follow Luigi.
> Server updated rewards and wallets.
list_following
< Luigi

> wallet
< Balance: 0,14 WINCOIN

Transactions:
Timestamp: 2022-07-19 13:02:32.583 - Increment: + 0,14 WINCOIN

> wallet_btc
< Balance: 299,89 BITCOIN

Transactions:
Timestamp: 2022-07-19 13:02:32.583 - Increment: + 299,89 BITCOIN

> logout Marcello
< User logged out.
> quit
See you soon! :)
```

Utente “Luigi” si registra, esegue il login, segue “Marcello”, vota e commenta il suo post (altre operazioni effettuate: “help”, “list_users”, “show_feed”, “show_post”, “wallet”, “wallet_btc”, “logout”, “quit”):

```
marcellosatta@MBP-di-Marcello winsome % bash run_client_jar.sh
< -----Welcome to WINSOME!-----
> register Luigi 1234 Libri
< User registered successfully.
> list_users
< You are not logged in.
> help
< Register or login to access WINSOME or quit if you want to leave.
Here are the possible operations:

 - register username password tags [from one to five] : register a new user;
 - login username password : login with the specified username and password;
 - login_after_crash username password : login with the specified username and password in case of crash of the client;
 - quit : close the application.
> login Luigi 1234
< User logged in.
> list_users
< Marcello

> follow Marcello
< Luigi started to follow Marcello.
> show_feed
< ID: 0
Author: Marcello
Title: ciao!
```

```
> Server updated rewards and wallets.
show_post 0
< Author: MarcelloTitle: ciao!
Content: ciao a tutti, popolo di Winsome!
Positive votes: 0
Negative votes: 0
Comments:

> rate 0 1
< Post rate successfully
> comment 0 "ciao marcello!"
< Comment created successfully.
> Server updated rewards and wallets.
wallet
< Balance: 0,04 WINCOIN

Transactions:
Timestamp: 2022-07-19 13:02:32.583 - Increment: + 0,02 WINCOIN
Timestamp: 2022-07-19 13:02:32.583 - Increment: + 0,02 WINCOIN

> wallet_btc
< Balance: 169,15 BITCOIN

Transactions:
Timestamp: 2022-07-19 13:02:32.583 - Increment: + 84,58 BITCOIN
Timestamp: 2022-07-19 13:02:32.583 - Increment: + 84,58 BITCOIN

> logout Luigi
< User logged out.
> quit
See you soon! :)
```