Università di Pisa
Corso di Laurea in Informatica

WINSOME: a reWardINg SOcial Media

Progetto di Fine Corso A.A. 2021/22
Reti di calcolatori e laboratorio

Francesco Botrugno – matr. 571436

Indice

1. Struttura di file e cartelle

```
src:
¦   ClientMain.java
¦   configClient.txt
¦   configServer.txt
¦   ServerMain.java
¦
+---classes
¦       Comment.java
¦       Hash.java
¦       Post.java
¦       processingClients.java
¦       Transaction.java
¦       User.java
¦       Wallet.java
¦       Winsome.java
¦
+---database
¦       lastRewarding.txt
¦       passwords.txt
¦       posts.txt
¦       progressiveID.txt
¦       users.txt
¦
+---exceptions
¦       UsernameAlreadyInUseException.java
¦       UserNotFoundException.java
¦
+---lib
¦       gson-2.8.2.jar
¦
+---rmi
¦       FollowerServiceImpl.java
¦       FollowerServiceInterface.java
¦       NotifyEventImpl.java
¦       NotifyEventInterface.java
¦       RegistrationServiceImpl.java
¦    RegistrationService.java
¦
+---runnables
        clientMulticastRunnable.java
        estimateRewardRunnable.java
        RequestHandler.java
        updateDatabaseRunnable.java
```

`classes`

Contiene le classi degli oggetti che formano la struttura del social.

Il file **`Winsome.java`** è la classe che rappresenta il social, contiene le strutture dati più importanti, ovvero la lista degli utenti, la lista dei post, le credenziali degli utenti e la lista dei post aggiornata all'ultimo calcolo delle ricompense così da poterli poi confrontare con i nuovi.

`User.java` contiene le informazioni relative ad un singolo utente, analogamente **`Comment.java`** e **`Post.java`** sono classi per i commenti e i post.
`Wallet.java` è il portafoglio di un utente, **`Transaction.java`** è una singola transazione.
`Hash.java` offre metodi per codificare le password.

processingClients.java rappresenta l'oggetto condiviso dai thread worker e dal main in cui vengono inseriti i client la cui richiesta è al momento in elaborazione dai thread worker.

database

Cartella contenente i dati persistenti del server.

posts.txt è la lista di tutti i post creati, **users.txt** la lista di tutti gli utenti registrati, **password.txt** contiene le credenziali di accesso degli utenti, **progressiveID.txt** memorizza l'id del prossimo post da creare in modo da garantirne l'unicità.
lastRewarding.txt contiene i post aggiornati all'ultimo calcolo delle ricompense.

exceptions

Contiene le eccezioni personalizzate

rmi

Contiene le interfacce e le relative implementazioni dei servizi RMI, ovvero il servizio di Registrazione e di Notifica Followers.

runnables

Contiene tutte le classi che implementano l'interfaccia java "**Runnable**", sono i task da passare ai thread. In particolare, **RequestHandler.java** contiene **tutta la logica** della gestione di una richiesta da parte di un client.
clientMulticastRunnable.java è un task utilizzato dal Client, che attiva un nuovo thread per ricevere le notifiche UDP per il calcolo delle ricompense.
estimateRewardRunnable.java è il task per il thread del Server dedicato al calcolo periodico delle ricompense.
updateDatabaseRunnable.java è il task per il thread del Server dedicato all'aggiornamento periodico dei dati sui file nella cartella database

I file **configClient.txt** e **configServer.txt** sono i file di configurazione cercati rispettivamente dal client e dal server al momento dell'esecuzione, devono trovarsi nella stessa cartella degli eseguibili.

ClientMain.java e **ServerMain.java** implementano il client e il server.

2. Architettura generale

Breve descrizione delle parti principali del progetto con riferimenti alla concorrenza e alle scelte progettuali più rilevanti. Per approfondire la logica delle singole funzionalità consultare il codice sorgente opportunamente commentato.

2.1 Client

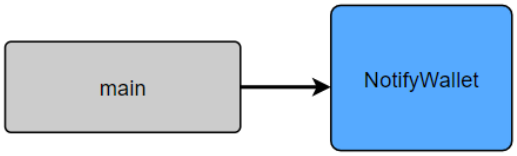


Figura 1 - Schema dei threads del client

Il client ha solo due threads: il main che aspetta che venga digitato un comando, invia i dati al server e stampa le risposte e il thread che si mette in ascolto sul multicast per ricevere la notifica di avvenuto calcolo delle ricompense.

Utilizza **Java NIO bloccante**, perciò prima di una nuova richiesta deve necessariamente aspettare di aver ricevuto risposta alla precedente. La scelta di NIO è stata fatta per mantenere simmetrica la comunicazione tra client e server.

Il client salva in locale la lista dei followers e i post del proprio blog, che si suppone essere in numero ridotto rispetto al feed, che invece non memorizza. Quando viene digitato il comando **`blog`** ci si aspetta che successivamente si riceva **`show post <idPost>`** quindi si controlla se il post è presente in cache e lo si stampa, altrimenti si manda la richiesta al server, il post potrebbe non essere aggiornato all'ultima interazione della commmunity.

Il client termina se viene digitato "quit"

2.2 Server

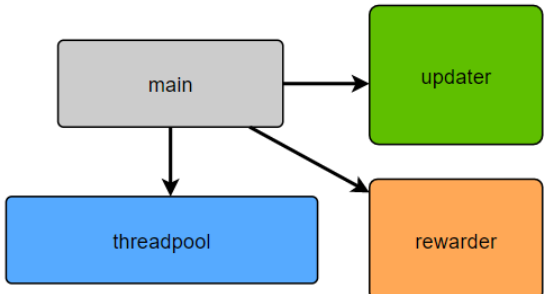


Figura 2 - Schema dei threads del server

Il server utilizza **Java NIO** non bloccante con **threadpool**, quindi all'arrivo di una nuova richiesta questa viene affidata ad un thread del threadpool che la gestirà autonomamente, mentre il thread main continuerà a cercare nuove richieste in arrivo tramite **selector**.

Nel caso in cui un client dovesse rispondere più volte al server (ad esempio rispondere di aver ricevuto una data stringa per poter ricevere la successiva) il selector lo rileverebbe e lo assegnerebbe ad un nuovo thread, per evitare questo si utilizza la struttura dati condivisa **processingClients** in cui vengono aggiunti tutti i canali dei client le cui richieste sono già in elaborazione, alla fine della richiesta il client viene rimosso e può essere nuovamente analizzato nel selector.

Il thread main si occupa anche di attivare il thread **updater**, che periodicamente aggiorna i dati su disco tramite Gson, e il thread **rewarder** che periodicamente calcola le ricompense.

Vengono poi inizializzati e attivati i servizi di registrazione e di notifica followers.

Il Server termina solo in caso di timeout

2.2.1 Gestione della concorrenza

Tutti i thread condividono l'oggetto WINSOME, perciò tutti i metodi che modificano la struttura del social sono **synchronized,** così che l'accesso all'oggetto sia regolamentato e non permetta modifiche in contemporanea.

Allo stesso modo anche i metodi delle altre classi che modificano gli oggetti sono **synchronized**, in questo modo, ad esempio, se due client stanno interagendo con lo stesso post le loro modifiche non vadano in conflitto.

Anche l'oggetto processingClients fa uso di metodi **synchronized.**

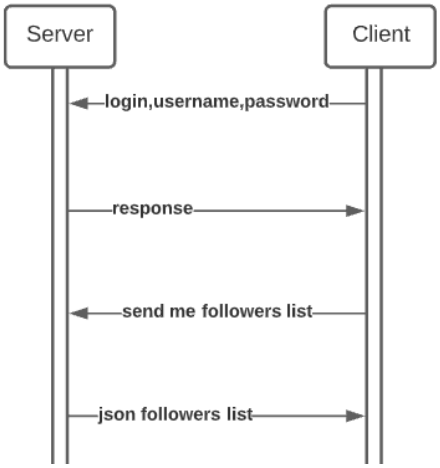
2.2.2 Task del threadpool (RequestHandler)

RequestHandler è la classe contenente quasi tutta la logica del social, rappresenta il task affidato ad un thread del threadpool.

Riceve la richiesta del client, la analizza dividendola sui separatori (le virgole) ed esegue la funzione associata al comando ricevuto, inviando al client i dati richiesti, poi rimuove il client dai **processingClients** e termina.
In caso di richieste particolari, come "post" o "comment" l'analisi del contenuto differisce in quanto i post o i commenti possono contenere qualsiasi tipo di carattere (anche le virgole), per dare completa libertà all'utente non è stato imposto alcun vincolo sui caratteri da usare nei post o nei commenti.

Di seguito, in breve, una visione dei protocolli di scambi di messaggi tra server e client per ogni tipo di richiesta con una breve descrizione per quelli più rilevanti:

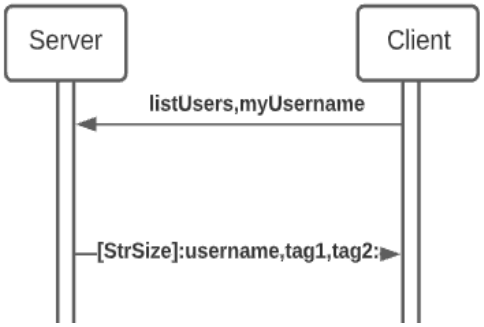
login



se “response” è il messaggio di avvenuto login allora il client chiede anche la lista dei followers per aggiornare quella in locale

list users

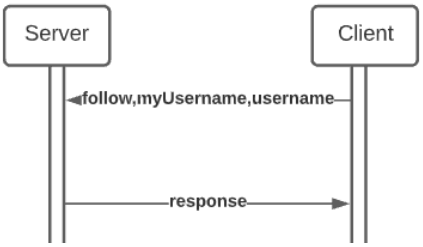
Viene inviata la lista degli utenti sottoforma di stringa, le virgole separano i tag e i due punti “:” separano gli utenti tra loro **“`:username,tag1,tag2,tag3,tag4,tag5:username2,tag1,tag2:`”**
Il server invia anche la dimensione della stringa, il client alloca il buffer adatto e riceve la lista da analizzare.



listFollowing

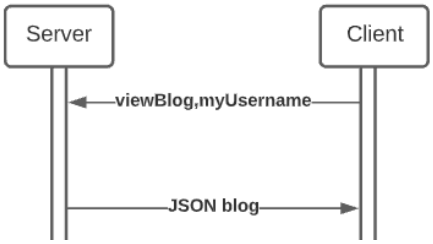
Il server invia la dimensione della stringa che contiene gli username degli utenti seguiti separati da virgole. Viene fatto un unico invio perché ci si aspetta che la dimensione sia limitata

followUser (unfollowUser)



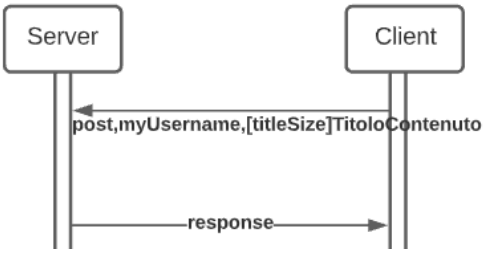
Il client invia la richiesta follow (unfollow) insieme al proprio username e all'username dell'utente da seguire. Il server si occupa anche di inviare la notifica all'altro utente.

viewBlog



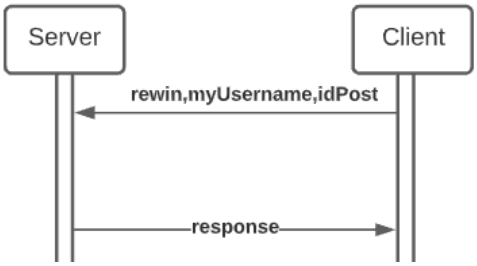
Il Server invia il blog sottoforma di lista di post in formato JSON, si fa un singolo invio perché si suppone che il numero di post nel blog non sia eccessivamente grande.

createPost

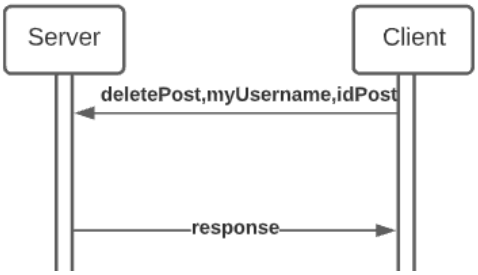


Il Client invia un buffer contenente gli elementi separati da virgole tranne per Titolo e Contenuto perché potrebbero contenere a loro volta virgole e vanificare la decifrazione della richiesta. Quindi il client inserisce prima la lunghezza del titolo, così che il server possa sapere quanti byte leggere per ottenerlo, i restanti byte corrisponderanno al contenuto.

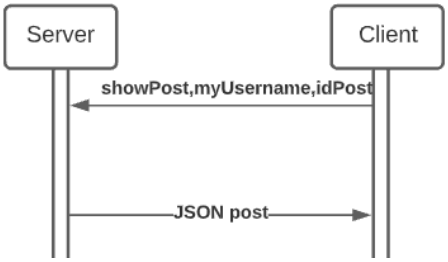
rewinPost



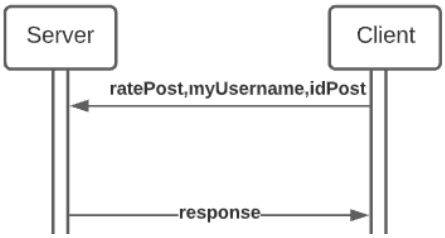
deletePost



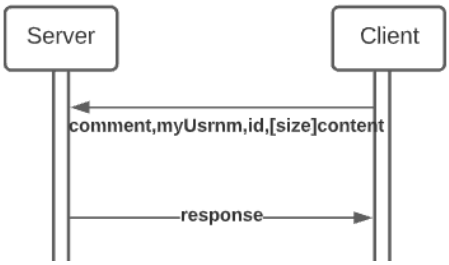
showPost



ratePost

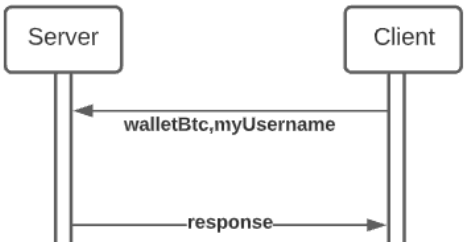


commentPost



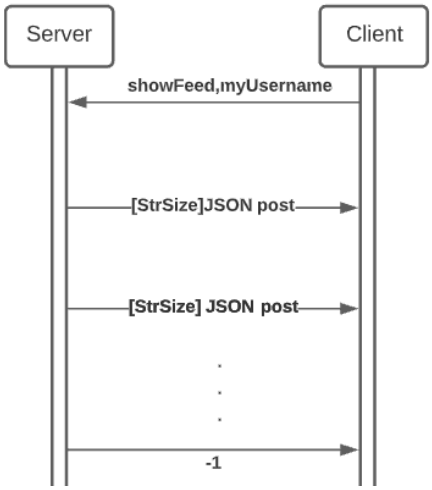
Come nel caso di **createPost** dato che il contenuto di un commento può contenere virgole allora viene inviata anche le dimensione.

walletBtc



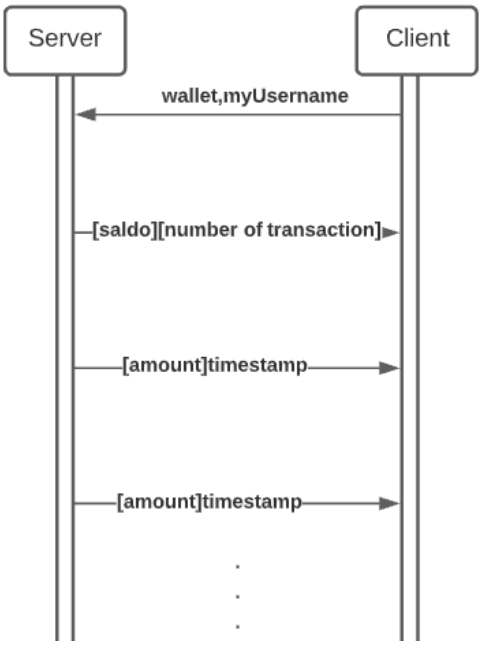
Il Server fa la richiesta a RANDOM.org e invia il valore aggiornato al client

showFeed



Il server carica più JSON post possibili nel buffer e poi li invia, svuota il buffer e lo "ricarica" con altri post, specificando anche la lunghezza di ognuno.

getWallet



Inizialmente il server invia il saldo e il numero di transazioni, così che il client sa quante volte deve leggere dal canale.

Inoltre la dimensione delle transazioni è fissata (8 byte per l'amount e 19 byte per il timestamp) e nota ad entrambi, così in modo che il client possa interpretare correttamente i dati ricevuti.

3. Istruzioni

NB: nella cartella del progetto è presente un file **`istruzioni.txt`** con le stesse indicazioni.

Posizionarsi da terminale nella cartella "src" e digitare:

```
javac -cp lib\gson-2.8.2.jar -d bin exceptions/*.java classes/*.java rmi/*.java
runnables/*.java *.java
```

Per avviare il Sever digitare:

```
java -cp lib\gson-2.8.2.jar;bin ServerMain.java
```

Per avviare il Client digitare:

```
java -cp lib\gson-2.8.2.jar;bin ClientMain.java
```

Non sono necessari argomenti da linea di comando, tutti i valori e i parametri si trovano all'interno dei due file **`configClient.txt`** e **`configServer.txt`** che devono trovarsi nella stessa cartella degli eseguibili. In mancanza di questi verranno utilizzati i valori di default.

Cancellare tutto il contenuto della cartella database prima di avviare il server se non si vogliono caricare i dati già in memoria.