RELAZIONE

RETI DI CALCOLATORI LABORATORIO
GIANLUCA PANZANI - 550358

INTRODUZIONE:

Il progetto che andró ad illustare é stato creato secondo le specifiche consegnateci alla fine delle lezioni. In quanto su tali specifiche non ho trovato restrizioni nella versione java da utilizzare ho preferito fare uso di una delle ultime, la JDK 17.0.1.

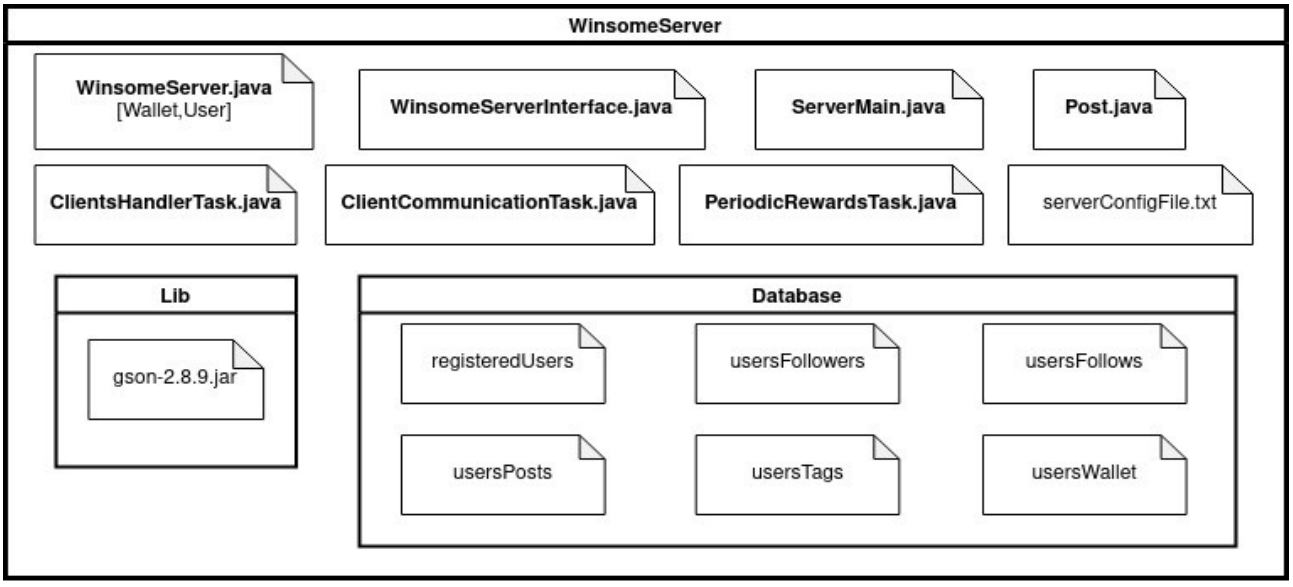
Vorrei inoltre specificare di aver preferito l'uso di alcuni metodi (circa un paio) anche se ritenuti dall'IDE "deprecated" perché visti a lezione (es: il metodo `leaveGroup`).

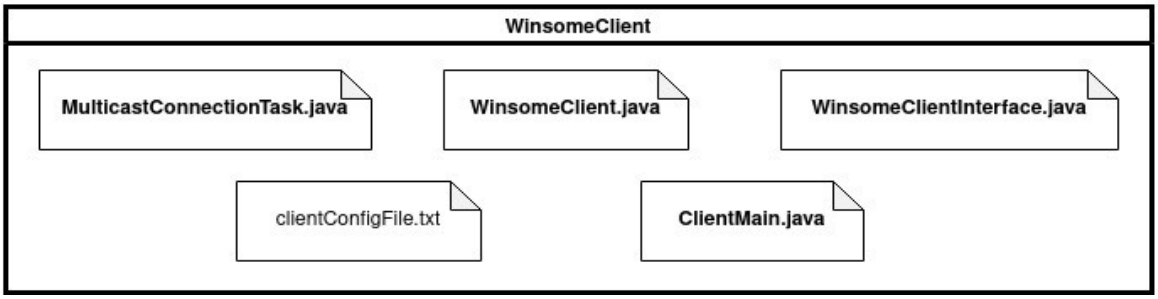
STRUTTURA DI FILE E DIRECTORIES:

Di seguito riporto uno schema abbastanza semplice ed intuitivo della struttura di file e directories.

Osservazione: bordi in grassetto indicano che l'oggetto in questione é una directory. I nomi dei file con estensione ".java" sono in grassetto.

Nell'immagine l'unica particolaritá si ha nel file **WinsomeServer.java** sotto al quale ho specificato le due classi private definite al suo interno.





DESCRIZIONE FILE:

Anche se dovrebbe essere abbastanza autoesplicativo grazie all'abbondante quantitá dei commenti, ho preferito dare comunque una spiegazione generale del ruolo e del comportamento dei file nel progetto.

I nomi dei file che finiscono per "Task" contengono classi che implementano *Runnable*, mentre i nomi dei file che finscono per "Main" contengono appunto il main.

FILE LATO SERVER (WinsomeServer/):

- **serverConfigFile.txt**: file di configurazione contenente le informazioni necessarie all'esecuzione del server (approfondiremo dopo).

- **ServerMain.java**: classe che si occupa della lettura del file di configurazione e della creazione dei thread, ognuno con il proprio scopo. Dopo aver fatto ció, attende l'istruzione `exit` da linea di comando per la terminazione dell'esecuzione del server. Dopo di che si occupa di far terminare tutti i thread correttamente.

OSS: uno di questi thread si occupa della memorizzazione periodica dei dati. Tale memorizzazione avviene sovrascrivendo ai file nella cartella database anche le informazioni che giá contenevano. Sono consapevole del fatto che sia una soluzione poco efficiente ma i problemi da risolvere in caso di "append" delle informazioni a fine file non erano pochi (es: il fatto che il formato json prevede la terminazione del file con una "]" e quindi avrei dovuto eliminare tale carattere prima di effettuare le append). Oltre al fatto che in caso di eliminazione di dati (es: eliminazione di un post), avrei dovuto rileggere interamente il file per ricercare tali dati ed eliminarli. Quindi alla fine la mia decisione è ricaduta sulla piú banale (seppur efficace).

- **WinsomeServerInterface.java**: interfaccia contenente i metodi invocabili da remoto tramite RMI e implementata dalla classe **WinsomeServer**.

- **WinsomeServer.java**: classe che implementa l'interfaccia **WinsomeClientInterface** e che offre tutte le funzionalitá per la gestione dell'account Winsome creato dagli utenti. In piú offre alcuni metodi protected in quanto riservati alle classi lato server nello stesso package. Al suo interno sono state definite due classi private: User, contenente username e password degli utenti, e Wallet, contenente le informazioni relative alle transazioni e al conto del portafogli dell'utente. Inoltre la classe WinsomeServer contiene tutte le strutture dati condivise necessarie alla memorizzazione di utenti registrati, post degli utenti, portafogli degli utenti, ecc… (tutte le istrutture dati sono commentate ad inizio file).

- **Post.java**: classe contenente le informazioni di ogni post e l'eventuale post di rewin (se assente non é stato fatto il `rewin` di nessun post).

- **ClientsHandlerTask.java**: classe che si occupa della gestione dei clients che vogliono connettersi al server. Crea un CachedThreadPool e poi si mette in un ciclo di attese in cui, ad ognuno di questi, instaura una connessione TCP con un client. Ad ogni iterazione aggiunge un thread sul thread pool al quale assegna un task di tipo **ClientCommunicationTask**. Questo si occuperá dell'interazione con quel singolo client.

- **ClientCommunicationTask.java**: classe che permette l'interazione con un singolo client. Si occupa di: istanziare l'oggetto remoto, inviare al client le informazioni per reperire l'oggetto remoto, ricevere le informazioni dell'oggetto remoto del client, memorizzare l'oggetto remoto del client, ricevere messaggi dal client, inoltrare i messaggi ricevuti alla classe **WinsomeServer** (che si occuperá di scegliere ed inviare la relativa risposta) e chiudere la connessione col client al momento della ricezione della stringa `exit`.

- **PeriodicRewardsTask.java**: classe che si occupa del calcolo periodico delle ricompense. Contiene i metodi per aggiornare la struttura dati contenente i "post recenti", ovvero quelli con cui gli utenti hanno interagito (messo like/dislike o commentato) nell'ultima iterazione. I post recenti sono stati memorizzati come una copia dei post reali ma privi di tutte le informazioni precedenti a quelle memorizzate nell'ultima iterazione. Quindi conterranno le sole informazioni aggiunte nell'ultimo periodo. Ovviamente dovrá contenere anche un oggetto WinsomeServer per poter interagire con questo tramite i suoi metodi.

- **lib/gson-2.8.9.jar**: libreria con archivio gson.

- **Database/***: sono 6 file che vengono usati per memorizzare le strutture dati contenenti le informazioni di post, registrazioni, portafogli, followers, follows e tags. Quandi abbiamo 1 file diverso per ogni struttura dati (quest'ultime sono descritte nel file **WinsomeServer.java**).

Ulteriori dettagli sono specificati nel codice attraverso i commenti.

FILE LATO CLIENT (WinsomeClient/):

- **clientConfigFile.txt**: file di configurazione contenente le informazioni necessarie all'esecuzione del client (approfondiremo di seguito).

- **ClientMain.java**: classe che permette l'interazione col server. Si occupa di: leggere il file di configurazione, aprire il thread che permette di stabilire la connessione col gruppo multicast (cioé un thread creato su un'istanza della classe **MulticastConnectionTask**), ricevere le informazioni per reperire l'oggetto remoto del server, memorizzare l'oggetto remoto del server, istanziare il proprio oggetto remoto, inviare al server le informazioni per reperire tale oggetto remoto, effettuare registrazione o login al client, inviare messaggi al server inseriti dall'utente tramite una semplice command line interface, ricevere i messaggi di risposta del server, stampare i messaggi ricevuti, chiudere il thread relativo al gruppo multicast e chiudere la connessione al momento dell'inserimento della stringa `exit`.

- **WinsomeClientInterface.java**: interfaccia contenente i metodi invocabili da remoto tramite RMI e implementata dalla classe **WinsomeClient**.

- **WinsomeClient.java**: classe che implementa l'interfaccia **WinsomeClientInterface** e che si occupa di aggiornare la struttura dati locale contenente i followers dell'utente. Inoltre permette di settare le informazioni utili a stabilire la connessione col gruppo multicast e la chiusura di tale connessione.

- **MulticastConnectionTask.java**: classe che permette di connettersi al gruppo multicast e di rimanere in ascolto in attesa dei messaggi che il server invierá periodicamente. Il thread rimane in attesa finché non saranno settati indirizzo multicast e porta. Dopo di ché potrá connettersi al gruppo.

FILE DI CONFIGURAZIONE:

Il formato delle righe dei file di configurazione é lo stesso sia lato client che lato server.

Tale formato é “NOMEVARIABILE=VALORE”. Tutte le righe che non lo rispettano saranno ignorate dal rispettivo file main (**ServerMain.java** lato server e **ClientMain.java** lato client), il quale é l’addetto alla lettura e al parsing del contenuto del file di configurazione.

Se il Main file, dopo aver letto tutto il file di configurazione, non avrá informazioni sufficienti ad eseguire il programma l’esecuzione sará interrotta e restituirá un messaggio d’errore/di usage.

Se invece tutte le informazioni verranno reperite correttamente l’effettiva esecuzione potrá iniziare.

Tutte le variabili e gli oggetti (sia lato client che lato server) che necessitano di inizializzazione a partire dai dati letti dai file di configurazione, sono state dichiarati/e all’inizio delle MainClass. Sopra di essi ci sono i rispettivi commenti che descrivono ogni campo.

Come ultima nota vorrei specificare il fatto che i file di configurazione contengono volutamente righe con valori errati che il programma scarta ed ignora correttamente.

METODI DELLE CLASSI:

Il comportamento dei metodi é descritto nel codice tramite specifiche e commenti.

Per ogni metodo ho indicato:

@effects: per descrivere gli effetti, il comportamento e l’eventuale valore di ritorno del metodo.

@param PARAMETRO: per descrivere il parametro indicato con le relative precondizioni necessarie affinché il metodo abbia il comportamento desiderato.

@throws EXCEPTION: per indicare quale eccezione puó sollevare e in che casi.

Dove non compare uno di questi campi implica il fatto che non fosse necessario.

Ho adottato come stile implementativo la *defensive programming* e di conseguenza ci sono molti controlli sui parametri, anche dove a volte non sarebbe necessario. La trovo una buona pratica e abitudine soprattutto per un fatto di debugging.

Di seguito riporteró l’elenco dei metodi implementati nelle varie classi che meritano qualche osservazione in piú. Ovviamente non ripeteró ció che é giá descritto nelle specifiche, mi limiteró ad aggiungere quelle informazioni secondarie, ma non per importanza, che permettono di chiarire meglio il loro ruolo all’interno del progetto.

METODI LATO SERVER:

ServerMain:

```
public static boolean getExitValue();
```
viene utilizzato nei cicli while presenti nei metodi *run()* dei file *ClientsHandlerTask.java* e *PeriodicRewardsTask.java*. Il valore viene settato a true dal *ServerMain* quando il server deve essere chiuso causando la terminazione degli altri thread.

WinsomeServer:

```
public WinsomeServer();
```
alla prima esecuzione di questo costruttore verrá ripristinato lo stato del sistema tramite il recupero delle informazioni dai file (in formato json) presenti nella cartella *Database*. Nelle successive esecuzioni viene solo richiamato il costruttore della superclasse.
```
private synchronized boolean registeredContains(User u);
```
usato per ricercare l'utente tra gli utenti registrati. Se avessi usato il metodo *contains()* sarebbe andato a ricercare l'oggetto user con lo stesso riferimento, non effettuando un controllo di uguaglianza campo per campo.
```
private synchronized boolean isRegistered(String username);
```
usato soprattutto per rendere il codice piú chiaro e leggibile.
```
public void searchUser(String start);
```
serve per cercare un utente sul social. Si potrebbe paragonare questa funzionalitá alla barra di ricerca di facebook.
```
protected void setOutputWriter(PrintWriter out);
```
metodo usato solo lato server che permette di settare il *PrintWriter* da utilizzare per inviare i messaggi al client. Quindi ogni istanza di *WinsomeServer* avrá un proprio *PrintWriter* (non a caso si ha un'istanza per client) che verrá settato dalla classe *ClientCommunicationTask*.
```
protected synchronized static boolean updateMemory();
```
metodo chiamato periodicamente dal thread che si occupa dell'aggiornamento della memoria (definito nel file *ServerMain*).
```
public synchronized boolean register(String user, String passw, LinkedList<String>
tags) throws RemoteException;
```
metodo che memorizza *user*, *passw* e *tags*. Di quest'ultimo memorizza i tag presenti nella lista in minuscolo e in particolare memorizza un numero di tag compreso (e uguale) tra 0 e 5. Quindi é ammesso anche il caso in cui l'utente non voglia inserire alcun tag. Mentre se vengono inseriti piú di 5 tag quelli superflui vengono ignorati.
```
public void turnOnNotify(WinsomeClientInterface clientRemoteObj) throws
RemoteException;
```
metodo invocabile da remoto. Dal punto di vista utente serve per attivare le notifiche, mentre dal punto di vista implementativo permette di memorizzare (lato server) l'oggetto remoto del client passatogli come parametro in modo tale da poter invocare i metodi remoti lato client (registrazione alle callback).
```
public void turnOffNotify(WinsomeClientInterface clientRemoteObj) throws
RemoteException;
```
metodo invocabile da remoto che ha l'effetto contrario del metodo *turnOnNotify*.
```
public void rewinPost(String idPost);
```
é interpretato come la creazione di un nuovo post uguale al post il cui id é passato come parametro ma con l'aggiunta di *{autore_post}* prima del titolo. Questo *autore_post* deve essere necessariamente l'autore del post che si é ricondiviso. Quindi se facessimo il rewin del post B ottenendo il post A, quando il post B aveva giá fatto il rewin del post C, l'autore che andrá a finire tra {} nel post A sará quello del post C. Se qualcun altro facesse nuovamente il rewin del post A l'autore tra {} rimarrebbe comunque quello del post C.
Questa scelta é dovuta al fatto che mi sembrava giusto "pubblicizzare" l'autore il cui post era di particolare interesse.
```
public void rate(String idPost, String vote);
```
i voti sono interpretati come like in caso di *vote = "+1"* e dislike in caso di *vote = "-1"*.

```
protected void startNewIteration();
```
metodo che serve al thread del calcolo periodico delle ricompense per ripristinare il campo *lastIter* dei post a false in modo tale da permettere il corretto incremento del campo *n_iterations* alla successiva iterazione. Questo perché il numero di iterazioni viene incrementato solo alla “prima aggiunta” (nell’ultimo periodo) di un post tra i “post recenti”. Se ad esempio un utente commenta il post A e un altro utente vota il post A nello stesso periodo la chiamata del metodo *incrementIterationsCounter()*, che avviene due volte, causerebbe il doppio incremento (cosa scorretta). Con l’uso della variabile booleana *lastIter* questo é impossibile (guardare l’implementazione di tale metodo nella classe Post).

Post.java:

```
protected static void setNextId(int id);
```
metodo che viene invocato soltanto al momento della prima chiamata del costruttore di *WinsomeServer*. Questo per il semplice fatto che l’id dei post é incrementale e di default ad ogni esecuzione il primo post creato avrá come id 10001. Siccome nel metodo *WinsomeServer()* ripristiniamo i post che erano giá presenti sul social, e siccome l’id deve essere univoco, l’id non dovrá partire da 10001 in quanto identificatore giá utilizzato precedentemente. Questo serve quindi a settare il prossimo id al valore del massimo id presente tra i post recuperati + 1. In questo modo ci accertiamo che l’id rimanga univoco.

PeriodicRewardsTask.java:

```
protected PeriodicRewardsTask();
```
costruttore chiamato solo dalla classe *WinsomeServer* affinché non debba necessariamente inizializzare i campi privati. Infatti l’istanza che crea serve solo ad accedere ai metodi forniti dalla classe *PeriodicRewardsTask.*
```
protected synchronized void deletePostFromRecentPosts(Post p);
```
in caso di eliminazione del post sul social non deve essere considerato il guadagno delle ricompense per quel post. Quindi l’eliminazione deve avvenire anche tra i post recenti (se presente).

METODI LATO CLIENT:

ClientMain:

```
private void parseMessage(String message);
```
i messaggi inviati dal server al client possono essere molto articolati. In tal caso vengono sostituiti i caratteri “\n” con i caratteri “/” in modo tale da permettere l’invio dell’intero messaggio su un’unica riga. In questo modo si riduce di gran lunga la complessitá dello scambio dei messaggi. Un unico messaggio di richiesta del client causa un unico messaggio di risposta del server.
Questo per dire che il client necessita di un metodo per il parsing dei messaggi del server in quando hanno un formato molto piú complesso dei messaggi del client.
```
private boolean registration(StringTokenizer tok, WinsomeServerInterface wsi,
BufferedReader reader);
```
metodo che aiuta il client durante la fase di registrazione

WinsomeClient:

```
public void setMulticastInfo(String IP, int PORT) throws RemoteException;
```
metodo remoto che permette al server (tramite l’oggetto remoto del client) di inviare le informazioni necessarie per permettere al client di unirsi al gruppo multicast.

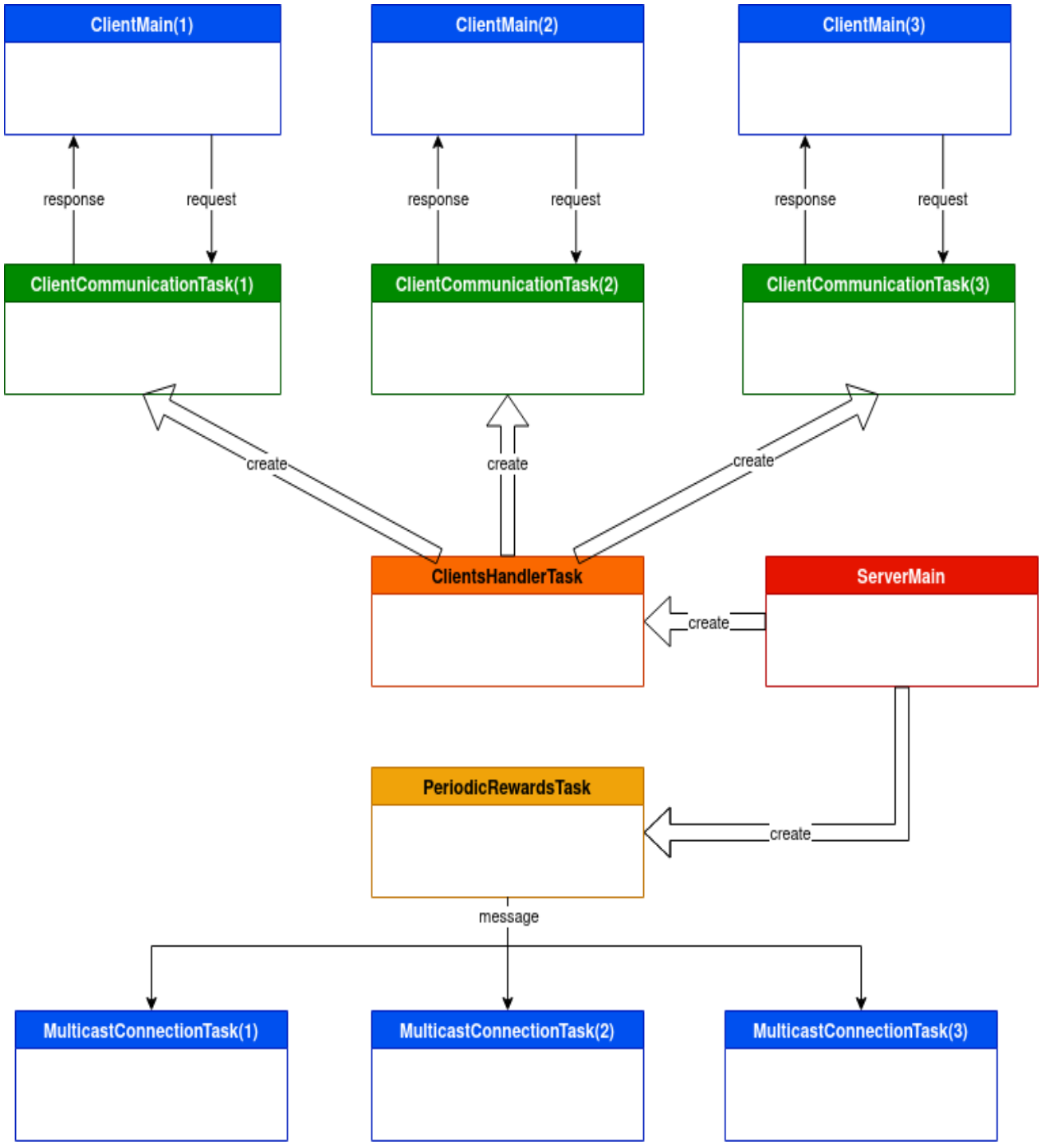
MulticastConnectionTask:

```
public void closeMulticast();
```
metodo che consente l’uscita del client dal gruppo multicast. Se non fosse chiamato dall’esterno del metodo *run()* non si chiuderebbe in quanto rimarrebbe bloccato sul metodo *recieve()*. Permette quindi la terminazione dell’esecuzione del client.

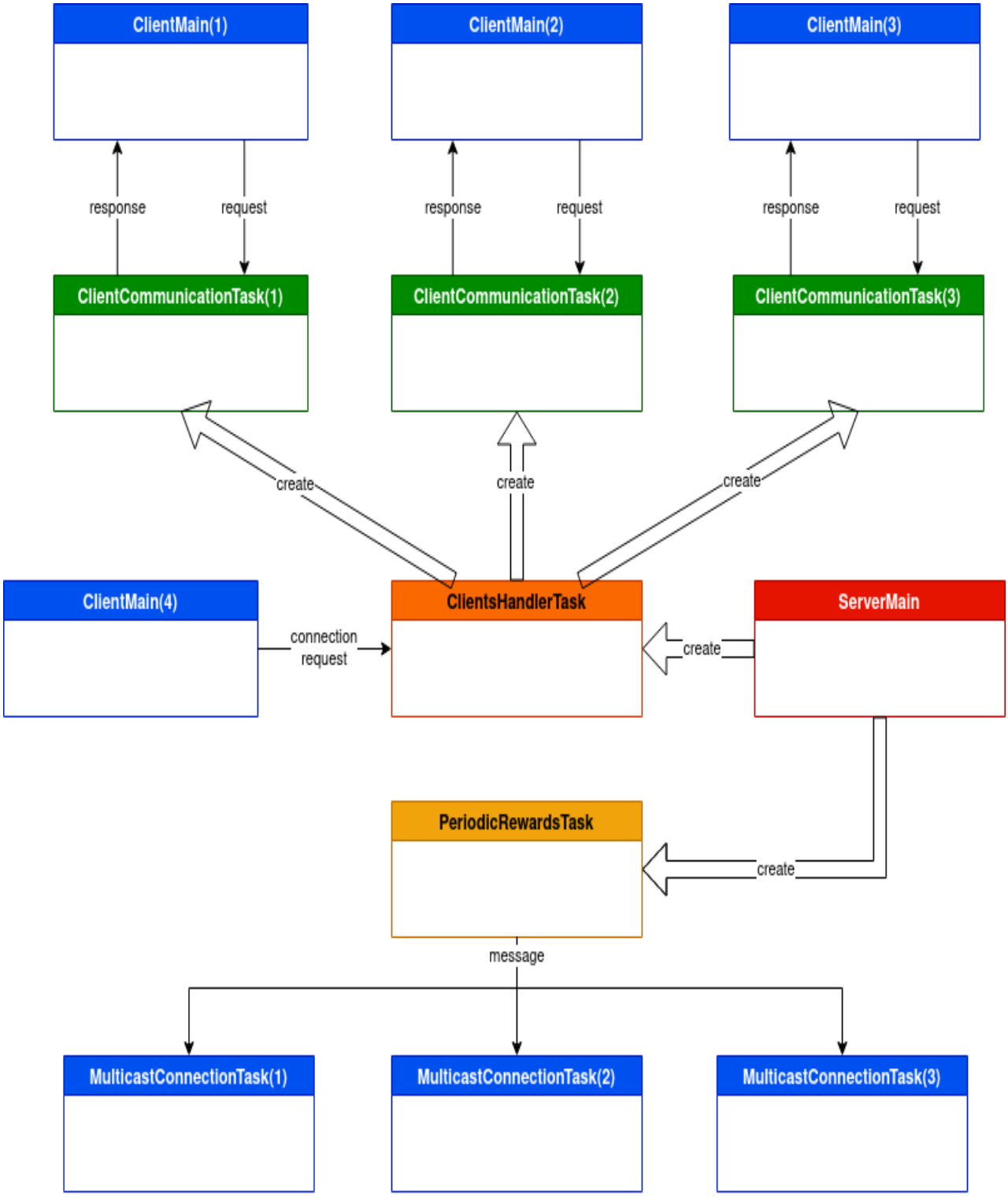
INTERAZIONE THREAD & CONCORRENZA:

Il sottostante diagramma contiene in blu i thread lato client e negli altri colori i thread lato server (un colore per ogni diverso ruolo). Ho scelto come caso quello di un'esecuzione con 3 clients che vengono gestiti quindi con 3 thread *ClientCommunicationTask* diversi.

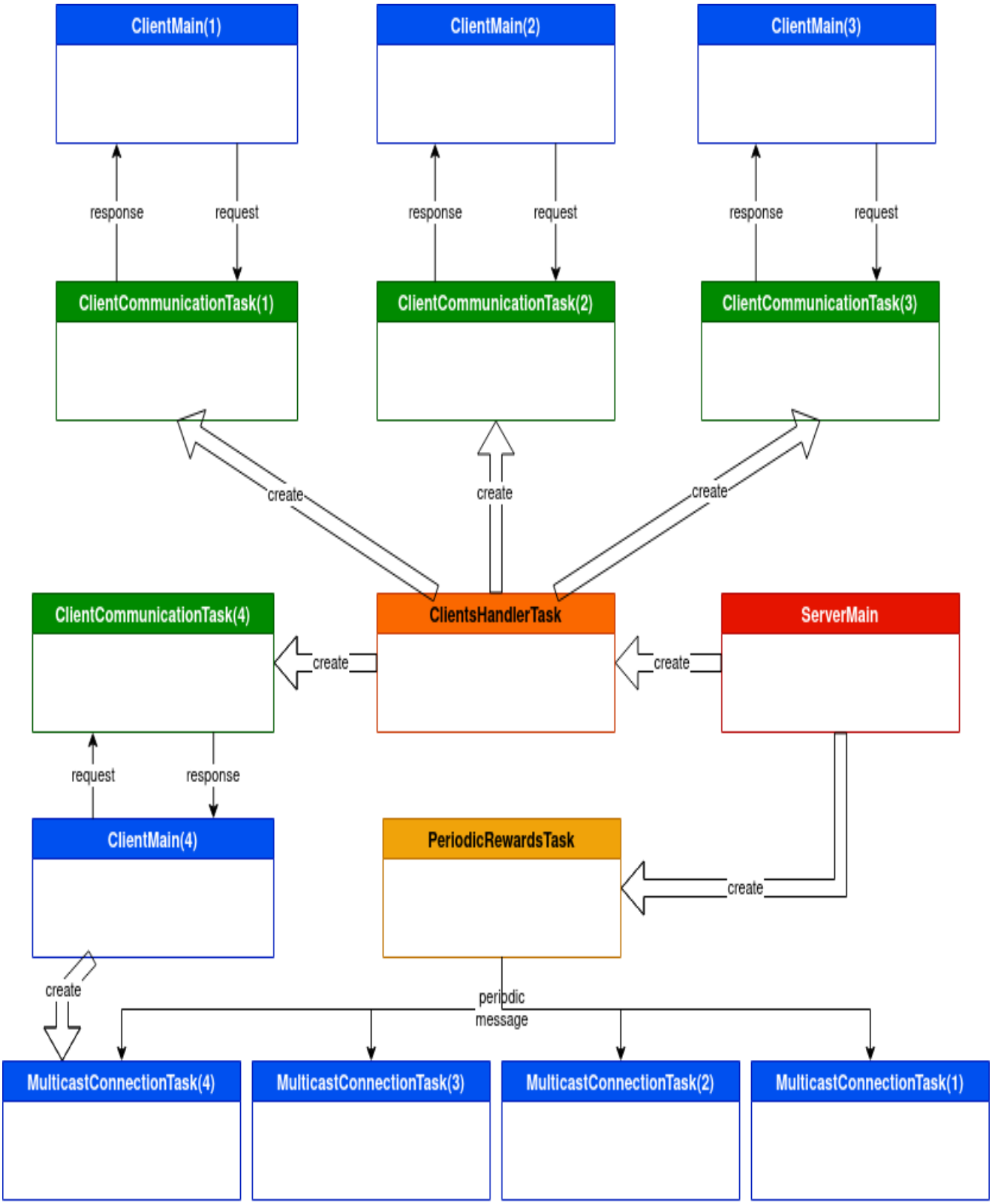
Per semplicitá ho evitato di inserire anche il thread che periodicamente aggiorna i dati in memoria.



Nell’immagine successiva un altro client richiede la connessione al server.



Nella seguente immagine il server ha accettato la richiesta di connessione del client ed ha conseguentemente creato un nuovo thread *ClientCommunicationTask*. Il client si é connesso ed ha quindi creato il thread *MulticastConnectionTask*.



COMPILAZIONE:

Considero la seguente compilazione effettuata a partire dalla cartella nella quale sono contenute le directories *WinsomeClient* e *WinsomeServer*.

Compilazione lato server:

```
javac -cp .:./WinsomeServer/lib/gson-2.8.9.jar WinsomeServer/*.java
```

Compilazione lato client:

```
javac WinsomeClient/*.java
```

ESECUZIONE:

Considero i seguenti comandi per l'esecuzione come eseguiti a partire dalla cartella nella quale sono contenute le directories *WinsomeClient* e *WinsomeServer*.

Esecuzione lato server:

```
java -cp .:./WinsomeServer/lib/gson-2.8.9.jar WinsomeServer.ServerMain
WinsomeServer/serverConfigFile.txt
```

oppure

```
java -jar jarServer.jar WinsomeServer/serverConfigurationFile.txt
```

Esecuzione lato client:

```
java WinsomeClient.ClientMain WinsomeClient/clientConfigFile.txt
```

oppure

```
java -jar jarClient.jar WinsomeClient/clientConfigurationFile.txt
```

COMPILAZIONE + ESECUZIONE:

Ho creato anche due semplici eseguibili che possono sostituire tutti i comandi precedenti e rendere l'esecuzione di client e server piú rapida (effettuano anche una `rm` di tutti i file.class creati dalla compilazione):

```
./execServer
```

```
./execClient
```

GUIDA ALL'UTILIZZO:

LATO SERVER:

In questo caso abbiamo poche cose da specificare.

Il server, durante la comunicazione coi client, resta in attesa di input da linea di comando. Qualunque comando venga inserito sará ignorato a meno che non si tratti del comando `exit` che ne causerá la terminazione.

Il resto viene eseguito automaticamente senza bisogno di alcuna interazione.

LATO CLIENT:

Per prima cosa ritengo importante specificare che ho cercato di creare un programma il piú semplice ed intuitivo possibile dal punto di vista dell'utente. Infatti questo si impegna a giudare l'utente dando suggerimenti nel caso in cui questo inserisca i comandi nel formato scorretto (o nel caso in cui non sappia quale comando utilizzare). Registrazione o login sono obbligatori a inizio esecuzione per poter accedere al proprio profilo Winsome.

Di seguito elencheró tutti i possibili comandi utilizzabili dall'utente accompagnati da una breve spiegazione:

`register <username> <password> <tag1> <tag2> <tag3> <tag4> <tag5>`

permette la creazione dell'account dell'utente. I tag sono opzionali ma non possono essere piú di 5.

`login <username> <password>`

permette il login da parte di un utente (se giá registrato e se non giá loggato su un altro dispositivo).

`help`

stampa tutti i possibili comandi utilizzabili dall'utente tranne il comando `register` in quanto utilizzabile solo all'avvio del programma al momento della richiesta di registrazione/login.

`search <startUsername>`

permette di ricercare e stampare tutti gli utenti registrati su Winsome che iniziano con la stringa di testo `startUsername`. Se viene inserito un username completo e viene stampato allora l'utente esiste. Cerca di replicare la barra di ricerca di facebook.

`tags <username>`

stampa tutti i tags dell'utente `username` se l'utente che ha usato il comando segue tale utente o se é il proprio nome utente.

`list users`

stampa tutti gli username degli utenti aventi almeno un tag in comune con chi ha usato il comando.

`notify on`

permette la registrazione alle callback che in questo caso, dal punto di vista utente, é una sorta di attivazione delle notifiche (da cui prende il nome).

`notify off`

permette la rimozione della registrazione alle callback che, dal punto di vista utente, come nel caso del comando precedente, é una sorta di disattivazione delle notifiche.

`logout`

permette di effettuare il logout permettendo di conseguenza il login da un altro dispositivo.

`list followers`

stampa tutti gli username dei propri followers (implementato lato client).

`list following`

stampa tutti gli username dei propri following.

`follow <username>`

permette di aggiungere ai following l'utente `username` e di inviargli una notifica (se registrato alle callback).

`unfollow <username>`

permette di rimuovere dai following l'utente `username` e di inviargli una notifica (se registrato alle callback).

`blog`

stampa id, autore e titolo dei propri post (ovviamente l'autore sará sempre lo stesso).

`post <title> | <content>`

crea un post con titolo `title` e testo `content` che sará poi visualizzabile sul proprio blog.

`show post <idPost>`

stampa le informazioni relative al post con id `idPost` se il post é sul proprio blog o se é si tratta del post di uno dei propri following. Le informazioni stampate sono: data e ora di creazione, id, autore, titolo, testo, likes (che corrispondono ai voti +1), dislikes (che corrispondono ai voti -1) e commenti.

`show feed`

stampa id, autore e titolo dei post condivisi dagli utenti seguiti.

`delete <idPost>`

permette di rimuovere il post con id `idPost` da Winsome.

`rewin <idPost>`

permette di ricondividere il post con id `idPost` se presente tra i post nel proprio feed.

`rate <idPost> <vote>`

permette di votare il post con id `idPost` del voto `vote` (che puó essere +1 o -1), se presente tra i post del proprio feed.

`comment <idPost> <comment>`

permette di commentare il post con id `idPost` con la stringa di testo `comment`.

`wallet`

permette di visualizzare il contenuto del proprio portafogli, cioé la history di tutte le transazioni e il conto totale, in wincoin (indicati per semplicitá con $).

`wallet btc`

permette di visualizzare il contenuto del proprio portafogli, cioé la history di tutte le transazioni e il conto totale, convertito in bitcoin (indicati con BTC).

OSS: l'interazione client server avviene tramite messaggi stampati a video dal client preceduti da <<<, in caso di messaggio di risposta del server, o da >>>, in caso di messaggio di richiesta da parte del client.

Esempio:

```
>>> login gianluca abcdef
<<< Login confirmed
>>> post piacere, gianluca | primo post su Winsome!! Ciao a tutti
<<< The post [10017] is now visible on Winsome
>>> show post 10017
<<< ....
```

CONCLUSIONI:

Spero di essere stato chiaro nelle spiegazioni.

Se alcune cose non lo sono abbastanza, consiglio di prendere visione del codice e delle strutture dati in quanto i relativi commenti dovrebbero essere abbastanza esaustivi.

Ovviamente il programma é pensato per un social network di piccole dimensioni e di conseguenza alcune scelte implementative possono essere rivedibili nel caso in cui tale app iniziasse ad avere un maggior numero di utenza.

Ad esempio l'uso di una Cached ThreadPool priva di limitazioni (in numero di thread) é una scelta che é stata presa pensando al fatto che non si avrá l'esigenza di creare 1000 e passa thread concorrenti.