WINSOME: a reWardINg SOcial Media

Progetto di Reti di Calcolatori

Leonardo Pantani

6 Gennaio 2022

1 Note

Il progetto è stato sviluppato utilizzando la Java 8 ed è disponibile su **GitHub** cliccando qui.
Per l'implementazione del salvataggio dei dati di persistenza (dati che servono per il mantenimento dello stato precedente del server) da file si è usata la libreria Gson, mentre si è usata Spring Security Crypto per implementare i meccanismi di *hashing con salt* e *verifica password*. Entrambe le librerie sono salvate nel percorso: **src/it/pantani/winsome/libs**.

2 Preparazione progetto

2.1 Esecuzione con file JAR già compilati (Ambienti WINDOWS & UNIX)

E' possibile eseguire il codice immediatamente estraendo i file contenuti nello zip in una nuova cartella.
Il client non ha dipendenze, mentre il Server ne ha due.
Nota: poiché il file JAR contiene nel proprio MANIFEST i percorsi *src/it/...* per le dipendenze, è necessario che la cartella *src* e il JAR siano allo stesso livello.

2.1.1 Comando per avviare il Server

```
java -jar ServerWinSome.jar
```

2.1.2 Comando per avviare il Client

```
java -jar ClientWinSome.jar
```

2.2 Compilazione e creazione manuali dei file JAR (Ambiente UNIX)

Salvare il contenuto del file zip in una nuova cartella ed entrare in quest'ultima. In seguito, eseguire questi comandi (se vengono copiati male usare il file "*Istruzioni per la compilazione*" fornito):

2.2.1 Client

```
mkdir buildClient
find src/it/pantani/winsome/client src/it/pantani/winsome/shared -name "*.
    java" > sourcesClient.txt
javac -d ./buildClient/ @sourcesClient.txt
cd buildClient
jar cfe ../ClientWinSome.jar it.pantani.winsome.client.ClientMain *
cd ..

# per eseguire
java -jar ClientWinSome.jar
```

2.2.2 Server

```
mkdir buildServer
find src/it/pantani/winsome/server src/it/pantani/winsome/shared -name "*.java" > sourcesServer.txt
javac -cp "src/it/pantani/winsome/libs/gson-2.8.9.jar:src/it/pantani/winsome/libs/spring-security-crypto-5.4.7.jar" -d ./buildServer/ @sourcesServer.txt
cd buildServer
jar mcfe ../src/it/pantani/winsome/shared/server-manifest/META-INF/manifest.txt ../ServerWinSome.jar it.pantani.winsome.server.ServerMain *
cd ..

# per eseguire
java -jar ServerWinSome.jar
```

3 Struttura generale del progetto

Il codice sorgente del progetto è stato suddiviso in vari *packages* per permettere una migliore comprensione della sua struttura.

- **libs** → contiene esclusivamente i file JAR delle *librerie* utilizzate dal server.
- **shared** → contiene le classi *condivise* di cui **Client** e **Server** hanno bisogno per funzionare. Contiene inoltre i file MANIFEST necessari per la creazione ed esecuzione corretta dei JAR.
- **client** → contiene le classi del **Client**.
  - **rmi** → questo sottopackage contiene solo la parte *implementativa* del servizio di aggiornamento della lista follower interna del client.
- **server** → contiene le classi del **Server**.
  - **entities** → contiene le classi che rappresentano le strutture dati fondamentali del social, come *WinSomeUser*, *WinSomePost* e diverse altre.
  - **exceptions** → contiene le classi che definiscono alcune eccezioni personalizzate. Si è preferito questo approccio invece di restituire errori tramite, per esempio interi, per aumentare la manutenibilità e comprensione del codice.
  - **rmi** → questo sottopackage contiene 2 servizi: la parte implementativa di *registrazione al servizio di callback* per la ricezione di aggiornamenti riguardo le liste follower dei client loggati e la parte implementativa della *registrazione di un utente al social WinSome* tramite la chiamata del metodo remoto **register**.
  - **utils** → contiene classi con responsabilità miste.

4 Definizione delle classi e scelte progettuali

4.1 Package Shared

4.1.1 Utils

Tutti i metodi e valori che non appartengono ad una specifica classe e possono essere utilizzati sia da **Client** che **Server** sono stati inseriti qui.
I metodi più importanti di questa classe sono *send* e *receive* che permettono al client e server di inviare e ricevere messaggi contenenti nuove righe o altri caratteri. Questi metodi sono stati creati per ovviare ad un problema che si verificava con l'iniziale modo di comunicare tra client e server, ovvero utilizzando i metodi **out.println** e **in.readLine** che però causavano problemi se le righe ricevute dalla

controparte erano più di una.
Sempre in questa classe, sono contenuti inoltre dei messaggi fissi che devono essere sempre uguali tra le due implementazioni sennò causerebbero problemi che impedirebbero il corretto funzionamento del sistema. Alcuni esempi sono il messaggio di LOGIN effettuato con successo o il messaggio di LOGOUT.
Infine, abbiamo la funzione *roundC* (dove "C" sta per "custom"), che approssima un numero all'intero più vicino con una quantità impostabile di cifre dopo la virgola. Da far notare che tutti i calcoli effettuati dal *RewardsManager* **non subiscono modifiche** e usano i double per avere una precisione più elevata; solo al momento della stampa, i numeri sono approssimati usando la funzione appena descritta.

4.1.2 ConfigManager

Si occupa di ottenere i valori impostati dall'utente sui file di preferenze tra client e server. Si è deciso di utilizzare la classe *java.util.Properties* perché possiede tutte le caratteristiche necessarie per leggere e scrivere ad alto livello su un file, inoltre i metodi forniti da questa classe sono facilmente comprensibili e utilizzabili. Quando viene istanziato un oggetto di questa classe, se il file di configurazione non viene trovato, sarà inizializzato con le opzioni di default del client o del server in base al valore (booleano) del parametro **isServer** fornito. Se il file non esiste, appunto, viene creato, inizializzato e messo in una cartella "data". Se il file esiste, viene caricato e da quel momento in poi, chiunque può accedere all'oggetto **config** per risalire ai valori delle preferenze del file. E' possibile anche salvare forzatamente delle preferenze personalizzate nel file in qualsiasi momento, grazie all'uso del metodo **forceSavePreference**. L'unica pecca del metodo scelto per salvare questi dati è il fatto che l'ordine delle preferenze non è mantenuto al salvataggio del file; questo sarebbe stato comodo per suddividere le varie preferenze in "categorie", di più facile comprensione all'utilizzatore.
Nota: alcune classi sfruttano i file di configurazione per immagazzinare dati persistenti invece di usare i file **.json**; si è fatta questa scelta perché in questo modo era più facile modificare le variabili in questione in fase di debugging.

4.2 Package Client

4.2.1 ClientMain

Come da specifica, il client è stato sviluppato con l'architettura "*thin (sottile)*"; questo significa che svolgerà il numero minimo di computazioni possibili e lascerà le restanti al server. L'unica struttura dati che il client terrà localmente è la **followersList** che viene aggiornata dal metodo **notificationEvent** interno alla classe *NotifyEvent* di cui parleremo in seguito. Per questa lista si è deciso di utilizzare una *ArrayList* perché sarà acceduta esclusivamente un thread alla volta e quindi non ha bisogno di controlli sulla concorrenza.
In breve, il client legge i parametri dal file di configurazione, li valida (nel caso uno non fosse presente, il client si arresta subito) e apre un *socket TCP* con il server all'indirizzo e porta specificati nel config. In seguito, si tenta di localizzare il registro all'indirizzo specificato; se questa operazione ha successo, si prepara i *reader e writer* per poi entrare in un loop dove si attende la richiesta dell'utente da inviare eventualmente al server.
Se la connessione col server viene persa, l'utente lo saprà quando proverà ad inviare una richiesta da far risolvere al server (non tutti i comandi inviano una richiesta remota!); in tal caso, il client chiederà all'utente se vorrà riconnettersi e se questa operazione ha successo e se il comando era un **register** oppure **login**, sarà ricordato e riproposto alla richiesta di input successiva per evitare all'utente il fastidio di riscriverlo.
Una breve nota prima di descrivere alcuni comandi della classe: nella specifica del progetto, il nome che definisce il comando è possibile che sia separato da uno spazio (come per esempio **list followers**); visto il modo con il quale il client divide il nome comando e i suoi argomenti, si è deciso che tutti i nomi saranno senza spazio, per capire meglio quali sono gli argomenti e quale è il comando. Comandi degni di nota di questa classe sono:

- **stopclient** → termina il client chiudendo immediatamente il socket col server, insieme ai due flussi di lettura e scrittura con quest'ultimo.

- **register** → sfrutta la tecnologia RMI per eseguire il metodo remoto **register** sul server; per questo motivo differisce un po' dal resto dei comandi che invece sono inviati al server tramite socket TCP. Il client verifica il numero di argomenti ed esegue il metodo.

- **login** → invia al server la richiesta di effettuare il login con i dati specificati come argomenti. Visto che il modo con il quale il server risponde influisce sul comportamento del client, si è deciso di verificare la risposta utilizzando delle costanti della classe *Utils* già descritte in precedenza. Se il client riceve **Utils.SOCIAL_LOGIN_SUCCESS**, allora considera l'operazione completata e si registra al callback del server per ricevere aggiornamenti sulla lista follower locale. Per terminare la procedura di accesso, il client esegue il metodo remoto **initializeFollowerList** per ottenere, solamente la prima volta, l'intera lista follower dell'utente passato come argomento (quello appena loggato). Notare come venga passata anche la password: se fosse stato richiesto solo l'username, *un client modificato avrebbe potuto ottenere l'intera lista follower di un utente che non è quello che si vuole loggare.*

- **logout** → manda una richiesta di disconnessione al server. Anche qui, visto che il comportamento del client varia in base alla risposta del server, si deve essere sicuri che questa sia sempre la stessa (e non vari a causa di aggiornamenti dei sistemi lato server). Si è deciso pertanto di usare la costante **Utils.SOCIAL_LOGOUT_SUCCESS**. Se l'operazione ha successo, il client si rimuove dalla lista callback del server e mostra un messaggio all'utente.

- **listfollowers** → unico metodo esclusivamente lato client. Stampa la lista di follower dell'utente attualmente collegato, che *viene aggiornata esclusivamente tramite tecnologia RMI.* Essendo un metodo locale, la gestione di questo comando contiene dei controlli non presenti in nessun altro comando implementato nel client.

4.2.2 WalletUpdateManager

Classe che si occupa di mostrare a schermo *una notifica di aggiornamenti dei portafogli degli utenti.* Come da specifica il server, dopo aver calcolato i premi relativi ai contributi dei post, deve inviare tramite un **Multicast Socket**, una notifica generica a tutti i client attualmente in ascolto. Quello di cui si occupa questo thread avviato nel main è appunto restare in ascolto di pacchetti in arrivo dal server. Visto che il messaggio da stampare a video ha *dimensione variabile* (in base al numero di cifre del premio o del nome della valuta modificabile nel file di config del server), inizialmente viene inviato un valore intero che rappresente la lunghezza dell'effettivo messaggio e poi il testo vero e proprio; si è preferito questo metodo invece di allocare un array di bytes statico per rendere più *dinamica e affidabile* la ricezione di messaggi di cui non si conosce a priori una lunghezza.
Questo thread viene terminato dal *ClientMain* tramite l'esecuzione del metodo pubblico **stopExecution**, che fa uscire dal loop il thread causando una *IOException* "controllata".

4.2.3 NotifyEvent (RMI)

Contiene esclusivamente il metodo *notificationEvent* che si occupa di aggiornare la **followersList** locale in base alla stringa "update" passata come argomento. Questa stringa ha una sintassi concordata con il server molto semplice: il primo carattere è il simbolo "+" o "-" e indica se riguarda un'operazione di aggiunta o rimozione dalla *lista follower locale*, mentre il testo successivo indica l'username dell'utente oggetto della modifica. Se viene ricevuta una stringa di "update" (aggiornamento) non valida, viene stampato un errore lato client, anche se ciò non dovrebbe mai verificarsi.

4.3 Package Server

4.3.1 ServerMain

Inizializza le strutture dati usate dalle altre classi che caratterizzano il **social WinSome**. Implementa il **Socket TCP Server** per accettare le connessioni, la **lista di Sockets connessi** e la **lista di sessioni attive**. All'avvio, vengono istanziati gli oggetti:

- **config** (ConfigManager) → genera o legge in base alla situazione il file di configurazione del server.
- **social** (SocialManager) → gestisce tutta la parte social del progetto.
- **rewards** (RewardsManager) → si occupa della parte di assegnazione dei premi in **WinCoins** agli utenti e di comunicare l'aggiornamento dei portafogli tramite notifica via **MulticastSocket**.
- **jsonmngr** (JsonManager) → genera o carica in base alla situazione i file contenenti i **dati di persistenza** del server (come gli utenti registrati, i post inviati, ecc).
- **psmngr** (PeriodicSaveManager) → come intuibile dal nome, effettua un salvataggio periodico dei dati di persistenza del server.

Finita la creazione delle istanze, vengono creati i registri della tecnologia RMI per: permettere al client di *registrare utenti* chiamando un metodo remoto e per permettere al client di *iscriversi ad una lista callback* per ricevere aggiornamenti sui propri follower.
Se durante l'istanziamento degli oggetti precedentemente elencati non si verifica nessuna *ConfigurationException*, il server apre il socket si prepara ad accettare connessioni.
Per gestire il client, si è deciso che dovrebbe essere generato un thread **gestore** che si occuperà di elaborare tutte le richieste di quel client dal momento in cui gli viene assegnato fino a quando quest'ultimo non si disconnette. Questo thread gestore si chiama *ConnectionHandler* e ne viene generato uno nuovo per ogni connessione accettata nel ServerMain. Per gestire meglio più thread in contemporanea, si è deciso di utilizzare una *pool* di tipo **cached** che è consigliata quando il numero di thread futuri non è conosciuto a priori e inoltre permette alla JVM di riutilizzare delle risorse quando ritenuto possibile.
Il ServerMain resta in attesa di nuove connessioni all'interno di un ciclo infinito, ma è quando questo terminerà? Solo quando si verifica una *SocketException*. Questa viene lanciata dal metodo **accept** (in cui il ServerMain resta per la maggior parte del tempo) solo quando l'amministratore del server invia da console il comando **stopserver**, che causerà la chiusura forzata del ServerSocket. Il server può elaborare richieste di input perché, oltre ai thread già accennati, ne viene avviato un altro (chiamato *InputHandler*) prima dell'inizio del ciclo. La classe di cui l'oggetto è istanza non verrà descritta in questa relazione perché prettamente usata a scopo di debugging e amministrativo, se non per il comando **stopserver** che invece è l'unico modo per terminare il server oltre al "*killing forzato*" del processo da parte del sistema operativo.
Quando il server esce dal ciclo, si occupa di chiudere i vari socket e terminare i thread ancora in esecuzione. Il salvataggio finale dei dati è eseguito sempre dal *PeriodicSaveManager* pertanto il ServerMain non si deve occupare di niente da questo punto di vista.

4.3.2 ConnectionHandler

Ogni oggetto istanza di questa classe gestisce, come dice il nome, una connessione tra un client e il server. Si è deciso di utilizzare questa implementazione a thread perché più comoda da usare e più "pulita" rispetto ad una possibile alternativa basata sull'utilizzo dei *selettori e canali*.
Il costruttore di questa classe riceve come parametri il **socket client-server** e un **codice di debug** (incrementale) per le stampe a schermo. All'*attivazione del thread*, questo si prepara a ricevere le richieste del client appena collegato e le gestisce di conseguenza. Nello switch che suddivide l'elaborazione di una richiesta rispetto ad un'altra, è contenuto codice di controllo della sintassi degli argomenti forniti. Se la richiesta passa questa verifica, viene chiamata la rispettiva funzione con lo stesso nome di quelle specificate nelle istruzioni del progetto. Le funzioni che eseguono metodi inerenti al **social**,

non fanno altro che chiamare un metodo della classe *SocialManager* e restituire un messaggio di conferma qualora non si verificassero eccezioni o, altrimenti, un messaggio di errore personalizzato. Per permettere al *ConnectionHandler* di riconoscere il motivo per cui una richiesta non è andata a buon fine, sono state create diverse **eccezioni** riutilizzabili in caso di errori comuni tra una richiesta e un'altra. Per esempio, la **UserNotFoundException** è lanciata da un metodo del *SocialManager* quando un'operazione su un utente è fallita perché non esiste nessun utente con username quello specificato come argomento.

4.3.3 SocialManager

Classe *cuore del progetto* che implementa tutti i metodi che salvano e manipolano informazioni relative al Social. Qui sono contenute le *HashMap Concorrenti* che salvano le associazioni tra nome utente $\leftrightarrow$ entità oppure nel caso dei post, tra id_post $\leftrightarrow$ oggetto post. Si è deciso di utilizzare questa struttura perché molto semplice da implementare e soddisfa tutti i requisiti per poter essere usata per salvare informazioni ad alta concorrenza di aggiornamenti.
Una cosa che può saltare all'occhio è la presenza di strutture molto simili: **followersList** e **followingList**; la prima associa un utente alla lista dei propri follower e la seconda associa un utente agli utenti che segue. Teoricamente si sarebbe potuto usare una sola struttura dati, ma si è fatta questa scelta per *velocizzare i tempi di accesso alle informazioni* di cui si ha bisogno e per *mantenere una ridondanza*: in questo modo se uno dei file si corrompe è possibile ricostruire l'altro senza problemi.
Per ricordarsi dell'id dell'ultimo post si usa il tipo *AtomicInteger* per assicurarsi che non si verifichino incongruenze tra accessi contemporanei dei thread che potrebbero accedere a questo dato.
Per quanto riguarda invece i metodi implementati, quelli che vengono chiamati nella classe *ConnectionHandler* utilizzano il sistema delle **Eccezioni** per far capire al chiamante quale operazione è andata storta. I metodi che non lanciano eccezioni sono più generici e possono essere usati in vari modi, alcune volte sono chiamati dalla stessa classe *SocialManager* per comodità, mentre altri ancora sembrano, all'apparenza, molto simili e duplicati di altri metodi ma in realtà sono destinati al lato amministrativo (*InputHandler* nella maggio parte dei casi), quindi effettuano meno controlli sui parametri per dare più libertà al gestore del server (per esempio permettendogli di aggiungere come follower utenti non validi).
Ultimo metodo degno di nota è *getFormattedCurrency*; questo metodo, dato un input double, lo formatta a **currency_decimal_places** cifre decimali con arrotondamento all'intero più vicino, ed è l'unico metodo che utilizza l'arrotondamento perché tutti i calcoli relativi ai premi e alle transazioni sono effettuati su double senza approssimazioni. Per questo motivo, se si specifica nel file di configurazione un numero di cifre decimali troppo basso è possibile che nell'elenco delle transazioni si veda qualcosa del tipo `+0,00 wincoins`; in tal caso basta aumentare il numero di cifre dopo la virgola. Si è preferito questo approccio invece di approssimare per eccesso per evitare fraintendimenti e pensare di avere più wincoins di quanti effettivamente se ne possiede. *Ultimo appunto*: questo metodo è chiamato ogni volta che si deve stampare a video un numero di WinCoins, dal **bilancio dell'utente** al **valore di una transazione**.

4.3.4 RewardsManager

Gestisce tutta la parte dell'assegnazione dei premi agli utenti che hanno interagito con un post. Quando questo thread viene avviato, apre un *DatagramSocket* ed inizia ad elaborare il premio in WinCoin totale per ogni singolo post. Il metodo **calculateRewards** calcola la somma da assegnare a curatori e autore del post in base alla formula scritta nella specifica del progetto, quindi si appoggia su tutte le variabili richieste, come per esempio il numero di iterazioni, numero di commenti, voti e altre ancora. Una volta ottenuto il valore del premio "globale" per il singolo post, questo viene spartito tra autore e curatori. E' possibile specificare nel config la percentuale da assegnare ad entrambe le parti, in modo che si possa inserire valori la cui somma non è proprio 100% (magari per imporre una "tassa" speciale). Infine, dopo l'aggiornamento degli utenti interessati, viene inviata una notifica tramite messaggio **UDP** alle coordinate (indirizzo ip e porta) specificate nel file di configurazione.

Nota: l'intero processo di verifica, calcolo e invio della notifica è effettuato con un *intervallo modificabile* a piacere.

4.3.5 PeriodicSaveManager

Si occupa del salvataggio periodico dei dati di persistenza. Il costruttore riceve molte strutture, ma è necessario se si vuole che tutta la parte del salvataggio venga gestita da un'unica entità, che se ci si pensa, è in realtà un *buon compromesso*. Il metodo **run** è molto semplice: contiene un ciclo da cui si può uscire nel caso di una eccezione, che si verifica se viene lanciata dall'esterno una *interrupt* che causa l'uscita dalla *sleep* in cui questo thread passa la maggior parte del tempo. Ogni **autosave_interval** millisecondi (specificabile nel file di configurazione del server), il thread salva tutti i dati delle varie strutture su disco. Per le informazioni "tradizionali" del social tutto è fatto in una singola istruzione: la **saveAll** della classe **JsonManager**; tuttavia due dati in particolare rispettivamente del *SocialManager* e del *RewardsManager* hanno bisogno di essere salvati nel config invece che nei file json, pertanto hanno dei metodi speciali che fanno uso del metodo **forceSavePreference** implementato nel *ConfigManager*.

4.3.6 JsonManager

Gestisce tutta la parte del salvataggio e caricamento dei file json contenenti i dati necessari per la ricostruzione dello stato precedente all'arresto del sistema. Questi file comprendono dati:

- utente
- portafogli
- followers
- posts

All'inizializzazione dell'oggetto di tipo **JsonManager**, questo genera i file (se non esistono) e poi col metodo pubblico **loadAll** preleva le informazioni necessarie dai file per il funzionamento del Social e le carica nelle strutture dati relative. Si è deciso di contenere le varie strutture in file diversi per evitare di avere un unico file di dimensioni considerevoli, sia per facilitare lo sviluppo che per ridurre la possibilità che esso *si corrompa*.
Notare come per leggere i file json, essendo possibilmente molto grandi, si utilizzi un *StringBuilder* in cui vengono salvati i caratteri del file interessato, man mano che vengono letti tramite il *ByteBuffer*.

4.3.7 PasswordManager

Sfrutta la libreria **org.springframework.security** per verificare che una password corrisponde e per generare *hash con salt* crittograficamente sicuri. Il metodo **hashPassword** non fa altro che chiamare il metodo **hashpw** della funzione BCrypt che genera un hash della password fornita in input con incorporato un salt generato casualmente. Anche **checkPSW** chiama il metodo **checkpw** per verificare se la password fornita è come quella memorizzata nel file di persistenza degli utenti. Notare che questo metodo **non legge** direttamente dal file la password, ma la riceve come argomento; in questo modo la classe può essere riutilizzata per progetti futuri e non necessariamente vincolata alla struttura di questo progetto.

4.3.8 RandomGenerator

Classe astratta che fornisce solo il metodo **generateRandomValue** che ottiene un numero pseudo-casuale contattando il servizio online *random.org*, come da specifica. Viene richiesto un numero da 20 cifre decimali, che poi saranno ridotte quando viene "parsato" come Double, ma questa operazione non ci dà problemi perchè non abbiamo bisogno di una precisione simile. Se si verifica un problema durante l'ottenimento del valore causale, il metodo stampa un errore e restituisce sempre 0.

4.3.9 WinSomeCallback (RMI)

Permette al client di invocare i metodi remoti **registerForCallback** e **unregisterForCallback**. Il primo aggiunge l'interfaccia del client e l'username dell'utente collegato ad una **HashMap** in modo che il server (tramite il metodo statico **notifyFollowerUpdate**), quando avviene una modifica alla lista follower dell'utente registrato al callback, possa chiamare il metodo remoto del client **NotificationEvent** che modifica la lista in base alla situazione (vedere la sezione **NotifyEvent** sopra). Il secondo metodo non fa altro che rimuovere l'utente dalla lista di callback in modo che non riceva più notifiche riguardo l'aggiornamento della lista follower. Questo metodo è richiamato dal client quando viene eseguito il comando *logout* o quando il collegamento col client termina.

4.3.10 WinSomeService (RMI)

Contiene due metodi che possono essere eseguiti dal client tramite tecnologia RMI: **register** e **initializeFollowerList**. Il primo permette di registrare l'utente al *Social WinSome* mentre il secondo è chiamato dal client appena effettuato il login per ottenere l'intera lista follower dal server. Se quest'ultimo metodo non fosse stato implementato, la lista follower locale del client conterrebbe solo gli utenti che hanno iniziato a seguire l'utente collegato solo dopo il login di esso.

4.3.11 Entità

- **WinSomeComment** → rappresenta un commento. Contiene il nome dell'autore, la data di invio e il contenuto del commento.
- **WinSomePost** → rappresenta un post. Contiene l'id del post (ridondanza perché già presente nella struttura della hashmap che contiene tutti i post), l'autore, il titolo, il contenuto, la data di invio, una *HashMap Concorrente* contenente l'associazione utente ↔ voto, una *ArrayList* contenente tutti i commenti relativi al post, una *ConcurrentLinkedQueue* contenente i nomi degli utenti che hanno fatto **rewin** di questo post e il *numero di iterazioni che l'algoritmo di calcolo premi ha già fatto su questo post.* Questa ultima informazione, se si voleva mantenere una perfetta modularità e indipendenza tra classi, non doveva esistere, ma l'idea di creare una struttura a parte per salvarsi solo questo parametro non è piaciuta e si è preferito giungere a questo compromesso. Una scelta implementativa riguardo a questa entità: si è deciso che un post resta *rewinnato* da un utente, anche se questo smette di seguire l'autore del post.
- **WinSomeSession** → permette di associare un socket TCP client con l'username dell'utente collegato con quel client. Oltre all'username dell'utente e al socket, contiene un timestamp che contiene la data di inizializzazione della sessione (scopo amministrativo).
- **WinSomeTransaction** → rappresenta una modifica al bilancio di un utente. Ogni transazione è caratterizzata dalla modifica in *WinCoins*, la causale e la data di esecuzione della transazione. Notare che una transazione può avere una "modifica" negativa; sarà la classe che implementa la transazione ad impedirlo con degli appositi controlli.
- **WinSomeUser** → rappresenta un utente del social. Contiene l'username, la password, la lista di tag e la data di creazione dell'account. La password contenuta non è ovviamente salvata in chiaro ma hashata. Per la lista di tags, si è deciso di usare un set perché non consente valori duplicati e ignora l'inserimento di un elemento che esiste già.
- **WinSomeVote** → rappresenta un voto su un post del social. Contiene l'username dell'autore, il valore del voto e la data di invio. Il valore del voto non è un boolean ma un **intero** perché sarà poi la classe che implementa l'oggetto ad imporre limitazioni.
- **WinSomeWallet** → rappresenta il portafoglio di un utente. Si è deciso di tenere questa struttura divisa dal *WinSomeUser* per mantenere una netta divisione delle responsabilità. Questa classe contiene il nome utente proprietario del portafoglio e una *ConcurrentLinkedQueue* di

transazioni. Si è deciso di optare per una struttura thread safe per evitare che, in future implementazioni, più thread ottengano dati inconsistenti tra di loro che causerebbero non pochi problemi.

5 Schema generale dei thread avviati

5.1 Client

Il client avvia 1 thread supplementare oltre al main thread stesso, cioè quello che gestisce le notifiche multicast ricevute in un particolare gruppo, le cui "coordinate" (indirizzo ip e porta) sono definite nel file di configurazione. Questo thread è avviato nel main del client e termina quando quest'ultimo riceve il comando **stopclient** dall'utente. Perché il thread esca dal loop infinito deve essere chiamato il metodo **stopExecution** dal main, che chiude forzatamente il socket multicast su cui il thread è in attesa di ricevere dati.

5.2 Server

Escludendo quelli generati per gestire le connessioni, vengono creati 3 thread supplementari oltre al main thread stesso, cioè:

- **RewardsManager** → avviato prima dell'apertura del socket TCP server nel main.
- **PeriodicSaveManager** → avviato anch'esso prima dell'apertura del socket TCP server nel main. Quando viene ricevuto il comando **stopserver**, il thread riceve una interrupt e viene eseguito il metodo **stopExecution**. Prima di uscire dal ciclo e non ricominciarlo più, viene effettuato un ultimo salvataggio dei dati di persistenza.
- **InputHandler** → avviato prima dell'apertura del socket TCP server nel main. Gestisce tutta la parte che permette all'amministratore del server di eseguire alcuni comandi utili per il debugging e la gestione del server attualmente aperto (come visualizzare la lista dei client connessi, aggiungere soldi e fare altre operazioni, non di importanza rilevante per il progetto).

Consideriamo ora invece i thread avviati per la gestione delle connessioni in ingresso. Ne viene avviato uno per connessione e viene inserito in una *CachedThreadPool*. Di ogni socket aperto (gestito da un thread) viene mantenuto un riferimento nel main nella **socketsList** per poterli chiudere in remoto alla ricezione del comando **stopserver**. Tutti i thread *ConnectionHandler* avviati durante la vita del server vengono tutti terminati tramite il comando **pool.shutdown**, anche se dovrebbero essere già terminati per conto proprio dopo la chiusura forzata dei sockets. Se questo comando non "basta" per terminare tutti i thread, si attende al massimo **max_timeout_pool_shutdown** millisecondi (specificabile nel config) prima di chiudere forzatamente il pool.