WINSOME: a Rewarding Social Media

Ahmad Shatti 578268

Settembre 2022

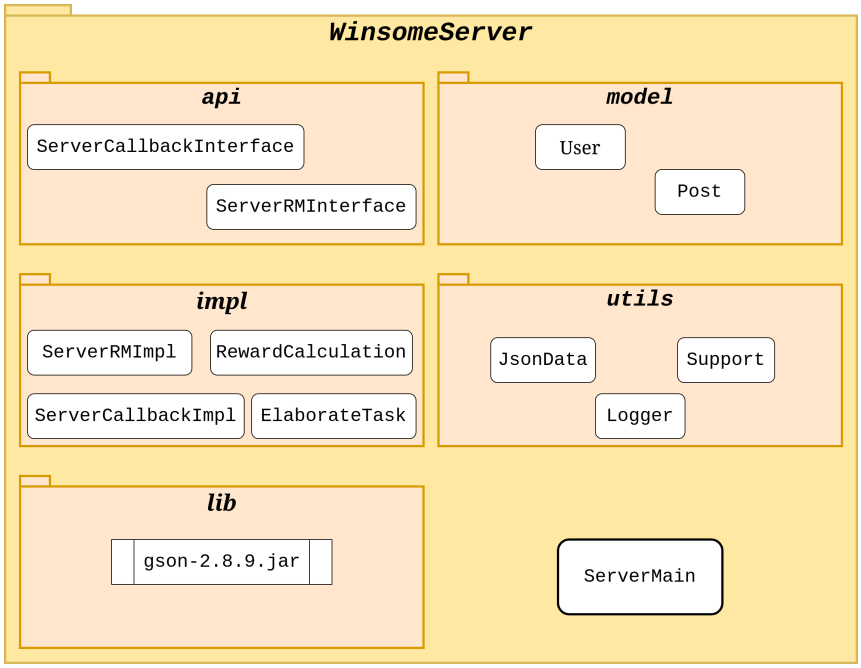
Contents

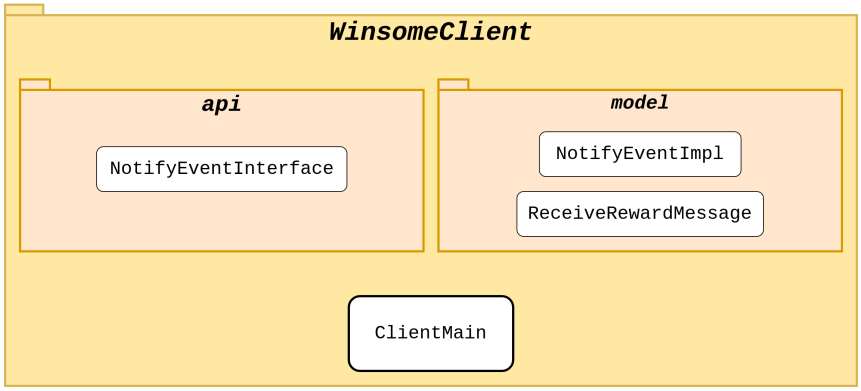
1 Introduzione e struttura del progetto

Il progetto è stato realizzato utilizzando Java 17 e la libreria *gson* per il salvataggio degli utenti e post di WINSOME. È composto da due package principali **WinsomeClient** e **WinsomeServer**, entrambi oltre a contenere la classe principale, rispettivamente **ClientMain** e **ServerMain**, sono suddivisi nel seguente modo:

- **api**, contiene le interfacce;
- **impl**, contiene le implementazioni delle interfacce;
- **model**, contiene la classe che modella un utente iscritto a WINSOME (**User**) e un un'altra classe che modella i post presenti all'interno del social network (**Post**);
- **utils**, contiene classi di supporto.

WinsomeClient contiene solo i primi due package. Client e Server comunicano principalmente tramite socket TCP e gli stream utilizzati sono *BufferedReader* per leggere stream di caratteri e *DataOutputStream* per scrivere tipi di dati primitivi in output. Le due classi all'avvio leggono il rispettivo file di configurazione contenente i parametri di input delle applicazioni (numeri di porta, indirizzi, timeout...). Il file di configurazione è composto da ⟨"chiave" = "valore"⟩ e se viene letto una chiave che la classe non "si aspetta" viene lanciata una *RuntimeException*. Il programma è stato testato su Ubuntu 22.04





2 WinsomeServer

Il Server è stato realizzato con JAVA I/O e threadpool. La classe principale **ServerMain**, oltre a definire le variabili per il file di configurazione e per gli oggetti remoti, crea due importanti variabili per il funzionamento del programma. La prima variabile globale *serverSocket* viene utilizzata per aspettare ed accettare le connessioni degli utenti; la seconda *serverActive* è una variabile atomica booleana che permetterà al Server Administrator di chiudere il server. Inoltre il ServerMain definisce ed inizializza tre strutture dati fondamentali: *usersMap*, hashmap contenente gli utenti iscritti a WINSOME, *postsMap*, hashmap contenente i post presenti in WINSOME e loggedUsers, lista degli utenti attualmente loggati. All'interno del main la classe legge il file di configurazione, effettua una chiamata al metodo statico *readData* all'interno di **JsonData** per ricostruire lo stato del sistema e crea e attiva due thread:

i) *SSthread*, all'interno di un ciclo, dopo aver instaurato una connessione TCP con il client, la socket viene assegnata ad un thread pool per gestire le richieste dell'utente. Quando il Server Administrator digita la stringa "exit", incomincia la terminazione graduale del pool e dopo "*TIMEOUT*" millisecondi la terminazione istantanea (nel file di configurazione è 20 secondi);

ii) *RCthread*, riceve come task la classe **RewardCalculation** che calcola periodicamente le ricompense d'autore e di curatore fino a quando non viene digitato "exit".

Il **ServerMain** mette a disposizione degli oggetti remoti per la registrazione e notifica, per essere invocati dai client. Quando verrà digitato "exit", nel caso ci siano degli utenti loggati, il server ne mostrerà il numero e il Server Administrator con "y" può decidere se chiudere effettivamente la serverSocket, il SSthread e RCthread o aspettare che i client abbiano terminato.

2.1 Classi di supporto

- *api package*

  ServerCallbackInterface, interfaccia remota che definisce i metodi per registrarsi e deregistrarsi al servizio di notifica;

  ServerRMInterface, interfaccia remota che definisce i metodi per registrare un utente a WINSOME e caricare i followers dell'utente al momento del login.

- *impl package*

  ServerCallbackImpl, implementa l'interfaccia *ServerCallbackInterface*. Contiene la struttura dati *clientsList* che comprende gli utenti che devono essere notificati al momento di un follow o unfollow. L'utente viene inserito nella struttura dati al momento del login, e rimosso al momento del logout;

  ServerRMImpl, implementa l'interfaccia *ServerRMInterface*. Contiene la struttura dati *usersMap* che comprende gli utenti registrati a WINSOME. Il metodo *registerUser* effettua un controllo sull'unicità dell'username e sulla lunghezza di username e password forniti dal client. Nel caso vadano bene, il metodo effettua l'hash della password grazie al metodo statico *hashPassword* fornito dalla classe **Support**, setta in minuscolo i tag e infine aggiunge il nuovo utente nella struttura dati. La classe implementa il metodo *loadFollower* che dato in input l'username restituisce la lista dei suoi followers;

  RewardCalculation, si occupa del calcolo periodico delle ricompense. Contiene le strutture dati *usersMap* per permettere l'aggiornamento del *wallet* degli utenti (nel caso in cui la ricompensa sia maggiore di 0) e *postsMap* per conoscere il numero di upvote, downvote e commenti in modo da poter calcolare la formula. Al termine del calcolo lo notifica tramite messaggio al gruppo di multicast con *DatagramSocket*. Il calcolo viene ripetuto ogni *REWARDTIME* millisecondi (nel file di configurazione è 1 minuto);

$$Guadagno = \frac{log(max(\sum_{p=0}^{new\_people\_likes}(Lp), 0) + 1) + log(\sum_{p=0}^{new\_people\_commenting}(\frac{2}{1+e^{-(Cp-1)}}) + 1)}{n\_iterazioni}$$

ElaborateTask, gestisce le richieste del client. Contiene le strutture dati: *usersMap* in modo da poter reperire le informazioni sugli utenti del social network, *postsMap* per poter aggiungere, votare, commentare, condividere e eliminare post e *loggedUsers* per conoscere gli utenti attualmente loggati. Sia U l'utente corrente, la classe contiene i metodi:

- *loginAccount*, controlla che l'username fornito da U esista, effettua l'hash della password per confrontarla e controlla che U non sia già *loggato* da un altro dispositivo;
- *logoutAccount*, rimuove U da *loggedUsers*, chiude la socket e gli stream;
- *followAccount*, se U vuole seguire un utente W, si effettuano i seguenti controlli: U e W non siano lo stesso utente, l'utente W esiste e che W non sia già presente nella lista di *following* di U. Alla fine si avvisa W che U ha incominciato a seguirlo tramite il metodo *updateNewFollower* presente nella classe **ServerCallbackImpl**;
- *unfollowAccount*, analogo di *followAccount*;
- *listUsers*, restituisce la lista degli utenti con un tag in comune con U. Per fare ciò utilizza un ciclo annidato che inserisce nella lista *usersWithCommonTags* gli utenti appena essi hanno un tag in comune con U;
- *listFollowing*, preso in input l'username di U restituisce la sua lista di following (struttura contenuta all'interno di **User**);
- *createPost*, permette di creare un post nel formato ⟨*"titolo"*⟩ ⟨*"contenuto"*⟩, come scelta di progettazione ho scelto che le virgolette siano obbligatorie in modo da poter separare le due parti. Il metodo controlla che siano presenti quattro virgolette, e poi separa la parte del titolo con la parte del contenuto. Quindi è possibile controllare che il titolo non sia superiore a 20 caratteri e il contenuto a 500 e inserire il post all'interno della struttura dati *postsMap* con l'idPost massimo +1. Queste operazioni (contare che ci siano quattro virgolette, separare la parte di titolo e contenuto e trovare l'id massimo) sono effettuate grazie a metodi statici presenti nella classe **Support**;
- *blogPost*, preso in input l'username di U restituisce in output una lista contente tutti i post creati da U. In risposta al client verrà inviato prima la dimensione della lista e se essa è maggiore di zero, verrà inviato anche il suo contenuto che comprende idPost, autore e titolo per ciascun post (si utilizza il metodo *showPost* contenuto all'interno di **Post**);
- *showFeed*, preso in input l'username di U restituisce in output la lista contenente tutti i post creati dai following di U. L'invio della risposta al client è analogo a *blogPost*;
- *showPost*, preso in input l'id di un post, controlla se corrisponde all'id di un post esistente e in quel caso utilizza il metodo *showInteraction()* presente nella classe **Post** per restituire titolo, contenuto, upvote, downvote e commenti del post;
- *deletePost*, preso in input l'id di un post, come prima, controlla se corrisponde all'id di un post esistente e se U è l'autore del post. Se queste condizioni sono verificate elimina il post;
- *rewinPost*, ripete gli stessi controlli effettuati in *deletePost*, e con l'aiuto di *Support.maxId* crea l'identificativo per quel post. Inoltre utilizza ancora **Support** per il metodo *addString* in modo da aggiungere al titolo del post la dicitura "*REWINED BY*";
- *ratePost*, prende come input l'username dell'utente U, l'id del post che U vuole votare e il voto che vuole dare. Il metodo controlla che il voto sia uguale a "+1" o "-1", altrimenti lancia un messaggio di errore. In seguito controlla che l'id fornito corrisponde ad un post esistente, che U non sia l'autore del post, che il post sia

di un utente presente nella lista di following di U e infine che non abbia già votato quel post. Se i controlli vanno a buon fine incrementa *upvote* o *downvote* in base se sia un voto positivo o negativo e aggiunge U a *whoRated*. Le variabili *upvote, downvote* e *whoRated* sono presenti all'interno di **Post**;

- *commentPost*, come con *createPost* ho voluto che un commento avesse obbligatoriamente le virgolette, quindi il metodo controlla che la stringa in input abbia due virgolette, in seguito, oltre ad effettuare gli stessi controlli presenti in *ratePost*, controlla che il commento non sia più lungo di 140 caratteri (anche se non era indicato nella specifica). Se tutto va a buon fine aggiunge un commento al post con la dicitura "⟨ *comment* ⟩ BY ";
- *getWallet* e *getTransaction*, entrambi prendono l'username dell'utente U e il primo restituisce il wallet, mentre il secondo la lista di transazioni che sono state create in **RewardCalculation**;
- *getWalletBTC*, tramite *URL* e *URLConnection* si collega al sito RANDOM.ORG per ottenere un valore decimale causale che verrà poi moltiplicato al wallet dell'utente per simulare il tasso di cambio;
- *sendOneReply*, prende in input una stringa e la invia tramite *DataOutputStream*.

- *model package*

  User, classe che modella un utente iscritto al social network WINSOME. Contiene metodi per reperire username, password, tag, lista following, lista followers, wallet e transazioni;

  Post, classe che modella un post di un utente. Contiene metodi per reperire informazioni sui post come titolo, autore, contenuto, upvote, downvote, commenti...

- *utils package*

  JsonData, classe che permette il salvataggio dello stato a ogni modifica del sistema serializzando e deserializzando gli user e i post utilizzano i json file *users.json* e *posts.json*. Questi file vengono letti all'avvio del server e scritti ad ogni cambiamento delle strutture dati usersMap e postsMap;

  Support, classe che come dice il nome è di supporto per il client e server. Contiene metodi per controllare se un utente può eseguire una determinata azione verificando se l'utente è loggato o se ha fornito gli argomenti giusti. Inoltre contiene metodi per: eliminare da una stringa parentesi quadre o virgolette; dividere una stringa composta da virgolette; aggiungere ad una stringa "REWINED BY" o "⟨ comment ⟩ BY"; contare quante virgolette una stringa possiede; fare l'hash di una password e restituire quale id deve avere il prossimo post che viene creato;

  Logger, classe che ha l'unica utilità di eliminare i troppi "System.out" nel progetto.

2.2 Strutture dati

- ⟨**String, User**⟩ **usersMap** , hashMap che contiene gli utenti iscritti a WINSOME. La chiave è l'username fornito dall'utente, che essendo unico, rende agevole reperire i dati del client contenuti all'interno della classe User. La classe **ServerMain** la inizializza, **JsonData** la popola ad un nuovo avvio del server, **ServerRMImpl** aggiunge un nuovo elemento alla mappa dopo aver controllato che l'utente non sia già all'interno, **RewardCalculation** la utilizza per permettere di modificare la variabile wallet contenuta in **User** e **ElaborateTask** per cercare un determinano utente e modificare le sue variabili followersList e followingsList;

- ⟨**String, Post**⟩ **postsMap**, hashMap che contiene i post presenti in WINSOME. La chiave è l'id del post, che essendo unico, permette di ottenere facilmente i dati di un post. In questo caso la chiave non poteva essere l'username dell'utente perche' altrimenti ogni client potrebbe creare un unico post. La classe **ServerMain** la inizializza, **JsonData** la popola ad un nuovo avvio del server, **RewardCalculation** la utilizza per reperire i upvote, downvote e commenti recenti per poter poi calcolare le ricompense e **ElaborateTask** aggiunge, rimuove e modifica post;

- ⟨**String**⟩ **loggedUsers**, lista che permette di controllare che un utente non sia collegato da due "dispositivi" diversi. **ServerMain** la inizializza e **ElaborateTask** la controlla, e inserisce un utente solo se non è già presente;
- ⟨ **NotifyEventInterface** ⟩ **clientList**, lista contenuta solo all'interno della classe **ServerCallBackImpl**. Un utente viene inserito nella struttura al momento del login, e rimosso al momento del logout. In seguito la lista viene utilizzata per notificare gli utenti al momento di un evento di follow o unfollow;
- ⟨ **String** ⟩ **followersList, followingList, transaction**, liste contenute all'interno di **User** che comprendono i followers, following dell'utente e tutte le transazioni che spiegano i movimenti del wallet;
- ⟨ **String** ⟩ **whoRated, comment, newPeopleUpvote, newPeopleDownvote, newPeopleComment**, liste contenute all'interno di **Post**, che contengono rispettivamente chi ha votato un determinato post, i commenti del post e gli upvote, downvote e commenti recenti. Le ultime tre vengono utilizzate nella classe **RewardCalculation** e vengono azzerate dopo ogni calcolo delle ricompense.

La maggior parte delle operazioni che il server svolge sulle strutture *usersMap, postsMap* e *loggedUsers* sono sincronizzate per evitare situazioni di errore ed inconsistenze.

3 WinsomeClient

La classe principale **ClientMain** permette l'interazione con il Server. La classe verifica se un utente è loggato tramite la variabile booleana *logged* che deve risultare false nel momento in cui un utente vuole registrarsi a WINSOME, fare il login nel proprio account o chiudere il client. Invece *logged* deve essere true prima di effettuare tutte le altre operazioni. In supporto è presente anche un'altra variabile *lastAccountLogged* che una volta che un utente effettua il login, memorizza il suo username. **ClientMain** presenta un ciclo (che viene chiuso solo quando client digita "exit") contenente tutte le operazioni disponibili in WINSOME. Prima di effettuare ogni operazione richiesta dal client si controlla il valore della variabile *logged* e se gli argomenti passati da riga di comando sono corretti. Metodi rilevanti:

- *sendTask*, invia al server la richiesta dell'utente e il suo nome utente;
- *registerAccount*, dopo aver ottenuto un riferimento all'oggetto remoto, invoca il metodo *registerUser* contenuto all'interno di **ServerRMImpl** per registrarsi a WINSOME;
- *loginAccount*, invoca il metodo remoto *loadFollower* per caricare i followers dell'utente, registra l'utente al servizio di notifica e lo inserisce al gruppo multicast in modo che possa ricevere le notifiche riguardo al proprio wallet;
- *logout*, rimuove l'utente dagli utenti registrati al servizio di notifica e dal gruppo di multicast, chiude la socket, azzera la lista followers e le variabili *lastAccountLogged* e *logged*;
- *listFollowers*, unico metodo che non ha bisogno di contattare il server perchè **ClientMain** possiede la struttura dati followerList che viene aggiornata in base alle notifiche callback. Il metodo quindi stampa gli utenti presenti all'interno della struttura dati.

3.1 Classi di supporto

- *api package*

  NotifyEventInterface, interfaccia remota che definisce i metodi per notificare un utente al momento di un follow o unfollow;

- *impl package*

  NotifyEventImpl, contiene la struttura dati *followersList* che viene aggiornata al momento dell'invocazione di *notifyNewFollow* e *notifyUnfollow*;

  ReceiveRewardMessage, è presente un ciclo infinito che riceve e stampa un messaggio che indica che è stato effettuato il calcolo delle ricompense;

3.2 Strutture dati

Il WinsomeClient come unica struttura dati contiene **followerList**. Essa viene inizializzata ad ogni avvio del **ClientMain**, caricata ad ogni login di un utente e azzerata al momento del logout. La lista viene utilizzata anche dalla classe **NotifyEventImpl** che la modifica aggiungendo o rimuovendo un utente al momento di un follow o unfollow.

4 Modello richiesta/risposta

L'utente interagisce con il sistema tramite la classe **ClientMain** dove sono definite tutte le funzioni che permettono di svolgere le varie attività. Ogni richiesta da parte del client viene presa sottoforma di stringa, per poi dividere ogni parola della stringa e inserirla in un array di stringhe. Questo servirà per scoprire quale comando il client ha digitato (prima parola) e controllare il numero di argomenti passati. Per esempio la richiesta "*follow user1*", verrà divisa in 2 stringhe "follow" e "user1". Le richieste vengono inviate al server principalmente tramite il metodo *sendTask* che prende come in input, oltre alla richiesta, anche il nome dell'utente che ha fatto richiesta. La classe **ElaborateTask** come **ClientMain** divide la stringa ricevuta in un array di stringhe per conoscere il comando, in seguito elabora la richiesta e manda la risposta tramite il metodo *sendTask*.

5 Multicast

Sia Client che Server tramite file di configurazione reperiscono indirizzo IP e porta multicast. Un utente dopo aver effettuato il login si unisce al gruppo multicast e tramite un thread (*RRMthread*) rimane in ascolto di futuri messaggi. Al momento del logout uscirà dal gruppo multicast. Il Server con un thread (*RCthread*), ogni volta che effettua un nuovo calcolo delle ricompense, spedisce un messaggio di notifica agli utenti presenti nel gruppo multicast.

6 Note per la compilazione ed esecuzione

I seguenti comandi devono essere effettuati posizionandosi all'interno della cartella **src**. Per la **compilazione** *lato client* utilizzare il comando:

```
javac WinsomeClient/api/*java WinsomeClient/impl/*java WinsomeClient/*java
```

mentre *lato server*:

```
javac -cp .:./WinsomeServer/lib/gson-2.8.9.jar WinsomeServer/api/*java
    WinsomeServer/impl/*java WinsomeServer/model/*java WinsomeServer/utils/*java
    WinsomeServer/*java
```

È allegato un *makefile* che automatizza la compilazione del client e server, semplicemente digitando `make` (con il target `make clean` si eliminano i file .class). Per effettuare l'**esecuzione** *lato client* utilizzare il comando:

```
java WinsomeClient.ClientMain
```

mentre *lato server*:

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

Altrimenti si possono utilizzare i rispettivi jar digitando i seguendi due comandi:

```
java -jar Client.jar
java -jar Server.jar
```