Università di Pisa

Dipartimento di Informatica

Relazione Progetto Winsome

Settembre 2022

Autore:
Giuseppe Conte

Indice

1 Introduzione

Winsome è una piattaforma social basata su client-server, in cui gli utenti possono registrarsi, pubblicare post, seguire altri utenti, visualizzare, commentare, votare e condividere i post degli utenti seguiti.
Winsome inoltre prevede di offrire una ricompensa ai propri utenti, in rapporto a quanta visibilità hanno i contenuti da essi pubblicati.

2 Compilazione ed esecuzione su terminale

Per compilare il progetto su terminale linux è sufficiente eseguire il comando:
javac -cp ./gson-2.8.9.jar *.java

Per eseguire il server su terminale linux è sufficiente eseguire il comando:
java -cp ./gson-2.8.9.jar: WinsomeServer

Per eseguire il client su terminale linux è sufficiente eseguire il comando:
java WinsonClient

3 Uno sguardo generale

Per il progetto è necessaria la libreria gson-2.8.9.jar

Il progetto è composto da 18 classi, di cui 4 dedicati esclusivamente ad RMI e RMI Callback, 3 di supporto alla grafica.
Non ho incluso un file di configurazione; tutti gli indirizzi e le porte sono dichiarati come variabili statiche all'inizio delle classi client e server.

Per eseguire il progetto è necessario avviare prima il server, successivamente uno o più client.

Lo stato del sistema viene salvato in tre file GSON differenti, che sono rispettivamente:

- **Utenti**: contiene le informazioni relative agli utenti registrati
- **Posts**: contiene le informazioni relative ai post pubblicati, con relativi commenti e voti
- **Wallets**: contiene il valore del portafoglio e lo storico degli incrementi

4 Il Server

Il metodo main del server è contenuto nella classe WinsomeServer. All'avvio il server innanzitutto ripristina lo stato del sistema recuperando i dati dai file GSON. Dopodichè il server inizializza le strutture per RMI, RMI Callback, TPC e UDP multicast.
Infine il server si mette in ascolto in attesa attiva di connessioni da parte di un client. Quando il server riceve una richiesta di connessione accetta subito la connessione, sottomette il task al threadpool e delega il lavoro di tutte le richieste del client alla classe **Worker**.
Non ho previsto la chiusura del server, che resta sempre attivo in un **while(true)**.

La richiesta di registrazione di un nuovo utente al servizio avviene mediante RMI, che invoca lo stub del metodo **RegistraUtente(...)** presente nella classe **WinsomeServer**.

4.1 La classe Worker

La classe **Worker** si occupa di tutte le richieste del client, tranne la registrazione.
La classe **Worker** resta in attesa attiva di richieste da parte del client connesso.
Le richieste ricevute sono di tipo stringa, il formato è *tipo richiesta::parametri*.
Il Worker invoca il metodo opportuno, a seconda del tipo richiesta, e calcola la risposta, che lo stesso sarà di tipo stringa.
Soltanto la richiesta di **exit** chiude la connessione verso il client.

4.2 Creare un nuovo post

Il metodo per creare un nuovo post è contenuto nella classe **Worker**.
Innanzitutto il metodo calcola il nuovo *IdPost*, di tipo int, il cui valore è 0 al primo post creato oppure il valore successivo dell'ultimo post presente.
Non utilizzo un contatore che si incrementa man mano per calcolare il nuovo idPost, e il nuovo idPost non è calcolato prendendo la dimensione + 1 dell'array posts, perchè i post che vengono cancellati ne diminiuiscono la dimensione.
Titolo e corpo del post il metodo li riceve come parametri.
Il metodo aggiorna il file GSON **posts** e l'array locale *posts*.

4.3 La conversione in bitcoin del portafoglio

Il metodo per convertire in bitcoin il valore del portafoglio è contenuto nella classe **Worker**. Il server utilizza il servizio di generazione di valori random decimali fornito da RANDOM.ORG per ottenere un tasso di cambio casuale e quindi calcola la conversione. Tale operazione viene eseguita solo se il valore del portafoglio dell'utente che ne fa richiesta è superiore a zero. In caso l'utente abbia un portafoglio uguale a zero il metodo restituisce come risposta il valore 0.

4.4 Il calcolo delle ricompense

Il metodo per calcolare le ricompense è contenuto nella classe **Timer**.
La classe **Timer** è implementata in modo tale che una volta avviata non termina mai. Il metodo **run()** contiene un **while(true)** con all'interno una **sleep(time)**, che simula l'attesa tra l'avvio dei vari calcoli, e l'invocazione del metodo **calcoloRicompense(...)**.
Il metodo **calcoloRicompense(...)** svolge il calcolo effettivo delle ricompense, aggiorna i file GSON **Posts** e **Wallets**, aggiorna l'array *posts*, e infine invia in UDP multicast a tutti i client registrati al gruppo multicast (ovvero a tutti i client online, perchè la registrazione dei client è automatica quando effettuano il login) il messaggio *"Ricompense aggiornate!"*, che i client stampano in output.

5 I file GSON

Per la conversione delle classi in formato GSON ho utilizzato la libreria esterna gson-2.8.9.jar.
Le classi convertite in formato GSON e salvate sul file sistem sono: User, Post e Wallet.

La classe **User**, il cui elenco è salvato nel file **utenti.json**, è composto dai campi:

- **nickname**: contiene il nome dell'utente, in formato stringa, il nickname è unico; quando viene creato un nuovo utente se il nickname selezionato è già presente viene restituito un messaggio di errore; il campo non può essere lasciato vuoto;
- **password**: contiene la password dell'utente, in formato stringa, salvata in chiaro; il campo non può essere lasciato vuoto;
- **tag1, tag2, tag3, tag4, tag5**: contengono i tag dell'utente, in formato stringa; i campi possono anche essere lasciati vuoti;
- **following**: è un array list di stringhe che contiene i nickname degli utenti che seguono l'utente;
- **followers**: è un array list di stringhe che contiene i nomi degli utenti seguiti dall'utente;
- **feed**: è un array list di interi che contiene gli *idpost* dei post condivisi dall'utente.

La classe **Post**, il cui elenco è salvato nel file **posts.json**, è composto dai campi:

- **idPost**: contiene l'id del post, in formato int; l'IdPost è unico; quando viene creato un nuovo Post l'idPost è il valore successivo all'ultimo post presente;
- **autore**: contiene il nickname dell'utente che ha scritto il post, in formato stringa;

- **n iterazioni**: contiene il numero di volte che il post è stato valutato dal metodo che calcola le ricompense, in formato int;
- **titolo**: contiene il titolo del post, in formato stringa, massimo 20 caratteri; il campo non può essere lasciato vuoto;
- **Textbody**: contiene il testo del post, in formato stringa, massimo 500 caratteri; il campo non può essere lasciato vuoto;
- **commenti**: è un array list di Comment, che contiene i commenti relativi al post che hanno scritto gli altri utenti;
- **voti**: è un array list di Voti, che contiene i voti che gli altri utenti hanno dato al post.

Ogni modifica al file **posts.json** non ne altera l'ordinamento.

La classe **Wallet**, il cui elenco è salvato nel file **wallet.json**, è composto dai campi:

- **proprietario**: contiene il nickname del proprietario del relativo wallet, in formato stringa;
- **portafoglio**: contiene il totale del portafoglio, in formato double;
- **portafoglio storico**: contiene un array list di Double, che rappresentano i vari incrementi del valore del portafoglio;
- **timestamp storico**: contiene un array list di String, che rappresentano i timestamp, ovvero la data, dei vari incrementi.

Gli array *portafoglio storico* e *timestamp storico* hanno lo stesso numero di elementi, li ho immaginati come array paralleli.
Il file **wallet.json** ha lo stesso numero di elementi del file **utenti.json**, per ogni utente è associato un wallet.

6 Il Client

Il metodo main del server è contenuto nella classe **WinsonClient**.
Le operazioni della classe **WinsonClient** sono implementate principalmente in due metodi: **window1()** e **window2()**.
Il metodo **window1()** viene avviato dal client main e si occupa delle operazioni di registrazione e login dell'utente.
Il metodo **window2()** viene avviato da **window1()** solo se la richiesta di login ha avuto esito positivo e si occupa di tutte le operazioni che può compiere l'utente: visualizzare la lista dei followers e dei following, cercare, seguire/non seguire più un utente, pubblicare o cancellare un post, visualizzare, condividere, commentare e votare un post di un altro utente, visualizzare il valore del proprio portafoglio, il timestamp degli incrementi, il relativo valore in bitcoin ed effettuare il logout.
All'avvio del metodo **window1()** viene effettuato il setup di RMI per la registrazione e TPC per il login. In caso di login viene effettuato il setup della RMI Callback per le notifiche di follow e unfollow di un utente.
All'avvio del metodo **window2()** viene effettutata la registrazione del gruppo UDP multicast per ricevere il messaggio del calcolo delle ricompense, e viene invocata la classe **UDPreceive** inserita in un nuovo thread che, in parallelo, si mette in ascolto sull'indirizzo di multicast. L'indirizzo IP e la porta sono uguali per tutti i client, e dichiarati all'inizio della classe come variabili statiche.
Tutte le richieste del client verso il server vengono fatte tramite il collegamento TPC.

Il Client è implementato con interfaccia grafica.
All'avvio si apre la seguente finestra, composta da sette Inputarea in cui scrivere username, password e i tag, e i bottoni di registrazione e login. In caso di login le eventuali scritte degli Inputarea dei tag vengono ignorati.

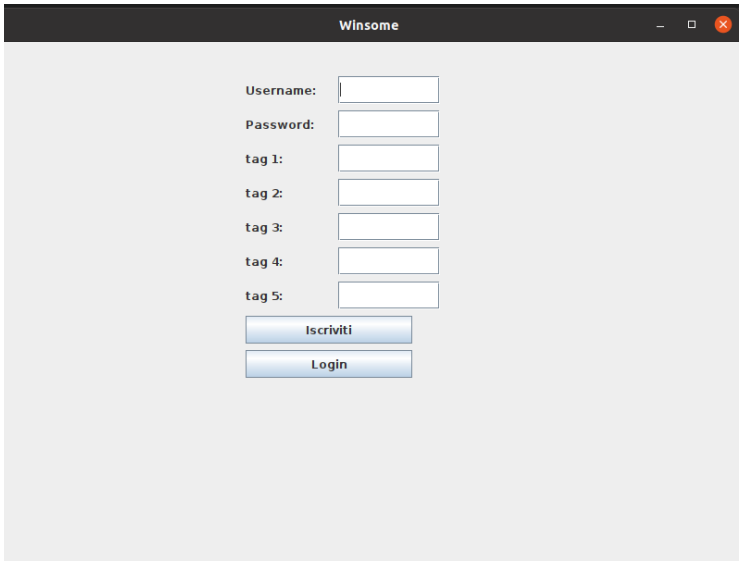

Figure 1: Finestra iniziale.

Dopo aver effettuato il login la finestra iniziale si chiude e si apre la finestra con gli Inputarea, Textarea e i bottoni per effettuare tutte le operazioni consentite dal client. Le prime tre TextArea sono per la scrittura, la Textarea più grande invece stampa i risultati delle varie richieste. Vari messaggi-risposta vengono stampati in una label basso al di sotto del bottone **Pulisci**.

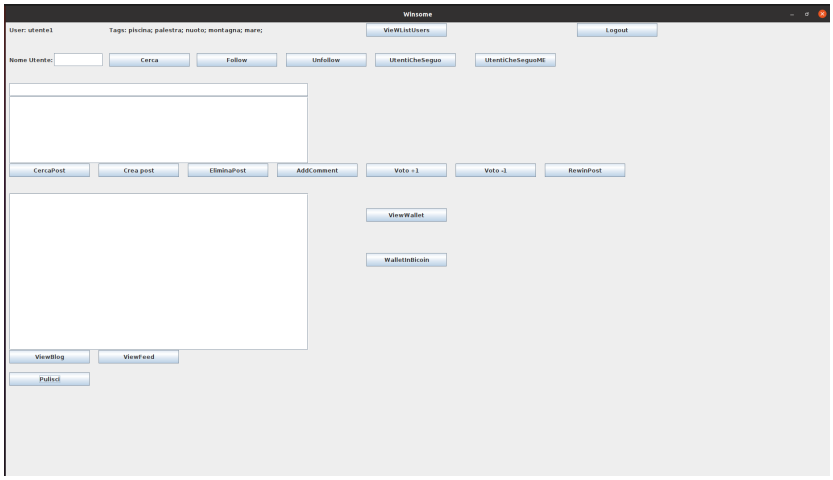

Figure 2:

Cliccando i bottoni **UtentiCheSeguo** o **UtentiCheSeguonoMe** vengono stampati nella textArea grande i dati salvati in locale; cliccando il bottone **Pulisci** vengono ripulite le Textarea e Inputarea.
Cliccando gli altri bottoni si invoca una chiamata al server con la richiesta, in formato *tipo richiesta::parametri* e si attende la risposta, che verrà stampata nella Textarea più grande oppure nella label in fondo.
In caso di **logout** la schermata corrente viene chiusa e viene riaperta la schermata iniziale.
Cliccando il pulsante di sistema **x** invece si apre una piccola finestra popup che chiede la conferma di chiusura del programma.

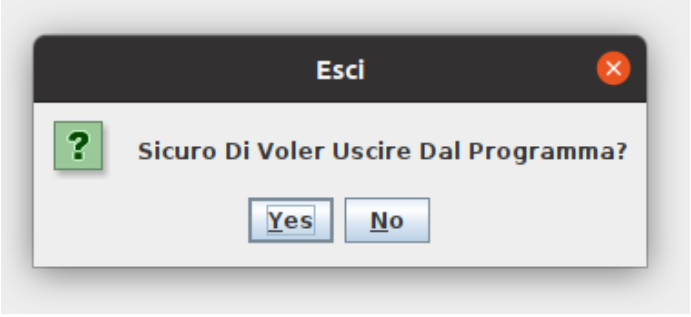

Figure 3:
Chiusura applicazione.

Di seguito un esempio di creazione di un nuovo post. Il titolo e il corpo del post vanno scritte in due textarea differenti. La conferma di *post creato con successo* viene stampata nella label in fondo.

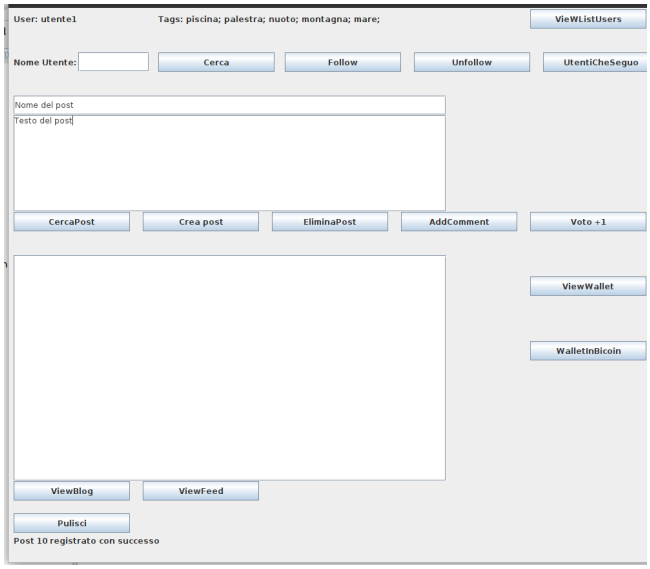

Figure 4:
Creazione di un nuovo post.

Di seguito un esempio di visualizzazione del proprio Blog. Il risultato viene stampato nella Textarea più grande.

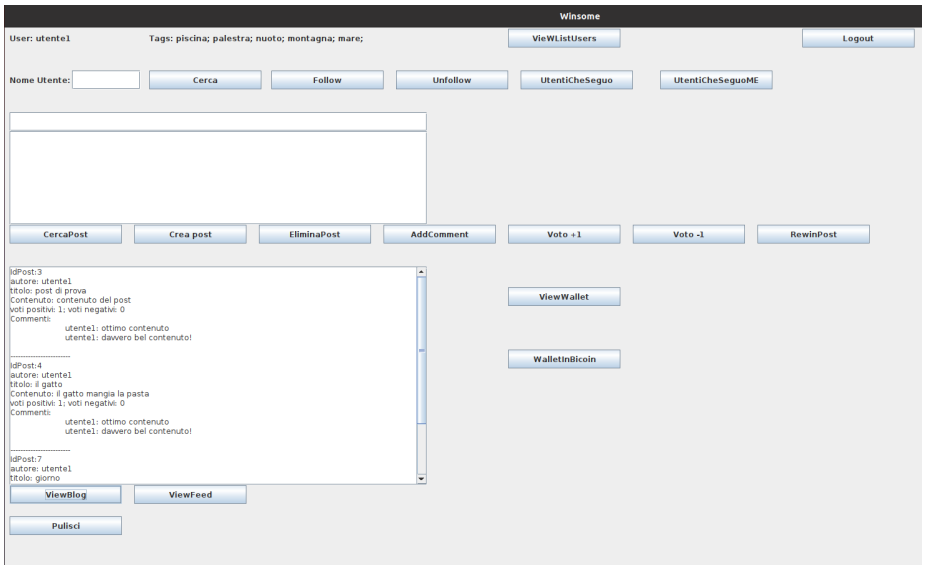

Figure 5:
Visualizzazione del proprio Blog.

7 I thread attivi

7.1 Lato server

All'avvio il server invoca *Thread Server = new Thread(WinsomeServer).start();*, creando così un ulteriore thread apparentemente superfluo, tuttavia tale invocazione è resa necessaria dal fatto che altrimenti il riferimento **this** nel main non avrebbe funzionato.
Il server inoltre attiva un thread demon per gestire il timer e il calcolo delle ricompense.
Per gestire le richieste dei client la classe Server fa uso di un CachedThreaPool, perchè la connessione di un client al server è dedicata e quindi ogni richiesta di un client, dal login fino alla chiusura del client, viene gestita dallo stesso thread Worker.

7.2 Lato client

Dal lato client viene attivato un unico ulteriore thread, oltre al principale.
Tale thread viene attivato successivamente al login, e il metodo all'interno del thread si occupa di restare in ascolto sull'indirizzo di multicast che riferisce la terminazione del calcolo delle ricompense.
Visto che il metodo resta sempre attivo in un **while(true)** ho preferito settare il thread come demone, in modo da avere la certezza che quando termina il thread main del client termina anche tale thread.

7.3 Sincronizzazione e concorrenza tra thread

Per la concorrenza delle varie strutture condivise non ho usato particolari costrutti, ogni metodo che gestisce strutture condivise l'ho dichiarato **synchronized**, ovvero ho usato i monitor. Tale scelta implementativa per via del loro uso estremamente facile.
Per la sincronizzazione non ho usato alcun costrutto, ho programmato in modo da non rendere necessario l'uso di sincronizzazione mediante **wait()** o **notify()**.