Davide Bulotta 596782

RELAZIONE PROGETTO WINSOME
Laboratorio di reti A.A. 2021/2022

Informazioni sul sistema

Il social media WINSOME è costituito da due componenti, ClientMain e MainServer.

Entrambi all'avvio leggono da un file di configurazione per settare i parametri necessari al loro funzionamento.

Successivamente, il client stabilisce una connessione con il server, lancia il thread CalculateClient e resta in attesa di accettare i comandi dell'utente.

Il server verifica se sono presenti file di backup da cui recuperare uno stato precedente, altrimenti ne crea dei nuovi files in cui salverà i dati che si verranno a creare durante il funzionamento di WINSOME (per costruire ed aggiornare i backup sono state utilizzate le librerie Jackson). Successivamente avvia un thread che si occupa del calcolo delle ricompense, stabilisce una connessione RMI, una connessione TCP e, successivamente si mette in attesa di ricevere comandi dal client. Il client utilizzerà l'RMI per i comandi register e showPost, mentre gli altri comandi saranno gestiti tramite la connessione TCP

La tecnologia RMI è stata utilizzata per le seguenti operazioni:

```
public interface ServerInterface extends Remote {
        // Registrazione
        public boolean registerUser(String username, String password, LinkedList<String> tags) throws RemoteException;
        //
        public String updateClientIdList(String username) throws RemoteException;
        //
        public Post showPost(int idPost) throws RemoteException;
        //
        public void resetClientId(String username) throws RemoteException;

        void registerForCallBacks(NotifyClientInterface clientInterface) throws RemoteException;

        void unregisterForCallBacks(NotifyClientInterface clientInterface) throws RemoteException;
}
```

La prima è utilizzata per la registrazione dell'utente, la seconda e la quarta vengono utilizzate per tenere traccia della relazione tra un utente specifico e il suo client, così da permettere di aggiornare le informazioni di callBack in modo totalmente soggettivo per ogni singolo utente.

La terza permette di ricevere dal server tutte le informazioni riguardo il post che si sta cercando.

Poco prima che il server sia definitivamente avviato parte un thread che richiama la classe **CalculateSystem**. Questa si occupa, ad intervalli di tempo costanti individuati dal parametro **sleepTime**, di calcolare e aggiornare tutte le ricompense e i portafogli degli utenti ed il loro storico, notificando l'avvenuto aggiornamento a tutti gli utenti collegati attraverso una connessione Multicast tra la **CalculateSystem** e la **CalculateClient**. Quest'ultima, che lavora come thread a parte rispetto al ClientMain, viene avviata quando un utente effettua il login sul proprio account e viene interrotta dal comand di **logout**.

Gli altri comandi vengono letti dal client, che verifica se rispettano i requisiti previsti e successivamente inviati al server.

Il server li riceve, li riconosce ed effettua l'attività prevista.

Le strutture dati

Tutte le informazioni all'interno del server sono raccolte mediante ArrayList<>, con le uniche eccezioni che riguardano la followers, la usersListTag e la feedList che sono delle HashMap. Ho scelto di adottare queste strutture dati perché il sistema di backup si implementa più semplicemente.
Infatti, facendo uso del package Jackson, tutte le informazioni della classe **User** vengono salvate all'interno della **ArrayList<User> users**. Le ArrayList vengono ricostruite utilizzando il backup e modificate dal server in tempo reale. Queste strutture consentono di aggiornare il suo blog, il suo feed rendendo soggettiva l'esperienza di ogni singolo utente.

Informazioni sulle classi

Il progetto è costituito dalle seguenti classi:

- CalculateClient
- CalculateSystem
- Comment
- NotifyClientInterface
- Post
- ServerInterface
- User

Le classi **CalculateClient e CalculateSystem** lavorano simultaneamente ai loro main. La prima viene eseguita come thread simultaneo del client dopo che l'utente ha eseguito il login. Se l'utente dovesse effettuare il logout, allora il thread riceverà un **interrupt()**. Lo scopo della CalculateClient è di ricevere le notifiche inviate dalla classe CalculateSystem mediante gruppo multicast.

Poco prima che il server sia definitivamente avviato viene fatto partire un thread che richiama la classe **CalculateSystem**. Questa si occupa, ad intervalli di tempo costanti individuati dal parametro **sleepTime**, di calcolare e aggiornare tutte le ricompense e i portafogli degli utenti ed il loro storico, notificando l'avvenuto aggiornamento agli utenti collegati attraverso una connessione Multicast tra la **CalculateSystem** e la **CalculateClient**. Quest'ultima, che lavora come thread a parte rispetto al ClientMain, viene avviata quando un utente effettua il login sul proprio account e viene interrotta dal comand di **logout**.

La CalculateSystem lavora in modo simultaneo al server, ogni tot SLEEPTIME (Valore in secondi del tempo di aggiornamento) esegue un calcolo di tutte le ricompense tenendo conto dei post del blog che hanno ricevuto delle modifiche, per esempio nuovi commenti o nuovi like, aggiorna il loro wallet ed il loro storico. Completate tutte queste operazioni notifica mediante gruppo multicast ai vari thread CalculateClient. Per gestire la concorrenza la CalculateSystem, prima di verificare i dati, effettua una lock di tutte le strutture dati e le libera non appena ha terminato i suoi aggiornamenti, in quanto il rischio di conflitti è abbastanza alto.

La Classe **Post** modella la struttura dei post e tiene traccia al suo interno delle newPeopleComment e del rewardsList. Queste strutture permettono al CalculateSystem di verificare quali sono i nuovi commenti che sono stati effettuati al post dall'ultimo calcolo delle ricompense e a chi assegnarle. I membri della rewardsList vengono modificati nel caso in cui qualche utente dovesse modificare il proprio voto al post da positivo a negativo o viceversa. Il valore newPeopleLikes tiene conto invece di tutti gli utenti che hanno inserito un voto (sia positivo che negativo).

La classe **Comment** è abbastanza semplice e rappresenta la struttura di un commento ad un post.

La classe **User** modella gli utenti del sistema, i loro dati e le attività che possono compiere. Il sistema di backup è basato sulle proprietà della classe User.

```
 */
private static final long serialVersionUID = 1L;
private String username;
private String password;
private int userId;
private String clientId;

//@JsonProperty("posts")
private ArrayList<Post> posts = null;

//@JsonProperty("followers")
private ArrayList<String> followers = null; // Lista degli utenti che lo seguono

private ArrayList<String> following = null; // Lista degli utenti che segue

//@JsonProperty("tags")
private ArrayList<String> tags = null;
//
private int numberComments;
//Per tenere traccia delle transazioni
private ArrayList<Double> history = null;
//
private double wallet = 0;

public User() {
        /*
         * Jackson neccesita' di un costruttore vuoto o del
         * @JsonProperty("example")
         * per riuscire a costruire il modello
         */
}
```

La list **ArrayList<Post> posts** è del tipo Post e farà riferimento al suo interno di una ArrayList<Comment> comments per poter tenere traccia dei commenti all'interno dei post.

Tutte le classi contengono i metodi get e set, così da poter ricavare e modificarne gli attributi pubblicamente. Ad esempio, il wallet ed l'history, che rispettivamente tengono conto del saldo del portafoglio e dello storico delle operazioni, possono ovviamente essere aggiunti e modificati questi valori.

Nel sistema ogni singola azione di qualsiasi client può avere ripercussioni sui dati degli altri utenti.