WINSOME: rewarding social media

Progetto di Reti di Calcolatori – Laboratorio Michela Deodati 597983

28 Giugno 2022

STRUTTURA:

La directory Winsome contiene la cartella **jar** che contiene i file WinsomeClientMain.jar e WinsomeServerMain.jar, la cartella libraries che contiene le librerie esterne utilizzate e in fine la cartella **src** che contiene i file sorgente del social Winsome. La cartella src contiene anche le cartelle **Color e CustomizedException.** Color ha all'interno la classe ColoredText.java che definisce una serie di stringhe static le quali inserite come prefisso su stdout permettono di colorare le scritte visualizzate da terminale (purtroppo su alcune shell il prefisso per il **bold** stampa una faccina sorridente). **CustomizedException,** invece, è la cartella che contiene le classi che definiscono le eccezioni personalizzate, che specificano errori di tipo diverso durante l'uso del social.
Alla prima esecuzione del progetto non esiste una cartella che contenga i Json e i file di configurazione perché verrà creata in seguito con il primo lancio del server e il primo lancio del client.

-SharedMethods:

Classe che contiene i metodi di scrittura e lettura da stream, alcune verifiche che sono utili sia lato client sia lato server, sostanzialmente una classe che contiene "metodi condivisi", come dice il nome. I metodi principali che questa classe implementa sono **sendToStream** e **readFromStream**. I messaggi sono divisi in 2 pacchetti, il primo contiene la dimensione del messaggio e l'altro il messaggio stesso. Implementa anche un metodo che permette di leggere da console.

Ci sono anche altre classi che sono condivise tra server e client:

-ConfigReader:

Il nome di questa classe è parzialmente improprio, in quanto non si occupa solamente di leggere i file di configurazione ma si occupa anche di crearli qualora mancassero. L'utilizzo di "*java.util.Properties*" è dovuto alle funzionalità che questa classe possiede ad alto livello per la lettura e scrittura di file (i file di config hanno estensione "**.properties*"). Quando viene istanziato un oggetto di questa classe, se non trova le cartelle e i file corrispondenti dove leggere le informazioni le crea ex-novo e sulla base di un flag che gli viene passato sa se deve leggere (o eventualmente creare) un file di configurazione Server o Client. L'id dell'ultimo post pubblicato in Winsome e l'ultima data di RewardCheck vengono salvate sul file di configurazione del server tramite il metodo **changeSaveConfig.**

ENTITÀ:

- **User:** Classe che rappresenta gli utenti del social Winsome, gli attributi sono **nickname, password** di tipo String e poi, per non avere duplicati, un HashSet <String> che contiene i tags (minimo 1 max 5). La password non viene cifrata, ma salvata in chiaro.
- **Wallet:** Rappresenta il portafoglio di un utente di Winsome. È di particolare rilevanza in questa classe il metodo *getWalletBitcoin* che ottiene un numero random da Random.org, come da specifica, con 20 cifre decimali; se si verifica un problema durante la connessione viene stampato un messaggio di errore. La lista dei **WalletMovement** è salvata in una *ConcurrentLinkedQueue.*

- **WalletMovement:** Rappresenta le transazioni in Winsome, ogni transazione è costituita da un importo, una causale e la data (*java.util.Date*) di esecuzione della transazione.
- **Vote:** Questa entità rappresenta il voto di un post di Winsome che può avere valore +1 -1, il controllo dei valori viene effettuato in altre classi.
- **Comment:** Le sue istanze sono i commenti sotto ai post, contengono il nome dell'autore e la data del commento che serve per il calcolo delle ricompense.
- **Post:** Rappresenta i post di Winsome:

```
// costruttore della classe post
public Post(String owner, String title, String text, int id) {
    this.owner = owner;
    this.date = new Date();
    this.postId = id;
    this.title = title;
    this.text = text;
    this.postComment = new HashSet<>();
    this.postRewinUser = new ConcurrentLinkedQueue<>();
    this.allPostVotes = new ConcurrentHashMap<>();
    this.nIterazioni = 0;// inizializzo a zero il numero di iterazioni
}
```

Owner = autore del post, date = data di pubblicazione del post, postID = identificativo del post, Title = titolo del post, text= contenuto del post, poi ci sono strutture come l'HashSet<Comment> ,ossia la lista dei commenti del post, una *ConcurrentLinkedQueue* contenente i nomi degli utenti che hanno fatto **rewin** di questo post, rimangono salvati anche se un utente fa l'unfollow dall'autore del post,e una *ConcurrentHashMap* che contiene l'associazione tra voto-nome_votante , e infine il numero di iterazioni che l'algoritmo di calcolo premi ha già fatto sul post.

- **ClientSession:** Associazione tra un Socket e il nome dell'utente loggato, utilizzato nel ConnectionHandler per effettuare controlli (ad esempio controllare se un utente è loggato o meno)

LATO CLIENT:

Il Client avvia 2 thread: il main thread e un thread che riceve notifiche multicast dal gruppo definito nel file di configurazione tramite indirizzo ip e porta; questo thread viene avviato dal thread main e rimane in ascolto fino a che non si chiude il client con il comando **"exit"** che chiude il multicast socket su cui il thread attende di ricevere pacchetti.

-WinsomeClientMain:

Il Client è stato creato per svolgere il minor numero di operazioni possibili, seppur effettuando alcuni controlli preliminari sulla sintassi dei comandi, lasciando tutta la parte computazionale al Server. La prima cosa che fa il Client all'avvio è leggere dal **ConfigReader** i file di configurazione che contengono tutte le informazioni utili per la connessione TCP con il server e per i servizi RMI. Dopo aver letto i file di configurazione apre una **connessione TCP** con il server all'indirizzo e porta letti in precedenza. Localizza il registro all'indirizzo specificato e prepara il reader e il writer, per poi entrare in un loop nel quale l'utente può inviare le sue richieste al server.
In caso di perdita di connessione con il server il client avvierà un tentativo di riconnessione. Il client si accorge di un eventuale interruzione di connessione solo alla mancata esecuzione di un comando (tranne help e exit).

****** **Comandi più importanti lanciati lato client** ******

Il client è in grado di interpretare i comandi anche se scritti in maiuscolo.

- exit → termina il client effettuando una logout se necessario, chiudendo reader, writer, thread in ascolto delle notifiche aggiornamento wallet e socket con il server.
- register → effettua la registrazione di un utente. Il client dovrà inviare nome utente (che verrà salvato in lowercase per praticità) password (in chiaro) e una lista di almeno 1 tag (che verranno salvati in uppercase, per scelta stilistica). Usa RMI per eseguire il metodo remoto
  ```
  public boolean registerNewUser(String nickname, String password, HashSet<String> tags) throws RemoteException
  ```
- login → manda al server la richiesta di accedere all'account utente con i parametri specificati, la richiesta ha successo se l'utente si è registrato in precedenza e nome utente e password corrispondono con quelli salvati. Se il metodo ha successo il server risponde al client **"OK"** l'operazione a questo punto è completata e si registra il client al *callback* del server per le notifiche di aggiornamento follower. La procedura di accesso termina con il metodo via RMI

  ```
  public Set<String> followerList(String nickname) throws RemoteException {
  ```

  che restituisce la **followersList** al client che ne tiene una copia locale, se non esiste ancora la inizializza .
- logout → manda al server una richiesta di disconnessione, la risposta può essere di diverso tipo in base allo stato del client, in sostanza la richiesta ha successo solamente se lato client è stato fatto un login prima, se l'operazione di logout ha successo il client viene rimosso dalla lista dei callback del server.
- listfollowers → il client tiene salvata in un **Set<String>** la **followersList** di un utente. Tale Set è il risultato di `ConcurrentHashMap.newKeySet()`, il quale si comporta come un hashset perché è implementato con un algoritmo hash-based e garantisce la consistenza delle informazioni. La followersList viene aggiornata tramite il metodo
  ```
  public void notifyEvent(String s) throws RemoteException {
  ```
  via RMI.
- help → stampa una lista di tutti i comandi che è possibile digitare lato Client con la loro spiegazione.
- Il resto dei comandi viene inviato come richiesta sulla connessione con il server, tramite **SharedMethods.sendToStream**.

È doveroso specificare che per via dell'interpretazione della richiesta eseguita dal ConnectionHandler i comandi (esclusi gli argomenti) devono essere digitati tutti attaccati (listfollowers invece di list followers, listfollowing invece di list following, listusers invece di list users ecc.).

-WalletRewardNotifier:

È la classe che si occupa di stampare su Stdout la notifica di aggiornamento wallet, inviandola a tutti i client connessi tramite **Multicast Socket**. Il thread che gestisce questa operazione viene avviato nel **WinsomeClientMain** e rimane in ascolto dei pacchetti in arrivo dal server. Il thread viene terminato quando si fa la "exit" dal client. Come per le richieste TCP si è scelto di inviare prima i byte di dimensione del pacchetto e poi l'effettivo contenuto.

-NotifyEvent:

Contiene il metodo `public void notifyEvent(String s) throws RemoteException {` che ha il compito di gestire la followersList lato client. La sintassi della stringa s passata come argomento è fissa: **FOLLOW+"nome utente"** oppure **UNFOLLOW+"nome utente"** che differenziano rispettivamente un'aggiunta alla followersList o una rimozione. È impossibile che si verifichino errori in questo metodo perché la stringa è preformattata dal server e non la deve scrivere l'utente.

LATO SERVER:

I thread generati lato server, escludendo quelli delle connessioni sono due più il mainThread stesso che li genera. Uno è il *RewardManager* che viene avviato prima dell'apertura del serverSocket TCP nel main. L'altro è il thread che si occupa di gestire i backup su file Json. I thread avviati per la gestione delle connessioni in ingresso sono uno per ogni connessione che viene stabilita e vengono istanziati tramite una **CachedThreadPool** che cresce dinamicamente riutilizzando i thread usati in precedenza, questa pool in generale migliora le prestazioni di programmi che eseguono piccole task asincrone.

-WinsomeServerMain:

Contiene e inizializza le strutture dati che sono usate dalle altre classi. È la classe che implementa il socketServer TCP per accettare le connessioni, conserva anche la lista delle **ClientSession** attive e quella dei **Socket** connessi. All'avvio vengono istanziati:

- configReader → si occupa della lettura (o creazione) dei file di configurazione.
- socialManager → gestisce la maggior parte delle funzionalità del progetto.
- fileManager → carica le informazioni dal file di configurazione.
- dataBackup → effettua un salvataggio periodico dello stato del server.
- rewards → si occupa dell'assegnazione dei premi in Wincoins e di comunicare l'aggiornamento dei *Wallet* tramite notifica su **MulticastSocket.**

Successivamente c'è la creazione dei registri RMI che permettono al client di fare la *register* e di iscriversi ad una lista di *callback* per ricevere aggiornamenti sui propri follower tramite metodi remoti. Ogni client viene gestito con un singolo thread gestore che si occupa di tutte le richieste: ***ConnectionHandler***. La threadpool utilizzata è di tipo *cached* che viene consigliata quando si devono gestire un numero ignoto di thread futuri.

-Backup:

Questa classe si occupa del salvataggio periodico dei dati persistenti (utenti registrati, transazioni, wallet, ecc). Implementa un runnable che verrà poi lanciato nel main del server. L'intervallo di salvataggio è stabilito nel file di configurazione del server nella cartella "*config*" sotto la voce di "BackupInterval", di default fissata a 30 secondi ma può essere cambiata (sia da codice che direttamente da file di config), le uniche due cose che vengono salvate nel file di configurazione invece che nei Json sono il valore dell'ultimo id dei post, in modo che possa essere ripristinato all'apertura del server, e la data dell'ultimo check delle ricompense. Il salvataggio e il caricamento dei dati dai file Json avvengono tramite la classe **JsonFileManager** in particolare per eseguire il backup è importante il metodo `public void save(SocialManager socialManager)` a cui viene passato il **SocialManager** che contiene tutte le informazioni su utenti, wallet, followers/following e post.

-JsonFileManager:

Il **JsonFileManager** gestisce tutta la parte di salvataggio e caricamento dei file Json che servono per ricostruire lo stato del sistema, in particolare i file Json salvati sono:

- *user.json*
- *post.json*
- *follower.json*
- *following.json*
- *wallet.json*

Si trovano nella stessa cartella dei file di configurazione, dentro **jsonFile**. Se i file Json non esistono li genererà automaticamente, altrimenti vengono semplicemente letti. Il metodo che preleva le informazioni è `public void loadBackupFile(SocialManager socialManager)` che esegue una ad una le load dei file. La lettura di questi avviene tramite **StringBuilder** in cui si salva mano a mano il file interessato e per poi essere letto tramite il **ByteBuffer**. Per implementare questa classe è stata usata la libreria **gson-2.8.2.jar** che è una libreria sviluppata da Google completamente open-source.

(https://github.com/google/gson).

-RmiCallback:

Contiene i metodi remoti invocati dal client per iscriversi alla callback. **CallbackRegister** aggiunge l'interfaccia del client e l'username dell'utente collegato ad una *ConcurrentHashMap* in modo che il server possa aggiornare la lista followers tenuta dal client tramite metodo remoto. Il metodo **CallbackUnregister** rimuove l'utente dalla lista del callback, in modo che non riceva più notifiche riguardo l'aggiornamento lista follower, viene chiamato dal client quando un utente esegue la logout.

-RmiService:

RmiService contiene 2 metodi che vengono eseguiti via RMI: **registerNewUser** che è il metodo che permette di iscrivere un utente al social winsome , per il quale è necessario specificare un nome utente , una password e una lista di tag che vanno da 1 a 5. Il secondo metodo è **followerList** che serve per recuperare l'intera lista follower dell'utente dal server, che è fondamentale per tentere la lista completa dei follower lato client, altrimenti rimarrebbe in memoria solo quella che viene salvata dopo la login.

-ConnectionHandler:

È la classe che gestisce le connessioni tra server e client, ogni istanza di questa classe gestisce una connessione in modalità bloccante. Il costruttore della classe riceve come parametri il *SocialManager* del Server, il *ConfigReader* del server e il *Socket client-server*. Allo **start()** del thread si prepara ad ascoltare i messaggi provenienti dal client e li gestisce di conseguenza tramite lo switch della richiesta che viene splittata sugli spazi (comandi digitati senza spazi rendono il codice del parsing più leggibile e semplice) e procede ad una serie di controlli sui parametri, sugli utenti e sulle operazioni da eseguire, parte dei controlli vengono effettuati già lato client, come ad esempio alcuni controlli sulla sitassi dei comandi: comment deve avere il testo del commento "tra virgolette". Se la richiesta passa la verifica viene chiamata la rispettiva funzione corrispondente nel SocialManager passato, che restituirà o un messaggio di conferma o un messaggio personalizzato contenete l'errore corrispondente all'eccezione sollevata, per esempio l'eccezione **PostNotFoundException** viene sollevata quando il post non è nella `private ConcurrentHashMap<Integer, Post> postList;` del socialManager.

-SocialManager:

Questa classe è il vero cuore pulsante di Winsome, implementa tutti i metodi che manipolano informazioni.

In questa classe sono salvate in delle ConcurrentHashMap<> tutte le associazioni che sono necessarie per gestire il social:

```
private ConcurrentHashMap<String, User> userList;
private ConcurrentHashMap<Integer, Post> postList;
private ConcurrentHashMap<String, Wallet> walletList;
private ConcurrentHashMap<String, HashSet<String>> followersList;
private ConcurrentHashMap<String, HashSet<String>> followingList;
```

La scelta della ConcurrentHashMap<> è stata ponderata: data la sua implementazione semplice e il fatto che è fortemente consigliata per salvare informazioni ad alta concorrenza di aggiornamenti. La prima cosa che salta all'occhio è la ridondanza di alcune strutture come la **followersList** e la **followingList** ridondate per velocizzare il tempo di accesso. L' id del post corrente in winsome è salvato in un *AtomicInteger* utile nel contesto multithreaded considerato. I metodi implementati in questa classe vengono chiamati nel **ConnectionHandler** che in base al tipo di eccezione custom sollevata capisce che tipo di errore si è incontrato, Il SocialManager ed il ConnectionHandler lavorano a braccetto per quanto riguarda il controllo degli errori e delle eccezioni, il **SocialManager** esegue controlli principalmente sulla logica dei comandi (post inesistente, utente non trovato) mentre il **ConnectionHandler** si occupa principalmente degli errori di sitassi e della comunicazione con il client. Il SocialManager contiene anche il metodo che formatta la stampa dei post:

```
Post 7
Titolo del post:        TIOTOLO DEL POST
Da: tizio       In Data: Sun Jun 26 11:31:41 CEST 2022
Testo: testo del post

UpVote: 0       DownVote: 0
2 utenti hanno commentato questo post.
michela:    commento da un'altro utente
tizio:    commento al post dell'autore
```

la stampa formattata del post comprende titolo in maiuscolo, autore sulla sinistra, data sulla destra a distanza "\t", testo del post, numero di upvote, numero di downvote, numero di commenti e commenti scritti sotto con annesso autore del commento.

Per scelta si è permesso agli utenti di commentare i loro stessi post, su esempio di social come facebook e instagram che lo permettono (ovviamente non viene incluso nel calcolo delle ricompense). Tuttavia un utente non può votare il suo stesso post.

-RewardManager:

Questa classe gestisce il calcolo delle ricompense e l'invio della notifica al client, all'avvio questo thread tramite una *try-with-resources* apre una DatagramSocket e inizia ad elaborare il premio tramite il metodo **gainFormula** secondo la formula indicata nelle specifiche del progetto. Il premio per autori è 70% del totale, mentre agli utenti che commentano e che votano vengono distribuiti i restanti Wincoins divisi in parti uguali. Vengono aggiornati i portafogli degli utenti che devono ricevere la ricompensa e viene mandata una notifica tramite **UDP** su indirizzo e porta specificati nel file di configurazione.

NOTE:

Compilazione: estrarre l'archivio, aprire la cartella **jar** da terminale e digitare:

```
java -jar .\WinsomeClientMain.jar
```

per compilare il client.

```
java -jar WinsomeServerMain.jar
```

per compilare il server.

(io ho compilato da PowerShell).
È indifferente se si lancia prima il client o il server, perché il client ha un meccanismo di riconnessione.
Se cade la connessione tra server e client, il client dà la possibilità di riconnettersi.