WINSOME: re**w**ard**in**g **so**cial **m**edia

Progetto di Reti di Calcolatori – Laboratorio Michela Deodati 597983

28 Giugno 2022

***LATO CLIENT:***

Il Client avvia 2 thread il main thread e un thread che gestisce notifiche multicast ricevute 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 forzatamente 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 **socketTCP** con il server all’indirizzo e porta letti in precedenza. Localizza il registro all’indirizzo specificato e prepara i reader e i writer, per poi entrare in “while(true){}” nel quale l’utente può inviare le sue richieste al server.

In caso di perdita di connessione con il server (*“l’orco cattivo che lo imprigiona o il perfido stregone”*) il client avvierà un tentativo di riconnessione con il server. Per capire se la connessione è ancora attiva è necessario che lato client venga fatta una richiesta che il server dovrà risolvere (non tutti i comandi generano una richiesta remota).

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

* exit 🡪 termina forzatamente il client 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



La “conta” dei parametri viene effettuata lato client.

* 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 cliente **“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
* 

che restituisce la **followersList** al client, 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 **HashSet<String>** la **followersList** di un utente, la quale viene aggiornata tramite il metodo  via RMI

questa lista viene acceduta esclusivamente da un thread, non c’è quindi motivo di avere controlli sulla concorrenza.

* help 🡪 stampa una lista di tutti i comandi che è possibile digitare lato Client con la loro spiegazione.
* Il resto dei comandi viene inoltrato come richiesta “raw” direttamente al server.

È doveroso specificare che per via dell’interpretazione della richiesta lato client i comandi, esclusi gli argomenti, devono essere digitati tutti attaccati.

**-WalletRewardNotifier:**

È la classe che si occupa di mostrare a schermo quando vengono aggiornati i portafogli degli utenti, invia una notifica generale a tutti i client connessi tramite **Multicast Socket** dopo aver calcolato le ricompense relative ai post tramite la formula contenuta nella specifica del progetto. 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. Il problema da risolvere per questa classe era quello della dimensione dei pacchetti che è variabile a seconda del numero di cifre delle ricompense, come soluzione ho deciso di inviare prima la dimensione del pacchetto in byte e poi l’effettivo contenuto, in modo da poter allocare la giusta quantità di spazio ne buffer, questo rende più affidabile la ricezione di messaggi di cui non si conosce a priori la lunghezza, rispetto al classico byte[] statico.

**-NotifyEvent:**

Contiene il metodo  che ha il compito di gestire la followersList. 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 viene passata la stringa preformattata e non la deve scrivere l’utente.

***LATO SERVER:***

I thread generati lato server, escludendo quelli delle connessioni sono tre più il main thread stesso che li genera. Il primo che cito è il thread dell’**ServerAdminHandler** viene avviato prima dell’apertura della TCP socket del server main, serve esclusivamente a scopo di debugging e controllo del server, arresta forzatamente il server tramite il comando **stopserver** è possibile controllare le statisticeh della connessione e del server tramite il comando **winsome** e ha altre funzioni che mi sono servite a scopo di debug come la possibilità di eseguire transazioni, controllare le liste followers aggiornate e chi sono gli utenti registrati al social (irrilevante per il funzionamento del social stesso ). Il secondo thread è quello del RewardManager che viene avviato prima dell’apertura del server socket TCP nel main. Ultimo ma non per importanza 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 messi dentro una **CachedThreadPool,** vengono inseriti anche nella *clientSocketList* in modo che possono essere chiusi quando si esegue un arresto forzato del Server.

**-WinsomeServerMain:**

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

* configReader 🡪 si occupa della lettura 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*  o di iscriversi ad una lista di *callback* per ricevere aggiornamenti sui propri follower tramite metodi remoti. I client vengono gestiti 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. E inoltre permette di controllare le risorse utilizzate riciclandole quando possibile. Il server resta in attesa di connessioni fino aquando non si verificherà una *SocketException* generata dal metodo *accept*( ). Il server è inoltre in grado di elaborare delle richieste di input grazie al thread *ServerAdminHandler*  che, ripeto ho creato a solo scopo di debugging e controllo del server, ad esempio il comando “**winsome**”permette di vedere delle statistiche del server e “**stopserver**” chiude il server in maniera sicura. Quando il server si spegne l’ultima azione è quella di salvare i dati che non è preoccupazione del server.

**-Backup:**

Questa classe si occupa del salvataggio periodico dei dati. 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 30 secondi ma può essere cambiata, 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 rispristinato all’apertura del server e la data dell’ultimo check delle ricompense. Il salavataggio e il caricamento dei dati dai file Json avvengono tramite la classe **JsonFileManager** in particolare per eseguire il backup è importante il metodo  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 serveono per ricostruire lo stato del sistema, in particolare i file Json salavati sono:

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

Si trovano nella stessa cartella dei file di configurazione. Se i file Json non esistono non c’è da preoccuparsi perché al primo salvataggio li genererà automaticamente. Il metodo che preleva le informazioni è  che esegue una ad una le load dei file. La scelta di tenere tutte le informazioni in file diversi anzi che in unico file è a scopo preventivo, cioè , file di grandi dimensioni aumentano al possibilità che questi possano corrompersi. 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 compeltamente open-source che rende facile, veloce e intutivo convertire oggetti java nella loro rapresentazione Json.

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

**-RmiCallback:**

Contiene i metodi remoti invocati dal client per iscriversi alla callback. **CallbackRegister** aggiunge lìinterfaccia 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, di modo che non riceva più notifiche riguaro 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 salvata solo quella che viene salavata dopo la login.

**-ConnectionHandler:**

È la classe che gestisce le connessioni tra server e client, ogni istanza di questa classe gestisce una connessione. L’implementazione come thread è stata adottata per comodità, rispetto ad una possibile alternativa su selector e channel. 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 le gestisce di conseguenza tramite lo switch della richiesta che viene splittata sugli spazi (motivo per cui ho preferito far digitare comandi come listusers, listfollowers, listfollowing ecc. tutti attaccati) 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 .