Laboratorio di Reti - Corso A

WINSOME: a reWardING SOcial MEdia

Progetto di Fine Corso A.A. 2021/22

Aru Giacomo

Matricola: 597700

Introduzione

Il progetto "Winsome" consiste in un sistema Client Server che realizza un Social Network che permette a utenti di condividere post e interagire con questi attraverso diverse funzionalità, come per esempio il voto. Esiste inoltre un meccanismo che assegna all'autore e a chi interagisce con un post una ricompensa in valuta digitale, in relazione a quanto questo post riceva commenti e voti positivi.

Per la realizzazione di questo progetto ho diviso il lavoro in due sotto-problemi distinti, affrontati singolarmente. Inizialmente ho sviluppato l'insieme di strutture dati che permettono al server di gestire un Social Network, con la memorizzazione di tutto il necessario. Poi ho creato la struttura di thread che gestisce il social e ne permette l'interazione con l'utente; questo in contemporanea con lo sviluppo del Client.

Il SocialNetwork

Questo rappresenta logicamente l'oggetto social che viene aggiornato dal server a seguito di richieste da parte di Client. Tutte le classi utili alla sua implementazione sono parte del package

SocialNetworkClasses, esso contiene al suo interno l'interfaccia SocialNetworkInterface che viene utilizzata dal Server, contiene anche la sua implementazione in SocialNetwork. Qui sono presenti anche classi di supporto quali: User, Post, PostRewin, Comment, PostToken, WalletTransaction.

Separando in questo modo il lavoro ho potuto semplificare notevolmente il problema più complesso della creazione del Server, creando prima il social e poi utilizzandolo come libreria per la creazione del Server. Facendo ciò ho implementato un insieme di funzioni (non tutte utilizzate) nel social in modo da poter agevolare l'utilizzo dello stesso.

Importante notare come il social non sia creato in stretta relazione con la necessità di memorizzare uno stato di un social network basato su un paradigma Client-Server, in cui la sicurezza di interazione tra utenti è basata sulle credenziali di accesso ad ogni "account"; Bensì il social riceve richieste sulla modifica dello stato da parte di un utente esterno (amministratore) ed esegue. Il concetto di account legato ad un'istanza di User all'interno del social è realizzato solamente da meccanismi di registrazione utente con relativa password, esterni al social stesso.

Nello svolgimento del progetto ho rispettato al massimo il principio di incapsulamento: lo stato del social è modificato esclusivamente attraverso l'interfaccia SocialNetworkInterface messa a disposizione dalla libreria. Ogni operazione passa attraverso di essa, ciò permette alla classe che la implementa di effettuare controlli in relazione al tipo di operazione, se essa è permessa o meno, e di evitare di condividere con l'esterno della classe puntatori a oggetti in essa contenuta, in modo che lo stato sia sempre un'istanza ben formata a prescindere dell'uso incauto che se ne faccia.

La classe SocialNetwork, per segnalare errori a chi la utilizza, sfrutta un insieme di eccezioni personalizzate, contenute nel package CustomException, quali: InvalidOperationException, PostNotFoudException, UserNotFoundException, UserAlreadyExistException. Tutte sottoclassi di WinsomeException. Ho voluto utilizzare un insieme di eccezioni personalizzate perché essendo un progetto a scopo didattico mi sono impegnato ad aggiungere varietà di scelte di implementazione dove possibile.

L'interazione Client Server

Il Server, rappresentato dalla classe principale chiamata ServerMain, rappresenta l'oggetto che svolge tutti i compiti descritti nella consegna del progetto e quindi accetta le connessioni dei Client, esegue le loro richieste, se ben formate, e fornisce ai Client stessi tutte le informazioni di cui hanno bisogno per la comunicazione con l'utente.

Il Client invece rappresenta le caratteristiche di "Thin Client", perché la classe di avvio del software CThinClientMain non implementa nessuna funzionalità se non quella dell'impostazione delle politiche di sistema e del caricamento della classe CClient attraverso l'RMIClassLoader. In seguito a queste due, lo svolgimento di ogni operazione supplementare, associata alla ricezione di input da riga di comando, verrà affidata all'oggetto CClient.

Esso si occupa di fare il parsing dei comandi ricevuti, di riconoscerli e di inviare e ricevere messaggi al Server in relazione a questi.

Un'interazione normale tra i due software presenta i seguenti passi:

- Il Client registra un account nel social con l'apposito comando, il Server riceve le credenziali di registrazione e si occupa della creazione di un User corrispondente all'interno del social e del salvataggio delle credenziali in un apposito file delle password. Notiamo che le password non solo salvate in chiaro ma al momento della registrazione viene generato un seed pseudocasuale che viene concatenato alla password. Il risultato di questa operazione viene digerito dalla funzione SHA256. È quindi salvata la corrispondenza nome utente, seed

generato e digest. Questo per evitare sia di memorizzare le password degli utenti in chiaro, sia per evitare che ad uguali password risultino digest corrispondenti.

- Ora il Client potrà eseguire il Login in cui verrà instaurata la connessione TCP che rimarrà attiva fino a quando il Client stesso non interromperà la connessione eseguendo un logout. Nel login vengono anche comunicati al Client indirizzo IP e porta della comunicazione Multicast per la ricezione di notifiche di avvenuto aggiornamento dei portafogli. Infine, sempre durante il login, il Client si registra al servizio di notifica SNotifyFollow che tramite RMI callback permetterà al Server di comunicare le modifiche della lista followers.
- Il Client a questo punto, connesso correttamente al Server, potrà inviare tutti i comandi messi a disposizione dal Client per l'interazione con il social e usufruire di tutte le sue potenzialità
- Infine, il Client invierà il comando di logout al Server e avverrà la chiusura della connessione TCP.

Gestione concorrenza: Client

Il Client, brevemente descritto sopra, non ha grandi precauzioni nei confronti di una esecuzione concorrente se non quelle presenti nell'implementazione del sistema di notifiche. A seguito del login il Client avvierà un thread di ricezione pacchetti UDP a cui verrà passata in input una lista di notifiche. Questa lista è implementata da una classe personalizzata CCustomDeque, sottoclasse di ConcurrentLinkedDeque, che è una struttura dati thread safe che permette la concorrenza. Ciò è necessario dato che la lista sarà acceduta dal thread appena nominato, dal Main Thread del Client che notifica l'utente rimuovendo elementi dalla lista, e da parte dell'esecuzione di metodi dell'oggetto CNotifyFollowImp, parte del meccanismo con cui il Server invia le notifiche follower al Client.

Gestione concorrenza: Server

Il Server è per necessità implementativa una applicazione multithread che gestisce multiple connessioni in entrata da parte di Client diversi, ed esegue contemporaneamente operazioni sullo stato del social come la serializzazione o l'aggiornamento dei portafogli. Per permettere la concorrenza, senza il presentarsi di errori, ho per prima cosa reso thread safe l'implementazione di SocialNetworkInterface.

L'oggetto SocialNetwork contiene una ReadWriteLock associata alla HashMap degli utenti presenti nel social. Questo perché ho valutato che le operazioni fatte su questa struttura dati saranno prevalentemente di lettura. Una lock di questo tipo permette tante operazioni di lettura contemporanee ma solo una operazione di scrittura contemporanea. E questo aumenta notevolmente le prestazioni rispetto all'utilizzo di un altro meccanismo.

Non sono presenti altre Lock esplicite del social, ma ogni oggetto da esso utilizzato è a sua volta thread safe: I Comment, PostToken e WalletTransaction sono oggetti immutabili, quindi, non presentano alcun problema di accesso concorrente. I Post, RewinPost e User sono thread safe perché ogni metodo che esegue una modifica o una lettura dello stato è sincronizzato presentando il tag synchronized. Infine, l'insieme di commenti del social e di post sono organizzati con l'ausilio di due ConcurrentHashMap, dal comportamento "weakly consistent". Se una modifica della collezione viene apportata, essa si riflette sulle viste create con il metodo values(), utilizzato nel progetto in casi in cui si voglia ottenere l'insieme di oggetti contenuti nella HashMap per una visita iterativa. Il ciclo "for each" basato su queste viste potrebbe o meno influire della nuova modifica; questo garantisce che non ci siano errori nel programma, in quanto al massimo una modifica contemporanea ad una lettura viene ignorata.

L'insieme di thread avviati dal Server comprende:

- ThreadMain: che avvia e detiene il controllo di tutti gli altri thread, non agisce in alcun modo sullo stato del social, ma si occupa di ricevere alcuni semplici comandi di debug da linea di comando, tra cui quello per terminare l'esecuzione.
- ServerCA: il "Client accepter" del Server si occupa di ricevere le connessioni TCP in ingresso, creare un eseguibile per ognuna di queste e inviarlo alla thread pool, non esegue alcuna operazione sullo stato del social.
- ServerRH: definisco così l'insieme di thread della thread pool che si occupa di ricevere le richieste del Client ed eseguirle. Ognuno di questi thread esegue operazioni sul social attraverso i metodi elencati nell'interfaccia. Infine, attraverso l'oggetto SNotifyFollowImp, di cui i metodi tutti synchronized, invia notifiche ai Client relative ai follower.
- ServerBK: il thread che si occupa di avviare il backup del social e della HashMap di registrazione. Per fare ciò esegue un blocco di codice, sincronizzato sulla HashMap stessa, in cui avviene la serializzazione in file json dello stato completo del social. Questo thread si occupa anche del ripristino dello stato all'avvio del Server. La HashMap è utilizzata anche dal servizio di registrazione, esportato attraverso il meccanismo RMI. Importante notare che per evitare errori a causa dell'accesso concorrente, viene sincronizzato sulla struttura dati stessa ogni blocco di codice che ne fa utilizzo.
- ServerWU: il thread che si occupa dell'aggiornamento dei portafogli associati agli utenti del social. Questo thread è il più semplice ed esegue un ciclo in cui, ad ogni intervallo di tempo fissato, chiama il metodo walletUpdate del social che aggiorna lo stato.

Descrizione classi

Package CustomException

Rappresentano l'insieme di eccezioni sollevate dal social, in relazione a richieste non accettate. Ognuna estende WinsomeException che a sua volta estende Exception.

Package SocialNetworkClasses

- **Comment**: un'istanza di Comment rappresenta logicamente un commento all'interno del social. È un oggetto immutabile che contiene id, autore, data, contenuto e post di riferimento del commento. Il social si occupa di associare ad ogni istanza un id differente: commenti appartenenti a social diversi non sono distinti univocamente dall'id.
- **Post**: un'istanza di Post rappresenta logicamente un post all'interno del social. Questa contiene id, autore, titolo, data, contenuto, insieme di upvote e downvote con relativi utenti, insieme di commenti associati. Il Post contiene anche un sistema di cache per memorizzare quali commenti o voti ha ricevuto dopo l'ultimo aggiornamento dei portafogli.
- **PostRewin**: un'istanza di PostRewin rappresenta logicamente un rewin di un post all'interno del social. Sottoclasse di Post contiene in aggiunta il riferimento al post di cui è rewin e il suo id. Sovrascrive la maggior parte dei metodi in modo tale che quando si vuole eseguire un'operazione di questo post la si esegue invece nel post di riferimento. L'implementazione di questa sottoclasse è dovuta alla necessità di assegnare un id diverso a post e rewin dello stesso post, in modo tale che ogni utente che esegue un rewin possa eliminare il proprio rewin dal suo blog senza eliminare il post vero e proprio e che quindi esista una differenza semantica tra le due operazioni.

- **PostToken**: oggetto immutabile utilizzato per rappresentare liste di post, associate ad un blog o ad un feed. Ogni istanza contiene id, autore, titolo, data del post originale e una variabile che specifica se questo post è un rewin o no.
- **WalletTransaction**: oggetto immutabile utilizzato per rappresentare una modifica di un portafoglio associato ad un utente. Ogni istanza presenta id, utente associato, data, descrizione e valore della transazione. La descrizione è un codice che contiene l'id del post a cui è associata la transazione e informazioni in relazione a come è stata generata: se deriva dalla creazione di un post oppure da una sua interazione attraverso commenti o voto positivo.
- **SocialNetwork**: Oggetto centrale per la rappresentazione del social. Implementa l'interfaccia SocialNetworkInterface. Qui sono presenti tre strutture dati: una HashMap per memorizzare User con associato nome univoco, una ConcurrentHashMap per memorizzare Post con associato id univoco e una ConcurrentHashMap per memorizzare commenti con associato id univoco. Questa classe presenta metodi per la creazione e l'eliminazione, se necessario, di nuovi utenti, post e commenti, metodi per l'ottenimento di blog e feed di ogni utente. Presenta i metodi per la serializzazione e deserializzazione e il metodo per avviare un aggiornamento dei portafogli.

Classi del Server

- **ServerMain**: classe di avvio del Server e di gestione dello stesso, si occupa di inizializzare le strutture dati, avviare l'insieme di thread utili alla gestione generale del social e di terminare correttamente l'esecuzione. Prende come argomento un file di configurazione contenente i parametri di input del Server stesso che comprendono numeri di porta delle varie connessioni, timer di backup e aggiornamento portafogli ecc.
- **ServerCA (ServerClientAccepter)**: implementazione di Runnable. Il task che esegue il thread "Client accepter" del Server che accetta le connessioni in entrata e crea un eseguibile inviato alla threadpool.
- **ServerRH (ServerRequestHandler)**: implementazione di Runnable. I task che vengono eseguiti dalla threadpool e che ricevono, comprendono, eseguono e rispondono ad ogni richiesta da parte del Client. Modificando lo stato del social e inviando notifiche ai Client in relazione ai follow.
- **ServerWU (ServerWalletUpdater)**: estende Thread. Il thread che si occupa di avviare l'aggiornamento dei portafogli all'interno del social. È implementato come Thread e non come eseguibile come i due precedenti solamente a scopo didattico e di pratica.
- **ServerBK (ServerBakcup)**: estende Thread. Il thread che si occupa di eseguire la serializzazione e la deserializzazione completa dello stato del social. Questo thread esegue un backup dello stato ogni intervallo di tempo impostato dal file di configurazione. Di default, è impostato per eseguire backup multipli dello stato del social. Ciò è necessario poiché in occasione di un malfunzionamento della macchina durante l'esecuzione di un backup, questo potrebbe corrompersi; Si cerca in questo modo di avere salvataggi precedenti funzionanti. All'avvio e alla chiusura il thread ripristina e salva lo stato nella cartella "StatoSocial0" contenuta nella cartella di salvataggio.
- **SHPsw (SHashPassword)**: un'istanza di SHPsw è utilizzata per memorizzare in un unico oggetto nome utente, seed generato al momento della registrazione e digest della password concatenata con il seed. Utilizzato nella fase di registrazione utente e login.
- **SHash**: classe che presenta solo metodi statici utile per eseguire la funzione di hashing usata nella fase di registrazione e login dei Client.

- **SRandomOrg**: classe che presenta solo metodi statici utile per connettersi al sito "Random.org" in due occasioni differenti:
    - Come richiesto nella consegna del progetto, il tasso di cambio randomico utilizzato per la conversione da valuta interna al Server a Bitcoin, in relazione al comando "wallet btc", è richiesto al sito tramite un messaggio HTTP. In caso il sito non sia disponibile (come accaduto diverse volte durante la creazione del progetto) il numero sarà generato in locale.
    - In relazione alla registrazione di un nuovo utente, la stringa rappresentante il seed casuale è richiesta anch'essa al sito "Random.org". Anche in questo caso se la richiesta non dovesse andare a buon fine la stringa sarà generata in locale con l'ausilio della funzione di hashing.
- **SNotifyFollowImp**: implementazione di SNotifyFollow. L'oggetto esportato con il meccanismo RMI per la registrazione al sistema di notifiche dei follower. È anche utilizzato dai thread ServerRH per inviare le notifiche attraverso i pacchetti Multicast UDP.
- **SRegisterServiceImp**: implementazione di SRegisterService. L'oggetto esportato con il meccanismo RMI per la registrazione al Server, una delle poche operazioni che non viene fatta attraverso la connessione TCP e precedente al login.

Classi del Client

- **CThinClientMain**: classe di avvio del Client che sfrutta il "Dynamic Class Loading" per creare un'istanza di CClient, scaricata da un URL preso come input.
- **Client**: implementazione di Runnable. Rappresenta il nucleo del Client, che si può utilizzare o creando un'istanza ed eseguendone il metodo "run" oppure attraverso il metodo stringParser, che riceve in ingresso i comandi descritti nella consegna del progetto e li esegue. Questa classe all'avvio esegue il parsing del file di configurazione, se dato in input, inizializza il thread di ricezione notifiche UDP, si registra al servizio di notifica follower del Server e instaura la connessione TCP al login.
- **CNotification**: oggetto immutabile che rappresenta logicamente una notifica generica del Client. Contiene il tipo della notifica, data, descrizione e azione. Dipendentemente dal campo tipo i campi descrizione e azione sono o meno utilizzati e assumono un significato diverso. Questa classe è pensata per rappresentare una notifica generica e non solo le due utilizzate attualmente dal Client:
    - **tipo = 0**: notifica follow. Nella descrizione è presente il nome dell'utente che ha generato la notifica, nella azione è codificato se questo utente ha iniziato a seguire (azione >= 0) o smesso di seguire (azione < 0) l'utente che ha eseguito l'accesso dal Client che riceve la notifica.
    - **tipo = 1**: notifica di aggiornamento portafogli. Questa ha descrizione e azione vuote e rappresenta l'avvenuto aggiornamento dei portafogli da parte del Server, che potrebbe rappresentare una modifica del portafoglio dell'utente che ha effettuato il login nel Client.
- **CCustomDeque**: estende ConcurrentLinkedDeque. Una semplice coda utilizzata per la memorizzazione sequenziale delle notifiche ricevute, unica peculiarità è che a seguito di un'aggiunta di una notifica di aggiornamento portafoglio, vengono rimosse tutte le altre notifiche dello stesso tipo presenti.
- **CPackReceiver**: implementazione di Runnable. Il task che rimane in attesa sulla connessione UDP Multicast di ricevere notifiche dal Server, e in caso di ricezione aggiorna la lista di notifiche del Client.

- **CNotifyFollowImp**: implementazione di CNotifyFollow. Oggetto esportato tramite il meccanismo RMI e utilizzato dal Server per la comunicazione al Client di notifiche follower. A seguito di questa chiamata viene aggiornata la lista delle notifiche.

Descrizione modalità di compilazione ed esecuzione:

NOTA: per la creazione del progetto mi sono basato su Java 8, così come per la compilazione e la creazione dei file jar. L'avvio del progetto con l'ausilio di versioni di Java più recenti potrebbe portare a errori in compilazione ed esecuzione per via di utilizzo di classi e meto di ormai deprecati.

Per la compilazione di tutto il codice sorgente, utilizzando il compilatore standard "javac", basta posizionarsi nella cartella generale del progetto (quella in cui è inclusa anche questa relazione), aprire il terminale e digitare il comando:

javac -d ./out -cp ./lib/gson-2.8.2.jar ./src/CustomException/.java ./src/SocialNetworkClasses/*.java ./src/*.java*

il comando compilerà tutti i file sorgente memorizzando i file .class nella cartella out e utilizzando la libreria "gson-282.jar" presente nella cartella "lib".

Per avviare il Server aprire il terminale nella cartella "jar" che contiene il file Server.jar e utilizzare il comando:

java -jar Server.jar ./configServer.txt

Questo comando avvierà il software con i parametri definiti nel file di configurazione configServer.txt presente nella stessa cartella.

Per avviare il Client aprire il terminale nella cartella "jar" che contiene il file Client.jar e utilizzare il comando:

java -jar Client.jar file:// ./configClient.txt

Questo comando avvierà il Client con i parametri definiti nel file di configurazione configClient.txt presente nella stessa cartella.

Con il Server sarà possibile interagire con i comandi:

- **help**, che stamperà a schermo i comandi accettati.
- **reg**, che stamperà lo stato attuale della lista di utenti registrati.
- **social**, che stamperà a schermo lo stato attuale del social.
- **update**, che aggiornerà i portafogli degli utenti nel social (non resetterà il timer per il prossimo aggiornamento).
- **stop**, che terminerà l'esecuzione dei software, attendendo però che tutti gli utenti connessi eseguano la disconnessione.

NOTA: questi comandi sono puramente a scopo di debug e da considerarsi non efficaci per una analisi attenta dello stato del Server, ma per visualizzare una sommaria descrizione dello stesso.

Con il Client sarà invece possibile interagire con tutti i comandi descritti nella consegna del progetto con in aggiunta i comandi:

- **help**, che stampa la lista dei comandi accettati e la loro sintassi.
- **close**, che terminerà l'esecuzione del software.

In risposta ad ogni comando il software stamperà l'esito del comando ricevuto dal server o generato dallo stesso client, e verranno stampate a schermo le notifiche ricevute durante il

periodo di attesa tra l'invio dell'ultimo comando e il precedente. Questo metodo di visualizzazione fa sì che eventuali notifiche ricevute durante la scrittura di un comando non interferiscano con quest'ultima.

Per avere uno stato del social già inizializzato con numerosi Utenti, post e commenti impostare la directory di salvataggio "stato1" dal file di configurazione del server. Lo stato è creato a partire di un dataset del videogioco "Pokemon", ogni utente ha come nome il nome di Pokemon e come password "password".

Esempi

Nome: Amoonguss	Password: password	Tags: "Grass","Poison","114","30"
Nome: Lunala	Password: password	Tags: "Ghost","Psychic","137","97"