WINSOME

Manneschi Leonardo 599933

26 giugno 2022

Indice

1 Introduzione al contesto

Il progetto WINSOME consiste in un social media ispirato a STEEMIT, Un social netword la cui caratteristica piu interessante è la ricompensa ad autori e curatori di post.

2 Files e directories

La cartella WINSOME contiene:

- **Out**, una sottodirectory che contiene tutti i file .class creati alla compilazione del codice.
- **Server_Client_Jar**, una sottodirectory contenente i file di configurazione e i jar per Server, Client e due file .sh per l'esecuzione di questi.
- **Src**, contenente tutte le classi contenute nel progetto e una cartella **My_Jar** con le librerie esterne utilizzate.
- inoltre troviamo due file .bash uno per l'avvio e compilazione(direttamente dai codici sorgenti) del server ed uno per il client

2.1 Librerie usate

Le uniche librerie utilizzate sono quelle per la serializzazione/deserializzazione di Jackson per i file .json

- jackson-core-2.9.7.jar
- jackson-databind-2.9.7.jar
- jackson-annotations-2.9.7.jar

Jackson è una suite di strumenti di elaborazione dati per Java (e la piattaforma JVM), tra cui la libreria JSON parser/generatore di streaming, la libreria di associazione dati (POJO-Plain old java object da e verso JSON) e moduli di formato dati aggiuntivi per elaborare i dati codificati in diversi format.

3 File di configurazione

Come da specifica, i parametri di input delle applicazioni devono essere letti automaticamente da appositi file di configurazione testuali.
Per cui è stato pensato di creare le classi **ClientConfig.java** e **ServerConfig.java**. Queste due classi, che hanno dei parametri standard, contengono tutte le informazioni necessarie per la corretta comunicazione tra server e client.

4 Lato server

Le principali componenti del server sono:

- **ServerMain.java** controlla il path del file di configurazione(se passato) e avvia il server.
- **Server.java** avvia thread che gestiscono : il salvataggio/recupero dati, l'update dei rewards per gli utenti interessati, la ricezione/invio di messaggi dai e per i client, e gestione di richieste da questi ultimi.
- **DTstructure.java** contenente metodi e dati usati per le funzionalita richieste.

4.1 ServerMain.java

ServerMain.java crea e avvia un'istanza di **Server.java**, dopo aver controllato se il path passato per le configurazioni sia corretto. Altrimenti procede a prendere le informazioni standard.

4.2 Server.java

Server.java, crea per prima cosa una classe che implementa un servizio remoto **ServerInterfaceImpl** per poi registrarla nel registry, tramite la bind. Solo dopo, tenta di recuperare informazioni dai file .json (se esistono). Infine fa partire 4 thread:

- **RewardHandler** che fa l'aggiornamento dei wallet per gli utenti autori e curatori dei post a cui sono stati aggiunti commenti e voti di recente.
- **BackUpHandler** che scrive i dati contenuti nelle strutture apposite nei file .json : usersBackUp.json, postsBackUp.json e rewardBackUp.json
- **ServerSelector** gestisce le connessioni TCP per ogni client che intende connettersi al server, tramite SELECTOR NIO, e usa un **ThreadPool** che avvia i task **ReqHandler** per gestire le richieste inviate dai client sulle connessioni TCP e l'invio del messaggio si risposta.
- **ShutDownHook** usato per gestire SIGINT.

4.3 Thread in Server.java

4.3.1 RewardHandler

Come si puo vedere dall'immagine, questo thread , ha una struttura piuttosto breve. Si crea un DatagramSocket e si itera fino alla ricezione di una interruzione, si crea un DatagramPacket (con un messaggio predefinito non presente nell'immagine) e dopo aver sfruttato i metodi **reward e BTCReward** della classe **DTstructure.java** che, rispettivamente, calcolano il reward per gli utenti e aggiornano il valore tasso di cambio dei bitcoin(simulato dal sito RANDOM.ORG) viene inviato il pacchetto in multicast su tutti i client connessi. Da sottolineare il fatto che il metodo **reward** sfrutta la struttura dati **modifies**, che verra brevemente descritta piu tardi, con una classe interna NewsOnPost che contiene il numero di voti positivi/negativi/commenti per ogni post e tutti gli utenti che ci hanno interagito.

```
public void run() {

    try {
        this.multicastGroup = InetAddress.getByName(mcAddress);
        if (!this.multicastGroup.isMulticastAddress()) { //check if the ip passed is correct
            throw new IllegalArgumentException();
        }
        try (DatagramSocket sock = new DatagramSocket()) { //used to send the message MC
            while (!Thread.currentThread().isInterrupted()) {
                //create the packet to send
                DatagramPacket dat = new DatagramPacket(
                        msg.getBytes(),
                        msg.length(),
                        this.multicastGroup,
                        this.mcPort
                );
                DTStructure.reward();
                DTStructure.BTCReward();
                //send message to clients
                sock.send(dat);
                Thread.sleep(this.interval);
            }
            DTStructure.reward();
            DTStructure.BTCReward();
        } catch (IOException | InterruptedException e) {
            //e.printStackTrace();
        }
    } catch (UnknownHostException e) {
        e.printStackTrace();
    }
}
```

Figura 1: RewardHandler run method

4.3.2 BackUpHandler

Anche per questo thread la struttura è abbastanza concisa. Viene subito avviato il metodo **createFile** che genera, se non esistono, i file **userBackUp.json** , **postBackUp.json** e **rewardBackUp.json**. In seguito si itera finche non viene ricevuta una interruzione e si esegue un'attesa attiva con la sleep per un certo tempo e col metodo **BackUp**(sempre fornito da DTsructure) si scrivono i dati(users, wallet , post e l'iterazione attuale del reward) negli appositi file. Per sviluppare le funzioni di questo thread sono state usate le librerie elencate precedentemente.

```java
public class BackUpHandler implements Runnable{
    private long bkupTimeOut;//timer used for sleep
    public BackUpHandler(long bkupTimeOut) { this.bkupTimeOut = bkupTimeOut; }


    @Override
    public void run() {
        DTStructure.createFile();
        while (!Thread.currentThread().isInterrupted()) {//until thread is interrupted, sleep and get BackUp
            try {
                Thread.sleep(bkupTimeOut);//sleep
            } catch (InterruptedException e) {
                return;
            }
            if (!Thread.currentThread().isInterrupted()) {//execute the writing of current data
                DTStructure.BackUp();
            }
        }
        DTStructure.BackUp();

    }
}
```

Figura 2: BackUpHandler

4.3.3 ServerSelector

```java
public void run() {
    try (ServerSocketChannel socketChannel = ServerSocketChannel.open() ) {//open socketChannel
        socketChannel.socket().bind(new InetSocketAddress(this.port)); //bind socketchannel to port passed by config
        //We set the socketChannel on to false blocking
        socketChannel.configureBlocking(false);
        //Open selector
        Selector sel = Selector.open();
        //then register on the selector the ACCEPT operation
        socketChannel.register(sel, SelectionKey.OP_ACCEPT);
        System.out.printf("Server: waiting for connection %d\n", this.port);
        //Creating CachedTrheadpool for handling threads
        workerPool = (ThreadPoolExecutor) Executors.newCachedThreadPool();
        while (!Thread.currentThread().isInterrupted()) {
            if (sel.select() == 0) {continue;}//blocking operation. it blocks until one channel is selected.
            Set<SelectionKey> selectedKeys = sel.selectedKeys(); //Set of keys corresponding to ready channels
            Iterator<SelectionKey> iter = selectedKeys.iterator();// Set's iterator
            while (iter.hasNext()) {
                SelectionKey key = iter.next();
                iter.remove(); //important to remove the element
                try {
                    if (key.isAcceptable()) {               // ACCEPTABLE
                        // Accept a new connection using a SocketChannel for the communication with the client that request it
                        ServerSocketChannel server = (ServerSocketChannel) key.channel(); //return the channel for witch the key was cr
                        SocketChannel client_Channel = server.accept();
                        client_Channel.configureBlocking(false);
                        System.out.println("Server: accepted new connection from client: " + client_Channel.getRemoteAddress());
                        System.out.printf("Server: number of open connection: %d\n", ++this.n_activeConnection);
                        this.registerRead(sel, client_Channel);//register a new connection
                    } else if (key.isReadable()) {this.readClientMessage(sel, key); // READABLE
                    } else if (key.isWritable()) {this.echoAnswer(sel, key);}// WRITABLE
                } catch (IOException e) {
                    this.n_activeConnection--;
                    key.channel().close();
                    key.cancel();
                }
            }
        }
```

Figura 3: ServerSelector Run method.

Questo thread, come descritto brevemente prima, si occupa della gestione delle connessioni principali (TCP) con i vari client. Per farlo viene utilizzato un Selector. In particolare, dopo l'apertura di quest'ultimo si itera finché, il thread non viene interrotto. All'interno di questo ciclo viene utilizzato un metodo di **attesa attiva select()** fino a che almeno un channel è pronto ad eseguire operazioni I/O. Dopodiché si itera sulle key selezionate che rappresentano i channel pronti alle operazioni I/O e si eseguono le operazioni per la gestione di read/write/accept di una connessione. Ritengo sia importante far notare che i channel sono settati come **NON BLOCCANTI** e che quindi alcuni metodi quali read o write possono non essere eseguiti completamente. Se la chiave viene ritenuta pronta e l'operazione era READ allora viene eseguito il metodo **readClientMessage** che recupera la channel e il bytebuffer posto come attachment antecedentemente e lo utilizza per leggere la richiesta. Altrimenti se l'operazione era di WRITE viene eseguito il methodo **echoAnswer** di cui vediamo l'immagine sotto.
Per la gestione delle varie richieste viene utilizzato invece un **newCachedThreadpool** che crea nuovi

thread quando arriva una richiesta di execute nuova, ma riutilizzerà i thread creati in precedenza se non sono ancora passati 60 secondi dal termine di uno di essi. Migliorano le prestazioni se vengono eseguiti task brevi (e per questo progetto ipotizzo task molto brevi e leggeri). I thread che non sono stati usati per sessanta secondi sono terminati e rimossi dalla cache. Altrimenti si sarebbe potuto implementare un **newFixedThreadpool**(ma il continuo utilizzo delle risorse anche se il pool è inattivo è stata una ragione per evitare questa scelta) o, ancora meglio il costruttore Thread Pool Executor.

```java
/**
 *
 * @param sel Selector
 * @param key  key used to take socketChannel and write the response message
 * @throws IOException
 */
private void echoAnswer(Selector sel, SelectionKey key) throws IOException {
    //Taking client's SocketChannel
     SocketChannel c_channel = (SocketChannel) key.channel();
    //retrieving attachment as string
    String req = (String) key.attachment();
    //create task to handle the request
    ReqHandler reqHandler = new ReqHandler(key, req);

    //use threadPool to solve the request and send the message to client
    this.workerPool.execute(reqHandler);
    //saving again the channel for next requests
    this.registerRead(sel, c_channel);
}
```

Figura 4: ServerSelector Echo method

4.3.4 Thread scatenati dal threadpool

```java
public class ReqHandler implements Runnable {
    //selectionKey used to get SocketChannel
    private SelectionKey key;
    //request to handle
    private String msg;
    public ReqHandler(SelectionKey key, String msg) {
        this.key = key;
        this.msg = msg;
    }

    @Override
    public void run() {
        String ans = DTStructure.handler(msg); //the static class will handle the request and send an answer
        System.out.println("Thread ricevuto : [" + this.msg + "] ... inviato: [" + ans + "]"); //TESTING
        // prepare the response
        ByteBuffer length = ByteBuffer.allocate(Integer.BYTES);
        length.putInt(ans.length());
        length.flip();
        ByteBuffer message = ByteBuffer.wrap(ans.getBytes());
        // toSend will be used as the last buffer to send to client
        ByteBuffer toSend = ByteBuffer
                .allocate(length.remaining() + message.remaining())
                .put(length)
                .put(message);
        toSend.flip();  // doing this will make it ready to be read
        length.clear();

        SocketChannel c_channel = (SocketChannel) key.channel();//taking the client channel to write on it

        try {
            c_channel.write(toSend);
            while (toSend.hasRemaining()) {
                c_channel.write(toSend);
            }
        } catch (IOException e) {
            //System.out.println("Exception in sending back response to client");
        }

    }
```

Figura 5: ReqHandler class

Come si evince dall'immagine della classe ReqHandler.java, viene passato alla chiamata del costruttore la richiesta e la key(da cui si prendera poi il channel per comunicare col client). Il metodo Run prende subito la risposta del metodo (offerto da DTstructure) **handler()** in formato stringa e dopo aver preparato un ByteBuffer con la lunghezza del messaggio e il messaggio stesso, tenta di scrivere il bytebuffer nel channel. Le richieste devono rispettare il formato della specifica del progetto (si veda il metodo PRINTUSAGE in Client.java). In caso di utenti/post non esistenti o errore di formato richiesta, viene inviata una risposta in formato stringa dal server. Il solo compito del client risulta quindi solo quello di stampare la risposta elaborata dal server

5 Gestione di concorrenza e strutture dati

Per la gestione dei dati in **DTstructure** vengono usate le classi:

- **User.java** che contiene tutte le informazioni su un utente.
- **Post.java** con tutti i commenti , voti e informazioni generali sul post.
- **Transaction.java** che rappresenta una transazione per il wallet di un utente.

$$ConcurrentHashMap < String, User > users$$

$$ConcurrentHashMap < String, User > users\_logged$$

$$ConcurrentHashMap < Integer, Post > post$$

$$ConcurrentHashMap < Integer, NewsOnPost > modifies$$

$$volatile < Integer > rewardIteration$$

Concurrenthashmap rappresenta una hashmap con piu lock interne.
In questa maniera piu thread possono lavorare su diversi elementi della struttura senza dover bloccare completamente l'intera struttura usando synchronized methods o lock esterne.
users e **post** sono soggetti al metodo **getBackUp** che, come abbiamo visto precedentemente, viene utilizzato all'avvio del server per rilevare i dati dai file .json presenti e anche al metodo **backUp** presente all'interno del thread che gestisce il salvataggio dei dati.
users_logged invece è soggetto a metodi per il login e logout di un utente e la gestione della concorrenza è tale e quale a users e post. Interessante è anche rewardIteration che viene gestito tramite un metodo synchronized per il recupero del valore e un secondo per l'incremento del valore.
users rappresenta gli utenti del social network associando al nome di ogni utente come stringa al riferimento **User** contenente tutti i dati rilevanti. Lo stesso dicesi per **user_logged** con l'unica differenza che rappresenta gli utenti che hanno effettuato una operazione di login con successo. **post** associa un valore intero (ID) ad ogni post in maniera univoca. Infine **modifies** contiene TUTTE le informazioni sulle modifiche recenti (rate positive/negativi e commenti) ai post esistenti. Infine **rewardIteration** rappresenta il numero di iterazioni alla quale il server è arrivato fino a quel momento.
Inoltre al fine di gestire la concorrenza anche per gli utenti e post, sono state create delle **ReentrantReadWriteLock** per lettura e scrittura dei dati e, nello specifico per i post, sono stati creati due metodi synchronized in DTstructure per l'incremento e il recupero del dato in maniera thread safe.

Vediamo qui un piccolo esempio del metodo **DeletePost** in DTstructure con uno snapshot del metodo **getAuthor** sull'oggetto Post p.

```
private static String deletePost(String[] args) {
    String response ;
    String username = args[2].trim();
    User user = DTStructure.findUser(username);
    int postID = Integer.parseInt(args[1]);
    if(user==null)return "< User not found";

    Post p = post.get(postID);
    if(p==null)return "< Post does not exist";
    if(p.getAuthor().equals(username)){ //REMOVE only if user is the owner
        if(user.removePost(postID)){
            post.remove(postID);
            response = "< Post removed correctly";
        }else{return "< Post not removed";}

    }else {return "< "+username+" is NOT the author of post ["+postID+"]";}
    return response;
}
```

Figura 6: DeletePost

(Viene mostrato il metodo scelto per semplicita dato che altri metodi possibili come removePost hanno la stessa struttura ma sono maggiormente complessi).

Come possiamo notare dopo aver recuperato il Post con il corretto ID si controlla se i nomi sono uguali usando il metodo **getAuthor** che sfrutta le readlock in Post p

```
public String getAuthor() {
    String response;
    readLock.lock();
    response = author;
    readLock.unlock();
    return response;
}
```

Figura 7: getAuthor in Post

Vediamo che viene bloccata e sbloccata la lock immediatamente dopo aver letto il dato. E cosi come in questo metodo, **in nessun'altro metodo viene acquisita la lock su piu istanze nello stesso momento**. In questa maniera non avremo problematiche di deadlock .

6 RMI

(Per descrivere al meglio questa componente del progetto si andra a descrivere anche il lato client).
Come descritto in precedenza, il server espone una interfaccia **serverInterfaceImpl** (senza esportazione per l'estensione UnicastRemoteObject).
I metodi fondamentali in ServerInterfaceImpl sono :

- **register** che viene usato quando vuole essere aggiunto un nuovo utente al server(come da specifica).
- **sendNotification** manda al Client interface associato ad un utente la lista di followers di quest'ultimo e ne fa l'aggiornamento grazie al metodo **update** presente in ClientInterfaceImpl. Viene eseguito anche per follow/unfollow di singoli utenti.
- **registerForCallBack** al corretto messaggio di "success login + ..." aggiunge al server il nuovo clientInterface di quest'ultimo utente appena loggato.
  Grazie ai metodi sopra elencati si puo ricavare molto facilmente la lista di followers di un utente.

Il client invece per prima cosa cerca di ottenere l'oggetto esportato dal server "serverInterface" e al termine del login, se risulta andato a termine con successo, esegue il metodo **registerForCallBack** dall'oggetto **ServerInterface** esportato dal server. Importante per ClientInterface invece è il metodo **update**, che aggiorna la lista di followers tenuta dall'implementazione.

7 Lato client

Per questo progetto la parte client è molto "light". Senza contare ClientInterface e la sua implementazione ci sono solo due componenti importanti.

- **ClientMain.java** Esattamente come ServerMain.java, ClientMain controlla e recupera il file di configurazione (se passato usa il path) e avvia il metodo **start()** della classe Client.java.
- **Client.java** Gestisce le richieste da inviare e le comunicazioni con il server in generale.

7.1 ClientMain.java

Crea e avvia un'istanza di **Client.java**, dopo aver controllato se il path passato per le configurazioni sia corretto e aver avviato un thread per la gestione di SIGINT **ShutdownHook**.

7.2 Client.java

Come detto prima, viene esportato come prima cosa un'oggetto remoto ServerInterface dal registry che verra utilizzato per metodi quali register, login e logout.
Le richieste, devono mantenere la struttura della specifica che comunque viene controllata dai metodi appropriati ad ogni richiesta. Ogni metodo manda al server un messaggio contenente la richiesta (e in alcuni casi anche il nome dell'utente settato dopo il login), la risposta e la stampa allo standard output. Alcuni metodi interessanti sono :

- **loginUser**
- **registerUser**
- **logoutUser**

Il metodo **loginUser**, dopo aver ricevuto una risposta di successo dal server, prende i parametri MCport e MCaddress passati sempre dalla risposta del server e, dopo essersi registrato per la callback, li passa al thread **rewardMsgReceiver** per ricevere messaggi di update sul completamento del metodo **reward** da parte del server in Multicast.
Il metodo **registerUser** utilizza, dopo aver controllato la struttura della richiesta passata da CLI, il metodo **register** dell'oggetto remoto **serverInterface**. **logoutUser** al contrario di loginUser interrompe il thread **rewardMsgReceiver** e avvia il metodo **unregisterCallBack** per togliere dalla lista di client in ServerInterfaceImpl il client che ha appena effettutato il logout.

7.3 Thread in Client.java

L'unico thread avviato da client e' il thread **rewardMsgReceiver** che riceve in multicast un messaggio prefissato dal server.

```
@Override
public void run() {

    try{
        this.welcomeGroup = InetAddress.getByName(mcAdd);//get the InetAddress

        if (!this.welcomeGroup.isMulticastAddress()) {//if not valid address
            throw new IllegalArgumentException();
        }
        try (MulticastSocket multicastWelcome = new MulticastSocket(this.mcport)) {//bind the multicast socket
            multicastWelcome.joinGroup(this.welcomeGroup); //join group
           while(!Thread.currentThread().isInterrupted()){
                DatagramPacket dat = new DatagramPacket(new byte[this.MSG_LENGTH], MSG_LENGTH);
                //receive the packet sent
                multicastWelcome.receive(dat);
                /* System.out.print("> ");
                System.out.printf("%s\n", new String(dat.getData(), dat.getOffset(), dat.getLength()));
                */
            }
        }
    }
    catch(IOException e){
        e.printStackTrace();
    }
}
```

Figura 8: rewardMsgReceiver run method

Avendo le informazioni sulla porta e address a cui connettersi(passate dalla risposta del server a login), si connette un socket multicast alla porta e si fa un join nel address Multicast passato. In seguito fino a che non viene interrotto il thread(nel metodo logout di client.java) si riceve il messaggio inviato dal thread rewardHandler.

8 Istruzioni

Per l'esecuzione del codice basta andare da terminale nella cartella WINSOME e da li si possono fare 2 cose:

- Se si vuole compilare e avviare direttamente i codici sorgenti allora basta digitare 'bash server.sh' per il server e 'bash client.sh' per il client. Si noti che vengono immediatamente creati file di backup e che per questo punto non sono stati volontariamente messi file di configurazione per osservare la corretta compilazione del codice nonostante la mancanza.

- Se altrimenti si vuole compilare i file .jar allora si puo andare nella cartella Server_Client_Jar e fare la medesima cosa. 'bash ServerJarRun.sh' per il server e 'bash ClientJarRun.sh' per i client. Da notare invece che qui sono presenti dei file di configurazione e che il pathname viene passato direttamente attraverso il file.sh