Relazione Progetto Laboratorio di Reti

Salvatore Guastella 598631

1 Introduzione

Il progetto è stato sviluppato seguendo le specifiche e le tecnologie che sono state fornite durante il corso. È stato utilizzato l'ambiente di sviluppo JetBrains IntelliJ IDEA 2021.3 e con Java Development Kit versione 8. Dettagli su compilazione ed esecuzione disponibili sul file README.pdf oppure alla sezione 7.

Librerie aggiuntive

Jackson `https://repo1.maven.org/maven2/com/fasterxml/jackson/core/`: per la conversione di oggetti in JSON (e viceversa) e la creazione dei messaggi di richiesta/risposta;

lombok `https://projectlombok.org/`: utilizzate prevalentemente le annotations @Getter @Setter;

Repository Github: `https://github.com/salvogs/progettoRETI`

2 Architettura generale

Il progetto è stato realizzato seguendo il paradigma Client-Server e lo scambio di informazioni avviene tramite l'invio di richieste/risposte su una connessione TCP, messaggi multicast e la tecnologia RMI. I messaggi sulla connessione TCP fra client e server sono codificati come stringhe in formato JSON che seguono un protocollo ben definito:

Ogni messaggio inviato dal client dovrà obbligatoriamente contenere il campo **request-type**, contenente una stringa indicante il tipo di richiesta che il server dovrà elaborare, e il campo **username**, contenente una stringa relativa all'username per il quale il client richiede il login o altre operazioni. Sia lato client che lato server si è cercato di mantenere il più possibile una coerenza tra comandi immessi dall'utente e metodi chiamati. Una volta che il server riceve una richiesta di un client, effettua una deserializzazione della stringa in un oggetto JsonNode e, assumendo che la richiesta abbia il formato prestabilito, legge il campo request-type e invoca il rispettivo metodo passando eventuali parametri aggiuntivi letti dalla richiesta. Una volta elaborata la richiesta, il server manda la risposta sempre in formato JSON con un campo **status-code**, contenente un numero intero che sta a indicare l'esito. I codici di risposta hanno una semantica strettamente legata agli status code del protocollo **HTTP**. In caso di successo il codice restituito sarà del tipo **2xx**; in caso di fallimento sarà un codice del tipo **4xx**, indicante l'errata sintassi o l'impossibilità nel soddisfarla (e.g. permessi mancanti). Insieme al codice di errore può essere presente anche un campo **message** contenente una breve descrizione dell'errore verificatosi.

3 Struttura del codice

Il codice è stato raggruppato in più packages al fine di organizzare in modo più semplice le varie classi in base alla loro funzione e alle dipendenze tra loro. Segue una lista con una breve descrizione di ogni file, le sezioni successive discuteranno nel dettaglio le varie implementazioni.

Il package principale è **com.salvo.winsome** che contiene due packages, due interfacce e una classe:

- **ConfigParser**: contiene i metodi per il parsing dei file di configurazione
- **RMIServerInterface**: contiene i metodi chiamati dai client per la registrazione a winsome e la registrazione di uno stub per l'aggiornamento dei followers.

- **RMIClientInterface**: contiene i metodi chiamati dal server (Callback) per notificare un nuovo follower/unfollower.
- **server**
  - MainServer: contiene il **main** del server e si occupa di leggere il file di configurazione, far partire il thread principale e gestire la terminazione.
  - RMIServer: implementa i metodi dell'interfaccia RMIServerInterface
  - WSServer: contiene il codice principale del server, che legge nuove richieste, manda le risposte e implementa tutte le funzionalità richieste e le strutture dati principali.
  - RequestHandler: contiene il codice necessario alla decodifica delle richieste e chiama i metodi di WSServer per la loro gestione.
  - RewardsHandler: contiene il codice per il calcolo periodico delle ricompense e l'invio di notifiche multicast ai client.
  - BackupHandler: contiene il codice per la scrittura periodica di un backup sul file system.
  - WSUser: un utente di winsome
  - WSPost: un post di winsome
  - WSTransation: una transazione relativa a un calcolo delle ricompense.
- **client**
  - MainClient: contiene il **main** del client e si occupa di leggere il file di configurazione, leggere i comandi dell'utente da CLI, invocare i metodi di WSClient per l'invio delle richieste e stampare gli esisti delle risposte.
  - RMIClient: implementa i metodi dell'interfaccia RMIClientInterface
  - WSClient: contiene i metodi per la formulazione delle richieste e la lettura/stampa delle risposte.
  - MulticastListener: codice necessario alla ricezione delle notifiche relative al calcolo delle ricompense.

4 Implementazione Server

4.1 File di configurazione

All'avvio deve essere passato come paramentro il path di un file di configurazione.
Il server **non prevede dei parametri di default** e quindi devono essere necessariamente specificati i seguenti parametri di configurazione:

TCPPORT porta tcp sulla quale il server si metterà in ascolto per nuove connessioni

MULTICAST indirizzo ip multicast per l'invio di notifiche al calcolo delle ricompense

MCASTPORT porta da comunicare ai client per ricevere correttamente le notifiche multicast

RMIREGPORT porta del registry RMI

RMISERVICENAME per la registrazione dello stub sul registry RMI

AUTHORPERCENTAGE percentuali di wincoin da assegnare all'autore del post

REWARDSPERIOD periodo calcolo ricompense (ms)

BACKUPPERIOD perido backup su file system (ms)

I valori ai parametri sono assegnati tramite la sintassi "parametro=valore". Vi è la possibilià di lasciare righe vuote e commentare inserendo a inizio riga il carattere '#'. Inoltre, il nome dei **parametri** non è case sensitive quindi, ad esempio, TCPPORT può essere scritto anche come tcpport.

4.2 Struttura generale

Il server è implementato come un singolo processo multi-threaded secondo lo schema master-worker. Una volta avviato il server, il metodo main della classe **MainServer**, legge il file di configurazione e istanzia un nuovo oggetto di tipo WSServer con il seguente costruttore:

```
public WSServer(int nWorkers,int tcpPort, int multicastPort, String multicastAddress,
    int regPort, String regServiceName, double authorPercentage, int rewardsPeriod,
    int backupPeriod);
```

Il costruttore si occupa di ripristinare un eventuale backup e inizializzare le varie strutture dati, variabili e oggetti necessari al corretto funzionamento del server.

Successivamente, il main, crea e avvia un nuovo Thread, che chiameremo **dispatcher**, il quale andrà ad eseguire il metodo run() implementato da WSServer.Il dispatcher, una volta avviato, crea un **threadpool di dimensione prefissata** [1] al quale far gestire le richieste e riuscire quindi a servire più client contemporaneamente. Il task passato al pool è un oggetto RequestHandler. In seguito crea e avvia un **thread per il backup** e un **thread per il calcolo delle ricompense** e l'invio delle notifiche multicast, gestiti dai rispettivi metodi run() delle classi BackupHandler e RewardsHandler, discussi nelle sezioni 4.8 e 4.9.

4.3 Gestione delle connessioni

Il server è stato realizzato con un comportamento non bloccante utilizzando la librerie Java NIO. Nel caso di una implementazione bloccante, ad esempio con Java IO, il server sarebbe stato costituito da almeno un thread, ognuno delegato alla gestione di una singola socket, **bloccato** in attesa che il client invii una richiesta sulla connessione; questo meccanismo porterebbe a un degrado delle performance quando il server è sotto carico e numerosi cambi di contesto. Con un'implementazione non bloccante un solo thread (o comunque un numero limitato di essi) si occupa di **gestire più connessioni** e gli altri thread si occupano solo di gestire le richieste/generare le risposte, sfruttando a pieno il multithreading.

All'avvio, il thread dispatcher, crea un **ServerSocketChannel** per accettare nuove connessioni TCP, un **SocketChannel** per ogni connessione attiva e un **Selector** per effettuare il multiplexing sulle connessioni.
Una volta che il metodo select() ritorna scorre tutte le key del selected-key set e, in base all'operazione pronta, su ciascuna chiave viene eseguita una **accept**, una **read** oppure una **write**. In caso di canale non leggibile o errori vari viene cancellata la key dal selector ed eseguito un disconnectionHandler che, tramite una struttura dati (hashUser), ricava l'eventuale username associato alla connessione, effettua il logout e chiude il channel.

Accettata una nuova connessione, il Channel del nuovo client viene **reso non bloccante** e registrato sul Selector per opeazioni di lettura (SelectionKey.OP_READ). Quando il channel è ritenuto pronto in lettura viene chiamato il metodo **readMessage** che legge dal channel, cambia l'interestOps della key associata ad esso e crea un nuovo task da far gestire a un thread del pool (vedi sezione successiva). Dato che il channel è non bloccante può capitare di dover effettuare più letture; invece di eseguire una *while(buffer.hasRemaining()) channel.read()* che farebbe perdere il vantaggio del Selector, è stato ritenuto opportuno eseguire più volte la readMessage che cambia l'interest set della key solo quando i buffer (in attachment alla chiave) non vengono completamente riempiti. Una volta chiamato il metodo **key.interestOps(0)** il channel **rimane registrato sul selector** ma il dispatcher non potrà mai essere risvegliato per operazioni pronte su di esso.
Quando il channel è ritenuto pronto in scrittura, viene chiamato il metodo **sendResponse** che, come per le letture, registrerà il channel in lettura solo dopo aver effettivamente scritto tutto il contenuto del ByteBuffer associato alla key.

4.4 Gestione delle richieste: RequestHandler

La classe RequestHandler rappresenta il *task* eseguito dai thread del pool. Convertita la richiesta testuale in un JsonNode legge il campo *request-type* ed esegue una chiamata a un metodo di WSServer passando ulteriori parametri letti dalla richiesta JSON. Se la conversione e l'elaborazione della richiesta

[1] Potrebbe essere sostituito da un threadpool costruito ad-hoc, non c'è un motivo particolare per il quale si è scelto il fixedthreadpool, sicuramente effettuare test con scenari e carico del server variabili porterebbero a una scelta più accurata.

è andata a buon fine, il metodo chiamato restituirà un JsonNode (potrebbe contenere eventuali codici e messaggi di errore). In caso contrario la risposta sarà del tipo **400: bad request** (in formato JSON). Il messaggio di risposta consiste in un ByteBuffer contenente la dimensione e la risposta in formato JSON. Il metodo privato **registerForResponse** si occupa di allocare quest'ultimo, di allegarlo alla key del client e di cambiare l'interestOps di quest'ultima in **SelectionKey.OP_WRITE** seguito da un selector.wakeup(), per 'svegliare' il dispatcher nel caso in cui fosse bloccato su una select().

4.5 Strutture dati

La classe WSServer mantiene in memoria diverse strutture dati durante l'esecuzione del server. Sono state utilizzate principalmente tre **ConcurrentHashMap** che contengono le informazioni necessarie per il corretto funzionamento di winsome. È stato scelto di utilizzare delle Map (invece ad esempio di List) perchè la memorizzazione e l'accesso sono sicuramente più efficienti nel caso di scenari di utilizzo con molti utenti, post, ecc. Sono dunque quattro le strutture dati principali:

- ConcurrentHashMap<String, WSUser> **registeredUsers**: gli utenti registrati a winsome;
- ConcurrentHashMap<Integer, WSPost> **posts**: tutti i post creati dagli utenti;
- ConcurrentHashMap<String, ArrayList<String>> **allTags**: corrispondenza Tag - Lista Utenti per risalire rapidamente agli utenti registrati con gli stessi tags;
- ConcurrentHashMap<Integer,String> **hashUser**: per risalire a quale utente si riferisce una connessione;

Sono inoltre presenti:

- Un AtomicInteger **idPostCounter**: per assegnare un id a un nuovo post
- Un int **idTransactionsCounter**: usato dal RewardsHandler per assegnare un id a una transazione;
- Un int **rewardsIteration**: usato dal RewardsHandler per tenere traccia dell'età dei post;
- Un double **lastExchangeRate**: necessario per mantenere una cache nel caso in cui RANDOM.ORG non fosse disponibile;
- Tre HashMap **newUpvotes**, **newDownvotes** e **newComments** per il calcolo delle ricompense.

Ulteriori dettagli riguardanti le singole Map e variabili sono discussi nelle sezioni successive.

4.6 Gestione concorrenza

L'accesso concorrente è garantito safe per quanto riguarda le ConcurrentHashMap discusse nella sezione precedente. Per l'accesso a oggetti di tipo WSUser e WSPost sono state utilizzate delle istanze di **ReentrantReadWriteLock**, dalle quali è stata ricavata una coppia di lock associate, una per le operazioni di lettura e una per le scritture:

```
1 private ReentrantReadWriteLock readWriteLock = new ReentrantReadWriteLock();
2 private Lock readLock = readWriteLock.readLock();
3 private Lock writeLock = readWriteLock.writeLock();
```

Prima di entrare nel dettaglio sul come vengono acquisite le lock è importante sottolineare che, in base agli argomenti visti durante il corso, si sarebbe potuto optare per l'utilizzo di 'classiche' ReentrantLock e Condition Variables o meccanismi di più alto livello come i monitor e i metodi synchronized. L'utilizzo delle ReentrantReadWriteLock è legato soprattutto al giusto compromesso fra prestazioni e semplicità di implementazione e al fatto che più thread possono acquisire la lock in lettura, riuscendo quindi ad accedere ad una risorsa in maniera concorrente.

Il pattern che si è cercato di seguire per gran parte delle operazioni è quello di:

1. Acquisire la writeLock se l'operazione necessita di modificare l'utente o il post, readLock altrimenti;

2. Mantenere la lock per l'intera esecuzione del metodo se l'operazione deve esclusivamente modificare dei flag o poco più; nel caso in cui si debbano scorrere strutture dati memorizzate dall'utente o dal post, viene effettuata una **copia** degli elementi.

In riferimento al punto 2 ecco un esempio tratto dalla **viewBlog**:

```
1   user.lockRead();
2   HashSet<Integer> blog = new HashSet<>(user.getBlog());
3   user.unlockRead();
4   for (Integer id : blog) {
5      WSPost p = posts.get(id);
6      if (p != null) {
7       .......
8       ....
9       ...
10      }
11  }
```

In questo caso viene effettuata una lock in lettura, copiato il blog, e rilasciata la lock. Scorrendo tutti i blog dell'HashSet copiato viene controllato che il post non sia stato eliminato cosa che può succedere nel caso in cui il post sia un rewin. È stato infatti assunto, come da specifica, che *un solo client alla volta può eseguire l'accesso con un determinato utente* e quindi situazioni in cui p sia == null non possono mai verificarsi se il post è dello stesso autore che ha eseguito la viewBlog. Quindi, a discapito di copiare delle entry del blog inutilmente, permettiamo ad altri thread di effettuare altre operazioni minimizzando la serializzazione.

Operazioni come follow/unfollowUser non necessitano di utilizzare read/write lock sull'utente **che effettua l'operazione** dato che la lista dei followed può essere modificata solo da se stesso (in riferimento all'assunzione sui login simultanei fatta in precedenza).

Osservazione su deletePost() Per assicurare che una rewin o una qualsiasi operazione di modifica del post come comment o rate falliscano nel caso in cui una deletePost sia in corso si è deciso di aggiungere il flag **deleted** alla classe WSPost. Quando l'autore vuole cancellare un proprio post viene acquisita una lock in scrittura su di esso, settato a *true* il flag deleted e rilasciata la lock.

Prendendo in considerazione il frammento di codice sottostante (riferito a rewin,rate ecc), se un altro utente (thread), durante l'esecuzione della delete, volesse ad esempio aggiungere un commento possono verificarsi i seguenti casi:
- il thread che esegue la comment riesce ad acquisire la lock e a portare a termine l'operazione;
- il thread che esegue la delete riesce a precedere la comment e a settare *post.deleted = true*.

A questo punto, nel secondo caso, se il thread della comment deve ancora eseguire la checkPost a riga 1 si accorge che il post non esiste e quindi fallisce, altrimenti, una volta riuscito ad acquisire la lock, si accorge che il post è stato cancellato, quindi sarebbe inutile, se non scorretto, portare a termine l'operazione.

```
1   WSPost post = checkPost(idPost,response);
2   if(post != null) {
3       post.lockWrite();
4       if(!checkDeleted(post,response) ... ...)
5       ...
6       ..
7       }
8       post.unlockWrite();
9   }
```

4.7 RMI e Callback

Come accennato nella sezione 2, le funzionalità RMI sono state implementate in **RMIServer.java**, classe che implementa l'interfaccia **RMIServerInterface.java** condivisa con il client. Una volta avviato il server, viene istanziato un oggetto RMIServer e, una volta esportato, lo stub viene pubblicato su un registry, associandogli il nome specificato sul file di configurazione. Seguendo la specifica sono quindi due i metodi RMI esposti dal server:

- public int **registerUser**(String username, String password, String[] tags) throws RemoteException: per permettere ai client di registrarsi a winsome;
- public int **registerForCallback**(RMIClientInterface client,String username) throws RemoteException : per permettere al server di memorizzare uno stub del client e mandargli così notifiche di un nuovo follow/unfollow

Osservazione In entrambi i casi si è deciso di escludere l'utilizzo di metodi synchronized dato che le operazioni sono garantite safe implicitamente dalle strutture dati e dai metodi utilizzati.

```
1    public RMIServer(ConcurrentHashMap<String,WSUser> registeredUsers,
2                     ConcurrentHashMap<String, ArrayList<String>> allTags,
3                     Lock allTagsWriteLock) {
```

Il costruttore della classe RMIServer prende tre parametri: le ConcurrentHashMap registeredUsers e allTags, per la memorizzazione di nuovi utenti e la writelock **allTagsWriteLock**, per permettere all'operazione di winsome **listUsers** di leggere senza inconsistenze le liste degli utenti registrati con lo stesso tag.
Per la registerUser, una volta creata un'istanza di WSUser, viene utilizzato il metodo *putIfAbsent* della ConcurrentHashMap che opera in maniera atomica e ritorna null se non esiste alcun utente registrato con lo stesso username. Quindi, se in contemporanea un altro thread RMI dovesse chiamare lo stesso metodo non ci sarebbero corse critiche.
Per la registerForCallback il metodo *setRemoteClient* utilizza la signature synchronized, quindi non possono verificarsi anche in questo caso corse critiche con thread che, ad esempio, mandano una notifica per un nuovo follower.

4.7.1 Callback

Il client, una volta registrato il riferimento all'oggetto remoto esportato (RMIClient), permette al server di chiamare i metodi **newFollow** e **newUnfollow**, al fine di mantenere aggiornata la struttura locale dei followers. Anche in questo caso potrebbe verificarsi una corsa critica se il server manda una o più notifiche mentre il client sta eseguendo l'operazione *listFollowers*. È un caso poco probabile però si è deciso comunque di rimediare utilizzando dei blocchi synchronized per accedere alla struttura dei followers.

4.8 Persistenza sul file system:

Come detto in precedenza, all'avvio del server viene ripristinato l'eventuale backup. Se il backup non è presente o non è valido tutte le strutture dati e variabili che ne fanno riferimento verranno inizializzate. Consiste in quattro file formato *json* memorizzati nella directory **backup**:

- **posts.json**
- **tags.json**
- **users.json**
- **variables.json**

I primi tre file sono rispettivamente i backups delle Maps posts, allTags e registeredUsers. L'ultimo file memorizza invece le variabili idPostCounter, idTransactionsCounter, rewardsIteration e lastExchangeRate.

Sia per la lettura che per la scrittura è stato utilizzato l'approccio **Data Binding** della libreria Jackson, che converte JSON da/verso oggetti e tipi Java.

Una volta avviato, il thread (BackupHandler), verifica l'esistenza dei file sopra citati e, nel caso non esistessero, li crea. Periodicamente, in base al campo BACKUPPERIOD del file di configurazione, viene chiamato il metodo pubblico di WSServer **performBackup()** che scriverà sui file il rispettivo contenuto.

4.9 Calcolo delle ricompense

Il thread delegato al calcolo delle ricompense (RewardsThread), come per il backup, periodicamente, in base al campo REWARDSPERIOD del file di configurazione, chiama il metodo privato **computeRewards** che si occupa di calcolare le ricompense sui post con nuove interazioni, aggiornare i wallets degli autori e dei curatori e notificare i client online (solo nel caso di effettive modifiche di almeno un wallet). La decandenza in base all'età del post viene calcolata sottraendo un counter all'interno di WSPost (inizializzato alla creazione del post = rewardsIteration) a rewardsIteration.

Nota su wallet btc Il servizio esterno random.org potrebbe non essere sempre disponibile; per questo motivo il server mantiene in memoria l'ultimo tasso di cambio restituito, in modo tale da garantire la disponibilità della wallet btc.

Osservazioni Invece di scorrere tutti i post verificando la presenza di nuove interazioni, si è preferito mantenere tre HashMap: **newUpvotes**, **newDownvotes** e **newComments**. Per operare su queste Maps in maniera thread-safe con i metodi ratePost e commentPost, il metodo computeRewards non fa una semplice get ma una **replaceAndGetNewUpvotes|Downvotes|Comments**: ecco l'implementazione della replaceAndGetNewUpvotes:

```
1    public HashMap<Integer, HashSet<String>> replaceAndGetNewUpvotes() {
2        synchronized (newUpvotes) {
3            HashMap<Integer, HashSet<String>> tmpUpvotes = newUpvotes;
4            newUpvotes = new HashMap<>();
5            return tmpUpvotes;
6        }
7    }
```

Come per i metodi ratePost e commentPost l'accesso alla Map avviene tramite dei blocchi synchronized. La particolarità della replaceAndGet è che sostituisce il riferimento all'oggetto newUpvotes con una nuova HashMap. Preso come riferimento il seguente frammento di WSServer.ratePost:

```
1    synchronized (newUpvotes) {
2        newUpvotes.putIfAbsent(idPost, new HashSet<>());
3        newUpvotes.get(idPost).add(username);
4    }
```

Supponiamo che il thread delle ricompense riesca ad accedere per primo al blocco synchronized, il thread che esegue la ratePost si metterà in attesa prima di segnalare un nuovo upvote/downvote. Una volta che il metodo replaceAndGetNewUpvotes ritorna, le righe 2 e 3 verrano eseguite in riferimento alla **nuova HashMap** e quindi senza intaccare il funzionamento del RewardsHandler.

4.10 Terminazione

Il thread main, una volta avviato il server, si mette in attesa di un eventuale input su stdin. Una volta letta la stringa "stop"[2] esegue il metodo **stop** della classe server ed effettua la join del thread dispatcher. La terminazione consiste nel settare un flag a true e invocare una *wakeup()* sulla select, nel caso in cui il thread dispatcher fosse 'bloccato' su quest'ultima e nel (in ordine): chiudere il selector, terminare il thread pool con una **shutdown**, in modo tale da far terminare eventuali richieste lasciando il server in uno stato consistente, terminare i thread delle ricompense e del backup (che effettueranno un'ultimo calcolo e un'ultimo backup prima di terminare).

5 Client

Il client permette agli utenti di winsome di interagire con esso, permettendogli di interfacciarsi con il server tramite un'interfaccia a riga di comando. Il processo client è composto dal thread **main** e dal thread **multicast**, che viene attivato al login di un utente e terminato al logout.

[2] non case sensitive

5.1 File di configurazione

Come per il server, all'avvio del client deve essere passato il path di un file di configurazione e non sono previsti parametri di default.

SERVER indirizzo ip del server

TCPPORT porta tcp del server

RMIREGHOST indirizzo ip del registry RMI

RMIREGPORT porta del registry RMI

RMISERVICENAME per il loockup dello stub sul registry RMI

5.2 Struttura generale

Si è cercato di attenersi il più possibile ai comandi descritti sulla specifica aggiungendo il comando **help** per visualizzare la lista dei comandi accettati dal client e i comandi **dnd on|off** per disabilitare/abilitare la stampa delle notifiche (abilitata di default). Dopo aver letto correttamente il file di configurazione, il client, tenta di stabilire una connessione TCP con il server e di effetturare un lookup del regisrty RMI per recuperare lo stub messo a disposizione dal server. Nel caso di fallimento di almeno una delle due operazioni vengono fatti altri tentativi (max 10) a distanza di un secondo.

6 I messaggi verso il Server

Alla ricezione di un comando da stdin, implementato utilizzando un BufferedReader, in caso di matching con un comando supportato dal client viene eseguito un metodo della classe WSClient. Una volta generato, il JsonNode relativo alla richiesta viene convertito in una String e inviato sul SocketChannel della connessione TCP. Successivamente il client si mette in attesa della risposta. In caso di errori il client viene terminato con un messaggio di errore, sarà poi il server a gestire un eventuale logout. In caso di ricezione di uno status-code diverso dal tipo **2xx** viene stampato il codice di errore allegato alla risposta sul campo JSON **message**.

Note particolari sulla login La risposta alla login contiene ulteriori informazioni oltre a uno status-code:

- Username, per tenere traccia che quel determinato client ha effettuato il login, in modo tale da poter eseguire (correttamente) tutti gli altri comandi;
- La lista dei followers dell'utente loggato, per aggiornare la struttura dati locale;
- I parametri per mettersi in ascolto sulla MulticastSocket.

Una volta recuperate queste informazioni viene creato e avviato il thread multicast (classe MulticastListener) che si occupa di: creare una MulticastSocket, effettuare la join al gruppo multicast e mettersi in attesa di eventuali notifiche relative al calcolo delle ricompense. Viene inoltre creato un oggetto RMIClient e, tramite lo stub RMI del server, memorizzato un suo riferimento remoto. Il thread multicast sarà poi interrotto una volta eseguito il comando logout, insieme alla 'pulizia' della lista dei followers e all'username di login.

7 Compilazione e Esecuzione

I comandi seguenti non sono stati utilizzati durante lo sviluppo ma solo in fase di consegna, perchè l'utilizzo di un IDE non ha reso necessaria la compilazione/esecuzione da Command Line.

Prerequisiti

Il progetto è stato compilato e testato sia su Windows che su Unix, in entrambi i casi era installata un JDK versione 1.8.0_321 (Java 8). Tutti i comandi seguenti devono essere eseguiti dalla directory principale del progetto.

Note aggiuntive

Attenzione: fare copia incolla sul terminale di comandi su più righe potrebbe portare a errori; Cancellare la cartella backup o il suo contenuto se si vuole che il Server venga inizializzato all'esecuzione.

Compilazione

Il risultato della compilazione saranno dei file .class contenuti nella directory **bin**

```
javac -d bin -cp .\lib\* .\src\com\salvo\winsome\*.java
.\src\com\salvo\winsome\client\*.java .\src\com\salvo\winsome\server\*.java
```

Creazione archivio jar

La directory principale contiene gia' i file Server.jar e Client.jar ma nel caso si volessero creare nuovamente a seguito di modifiche al codice e' possibile lanciare i comandi:

```
cd bin  # vanno creati dalla directory bin

jar cvfm ../Server.jar ../META-INF/smanifest.mf com/salvo/winsome/*.class
com/salvo/winsome/server/*.class

jar cvfm ../Client.jar ../META-INF/cmanifest.mf com/salvo/winsome/*.class
com/salvo/winsome/client/*.class

cd ..
```

Esecuzione

Comandi per l'esecuzione del server e del client. In entrambi i casi possono essere terminati in maniera corretta con il comando 'stop'.

Server

```
java -cp "lib\*;bin" com.salvo.winsome.server.MainServer .\serverconfig.txt
```

oppure

```
java -jar Server.jar .\serverconfig.txt
```

Oltre all'argomento `config_path` e' possibile passare il parametro `n_workers` per personalizzare il numero thread del pool delle richieste

Client

```
java -cp ".\lib\*;.\bin" com.salvo.winsome.client.MainClient .\clientconfig.txt
```

oppure

```
java -jar Client.jar .\clientconfig.txt
```