Laboratorio di Reti

Progetto di Fine Corso A.A 2021/22

Oleksiy Nedobiychuk
Matricola: 597455 Corso B
o.nedobiychuk@studenti.unipi.it

Indice

1 Introduzione

Sono stato piacevolmente sorpreso da questo progetto. Il mio obiettivo principale è quello di lavorare nel ambito del web quindi posso dire che è stato un assaggio di quello che mi potrei aspettare. L'ho anche appezzato per fatto che mi ha invogliato a capire tecnologie da me sconosciute come ad esempio la RMI. Ho speso tanto tempo a ragionare sulle possibile soluzioni alternative , cercando sempre un bilanciamento tra difficoltà ed efficienza. Ogni scelta importante è spiegata in questa relazione al meglio delle mie capacità con qualche esempio e figura.

2 Architettura

Io ho realizzato questo progetto con l'obiettivo di creare un "sistema" efficiente; limitata dalla mia conoscenza di base, dalla difficoltà di realizzazione dell'idea, dalla gamma d'idee che ebbero la voglia di presentarsi nella mia testa quando cercai di trovare una soluzione a uno specifico problema e dalla mancanza di tempo. Io ho realizzato il software su una macchina `Ubuntu`, che usava: `java 11.0.13`. Per curiosità ho provato a compilare ed eseguire il codice con `java 8`, non funziono, e con `java 17` funziono correttamente. C'è un vincolo per far compilare il programma (se non erro) ed è quello d'inserire il certificato nel file soprannominato cacerts che si trova nella cartella di java poichè ho deciso di usare l'oggetto di tipo `SslRMIClientSocketFactory` e di tipo `SslRMIClientClientFactory` per la craezione del registry siccome che la comunicazione è cifrata per maggiore chiarezza vedere la sezione **Manuale**.
Illustrerò l'architettura del sistema e il suo funzionamento attraverso il linguaggio di modellazione UML e il linguaggio naturale. Premesso che non sono pratico nel formalismo del primo quindi mi vorrei scusare in anticipo di eventuali imprecisioni e sciocchezze. Il primo diagramma che presento è quello di dislocazione che è piuttosto semplice e non ha bisogno di spiegazioni.

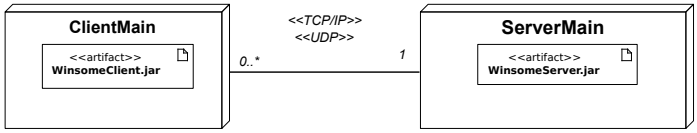

Figura 1: Diagramma di Dislocazione

La seconda figura lo creata per fornire una vista logica delle classi più importanti e le varie interazioni tra di essi. Io avevo cercato di raggruppare funzionalità simili nelle stesse classi per avere la coesione funzionale di conseguenza ho separato in base all'analisi delle richieste, esecuzione dell'ultime, risposte e registrazione dei clienti. In breve; il cliente invia una richiesta nel formato che rispetta il protocollo HTTP/1.1. I metodi supportati sono GET, PUT, DELETE e POST. La classe WinsomeServer usa una `ServerSocketChannel` nella configurazione non bloccante in quanto ho assunto che la gestione dei clienti con questo aspetto è più efficiente rispetto alle altre soluzioni che io avessi potuto utilizzare, come ad esempio quelle bloccante oppure `ServerSocket` tradizionale. La gestione di una richiesta, valida o invalida, e la relativa risposta avviene in tre fasi:

1. **accettazione:** il server accetta la nuova connessione; registra il nuovo `SocketChannel` con l'operazione di `OP_READ` e si mette in ascolto per canali pronti. Nell'istante in cui trova un cliente pronto per la lettura cancella la chiave di tipo `SelectionKey` affiliata al canale, crea una istanza della classe `ReaderClientMessages` e chiama il metodo `execute` appartente ad un oggetto di tipo `ThreadPoolExecutor`.
2. **analisi-esecuzione-costruzione della risposta:** Il thread analizza gli *header* ed estrae le informazioni necessarie per chiamare il metodo statico correlato alla richiesta. Costruisce la risposta HTTP, infine inserisce in una coda bloccante un involucro che racchiude il socket, la risposta, l'operazione `OP_WRITE` per far registrare al thread principale il canale. In caso in cui il main thread sia in attesa e nessuno ha chiamato `wakeup` in precedenza, dopo l'ultima chiamata alla `select`, invoca il metodo per risvegliarlo.
3. **scrittura:** Il main preleva il canale pronto per la scrittura, cancella la chiava ed infine chiama nuovamente il metodo `execute`. Il thread assegnato alla gestione recupera l'allegato e scrive la risposta.

Io ho deciso di usare il protocollo HTTP per la comunicazione tra client-server per la sua semplicità, permette di inviare richieste al server con vari programmi oltre al mio cliente come ad esempio il browser ed infine poiché fu studiato durante il corso. Io ho optato di sfruttare il sistema operativo in particolare il file system per gestire i profili, in sintesi: nel momento della registrazione si darà vita ad una cartella avente come nome quello dell'utente, nella directory `/src/Server/User_Data/Profiles` e per ogni tag non presente nel sistema, si creerà una cartella nella directory `/src/Server/User_Data/Tags` e al suo interno si farà nascere un file json per salvare all'interno di un array JSON il nome dell'utente; in seguito per ogni cliente che manifesterà la stessa preferenza si aggiornerà il file json inserendo il nome del cliente. All'interno della cartella `Profiles/username` si creerà la cartella `Blog` che contenerà i collegamenti simbolici dei post pubblicati più quelli che sono stati condivisi, la cartella `Followers` accoglierà i collegamenti simbolici alle cartelle dei follower e viceversa nella cartella `Following` di ogni follower ci sarà presente il collegamento simbolico alla cartella dell'utente seguito. Si darà origine anche alla cartella `/Profiles/username/Posts` che racchiuderà i post (cartelle aventi come nome id del post) pubblicati dall'utente di riferimento e i post che saranno divulgati da user seguiti. All'interno di ogni post si darà vita a sottocartelle `Comments`, nel momento in cui gli user decideranno di commentare il

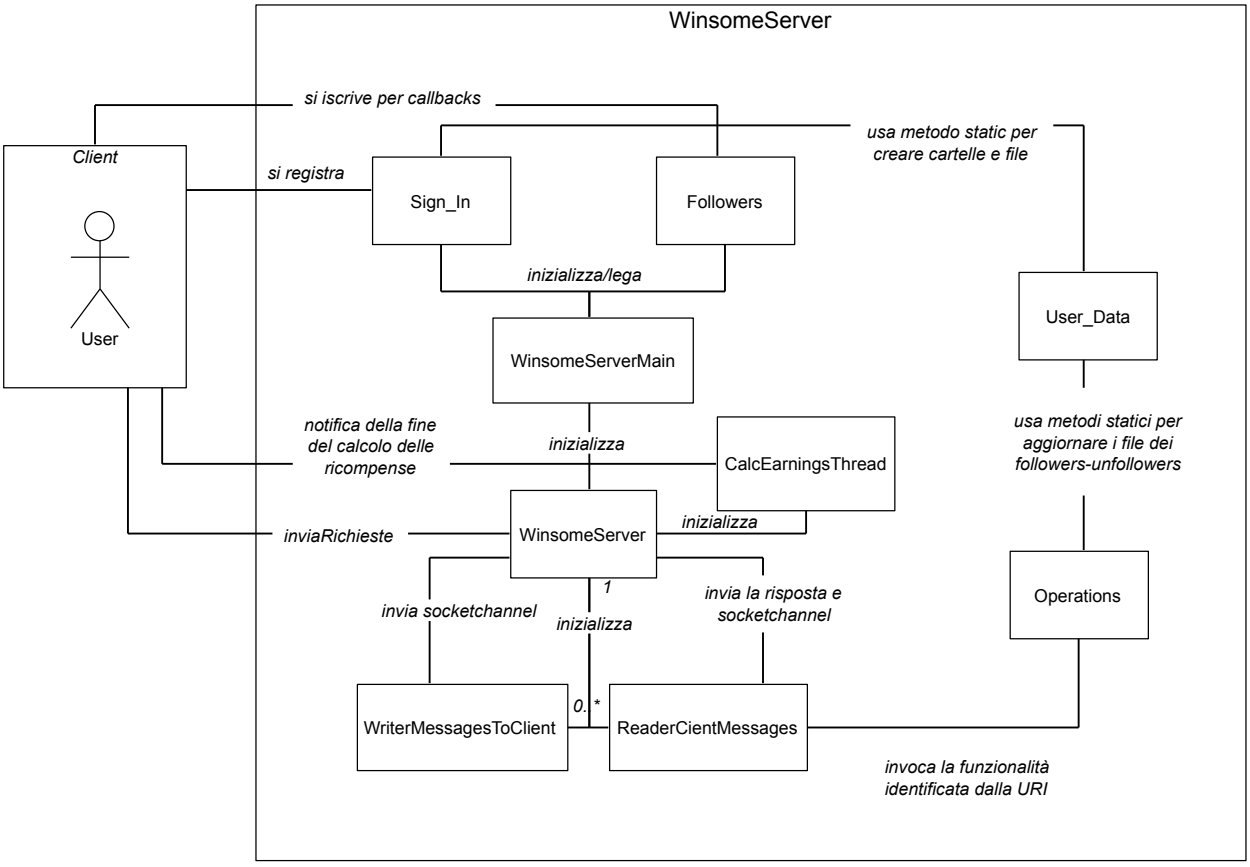

Figura 2: Vista logica del Server

post si creerà una cartella per ogni utente racchiudendo un file json per ogni commento dello stesso cliente, `Thumbs_up`

e `Thumbs_down` . Cerco di chiarire meglio con un esempio; un post pubblicato da Celentano che è stato commentato da Vasco per 3 volte, da Antonello per 2 volte, e ha ricevuto una reazione positiva da Lucio e una negativa da Vasco (stessa persona di prima) avrà la seguente struttura:

`/src/Server/User_Data/Profiles/Celentano/idPost/Thumbs_down/Vasco.json`

`/src/Server/User_Data/Profiles/Celentano/idPost/Thumbs_up/Lucio.json`

`/src/Server/User_Data/Profiles/Celentano/idPost/Comments/Vasco/idComment1.json`

`/src/Server/User_Data/Profiles/Celentano/idPost/Comments/Vasco/idComment2.json`

`/src/Server/User_Data/Profiles/Celentano/idPost/Comments/Vasco/idComment3.json`

`/src/Server/User_Data/Profiles/Celentano/idPost/Comments/Antonello/idComment1.json`

`/src/Server/User_Data/Profiles/Celentano/idPost/Comments/Antonello/idComment2.json`

nella cartella del singolo post sarà presente anche un file json con il titolo contenuto e altro, e un file chiamato stats.json (creato nel momento in cui il thread calcolerà il valore del posto per la prima volta) con dati necessarie per futuri calcoli. Nella cartella dell'utente ci sarà presente un file che conterrà l'informazioni sull'ultimo, file che rappresenterà il portafoglio, un file con nuovi follower (non notificati attraverso la callback) e un file con gli unfollower. La cartella nella directory `/src/Server/User_Data/Posts` racchiuderà tutti i post pubblicati da ciascun utente. In conclusione vorrei affermare che questa organizzazione, secondo il mio parere, ha permesso d'implementare le funzionalità richieste in maniera efficiente e semplice.

La procedura che verrà eseguita per calcolare la ricompensa spettata all'autore del post sarà la seguente: (si omette dettagli considerati non informativi) Si preleva il timestamp dell'ultimo calcolo dal file `stat.json` e si confronta con l'ultima data di modifica dei file presenti nelle cartelle `Thumbs_up` e `Thumbs_down` per recuperare il numero di reazioni positive e negative recenti. L'argomento per il secondo logaritmo viene ricavato confrontando il timestamp in precedenza recuperato con l'ultima data di modifica delle cartelle aventi come nome quello dell'autore del commento quindi si scelgono persone che hanno commentato ultimamente con il relativo numero di file presenti nella stessa cartella.

La terza figura rappresenta la vista logica delle classi più rilevanti del cliente. Per ogni persona che si registrerà, si creerà

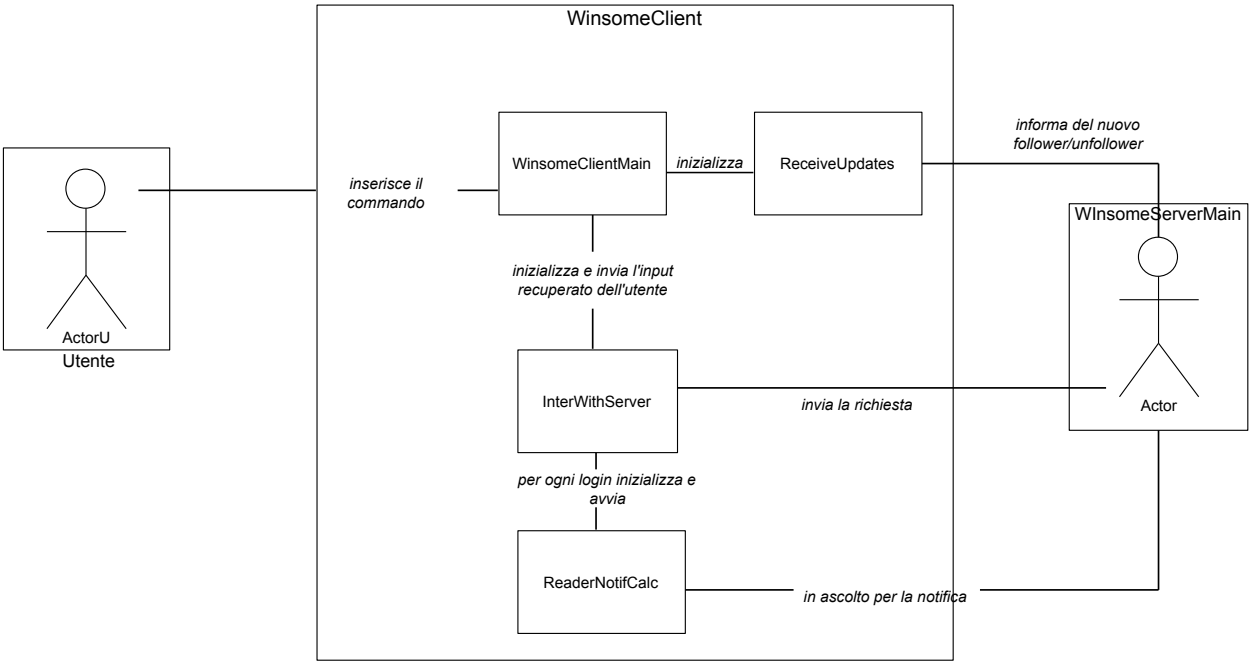

Figura 3: Vista logica del Cliente

una cartella nella directory `/src/Client/Users` con un file json che racchiuderà tutti gli utenti che decideranno di seguirla. Io ho implementato il server in modo che aggiorni il cliente nel momento in cui effettua il login circa le nuove persone che hanno cominciato a seguirlo o l'individuo che hanno smesso di seguire, quindi prima di effettuare il login il cliente preleva dal file json i follower e anteriormente all'invio del commando di uscita, sovrascrive il file json con i follower presenti a quel istante, pertanto il server non notifica il cliente circa tutti gli individui nell'attimo del login. Inoltre a ogni login il server invia al cliente i parametri necessarie per unirsi al gruppo di multi cast, successivamente il programma avvia il thread a parte che si mette in ascolto della notifica. Io ho deciso d'inviare i dati necessarie per aderire al gruppo nel momento del login e con il protocollo HTTP/1.1 siccome è la scelta più semplice che si presentò

in testa. Nella modellazione delle due viste mi sono ispirato alla figura presente alla pagina 45 dei lucidi del professore Paolo Ciancarini.

3 Concorrenza

Segue una breve descrizione di come tutti thread sono terminati, avviati. Il cliente attiva a ogni richiesta di login un thread che ha il compito di unirsi al gruppo per ascoltare le notifiche da parte del server. Il thread è terminato nel momento che lo stesso utente richiedere di uscire. Il server usa una *chahedThreaPool* per analizzare, eseguire le richieste infine rispondere ai clienti. Inoltre all'avvio del server si chiama il metodo `schedule` appartenente al oggetto di tipo `Task` che calcola le ricompense in base al periodo specificato, in un thread separato. Per maggiore chiarezza fornisco un diagramma di attività che descrive l'avvio del server, la terminazione dell'ultimo, e la gestione dei canali. Ulteriormente ho allegato un altro diagramma che descrive le azioni principali eseguiti nel momento in cui si richiede il login al cliente. Da notare che ogni fork corrisponde alla creazione di un thread o l'invio dell'oggetto che implementa `Runnable` alla *threadPool* Il server usa Set, Map come struture dati per memorizzare username, tag, utenti loggati, follower e i nomi delle funzionalità implementate. La sincronizzazione è realizzata attraverso gli oggetti di tipo `ConcurrentMap` : il primo memorizza tutti gli username, come chiave, specificati e come valore un oggetto di tipo `ReadWriteLock` , il secondo contiene i nomi di tag, sempre come chiave, dichiarati e come valore, stesso tipo del primo per sincronizzare la lettura e la scrittura di file presenti nelle cartelle dei tag. Il terzo ha tutti gli utenti attualmente connessi con il relativo identificatore di sessione. Infine il quarto contiene tutti gli user che si sono registrati per i callback e thread che calcola il guadagno usa una `HashMap` per memorizzare nomi e guadagni. Ogni funzionalità che esige solo la lettura dei dati di un utente generico, richiede all'inizio il lock di lettura invece quelli che modificano (creare post, aggiornamento del wallet, cancellazione del post) chiedono il lock di scrittura. Lo stesso discorso vale per la gestione dei file nelle cartelle dei tag. Per il passaggio d'involucri tra i tre thread che gestiscono la richiesta si usa una coda bloccante. Anche il cliente usa un Set sincronizzato per gestire i follower, esso è inizializzato attraverso la `ConcurrentMap` .

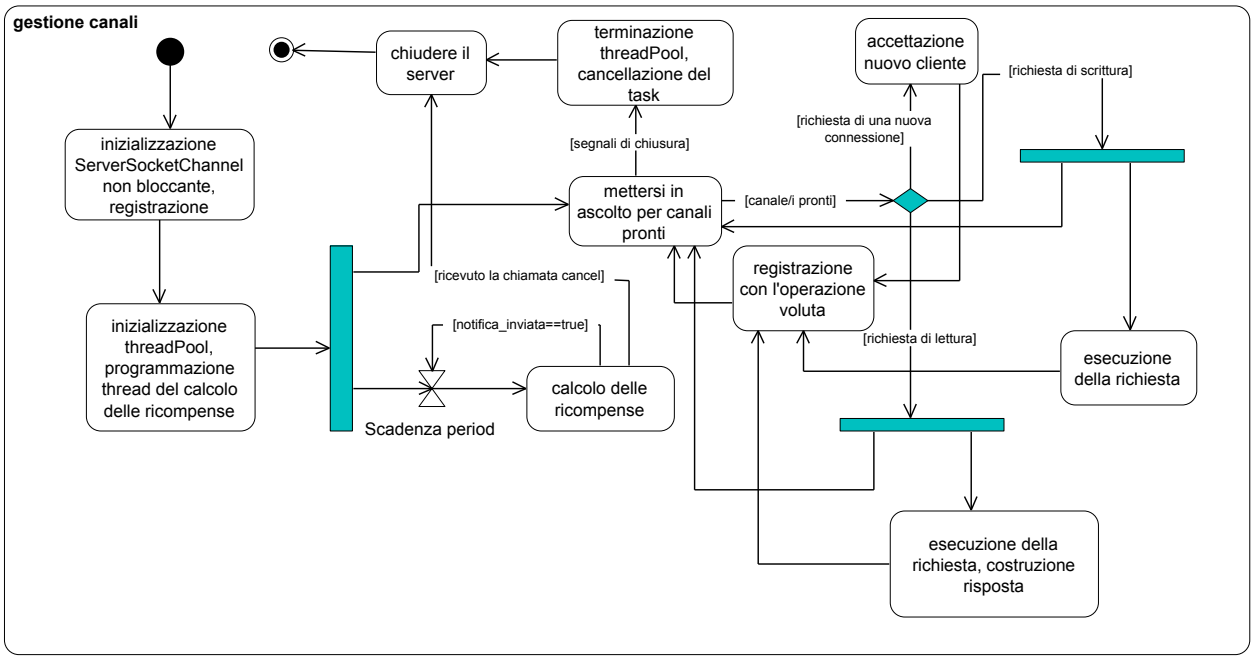

Figura 4: Diagramma di attività del server

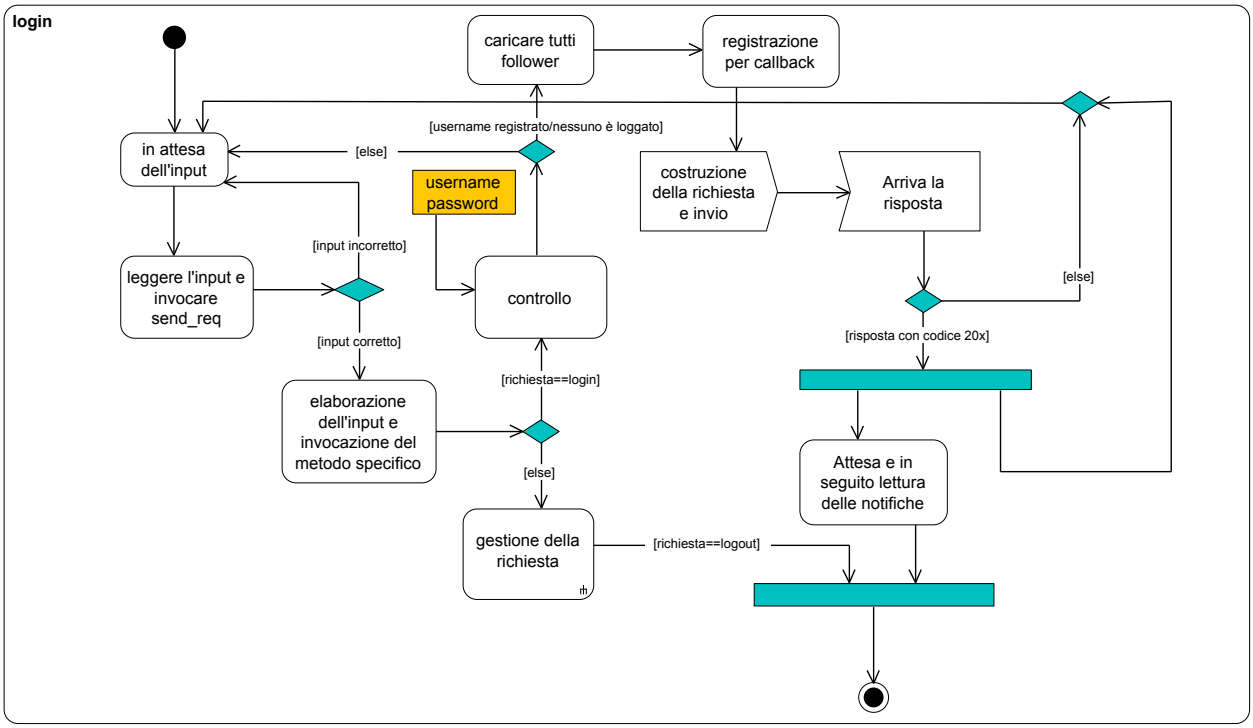

Figura 5: Diagramma di attività del cliente

4 Classi

Nelle sottosezioni do una descrizione sintetica delle classi più importanti.

4.1 WinsomeServerMain.java

Analizza il file di configurazione salvando i valori presenti nelle variabili; tutte le configurazioni accettate sono: *"SERVER", "TCPPORT", "UDPPORT", "MULTICAST", "MCASTPORT", "REGHOST", "REGPORT", "TIMEOUT", "GAINPERIOD", "BUFF_LIMIT", "REWARD_AUTHOR"*.
"GAINPERIOD": specifica il periodo che passa tra un calcolo e l'altro.
"BUFF_LIMIT": è usato come la dimensione massima del buffer che salva le richieste.
"REWARD_AUTHOR": è la percentuale che spetta all'autore del post.

4.2 User_Data.java

Contiene metodi static al supporto delle funzionalità richieste per modificare/aggiornare i file json.
Il metodo `add_user` creare le varie cartelle, file spiegate in precedenza, il metodo `create_addTags` aggiunge cartelle alla directory `Tags` con tag non presenti e aggiorna i file inserendo i nomi degli utenti che hanno specificato i tag già presenti nel sistema. Il metodo `setSettings_Server` dice dove si trova il file che contiene le chiavi per la comunicazione sicura con la relativa password, e il file che contiene i certificati di cui java si deve fidare (il mio certificato, si trova dentro questo file). Per vedere i certificati andare nella cartella `/src/Server/ssl`, aprire il terminal ed eseguire il commando: `keytool -list -keystore truststore.jks`, la password è password.

4.3 CalcEarningsThread.java

Estende la classe TimerTask fornita da java. Per ogni user e per ogni post pubblicato da esso recupera i dati salvati nel file `stats.json` che contiene i valori del numero di "mi piace", "non mi piace", commenti, numero d'iterazioni fatti sul post, e il timestamp dell'ultima computazione; calcola il guadagno, aggiorna il borsellino dell'utente che ha interagito con i post e comunica dell'avvenuto aggiornamento a tutti i membri del gruppo.

4.4 Operations.java

Contiene metodi statici per la realizzazione delle funzionalità richieste nel progetto.

4.5 ReaderClientMessages.java

La classe che legge i byte dal `InputStream`, e gli analizza; ho deciso di utilizzare parti della libreria *Apache http* per facilitare la scomposizione ed estrazione degli Header e la costruzione della risposta. Ho deciso di creare una struttura statica di tipo `Map` per contenente i metodi supportati (GET, PUT, DELETE, POST) come chiave e come valore un altra `Map` con chiavi il nome delle funzionalità supportate e come valori un oggetto di tipo `BiFunction`. Questa organizzazione permette di raggruppare insieme le funzionalità con lo stesso metodo e in modo efficiente capire la validità della richiesta: se la chiamata, `METHODS_OP.get(METHOD).get(OPERATION).apply(this, HttpRequest)` invoca l'eccezione allora la richiesta è malformata altrimenti si invoca un metodo privato dell'istanza di questa classe. C'è una corrispondenza biunivoca tra i metodi privati dell'ultima e i metodi statici della classe *Operations.java*.

4.6 WinsomeServer.java

Ha il compito principale di gestire i canali e registrarli con operazioni richieste da thread servitori.

4.7 WriterMessagesToClient.java

Scrive sul canale la risposta formattata in base al protocollo HTTP/1.1 e inserisce nella coda bloccante l'involucro.

4.8 Test

La classe `Sign_InTest.java` testa la correttezza dell'implementazione del processo di registrazione utente. La classe `OperationsTest.class`, come dice il nome, testa i metodi statici della classe `Operations.class` con vari input, anche incorretti. Il teste ha bisogno degli argomenti `-ea` `-Djunit.jupiter.execution.parallel.enabled=true` `-Djunit.jupiter.execution.parallel.mode.default=concurrent` per non andare in ciclo infinito.
La classe `ClientTest.class` infine testa il cliente, la comunicazione client-server, il server attraverso l'invocazione dell'intera gamma di funzionalità richieste. Nel momento dell'esecuzione si consiglia d'includere gli argomenti sopra indicati. Da notare che il test reindirizza lo standart input verso un oggetto di tipo `ByteArrayOutputStream` quindi per stampare sullo schermo usare il `System.err.print`.

5 Manuale

Per compilare il server, aprire il terminale nella directory `some/path/WINSOME` ed eseguire il commando:

```
javac -sourcepath src -d bin_s -cp '.:./libs/*' ./src/Server/*.java ./src/Server/notify_client/*.java
./src/Server/sign_in/*.java ./src/Server/user/*.java ./src/Server/utils/*java
./src/Server/winServ/*.java ./src/Client/rec_fol/*.java
```

Per avviarlo inserire:

```
java -cp '.:./libs/*:./bin_s' WinsomeServerMain
```

Per compilare il cliente:

```
javac -d bin_c -sourcepath src -cp '.:./libs/*' ./src/Client/rec_fol/*.java
./src/Client/serv_inter/*.java ./src/Client/utils/*.java ./src/Server/notify_client/*.java
./src/Server/sign_in/Sign_In_Interface.java ./src/Server/sign_in/TooManyTagsException.java
./src/Server/sign_in/UsernameAlreadyExistsException.java
```

Per avviarlo:

```
java -cp '.:./libs/*:./bin_c' WinsomeClientMain
```

Per far partire il **Server.jar**:

```
java -cp '.:./libs/*:Server.jar' WinsomeServerMain
```

Per far partire il **Client.jar**:

```
java -cp Client.jar WinsomeClientMain
```

Importante: In caso di eccezioni di tipo `NoSuchAlgorithmException` si deve importare i certificati del server e client,

creato da me nel truststore di java (o creare certificati nuovi e aggiornare keystore.jks e truststore.jks nella directory `/src/Client/ssl` e `/src/Server/ssl`, per maggiori dettagli: link)
Aprire il terminale nella cartella `/WINSOME/src/Server/ssl`, ed eseguire il commando:
`keytool -import -alias server.pem -file ca-cert -keystore $JAVA_HOME/lib/security/cacerts` la password è: changeit. Ripetere lo stesso passaggio per il certificato del cliente. Per concludere ho implementato il commando help per elencare tutte le funzionalità implementate.