Relazione progetto di reti a. a. 2021/2022 – Winsome

L'obiettivo del progetto era quello di creare Winsome, un social media che si ispira a Steemit. Nel seguito saranno analizzate l'architettura complessiva del server e del client, lo schema generale dei threads attivati, una breve descrizione delle classi definite, le istruzioni per la compilazione e l'esecuzione e infine verrà presentata una possibilità per effettuare il testing.

Architettura complessiva del Server

Durante l'inizializzazione del server viene recuperato il nome del file di configurazione, se specificato da riga di comando, successivamente viene eseguito il parsing del file, vengono inizializzate le strutture dati che conterranno lo stato del server, viene ripristinato lo stato del server dai due file json (uno per gli user e uno per i post), vengono inizializzati i servizi RMI (uno per la registrazione di nuovi utenti e uno per permettere al client di registrarsi per ricevere la callback della notifica di avere un nuovo follower o di averne perso uno), vengono inizializzati una serie di thread di supporto (che verranno discussi in seguito) e infine viene creato uno shutdown hook che permette la corretta terminazione del server.
Successivamente il server aprirà il channel socket per la connessione TCP, imposterà la modalità non bloccante e si metterà in attesa di connessioni da parte dei client.

Lo stato del server è salvato in due strutture dati diverse, una per gli utenti e una per i post. Entrambe le strutture dati sono delle Map che associano rispettivamente lo username all'utente e l'id al post. Per tenere traccia degli utenti che hanno effettuato il login c'è un'ulteriore Map che associa lo username alla corrispondente Socket.
Le Map sono implementate mediante delle ConcurrentHashMap che sono strutture thread-safe e quindi offrono operazioni di base per la gestione della concorrenza.
Per quanto riguarda la persistenza dello stato del server, gli utenti e i post vengono salvati in due file json distinti, come detto sopra. Serializzare e deserializzare le Map che contengono gli utenti e i post utilizzando direttamente i metodi offerti da Gson non offre scalabilità, in quanto molto facilmente si supererebbe la dimensione massima possibile per una stringa. Quindi per rendere queste due operazioni scalabili sono stati usati gli oggetti JsonWriter e JsonReader.
Attraverso JsonReader sono stati letti, dal file json, tutti gli attributi degli oggetti che possono crescere nel tempo (es.: Il singolo oggetto Post. Un post può crescere in upvotes, downvotes e commenti. Infatti queste liste vengono lette un elemento per volta), mentre per oggetti di dimensione fissa (es.: il singolo commento) è stato usato il metodo `fromJson()`, offerto dall'oggetto Gson, che permette di deserializzare il prossimo oggetto Json che dev'essere letto dal JsonReader (La differenza rispetto al metodo visto a lezione è proprio il passaggio dell'oggetto di tipo JsonReader al posto dell'oggetto Reader).
La stessa linea di principio è stata usata per la scrittura sul file json usando JsonWriter. Per poter scrivere un metodo generico per serializzare una Map su un file, è stata creata una interfaccia BufferedSerialization che definisce un metodo `toJsonFile` che viene implementato da tutti gli oggetti che devono essere serializzati. In questo modo non è stato necessario scrivere due volte lo stesso metodo sostituendo solo i parametri generici della Map.

Il server gestisce le connessioni mediante la combinazione di Java NIO e thread pool. Attraverso NIO si fa il multiplexing dei canali, in modo tale da eseguire i task solo quando il client è pronto per effettuare la richiesta o per ricevere la risposta. I task vengono passati al thread pool che si occuperà di eseguirli, rendendo possibile l'esecuzione in parallelo di più task da client diversi.
Questa combinazione permette di evitare la creazione di un thread per ogni client (permettendo quindi di non avere un grande overhead nel caso di molte connessioni in contemporanea) senza rinunciare a servire più client contemporaneamente, dato che utilizzando solamente NIO si sarebbe introdotto un ritardo nelle risposte in quanto le richieste sarebbero state servite una alla volta. Inoltre, così facendo si evita che i

thread siano occupati inutilmente in attesa di poter leggere/scrivere dal/sul canale del client.
Per implementare questa coesione tra thread pool e Java NIO, si usa un set thread-safe (ottenuto mediante un metodo statico della classe ConcurrentHashMap) che permette di segnalare i client che sono stati serviti e che sono pronti per essere registrati nuovamente sul selector. Questo set contiene oggetti della classe Registrable, che verrà illustrata nel seguito.
Il server permette di specificare, nel file di configurazione, il numero base di thread da attivare, il numero massimo di thread che possono essere attivati in supporto ai thread base, nel caso ci siano più task di quanti ne riescano a soddisfare, e un valore di keep alive, configurabile dal file di configurazione mediante il campo "KEEPALIVE", per far terminare i thread di supporto inutilizzati in modo tale da avere una maggiore gestione delle risorse.

Quando il selector recepisce che il canale di un client è pronto in lettura, crea un task ReaderWorker che si occuperà della lettura e della soddisfazione della richiesta. Mentre quando il canale del client è pronto per la scrittura creerà un task WriterWorker per inviare la risposta al client.
Il task ReaderWorker legge dal canale la richiesta e, dopo averla analizzata e aver verificato la bontà dei parametri, la esegue.

I messaggi di richiesta sono tutti del formato:

```
 ---------------------------------
|lunghezza richiesta | richiesta |
 ---------------------------------
```

dove `lunghezza richiesta` è un intero che indica la lunghezza della richiesta in byte mentre `richiesta` indica la richiesta vera e propria in byte. La richiesta è del tipo:

`<comando digitato sul client> <username dell'utente che effettua la richiesta>`

Le richieste di `register` e `list followers` sono un'eccezione, in quanto non sono richieste che il client invia al server mediante messaggi di richiesta. Anche le richieste `post` e `comment` hanno un'eccezione sul formato della richiesta in quanto aggiungono uno "`/`" per riuscire a separare i parametri che contengono dei doppi apici, lato server.
Un'altra richiesta, che però non fa parte di quelle derivate dai comandi digitati sul client, ma che rispetta il formato descritto sopra, è la `getFollowers` che è una richiesta inviata successivamente al login con successo per recuperare la lista dei followers dell'utente che ha appena effettuato il login.

I messaggi di risposta sono di due tipi:

```
 --------------
| status code |
 --------------
```

se la risposta non deve restituire ulteriori dati al client, oppure:

```
 ------------------------------------
| status code | lunghezza dati | dati |
 ------------------------------------
```

se la risposta deve restituire dei dati.

`Status code`**:** segnala l'esito della richiesta. Viene restituito 0 se la richiesta è stata servita correttamente, 1 se il client che effettua la richiesta non ha fatto il login, -1 se i parametri forniti non sono corretti, -2 se la richiesta non è valida. Mentre altri codici di errore sono restituiti in base al tipo di operazione. Questi codici sono documentati all'interno del codice, nella classe ReaderWorker.

`Lunghezza dati`: il numero di byte che compongono il campo dati.

Lo status code viene scritto nel buffer che si occuperà di inviare la risposta non appena viene servita la richiesta. Mentre la scrittura dei dati verrà fatta nel task WriterWorker. Questa scelta è legata al fatto che il buffer dev'essere necessariamente grande almeno 16 byte (sia lato server che lato client); quindi, lo status code sicuramente potrà essere immediatamente scritto sul buffer. Questo vincolo sulla dimensione del buffer serve per assicurare che ci sia sempre spazio per scrivere sul buffer almeno due interi e qualche byte. È fortemente consigliato avere byte almeno di qualche KB per garantire un minore scambio di messaggi sulla rete.
Non è necessario che i client che utilizzano l'API Winsome controllino la ricezione di status code `-1` o `-2`. Questi status code sono pensati per quei client che effettueranno richieste costruendosi i propri messaggi da inviare al server e quindi potrebbero inviare una richiesta non valida dal punto di vista dei parametri (-1) oppure una richiesta non valida con una funzionalità non esistente (-2).

Tutti i metodi che servono le richieste che modificano lo stato del server impostano un flag booleano per segnalare che lo stato del server è diverso da quello salvato nei file json. Il flag booleano è implementato mediante un AtomicBoolean per garantire la thread-safeness.

Durante l'operazione di login il server si assicura che l'utente non lo abbia già effettuato e che non lo effettui mentre lo sta loggando (questo secondo vincolo viene controllato mediante un blocco synchronized). Inoltre, una volta effettuato il login, il server invia al client i riferimenti per aggiungersi al gruppo multicast per ricevere le notifiche del calcolo delle ricompense. I riferimenti, ovvero indirizzo IP di multicast e la porta, vengono inviati mediante una stringa json. Il client a quel punto estrarrà l'indirizzo IP e la porta, verificherà che sia effettivamente un indirizzo di multicast e poi si aggiungerà al gruppo multicast.

Il formato della stringa json è:

```
{
 "multicastIP": "239.255.32.32",
 "mutlicastPort": 4444
}
```

Nel server non viene salvata la password in chiaro ma il digest ottenuto dall'applicazione della funzione hash SHA-256 sulla concatenazione dello username dell'utente e della password in chiaro. Questo permette di non avere una dispersione delle password nell'eventualità di attacchi mirati al sistema. La funzione hash viene applicata alla concatenazione dello username e della password per evitare che una stessa password utilizzata da utenti diversi generi lo stesso digest. Lo username è univoco all'interno del sistema, quindi, non è possibile avere due coppie con lo stesso username e la stessa password. Per verificare la password durante il login, si calcola il digest ottenuto dalla concatenazione di username e password inviati dal client e si confronta con quello salvato.
Durante la fase di registrazione e di login la password viene inviata in chiaro. Questo è un punto critico nella comunicazione che rende il sistema vulnerabile ad attacchi di tipo "Man-in-the-middle". Per avere una sessione di comunicazione sicura sarebbe stato meglio utilizzare una sessione TLS, magari facendo spedire direttamente dal client il digest.

Viene ritenuta necessaria sottolineare una scelta progettuale legata alla possibilità di seguire utenti. In linea generale un utente può seguire solo gli utenti di cui conosce lo username, quindi gli utenti risultanti dopo il comando `list users`. Un utente però, può venire a conoscenza dello username di altri utenti

leggendo i commenti di un post, leggendo lo username dell'autore di un post rewinnato o magari leggendo lo username di un utente direttamente nel contenuto di un post (es.: un utente che segue cita lo username di un altro utente all'interno del post). Quindi è stata lasciata piena libertà agli utenti di seguire anche persone con cui non hanno interessi in comune. Anche perché, se dovessero rendersi conto che i contenuti di quegli utenti non sono di loro gradimento, potrebbero sempre ricorrere al comando `unfollow`. Da qui si generano anche le casistiche in cui un utente potrebbe aver messo un upvote o un commento ad un post di un utente che non segue più. Questo è perfettamente tollerato dal sistema in quanto il non voler più seguire un utente non implica la rinnegazione dei propri pensieri sui contenuti precedentemente visionati, votati e commentati. Ovviamente il sistema dei commenti rispetta la specifica. Per essere commentato, il post deve apparire all'interno del feed dell'utente. Questo vuol dire che un utente può aggiungere un commento ad un post solo se segue l'autore oppure se uno degli utenti che segue ha fatto il rewin del post.
Come in tutti i social network più popolari, un utente non può seguire sé stesso. Conseguentemente non è abilitata nemmeno l'operazione di unfollow su sé stesso.
Si vuole sottolineare come la funzionalità di rewin sia stata progettata e implementata come una sorta di sponsorizzazione al post, in quanto non ne fa aumentare direttamente il guadagno. A supporto di questa tesi, nel feed di un utente compare specificato se il post è un rewin e da chi è stato rewinnato (quindi essenzialmente viene introdotto anche un meccanismo di "influencing". Un utente che apprezza particolarmente i contenuti di un altro sarà invogliato a leggere il post di un terzo utente, se questo è stato rewinnato da questo "influencer"). Quindi, seguendo questa linea, voti e commenti ad un post soggetto a rewin vengono assegnati direttamente al post originale (quindi il rewin fa aumentare il guadagno generato da un post indirettamente). Nel feed di un utente possono apparire più post uguali a causa del rewin, questo è un indicatore per l'utente della popolarità del post. Inoltre, un utente potrebbe trovarsi nel proprio feed un proprio post rewinnato da uno degli utenti che segue. Per concludere, all'eliminazione del post rewinnato segue la rimozione della sponsorizzazione.
Come da specifica, un utente non può votare un proprio post e non può votare due volte uno stesso post, nemmeno per cambiare il proprio voto.

Per quanto riguarda la gestione della concorrenza nella classe ReaderWorker, la maggior parte del lavoro viene affidato alla ConcurrentHashMap. Solo tre metodi hanno chiesto una maggiore attenzione perché combinavano inserimenti su entrambe le strutture e quindi una scorretta sincronizzazione poteva portare a inconsistenze nello stato salvato sui file json. I tre metodi in questione sono `createPost()`, `rewinPost()` e `deletePost()`.

Una volta che la richiesta del client è stata servita, che sia andata a buon fine o meno, viene segnalato che il client è pronto per essere registrato nuovamente sul selector per l'invio della risposta (quindi il client, quando sarà pronto, sfrutterà un task di tipo WriterWorker). Viene invocata la `wakeup()` sul selector per sbloccare il selector e registrare nuovamente il client.

Il task WriterWorker è molto più semplice rispetto al ReaderWorker, riceve la SelectionKey (per recuperare il SocketChannel e il ByteBuffer), i dati da scrivere (se presenti) e gli oggetti per permettere di segnalare la disponibilità del client nell'essere ri-registrato per inviare una nuova richiesta, sfruttando nuovamente un task di tipo ReaderWorker (gli oggetti che permettono ciò sono il Set readyToBeRegistered e il Selector che server per invocare la `wakeup()`).
Se il WriterWorker deve inviare solo lo status code, scriverà direttamente il contenuto del buffer sul channel, segnalerà che il client è pronto per essere ri-registrato e sveglierà il selector.
Se invece devono essere inviati anche dei dati, per prima cosa scriverà sul buffer la lunghezza del campo dati e successivamente invierà i dati in maniera sicura (che verrà spiegata subito). Da qui si comporterà come specificato prima.

Dato che server e client potrebbero avere buffer di dimensioni diverse e che le richieste, ma soprattutto, le risposte possano essere più grandi della dimensione del buffer, si effettuano letture e scritture dal/sul buffer in maniera sicura. Per quanto riguarda la scrittura sul buffer, si verifica che la richiesta/risposta entri tutta in una volta nel buffer e in quel caso viene inviato tutto insieme. Altrimenti, se dovessero essere più grandi della dimensione del buffer, questo viene riempito il più possibile, viene effettuata la scrittura sul channel e viene riempito nuovamente, ripetendo questi passaggi fino a quando non è stata scritta tutta la richiesta/risposta.
Durante la fase di lettura invece, disponendo della lunghezza della parte dati, si continuano a fare letture dal canale fino a quando non è stato letto il numero di byte sufficiente. Tutte queste letture frammentate vengono ricomposte (l'ordine e l'affidabilità della consegna sono garantiti dal protocollo TCP) e poi si procede con l'elaborazione.

Architettura complessiva del Client

Durante l'inizializzazione del client viene recuperato il nome del file di configurazione, se specificato da riga di comando, successivamente viene fatto il parsing del file e viene creato l'oggetto Winsome, l'API per del social network. Successivamente il client si prepara a ricevere i comandi digitati dall'utente.
Una volta ricevuto un comando, viene fatta un'analisi per verificare la bontà dei parametri prima di chiamare i metodi dell'API e in caso negativo vengono restituiti messaggi di errore significativi.
Oltre ai comandi richiesti dalla specifica, sono stati aggiunti i comandi `exit` e `help` che permettono rispettivamente di terminare il client e di mostrare una lista di tutti i comandi disponibili con i relativi parametri. Il comando exit invoca un metodo `close` che si occupa di terminare tutti i thread e i servizi di supporto avviati.
Il core del client è quindi spostato sull'API. Alla creazione dell'oggetto viene recuperato il registry e da lì si recupera lo stub per la registrazione di un nuovo utente e quello per la registrazione alle callback.
I vari metodi dell'API che implementano i comandi si limitano a controllare che esista una connessione verso il server e che l'utente sia loggato (ad eccezione del `login`), costruiscono il messaggio di richiesta, come precedentemente descritto, e lo inviano al server mediante il channel. Infine, si mettono in attesa di ricevere la risposta e analizzarla. Le risposte che contengono anche il campo dati vengono analizzate e "parsate" mediante la classe JsonParser che permette, attraverso i metodi opportuni, di restituire la stringa json sottoforma di JsonArray e JsonObject. Da questi oggetti vengono recuperate le varie entry del json.
Di maggior rilievo sono i metodi `login`, che si occupa di avviare la connessione TCP con il server, avviare un thread per ricevere i messaggi multicast, di inizializzare la lista in cui verranno inseriti i follower dell'utente loggato, esportare il servizio per notificare cambiamenti nei follower, registrarsi per la callback, e `logout` che esegue le operazioni duali. Quindi chiude la connessione TCP, interrompe il thread che si mette in attesa dei messaggi multicast, resetta la lista dei follower, rimuove la registrazione per la callback e "unexporta" il servizio per notificare cambiamenti nei follower.
La differenza è fatta dal metodo `register` che invocherà il metodo `register` dell'oggetto remoto e il metodo `listFollowers` che si limiterà a scrivere sullo standard output la lista che viene compilata attraverso le callback.

Schema generale dei threads attivati

Lato server sono stati attivati:

- Il thread Main che si occupa di fare il multiplexing dei canali.

- I core thread del thread pool per eseguire i task, come già precedentemente descritto. Essenzialmente i task ReaderWorker e WriterWorker sono una trasposizione, rispettivamente, del problema del produttore/consumatore. L'unica differenza è che il passaggio delle risorse dal produttore al consumatore è mediato dal selector che dà il via libera al consumatore per svolgere il proprio task. E poi il consumatore deve "riabilitare" il produttore a produrre, e anche in questo caso passa tutto dalla mediazione del selector. La risorsa passata è proprio l'oggetto Registrable. Eventualmente verranno creati anche altri thread di supporto in base alle possibilità e alla richiesta del thread pool.
- Il thread che esegue il task definito in RevenueCalcolator che si occupa di calcolare, ogni `calculationTime ms`, il guadagno generato da ogni post. Una volta calcolato il guadagno di tutti i post, viene inviata una notifica sul gruppo multicast.
  Questo task possiede il riferimento alla Map che contiene i post.
- Il thread che esegue il task definito in SaveState che si occupa di salvare lo stato del server, ogni `iterationTime ms`, sui due file json solo se lo stato del server è stato modificato dall'ultimo salvataggio. Questo viene controllato mediante la variabile booleana atomica citata precedentemente.
  Questo thread possiede il riferimento ad entrambe le Map che contengono i post e gli utenti. Per la gestione della concorrenza si acquisisce il recinto di mutua esclusione su entrambi gli oggetti in modo tale da consentire il corretto salvataggio dello stato del server.
- Il thread che esegue il task definito in AutomaticLogoutHandler che si occupa di rimuovere, ogni `checkTime ms`, dalla lista degli utenti loggati quelli che non sono più connessi e che quindi non hanno effettuato il logout.
  Questo thread possiede il riferimento alla Map che contiene gli utenti loggati. Non sono necessarie ulteriori accortezze per quanto riguarda la gestione della concorrenza in quanto la Map è una ConcurrentHashMap e gli iteratori su questa struttura dati sono "weakly consistent".
- Il thread che viene creato quando viene aggiunto lo ShutdownHook, nella classe ShutdownHandler, per la corretta terminazione del server. Questo thread, una volta ricevuto il segnale, si occupa di: chiudere il selector, interrompere il thread pool, interrompere tutti i thread precedentemente descritti e, se lo stato del server sui file json non è aggiornato con quello attuale, salva lo stato.
  Questo thread possiede il riferimento alle Map che contengono lo stato del server, più altri riferimenti che servono per la corretta terminazione (es.: il riferimento al thread pool, alla lista dei thread attivi, alla variabile atomica che stabilisce se lo stato del server è stato modificato dall'ultimo salvataggio). La lista activeThreads, una volta inizializzato il server, non potrà più ricevere inserimenti, quindi, non necessita di sincronizzazione in fase di chiusura di tutti i thread.
  Il salvataggio dello stato del server è sicuro in quanto si attende la terminazione del thread pool e tutti i thread di suporto, nel frattempo, saranno già stati terminati. Quindi durante l'ultimo salvataggio, lo stato del server non potrà mai essere modificato.

Lato client è stato creato un unico thread di supporto al thread Main. Il thread esegue il task NotifyHandler che si occupa di unirsi al gruppo multicast e aspettare la ricezione delle notifiche da parte del server. Una volta arrivata una notifica, viene stampata sullo standard output

Classi definite

Le classi Comment, Post, Transaction, User e Wallet sono le classi fondamentali per lo stato del server. Implementano tutte, tranne la classe Comment, l'interfaccia BufferedSerialization per permettere il salvataggio del proprio stato su un file mediante JsonWriter. Durante il salvataggio dello stato è bene

entrare nel recinto di mutua esclusione per quell'oggetto. Tutti i Set presenti nelle classi sono thread-safe, e sono ottenuti dal metodo statico `ConcurrentHashMap.newKeySet()`.

La classe User contiene vari Set che contengono le informazioni per i follower, i followed e il blog. I metodi per aggiungere valori a questi Set sfruttano la thread-safeness offerta da questa struttura dati. La sincronizzazione entra in gioco in quei metodi che devono restituire delle copie o devono salvare lo stato attuale dell'oggetto. Questi Set sono comunque "weakly consistent".

La classe Post contiene più elementi che richiedono una cura maggiore della sincronizzazione. Soprattutto quando dev'essere creata una copia di uno dei Set (che contengono le informazioni sui commenti, upvote, downvote, rewinner. In più sono presenti tre Set di supporto al calcolo delle ricompense che contengono upvote recenti, downvote recenti e commenti recenti). La sincronizzazione viene fatta sull'oggetto specifico da copiare, in modo tale da continuare a garantire altre possibili operazioni che non influenzano quella struttura. Da sottolineare sono i metodi `addUpvote()`, `addDownvote()` e `addComment()` in cui la sincronizzazione viene fatta non sulla variabile "persistente" (rispettivamente upvotes, downvotes e comments) ma sulle versioni aggiornate in base all'ultima valutazione del guadagno del post (rispettivamente recentUpvotes, recentDownvotes e recentCommenters). Questo viene fatto per garantire che il voto/commento "persistente" sia aggiunto solo quando può essere aggiunto anche al set che permette all'utente di essere considerato per il guadagno. Questa caratteristica è fondamentale in un contesto in cui gli upvotes, downvotes e commenti sono fonte di guadagno e i post "decadono" col tempo.

Nella classe Wallet la lista delle transazioni è implementata mediante un ArrayList, la gestione della concorrenza viene fatta manualmente mediante blocchi e metodi synchronized. Di rilievo è la sincronizzazione nel metodo `wincoinToBTC()` in cui è necessaria la sincronizzazione su tutto l'oggetto mentre si calcola la conversione. Per garantire la correttezza però, è necessaria anche la sincronizzazione del metodo `addWincoin()`.

Registrable è una classe che permette di salvare le informazioni fondamentali per servire un client. Contiene quattro attributi:

- clientChannel che identifica il socketChannel del client pronto per essere riregistrato
- operation che specifica l'operazione, di scrittura o di lettura (`OP_WRITE` o `OP_READ`), su cui il client dev'essere registrato
- byteBuffer che identifica il buffer associato a quel client
- jsonElement che identifica il campo dati da scrivere sul channel del client. Questo campo può essere null nel caso in cui il client debba essere registrato in lettura oppure nel caso in cui non ci siano dati da inviare al client

La classe CallbackHandler implementa i metodi per consentire ad un client di registrarsi ed eliminarsi dalle callback e i metodi che implementano le callback vere e proprie (`notifyNewFollower` e `notifyLostFollower`).
La classe NotifyNewFollowerService, legata strettamente a quella appena descritta, implementa i metodi per ricevere la callback e aggiungere/rimuovere i follower alla lista locale del client.

Le altre classi definite, come quelle che implementano dei task (es. AutomaticLogoutHandler, NotifyHandler, ReaderWorker, ecc.), sono state già ampiamente discusse con annessi riferimenti alla gestione della concorrenza precedentemente.

Compilazione ed esecuzione

L'unica libreria esterna utilizzata è la libreria Gson per la serializzazione e la deserializzazione degli oggetti. Per la compilazione è necessario avviare un terminale all'interno della directory del progetto ed inserire il comando:

```
javac -cp ".;./lib/gson-2.8.9.jar" *.java
```

I file di configurazione devono trovarsi nella stessa directory in cui si trovano i rispettivi file compilati (o i file JAR) e devono avere i seguenti nomi:

- "serverConfig.txt" – per il file di configurazione del server
- "clientConfig.txt" – per il file di configurazione del client

Altrimenti, per entrambi gli applicativi, è possibile passare mediante riga di comando il path di un file configurazione situato anche in una directory diversa da quella dell'applicativo, anche con un nome diverso da quello indicato sopra.
I parametri del file di configurazione del server sono i seguenti:

- `SERVER-IP` – l'indirizzo IP del server
- `TCP-PORT` – la porta utilizzata per il protocollo TCP
- `MULTICAST-IP` – l'indirizzo IP multicast per inviare le notifiche di calcolo delle ricompense
- `MCAST-PORT` – la porta utilizzata per il multicast
- `REGISTRY-PORT` – la porta utilizzata il registry che contiene gli stub
- `BUFFER-SIZE` – la dimensione del buffer (la dimensione dev'essere almeno di 16 byte)
- `RMI-REGISTER-SERVICE` – il nome che corrisponde al servizio di registrazione nel registry
- `POOL-SIZE` – il numero di core thread nel thread pool
- `MAX-POOL-SIZE` – il numero massimo di thread nel thread pool
- `KEEPALIVE` – il timer di keep alive dei thread di supporto generati dal thread pool
- `THREAD`POOL-TIMEOUT – il tempo di attesa massimo per la terminazione del thread pool
- `RMI-FOLLOWER-SERVICE` – il nome che corrisponde al servizio per registrarsi alle callback
- `REVENUE-TIME` – l'intervallo di tempo tra un calcolo delle ricompense e l'altro
- `AUTHOR-PERCENTAGE` – la percentuale di guadagno che deve ricevere l'autore per ogni post
- `BACKUP-TIME` – l'intervallo di tempo tra un salvataggio dello stato del server e l'altro
- `AUTOMATIC-LOGOUT` – l'intervallo di tempo tra un controllo dei client non più connessi da disconnettere e l'altro

Mentre i parametri del file di configurazione del client sono:

- `SERVER-IP` – l'indirizzo IP del server
- `TCP-PORT` – la porta utilizzata per il protocollo TCP
- `REGISTRY-HOST` – l'indirizzo IP nel quale è situato il registry
- `REGISTRY-PORT` – la porta utilizzata il registry che contiene gli stub
- `BUFFER-SIZE` – la dimensione del buffer (la dimensione dev'essere almeno di 16 byte)
- `RMI-REGISTER` – il nome che corrisponde al servizio di registrazione nel registry
- `RMI-CALLBACK` – il nome che corrisponde al servizio per registrarsi alle callback

Per l'esecuzione dei file compilati basterà utilizzare i comandi:

- `java -cp ".;./lib/gson-2.8.9.jar" ServerMain` – per eseguire il server
- `java -cp ".;./lib/gson-2.8.9.jar" ClientMain` – per eseguire il client

Mentre, per l'esecuzione dei file JAR basterà utilizzare:

- `java -jar ServerMain.jar` – per eseguire il server
- `java -jar ClientMain.jar` – per eseguire il client

Una volta avviato il server, può essere avviato il client. Il client successivamente sarà a disposizione dell'utente per eseguire tutti i comandi supportati dal sistema:

- `register <username> <password> <tags>` – dove <tags> è una lista di massimo cinque tag separati da uno spazio. Permette di registrare un nuovo utente su Winsome
- `login <username> <password>` – Permette di effettuare il login dell'utente specificato
- `logout` – Permette di effettuare il logout dell'utente attualmente loggato
- `list users` – Permette di mostrare la lista di tutti gli utenti che hanno almeno un tag in comune con l'utente loggato
- `list followers` – Permette di mostrare la lista di tutti gli utenti che seguono l'utente loggato
- `list following` – Permette di mostrare la lista di tutti gli utenti che l'utente loggato segue
- `follow <username>` – Permette di seguire un utente
- `unfollow <username>` – Permette di smettere di seguire un utente
- `blog` – Permette di mostrare la lista dei post creati dall'utente loggato
- `post <title> <content>` – Permette di creare un nuovo post
- `show feed` – Permette di mostrare la lista dei post creati dagli utenti seguiti dall'utente loggato
- `show post <id>` – Permette di mostrare il contenuto di un post
- `delete <idPost>` – Permette di eliminare un post se l'autore del post corrisponde con l'utente che ha fatto la richiesta
- `rate <idPost> <vote>` – Permette di assegnare un voto ad un post. Il voto può essere scelto tra "`+1`" e "`-1`"
- `comment <idPost> <comment>` – Permette di aggiungere un commento ad un post
- `wallet` – Permette di mostrare il portafoglio dell'utente loggato
- `wallet btc` – Permette di mostrare il portafoglio, convertito in BitCoin, dell'utente loggato
- `exit` – Permette di chiudere il servizio Winsome
- `help` – Permette di mostrare su schermo tutti i comandi disponibili

Testing

Il progetto viene consegnato con già qualche dato inserito all'interno del server, in modo da facilitare il testing con utenti già avviati. È possibile avere un'esperienza da zero eliminando il contenuto dei due file "users.json" e "posts.json".
Di seguito vengono elencate le credenziali per questi utenti:

- `Francesco DiLux`
- `Stefano Steff`
- `Fabio Faaf1234`
- `Tommaso 3579dm`
- `Vanessa 32Robo_C0p`
- `Lorenzo ArezzosCastle`
- `Romina romiromi`