# Progetto Winsome

DAVIDE PARRINI, CORSO A, 599858

## Organizzazione progetto

Il progetto è suddiviso in 3 package:

-server: dove sono implementate le classi per l'esecuzione del server,
una directory contente le strutture dati esclusive del server,
ed una directory filesystem dove sono contenuti i file json per salvare lo stato del server, informazioni riguardo ai post, utenti registrati, id corrente dell'ultimo post scritto, varie Map utilizzate dal server e il file configurazione relativo al server;

-client: dove sono implementate le classi per l'esecuzione del client e il file di configurazione relativo al client

-utils: dove sono implementate classi utili sia al client sia al server

## Descrizione generale

### Lato server

All'avvio del server, Winsome scandisce un file di configurazione e ricostruisce lo

stato del sistema tramite il metodo

```
initializateFileSystem();
```

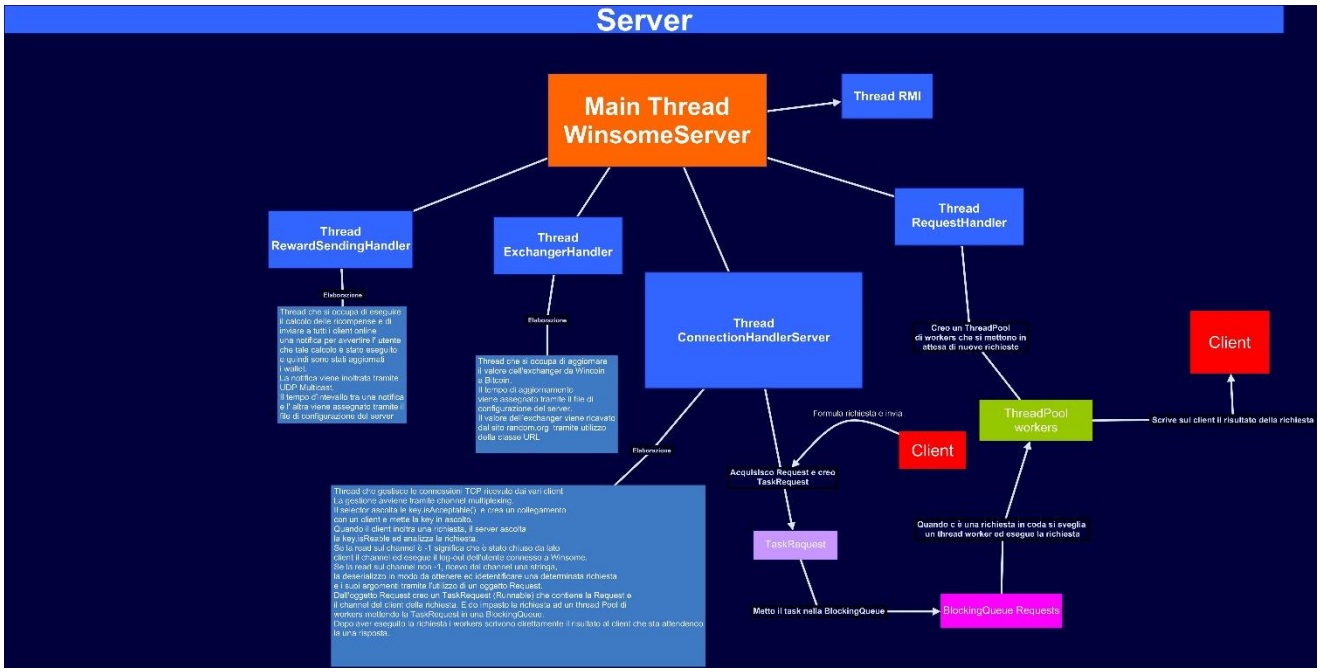
dove vengono recuperate tutte le informazioni delle strutture dati utili al server da dei file json, utilizzando l'ausilio della classe Serializator, dove ci sono i metodi per la deserializzazione di ogni struttura dati usando Gson.

Dopo aver recuperato lo stato, il server avvia una serie di thread :

- **Thread RMI**, non dichiarati esplicitamente, RMI di base attiva dei sui thread per essere eseguito
- **Thread RewardSendingHandler**, si occupa del calcolo delle ricompense analizzando post per post assegna ad ogni utente la sua ricompensa. Tra un calcolo delle ricompense e l'altro viene atteso un intervallo di tempo deciso tramite il file di configurazione. Ogni volta dopo il calcolo, viene mandata una notifica ai client online, comunicandogli che sono stati aggiornati i wallet. Il servizio di notifica deve essere implementato con UDP Multicast, quindi da lato server scrivo su un DataGramSocket il packet che voglio spedire a tutti.
- **Thread ExchangerHandler**, si occupa dell' aggiornamento del valore dell' exchanger da Wincoin a Bitcoin. Viene passato al file un intervallo di tempo di attesa tra un aggiornamento e l'altro (deciso nel file di configurazione). Il valore dell'exchanger viene recuperato dal sito Random.org tramite l'utilizzo della classe URL. La variabile del exchanger da aggiornare è presente nel server. Ho preferito far si che un thread aggiornasse tale valore invece di elaborare l'exchanger ogni qual volta viene utilizzato il

comando Wallet BTC,  per rendere piu verosimile la stabilità del valore ( evitando che un utente utilizzasse piu volte in successione il comando ed avere valori completamente sballati e non coerenti in Bitcoin )

- **Thread ConnectionHandlerServer**, thread che gestisce le connessioni TCP arrivate dai vari client. L'idea di massima per l'implementazione lato server è un server Channel Multiplexing (essendo un social network, ci si aspetta scalabilità ed efficienza elaborazione delle richieste) che analizza le richieste fatte dai client, associa le richieste ad ogni singolo client di provenienza e le mette in una coda di TaskRequest dove vengono prese ed elaborate da un ThreadPool di workers, che dopo aver eseguito tali richieste scrivono il "risultato" direttamente ai client.
  Esempio:
  Arriva una connessione di un nuovo client, accetto la connessione e il server sta pronto ad ascoltare il client. Arriva una richiesta da tale client, avviene un analisi della richiesta fatta, deserializzando la stringa arrivata dal client in un oggetto Request, associo la Request al channel del client e formulo un TaskRequest (dove c è l'implementazione necessaria per svolgere ogni tipo di richiesta e implementa Runnable) la metto in una coda (BlockingQueue). Dalla coda, un worker della Threadpool prende dalla coda la richiesta e la esegue, e scrive il risultato/feedback direttamente sul channel del client.
  -**Thread RequestHandler**, il thread che implementa la threadpool dei workers.




Dopo aver avviato i thread inizializzo un TerminatoreServer ovvero un Handler di Ctrl+C, per far terminare il server nella "maniera giusta", interrompendo tutti i thread e salvando tutti i file necessari per ricostruire il server.

## Lato Client:

All'avvio del client viene subito parsato il file di configurazione relativo al client e durante l'esecuzione del client verrà richiesto all'utente di seguire le istruzioni e di scandire in modo corretto le richieste da fare al client. Se non vengono rispettate le indicazioni la richiesta non viene inviata al server e si invita l'utente a riprovare nel modo corretto.

Inizialmente l'utente può scegliere di usare il comando register o login.

Dal momento in cui avviene fatto il login il client si registra ad un servizio di notifica alle CallBacks di RMI per ricevere notifiche di nuovi followers/unfollowers durante l'utilizzo di Winsome e aggiornando la lista dei followers dell'utente (salvata in locale) e notificando pure gli eventuali nuovi followers/unfollowers che ci sono stati mentre l'utente era offline. Ed infine avvia 2 thread in modo esplicito per gestire il collegamento TCP ed il collegamento UDP per il Multicast.

-Thread ConnectionHandlerClient, che prevede la gestione TCP, parsando le richieste ed inviandole al server tramite il metodo

```
public static void sendMessageTCP(String msg, SocketChannel
channel)
```

aspettando una replica da parte del server tramite il metodo

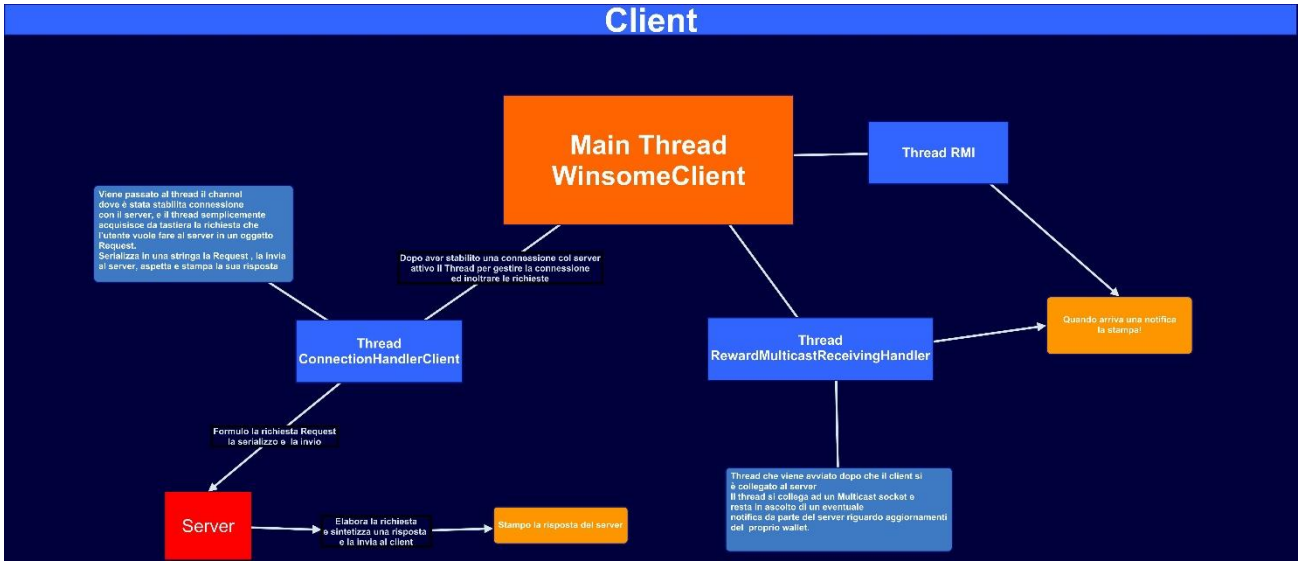
```
public static String receiveMessageTCP(SocketChannel channel)
```

-Thread RewardMulticastReceivingHandler, il thread collega il client ad un MulticastSocket e attende una notifica da parte del server.

La connessione TCP e del Multicast UDP viene mantenuta fino a quando viene chiuso il client tramite Ctrl+C o fino a che non viene mandata la richiesta di Logout.

Viene utilizzato un oggetto Console per sincronizzare le stampe delle varie notifiche e le risposte da parte del server.

(Ovviamente è possibile che arrivi una notifica mentre l'utente sta scrivendo a tastiera un comando, purtroppo bisogna riformulare il comando da capo)




Esecuzione: non serve passare parametri al ClientMain.

Per quanto riguarda i comandi da passare a tastiera sono stati utilizzati i comandi dell'esempio nella specifica del progetto.

In ogni caso basta digitare 'help' (senza virgolette) per avere a disposizione la lista dei comandi utilizzabili in quel momento!

Per terminare il client è a disposizione anche il Ctrl+C.

## Gestione strutture dati/concorrenza :

Per gestire la concorrenza tra le strutture dati ho utilizzato principalmente Concurrent Collections, più nello specifico, ConcurrentHashMap e CopyOnWriteArrayList.
Utilizzando strutture thread safe e usando operazioni composte atomiche adeguate ho reso il server e il client thread safe.
Come già detto in precedenza, per quanto riguarda la gestione della threadpool di workers e delle richieste, ho utilizzato una BlockingQueue.
E lato client per evitare che si sovrascrivessero stampe tra notifiche varie e risposte dal server ho utilizzato un monitor nella classe MyConsole.

## Classi utilizzate:

Utils:

Utils: classe dove sono presenti metodi utili per l'esecuzione del client e del server.
Serializator: classe che implementa i metodi per serializzare e deserializzare ogni tipo di oggetto utilizzato.
Utente: classe che implementa un utente di Winsome, utile solo per associare i tags agli Utenti.
Request: classe che implementa una richiesta del client al server.

Server:

WinsomeServer : thread principale del server.
ConnectionHandlerServer: classe che implementa il thread che gestisce le connessioni TCP.
RequestHandler: classe che implementa il thread gestore del threadpool di workers, i quali eseguono le richieste arrivati dai vari.
TaskRequest: classe che associa un oggetto Request (quindi una richiesta) e un Channel di connessione TCP tra server e client.
(implementa Runnable ed è l'oggetto passato al threadpoool di workers).
ExchangerHandler: classe che implementa il thread per cambiare il valore dell'exchanger dopo un intervallo di tempo assegnato.
RewardSendingHandler: classe che implementa il thread gestore del Multicast UDP.
TerminatoreServer: classe che implementa un handler di CtlrC facendo terminare il Server in maniera corretta.
Post: classe che implementa un post di Winsome
CommentPost: classe che implementa un commento di post
Transition: classe che implementa una transazione d'incremento del Winsome
Wallet: classe che implementa il wallet di un utente di Winsome.

Client:

WinsomeClient: thread principale del client
ConnectionHandlerClient: classe che implementa il thread per la gestione della connessione TCP,parsing delle richieste e invio dell' oggetto Request al server.
RewardMulticastReceivingHandler: classe che implementa il thread per la gestione del Multicast UDP.
MyConsole: classe che implementa un monitor per la sincronizzazione delle stampe a Schermo del client
CtrlCHandlerClient: classe che implementa un handler di CtrlC facendo terminare in modo Corretto il client.

## Annotazioni importanti:

Il rewin: quando si fa un rewin di un post, è come un retweet di Twitter, quindi se un post viene rewinnato, i commenti e voti non sono del rewin, ma bensì del post originale.
Quindi i rewin post non vengono calcolati per le ricompense, ma solo i post originali, in quanto come specificato prima i commenti e i voti vanno direttamente sul post originale.
Non si può fare il rewin di un rewin.

Il calcolo delle ricompense viene fatto per ogni singolo Post, quindi la classe Post ha un metodo interno per calcolare il valore della ricompensa globale

Per rendere piu verosimile il valore dei Wincoins, quando si cambia da Wincoin a Bitcoin, dopo aver applicato l'exchanger viene diviso il valore per 1'000'000. Se il valore ottenuto in Bitcoin non è molto significativo viene ritornato anche il valore in Satoshi.

Consigli per l'esecuzione del client: io per semplicità durante i test ho registrato gli utenti usando le lettere dell'alfabeto come username e password onde evitare di scordarmi le password (essendo Hashate da lato client), es. username: a ,password: a (ovviamente ho testato anche con username/password piu lunghe e diversi, è solo una comodità)

## Compilazione ed esecuzione:

**Compilazione:**
Partendo dalla directory src :
javac -cp .;../lib/gson-2.8.6.jar ./client/*.java ./server/*.java ./server/data_structure/*.java ./utils/*.java

**Esecuzione:**
Non serve passare parametri da linea di comando né al Server né al Client.

- Server: java -cp .;../lib/gson-2.8.6.jar server.ServerMain
- Client: java -cp .;../lib/gson-2.8.6.jar client.ClientMain

Per chiudere il server è necessario usare Ctrl+C.
Per chiudere il client è possibile usare Ctrl+C, oppure seguendo i comandi da linea di comando.