



# UNIVERSITÀ DI PISA

UNIVERSITÀ DI PISA, DIPARTIMENTO DI INFORMATICA

LABORATORIO DI RETI - A.A. 2021-2022 - CORSO B

DOCENTE: FEDERICA PAGANELLI

---

## Relazione progetto Winsome

---

Nicola Vetrini, matricola 600199

[n.vetrini@studenti.unipi.it](mailto:n.vetrini@studenti.unipi.it)

10 gennaio 2022

# Indice

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduzione</b>                           | <b>2</b>  |
| <b>2</b> | <b>Architettura del sistema</b>               | <b>3</b>  |
| 2.1      | Protocollo Richiesta/Risposta . . . . .       | 3         |
| 2.2      | RMI ed RMI callback . . . . .                 | 3         |
| 2.3      | UDP multicast . . . . .                       | 4         |
| <b>3</b> | <b>Server Winsome</b>                         | <b>5</b>  |
| 3.1      | Argomenti da riga di comando . . . . .        | 5         |
| 3.2      | Configurazione . . . . .                      | 5         |
| 3.3      | Esecuzione . . . . .                          | 6         |
| 3.3.1    | Thread e gestione della concorrenza . . . . . | 7         |
| 3.3.2    | Strutture dati . . . . .                      | 7         |
| <b>4</b> | <b>Client Winsome</b>                         | <b>8</b>  |
| 4.1      | Argomenti da riga di comando . . . . .        | 8         |
| 4.2      | Configurazione . . . . .                      | 8         |
| 4.3      | Esecuzione . . . . .                          | 9         |
| 4.3.1    | Thread e gestione della concorrenza . . . . . | 9         |
| 4.3.2    | Strutture dati . . . . .                      | 9         |
| <b>5</b> | <b>Compilazione ed esecuzione</b>             | <b>11</b> |
| 5.1      | Compilazione . . . . .                        | 11        |
| 5.2      | Esecuzione . . . . .                          | 11        |
| 5.3      | Documentazione . . . . .                      | 11        |
| <b>6</b> | <b>Librerie utilizzate</b>                    | <b>12</b> |
| 6.1      | Apache Commons CLI . . . . .                  | 12        |
| 6.2      | Jackson . . . . .                             | 12        |

# 1 Introduzione

Lo scopo del progetto è stata la realizzazione di una rete sociale, Winsome, basata sulla condivisione di contenuti tra un insieme di utenti.

Il social network Winsome consiste di due componenti software: **WinsomeClient** (client nel seguito) e **WinsomeServer** (server nel seguito). Il client ed il server comunicano principalmente tramite un protocollo richiesta/risposta su un socket TCP subito prima della login di ciascun utente e chiuso al logout. Pertanto il client ed il server possono essere avviati in modo indipendente, avendo però cura di notare che molti comandi del client non danno alcun risultato se la connessione al server non può essere stabilita.

Vi sono inoltre gli oggetti remoti per la comunicazione tramite RMI e RMI callback: il server crea un registry RMI che utilizza per fornire ai client gli stub per la registrazione e la sottoscrizione al servizio di callback.

Il server può inoltre fornire ai client che lo richiedano l'indirizzo di un gruppo UDP multicast, sul quale notifica regolarmente che i wallet degli utenti sono stati aggiornati (non viene comunicata alcuna informazione riguardo le ricompense calcolate). I client eseguono la richiesta di tale indirizzo ed effettuano la join su di esso soltanto al login di un utente, e lasciano il gruppo al logout: ho ritenuto non efficiente mantenere un socket in attesa di datagrammi (in un thread separato, come illustrato nel seguito) se non vi è alcun utente che possa controllare ontestualmente lo stato del proprio wallet dopo aver ricevuto la notifica.

Un aspetto dell'implementazione di Winsome proposta che assume particolare rilievo è la gestione dei rewin: l'azione compiuta dal rewin di un post è quella di spostare un post dal proprio feed al proprio blog. Dato che nella mia implementazione vi è una mappa globale dei post vi erano due approcci possibili: un primo approccio consiste nell'inserire il riferimento al post nel blog, l'altro nel creare un nuovo post che contenga l'id del post originale ed il suo autore, con una flag che indica che quel post è un rewin.

Nell'implementazione è stato adottato il secondo approccio, poiché consentirebbe, anche se non è stato implementato, di attribuire una ricompensa anche agli utenti che contribuiscono alla diffusione di un post, e quindi al suo successo nella rete sociale. Un punto importante è l'inserimento di voti e commenti: tali modifiche al post devono essere effettuate sul post originale, in modo tale che il calcolo dei commenti e dei voti possa correttamente assegnare all'autore, e non a chi ha effettuato il rewin, la somma dovuta. Tale decisione è motivata principalmente dal seguente scenario: se vi fossero due blog, A che ha un numero esiguo di followers e B che ha un gran numero di followers, ed il blog B effettuasse il rewin di un post di A esso potrebbe ricevere molti commenti e voti, in quanto compare sul feed di un gran numero di blog. Di conseguenza il blog B riceverebbe una grande ricompensa, sfruttando il post prodotto da un altro utente. La logica implementata previene questo tipo di situazioni, assegnando al creatore del post originale la ricompensa dovuta alla diffusione del post.

Se il post originale al quale un rewin si riferisce venisse cancellato, allora fallirebbe anche l'aggiunta di voti o commenti (il modo in cui questo controllo viene effettuato è commentato nel codice), seppure l'id del post cancellato rimanga visibile nella rete sociale sui blog di utenti che hanno effettuato il rewin di tale post. La cancellazione in cascata di tale id sarebbe un'operazione in generale molto dispendiosa, che può essere evitata effettuando dei controlli aggiuntivi molto più veloci.

## 2 Architettura del sistema

Di seguito sono illustrati i principali protocolli di comunicazione in uso nell'implementazione proposta di Winsome.

### 2.1 Protocollo Richiesta/Risposta

Il protocollo coinvolge principalmente due packages: **WinsomeRequests** e **WinsomeTasks**. I due package contengono una superclasse (**Request** e **Task** rispettivamente), la cui utilità è motivata dalla necessità di avere una classe base da poter serializzare/deserializzare. Infatti, dovendo gestire molti tipi di richieste diverse, il server ha la necessità di avere un modo semplice per distinguerle. Il meccanismo implementato è quello di serializzare in JSON (utilizzando la libreria Jackson) delle richieste sottoclassi di Request lato client, che quindi contengono tutte le informazioni inerenti alla specifica richiesta, per poi deserializzarle lato server come la superclasse Request e successivamente verificare di che tipo di richiesta si tratti, andando ad effettuare un casting al sottotipo appropriato per estrarre le informazioni dall'oggetto deserializzato.

Vi è quasi un mapping 1:1 tra i comandi del client e le sottoclassi di Request, tranne i seguenti comandi: **register**, **list followers**, **help**. La superclasse Request ha il campo (privato, ma con relativi metodi get e set) **kind**: la stringa contenuta in tale campo è necessaria al server per distinguere il tipo di richiesta in arrivo, per cui ogni sottoclasse di Request nel costruttore setta il campo alla stringa appropriata.

Associata ad una richiesta vi è la corrispondente Task lato server. (Quasi) tutte le sottoclassi di Task implementano l'interfaccia **Callable**, in quanto vengono eseguite da una threadpool e restituiscono un risultato che deve essere comunicato al client. Tale risultato può avere tipi diversi (dipende dall'operazione specifica: vi sono Task che restituiscono un Integer, alcune che restituiscono una String, ...), ma in ogni caso viene inserito in un ByteBuffer ed inviato al client.

Un esempio: il comando `login <user> <pwd>` provoca la creazione di una **LoginRequest**, i cui parametri del costruttore sono lo username e la password letti; la proprietà **kind** viene settata alla stringa "Login". Tale richiesta viene serializzata in JSON con un ObjectMapper e poi scritta sul socket TCP.

Il server deserializza la richiesta, crea una **LoginTask**, che viene sottomessa ad una threadpool; quando la task termina il risultato viene scritto sul socket del client richiedente.

### 2.2 RMI ed RMI callback

Attraverso il client è possibile registrare un nuovo utente su Winsome: tale operazione viene eseguita utilizzando RMI, come da specifica (l'interfaccia implementata dal server è **Signup**, il cui unico metodo è **register()**, che restituisce un intero).

Il comando **list followers** del client non genera una richiesta sincrona al server: è presente una struttura dati in ciascun client il cui contenuto viene mostrato all'utente. Tale elenco di followers viene aggiornato periodicamente dal server utilizzando il meccanismo di RMI callback: al login di un utente viene effettuata la registrazione al servizio di aggiornamento presso il server tramite RMI (interfaccia **FollowerUpdaterService**, metodo **subscribe()**) passando il nome dell'utente (loggato) che richiede il servizio ed un oggetto che implementa l'interfaccia **FollowerCallback** (**FollowerCallbackImpl**) al server. Su tale oggetto remoto il server invoca il metodo **updateFollowers()** e successivamente **setUpdateTimestamp()** per eseguire l'aggiornamento dell'insieme dei followers di tale utente.

L'aggiornamento avviene a cadenza regolare, compatibilmente con il carico del server, che può essere impostata nel file di configurazione del server (maggiori dettagli in seguito).

Durante l'operazione di logout dell'utente viene chiamato anche il metodo per la cancellazione del

servizio (metodo `unsubscribe()` di **FollowerUpdaterService**), al fine di rimuovere dal server il riferimento all'oggetto del client usato per il callback. Dato che il comando `quit` del client richiama al suo interno, se vi è un utente loggato, la procedura di logout, di fatto la deregistrazione avviene in ogni caso in modo automatico.

### 2.3 UDP multicast

Per ricevere update sull'aggiornamento dei wallet un client può inviare una Request di tipo "Multicast" al server, che risponde fornendo l'indirizzo (IP + porta) del gruppo multicast sul quale invia tali notifiche. I client inviano automaticamente tale richiesta soltanto al login di un utente. A quel punto viene creato un thread apposito (**McastListener**) che sta in ascolto. Il socket UDP ha un timeout settato, in modo che non si blocchi in attesa in modo indefinito. Al logout dell'utente, infatti, tale thread viene fatto terminare.

La motivazione di tale scelta è dovuta alla considerazione che sia poco utile tenere un thread attivo per tutta la durata dell'esecuzione del server

## 3 Server Winsome

Il server è contenuto in un package, **WinsomeServer**, che contiene le seguenti classi principali:

- **ServerMain.java**, contenente la classe main del server
- **WinsomeServer.java**, contenente la classe che incapsula i dati della rete sociale e le funzionalità principali
- **User.java**, **Post.java**, **Comment.java** ed altre, che definiscono il formato dei dati all'interno della rete sociale

### 3.1 Argomenti da riga di comando

Il server può ricevere uno o più dei seguenti parametri da riga di comando, che ne influenzano il comportamento all'avvio. Si noti che i valori passati da riga di comando hanno la precedenza su quelli eventualmente letti dal file di configurazione caricato.

Se non è stato specificato alcun valore per un parametro, verrà usato il valore di default statico presente nel file **ServerConfig.java**.

Per un riepilogo delle opzioni disponibili è sufficiente passare la flag **-h** o **--help**, in ogni caso le opzioni disponibili sono le seguenti:

- **-c --configure <FILE>**: path del file di configurazione che il server deve caricare
- **-h --help**: messaggio di uso del server
- **-p --socket-port <PORT>**: porta sulla quale, se possibile, viene creata la **ServerSocketChannel** sulla quale il server si mette in ascolto per accettare le connessioni dai client
- **-r --registry <PORT>**: porta sulla quale, se possibile, viene creato il registry RMI utilizzato sia per la registrazione, che per l'iscrizione/disiscrizione dal servizio di aggiornamento dei followers

### 3.2 Configurazione

Il server è configurato attraverso un file JSON che viene cercato, nell'ordine, ai seguenti path:

1. il path passato attraverso l'opzione **-c <path>**
2. **config.json** nella directory corrente
3. **data/WinsomeServer/config.json** il file di configurazione di default

Se tutte le opzioni precedenti non contengono un file di configurazione valido allora il server termina immediatamente.

I parametri configurabili sono i seguenti:

**dataDir** : Il path alla directory dalla quale caricare i dati della rete sociale, ovvero la directory che contiene il file **users.json**, la directory **blogs** e opzionalmente il file di configurazione

**outputDir** : Il path della directory nella quale salvare lo stato del server alla terminazione. Se non esiste il server tenta di crearla

**registryPort** : La porta del registry RMI nel quale inserire lo stub per la registrazione. Il valore deve essere un intero nel range [0, 65535]

**serverSocketAddress** : Nome host o indirizzo IP del **ServerSocket** creato dal server per accettare le connessioni dai client (IP pubblico del server nel caso generale, ma in questo caso localhost)

**serverSocketPort** : Porta alla quale deve essere legato il socket sopracitato, nel range  $[0, 65535]$

**multicastGroupAddress** : Indirizzo IP del gruppo multicast sul quale inviare le notifiche di aggiornamento dei wallet

**multicastGroupPort** : Porta relativa all'indirizzo multicast precedente

**minPoolSize** : Intero ( $> 0$ ) che rappresenta il numero di core thread nella threadpool per la gestione delle richieste del server

**maxPoolSize** : Intero ( $x : INT\_MIN \leq minPoolSize \leq x < INT\_MAX$ ) che rappresenta il numero massimo di thread che possono essere gestiti contemporaneamente dalla threadpool

**workQueueSize** : Dimensione della coda di task che la threadpool può accumulare in attesa che un thread del pool sia libero, in accordo con la politica di gestione delineata dalla documentazione di ThreadPoolExecutor

**retryTimeout** : long ( $> 0$ ) che rappresenta il numero di millisecondi che l'handler per la gestione delle richieste rifiutate dal threadpool attende prima di provare a sottomettere nuovamente la richiesta

**callbackInterval** : long ( $> 0$ ) che rappresenta l'intervallo tra due consecutivi aggiornamenti alla lista dei follower del client registrato al servizio. L'unità di misura che quantifica il valore è specificata dal parametro indicato di seguito

**callbackIntervalUnit** : Unità di misura di callbackInterval. Essendo rappresentato con una [TimeUnit](#) i valori possibili sono quelli dell'enumerazione indicati nel link. Di default l'unità di tempo sono i secondi. Nel file JSON una TimeUnit è serializzata come stringa, quindi per indicare di misurare callbackInterval in secondi è sufficiente scrivere "callbackIntervalUnit": "SECONDS" nel file JSON.

**authorPercentage** : Percentuale della ricompensa calcolata che deve essere assegnata all'autore. Il valore specificato deve essere un double  $x : 0 < x < 1$

Non è necessario specificare tutti i parametri nel file, poiché quelli assenti assumeranno i valori di default definiti nella classe `ServerConfig.java`. Il file JSON viene deserializzato utilizzando Jackson con ObjectMapper in un'istanza della classe menzionata, ed un riferimento ad essa viene passato alla creazione del server.

Nella classe è in realtà presente anche il campo (non serializzato) `configFile`, che serve soltanto per determinare, nella funzione `getServerConfiguration()` in `main()`, se era stato settato con l'opzione `-c` il path per un file di configurazione, che ha la precedenza rispetto a quello di default.

### 3.3 Esecuzione

Il codice del server Winsome è contenuto all'interno del package WinsomeServer; la classe ServerMain in tale package contiene il metodo main, quindi è quella da eseguire per far partire il server. Il server stampa su standard output all'avvio tutti i parametri della propria configurazione, con altre informazioni di supporto relative al caricamento dei dati ed eventuali errori.

In particolare, prima viene caricato il file degli utenti; al termine, per ciascun utente, viene caricato il suo blog. Se il file contenente il blog non esiste viene notificato su stderr ed il caricamento prosegue (si crea nel server un nuovo blog vuoto per tale utente). Successivamente viene creato il ServerSocketChannel non bloccante che si mette in ascolto di nuove connessioni e viene attivato un selector per il multiplexing delle richieste sui socket attivi.

Quando un client richiede la connessione viene creato il SocketChannel non bloccante e viene registrato per l'operazione di lettura, con l'attachment **ClientData** che incapsula lo stato di quel client. Quando

un `SocketChannel` diventa readable si effettua la read della richiesta e si registra quel `SocketChannel` per l'operazione di scrittura (`setWritable()`), quindi il selector non è più in ascolto per operazioni di lettura da quel socket). Una volta pronta la risposta ed il socket per la scrittura la si effettua; successivamente si registra il socket solo per la lettura (`setReadable()`) in modo analogo.

La terminazione del server avviene interrompendo la JVM, ad esempio con `Ctrl+C`, ma dato che all'avvio sono registrati degli shutdown hook per la sincronizzazione dei contenuti del social network lo stato corrente al momento della terminazione viene salvato sovrascrivendo il file degli utenti, dei blog ed il file di configurazione. Una volta terminati i thread di sincronizzazione (tutti operanti su un set di dati indipendenti, quindi non soggetti a particolare attenzione per quanto riguarda la sincronizzazione) la JVM su cui eseguiva il server può terminare.

### 3.3.1 Thread e gestione della concorrenza

Il server `Winsome` è multithreaded: vi è il thread del main, il cui compito è l'inizializzazione del server e del registry; dal thread principale è fatta partire un'istanza di `WinsomeServer`, sottoclasse di `Thread`, che si occupa di gestire lo smistamento delle richieste. Una volta lanciato con successo il `WinsomeServer` il thread main termina.

Nel thread del `WinsomeServer` vi è un selector che permette di leggere le richieste provenienti dai client e sottometterle ad una threadpool le cui dimensioni minime e massime sono fissate dal file di configurazione.

Nel costruttore di `WinsomeServer`, inoltre, viene attivato un thread per leggere gli utenti di `Winsome` dal file `users.json` e caricarli in memoria. Una volta letti gli utenti viene attivato un `BlogLoaderThread` per ciascun utente letto dal file, il cui compito è deserializzare dal file `<datadir>/blogs/<user>.json` tutti i post presenti in tale blog e caricarli nelle strutture dati del server (ogni post è inserito nella mappa post dei globale e nella lista dei post di ogni blog). Tali thread terminano una volta caricato il blog (il thread che esegue il costruttore di `WinsomeServer` si blocca finché la join su ciascuno di essi non ritorna).

Alla terminazione del server, poiché lo stato deve persistere, ogni utente ed ogni post dei loro blog devono essere scritti su file. Per fare questo è stato utilizzato il meccanismo degli shutdown hook, settati come prima istruzione del metodo `run()` del `WinsomeServer`.

Vi è un hook (thread su cui non è stato invocato `start()`) per la sincronizzazione del file degli utenti: `SyncUsersThread.java`. L'altro hook è `SyncBlogsThread.java`, il cui unico compito è quello di creare e far partire un `SyncPostsThread.java` per ogni utente `Winsome`, che sincronizza i post di un solo blog. Tali thread operano su dati totalmente indipendenti, per cui non è richiesta alcuna sincronizzazione delle loro operazioni, il che consente di avere il massimo grado di parallelizzazione del processo consentito dalla macchina. L'unica accortezza è che il thread che li ha creati non termini fino a che tutti questi thread non sono terminati, altrimenti la JVM potrebbe terminare lasciando uno o più blog in uno stato inconsistente, che provocherebbe degli errori al riavvio del server.

### 3.3.2 Strutture dati



## 4 Client Winsome

Il client è contenuto in un package, **WinsomeClient**, che contiene le seguenti classi principali:

- **ServerClient.java**, contenente la classe main del client
- **WinsomeClientState.java**, contenente la classe che incapsula lo stato corrente del client
- **McastListener.java**, che contiene il Thread che ascolta i messaggi di update dei wallet sul gruppo multicast

### 4.1 Argomenti da riga di comando

Il client può ricevere uno o più dei seguenti parametri da riga di comando, che ne influenzano il comportamento all'avvio. Si noti che i valori passati da riga di comando hanno la precedenza su quelli eventualmente letti dal file di configurazione caricato.

Se non è stato specificato alcun valore per un parametro, verrà usato il valore di default statico presente nel file **ServerConfig.java**.

Per un riepilogo delle opzioni disponibili è sufficiente passare la flag **-h** o **--help**, in ogni caso le opzioni disponibili sono le seguenti:

- **-c --configure <FILE>**: path del file di configurazione che il server deve caricare
- **-h --help**: messaggio di uso del server
- **-s --host <HOSTNAME>**: indirizzo IP o hostname del server
- **-p --socket <PORT>**: porta sulla quale connettersi al server al login
- **-r --registry <PORT>**: porta sulla quale cercare il registry RMI del server

### 4.2 Configurazione

Il client è configurato attraverso un file JSON che viene cercato, nell'ordine, ai seguenti path:

1. il path passato attraverso l'opzione **-c <path>**
2. **config.json** nella directory corrente
3. **data/WinsomeClient/config.json** il file di configurazione di default

Se tutte le opzioni precedenti non contengono un file di configurazione valido allora il server termina immediatamente.

I parametri configurabili sono i seguenti:

**dataDir** : Path della directory nella quale sono

**registryPort** : Porta sulla quale effettuare il lookup del registry RMI

**serverHostname** Hostame o indirizzo IP del server ("localhost" in questo caso)

**serverPort** : Porta del server

**netIf** : Nome dell'interfaccia di rete sulla quale il MulticastSocket effettua la join su un indirizzo comunicato dal server. Molto probabilmente è l'unico parametro da settare ad un'intefaccia di rete che supporta la ricezione di datagrammi UDP su gruppi multicast

Non è necessario specificare tutti i parametri nel file, poiché quelli assenti assumeranno i valori di default definiti nella classe `ClientConfig.java`. Il file JSON viene deserializzato utilizzando Jackson con `ObjectMapper` in un'istanza della classe menzionata, ed un riferimento ad essa viene passato alla creazione del client.

Nella classe è in realtà presente anche il campo (non serializzato) `configFile`, che serve soltanto per determinare, nella funzione `getClientConfiguration()` in `main()`, se era stato settato con l'opzione `-c` il path per un file di configurazione, che ha la precedenza rispetto a quello di default.

### 4.3 Esecuzione

Il codice del client Winsome è contenuto all'interno del package `WinsomeClient`; la classe `ClientMain` in tale package contiene il metodo `main`, quindi è quella da eseguire per far partire il server. Il client stampa su standard output all'avvio tutti i parametri della propria configurazione e presenta un prompt sul quale l'utente può digitare dei comandi. Per un riepilogo della sintassi e l'obiettivo di ciascun comando si può digitare il comando `help`.

Il comando `login <user> <pwd>` consente di effettuare il login di un utente registrato; una volta loggati il prompt mostra il nome dell'utente, mentre a seguito del `logout` ritorna quello di default. Con il comando `quit` è possibile effettuare, se necessario, il `logout` e terminare il client.

Il client all'avvio non è connesso al server, per cui appena prima di effettuare il login tenta di aprire una nuova connessione tramite la creazione di un socket TCP verso il server (con indirizzo e porta specificati in base alla procedura presentata in precedenza): se fallisce viene riportato l'errore e l'esecuzione del client continua normalmente, mentre se ha successo viene automaticamente inviata una richiesta del gruppo multicast su cui ricevere update dei wallet ed il client effettua l'iscrizione al servizio di callback per i follower.

#### 4.3.1 Thread e gestione della concorrenza

Il client è un processo multithreaded: vi è un thread principale (quello in cui esegue `ClientMain`) ed una serie di thread la cui creazione e terminazione è variabile: il thread `McastListener.java` viene creato dal thread `main` dopo aver ricevuto il gruppo multicast su cui mettersi in ascolto, per cui rimarrà in esecuzione fino a che l'utente non effettua il `logout`. A quel punto il thread (dopo un delay dovuto al timeout del socket) termina. Avrei potuto prevedere un thread che, una volta ottenuto al primo login il gruppo multicast, restasse in esecuzione per tutti i client, imponendo il vincolo che il gruppo comunicato dal server fosse lo stesso per ciascun client. Tuttavia l'uso di un protocollo richiesta/risposta ed un thread il cui tempo di vita è legato al login dell'utente mi è sembrata una scelta migliore, in quanto consente maggiore flessibilità da parte del server Winsome.

#### 4.3.2 Strutture dati

Lo stato del client è contenuto principalmente nella classe dedicata `WinsomeClientState`, la quale memorizza le seguenti informazioni:

**currentClient** Una stringa contenente l'username dell'utente loggato (di default è "")

**signupStub** Riferimento all'oggetto remoto sul quale effettuare l'operazione di registrazione di un nuovo utente

**isQuitting** Flag booleana che indica se il client deve terminare (settata dal comando "quit")

**tcpConnection** Riferimento al socket TCP bloccante che connette un client in cui un utente è loggato al server

**callbackRef** Riferimento all'oggetto remoto che implementa la procedura di callback utilizzata dal server. Questo oggetto è settato da ciascun client prima di registrarsi per il callback e resettato al logout

**mcastAddr, mcastSock, mcastThread** Riferimenti alle entità coinvolte nella gestione del thread in ascolto sul gruppo multicast

**run\_thread** Un AtomicBoolean che controlla se il thread sopracitato debba o meno continuare la propria esecuzione. Ho utilizzato un AtomicBoolean invece di una lock perché non vi è alcuna condizione particolare che il thread debba attendere prima di entrare in esecuzione

Un'altra struttura dati di rilievo è la classe **FollowerCallbackImpl**, che implementa l'interfaccia omonima. Tale classe ha metodi per aggiornare i follower, resettare la lista ed aggiornare il timestamp dell'ultimo update della lista. Lo stato aggiornato dai metodi viene mantenuto internamente nell'istanza ed è ottenuto tramite i rispettivi getters.

Infine vi è la classe **ClientCommand** e l'enumerazione **Command**, alle quali spetta il parsing dei comandi: a seconda del Command ottenuto viene scelto un ramo dello switch in ClientMain, che richiama la funzione che si occupa di inviare la richiesta appropriata e visualizzarne la risposta a video.

## 5 Compilazione ed esecuzione

I comandi elencati di seguito assumono che la directory corrente sia quella base del progetto (quella con la subdirectory `src`). Sono inoltre allegati un Makefile e degli script per la compilazione ed esecuzione (`serv_run.sh` e `client_run.sh`), il cui scopo è semplicemente quello di automatizzare l'esecuzione dei comandi riportati di seguito.

La compilazione ed esecuzione è stata testata su Java 11, ma dovrebbe compilare ed eseguire senza modifiche su Java  $\geq 8$

### 5.1 Compilazione

Il comando per la compilazione del server è il seguente:

```
javac -d bin -cp "libs/*" -sourcepath src/ src/Winsome/WinsomeServer/*.java
```

Per compilare il client è sufficiente sostituire `WinsomeClient` al posto di `WinsomeServer`.

Il risultato della compilazione è nella directory `bin/Winsome`.

Sono disponibili anche i comandi per la generazione di eseguibili in formato jar come segue:

```
jar cfm bin/WinsomeServer.jar server-manifest.txt -C bin Winsome per il server e
```

```
jar cfm bin/WinsomeClient.jar client-manifest.txt -C bin Winsome per il client.
```

### 5.2 Esecuzione

Per eseguire il server è sufficiente il comando

```
java -cp ".:bin/:libs/*" Winsome.WinsomeServer.ServerMain
```

Sostituendo a `"WinsomeServer.ServerMain"` `"WinsomeClient.ClientMain"` si può eseguire il client.

Eventuali argomenti da riga di comando al client o al server possono essere aggiunti in fondo.

Per eseguire i jar è sufficiente il comando `java -jar bin/WinsomeServer.jar` ed uno analogo per il client.

### 5.3 Documentazione

È disponibile nel Makefile un target (`make doc`) per generare con javadoc la documentazione delle classi realizzate, navigabile tramite browser, anche se il comando (eliminando gli `'`) può essere eseguito anche da terminale. La documentazione viene generata nella subdirectory `doc`, per cui è sufficiente aprire il file `doc/index.html` con il browser per sfoglarla.

Con l'opzione `-link` di javadoc si ottengono link cliccabili alla documentazione delle classi della libreria standard Java e delle librerie utilizzate, anche se ciò potrebbe introdurre un leggero ritardo nella generazione, per cui le righe contenenti `-link` possono essere tolte, se necessario.

## 6 Librerie utilizzate

Le librerie esterne utilizzate dal software sono state incluse, sotto forma di file .jar, nella cartella *libs*

### 6.1 Apache Commons CLI

È stata utilizzata la libreria Apache Commons CLI versione 1.5.0 per il parsing degli argomenti da riga di comando sia in WinsomeServer che in WinsomeClient.

### 6.2 Jackson

All'interno del software, soprattutto nella componente server, è stata utilizzata ripetutamente la serializzazione e deserializzazione tra classi ed oggetti JSON scritti su file. A tale scopo ho scelto di utilizzare la libreria Jackson, versione 2.9.7.