Relazione progetto Winsome

Reti di Calcolatori E Laboratorio 2021/2022

Studente: Fabrizio Cau - Corso A - Matricola 508700
Docente: Laura Emilia Maria Ricci

Introduzione

Questa relazione illustra l'architettura e la struttura del sistema Winsome realizzato; nonostante ciò, il codice fornito è stato ampiamente commentato in ogni suo metodo, classe e struttura dati di particolare importanza (rispettando la javadoc, e quindi facilmente consultabile con i principali IDE), oltre ai commenti relativi ad ogni funzionalità.

Il codice realizzato è conforme al **JDK versione 8** di Java come utilizzato a lezione, e non presenta errori di compilazione o a runtime (non previsti). Non è però garantita la corretta compilazione e funzionamento sotto versioni diverse, in quanto alcune librerie e metodi utilizzati potrebbero essere non disponibili in versioni precedenti o deprecati in versioni successive.

Descrizione architettura del sistema

Si è realizzato un sistema client-server multithread in entrambi i lati con comunicazione mediante protocolli TCP, RMI e RMI Callback

La **comunicazione principale** avviene mediante connessione persistente TCP realizzata nella classe *ClientReception* mediante un canale `ServerSocketChannel` di Java NIO. Ogni nuova connessione di client verrà inviata come parametro al costruttore di un nuovo task per il threadpool.

Il **threadpool** viene inizializzato con il predefinito `newCachedThreadpool` che gestisce l'esecuzione assegnando i task ClientHandler ad un thread.

Il **ClientHandler** gestirà interamente la comunicazione bilaterale con il client scrivendo e leggendo nel `SocketChannel` ricevuto e usando come intermediario un `ByteBuffer` di dimensione definita nel file di configurazione (per poterlo ampliare in futuro). Analogamente questa parte viene eseguita nel client, in particolare nella classe ClientMain in un metodo separato.

Il **ClientMain all'avvio** apre la SocketChannel e tenta la connessione al server con un timeout configurabile, allo scadere del timeout viene catturata l'eccezione e viene eseguito nuovamente un tentativo fino ad un massimo numero di tentativi stabiliti nel file di configurazione. Se entro questi tentativi non viene accettata la connessione, il Client termina.

La **collezione di utenti e post** viene mantenuta nella classe *db* del server che definisce metodi e strutture dati `public` per rendere disponibile l'accesso a tutte le altre classi. All'avvio, il main del server avvia il costruttore del database e questo a sua volta avvia il recupero delle collezioni dai file JSON. Successivamente le due collezioni saranno disponibili a tutti i thread del server (approfondimento nel paragrafo sulla concorrenza).

Serializzazione e deserializzazione sono effettuate accedendo ai file mediante un `FileChannel` (Java NIO) e utilizzando metodi della libreria `Gson` per la traduzione da e verso Json e metodi per analizzare i dati letti specificando il tipo di dato (`TypeToken`)

Le due **strutture dati** sono realizzate con oggetti di tipo `Map` in cui si associa l'id del post (chiave univoca) ad un oggetto di tipo Post e lo username dell'utente ad un oggetto di tipo User. Ognuna possiede comunque all'interno la stessa chiave/id per comodità ed ottimizzazione di implementazione.

Per quanto riguarda la collezione dei post, viene definita una variabile intera usata per assegnare un nuovo **postId** alla creazione di un post, ma dal momento che la struttura non preserva un ordine e poiché vi è la possibilità di eliminare un elemento, si è scelto di inizializzare a zero questo valore all'avvio; successivamente, durante la creazione del post, si incrementa tale valore fino a trovare una posizione libera e alla delete si riporta al valore del post eliminato, in modo da riempire sempre i postId "non occupati" (consecutivamente). Questo può portare un leggero overhead all'avvio del server in caso di collezioni molto grandi.

La **registrazione** di un utente viene effettuata tramite **RMI**: il server nel main crea un registry e configura il suo stub mettendo a disposizione il metodo register, implementato nella classe *RMIcomandList* e di cui anche il client possiede l'interfaccia.

Il client, dopo aver instaurato la connessione TCP, avvia la ricerca del servizio nel registry del server e al comando register dell'utente invoca il metodo remoto omonimo.

Il **servizio RMI callback** per l'aggiornamento dei follower viene realizzato analogamente a quello precedente; in questo caso il main del client ricerca il servizio nel registry e dopo la conferma del login chiama il metodo remoto per registrare il proprio stub tramite il quale il server può invocare i metodi per aggiornare in modo asincrono i follower nella struttura dati dell'utente.

Per implementare la **struttura dati che mantiene gli stub degli utenti**, nella classe *ServerFollowersUpdImpl*, è stato definito un oggetto `Map` che, ad ogni Utente (username) registrato alle notifiche, mappa una lista di stub (di tipo *CliFollowersUpdInterface*). In questo modo il server può gestire la registrazione alle notifiche di più client loggati con lo stesso utente contemporaneamente.

Tutte le strutture dati che ospitano una lista di username sono realizzate mediante `Hashset` poiché gli username sono chiavi che identificano univocamente gli utenti. Inoltre, rende impossibile inserire due volte lo stesso username e allo stesso tempo risulta computazionalmente semplice l'accesso al singolo elemento.

Il **Rewarder** è un task eseguito da un thread apposito che svolge i calcoli necessari per assegnare un guadagno ad ogni post e ripartire questo tra gli utenti curatori e l'autore. Questo calcolo viene fatto ad intervalli configurabili, e al termine viene inviata una notifica mediante l'invio di un pacchetto UDP ad un indirizzo multicast che i client in ascolto riceveranno.

Il calcolo viene fatto analizzando dapprima se il post era presente l'ultima volta che il calcolo è stato effettuato, e se già visitato, l'algoritmo procede effettuando una "sottrazione" delle iterazioni del post già visitato e quello attuale. Questo è possibile utilizzando strutture dati quali `HashSet` e salvando i post "già visitati" in una struttura dati ausiliaria mediante deep-copy (metodo implementato nelle classi Post e Comment) per evitare di copiare il riferimento all'oggetto.

Il client dopo la conferma di login richiede al server l'invio in blocco di tutti i followers per aggiornare la collezione locale e successivamente richiede l'invio dei parametri multicast su cui mettersi in ascolto. Questi, insieme al messaggio di uscita (volontaria) del client sono comandi interni non accessibili all'utente.

Il server inoltre utilizza il **protocollo HTTP** nella classe ClientHandler per recuperare un numero casuale generato da un sito web esterno mediante una query URL. Il range di valori richiesto nella query viene

calcolato a partire da un numero di base a cui viene sommato (per il max) e sottratto (per il min) la percentuale di scostamento stabilita nel file di configurazione. Il valore ricevuto sarà diviso per il numero base e moltiplicato al valore dei Wincoin per ottenere il cambio in Bitcoin. In pratica, configurando il range a 0.5 si può ottenere un tasso di cambio pari ad un multiplo che varia da 0.50000 a 1.50000

Ai fini di debug si è realizzato un meccanismo di stampa di messaggi di log sia su console (configurabile) che sul file di testo log.txt

Struttura dati condivisa e concorrenza

La struttura dati che ospita la collezione di utenti e post è stata implementata istanziando una `ConcurrentHashMap`, ovvero una `HashMap` che gestisce autonomamente gli accessi concorrenti. Non è stato necessario utilizzare altri strumenti per l'accesso thread-safe a questi dati, ad eccezione di un blocco sincronizzato nella creazione del post per evitare problemi di concorrenza sulla variabile intera usata come indice incrementale di post. La struttura dati infatti, effettua internamente l'accesso in mutua esclusione ad ogni elemento della `Map`, pertanto, un solo thread per volta può accedere allo stesso elemento.

Inoltre, l'intero usato come indice viene definito nella classe *db* con keyword `volatile` per permettere un accesso a tutti i thread del valore aggiornato.

Dal lato client, seppur vi siano solo due thread, viene definita una variabile di tipo `AtomicBoolean` che indica se l'utente è loggato oppure no e permette al main di notificarlo all'altro thread che ha immediata visibilità della modifica.

Sicurezza e controllo degli errori

La sicurezza è gestita in piccola parte mediante il calcolo della funzione hash delle password inserite alla registrazione e dunque alla verifica quando viene richiesto il login.

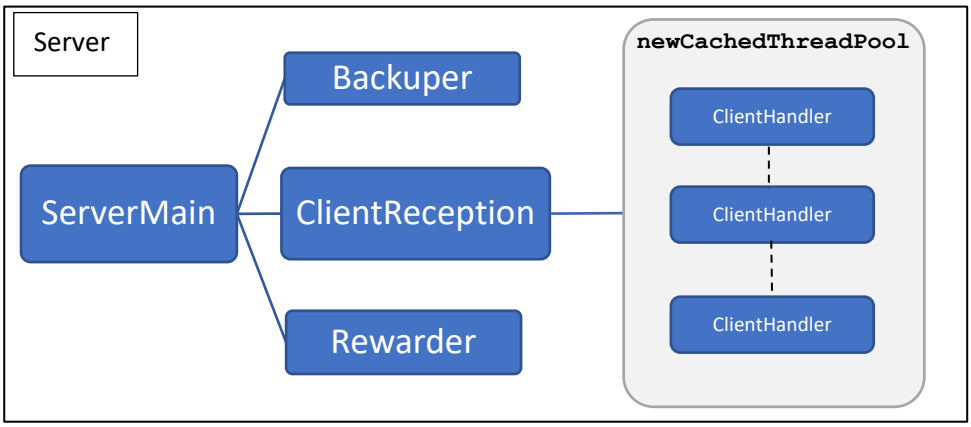
Vi è nel sistema un controllo degli errori accurato, a partire dalla lettura dei file di configurazione (nel quale devono essere rispettati determinati criteri per ogni valore) e soprattutto nella classe ClientHandler che gestisce l'iterazione con il client: qui viene effettuato il parsing del comando richiesto e vi è un controllo su tutti i parametri inseriti e sulla possibilità di eseguire l'operazione.

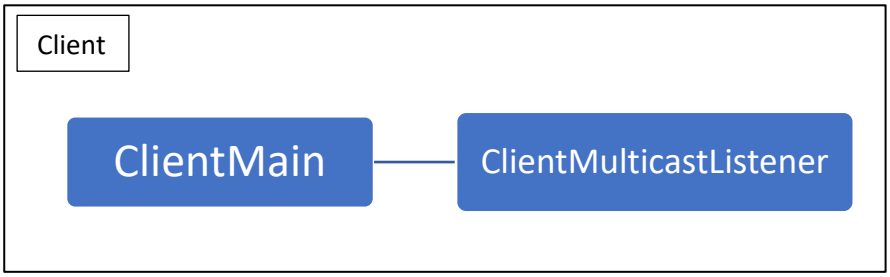
Dal lato client viene effettuato un controllo mediante i metodi della classe ComTools.

Ogni operazione produce un risultato e/o un messaggio di risposta sia all'utente che nella console e nel log del server.

Sono stati inseriti molti blocchi try/catch per assicurarsi che non venga interrotto il server in modo imprevedibile ma che, se l'errore non è fatale, si possa inviare un messaggio di errore e ripetere l'operazione oppure terminare mediante gli appositi strumenti che chiudono tutte le strutture dati aperte e salvano le collezioni nei database Json. Questi blocchi, purtroppo, causano un maggiore overhead.

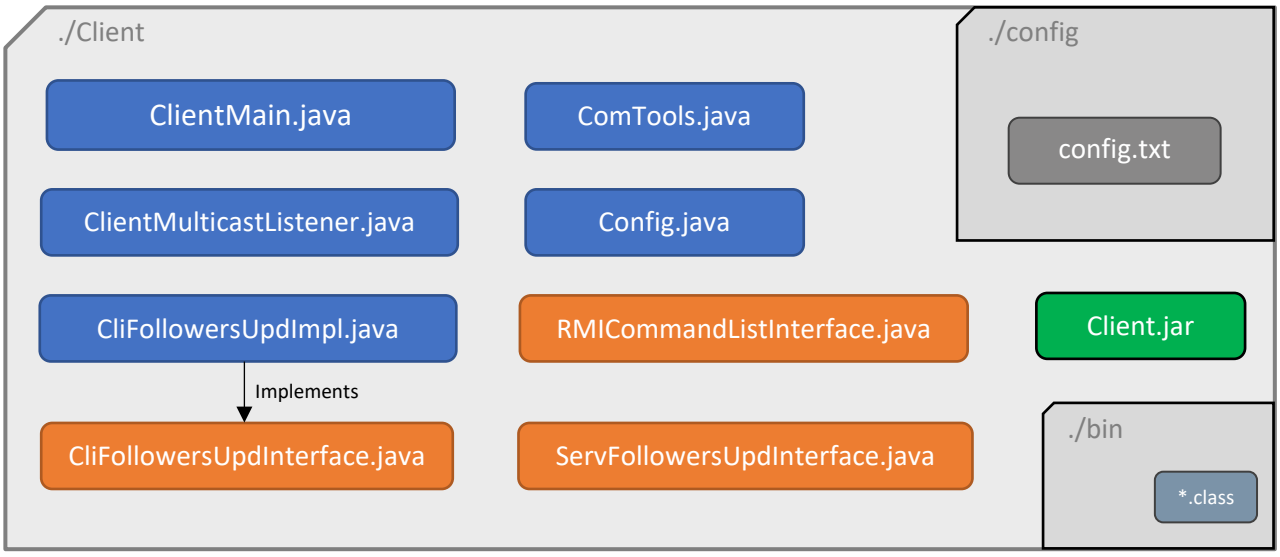
Schema gerarchico dei threads





Schema files e directories

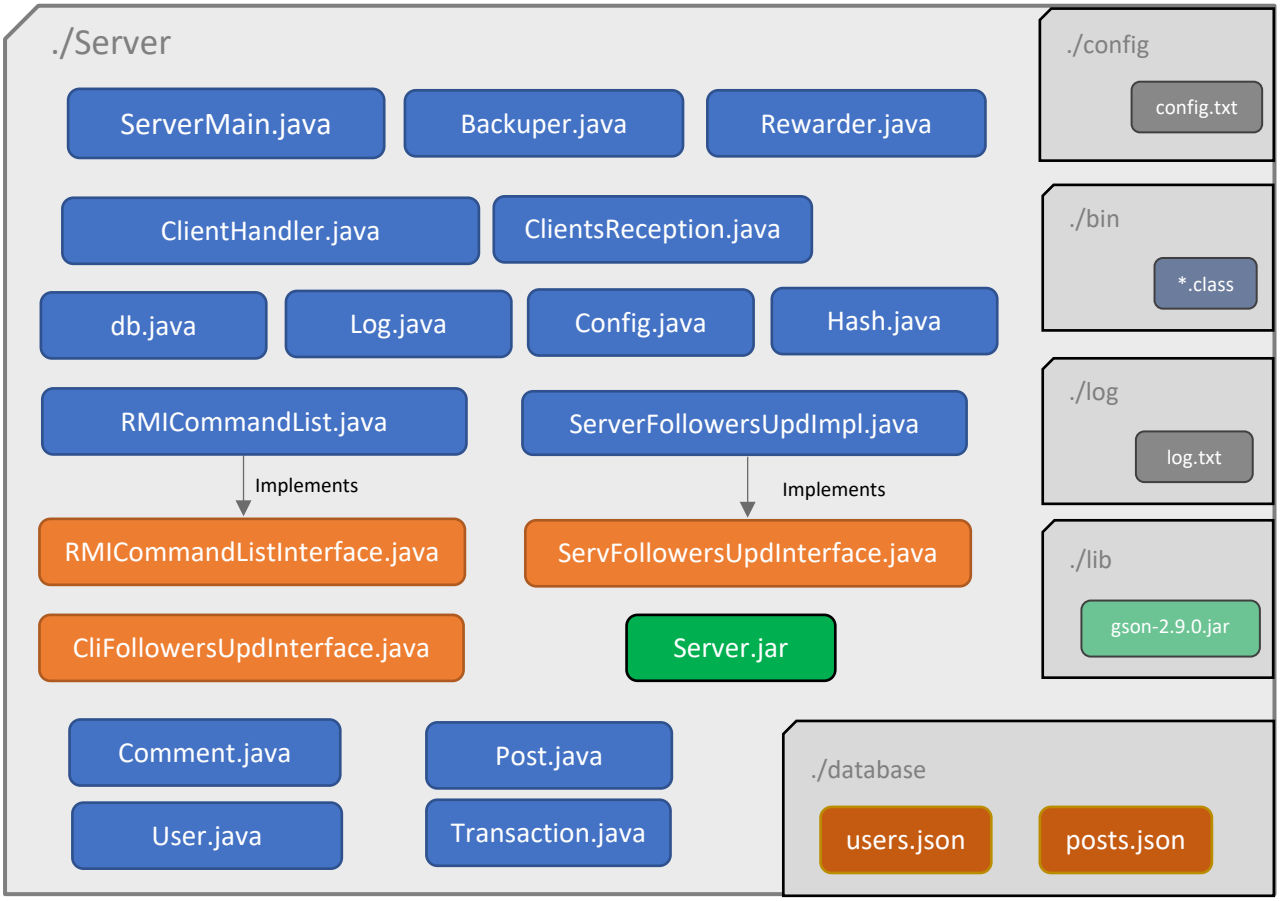
Descrizione file directory Client



- **ClientMain.java:** contiene il main dell'applicazione client, definisce le strutture dati utili per effettuare la connessione con il server mediante le varie tecnologie menzionate nei paragrafi precedenti. Implementa una comunicazione con l'utente mediante CLI e delega ai metodi della classe *ComTools* il controllo del comando inserito. In base al comando inserito, effettua verifiche sulla risposta del server e restituisce un messaggio nella console all'utente. All'interno della classe vi sono due metodi interni per gestire correttamente la chiusura e l'invio/ricezione di messaggi con il server.

- **ComTools.java:** implementa alcuni metodi per il controllo della correttezza del comando inserito dall'utente e per la stampa dei comandi disponibili (`> help`)
- **ClientMulticastListener.java:** definisce il task che si occupa della ricezione di pacchetti multicast
- **Config.java:** definisce i parametri di configurazione di default e la lettura dei parametri aggiornati dal file config.txt
- **CliFollowersUpdImpl.java:** implementa i metodi dell'interfaccia *CliFollowersUpdInterface* che saranno invocati dal server per inviare aggiornamenti asincroni (RMI Callback) sui followers
- **CliFollowersUpdInterface.java:** interfaccia della classe *CliFollowersUpdImpl*
- **RMICommandListInterface.java:** interfaccia che definisce il metodo remoto (RMI) per la registrazione di un utente
- **ServFollowersUpdInterface.java:** interfaccia che definisce i metodi remoti utilizzabili dal client per registrare il proprio stub nel registry pubblicato dal server
- **config.txt:** file di configurazione
- **Client.jar:** applicazione del client

Descrizione file directory Server



- **ServerMain.java:** contiene il main dell'applicazione server, avvia i costruttori delle classi che definiscono i parametri di default, il log su console e la struttura dati che conterrà post e utenti, crea i registry per i comandi remoti RMI e RMI Callback, e avvia i thread principali (vedi sezione successiva). Successivamente resta in attesa di un comando da console per effettuare la chiusura correttamente.
- **Backuper.java:** task che ad intervalli definiti nel file di configurazione chiama i metodi della classe *db* per il backup dei dati di post e utenti sui file .json

- **Rewarder.java**: task che ad intervalli definiti nel file di configurazione implementa un metodo per il calcolo dei guadagni di ciascun post. Possiede una collezione di post analoga a quella principale. Al termine di ogni iterazione invia un messaggio ad un gruppo multicast i cui parametri sono definiti nel file di configurazione.
- **ClientsReception.java:** task che apre la connessione TCP e resta in attesa di richieste di connessioni dagli utenti, apre un threadpool e all'arrivo di una connessione immette un nuovo task (implementato con la classe *ClientHandler*) a cui passa il canale socket connesso con il client, rimettendosi in attesa di nuove connessioni.
- **ClientHandler.java:** task che si occupa della comunicazione con un client connesso, entra in un loop in cui riceve le richieste dal client, effettua i parsing necessari per ogni comando disponibile (eccetto registrazione) ed invia la risposta elaborata. All'interno sono implementati anche alcuni metodi di supporto
- **db.java:** definisce la struttura dati che ospita la collezione di utenti e post, all'avvio (costruttore) inizializza le collezioni e recupera i dati dai rispettivi file JSON. Implementa inoltre i metodi per la scrittura delle collezioni sugli stessi file .json (utilizzati dal Backuper)
- **Log.java:** implementa dei metodi di supporto per l'output di messaggi di comunicazione, errore e informazione nella console e il salvataggio di ogni messaggio sul file log.txt in modalità append.
- **Config.java:** definisce i parametri di configurazione di default e implementa la lettura dei parametri aggiornati dal file config.txt
- **Hash.java:** classe messa a disposizione nel moodle che implementa i metodi per calcolare la funzione hash di una stringa, con una breve modifica per semplificare l'invocazione. Viene usata per calcolare l'hash delle password registrate e immesse al login
- **RMICommandList.java:** classe che implementa il metodo remoto definito nell'interfaccia successiva. Il metodo remoto register implementa la registrazione di un nuovo utente nel sistema
- **RMICommandListInterface.java:** interfaccia che definisce il metodo remoto della registrazione
- **ServerFollowersUpdImpl.java:** classe che implementa i metodi remoti dell'interfaccia successiva. Implementa una struttura dati che ospita gli stub dei client associati allo username con cui hanno fatto il login, e i metodi per effettuare o cancellare la registrazione (ad uso remoto del client), e per notificare l'aggiunta di uno o più follower o la rimozione di uno (ad uso del server). Questi ultimi al loro interno invocano i metodi remoti implementati dal client e definiti nell'interfaccia *CliFollowersUpdInterface*
- **ServFollowersUpdInterface.java:** interfaccia che definisce i task implementati nella classe precedente ad uso remoto del client (registrazione e cancellazione della registrazione)
- **CliFollowersUpdInterface:** interfaccia che definisce i metodi remoti implementati dal client per l'aggiunta di uno o più followers, la rimozione di un follower o per ottenere lo username con cui il client ha effettuato il login (utile al server per associarlo allo stub ricevuto dal client)
- **Comment.java:** definisce il tipo di dato che rappresenta il commento al post, formato da username e testo, al suo interno contiene i metodi per restituire il commento o l'autore, oppure effettuare una deep-copy del commento.
- **Post.java:** definisce il tipo di dato che rappresenta il post, che comprende: id, autore, titolo, contenuto, lista degli username di chi ha votato positivamente, negativamente, di chi ha fatto rewin al post e lista dei commenti. Al suo interno sono state implementati diversi metodi per aggiungere iterazioni e restituire informazioni sul post. Include anche un metodo per la deep-copy del post.
- **User.java:** definisce il tipo di dato che rappresenta l'utente, comprende: username, password (hashed), lista dei tag, lista di chi segue e da cui è seguito l'utente, lista delle transazioni e valore del portafoglio. Al suo interno sono stati implementati i metodi per aggiungere o rimuovere utenti seguiti o che seguono, per aggiungere modifiche relative al portafoglio o per restituire informazioni sull'utente.
- **Transaction.java:** definisce il tipo di dato che rappresenta la transazione del portafoglio, comprende il valore e il timestamp della transazione oltre due metodi per restituire ciascun valore.
- **config.txt:** file di configurazione

- **log.txt:** file di log, contiene tutti i messaggi generati durante l'esecuzione relativi alle operazioni del server a partire dal primo avvio.
- **users.json/posts.json:** file JSON nel quale viene effettuata la serializzazione (o backup) delle collezioni degli utenti/post e da cui viene effettuata la deserializzazione delle informazioni all'avvio del server
- **Server.jar:** applicazione del server

Compilazione ed esecuzione

I due sistemi Server e Client devono essere compilati e avviati separatamente, anche perché in un sistema reale avverrebbe in macchine diverse. Si considera dunque la compilazione a partire dalla cartella di ciascuno, quindi all'interno della cartella Client e della cartella Server.

I comandi elencati di seguito si riferiscono all'esecuzione nella CLI su sistema Windows.

Compilazione Client

- `javac *.java -d ./bin`

Compilazione Server

- `javac -cp ".;./lib/gson-2.9.0.jar" *.java -d ./bin`

Esecuzione Client

- `java -cp ".;./bin" ClientMain`

Esecuzione Server

- `java -cp ".;./bin;./lib/gson-2.9.0.jar" ServerMain`

Esecuzione applicazione .jar Client

- `java -jar Client.jar`

Esecuzione applicazione .jar Server

- `java -jar Server.jar`

Comandi disponibili

Nell'applicazione server vi è un solo comando disponibile `exit` che avvia la corretta chiusura del sistema lato server, mentre nel client oltre la stessa exit vi sono numerosi comandi che per semplicità si rimanda all'applicazione client stessa; infatti, utilizzando il comando `help` verranno stampati nella console tutti i comandi disponibili.

Da tenere presente che all'avvio del client e prima del login, non vi è alcun utente autenticato pertanto i comandi disponibili mostrati saranno diversi da quelli mostrati dopo il login.