

Relazione Progetto WORTH

Questo progetto è la mia implementazione di **WORTH**.

È diviso in Server e Client e prevede l'interazione di quest'ultimo tramite **CLI**.

Durante lo sviluppo ho utilizzato [IntelliJ](#) su Windows per la scrittura e il debug ma la parte final di testing è stata eseguita su Linux con [WSL](#) e [Raspbian](#).

Ho utilizzato 2 librerie esterne:

- [Apache Commons CLI](#) v1.4 per il parsing degli argomenti.
- [Jackson](#) v2.11.3 per la serializzazione/deserializzazione utenti e progetti su/da file **JSON**.
 - Le classi da serializzare/deserializzare implementano **Serializable** ma ho preferito far generare e associare automaticamente **serialVersionUID** a Java omettendone la dichiarazione.

Contenuto, compilazione ed esecuzione

Nella root del progetto sono presenti, oltre a questa relazione:

- Una directory **Server** con al suo interno
 - I file **.java**
 - La cartella **lib** che contiene i files **.jar** delle 2 librerie esterne
 - Il file bash **run.sh** per la compilazione ed esecuzione automatica del **Server**
 - La cartella **build** per i files **.class**
- Una directory **Client** con al suo interno
 - I file **.java**
 - La cartella **lib** che contiene i files **.jar** della libreria **Apache Commons**
 - Il file bash **run.sh** per la compilazione ed esecuzione automatica del **Client**
 - La cartella **build** per i files **.class**
- Un **Makefile** con 8 phony target:
 - **both** (o **make** senza argomenti) per compilare sia Server che Client
 - **compileServer** per compilare il Server
 - **compileClient** per compilare il Client
 - **runServer** per eseguire il Server
 - **runClient** per eseguire il Client
 - **Server** per compilare ed eseguire il Server
 - **Client** per compilare ed eseguire il Client
 - Nei 4 phony target precedenti qualsiasi argomento (più dettagli su quelli disponibili in seguito) che si desidera passare va specificato sotto forma di variabile del **Makefile**.
 - Es. `make Server ADDRESS=192.168.1.100 PORT=19999 DEBUG=-d`
 - Es. `make Client ADDRESS=192.168.1.100 PORT=19999`
 - Se non specificato **ADDRESS=127.0.0.1**, **PORT=10002** e **DEBUG** (presente solo nel Server) è **nullo**
 - Ho deciso di utilizzare di default l'indirizzo di **loopback** per rendere il più immediato possibile il testing in qualsiasi situazione di rete.
 - **clean** per eliminare tutti i dati serializzati (cioè la cartella **data** nella cartella Server)
- Un **PDF** con uno schema (riportato anche a fine relazione) delle componenti del progetto in fase di esecuzione e delle loro interazioni.

Qualora si volessero separare le directories **Client** e **Server** rispetto a come vengono presentate, lo script **run.sh** al loro interno permette di compilare ed eseguire il codice comodamente. Qualsiasi argomento passato a **bash run.sh** diventa argomento dell'esecuzione del programma (si veda il parsing degli argomenti più avanti).

Es.

```
$ cd Server/  
$ bash run.sh -h 192.168.1.100 -p 10003 -d
```

Di default ridirigo il risultato della compilazione (quindi i files **.class**) in una cartella (interna sia a **Client** che a **Server**) chiamata **build**, per questione di ordine. Per farlo utilizzo il flag **-d**, il quale non prevede però la creazione della directory (anche se da i miei test risulta il contrario) quindi è già presente in origine.

Se si desidera compilare ed eseguire manualmente il codice è necessario specificare il **CLASSPATH** che il compilatore e l'esecutore dovranno considerare per trovare le dipendenze delle librerie (**Jackson** e **Apache Commons**) utilizzate nel codice.

Lato Server

E' composto dalle seguenti classi e interfacce:

- **MainServer**: Classe contenente il metodo **main** da eseguire se si vuole avviare il server.
- **RegisterService**: Classe che estende **RemoteServer** e implementa **RegisterServiceInterface**, contiene il metodo per la registrazione tramite **RMI** degli utenti.
- **ServerUpdater**: Classe che estende **RemoteServer** e implementa **ServerUpdaterInterface**, contiene metodi per le **RMI callback** con le quali fare l'aggiornamento sullo status degli utenti e permettere agli utenti di joinare/unjoinare le chat dei progetti.
- **LoginService**: Classe che estende **Thread**. Viene startata e utilizza una **ThreadPool** per servire i client che si connettono su connessioni **TCP**.
- **ClientHandler**: Classe che implementa **Runnable**. Viene eseguita su un thread della **ThreadPool** del **LoginService** ogni volta che un client si connette.
- **ClientUpdaterInterface**: Interfaccia sulla quale **ServerUpdater** (il quale ha un'**HashMap** per contenere oggetti che la implementano) effettua le callback.
- **Card**: Classe che rappresenta una card. Mi sono limitato a introdurre attributi riportati nella specifica. La lista di appartenenza corrente è indicata con l'attributo String **status**.
- **Progetto**: Classe che rappresenta un progetto. Data l'univocità del nome della carta all'interno di un progetto ho utilizzato un'**HashMap** con key **nomeCard** per immagazzinare le cards presenti all'interno di esso. Per immagazzinare i membri del progetto, identificandoli qui come stringhe, ho optato per una **LinkedList**.
- **Utente**: Classe che rappresenta un utente. La password è salvata e scambiata sempre in chiaro. Non ho ritenuto nello scope di questo corso doverla proteggere. Sarebbe sufficiente combinarla con un **salt** di lunghezza fissa generato casualmente, fidarsi del processo di registrazione e utilizzare un'immagine della password e del **salt** combinati per verificare che chi cerca di accedere sia a conoscenza della password (similmente a come viene effettuato su Linux).
- **DataBaseUsers** e **DataBaseProjects**: Classi che rappresentano rispettivamente lo stub di un database di utenti e di progetti.

L'esecuzione inizia dal **MainServer**. All'interno della classe vengono definite e inizializzate alcune costanti e variabili che saranno poi accessibili alle altre classi dipendenti da questa. Qui, come in **MainClient**, ho utilizzato la keyword **protected** per permetterne l'accesso alle altre classi e le keywords **static** e **final** per creare solo un oggetto per variabile e renderlo immutabile.

La prima operazione effettuata all'interno del main è il parsing degli argomenti con **argParser**. Ne ho introdotti 3:

- **-h, --host** : Indirizzo sul quale avviare il server. Opzionale, **127.0.0.1** di default.
- **-p, --port** : Porta sulla quale fare il bind della socket. Opzionale, **10002** di default.
- **-d, --debug** : Modalità di debug nella quale verranno stampati informazioni e messaggi di errore. Opzionale, **false** di default.

Ho optato per la gestione esplicita (anche lato client) solamente dell'indirizzo e porta del server, li altri valori delle porte si ipotizza che siano fissi e noti a entrambe le parti.

Controllo se posso aprire il server con i dati input (provando a fare il bind di una **ServerSocket** e controllando le collisioni delle altre porte fissate).

Deserializzo utenti e progetti (se presenti). Grazie all'uso di 2 oggetti stub (**usersDB** e **projectsDB**) posso simulare un database. Contengono entrambi un'unica lista (**ArrayList** per li utenti e **LinkedList** per i progetti) che uso come riferimento del metodo **readTree** della libreria **Jackson** per ottenere tutte le entries che ho serializzato. Le operazioni di scrittura avvengono tutte in blocchi **synchronized** e sono sicuramente costose (a ogni modifica vado a riscrivere l'intero stub) però ho ritenuto che l'integrazione di un vero database non fosse nello scope di questo corso.

Contestualmente alla deserializzazione costituisco anche 3 strutture dati (2 per li utenti e 1 per i progetti) che conterranno i valori aggiornati degli attuali utenti e progetti (li stessi valori, in ogni momento, possono essere trovati anche nei rispettivi files **JSON** ma è più comodo appoggiarsi a strutture dati sulla **JVM**).

Per gestire li utenti utilizzo 2 **HashMaps**: una contiene li oggetti **Utente** ed è accessibile solo lato server (il metodo **getPassword** è public) e l'altra, esposta (come suggerisce il nome) contenente lo status (Online/Offline) degli utenti registrati. Entrambe sono indicizzate con l'username che, essendo univoco, mi permette una ricerca efficiente all'interno della struttura dati. Non ho optato per la versione **Concurrent** delle **HashMap** perché ho la necessità di effettuare più operazioni che dovrebbero essere atomiche all'interno di esse e ciò non era comunque garantito. Ho preferito accedere a queste strutture in blocchi **sincronized** così da poterne garantire la mutua esclusione anche in caso di più accessi consecutivi.

Per i progetti uso un **HashMap** contenente li oggetti **Progetto** indicizzata con il nome del progetto (univoco, come richiedo per essere una key). Ogni accesso a questa struttura dati avviene in un blocco sincronizzato in modo che qualsiasi cambiamento al suo interno sia effettuato da un utente contemporaneamente. Avrei potuto sincronizzare le singole entry ma, andando a scrivere sul file **JSON** ad ogni modifica, avrei avuto episodi di inconsistenza dovuti alla modifica concorrente di progetti diversi con risultato la predominanza dell'ultimo thread che serializza.

Nella deserializzazione dei progetti formo anche quello che sarà il primo indirizzo di **Multicast** che andrò ad assegnare ai nuovi progetti.

Fisso come primo byte **225**, i successivi 3 bytes sono salvati in un array di interi condiviso tra li handler (**addressBytes**). Scandendo progetto per progetto ottengo i singoli indirizzi, li converto in numero decimale e li confronto con il più alto numero fin'ora raggiunto (inizialmente 0). Il massimo risultante viene riconvertito in notazione puntata e verrà usato come base dagli handler dei client per assegnare (in modo incrementale) li indirizzi di multicast. Questo accade ad ogni riavvio del server e permette di riutilizzare li indirizzi di progetti recenti cancellati. Una limitazione di questo metodo è dovuta al fatto che l'ultimo progetto creato ancora attivo crea un gap di indirizzi inutilizzati lasciati da progetti cancellati. Riservo l'indirizzo **225.0.0.0** come indirizzo di multicast al quale tutti i client si uniscono per ricevere pacchetti di **wakeup** (più informazioni nella sezione del Client).

Registro 2 servizi di **RMI**:

Il primo, ottenibile con una **lookup** su **"UPDATE"**, espone un oggetto sul quale i client possono registrarsi per delle **callback** di **update**. Queste chiamate (**synchronized**) sono effettuate dall'oggetto **ServerUpdater** su stub **ClientUpdater** lato client e sono inerenti a:

- cambiamento di status di un utente -> **notifyStatus**
- aggiunta alla chat di progetto -> **joinChat**
- abbandono di una chat di progetto (in seguito alla cancellazione di quest'ultimo) -> **unjoinChat**
- logout -> **unjoinAll**
- invio di messaggi di wakeup -> **wakeup**

Il secondo, ottenibile con una **lookup** su **"REGISTRATION"**, espone un oggetto remoto che permette ai client che ne ottengono uno stub di chiamare il metodo **addUser** che aggiungerà, se non ancora presente, l'utente e la sua password specificati nel "database" (**usersDB**) e nella struttura dati che contiene a tempo di esecuzione li utenti registrati (**users**). Contestualmente notifico con una callback che l'utente appena registrato è offline e lo aggiungo anche all'**HashMap** esposta agli utenti (**exposedUsers**). Tutto ciò avviene sincronizzando su **users**.

Non uso **rebind** perché, in caso di avvio di un secondo server sulla stessa macchina, il primo server perde l'uso dei servizi a vantaggio del secondo. Quindi in caso di fallimento del **binding** esco dal programma (non potendo garantire ai client anche solo uno dei due servizi).

Come ultima operazione di **MainServer** starta un thread **LoginService** che gestirà le connessioni.

LoginService effettua il bind all'indirizzo e alla porta indicati nel **MainServer** dopo di che crea una **ThreadPool**. Ho utilizzato una **CachedThreadPool** per gestire in modo elastico la quantità di clients connessi.

In un ciclo `while(true)` il thread si blocca sull'**accept**, all'arrivo di una connessione ottengo la socket associata e la passo al **ClientHandler** (che implementa **Runnable**) che verrà eseguito dall'**ExecutorService**. Dopo di che il ciclo ricomincia.

Il thread appena eseguito rimarrà quindi associato al client che ha richiesto la connessione finché esso non terminerà. Ho scelto questo approccio **multithreaded** perché ritengo che per come ho pensato la classe **ClientHandler** (cioè con una variabile **currentUser**, un controllo della connessione costante, nessun **TimeOut** per la ricezione dei dati) **NIO** avrebbe solo creato confusione. In applicazioni di portata maggiore avrei utilizzato **NIO**.

La classe **ClientHandler** è il cuore del backend di **WORTH**. Li handler comunicano con i client tramite **TCP** e assolvono alla maggior parte delle funzioni alle quali un utente ha accesso.

Il metodo **run** inizia con una **try-with-resources** (così da effettuare automaticamente la close all'uscita dal blocco) sui canali (**input** e **output**) ottenuti dalla **Socket** sulla quale il client è connesso. Ho deciso di **bufferizzarli** e **wrapparli** con le classi **BufferedReader** e **BufferedWriter** perché sul canale passeranno solo caratteri, questo facilita operazioni di lettura e scrittura.

Il thread sta in attesa di leggere dati sullo stream in input. I dati che posso essere trasmessi seguono un protocollo (con relativi codici di stato) prestabilito e noto sia al client che al server. Solo dati testuali possono essere scambiati. Utilizzo i metodi **write** e **newline** (per una maggiore compatibilità in termini di **Line Separator**) di **BufferWriter** per scrivere (**flush** solo quando voglio inviare i dati esplicitamente senza dover aspettare il riempimento del buffer) e **readLine** per leggere dallo stream.

L'**Handler** tokenizza ciò che riceve e, con uno switch sulla stringa che rappresenta il comando, va a verificare le intenzioni dell'utente. Ogni comando invoca un metodo privato ad hoc che assolve alle funzioni richieste dalla specifica. Questi metodi ricevono come argomento, oltre ad eventualmente gli streams **writer** e **reader**, ciò che il client ha comunicato all'handler (escluso il comando stesso). Il parsing dell'input è effettuato lato server in modo da alleggerire il client che verrà avisato, tramite codici di stato concordati, dell'esito delle richieste.

I comandi sono:

- **login** : Comando per effettuare il login di un utente.
 - Controllo la correttezza dell'input
 - Rimuovo li spazi prima dell'username
 - Non sono ammessi spazi all'interno dell'username
 - Qualsiasi stringa non vuota dopo il primo spazio diventa la password (spazi compresi).
 - Sincronizzo **users**
 - Controllo se l'utente sia tra quelli registrati e se la password fornita corrisponda a quella salvata.
 - Sincronizzo **exposedUsers**.
 - Controllo se l'utente è già loggato su un altro client.
 - Inserisco il **currentUser** (Stringa che rimarrà valida per tutta la durata dell'handler) e tra li **exposedUsers** in modo che tutti possano verificare che abbia loggato.
 - Comunico al client tutte le coppie **<username,status>** (separati ds **newline**) che ho attualmente in modo che si possa ricostruire la situazione attuale di tutti li utenti registrati.
 - Tramite un RMI avviso tutti li utenti attualmente registrati al servizio di **callback** che **currentUser** è online.

- Attendo un “ok” da parte del client (dopo l’avvenuta registrazione al servizio di callback) dopo di che, per ogni progetto di cui è membro, effettuo una callback che lo farà registrare al relativo servizio di chat.
 - Invio un “done” per stabilire il completamento del login.
- **logout** : Comando per effettuare il logout di un utente
 - Controllo se **currentUser** è diverso da **null** (questo controllo salvo problemi di comunicazione è superfluo)
 - Sincronizzo **exposedUsers**
 - Setto la entry corrispondente a **currentUser** a **false**.
 - Chiamo **update** di **ServerUpdater** che effettuerà le callback su ogni client registrato per esse e informerà del cambiamento di stato di **currentUser**.
- **createProject** : Comando per creare un progetto
 - Controllo la correttezza dell’input
 - Nel nome del progetto sostituisco gli spazi con degli underscore e rimuovo tutti i caratteri non alfanumerici (incluso anche . - _) in modo da non creare problemi durante la creazione della cartella col nome del progetto.
 - Sincronizzo **progetti**
 - Controllo se un progetto con lo stesso nome è già presente
 - Costruisco l’indirizzo di multicast della chat a partire dall’ultimo assegnato (tramite l’array **addressBytes**) controllando che non abbia esaurito gli indirizzi.
 - Aggiungo il nuovo progetto a **progetti**.
 - Serializzo
 - Creo una cartella con lo stesso nome del progetto
 - Aggiungo il progetto allo stub **projectsDB** e chiamo **writeValue** sull’**ObjectMapper**.
 - Se fallisco la serializzazione restituisco un codice di errore adeguato e ripristino lo stato precedente di **projectsDB** e **progetti**
- **cancelProject** : Comando per la cancellazione di un progetto
 - Controllo la correttezza dell’input
 - Sincronizzo **progetti**
 - Controllo se il progetto esiste, se **currentUser** (il richiedente) è membro del progetto e se tutte le carte presenti nel progetto abbiano status **DONE**.
 - Rimuovo il progetto dal database e da **progetti**
 - Serializzo
 - Se fallisco la serializzazione ripristino i cambiamenti e restituisco un errore
 - Elimino ricorsivamente i files delle cards nella directory del progetto e la directory stessa.
- **listProjects** : Comando per mostrare i progetti dei quali **currentUser** è membro
 - Sincronizzo **progetti**
 - Itero su tutte le entry di **progetti** e comunico al client il nome che lo contengono tra i membri.
- **addMember** : Comando per aggiungere un utente ai membri di un progetto
 - Controllo la correttezza dell’input
 - Ottengo il nome del progetto e l’username dell’utente da aggiungere
 - Controllo che l’utente esista
 - Sincronizzo **progetti**
 - Controllo che il progetto esista, che il richiedente sia membro e che l’utente da aggiungere non sia già membro
 - Aggiungo l’utente
 - Serializzo
 - Se fallisco rimuovo l’utente dai membri e restituisco un codice di errore
- **showMembers** : Comando per ottenere la lista dei membri di un progetto
 - Controllo la correttezza dell’input
 - Sincronizzo **progetti**
 - Controllo se il progetto esiste e se il richiedente ne è membro

- Itero sulla lista dei membri e ne invio li username al client separati da **newLine**.
- **addCard** : Comando per aggiungere una card a un progetto
 - Controllo la correttezza dell'input
 - Nel nome del progetto e della carta sostituisco li spazi con degli underscore e rimuovo tutti i caratteri non alfanumerici (include anche . - _) in modo da non creare problemi durante la creazione della cartella col nome del progetto.
 - Nella descrizione non sono ammessi spazi
 - Sincronizzo **progetti**
 - Controllo che il progetto esista, che il richiedente ne sia membro e che non vi sia una carta con lo stesso nome al suo interno
 - Serializzo
 - Creo un file con lo stesso nome della card, fungerà da storico delle liste, e tramite **FileOutputStream** scrivo "**TODO**" (in quanto tutte le carte partono da quella lista).
 - Se fallisco ripristino lo stato precedente.
 - Notifico tramite messaggio **Multicast** sulla chat del progetto l'aggiunta di una nuova card.
 - Se fallisco la notifica l'inserimento va comunque a buon fine ma avverto il client.
- **showCards** : Comando per ottenere i nomi e la lista delle cards di un progetto
 - Controllo la correttezza dell'input
 - Sincronizzo **progetti**
 - Controllo che il progetto esista e che il richiedente appartenga al progetto.
 - Itero sulle cards del progetto specificato e invio nome e descrizione separati da **newLine** per ogni card.
- **showCard** : Comando per ottenere le informazioni di una carta di un progetto
 - Controllo la correttezza dell'input
 - Sincronizzo **progetti**
 - Controllo che il progetto esista, che il richiedente ne sia membro e che la carta sia presente.
 - Invio nome, descrizione e lista di appartenenza separati da **newLine**.
- **moveCard** : Comando per spostare una card all'interno delle liste di un progetto
 - Controllo la correttezza dell'input
 - Sincronizzo **progetti**
 - Controllo che il progetto esista, il richiedente ne sia membro e che la carta vi sia contenuta.
 - Controllo che le liste di partenza e di arrivo rientrino tra quelle stabilite e i vincoli imposti dall'automa a stati finiti.
 - Se il controllo ha successo chiamo una funzione ausiliaria **auxmoveCard** (creata con lo scopo di evitare di scrivere lo stesso codice più volte) che effettua la serializzazione.
 - Se la serializzazione non va a buon fine ripristino la lista di partenza.
 - Se la serializzazione va a buon fine notifico sulla chat del gruppo lo spostamento della card.
- **getCardHistory** : Comando per mostrare lo storico delle liste di una card
 - Controllo la correttezza dell'input
 - Sincronizzo **progetti**
 - Controllo che il progetto esista, che il richiedente sia membro e che la card esista.
 - Wrappo in un **BufferedReader** un **FileReader** con filename il path della carta.
 - Leggo fino all'**EOF** tutte le righe contenenti lo storico degli stati (dal meno recente al più recente) e le invio una ad una, separate da **newLine**, al client.

Nel caso in cui il client interrompa la connessione "correttamente" (dopo la procedura di logout) l'**handler** termina senza problemi. Se invece il client viene chiuso forzatamente (es. **SIGKILL**) non ho modo di disiscriverlo dalle **callback** e continuerà a risultare online fino al prossimo riavvio del server. Nel caso in cui il client riceva **SIGINT** o **SIGTERM** riesce comunque ad effettuare il logout grazie a uno **ShutdownHook** (più dettagli in seguito).

Lato Client

È composto dalle seguenti classi e interfacce:

- **MainClient**: Classe contenente il metodo **main** da eseguire se si vuole avviare il client.
- **ClientUpdater**: Classe che implementa **ClientUpdaterInterface**. Un'istanza verrà registrata per le callback sull'oggetto remoto **ServerUpdater**. Implementa metodi per l'aggiornamento dello status degli utenti e la gestione della chat.
- **RegisterServiceInterface** : Interfaccia della quale ottenere uno stub sul quale chiamare il metodo **addUser** per la registrazione di un nuovo utente.
- **ServerUpdaterInterface** : Interfaccia della quale ottenere uno stub sul quale chiamare il metodo **registerForCallback/unregisterForCallback** (a seconda se si sia nella fase di **login** o **logout**).

L'esecuzione del **Client** comincia dal **main**. Prima però definisco alcune costanti e variabili statiche tra cui un **InputStream** (wrappato in un **BufferedReader**) da **stdin** per l'inserimento dei comandi dell'utente.

Effettuo il parsing degli argomenti con la funzione **argParser**. Ne ho introdotti 2:

- **-h, --host** : Indirizzo al quale collegarsi. Opzionale, **127.0.0.1** di default.
- **-p, --port** : Porta alla quale collegarsi. Opzionale, **10002** di default.

Non ho introdotto il flag **-d** per la modalità **debug** (come per il **server**) perché non trovo giusto che l'utente riceva messaggi di errore complessi e riguardanti il codice. È però possibile settare **DEBUG** a **true** nel codice e ottenere maggiori informazioni sui messaggi di errore.

Sullo **stdout** viene proposto all'utente un menù con 4 comandi. Ciascuno può essere acceduto digitando il numero associato, la sua prima lettera o il nome completo (non case-sensitive). Dopo di che, a seconda del comando scelto, il sistema chiede esplicitamente all'utente i parametri da inserire. Ho preferito questo approccio (valido anche per i comandi degli utenti loggati) rispetto a scrivere comando e input sulla stessa riga in modo da semplificare e velocizzare il testing e l'utilizzo.

L'input richiesto all'utente sarà preceduto da **>**, le risposte di **WORTH** (non necessariamente in seguito a interazioni col server) saranno precedute da **<**, errori tecnici saranno preceduti da *****.

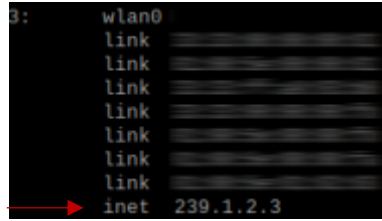
I comandi disponibili sono:

1. **Register** : Permette all'utente di registrare un nuovo username.
 - Dato che il metodo per la registrazione è comune a tutti i client ed è **synchronized** ho preferito appesantire il client del parsing dell'input.
 - Dopo aver estrapolato dall'input **username** e **password** da registrare ottengo il registro alla porta **RMIPORT** e faccio una **lookup** di "**REGISTRATION**". L'oggetto che ottengo (castando il risultato della **lookup** con l'interfaccia **RegisterServiceInterface**) sarà lo stub sul quale effettuerò **RMI**.
 - La chiamata **addUser** restituirà **true** se è andata a buon fine o **false** altrimenti.
2. **Login** : Inizia la procedura per l'accesso dell'utente, maggiori dettagli in seguito.
3. **Help** : Verranno mostrate delle indicazioni sull'utilizzo di questi comandi.
4. **Quit** : Termina il programma con **exit code 0**.

Se l'utente decide di accedere viene avviata la procedura di login.

Il client si collega al server e ne ottiene lo **stream (input e output)** dalla **Socket**. Volendo utilizzare **NIO** (più informazioni in seguito) per la gestione delle chat mi sono documentato sull'utilizzo delle **NewtworkInterface**. Dato che è sempre possibile utilizzare l'interfaccia di **loopback** la setto di default e la utilizzo per l'invio di datagrammi su gruppi **multicast** (la **join** verrà effettuata dal **ClientUpdater**, più dettagli in seguito). In questo modo garantisco a tutti i client sulla stessa macchina di ricevere e inviare datagrammi UDP multicast. Nonostante non fosse richiesto ho provato ma non sono riuscito a inviare/ricevere datagrammi da/ad altre macchine (sempre sulla stessa rete) quindi ho preferito non cercare interfacce diverse da quella di loopback (lascio comunque il codice commentato per completezza). Nel prossimo paragrafo descrivo come ho potuto verificare ciò.

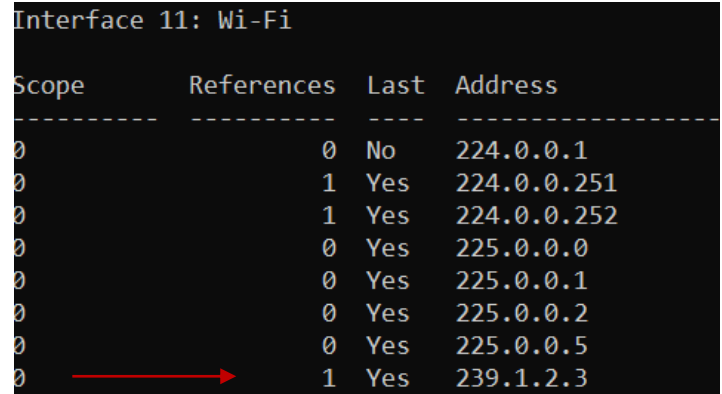
Col comando **ip maddr show** (Linux, sinistra) e **netsh interface ip show joins** (Windows, destra) ho potuto riscontrare che effettivamente, anche



```

# ip netns exec
0:      wlan0
      link
      link
      link
      link
      link
      link
      link
      inet 239.1.2.3
  
```

utilizzandoli interfacce non di loopback, degli sniffer di prova fossero joinati correttamente ai gruppi ma sono stati in grado di ricevere datagrammi solo da sender sulla stessa macchina.



Scope	References	Last	Address
0	0	No	224.0.0.1
0	1	Yes	224.0.0.251
0	1	Yes	224.0.0.252
0	0	Yes	225.0.0.0
0	0	Yes	225.0.0.1
0	0	Yes	225.0.0.2
0	0	Yes	225.0.0.5
0	1	Yes	239.1.2.3

Inizializzo anche un **DatagramChannel** che utilizzerò per inviare datagrammi UDP in multicast. Istanziò un **HashMap** di utenti (indicizzata con lo username e contenente lo status attuale associato) che mi permetterà di controllare localmente li utenti registrati (i valori saranno sempre aggiornati grazie alle callback di **ClientUpdater**).

Le chat vengono salvate in una **CocurrentHashMap** indicizzata col nome del progetto contenente **LinkedLists** di **String** che rappresentano, in ordine di ricezione, i messaggi della chat del progetto rappresentato dalla key (maggiori informazioni sul loro inserimento in seguito).

Ho deciso di salvare localmente anche un'associazione tra nome del progetto e il suo indirizzo di **multicast** nella **ConcurrentHashMap multicastGroup** in modo da non dover interrogare il server ad ogni invio di messaggio.

La scelta di una **ConcurrentHashMap** per queste ultime 2 strutture dati è dovuta la fatto che in ogni situazione viene effettuato solo un accesso quindi, nel caso pessimo, sarà sufficiente ripetere il comando per ottenere il valore aggiornato (ad esempio nella lettura di una chat per la quale non è ancora presente una entry).

Il metodo **login** prende in input username e password e li comunica al client preceduti dal nome del comando che dovrà interpretare il server. Questa forma protocollare è utilizzata in tutti i metodi che comunicano col client.

Con uno switch sullo status code di risposta del server valuto l'esito della richiesta di login. In caso di riscontro positivo (codice "0") ricevo dal server successioni di username e status attuale per aggiornare l'**HashMap allUsers** per far sì che sia identica alla situazione riscontrabile nel server e da tutti i client. La ricezione di dati termina con "end".

Segue l'ottenimento di un oggetto remoto tramite lookup su "UPDATE" sul quale chiamare il metodo **registerForCallback** dando come parametro un oggetto **ClientUpdater** che verrà registrato per le callback da parte del server. Chiamando il metodo **registerForCallback** sullo stub fornisco anche l'username corrente in modo che il mio oggetto **ClientUpdater** possa essere indicizzato nell'**HashMap clients** di **ServerUpdater**, questa faciliterà le update mirate a singoli utenti. Fatto ciò il client invia "ok" al server che farà chiamate **makeJoin** su ogni progetto del quale l'utente è membro per far joinare la chat. Solo dopo la ricezione di "done" da parte del server l'utente potrà considerarsi effettivamente loggato. Quindi la funzione restituirà **currentUser** ("errore" in tutti li altri casi, non potendo contenere spazi questo username fittizio verrà riconosciuto dal ciclo di inserimento dei comandi degli utenti loggati e chiuderà la connessione col server).

L'oggetto **ClientUpdater** appena creato contiene i metodi che invocherà come callback il **ServerUpdater**. Alla sua creazione viene creato un **DatagramChannel** per l'invio di messaggi di **wakeup**, una **LinkedList** si stringhe **toJoin** (che conterrà li **InetAddress** dei gruppi che devono ancora essere joinati dal client) e avviato un thread **Listener** che effettuerà **demultiplexing** con **NIO** sui datagrammi in arrivo dalle chat. Eredita dal **MainClient** li oggetti di cui ha bisogno.

In questa classe sono presenti 5 metodi sui quali il server può fare le callback:

- **notifyStatus** : Metodo per aggiornare lo status di un utente nella struttura dati interna al client.
 - Viene invocato in seguito alla registrazione di un nuovo utente o dopo il login/logout di un utente registrato.
- **joinChat** : Metodo per aggiungere un client al gruppo multicast di una chat.
 - Viene invocato una volta per ogni progetto al quale partecipa l'utente appena loggato e, in seguito, ogni volta che crea un progetto o verrà aggiunto a uno già esistente.
 - Viene aggiunta una nuova entry nelle **chats** (come da specifica non è possibile risalire a messaggi precedenti al login), viene aggiunta una nuova entry in **multicastGroups** (così da poter ottenerne l'indirizzo con un accesso),

viene aggiunto l'**InetAddress** corrispondente alla chat che dovrà essere joinata, infine viene inviato un messaggio di **wakeup**.

- **unjoinChat** : Metodo per lasciare la chat di un gruppo.
 - In caso di rimozione del progetto vengono semplicemente rimosse le entry indicizzate con esso da **chats** e **multicastGroups**. Nessun altro progetto durante questa sessione del server potrà avere lo stesso indirizzo (in caso di riavvio sì ma a quel punto i client dovranno rieffettuare il login e il conseguente join su tutti i gruppi).
- **unjoinAll** : Metodo per lasciare tutte le chat di progetto. Viene invocato con il **logout**.
 - Lancia un'interruzione al **listener** e manda un messaggio di **wakeup**. Non ho necessità di chiamare una **cancel** del **DatagramChannel** su ogni gruppo perché l'utente sta effettuando il logout e a un nuovo login il **DatagramChannel** verrà resettato.
- **wakeup** : Metodo per l'invio di un messaggio di **wakeup**.
 - Ho introdotto questa tecnica per risolvere il problema che si creava quando, aggiungendo un utente online a un progetto, questo non riusciva a joinare perché il **listener** era attivo ma fermo in attesa di datagrammi su gruppi già joinati. Inviando però un datagramma a un gruppo comune al quale tutti partecipano di default (225.0.0.0, non assegnabile ad alcun progetto) si può "svegliare" il ciclo di **NIO** che, come prime istruzioni, prevede il check della propria interruzione (in caso di **interrupt** da **unjoinAll**) e il join dei gruppi ancora da joinare effettuando uno **spooling** della **LinkedList toJoin**. Questo meccanismo non crea problemi a chi non ha bisogno di un messaggio di wakeup perché, essendo esso composto solamente da ".", verrà scartato nella valutazione del contenuto del **DatagramChannel**.
 - Questa strategia in combinazione con **NIO** mi ha permesso di utilizzare un solo thread che sta in ascolto su tutti i gruppi di multicast invece di adoperare una soluzione "forza bruta" che avrebbe previsto, per ogni utente, l'avvio di un **thread daemon** per ogni chat della quale fosse membro.

Prima di mostrare i comandi disponibili agli utenti installo uno **ShutdownHook** (solo se l'utente ha loggato con successo). Esso è rappresentato da una sottoclasse di **MainClient** che estende **Thread**. Una sua istanza viene associata al runtime corrente. La **JVM** manderà in esecuzione l'**hook** che effettuerà la procedura di **logout** in seguito al ricezione di un segnale **SIGINT** o **SIGTERM**, evitando così che il **ServerUpdater** abbia una entry nulla in **client** (come accade invece in caso di terminazione per **SIGKILL**).

Se il login è effettuato con successo verrà presentato all'utente un menù di comandi disponibili. Li ho divisi per pertinenza (welcome, utenti, progetti, cards e chat) sia nella loro enumerazione che nel codice in modo che fosse più facile individuare il comando richiesto. Con uno **switch** sulla scelta viene matchato il case richiesto e chiamato il metodo corrispondente:

1. **List Users** : Mostra li utenti registrati
 - Dato che la lista aggiornata di utenti e il loro stato è mantenuta in locale da ogni client basta ottenere uno "snapshot" dell'**HashMap allUsers** e iterare su di esso. Non utilizzo **allUsers** stesso per evitare un'eccezione di modifica concorrente in caso di aggiornamento della struttura in seguito a un login o una registrazioni.
2. **List Online Users** : Mostra li utenti registrati attualmente online
 - Simile a **List Users** ma stampo solo li utenti il quale stato risulta attualmente **true** (Online).
3. **Create Projects** : Crea un nuovo progetto
 - Viene richiesto l'inserimento del nome del progetto che viene poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito dell'inserimento verrà mostrato un messaggio di risposta sul client.
 - Se la creazione ha successo il creatore viene aggiunto alla chat con una chiamata **makeJoin** al **ServerUpdater** da parte del **ClientHandler** con cui si sta comunicando. Questo provocherà una callback a **joinChat** con li effetti sopra descritti.
4. **List Projects** : Mostra i progetti di cui **currentUser** è membro
 - Invio la richiesta al server senza argomenti e ottengo, nome per nome, tutti i progetti di cui sono membro.
5. **Add Member** : Aggiunge un membro a un progetto
 - Viene richiesto l'inserimento del nome del progetto e dell'utente da aggiungere. Tutto ciò verrà poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.

- Se l'operazione ha successo il nuovo membro viene aggiunto alla chat con una chiamata **makeJoin** al **ServerUpdater** da parte del **ClientHandler** con cui si sta comunicando. Questo provocherà una callback a **joinChat** dell'oggetto **ClientUpdater** appartenente al membro aggiunto (se online) con li effetti sopra descritti.
6. **Show Members** : Mostra i membri di un progetto
- Viene richiesto l'inserimento del nome del progetto che viene poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Se l'operazione ha successo verrà inviato dal server l'elenco di progetti dei quali **currentUser** è membro.
7. **Cancel Projects** : Elimina un progetto
- Viene richiesto l'inserimento del nome del progetto che viene poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Se l'operazione ha successo l'handler invocherà il metodo **makeUnJoin** del **ServerUpdater** su ogni membro del progetto. La callback sortirà li effetti sopra descritti per ogni utente online interessato dalla cancellazione.
8. **Add Card** : Aggiunge una card a un progetto.
- Viene richiesto l'inserimento del nome del progetto, il nome della card e la sua descrizione. Tutto ciò viene poi inviato, antepoendo il nome del comando, al server.
 - Non sono ammessi spazi o caratteri speciali né per il nome della card né per la sua descrizione
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Il Server provvede anche alla notifica di inserimento sulla chat del progetto (nella specifica è richiesta la notifica solo dello spostamento di una card ma ho ritenuto che fosse corretto notificare li utenti online anche dell'inserimento di una nuova card dato che la lista **TODO** subisce una variazione).
9. **Show Cards** : Mostra nome e lista di tutte le cards di un progetto.
- Viene richiesto l'inserimento del nome del progetto che viene poi inviato antepoendo il nome del comando al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Se l'operazione ha successo verrà inviato dal server l'elenco di valori richiesti.
10. **Show Card** : Mostra le informazioni di una carta di un progetto.
- Viene richiesto l'inserimento del nome del progetto e il nome della card. Tutto ciò viene poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Se l'operazione ha successo verranno inviati dal server i valori richiesti.
11. **Move Card** : Sposta una carta all'interno di un progetto
- Viene richiesto l'inserimento del nome del progetto, il nome della card, la lista di partenza e quella di destinazione. Tutto ciò viene poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Il Server provvede anche alla notifica dello spostamento sulla chat del progetto.
12. **Get Card History** : Mostra lo storico delle liste di una card
- Viene richiesto l'inserimento del nome del progetto e il nome della card. Tutto ciò viene poi inviato, antepoendo il nome del comando, al server.
 - A seconda dell'esito della richiesta verrà mostrato un messaggio di risposta sul client.
 - Se l'operazione ha successo verrà inviato dal server l'elenco di valori richiesti.
13. **Send Chat Message** : Invia un messaggio sulla chat di un gruppo
- Viene richiesto l'inserimento del nome del progetto e il messaggio. Li iso lo entrambi in 2 variabili.
 - Controllo se l'indirizzo di multicast per il progetto richiesto è presente in **multicastGroups**.
 - Se così non fosse o non sono membro del progetto o non esiste, avviso l'utente.
 - Altrimenti invio un messaggio all'InetSocketAddress giusto (combinò l'indirizzo specificato con la porta nota).
 - Tutti i messaggi, compresi quelli inviati a seguito di modifica delle liste, sono formattati in modo che i threads listener possano destinarli correttamente i messaggi nelle chat.
 - Es. `call_skype_quarantena: drake: "bro, nice job, cya l8r"`

- `Es.call_skype_quarantena`: Callie ha spostato la card `preparare_meme` da `inprogress` a `done`

14. **Read Chat** : Mostra i messaggi di un progetto non ancora letti che sono stati inviati da quando è stato effettuato l'ultimo login.

- Viene richiesto l'inserimento del nome del progetto.
- Controllo se la chat (**LinkedList** di stringhe) è presente nelle chat utilizzando come indice il nome del progetto specificato.
 - In caso di esito negativo (**tmpChat** nullo) **currentUser** non fa parte del progetto o il progetto richiesto non esiste.
 - Altrimenti apro un blocco **synchronized** sulla chat del progetto selezionato in modo che possa finire di scandirla prima di inserire nuovi messaggi al suo interno (questo blocca eventualmente solo il **listener** del client che avrà comunque modo di riprendere la lettura su altri canali dopo questa attesa).

15. **Logout** : Effettua il logout di **currentUser**.

- Invio al server il comando **logout**.
- Chiamo il metodo **unregisterForCallback** di **ServerUpdater** per non essere più aggiornato (altrimenti avrebbe la entry associata a **currentUser** uguale a **null** in **clients**).
- Rimuovo lo **ShutdownHook**.
- Setto il flag **exit** a **true** per evitare di rientrare nel ciclo di inserimento dei comandi.

16. **Help** : Stampa una del testo di supporto all'utente.

- Ho inserito anche una raffigurazione in ASCII dell'automa a stati finiti fatta su asciiflow.com.

Ogni metodo che comunica con il **Server** ha nella firma **throws IOException**, questo perché se durante qualsiasi fase della comunicazione con l'handler il Server viene chiuso letture e scritture solleveranno un'eccezione. Questa galleggerà fino al main dove verrà catchata e verrà presentato un messaggio di errore all'utente. In seguito a questo scenario verrà riproposta la schermata di login e registrazione (non ho la necessità di effettuare la corretta procedura di **logout** perché il server non avrebbe modo di completarla essendo down). Ho catchato sia **SocketException** che **NullPointerException** perché su Windows (IntelliJ) ho riscontrato comportamenti diversi rispetto a Linux e questa combinazione di eccezioni mi ha permesso di assecondare entrambi.

Per testare le situazioni di concorrenza ho utilizzato questa funzione scritta da me che mi ha garantito una sufficiente approssimazione di simultaneità:

```
private static void sinc() {  
    SimpleDateFormat sdf = new SimpleDateFormat("mm:ss");  
    while (!sdf.format(new Date(System.currentTimeMillis())).equals("00:00")) {}  
}
```

Con un (inefficiente) ciclo in attesa attiva vado a verificare i minuti e secondi attuali. Antepoendo una chiamata a **sinc()** prima di lavorare con più client su sezioni critiche ho potuto scovare così problemi di concorrenza e introdurre opportune sincronizzazioni.

Ho rimosso le vocali accentate per una compatibilità con il **Character Set UTF-8**.

