

Relazione Progetto Worth

Laboratorio di Reti



A cura di Maurizio Cascino
Professoressa Laura Ricci

INDICE

Introduzione	5
1. Architettura del progetto e RMI	6
1.1. PARADIGMI UTILIZZATI.....	7
1.2. Operazioni svolte tramite RMI (lato Server)	8
1.2.1. <i>Register e SendStatusClient</i>	8
1.2.2. <i>RegisterForCallBack e DeleteForCallback</i>	9
1.3. Operazioni svolte tramite RMI (lato Client).....	10
1.3.1 <i>NotifyEvent</i>	10
1.3.2. <i>NotifyProject</i>	10
1.3.3. <i>NotifyDeleteProject</i>	11
1.4. Sincronizzazione metodi remoti (lato Server).....	12
1.5. Sincronizzazione metodi remoti (lato Client)	12
2. SERVERCLASS	13
2.1. Server multiplexing con NIO	14
2.2. Strutture dati utilizzate dal server	15
2.3. Persistenza dello stato del sistema.....	16
2.3.1. <i>Serializzazione</i>	16
2.3.2. <i>Deserializzazione</i>	18
2.4. GESTOREIPMULTICAST.....	19
2.4.1. <i>GeneraMulticastIp</i>	19
3. CLIENT CLASS	20
3.1. Strutture dati utilizzate e accessi concorrenti	21
3.2. Chat	22
4. CLASSI PROGETTI, CARD E UTENTI	23
4.1. Card	24
4.2. progetti.....	24
4.3. Utenti	24

5. INFO GENERALI	25
5.1. Test del progetto	26
5.2. Come avviare il progetto da terminale?.....	26
5.3. Comando HELP.....	27

Introduzione

Il progetto Worth, che verrà presentato in dettaglio nelle prossime sezioni, mira a implementare funzionalità che possano aiutare le persone a organizzarsi e coordinarsi nello svolgimento di compiti comuni. Quest'ultimi possono essere progetti professionali o in generale qualsiasi evento che possa essere organizzato in una serie di compiti, comunemente chiamati *task*, i quali sono svolti dai membri di un gruppo.

Tale progetto s'ispira e implementa la metodologia Kanban, la quale fornisce una visione completa dei task.

Infatti, attraverso l'utilizzo di specifiche operazioni da linea di comando, l'utente potrà registrarsi al servizio e dopo aver effettuato l'accesso, potrà creare un progetto e aggiungervi nuovi utenti già iscritti al servizio. I membri dello stesso progetto potranno creare nuovi task, chiamati *cards*, le quali verranno spostate in diverse liste in base al loro stato di avanzamento dei progressi.

Infine, ad ogni progetto vi è associata una chat, con la quale i membri dello stesso progetto potranno comunicare per ottimizzare al meglio il loro lavoro.

1. Architettura del progetto e RMI

1.1. Paradigmi Utilizzati

L'architettura del sistema è basata su due paradigmi:

- **Client-Server**
- **RMI**

La fase di registrazione di un utente viene implementata tramite il meccanismo RMI. In seguito, l'utente stabilisce una connessione TCP col Server. D'ora in poi, l'utente potrà richiedere operazioni al Server Worth¹, seguendo quindi il paradigma Client-Server. Inoltre, dopo che l'utente avrà effettuato l'operazione di login, potrà ricevere aggiornamenti sullo stato degli altri utenti registrati tramite il servizio di notifica messo a disposizione dal Server Worth, implementato con il meccanismo di RMI callback. Infine, le chat dei progetti sono state realizzate usando UDP multicast. Nei prossimi paragrafi si discuterà degli altri metodi RMI messi a disposizione sia per il Client che per il Server.

¹ Tutte le operazioni disponibili verranno descritte in seguito, sia quelle effettuate tramite TCP che quelle effettuate tramite RMI.

1.2. Operazioni Svolte Tramite Rmi (Lato Server)

```
public interface ServerInterface extends Remote {
    /*Se la registrazione va a buon fine restituisce 1, altrimenti restituisce -1*/
    int register(String nickUtente, String password) throws RemoteException;

    /* Registrazione del client al servizio di aggiornamenti, offerto dal server*/
    void registerForCallback(String nickUtente, ClientInterface stub) throws RemoteException;

    /* Annulla la registrazione del client al servizio di aggiornamenti, offerto dal server */
    void deleteRegForCallback(String nickUtente) throws RemoteException;

    /* Notifica i client del cambiamento di stato*/
    void sendStatusClient() throws RemoteException;
}
```

Interfaccia del Server con i metodi remoti messi a disposizione per il client

1.2.1. Register e SendStatusClient

Il server mette a disposizione del client il metodo remoto **register**, con il quale il client, subito dopo aver recuperato dal registry lo stub dell'oggetto esportato del server, potrà effettuare la registrazione al servizio. Il server memorizzerà le credenziali del client nella struttura dati *credenziali*. L'utente, appena registrato, avrà lo stato impostato di default a "offline", mentre le informazioni riguardanti il nome utente e il suo stato verranno memorizzate nella struttura dati *statusForClient*. Quest'ultima struttura dati tornerà utile quando verrà chiamato il metodo remoto **sendStatusClient**, il quale notificherà tutti i clients già registrati al servizio di notifica del server dell'iscrizione di un nuovo utente.

Si noti che, nel caso della *registrazione*, del *login* o *logout* di un utente, il metodo *sendStatusClient* verrà chiamato dal Server per aggiornare gli altri utenti del cambiamento di stato. Invece, verrà utilizzato *come metodo remoto dal client* ogni qualvolta il client avvierà una nuova connessione col Server. Tale scelta è stata adottata affinché il client, appena connesso, possa controllare che la sessione precedente col server sia stata chiusa correttamente, altrimenti il client potrebbe trovarsi in uno stato di inconsistenza. Ad esempio, l'utente potrebbe essersi collegato e poi la sua applicazione potrebbe essersi arrestata in modo anomalo. Infatti, nel caso in cui lo stesso utente si

collegli nuovamente, non riuscirebbe ad effettuare il login, poiché il server sarebbe ignaro della disconnessione avvenuta precedentemente. Inoltre, il suo stub dovrebbe essere rimosso poiché non più raggiungibile e gli altri client dovrebbero essere avvisati del cambiamento di stato. Per tali ragioni, dopo che il client ha stabilito la connessione al servizio, invoca il metodo remoto *sendStatusClient* per ovviare a tali inconvenienti.

1.2.2. RegisterForCallback e DeleteForCallback

Il metodo remoto **registerForCallback** viene chiamato dal client al momento della login per passare il suo stub al Server e sottoscrivere l'interesse da parte del client a ricevere aggiornamenti su nuove registrazioni o cambiamenti di stato degli altri utenti. La coppia $\langle \textit{nome Utente}, \textit{stubClient} \rangle$ viene memorizzata nella struttura dati del server *tabClient* e verrà recuperata nel momento in cui il server notificherà un cambiamento di stato di un utente.

Specularmente, il metodo remoto **deleteRegForCallback** viene chiamato dal client al momento della logout o nel caso in cui la login non sia andata a buon fine. In tal caso, lo stub del client verrà rimosso dalla struttura dati *tabClient*.

1.3. Operazioni Svolte Tramite Rmi (Lato Client)

```
public interface ClientInterface extends Remote {  
    public void notifyEvent(ConcurrentHashMap<String, String> statusForClient) throws RemoteException;  
    public void notifyProject(String progetto, String multicast) throws RemoteException;  
    public void notifyDeleteProject(String progetto) throws RemoteException;  
}
```

Interfaccia del Client con i metodi messi a disposizione per il Server

1.3.1 NotifyEvent

Il client mette a disposizione del server il metodo **notifyEvent**, affinché il server possa passare all'utente la struttura dati *statusForClient*. Il client salverà tale struttura dati in locale e potrà accedervi ogni qualvolta sia necessario. Si noti che verranno creati due oggetti di *statusForClient*, uno lato server e l'altro lato client. Infatti, lato client arriverà un oggetto serializzato che verrà ricostruito e riferito da *statusClients*.

1.3.2. NotifyProject

Il metodo remoto **NotifyProject**, implementato nel client e utilizzato dal server, permette agli utenti di recuperare l'indirizzo IP multicast associato al progetto. L'utente salverà la coppia *<nome del progetto, indirizzo IP>* nella struttura dati locale *tabIp*. Inoltre, ad ogni progetto viene associato un thread, che verrà attivato tramite il metodo remoto *notifyProject* e resterà in ascolto sulla chat per ricevere eventuali messaggi da altri membri dello stesso progetto. La coppia *<nome del progetto, riferimento al thread>* verrà memorizzata nella struttura dati locale del client *tabThreads*.

NotifyProject viene richiamato dal server in 3 casi:

1. Un utente crea un nuovo progetto tramite il metodo *createProject*. All'utente verrà notificato l'indirizzo IP multicast associato al progetto appena creato.
2. Un utente viene aggiunto al progetto tramite il comando *addMember* e dovrà essere notificato del nuovo indirizzo IP multicast associato al progetto per poter inviare messaggi nella chat del progetto. Se l'utente aggiunto si trova *online*, il metodo remoto *notifyProject* verrà eseguito correttamente. Nel caso in cui l'utente aggiunto si

trovi *offline* non gli verrà notificato l'indirizzo IP associato al progetto in cui è stato aggiunto, poiché non sarà possibile recuperare il suo stub associato. Questa situazione verrà gestita nel terzo caso.

3. Ogni volta che un utente effettua l'accesso tramite il metodo *login*, controlla che sia stato aggiunto ad un progetto e richiama il metodo remoto *notifyProject* per recuperare l'indirizzo Ip associato al progetto di cui è membro.

1.3.3. NotifyDeleteProject

Il metodo remoto **NotifyDeleteProject**, implementato nel client e utilizzato dal server quando viene eseguito il metodo *cancelProject*, permette agli utenti di rimuovere l'indirizzo IP multicast associato al progetto dalla struttura dati *tabIp*. Inoltre, da *tabThreads* rimuove il thread, associato al progetto, che è in ascolto sulla chat per ricevere e memorizzare i messaggi ricevuti.

1.4. Sincronizzazione Metodi Remoti (Lato Server)

Poiché avrebbero potuto presentarsi situazioni di inconsistenza, i metodi remoti sono stati resi **synchronized**. I casi analizzati e gestiti sono:

1. Vengono attivati più clients che potrebbero invocare contemporaneamente i metodi remoti del server. Quindi, vengono attivati più **threads RMI**, uno per ogni clients, e l'accesso alle risorse è gestito correttamente.
2. Il **thread main del Server** potrebbe entrare in concorrenza con i **threads RMI**. È stata riscontrata una possibile situazione di inconsistenza quando viene chiamato il metodo login. Tale situazione è stata gestita e risolta, incapsulando il metodo login in un blocco *synchronized*.

1.5. Sincronizzazione Metodi Remoti (Lato Client)

I metodi remoti *notifyProject* e *notifyDeleteProject* sono stati resi *synchronized*, poiché il thread principale del client potrebbe entrare in concorrenza con il thread RMI, attivato dall'invocazione di un metodo remoto da parte del server. Per ovviare a tale inconveniente, il metodo *logout* è stato reso *synchronized*.

Si noti che, rispetto ai metodi RMI lato server, i metodi RMI del client non potranno mai entrare in concorrenza tra loro, poiché il server è single thread. Di conseguenza, il server potrà invocare solo un metodo remoto per volta e lato client verrà attivato solo un thread RMI.

2. SERVERCLASS

2.1. Server Multiplexing Con Nio

Nel progetto è stata effettuata la scelta di implementare un server con comportamento non bloccante che effettua il **multiplexing attraverso NIO**. Tale scelta è dettata dall'esigenza di ridurre l'utilizzo delle risorse che porterebbe un server multithreading. Infatti, con NIO multiplexing sarà possibile gestire un numero arbitrario di sockets con un singolo threads, portando miglioramenti di performance e scalabilità del servizio.

Nel metodo **connectTCP** viene creata la connessione e viene impostata la modalità non bloccante. All'inizio la selector registra solo l'operazione di tipo **accept** e nel corpo del **while(true)**, il server si blocca sulla **select()** in attesa che arrivi la prima richiesta di connessione.

In seguito, le operazioni di lettura, scrittura e nuove richieste di connessione verranno gestite dai rispettivi metodi:

- **readSocketChannel**: il canale è pronto per un'operazione di lettura. Si effettua un'operazione di lettura dal canale e si scrive il contenuto nel buffer, affinché i dati possano essere ricevuti dal Server. Infine, si registra una nuova operazione di scrittura su quel canale.
- **writeSocketChannel**: il canale è pronto per un'operazione di scrittura. Si effettua l'operazione di scrittura nel buffer associato al canale affinché i dati vengano inviati al client. Infine, si registra una nuova operazione di lettura per lo stesso canale.
- **acceptConnection**: il canale è pronto per accettare una nuova connessione. Viene creata una nuova connessione non bloccante monitorata dalla selector e associato un buffer al canale. Infine, viene registrata una nuova operazione di lettura sul canale.

2.2. Strutture Dati Utilizzate Dal Server

Le strutture dati utilizzate dal server sono state rese **thread-safe**, poiché avrebbero potuto presentarsi situazioni di inconsistenza, nel caso in cui vi fosse stato richiesto l'accesso concorrente sia dal thread principale del Server sia dai threads dei metodi remoti. Di seguito vengono presentate le strutture dati utilizzate:

- **Collection<Utenti> credenziali:** utilizzata per memorizzare le credenziali di un utente dopo aver effettuato la registrazione. Tale struttura dati è un **ArrayList<Utenti>** ed è stata resa thread-safe con la classe *Collections*.
- **ConcurrentHashMap<String,ClientInterface> tabClient:** utilizzata per memorizzare le coppie *<nome utente, stub client>*.
- **ConcurrentHashMap<String,String> statusForClient:** utilizzata per memorizzare le coppie *<nome utente, stato online/offline>*.
- **ArrayList<Progetti> progetti:** struttura dati utilizzata per memorizzare i progetti. È l'unica struttura dati non resa *thread-safe*, poiché non viene utilizzata anche nei metodi remoti.

2.3. Persistenza Dello Stato Del Sistema

Lo stato del sistema deve essere **persistente**, poiché in caso di chiusura inattesa del server, deve essere possibile recuperare lo stato precedente alla chiusura del server senza perdere alcuna informazione. Perciò, le strutture dati implementate nel server sono state serializzate su *file system* con il supporto delle librerie **Jackson** versione 2.13.

2.3.1. Serializzazione

- **SerializzaCredenziali()**: questo metodo della classe **ServerClass** viene chiamato quando il client invoca i comandi *register*, *login* e *logout*. Ciò permette di serializzare oggetti di tipo Utenti.

```
1 [ {
2   "nickUtente" : "mauro",
3   "password" : "passWordSicura@12",
4   "stato" : "offline"
5 }, {
6   "nickUtente" : "lucia",
7   "password" : "Ciao32",
8   "stato" : "offline"
9 }, {
10  "nickUtente" : "omar",
11  "password" : "Ape44",
12  "stato" : "offline"
13 }, {
14  "nickUtente" : "robi",
15  "password" : "fiore43c",
16  "stato" : "offline"
17 } ]
```

Credenziali.json

- **Addcard()** e **moveCard()**: in questi metodi della classe **Progetti** vengono rispettivamente create e spostate le card nelle liste e infine vengono serializzate. Le card serializzate sono oggetti di tipo **Card**. Nel caso di una nuova card, viene creata una directory con il nome del progetto e all'interno della directory viene creato un file con il nome della card serializzata. Inoltre, la serializzazione avviene anche ogni volta che la card viene spostata in una nuova lista. Di seguito, vengono riportati due esempi di card serializzate, nel primo caso la card si trova nella lista TODO, quindi la card è stata solo creata. Nel secondo caso si trova nella lista DONE, ciò vuol dire che la card è stata completata.

```
{
  "descrizione" : "sostituire le gomme",
  "list" : "TODO",
  "history" : [ "TODO" ],
  "card" : "gomme"
}
```

Gomme.json

```
1 {
2   "descrizione" : "pulire la camera da pranzo",
3   "list" : "DONE",
4   "history" : [ "TODO", "INPROGRESS", "TOBEREVISED", "INPROGRESS", "DONE" ],
5   "card" : "stanza"
6 }
7
```

Stanza.json

- **SerializzaProgetti()**: questo metodo della classe **ServerClass** viene chiamato quando il client invoca i comandi *createProject*, *addMember*, *addCard* e *cancelProject*. Ciò permette di serializzare oggetti di tipo Progetti.

```
[ {
  "nomeProgetto" : "casa",
  "membri" : [ "mauro", "omar", "lucia", "robi" ],
  "multicastIp" : "224.0.0.0",
  "card" : [ "stanza", "cucina" ]
}, {
  "nomeProgetto" : "auto",
  "membri" : [ "mauro" ],
  "multicastIp" : "224.0.0.1",
  "card" : [ "gomme", "patente" ]
} ]
```

Progetti.json

- **SerializzaMulticastIp()**: questo metodo della classe **ServerClass** viene chiamato quando il client invoca i comandi *createProject* e *cancelProject*. Ciò permette di serializzare gli oggetti di tipo *GestoreMulticast*, i quali conterranno informazioni relativi agli indirizzi IP Multicast.

2.3.2. Deserializzazione

Per ripristinare correttamente l'ultimo stato, il Server utilizza il metodo *ripristinaStato*, che viene chiamato appena viene avviata l'applicazione tramite il metodo *ConnectTCP*.

Nel metodo *ripristinaStato*, si controlla che nel *file system* siano presenti i file **“credenziali.json”**, **“progetti.json”** e **“indirizziMulticast.json”** e si procede alla deserializzazione.

Si evidenzia che, dopo aver recuperato l'*ArrayList* dei progetti, viene invocato il metodo *ripristinaStatoCard* della classe *Progetti* affinché vengano deserializzate tutte le card relative a ogni progetto.

2.4. Gestoreipmulticast

La classe **GestoreIpMulticast** permette di generare nuovi indirizzi IP multicast. Nella classe sono presenti la variabile di istanza *ip*, che viene incrementata ogni volta che si genera un nuovo indirizzo IP multicast, e l'*ArrayList<String> ipDelete*, il quale memorizza gli indirizzi IP multicast che sono stati generati, ma non vengono più utilizzati, dato che il progetto è stato rimosso e di conseguenza la chat è inutilizzata. Lo scopo di *ipDelete* è quello di memorizzare tali indirizzi IP inutilizzati e recuperarli, tramite il metodo *retrieveMulticastIp*, nel momento in cui verrà creato un nuovo progetto.

Di seguito vengono presentati due possibili stati dell'oggetto GestoreIPmulticast serializzato nel file *indirizziMulticast.json*:

```
1 {  
2   "ip" : 2,  
3   "ipDelete" : [ ]  
4 }  
5
```

Sono stati generati due indirizzi IP multicast e non è stato rimosso alcun progetto

```
1 {  
2   "ip" : 5,  
3   "ipDelete" : [ "224.0.0.2", "224.0.0.4" ]  
4 }  
5
```

Sono stati generati cinque indirizzi IP multicast, ma due indirizzi IP sono stati rimossi e sono pronti per essere riutilizzati

2.4.1. GeneraMulticastIp

Questo metodo permette di generare nuovi indirizzi IP multicast dopo aver controllato in *ipDelete* se è possibile utilizzare un indirizzo IP inutilizzato.

Tutto il metodo ruota intorno alla variabile *ip*, che viene incrementata ogni volta che viene generato un nuovo indirizzo. La variabile *ip* deve essere necessariamente serializzata affinché il server possa sapere quale sia stato l'ultimo indirizzo generato. Gli indirizzi IP multicast che possono essere generati vanno da 224.0.0.0 fino a 239.255.255.255.

3. CLIENT CLASS

3.1. Strutture Dati Utilizzate E Accessi Concorrenti

Le strutture dati utilizzate dalla classe client sono state rese **thread-safe**, poiché vi possono accedere sia il thread della classe main del Client che il thread che viene attivato da uno dei metodo remoto. Di seguito, vengono elencate le strutture dati utilizzate nella classe **ClientClass**:

- **ConcurrentHashMap<String, String> statusClients** : in questa struttura dati vengono memorizzate le coppie *<nome utente, stato online/offline>*. Ogni utente ha questa struttura dati salvata in locale, che viene aggiornata² ogni volta che il server invoca il metodo remoto *notifyEvent*.
- **ConcurrentHashMap<String, String> tabIp** : in questa struttura dati vengono memorizzate le coppie *<nome progetto, indirizzo IP multicast>*.
- **ConcurrentHashMap<String, ArrayList<String>> chat** : in questa struttura dati vengono memorizzate le coppie *<nome progetto, oggetto di tipo ArrayList<String>>*. Infatti, ad ogni progetto viene associato un *ArrayList<String>*, nel quale verranno memorizzati i messaggi ricevuti dagli altri membri del progetto. Si noti che la struttura dati in cui vengono memorizzati i messaggi, è stata incapsulata in un blocco *synchronized*, in modo tale da gestire la concorrenza tra il thread del Server e il thread attivato dal metodo RMI. Infatti, il thread del Server potrebbe richiedere l'accesso alla struttura contenente i messaggi con il comando *readChat* (anche in questo caso è stato reso *synchronized*), mentre il thread, che monitora la chat, potrebbe accedere alla struttura per scrivere un nuovo messaggio ricevuto, senza che si creino situazioni di inconsistenza.
- **ConcurrentHashMap<String, ThreadMsg> tabThreads**: in questa struttura dati vengono memorizzate tutte le coppie *<nome progetto, oggetto di tipo ThreadMsg>*. Infatti, per ogni progetto viene associato un thread, che ha il compito di ricevere e scrivere i messaggi in arrivo nell' *arrayList* presente in *chat*.

Infine, durante la fase di **logout** vengono ripulite tutte le strutture dati utilizzate dal client, poiché, finché la connessione col server non viene chiusa, potrebbe collegarsi un nuovo utente dallo stesso client. Inoltre, tramite il metodo **closeAllChatThreads**, vengono terminati tutti i threads che si trovano in ascolto sulle chat dei progetti di cui l'utente è membro. Anche il metodo *logout* è stato reso *synchronized* per evitare situazioni di inconsistenza simili a quelle descritte in precedenza.

² Si noti che il riferimento *statusClients* punterà ad un nuovo oggetto *statusForClients* ogni volta che verrà invocato il metodo *notifyEvent*, quindi non c'è rischio di accessi concorrenti.

3.2. Chat

Tutti i membri di uno stesso progetto possono inviare e ricevere messaggi tramite connessione UDP.

- Quando un membro vuole inviare un messaggio sulla chat, utilizza il comando **sendchatmsg**: recupera l'indirizzo IP multicast dalla struttura dati *tabIp*, crea un pacchetto di tipo *DatagramSocket* e lo spedisce all'indirizzo IP multicast recuperato precedentemente e alla porta 5000.
- Quando un membro vuole visualizzare un messaggio sulla chat, utilizza il metodo **readchat**, con il quale il thread del Client recupera il messaggio dalla struttura dati *chat*. Si noti che i membri possono inviare e ricevere messaggi in modo asincrono.

Si evidenzia che anche il Server può mandare messaggi sulla chat del progetto per notificare ai membri l'avvenuto spostamento di una card da una lista all'altra tramite il metodo **notifyMoveCard**.

4. CLASSI PROGETTI, CARD E UTENTI

4.1. Card

La classe **Card** definisce i metodi per reperire informazioni relative ad una card. Inoltre, quando viene creata una nuova card tramite il metodo *addCard* della classe *Progetti*, viene invocato il metodo costruttore della classe *Card*, il quale, oltre a modellare il nuovo oggetto con il nome della card e la sua descrizione, etichetta la card come “TODO” sia nella variabile *list* sia nella struttura dati *history* di tipo *ArrayList<String>*. La variabile *list* memorizza l’ultima lista in cui si trova la card, mentre *history* memorizza tutte le liste in cui la card è stata spostata.

4.2. Progetti

La classe **Progetti** contiene le liste in cui le card potranno essere spostate: *todo*, *inprogress*, *toberevised* e *done*. Inoltre, contiene le strutture dati *membri*, che memorizza i membri di un progetto, e *nomeCards*, che memorizza i nomi delle cards del progetto. Infine, fornisce metodi per gestire agevolmente operazioni sulle cards.

4.3. Utenti

La classe **Utenti** viene usata per reperire informazioni relative alle credenziali di un utente iscritto al servizio e impostare il suo stato offline/online.

5. INFO GENERALI

5.1. Test Del Progetto

L'ambiente di sviluppo utilizzato per il progetto è stato **Eclipse** con la versione 16 di Java. Le librerie utilizzate sono *jackson-databind*, *jackson-annotations*, *jackson-core* nella versione 2.13.0.

Il progetto è stato testato sia da Eclipse che da terminale su sistema operativo macOS con processore ARM, Windows 10 e Linux (Ubuntu 20.04.3 LTS).

5.2. Come Avviare Il Progetto Da Terminale?

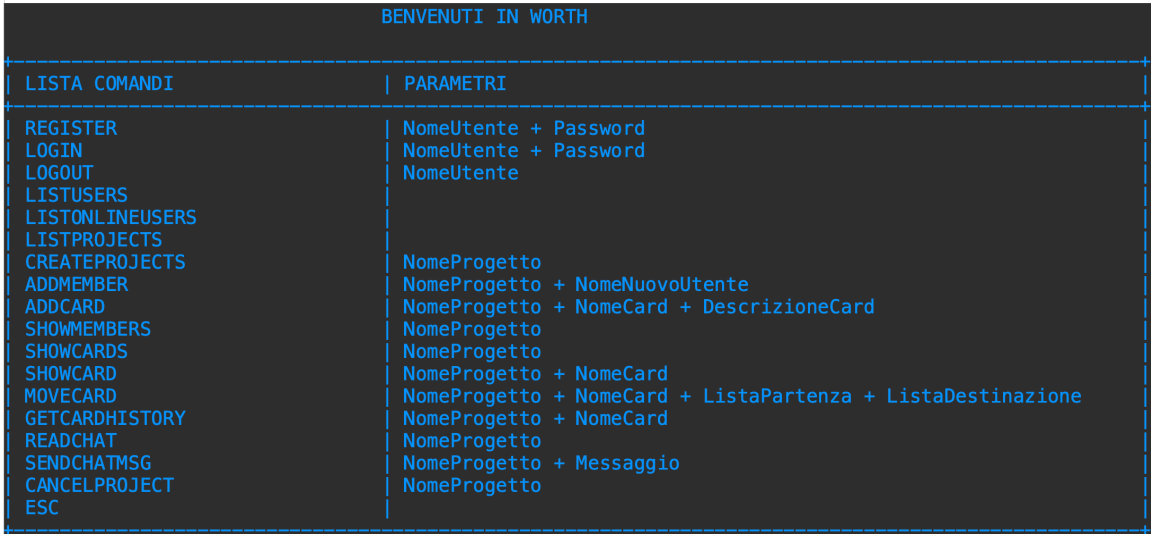
Per eseguire il progetto da terminale, bisogna spostarsi nella cartella **src** e lanciare i seguenti comandi in questo ordine:

1. **./server.sh** : in questo modo verrà compilato ed eseguito il server
2. **./client.sh** : in questo modo verrà compilato ed eseguito il client
3. **./clean.sh**: nel caso in cui si vogliano cancellare i file.class

Si evidenzia che a volte per eseguire gli script c'è bisogno dei permessi, quindi bisogna utilizzare il comando **chmod +x ./script.sh**

5.3. Comando Help

Il comando **help** è utile all'utente per visualizzare tutte le operazioni disponibili e i relativi parametri che deve utilizzare affinché il comando sia eseguito correttamente.



BENVENUTI IN WORTH	
LISTA COMANDI	PARAMETRI
REGISTER	NomeUtente + Password
LOGIN	NomeUtente + Password
LOGOUT	NomeUtente
LISTUSERS	
LISTONLINEUSERS	
LISTPROJECTS	
CREATEPROJECTS	NomeProgetto
ADDMEMBER	NomeProgetto + NomeNuovoUtente
ADDCARD	NomeProgetto + NomeCard + DescrizioneCard
SHOWMEMBERS	NomeProgetto
SHOWCARDS	NomeProgetto
SHOWCARD	NomeProgetto + NomeCard
MOVECARD	NomeProgetto + NomeCard + ListaPartenza + ListaDestinazione
GETCARDHISTORY	NomeProgetto + NomeCard
READCHAT	NomeProgetto
SENDCHATMSG	NomeProgetto + Messaggio
CANCELPROJECT	NomeProgetto
ESC	

Comando help()