Relazione del progetto

Paolo Junior Mollica

Matricola 564222

1 Introduzione

Il progetto contiene due cartelle, Client e Server. All'interno della cartella Client si trova il file `ClientMain.java`, cioè la classe contenente il metodo `main()`, e gli altri file del client. Analogamente, all'interno della cartella Server si trova il file `ServerMain.java` e gli altri file del server.

2 Le enum: Commands e Replies

All'interno dei file `Commands.java` e `StatusCodes.java`, presenti sia in Client che in Server, si trovano le omonime enum. La enum `Commands` contiene le codifiche dei comandi che richiedono di essere soddisfatti mediante una connessione TCP (ad esempio i comandi login, logout, list_projects, ecc). La enum `Replies` contiene invece le codifiche dei possibili esiti delle operazioni richieste, ovvero se l'operazione è riuscita oppure quale errore si è verificato.

Come spiegato successivamente, client e server comunicano scambiandosi messaggi, per cui il client includerà nel messaggio di richiesta un valore di `Commands` e il server includerà nel messaggio di risposta un valore di `Replies`. In questo modo il server, leggendo il valore di `Commands` nel messaggio del client, viene a conoscenza della richiesta che gli è stata fatta. Il client, invece, leggendo nel messaggio di risposta il valore di `Replies`, viene a conoscenza dell'esito dell'operazione e può agire di conseguenza.

3 TCP: la classe ClientServerMessage

La comunicazione tra client e server è implementata per mezzo della classe `ClientServerMessage`, ovvero client e server per comunicare si mandano esclusivamente oggetti di questa classe. Un oggetto di tipo `ClientServerMessage` ha quindi al suo interno molti campi, che verranno inizializzati dal client o dal server all'occorrenza. Il file `ClientServerMessage.java` è quindi presente in entrambe le cartelle.

Quando un utente digita un comando che richiede di utilizzare la connessione TCP, il client crea un oggetto di tipo `ClientServerMessage`, setta il

campo apposito dell'oggetto con il comando arrivato dall'utente (codificato con un valore della enum `Commands`), setta altri campi con gli argomenti che l'utente ha digitato insieme al comando e invia il messaggio.

Il server, alla ricezione del messaggio, crea un task (definito dalla classe `WorkerTask`) incaricato di soddisfare la richiesta del client e lo passa ad un thread pool. Quando viene eseguito, il thread legge il messaggio arrivato dal client, soddisfa la richiesta e invia a sua volta un oggetto di tipo `ClientServerMessage` al client. All'interno di questo messaggio, il thread setta il campo apposito dell'oggetto con l'esito dell'operazione (un valore della enum `Replies`) per informare il client sul fatto che la richiesta sia stata soddisfatta o meno, e altri campi nel caso debba restituire degli oggetti al client (ad esempio, per soddisfare una richiesta di login il server include nel messaggio l'oggetto di tipo `User` corrispondente all'utente).

Infine, una volta ricevuto il messaggio, il client legge il valore della enum `Replies` per sapere se l'operazione ha avuto successo oppure no, e che tipo di errore si è verificato in caso di esito negativo. Successivamente, se l'operazione è terminata con successo, il client, a seconda di che tipo di richiesta aveva fatto al server, legge l'oggetto che il server ha restituito nel rispettivo campo.

Ultima nota da fare è che i messaggi vengono inviati mediante due metodi: `sendToServer()` nel client e `sendToClient()` nel server. Entrambi i metodi funzionano allo stesso modo:

1. Serializzano il messaggio, ottenendo una stringa che viene a sua volta codificata in un array di bytes;
2. Allocano un oggetto `ByteBuffer sizeBuffer` con la dimensione esatta per contenere un intero, e lo inizializzano con la lunghezza dell'array di bytes creato in precedenza;
3. Allocano un oggetto `ByteBuffer dataBuffer` contenente i bytes del messaggio serializzato;
4. Inviano un array di `ByteBuffer` contenente i due byte buffers creati in precedenza.

In questo modo, quando si deve leggere un messaggio si legge prima la sua dimensione in bytes, poi si alloca un byte buffer della grandezza giusta e infine si leggono i bytes del messaggio, da cui poi viene ricostruito l'oggetto.

4 Gli altri file condivisi

Oltre a `Commands.java`, `Replies.java` e `ClientServerMessage.java`, ci sono altri file presenti in entrambe le cartelle. Questi file sono quelli che implementano le componenti essenziali di Worth (utenti, progetti e carte) e le interfacce degli oggetti remoti.

4.1 User.java

La classe `User` implementa un utente in Worth. Un utente è identificato univocamente dal nickname con cui si registra, per cui il metodo `equals()` è ridefinito per restituire `true` nel caso in cui i due utenti abbiano lo stesso nickname. La classe è composta dalle seguenti variabili d'istanza:

- `String nickname`: l'identificativo dell'utente;
- `String password`: la password dell'utente;
- `boolean online`: vale `true` se e solo se l'utente è online;
- `ArrayList<User> users`: la lista contenente tutti gli utenti registrati a Worth, aggiornata mediante il servizio di RMI callback;
- `ArrayList<Chat> chats`: la lista contenente tutte le chat dei progetti di cui l'utente è membro, anch'essa aggiornata mediante il servizio di RMI callback.

Per quanto riguarda i metodi, oltre ad alcuni metodi getter e setter base, contiene tre metodi molto importanti: `setChatsList()`, `readChat()` e `sendChatMsg()`.

Il metodo `setChatsList()` richiede attenzione perché è diverso nei file `User.java` delle due cartelle. Infatti, nel file `User.java` della cartella Server è un semplice metodo setter che aggiorna la lista `chats` con quella passata come argomento, mentre nel file `User.java` della cartella Client il metodo è più complicato. Questo perché mentre il server utilizza il metodo unicamente per azzerare la lista `chats` nell'oggetto di tipo `User` (per non trascriverla quando crea il file `.json` per gli utenti registrati, al momento di salvare lo stato del sistema), il client invece utilizza il metodo per aggiornare la lista `chats` dell'utente con quella arrivata mediante RMI callback. Quando arriva un aggiornamento, per ogni chat presente nella lista dell'aggiornamento, ci sono due casi:

1. La chat si riferisce a un progetto a cui l'utente è stato appena aggiunto, per cui dovrà essere inserita nella nuova lista `chats` dell'utente;
2. La chat si riferisce a un progetto per cui l'utente era già membro, per cui la chat corrispondente nella lista `chats` dell'utente deve persistere ma senza essere sovrascritta. Se infatti semplicemente si assegnasse la lista dell'aggiornamento alla lista `chats` dell'utente (come avviene nel server) si sovrascriverebbero le chat che già esistevano, provocando così una perdita dei messaggi non ancora letti.

Il metodo `setChatsList()` deve quindi fare in modo di aggiornare la lista `chats` dell'utente, ma senza perdere i messaggi non ancora letti delle chat

a cui già partecipava e che si trovano nell'aggiornamento. Le chat che erano nella lista `chats` dell'utente e che non si trovano nell'aggiornamento si riferiscono a progetti appena cancellati, per cui non si troveranno nella lista aggiornata.

Il metodo `readChat()` serve a leggere la chat del progetto passato come argomento, mentre il metodo `sendChatMsg()` serve a inviare un messaggio su una chat utilizzando UDP multicast (invia un oggetto di tipo `DatagramPacket` all'indirizzo multicast e alla porta assegnati dal server al progetto). Entrambe vengono utilizzate nel metodo `main()` del client.

4.2 Project.java

La classe `Project` implementa un progetto in Worth. Un progetto è identificato univocamente dal suo nome, per cui il metodo `equals()` è ridefinito per restituire `true` nel caso in cui il nome dei due progetti sia lo stesso. La classe è composta dalle seguenti variabili d'istanza:

- `String name`: l'identificativo del progetto;
- `ArrayList<Card> toDo`: la lista TODO descritta nella consegna;
- `ArrayList<Card> inProgress`: la lista INPROGRESS descritta nella consegna;
- `ArrayList<Card> toBeRevised`: la lista TOBEREVISED descritta nella consegna;
- `ArrayList<Card> done`: la lista DONE descritta nella consegna;
- `ArrayList<Card> cards`: una lista contenente tutte le carte che si trovano nelle 4 liste sopra elencate;
- `ArrayList<String> members`: una lista dei nickname dei membri del progetto;
- `InetAddress chatAddress`: l'indirizzo IP multicast per la chat;
- `String multicastAddress`: la rappresentazione testuale dell'indirizzo IP multicast per la chat, assegnata dal server;
- `int chatPort`: la porta per la chat, assegnata dal server.

La classe contiene inoltre vari metodi getter e setter per le liste e gli indirizzi.

4.3 Card.java

La classe `Card` implementa una carta in Worth. Una carta è identificata univocamente dal suo nome, per cui il metodo `equals()` è ridefinito per restituire `true` nel caso in cui il nome delle due carte sia lo stesso. La classe è composta dalle seguenti variabili d'istanza:

1. `String name`: l'identificativo della carta;
2. `String description`: la descrizione della carta;
3. `String history`: la storia della carta.

La classe contiene inoltre vari metodi getter e setter, e un metodo per aggiornare la variabile `history` quando la carta viene spostata da una lista del progetto a un'altra.

4.4 NotifyEventInterface.java

`NotifyEventInterface` è l'interfaccia dell'oggetto esportato dal client. I metodi dichiarati al suo interno sono `notifyUsersEvent()` e `notifyChatsEvent()`, entrambi remoti e che verranno chiamati dal server. Il primo metodo viene utilizzato dall'oggetto remoto per aggiornare la lista `users` all'utente, mentre il secondo viene utilizzato dall'oggetto remoto per aggiornare la lista `chats` (per mezzo del metodo `setChatsList()` della classe `User` descritto nella sezione 4.1). Il server chiama i due metodi passandogli come argomento, rispettivamente, la lista degli utenti registrati e la lista di tutti i progetti creati in Worth (come detto più avanti, sono le liste `registeredUsers` e `createdProjects`).

4.5 WorthInterface.java

`WorthInterface` è l'interfaccia dell'oggetto esportato dal server. I metodi dichiarati al suo interno sono `register()`, `registerForCallbacks()` e `unregisterForCallbacks()`, tutti e tre remoti e che verranno chiamati dai client. Il primo metodo serve a registrare un utente a Worth, il secondo a registrare un utente al servizio di callback e il terzo per cancellare la registrazione.

5 Threads e concorrenza

All'interno del progetto ci sono due file, uno per ogni directory, che descrivono ciascuno una classe che implementa l'interfaccia `Runnable`. Nella directory Client si trova il file `ChatSniffer.java`, mentre nella directory Server si trova il file `WorkerTask.java`, già accennato prima.

Come detto, quando il server (in particolare `ServerMain`) riceve un messaggio da un client, esso crea un oggetto di tipo `WorkerTask` e lo passa a un thread pool. Un oggetto di tipo `WorkerTask` viene creato passando al costruttore due cose: un buffer contenente la codifica in bytes del messaggio serializzato e il canale per la comunicazione con il client. Il costruttore si occupa di ripristinare l'oggetto di tipo `ClientServerMessage` mediante un processo di deserializzazione.

Quando il thread viene eseguito, controlla innanzitutto il campo del messaggio che contiene un `Commands`. Per ogni possibile valore di questo campo, chiama un diverso metodo della classe `ServerMain`, passandogli come argomenti i contenuti di altri campi del messaggio. Questi metodi sono tutti accumunati da due caratteristiche:

1. Restituiscono un oggetto di tipo `ClientServerMessage`, che consiste nella risposta che il thread deve mandare al client;

2. Sono thread safe.

Per quanto riguarda il secondo punto, questi metodi possono accedere (in lettura e/o in scrittura) a due variabili d'istanza della classe `ServerMain`, ovvero le due liste `registeredUsers` e `createdProjects`. La prima contiene gli utenti registrati a Worth (oggetti di tipo `User`), mentre la seconda contiene tutti i progetti esistenti (oggetti di tipo `Project`). Per gestire la concorrenza, quindi, la classe `ServerMain` dispone di altre due variabili d'istanza, entrambe di tipo `ReentrantReadWriteLock`, ovvero `usersLock` e `projectsLock`. In questo modo, quando un metodo richiede degli accessi in sola lettura a una delle due liste, esso viene sincronizzato con la relativa read lock. Se invece un metodo richiede di modificare una delle due liste, esso viene sincronizzato con la relativa write lock.

Il thread worker, una volta ottenuto il messaggio di risposta per il client, lo invia mediante il metodo `sendToClient()` (descritto nella sezione 3) sul canale che era stato passato al costruttore. A questo punto il thread termina.

Per quanto riguarda `ChatSniffer`, un thread creato con un task di questa classe ha il compito di ricevere i messaggi su una determinata chat. La classe `ClientMain` contiene, tra le altre, una variabile d'istanza `sniffers`, che consiste in una lista di threads sniffer, uno per ogni progetto dell'utente loggato sul client. Ognuno di questi threads, alla sua creazione, si unisce al gruppo multicast della chat e si mette in attesa di ricevere un messaggio. Ogni volta che ne riceve uno, lo aggiunge alla chat dell'utente e si rimette in attesa. Il thread termina quando l'utente effettua il logout oppure quando il progetto viene cancellato. Un thread sniffer viene creato ogniqualvolta l'utente diventa membro di un nuovo progetto.

La classe `ClientMain` contiene anch'essa una variabile d'istanza di tipo `ReentrantReadWriteLock`, ovvero `userLock`, per sincronizzare gli accessi all'oggetto `User user` che rappresenta l'utente loggato. Dato che un thread

sniffer deve accedere e modificare una chat dell'oggetto `user`, l'operazione deve essere sincronizzata mediante la write lock di `userLock`.

Infine, la variabile `userLock`, oltre che nei metodi di `ClientMain`, viene utilizzata anche dall'oggetto remoto del client (definito dalla classe `NotifyEventImpl`) nel momento in cui deve modificare una delle liste dell'oggetto `user` per una callback (sono due le liste aggiornate mediante RMI callback, ovvero `users` e `chats`).

6 Il client

Oltre ai file descritti in precedenza, nella cartella Client ci sono altri due file, ovvero `NotifyEventImpl.java` e `ClientMain.java`. Tutto il client gira intorno a una variabile d'istanza di `ClientMain`, la variabile `User user`, che come detto rappresenta l'utente attualmente connesso a Worth sul client.

6.1 NotifyEventImpl.java

La classe `NotifyEventImpl` è l'implementazione dell'interfaccia `NotifyEventInterface`, descritta nella sezione 4.4. Al suo interno sono implementati i due metodi remoti, `notifyUsersEvent()` e `notifyChatsEvent()`. Il metodo `notifyUsersEvent()` semplicemente setta la lista `users` di `user` con quella passata come argomento, ovvero la lista del server `registeredUsers` contenente gli utenti registrati. Il metodo `notifyChatsEvent()`, invece, riceve come argomento la lista del server `createdProjects`, contenente tutti i progetti esistenti in Worth. Per prima cosa il metodo deve creare una lista di chat (vuote) relative ai soli progetti di cui `user` è membro, per poi passarla come argomento al metodo `setChatsList()` di `user` (sezione 4.1).

Come detto prima, i due metodi `notifyUsersEvent()` e `notifyChatsEvent()` vanno a modificare asincronamente l'oggetto `user`. Per questo, viene fatto uso di un'altra variabile d'istanza di `ClientMain`, ovvero `userLock`. Le modifiche all'oggetto `user` sono quindi racchiuse dalla write lock di `userLock`.

6.2 ClientMain.java

La classe `ClientMain`, come detto, è la classe che contiene il metodo `main()` del client. Essa contiene le seguenti variabili d'istanza:

- `User user`: l'oggetto che rappresenta l'utente attualmente collegato in Worth;
- `ReentrantReadWriteLock userLock`: l'oggetto per la sincronizzazione degli accessi a `user`;
- `int registryPort`: la porta per la connessione al registry del server;
- `int TCPport`: la porta per la connessione TCP con il server;

- `ArrayList<Thread> sniffer`: una lista di thread sniffer, ciascuno creato passandogli un task di tipo `ChatSniffer`;
- `SocketChannel socketChannel`: il canale per la comunicazione TCP con il server;
- `NotifyEventImpl callbackObj`: l'oggetto esportato dal client per ricevere le callbacks;
- `NotifyEventInterface stub`: lo stub dell'oggetto remoto del client;
- `WorthInterface serverStub`: il riferimento all'oggetto remoto del server.

Per quanto riguarda i metodi, oltre al metodo `main()`, essa contiene un metodo per ogni possibile comando dell'utente e alcuni metodi ausiliari.

Quando l'utente digita un comando, viene eseguito il metodo di `ClientMain` associato. I metodi che sfruttano la connessione TCP seguono tutti uno scheletro generale: creano un oggetto di tipo `ClientServerMessage`, lo inviano al server mediante il metodo `sendToServer()` (descritto nella sezione 3), leggono la risposta mediante il metodo `receiveFromServer()` e infine stampano cose diverse a seconda di che metodo si tratta.

I metodi che invece non sfruttano la connessione TCP sono `help()`, `sendChatMsg()` e `readChat()`. Il primo stampa i comandi disponibili all'utente in un determinato momento, per cui non necessita di alcuna connessione. Il metodo `readChat()` fa semplicemente una stampa di una chat di `user`, per cui anch'esso non necessita di connessioni. Il comando `sendChatMsg()`, invece, fa uso di una connessione UDP per mandare messaggi a un gruppo multicast relativo a un progetto (ricevuti dai threads sniffer dei membri del progetto e aggiunti alle rispettive chat).

Infine, il metodo `main()` è composto sostanzialmente da due cicli `while`: il primo riceve dei comandi finché non viene completato il login dell'utente, il secondo finché non viene completato il logout. Una volta effettuato il logout, il programma termina.

7 Il server

Anche nel server, oltre ai file già citati, sono presenti altri due file, ovvero `WorthImpl.java` e `ServerMain.java`. Tutto il server gira intorno a due variabili d'istanza di `ServerMain`: la variabile `registeredUsers` e la variabile `createdProjects`. Come detto, la prima è la lista di tutti gli utenti registrati a Worth, la seconda la lista di tutti i progetti esistenti in Worth.

7.1 WorthImpl.java

La classe `WorthImpl` è l'implementazione dell'interfaccia `WorthInterface`, descritta nella sezione 4.5. Al suo interno sono implementati i tre metodi remoti dichiarati nell'interfaccia: `register()`, `registerForCallbacks()` e `unregisterForCallbacks()`.

Il metodo `register()` registra l'utente a Worth. La registrazione è vista come la presenza dell'oggetto di tipo `User` dell'utente nella lista `registeredUsers`. Pertanto, il metodo `register()` crea un oggetto di tipo `User` con `nickname` e `password` specificati dall'utente, controlla che non sia già nella lista `registeredUsers` e la aggiunge. Restituisce un valore di `Replies` per informare il `ClientMain` sull'esito dell'operazione.

Il metodo `registerForCallbacks()` aggiunge lo stub dell'oggetto remoto del client, passato come argomento, a un'altra lista di `ServerMain`, ovvero la lista `clientsRegisteredForCallback`. In questo modo, ogniqualvolta vadano mandate delle callback, è sufficiente scandire la lista per sapere quali client sono registrati. Il metodo `unregisterForCallbacks()` consiste pertanto nel rimuovere dalla lista lo stub del client di cui cancellare la registrazione.

Tutti e tre i metodi accedono a delle liste della classe `ServerMain`. Per farlo, utilizzano dei metodi forniti dalla classe `ServerMain` e opportunamente sincronizzati con le variabili d'istanza `usersLock` e `callbackLock`.

7.2 ServerMain.java

La classe `ServerMain`, come detto, è la classe che contiene il metodo `main()` del server. Essa contiene le seguenti variabili d'istanza:

- `ArrayList<User> registeredUsers`: la lista contenente tutti gli utenti registrati a Worth;
- `ReentrantReadWriteLock usersLock`: un oggetto di sincronizzazione per la lista `registeredUsers`;
- `ArrayList<Project> createdProjects`: la lista contenente tutti i progetti esistenti in Worth;
- `ReentrantReadWriteLock projectsLock`: un oggetto di sincronizzazione per la lista `createdProjects`;
- `ArrayList<NotifyEventInterface> clientsRegisteredForCallback`: una lista contenente i riferimenti agli oggetti remoti dei client registrati al servizio di callback;
- `ReentrantReadWriteLock callbackLock`: un oggetto di sincronizzazione per la lista `clientsRegisteredForCallback`;

- `ThreadPoolExecutor threadPool`: il thread pool contenente i thread worker (creati con un oggetto di tipo `WorkerTask`);
- `int registryPort`: la porta del registry;
- `int TCPport`: la porta per le connessioni TCP;
- `String multicastAddress`: indirizzo IP multicast di partenza, a partire dal quale assegna gli indirizzi per le chat ai progetti;
- `LinkedBlockingQueue<String> addressesToBeReallocated`: una coda contenente gli indirizzi multicast assegnati a dei progetti che sono stati cancellati, e quindi da riassegnare ai progetti che verranno creati in futuro;
- `int multicastPort`: porta per il multicast, da assegnare ai progetti per le chat;
- `String stateDirName`: nome della directory contenente lo stato del sistema;
- `String usersFilename`: nome del file contenente lo stato degli utenti registrati;
- `String projectMembersFilename`: nome del file contenente i nomi dei membri di un progetto. Ne viene creato uno per ogni directory relativa a un progetto.

All'interno della classe `ServerMain`, come detto, si trovano tutti i metodi utili ai thread worker per soddisfare le richieste dei client, e i metodi utili all'oggetto remoto. All'interno di questi metodi, gli accessi alle liste sono sincronizzati mediante `usersLock`, `projectsLock` e `callbackLock`, in modo da renderle thread safe.

Oltre ai metodi per i thread worker, all'interno della classe si trovano i metodi per il salvataggio e il ripristino dello stato. Grazie a questi metodi, ogni volta che il server termina salva il suo stato in una directory il cui nome è contenuto in `stateDirName`. Dentro questa directory viene creato un file `.json`, il cui nome è contenuto nella variabile `usersFileName`, contenente la lista `registeredUsers` serializzata, e vengono create tante directories quanti sono i progetti in `createdProjects`. Ognuna di queste directories ha lo stesso nome del progetto a cui si riferisce, e al suo interno ci sono un file `.json` per ogni carta e uno dove al suo interno vi è la lista dei membri del progetto. I file delle carte hanno lo stesso nome della carta a cui si riferiscono, e all'interno contengono l'oggetto di tipo `Card` serializzato, mentre il nome del file dei membri del progetto è contenuto nella variabile `projectMembersFilename` e il file contiene la lista `members` dell'oggetto di tipo `Project` serializzata.

Nella classe `ServerMain` si trova anche il metodo `BindChatAddress()` che serve ad assegnare la porta e l'indirizzo multicast a un nuovo progetto. Per farlo controlla prima la lista `addressesToBeReallocated`: se ci sono indirizzi all'interno assegna quelli, altrimenti assegna il successivo rispetto a quello contenuto nella variabile `multicastAddress`, che viene poi aggiornata con l'indirizzo assegnato. In questo modo la variabile `multicastAddress` contiene sempre l'indirizzo più grande che si stato assegnato ai progetti.

Infine, il metodo `main()` di `ServerMain` consiste in un ciclo infinito, nel quale vengono accettate le connessioni dei client e vengono ricevuti i messaggi da essi, mediante l'uso di un selettore. Nel momento in cui viene letto un messaggio di un client, viene creato un task di tipo `WorkerTask` a cui viene passato il messaggio (o meglio il buffer contenente la codifica in bytes del messaggio serializzato) e il canale per la comunicazione con il client, il quale viene inserito ed eseguito in `threadPool`.

8 Eseguire il progetto

Per eseguire il progetto, aprire due o più shell ed entrare con una nella directory Server, e con le altre nella directory Client. I comandi per eseguirlo variano in base al sistema operativo.

8.1 Eseguire in Windows

Nella shell del server compilare utilizzando il comando

```
javac -cp lib\gson-2.8.6.jar *.java
```

ed eseguire il server utilizzando il comando

```
java -cp lib\gson-2.8.6.jar;.  ServerMain
```

Successivamente, in una shell di un client compilare utilizzando il comando

```
javac -cp lib\gson-2.8.6.jar *.java
```

Adesso è possibile eseguire ogni client utilizzando, nella rispettiva shell, il seguente comando:

```
java -cp lib\gson-2.8.6.jar;.  ClientMain
```

Infine, nel caso si volessero cancellare i file `.class` da una directory utilizzare il seguente comando:

```
del *.class
```

8.2 Eseguire in Linux

Nella shell del server compilare utilizzando il comando

```
javac -cp lib/gson-2.8.6.jar *.java
```

ed eseguire il server utilizzando il comando

```
java -cp lib/gson-2.8.6.jar:.  ServerMain
```

Successivamente, in una shell di un client compilare utilizzando il comando

```
javac -cp lib/gson-2.8.6.jar *.java
```

Adesso è possibile eseguire ogni client utilizzando, nella rispettiva shell, il seguente comando:

```
java -cp lib/gson-2.8.6.jar:.  ClientMain
```

Infine, nel caso si volessero cancellare i file `.class` da una directory utilizzare il seguente comando:

```
rm -f *.class
```

9 I comandi di Worth

All'avvio del client verrà mostrato un piccolo menù in cui è possibile cominciare a scrivere i comandi. I comandi disponibili sono:

- `help`: stampa una lista dei comandi disponibili in quel momento, con una breve spiegazione della loro funzione;
- `register [nickname] [password]`: registra l'utente con il nickname e la password specificati;
- `login [nickname] [password]`: fa il login dell'utente con nickname e password specificati;
- `logout [nickname]`: fa il logout dell'utente;
- `list_users`: stampa la lista degli utenti registrati, ciascuno con il suo stato online/offline;
- `list_online_users`: stampa la lista degli utenti online;
- `list_projects`: stampa la lista dei progetti di cui l'utente fa parte;
- `create_project [project_name]`: crea un progetto di nome `project_name`;
- `add_member [project_name] [nickname]`: aggiunge un membro `nickname` al progetto di nome `project_name`;

- `show_members [project_name]`: stampa la lista dei membri del progetto di nome `project_name`;
- `show_cards [project_name]`: stampa la lista delle carte del progetto di nome `project_name`;
- `show_card [project_name] [card_name]`: mostra le informazioni della carta di nome `card_name` che si trova nel progetto di nome `project_name`;
- `add_card [project_name] [card_name] [description]`: aggiunge una carta di nome `card_name` e descrizione `description` al progetto di nome `project_name` (la descrizione non può contenere spazi);
- `move_card [project_name] [card_name] [source_list] [dest_list]`: all'interno del progetto di nome `project_name`, sposta la carta di nome `card_name` dalla lista `source_list` alla lista `dest_list`;
- `get_card_history [project_name] [card_name]`: stampa la storia della carta di nome `card_name` che si trova nel progetto di nome `project_name`;
- `send [project_name] [message]`: invia sulla chat del progetto di nome `project_name` il messaggio `message` (il messaggio può contenere spazi);
- `receive [project_name]`: stampa i messaggi non letti della chat del progetto `project_name`;
- `cancel_project [project_name]`: cancella il progetto `project_name` se tutte le carte si trovano nella lista DONE.

Una volta completata la richiesta di logout, il programma client termina. Per terminare il server, è sufficiente fare un CTRL-C sulla shell corrispondente ed esso provvederà a salvare lo stato del sistema.