Università di Pisa

Relazione Progetto WORTH

Corso di Laboratorio di Reti di Calcolatori

A.A 2020/21

Studente:
Luca Miglior - mat. 580671

Docente:
Prof.ssa Federica Paganelli

Indice

1 Introduzione

WORTH (WorkTogetHer) è un progetto didattico per la gestione di lavori di gruppo in modo collaborativo, da remoto. Ispirato dalla metodologia *Kanban*, che prevede l'utilizzo di schede mobili (solitamente post-it da affiggere su una lavagna) per indicare i task da eseguire per portare a termine un'attività, il software WORTH si propone di virtualizzare il processo di sviluppo Kanban e rendere accessibili da remoto lavori collaborativi, siano essi professionali, o amatoriali.

WORTH riproduce l'esperienza del lavoro "agile", simulando ogni componente essenziale del *Kanban*: attraverso l'utilizzo della piattaforma, gli utenti potranno registrarsi al servizio e, tramite un'interfaccia grafica o testuale a riga di comando, potranno creare nuovi progetti e collaborare agilmente con altri utenti iscritti al servizio. Sarà possibile creare nuove card per ogni lavagna virtuale, e ogni membro del team potrà riorganizzare le attività, spostando le card da una lista all'altra, o aggiungendo nuovi membri al gruppo di lavoro. WORTH predispone inoltre un sistema di chat per favorire la comunicazione e la condivisione di idee fra i membri di ogni progetto.

L'intero sistema è supportato da un server centrale remoto, che si occuperà di ricevere, gestire e orchestrare la collaborazione fra gli utenti della piattaforma.

2 Architettura generale del Sistema

L'architettura generale del sistema è basata sul paradigma Client-Server. Gli utenti del servizio possono accedere al sistema mediante un Client, che invia le richieste ad un server (in rete locale o Internet) che le elabora e restituisce le informazioni richieste dal client. Le connessioni instaurate fra client e server sono affidate al protocollo TCP/IP. Al momento del lancio di un client, questo instaura una connessione TCP con il server, presso un indirizzo IP e porta noti; inoltre, tramite il paradigma RMI, il client crea un riferimento agli oggetti remoti pubblicati dal server su un registry noto: questi saranno necessari per l'operazione di registrazione al servizio, e verranno invocati su richiesta dell'utente. In seguito alla registrazione, il client potrà quindi effettuare l'operazione di login, unica operazione che permetterà di inviare ulteriori richieste. In seguito al successo dell'operazione di login, il server è pronto a ricevere nuovi messaggi.

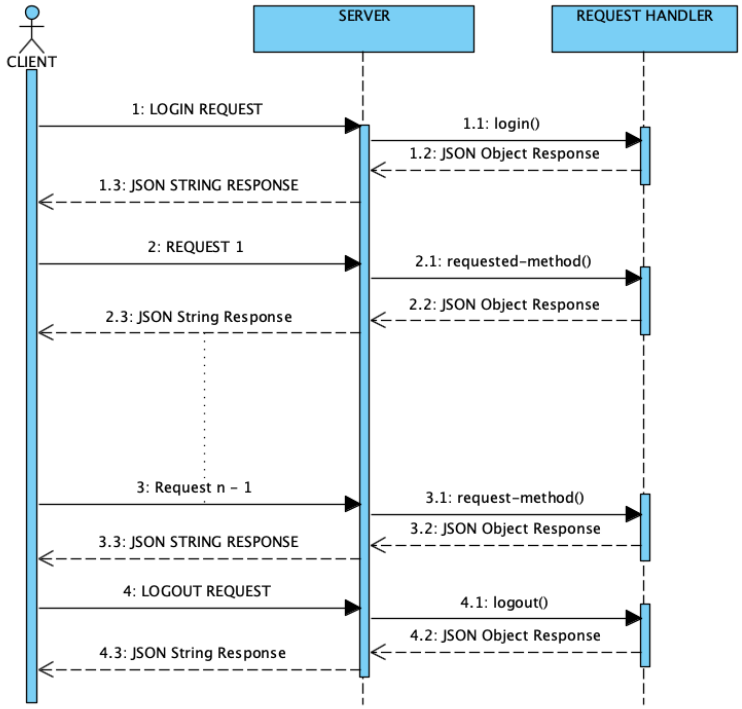

Figura 1: Diagramma di sequenza che mostra l'interazione fra client e server

I messaggi scambiati fra il client ed il server sono codificati come stringhe in formato JSON[1]. Ogni messaggio JSON inviato dal client dovrà obbligatoriamente contenere un campo `login-name` contenente una stringa indicante il nome utente associato al client che ha effettuato la richiesta, e un campo `method`, contenente una stringa che identifichi univocamente il metodo e il tipo della richiesta che il server dovrà elaborare. Ogni metodo, necessita a sua volta di altri parametri obbligatori, che saranno aggiunti alla stringa JSON della richiesta: ogni parametro necessario al metodo per essere eseguito è identificato da un nome noto, per permettere al server di effettuare il parsing della stringa JSON in oggetti locali, e invocare i metodi correttamente e senza ambiguità.

Una volta ricevuta una richiesta di un client, il server effettua la deserializzazione della stringa in un generico oggetto JSON[2], e procede all'invocazione del metodo richiesto; in seguito all'elaborazione della richiesta, viene generata una stringa codificata JSON che contiene il risultato dell'elaborazione. I metodi invocati dal server restituiscono un ulteriore oggetto JSON, che conterrà obbligatoriamente un campo numerico `return-code`, ad indicare in via preliminare il risultato dell'elaborazione della richiesta. La semantica dei codici di ritorno è la stessa utilizzata dal protocollo HTTP. In caso di successo, l'oggetto JSON conterrà altri campi con i dati prodotti dall'elaborazione della funzione, e il codice restituito avrà un valore del tipo `2xx`; in caso di fallimento, verrà restituito un codice del tipo `4xx` oppure `5xx`, utilizzati rispettivamente in caso di errore riscontrato nella sintassi o nel contenuto della richiesta, oppure in caso di errore interno del server durante l'handling della stessa. L'oggetto risposta JSON è quindi serializzato in una stringa, e inviato sulla connessione TCP al client corrispondente. Il client, rimasto in ascolto, elaborerà i dati ricevuti dal server, che saranno visualizzati in maniera appropriata all'utente.

La connessione TCP sulla quale viaggiano le richieste è mantenuta aperta, salvo eccezioni, finché non si riceve una richiesta di tipo `logout` dal client. In seguito alla richiesta di logout il server effettua il logout dell'utente e chiude la connessione; a questo punto anche il client terminerà la connessione invocando il metodo close sul proprio socket TCP. In caso di chiusure inattese della connessione, il server mantiene consistente lo stato del sistema, individuando la connessione che ha causato l'errore ed effettuando le opportune modifiche alle strutture dati locali che identificano il client corrispondente alla socket.

3 Implementazione del Server

Il server è il cuore dell'applicazione WORTH: si occupa di gestire le connessioni di rete verso i client e mantiene al suo interno le strutture dati necessarie al soddisfacimento delle richieste; si occupa di salvare in memoria secondaria i dati relativi allo stato della piattaforma, nonché del ripristino delle strutture in seguito a riavvio. Al server è inoltre delegato il compito della gestione e dell'assegnamento degli indirizzi multicast utilizzati dalle chat di progetto. È stato sviluppato in ambiente UNIX, utilizzando Java 8. È inoltre stata utilizzata la libreria GSON, per la serializzazione in stringhe degli oggetti Java e la deserializzazione, e la libreria Apache Commons CLI, per il parsing degli argomenti da riga di comando.

L'implementazione delle funzionalità di rete e la gestione delle connessioni è racchiusa interamente nella classe `Server.java`. Un'istanza della classe `Server` è creata al momento dell'avvio del programma. Il costruttore del server accetta come argomenti cinque parametri: indirizzo IP e porta per effettuare il bind della socket, nome e porta del Registry RMI e directory della cartella nella quale è mantenuto lo stato del sistema. Al momento dell'istanziazione, il costruttore inizializza le variabili locali della classe, crea un socket per la gestione delle connessioni e verifica se è presente su disco, nel path specificato, un backup di una precedente esecuzione da ripristinare. In caso di errore viene lanciata un'eccezione del tipo `IOException`. In caso di successo, la funzione main invoca il metodo `start()` sull'oggetto server.

```
public Server(String address, int channelport, String registryname, int rmiport,
    String workingdir)
```

Listing 1: Firma del metodo costruttore della classe `Server.java`

Se tutto è andato a buon fine, il metodo main procede con l'esecuzione e invoca il metodo `start()` sull'oggetto appena creato. A questo punto il nuovo oggetto avvia il server RMI e sarà pronto ad accettare nuove richieste all'indirizzo IP e porta specificati dall'utente.

[1] Per effettuare la serializzazione in stringhe e la deserializzazione in oggetti è stata utilizzata la nota libreria esterna GSON.

[2] Con oggetto JSON, si intente un'istanza della classe `JsonObject` della libreria GSON.

3.1 Funzionalità di rete e gestione delle connessioni

Nella tradizionale visione bloccante, proposta dalla libreria Java IO, il server sarebbe stato costituito da almeno un thread, con un oggetto della classe ServerSocket in ascolto, bloccato, per nuove connessioni, e da n thread (o da una threadpool), ognuno relegato alla gestione di una singola socket, bloccato in attesa che il client invii dati da elaborare sulla connessione; questo meccanismo non è adatto per sopportare alti carichi di lavoro: il processo di creazione dei thread è molto costoso, la creazione di un thread comporta notevole overhead da parte del sistema operativo, così come le commutazioni di contesto e i meccanismi di sincronizzazione necessari per l'esecuzione del codice concorrente. Inoltre, si deve considerare che la maggior parte dei thread potrebbe rimanere bloccata in ascolto di dati da parte del client per gran parte della durata del suo ciclo di vita. Il server sviluppato con Java IO, subirebbe un degrado delle performance sotto carico. Per questo progetto, è stato quindi deciso di implementare un server con comportamento non bloccante, utilizzando la libreria Java NIO.

L'implementazione del server attraverso NIO effettua il multiplexing delle richieste ed è basata su tre componenti fondamentali:

- un oggetto della classe `ServerSocketChannel` che rimane in ascolto delle connessioni in arrivo;
- un `SocketChannel` per ogni connessione attiva;
- un `Selector` per iterare sulle connessioni attive ed effettuare il multiplexing di queste.

Al momento della creazione del server viene istanziato un `Selector` e il `ServerSocketChannel`, tramite l'invocazione del metodo `open()`; per il corretto funzionamento in modalità non bloccante, l'oggetto `serverSocketChannel` è registrato sul selettore appena creato. In seguito all'invocazione del metodo `start()`, il server entra in un ciclo `while(true)`; ad ogni iterazione è verificata la presenza o meno di canali pronti per effettuare qualche operazione. Se il selettore non è vuoto (ovvero, restituisce un valore diverso da zero invocando su di esso il metodo `select()`), si crea un oggetto di tipo `Iterator<SelectionKey>` per iterare sul set di chiavi corrispondenti ai canali disponibili. Per ogni chiave restituita dall'iteratore, si individua il tipo di evento pronto da essere elaborato: se la chiave corrisponde ad un evento di tipo `OP_ACCEPT`, vi è una nuova connessione in arrivo e si invoca il metodo privato `accept()`, che si occupa di creare un nuovo SocketChannel per la connessione, di registrarlo sul selettore e di allocare un buffer[3] per la memorizzazione dei dati; se il canale invece risulta *readable*, si procede con l'handling della richiesta chiamando il metodo privato `readSocketChannel()`.

3.1.1 Il metodo readSocketChannel

Quando invocato, si occupa di leggere i dati presenti sul buffer associato alla chiave selezionata e di fornire una risposta appropriata al messaggio in esso contenuto. Dopo aver istanziato le variabili locali, il metodo si occupa di controllare che il canale sia effettivamente aperto e leggibile; in caso di canale non leggibile, in caso di errori, o quando un client chiude la connessione in maniera inaspettata, l'utente associato a quella particolare socket dev'essere eliminato dalla lista degli utenti online: per consentire questa operazione, il server mantiene nelle variabili locali di ogni utente un riferimento alla connessione in uso. [4]

Se il canale è leggibile, incomincia la trascrizione dei dati dal buffer in oggetti locali: il contenuto del buffer viene decodificato in stringa, e la stringa viene deserializzata in un oggetto JSON. Se l'operazione va a buon fine, ovvero, se la stringa rappresenta un oggetto JSON valido, la funzione recupera la proprietà `method` dall'oggetto e tenta di recuperare le altre proprietà richieste dal metodo per essere eseguito; a questo punto, è chiamata la funzione locale corretta che restituirà un nuovo oggetto JSON contenente i dati dell'elaborazione (eventualmente, anche un codice di errore in caso di elaborazione non andata a buon fine); al ritorno di un handler, readSocketChannel serializza l'oggetto risposta in stringa, trascrive la stringa sul buffer e scrive i dati sul SocketChannel. Infine il canale è nuovamente registrato in lettura sul selettore, per permettere al client di inviare nuovi dati.

[3] Per l'allocazione dei buffer sono stati utilizzati oggetti della classe `ByteBuffer`. La dimensione è fissata di default in 8KB.

[4] Il sistema consente che sia attiva al più un'istanza di un singolo utente durante l'esecuzione del programma.

3.2 Strutture dati principali della classe Server

La classe Server, durante l'esecuzione, mantiene i dati in memoria principale attraverso mappe chiave-valore del tipo `ConcurrentHashMap<>`. La scelta di oggetti di tipo Map, invece che di oggetti di tipo List per la memorizzazione dei dati, è stata ritenuta più efficiente dal momento che la maggior parte degli accessi alle risorse sono effettuati per nome: si pensi all'operazione di recupero delle informazioni di un utente, o alle informazioni relative alla lista dei membri di un singolo progetto. Una soluzione con strutture del tipo `ArrayList<>` avrebbe probabilmente avuto problemi di performance in riferimento a questo particolare caso d'uso, specialmente con un alto numero di utenti o progetti e considerata la natura single-threaded dell'applicazione. Per ottimizzare le performance in tempo della HashMap, sarà inoltre possibile modificare in futuro il fattore di carico della struttura per evitare il numero di collisioni interne. Dunque, il server possiede due strutture dati principali per la memorizzazione dei contenuti:

- Una `ConcurrentHashMap<String, User>` per la memorizzazione degli utenti a runtime;
- Una `ConcurrentHashMap<String, Project>` per la memorizzazione dei progetti.

Le strutture dati sono state scelte *thread-safe* per evitare problemi di sincronizzazione e consistenza dei dati in caso di accessi concorrenti da parte del thread main e dagli eventuali thread RMI: sebbene la mutua esclusione sull'esecuzione dei metodi RMI sia garantita dall'implementazione dei metodi della classe stessa (della quale si discuterà nel sottoparagrafo seguente), nulla avrebbe garantito l'accesso in mutua esclusione alle strutture dati condivise fra le classi Server e RMIServer. *Inoltre, se in un futuro si volesse espandere l'applicazione per far gestire le richieste da più thread workers, in maniera asincrona, non sarà necessario implementare ulteriori meccanismi di sincronizzazione sulle strutture dati di base del sistema, garantendo un minimo di funzionalità out-of-the-box.*

3.3 RMI per le operazioni di Registrazione e Callback

Come richiesto nelle specifiche dell'implementazione, le funzionalità di registrazione di nuovi utenti al servizio e callback per notifiche di login e logout verso i client, sono state implementate con il protocollo RMI. Il meccanismo delle callback RMI è stato inoltre utilizzato per comunicare al client gli estremi per unirsi alla chat di un progetto, nel momento in cui questi vengono aggiunti. Le funzionalità del server RMI sono implementate nella classe `RMIServer.java`, che a sua volta implementa l'interfaccia `RMIServerInterface.java`

La classe è istanziata all'interno del metodo `startRMI()` della classe Server, in seguito alla creazione di un nuovo oggetto, il metodo provvede anche ad effettuare il bind di un nuovo registry e al contestuale avvio del server RMI.

```
public interface RMIServerInterface extends Remote {
    int  signUp(String Username, String password)         throws RemoteException;
    void registerForCallback(RMIClientInterface client)   throws RemoteException;
    void unregisterForCallback(RMIClientInterface client) throws RemoteException;
}
```

Listing 2: Interfaccia `RMIServerInterface.java`

Il costruttore della classe `RMIServer` accetta due parametri: la HashMap degli utenti iscritti al servizio, e un'istanza della classe FileHandler, della quale si illustreranno le funzionalità in seguito, per permettere la memorizzazione dei nuovi profili utente al momento della registrazione. Tutti i metodi sono di tipo `synchronized void`: questa sincronizzazione è necessaria, dal momento che non è possibile sapere a priori quanti e quali thread RMI eseguiranno i metodi della classe. RMIServer possiede inoltre una struttura dati locale per mantenere in memoria i client registrati alle callback di Utenti e Chat di progetto. L'operazione di registrazione è implementata dal metodo `signUp(String username, String password)`: quando un client invoca questo metodo, viene creata un nuovo oggetto della classe User e aggiunto alla mappa degli utenti: l'utente appena registrato potrà ora effettuare il login sul server. Per quanto riguarda le Callback per le notifiche di eventi, il client si registra invocando il metodo `RegisterForCallback(RMIClientInterface)`, e fornisce una sua interfaccia pubblica contenente la firma dei metodi che il server potrà chiamare per notificare l'iscrizione alla chat di un nuovo progetto o il cambiamento di stato (online, offline) di un utente. In caso di eccezioni RMI dipendenti da disconnessioni inaspettate dei client, è individuata la connessione che ha sollevato l'eccezione, e rimosso il client associato dalla lista dei client registrati per le notifiche.

3.4 La classe User

Questa classe rappresenta un utente della piattaforma WORTH. Un'istanza di una classe User è creata dal server RMI al momento dell'iscrizione. Inoltre, gli oggetti di tipo User sono ripristinati dal server in caso di riavvio del sistema.
Ogni oggetto della classe consta di cinque variabili di istanza:

1. Tre oggetti della classe `String` rispettivamente per username, password e *salt* della password;
2. Una variabile di tipo `boolean`, per indicare lo stato (online, offline) dell'utente;
3. Una variabile di tipo `int` che contiene l'hash dell'oggetto SocketChannel [5], in modo da poter recuperare facilmente e univocamente la connessione associata all'utente.

Il costruttore della classe accetta esattamente tre argomenti, di cui al punto uno. Lo stato dell'utente è impostato di default con il valore `false` e il codice hash relativo alla socket associata è impostato dal server al momento del login. Tutte le variabili d'istanza di User sono accessibili mediante i metodi di default `get` (informalmente detti *"getters"*), e sono modificabili a runtime mediante i metodi `set` (*"setters"*)

3.4.1 Cenni sulla memorizzazione delle password

Per quanto WORTH sia un progetto sviluppato puramente a scopo didattico, nel quale alcuni dettagli *potrebbero* essere trascurati (e.g. piccoli aspetti riguardanti la sicurezza), è stato scelto, almeno limitatamente alla classe User, di adottare una strategia *puramente dimostrativa* che si proponga di oscurare le password per ogni utente, in modo da non rendere accessibili dai files di backup *"in chiaro"* le credenziali di accesso degli utenti. L'hashing delle password degli utenti è delegato alla classe `PasswordHandler`. Nel momento in cui un nuovo utente si registra, viene generato automaticamente un *salt* [6] e l'hash della password inserita, attraverso la creazione di un oggetto del tipo `SecretKeyFactory` [7], che esegue l'hashing utilizzando l'algoritmo SHA-1; al termine del processo, l'hash e il salt sono memorizzati all'interno dell'oggetto utente con codifica Base64, in modo da rendere tutti i caratteri stampabili, nonché sul filesystem. Il riconoscimento della password è effettuato in maniera analoga: per ogni tentativo di login è calcolato l'hash della password inserita e lo si confronta con l'hash presente sul server: se le due stringhe coincidono, la password inserita è corretta.

3.5 La classe Project

Analogamente alla classe User, `Project.java` rappresenta un progetto del sistema WORTH, tuttavia, a differenza di `User`, non si occupa solamente di memorizzare i dati relativi al progetto, ma effettua operazioni sui dati di progetto (come gestione persistenza sul disco, spostamento di card, e controlli sui vincoli imposti da specifica) su richiesta del client. La classe Project possiede due campi `String`, per la memorizzazione del nome di progetto e per l'indirizzo della chat, e possiede quattro strutture di tipo `HashMap<String, Card>` per la memorizzazione delle Card rappresentanti i task da svolgere per portare a termine l'attività. Come già discusso in precedenza (si veda 3.2), è stato scelto di utilizzare una mappa per la memorizzazione delle Card dal momento che l'accesso alle risorse è effettuato prevalentemente per nome. I membri di progetto sono memorizzati in una struttura `ArrayList<User>`

La classe Project possiede due costruttori: entrambi accettano due oggetti di tipo `String` per memorizzare il nome di progetto e l'indirizzo multicast relativo alla chat; i costruttori richiedono inoltre un'istanza della classe `FileHandler`, per mantenere sul filesystem le modifiche, quando avvengono. La differenza fra i due costruttori, consiste nel modo in cui essi inizializzano i membri di progetto: un costruttore, infatti, accetta un'oggetto di tipo `User`[8], ed è utilizzato al momento della creazione di un nuovo progetto, a runtime, quando questa richiesta proviene da un client; l'altro prevede il passaggio di un riferimento ad un `ArrayList<User>`, e si utilizza al momento del rirpristino, quando è già nota, dai files di backup, la lista dei membri del progetto. Oltre ai metodi standard *getters* e *setters* per il recupero dei riferimenti agli oggetti locali e la loro modifica, sono

[5] Il codice hash della socket è ottenuto chiamando il metodo `hashCode()` sull'oggetto SocketChannel associato.
[6] Il *salt* è una sequenza di numeri casuali posti prima o dopo la password, per evitare attacchi di tipo dizionario
[7] La classe SecretKeyFactory fa parte della libreria embedded `javax.crypto`
[8] Un nuovo progetto è inizializzato ponendo come unico membro l'utente che ha effettuato la richiesta di creazione.

presenti alcuni metodi fondamentali per la gestione delle funzionalità della piattaforma WORTH, in particolare i metodi relativi alla gestione delle card di progetto, elencati di seguito:

- `public void addCard(Card card)`: aggiunge un oggetto di tipo card al progetto, nella lista corretta. In alternativa è possibile delegare alla classe project la creazione di una nuova Card, passando al metodo il nome e la descrizione della card. Se la card è già esistente, verrà sollevata un'eccezione `CardAlreadyExistsException`. La gestione degli errori è analoga fra i due metodi.

- `public void moveCard(String name, String from, String to)`: si occupa di spostare una card da una lista from ad una lista to, effettuando controlli sui vincoli di movimento delle card, richiesti dalle specifiche del progetto, e controlli sull'effettiva esistenza della card. Se si dovesse rilevare una violazione dei vincoli, sarà sollevata un'eccezione `CardMoveForbidden`; in caso di card non esistente l'eccezione sarà `CardNotFoundException`.

In caso di richiesta di eliminazione di un progetto, il server invoca il metodo `public boolean isAllDone()`, che verifica l'effettivo completamento di tutti i task di progetto, prima di procedere con l'eliminazione. Ogni modifica alle proprietà di un progetto, inclusa la cancellazione, è salvata sul filesystem mediante l'invocazione dei metodi della classe FileHandler.

3.6 Le classi Card e CardEvent

Rappresentano rispettivamente una card e un evento di spostamento della card all'interno di un progetto. La classe Card possiede le informazioni strettamente necessarie per l'identificazione di essa, ossia nome, descrizione testuale e lista nella quale si trova: sono quindi nuovamente stati implementati i metodi get e set di default per recuperare e impostare le variabili di istanza di ogni oggetto. La cronologia degli spostamenti di una card da una lista all'altra è mantenuta in una lista ordinata, in ordine crescente, per data. Ogni evento di spostamento è rappresentato da un'istanza della classe CardEvent, che mantiene fra le variabili locali un riferimento alla data dello spostamento (tipo `long`, rappresentante la timestamp dell'evento in secondi trascorsi da `Epoch`), una stringa indicante la lista di partenza, e un'altra indicante la lista di destinazione. Per garantire l'ordinamento, la classe implementa l'interfaccia Comparable, e per permettere il confronto cronologico fra due eventi, è stato implementato l'override del metodo `compareTo()` di Comparable. In caso di ripristino da backup, la classe Card ammette nel costruttore un riferimento ad un oggetto `ArrayList<CardEvent>`.

3.7 Generazione e riuso di indirizzi multicast con MulticastBaker

La classe MulticastBaker è responsabile della generazione degli indirizzi Multicast necessari al funzionamento della chat. La classe implementa due metodi statici, invocati dal server al momento della creazione di un nuovo progetto, o quando un nuovo progetto verrà eliminato e il relativo indirizzo chat sarà nuovamente disponibile. Dal momento che l'ultimo byte dell'indirizzo multicast è costante e vale 224, la classe mantiene in memoria solamente la variabile intera `multicastSuffix`, rappresentante i primi 24 bit dell'indirizzo, inizialmente posta a uno. Ad ogni invocazione del metodo `getNewMulticastAddress()`, se non sono stati rilasciati indirizzi generati precedentemente, il valore della variabile è incrementato di uno. L'indirizzo creato, è memorizzato mediante un array di interi: ogni posizione rappresenta un otteto dell'indirizzo. La prima posizione dell'array è inizializzata con il valore fisso 0xE0 (224) [9]; i restanti tre byte sono invece calcolati effettuando l'operazione di *shift destro* dei bit e applicando una maschera del valore 0xFF (255)[10] al risultato, per recuperare solamente i primi 8 bit meno significativi: il primo byte è ottenuto applicando direttamente la maschera al suffisso, i restanti due sono calcolati effettuando lo *shift destro* del suffisso rispettivamente di 8 e 16 bit, in seguito è applicata la maschera. L'indirizzo è restituito sotto forma di stringa del formato 224.x.x.x; In caso di indirizzi esauriti (quando il suffisso vale 0xFFFFFF), è restituito al chiamante il valore nullo.

```
    address[2] = ((multicastSuffix >> 0x8) & 0xFF);
```

Listing 3: Inizializzazione del secondo byte dell'indirizzo

[9] Non sono calcolati gli indirizzi a scope amministrativo.
[10] Applicata con l'operazione AND bit a bit

3.8 La persistenza sul filesystem con FileHandler

La fine del capitolo riguardante l'implementazione del server è dedicata alla persistenza dei dati e al ripristino dello stato del sistema in caso di riavvio della piattaforma. La classe FileHandler è implementata utilizzando la libreria Java IO, che offre l'implementazione dei metodi per l'accesso al filesystem. FileHandler possiede tutti i metodi necessari per la lettura e la scrittura dei dati WORTH in memoria secondaria. Un'istanza della classe è creata al momento dell'avvio del server. Il costruttore della classe accetta un parametro, stringa, che individua la directory nella quale sarà presente la directory del sistema worth contenente i dati degli utenti e dei progetti. Al momento della chiamata, è verificata l'esistenza della sottodirectory `./worth` (che chiameremo per semplicità directory base), nel path ricevuto. Se è presente, ritorna; se non è presente tenta la creazione di tali cartelle: in caso di fallimento, è forzata l'uscita dal programma, in quanto non è possibile il funzionamento corretto del server. I dati sono salvati su disco secondo questo schema: la directory base contiene due sottocartelle, una denominata projects, l'altra denominata users. All'interno della prima vi sono ulteriori cartelle, ognuna per ogni progetto: all'interno sarà immagazzinato un file contenente le proprietà del progetto (nome, lista dei membri) e ulteriori n files, ognuno per le n card associate al progetto. La cartella Users, invece conterrà un file per ogni utente registrato alla piattaforma.

I files sono salvati tutti utilizzando il formato JSON. Per serializzare gli oggetti locali in stringhe JSON, è stata anche in questo caso utilizzata la libreria GSON; tuttavia, anziché utilizzare oggetti "generici" della classe JsonObject, sono stati implementati metodi custom per la serializzazione e la deserializzazione delle classi: questa scelta implementativa ha permesso un maggiore controllo sulle proprietà inserite all'interno dei files di backup, permettendo una serializzazione più concisa. I metodi per la serializzazione e la deserializzazione sono contenuti nella classe `shared.serializers.FileSerializerHelper.java`. Un riferimento ai metodi è stato poi passato ad un'istanza della classe GSON, all'interno del metodo privato `setJson()`.

```
public JsonSerializer<User> userJsonSerializer = (user, type, context) -> {

        JsonObject serialized = new JsonObject();

        serialized.addProperty("name",     user.getUsername());
        serialized.addProperty("password", user.getPassword());
        serialized.addProperty("salt",     user.getSalt());
        return serialized;
    };
```

Listing 4: Esempio di serializzatore per la classe User

I metodi della classe FileHandler si suddividono in metodi *loader* e metodi *saver*. I metodi loader sono chiamati solo all'avvio del sistema, e ricercano nei path noti i files relativi ad uno stato precedente, ritornando le strutture dati così come si trovavano al momento dell'arresto del server; in caso di errore nel ripristino viene sollevata una `IOException`. I metodi *saver*, invece, sono invocati dal server ogni volta che lo stato del sistema subisce modifiche: si pensi alla creazione o eliminazione di un progetto, allo spostamento di una card o all'iscrizione di un utente. Questi metodi sono stati dichiarati utilizzando la keyword synchronized, dal momento che un'unica istanza di FileHandler è creata e condivisa dal sistema: possono essere infatti invocati sia dal thread main, sia dai thread RMI; anche in questo caso non è possibile fare assunzioni sulla gestione della concorrenza, dunque è stato necessario introdurre manualmente la sincronizzazione su accessi concorrenti all'istanza.

4 Implementazione del Client

Il Client implementa le funzionalità con cui gli utenti di WORTH interagiscono col server e svolgono le proprie azioni. È stato sviluppato sia un client a riga di comando, sia un'interfaccia grafica. Per quanto riguarda l'interfaccia grafica, la maggior parte dei costrutti sono stati ereditati dal client CLI e riadattati.

4.1 Interfaccia a riga di comando (CLI)

Il client a riga di comando, è stato sviluppato utilizzando Java 8. Si compone di tre classi principali: la classe Main, che si occupa dell'interazione con gli utenti, la classe Client, che implementa i metodi per gestire la connessione TCP col client, e la classe ChatHelper, che si occupa di ricevere e inviare i messaggi relativi alle chat di progetto. I riferimenti al server sono specificati dall'utente al momento dell'avvio del programma, attraverso riga di comando; se non sono specificati argomenti sono utilizzati i valori di default: in particolare, in caso di assenza di riferimenti al server, il client tenterà la connessione sull'indirizzo IP locale della macchina su cui si trova in esecuzione. In caso di connessione impossibile da stabilire, è visualizzato un messaggio di errore e l'esecuzione è terminata. Inoltre, al momento dell'avvio, è creato un riferimento anche al registry RMI del server, per le operazioni di registrazione e callback.

4.1.1 I messaggi verso il Server

All'inizio dell'esecuzione, sono stampati a terminale i comandi disponibili per essere selezionati. L'utente è guidato durante l'esecuzione di un comando: l'input da tastiera è stato realizzato utilizzato un `BufferedStreamReader`. Al completamento di un comando, è effettuato il parsing degli argomenti in un oggetto JsonObject: come per il server è stata utilizzata la libreria GSON. Ogni messaggio inviato dal Client, contiene il campo `method` per specificare l'operazione che si intende eseguire, e il campo `login-name`: questi campi sono obbligatori perché il server possa accettare la richiesta. Al termine della serializzazione in Stringa JSON, il messaggio è inviato attraverso un SocketChannel, e si rimane in attesa di una risposta dal Server: se la richiesta è andata a buon fine, viene visualizzato il contenuto richiesto, altrimenti è visualizzato un messaggio di errore con relativo codice.

4.1.2 RMI e Callback

Al momento dell'operazione di login, il client registra uno stub con i riferimenti agli oggetti locali del tipo `RMIClientInterface` sul server, in modo che possano essere utilizzati meccanismi di callback al verificarsi di determinati eventi: le callback RMI sono state utilizzate per mantenere aggiornata sul client la lista degli utenti iscritti al servizio, e per comunicare i riferimenti relativi alla chat di un progetto durante l'esecuzione del client; sebbene, infatti, il client recuperi la lista degli indirizzi multicast relativi alle chat dei progetti al momento del login, utilizzando le callback è possibile ricevere una notifica relativa a nuovi progetti in tempo reale, senza che l'utente debba aggiornare manualmente le liste di progetto; analogamente viene ricevuta una notifica anche al momento dell'eliminazione di un progetto, con conseguente abbandono del gruppo Multicast.

```
public interface RMIClientInterface extends Remote {
    void notifyUser(String username, Boolean status)    throws RemoteException;
    void notifyChat(String address, String projectname) throws RemoteException;
    void leaveGroup(String address, String projectname) throws RemoteException;
    String getUsername()                                throws RemoteException;
}
```

Listing 5: Interfaccia `RMIClientInterface.java`

4.1.3 Implementazione della chat con DatagramChannel

La chat di progetto è implementata utilizzando IP Multicast: in questo modo, un singolo client può inviare un datagramma UDP ad un gruppo di client interessati, e i messaggi chat possono essere inviati direttamente da client a client, senza utilizzare un server intermedio. Le funzionalità della chat sono state implementate nella classe `ChatHelper`. Al momento dell'istanziazione della classe, sono inizializzate le strutture dati per la memorizzazione dei riferimenti e dei messaggi ricevuti durante la sessione, ed è aperto un socket UDP per la ricezione dei pacchetti Multicast. Per la ricezione dei datagrammi UDP, è stato utilizzato un oggetto della classe `DatagramChannel`, in modalità non

bloccante. Al momento del login, il server comunica al client una mappa chiave-valore, in formato JSON, contenente per ogni entry una coppia del tipo (nome-progetto, indirizzo-chat). Per ogni entry della mappa, il client invoca sull'oggetto ChatHelper il metodo `joinGroup`, passando come argomento una entry della mappa; a questo punto, ChatHelper invoca il metodo `join()` sul DatagramChannel locale. E' mantenuta inoltre in memoria la coppia (nome-progetto, `MembershipKey`), in modo da poter revocare la sottoscrizione ad un gruppo multicast in caso di cancellazione di un progetto. Il metodo `leaveGroup` per l'abbandono di un gruppo è invocato dal server tramite Callback RMI, a cura del metodo `Server.deleteProject()`. Per la lettura dei dati dal canale, è utilizzato un thread che invoca periodicamente il metodo `readDatagramChannel()`, in modo da avere la lista dei messaggi per ogni progetto sempre aggiornata. Il controllo degli accessi concorrenti all'oggetto è garantito dall'utilizzo di strutture dati concorrenti; Inoltre le classi DatagramChannel e Selector, sono *thread safe*, secondo quanto specificato nella documentazione ufficiale di Java. Al momento dell'invio di un messaggio, il client chiama il metodo `sendmessage()` sull'oggetto chatHelper. Il DatagramChannel si occuperà di inviare i messaggi all'indirizzo specificato.

4.2 Interfaccia grafica (GUI)

Per questo progetto, è stato sviluppato anche un client GUI, in modo da rendere più gradevole e intuitivo l'utilizzo della piattaforma. L'interfaccia grafica è stata sviluppata utilizzando il framework JavaFX, supportato da Java 11. Gran parte del *backend* è stato ereditato dal codice sorgente del client CLI, tuttavia sono state eliminate le parti relative alla gestione I/O testuale, in particolare gli oggetti InputStream per la lettura dell'input da tastiera. Le finestre e il layout della GUI sono descritti in files FXML, per semplicità di sviluppo. Ad ogni finestra è stato associato un Controller per gestire le operazioni dell'utente: i metodi della classe Client sono ora invocati attraverso eventi sulle tabelle, o tramite eventi rilevati su oggetti Button. Il client GUI possiede anche classi "lite" delle componenti della piattaforma WORTH, istanziate al momento del recupero di informazioni dal server, in modo da avere strutture dati più robuste rispetto al solo JSON utilizzato dalla CLI.

4.2.1 Implementazione della chat grafica

Sostanziale è la differenza nel modo in cui gli utenti possono accedere alla chat, rispetto al client testuale. Per migliorare l'esperienza d'uso, la TextArea riguardante la chat di progetto è ricaricata al momento della selezione di un progetto dalla lista, e viene aggiornata attraverso un meccanismo di callback direttamente dal thread ChatListener, alla ricezione di un nuovo datagramma.

Al momento della creazione della finestra principale, in seguito al login, è creato anche un oggetto della classe ChatCallback, implementante l'interfaccia ChatCallbackHandler, nota a ChatHelper, che si occupa di aggiornare le strutture dati locali del client che persistono i messaggi. Un riferimento a ChatCallback è impostato sull'istanza ChatHandler: ora, alla ricezione di un nuovo messaggio, ChatHandler invocherà il metodo `handleMessage()` sul proprio riferimento a ChatCallback. È chiamato dunque il metodo `notifyMessage()` del Controller JavaFX [11] associato alla finestra principale: a questo punto, il messaggio ricevuto è salvato in una lista contenuta all'interno di un'istanza di `ConcurrentHashMap`, in modo da essere visualizzato sulla TextArea alla selezione del progetto corrispondente. Se in quel momento è selezionato il progetto relativo al messaggio ricevuto, la TextArea è aggiornata, e il messaggio visualizzato in tempo reale.

```
public void notifyMessage(String projectname, String message){
    if(!messages.containsKey(projectname))
        messages.putIfAbsent(projectname, new ArrayList<>());

    messages.get(projectname).add(message);
    if (projectsTable
        .getSelectionModel()
        .getSelectedItem()
        .getName()
        .equals(projectname)){
        chatArea.appendText(message + "\n");
    }
}
```

Listing 6: Il metodo `notifyMessage()`

[11] Il controller della main window è implementato dalla classe `WorthController.java`.

5 Quick start guide

Il software è stato sviluppato utilizzando l'ambiente di sviluppo Jetbrains Intellij IDEA (ver. 2021.1). Le versioni Java di riferimento sono la versione 8, per Server e client CLI, e Java 11 per il client GUI; il client GUI è stato sviluppato affidandosi anche al framework JavaFX 11.0.2. Sono state utilizzate le seguienti librerie esterne:

- Apache Commons CLI, ver. 1.4 per il parsing degli argomenti da riga di comando;
- Google GSON, ver. 2.8.5, per la gestione dei contenuti JSON.

Le dipendenze sono state gestite utilizzando Maven.

5.1 Istruzioni per la compilazione e l'esecuzione

Sarà possibile compilare il codice facilmente utilizzando l'installazione di Maven presente sul sistema (consigliato). Per compilare il codice sorgente, incluse le dipendenze, sarà sufficiente, da riga di comando, posizionarsi nella directory ./WORTH del pacchetto estratto e lanciare il comando

```
mvn package
```

Maven si occuperà di scaricare e installare le dipendenze sul sistema, di compilare il codice e di creare due pacchetti con estensione .jar, contenenti il codice e le librerie esterne. Per eliminare la cartella target contenente il codice eseguibile, sarà poi possibile eseguire il comando `mvn clean`. In alternativa alla compilazione del codice sorgente con Maven, sarà possibile importare il codice come un nuovo progetto, Intellij IDEA, direttamente dall'IDE. Si dovranno poi aggiungere le dipendenze manualmente, (se non riconosciute automaticamente dal file pom.xml) attraverso il menu `Project Structure -> Libraries` e importando i files jar con le librerie esterne necessarie per l'esecuzione del codice. I files jar sono inclusi nel pacchetto, nella sottocartella `./WORTH/lib`. A seconda delle impostazioni dell'ambiente di sviluppo, potrebbe essere necessario eseguire il programma modificando la variabile d'ambiente `CLASSPATH`, in modo da includere le librerie in formato jar. [12]

5.2 Utilizzo del Server

Una volta compilato il codice sorgente, e creato l'artefatto java, sarà possibile eseguire il Server con il comando:

```
java -jar ./target/Server-jar-with-dependencies.jar [args]
```

Per una lista degli argomenti disponibili, è possibile eseguire il comando con l'opzione `--help`. Se client e server sono eseguiti sulla stessa macchina, è consigliato non specificare argomenti da riga di comando, in modo da utilizzare le impostazioni di default. Non è invece sconsigliato specificare una directory per l'esecuzione attraverso l'opzione `-d <path-to-my-dir>`. Di default, se non esistente, la directory contenente le informazioni di sistema sarà creata in `./worth`. Per interrompere il server, semplicemente inviare SIGINT (Ctrl + C). Il server mostra sul terminale alcuni messaggi di debug durante la sua esecuzione: sono evidenziati con il colore giallo. I messaggi di informazione, sono visualizzati col colore verde.

5.3 Utilizzo dell'interfaccia a riga di comando

Analogamente al server, per eseguire il client CLI si dovrà eseguire il comando:

```
java -jar ./target/Client-jar-with-dependencies.jar [args]
```

Anche in questo caso, se server e client sono eseguiti localmente sullo stesso pc, è consigliato non specificare argomenti da linea di comando e utilizzare le impostazioni di default. Una volta lanciato il client, verrà visualizzato un messaggio con i comandi disponibili. Sarà possibile registrarsi con il comando (signup), o effettuare il login in caso di registrazione già avvenuta. Durante l'esecuzione dei comandi l'utente è guidato nell'inserimento delle informazioni richieste. Per visualizzare la lista dei comandi disponibili, è possibile digitare `help` sulla console. Il client CLI offre tutte i comandi

[12](e.g.: `export CLASSPATH=$CLASSPATH:./WORTH/lib/mylib.jar`)

necessari per l'accesso alle funzionalità del server. In caso di errori, verrà visualizzato un messaggio in colore rosso contenente il codice di errore restituito dal server e una breve descrizione testuale. In caso di successo, verrà visualizzata la risposta del server con i contenuti richiesti, formattati in modo da rendere immediata la lettura, oppure verrà semplicemente visualizzato un messaggio di colore verde contenente il codice di successo restituito del server.

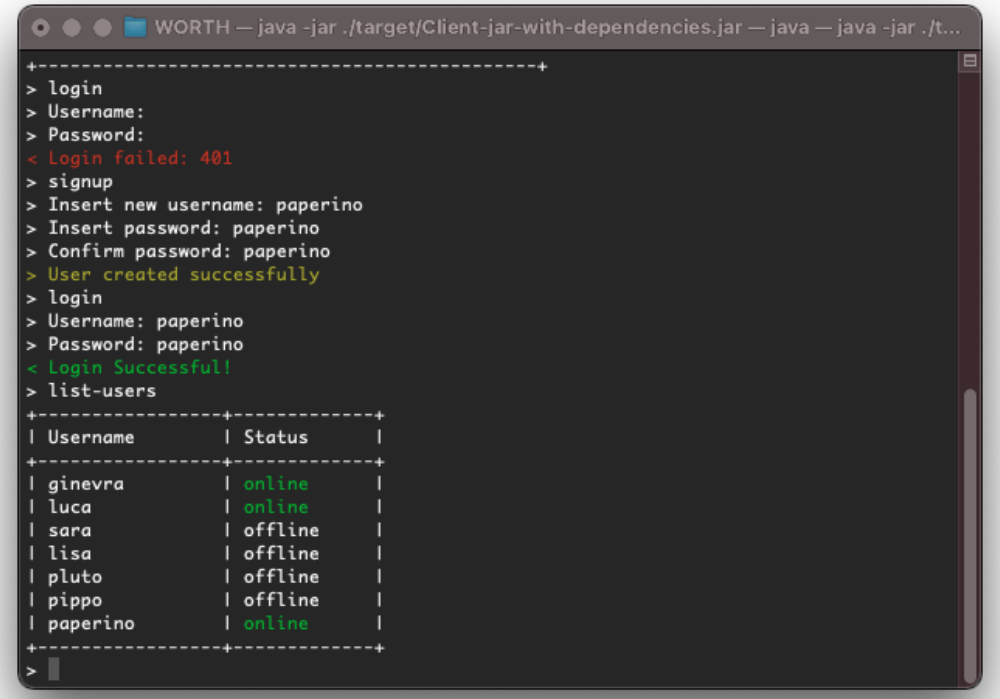

Figura 2: Creazione di un utente, login e lista utenti online. È presente anche un esempio di messaggio di errore

5.4 Utilizzo della GUI

Per compilare e utilizzare il client GUI, è fortemente consigliato l'utlizzo di Maven per recuperare le librerie esterne necessarie: è stato infatti incluso nel file `pom.xml` il plugin esterno `javafx-maven-plugin` per automatizzare la risoluzione dei percorsi delle librerie di JavaFX. Per compilare il codice sorgente della GUI ed eseguirlo posizionarsi nella directory ./WorthGUI ed eseguire il comando:

```
mvn compile javafx:run
```

Gli eventuali argomenti da passare al programma per l'esecuzione dovranno essere specificati all'interno del file pom.xml, in corrispondenza del tag `<commandlineArgs>`; per esempio, per specificare l'indirizzo IP del server e la porta UDP 5678 per la chat, il file andrà editato come segue:

```
<commandlineArgs>-s192.168.0.1 -c5678</commandlineArgs>
```

In alternativa a Maven, sarà possibile anche in questo caso importare il progetto su Intellij IDEA o compilarlo manualmente con il compilatore javac. Sarà però necessario scaricare l'SDK di JavaFX dal sito, oltre alle librerie GSON e Apache CLI, e specificare al momento della compilazione il path contenente le librerie. Dal momento che JavaFX non è incluso nella JRE di Java 11, sarà anche necessario indicare nelle opzioni della JVM il path alle librerie esterne, come segue:

```
--module-path /path/to/javafx/sdk/javafx-sdk-11.0.2/lib --add-modules
javafx.controls,javafx.fxml
```

In seguito all'esecuzione, verrà visualizzata la schermata per il login al servizio e, in seguito, la schermata principale. L'utilizzo del client GUI è molto intuitivo: per iniziare è sufficiente selezionare un progetto dall'elenco in alto a sinistra, a questo punto la schermata mostrerà le informazioni relative al progetto, quali membri e card; la chat sarà aggiornata automaticamente ripristinando i messaggi ricevuti relativi al progetto selezionato e i nuovi messaggi saranno visualizzati automaticamente.

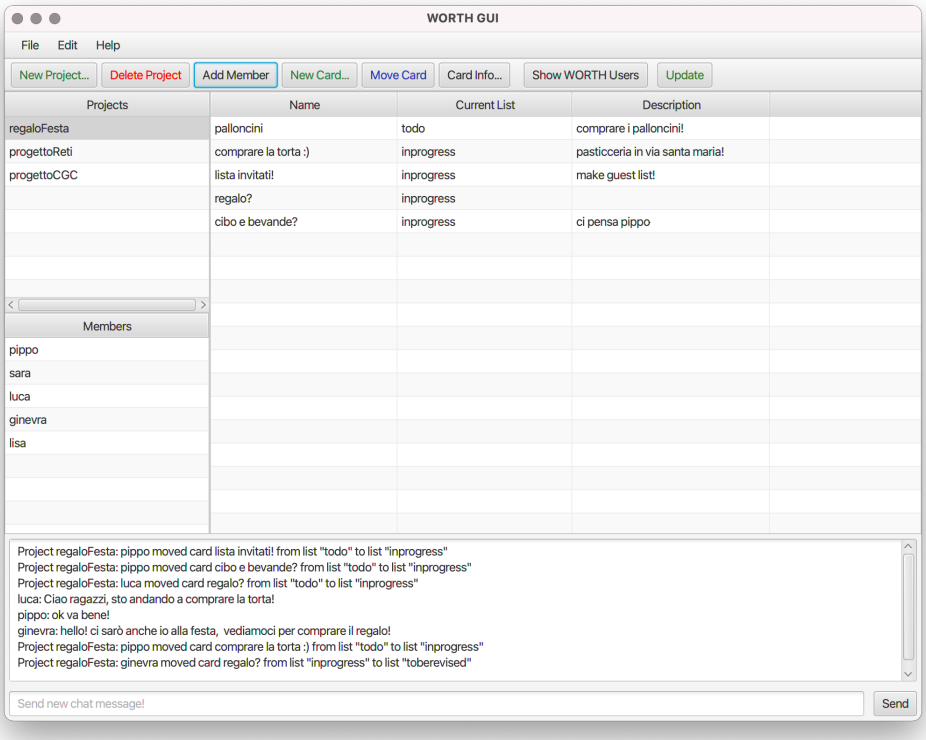

Figura 3: Schermata principale client GUI.

Per invocare le funzioni del server, è possibile utilizzare il menu superiore o i pulsanti sulla Action Bar. In seguito ad un'azione, la schermata è aggiornata automaticamente; tuttavia, in caso ciò non avvenisse, sarà possibile aggiornare manualmente la pagina tramite il pulsante "update", in alto a destra. Se alcuni pulsanti non risultano abilitati durante l'utilizzo, significa che non è possibile eseguire il comando associato: per esempio, per muovere una card, sarà necessario che una card sia selezionata dalla lista; analogamente per cancellare un progetto, aggiungere card o membri al gruppo di lavoro, sarà necessario che un progetto sia selezionato.

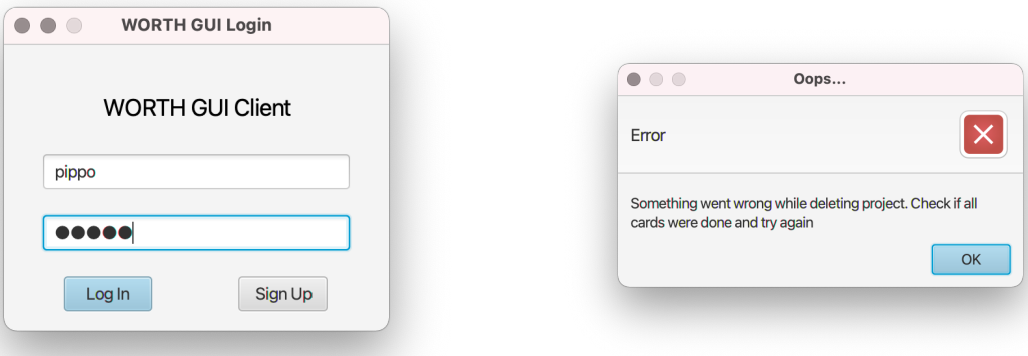

Figura 4: Finestra di login, e un messaggio di errore.