LPR-B-2021: Worth

Gabriele Pappalardo
g.pappalardo4@studenti.unipi.it

Università di Pisa — January 28, 2021

Introduzione

Il corso di laboratorio di reti, tenutosi durante l'anno accademico 2020/2021, ci ha permesso di apprendere il processo di creazione delle applicazioni di rete.
Di seguito sarà riportato il processo di creazione di **Worth**, uno strumento per la gestione di progetti collaborativi che si ispira ad alcuni principi della metadologia *Kanban*.

Nella prima sezione della relazione verrà descritto come è stato realizzato il *programma server*; nella seconda sezione come è stato realizzato il *client* e nelle due ultime sezioni verrà descritta una guida sulla compilazione del programma *client/server* seguita da una serie di immagini illustrative della GUI realizzata.

Struttura del Progetto

Il codice sorgente del progetto è strutturato in tre packages:

- `client`: contiene il codice usato dall'applicativo client;
- `common`: contiene il codice usato in comune dal server e dal client;
- `server`: contiene il codice usato dall'applicativo server.

Le librerie di terze parti sono contenute all'interno della cartella `lib/`.

1 Server

1.1 Architettura del Server

Per l'architettura del Server si è scelto un approccio di tipo *non bloccante con demultiplexing*. Nello specifico, l'architettura del server Worth adotta lo stile del pattern *Reactor*: una volta avviato il server, viene creato un thread `Reactor` che iscrive una `ServerSocketChannel` in attesa di connessioni da parte dei client in un proprio selettore. Parallelamente al thread, che contiene il selettore, viene creata un'instanza della classe `Dispatcher` il cui compito è di sottomettere dei task ad un proprio `ThreadPool`.

1.1.1 Funzionamento del Server

Una volta avviato il main loop principale, il thread Reactor, verifica se ci sono richieste di cambio di tipo di una `SelectionKey` e invoca il metodo `select()` del selettore.

Il thread Reactor bloccato sulla syscall `select()` si risveglia quando un nuovo evento di I/O viene generato o a seguito di una richiesta di un cambio di tipo della chiave da parte di un task che chiama il metodo `wakeup()` del selettore.
A questo punto, dopo aver selezionato le chiavi pronte per delle operazioni, il Reactor le itera verificando che se una chiave selezionata è di tipo *Accettable*, allora vuol dire che un nuovo client desidera connettersi col server. Quest'ulitmo verrà registrato in modalità non bloccante pronto per le operazioni di lettura (`OP_READ`) successive.

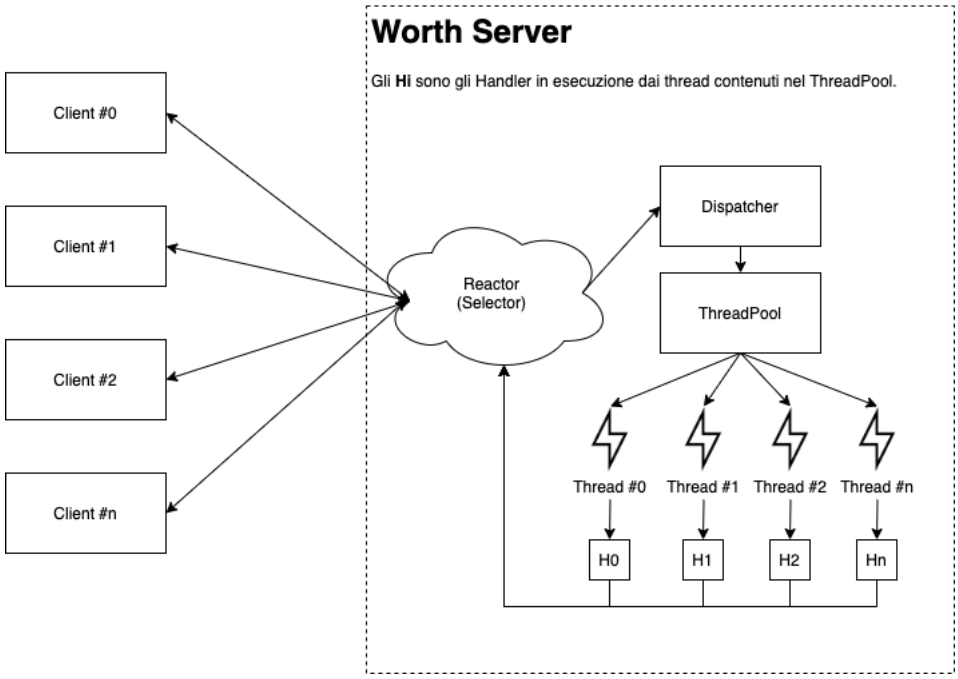


Figure 1: Architettura di Worth

Al momento dell'accettazione di un client da parte del `ServerSocketChannel` viene *allegata* alla nuova chiave un'instanza della classe `RequestHandler`, il cui compito sarà di leggere in primo tempo la richiesta in arrivo all'interno di un proprio buffer in entrata, successivamente farne il parsing, l'elaborazione, e la creazione di una risposta usando il `CommandParser`. Questa verrà poi inserita in un buffer di uscita e infine sarà cambiato l'insieme di appartenenza delle chiave in scrittura.
Il `RequestHandler` è un task che verrà eseguito in maniera concorrente dal ThreadPool contenuto nel `Dispatcher`. L'`Handler` viene sottomesso all'interno del pool se e solo se la chiave selezionata è *Readable* e dopo aver invocato la sua funzione `read()` definita dall'interfaccia `ReadableHandler`. Se la `read()` di un RequestHandler verifica che sono stati letti 0 bytes allora non accade nulla; se il valore ritornato è l'EOF allora il metodo chiama sia la cancellazione della chiave, sia la richiesta di un logout forzato nel momento in cui un client avesse fatto accesso al server precedentemente.

Un `RequestHandler` che ha pronta una risposta da fornire ad un client richiede un cambio operazione della `SelectionKey` di appartenenza in scrittura, in modo tale che nella prossima iterazione del main loop il cambio richiesto venga effettuato.

Il thread Reactor è il responsabile effettivo del cambio del tipo della chiave. Quando una chiave viene selezionata dall'insieme delle scritture, viene chiamato il metodo `write()` dell'handler (definita dall'interfaccia `WritableHandler`) contenuto nell'attachment della chiave e la risposta per il client verrà spedita.

Oltre al `RequestHandler` vi è anche un `ChatHandler` che si occupa di ascoltare i messaggi spediti in un gruppo Multicast e di conseguenza inserirli nella chat di progetto per essere letti in futuro dagli utenti di progetto.

Gli Handler usati dal server Worth sono mostrati nella *Figura 2*.

1.2 Protocollo Richiesta-Risposta

Il server e il client comunicano attraverso lo scambio di specifici messaggi. L'insieme dei messaggi di richiesta e di risposta formano il protocollo di Worth.
Un **messaggio di richiesta** è formato da un corpo codificato in JSON contenente due parti fondamentali

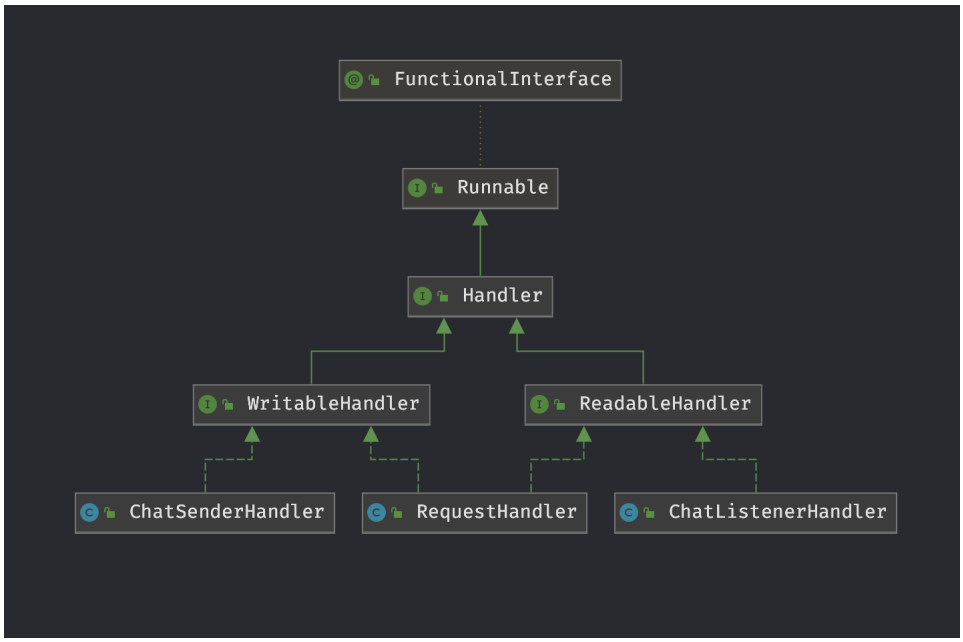


Figure 2: Gerarchia degli *Handler*

e una terza opzionale:

- un'operazione (`operation`);
- un `payload` (non obbligatorio);
- infine, un `sessionId` (obbligatorio per determinate operazioni).

Il payload del messaggio assume significato in base all'operazione fornita, ad esempio, se un client volesse effettuare l'autenticazione all'interno della piattaforma dovrà inviare la seguente richiesta:

Command Line

```
{
    "operation": "LOGIN",
    "payload": {
        "email": "g.pappalardo4@studenti.unipi.it",
        "password":"gabriele"
    }
}
```

Il payload del messaggio di richiesta contiene le credenziali dell'utente che vuole effettuare il login all'interno di Worth. Il server, una volta ricevuta la richiesta, controlla se l'operazione contenuta in `operation` è valida, se l'esecuzione richiede accessi priveligiati e se richiede la presenza di un `payload`.

Dopo aver effettuato tutti i controlli richiesti, il server risponderà con un **messaggio di risposta**, strutturato nel seguente modo:

- lo stato della richiesta (`status`);
- ed un `payload`, anche questa volta, non obbligatorio.

Il campo `status` può assumere soltanto due valori dal signficato immediato: `SUCCESS` o `FAILURE`. In caso di fallimento della richiesta, il server inserirà all'interno del payload un messaggio esplicativo del problema.

Il messaggio di risposta di `LOGIN` contiene, all'interno del proprio payload, un `sessionId`. Questo identificativo servirà al client per effettuare delle operazioni che possano modificare lo stato del server come, ad esempio, la creazione dei progetti.

Esempio di una richiesta con `sessionId`:

Command Line

```
{
    "operation": "LIST_ONLINE_USERS",
    "sessionId": 546856406
}
```

Il `sessionId` viene calcolato in base all'utente che ha effettuato l'autenticazione e all'orario dell'operazione, infatti esso è l'intero calcolato dalla funzione: `Objects.hash(sessionUser, timestamp)`.
Quando un utente decide di effettuare il `LOGOUT` allora il `sessionId` verrà marcato come invalido e dunque, non potrà essere più riutilizzato nelle richiesta successive.

Table 1: **Operazioni di richiesta**

Operazione	Richiede Autenticazione	Partecipazione ad un progetto
LOGIN	No	No
LOGOUT	No	No
LIST_ONLINE_USERS	Sí	No
LIST_USERS	Sí	No
LIST_PROJECTS	Sí	No
CREATE_PROJECT	Sí	No
CANCEL_PROJECT	Sí	Sí
ADD_MEMBER	Sí	Sí
SHOW_MEMBERS	Sí	Sí
ADD_CARD	Sí	Sí
MOVE_CARD	Sí	Sí
SHOW_CARD	Sí	Sí
SHOW_CARDS	Sí	Sí
READ_CHAT_MESSAGES	Sí	Sí

1.3 Utilizzo di RMI

Il server mette a disposizione un registro `RMI` contenente tre servizi:

- *servizio registrazione*;
- *servizio di iscrizione alle notifiche di stato degli utenti*;
- *servizio di iscrizione alle notifiche di stato dei progetti*.

Il *servizio di registrazione* permette al client di iscrivere un nuovo utente all'interno della piattaforma. La classe che implementa il servizio di registrazione richiede solo *nome*, *cognome*, *email* e *password* da parte dell'utente. Se all'interno vi è già presente un utente con un'email già registrata, allora il servizio solleverà un'eccezione `UserRegisteredException`.

Il *servizio di iscrizione alle notifiche di stato degli utenti*, implementato tramite RMI Callbacks permette, ad un client iscritto, di ricevere informazioni sui cambi di stato di un utente, quindi se è online o offline. Il servizio RMI invia un'instanza della classe `UserStatus` che contiene un utente associato e il nuovo stato (rappresentato da un booleano).

Il *servizio di iscrizione alle notifiche di stato dei progetti*, implementato anch'esso tramite RMI Callback,

invece, permette ad un client di essere notificato sull'avvenire di un evento su un progetto a cui partecipa, ad esempio, l'aggiunta di una nuova Card, lo spostamento da un flusso ad un altro, l'aggiunta ad nuovo progetto e, per finire, l'eliminazione di un progetto. Questo componente permetterà alla GUI del Client una certa interattività.

1.4 Server's Threads

All'interno di Worth sono presenti dei Threads secondari che svolgono diversi ruoli, quali:

- `SessionRipper`: questo thread demone si occupa di eliminare le sessioni che presentano inattività dopo 30 minuti;
- `ProjectSaver`: è un thread che si occupa di salvare modifiche dei progetti periodicamente in modo asincrono, altrimenti, se il server viene avviato con l'opzione `-s 0` il salvataggio del progetto verrà affidato ad un task di un `ThreadPool` interno.

1.5 Gestione dei Progetti

Le richieste dirette ai progetti vengono gestite dalla classe `ProjectManager` che controlla i permessi degli utenti che richiedono modifiche ad un progetto. Ogni operazione che comporta la modifica di stato di un progetto invoca l'intervento della classe `ProjectSaver`. Infine i progetti salvati nel file system della macchina che esegue il server vengono ripristinati dal `ProjectStorage`.

1.5.1 Creazione di un progetto

Un progetto viene creato a seguito di una richiesta da parte di un utente. Se il nome del nuovo progetto è già stato utilizzato allora l'utente verrà avvisato del problema, altrimenti, viene creato un nuovo progetto a cui verrà assegnato un Multicast Group. Il gruppo viene generato dalla classe `MulticastGroupDispenser` che assicura l'unicità degli indirizzi tra i progetti. Quando un nuovo progetto viene creato viene creato anche un nuovo `ChatHandler` che si occuperà di salvare i messaggi inviati tra gli utenti nella chat di progetto in modo tale che quando un client invia la richiesta `READ_CHAT_MESSAGE` si possano leggere i vecchi messaggi [1].

1.5.2 Come viene salvato un progetto all'interno del File System

Il salvataggio di un progetto viene affidato a due classi fondamentali: `ProjectSaver` e `ProjectStorage`. Il `ProjectStorage` salva un progetto all'interno del file system del computer seguendo una serie di passi:

1. viene effettuato un hash in `MD5` [2] del nome del progetto;
2. all'interno della cartella `APP_DATA/projects` [3] viene creata una nuova cartella con nome l'hash;
3. se non presente, viene creato un file `manifest.json` contenente tutti i metadati del progetto, quali: nome, creatore, Mutlicast Group, membri e infine i flussi che contengono le card;
4. se non presente, viene creata una cartella denominata `cards` che contiene tutti i file rappresentanti le cards all'interno del progetto;
5. se all'interno del progetto vi sono delle cards allora per ciascuna di essere viene effettuato l'hash in `MD5` del nome e di conseguenza viene creato un file JSON contenente i metadati della card, ovvero: nome, descrizione e una history.

Il salvataggio del progetto è un'operazione atomica, grazie ad una `ReadWriteLock` all'interno di ogni singola instanza di `Project` viene garantito che nessuno Thread possa fare delle modifiche durante la fase di scrittura nel File System.

Nel listato di sotto vi sono presenti un file manifest di un progetto e una card.

[1] Ogni chat di progetto può contenere fino ad un massimo di 32 messaggi, i messaggi dopo la chiusura del server non saranno scritti in memoria fisica.

[2] L'hash viene usato per creare dei filename non contenenti caratteri speciali che non sarebbero consentiti normalmente dal file system di un Sistema Operativo (i.e. caratteri non ASCII).

[3] La cartella contiene tutti i file di configurazione del server, un file JSON con gli utenti iscritti e una cartella dei progetti. Questo path viene definito dall'utente al momento dell'esecuzione del programma server di Worth.

Command Line

[manifest.json]

```
{
  "projectName": "Worth",
  "creator": "g.pappalardo4@studenti.unipi.it",
  "chatAddress": "239.0.0.1",
  "members": [
    "g.pappalardo4@studenti.unipi.it",
  ],
  "flows": {
      "TODO":["027acbd311b5294f84eb9b05258eb674"],
      "WORKING_IN_PROGRESS": [],
      "TO_BE_REVISED": [],
      "DONE": []
  }
}
```

Command Line

[027acbd311b5294f84eb9b05258eb674.json]

```
{
    "name": "Drinks for the Journey",
    "description": "We have to order 1 Pan Galactic Gargle Blaster",
    "history": [
        {
            "event": "CREATED",
            "timestamp": 1610893944456
        }
    ]
}
```

Command Line

[Esempio di una directory di progetto]
<Hash del nome di Progetto>
|– cards
| |– <Hash del nome di una card>.json
|– manifest.json

1.5.3 Restore dei Progetti

Quando il server viene avviato, un'instanza della classe `ProjectStorage` verifica se all'interno della cartella `APP_DATA/projects` vi siano dei progetti da caricare in memoria.
I progetti vengono caricati in una mappa concorrente grazie all'ausilio dello `UserStorage` e della classe `PersistentProjectDeserializer` che implementa l'interfaccia `JsonDeserializer` provvisti dalla libreria `GSON` di *Google*.
Lo `UserStorage` viene utilizzato per poter mappare un'email di un membro alla rispettiva instanza della classe `User`.
Le card del progetto vengono caricate usando il campo `flows` del manifest del progetto, per ogni nome di card hashato viene concatenata l'estensione `.json` e verifica se all'interno della cartella `cards` sia presente la card e in caso affermativo la inserisce nella lista di progetto corretta.

2 Client

L'applicativo Client di Worth è stato implementato attraverso una GUI usando il framework `JavaFX`.

2.1 Graphical User Interface

La GUI di *Worth Client* sfrutta tutte le operazioni che il server mette a disposizione. Attraverso l'applicativo è possibile registrarsi al servizio di Worth e successivamente fare il login.
Come richiesto da progetto la registrazione di un utente avviene usando `RMI`, mentre il login usa una connessione `TCP` che viene mantenuta per tutta la durata della sessione di un utente fino all'operazione di logout.

2.1.1 Login e Logout

Il login dell'utente avviene attraverso *email*, *password* e l'invio del comando `LOGIN` al server contenente le credenziali. Se quest'operazione avviene con successo allora il payload della risposta conterrà due informazioni che verranno utilizzate durante tutta la sessione dell'utente:

- una lista degli utenti di Worth con relativo stato (Online/Offline) che verrà aggiornata grazie al servizio offerto dal registro RMI;
- un `sessionId` che verrà automaticamente cachato per le richieste che modificano lo stato del server future.

Altrimenti, se l'operazione dovesse fallire, verrà mostrato un messaggio di errore, come mostrato nella figura 5.3.

Un utente dopo aver effettuato il login con successo, si iscrive automaticamente a due servizi offerti dal registry `RMI` del server. Il primo servizio invia delle notifiche push che permettono di aggiornare una lista locale degli stati degli utenti come richiesto da specifica. Il secondo servizio, invece, invia delle notifiche che permettono di ricevere informazioni sul cambio di stati di progetti. Infatti viene creato un Thread JavaFX che resta in attesa e che, all'emissione di un evento da parte del server, il client si aggiorna di conseguenza.

Al momento del logout il client si disiscrive dai servizi RMI elencati precedentemente ed invia un messaggio di `LOGOUT` al server che invaliderà il `sessionId` per gli usi futuri.

2.1.2 Dashboard

La **dashboard** permette di creare nuovi progetti, visualizzare un progetto esistente, creare nuove card, aggiungere membri al progetto, aprire la chat di progetto e infine l'eliminazione di un progetto.
Ogni operazione elencata effettua una richiesta al server che verrà inviata attraverso la connessione `TCP` instarauta al momento del login. Ad esempio, quando una card viene spostata da una lista ad un'altra viene inviata una richiesta `MOVE_CARD`.

Quando si seleziona un progetto dalla *Projects Lists* vengono caricate le card all'interno delle quattro liste presenti nella dashboard. Una card è composta da un titolo, una descrizione e tre bottoni. Il primo bottone *Details* permette di vedere la descrizione completa di una card se questa dovesse essere eccessivamente grande, il seconodo bottone *Log* permette di vedere la history di una card e l'ultimo bottone *Move* permette di spostare una card da una lista all'altra in base all'automa previsto dall'assignment.

2.1.3 Chat

Ad ogni progetto viene associato un gruppo multicast che verrà utilizzato da un client al fine di potere accedere alla chat per leggere e inviare i messaggi.
Nel client grafico di Worth quando un utente clicca il pulsante *Show Chat* viene creato un nuovo thread Java FX, il quale crea un `MulticastSocket` in modalità bloccante tale che attenda, in modo asincrono, l'arrivo di messaggi da parte dei membri del gruppo.
Successivamente viene effettuata una richiesta di tipo `READ_CHAT_MESSAGE` al server al fine di recuperare

la lista di messaggi di un progetto dall'avvio del server. Questo serve a garantire che un membro sia sempre aggiornato sull'avvenire degli eventi del gruppo. Tali messaggi possono essere un massimo di 32. Un utente per inviare un messaggio scrive un testo nella casella di testo libero e preme il pulsante *Send* per inviarlo attraverso un normale `DatagramSocket`. Inoltre, quando una card del progetto viene spostata da una lista ad un'altra il server invierà in chat un messaggio esplicativo dell'avvenimento, il mittente del messaggio è il server.

3 Compilazione ed Esecuzione

Il client e il server di Worth usano la OpenJDK 15. Per facilitare la compilazione e dunque l'esecuzione consiglio l'uso dell'IDE: *IntelliJ IDEA* creato da *JetBrains*. Una volta scaricato l'IDE basta aprire la cartella del progetto ed eseguire la configurazione `MainServer` e `MainClient`.

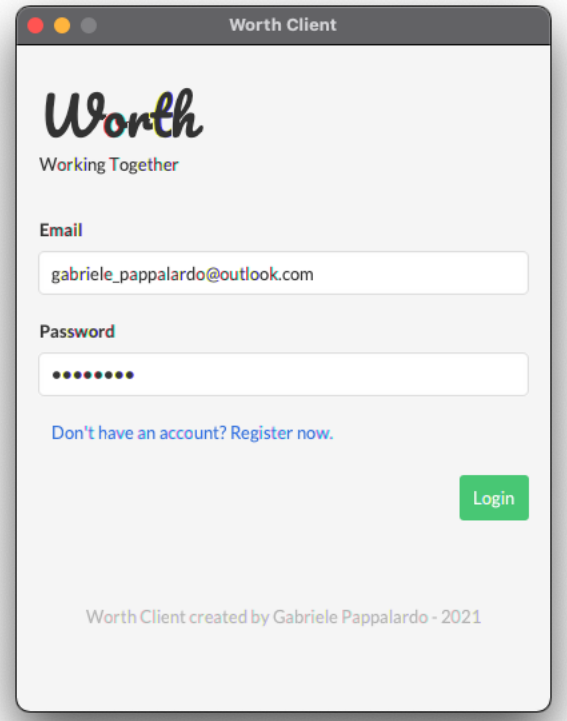
Alternativamente, all'interno della cartella `src` vi è contenuto un file `README.md` che spiega come compilare Worth con `javac`.

4 Librerie e Framework Usati

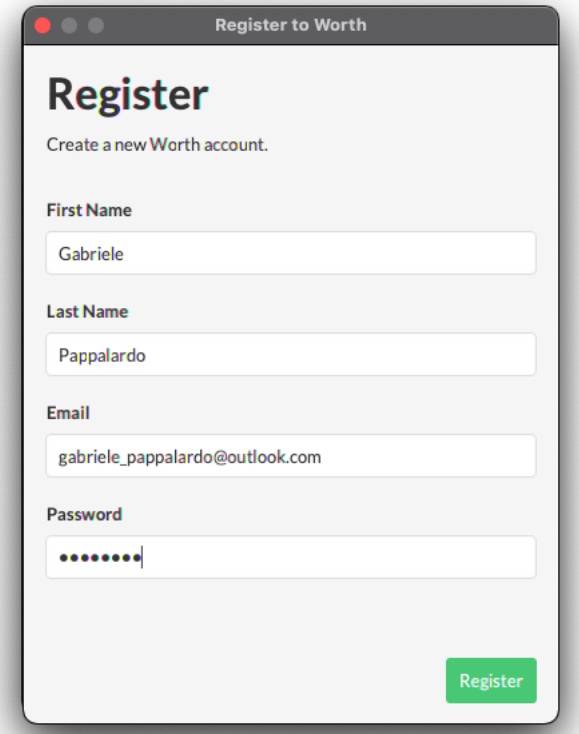
Worth Client e Server utilizzano le seguenti librerie Open Source:

- `OpenJFX`: usata per la creazione della GUI del client (*GPL v2 with the Classpath exception*, https://github.com/openjdk/jfx);
- `ControlsFX`: usata per estendere i componenti provvista da `OpenFX` (BSD, https://github.com/controlsfx/controlsfx);
- `Gson`: usata per effettuare l'encoding/decoding (*Apache License, Version 2.0*, https://github.com/google/gson);
- `Guava`: usata per le implementazioni delle hash functions (*Apache License, Version 2.0*, https://github.com/google/guava)
- `Apache Commons CLI`: usata per effettuare il parsing delle options inserite da terminale (*Apache License, Version 2.0*, https://github.com/apache/commons-cli);
- `Apache Commons Collection`: usata per implementare strutture dati di supporto come il Ring Array per i messaggi della chat (*Apache License, Version 2.0*, https://github.com/apache/commons-collections)

5 Immagini



(a) Login Screen



(b) Register Screen

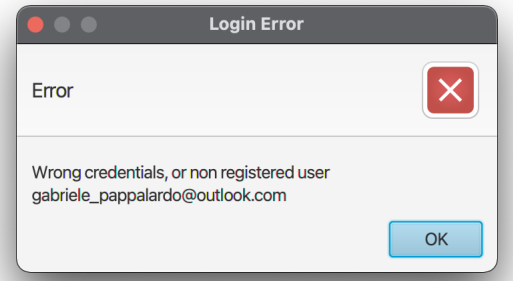


Figure 4: Login Error

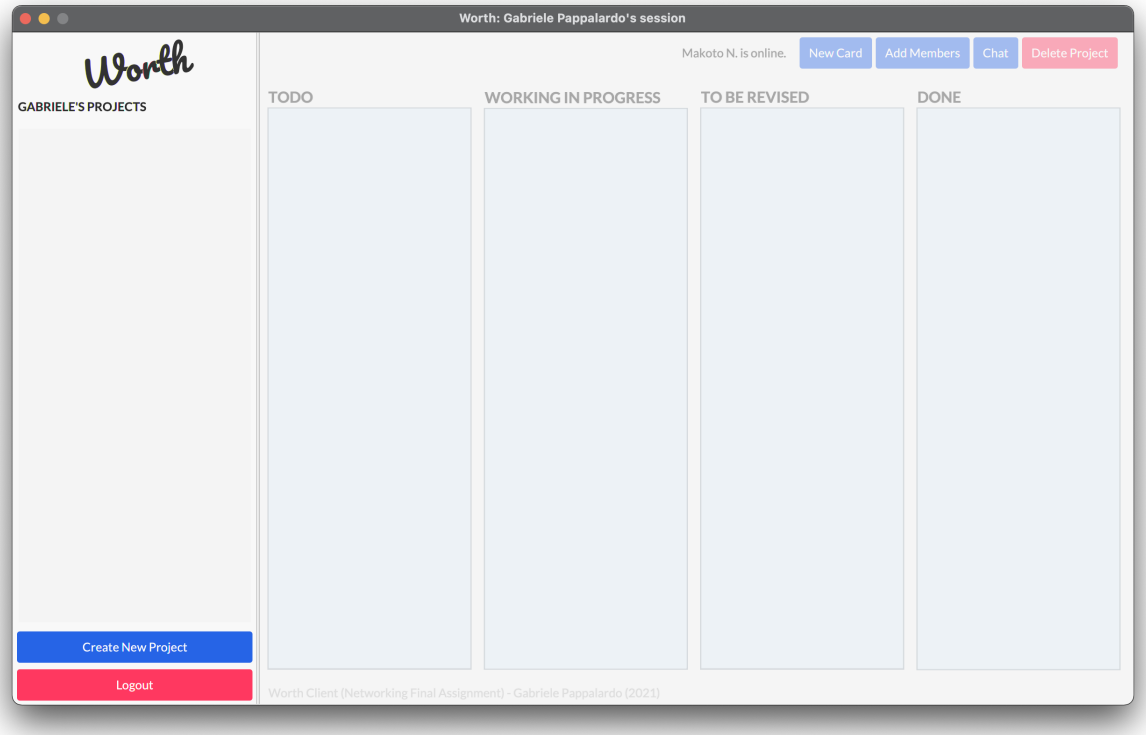


Figure 5: Project Dashboard

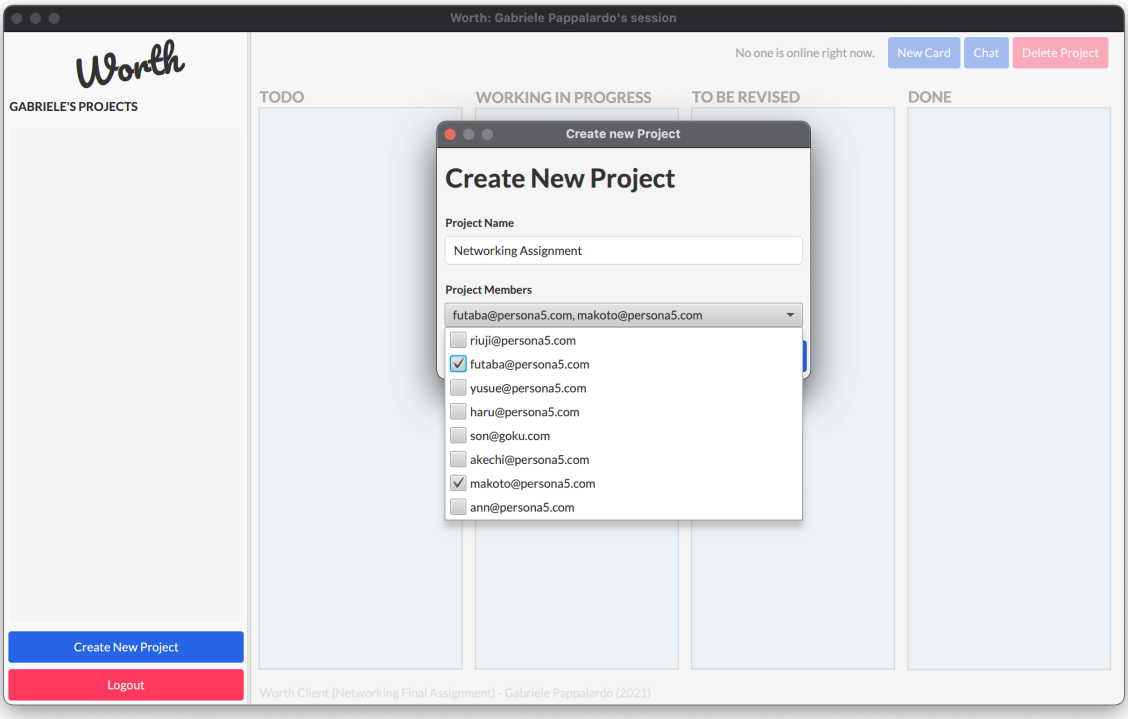


Figure 6: Creazione di un progetto

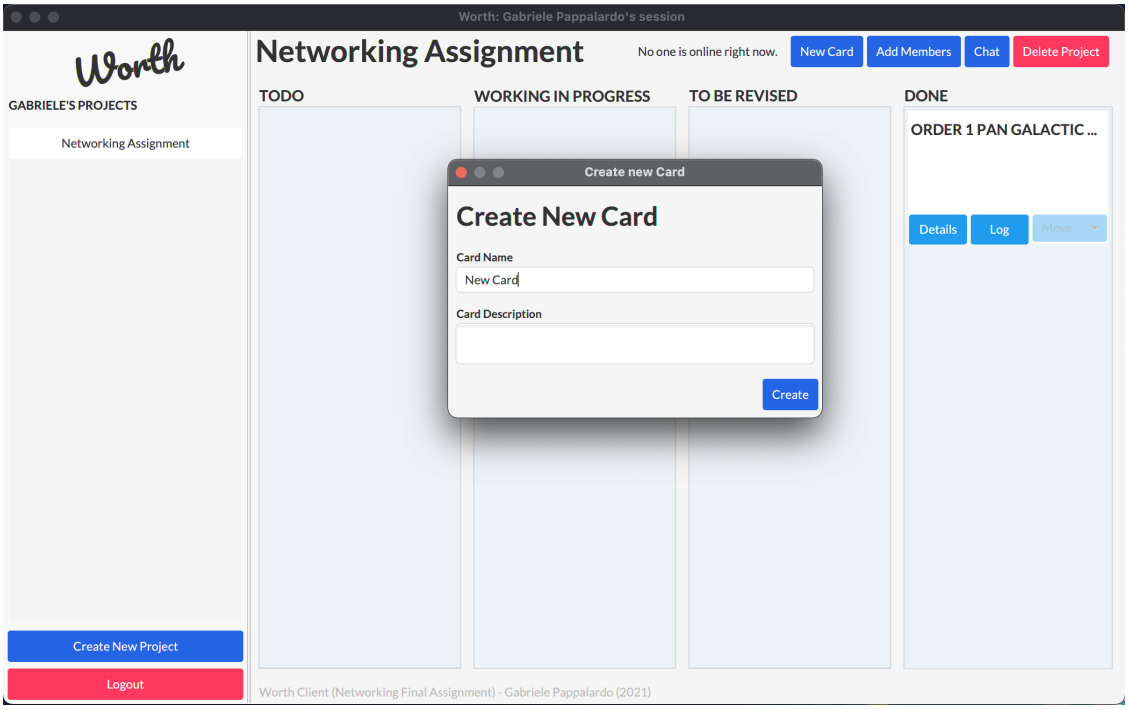


Figure 7: Creazione di una card

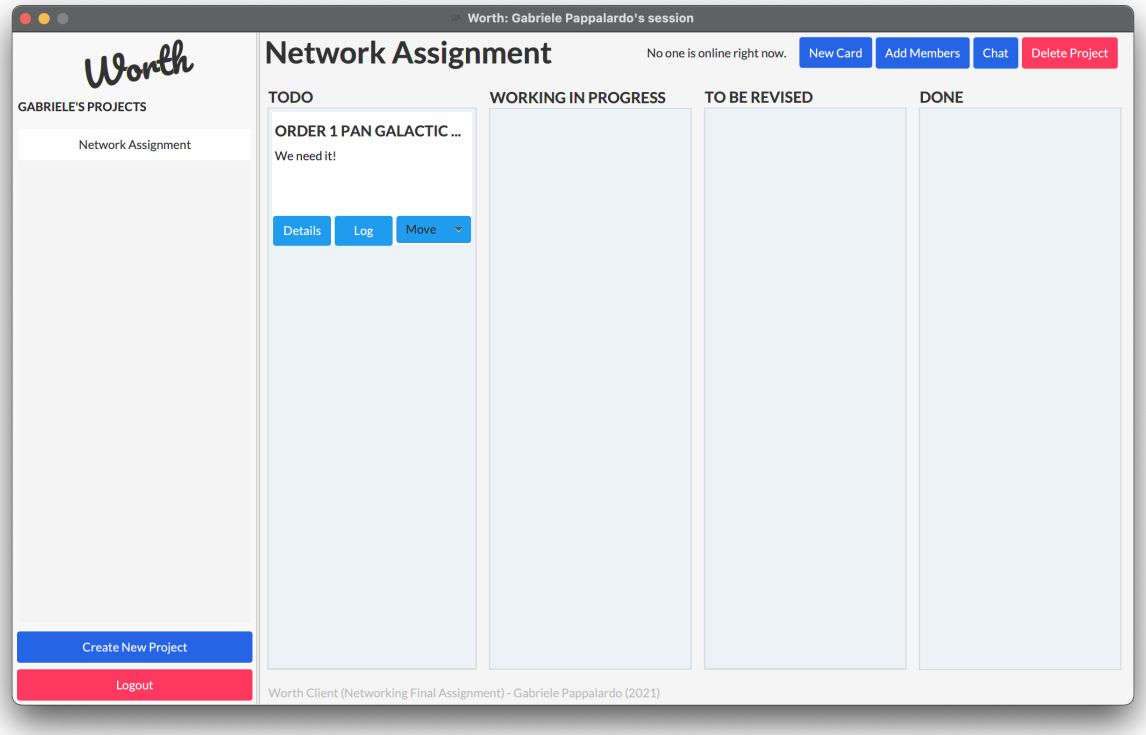


Figure 8: Dashboard con una card

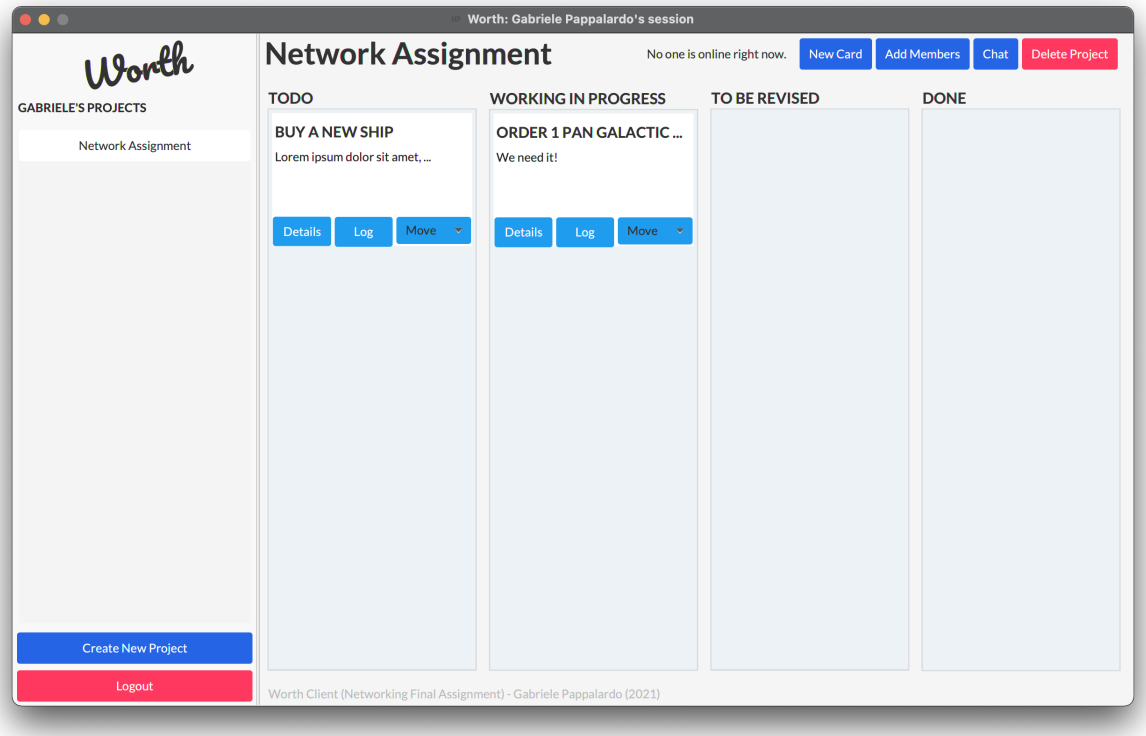


Figure 9: Dashboard con due card

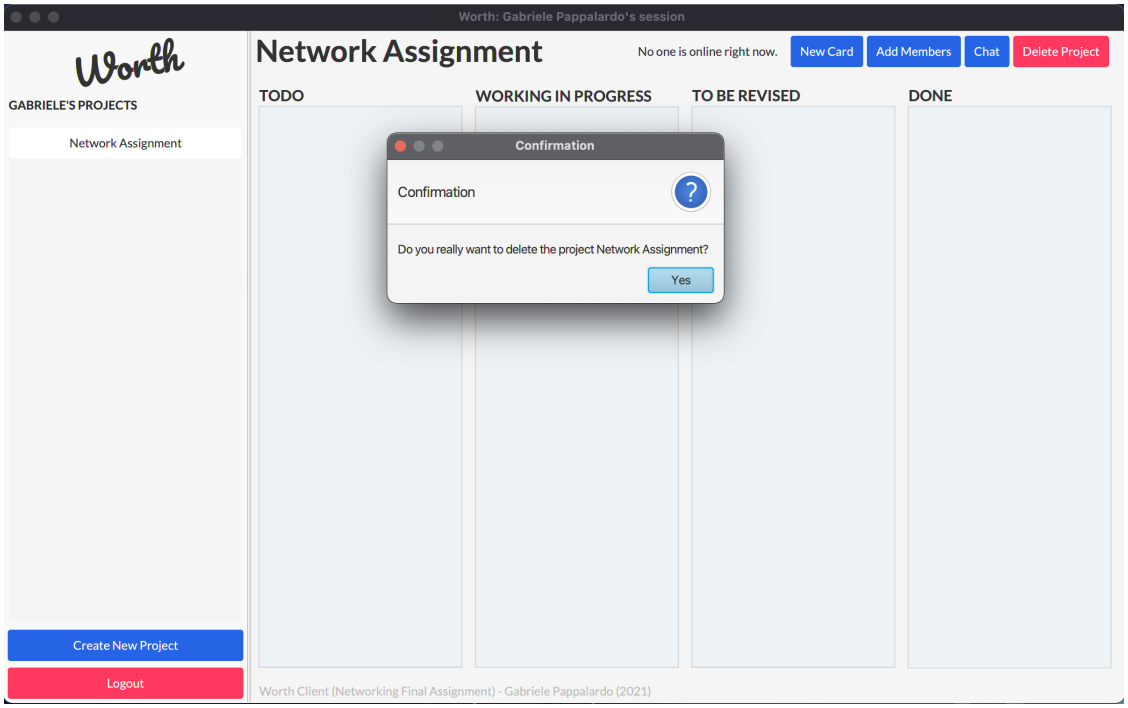


Figure 10: Eliminazione di un progetto

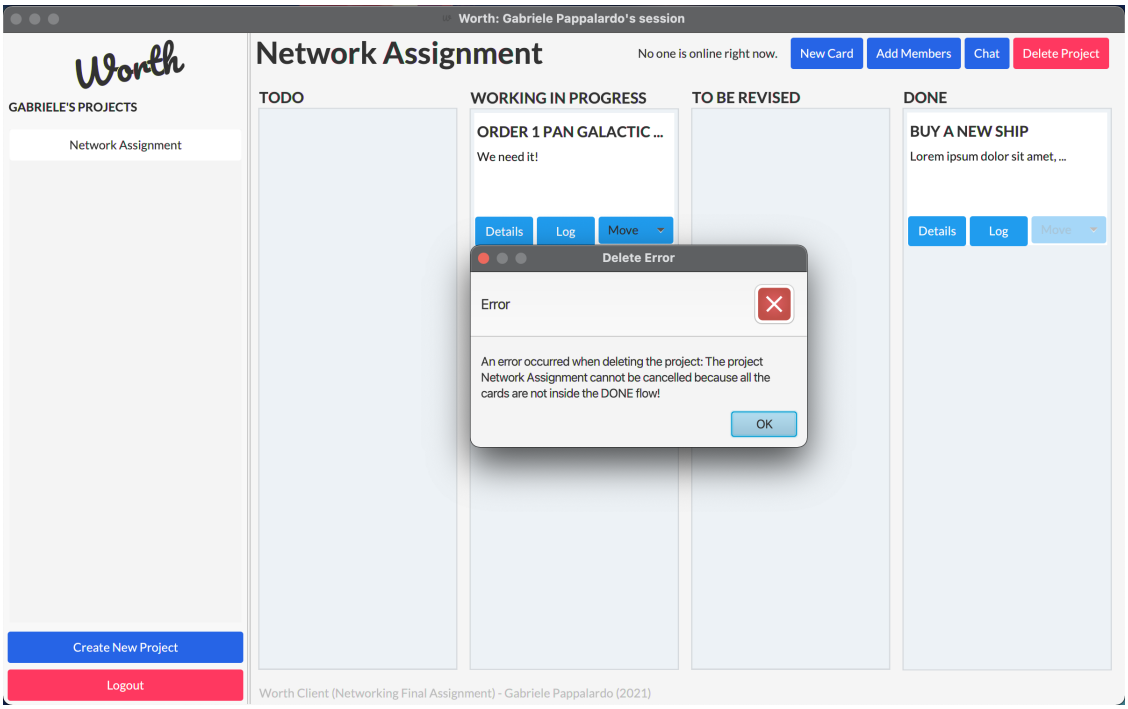


Figure 11: Eliminazione progetto fallita

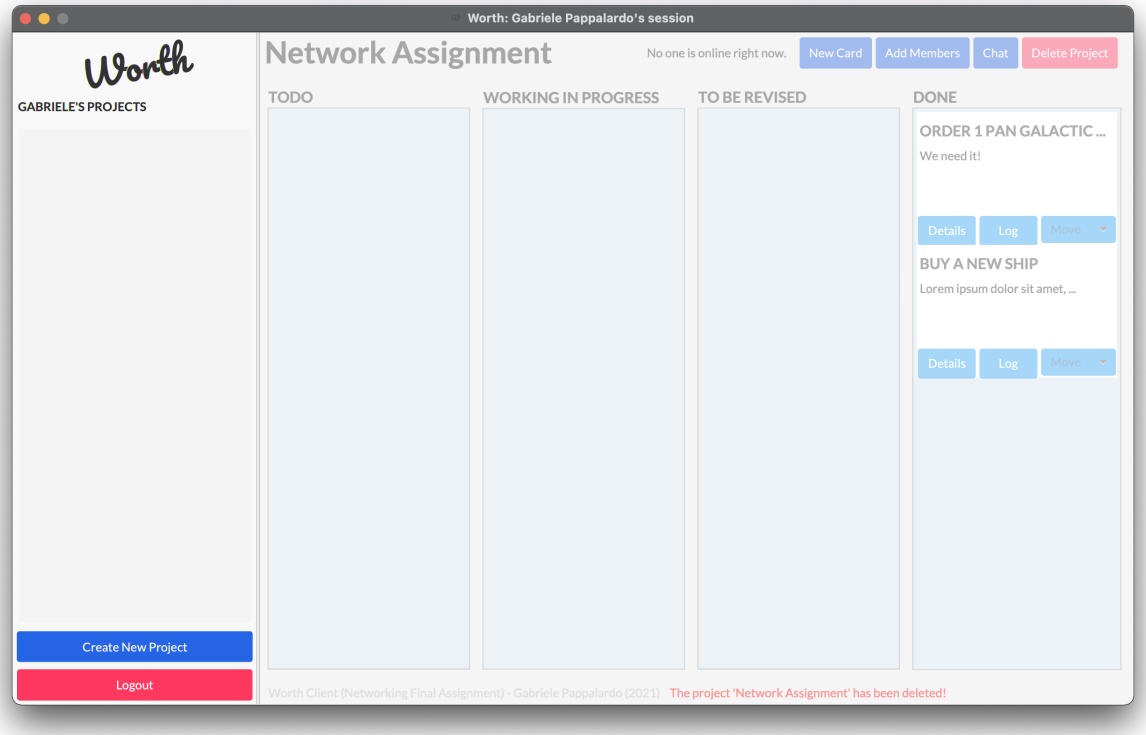


Figure 12: Eliminazione progetto riuscita

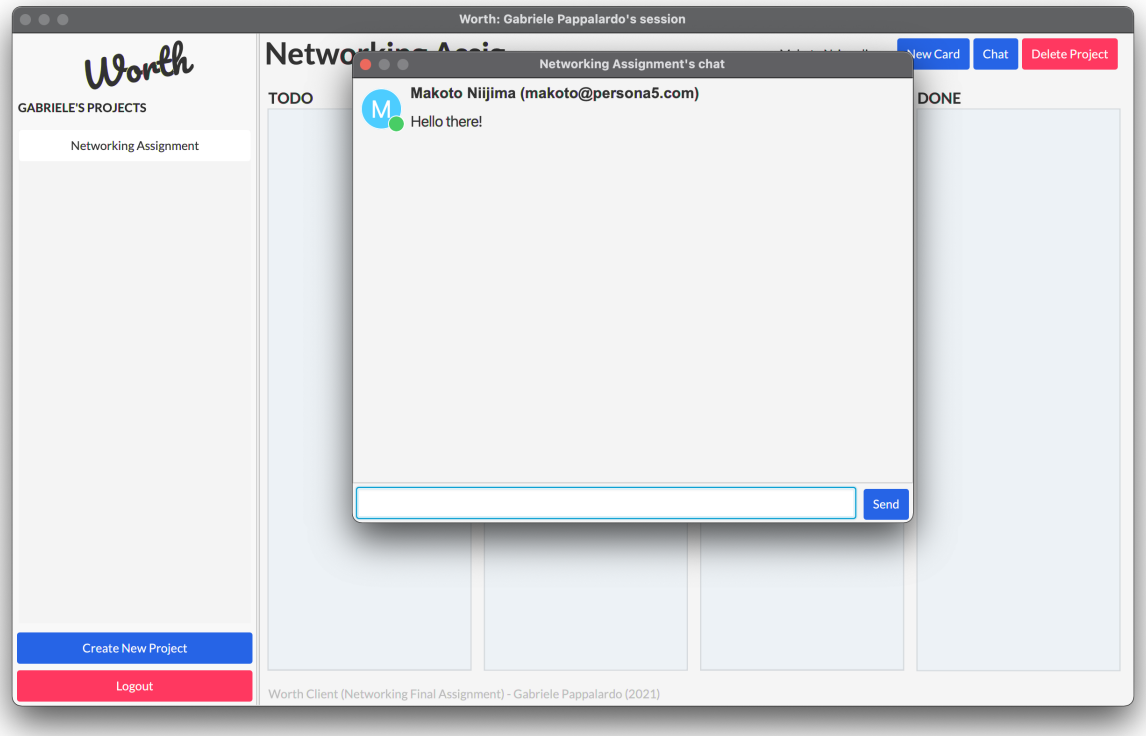


Figure 13: Chat di progetto