Cluedo

Marica Pasquali
marica.pasquali@studio.unibo.it

Settembre 2023

Il progetto consiste nella realizzazione di una versione distribuita del noto gioco da tavolo *Cluedo* [3].

In questa versione del gioco, gli utenti possono creare o partecipare a partite create da altri giocatori. Inizialmente viene creata un "room" di attesa, in cui gli utenti attendono l'arrivo di altri utenti. Appena raggiunto il numero minimo di giocatori ossia 3, la partita può essere avviata da qualsiasi giocatore nel gioco. La partita può avere al massimo 6 giocatori.
Durante il proprio turno, un giocatore potrà :

- tirare un dado per potersi muove nella casa. Le facce del dado raffigurano le stanze della casa + uno o più ingressi;
- formulare un'*ipotesi* nel caso sia in una stanza;
- formulare un'*accusa*.

Nel caso di un'*ipotesi* gli altri giocatori devono allora confutarla (se possibile), mostrando al massimo una carta che contenga uno degli elementi nominati dal giocatore di turno; tale azione dimostra che la carta sicuramente non può essere all'interno della busta contenente la soluzione dell'assassinio. Nel caso di un'*accusa*, il giocatore di turno, se l'accusa non è corretta, esce dal gioco; altrimenti se "pronuncia" un'accusa esatta, le carte della soluzione sono rivelate agli altri giocatori e la partita finisce. I giocatori che hanno effettuato accuse sbagliate devono continuare a esibire le carte agli altri partecipanti per il resto della partita, oppure possono redistribuire le proprie carte e abbandonare il gioco.

Indice

1 Obiettivi del progetto

L'obiettivo del progetto è quello di realizzare una versione digitale e distribuita del gioco *Cluedo* adottando un'architettura distribuita peer-to-peer con un discovery server.

Il discovery server permetterà ai peer appena connessi di conoscere gli indirizzi degli altri peer online e quindi aggiungersi alla lista dei peer connessi. Successivamente, il peer potrà comunicare con uno o più peer per scoprire se è possibile entrare nella futura partita (max peer 6 per partita). Ogni peer gestiranno lo stato della partita in cui partecipano in modo distribuito comunicando tra di loro. Dopo che l'utente avrà avviato la partita, il peer inizializzerà la partita ossia posizionerà le pedine nel tabellone, posizionerà le armi nelle varie stanze, selezionerà in modo casuale le tre carte soluzione e distribuire le carte rimanenti.

Le comunicazioni tra peer e i server avverranno tramite API RESTful e socket. Le specifiche delle API RESTful e degli eventi socket saranno fornite tramite Swagger (rispettivamente *openapi* e *asyncapi*).

Di seguito viene riportato il diagramma dei casi d'uso di come le varie componenti (peers e discovery server) interagiscono tra di loro.

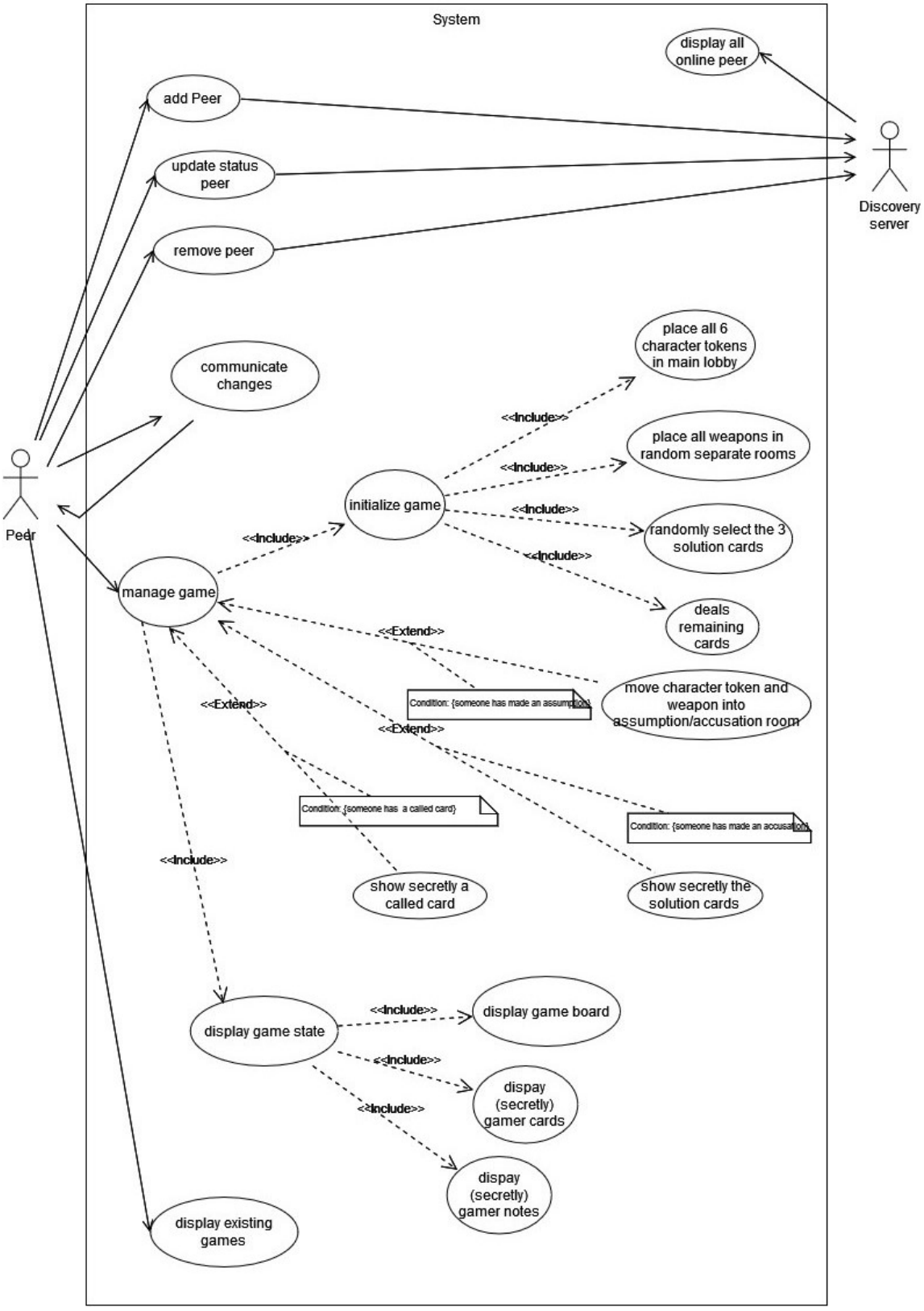


Figura 1: Diagramma dei casi d'uso del sistema

1.1 Expected Deliverables

Ci si impegna a consegnare:

- i sorgenti del discovery server
- i sorgenti del peer
  - i sorgenti del peer lato server per la gestione della comunicazione con gli altri peer e servers
  - i sorgenti del peer lato client per la visualizzazione e l'interazione da parte dell'utente
- Dockerfile che consente di facilitare il deployment dell'applicazione.

1.2 Scenarios

Ogni utente quando accede al client web potrà scegliere di creare una nuova partita o di unirsi ad una già esistente. Dopo aver creato la partita, l'utente verrà inserito nella rispettiva waiting room in attesa dei restanti giocatori (massimo 6). Nel frattempo avrà scelto quale personaggio essere durante il gioco. Quando il numero minimo di giocatori (ossia 3) sarà raggiunto, la partita potrà iniziare e ad ogni giocatore verrà mostrata l'interfaccia di gioco.

Avviato il gioco, ogni giocatore a turno potrà "tirare il dado" per muoversi all'interno della casa e quando si troverà all'interno di una stanza, potrà formulare un'ipotesi oppure formulare un'accusa composte dalla *vittima*, dall'*arma*, e del *luogo* del delitto; o spostarsi attraverso passaggi segreti. La stanza che verrà nominata in un'ipotesi dovrà essere la stessa in cui il giocatore di turno si troverà. Qualunque giocatore potrà prendere appunti (segreti) sulla risoluzione del delitto.

Successivamente alla formulazione di un'ipotesi, gli altri giocatori, uno per volta, dovranno allora confutarla (se possibile), mostrando al massimo una carta che contenga uno degli elementi nominati dal giocatore di turno; tale azione dimostra che la carta sicuramente non potrà essere una delle "carte soluzione" dell'assassinio.

Invece, successivamente alla formulazione di un'accusa, nel caso il giocatore di turno abbia formulato l'accusa corretta, le carte della soluzione verranno rivelate agli altri giocatori e la partita finirà, altrimenti avrà due opzioni o continua a esibire le carte agli altri partecipanti per il resto della partita, oppure potrà redistribuire le proprie carte e abbandonare il gioco.

Di seguito viene riportato il diagramma dei casi d'uso di come gli utenti potranno interagire con il sistema.

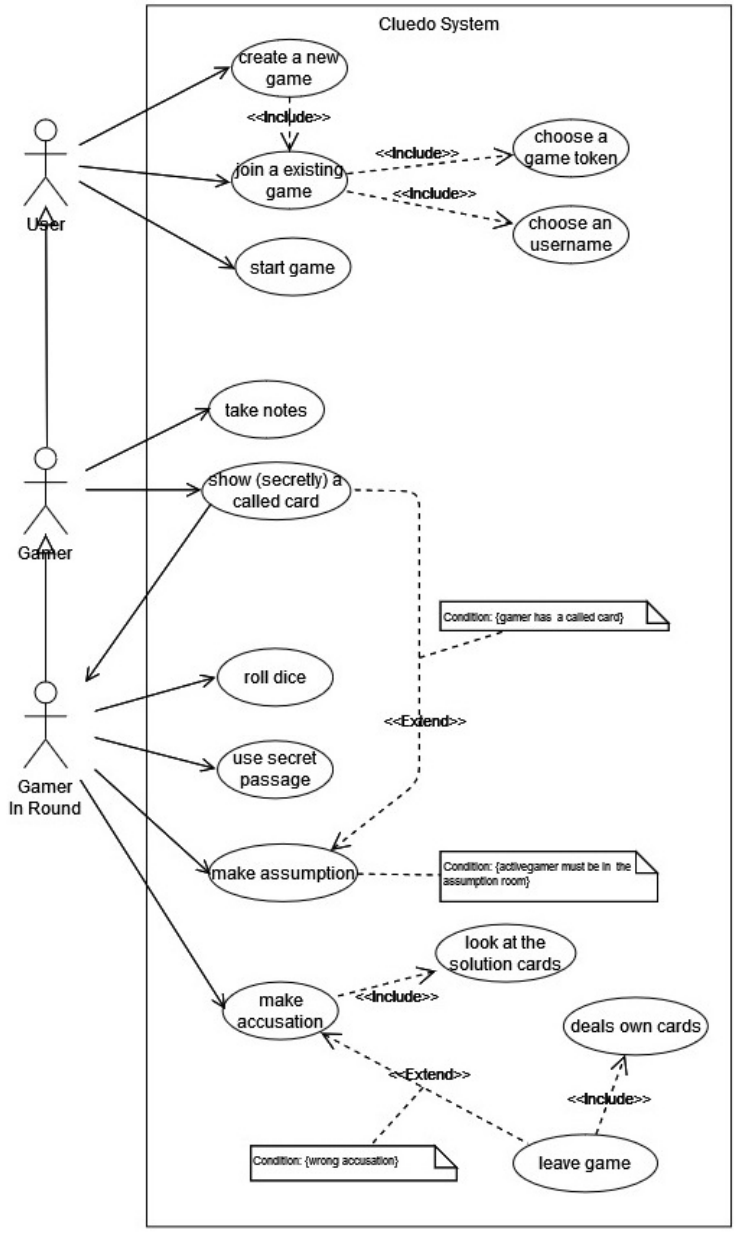


Figura 2: Diagramma dei casi d'uso delle utente

2 Analisi dei requisiti

2.1 Requisiti funzionali

User e Peer

Il sistema dovrà fornire all'utente la possibilità di effettuare alcune operazioni e quindi anche garantire alcune funzionalità ad esse collegate:

- visualizzare partite esistenti
- creare una partita
- entrare in una partita già esistente che non ancora avviata
- scegliere una pedina (character tokens) per una partita
- avviare la partita in cui si è entrati (e di conseguenza di inizializzare la partita, come mostrato nella figura 1)
- visualizzare lo stato della partita (il tabellone di gioco, i giocatori in partita, ecc..)
- tirare il dado e muovere la pedina del giocatore di turno
- fare un'ipotesi, se presente nella stanza nominata nell'ipotesi
- fare un'accusa
- muovere la pedina del personaggio ed dell'arma nella stanza nominati nell'ipotesi
- visualizzare le carte della soluzione solo per chi ha formulato un'accusa oppure nel caso ci sia rimasto un giocatore attivo
- ridistribuire le carte di un giocatore che ha effettuato un'accusa errata
- lasciare la partita dopo un'accusa errata
- usare un passaggio segreto se presente in una stanza
- mostrare una carta, nominata in un'ipotesi e se presente, solamente al giocatore che ha formulato l'ipotesi
- prendere appunti privati

Discovery server

- visualizzare i peers online
- aggiungere peer
- aggiornare lo stato dei peer memorizzati nel server
- rimuovere peer

2.2 Requisiti non funzionali

Il sistema dovrà

- garantire la comunicazione "sicura" tra i peers e i server attraverso meccanismi di autorizzazione (Role-Based Access Control) e usando il protocollo di comunicazione HTTPS (HTTP over TLS) e WSS (Web Socket over TLS)
- essere facilmente estendibile e quindi l'integrazione di nuove funzionalità dovrà risultare poco onerosa
- essere scalabile

2.3 Tecnologie e paradigmi

Per l'implementazione del sistema verrà adottata un'architettura p2p e saranno inoltre sfruttate le seguenti librerie e framework:

Node + Express

Node.js [9] è un ambiente di runtime JavaScript open source e multipiattaforma per l'esecuzione di codice JavaScript, costruita sul motore JavaScript V8 di Google Chrome. Node.js dispone di una grande quantità di moduli scritti completamente in Javascript. Essendo il progetto open source è inoltre possibile per gli sviluppatori aggiungere i propri moduli in modo da renderli disponibili pubblicamente.
Il gestore di pacchetti predefinito per Node.js si chiama *npm* [10] (Node Package Manager).
Express [5] è un framework per applicazioni Web Node.js minimale e flessibile che fornisce un robusto set di funzionalità per applicazioni Web e mobili.
Verranno utilizzate per l'implementazione dei servers e dei peers.

Socket.io

Socket.io [13] è una libreria che consente la comunicazione a bassa latenza, bidirezionale e basata su eventi tra un client e un server.
Si basa sul protocollo WebSocket e offre garanzie aggiuntive come il fallback al polling lungo HTTP o la riconnessione automatica.
WebSocket è un protocollo di comunicazione che fornisce un canale full-duplex e a bassa latenza tra il server e il browser.
Verrà utilizzata per

- la comunicazione real-time tra i peers e tra alcune comunicazioni peers e server.
- intercettare eventi come la connessione e la disconnessione dei peers
- permettere la comunicazione da un peer all'altro o al discovery server attraverso la pubblicazione di messaggi sull'eventbus

MongoDB + Mongoose

Mongoose [8] è un Object Document Mapper (ODM) che semplifica l'utilizzo di MongoDB [7] traducendo i documenti di un database MongoDB in oggetti JavaScript. Fornisce una soluzione basata su schemi per modellare i dati dell'applicazione. Include il casting di tipo integrato, convalida, creazione di query, ecc. Mongoose utilizza schemi per modellare i dati che un'applicazione desidera archiviare e manipolare in MongoDB, lo schema descrive gli attributi delle proprietà (ovvero i campi) che l'applicazione manipolerà. Tra i principali attributi troviamo:

- Data type (ad esempio String, Number, etc.)
- Descrivere se il campo è obbligatori oppure opzionale
- Descrivere se il campo è un valore univoco oppure no

Rispetto all'API nativa di MongoDB in Mongoose vengono semplificate le query. Le query MongoDB sono costituite da BSON strutturato che specifica nomi di proprietà del documento, operatori e valori, che insieme specificano come filtrare i documenti; Mongoose invece utilizza la combinazione e il concatenamento di funzioni, piuttosto che operatori per filtrare i documenti. Sebbene più lunga del suo equivalente MongoDB per le query più semplici, nelle query più complesse la versione Mongoose risulta più facile sia da costruire che da leggere.

Axios

Axios [1] è una libreria che permette di fare richieste HTTP basata sull'utilizzo delle Promise. Questo permette di scrivere codice asincrono con facilità, oltre ad evitare il problema noto della "Pyramid of Doom".
È isomorfo, ovvero può essere eseguito nel browser e su nodejs con la stessa base di codice. Sul lato server utilizza il modulo http nativo node.js, mentre sul client (browser) utilizza XMLHttpRequests.

Mocha + Chai

Mocha [6] è un framework di test JavaScript ricco di funzionalità in esecuzione su Node.js e nel browser, che rende i test asincroni semplici e divertenti. I test Mocha vengono eseguiti in serie, consentendo rapporti flessibili e accurati, mappando le eccezioni non rilevate sui casi di test corretti.
Chai [2] è una libreria di assertions BDD/TDD che consente di creare le asserzioni multiparadigma per i test.
Verranno utilizzate per testare le varie funzionalità del sistema.

Vue.js

VueJS [15] è un framework accessibile, performante e versatile per la creazione di interfacce utente web.

Il modello base di Vue prevede che ciascun componente sia composto dai seguenti blocchi logici:

- **state** rappresenta il contesto dei dati utilizzati dall'applicazione (concretizzato dall'oggetto data);
- **view** permette, tramite un mapping dichiarativo, di rappresentare lo stato all'utente tramite un'interfaccia (concretizzata tramite template HTML);
- **actions** permettono di modificare lo stato reagendo alle interazioni che gli utenti fanno tramite la view (concretizzata tramite eventi e metodi).

Swagger - OpenAPI

La specifica OpenAPI (OAS) [11] definisce un'interfaccia standard indipendente dal linguaggio per le API RESTful che consente a persone e computer di scoprire e comprendere le capacità del servizio senza accedere al codice sorgente, alla documentazione o tramite l'ispezione del traffico di rete. Se correttamente definito, un utente può comprendere e interagire con il servizio remoto con una quantità minima di logica di implementazione. Una definizione OpenAPI può quindi essere utilizzata da strumenti di generazione della documentazione per visualizzare l'API, strumenti di generazione di codice per generare server e client in vari linguaggi di programmazione, strumenti di test e molti altri casi d'uso.

Docker

Docker [4] è un progetto open-source che automatizza il processo di deployment di applicazioni all'interno di contenitori software (*container*), fornendo un'astrazione aggiuntiva grazie alla virtualizzazione a livello di sistema operativo di Linux.
I container sono unità software che permettono di isolare una singola applicazione e il proprio ambiente, facilitandone così l'installazione sulle macchine dotate di Container Engine.
I concetti fondamentali su cui si basa Docker sono:

- **Dockerfile**: file di configurazione che descrive tutti i comandi necessari per assemblare una nuova immagine.
- **Docker Images**: file eseguibile che rappresenta la descrizione statica di un container, l'esecuzione dell'immagine determina l'istanziazione del container stesso.
- **Docker Container**: macchina virtuale "leggera" che condivide con la macchina host il kernel garantendo però l'isolamento sia dall'ambiente di esecuzione che dagli altri container.

3 Design

L'architettura complessiva del sistema segue un approccio decentralizzato rispetto a un approccio centralizzato. Come mostrato nella figura 3, un approccio centralizzato in un sistema software distribuito comporta che tutta l'elaborazione e la gestione dei dati vengano eseguite da un singolo server, mentre un approccio decentralizzato richiede la distribuzione di queste attività su più nodi della rete. I vantaggi di un approccio centralizzato includono una gestione e una manutenzione più semplici, nonché una comunicazione più rapida ed efficiente grazie alla presenza di un unico punto di contatto. Tuttavia, l'approccio centralizzato è vulnerabile al *Single Point of Failure* e la capacità di scalabilità è limitata dal server centrale. I vantaggi di un approccio decentralizzato includono una maggiore disponibilità grazie all'assenza di *Single Point of Failure* e una maggiore scalabilità attraverso l'aggiunta di nuovi nodi. Tuttavia, i sistemi decentralizzati sono più complessi da gestire e mantenere e la comunicazione può essere più lenta a causa di più punti di contatto.

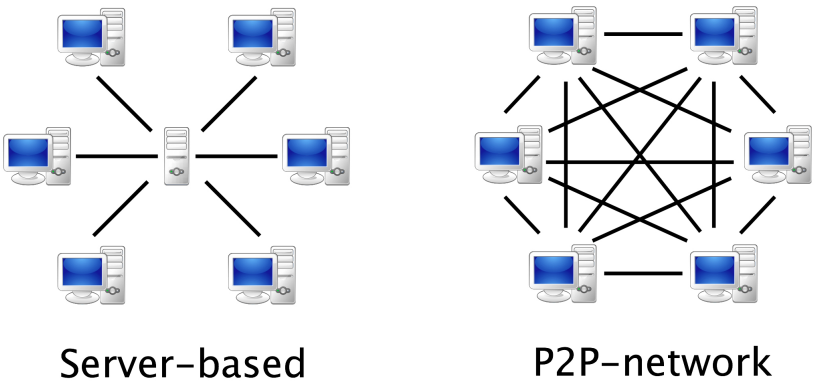
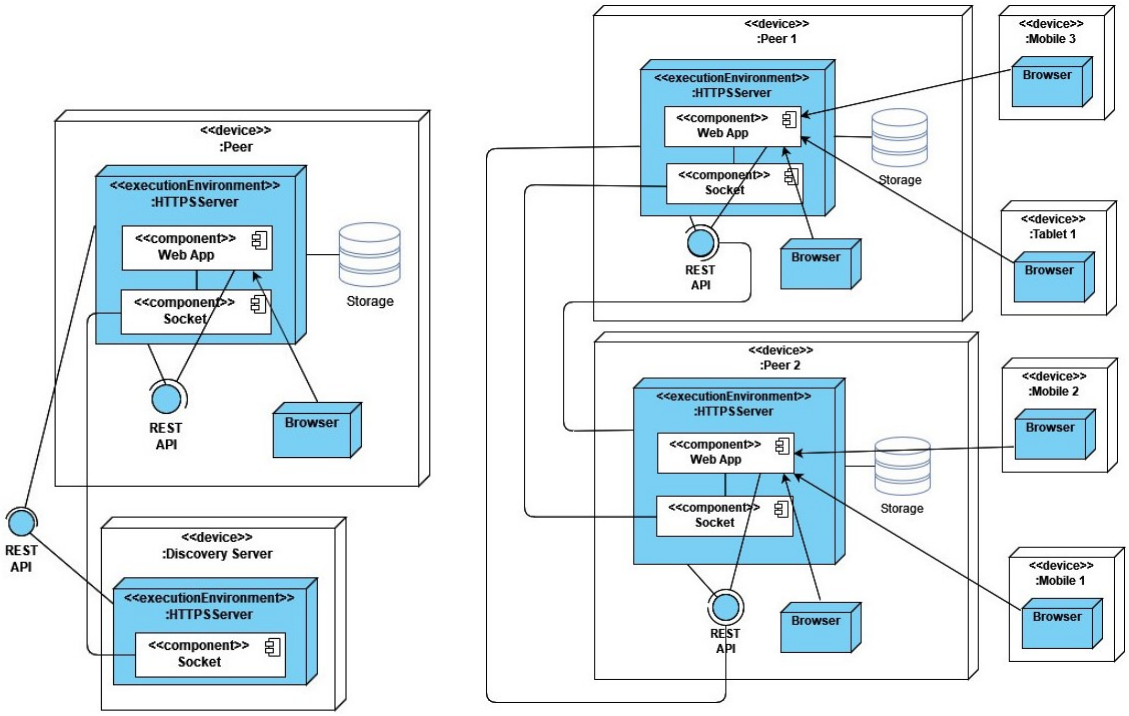


Figura 3: Architettura ad alto livello a confronto

3.1 Struttura

Si è scelto di realizzare un'architettura Peer-to-Peer con Discovery Server. Questo tipo di architettura prevede un server centrale chiamato, appunto, *Discovery*, che consente ai Peer appena avviati di conoscere gli indirizzi di altri Peer.
Come mostrato in figura 4, sia il Discovery Server che il Peer espongono sia RESTful APIs che Socket, le quali che permetto la comunicazione tra i vari componenti, come ad esempio la comunicazione tra il Discovery Server e un Peer, oppure tra un Peer e un altro Peer, oppure tra il lato server del Peer (HTTPS Server) e il lato client dello stesso (Web App).



(a) Discovery Server e Peer (b) Peers

Figura 4: Diagramma di Deployment.

Come mostrato nella figura 4b, la struttura del Peer è stata progettata in modo tale che anche se non ci sono altri peer connessi, l'applicativo funzioni come un'architettura Client-Server con lo stesso Peer come server centrale.

A differenza dello Discovery Server, che ha una natura dinamica nel sistema, il lato server del Peer ha anche la funzione di Backup server per le partite di altri peer, e quindi è necessario utilizzate uno storage. L'idea iniziale era quella di creare un ulteriore server che fungeva da Backup Server, ma questa idea è stata scartata perché alla fine la comunicazione dei vari updates tra le partite devono passare comunque attraverso tutti i Peer connessi.

Di seguito viene illustrato il diagramma (figura 5) dell'entità principali dell'applicativo.

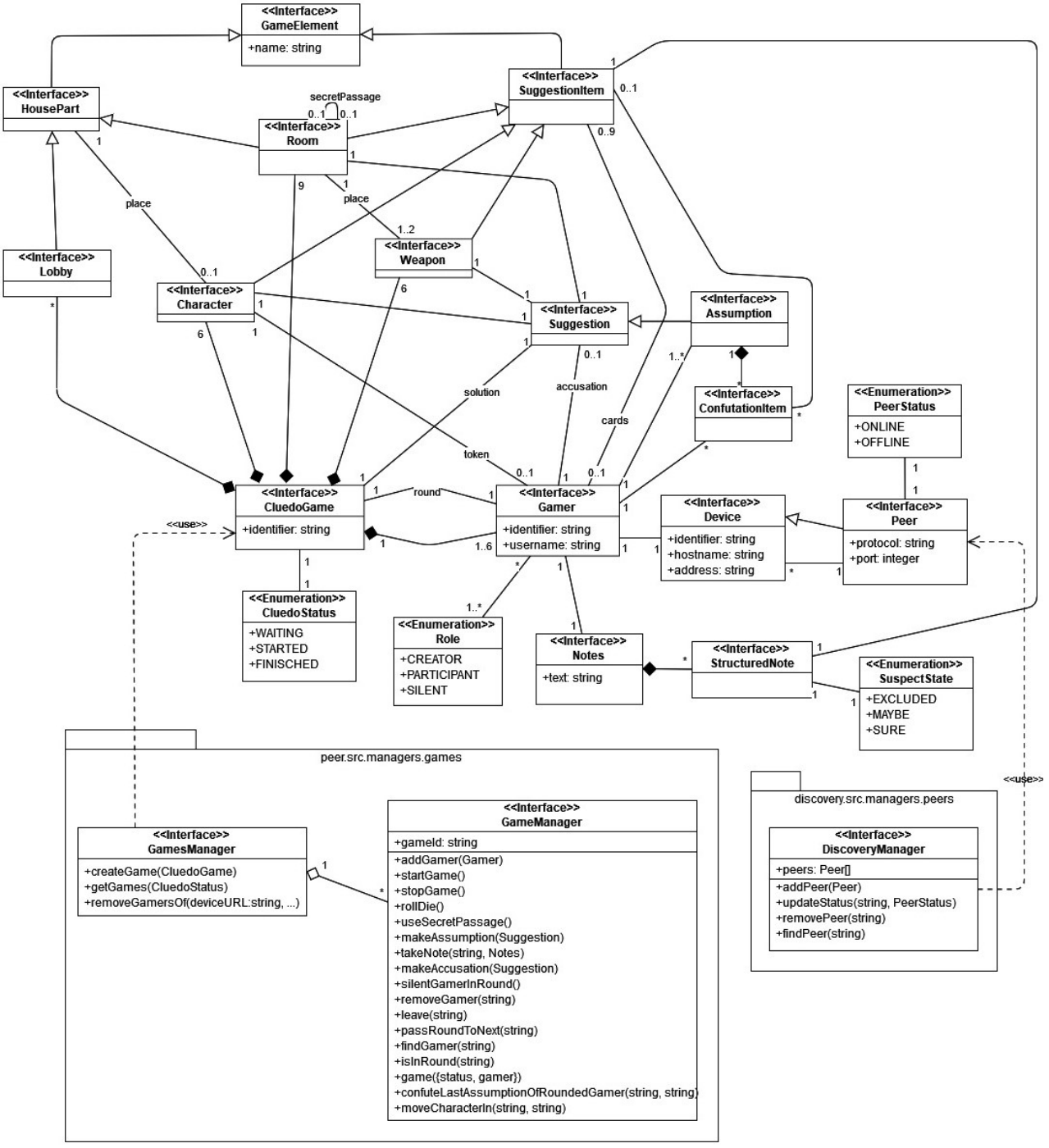


Figura 5: Diagramma delle classi del gioco Cluedo

L'entità principale del sistema è rappresentata dall'interfaccia *CluedoGame*, la quale descrive una partita di Cluedo.
Essa si compone di

- una lista di *Gamer*, ossia gli utenti che partecipano alla partita. Essa può contenere da 1 a 6 giocatori;
- una lista di 6 *Character*, ossia i personaggi che i giocatori "interpretano" durante la partita;
- una lista di 9 *Room*, ossia le stanze in cui il giocatore può muoversi durante la partita. Una stanza potrebbe essere collegata ad un'altra stanza attraverso un passaggio segreto rappresentato dall'associazione riflessiva *secretPassage*;
- una lista di N *Lobby*, ossia "parti della casa" in cui giocatori saltano il proprio turno durante la partita;
- una lista di 6 *Weapon*, ossia le possibili armi del delitto. Esse sono collocante una in ogni stanza;
- una soluzione del gioco (ossia del delitto) rappresentata dall'interfaccia *Suggestion*. Essa è caratterizzata da un sospettato (*Character*), un'arma (*Weapon)* e una stanza (*Room*).

Character, *Weapon*, *Room* e *Lobby* sono caratterizzato da un nome (*name*) che rappresenta l'identificatore, il quale è ereditato direttamente o indirettamente dall'interfaccia *GameElement*.

Per quanto riguarda i giocatori, ogni *Gamer* è caratterizzato da un identificatore e un username. Nell'ambito della partita il *Gamer* è caratterizzato anche da

- un *Character* che rappresenta la pedina con cui è identificato nella partita;
- un *Role* che rappresenta il ruolo del giocatore in una partita. I possibili valori sono:
  - CREATOR: colui che crea la partita
  - PARTICIPANT: colui che si unisce a un partita
  - SILENT: colui che, avendo effettuato un'accusa sbagliata, decide di rimanere. L'unica azione che potrà effettuare è la confutazione di un'ipotesi di un altro giocatore.
- un *Device*. In una prima progettazione rappresentava il dispositivo con cui il giocatore si connette al Peer. Alla fine si è deciso che rappresentasse il *Peer* a cui si è connesso.
  Un *Peer* è caratterizzato da
  - un identificatore (*identifier*)
  - un hostname

  - un indirizzo ip (*address*)
  - una porta (*port*)
  - un protocollo di comunicazione (*protocol*)
  - uno *PeerStatus*, che rappresenta lo stato corrente del peer. I possibili valori sono:
    * ONLINE: un determinato Peer è raggiungibile anche da altri Peer.
    * OFFLINE: un determinato Peer non è raggiungibile da nessun client, peer e server.

- una lista di ipotesi (*Assumption*) e un'accusa (*Suggestion*). Un'ipotesi è un' estensione di un'accusa, con in più la lista dei giocatori che hanno effettivamente confutato la stessa. Questo concetto è rappresentato dall'interfaccia *ConfutationItem*, la quale tiene anche il riferimento a quale "parte" dell'ipotesi il giocatore a confutato;

- una lista di *SuggestionItem* che rappresentano le "carte" che a inizio partita vengono distribuite al giocatore e sono raffiguranti i componenti chiave del gioco, ossia *Character*, *Room*, *Weapon*;

- un taccuino, rappresentato attraverso l'interfaccia *Notes*. Questa interfaccia è caratterizzata da
  - un campo *text*, che rappresenta le note "libere" che il giocatore "scrive" durante la partita;
  - una lista di tutti i componenti chiave del gioco (*Character*, *Room*, *Weapon*) con indicato un valore (*SuspectState*), che rappresenta uno dei seguenti termini
    * EXCLUDED: sicuramente il componente non è parte della soluzione o perché il giocatore ha in "mano" la carta relativa oppure perché è stata confutata da un altro giocatore.
    * MAYBE: forse il componente può essere parte della soluzione
    * SURE: sicuramente il componente è parte della soluzione

Per quanto riguarda i *Managers* sono delle interfacce che aiutano nella creazione, nell'aggiornamento e nella rimozione delle istanze dell'interfacce viste in precedenza. Come si vedrà nella sotto sezione Interazione, il *DiscoveryManager* si occupa di tenere in memoria lo stato di tutti i peer connessi, mentre *GamesManager* e *GameManager* si occupano rispettivamente di tenere in memoria (nello storage) le partire e lo stato di gioco di una singola partita.

3.2 Comportamento

Come mostrato nella figura 6, quando un Peer viene avviato va in uno stato *Idle* ossia in uno stato in cui è possibile già utilizzare l'applicativo come architettura client-server. Dal momento che viene connesso al Discovery Server, il Peer va in uno stato *Online*, ossia in uno stato in cui è possibile raggiungere il Peer tramite altri Peer. Nel caso di errori di connessioni o di chiamata esplicita di disconnessione, il Peer dallo stato *Online* torna nello stato *Idle*. Nel caso il Peer dovesse terminare, lo stato che il Discovery Server "riceve" è lo stato *Offline*, ossia uno stato in cui non è possibile raggiungere il Peer tramite la rete.

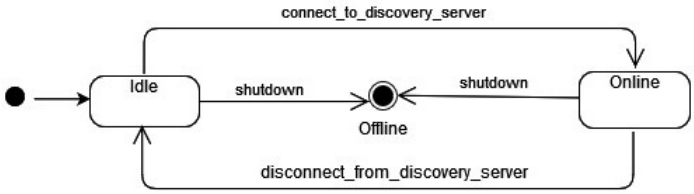


Figura 6: Diagramma di stato del Peer

Di seguito è illustrato il diagramma degli stati (figura 7) per quanto riguarda il comportamento dell'applicativo (e di una partita) dal punto di vista dell'utente.

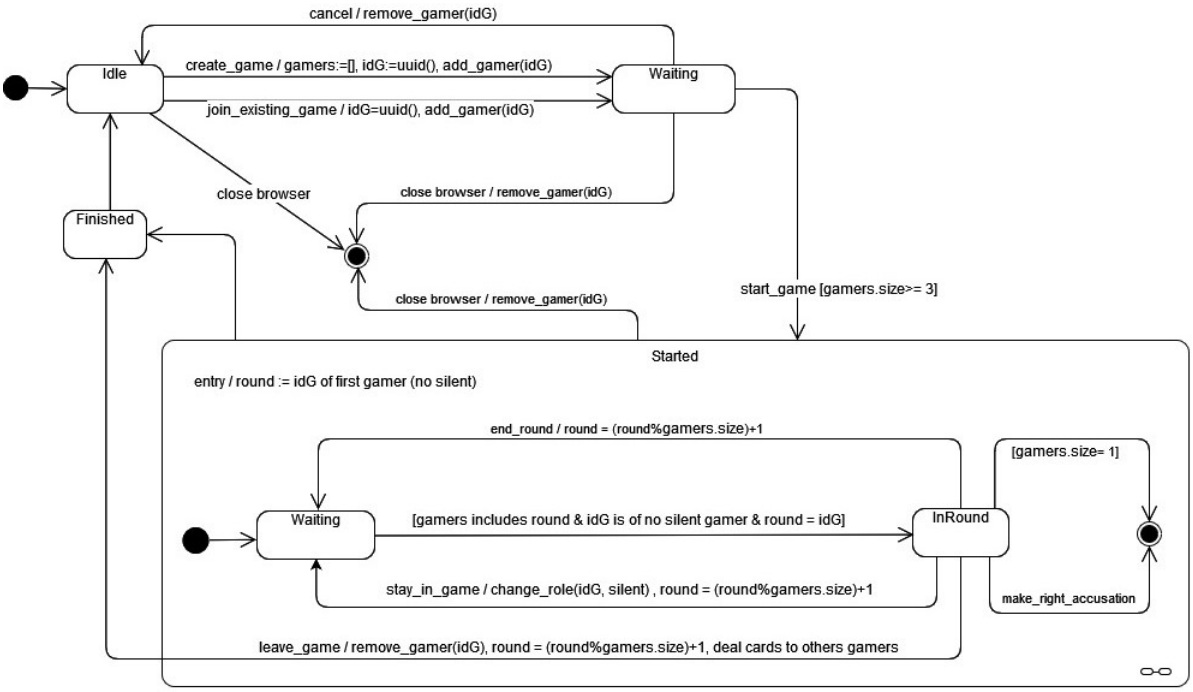


Figura 7: Diagramma di stato dell'utente

Quando l'utente avvia l'applicativo (ossia il front-end del Peer) tramite il browser, l'applicativo si trova in uno stato *Idle*, stato in cui l'utente può interagire con esso. Quando l'utente crea una nuova partita oppure si unisce a una già esistente, l'applicativo passa dallo stato *Idle* a uno stato di *Waiting*, ossia uno stato in cui l'utente deve aspettare l'entrata di altri utenti. Dallo stato *Waiting* si può ripassare allo stato *Idle* nel caso in cui l'utente decida di uscire dalla partita (non ancora avviata).

Invece quando il numero minimo dei giocatori (ossia 3) è stato raggiunto, un giocatore può decidere di avviare la partita e così facendo si passa a uno stato *Started*.
All'entrata dello stato *Started*, viene settato il giocatore che deve iniziare la partita ed ogni giocatore va in un sotto-stato *Waiting*, stato in cui si deve aspettare il proprio turno oppure si aspetta che il gioco finisca; solo il giocatore con identificatore uguale *round* (ossia il giocatore in turno) può passare nel sotto-stato *InRound*, ossia stato in cui il giocatore può inviare a giocare; di seguito viene illustrato tramite un diagramma di attività (figura 8).

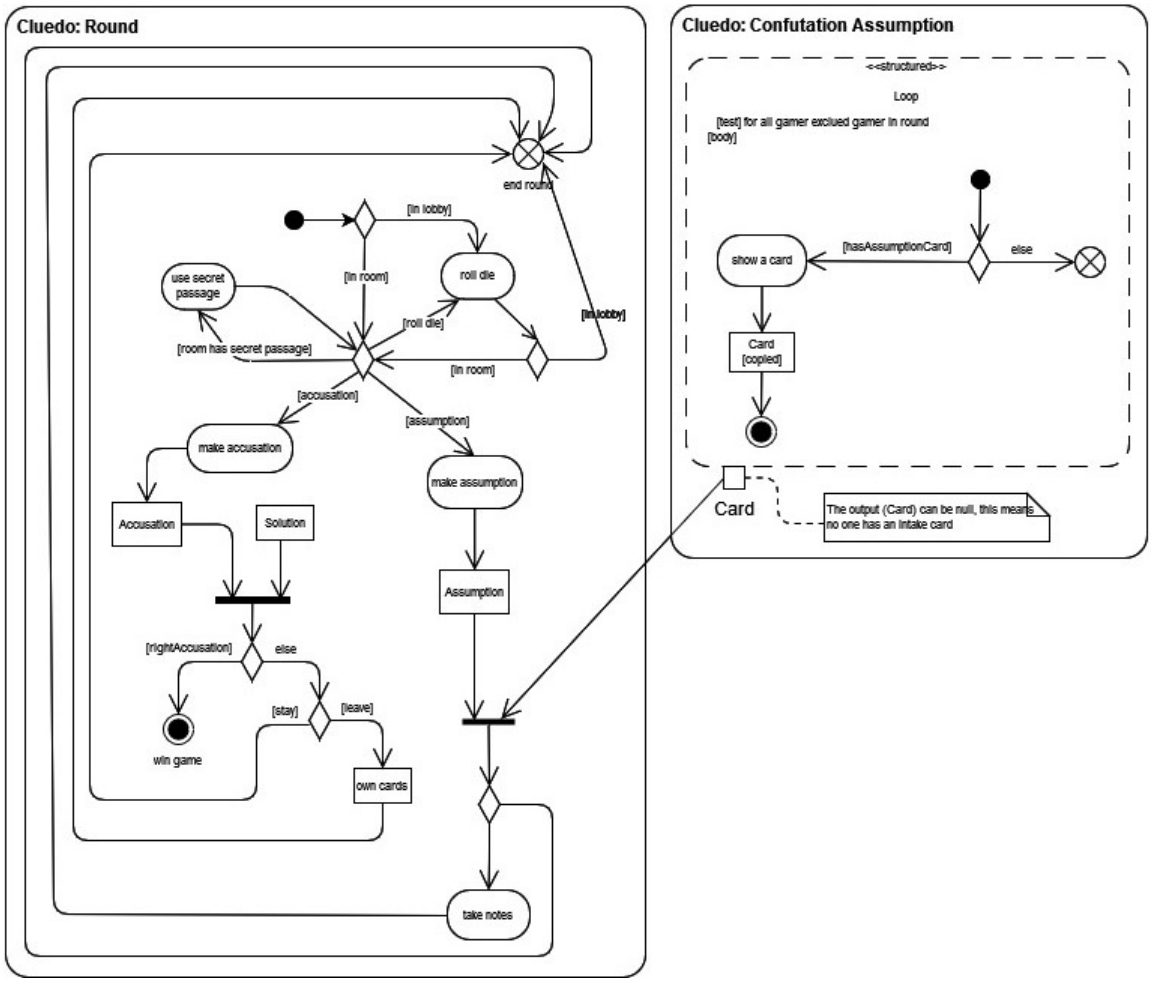


Figura 8: Diagramma di attività di una partita

Il giocatore in turno può effettuare diverse attività:

- **roll die**: questa attività deve essere svolta sicuramente la prima volta che il giocatore giocherà il suo turno, perché egli si trova in una lobby. Successivamente può effettuare questa attività quante volte vuole a meno che il risultato non sia una *lobby*, in tal caso il flusso del giocatore in turno termina (*end round* in figura 8) e lo stato passa da *InRound* a *Waiting* attraverso l'evento *end_round* (figura 7), che permette il passaggio del turno all'utente successivo;

- **make assumption**: questa attività permette di effettuare un'ipotesi sul delitto. Successivamente, l'utente deve attendere che un altro giocatore gli "mostri una carta". Dopo che ha ricevuto almeno una "carta" oppure ha avuto conferma che nessun altro giocatore abbia una "carta" relativa all'ipotesi, il giocatore può "prendere appunti" (attività **take notes**). Successivamente, il flusso del giocatore in turno termina come per il caso dell'attività di *roll die*;

- **make accusation**: questa attività permette di effettuare un'accusa sul delitto.
  Se l'accusa risulta corretta (ossia il giocatore ha vinto la partita), il flusso del gioco termina con l'evento *make_right_accusation* e l'applicativo, per tutti i giocatori, passa allo stato *Finished* (stato transitorio) e subito dopo passa allo stato *Idle*.
  Nel caso di un'accusa scorretta (ossia il giocatore ha perso la partita), il giocatore deve decidere se restare (*stay*) oppure abbandonare (*leave*) il gioco.
  Nel caso decidesse di abbandonare (evento *leave_game*, figura 7), il passaggio dallo stato *InRound* allo stato *Finished* (per il solo giocatore uscente) permette la ridistribuzione delle carte agli altri utenti, la rimozione dell'utente uscente e il passaggio del turno all'utente successivo. Nel caso decidesse di restare (evento *stay_in_game*, figura 7), il passaggio dallo stato *InRound* allo stato *Waiting*, permette il cambio di ruolo del giocatore uscente da *participant* in *silent* e il passaggio del turno all'utente successivo.

- **use secret passage**: questa attività può essere eseguita solo nel caso il giocatore di turno sia in una stanza che abbia un passaggio segreto.

L'utente può chiudere l'applicativo (ossia chiudere il browser) e quindi terminare definitivamente il flusso dell'applicativo; questo comporta l'azione di rimozione del giocatore nel caso si dovesse trovasse nello stato *Waiting* o nello stato *Started*.

3.3 Interazione

In questa sotto sezione viene descritto come interagiscono i diversi componenti dell'architettura durante le diverse fasi di gioco, fornendo ogni scenario con il corrispondente diagramma di sequenza ed eventuale mockup dell'interfaccia grafica.

Avvio del Peer (back end)

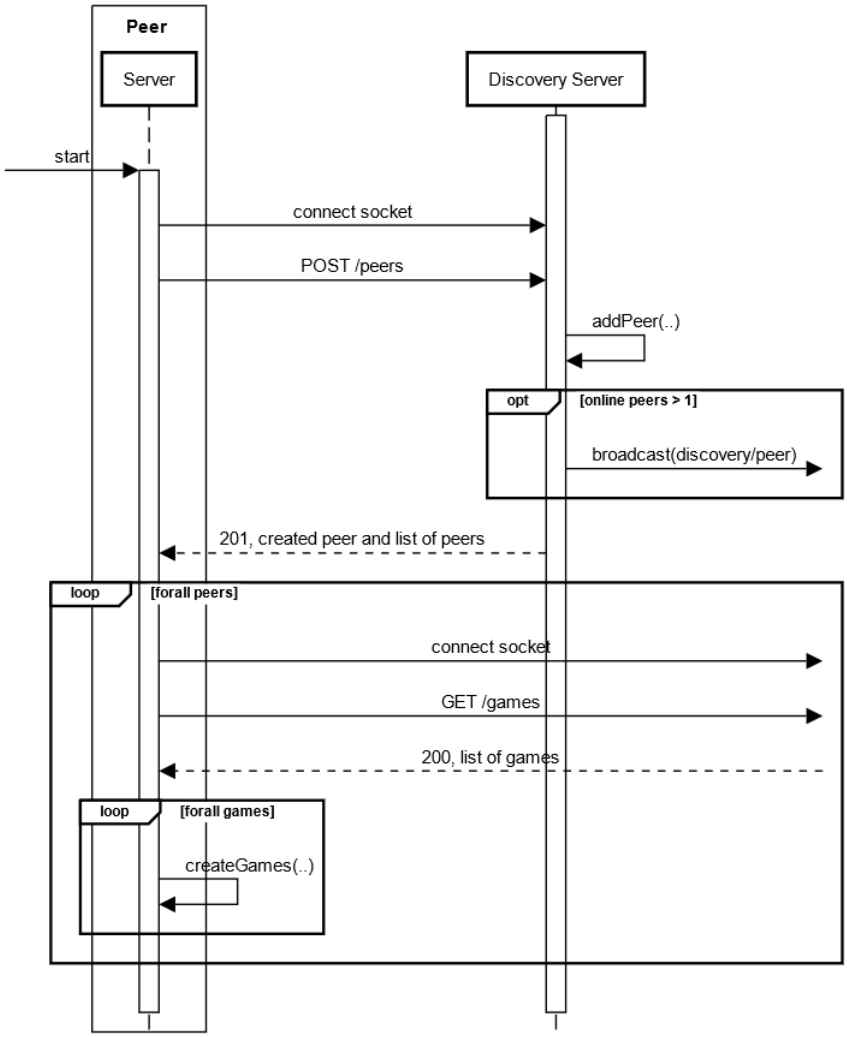


Figura 9: Diagramma di sequenza: Avvio di un peer

Come mostrato in figura 9, appena avviato il Peer, si connette al Discovery Server attraverso le socket, e successivamente invia in POST una richiesta PEERS. Questa richiesta permette al Discovery Server di registrare in memoria le informazione del Peer attraverso *addPeer* (un metodo del *DiscoveryManager*); inoltre vengono inviate (le infor-

mazioni relative al Peer appena connesso) anche a tutti i possibili Peer connessi attraverso l'evento socket *discovery/peer*.

Alla ricezione della lista di tutti i peer connessi, per ognuno viene effettuata la connessione socket e inviata in GET una richiesta GAMES. Questa richiesta permette di "prelevare" ogni partita creata, così d'avere una copia locale. In fine, ogni partita viene memorizzata nel Peer attraverso *createGames* (un metodo del *GamesManager*).

Avvio del Peer (front end)

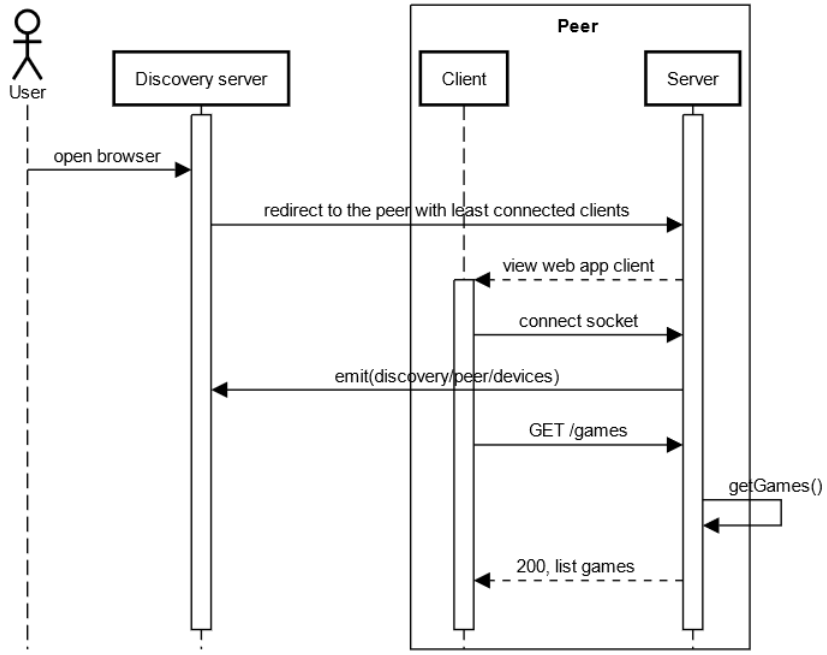


Figura 10: Diagramma di sequenza: Avvio applicazione di un utente

Come mostrato in figura 10, quando l'utente apre l'applicazione (ossia apre il browser) attraverso l'indirizzo del Discovery Server, viene reindirizzato al l'indirizzo del Peer con meno clients (ossia clients veri e propri + peers) connessi. Successivamente il Peer mostra all'utente l'interfaccia con cui interagire. Prima di poter interagire il Peer (Client) apre una connessione socket con il Peer su cui è hostato. Il Peer invia al Discovery Server il numero aggiornato di dispositivi connessi attraverso l'evento socket *discovery/peer/devices*.

In fine, viene inviata in GET una richiesta GAMES al Peer (Server), la quale permette di recuperare tutte le partite attraverso *getGames* (un metodo del *GamesManager*).

Figura 11: Mockup: HomePage

Creazione di una nuova partita

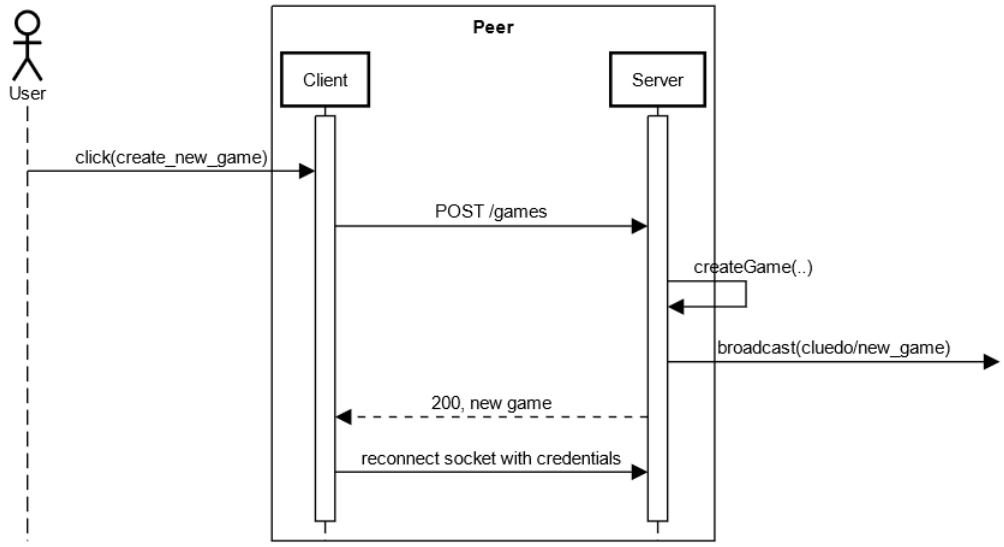


Figura 12: Diagramma di sequenza: Creazione di una nuova partita

Come mostrato in figura 12, quando l'utente decide di creare una nuova partita, il

Peer (Client) invia in POST una richiesta GAMES al Peer (Server). La richiesta permette la memorizzazione della nuova partita attraverso il metodo del *GamesManager*, *createGame* e l'invio a tutti i clients connessi (escluso il chiamante) attraverso l'evento socket *cluedo/new_game*.

Successivamente alla ricezione della risposta alla richiesta, il Peer (Client) effettua una riconnessone socket aggiornando le credenziali del giocatore ricevute.

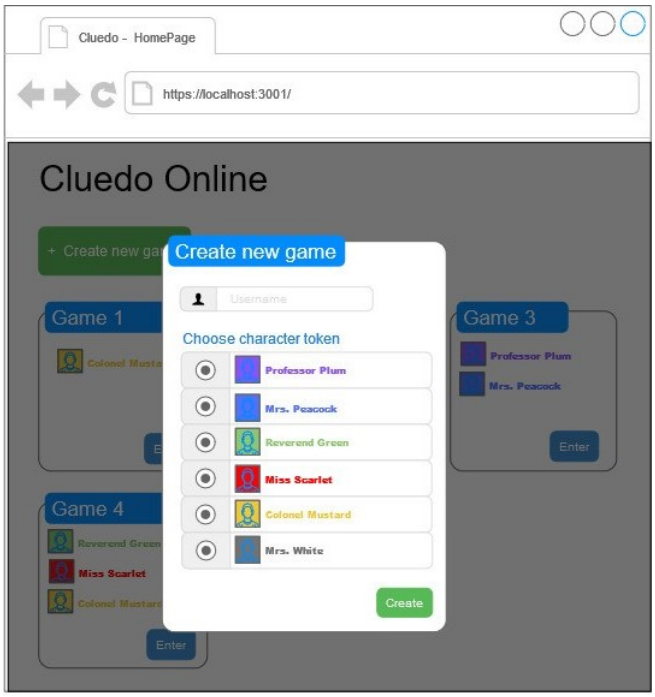


Figura 13: Mockup: Creazione di una nuova partita

Unione di un giocatore ad una partita

Come mostrato in figura 14, quando l'utente decide di unirsi a una partita, il Peer (Client) invia in POST una richiesta GAMES/{gameId}/GAMERS al Peer (Server). La richiesta permette l'aggiunta del nuovo giocatore nella partita scelta attraverso il metodo del *GameManager*, *addGamer* ed l'invio a tutti i clients connessi (escluso il chiamante) attraverso l'evento socket *cluedo/new_gamer*.

Successivamente alla ricezione della risposta alla richiesta, il Peer (Client) effettua una riconnessone socket aggiornando le credenziali del giocatore ricevute.

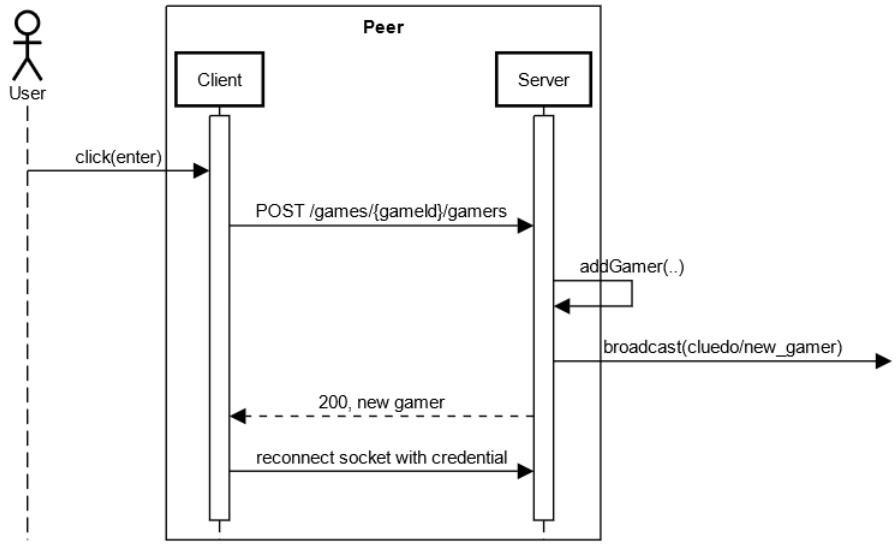


Figura 14: Diagramma di sequenza: Unione in una partita

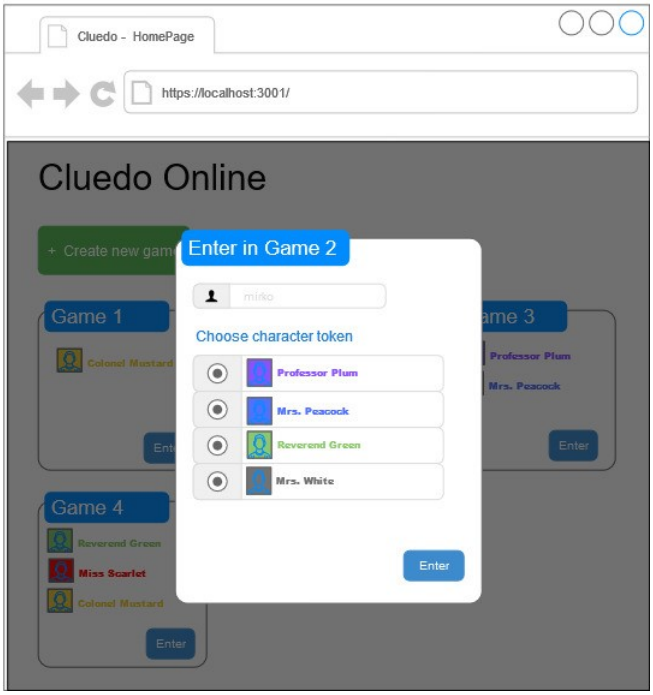


Figura 15: Mockup: Unirsi a una partita

Uscita di un giocatore da una partita (non avviata)

Come mostrato in figura 16, l'utente può decide di uscire da una partita (non ancora avviata), in tal caso il Peer (Client) invia in DELETE una richiesta

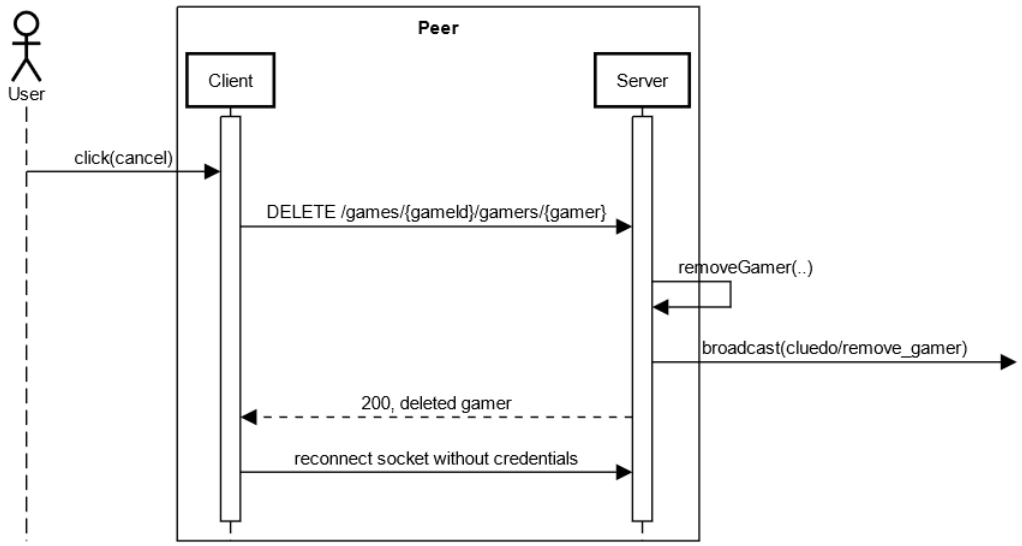


Figura 16: Diagramma di sequenza: Uscita da una partita (non avviata)

GAMES/{gameId}/GAMERS/{gamer} al Peer (Server). La richiesta permette la rimozione del giocatore attraverso il metodo del *GameManager*, *removeGamer* e l'invio a tutti i clients connessi (escluso il chiamante) attraverso l'evento socket *cluedo/remove_gamer*.

Successivamente alla ricezione della risposta alla richiesta, il Peer (Client) effettua una riconnessone socket rimuovendo le credenziali del giocatore appena rimosso.

Figura 17: Mockup: Uscire da una partita (non avviata)

Ricezioni di creazione di una partita o unione o uscita da partita

Come mostrato in figura 18, alla ricezione degli eventi *cluedo/new_game*, *cluedo/new_game* o *cluedo/remove_gamer*, il Peer (Server) chiama i relativi metodi *createGame*, *addGamer* e *removeGamer* ed invia ai clients veri e propri (ossia i Peer (Client) connessi) i vari messaggi ricevuti attraverso gli stessi eventi socket ricevuti.

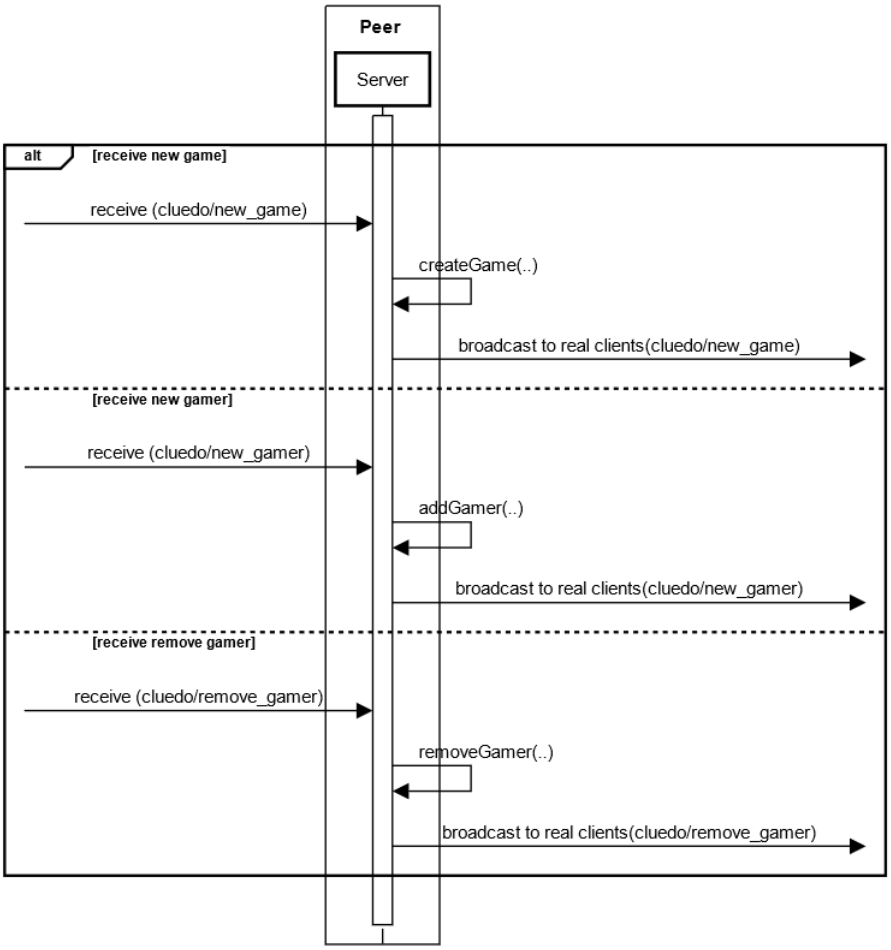


Figura 18: Diagramma di sequenza: Ricezioni di creazione di una partita, unione o uscita da partita

Inizio di una partita

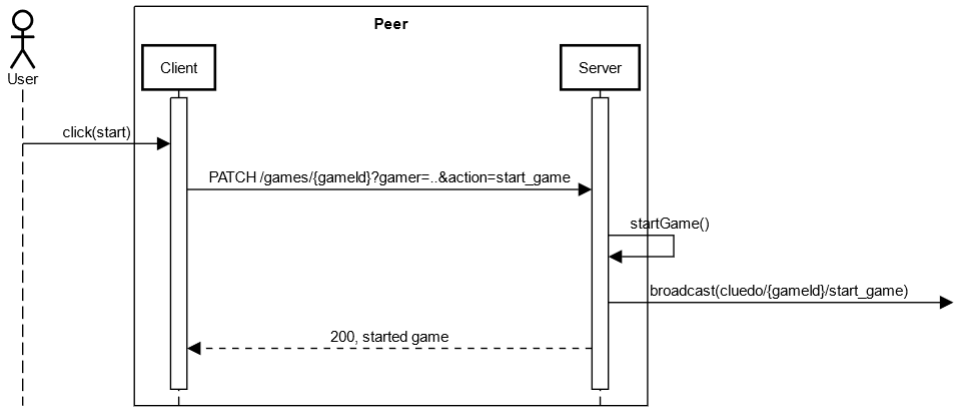


Figura 19: Diagramma di sequenza: Inizio di una partita

Come mostrato in figura 19, quando un giocatore decide di iniziare una partita, il Peer (Client) invia in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che decide di iniziare la partita* e *action = start_game* al Peer (Server). La richiesta permette l'inizializzazione della partita (ossia il settaggio nella soluzione, della distribuzione delle "carte" ecc.) attraverso il metodo del *GameManager*, *startGame* ed l'invio a tutti i giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/start_game*. Ovviamente anche i clients che non fanno parte della partita devono ricevere un feedback, questo avviene sempre attraverso l'evento socket *cluedo/{gameId}/start_game*, ma con un messaggio leggermente diverso ossia senza l'inizializzazione della partita.

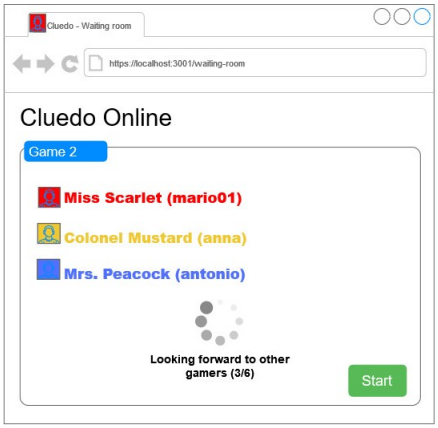


Figura 20: Mockup: Iniziare una partita

Di seguito è mostrato il mockup della game board:

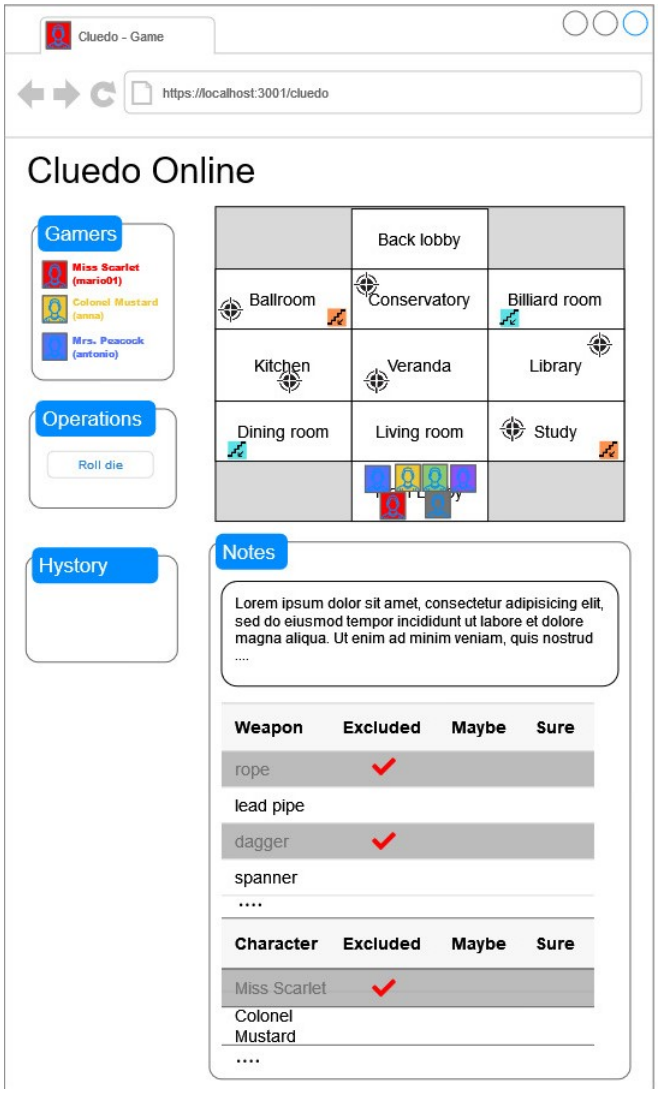


Figura 21: Mockup: Game Board

Azioni in partita

In questa sotto sezione, vengono descritte le varie interazioni tra i vari componenti durante un partita a *Cluedo* ossia gli "eventi" che i giocatori scatenano durante la partita.

Essendo *Cluedo* un gioco a turni, dopo un certo tipo di azioni il giocatore deve passare il turno al successivo; questo avviene attraverso l'invio in PATCH di una richiesta GAMES/{gamerId} con query *gamer=identificatore del giocatore che passa il turno* e *action=end_round*. (figura 22).
Quando il Peer (Server) riceve la richiesta, "passa" il turno al giocatore successivo (con ruolo "*participant*") attraverso il metodo *passRoundToNext* del *GameManager*. Nel caso ci sia rimasto un solo giocatore con il ruolo "*participant*", viene inviato a tutti gli altri giocatori connessi (anche in diversi Peer) la soluzione della partita (figura 23), altrimenti viene inviato l'identificatore del giocatore successivo attraverso l'evento socket *cluedo/{gameId}/end_round*.

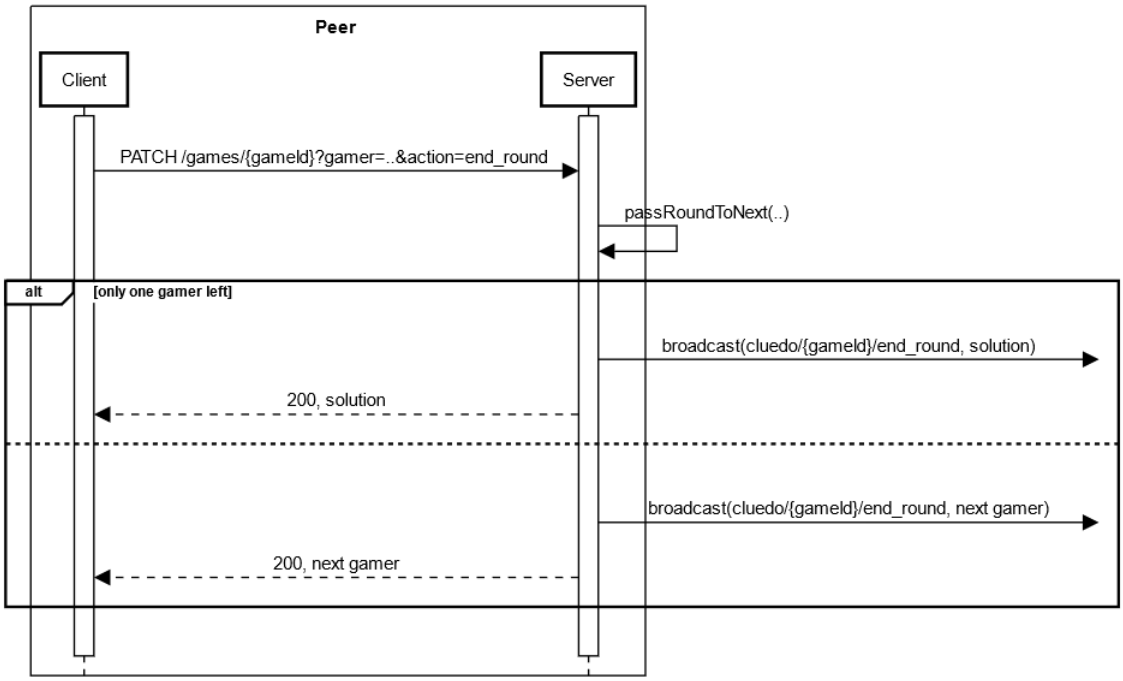


Figura 22: Diagramma di sequenza: Passare il turno

Questa richiesta può provocare la fine di una partita senza un vincitore, come mostrato nel mockup in figura 23.

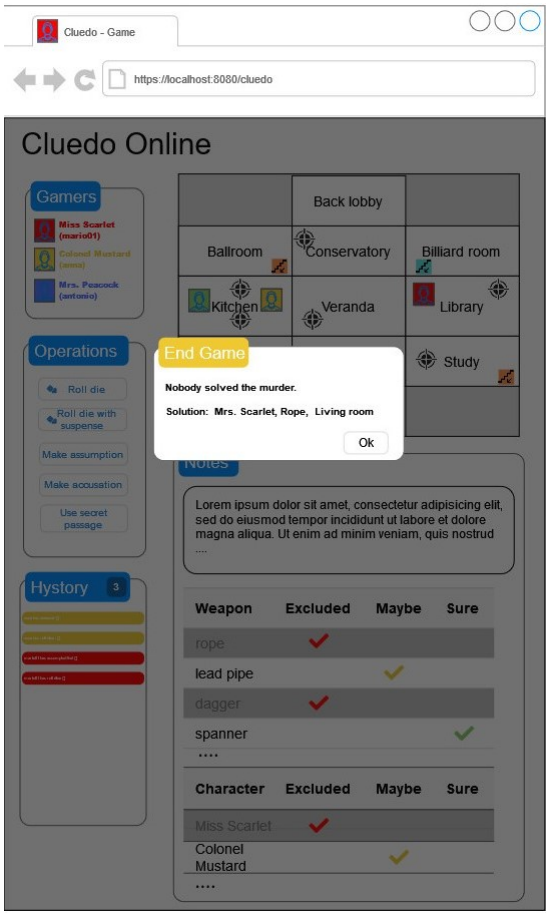


Figura 23: Mockup: Nessuno vincitore nella partita

Tirare il dado

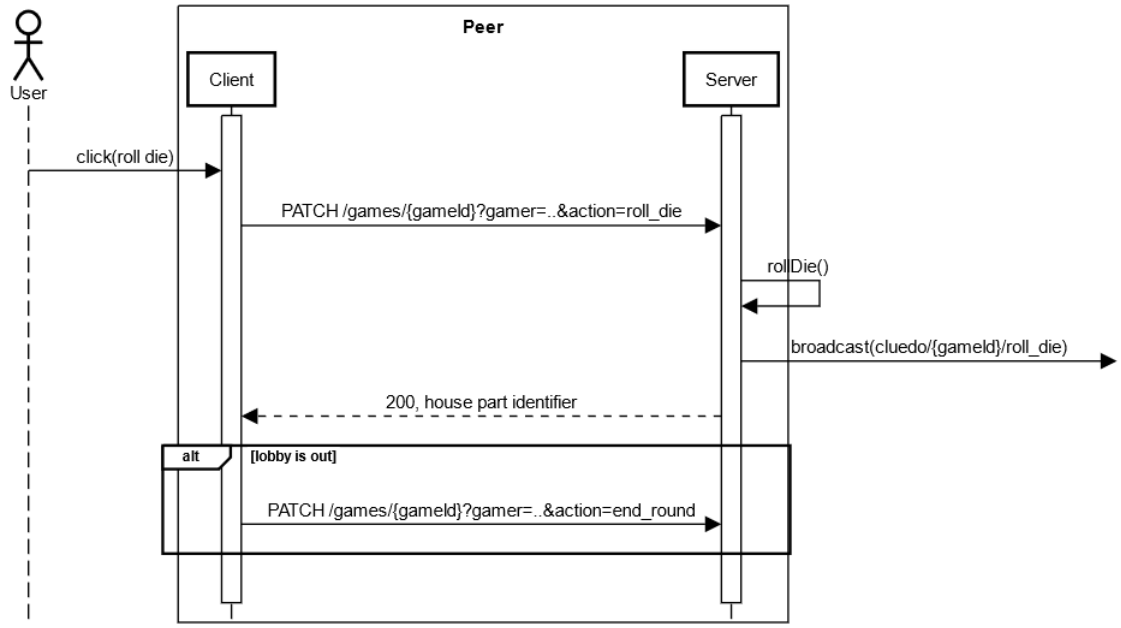


Figura 24: Diagramma di sequenza: Tirare il dado

Come mostrato nella figura 24, quando un giocatore "tira il dado", viene inviata in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che tira il dado* e *action = roll_die*. La richiesta permette di generare casualmente e memorizzare la "parte della casa" in cui il giocatore deve posizionarsi attraverso il metodo *rollDie* del *GameManager* e successivamente inviarla a tutti gli altri giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/roll_die*. Nel caso la "parte della casa" fosse una "lobby" viene inviata una richiesta di fine turno (vista in precedenza).

Usare un passaggio segreto

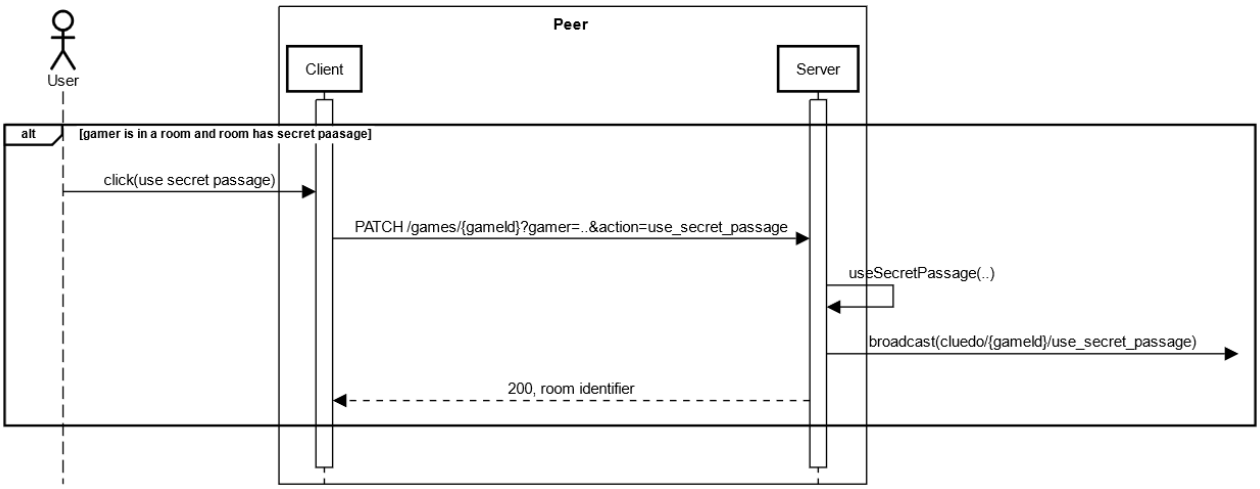


Figura 25: Diagramma di sequenza: Usare un passaggio segreto

Come mostrato nella figura 25, quando un giocatore è in una stanza dove è presente un passaggio segreto per un'altra stanza e il giocare l'utilizza, viene inviata in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che usa il passaggio segreto* e *action = use_secret_passage*. La richiesta permette di spostare il giocatore nella stanza relativa al passaggio segreto attraverso il metodo *useSecretPassage* del *GameManager* e successivamente di inviare le modifiche a tutti gli altri giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/use_secret_passage*.

Fare e confutare un'ipotesi

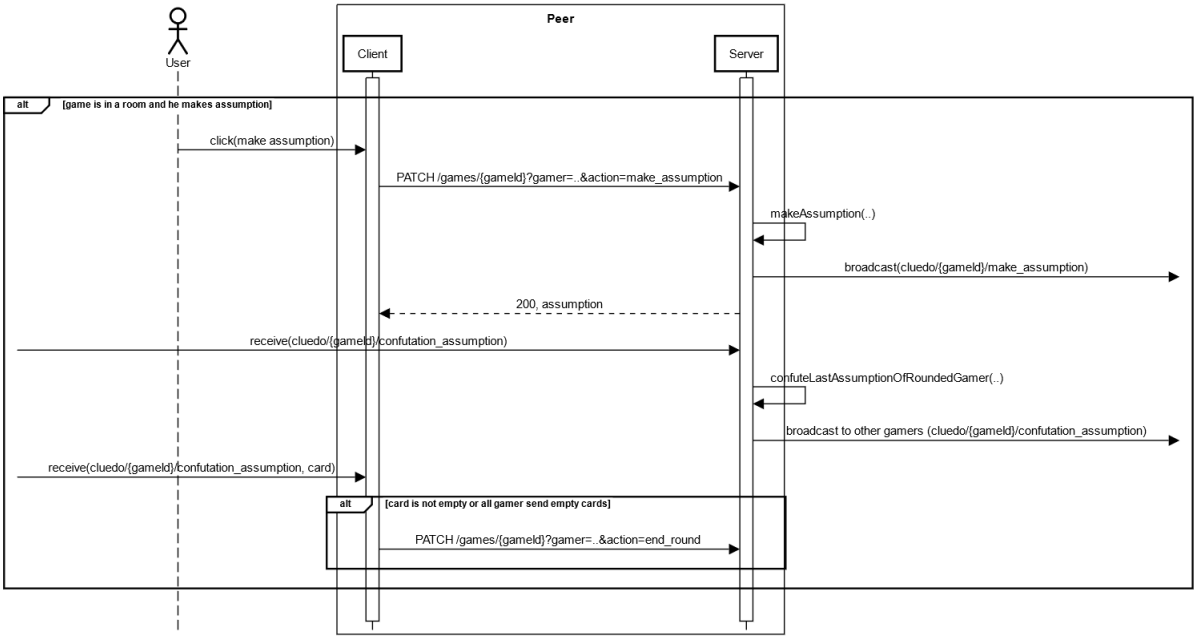


Figura 26: Diagramma di sequenza: Fare un'ipotesi

Come mostrato nella figura 26, quando un giocatore vuole effettuare un'ipotesi, il Peer (Client) invia in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che effettua l'ipotesi* e *action = make_assumption*. La richiesta permette di memorizzare l'ipotesi effettuata dal giocatore di turno sul Peer (Server) attraverso il metodo *makeAssumption* del *GameManager* e successivamente inviarla a tutti gli altri giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/make_assumption*. Dopo di ché, il giocatore deve attendere che l'ipotesi venga confutata dagli altri giocatori.

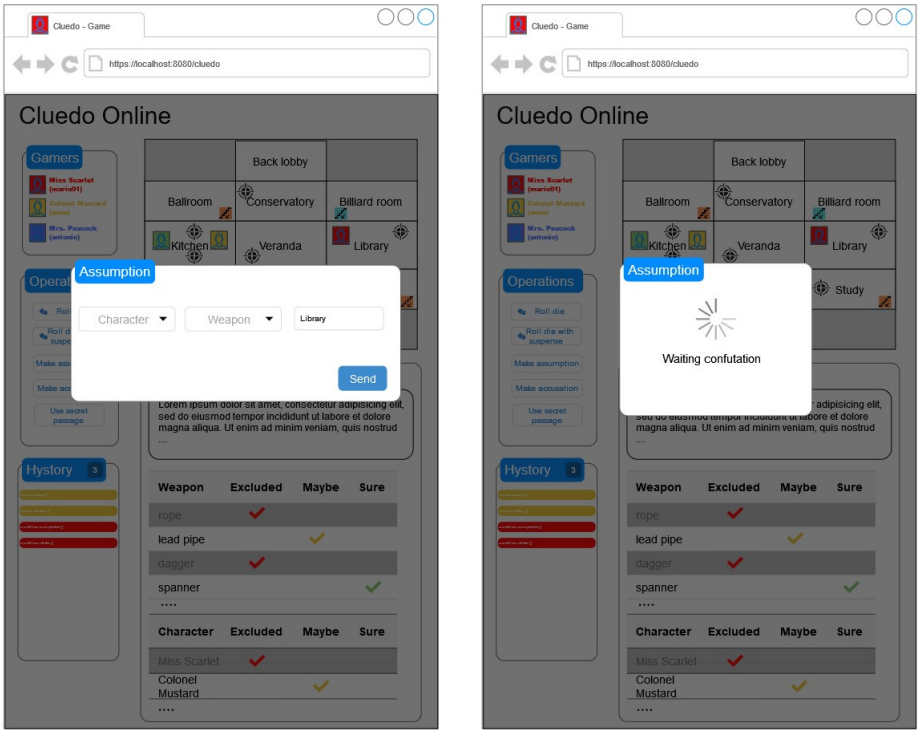


Figura 27: Mockup: Fare un'ipotesi

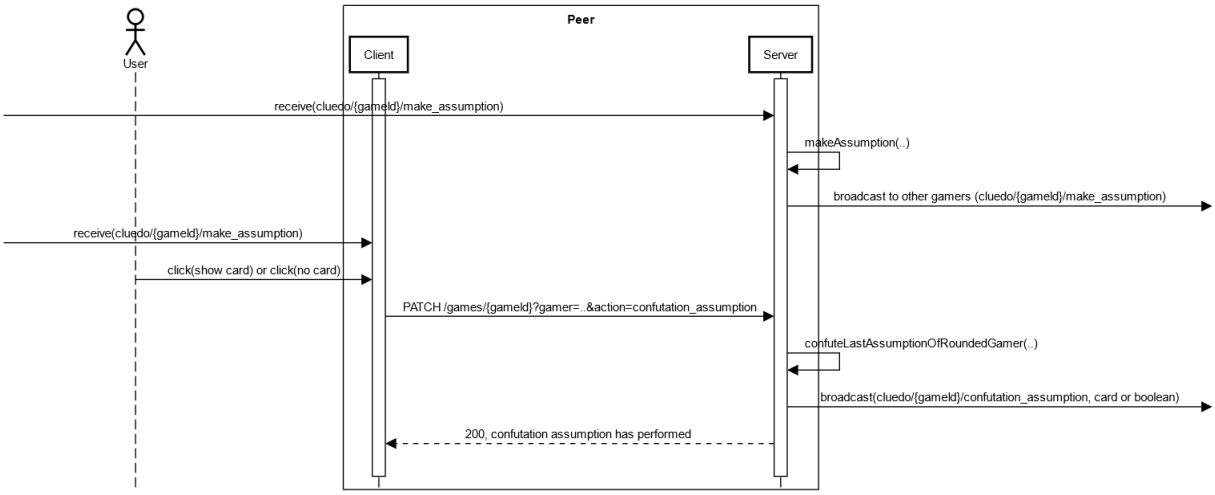


Figura 28: Diagramma di sequenza: Confutazione di un'ipotesi

Dopo che un giocatore ha ricevuto l'ipotesi dal giocatore di turno (figura 28), egli deve scegliere una "carta" (tra le sue) relativa all'ipotesi effettuata, nel caso ne abbia una. A questo punto il Peer (Client) invia in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che effettua la confutazione* e *action = con-*

futation_assumption. La richiesta permette di memorizzare la confutazione effettuata sul Peer (Server) attraverso il metodo *confuteLastAssumptionOfRoundedGamer* del *GameManager* e successivamente inviarla al giocatore di turno attraverso l'evento socket *cluedo/{gameId}/confutation_assumption*. Però anche gli altri giocatori devono ricevere un input, sempre attraverso l'evento *cluedo/{gameId}/confutation_assumption* però come messaggio viene inviato un booleano che conferma o meno l'avenuta confutazione. Nel caso negativo, il giocatore successivo cerca di confutare l'ipotesi nello stesso modo e così via finché almeno un giocatore non ha confutato l'ipotesi oppure tutti i giocatori (escluso quello di turno) hanno inviato una "carta vuota".

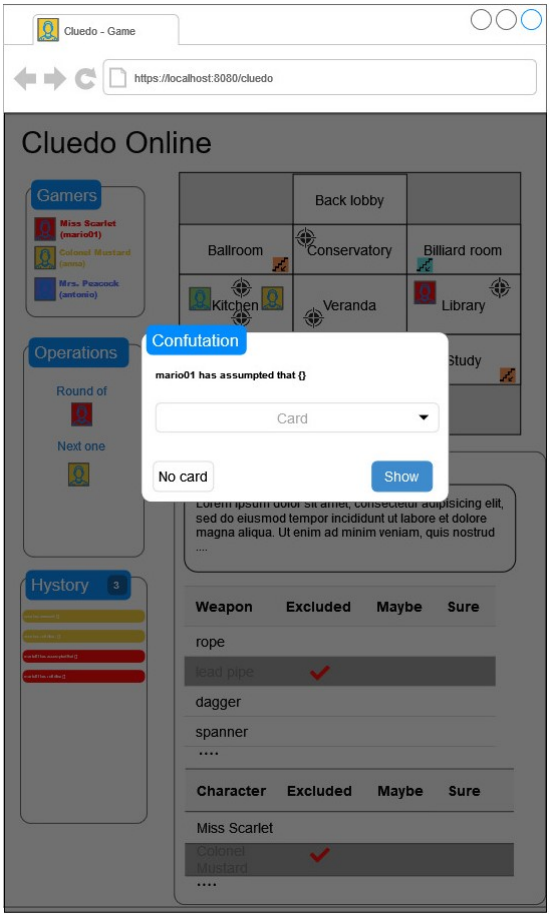


Figura 29: Mockup: Confutazione di un'ipotesi

In fine quando il giocatore di turno riceve almeno una "carta" oppure, per tutti gli altri giocatori, ha ricevuto una "carta vuota", il suo turno finisce e quindi viene inviata una richiesta di fine turno (vista in precedenza).

Fare un'accusa

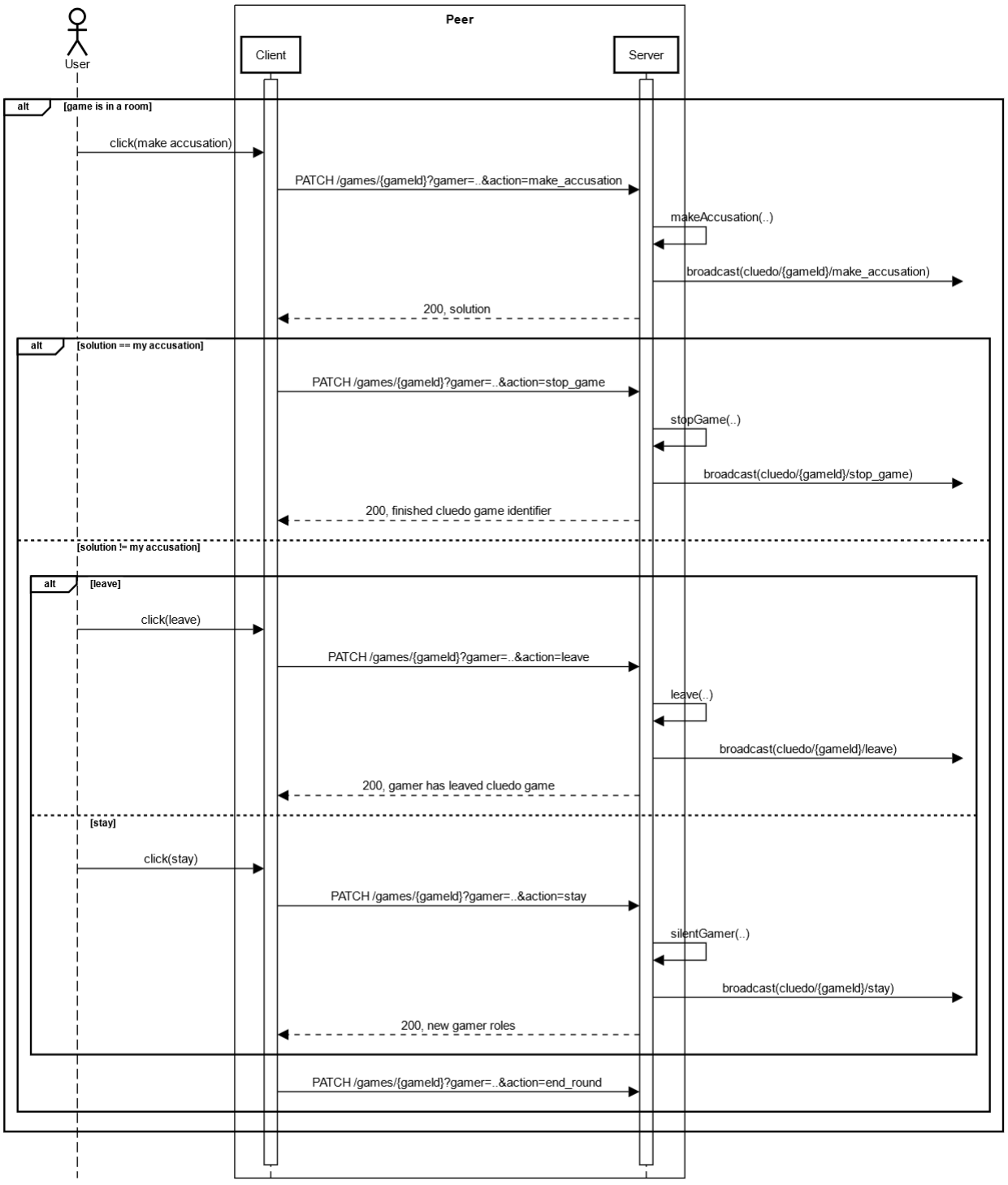


Figura 30: Diagramma di sequenza: Fare un'accusa

Come mostrato in figura 30, quando un giocatore vuole effettuare un'accusa, il Peer (Client) invia in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che effettua l'accusa* e *action = make_accusation*. La richiesta permette di memorizzare l'accusa effettuata dal giocatore di turno sul Peer (Server) attraverso il metodo *makeAccusation* del *GameManager* e successivamente inviarla a tutti gli altri giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/make_accusation*.

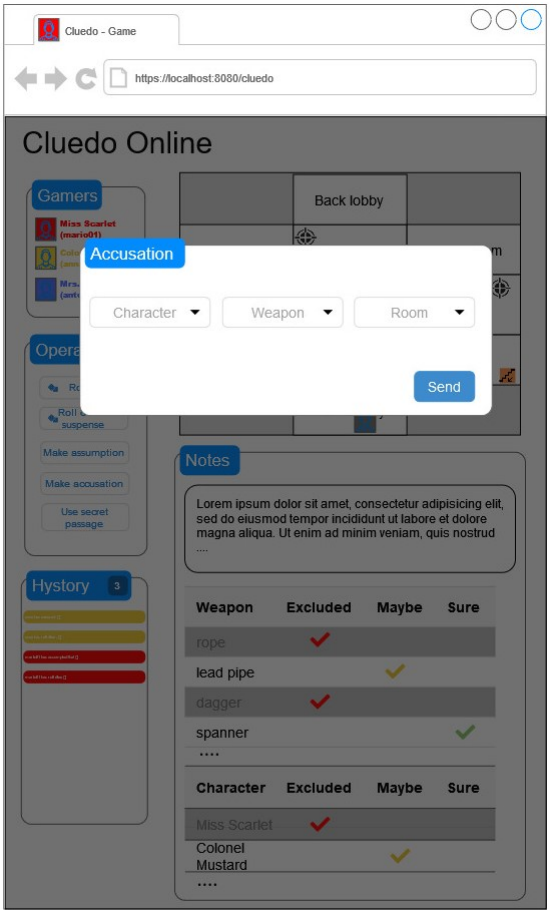


Figura 31: Mockup: Fare un'accusa

Nel caso l'accusa sia uguale alla soluzione, il gioco termina con vincitore il giocatore e viene inviata in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore* e *action = stop_game*. La richiesta permette di fermare il gioco attraverso il metodo *stopGame* del *GameManager* e successivamente propagare la fine del gioco anche a tutti i clients connessi (escluso il chiamante) attraverso l'evento socket *cluedo/{gameId}/stop_game*.

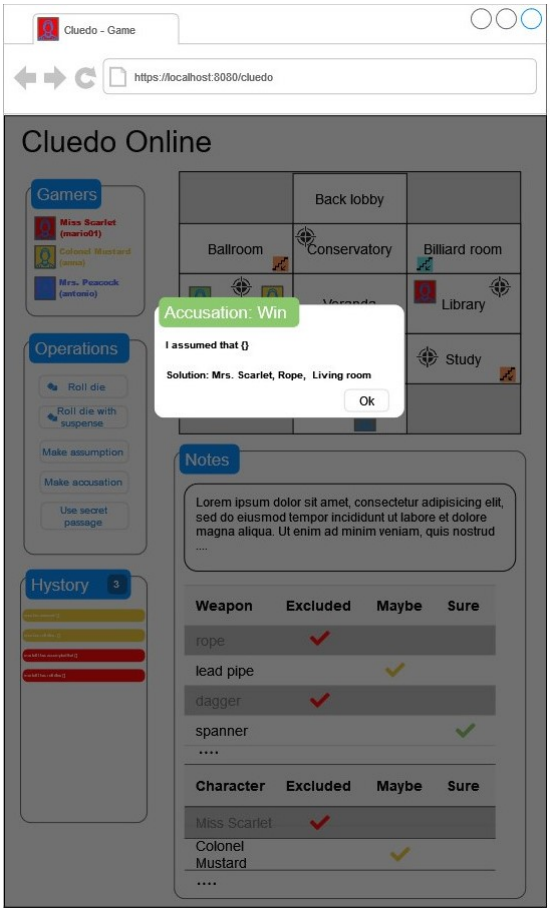


Figura 32: Mockup: Vincere la partita

Nel caso contrario, il giocatore deve decidere se rimanere oppure abbandonare il gioco.

Nel caso il giocatore decida di restare, viene inviata in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che vuole rimanere* e *action = stay*. La richiesta permette l'assegnamento del nuovo ruolo (*silent*) al giocatore attraverso il metodo *silentGamerInRound* del *GameManager* e l'invio delle modifiche anche a tutti gli altri giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/stay*.

Nel caso il giocatore decida di abbandonare la partita, viene inviata in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore che vuole abbandonare la partita* e *action = leave*. La richiesta permette di rimuovere il giocatore e ridistribuire le "carte" del giocatore uscente attraverso il metodo *leave* del *GameManager* e successivamente propagare le modifiche anche a tutti gli altri giocatori connessi (anche in diversi Peer) attraverso l'evento socket *cluedo/{gameId}/leave*.

Dopo tutto ciò, il suo turno finisce e quindi viene inviata una richiesta di fine turno (vista in precedenza).

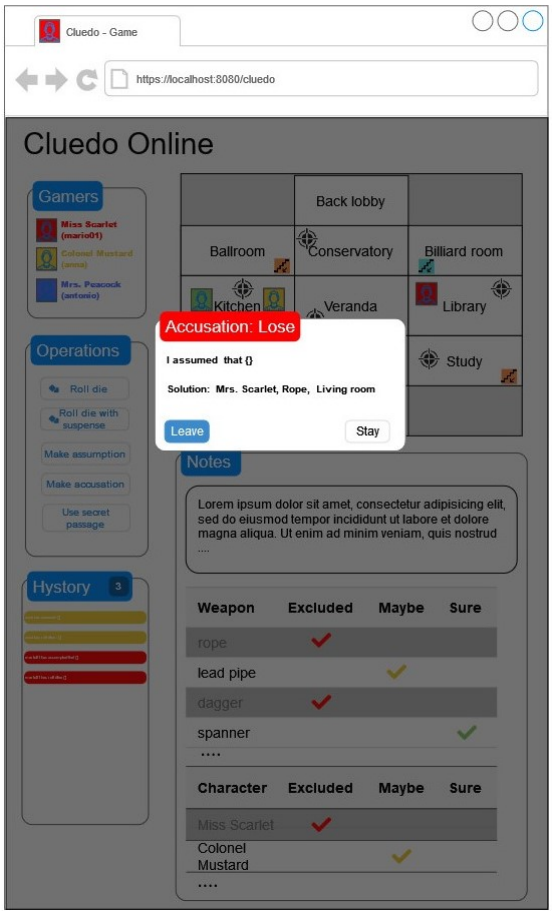


Figura 33: Mockup: Perdere la partita

Prendere appunti

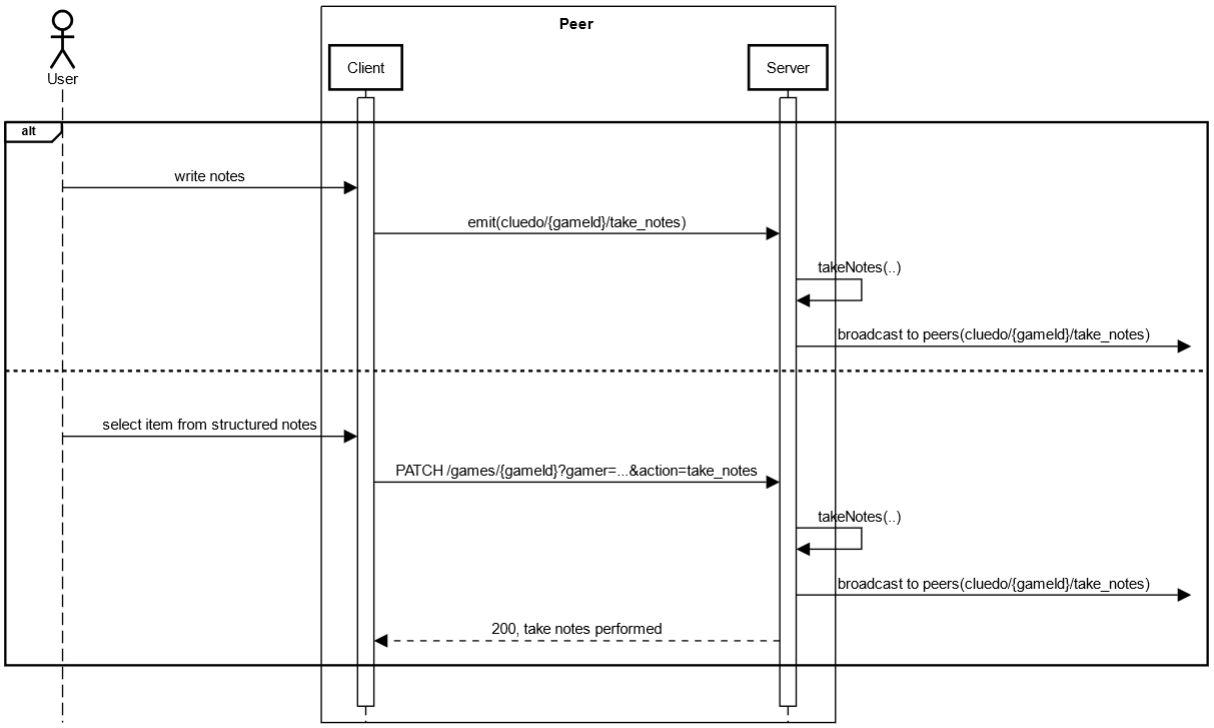


Figura 34: Diagramma di sequenza: Prendere appunti durante una partita

Come mostrato in figura 34, quando un giocatore "scrive" qualche appunto sul delitto, viene inviata in PATCH una richiesta GAMES/{gameId} con query *gamer=identificatore del giocatore* e *action = take_notes* oppure viene emesso l'evento socket *cluedo/{gameId}/take_notes* a seconda se le note sono strutturate oppure libere. Questa richiesta o evento permette di memorizzare nel Peer (Server) le note attraverso il metodo del *GameManager*, *takeNotes* ed inviarle a tutti i peer connessi (escluso il chiamante) attraverso l'evento socket *cluedo/{gameId}/take_game*.
A differenza degli altri eventi socket, dopo averlo ricevuto un Peer non lo propaga agli altri giocatori (o agli altri clients), ma semplicemente lo memorizza nello storage e lo utilizza come backup in caso che il Peer in cui è collegato il giocatore termini.

Disconnessione di un giocatore

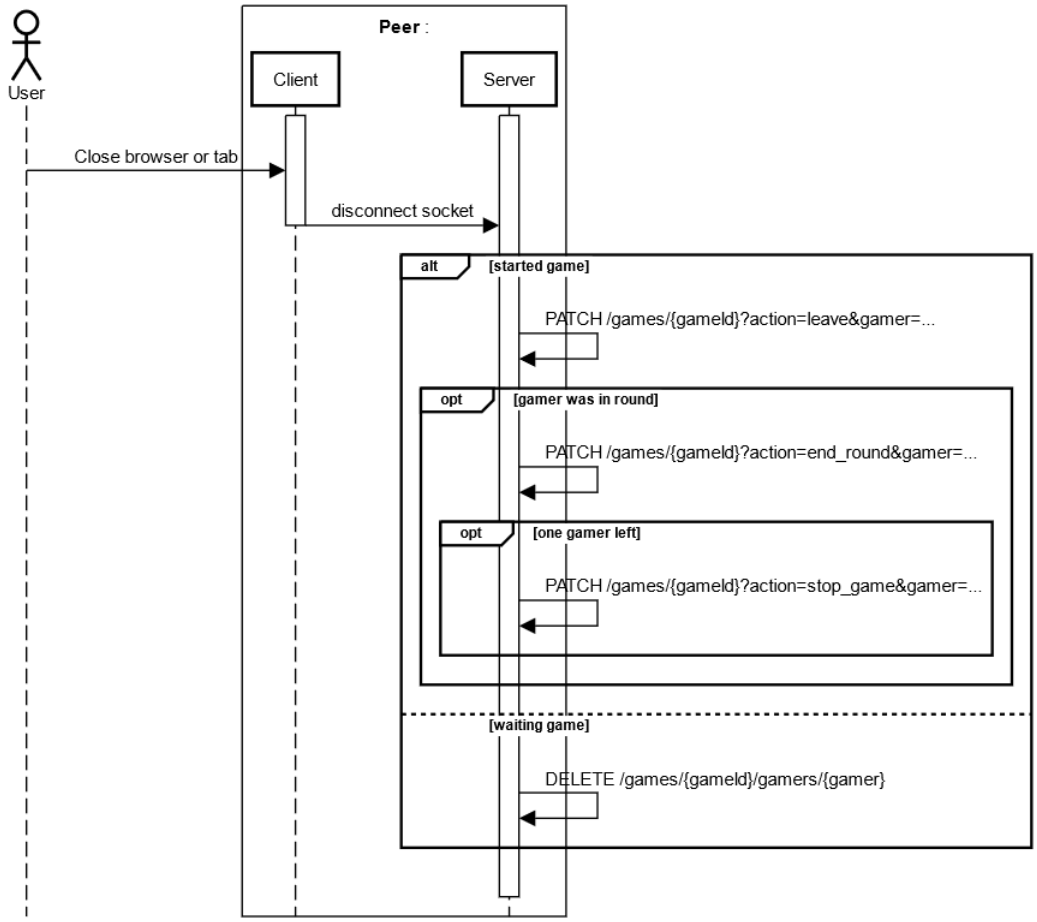


Figura 35: Diagramma di sequenza: Disconnessione di un giocatore

Come mostrato in figura 35, quando un giocatore chiude l'applicativo (ossia chiude il browser o il tab del browser), il Peer (Client) scatena la disconnessione della socket collegata al Peer (Server). Come conseguenza nel Peer (Server) vengono eseguite dell'operazioni per la rimozione dell'eventuale giocatore associato alla socket:

- *PATCH /games/{gameId}?action=leave&gamer=...* (descritta nella sezione *Azioni in partita*): la richiesta viene eseguita nell'eventualità che la partita sia stata avvita.
- *DELETE /games/{gameId}/gamers/{gamer}* (descritta nella sezione *Uscita di un giocatore da una partita (non avviata)*): la richiesta viene eseguita nell'eventualità che la partita non sia ancora stata avvita.

Altre operazioni che possono essere eseguite sono:

- *PATCH /games/{gameId}?action=end_round&gamer=...* (descritta nella sezione *Azioni in partita*): la richiesta viene eseguita nell'eventualità che il giocatore rimosso fosse in turno.
- *PATCH /games/{gameId}?action=stop_game&gamer=...* (descritta nella sezione *Azioni in partita*): la richiesta viene eseguita nell'eventualità che nella partita ci fosse rimasto solo un giocatore con ruolo *participant*.

Disconnessione di un Peer

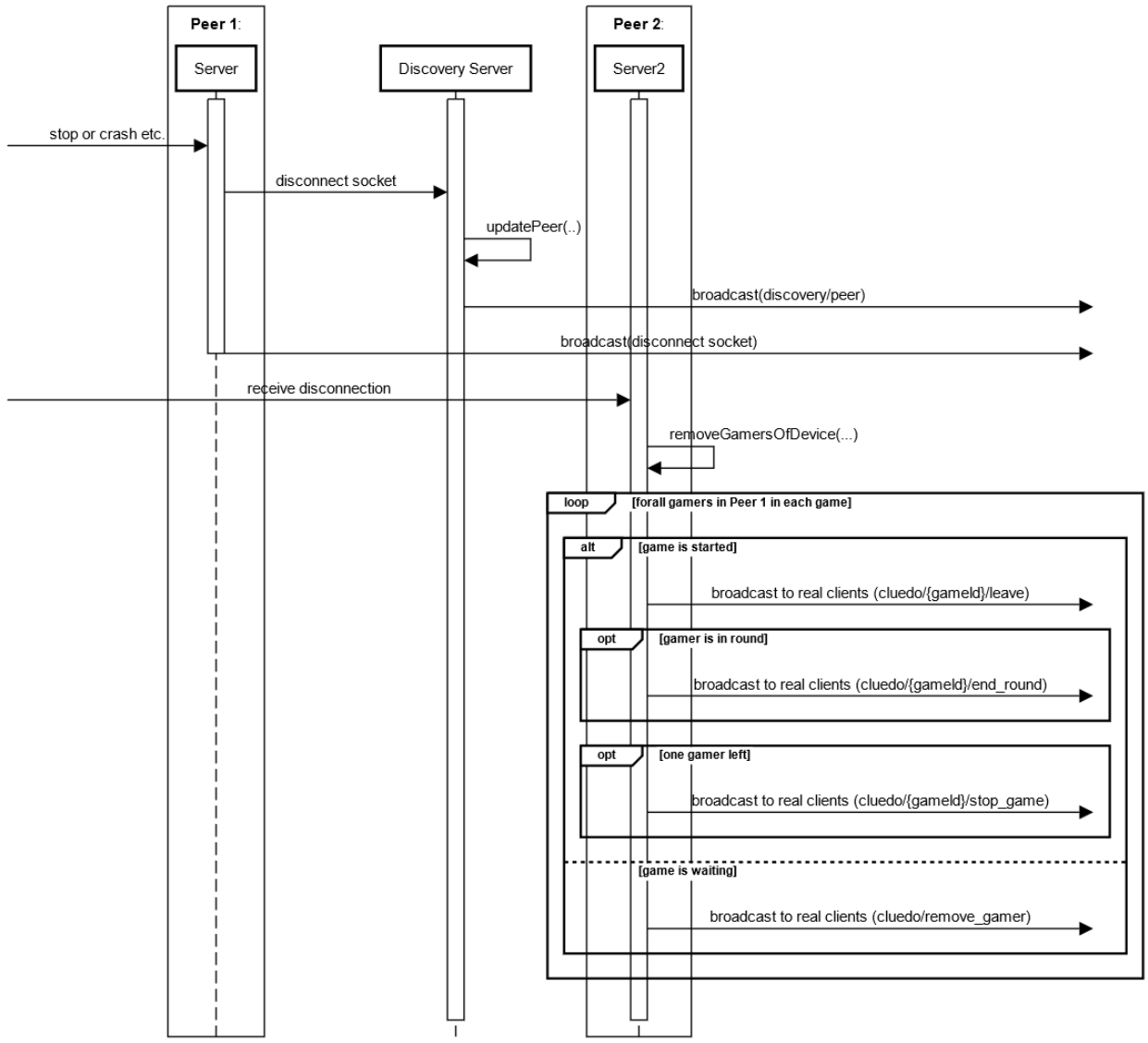


Figura 36: Diagramma di sequenza: Disconnessione di un peer

Come mostrato in figura 36, quando il Peer (Server) viene terminato oppure in caso di crash, viene scollegato da tutti i servers e clients. In conseguenza nel Discovery Server, viene aggiornato lo stato del Peer attraverso il metodo *updatePeer* del *DiscoveryManger* e inviato a tutti gli altri peer.

Quando un Peer (Server) riceve il messaggio di disconnessione di un altro Peer, rimuove ogni giocatore associato a quel Peer per ogni partita memorizzata. Di conseguenza verranno inviati degli eventi socket:

- *cluedo/{gameId}/leave*: l'evento inviato nel caso il giocatore fosse in una partita avviata.
- *cluedo/{gameId}/end_round*: l'evento viene inviato nel caso il giocatore fosse in turno in una partita avviata.
- *cluedo/{gameId}/stop_game*: l'evento viene inviato nel caso nella partita (avviata) ci fosse rimasto solo un giocatore con ruolo *participant*.
- *cluedo/remove_gamer*: l'evento viene inviato nel caso il giocatore fosse in una partita non ancora avviata.

4 Dettagli di implementazione

In questa sezione verranno discussi alcuni tra gli aspetti più interessanti e rilevanti dell'implementazione.

Di seguito è illustrata tramite uno schema dei package (figura 37) la struttura del progetto.

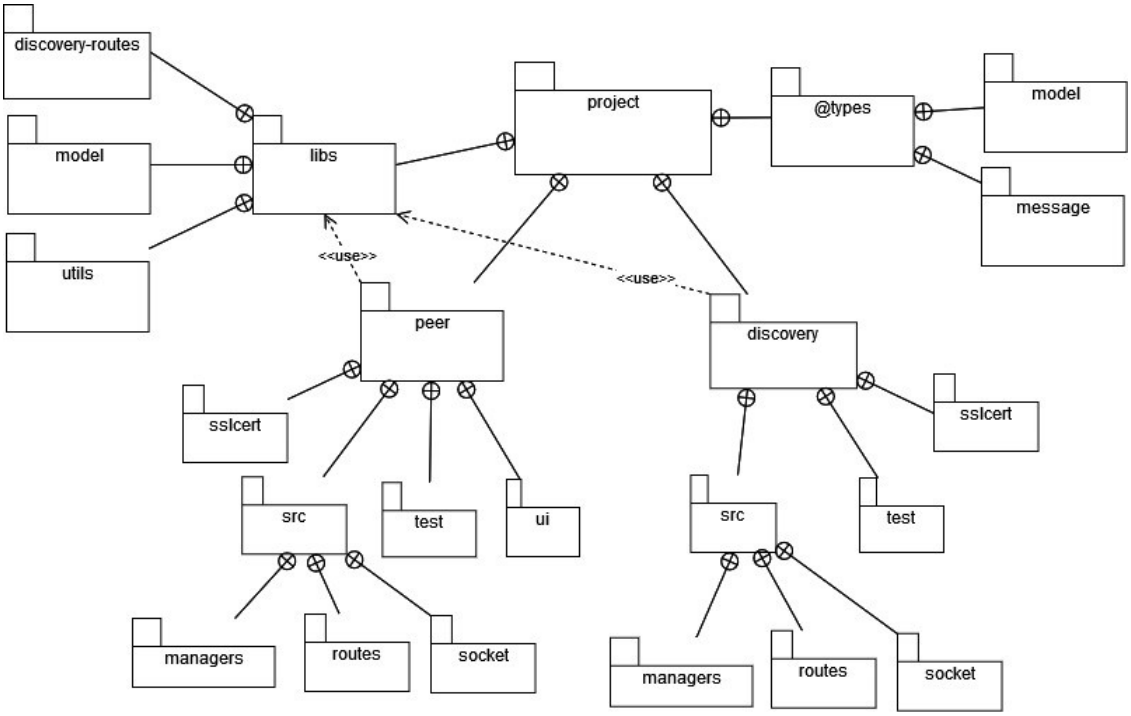


Figura 37: Struttura del progetto

Model e Message

Per l'implementazione dell'entità principali e dei messaggi (socket) vengono sfruttati i *Declaration files* (.d.ts) di Typescript [14], ossia files in cui vengo dichiarate le varie interfacce. Queste interfacce vengo rese globali in tutto il progetto inserendo i *declaration files* nel file di configurazione del progetto (tsconfig.json) tramite il campo *compilerOptions.typeRoots*.

Di seguito viene riportato un esempio di *Declaration files*:

```
/**
 * It represents a gamer of a cluedo game.
 */
5+ usages
declare interface Gamer {
  identifier: string;
  username: string;
  characterToken: string;
  role?: string[];
  device?: Device;
  assumptions?: Assumption[];
  accusation?: Suggestion;
  cards?: string[];
  notes?: Notes;
}

/**
 * It represents an item of structured note.
 */
5+ usages
declare interface StructuredNoteItem {
  name: string;
  suspectState: string;
  confutation?: true;
}

/**
 * It represents gamer's notes taken during the cluedo game.
 */
5+ usages
declare interface Notes {
  text?: string;
  structuredNotes?: StructuredNoteItem[];
}
```

Figura 38: File *./@types/model/gamer.model.d.ts*

Come mostrato in figura 38, alcuni campi come ad esempio *characterToken*, *role* oppure *suspectState*, sono dichiarati di tipo *string*, il quale è un tipo differente da quello modellato nella sezione Design. Questa scelta implementativa è stata fatta per facilitare l'estensione. Quindi se in futuro si volessero aggiungere o rimuove dei valori di questi campi, le interfacce non andrebbero modificate.

Le variabili con i valori relativi a questi campi sono dichiaranti nel package *./libs/model*. Per dichiararne i valori sono stati sfruttati le enumerazioni (figura 39).

```ts
export namespace Gamer {

  export enum Role {
    CREATOR = 'creator',
    PARTICIPANT = 'participant',
    SILENT = 'silent',
  }
}
```

(a) Variabili in ./libs/model/game.model.ts

```ts
export namespace GameElements {
  ...
  export enum CharacterName {
    PROFESSOR_PLUM = 'professor plum',
    MRS_PEACOCK = 'mrs. peacock',
    REVEREND_GREEN = 'reverend green',
    MISS_SCARLET = 'miss scarlet',
    COLONEL_MUSTARD = 'colonel mustard',
    MRS_WHITE = 'mrs. white',
  }
  ...
  export enum SuspectState {
    EXCLUDED = 'excluded',
    MAYBE = 'maybe',
    SURE = 'sure',
  }
  ...
}
```

(b) Alcune variabili in ./libs/model/game-element.model.ts

Figura 39: Alcune variabili del *model*

Protocollo di comunicazione: HTTPS

Sia il Discovery server che il lato server del Peer fruttano il protocollo di comunicazione HTTPS. Esso consiste nella comunicazione tramite il protocollo HTTP all'interno di una connessione criptata, tramite crittografia asimmetrica, dal Transport Layer Security (TLS) o dal suo predecessore, Secure Sockets Layer (SSL) fornendo come requisiti chiave:

- un'autenticazione di un giocatore in una specifica partita
- protezione della privacy (riservatezza o confidenzialità)
- integrità dei dati scambiati tra le parti comunicanti

Per creare i certificati SSL, è stato richiesto il software **openssl**[12].
Di seguito i comandi utilizzati:

```
$ openssl genrsa -out privatekey.pem
$ openssl req -new -key privatekey.pem -out csr.pem
$ openssl x509 -req -days 365 -in csr.pem -signkey privatekey.pem \
  -out cert.pem
```

Rest API e Socket API

Le comunicazioni tra diversi Peer e il Discovery Server avvengono tramite API RESTful ed eventi SOCKET.
Le specifiche dell'API RESTful e degli eventi SOCKET vengono fornite tramite Swagger:

- Cluedo RESTful API (Open API)
- Cluedo Socket API (Async API)

Per l'hosting e distribuzione delle API RESTful si è optato per la combinazione *Node.JS + Express*. Questa combinazione fornisce un'interfaccia uniforme per l'accesso e la manipolazione delle risorse sul web e restituiscono i dati in un formato comune come JSON oppure text/pain. Un'applicazione Express è essenzialmente una serie di chiamate di funzioni middleware (figura 40). Le funzioni middleware sono funzioni che hanno accesso all'oggetto richiesta (req), all'oggetto risposta (res) e alla successiva funzione middleware nel ciclo richiesta-risposta dell'applicazione. La successiva funzione middleware è comunemente indicata da una variabile denominata *next*.

```
app.route(DiscoveryRouteName.BASE).get(controller.baseHandler);

app
  .route(RestAPIRouteName.PEERS)
  .post(
    middleware.post.handlerBadRequest,
    middleware.post.handlerForbiddenRequest,
    controller.restApi.postPeer
  )
  .get(controller.restApi.getPeers);

app
  .route(RestAPIRouteName.PEER)
  .patch(
    middleware.patch.handlerBadRequest,
    middleware.patch.handlerNotFoundRequest,
    middleware.patch.handlerUnauthorizedRequest,
    middleware.patch.handlerForbiddenRequest,
    controller.restApi.updateStatusPeer
  )
  .delete(
    middleware.remove.handlerBadRequest,
    middleware.remove.handlerNotFoundRequest,
    middleware.remove.handlerUnauthorizedRequest,
    middleware.remove.handlerForbiddenRequest,
    controller.restApi.deletePeer
  );

app.use(serverError);

app.use(pathNotFound);
```

(a) Discovery Server

```
if (process.env.NODE_ENV === 'production') {
  app.use(serveStatic(path.resolve( paths: '..', 'build', 'peer', 'ui', 'dist')));
}

app
  .route(RestAPIRouteName.GAMES)
  .post(
    middleware.games.handlerBadRequest,
    controller.restApi.postGames(peer)
  )
  .get(middleware.games.handlerBadRequest, controller.restApi.getGames);

app
  .route(RestAPIRouteName.GAME)
  .get(
    middleware.games.handlerBadRequest,
    middleware.games.handlerNotFoundRequest,
    middleware.games.extractAccessToken,
    controller.restApi.getGame
  )
  .patch(
    middleware.games.handlerBadRequest,
    middleware.games.handlerNotFoundRequest,
    middleware.games.handlerUnauthorizedRequest,
    middleware.games.handlerForbiddenRequest,
    middleware.games.handlerGoneRequest,
    controller.restApi.patchGame
  );

app
  .route(RestAPIRouteName.GAMERS)
  .post(
    middleware.games.handlerNotFoundRequest,
    middleware.gamers.handlerBadRequest,
    middleware.gamers.handlerGoneRequest,
    controller.restApi.postGamers(peer)
  );

app
  .route(RestAPIRouteName.GAMER)
  .delete(
    middleware.gamers.handlerNotFoundRequest,
    middleware.gamers.handlerUnauthorizedRequest,
    middleware.gamers.handlerForbiddenRequest,
    middleware.gamers.handlerGoneRequest,
    controller.restApi.deleteGamer
  );

app.use(serverError);

app.use(pathNotFound);
```

(b) Peer

Figura 40: Esempi di utilizzo delle funzioni *middleware* di *Express*

Per la comunicazione in real time si è deciso di utilizzare la libreria *Socket.io*. La libreria lavora tramite canali di comunicazione su cui vengono inviati dei messaggi. I messaggi vengono intercettati dal destinatario solamente se nel canale del ricevente è presente un *listener* associato a uno specifico evento (figura 41).

```
socket
  .on(CluedoGameEvent.CLUEDO_NEW_GAME, (game: CluedoGameMessage) : void  => {...})
  .on(CluedoGameEvent.CLUEDO_NEW_GAMER,  listener: (message: GamerMessage) : void  => {...})
  .on(CluedoGameEvent.CLUEDO_REMOVE_GAMER,  listener: (message: ExitGamerMessage) : void  => {...})
```

Figura 41: Esempi di utilizzo dei *listeners* di *Socket.io*

5 Self-assessment / Validation

5.1 Test automatizzati

La validazione del software prodotto è stata, per quanto possibile, effettuata adottando un approccio di sviluppo Test-Driven utilizzando il framework di testing Mocha.

Il ruolo dei test automatizzati in un sistema software è quello di garantire che il sistema si comporti come previsto e di rilevare eventuali problemi nelle prime fasi del processo di sviluppo, contribuendo così a migliorare la qualità complessiva del software.

Alcuni dei test automatizzati implementati, come quelli del *Discovery Manager* (figura 42) e quelli del *Games Manager* (figura 43), sono *unit tests* eseguiti su singole unità o componenti di codice per garantire che funzionino correttamente in isolamento.

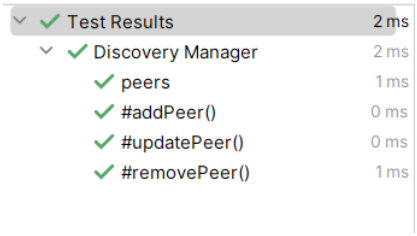


Figura 42: Risultati dei Test Discovery Manager

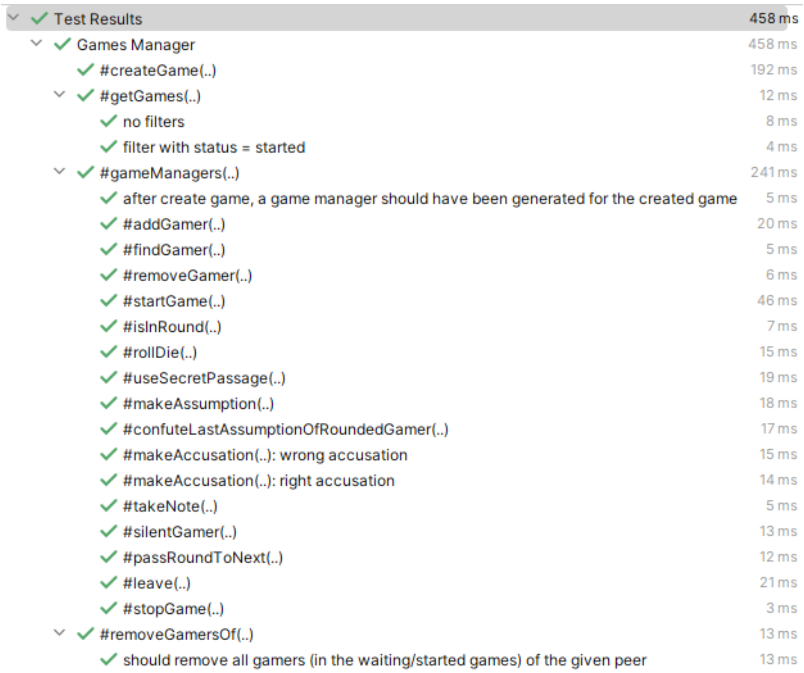


Figura 43: Risultati dei Test Games Manager

Mentre gli altri tests si concentrano anche sull'integrazione di più componenti, ossia quelli che testano le REST API (figure 44 e 46) e SOCKET API (figure 45 e 47).

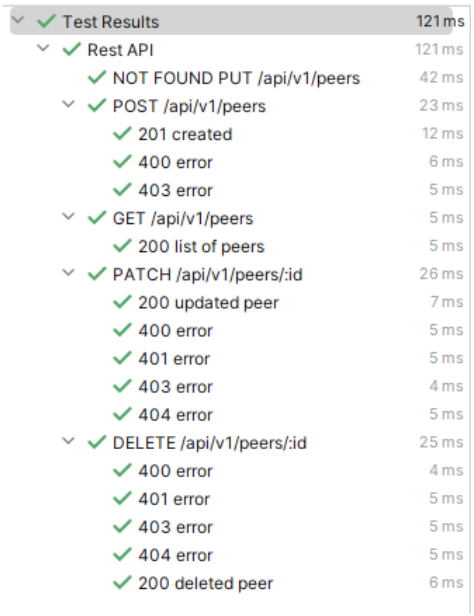


Figura 44: Risultati dei Test DISCOVERY REST API

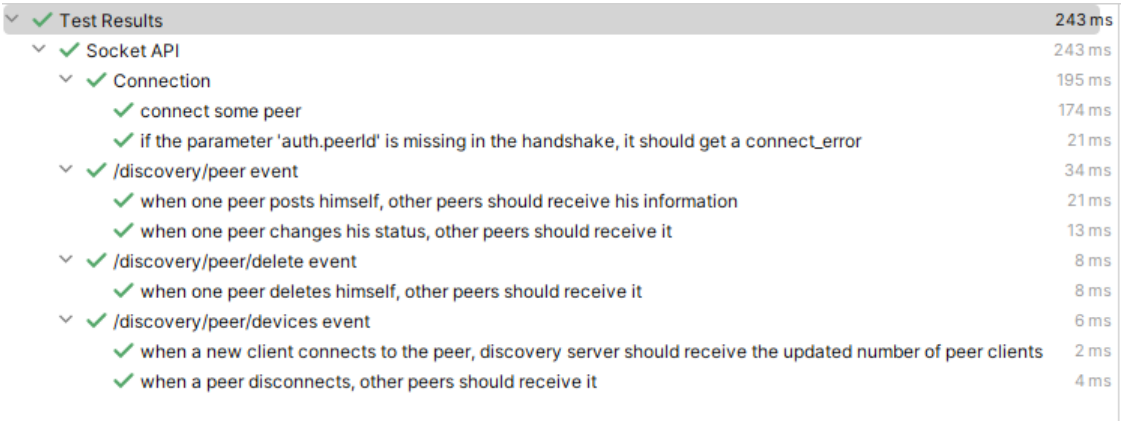


Figura 45: Risultati dei Test DISCOVERY SOCKET API

Figura 46: Risultati dei Test PEER REST API

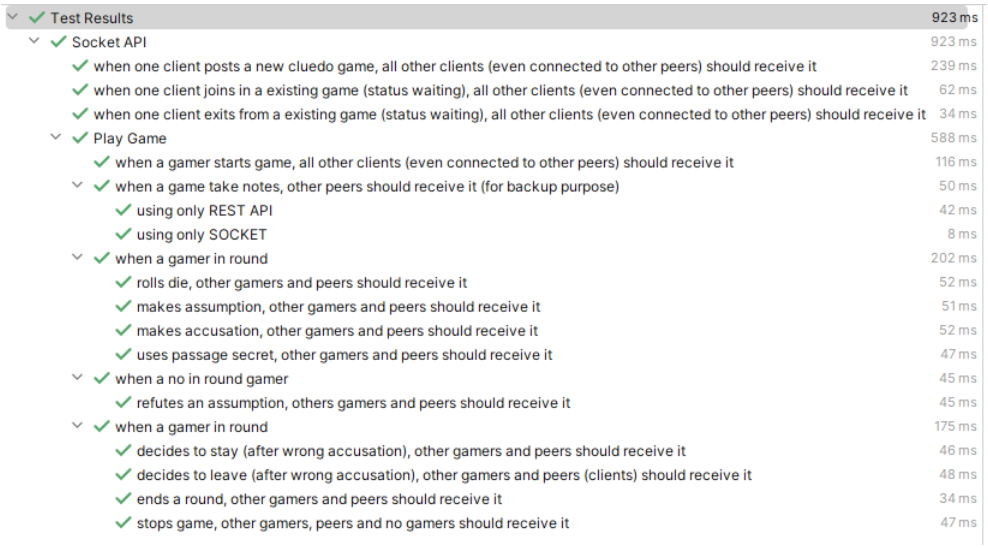


Figura 47: Risultati dei Test PEER SOCKET API

Coverage

La code coverage, o copertura del codice, viene solitamente definita come la percentuale di codice attraversato dei test rispetto al totale della code base. Il testing mostra la presenza di errori per cui, non si è cercato di avere necessariamente un testing con il 100% di coverage, in quanto non sarebbe comunque immune da bugs, lo scopo infatti non è stato quindi aumentare al massimo la coverage, ma testare le parti appropriate dell'applicazione (figure 48 e 49).

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	93.77	80.23	100	93.72	
managers/peers	94.11	66.66	100	93.75	
index.ts	94.11	66.66	100	93.75	49
managers/peers/devices	100	100	100	100	
index.ts	100	100	100	100	
routes	100	100	100	100	
index.ts	100	100	100	100	
routes/controller	92.98	81.81	100	92.98	
index.ts	86.2	60	100	86.2	47-52,64
rest-api.controller.ts	100	100	100	100	
routes/middleware	91.5	80	100	91.5	
delete.middlewares.ts	92.1	80	100	92.1	29,76,100
index.ts	100	100	100	100	
patch.middlewares.ts	90.47	79.16	100	90.47	26,32,88,112
post.middlewares.ts	91.3	83.33	100	91.3	24,32
socket	100	83.33	100	100	
index.ts	100	83.33	100	100	58

Figura 48: Coverage dei tests del Discovery Server

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	86.42	71.12	81.84	86.52	
src	73.33	41.66	60	73.33	
utils.ts	73.33	41.66	60	73.33	17-21
src/managers/games/mongoose	93.85	72.13	94.62	93.65	
errors.ts	90.9	50	66.66	90.9	15
index.ts	94	72.36	95.55	93.81	261,267,323,377,428-444
src/managers/games/mongoose/schemas	90	100	66.66	90	
index.ts	90	100	66.66	90	90
src/managers/peers-servers	70.37	37.5	76.92	77.27	
index.ts	70.37	37.5	76.92	77.27	47,57-59,67
src/routes	97.05	87.5	100	97.05	
index.ts	93.75	50	100	93.75	17
parameters.ts	100	100	100	100	
routesNames.ts	100	100	100	100	
src/routes/controller	100	100	100	100	
index.ts	100	100	100	100	
src/routes/controller/rest-api.controller	89.87	74.35	84.88	90.2	
game.controller.ts	86.2	73.8	79.03	86.2	90,127,162,204-228,270-275,376,411-413,425,467,545-550
index.ts	97.8	80.64	100	98.87	224
utils.ts	72.72	40	100	72.72	38-43
src/routes/middleware	90.11	80.53	97.05	90	
gamers.middlewares.ts	93.47	81.81	100	93.47	89,118,141
games.middlewares.ts	88.7	80.21	95	88.52	81,104,152,190,224,238,259,272,279-284,288,296,320
index.ts	100	100	100	100	
src/socket	100	83.33	100	100	
utils.ts	100	83.33	100	100	7
src/socket/checker	100	100	100	100	
index.ts	100	100	100	100	
src/socket/client	44.44	0	31.57	44.44	
index.ts	44.44	0	31.57	44.44	48,59,66,86-136
src/socket/events	100	100	100	100	
index.ts	100	100	100	100	
src/socket/handlers	87.5	77.77	100	87.5	
utils.ts	87.5	77.77	100	87.5	23-25
src/socket/handlers/gamer	68.62	30	50	69.38	
index.ts	68.62	30	50	69.38	40-61,68-85,120
src/socket/handlers/peer	77.38	64.28	62.06	77.38	
game.handlers.ts	93.87	95.83	78.57	93.87	131,142,153
index.ts	54.28	22.22	46.66	54.28	28-74,97
src/socket/handlers/peer/game.actions	83.83	40	78.26	83.83	
index.ts	83.83	40	78.26	83.83	44,68,84,115,140,163,209,236,256-260,271-273,295
src/socket/server	100	87.5	100	100	
index.ts	100	87.5	100	100	38,75
src/socket/server/clients	84	64.28	88.88	83.33	
index.ts	84	64.28	88.88	83.33	69,116-126

Figura 49: Coverage dei tests del Peer

Per lo sviluppo di questi test vengono usati principalmente i seguenti costrutti forniti dal framework Mocha:

- **describe**: crea un blocco di codice che raggruppa diversi test correlati, non è necessario infatti si possono scrivere i blocchi di test direttamente al livello superiore, tuttavia il *describe* è utile se si preferisce organizzare i test in gruppi.

  Nella scrittura dei test viene utilizzata la *should* per controllare che i valori soddisfino determinate condizioni, infatti vengono forniti una serie di "matcher" che consentono di convalidare valori diversi.

- **before**: esegue una funzione prima che venga eseguito uno qualsiasi dei test, e se la funzione restituisce una promise, Mocha attende che la promise si risolva prima di eseguire i test. Viene utilizzato per creare i broker che servono per richiamare le azioni nei servizi da testare, e in generale impostare uno stato globale che verrà utilizzato da molti test.

- **after**: similmente alla *before* esegue una funzione dopo che tutti i test sono stati completati, se la funzione restituisce promise, Mocha attende che la promise si risolva prima di continuare. Viene utilizzata principalmente dopo aver effettuato test che andassero a chiudere le varie connessioni create con il database o connessioni socket.

5.2 CI/CD

La Continuous integration (CI) è una pratica software che richiede il commit frequente di codice in un repository condiviso. I commit frequenti del codice rilevano spesso prima gli errori e riducono la quantità di codice di cui uno sviluppatore ha bisogno per eseguire il debug quando trova l'origine di un errore. Gli aggiornamenti frequenti del codice semplificano inoltre l'unione delle modifiche apportate da diversi membri di un team di sviluppo software. Questo è ottimo per gli sviluppatori, che possono dedicare più tempo alla scrittura di codice e meno tempo al debug degli errori o alla risoluzione dei conflitti di unione. Quando si esegue il commit del codice nel repository, si può creare e testare continuamente il codice per assicurarsi che il commit non introduca errori. I test possono includere code linter (che controllano la formattazione dello stile), controlli di sicurezza, copertura del codice, test funzionali e altri controlli personalizzati. La creazione e il test del codice richiedono un server. È possibile creare e testare gli aggiornamenti in locale prima di eseguire il push del codice in un repository oppure è possibile utilizzare un server CI che verifica la presenza di nuovi commit di codice in un repository.

Il repository del progetto è ospitato su GitLab e la configurazione della pipeline CI è visibile nel file *.gitlab-ci.yml* all'interno della cartella principale del progetto.
Fondamentalmente, questo file coordina l'esecuzione di pipeline indipendenti tramite un meccanismo basato su trigger.

5.3 Test manuali

Non tutte le funzionalità dell'applicazione sono state testate utilizzando test automatizzati come ad esempio l'interfaccia utente. I test manuali vengono eseguiti da un tester umano che esegue l'applicazione sotto test, ne osserva il comportamento e confronta i risultati con quelli attesi. A differenza dei test automatizzati, i test manuali si basano sulla conoscenza, sull'esperienza e sul giudizio del tester invece di essere eseguiti da una macchina o da uno strumento. Utilizzando un approccio manuale, l'intero sistema è stato testato nel suo insieme per garantire che soddisfi le specifiche richieste e funzioni correttamente in diversi scenari. Tuttavia non è stato possibile testare il comportamento dell'architettura quando un gran numero di peers o clients interagiscono contemporaneamente tra di loro.

6 Istruzioni per il deployment

Le istruzioni di deployment del progetto in production mode sono descritte nel file README.md. Si è voluto anche riportarle in questa sezione.

6.1 Locale

Questa versione di installazione ha bisogno che siano installati i seguenti componenti:

- Node.JS
- MongoDB

Dopo di ché, per installare l'applicativo è sufficiente eseguire le seguenti istruzioni:

```
# Install dependecies
$ npm install
$ npm run install:peer-ui

# Build applications
$ npm run build

# Start Dicovery Server
$ PORT=<some-free-port> npm start -w discovery

# Start one Peer
$ MONGODB_ADDRESS=<database-uri-connection> PORT=<some-free-port> \
  DISCOVERY_SERVER_ADDRESS=<https-address-of-discovery-server> \
  npm start -w peer
```

Ora é possibile collegarsi alla piattaforma tramite un browser: `https://localhost:<port-used-to-discovery-server>` oppure di default `https://localhost:3000`.

6.2 Docker

Essendo un'architettura p2p, le applicazioni (discovery peer e peer) dovrebbero essere installate su macchine diverse. Per simulare questo tipo di installazione viene utilizzato Docker.

Sono stati scritti degli script per facilitare l'installazione dell'applicativo:

- ***./deploy-scripts/all.up.sh***: build e avvio di tutti i componenti del progetto.

```
$ bash ./deploy-scripts/all.up.sh \
  -n <number-of-peer-to-up> \
  -d <host-port-discovery-server> \
  -p <host-port-first-peer>
```

- ***./deploy-scripts/discovery.build.sh***: build dell'immagine del discovery server.

```
$ bash ./deploy-scripts/discovery.build.sh
```

- ***./deploy-scripts/discovery.start.sh***: avvio del discovery server container.

```
$ bash ./deploy-scripts/discovery.start.sh \
  -p <host-port-discovery-server>
```

- ***./deploy-scripts/peer.build.sh***: build dell'immagine del peer.

```
$ bash ./deploy-scripts/peer.build.sh
```

- ***./deploy-scripts/peer.start.sh***: avvio di uno o più peer.

```
$ bash ./deploy-scripts/peer.start.sh \
  -n <number-of-peer-to-up> \
  -p <host-port-first-peer> \
  -d <https-address-discovery-server>
```

7 Esempi d'uso

Di seguito, vengono riportate una serie di screenshot che mostrano come un utente può interagire con l'applicativo.

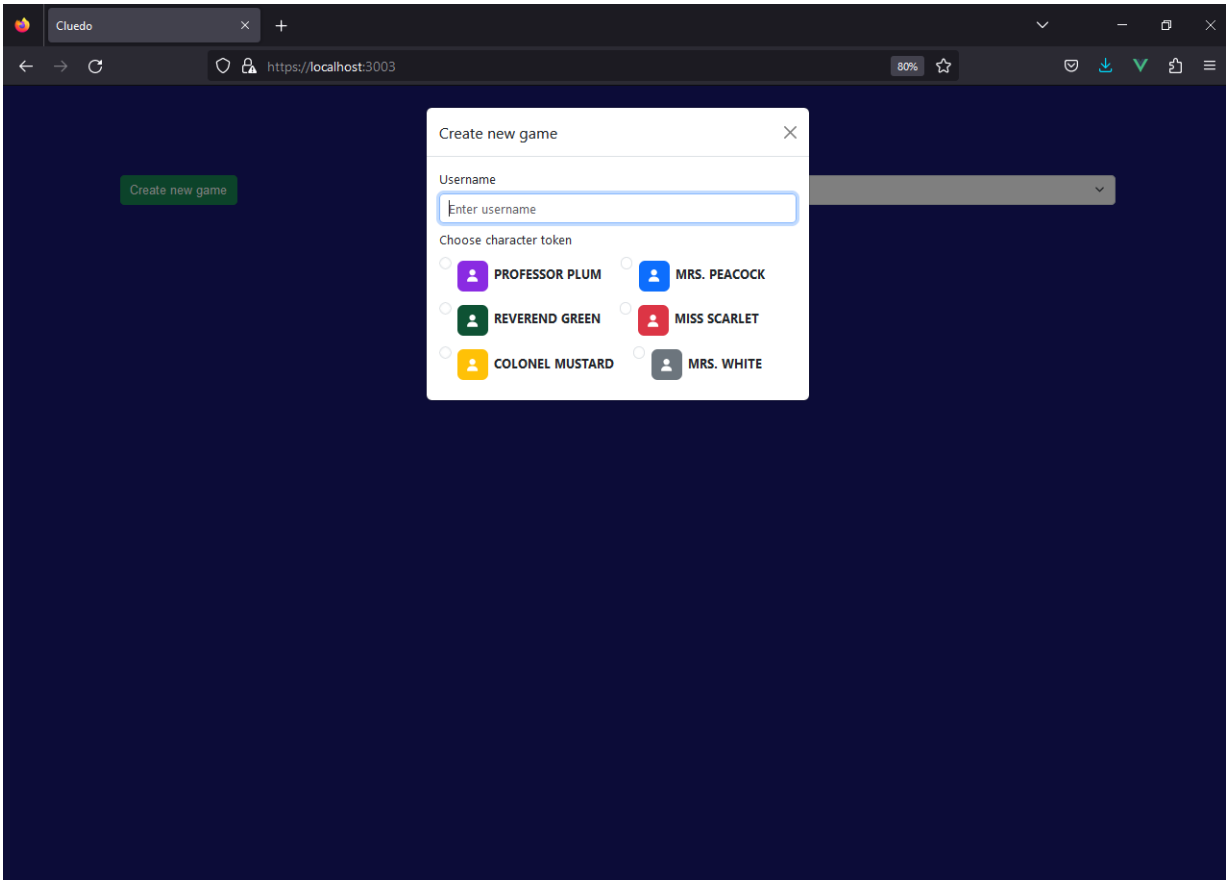


Figura 50: Esempio di creazione di una partita

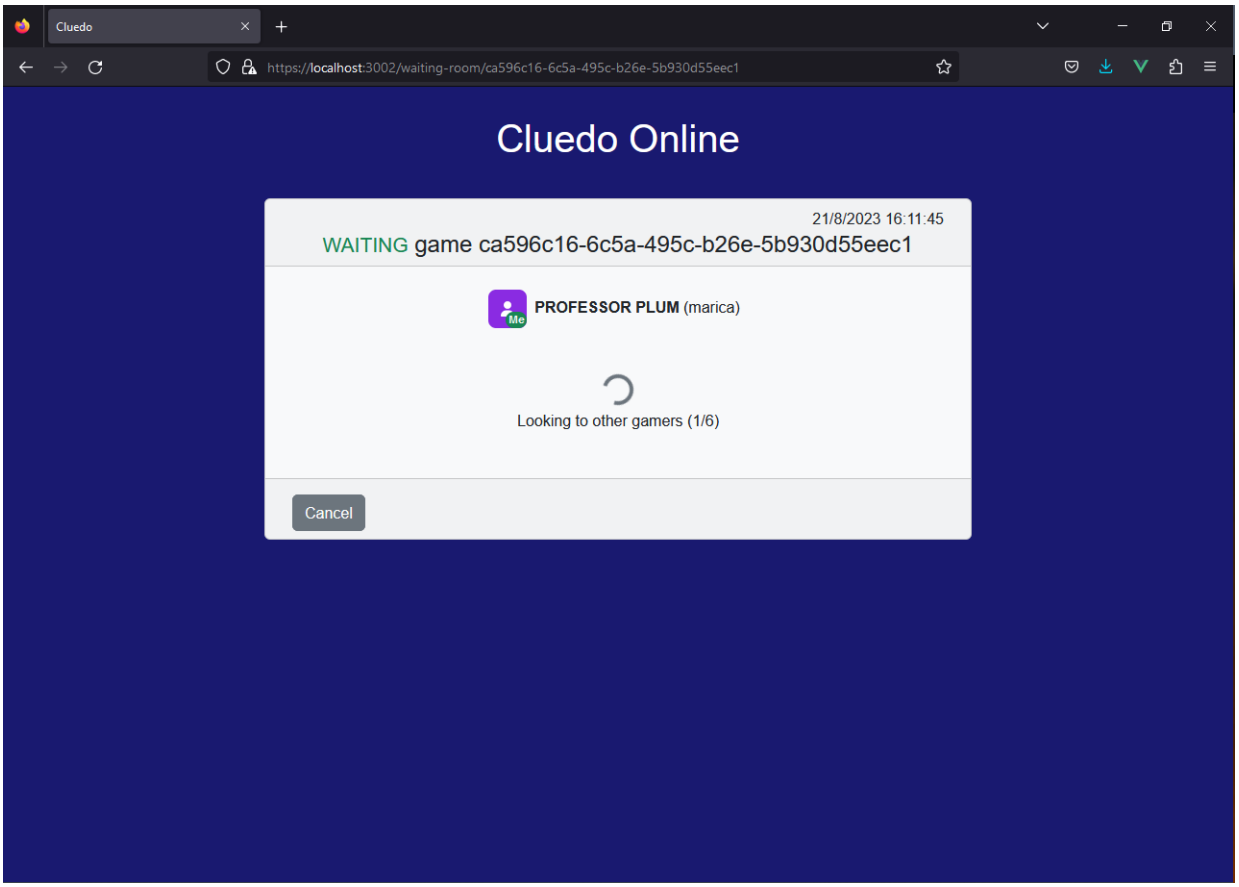


Figura 51: Esempio di una waiting room

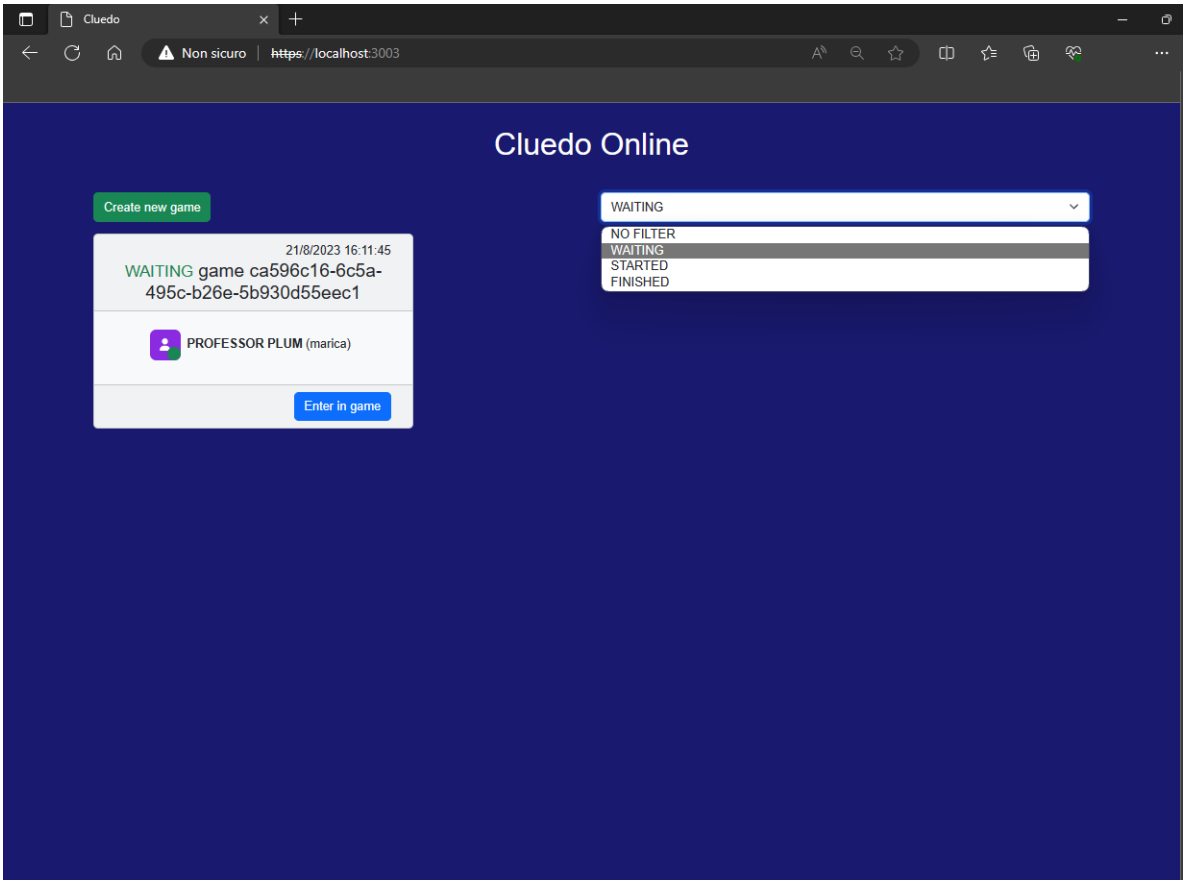


Figura 52: Esempio di “join” in una partita esistente

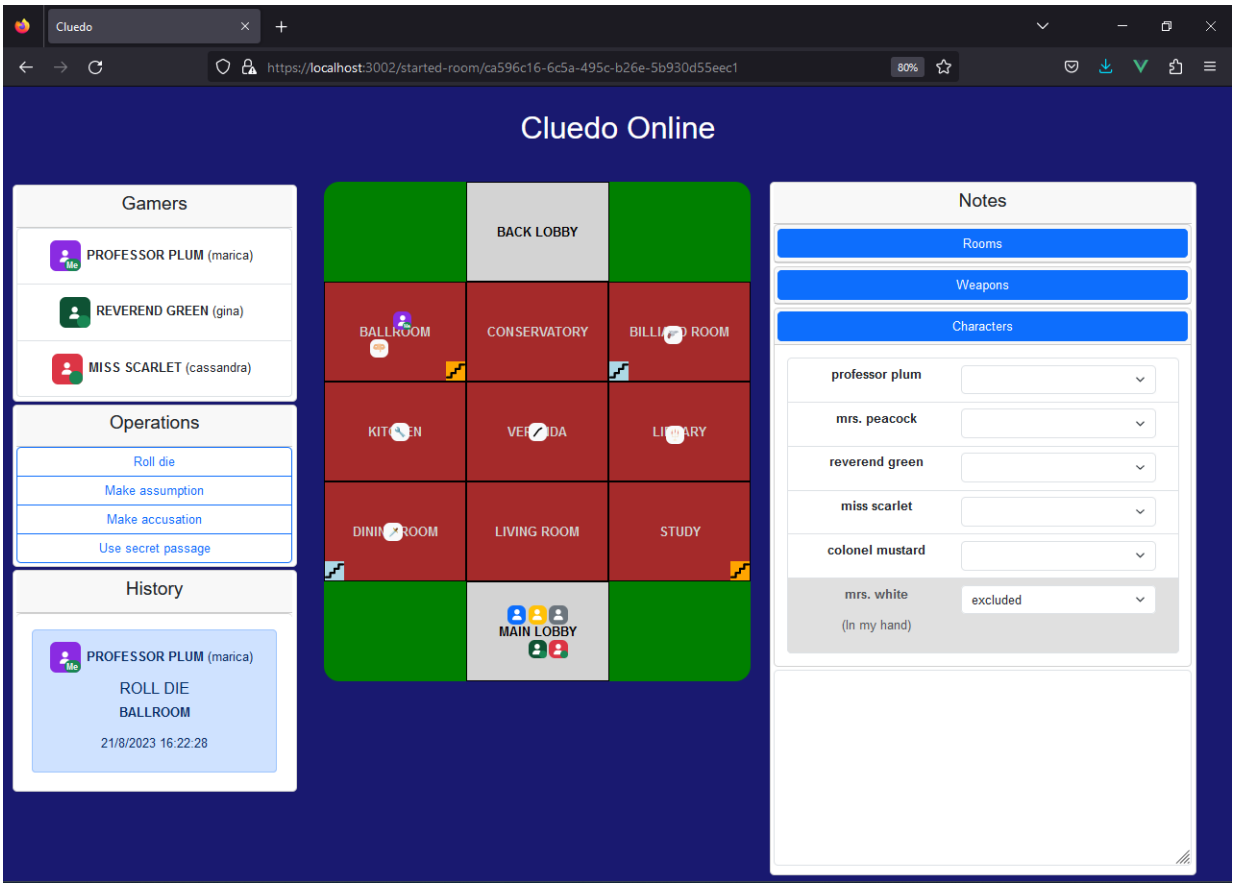


Figura 53: Esempio di schermata di gioco di un giocatore in turno

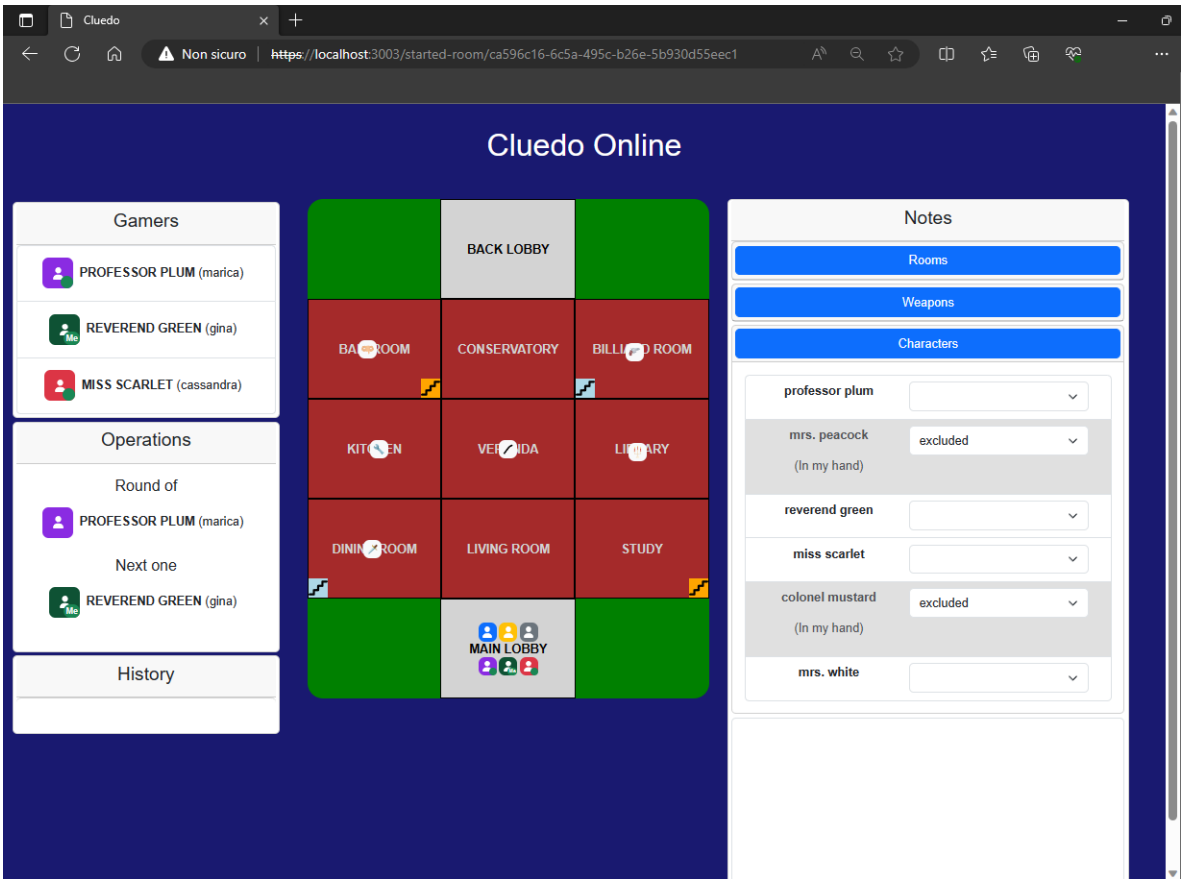


Figura 54: Esempio di schermata di gioco di un giocatore non in turno

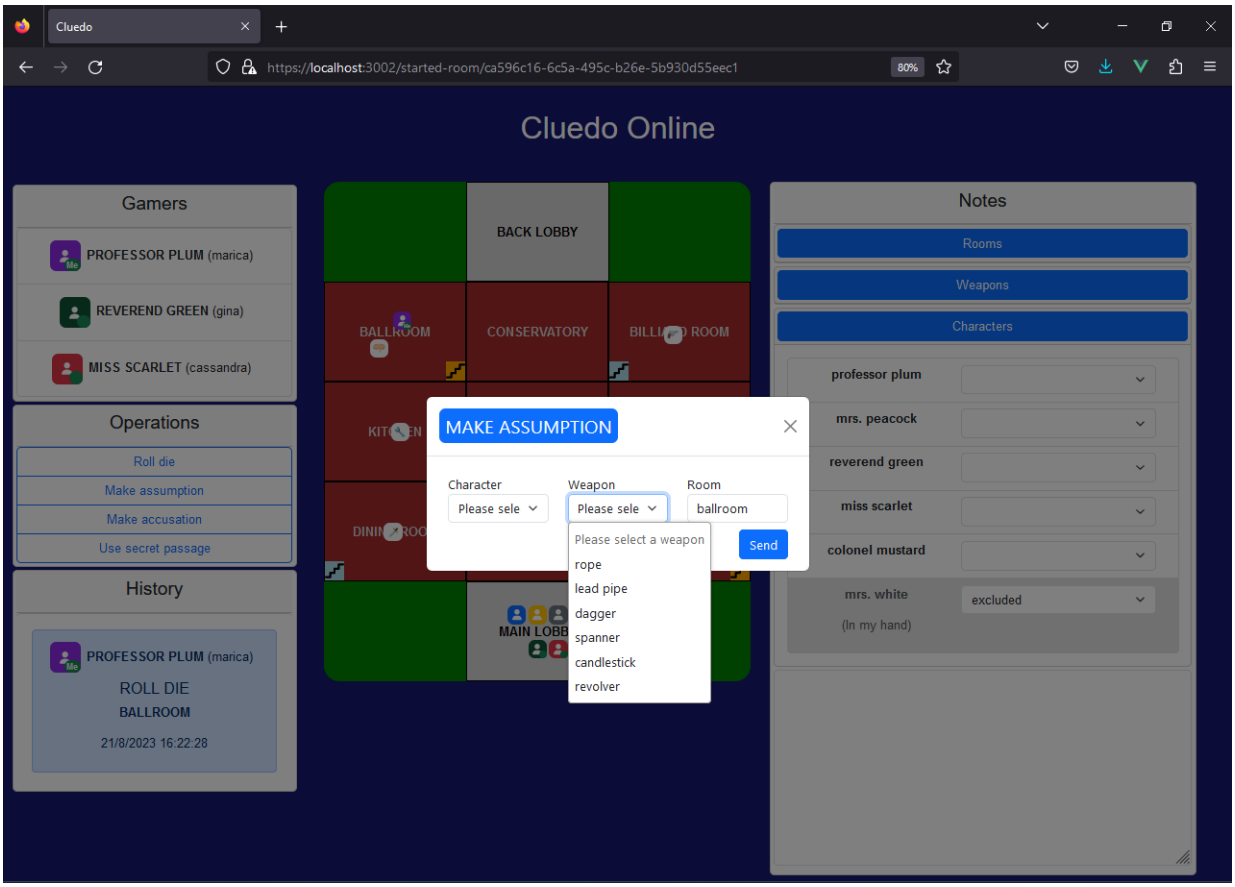


Figura 55: Esempio di “make assumption”

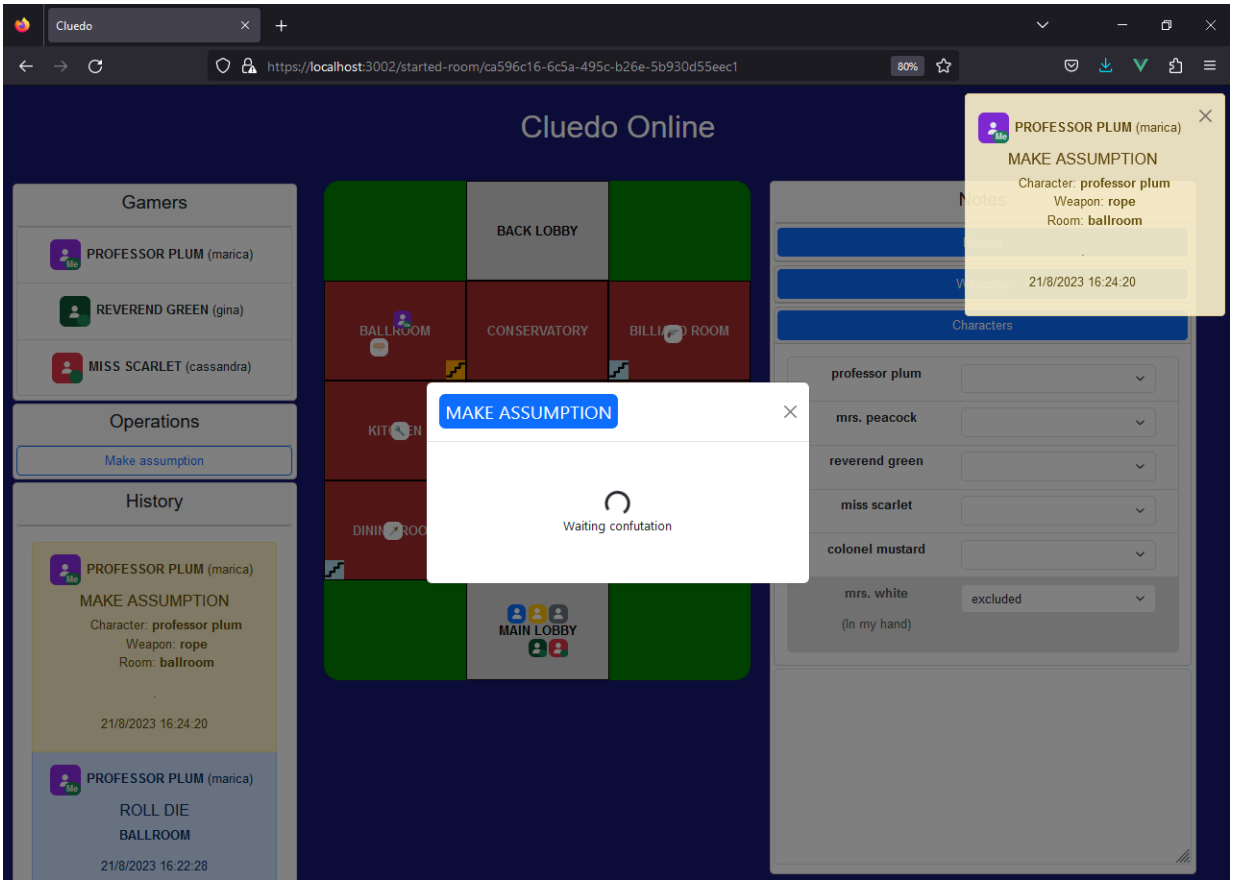


Figura 56: Esempio di attesa di una confutazione

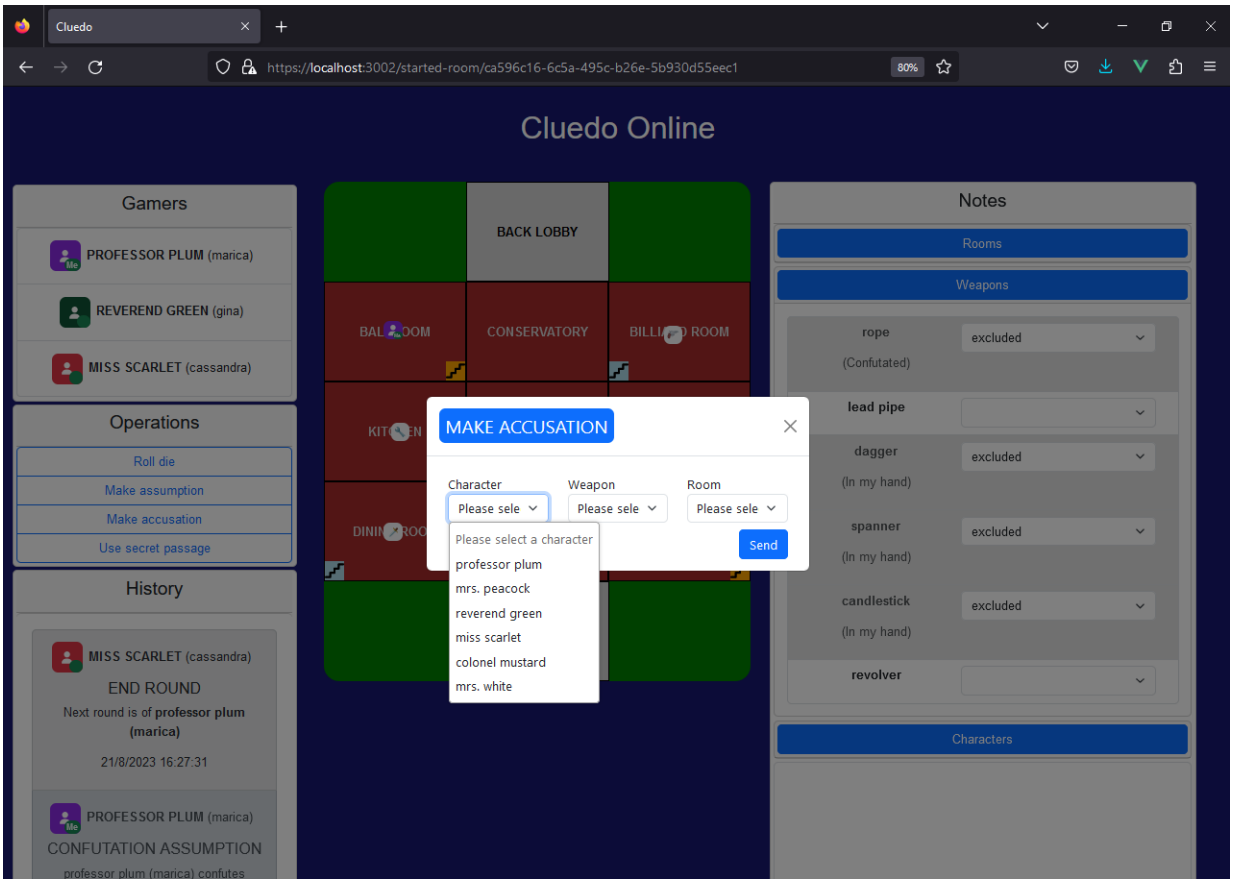


Figura 57: Esempio di “make accusation”

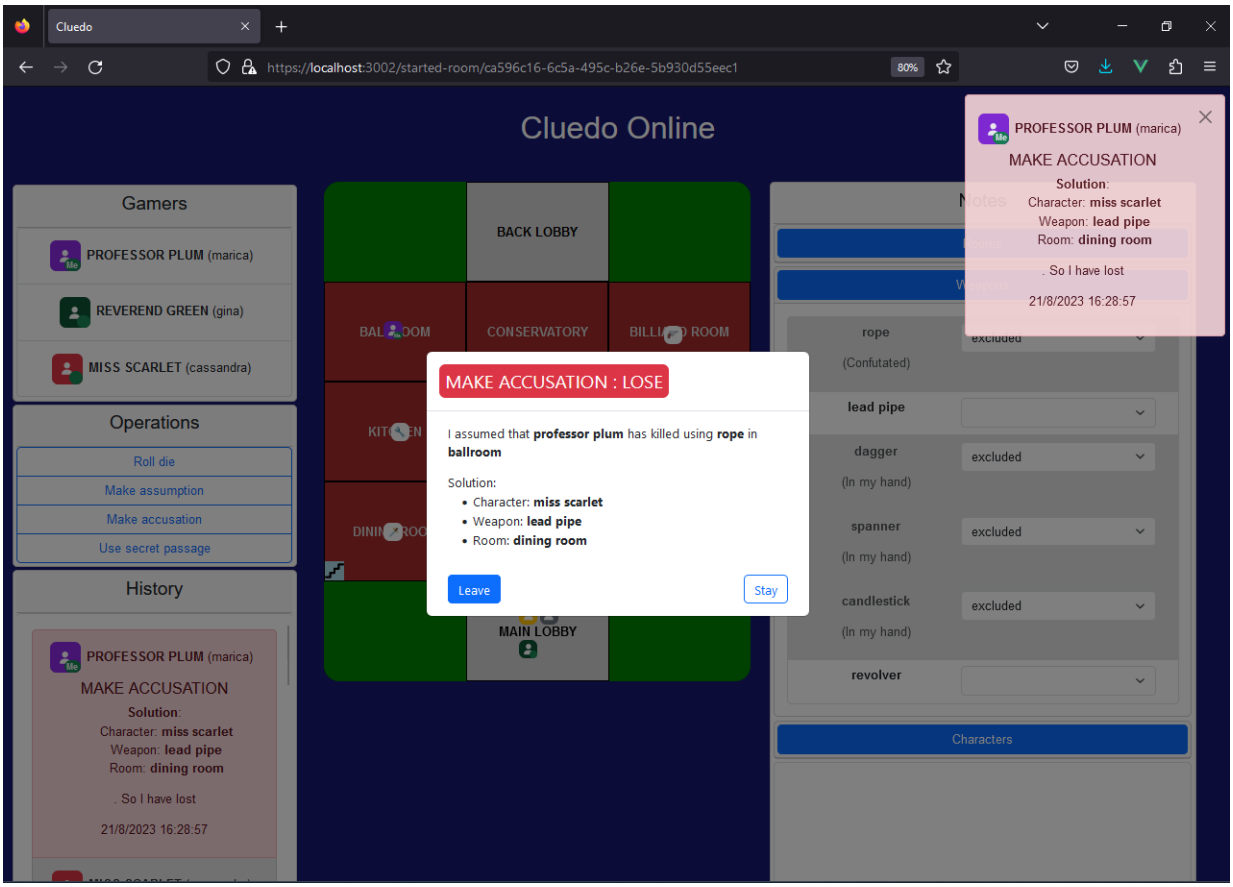


Figura 58: Esempio di un'accusa sbagliata

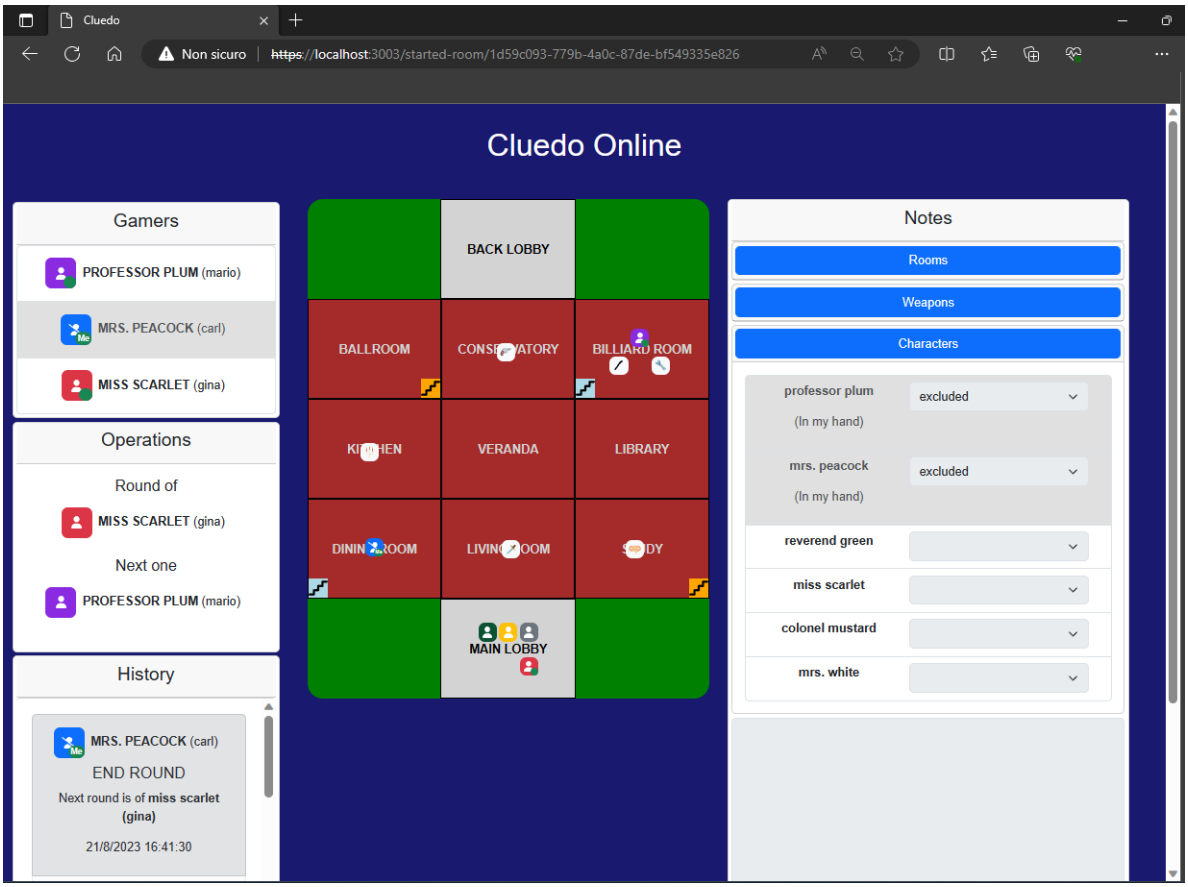


Figura 59: Esempio di schermata di gioco di un giocatore con ruolo *silent*

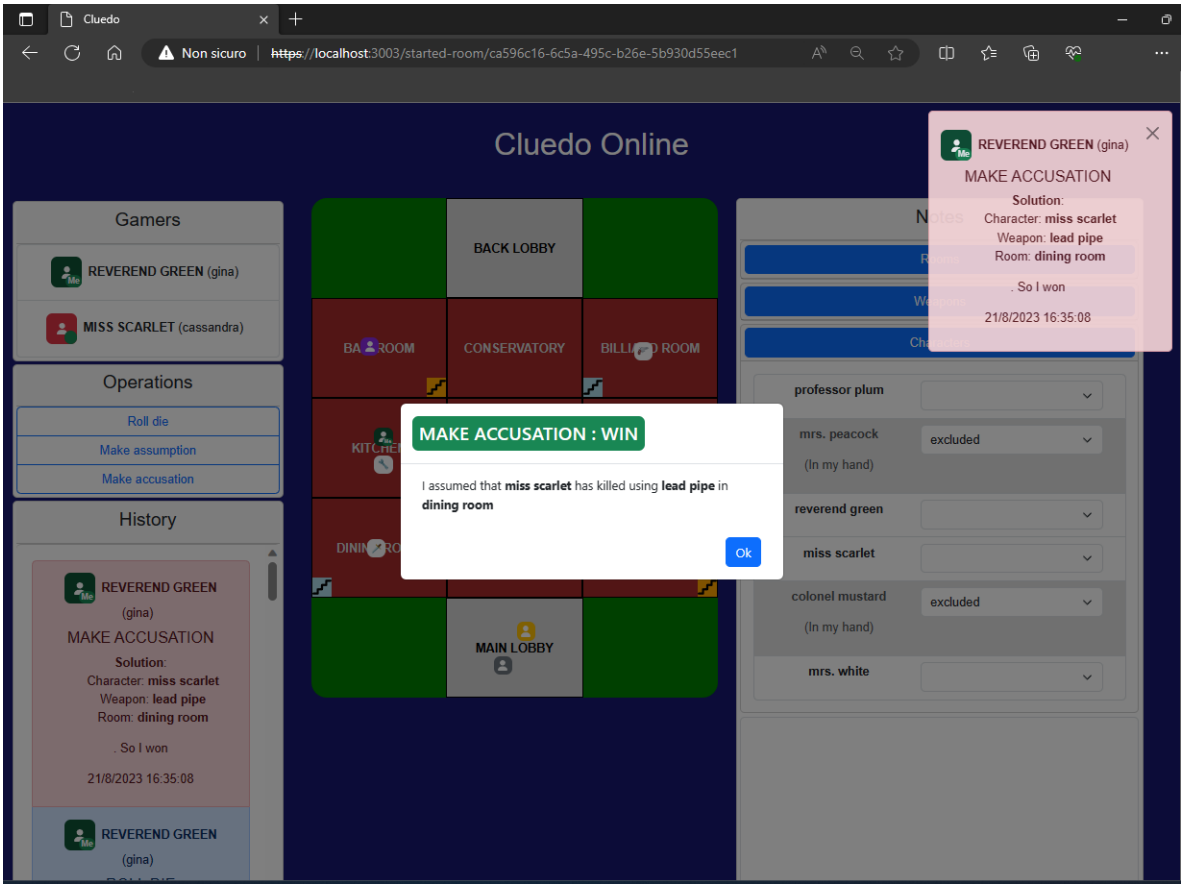


Figura 60: Esempio di un'accusa corretta

8 Conclusioni

Il progetto realizzato mi ha permesso di implementare per la prima volta un sistema realmente distribuito e il risultato ottenuto mi soddisfa. La scelta di implementare una versione distribuita di *Cluedo*, un gioco che presenta una logica semplice, si è rivelata vincente in quanto mi ha permesso di concentrare la maggior parte del tempo sullo studio e gestione di aspetti legati alla programmazione distribuita come, ad esempio, la gestione di connessioni e disconnessioni dei client (peer come client verso il discovery server o verso altri peer o client reali) tramite *Socket.io* e la gestione di uno stato condiviso tra più peer. Scegliendo un'architettura peer-to-peer invece della classica architettura client-server, ho acquisito una migliore comprensione delle difficoltà di progettazione e testing di questo tipo di architettura, dei vari stack tecnologici e sono stata in grado di affrontare alcune delle difficoltà nella creazione di una soluzione, si spera, solida. In conclusione, si ritiene di aver raggiunto un ottimo risultato, considerando anche la sfida affrontata nell'integrare molteplici nuove tecnologie all'interno del progetto.

8.1 Sviluppi futuri

In una versione futura dell'applicazione sarebbe sicuramente necessario un miglioramento dell'interfaccia grafica, la quale al momento risulta volutamente semplice. Si potrebbe inoltre, inserire una chat all'interno della waiting room o durante una partita oppure avviare direttamente una video chiamata. Un altro sviluppo potrebbe essere l'utilizzo intensivo da parte di numerosi utenti allo scopo di verificare l'efficienza del sistema posto sotto stress e confermarne il corretto funzionamento.

Attualmente l'applicazione può essere solo eseguita all'interno di una rete privata, e quindi non è possibile che le applicazioni *peer* e *discovery server* comunichino al di fuori della rete privata; questo si potrebbe risolvere durante uno sviluppo futuro.

8.2 Cosa si è imparato

La realizzazione di questo progetto ha contribuito ad arricchire il bagaglio culturale delle tecniche acquisite durante il corso, dandomi la possibilità di mettere alla prova le mie conoscenze nella realizzazione di funzionalità che coinvolgano tecnologie diversificate nella stessa soluzione. Lo sviluppo di questo software mi ha permesso di comprendere al meglio ed imparare numerosi aspetti riguardanti lo sviluppo di sistemi distribuiti: analizzando le varie architetture di rete e le varie tipologie di comunicazioni che ci possono essere tra client e server e tra peer e peer.
In conclusione, completare questo progetto mi ha aiutato a comprendere meglio il funzionamento di un sistema distribuito. Insieme al corso, mi ha fornito conoscenze concrete che ritengo essenziali per chi opera nel nostro settore e per comprendere argomenti che si sviluppano su questi concetti.

Riferimenti bibliografici

[1] Axios. https://axios-http.com/.

[2] Chai. https://www.chaijs.com/.

[3] Cluedo. https://it.wikipedia.org/wiki/Cluedo.

[4] Docker. https://it.wikipedia.org/wiki/Docker.

[5] Express. https://expressjs.com/.

[6] mocha. https://mochajs.org/.

[7] Mongodb. https://www.mongodb.com.

[8] Mongoose. https://mongoosejs.com.

[9] Node.js. https://nodejs.org/en.

[10] npm. https://docs.npmjs.com/about-npm.

[11] Openapi specification. https://swagger.io/specification/.

[12] Openssl. https://it.wikipedia.org/wiki/OpenSSL.

[13] socket.io. https://socket.io/.

[14] Typescript. https://www.typescriptlang.org.

[15] Vue.js. https://v2.vuejs.org/v2/guide/.