Descrizione UML di rete

Abbiamo deciso di utilizzare due pattern MVC, uno per gestire la Lobby e l'inizializzazione del gioco e uno per gestire le interazioni del gioco. Entrambi i pattern MVC utilizzano la stessa view.

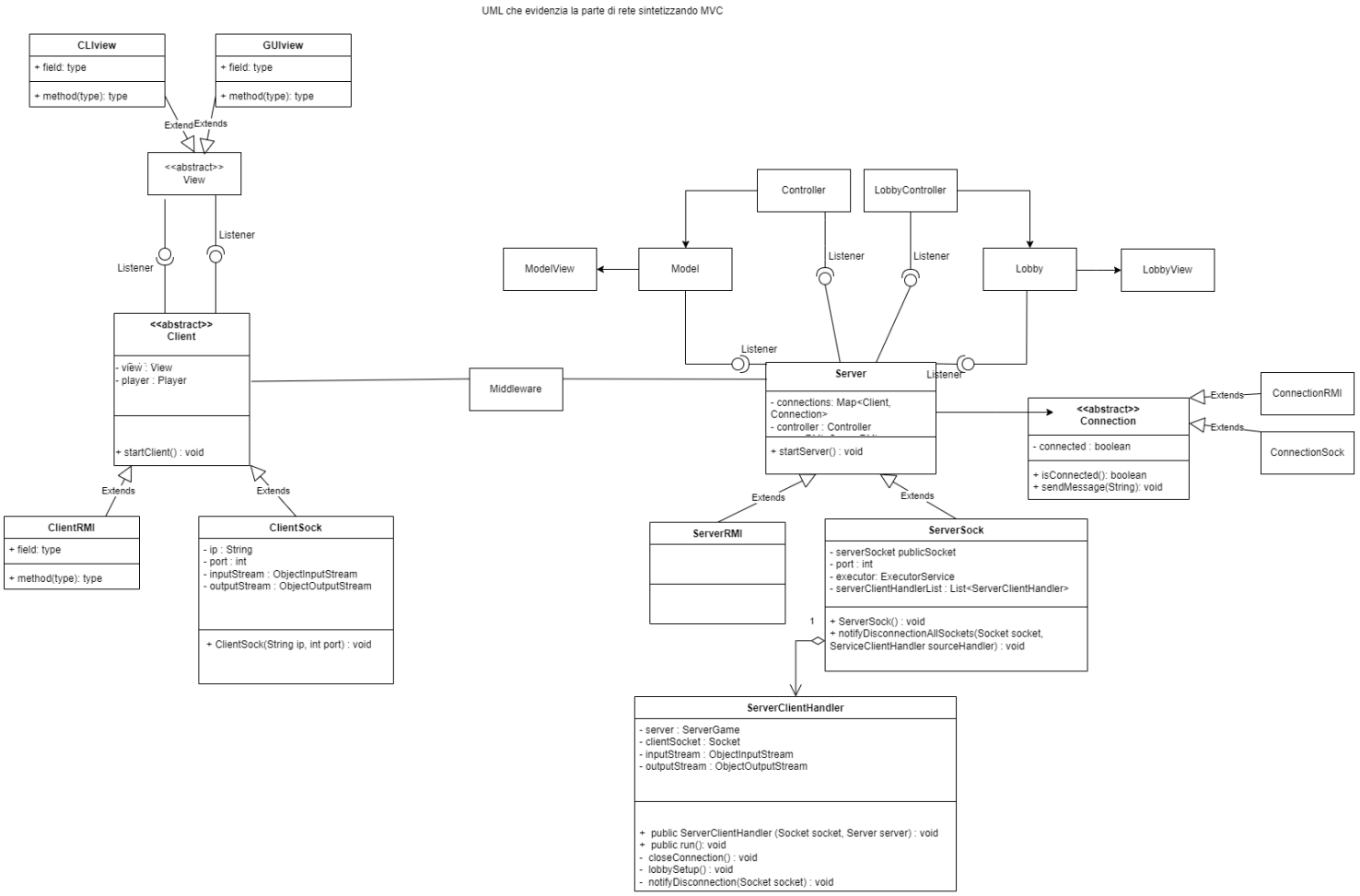
Le classi di rete vengono utilizzate essenzialmente per mandare pacchetti tra client e server contenenti i comandi mandati dai giocatori con i relativi parametri. Questi comandi sono gestiti tramite una classe Choice. Questa classe ha al suo interno un enum di tipo CHOICE_TYPE che rappresenta il comando e una lista di stringhe che contiene tutti i parametri disponibili per il comando selezionato (ad esempio, se si decide di prelevare una tessera dalla board, bisogna passare anche le coordinate della tessera).

Lato client abbiamo utilizzato una classa astratta chiamata Client e due estensioni di essa che implementano rispettivamente il protocollo RMI e il protocollo Socket (Strategy Pattern). Analogamente, lato server Esiste una classa padre (Concreta) Server che viene estesa ancora da due implementazioni diverse in base al protocollo. La classe Server si avvale anche della classe Connection che ancora una volta si estende nelle due rispettive logiche Socket ed RMI. Questa classe server per gestire la tipologia di connessione e utilizzare il giusto protocollo per inviare a ogni client la notifica della GameView. Per quanto riguardo le connessioni socket, viene utilizzata una classe ServerClientHandler per gestire il multithreading e la molteplicità delle connessioni.

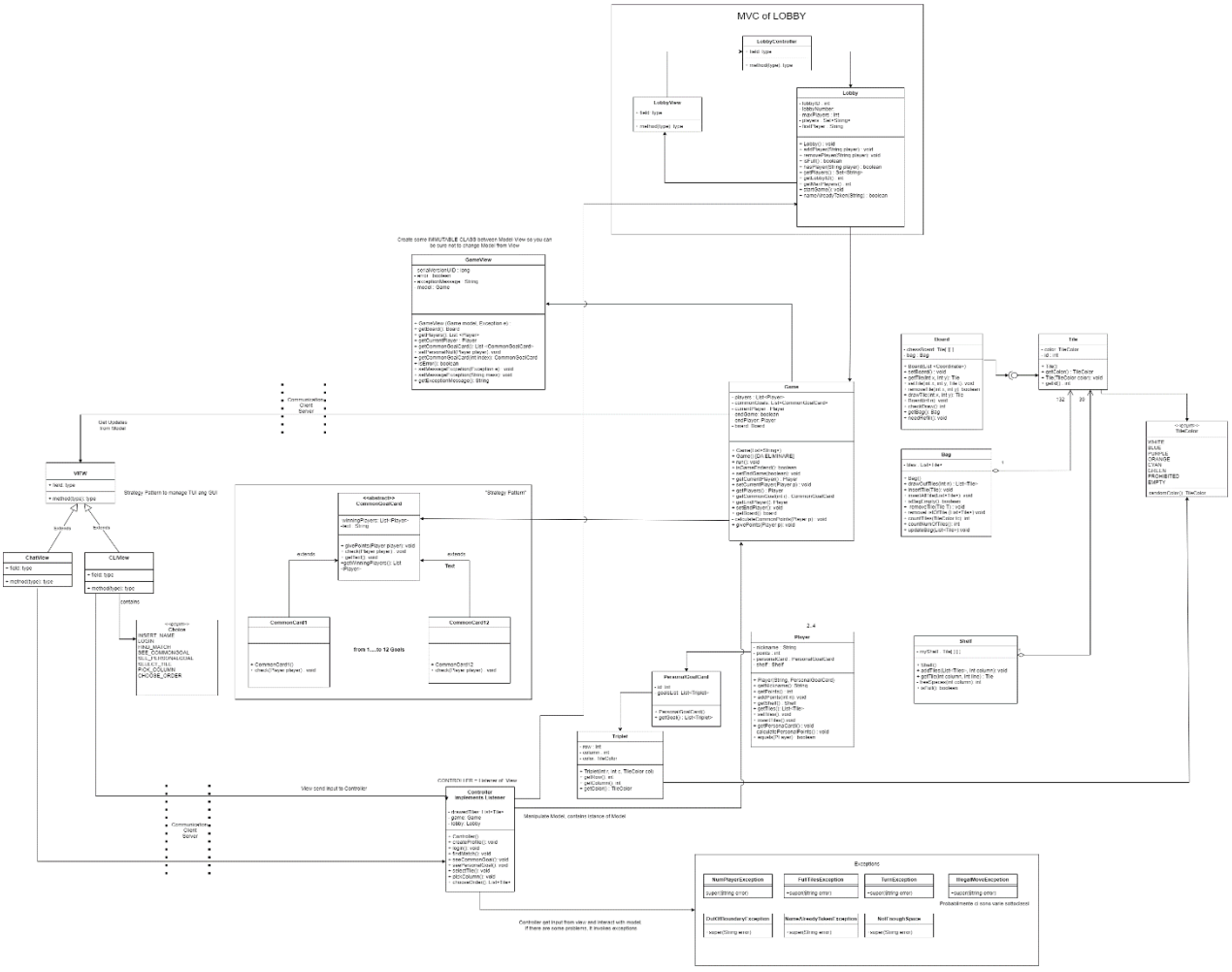
Il client è un listener della view e tutte le volte che la view riceve un input dall'utente il client viene triggerato e invia il messaggio al server. Il server riceve il messaggio e notifica il controller che a sua volta è listener del server. Il controller esegue il controllo dei comandi ed esegue l'operazione apportando le modifiche sul model e gestendo le eventuali eccezioni. Il model, una volta eseguite le modifiche, notifica il server (che è listener del model) e infine il server invia una GameView al client. La GameView contiene un'istanza del Game modificata in seguito all'azione di un player, se l'operazione effettuata è lecita, oppure contiene un messaggio di errore qualora la mossa effettuata non fosse corretta o ci fossero stati problemi a livello di eccezioni. La struttura della GameView rimane la medesima sia in caso di successo che in caso di errore, vengono solamente settati gli attributi interessati. La differenza tra le due tipologie di messaggio viene gestita utilizzano un attributo booleano che segnala la presenza di un errore o meno. Il client, una volta ricevuto il messaggio dal server, lo notifica alla view sempre utilizzando il pattern observer-observable implementato per mezzo dell'interfaccia PropertyChangeListener.

La configurazione iniziale del client, con il setup dei parametri di connessione (indirizzo IP, porta e tipologia di connessione) viene effettuato in maniera locale e sincrona sul client. Una volta effettuato il setup e la connessione, il client manda i messaggi di comando al server e questo risponde.

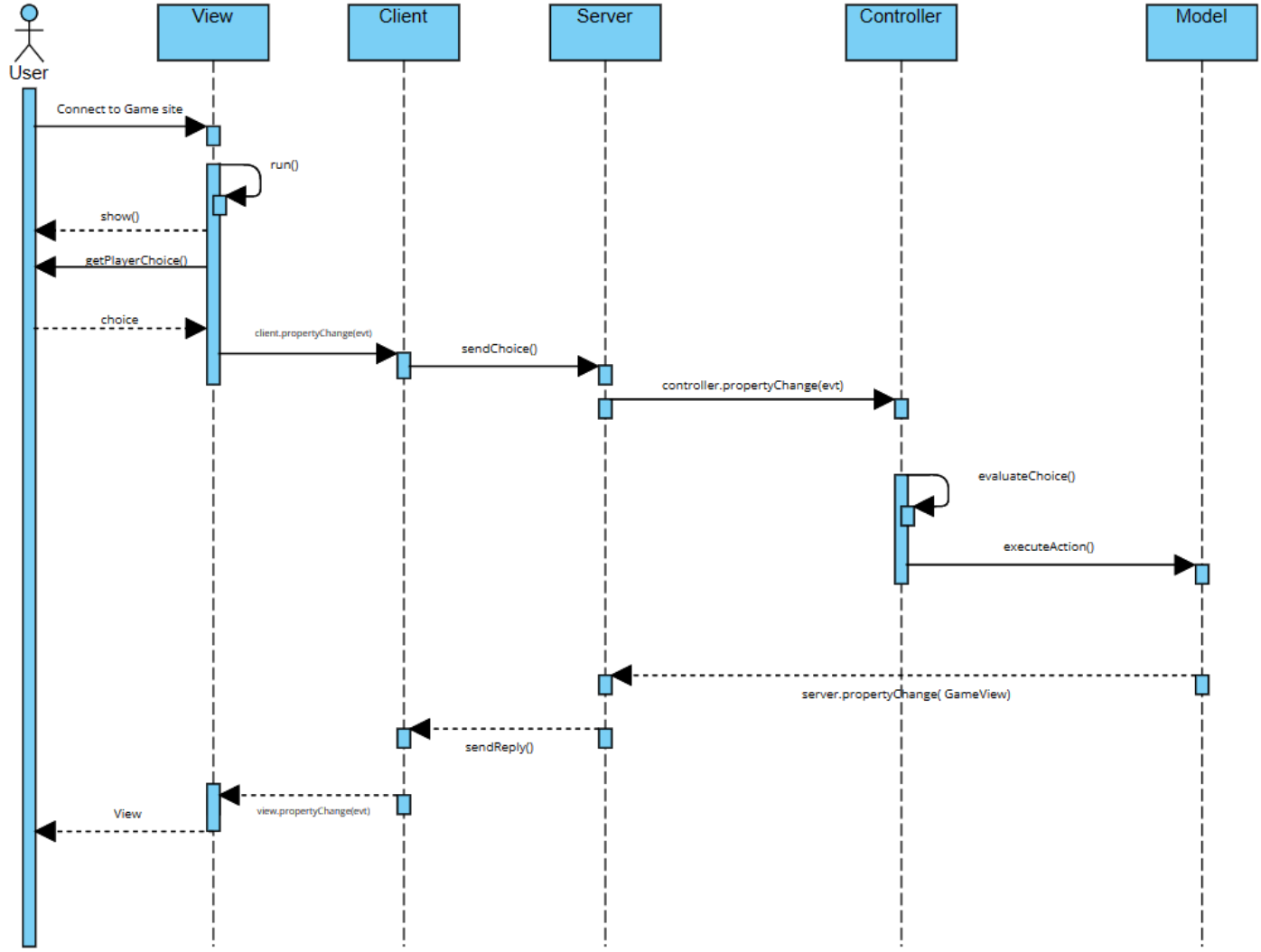
Il parsing delle istruzioni di comando viene effettuato lato client per gestire la costruzione dell'istanza della classe Choice relativa al comando inserito dall'utente. I controlli sulla legalità del comando (numero e tipo di parametri coerenti con il comando selezionato) vengono effettuati sia a lato client che a lato server, rispettivamente dalla view e dal controller. I controlli effettuati su quest'ultimo vengono fatti per garantire maggior robustezza al codice, dare la possibilità di utilizzare un client con diversa natura da quella fornita dal gioco e sono più stringenti poiché accedono ai dati di gioco sul server.



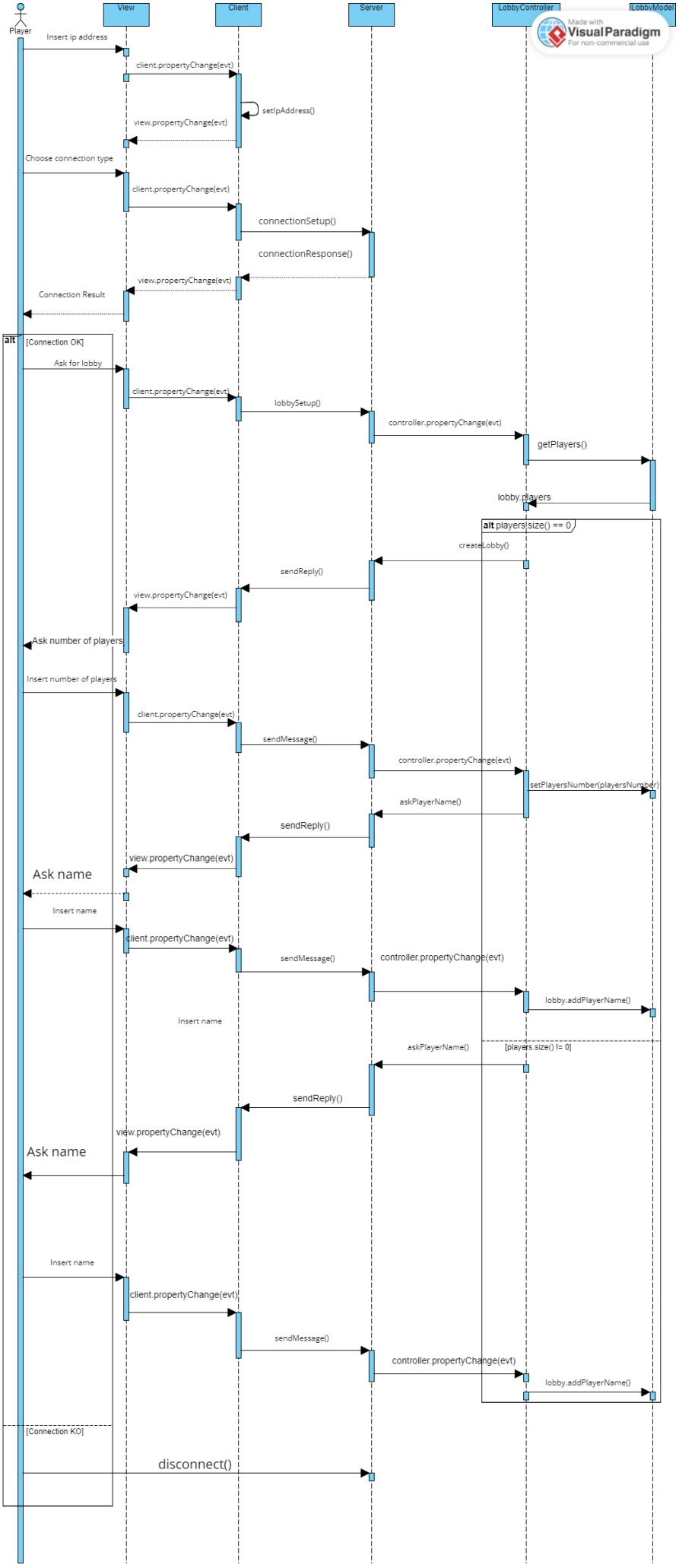
Per quanto riguardo il diagramma UML con il pattern MVC è riportato qui sotto.



Di seguito riportiamo i sequence diagram delle azioni di gioco principali

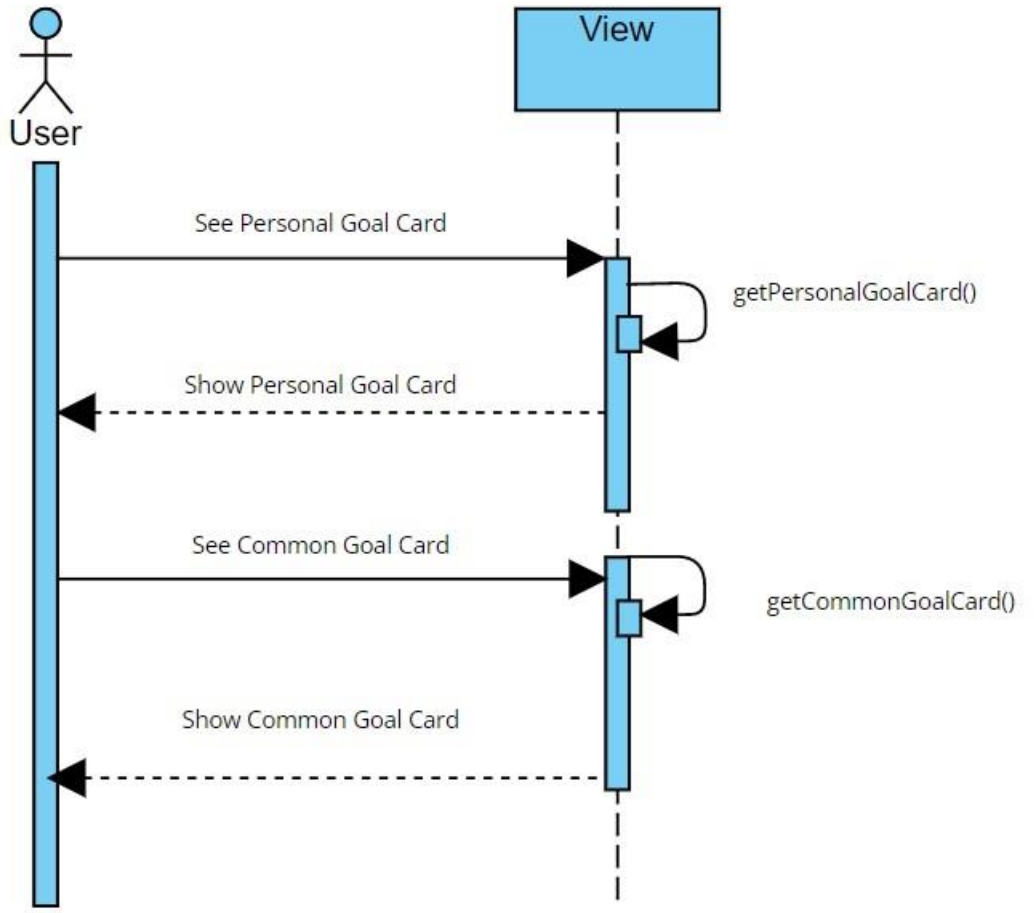


Questo suquence diagram descrive una **generica** azione di gioco come riportato nel paragrafo precedente. La view, dopo aver ricevuto un input dall'utente, scatena il metodo propertyChange() sul client che a sua volta chiama il metodo sendChoice() e manda la Choice al server. Il server chiama l'analogo metodo propertyChange sul controller che eseguirà gli aggiornamenti del model. Dopodiché il model notifica il Server con il metodo propertyChange e a sua volta il server invierà il messaggio al client. Una volta ricevuto il messaggio, il client notificherà la view sempre con il metodo propertyChange. Qualora ci fosse un errore lato server (mossa illegale) il flusso non cambia poiché l'informazione sull'errore viene inserita all'interno della GameView.

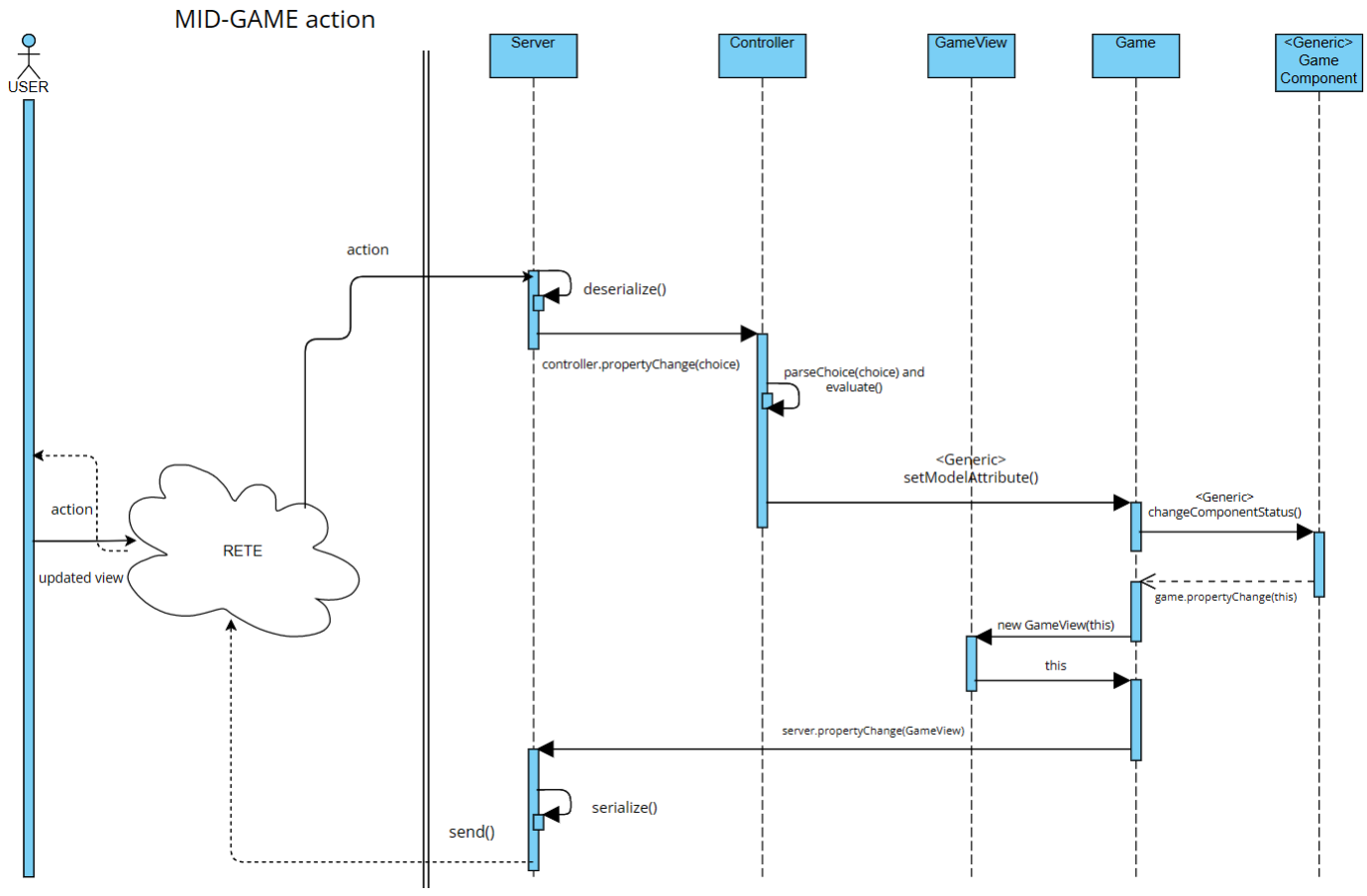


Questo diagramma rappresenta invece la sequenza iniziale di setup della connessione e creazione della lobby.

Quando un utente avvia il gioco, inserisce l'indirizzo IP del server. Successivamente sceglie il tipo di connessione ed effettua il setup della stessa contattando il server. A questo punto, se la connessione non è andata a buon fine, l'utente effettua la disconnessione e può riprovare a connettersi. Se invece la connessione è effettuata, si procede con la creazione o sottoscrizione alla lobby. Il server notifica il LobbyController che verifica quanti giocatori sono iscritti alla lobby. Se nessun giocatore è ancora iscritto, si procede con la creazione. L'utente inserisce il numero di giocatori e successivamente il proprio nickname venendo messo come giocatore principale. Se invece il numero di giocatori è almeno uno e minore del numero di giocatori massimo scelto dal primo giocatore, l'utente procede solamente con l'aggiunta del proprio nickname.



Questo sequence diagram descrive una **generica** interazione tra user e view.



Anche in questo diagramma è descritta una generica interazione però lato server. Avviene quanto descritto sopra, ovvero la notifica ai listener di tutte le componenti implementate sul server.