Prova finale di Ingegneria del Software 2024/2026

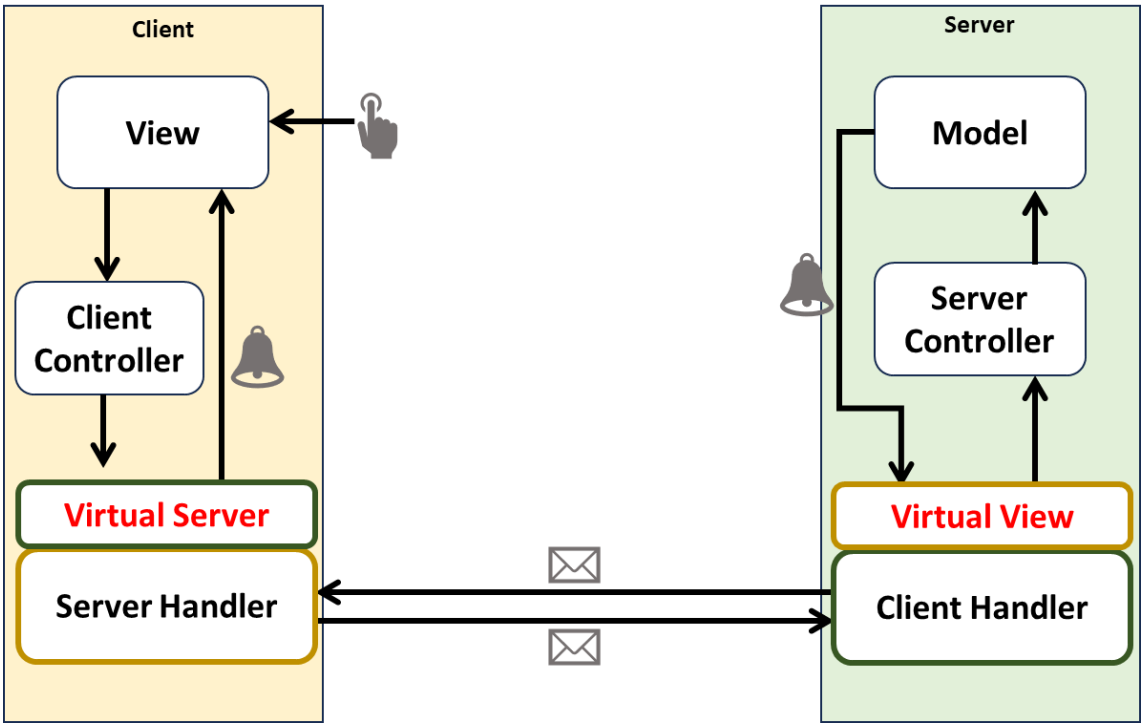
PROTOCOLLO DI RETE

IS25-AM02

- Crimella Gabriele
- Cordioli Anna
- Donno Erica
- Faccini Davide

Architettura della soluzione

Abbiamo progettato una soluzione **"hybrid client"**, in cui il **Controller** è presente sia sul **Server** che sul **Client**. Questo approccio consente al Client di eseguire alcune operazioni localmente, riducendo il numero di richieste al server e migliorando l'efficienza complessiva del sistema. (Nell'immagine è presente un esempio di comunicazione tramite socket)



Comunicazione tra Client e Server

Comandi inviati dalla View al Controller

Il controller lato server si interfaccia con il client con due tipologie di metodo:

- Inizializzazione del gioco e creazione partite
  - Verifica disponibilità username: Controlla l'unicità del nome scelto dal giocatore.
  - Creazione di una nuova lobby: Un giocatore può creare una lobby e specificare il numero massimo di partecipanti.
  - Ingresso in una lobby esistente: Un giocatore può unirsi a una partita già creata.
  - Avvio della partita: Il server inizializza la sessione di gioco quando la lobby raggiunge il numero di giocatori richiesto.

- Fasi di gioco
  Il controller presenta inoltre i metodi per poter gestire le fasi di gioco. I metodi che implementa sono quelli contenuti nell'interfaccia Game_controller, con l'unica aggiunga che, in alcuni di essi, nel model sono presenti senza parametri (in quanto non richiesti, come la richiesta dei vincitori) e nel controller sono richiesti con l'ID della lobby appartenente al gioco. Il ruolo del ServerController, in questo caso, è quello di trovare il giusto gioco a cui il player fa riferimento e, conseguentemente, invocare tale metodo sul "model" corretto.

Abbiamo gestito i casi di concorrenza dei giochi e delle strutture dati presenti nel controller, usando strutture dati dinamiche appropriate.

```
private final Map<Integer, Lobby> lobbies = new ConcurrentHashMap<>();
private final Map<Integer, GameSession> activeGames = new ConcurrentHashMap<>();
```

Update del Model alla View

In accordo col pattern MVC, abbiamo scelto di inserire nell'interfaccia del model solo metodi void in quanto interpretiamo il model come un componente reattivo, al quale i client si possono mettere in ascolto in attesa di aggiornamenti per la view.

Ogni metodo del model, quindi, genererà eventi che interesseranno i giocatori presenti in una specifica partita.

Fasi di gioco

Abbiamo racchiuso le fasi in cui si può trovare il **gioco** in un'enumeration presente nel model. Essa contiene i seguenti valori:

BUILD, //the spaceship is being built
CHECK, //the user can only call checkSpaceship
CORRECTION, //if the check phase went wrong, the user can correct the spaceship with the remove method. at the end of correction, if the spaceship is correct, the next sstate will be the inizializazion one, else ...
INITIALIZATION_SPACESHIP, //the user can add the crew. the battery tile is filled.
TAKE_CARD, //the user can call nextCard. if the deck is empty, the next state will be the result.
EFFECT_ON_PLAYER,
RESULT

Il giocatore, invece, può trovarsi nelle seguenti fasi durante la partita:

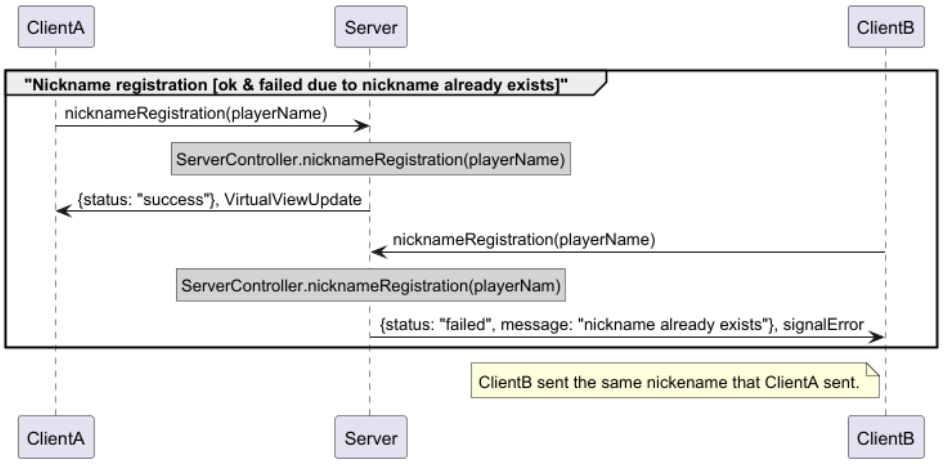
FINISHED,
NOT_FINISHED,
CORRECT_SHIP,
WRONG_SHIP,
IN_GAME,
OUT_GAME

Il model, al suo interno, gestisce gli stati verificando che le invocazioni dei metodi da parte del client siano consistenti con lo stato in cui il gioco si trova in un dato istante. Qualora si verificasse un'incongruenza, il model non eseguirà il metodo e verrà eseguita una ViewUpdate ai client direttamente interessati.

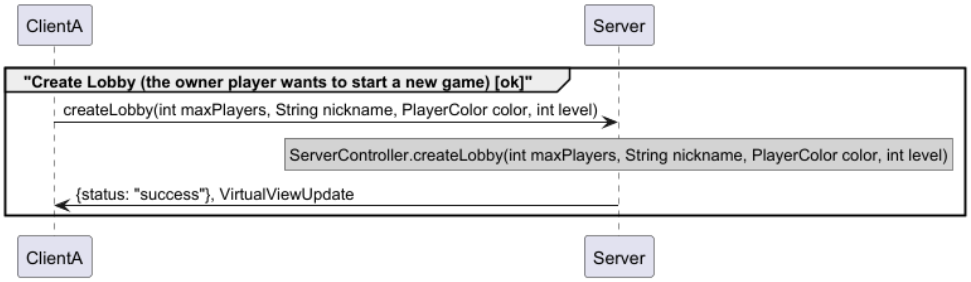
Descrizione schematica delle interazioni Client-Server

Di seguito si riportano alcuni esempi di interazione tra il client e il server (ServerController). Non è stato effettuato un diagramma per ogni possibile interazione, ma si è cercato di raggruppare queste ultime per "categoria e comportamento", facendo notare che l'interazione è simile per molti metodi.

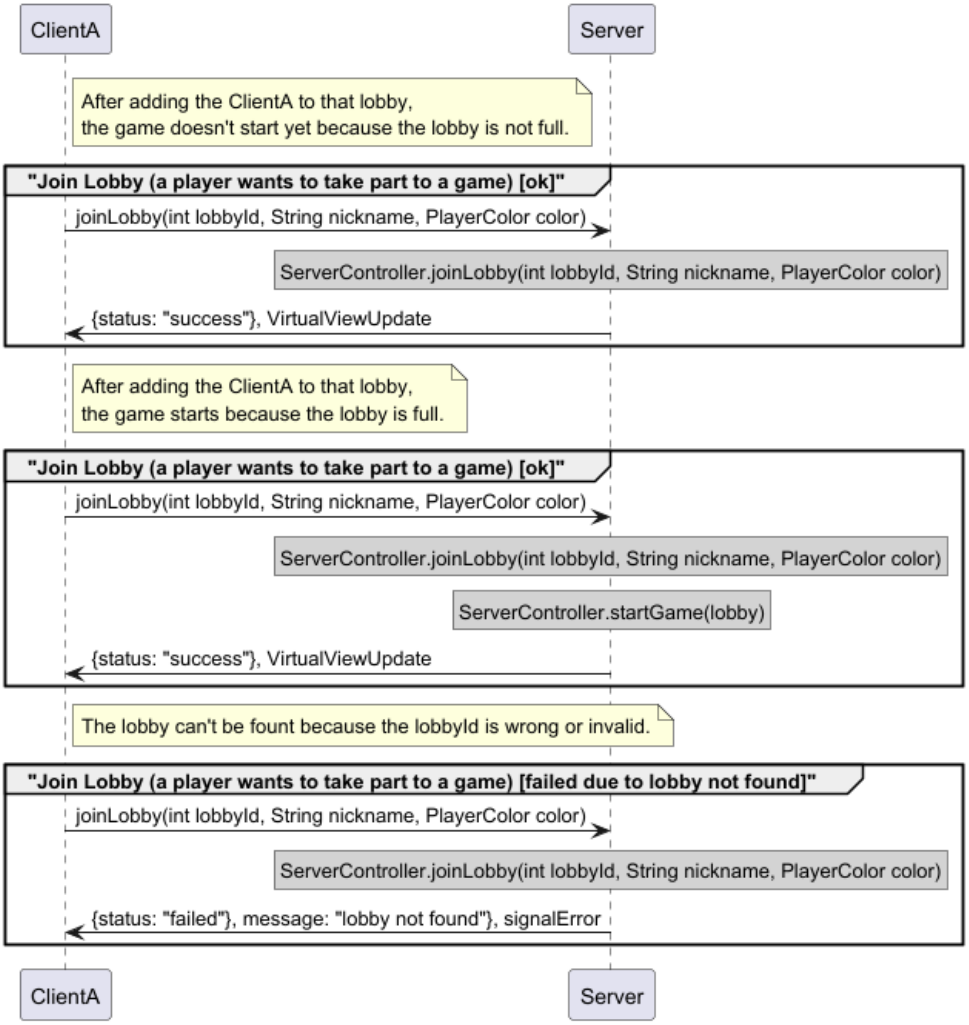
nicknameRegistration



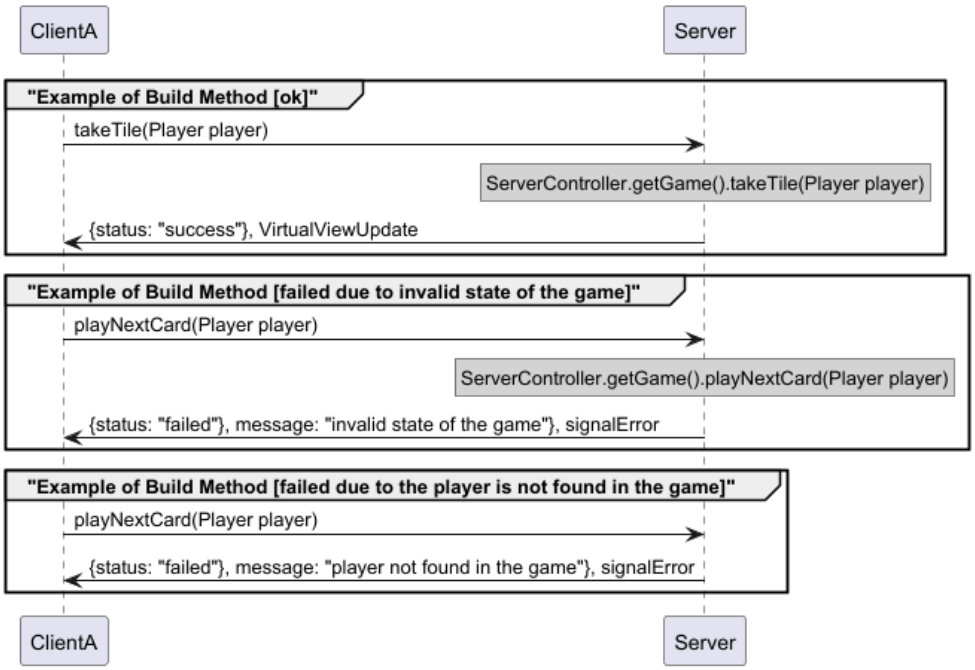
createLobby



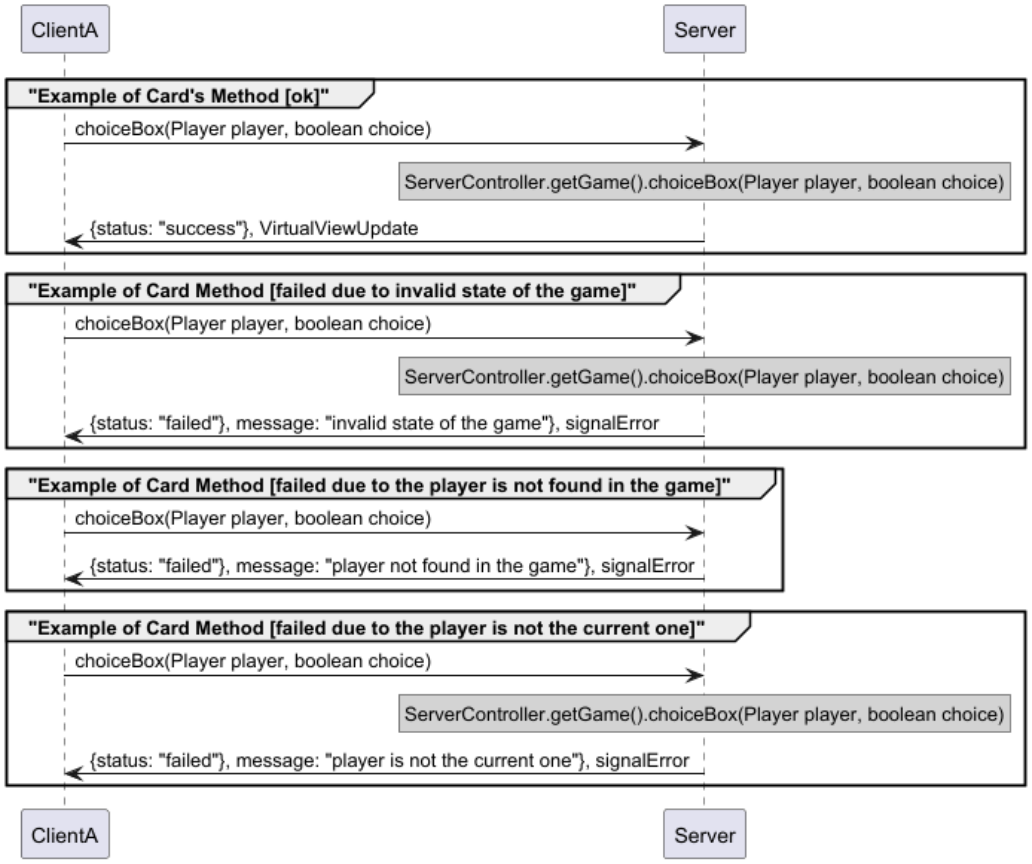
joinLobby


ClientA
Server
After adding the ClientA to that lobby,
the game doesn't start yet because the lobby is not full.
"Join Lobby (a player wants to take part to a game) [ok]"
joinLobby(int lobbyId, String nickname, PlayerColor color)
ServerController.joinLobby(int lobbyId, String nickname, PlayerColor color)
{status: "success"}, VirtualViewUpdate
After adding the ClientA to that lobby,
the game starts because the lobby is full.
"Join Lobby (a player wants to take part to a game) [ok]"
joinLobby(int lobbyId, String nickname, PlayerColor color)
ServerController.joinLobby(int lobbyId, String nickname, PlayerColor color)
ServerController.startGame(lobby)
{status: "success"}, VirtualViewUpdate
The lobby can't be fount because the lobbyId is wrong or invalid.
"Join Lobby (a player wants to take part to a game) [failed due to lobby not found]"
joinLobby(int lobbyId, String nickname, PlayerColor color)
ServerController.joinLobby(int lobbyId, String nickname, PlayerColor color)
{status: "failed"}, message: "lobby not found"}, signalError
ClientA
Server

BUILD PHASE (just an example…)



CARD PHASE (just an example…)



Protocolli Sviluppati

Nel nostro progetto abbiamo deciso di implementare entrambi i protocolli di comunicazione di rete: RMI e Socket. Le due soluzioni presentano differenti livelli di astrazione e complessità implementativa.

Nel caso dell'RMI (Remote Method Invocation), le chiamate remote vengono effettuate direttamente verso il ServerController, sfruttando l'astrazione offerta dal framework stesso, che gestisce automaticamente la serializzazione e il trasporto dei dati.

Per quanto riguarda il protocollo Socket, invece, abbiamo adottato una struttura più articolata: le comunicazioni sono gestite tramite specifici Handler, responsabili della gestione degli stream di input/output, della serializzazione e deserializzazione dei dati e del riconoscimento dei comandi ricevuti. Tali Handler si occupano quindi di interpretare i messaggi e inoltrarli ai componenti logici appropriati all'interno del server.

È importante sottolineare che, una volta che le richieste client raggiungono il server, entrambi i protocolli convergono verso un'interfaccia comune rappresentata dal ServerController. Tuttavia, la gestione delle chiamate di update della view lato client è stata trattata in modo differente: abbiamo ritenuto opportuno, considerando che RMI si basa internamente su socket, che fosse il ServerHandler a identificare il metodo da invocare e a delegare l'effettiva esecuzione al NetworkController. Quest'ultimo, grazie all'implementazione del protocollo RMI e dei relativi metodi di update, è in grado di notificare in modo efficiente il client.

Questa scelta progettuale consente di:

1) Mantenere una chiara separazione delle responsabilità tra i componenti;
2) Evitare duplicazioni di codice lato client per la gestione degli stessi metodi;
3) Facilitare l'estensibilità e la manutenzione del sistema.

A seguire, è riportato uno schema esplicativo dell'intero protocollo di rete, con particolare attenzione alle differenze nei livelli di astrazione tra le due tecnologie adottate.