GESTIONE DELLA COMUNICAZIONE RMI NEL PROGETTO CLIENT – SERVER

1. INTRODUZIONE

Per garantire una comunicazione efficace nel progetto *Galaxy Trucker*, è stato adottato il protocollo di comunicazione di tipo **RMI** (*Remote Method Invocation*). Questo protocollo consente uno scambio di informazioni preciso e permette di gestire in modo strutturato le interazioni tra client e server.

Ogni messaggio scambiato è associato ad un payload, ovvero un contenuto informativo che descrive l'azione da eseguire o lo stato da aggiornare

2. FLUSSO DELLA COMUNICAZIONE

2.1 REGISTRAZIONE DEL CLIENT

Il processo inizia con l'inserimento dell'IP e dell'username da parte di un client tramite **CLI** (*Command Line Interface*). L'username del giocatore viene memorizzato nel Model del client nella classe ***ClientGame*** *per il modello di gioco e nella classe* ***RMIServerHandler*** *per la gestione delle connessioni***.**

Successivamente, viene attivata la classe ***CommandNotifier***, un'interfaccia che funge da publisher degli input ricevuti dalla CLI. La sua funzione è inoltrare i comandi alla classe ***UIManager***, incaricata della gestione dell'interfaccia utente lato client. Quest'ultima, dopo aver ricevuto il comando, lo elabora e lo trasmette al server.
L'invio del comando avviene tramite il metodo *commandNotifier.sendCommand(...)*, che utilizza un valore dell'enumerazione ***CommandEnum*** per identificare in modo univoco l'azione richiesta.

L'UIManager ascolta il comando (tramite *listen*) e, in base al tipo, lo prepara aggiungendo eventuali argomenti necessari, per poi inoltrarlo al server utilizzando il metodo *invoke( )* che permette di effettuare l'invocazione remota (RMI).
(per esempio, *RMIClient.tryDrawCard( )* invoca *tryDrawCard( )* sul server)

Nel caso in cui la request abbia lo scopo di registrare un client (*setUsername*), viene effettuata l'iscrizione del client all'interno dell'implementazione del server: la classe ***RMIServerHandler***. Essa utilizza la classe ***MessageController*** per validare il messaggio e gestire la logica di gioco.

Il server implementa l'interfaccia ***RemoteServer*** e salva una mappa con i ***RemoteClient****, che è l'interfaccia implementata dal client, connessi al server*. Se la richiesta è valida, il ***MessageController*** invoca il Controller, rappresentato dalla classe ***GameController***, che interagirà direttamente con il Model.

La risposta viene generata tramite la classe ***PayloadCreator***, restituita all'RMIServerHandler e quindi viene inviata a tutti i client, con il metodo *broadcast*( ).

Il server registra il riferimento remoto al client e associa l'username alla rispettiva entry nella variabile *clients* di tipo *Map <String, RemoteClient>.*

2.2 GESTIONE DELLA RESPONSE LATO CLIENT

Il client riceve la Response all'interno dell'RMIClient, che tramite un ***listener*** la reinvia all'UIManager. Quest'ultimo la inserisce in una **coda** per garantirne l'elaborazione ordinata anche in presenza di risposte multiple.

La risposta viene processata tramite il metodo *processResponse( )* nella classe UIManager e il ClientGame verrà aggiornato di conseguenza. Al termine dell'aggiornamento, la CLI aggiorna l'interfaccia utente ponendosi in attesa di un nuovo input.

2.3 GESTIONE DEI FALLIMENTI

In caso di errore durante l'elaborazione della richiesta, ad esempio se non sono rispettati dei vincoli di gioco, il MessageController genera una Response che riporta il *flag success = false, e l'enumeration ErrorType, con il tipo di errore che si è verificato*.

Questa risposta, a differenza di quelle che vanno a buon fine, viene inviata solo al client che ha effettuato la richiesta fallita.
In questo caso, il Model del client (ClientGame) non viene aggiornato e la CLI visualizza un messaggio di errore e consente di inserire nuovamente il comando.

3. CASI DI UTILIZZO DEL PROTOCOLLO RMI

Vengono in seguito mostrati i seguenti casi:

- Gestione del collegamento e delle disconnessioni
- Esempio di Request e di Response
- Funzionamento della Response lato server

3.1 COLLEGAMENTO E DISCONNESSIONE

Prevede i seguenti punti:

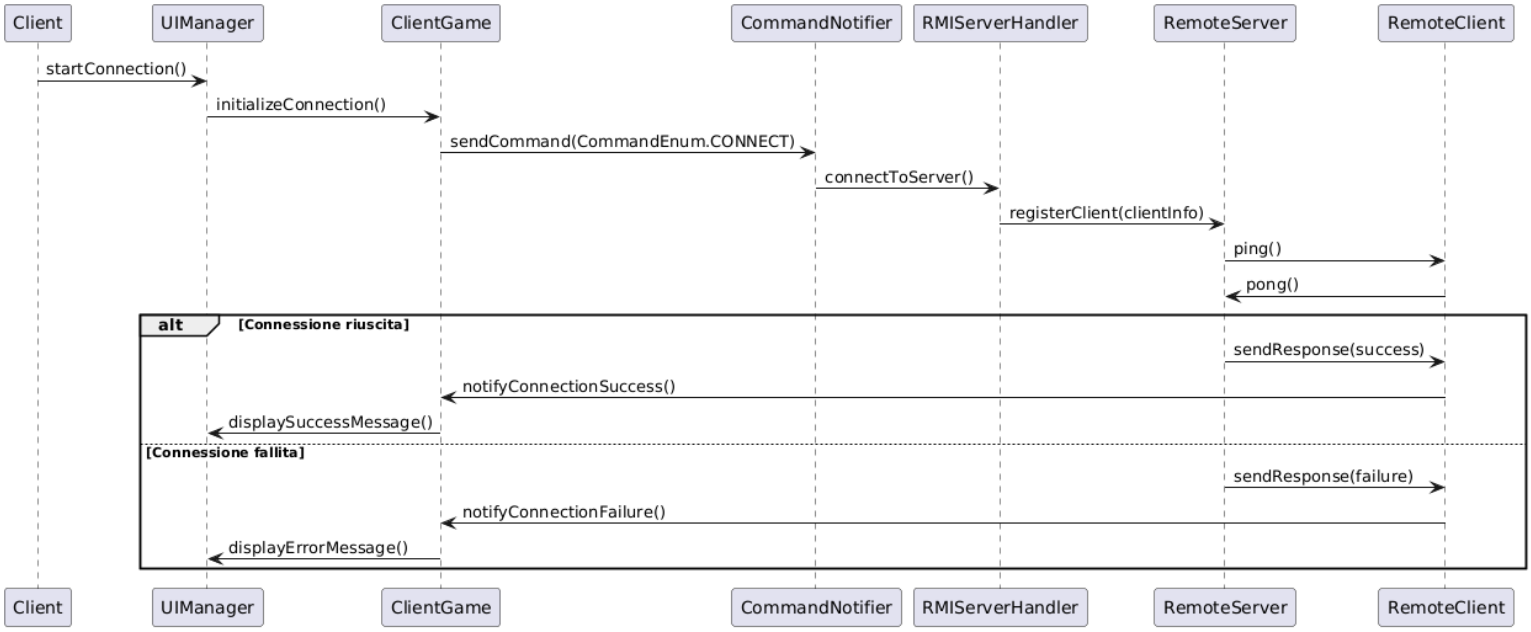
1) **Inserimento IP e nickname** da parte del client, per collegarsi al server.
2) **Meccanismo di ping/pong**:
    - Il client invia il ping e si pone in attesa di un pong
    - se non riceve risposta, vengono inviati fino a tre nuovi tentativi a distanza di 15 secondi
    - se anche il terzo fallisce, il client verrà disconnesso e rimosso dalla lista dei giocatori.

    In generale, tale meccanismo funziona grazie a un thread in esecuzione continua.
3) Disconnessione immediata di un client se si fallisce nell'invio di una Response di tipo broadcast dal server.

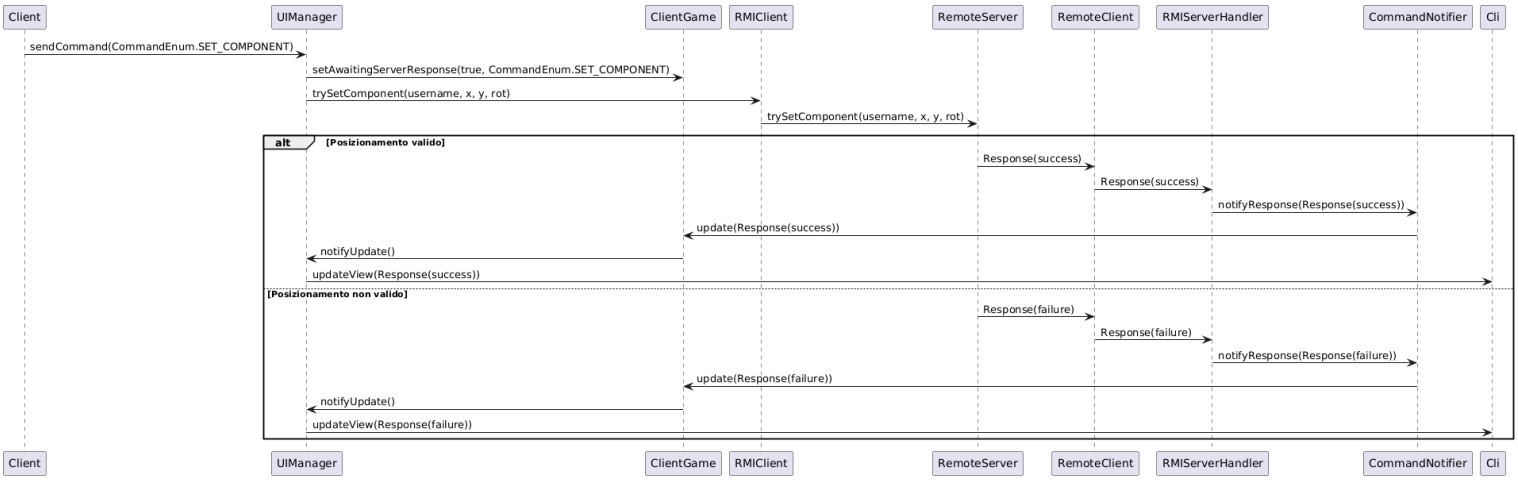
3.2 ESEMPI DI REQUEST E RESPONSE – casi rappresentativi di interazioni client - server

In particolare, analizziamo i seguenti casi rappresentativi:

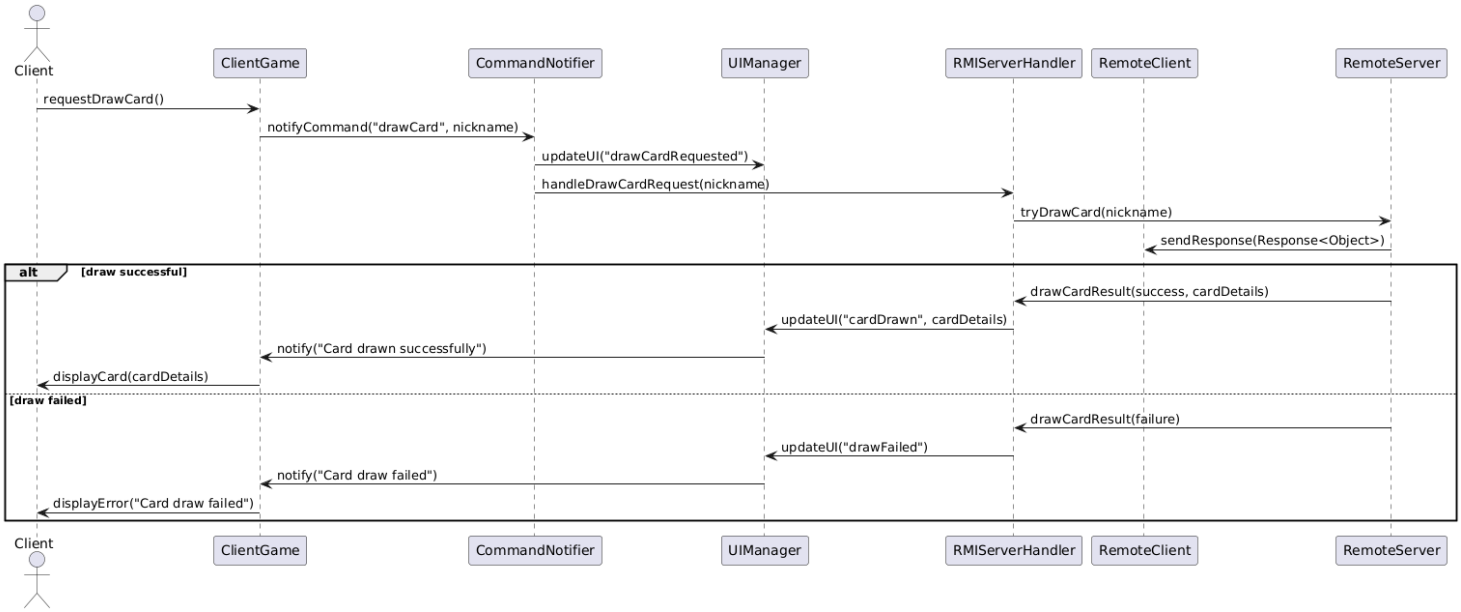
1) Inserimento del nickname e successiva registrazione del client (**input di un nickname**)



2) Tentativo di posizionamento di un componente sulla shipboard (**input di coordinate**)



3) Estrazione della prima carta del mazzo (nessun input, **solo click 'INVIO'**)



2.3 STRUTTURA DI UNA RESPONSE

Ogni Response include:

- Un flag ***success*** (boolean) che indica l'esito dell'operazione. Esso è determinato dai controlli nella classe MessageController
- Un ***payload***, il contenuto variabile della risposta, a seconda del comando (CommandEnum), o dall'evento che l'ha generata.
- Un eventuale ***ErrorType***, in caso di errore

I tipi di payload delle Response sono:

- *List <String>* — lista di username
- *List <ClientShipboard>* — le shipboard di tutti i giocatori
- *Integer*
- *ClientShipboard* — una singola shipboard
- *List <ClientComponent>* — lista di componenti della shipboard
- *ClientComponent* — un singolo componente della shipboard
- *List <List <ClientCell >>* — parti di una shipboard
- *MapShipCrew*
- *ClientPlayersFlightboardCard*
- *ClientGame* — il game, utile per il calcolo dei crediti
- *ClientBoards*
- *ClientShipboardDecks*
- *Map <String, Integer>*

Infine, i tipi di Response che vengono essere mandate ai client sono i seguenti:

- **AZIONATA DA UNA REQUEST**: è la risposta tipica in cui il client invia una richiesta esplicita ed il server risponde. Include sempre l'username del giocatore che effettua la richiesta.
- **AZIONATA DA UN LISTENER OPPURE DA UN EVENTO ASINCRONO**: è inviata a prescindere dalla richiesta da parte del client, ma in seguito ad un cambiamento di stato. In questo caso, il campo dell'username viene lasciato vuoto.
  Questi eventi sono:
    - ***GameStatusChange***: cambio nello stato di gioco
    - ***CardPhaseChange***: avanzamento nella fase della carta
    - ***CardEffectFinish***: termine dell'effetto di una carta
    - ***PlayerDisconnected***: disconnessione di un giocatore
    - Frammentazione della nave

4. CONCLUSIONI

La comunicazione client-server del progetto è stata strutturata secondo l'architettura MVC, con l'ausilio del protocollo RMI per l'invocazione remota dei metodi. Il flusso dei dati è stato gestito in maniera modulare e responsabile, attraverso classi ben definite come *CommandNotifier*, *UIManager*, *RMIClient*, *RMIServerHandler* e *MessageController*, ciascuna con un ruolo preciso.

I messaggi scambiati sono veicolati da oggetti Response dotati di payload informativi e gestiti tramite una coda asincrona, garantendo robustezza anche in presenza di eventi concorrenti. Le situazioni di errore sono gestite in modo isolato, senza compromettere la stabilità del sistema per gli altri client.
L'implementazione di meccanismi di ping e disconnessione automatica assicura un controllo continuo sulla stabilità della rete, migliorando l'affidabilità complessiva dell'applicazione.

In sintesi, l'adozione di RMI si è dimostrata efficace per questo tipo di applicazione distribuita, permettendo un'astrazione pulita delle comunicazioni e facilitando la gestione degli eventi sincroni e asincroni. L'architettura è estendibile e potrebbe facilmente essere adattata per introdurre nuove funzionalità o supportare protocolli di comunicazione più avanzati, se necessario.