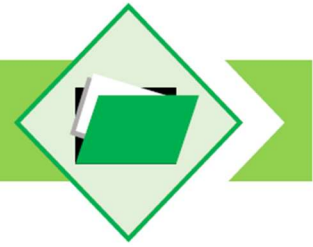


PROTOCOLLO DI RETE



Documentazione relativa al protocollo di rete del **gruppo32**:
“ing-sw-2024-rocca-pasqual-paone-tonni”

Per la connessione client-server abbiamo scelto di implementare la comunicazione RMI e Socket in modo da rendere trasparente sia dalla parte della view sia dalla parte del model quale tipo di comunicazione si sta realmente usando.

Per far ciò abbiamo utilizzato 3 interfacce principali:

- **GameListener**: è un'interfaccia remota implementata direttamente dal client, che espone i metodi necessari per mostrare gli aggiornamenti relativi al model nella view. Questo permette al client di rimanere in ascolto delle modifiche apportate al model e di aggiornare la sua interfaccia grafica o testuale di conseguenza.
- **NotifierInterface**: è un'interfaccia che definisce i metodi utilizzati dai listener per notificare le modifiche del model al client. Ci permette di non differenziare il comportamento dei listener all'interno del model a seconda del tipo di comunicazione. E' implementata per RMI da una classe RMINotifier che a sua volta chiamerà i metodi direttamente sull'oggetto remoto del client, mentre per socket è implementata da ClientRequestHandler, la quale essendo lato server manderà dei messaggi al clientSocket che, una volta ricevuti, chiamerà sul client gli stessi metodi chiamati da RMI.
- **ServerInterface**: è un'interfaccia che definisce i metodi accessibili dal client per interagire con il server e gestire le operazioni di gioco. Questa interfaccia è implementata sia da SocketClient che da RMIServerStub, così che queste classi possano poi usare i rispettivi procedimenti per propagare la richiesta al model.

Usiamo una mappa di listener e notifier all'interno del model (classe ListenersHandler) per propagare le modifiche a tutti i client collegati al gioco in un certo istante.

FUNZIONAMENTO RMI

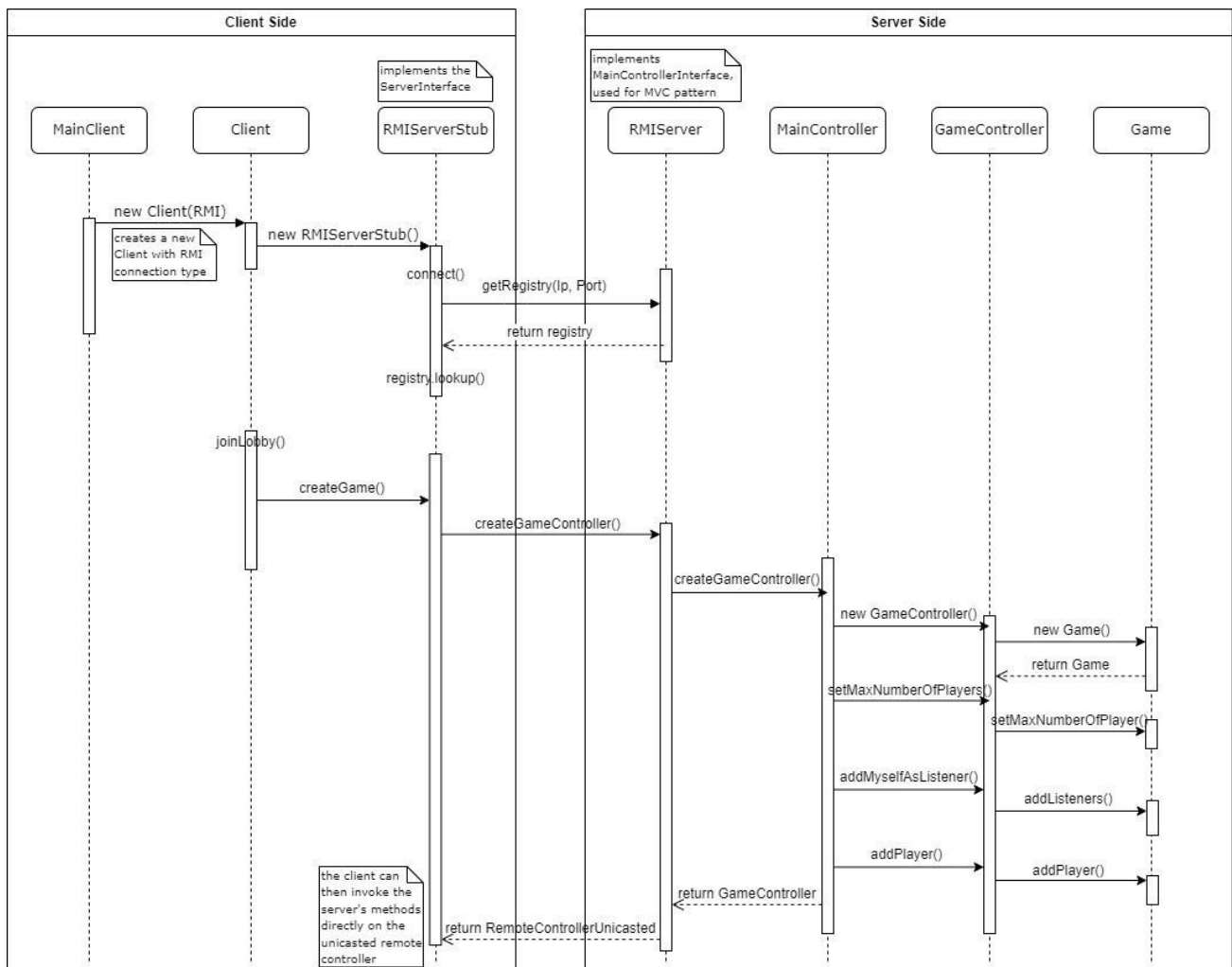
Per gestire la connessione RMI usiamo 3 classi: **RMIServer**, **RMIServerStub** e **RMINotifier**.

Quando un client seleziona RMI creerà un oggetto di tipo **RMIServerStub**, che si connette all'**RMIServer** tramite la lookup del registry. **RMIServer** contiene il **MainControllerInterface**, e una volta in possesso del riferimento al server possiamo richiedere di creare o aggiungerci ad un game, ricevendo indietro l'oggetto remoto **GameController**.

Inoltre, è in questo momento che aggiungiamo listener e notifier al model, così da poter far tornare i cambiamenti alla view con il pattern MVC.

L'**RMIServerStub** contiene i metodi attui alla modifica del model, in quanto, come fa intendere il nome, è essa ad avere lo stub del gameController.

Riportiamo di seguito un sequence diagram per spiegare meglio il funzionamento della comunicazione client-server lato RMI.



FUNZIONAMENTO SOCKET

L'implementazione di socket risulta diversa in quanto non può usare oggetti remoti, ma deve scambiare messaggi attraverso l'input e l'output stream presenti nella connessione.

Nonostante ciò, implementando le stesse interfacce di RMI siamo riusciti a mantenere la parvenza di avere lo stesso comportamento per entrambi.

In socket abbiamo un **ClientSocket**, creato allo stesso modo di RMIServerStub, che si connette al **SocketServer**. Implementando ServerInterface manda al server i messaggi contenenti le azioni da svolgere sul model, oltre che i messaggi iniziali per la creazione/aggiunta a un gioco, e al contempo riceve sul suo inputStream gli aggiornamenti dal model segnalati dai listener.

Quando SocketServer riceve una richiesta di connessione crea per il singolo client una classe **ClientRequestHandler**, che rappresenta la singola connessione client-server. Essa gestisce il ricevimento dei messaggi dal client sull'inputStream, li processa e li esegue nel controller (che tiene come parametro a seguito della prima chiamata che lo setta).

Inoltre, come detto, implementando la NotifierInterface passa gli update dei listener al clientSocket attraverso dei messaggi riposti sull'outputStream del singolo client.

