DOCUMENTO INTEGRATIVO
UML Rete GC48

1) DESCRIZIONE DEI COMPONENTI

Innanzitutto, è fondamentale definire che l'architettura di rete prevede la presenza di due principali interfacce implementate sia dai componenti RMI che da quelli Socket. In particolare, l'interfaccia VirtualClient si trova sul lato Server ed ha metodi che vengono chiamati nella sequenza di comunicazione Server-Client; l'interfaccia VirtualServer, invece, si colloca sul lato client ed ha metodi che vengono chiamati nella sequenza Client-Server.
Una terza interfaccia, chiamata CommonClient, è invece utilizzata per far si che l'utente, tramite la view, non debba essere al corrente del tipo di comunicazione (RMI o Socket) utilizzata, ma si limiti a chiamare metodi su questa interfaccia implementata sia dai componenti RMI che da quelli Socket.

Sia nell'architettura RMI che in quella Socket è stata utilizzata la struttura dati Queue, che permette di tenere memoria di tutte le chiamate da svolgere senza bloccare lo svolgimento di azioni nei due flussi di comunicazione.

[RMI]

Lato Client:

La classe RmiClient, che implementa l'interfaccia remota VirtualClient, invia una richiesta di connessione al lato Server e comunica con esso in modo remoto chiamando metodi dell'interfaccia VirtualServer. Inoltre, RmiClient contiene metodi comunicanti con la view utili nel flusso di comunicazione Server-Client. La classe RmiClient implementa anche l'interfaccia CommonClient sopracitata.

Lato Server:

La classe RmiServer, che implementa l'interfaccia remota VirtualServer, si occupa di ricevere le richieste di connessione da parte della classe RmiClient e di effettuare quindi le chiamate ai metodi del controller nel flusso di comunicazione Client-Server. Inoltre, RmiServer comunica in modo remoto con il lato Client, tramite l'interfaccia VirtualClient, per mostrare le modifiche avvenute nel model.

Action:

In RMI, le chiamate ai metodi con relativi parametri arrivate in RmiServer vengono incapsulate in classi "Action" e inserite in una coda. Questa coda serve per rendere asincrone, come in Socket, le chiamate RMI che altrimenti sarebbero sincrone.

[Socket]:

Lato Client:

La classe ClientSocket invia una richiesta di connessione al Server e successivamente comunica con la classe ClientHandler tramite messaggi [vedi sezione “Messaggi”] per eseguire azioni sul model.
Sul lato Client si trova inoltre la classe ServerHandler, che si occupa di rimanere in ascolto di messaggi provenienti dal lato Server per poi comunicare con la classe ClientSocket (che nel flusso di comunicazione server-client comunica con la view). La classe ClientSocket implementa anche l’interfaccia CommonClient sopracitata.

Lato Server:

La classe SocketServer si occupa di rimanere in ascolto di nuove connessioni e quindi istanziare oggetti di tipo ClientHandler con cui i corrispettivi ClientSocket comunicheranno. Nel flusso di comunicazione Client-Server, SocketServer chiama i metodi del controller per modificare il model. Nel flusso di comunicazione Server-Client, ClientHandler, implementazione dell’interfaccia VirtualClient, si rivolge tramite messaggi alla classe ServerHandler, che comunica con ClientSocket e da lì con la view, per mostrare le modifiche avvenute nel model.

Messaggi:

Nell’architettura di rete di tipo Socket, sono adottati messaggi serializzati che incapsulano tutti i parametri necessari alla specifica chiamata del metodo per la quale sono stati istanziati. I diversi tipi di messaggio estendono la classe astratta Message, che contiene i due metodi “execute” (che differiscono per parametri d’ingresso) soggetti all’Override delle sottoclassi, permettendo la chiamata agli specifici metodi di SocketServer (nel flusso Client-Server) o ClientSocket (nel flusso Server-Client), che comunicheranno, rispettivamente, con il controller e con la view. I due metodi execute ricevono come parametro d’ingresso il riferimento necessario di tipo SocketServer o ClientSocket su cui verrà chiamato il metodo specifico.

[Classi serializzate]:

Una classe PlayerInfromation, serializzata e immutabile, incorpora tutte le informazioni utili del giocatore (Station, hand, points) per passarle attraverso la rete. Abbiamo serializzato solo lo stretto necessario delle classi del Model (principalmente solo le carte e i loro componenti).
Una classe WaintingRoomImmutable incorpora le informazioni delle lobby attualmente disponibili (come i gameId e colori disponibili) per permettere agli utenti che vogliono unirsi una selezione più semplice.

2) SEQUENZA DI COMUNICAZIONE

[RMI]

Da Client a Server:

La view chiama i metodi sull'interfaccia CommonClient che, in RMI, è implementata da RmiClient. Da lì il metodo è chiamato sull'interfaccia VirtualServer implementata da RmiServer. Nel Server la chiamata e i parametri sono incapsulati in una classe "Action" e messi in coda. Un thread si occupa di gestire gli elementi in coda e eseguirli chiamando metodi di ControllerManager e Controller, rispettivamente per azioni generali (create, join, leave game…) o per azioni di gioco (playCard, getCard…). I due controller modificano il Model.

Da Server a Client:

Le modifiche al model o eventuali eccezioni sono segnalate dal Model stesso che chiama un ObserverManager che ha la lista di tutti gli Observer dello stesso game. Gli Observer sono le interfacce VirtualClient e, in RMI, sono implementate direttamente in RMIClient.

[Socket]

Da Client a Server:

La view chiama i metodi sull'interfaccia CommonClient che, in Socket, è implementata da ClientSocket che li incapsula in Message e li trasmette.
Lato server, ClientHandler è in ascolto e arrivato il messaggio lo inoltra a SocketServer dove viene inserito in una coda. Un thread si occupa di gestire i messaggi in coda e eseguirli chiamando metodi di ControllerManager e Controller, per azioni generali (create, join, leave game…) o per azioni di gioco (playCard, getCard…) rispettivamente. I due controller modificano il Model.

Da Server a Client:

Le modifiche al model o eventuali eccezioni sono segnalate dal Model stesso che chiama un ObserverManager che ha la lista di tutti gli Observer dello stesso game. Gli Observer sono le interfacce VirtualClient e, in Socket, sono implementate da ClientHandler che si occupa di incapsulare le notifiche in messaggi e trasmetterli.
Lato client, un ServerHandler è sempre in ascolto di comunicazioni dal Server e quando arriva un messaggio lo ritraduce in una chiamata ad un metodo di ClientSocket.