Descrizione UML Model e Controller – GC42

Introduzione

In questa *Peer Review* portiamo il progetto delle componenti *Model*, *Controller* e *della parte di rete*.
La struttura di base del Model è rimasta perlopiù invariata, se non per alcune modifiche consigliateci nella scorsa *Peer Review*, che abbiamo deciso di implementare, e per alcune modifiche secondarie resesi necessarie durante i lavori di progettazione ed implementazione della connettività di rete.
Entrambi questi aspetti saranno approfonditi nella sezione dedicata al *Model*.

Novità inedita rispetto alla scorsa volta è, invece, il *Controller*: qualche settimana fa avevamo detto che esisteva un *GameController*, e avevamo evidenziato una serie di metodi del *Model* pensati per l'interazione con esso; tuttavia, essendo allora il suo sviluppo ancora piuttosto embrionale, non era stato incluso nel materiale da revisionare.
Ora, invece, il *GameController* ha raggiunto uno stadio di sviluppo molto avanzato, al punto da permettere già di giocare delle partite quasi complete, ed è affiancato da tutta una serie di classi che formano il package *Network*. Anche questo verrà approfondito.

Segnaliamo, inoltre, che i lavori di sviluppo della componente *View* procedono spediti, con una *GUI* già molto matura ed una *TUI* che segue a ruota; tuttavia, non saranno oggetto di revisione; dunque, verranno condensati in un generico componente *ViewController*.
Basti sapere che ViewController è un'interfaccia comune ad entrambe le componenti che gestiscono, rispettivamente, *GUI* e *TUI*: dunque è possibile per il *Controller* operare indistintamente su ognuno dei due, rendendoli di fatto intercambiabili e permettendo anche di giocare partite "ibride", dove i vari giocatori possono non usare tutti la stessa interfaccia.

Model

Come detto, la struttura del *Model* è rimasta perlopiù invariata: rimane la classe *Game* come "classe principale", ovvero la classe che contiene tutte le altre: gli oggetto di tipo *Game* rappresentano, quindi, un intero *Model*, come vedremo più avanti.

Evitiamo di soffermarci sulla spiegazione dettagliata del *Model* laddove non risultano sostanziali differenze: in caso di dubbi invitiamo a recuperare la scorsa relazione.
Approfondiamo invece quello che è cambiato nel corso delle ultime settimane.

Modifiche suggerite

In sede di *Peer Review* erano stati avanzati diversi suggerimenti.
Di questi, abbiamo deciso di implementarne tre:

1. Migliorata la gestione delle carte scoperte: Il metodo per prendere le carte scoperte nella scorsa implementazione avevamo un *PlayingDeck* contenente un *Deck* e due *Card*: quando una carta veniva presa (*grab*), un'altra carta veniva pescata da quel mazzo e messa al suo posto (*putDown*).
   Questo presentava un problema nel caso in cui il mazzo era vuoto: se l'altro mazzo non era vuoto allora doveva essere lui ad eseguire il *putDown*.
   Per risolvere questo problema abbiamo deciso di spostare all'esterno del *PlayingDeck* l'implementazione delle funzioni di *grab* e *putDown*: la prima ora è gestita dal *GameController* (anche per altre ragioni), mentre *putDown* direttamente da *Game*, che può così interagire contemporaneamente con entrambi i *PlayingDeck* interessati, permettendo di risolvere la problematica senza dover ripensare tutte le interazioni in cui il confinamento di *Deck* e *Card* dello stesso tipo in un solo oggetto risultava, invece, comodo.
2. Unite le classi *KingdomResource* e *Resource*: ora queste due enumerazioni, che abbiamo comunque deciso di mantenere separate, sono unite sotto il nome di *Item*, l'interfaccia che entrambi implementano.
3. Divise le carte in *PlayableCard* e *ObjectiveCard*: è stata creata una distinzione tra i due tipi di carte: nel primo gruppo rientrano le carte Risorsa, Oro e Iniziali, mentre le carte Obiettivo esistono in modo separato ed indipendente.
   Facciamo notare, inoltre, che le *frontImage* e *backImage* ora contengono delle *String* e non più delle *Image*: questa è una delle modifiche imposte dalla parte di rete, in quanto le classi della libreria *JavaFx* (di cui *Image*) non sono serializzabili.
   Al loro interno ora si trova l'*URL* dell'immagine, che viene caricata direttamente dalla *View* invece che dal *Model*.
   P.S.: Siamo consapevoli che in alcuni punti (ad esempio *Deck* e *PlayingDeck*), la presenza di *Card* invece che *PlayableCard* non sia il massimo: stiamo valutando se separare il mazzo delle carte Obiettivo dai mazzi di carte giocabili, tuttavia la parte di rete ci ha tenuti occupati per più tempo del previsto.

Rimaniamo, invece, fermamente convinti dell'implementazione del *PlayField* come *ArrayList*: nel tempo ha dimostrato una serie notevole di vantaggi, tra cui la naturale capacità di tener traccia dell'ordine in cui vengono giocate le carte, con tutti i vantaggi che ne conseguono dal lato della *View*.
Non abbiamo notato, invece, problemi di prestazioni legate allo scorrimento della lista, neanche in alcune situazioni particolarmente complesse che abbiamo avuto modo di testare.

Modifiche introdotte per la connettività

La parte di *Network* è stata forse la più complessa fino ad ora, e ha richiesto numerosi test ed aggiustamenti per poter essere inserita nel progetto.
Quello che è presente ora, che verrà spiegato meglio nella sezione dedicata, è il frutto di diversi giorni di progettazione , test ed implementazione che hanno richiesto anche di ripensare alcune delle interazioni con e dentro il *Model*.

Innanzitutto, chiariamo che l'intero *Model*, tutte le classi, ora implementano l'interfaccia *Serializable* (*java.io.Serializable*): questo consente di serializzare (convertire in stringhe) qualunque oggetto al suo interno, e ovviamente di de-serializzarlo: questo è necessario per le operazioni in lettura via *RMI*, ma potrà anche essere utilizzato manualmente in *Socket*.
Per evitare di riempire l'*UML* di frecce, abbiamo evitato di rappresentare questa interfaccia, ma ripetiamo che tutto il *Model* la implementa.

Un' altra modifica, già discussa sopra, riguarda la sostituzione delle *Image* in *Card* con delle *String*.

Controller

Durante la prima *Peer Review* avevamo definito la nostra implementazione *Thin Controller / Fat Model*: questo perché avevamo inserito il grosso della logica all'interno del *Model*, ed il *Controller* si occupava, invece, solamente dello "*scripting*" della partita.

GameController

Il *GameController* è la componente principale, nonché più importante, dell'intero progetto.

Nato come "direttore d'orchestra" per il gioco, oggi esegue questo compito in tre modi diversi: unendone i risultati per permettere l'esecuzione del gioco nella sua interezza.

1. **Gestione delle fasi di gioco**
   Il *GameController* si occupa di gestire il corretto funzionamento ed avanzamento del gioco: gestisce le fasi di gioco, il sistema dei turni e si occupa di eseguire le azioni più complesse, quelle che coinvolgono più classi appartenenti al *Model* (come *draw* o *grab*) implementando dei controlli nel caso mi trovi nelle fasi finali del gioco (mazzi finiti), ed invece reindirizza le chiamate ai metodi interni del *Model* nel caso delle azioni più semplici, che possono essere gestite internamente.
   Tutte le azioni possibili nel gioco passano per il *GameController*, che è l'unico che può fisicamente modificare il *Model*.
2. **Controllo del *Model***
   Il *GameController* contiene un `private final Game game`: creare un *GameController* significa creare un *Game*, e dunque un *Model*.
   Questo rende il *GameController*, di fatto, una partita o *match*: vedremo che questo aspetto è importante nella parte di rete.
   Per lo stesso motivo abbiamo pensato di dare al *GameController* un nome, sotto forma di *String*: avendo deciso di supportare la FA delle partite multiple, abbiamo pensato di mostrare ai giocatori una lista di partite disponibili, ognuna riconoscibile dal proprio nome.
3. **Controllo delle *View***
   Il *GameController* contiene al suo interno un *ArrayList* di *RemoteViewController*: vedremo a breve cosa sono, ma per ora basti sapere che sono oggetti strettamente legati alle *View* (*GUI e TUI*).
   GameController ha dunque anche la funzione di "punti di contatto" tra *Model* e *View*.

Network

Il *package Network* contiene tutte le classi che entrano in gioco per garantire la connettività di rete.
Abbiamo ovviamente deciso di sviluppare sia la connessione via *RMI* che *Socket*: dunque abbiamo progettato una soluzione il più generale possibile, in modo che entrambe le modalità di connessione possano essere perfettamente intercambiabili.

Server

Il primo elemento degno di nota nel *Server* è, appunto, *Server*: questa classe è una *Application*, dunque ha una propria *GUI*, e crea al suo interno due *ServerNetworkController*: un *RmiControllerServer* ed un *SocketControllerServer*.
Questi vengono creati contemporaneamente nel momento in cui viene premuto il pulsante "*Start*" nella GUI.

ServerNetworkController è un'interfaccia pensata per astrarre il più possibile i due *controller* di rete: è implementata sia da *RmiControllerServer* che da *SocketControllerServer*, e questo permette di avere gli stessi identici metodi da entrambe le parti.

Server, infine, si occupa di un'ultima cosa: crea un'istanza di *GameCollection*, che poi passa ad entrambi i *ServerNetworkController*: questo permette ai due *controller* di lavorare sugli stessi dati, e dunque di controllare le stesse partite allo stesso momento.

GameCollection

GameCollection è un semplice contenitore di *GameController*: come avevamo detto prima, ogni *GameController* è una partita; dunque, questo contenitore è un contenitore di partite.
Essendo lo stesso oggetto passato ad entrambi i *ServerNetworkController*, entrambi operano sulle stesse partite.
GameCollection ha anche il compito di creare nuovi *GameController*, ovvero nuove partite.

ServerManager

Entrambi i *ServerNetworkController*, poi, creano un proprio *ServerManager*: questa è forse la componente più importante della parte di rete.
Al suo interno entrambi mettono la *GameCollection* ricevuta dal *Server*, e dunque i due *ServerManager* operano sulle stesse partite.
La scelta di utilizzare due *ServerManager* separati, invece di uno solo, deriva dal fatto che l'uso che ne viene fatto varia di molto tra *RMI* e *Socket*: per evitare problemi abbiamo deciso di separarli.

ServerManager è dunque la componente dedicata al trasferire la volontà dei *Client* al *GameController*: implementa l'interfaccia *RemoteServer*, che serve principalmente per *RMI*, ma permette anche a *Socket* di accedere direttamente a tutti i metodi di interazioni in ingresso ed in uscita possibili, senza doverli re-implementare nuovamente.
Nell'UML è possibile vedere quali sono questi metodi: notare che i metodi che arrivano a

ServerManager contengono quasi tutti informazioni semplici quali numeri o valori di *enum*: questo è importante per 2 ragioni:

1. Porta l'esecuzione dei metodi direttamente dentro il *Server*: *ServerManager* costruisce le chiamate ai metodi di *GameController* utilizzando le informazioni ricevute.
   Nota: nell'*UML* gli *int* in rosso rappresentano il *gameID*, quelli in verde il *playerID*.
   Il *GameController*, così come il *Player* su cui eseguire l'azione, così come in altri casi per il *cardID*, vengono "scelti" da GameController stesso, mentre i *Client* dicono solo "quale *GameController*", "quale *Player*", ecc...
   In questo modo si garantisce che la modifica venga fatta sugli oggetti su *Server*, e dunque può essere letta da tutti i *Client*.
2. Rende l'implementazione molto simile tra *RMI* e *Socket*: come detto, le chiamate ai metodi remoti via *RMI* di fatto trasferiscono informazioni semplici, numeri interi in genere, e dunque sono facilmente traducibili in equivalenti messaggi nella parte *Socket*.

Se fino a qui l'implementazione di *RMI* e *Socket* sembrano molto simili, le cose cambiano per i *ServerNetworkController*.

RmiControllerServer

RmiControllerServer, che, come detto, riceve una *GameCollection* da *Server* e la mette nel proprio *ServerManager*, poi condivide *ServerManager* tramite l'*RMIRegistry*, rendendolo disponibile agli *RmiClient* come *Stub*.
Questo è il motivo per cui *ServerManager* implementa l'interfaccia *RemoteServer*: viene usato come oggetto remoto dai *Client*, che chiamano direttamente i suoi metodi passandogli i parametri richiesti.

SocketControllerServer

SocketControllerServer, invece, usa il proprio *ServerManager* localmente: apre una propria connessione con *SocketClient*, e traduce i messaggi ricevuti in equivalenti chiamate ai metodi di *ServerManager*.

Client

Guardiamo adesso al *Client*: come abbiamo appena detto, il modo in cui si chiamano i metodi è diverso: in *RMI* si interagisce sull'oggetto remoto, mentre in *Socket* si mandano messaggi.
Per rendere anche qui l'implementazione completamente intercambiabile abbiamo creato un'interfaccia *NetworkController*, implementata sia da *RmiClient* che da *SocketClient*: in questo modo i *ViewController* vedono solamente un generico *NetworkController*, su cui chiamano i metodi sulla falsa riga di *RMI*.
Se il *NetworkController* è effettivamente un *RmiClient*, allora le chiamate vengono semplicemente passate sullo *Stub*, mentre se è un *SocketClient* queste chiamate vengono tradotte in messaggi, che vengono inviati via *Socket*, ricevuti e ri-tradotti in chiamate ai metodi.

Facciamo notare che l'interfaccia *NetworkController*, quella attraverso cui i *ViewController* "vedono" *RmiClient* e *SocketClient*, contiene una copia di tutti i metodi contenuti in *RemoteServer*: l'unica differenza è che i metodi di *NetworkController* non richiedono il *gameID* come parametro.
Abbiamo deciso di isolare il *gameID* come parametro interno ai *controller* di rete in modo da semplificare il lavoro nei *ViewController*. Il *gameID*, richiesto invece nei metodi di *RemoteServer*, viene "iniettato" dai *NetworkController* nel momento in cui costruiscono il metodo remoto / messaggio *Socket*.

Ovviamente facciamo notare, nel caso non fosse chiaro, che questo ragionamento vale solo ed unicamente per i metodi di modifica: risulta così impossibile inviare un comando "a nome di un altro giocatore" da un *Client*, perché verrà sempre inviato con il proprio *playerID*.
Per la lettura di dati, invece, è possibile utilizzare i metodi `getGame()`, `getPlayer()` ecc, che permettono di specificare il *playerID*, e dunque di leggere lo stato degli altri giocatori: questo è importante, in quanto la nostra *GUI* permette di vedere in tempo reale i tavoli degli altri giocatori.

Concludiamo parlando di come abbiamo gestito i "messaggi di aggiornamento", ovvero i messaggi che partono in automatico dal *Server* verso i *Client* per comunicare la modifica che è stata appena fatta.

Il nostro approccio iniziale è stato di utilizzare dei *Listener*, come già facciamo internamente al *Model* e tra *Model* e *Controller*; tuttavia, ci siamo scontrati con dei seri problemi in fase di implementazione di *RMI*, legati al fatto che gli oggetti inviati tramite *RMI* devono essere serializzati, e la serializzazione di oggetti contenente codice (i *Listener*, per l'appunto) richiede di serializzare l'intero *context*, ovvero la classe che li ha generati.
Questo crea problemi quando la classe contiene oggetti non serializzabili, come ad esempio l'intera libreria di *JavaFx*...

ClientController

Per risolvere questo problema abbiamo trovato una soluzione, forse un po' complicata, ma che funziona alla perfezione.
Essenzialmente, quello che abbiamo fatto è stato creare una classe *ClientController*, che contiene al suo interno un *ViewController* (ovvero uno tra *GUI* e *TUI*) ed implementa l'interfaccia *RemoteViewController*.
Questa interfaccia è una "gemella remota" dell'interfaccia *ViewController*, ovvero contiene gli stessi metodi che le *View* contengono ed utilizzano, ma è pensata per essere usata come oggetto remoto.

RmiClient

Quando *RmiClient* recupera lo *Stub* del *Server* dall'*RMIRegistry*, invece di utilizzare la connessione *RMI* in maniera unidirezionale, *RmiClient* fa un *bind* sullo stesso *Registry* del *Server*, inserendoci questo *ClientController*: il *Server* ha così un oggetto remoto (detto *Skeleton*, l'equivalente di uno *Stub* ma del *Client*) su cui può chiamare i vari "metodi di

aggiornamento", indicati come `notify…()`, che indicano ai *ViewController* di leggere le informazioni aggiornate e di eseguire i *refresh* necessari.

Essendo che in una partita ci sono fino a 4 giocatori, ed essendo che il *Server* supporta un numero potenzialmente molto grande di partite (non abbiamo inserito un limite), non potevamo predisporre dei "nomi standard" per questi *binding* degli *Skeleton*;

Dunque, la nostra soluzione prevede la generazione di un numero casuale tra 1 e 100.000.000 (potenzialmente aumentabile), che viene usato come "nome" e poi notificato all'*RmiControllerServer* tramite il metodo `lookupClient()`.
Se è già stato effettuato un bind sul numero generato, se ne genera uno nuovo.

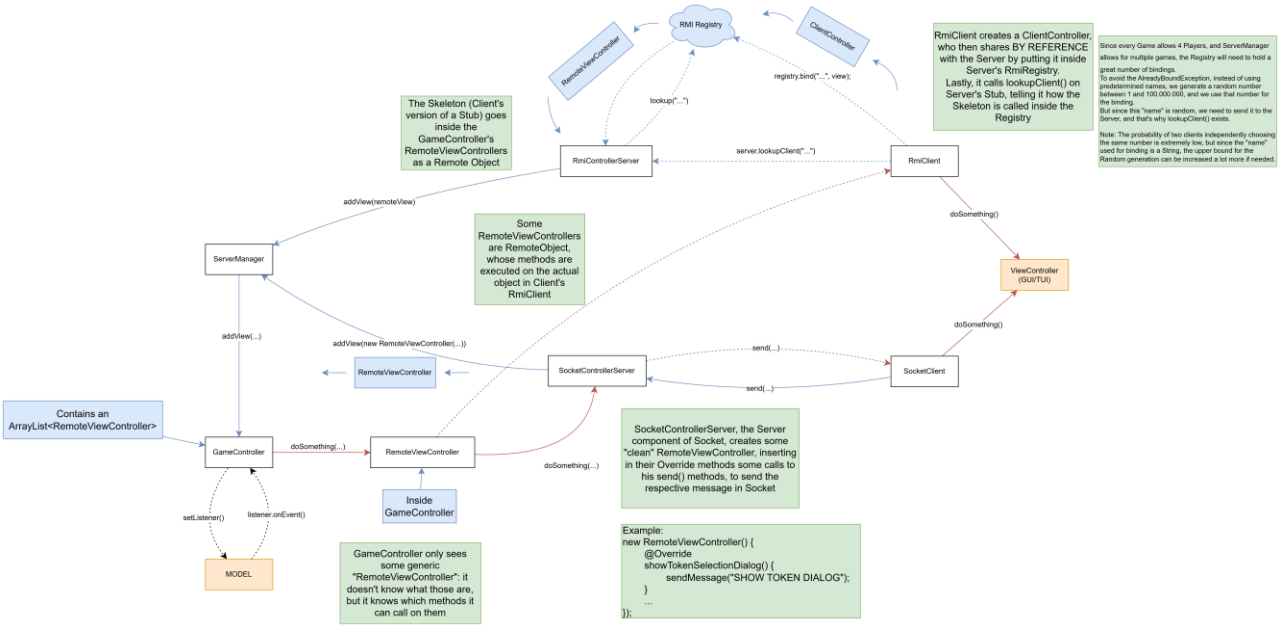
SocketClient

In *Socket*, invece, viene aperta una normale connessione, e mentre *SocketClient* conserva un riferimento al proprio *ViewController*, è invece *SocketControllerServer* a creare il *RemoteViewController*: nei suoi metodi `@Override` viene inserito il codice necessario ad inviare un messaggio equivalente via *Socket*, che viene ricevuto da *SocketClient* e tradotto nell'equivalente metodo sul *ViewController*.

Messaggi per socket

Per lo scambio di messaggi tramite S*ocket* è stato scelto di utilizzare una sorta di *Command Pattern*: il *Client* invia al server dei *Message* (o sue sottoclassi) che contengono un attributo che ne identifica il tipo tramite una *Enum* per comunicare al server quale metodo si vuole chiamare e gli attributi di questo metodo che verranno prelevati tramite un metodo `toString()`.
Il S*erver* avrà il compito di tradurre questi messaggi in azioni da eseguire, ed a sua volta di inviare dei Message per comunicare al *Client* la presenza di aggiornamenti.

Di seguito uno schema del *setup* della connessione sia in *RMI* che in *Socket*, e dello scambio di messaggi di aggiornamento (in rosso):

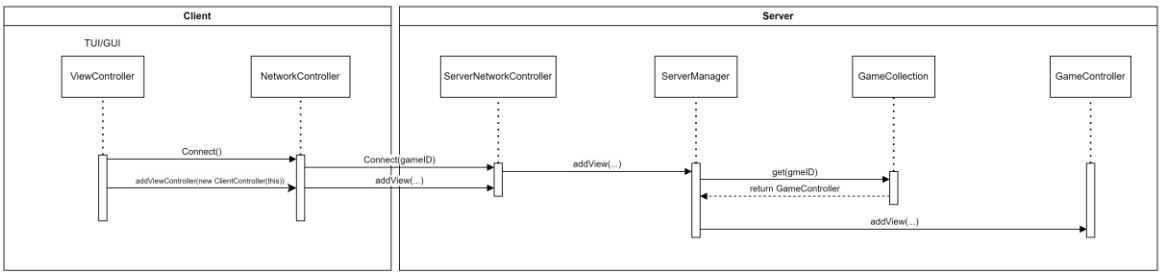


Simone Rodari, Cristian Summa, Mathias Rodigari, Gabriele Pignataro
Peer Review 2

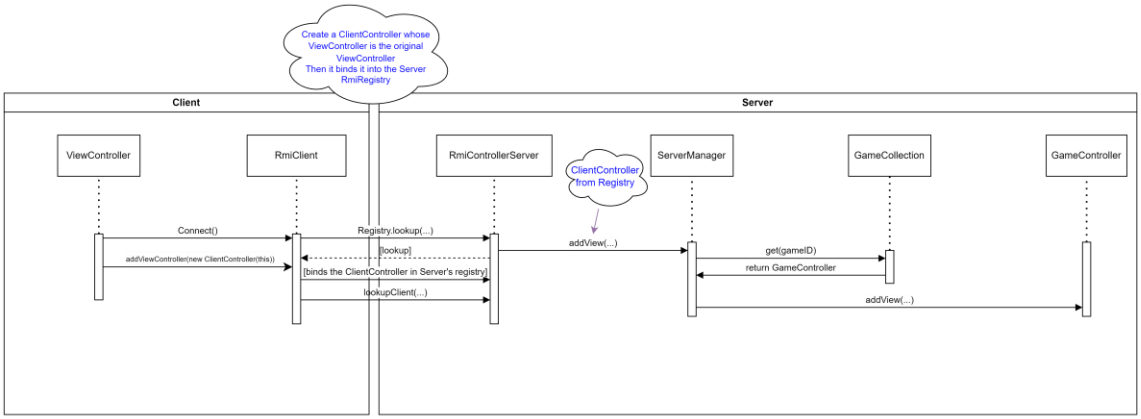
Sequence Diagrams

Connection and Setup

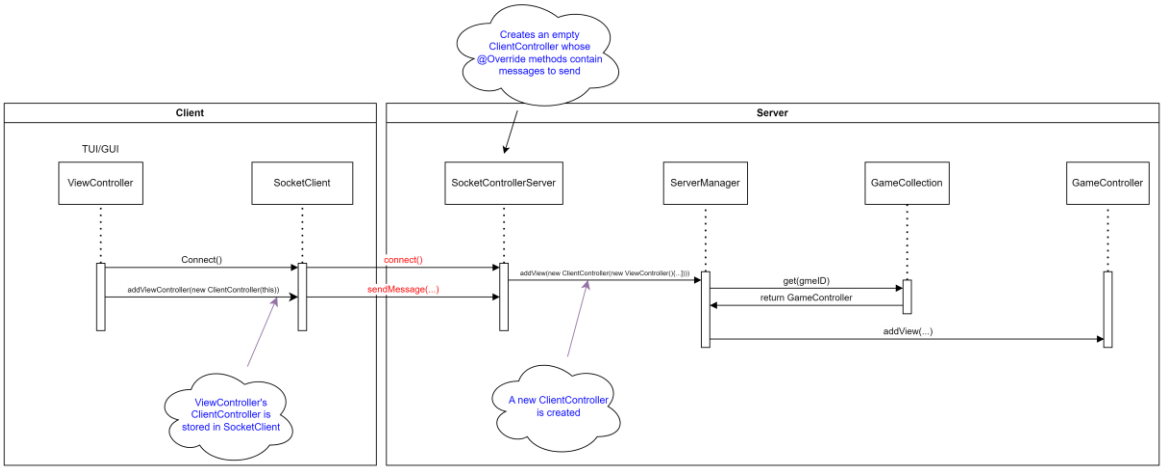
General



Rmi

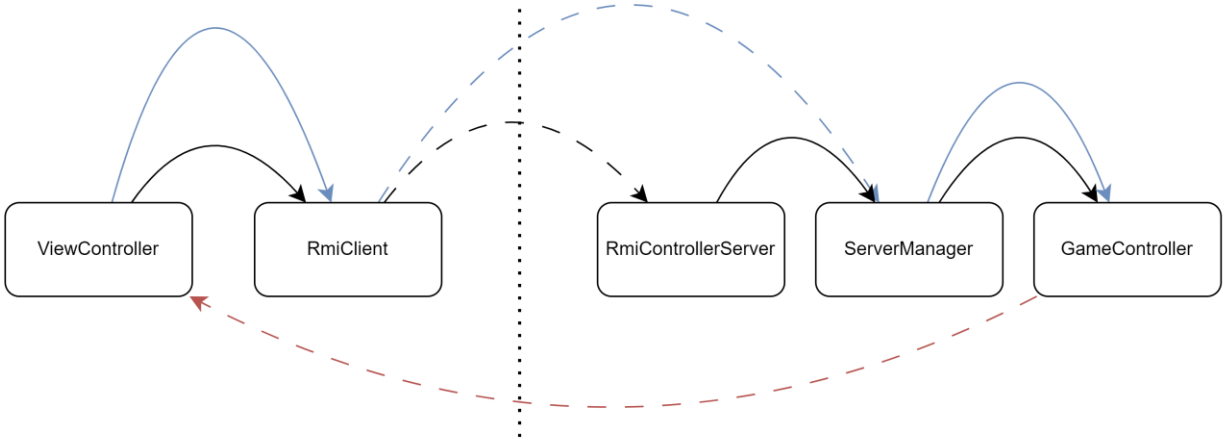


Socket



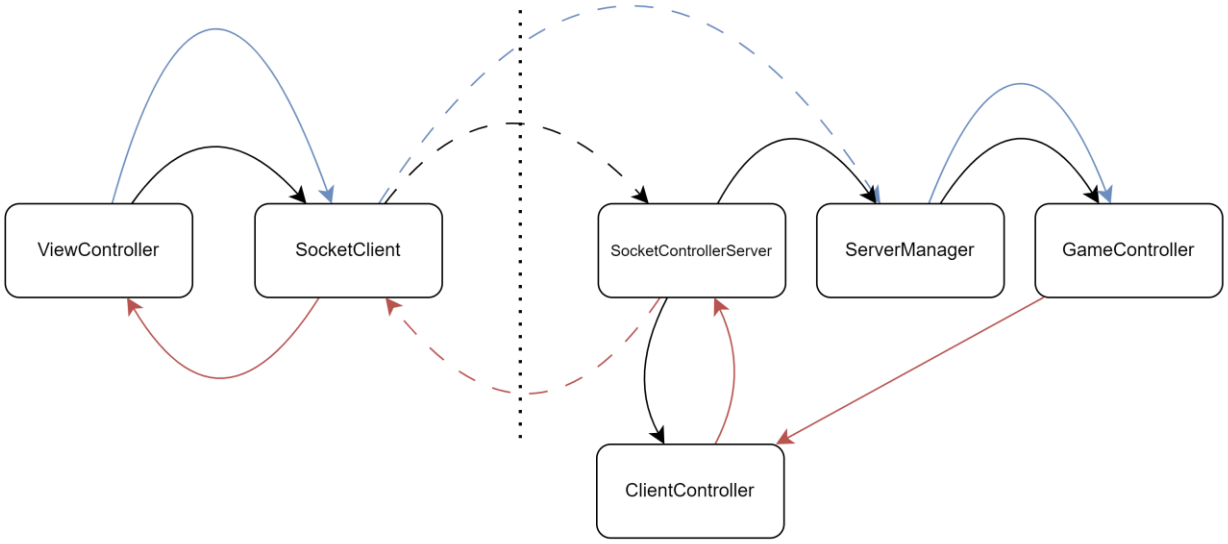
RMI:

white: connection setup
blue: sending actions
red: receiving automatic updates



SOCKET:

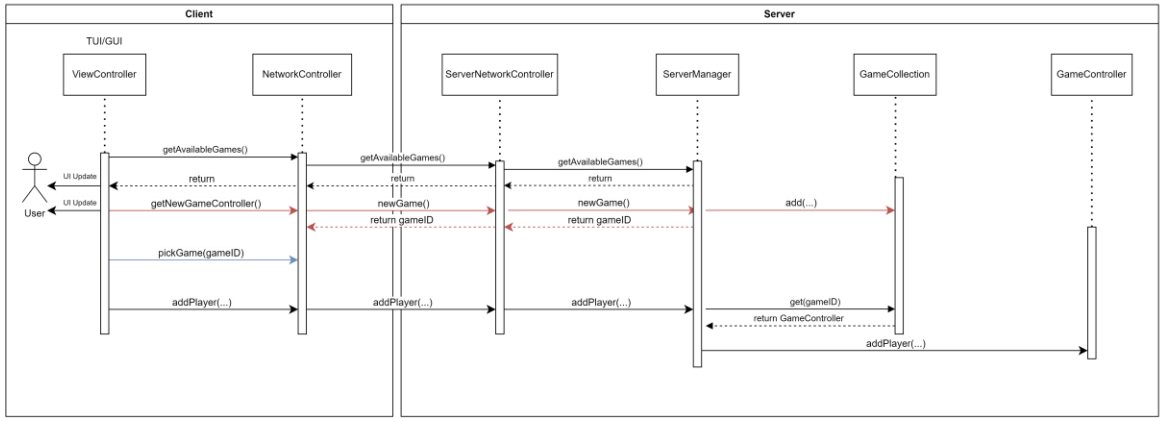
white: connection setup
blue: sending actions
red: receiving automatic updates



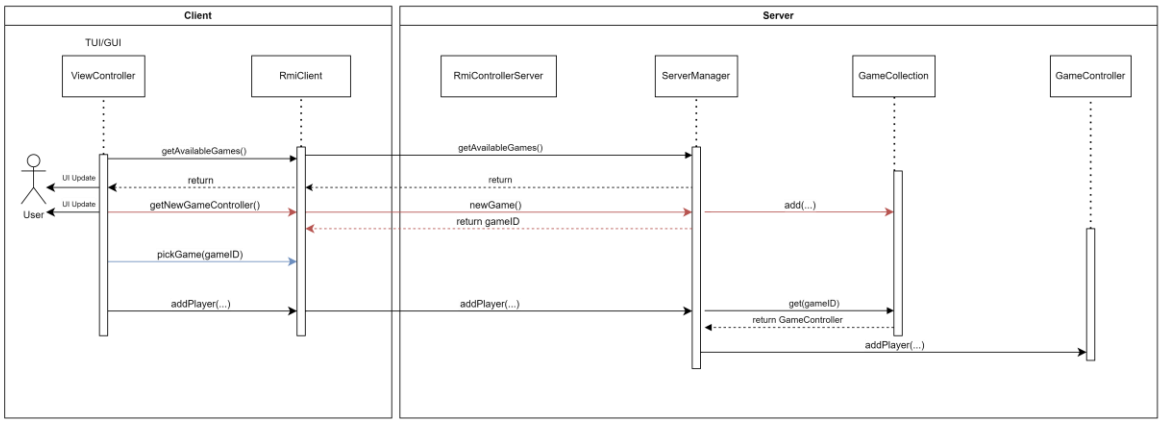
Simone Rodari, Cristian Summa, Mathias Rodigari, Gabriele Pignataro
Peer Review 2

Creating/Joining a Game

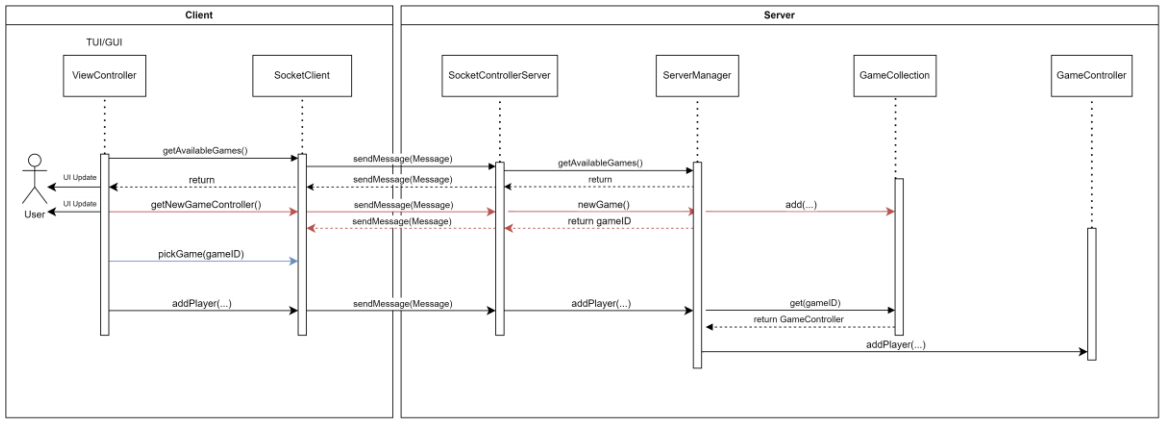
General



Rmi



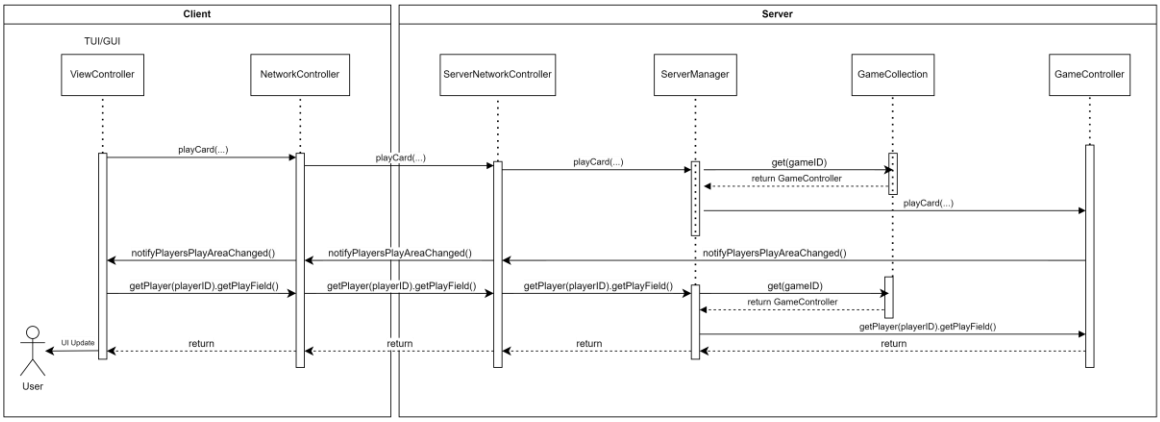
Socket



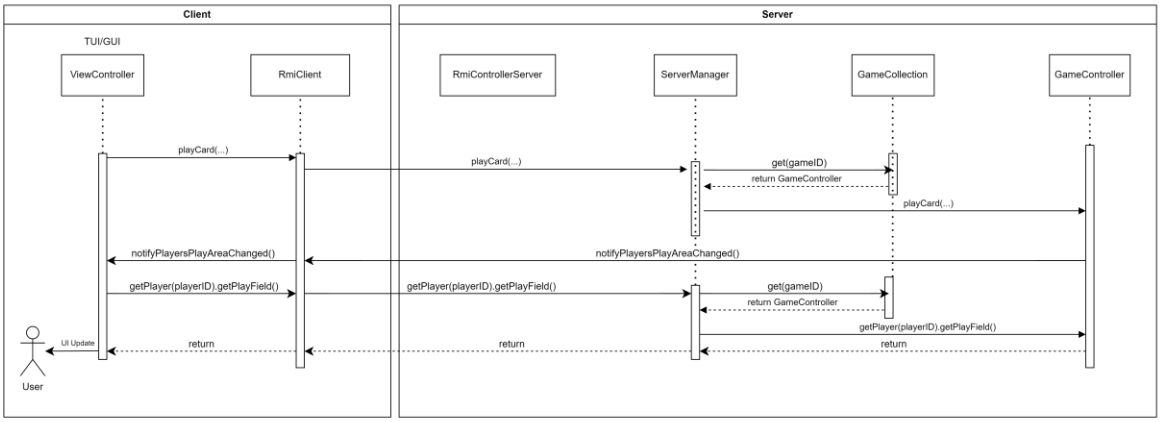
Simone Rodari, Cristian Summa, Mathias Rodigari, Gabriele Pignataro
Peer Review 2

Performing an Action (for example “playCard”)

General



Rmi



Socket