Peer-Review #2:

Pietro Roggero, Lorenzo Tallarico, Matteo Trombetta, Tommaso Razza | ***Gruppo GC06***.

Attualmente abbiamo scelto di implementare la comunicazione tramite **RMI**.
I clients e il server comunicano tra di loro invocando alcuni metodi remoti e per la maggior parte delle volte si passano come parametri degli oggetti ***"Action"***.
Esistono varie sottoclassi della omonima classe e rappresentano ognuna un tipo di azione diversa, all'interno hanno una *enum* per identificarle con più facilità e diversi attributi in base alla necessità.
A seconda dell'*Action* ricevuta, verranno invocati metodi diversi del game controller, il quale si occuperà di gestire una singola partita, dato che non implementeremo la funzionalità per partite multiple. Il controller gestirà quindi la creazione della partita, l'entrata dei giocatori in essa e tutte le fasi di gioco.

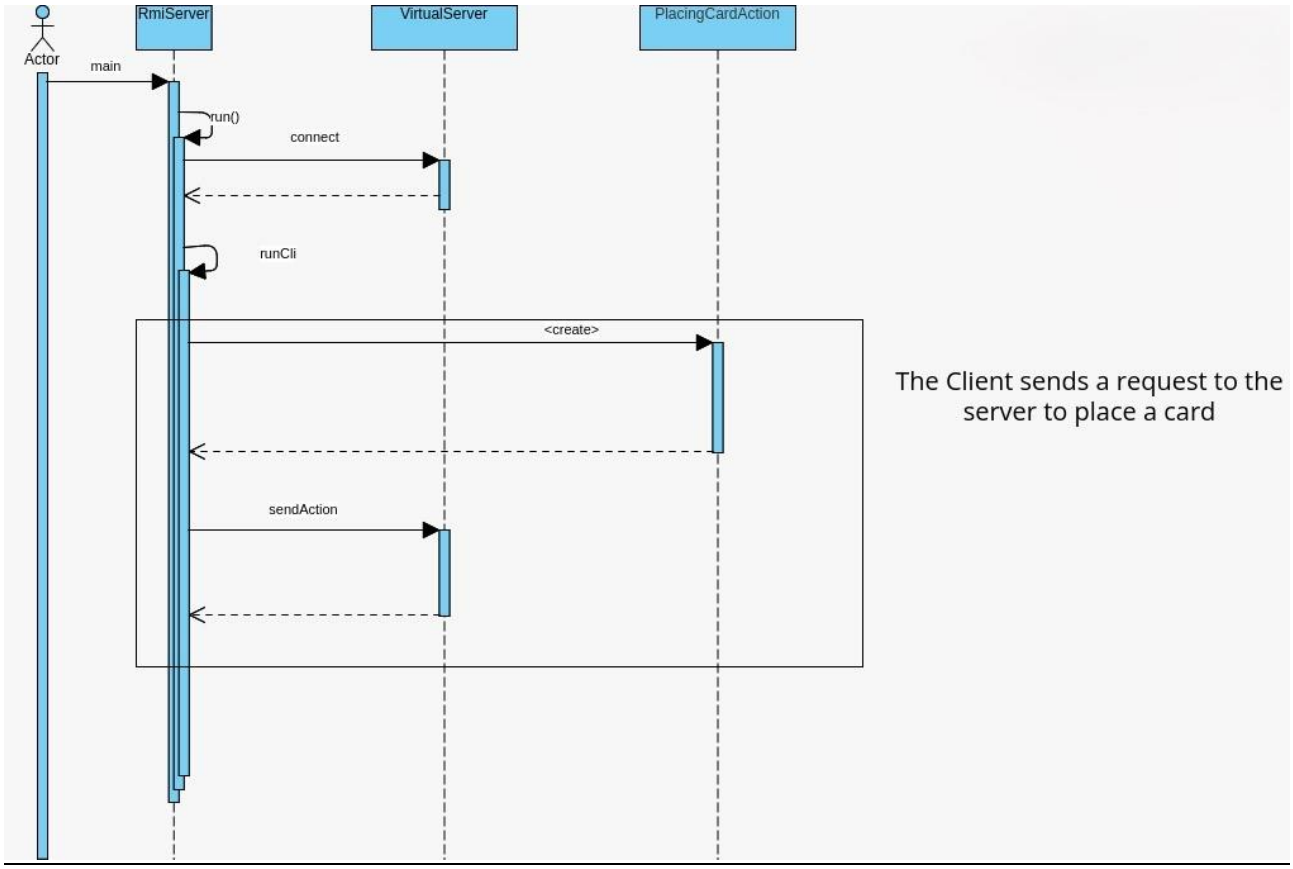
Nel nostro progetto gestiamo la comunicazione tra client e server servendoci di due *Blocking Queue*: ***"serverActions"*** e ***"clientActions"***. Queste code vengono gestite tramite l'utilizzo di due threads, che vengono eseguiti in loop contemporaneamente, e i loro rispettivi metodi. Si può aggiungere un'azione in una delle due code e verrà eseguita dal server o dal client, ogni azione verrà eseguita in modo sequenziale evitando problemi di inconsistenza sul model.

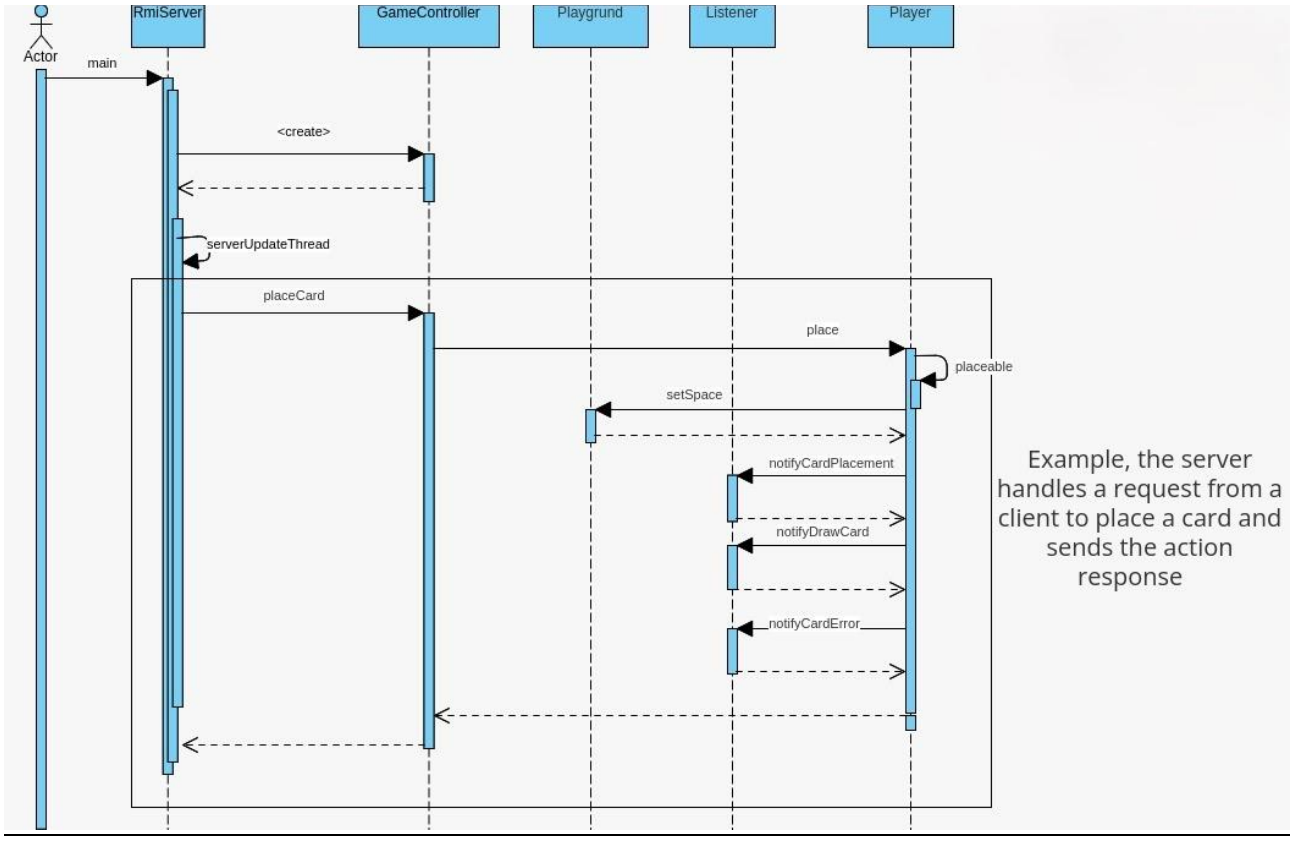
In ***serverActions*** sarà il server ad aggiungere istanze delle sottoclassi di Action tramite un metodo che verrà invocato da parte del client dopo l'invio di un determinato comando.
L'andamento sarà quindi questo: si avrà l'invio al server di un comando da parte del client, il comando verrà eseguito dal server, il quale, mediante un metodo, aggiungerà alla coda *serverActions* una istanza di un oggetto sottoclasse di Action apposita per il comando che è stato inviato. Il server avrà un metodo che si occuperà continuamente di monitorare la coda, questo metodo non appena noterà l'aggiunta di un elemento sulla coda lo prenderà rimuovendolo da questa. L'oggetto ottenuto verrà processato in maniera differente a seconda del suo tipo e verrà invocato il controller che chiamerà un metodo (differente in base al tipo di Action) che a sua volta invocherà un metodo del model che attuerà la modifica.

Ogni qual volta che il model viene modificato, ci sono dei ***listeners*** appositi che notificano i clients e gli comunicano le modifiche. Il *listener* è definito tramite la sua propria classe, è istanziato dentro il model e ha tra gli attributi la lista di clients. Il *listener* mediante diversi metodi ***"notify()"*** manda delle istanze di sottoclassi di Action ai client (ciascuna specifica per il tipo di modifica che è avvenuta sul model), il client a sua volta processerà le azioni ricevute in base al tipo e si comporterà di conseguenza. Tramite queste azioni verrà gestita anche lo stato in cui ogni player si trova e determineranno quindi anche quali azioni potrà svolgere e quali no.

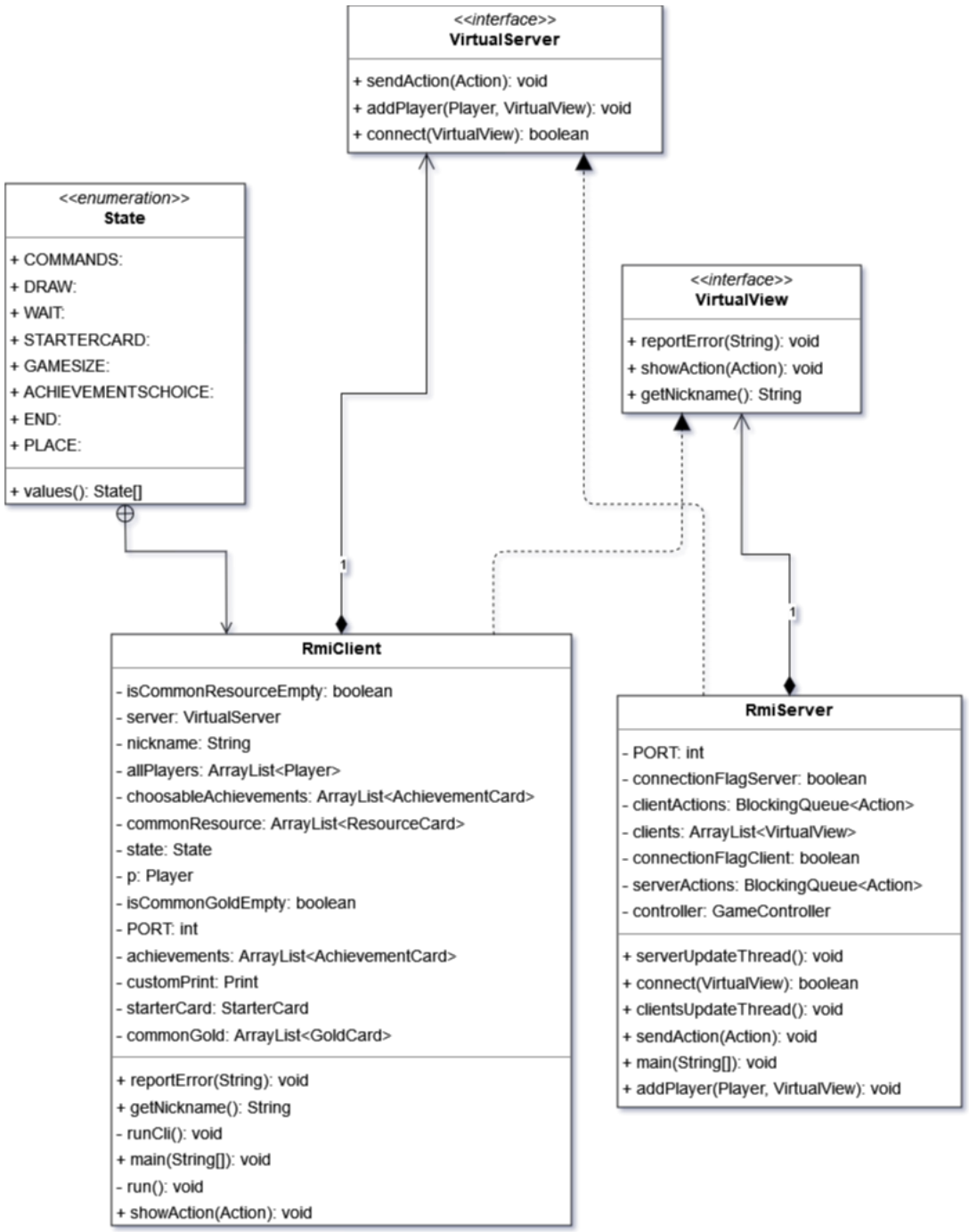
La coda ***clientActions*** entrerà in gioco solamente per alcune operazioni che riguardano il comunicare determinati messaggi di errore, per la gestione della chat di e per la gestione delle connessioni al server. Tutte le altre comunicazioni server -> client partono dal model tramite i listeners rispettando così il pattern MVC.

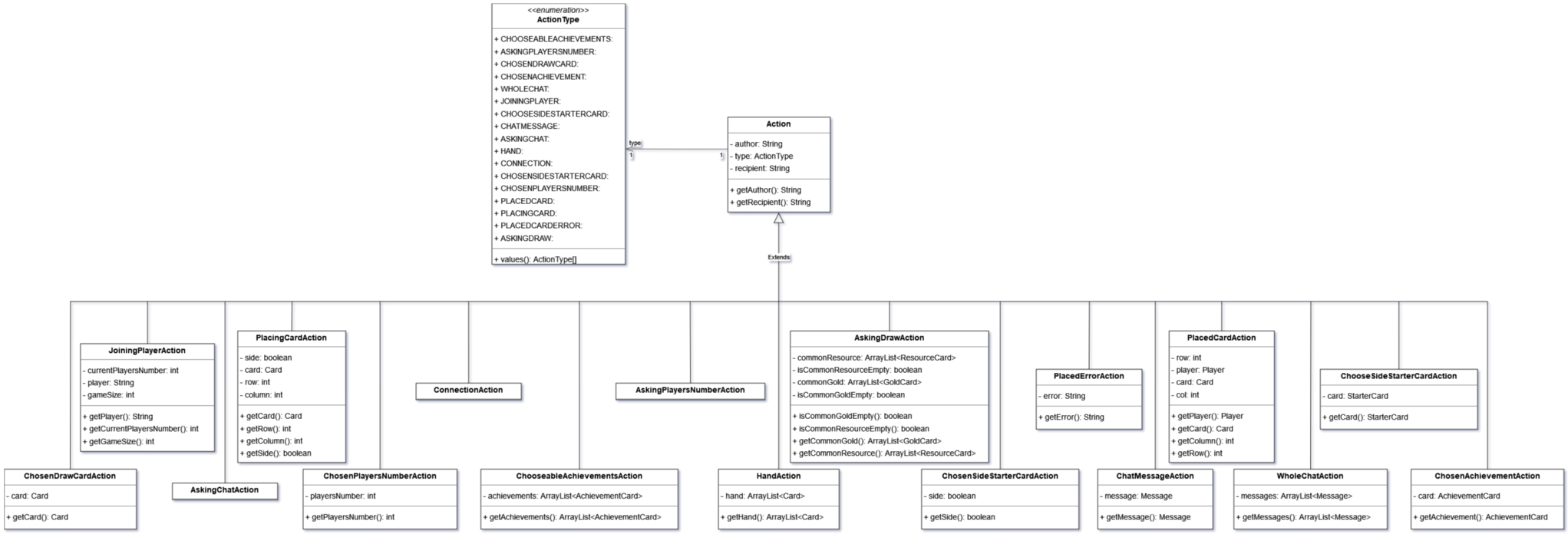
Riportiamo per completezza dei ***Sequence Diagram*** con degli **<u>esempi</u>** di interazione Client-Server:





Di seguito una gli UML delle varie classi Action e della connessione tra Client e Server RMI.
(Per una visione migliore sono stati anche caricati individualmente come allegati della mail)




<<enumeration>>
ActionType
+ CHOOSEABLEACHIEVEMENTS:
+ ASKINGPLAYERSNUMBER:
+ CHOSENDRAWCARD:
+ CHOSENACHIEVEMENT:
+ WHOLECHAT:
+ JOININGPLAYER:
+ CHOOSESIDESTARTERCARD:
+ CHATMESSAGE:
+ ASKINGCHAT:
+ HAND:
+ CONNECTION:
+ CHOSENSIDESTARTERCARD:
+ CHOSENPLAYERSNUMBER:
+ PLACEDCARD:
+ PLACINGCARD:
+ PLACEDCARDERROR:
+ ASKINGDRAW:
+ values(): ActionType[]
type
1
1
Action
- author: String
- type: ActionType
- recipient: String
+ getAuthor(): String
+ getRecipient(): String
Extends
JoiningPlayerAction
- currentPlayersNumber: int
- player: String
- gameSize: int
+ getPlayer(): String
+ getCurrentPlayersNumber(): int
+ getGameSize(): int
PlacingCardAction
- side: boolean
- card: Card
- row: int
- column: int
+ getCard(): Card
+ getRow(): int
+ getColumn(): int
+ getSide(): boolean
ConnectionAction
AskingPlayersNumberAction
AskingDrawAction
- commonResource: ArrayList<ResourceCard>
- isCommonResourceEmpty: boolean
- commonGold: ArrayList<GoldCard>
- isCommonGoldEmpty: boolean
+ isCommonGoldEmpty(): boolean
+ isCommonResourceEmpty(): boolean
+ getCommonGold(): ArrayList<GoldCard>
+ getCommonResource(): ArrayList<ResourceCard>
PlacedErrorAction
- error: String
+ getError(): String
PlacedCardAction
- row: int
- player: Player
- card: Card
- col: int
+ getPlayer(): Player
+ getCard(): Card
+ getColumn(): int
+ getRow(): int
ChooseSideStarterCardAction
- card: StarterCard
+ getCard(): StarterCard
ChosenDrawCardAction
- card: Card
+ getCard(): Card
AskingChatAction
ChosenPlayersNumberAction
- playersNumber: int
+ getPlayersNumber(): int
ChooseableAchievementsAction
- achievements: ArrayList<AchievementCard>
+ getAchievements(): ArrayList<AchievementCard>
HandAction
- hand: ArrayList<Card>
+ getHand(): ArrayList<Card>
ChosenSideStarterCardAction
- side: boolean
+ getSide(): boolean
ChatMessageAction
- message: Message
+ getMessage(): Message
WholeChatAction
- messages: ArrayList<Message>
+ getMessages(): ArrayList<Message>
ChosenAchievementAction
- card: AchievementCard
+ getAchievement(): AchievementCard