Distributed UNO card game clone

Lorenzo Gardini
lorenzo.gardini7@studio.unibo.it

1 luglio 2024

Il progetto ha lo scopo di studiare e presentare una versione distribuita multiplayer del gioco di carte UNO. Il gioco è realizzato utilizzando l'architettura client-server e il paradigma ad attori. Il linguaggio utilzzato per sviluppare l'intero progetto è Scala.

1 Obbiettivi del progetto

L'obbiettivo del progetto è quello di sviluppare una versione distribuita del gioco UNO che possa essere eseguita con Docker ed una connessione ad internet.

Il piano di lavoro è stato diviso nelle seguenti parti:

1. Individuazione del dominio e dei requisiti.
2. Studio e confronto delle tecnologie applicate allo sviluppo di videogiochi distribuiti.
3. Creazione di un modello, basato sulla soluzione trovata grazie al punto precedente, che soddisfi tutti i requisiti.
4. Produzione di documentazione e schemi a supporto dello sviluppo del modello.
5. TDD per una prima stesura di test.
6. Sviluppo dell'applicativo vero e proprio.
7. Alpha-testing su casi reali.

2 Analisi dei requisiti

Nella prima parte dello sviluppo del progetto mi sono occupato di delineare quali fossero le regole del gioco e sono giunto alla decisione di modificare/semplificare alcune meccaniche del gioco (discusse successivamente) per motivi sia pratici che a causa di limitazioni imposte dalle modalità "non in presenza".

2.1 Regole del gioco

In questa parte vengono trattate le regole utilizzate per questo progetto che sono una versione **riadattata** di quelle originali.

UNO è un gioco di carte per 2-7 giocatori. L'obbiettivo del gioco è quello giocare dalla propria mano le carte a disposizione in modo da rimanerne senza. Il primo giocatore che raggiunge questo scopo viene considerato il vincitore. All'inizio di ogni partita a ciascun giocatore vengono consegnate sette carte casuali prese dal mazzo. Successivamente, vengono scoperte dalla cima del mazzo carte fino a che non si scopre un numero che viene posto come prima carta nella pila degli scarti. Le altre carte vengono mescolate con il resto del mazzo. Un giocatore scelto casualmente tra i partecipanti effettua il primo turno, poi si procede in senso orario.Nel suo turno ogni giocatore può:

1. giocare una carta e terminare il turno.
2. pescare una carta. A questo punto deve decidere se giocare la carta pescata oppure passare.
3. dichiarare "UNO" avendo obbligatoriamente due carte in mano e giocare una di quelle carte.

Considerazioni:

1. Ogni volta che si gioca una carta, quest'ultima deve essere compatibile con la prima carta della pila degli scarti (discusso successivamente).
2. La carta giocata diventa la prima carta sulla pila degli scarti.
3. Se il giocatore non ha a disposizione carte adatte da giocare è obbligato a pescare.

Se un giocatore vuole giocare una carta e in quel momento possiede esattamente due carte nella propria mano deve dichiarare "UNO" prima di giocare. In caso si dimentichi, subisce una penalità ed è costretto a pescare due carte dal mazzo.

Il **primo** giocatore resta senza carte vince la partita.

2.2 Carte utilizzate

Il mazzo contiene 108 carte cosi distribuite:

- 19 carte di colore Blu, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Rosso, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Verde, numerate dall'1 al 9 (2 serie) più uno 0
- 19 carte di colore Giallo, numerate dall'1 al 9 (2 serie) più uno 0
- 8 carte *Cambio Giro* dei quattro colori sopracitati

- 8 carte *Salta Turno* dei quattro colori sopracitati
- 8 carte *Pesca +2* dei quattro colori sopracitati
- 4 carte *Cambio Colore*
- 4 carte *Pesca +4*

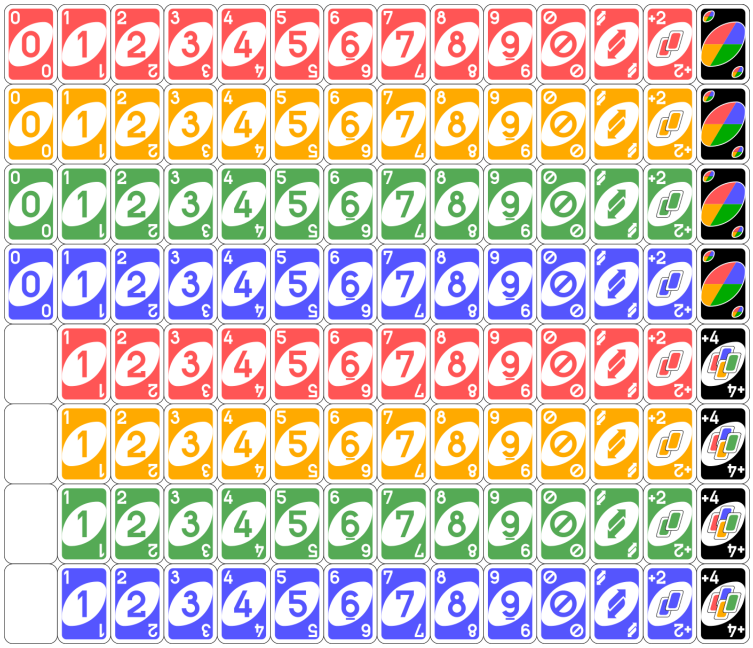


Figura 1: carte utilizzate per creare il mazzo

2.2.1 Regole compatibilità carte

- Una carta colorata può essere sovrapposta ad un'altra se condividono il tipo di carta o hanno lo stesso colore. Per esempio una carta 5 verde può essere giocata solamente se la prima carta nella pila degli scarti è un 5 (stesso tipo) o una carta verde (stesso colore).
- Le carte Jolly (*Pesca Quattro* e *Cambia Colore*) possono essere sempre sovrapposte.

2.2.2 Effetti delle carte

- Carte numero: non hanno nessun effetto
- *Cambio Giro*: inverte l'ordine di gioco. In caso ci siano due giocatori non ha effetto.
- *Salta Turno*: permette di far saltare un turno al giocatore successivo

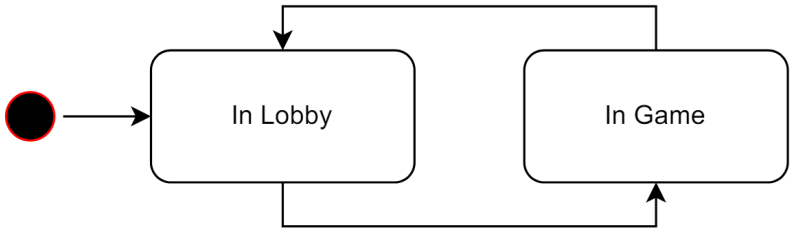
- *Pesca +2*: permette di fare pescare 2 carte al giocatore successivo
- *Cambio Colore* (Jolly): una volta giocata si sceglie un colore. La carta *Cambio Colore* viene considerata del colore scelto.
- *Pesca + 4*: permette di fare pescare 4 carte al giocatore successivo. Ha anche l'effetto della carta *Cambio Colore*.

2.3 Fasi

Come la maggior parte dei giochi multiplayer, il sistema si può trovare in una su due fasi differenti:

- fase di *lobby*
- fase di gioco

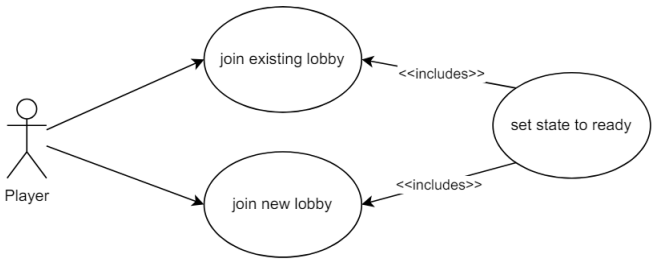
Una volta terminata la fase di gioco il sistema torna alla fase di *lobby*.



2.3.1 Fase di lobby

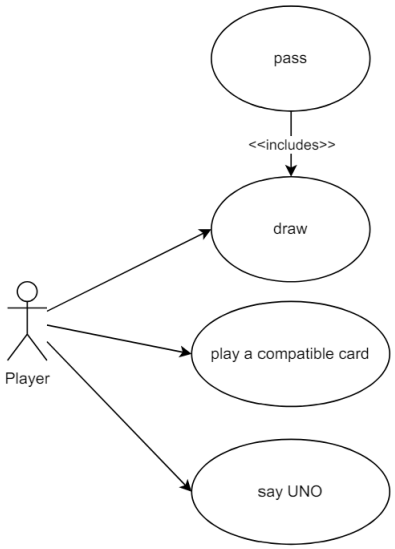
La fase di *lobby* corrisponde ad una sala di attesa che riunisce tutti i giocatori prima dell'inizio della partita. Ogni *lobby* è identificata da un identificatore univoco. Il sistema deve permettere a due giocatori di avere lo stesso username.

La *lobby* di gioco consente ai giocatori di ricevere aggiornamenti sullo stato dei partecipanti. La partita inizia solo quando tutti sono pronti e il numero minimo di giocatori è soddisfatto. Se la *lobby* è piena, nuovi giocatori vengono respinti.



2.3.2 Fase di gioco

Il gioco si svolge seguendo le regole descritte precedentemente. Ogni giocatore visualizza lo stato del gioco e, se è il suo turno, può svolgere azioni. Una volta terminata la partita il sistema torna alla fase *lobby*.



2.4 Gestore delle partite

Il sistema deve essere in grado di poter gestire contemporaneamente diverse *lobby* in base al loro codice. Ogni giocatore, quando si connette, deve poter specificare l'identificatore della *lobby* alla quale vuole collegarsi. Nel caso quella specifica *lobby* non fosse già presente il sistema deve crearla. Se una *lobby* viene abbandonata da tutti i giocatori deve essere rimossa.

2.5 Load Balancer

Deve essere sviluppato anche un semplice sistema di *load balancing*. Il *load balancer* è un componente che ha il ruolo di suddividere il carico di richieste tra i vari *gestori delle partite* allo scopo di migliorare le prestazioni, l'affidabilità e la scalabilità.

Deve poter gestire la registrazione dei vari *gestori delle partite* e deve indirizzare, in base alla *lobby* specificata, ogni giocatore che si collega.

2.6 Scenarios

Un utente può decidere di voler ospitare uno o più *gestori delle partite*, *load balancer* e/o di partecipare come giocatore.

Il sistema deve permettere ai giocatori di creare o partecipare ad una partita. Ogni giocatore, quando si collega, deve poter specificare il nome della *lobby* a cui vuole con-

nettersi e il suo *username* che verrà utilizzato in tutte le fasi di gioco. Ci possono essere *username* duplicati.

Per semplicità, l'interazione dei giocatori con il gioco avverrà lato utente tramite terminale a linea di comando. Qui verranno stampate le varie informazioni e verrà chiesto ai giocatori di effettuare scelte in base alla situazione.

2.7 Self-assessment policy

Per testare la correttezza del progetto verrà utilizzato un approccio **Test Developement Driven** per poter creare e testare i vari componenti in modo che seguano i requisiti specificati. Non è da escludere che ci possano essere interventi di modifica a posteriori o di refactoring in modo da migliorare o modificare il comportamento dei vari componenti.

Successivamente verranno effettuati *alpha tests* per verificare che tutto funzioni correttamente. Nello specifico, si proverà soprattutto l'input e l'ouput dell'utente e le varie interazioni di gioco, difficilmente testabili utilizzando test automatici.

2.8 Requisiti del sistema

In questo progetto devono essere garantiti **consistency** e **availability** del *CUP theorem* ma non viene garantita la **partition tolerance**:

- essendo la partita centralizzata tutti i giocatori avranno come *punti di verità* il *server*. Questo garantisce *consistency*
- il sistema deve gestire la disconnessione dei vari giocatori (come se nella partita usando il gioco fisico un giocatore se ne andasse). In questo caso si assicura **partition tolerance**
- non viene assicurata l'**availability**, in quanto se il server o il *load balancer* andassero in *crash*, tutte le richieste effettuate verso di loro non riceverebbero risposta

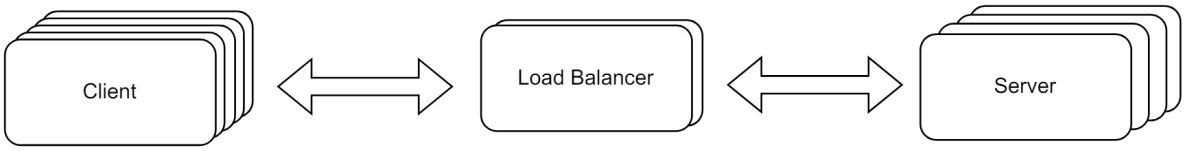
3 Design

3.1 Architettura generale

Il sistema nel complesso utilizza un'architettura **client-server** e può essere suddivisa in 3 macro componenti:

- *client*
- *load-balancer*
- *server*

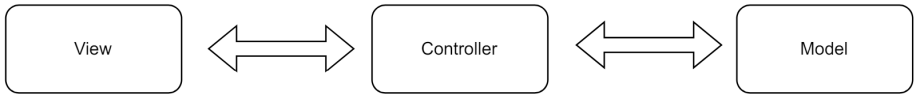
I vari *client* (giocatori) contattano il *load balancer* che a sua volta contatta i *server* (*gestori delle partite*) per instaurare una comunicazione bidirezionale.



Di seguito una descrizione di ogni componente

3.2 Client

Il client rappresenta il singolo giocatore. La sua architettura è stata realizzata usando il pattern *Model-View-Controller*. D'ora in avanti *client* è sinonimo di **giocatore**.



3.2.1 Model

Rappresenta un'interfaccia tra il giocatore e il la logica del **server** tramite una comunicazione bidirezionale:

- Riceve gli input del giocatore e li invia al **server**.
- Riceve gli aggiornamenti dal **server** e li invia al *controller* che a sua volta le manda alla *view* affinché il giocatore possa interagirci

Alla creazione contatta il server utilizzando il nome della *lobby* e l'*username* specificati.
In caso di errori di comunicazione con il **server** contatterà il *controller* che successivamente gli comanderà di stopparsi.
Si comporta come *layer* distribuito lato *client*. Inoltre, implementa il pattern *adapter*, in quanto nasconde completamente al *server* l'utilizzo del metodo di comunicazione optato.

3.2.2 View

Si occupa della presentazione dell'interfaccia utente:

- Durante la fase di *lobby* permette di inviare l'informazione che l'utente è pronto per giocare
- Riceve l'input delle varie scelte dell'utente e le invia al *controller* che a sua volta le manda al *model* affinché il gioco possa aggiornarsi
- Riceve gli aggiornamenti dal *model* e mostra le informazioni al giocatore

In caso di errore dell'interfaccia con il giocatore contatterà il *controller* che successivamente gli comanderà di stopparsi

3.2.3 Controller

Crea *model* e *view* e si pone come collegamento tra di essi. In caso un componente andasse in errore ha il compito di ordinare ad entrambi i componenti di stopparsi per poi anch'esso terminare l'esecuzione.

3.3 Load Balancer

Il *load balancer* si pone come intermediario tra client e server.

Ogni *server* si registra presso il *load balancer* specificando il suo indirizzo, la sua porta e il suo carico massimo. I giocatori non contattano direttamente i *server* ma passano per il *load balancer*. Quest'ultimo ha il seguente comportamento:

- Se il *client* che si connette ha specificato il nome di una *lobby* non ancora creata il *load balancer* sceglie dall'elenco dei *server* quello che ha il rapporto numero giocatori-carico massimo minore.
- Se il *client* che si connette ha specificato il nome di una *lobby* già creata il *load balancer* sceglie tra dall'elenco dei *server* quello che contiene quella *lobby*.
- Una volta che tutti i *client* di una *lobby* si sono disconnessi quest'ultima viene rimossa.
- Se un *client* si connette ma nessun *server* si è registrato, la connessione viene rifiutata.

3.4 Server

Il *server* rappresenta il gioco vero e proprio. Ha il compito di creare e gestire diverse partite in contemporanea. È composto principalmente da 3 macro componenti: *lobbies manager*, *lobby* e *game model*. Nello specifico, ogni componente viene modellato come uno o più attori.

Di seguito una descrizione di ogni componente

3.4.1 Lobbies manager

Ha il compito di accogliere le connessioni dei vari *client* ed effettuano un controllo sul nome della *lobby* da loro specificata:

- se esiste già aggiunge il *client* a quella *lobby*
- se non esiste ne crea una nuova e poi lo aggiunge

Si comporta come *layer* distribuito lato *server*. Inoltre, implementa il pattern *adapter*, in quanto nasconde completamente al *server* l'utilizzo del metodo di comunicazione optato.

3.4.2 Lobby

Rappresenta la *lobby* di una singola partita. Svolge diversi compiti:

- Mantiene lo stato attuale dei *client*.
- Intanto che la partita non è inizia ha il compito di inviare a ogni *client* aggiornamenti sui componenti attuali e sul loro stato.
- Ogni *client* può cambiare, solamente una volta, il suo stato da *not ready* a *ready* comunicando agli altri *client* che è pronto per iniziare.
- Se un *client* abbandona la *lobby* lo stato di ogni *client* viene impostato a *not ready*
- Se tutti i *client* in *lobby* sono *ready* ed è presente il numero minimo di *client* (in questo caso 2) la partita ha inizio.
- Se un *client* si connette ma la *lobby* ha raggiunto il numero massimo di *client* (in questo caso 7) la richiesta del *client* viene rifiutata
- Se un *client* si connette durante la partita ma la *lobby* ha raggiunto il numero massimo di *client* (in questo caso 7) la richiesta del viene rifiutata.
- Se un *client* si connette durante la partita e la *lobby* non ha raggiunto il numero massimo di *client* (in questo caso 7), la richiesta viene accettata e viene comunicato al nuovo *client* che la partita in corso non è ancora terminata e che deve attendere. Una volta terminata il *client* viene gestito come tutti gli altri.
- Se la partita in corso termina per qualunque motivo ogni giocatore viene riannesso alla *lobby* con stato *not ready*
- Se un *client* si disconnette mentre è in corso una partita questa viene terminata prematuramente tornando in *lobby*.

3.4.3 Game model

Modella la partita vera e propria. Riceve gli aggiornamenti, le scelte effettuate dai giocatori, si aggiorna e comunica questi aggiornamenti ai vari *client*.

Le informazioni globali:

- L'azione effettuata con successo dal giocatore di turno (e.g. carta giocata, aver passato etc)
- Un giocatore si dimentica di dire "UNO" prima di giocare
- Il motivo del *game over*

Le informazioni private:

- Lo stato attuale del giocatore. Questo comprende sia informazioni personali (le carte che possiede) sia informazioni globali (la prima carta nella pila degli scarti, l'ordine di turno, il numero di carte posseduto da ciascun giocatore)

La partita termina nei seguenti casi:

- Un giocatore vince
- Il mazzo termina. Questo scenario si verifica solamente se ogni giocatore durante il suo turno non gioca carte ma pesca.
- La partita non può essere creata

A partita terminata:

1. Viene mostrata la motivazione del termine della partita
2. Ogni giocatore viene riannesso alla *lobby* con stato *not ready*

3.5 Scalabilità

Così strutturato il sistema è in grato di poter ospitare un numero potenzialmente molto alto di partite. Questo perché in primis ogni partita richiede poche risorse computazionali, quindi un *server* moderno anche non molto performante può gestirne un numero elevato di *lobbies*. Inoltre, è possibile creare più *server* in più nodi computazionali diversi e, grazie all'aiuto di qualche *load balancer*, si può distribuire il carico equamente. In questo modo non solo si può assicurare una buona *availability*, ma anche la possibilità di reggere e gestire un'utenza corposa. Bisogna anche aggiungere che essendo così granulare l'architettura può essere modulata a necessità: in caso di pochi utenti si possono usare poche repliche del server e in caso di necessità aumentarle.

4 Dettagli implementativi

4.1 Storico di design e implementazioni

Inizialmente si era pensata una soluzione **client-server** realizzato con **TuCSoN**. Il *server* conteneva diversi *spazi di tuple*: uno per memorizzare le richieste di creazione delle partite, uno per le lobby, uno per ogni partita per permettere uno scambio di messaggi ai giocatori. Questa soluzione è risultata impraticabile per via della complessità nella gestione delle connessioni e della gestione di *spazi di tuple* e della lentezza che avrebbero causato le *primitive* chiamate su di essi.

La seconda soluzione pensata riguardava l'utilizzo di **Akka Cluster** per una versione *P2P* del gioco. L'idea era quella di non avere un *server*, ma che ogni giocatore potesse sincronizzarsi con gli altri grazie a messaggi *broadcast*. Questa soluzione non è risultata adatta, in quanto solamente un giocatore può iniziare la partita e, in certi scenari, il sistema si sarebbe trovato in uno stato *inconsistente* o *partizionato*. Inoltre non mostrava

correttamente come sviluppare un sistema distribuito, dato che l'infrastruttura sottostante svolgeva buona parte del lavoro, soprattutto nella gestione dello scambio remoto di messaggi. Inoltre, diverse pubblicazioni sullo sviluppo di giochi distribuiti mostrano come sviluppare giochi 100% *P2P* è difficile a causa della totale distribuzione non avendo né un coordinatore stabile (potrebbe andare in *crash* e bisognerebbe rieleggerne un altro) né un unico punto di verità (come invece accade per una soluzione che comprende un *server*).

4.2 Implementazione corrente

Per la realizzazione dell'intero progetto si è scelto:

- come linguaggio di programmazione **Scala 2**
- come *framework* per l'implementazione degli attori **Akka-Scala**
- come *framework* per connessioni **Scala Vertx**
- come *framework* per i test **ScalaTest**
- come *build tool* **sbt**
- per la conteinerizzazione **Docker**

Ogni componente viene modellato usando il paradigma ad *attori* grazie ad **Akka**:

- Ogni *attore* è un'entità indipendente che possiede un proprio stato interno e può eseguire operazioni autonome. Gli attori sono isolati e comunicano solo attraverso il passaggio di messaggi.
- gli *attori* comunicano inviandosi messaggi asincroni. Un *attore* può inviare un messaggio a un altro attore, specificandone l'indirizzo
- ogni attore ha un indirizzo unico attraverso il quale può essere identificato.
- lo stato di un *attore* è protetto e può essere modificato solo dall'*attore* stesso
- gli attori sono responsabili della gestione dei propri errori.

Questa soluzione ha reso la modellazione molto più lineare ed è stato possibile dividere facilmente in moduli l'intero sistema.

4.3 Divisione delle componenti in moduli

Il progetto è diviso in quattro moduli principali: **common**, **server**, **client** e **load balancer**

4.4 WebSocket

Web-socket è un protocollo di comunicazione che consente una connessione bidirezionale e full-duplex tra un *client* e un *server*. È utile per applicazioni che richiedono aggiornamenti in tempo reale e bidirezionale, come in questo caso.

Come funziona:

1. **handshake iniziale**: la connessione inizia con un *handshake* HTTP. Il *client* invia una richiesta HTTP al *server* con un campo `Upgrade` che indica la richiesta di aggiornamento della connessione a *web-socket*. Se il server supporta WebSocket, risponde con un messaggio di accettazione, completando così l'upgrade.

2. **comunicazione continua**: dopo l'handshake, la connessione viene mantenuta aperta. Il client e il server possono quindi scambiarsi messaggi (testuali o binari) in modo asincrono e bidirezionale (Message-oriented Communication).

3. **ping/pong e timeout**: per mantenere viva la connessione e rilevare eventuali disconnessioni, si utilizzano i frame *ping/pong*. Il *client* o il *server* possono inviare un *frame ping*, al quale l'altra parte risponde con un *frame pong*. Se non viene ricevuto un *frame pong* entro un certo intervallo di tempo, si può considerare che la connessione sia interrotta.

4. **chiusura della connessione**: la connessione *web-socket* può essere chiusa da entrambe le parti tramite un *frame di chiusura*. Questo processo avviene in modo ordinato, permettendo di completare eventuali operazioni pendenti prima della disconnessione definitiva.

Vantaggi:

- **comunicazione bidirezionale**: permette una comunicazione bidirezionale continua.

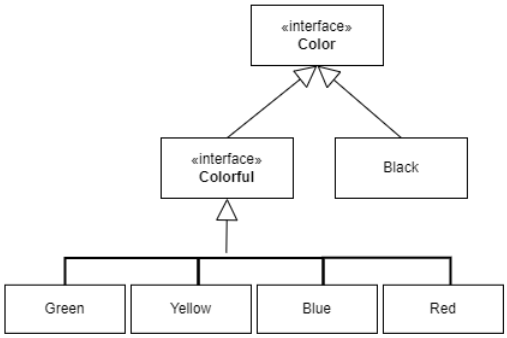
- **efficienza**: una connessione rimane aperta, eliminando il sovraccarico e la latenza di dover stabilire nuove connessioni per ogni messaggio.

In questo progetto viene preferito questo protocollo rispetto a REST per via della necessità di una comunicazione non vincolata tra *client* e *server*. Inoltre, permette di gestire più facilmente problemi di disconnessioni improvvise delle varie componenti. I messaggi mandati e ricevuti sono JSON in forma testuale contenenti le informazioni scambiate tra gli attori.

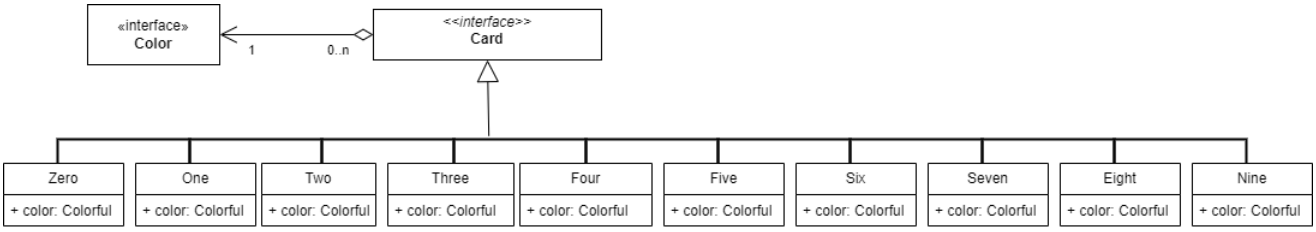
4.5 Common

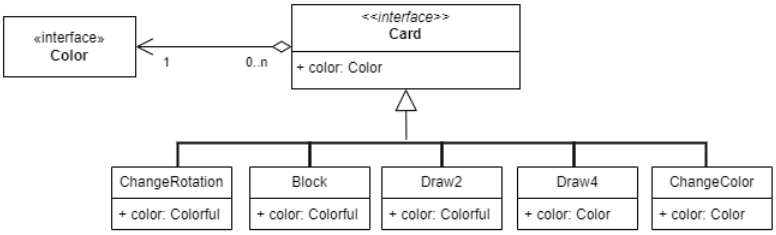
Questo modulo contiene tutti quegli elementi comuni in tutto il progetto. Spazia da funzioni di utilità a elementi di gioco a strutture dati. Le entità più importanti sono:

- *trait* **Color**: rappresenta il colore della carta. Viene estesa dal colore **Black** e dal *trait* **Colorful** le cui implementazioni sono i 4 colori di gioco: **Green**, **Red**, **Blu**, **Yellow**

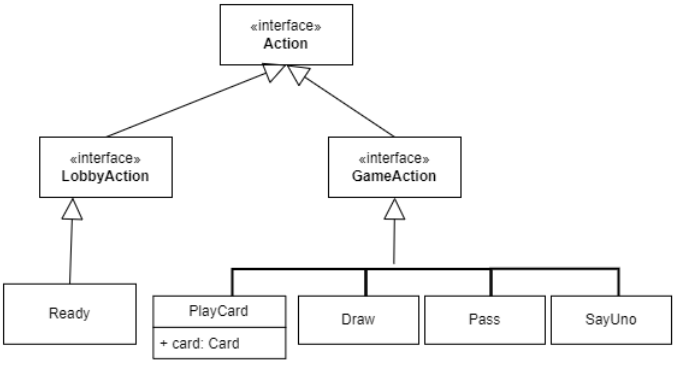


- *trait* **Card**: rappresenta una carta di gioco e come attributo ha un **Color**. Le sue implementazioni sono tutte le possibili carte (**Draw4**, **ChangeColor**, **ChangeRotation**, **Draw2**, **Block**, **Zero**, **One**, **Two**, **Three**, **Four**, **Five**, **Six**, **Seven**, **Eight**, **Nine**)

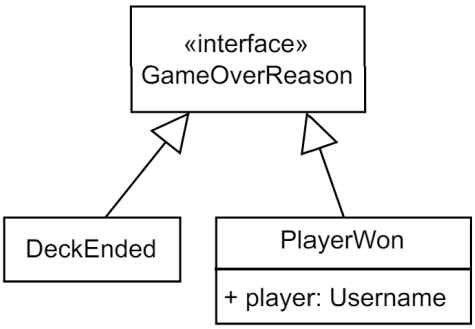




- *trait* **Action**: rappresenta un azione del giocatore. È diviso in *trait* **LobbyAction**, che comprende solo **Ready** per cambiare stato da *not ready* a *ready* in *lobby*, e nel *trait* **InGameAction** che comprende tutte le azioni di gioco:
  - **PlayCard**: l'azione di giocare una carta
  - **Draw**: l'azione di pescare
  - **Pass**: l'azione di passare
  - **SayUno**: l'azione di dire "UNO"



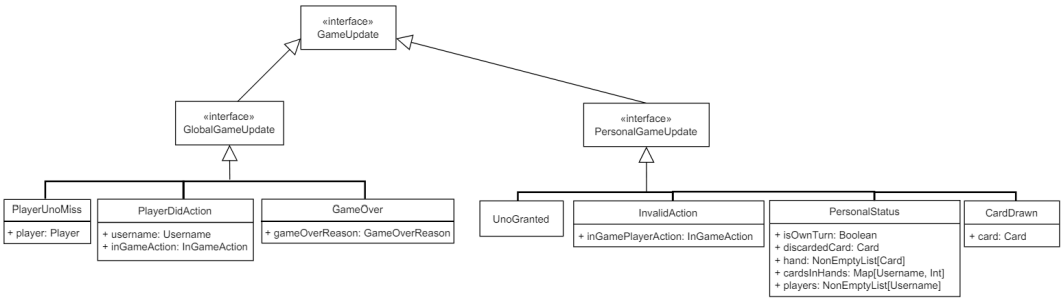
- *trait* **GameOverReason**: rappresenta la ragione di fine partita. Le sue implementazioni sono:
  - **DeckEnded**: fine partita a causa della fine del mazzo
  - **PlayerWon**: fine partita a causa della vittoria di un giocatore



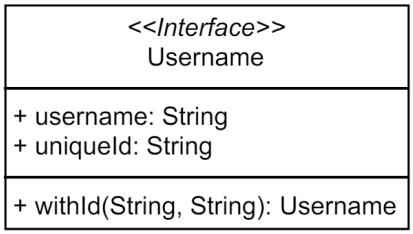
- *trait* **GameUpdate**: rappresenta un aggiornamento dal parte del *model*. È diviso in *trait* **GlobalGameUpdate**, che rappresenta tutti quegli aggiornamenti da inviare a ogni giocatore e le sue implementazioni sono:
  - **PlayerUnoMiss**: il giocatore di turno si è dimenticato di dire "UNO" quando aveva due carte in mano
  - **PlayerDidAction**: il giocatore ha effettuato una mossa (una **InGameAction**)
  - **GameOver**: il gioco è terminato

  e dal *trait* **PersonalGameUpdate** che rappresenta un aggioramento personale di per ciascun giocatore e le sue implementazioni sono:
  - **UnoGranted**: il permesso di poter giocare una carta dopo aver detto "UNO"
  - **InvalidAction**: un'azione invalida da parte di un giocatore
  - **CardDrawn**: la carta pescata con l'azione **Draw**
  - **PersonalStatus**: lo status personale del giocatore. È composto da:

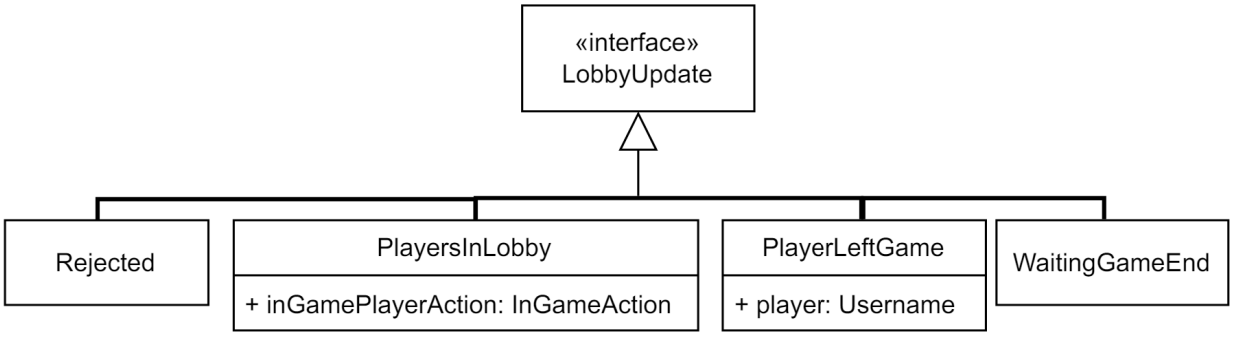
* informazione se è il proprio turno o no
* l'ultima carta nella pila degli scarti
* la propria mano
* il numero di carte in mano a ogni giocatore
* l'ordine di turno



- *trait* **Username**: rappresenta il nome del giocatore in modo univoco



- *trait* **LobbyUpdate**: rappresenta un aggiornamento della *lobby*. Le sue implementazioni sono:
  - **PlayersInLobby**: l'elenco di tutti i giocatori in *lobby* e se sono nello stato *ready* o *not ready*
  - **Rejected**: il giocatore è stato rifiutato perchè la *lobby* è piena
  - **WaitingGameEnd**: il giocatore è stato messo in attesa che la partita corrente finisca
  - **PlayerLeftGame**: un giocatore ha abbandonato la partita

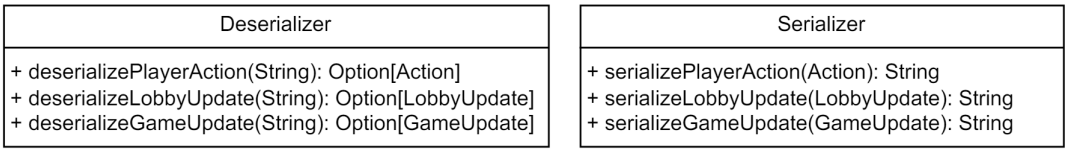


In questo modulo contiene anche tutta la parte che riguarda la *serialization-deserialization* dei messaggi da mandare in rete, chiamata **presentation**.

Le entità serializzabili e deserializzabili sono:

- **Action**
- **LobbyUpdate**
- **GameUpdate**

Ogni entità serializzata è in realtà un *Option* di quest'ultima in quanto non è detto che la serializzazione sia andata a buon fine.

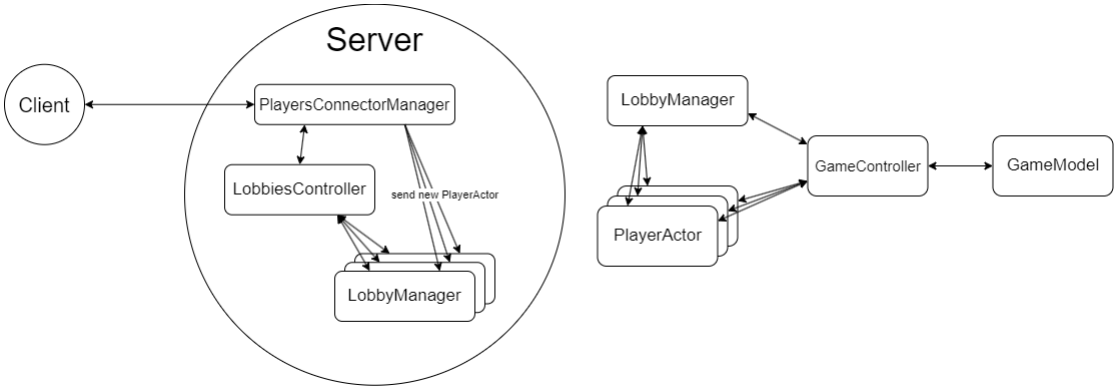


Infine, tramite l'utilizzo degli impliciti, sono stati aggiunti alla **NonEmptyList**, che rappresenta una lista generica non vuota, metodi per scorrerla ed invertirla modellando quindi un l'ordine di gioco.

4.6 Server

Questo modulo contiene gli elementi del *server* e di tutta la gestione di *lobby* e partite. Il *server* è composto dalla coordinazione di diversi attori: **GameModel**, **GameController**, **LobbiesController**, **LobbyManager**, **PleyerActor**, **PlayersConnectorManager**.

La struttura e tutti i collegamenti possono essere rappresentati con la seguente figura:



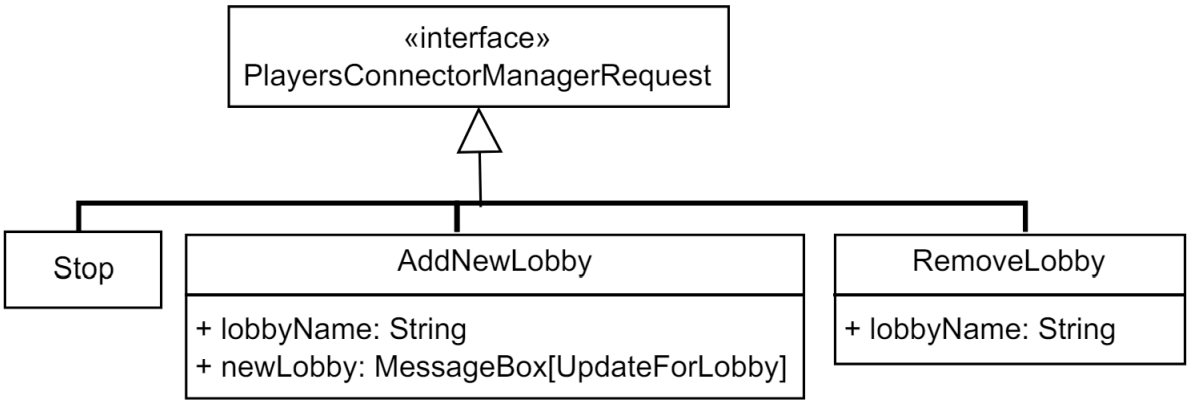
4.6.1 PlayersConnectorManager

Attore che ha il compito di accettare le nuove connessioni di tipo *web-socket* da parte dei giocatori e indirizzarli nella *lobby* specificata o chiedere al **LobbiesController** di crearla nel caso non esistesse.

Riceve messaggi di tipo **PlayersConnectorManagerRequest** mandati dal **LobbiesController**:

- **AddNewLobby(lobby)**: aggiunge la *lobby* specificata tra quelle disponibili
- **RemoveLobby(lobby)**: rimuove la *lobby* specificata da quelle disponibili
- **Stop**: manda un messaggio di **PlayersConnectorManagerStopped** al **LobbiesController**, termina tutte le connessioni e poi si stoppa

Se, in qualunque momento, una connessione generasse un errore, il **PlayersConnectorManager** manda un messaggio di **PlayersConnectorManagerError** al **LobbiesController**.

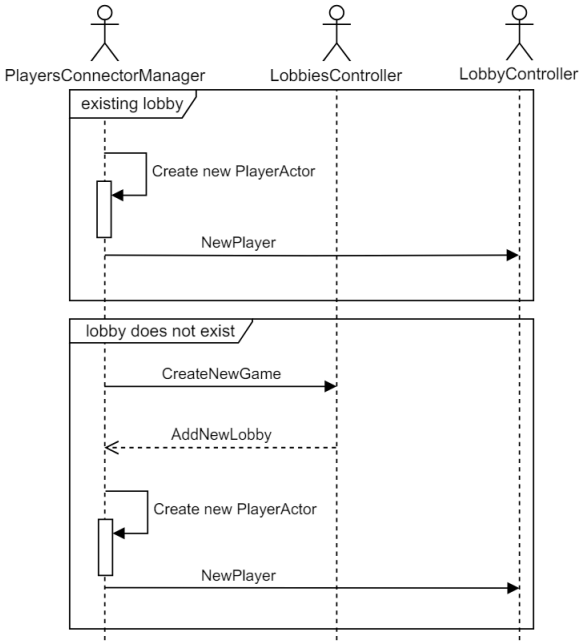


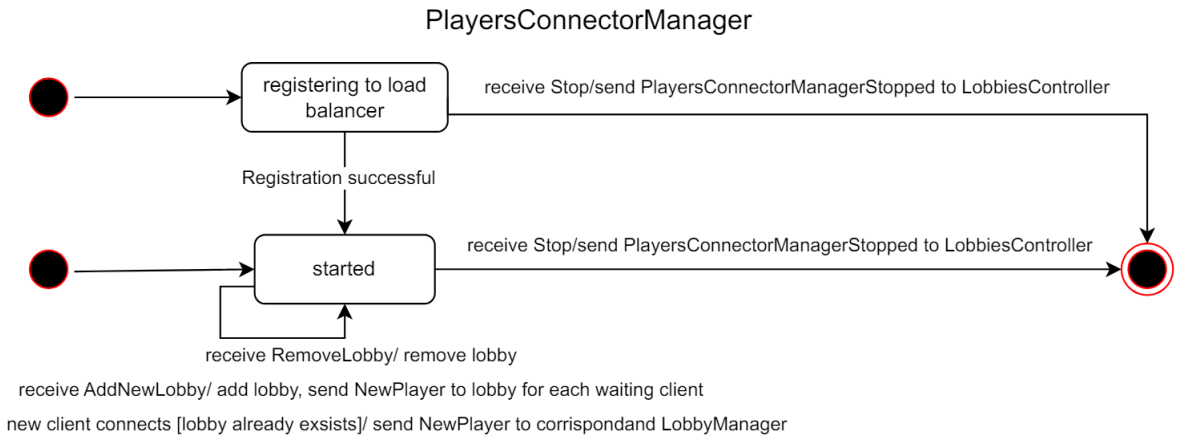
Quando arriva una nuova connessione e la *lobby* esiste:

1. controlla internamente se una *lobby* associata al nome specificato esiste. Esiste
2. crea un nuovo **Username** con l'username specificato
3. crea un nuovo **PlayerActor** e lo invia al **LobbyManager** corrispondente con un messaggio di **NewPlayer**

Quando arriva una nuova connessione e la *lobby* non esiste:

1. controlla internamente se una *lobby* associata al nome specificato esiste. Non esiste.
2. chiede al **LobbiesController** di creare una nuova *lobby* con il messaggio **CreateNewGame** specificando il nome della *lobby*.
3. crea un nuovo **Username** con l'username specificato e lo aggiunge nella lista dei giocatori che stanno aspettando la creazione di quella *lobby*
4. quando riceve il messaggio **AddNewLobby(lobby)** dal **LobbiesController** effettua il punto 3. del caso di *lobby* esistente.
5. aggiunge la nuova *lobby* tra quelle disponibili.





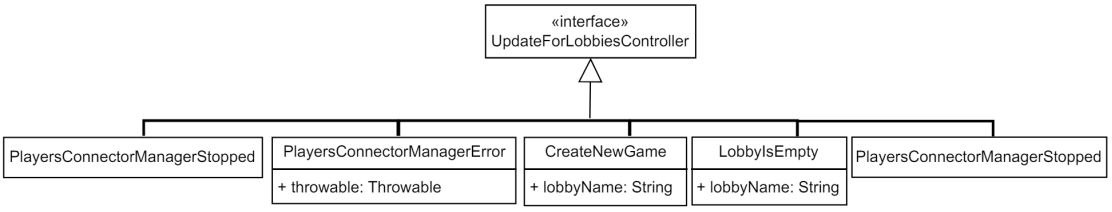
Internamente il *PlayerActor* contiene un riferimento alla *web-socket* del *client*, che usa sia per ricevere che per mandare i messaggi ricevuti. In questa parte viene usato il modulo della *presentation*.

4.6.2 LobbiesController

Attore che ha il compito di creare e gestire tutte le *lobby*.

Riceve messaggi di tipo **UpdateForLobbiesController**:

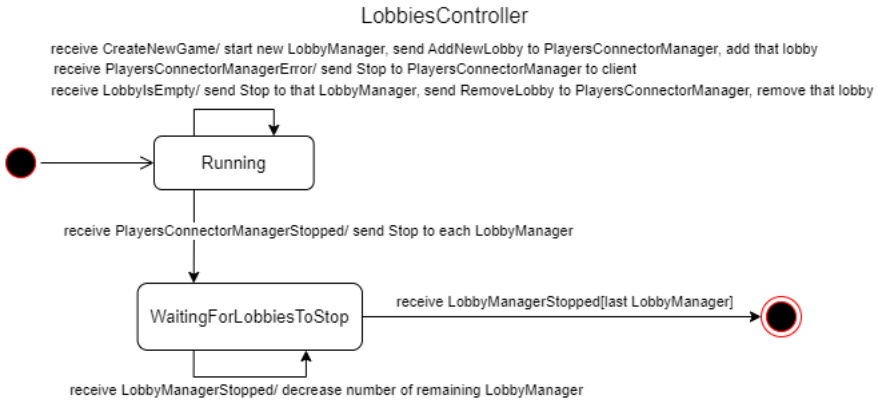
- **PlayersConnectorManagerStopped**: viene mandato dal **PlayersConnectorManager** informando il **LobbiesController** che è terminato
- **PlayersConnectorManagerError**: viene mandato dal **PlayersConnectorManager** informando il **LobbiesController** che c'è stato un errore
- **CreateNewGame**: viene mandato dal **PlayersConnectorManager** chiedendo al **LobbiesController** di creare una nuova *lobby* con in nome specificato
- **LobbyIsEmpty**: viene mandato dal **LobbyManager** informando il **LobbiesController** che quella specifica *lobby* è vuota e deve essere rimossa
- **LobbyManagerStopped**: viene mandato dal **LobbyManager** informando il **LobbiesController** che è terminato



Comportamento generale:

- La prima cosa che svolge è creare il **PlayersConnectorManager**
- Quando il **PlayersConnectorManager** gli manda un messaggio di **CreateNewGame** lui crea una nuova istanza di **LobbyManager** e risponde **PlayersConnectorManager** con il messaggio di **AddNewLobby** con l'indirizzo di quest'ultima
- Quando riceve il messaggio di **LobbyIsEmpty** dal **LobbyManager** manda un messaggio di **RemoveLobby** al **PlayersConnectorManager** in modo da fargli rimuovere quella *lobby* da quelle disponibili. Inoltre, manda un messaggio di **Stop** a quello specifico *LobbyManager* in modo da farlo terminare ricevendo in risposta un messaggio di **LobbyManagerStopped**

Se riceve un messaggio di **PlayersConnectorManagerError** da parte del **PlayersConnectorManager** allora si inizia il processo di stop di tutto il *server*. Il **LobbiesController** manda un messaggio di **Stop** al **PlayersConnectorManager** che prima di terminare risponde con un **PlayersConnectorManagerStopped**. Una volta ricevuto quest'ultimo messaggio il **LobbiesController** si stoppa.



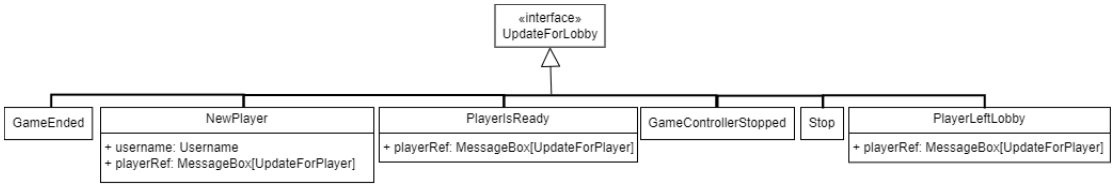
4.6.3 LobbyManager

Attore che ha il compito di creare e gestire la singola *lobby* con tutti i partecipanti.

Riceve messaggi del tipo **UpdateForLobby**:

- **NewPlayer**: messaggio che viene mandato dal **PlayersConnectorManager** quando si connette un nuovo giocatore
- **PlayerLeftLobby**: messaggio mandato dal **PlayerActor** nel caso il *client* si disconnette
- **PlayerIsReady**: messaggio mandato dal **PlayerActor** nel caso il **client** fosse **ready**
- **GameEnded**: messaggio mandato dal **GameController** nel caso la partita fosse finita

- **Stop**: messaggio mandato dal **LobbiesController** ordinando al **LobbyManager** di interompere la sua esecuzione
- **GameControllerStopped**: messaggio mandato dal **GameController** per informare il **LobbyManager** della sua terminazione



Comportamento in *lobby*:

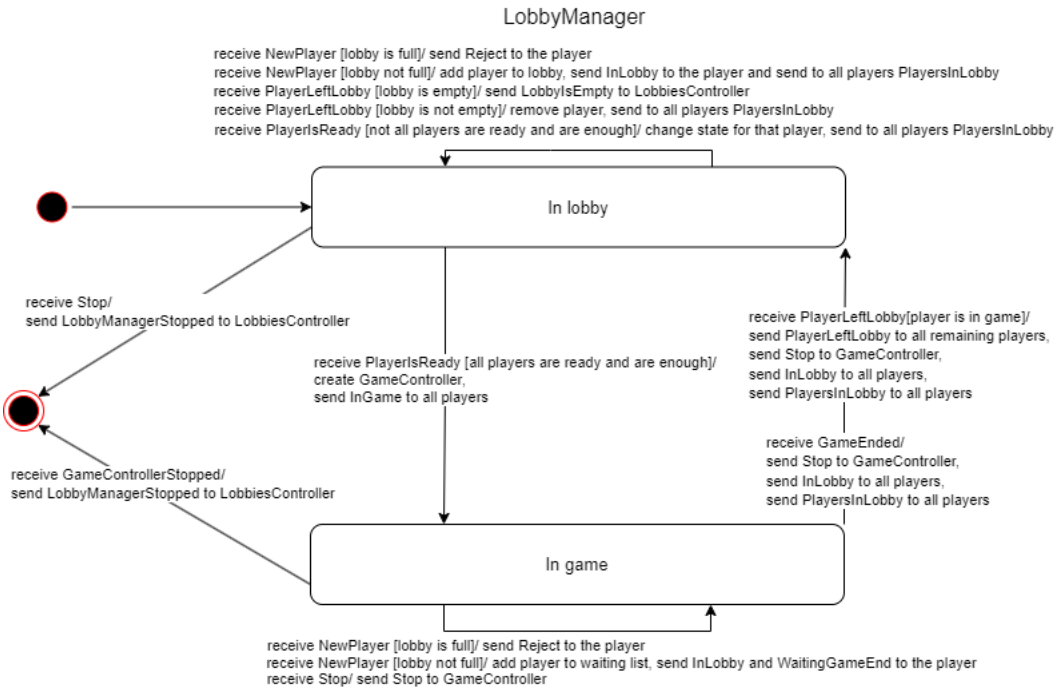
- quando riceve un messaggio di **NewPlayer** controlla se il numero di giocatori attualmente in *lobby* è minore uguale al numero massimo (in questo caso 7). Se ci sono posti disponibili, aggiunge il giocatore a quelli disponibili, gli manda un messaggio di **InLobby** e poi manda un messaggio di **UpdateFromLobby(PlayersInLobby)** a tutti i giocatori. Altrimenti lo rifiuta mandandogli un messaggio di **UpdateFromLobby(Rejected)**
- quando riceve un messaggio di **PlayerLeftLobby** rimuove il giocatore da quelli disponibili, manda un messaggio di **UpdateFromLobby(PlayersInLobby)** a tutti i giocatori e, in caso non ci fossero più giocatori rimasti, manda un messaggio di **LobbyIsEmpty** al **LobbiesController**
- se riceve un messaggio di **Stop** manda un messaggio di **Stop** al **GameController**. Una volta ricevuto la conferma che il **GameController** sia terminato ricevendo il messaggio **GameControllerStopped** allora manda un messaggio di **LobbyManagerStopped** al **LobbiesManager** e poi termina.
- quando riceve un messaggio di **PlayerIsReady** da parte di un giocatore cambia il suo stato da *not ready* a *ready*. Se tutti i giocatori in lobby sono in stato *ready* e sono almeno il numero minimo (in questo caso 2), cambia stato in *game*

Una volta che tutti i giocatori sono pronti crea un'istanza di **GameController** che dirigerà la comunicazione **GameModel-PlayerActor**

Comportamento in *game*:

- in caso di **NewPlayer**, controlla se il numero di giocatori attualmente in partita è minore uguale al numero massimo (in questo caso 7). Se ci sono posti disponibili, aggiunge il giocatore ai giocatori in attesa di entrare in *lobby*, gli manda sia un messaggio di **InLobby** che di **UpdateFromLobby(WaitingGameEnd)**. Altrimenti lo rifiuta mandandogli un messaggio di **UpdateFromLobby(Rejected)**

- se riceve un messaggio di **Stop** manda un messaggio di **Stop** al **GameController** e poi termina la sua esecuzione
- se riceve un messaggio di **GameEnded** manda un messaggio di **Stop** al **GameController**, manda un messaggio di **InLobby** a tutti i giocatori, sia quelli che stanno aspettando che quelli che erano in partita e poi cambia stato tornando in **lobby**
- se un giocatore lascia la partita con un messaggio di **PlayerLeftLobby** controlla se il giocatore che ha abbandonato la partita fa parte dei giocatori in attesa oppure di quelli in partita. Nel primo caso semplicemente il giocatore viene rimosso dai giocatori in attesa. Nel secondo caso, ogni giocatore riceve un messaggio di **UpdateFromLobby(PlayerLeftGame)** e si procede nello stesso modo del messaggio di **GameEnded**

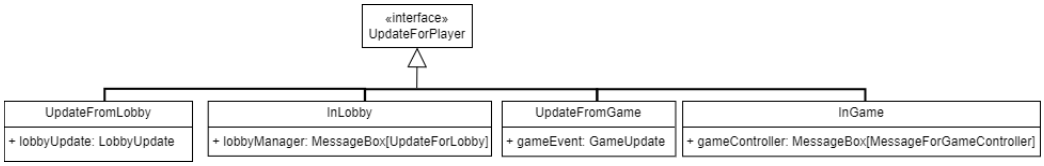


4.6.4 PlayerActor

Attore che ha il compito di creare un'interfaccia tra il giocatore lato *client* e la sua rappresentazione lato *server* tramite una connessione *web-socket*. Internamente, riceve i messaggi dal *client*, li deserializza e li invia, in base al tipo di messaggio, al **LobbyManager** o al **GameModel** e serializza i messaggi di questi ultimi mandandoli al *client* con l'operazione contraria.

Riceve messaggi del tipo **UpdateForPlayer**:

- **UpdateFromLobby**: aggiornamento, da parte del **LobbyManager**, come **LobbyUpdate**
- **InLobby**: ricevuto dal **LobbyManager**, informa il **PlayerActor** che è stato accettato nella *lobby*
- **UpdateFromGame**: aggiornamento, da parte del **GameModel**, come **GameUpdate**
- **InGame**: ricevuto dal **LobbyManager**, informa il **PlayerActor** che la partita è iniziata



Appena avviato il **PlayerActor** può essere accettato in *lobby*, tramite il messaggio **InLobby**, oppure rifiutato, con un **UpdateFromLobby(Rejected)**. Nel caso il giocatore venisse rifiutato il **PlayerActor** chiude la connessione e termina la sua esecuzione.

Ci sono casi in cui il giocatore viene accettato in *lobby* ma la partita è già iniziata. In quel caso riceve un messaggio di **UpdateFromLobby(WaitingGameEnd)** per avvisarlo di attendere. Questa informazione viene mandata al *client* tramite la *web-socket*.

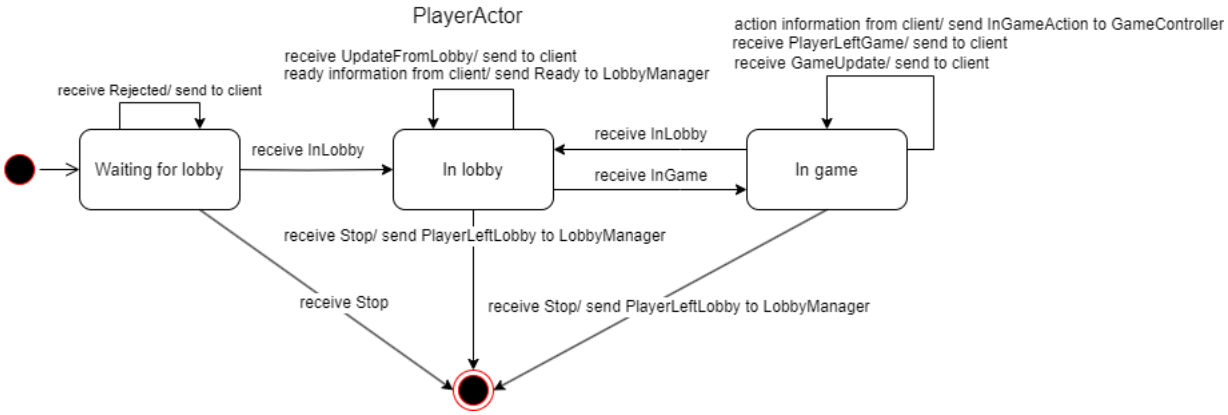
Comportamento in *lobby*:

- in caso di disconnessione da parte del *client* l'attore avverte il **LobbyManager** che ha lasciato la *lobby*, con un messaggio di **PlayerLeftLobby**, e poi termina la sua esecuzione
- in caso ricevesse un messaggio dalla *web-socket* che indica che il **client** sia *ready*, manda al **LobbyManager** un messaggio di **PlayerIsReady**
- il **PlayerActor** invia al **client** tramite la *web-socket* tutti i messaggi di **UpdateFromLobby(PlayersInLobby)** mandati dal **LobbyManager**
- se riceve un messaggio di **InGame** cambia stato per poter ricevere i messaggi di tipo **UpdateFromGame**

Comportamento in *game*:

- anche in questo caso se il *client* si disconnette l'attore avverte il **LobbyManager** che ha lasciato la *lobby*, con il messaggio di **PlayerLeftLobby**, e poi termina la sua esecuzione
- invia tramite *web-socket* tutti i messaggi di **UpdateFromGame** e di **UpdateFromLobby(PlayerLeftGame)** che riceve

- inoltra tutti i messaggi di tipo **InGamePlayerAction**, ricevuti dalla *web-socket*, al **GameController**
- torna nello stato *lobby* se riceve un messaggio di **InLobby** (la partita è per qualche ragione terminata)

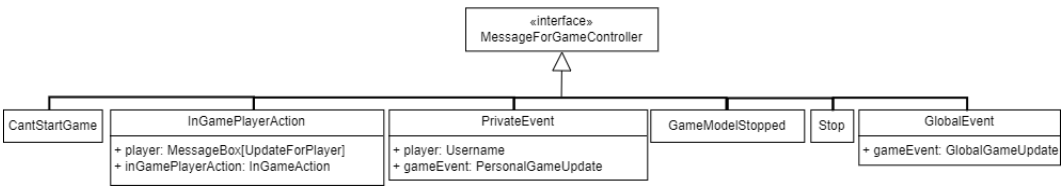


4.6.5 GameController

Attore che ha il compito di porsi come intermediario tra il **GameModel** e i vari **PlayerActor** che stanno partecipando alla partita.

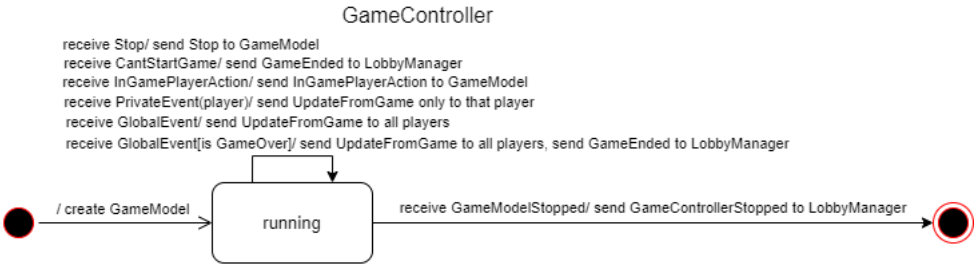
Riceve messaggi del tipo **MessageForGameController**:

- **GlobalEvent(GlobalGameUpdate)**: mandato dal **GameModel**, rappresenta un messaggio da mandare a tutti giocatori
- **PrivateEvent(PersonalGameUpdate)**: mandato dal **GameModel**, rappresenta un messaggio da mandare ad un giocatore specifico
- **InGamePlayerAction(InGameAction)**: mandato da un **PlayerActor**, rappresenta una mossa effettuata da quel giocatore
- **CantStartGame**: mandato dal **GameModel** è un errore nell'avvio della partita
- **Stop**: mandato dal **LobbyManager** è un ordine di interrompere la sua esecuzione
- **GameModelStopped**: mandato dal **GameModel** per informarlo della sua terminazione



Appena avviato, crea una istanza di **GameModel**.
Comportamento generale:

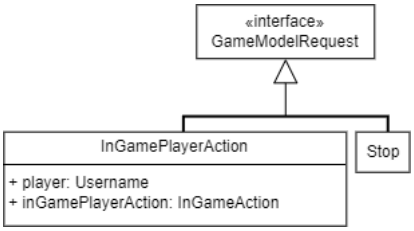
- ogni volta che riceve un messaggio di **InGamePlayerAction** lo inoltra al **GameModel**
- ogni **PrivateEvent** viene mandato solamente al giocatore corrispondente
- ogni **GlobalEvent** viene mandato a tutti i giocatori. Nel caso fosse un **GameOver** invia un messaggio di **GameEnded** al **LobbyManager**
- se riceve un messaggio di **CantStartGame** manda un messaggio di **GameEnded** al **LobbyManager**
- se riceve un messaggio di **Stop** invia un messaggio di **Stop** al **GameModel**. Una volta ricevuto il messaggio **GameModelStopped** dal **GameModel** manda un messaggio di **GameControllerStopped** al **LobbyManager** per poi stopparsi



4.6.6 GameModel

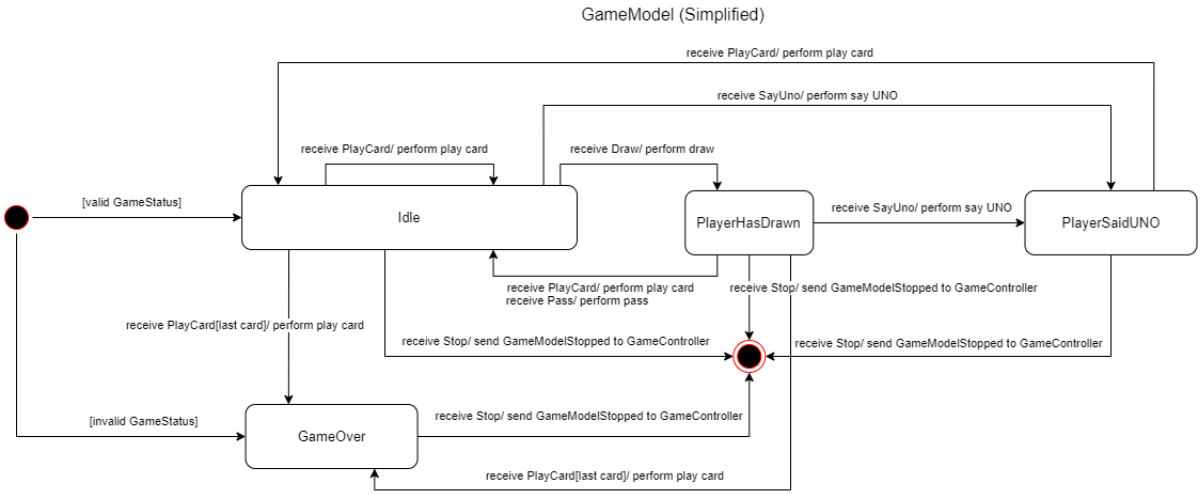
Attore che rappresenta la partita vera e propria. Mantiene le informazioni sullo stato del gioco e dei giocatori.
Riceve messaggi del tipo **GameModelRequest**:

- **InGamePlayerAction(InGameAction)**: messaggio dal **GameController** è l'azione effettuata da un giocatore
- **Stop**: messaggio dal **GameController** che gli ordina di stopparsi



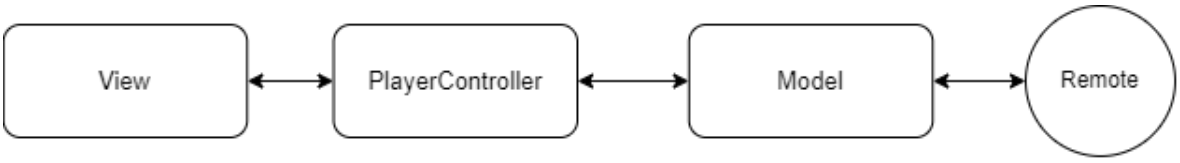
All'avvio cerca di creare uno stato di partita consistente, se non riesce manda un messaggio di **CantStartGame** al **GameController**
Comportamento generale:

- quando riceve un messaggio di **InGamePlayerAction(InGameAction)** verifica se l'azione è legale. In caso non lo fosse manda al **GameController** un messaggio di **PrivateEvent(InvalidAction)**. Nel caso fosse legale si aggiorna, cambia stato e invia, dipendentemente dall'azione svolta dal giocatore, una serie di **PrivateEvent** e **GlobalEvent** contenenti tutti gli aggiornamenti della partita
- quando un giocatore vince o il mazzo termina manda un messaggio di **GlobalEvent(GameOver)** al **GameController** che poi provvederà a mandargli il messaggio di **Stop**
- se riceve il messaggio di **Stop** manda un messaggio di **GameModelStopped** al **GameController** e termina la sua esecuzione. Si ricorda che questo messaggio viene propagato dal **LobbyManager** fino al **GameModel** anche nel caso in cui un giocatore dovesse lasciare la partita



4.7 Client

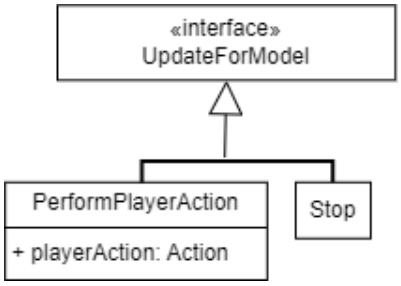
Questo modulo contiene gli elementi del *client*. Il *client* è composto dalla coordinazione di diversi attori: **Model**, **PlayerController**, **View**. La struttura e tutti i collegamenti possono essere rappresentati con la seguente figura:



4.7.1 Model

Attore che rappresenta il collegamento con il **server** incapsulando la distribuzione della connessione *web-socket*. Invia le azioni generate dall'utente e riceve gli aggiornamenti. Riceve messaggi del tipo **UpdateForModel**:

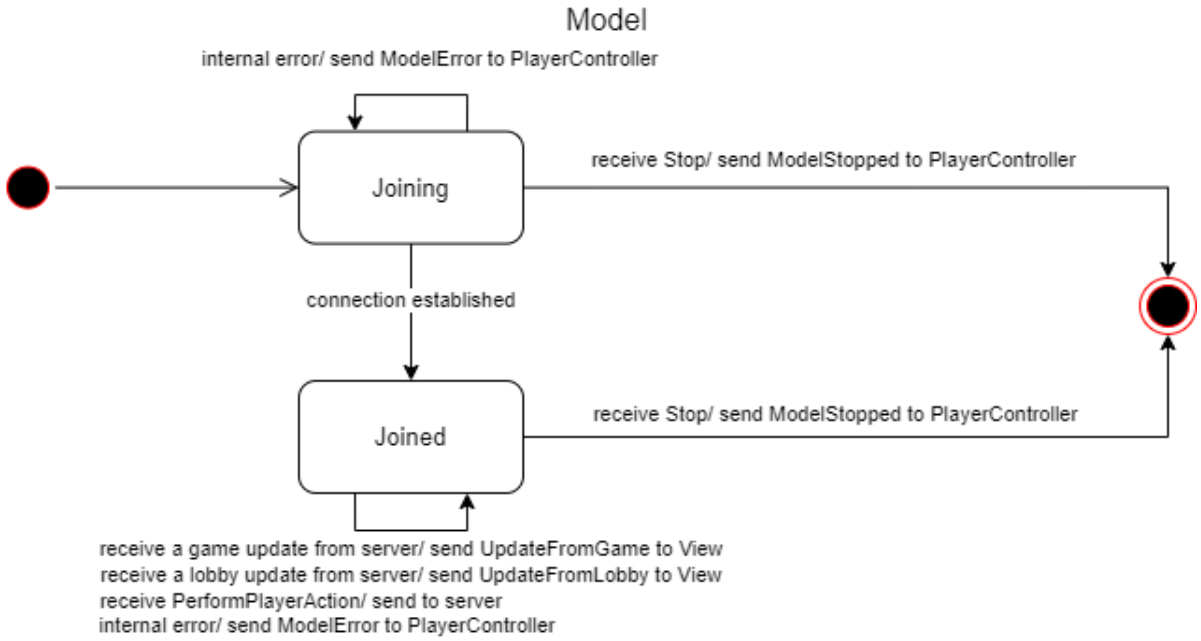
- **PerformPlayerAction**: inviato dalla **View** rappresenta un'azione del giocatore da inviare al *server*
- **Stop**: inviato dal **PlayerController** ordinando la sua terminazione.



All'avvio cerca di connettersi con il *server* specificato. In caso non fosse raggiungibile o ci fossero dei problemi manda un messaggio di **ModelError** al **PlayerClient**.

Comportamento generale:

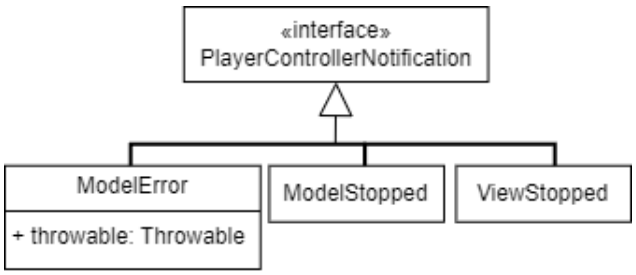
- quando riceve un messaggio di **PerformPlayerAction** da parte della **View** lo serializza e lo invia al *server*
- quando riceve un messaggio di **Stop** termina la sua istanza di **Vertx**, manda un messaggio di **ModelStopped** al **PlayerController** e poi termina
- se riceve dei messaggi dal *server* li deserializza e li indirizza alla **View**
- nel caso ci fossero problemi con la connessione invia un messaggio di **ModelError** al **PlayerController**



4.7.2 PlayerController

Attore che funge da intermezzo tra i messaggi del **Model** e della **View**. Riceve messaggi del tipo **PlayerControllerNotification**

- **ModelStopped**: mandato dal **Model** per notificare la sua terminazione
- **ViewStopped**: mandato dalla **View** per notificare la sua terminazione
- **ModelError**: mandato dal **Model** per notificare che è avvenuto un errore (in questo caso di rete)



Appena avviato crea un'istanza di **View** e una di **Model**. Grazie a *message adapters* può ricevere messaggi da questi ultimi attori.

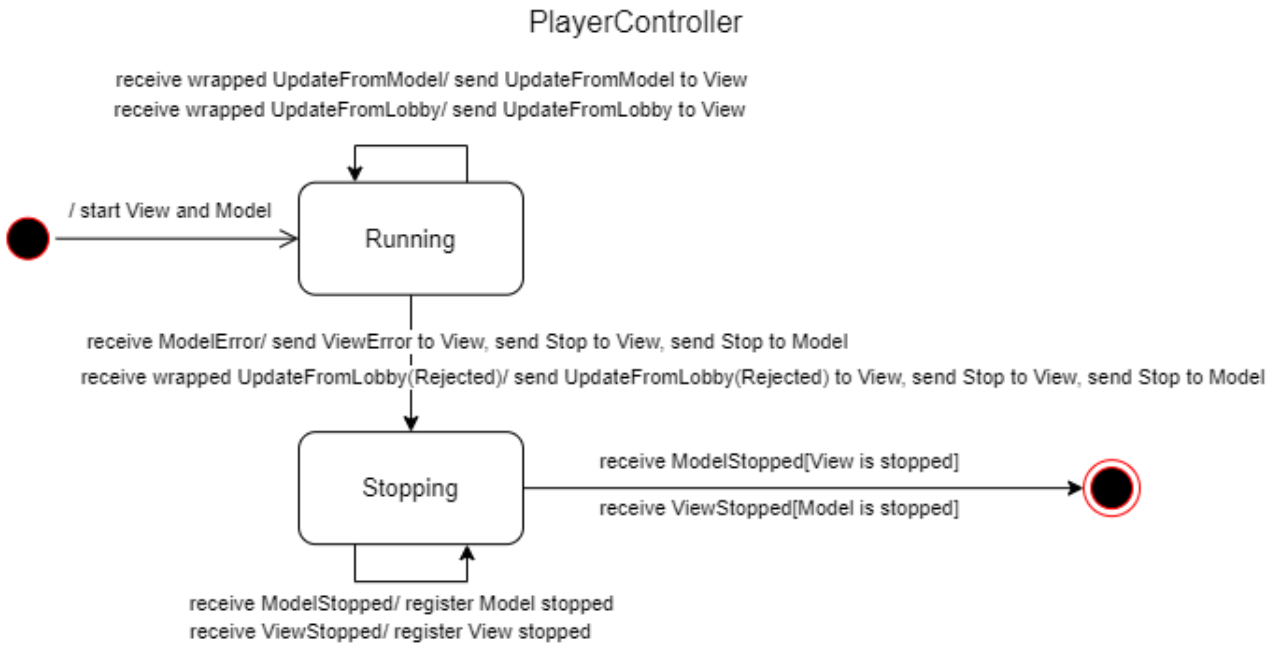
Comportamento generale:

- se riceve un **UpdateForView** lo indirizza verso la **View**. Se questo contiene il messaggio di **UpdateFromLobby(Rejected)** va inoltre nello stato **stopping**

- se riceve un **ModelError** lo inoltra alla **View** e cambia nello stato *stopping*

Comportamento nello stato *stopping*:

1. per prima cosa, il **PlayerController** invia un messaggio di **Stop** sia alla **View** che al **Model**
2. una volta terminati risponderanno rispettivamente con i messaggi di **ViewStopped** e **ModelStopped**
3. una volta ricevuti i messaggi sopra citati il **PlayerController** si stoppa

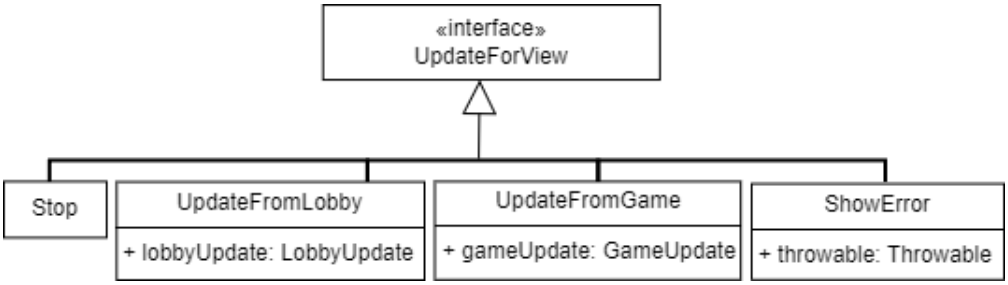


4.7.3 View

Attore che rappresenta l'interfaccia grafica con l'utente. Ha il ruolo di ricevere in input le sue scelte e mostrare gli aggiornamenti del gioco.

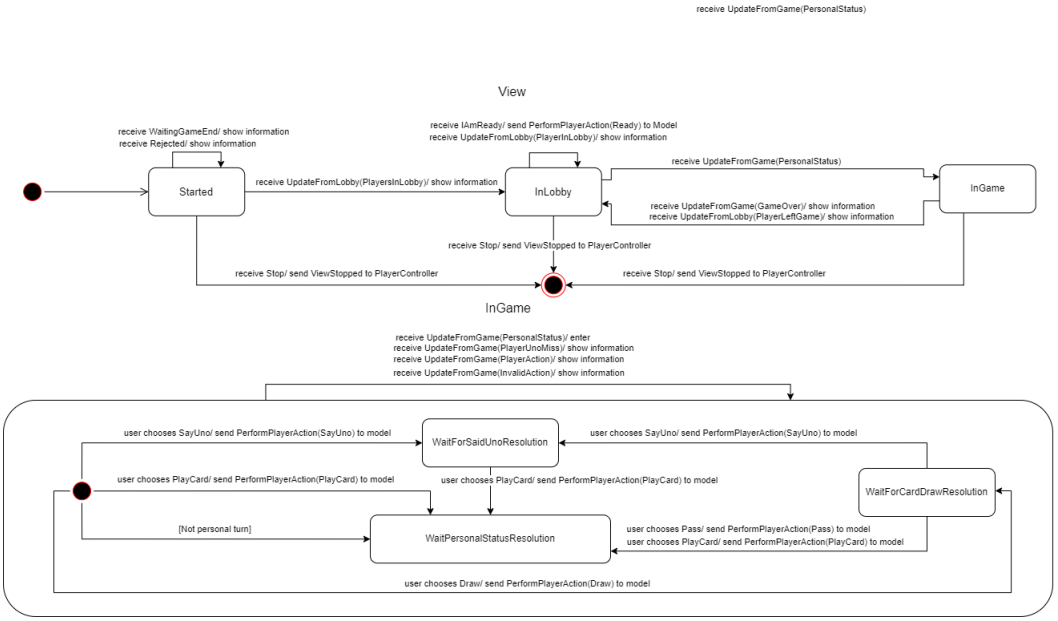
Riceve messaggi del tipo **UpdateForView**:

- **UpdateFromLobby**: mandato dal **Model** rappresenta un aggiornamento della *lobby*
- **UpdateFromGame**: mandato dal **Model** rappresenta un aggiornamento del gioco
- **ShowError**: mandato dal **PlayerController** rappresenta un errore da visualizzare
- **Stop**: mandato dal **PlayerController** con l'ordine di stopparsi



Funzionamento generale:

- in base alla fase di gioco mostra informazioni e riceve input dell'utente diversi
- durante la fase di *lobby* mostra i giocatori presenti e il loro stato (*ready* o **not ready**) e permette all'utente di poter andare, qualora non lo fosse, nello stato *ready*
- durante la fase di gioco mostra le informazioni sull'evoluzione della partita e permette all'utente, durante il suo turno, di poter scegliere quale azione svolgere



4.8 Load Balancer

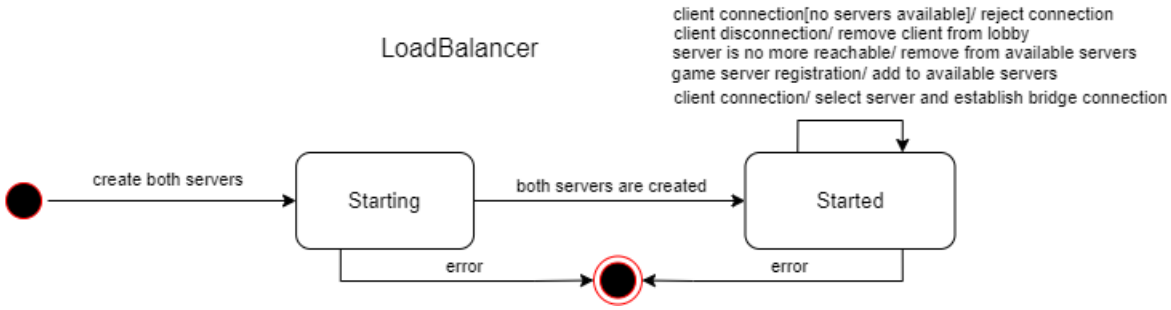
Un *load balancer* distribuisce il traffico di rete tra vari *server* per garantire prestazioni e affidabilità ottimali. Monitora il loro carico e reindirizza le richieste dei *clienti* al *server* meno carico disponibile. Ogni server può eventualmente registrarsi presso il *load balancer* rendendosi disponibile.

Questo componente viene modellato dall'attore **LoadBalancer**

Riceve messaggi del tipo **LoadBalancerMessage** ma nessuno dei messaggi è pubblico.

Comportamento generale:

- una volta avviato crea due **server** in due porte diverse specificate. Uno viene contattato dai *client* che voglino partecipare ad una partita, l'altro dai **server** che vogliono registrarsi.
- un *server* può, con una richiesta *POST* registrarsi dal **LoadBalancer** per essere contattato.
- quando un giocatore si connette, tramite connessione *web-socket*, il **LoadBalancer** effettua un controllo:
  - se la *lobby* specificata dal giocatore non esiste allora seleziona il server meno carico e instaura una nuova connessione *web-socket*. Ogni messaggio che viene ricevuto dal giocatore viene inviato al server e viceversa
  - se la *lobby* specificata dal giocatore esiste già allora si seleziona il *server* che contiene quella *lobby* e si effettuano gli stessi collegamenti illustrati al punto precedente
- quando un giocatore si disconnette viene decrementato il numero di giocatori facenti parte la sua *lobby*. Se una *lobby* contiene 0 giocatori viene rimossa da quelle già create
- se un giocatore si connette ma non sono presenti *server* la connessione viene rifiutata con un 500
- se un *server* non è più raggiungibile o smette di rispondere viene rimosso da quelli disponibili e, nel caso di connessione di un nuovo giocatore, se ne seleziona uno nuovo



5 Validazione

Le funzionalità del sistema vengono testate tramite la combinazione di diversi frameworks usati in modo combinato:

- **ScalaTest**: permette di creare delle suites di test. Nello specifico è stata usata **AnyFunSpec** in combinazione con **Matchers** in modo da poter specificare i requisiti del test chiaramente usando il linguaggio naturale.
- **AkkaTest**: mette a disposizione il **ActorTestKit** con cui si possono testare gli attori di **Akka**.

Per ogni test si è cercato creare il massimo di indipendenza tra le risorse utilizzate, sopratutto la rete. Inoltre sono state seguite le *best practices* descritte nella documentazione ufficiale di *Akka*

Inoltre, è stata introdotta la *code coverage* grazie al *plugin* di sbt **ScoverageSbtPlugin** in modo da monitorare e ampliare in maniera iterativa la copertura dei test sul codice. La soglia che si è decisa di impostare per la copertura è del 70%.

Ogni componente è stato testato tranne per **View** e **GameModel**. La scelta è stata fatta a fronte di una elevata difficoltà nel testing. Si è così proceduto a testare queste componenti durante una fase di *alpha-testing* interagendoci direttamente.

6 Installazione

I requisiti di progetto comprendevano la creazione di container **Docker** per ogni componente in modo da renderli facilmente eseguibili su macchine diverse e deployabili in sistemi cloud.

Ciò non toglie che si possano eseguire direttamente usando come strumento di automazione quello con cui è stato sviluppato l'intero progetto, sbt.

Ci sono quindi due diverse modalità

6.1 Installazione con sbt

Per prima cosa si utilizza il comando `sbt compile` per compilare il codice sorgente. Successivamente è necessario eseguire i seguenti comandi in base al componente di cui hai bisogno:

- Per eseguire il **Load Balancer**

```
sbt runMain launchers.LBLauncher \
[--player-registration-port=<default=4444> \
--server-registration-port=<default=2222>]
```

- Per eseguire il *Server*

```
sbt runMain launchers.ServerLauncher \
[--port=<default=4444> \
--host=<default=localhost> \
--lb-port=<default=2222> \
```

```
--lb-host=<default=localhost> \
--load-balancer=<default=false>]
```

- Per eseguire il *Client/Player*

```
sbt runMain launchers.PlayerLauncher \
--username=<valore obbligatorio> \
--lobby-name=<valore obbligatorio> \
[--server-port=<default=4444> \
--server-host=<default=localhost>]
```

6.2 Installazione con Docker

Utilizzando **Docker**, per ogni componente è necessario costruire l'*immagine* e quindi eseguirla:

- Per il **Load Balancer**

```
docker build -t uno-load-balancer -f load_balancer .
```

per creare l'immagine e poi

```
docker run -p <server-registration-port>:<server-registration-port> \
-p <player-registration-port>:<player-registration-port> \
uno-load-balancer:latest  \
[--player-registration-port=<default=4444> \
--server-registration-port=<default=2222>]
```

```
ad esempio:
docker run -p 2222:2222 -p 4444:4444 \
uno-load-balancer:latest \
--player-registration-port=4444 \
--server-registration-port=2222
```

- Per il *server*

```
docker build -t uno-server -f server .
```

per creare l'immagine e poi

```
docker run -p <port>:<port> uno-server:latest \
[--port=<default=4444> \
--host=<default=localhost> \
--lb-port=<default=2222> \
--lb-host=<default=localhost> \
--load-balancer=<default=false>]


ad esempio con Load Balancer
docker run -p 5555:5555 \
uno-server:latest \
--port=5555 \
--host=host.docker.internal \
--lb-port=2222 \
--lb-host=host.docker.internal \
--load-balancer=true


ad esempio senza Load Balancer
docker run -p 4444:4444 uno-server:latest --port=4444
```

- Per il *client*

```
docker build -t uno-player -f player .
```

per creare l'immagine e poi

```
docker run uno-player:latest \
--username=<valore obbligatorio> \
--lobby-name=<valore obbligatorio> \
[--server-port=<default=4444> \
--server-host=<default=localhost>]


ad esempio
docker run -it uno-player:latest \
--username=Lorenzo \
--lobby-name=test \
--server-port=4444 \
--server-host=host.docker.internal
```

7 Esempio di utilizzo

Una volta avviati **Load Balancer**

```
PS C:\Users\Gardo\Desktop\ds-project-gardini-ay2021> docker run -p 2222:2222 -p 4444:4444 uno-load-balancer:latest --player-registration-port=4444 --server-registration-port=2222
[info] welcome to sbt 1.5.6 (AdoptOpenJDK Java 11.0.11)
[info] loading settings for project app-build from plugins.sbt,wart.sbt ...
[info] loading project definition from /app/project
[info] loading settings for project root from build.sbt ...
[info] set current project to UNO scala (in build file:/app/)
[info] running launchers.LBLauncher --player-registration-port=4444 --server-registration-port=2222
Staring Load balancer with server-registration-port = 2222 and player-registration-port = 4444
Registered new server host host.docker.internal port 5555 size 21
```

e *server*

```
PS C:\Users\Gardo\Desktop\ds-project-gardini-ay2021> docker run -p 5555:5555 uno-server:latest --port=5555 --host=host.docker.internal --lb-port=2222 --lb-host=host.docker.internal --load-balancer=true
[info] welcome to sbt 1.5.6 (AdoptOpenJDK Java 11.0.11)
[info] loading settings for project app-build from plugins.sbt,wart.sbt ...
[info] loading project definition from /app/project
[info] loading settings for project root from build.sbt ...
[info] set current project to UNO scala (in build file:/app/)
[info] running launchers.ServerLauncher --port=5555 --host=host.docker.internal --lb-port=2222 --lb-host=host.docker.internal --load-balancer=true
Starting Server at port: 5555 size: 21 seed: 1234 with Load Balancer host: host.docker.internal port: 2222 uri: /register
```

si può creare un utente. Questo entra direttamente in lobby aspettando un comando per essere pronto.

```
PS C:\Users\Gardo\Desktop\ds-project-gardini-ay2021> docker run -it uno-player:latest --username=Lorenzo --lobby-name=test --server-port=4444 --server-host=host.docker.internal
[info] welcome to sbt 1.5.6 (AdoptOpenJDK Java 11.0.11)
[info] loading settings for project app-build from plugins.sbt,wart.sbt ...
[info] loading project definition from /app/project
[info] loading settings for project root from build.sbt ...
[info] set current project to UNO scala (in build file:/app/)
[info] running launchers.PlayerLauncher --username=Lorenzo --lobby-name=test --server-port=4444 --server-host=host.docker.internal
Player Lorenzo, lobby test, server host host.docker.internal, server port 4444

--------------- Players in lobby ---------------
Lorenzo - false
Press enter to be ready!
```

Se viene avviato un secondo utente vengono visualizzati entrambi nella *lobby*.

```
Player Alessandro, lobby test, server host host.docker.internal, server port 4444


--------------- Players in lobby ---------------
Lorenzo - false
Alessandro - false
Press enter to be ready!
```

Se un utente è pronto viene visualizzato.

```
--------------- Players in lobby ---------------
Lorenzo - false
Alessandro - true
```

Quando entrambi i giocatori sono pronti la partita inizia.

```
-------------- New Turn: Personal Status --------------

Is your turn? Yes
Players order: Alessandro -> Lorenzo
Discarded card: 2 Blue
Players cards in hand:
Alessandro -> 7
Lorenzo -> 7
Choose
[1] Play Card: 2 Blue
[2] Play Card: 4 Green
[3] Play Card: 4 Blue
[4] Play Card: 9 Yellow
[5] Play Card: 8 Green
[6] Play Card: 4 Yellow
[7] Play Card: 2 Red
[8] Say Uno
[9] Draw
```

Figura 2: È il turno del giocatore "Alessandro"

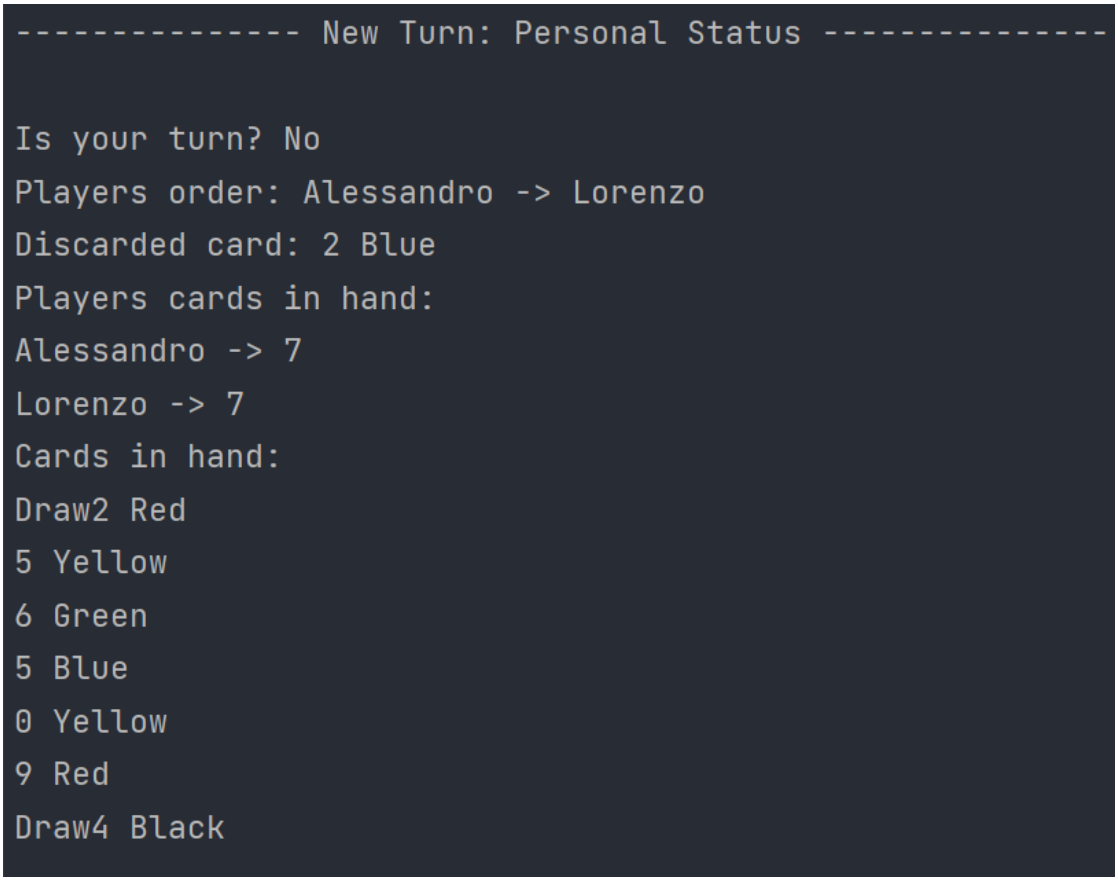


Figura 3: Non è il turno del giocatore "Lorenzo"

Se il giocatore effettua una mossa viene registrata e vengono mostrati gli aggiornamenti

```
1
Alessandro did Play Card: 2 Blue

--------------- New Turn: Personal Status ---------------

Is your turn? No
Players order: Lorenzo -> Alessandro
Discarded card: 2 Blue
Players cards in hand:
Alessandro -> 6
Lorenzo -> 7
Cards in hand:
4 Green
4 Blue
9 Yellow
8 Green
4 Yellow
2 Red
```

Figura 4: "Alessandro" gioca il 2 blu e viene mostrato come messaggio

Si continua così fino a quando o un giocatore vince o un giocatore non si disconnette.

```
Player left the game: Lorenzo. Return to lobby
Press enter to be ready!

--------------- Players in lobby ---------------
Alessandro - not ready
```

Figura 5: "Lorenzo" si disconnette (stava perdendo). Tutti i giocatori tornano alla lobby

8 Conclusioni

8.1 Che cosa ho imparato

In questo progetto ho imparato molte cose, ognuna delle quali mi ha richiesto però diverso tempo di approfondimento e di pratica portandomi a sforare il numero di ore previste (in questo caso 90). Ecco un elenco di quello che ho imparato:

- come usare e testare il framework **Vertx**
- come progettare e testare applicazioni con il framework **Akka**
- quale architetura fosse la più adatta (quella *client-server*, dopo diversi tentativi fallimentari)
- sviluppare con sbt
- un minimo di scrittura di pipeline per **GitLab**
- come gestire la coverage
- come scrivere il **Dockerfile** parametrico, e come usarlo
- come funziona un *load balancer*
- gestione più pulita del *VCS*

8.2 Lavori futuri

Il progetto si potrebbe estendere ulteriormente:

- aggiungendo una schermata grafica lato *client*
- rendendo il *load balancer* più adatto a gestire meglio i server e il carico.
- rilasciarlo usando un servizio cloud come *AWS* o *GCP*.