PROGETTO DI SOFTWARE ARCHITECTURE DESIGN

Studenti:
Mazza Francesco M63001338
Nelli Marisanna M63001323
Paesano Alessio M63001347
Reggio Raimondo M63001328
Pascarella Vincenzo M63001337

Anno accademico 2021-2022

Indice

1 Avvio del progetto

1.1 Introduzione

Si vuole realizzare un multiplayer game che permetta a quattro giocatori di sfidarsi in una competizione di disegno. Lo scopo del gioco é quello di disegnare nella maniera più precisa possibile gli oggetti che vengono suggeriti a schermo, in modo tale che una rete neurale sia in grado di classificarli il più velocemente possibile. Vince la partita il giocatore che ha acquisito il punteggio massimo cioè colui che ottiene il maggior numero di disegni correttamente riconosciuti.

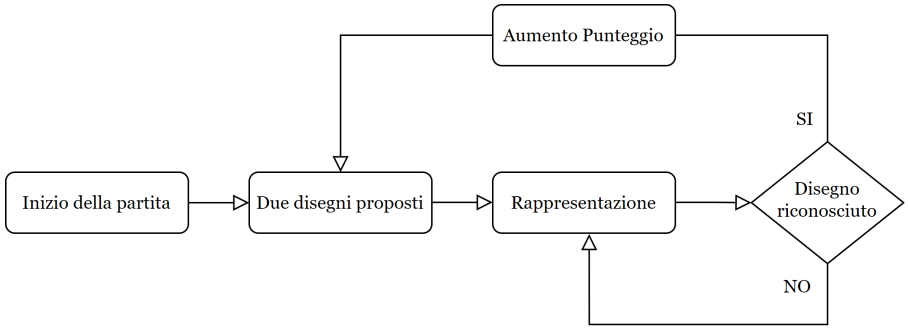

Figura 1: Dinamica di gioco

1.2 Parti interessate

Il sistema software é utilizzato dall'attore **Utente** e dipende da un **riferimento temporale**.

- Utente: é in grado di effettuare le operazioni di registrazione e di log-in, é in grado di iniziare una nuova partita con altri giocatori. Durante la partita ha a disposizione una drawing area in cui può disegnare. Durante la partita il giocatore può visualizzare l'attività degli altri giocatori collegati. Il giocatore può consultare la classifica aggiornata con i punteggi degli altri giocatori.

1.3 Funzionalità del sistema

Il sistema deve:

- Permettere ad un utente di registrarsi.
- Permettere all'utente di effettuare il login.
- Permettere all'utente autenticato di entrare in una partita.
- Associare un insieme di utenti in una stessa sessione di gioco.
- Permettere all'utente di disegnare su schermo durante una partita.
- Permettere all'utente di vedere i disegni degli avversari durante una partita.
- Permettere all'utente di visualizzare il tempo rimanente durante una partita.
- Riconoscere il soggetto disegnato dall'utente.
- Analizzare in real-time il disegno oppure la parte disegnata.
- Memorizzazione dello storico delle partite.
- Assegnare ad ogni utente registrato una posizione in classifica sulla base delle partite vinte.
- Visualizzazione della classifica delle partite vinte.

1.4 Glossario dei termini

Termine	Descrizione	Attinenza
Utente	Colui che usufruisce del servizio.	Utente
Giocatore	Sinonimo di Utente.	Utente
Partita	Competizione tra giocatori.	Giocatore
Lobby	Insieme di giocatori in attesa di partecipare ad una partita.	Giocatore,Partita
Disegno	Rappresentazione grafica prodotta da un giocatore.	Giocatore,Partita
Soggetto	Tema del disegno da rappresentare.	Giocatore,Partita,Disegno
Scoreboard	Classifica globale dei punteggi dei giocatori.	Giocatore
Storico	Sinonimo di Scoreboard	Scoreboard

2 Processo di Sviluppo

Per la realizzazione di questo progetto é stato deciso di adottare un approccio agile, seguendo, in particolar modo, il framework Scrum.

2.1 Scrum

Scrum è un framework euristico: si basa sull'apprendimento continuo e l'adattamento a fattori variabili. Un approccio di tipo agile deve tener conto del fatto che il team non dispone di tutte le conoscenze all'inizio di un progetto e che si evolverà progressivamente con acquisizione di esperienza ed il consolidamento del team di lavoro. Scrum è strutturato per aiutare i team ad adattarsi in modo naturale alle mutevoli condizioni ed esigenze degli utenti con una ridefinizione delle priorità integrata nel processo e cicli di rilascio brevi che consentono al team di imparare e migliorare costantemente.

Lo sviluppo secondo Scrum é basato su un insieme di artefatti:

- **Incremento:** è un passo concreto verso l'obiettivo del prodotto. Ogni incremento è additivo rispetto a tutti quelli precedenti e viene accuratamente verificato, per garantire che tutti gli incrementi funzionino insieme. Per fornire valore, l'incremento deve essere utilizzabile.
- **Product Backlog:** è un elenco di lavori prioritari per il team di sviluppo che deriva dalla roadmap e dai suoi requisiti. Gli elementi più importanti sono indicati in cima al backlog del prodotto, in modo che il team sappia cosa consegnare per primo.
- **Sprint Backlog:** insieme degli elementi del product backlog che vengono selezionati per uno sprint. Il backlog contiene anche le informazioni necessarie per consegnare l'incremento del prodotto e realizzare l'obiettivo dello sprint (*sprint goal*).

Lo scrum team è composto da diverse figure:

- **Product owner:** figura responsabile di massimizzare il valore del prodotto da realizzare nonché unico responsabile del product backlog. Ha il compito di interfacciarsi con il cliente ma anche avere una visione del valore che lo scrum team sta apportando al progetto. Inoltre, ha la responsabilità di bilanciare le esigenze degli stakeholder dell'organizzazione.

- **Scrum master:** figura che aiuta il product owner a comprendere e comunicare meglio il valore delle pratiche Scrum, gestire al meglio il backlog, pianificare il lavoro con il team e suddividere il lavoro per ottenere un apprendimento più efficace.
  Aiuta il team ad auto-organizzarsi, superare gli impedimenti riscontrati e gestire le interazioni tra team di sviluppo ed il resto dell'organizzazione.

- **Development team:** insieme di professionisti responsabili della realizzazione degli incrementi di cui prodotto ha bisogno. Secondo la Guida Scrum, può essere composto da tutti i tipi di persone, compresi designer, scrittori, programmatori, ecc.
  La dimensione del team di sviluppo può variare dai 3 ai 9 elementi.
  Il ruolo di "sviluppatore" in Scrum indica un membro del team che ha le giuste competenze, come parte del team per svolgere il lavoro.

Scrum durante il processo di sviluppo prevede diversi *eventi* riportati di seguito:

- **Sprint:** [4 settimane] è un breve periodo di tempo, o timebox, in cui un team Scrum collabora per completare una determinata quantità di lavoro.

- **Pianificazione dello Sprint:** [4 ore] è un evento in cui si pianifica il lavoro da svolgere durante uno sprint. In particolare viene definito cosa deve essere realizzato e come deve essere svolto il lavoro. Durante tale evento viene definito lo sprint goal, ovvero l'obiettivo da raggiungere al termine dello Sprint.

- **Daily Scrum:** [15 minuti] è un evento di 15 minuti per gli sviluppatori del team Scrum. Per ridurre la complessità, si tiene alla stessa ora e nello stesso luogo ogni giorno lavorativo. Si pianifica il lavoro da svolgere nelle successive 24 ore.

- **Sprint Review:** [3 ore] evento in cui avviene la revisione dell'incremento rilasciato al termine dell'ultimo sprint concluso. In questa fase avviene il confronto con gli stakeholder e vengono proposti spunti per l'incremento successivo.

- **Sprint Retrospective:** [1 ora] evento che avviene nell'ambito del team allo scopo di migliorare il proprio rendimento in occasione dello sprint successivo.

2.2 Pratiche agili

Le metodologie agili hanno l'intento di presentare nel minor tempo possibile un risultato concreto ed eseguibile e sopratutto meglio adattabile a cambiamenti dei requisiti, per questo motivo, la produzione della documentazione del software é presente in misura minore nelle prime fasi del progetto, é viene costantemente aggiornata e raffinata fino al rilascio del prodotto finale. Per documentare il software, allora, é stato deciso di introdurre i soli diagrammi indispensabili alla comprensione e le opportune viste, aggiornando progressivamente sulla base dei cambiamenti e risultati ottenuti.

L'utilizzo di un approccio agile ha favorito alcune pratiche che favoriscono l'interazione e la collaborazione tra i membri del development team. Alcune di queste sono:

- Pair Programming
- Code refactoring
- Time boxing

Per poter rendere l'ambiente di lavoro simile ad un contesto reale, sono stati assegnati i seguenti ruoli:

- Scrum Master: Paesano.
- Development Team: Mazza, Nelli, Paesano, Pascarella, Reggio.

2.3 Strumenti software di supporto

Per poter mettere in pratica il frameowrk scrum, é stato sufficiente utilizzare la piattaforma Jira Software, essa permette agli utenti di creare un team di lavoro e di utilizzare un insieme di funzionalità tra le quali:

- Definire i product goal.
- Definire product backlog.
- Definire un task.
- Destione delle storie utente.
- Avvio di uno sprint.
- Gestione di ruoli e responsabilità.

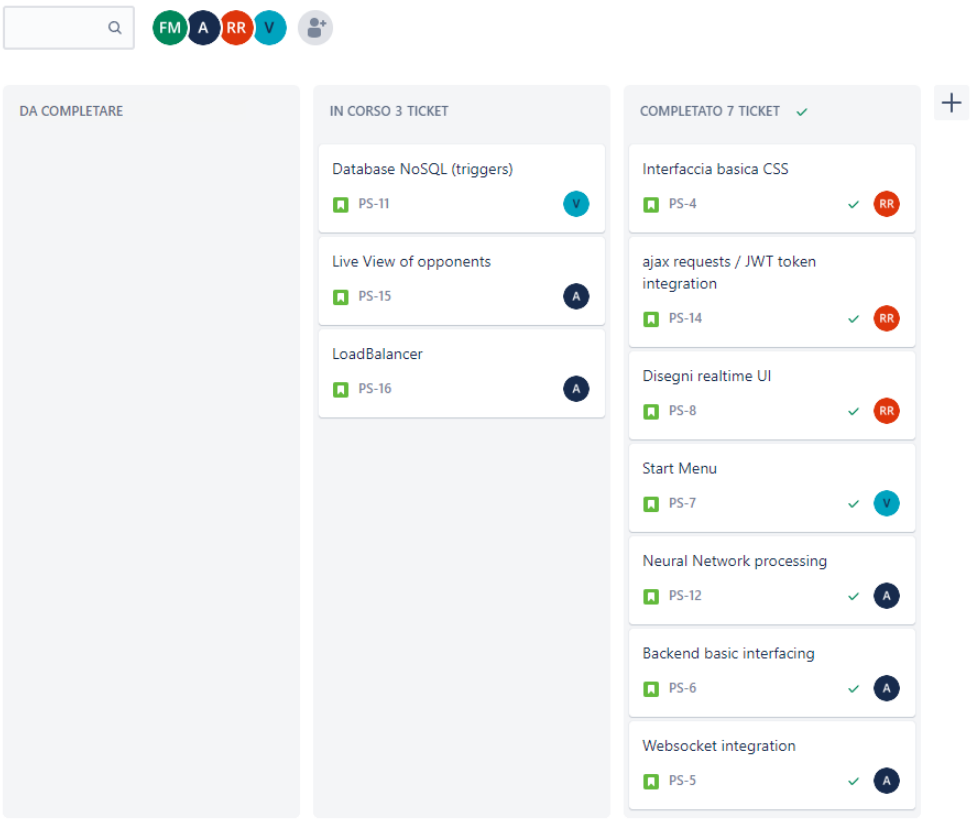

Figura 2: Jira

Per la gestione del versioning del software e allo scopo di favorire la collaborazione tra i membri del development team é stato utilizzato **GitHub**, la cui repository é accessibile al seguente link:

`https://github.com/Alessio-jpg/SAD-Project`

3 Specifica dei requisiti

3.1 Diagramma dei casi d'uso

Il diagramma dei casi d'uso descrive le interazioni tra gli attori ed il sistema.

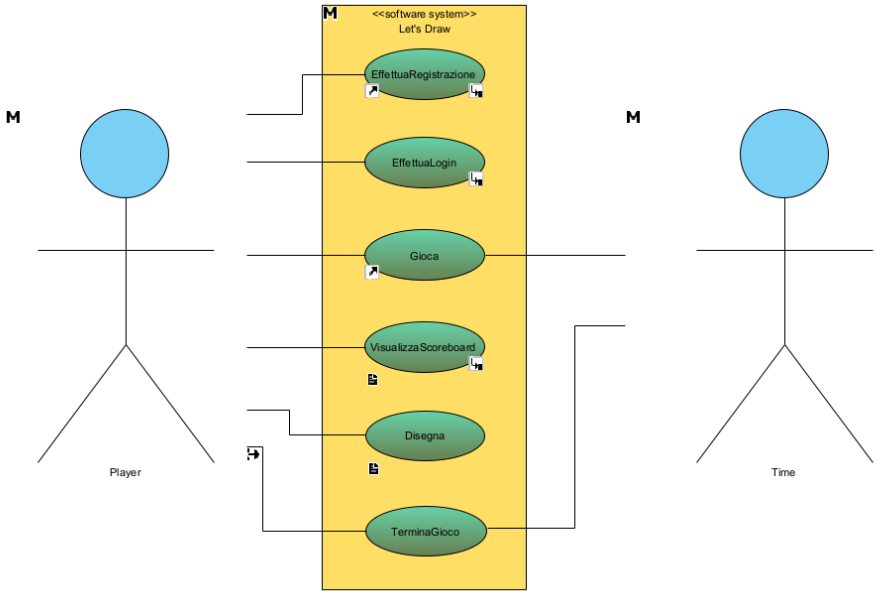

Figura 3: Diagramma dei casi d'uso

3.2 Descrizione casi d'uso

- **EffettuaRegistrazione**: Il giocatore richiede al sistema di registrarsi fornendo l'username scelto e la password.
- **EffettuaLogin**: Il giocatore richiede al sistema di effettuare il login fornendo i suoi dati d'accesso.
- **Gioca**: Il giocatore richiede al sistema di iniziare una partita.
- **VisualizzaScoreboard**: Il giocatore richiede al sistema di visualizzare la classifica dei migliori giocatori.
- **Disegna**: Il giocatore in partita disegna uno dei soggetti proposti.
- **TerminaGioco**: Allo scadere del tempo, la partita corrente termina.

3.3 Scenari

Caso d'uso	EffettuaRegistrazione
Attore primario	Giocatore.
Precondizioni	nessuna.
Sequenza di eventi	1. Il caso d'uso inizia quando l'Utente richiede di registrarsi. 2. L'utente inserisce username e password che vuole utilizzare. 3. if l'username è già utilizzata 3.1. SYSTEM "ERROR: username già utilizzato" 4. else if il campo password è vuoto 4.1. SYSTEM "ERROR: password non valida" 5. else if il campo username è vuoto 5.1. SYSTEM "ERROR: usurname non valido" end if 6. Il sistema registra l'utente.
Postcondizioni	L'utente é stato registrato correttamente.

Caso d'uso	EffettuaLogin
Attore primario	Giocatore.
Precondizioni	L'utente é registrato.
Sequenza di eventi	1. Il caso d'uso inizia quando l'utente chiede di effettuare il login 2. L'utente inserisce username e password 3. if username e password sono errati 3.1. SYSTEM Error 4. L'utente effettua il login
Postcondizioni	L'utente ha effettuato il login correttamente.

Caso d'uso	Gioca
Attore primario	Giocatore.
Attore secondario	Tempo.
Precondizioni	L'utente ha effettuato il login.
Sequenza di eventi	1. Il caso d'uso inizia quando l'utente avvia una partita. 2. Il sistema associa l'utente ad altri giocatori che hanno avviato una partita. 3. Il sistema avvia la partita. 4. Il sistema fornisce l'elemento da disegnare al giocatore. 5. Il sistema notifica l'inizio della partita.
Postcondizioni	nessuna.

Caso d'uso	VisualizzaScoreboard
Attore primario	Giocatore.
Precondizioni	L'utente ha effettuato il login.
Sequenza di eventi	1. Il caso d'uso inizia quando l'utente richiede di visualizzare la scoreboard. 2. Il sistema preleva le informazioni riguardanti i migliori giocatori. 3. Il sistema mostra i punteggi all'utente.
Postcondizioni	nessuna.

Caso d'uso	Disegna
Attore primario	Giocatore.
Precondizioni	L'utente ha iniziato una partita.
Sequenza di eventi	1. Il sistema presenta un **template** di disegno al giocatore 2. Il giocatore traccia il disegno 3. Il sistema elabora il disegno dell'utente 4. Il sistema presenta all'utente il soggetto del disegno 5. **if** il disegno è corretto 5.1. Il sistema incrementa il punteggio dell'utente 5.2. Il sistema comunica all'utente un nuovo soggetto da disegnare
Postcondizioni	nessuna.

Caso d'uso	TerminaGioco
Attore primario	Tempo.
Attore secondario	Giocatore.
Precondizioni	L'utente ha iniziato una partita.
Sequenza di eventi	1. Il caso d'uso inizia quando si è raggiunto il limite di tempo di una partita. 2. Il sistema termina la partita in corso. 3. Il sistema decreta il vincitore. 4. Il sistema memorizza il punteggio degli utenti.
Postcondizioni	La scoreboard viene aggiornata dal sistema.

3.4 Requisiti non funzionali

- **Scalabilità**: Il sistema deve essere facilmente modificabile sulla base della mole di richieste da gestire. Maggiore è la quantità di utenti connessi, maggiore deve essere la quantità di partite gestibili.
- **Usabilità**: Il sistema software deve essere intuitivo e facile da comprendere, in modo da poter essere utilizzato da un insieme di utenti il più vasto possibile.
- **Portabilità**: Il sistema software deve poter essere utilizzato in ambienti di esecuzione diversi, in modo tale da consentire ad utenti con diversi dispositivi di poter competere tra loro.
- **Performance**: Il sistema deve funzionare in tempo reale. La gestione delle partite, in particolare, deve poter rispondere immediatamente a variazioni di punteggio degli utenti e consentire che il riconoscimento dei disegni sia sufficientemente veloce da evitare rallentamenti nell'ottenimento del risultato.
- **Dispiegabilità**: Il sistema deve essere consegnato all'utente nel modo più semplice possibile ed essere fin da subito funzionante per l'inizio di una partita.

4 Analisi dei requisiti

4.1 Diagramma di contesto

Il primo dei diagrammi di analisi é il diagramma di contesto. Definisce il confine tra il sistema , o parte di un sistema, e il suo ambiente, mostrando le entità che interagiscono con esso. Costituisce una vista ad alto livello in cui il sistema viene trattato come una black box.
In questo caso, il sistema usufruisce di servizi esterni (Database) ed ha come input il tempo.

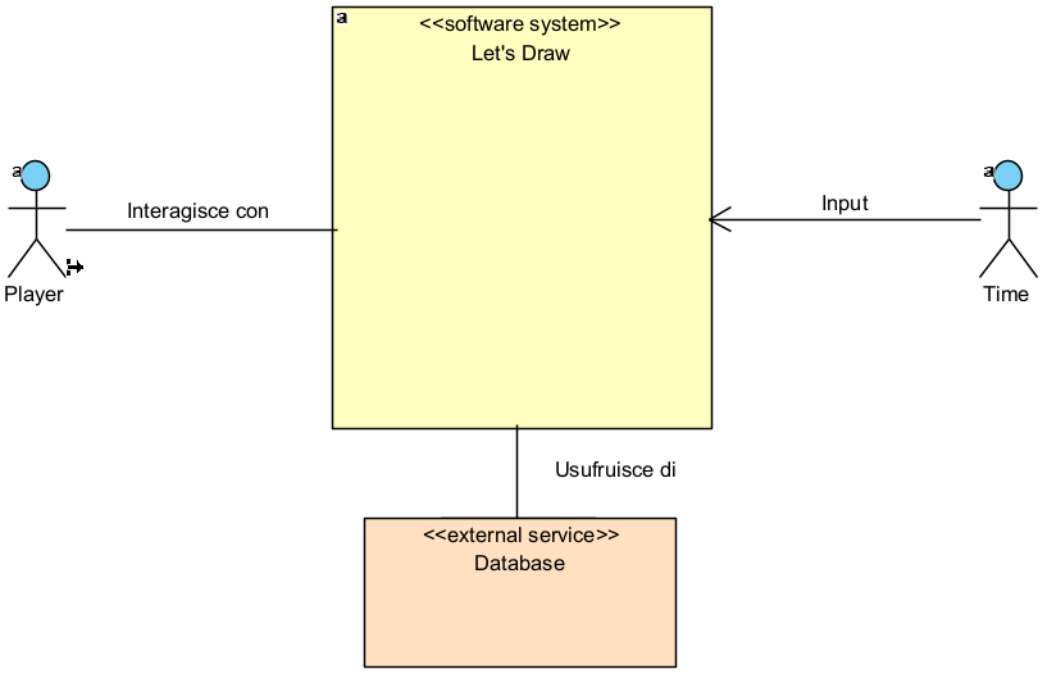

Figura 4: Diagramma di contesto

4.2 Diagramma delle classi

Il diagramma della classi rappresenta un *"system domain model"* in cui sono presentate le principali enetitá del sistema e le relazioni che intercorrono tra esse. Rappresenta una visione statica di un'applicazione. Descrive gli attributi e le operazioni di una classe e anche i vincoli imposti al sistema.

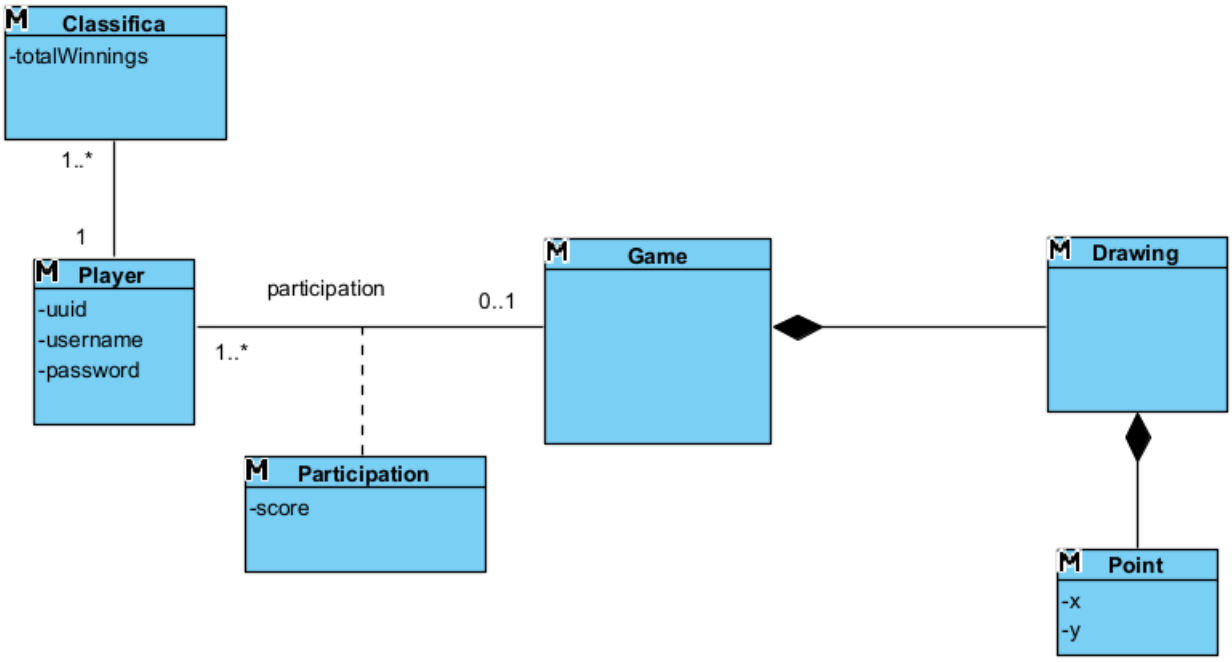

Figura 5: Diagramma delle classi - Analisi

Il diagramma presenta le classi *"Game"* e *"Player"* che messe in relazione dalla classe associativa *"Partecipation"*, la quale memorizza lo score e ne permette l'aggiornamento.
La classe *"Game"* si compone della classe *"Drawing"* che permette di inserire linee, cancellare e confermare l'invio del disegno.
A sua volta essa si compone di punti con rispettive coordinate spaziali (classe *Point*).

Il diagramma delle classi è stato raffinato aggiungendo ulteriori classi e relazioni, ottenendone uno di dettaglio.
Come é possibile notare, è stato inserita una classe *controller*.
Un **Controller** impone di assegnare la responsabilità di gestire gli eventi del sistema ad una classe non appartenente all'interfaccia utente. In altre parole, non è altro che la prima classe posta, oltre la UI, che riceve le richieste dell'utente e coordina le operazioni del sistema. Nel fare ciò, dovrebbe delegare ad altre classi di dominio il lavoro da svolgere per soddisfare le richieste dell'utente. In questo caso, è utilizzato dalla classe *Player* e *Game* e ad esso è demandata la responsabilità di aggiornare la *View*.
Lo stereotipo **"Singleton"**, infine, stabilisce che soltanto un'unica istanza della classe stessa possa essere creata all'interno di un programma.

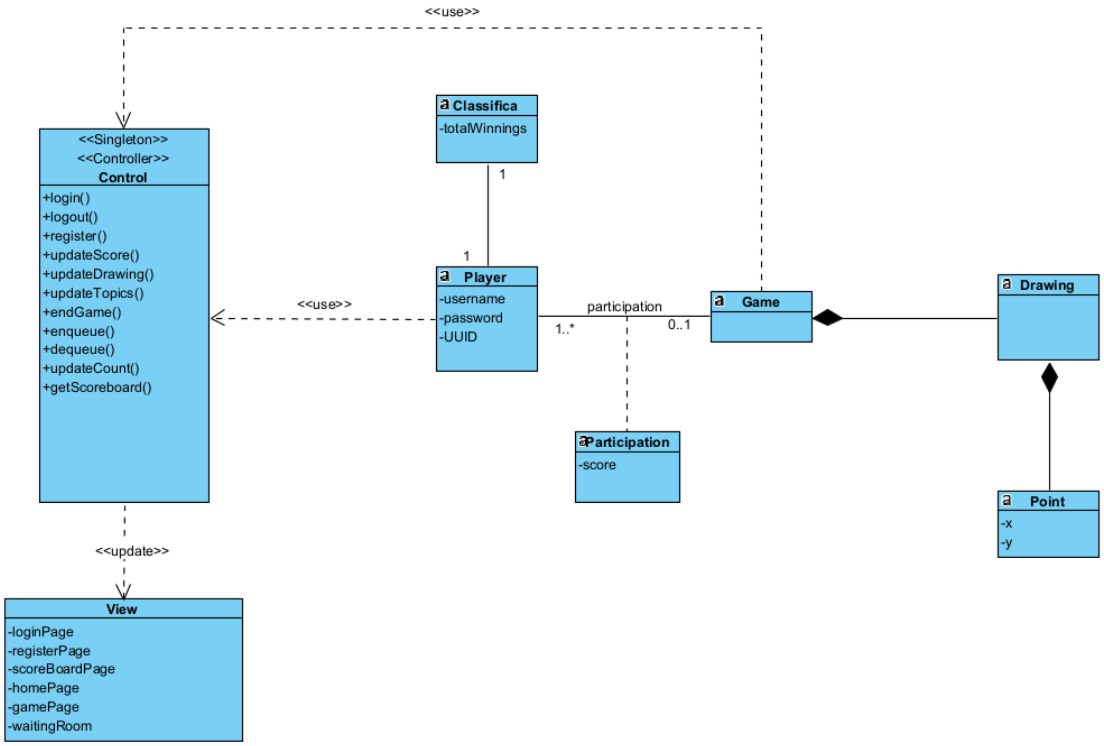

Figura 6: Diagramma delle classi raffinato

4.3 Diagramma di stato

Il diagramma di stato è una formalizzazione grafica del susseguirsi di eventi. Puó essere utilizzato per descrivere le transizioni che caratterizzano alcune sequenze di operazioni.
Di seguito sono riportati i diagrammi di stato dei casi d'uso "*Gioca*" e "*Termina gioco*".

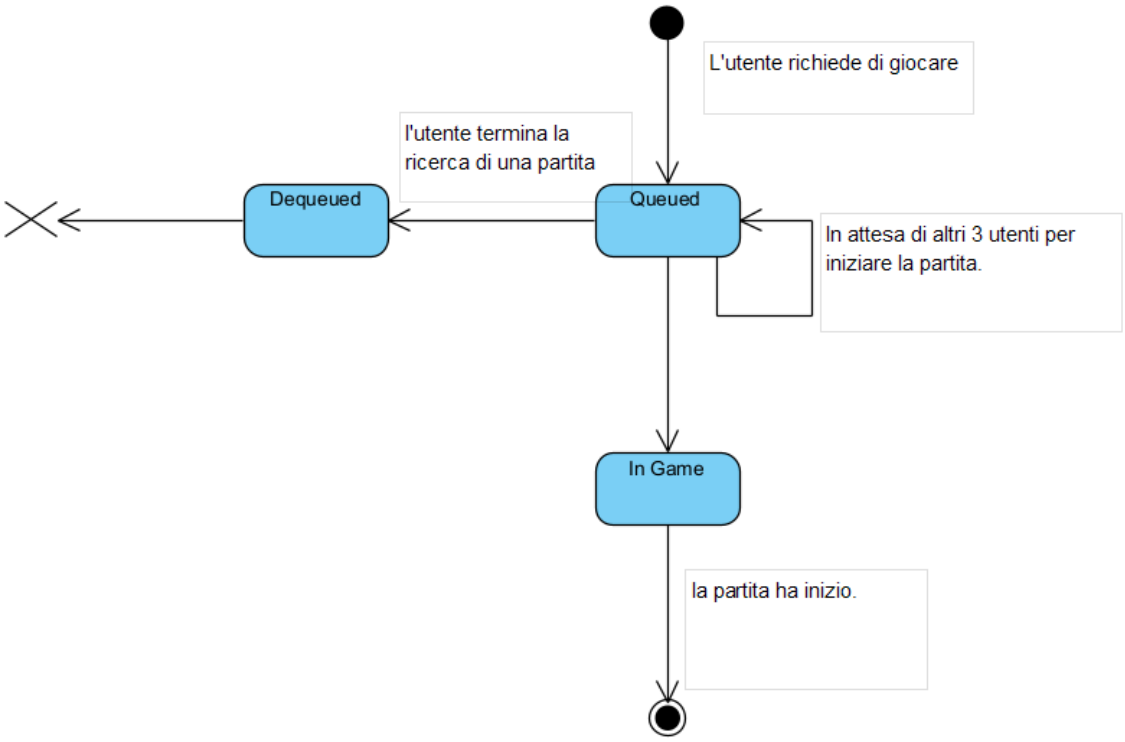

Figura 7: Diagramma di stato del caso d'uso "Gioca"

L'utente, nel momento in cui richiede di giocare, passa nello stato *"Queued"* dove resta in attesa di altri 3 giocatori per iniziare la partita.
Se l'utente mostra l'interesse ad abbandonare l'attesa per la partita, transita nello stato di *"Dequeued"*. Altrimenti, al raggiungere dei 4 giocatori in attesa, si passa allo stato di *"In Game"* e la partita ha inizio.

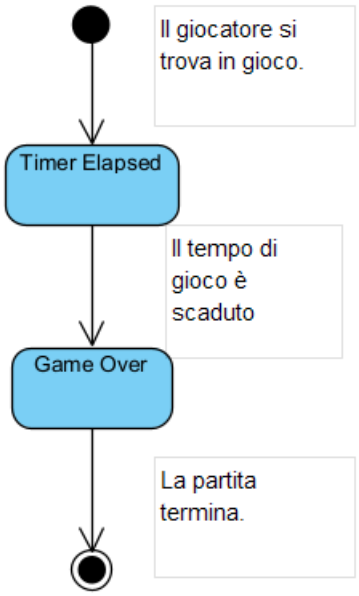

Figura 8: Diagramma di stato del caso d'uso "Termina gioco"

Il giocatore è in gioco. Quando trascorre il tempo prestabilito, si passa allo stato di *"Timer Elapsed"*.
Essendo scaduto il tempo, si transita in *"Game Over"* e termina il gioco.

4.4 Diagramma di sequenza - Analisi

Per effettuare il *login* l'utente invia username e password al sistema che provvede a controllarne l'esistenza. Se esiste, restituisce un messaggio di login avvenuto con successo altrimenti di login fallito.

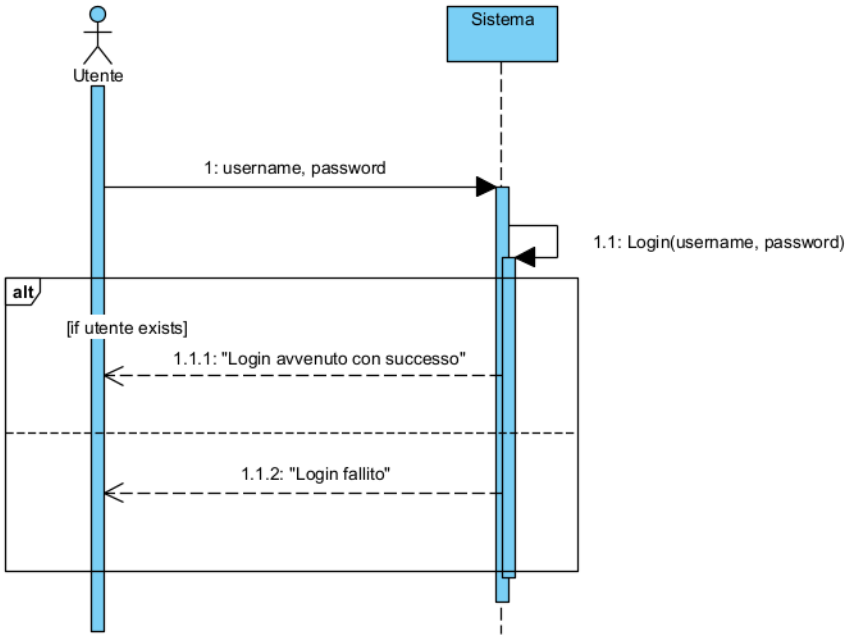

Figura 9: Diagramma di sequenza "EffettuaLogin"

Per effettuare la *registrazione* l'utente invia username e password al sistema. Se la registrazione avviene senza errori, il sistema restituisce un messaggio di successo altrimenti avvisa l'utente del fallimento della registrazione.

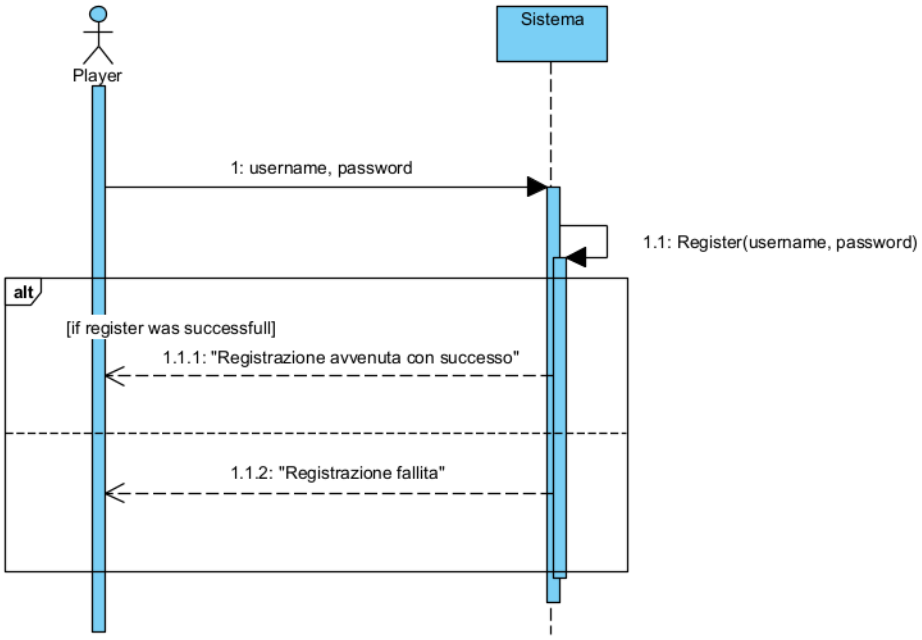

Figura 10: Diagramma di sequenza "EffettuaRegistrazione"

L'utente mostra l'intenzione di visualizzare la classifica al sistema. Questo provvede a recuperare lo storico e lo invia al player.

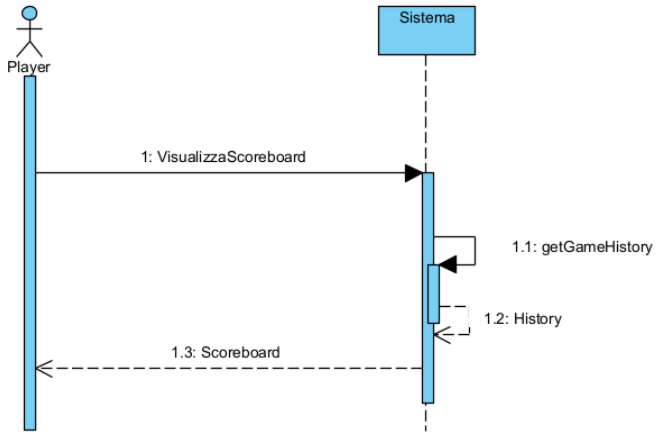

Figura 11: Diagramma di sequenza "VisualizzaScoreboard"

Finché l'utente è in gioco invia dei punti al sistema che li inoltra al gioco. Se il disegno non esiste, viene creato, altrimenti viene aggiornato con le nuove linee. Il disegno viene mostrato in tempo reale all'utente e se il sistema lo riconosce, viene mostrato anche il risultato.

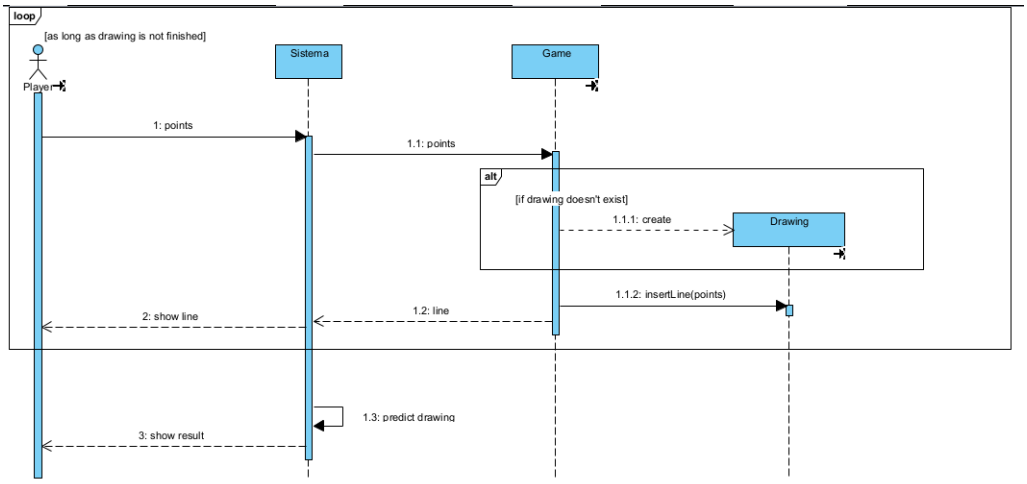

Figura 12: Diagramma di sequenza "Disegna"

4.5 Analisi architetturale

Il multiplayer game si figura come un software distribuito, in cui gli utenti utilizzano un client per interagire con l'interfaccia ed avere accesso alle funzionalità offerte. Un server, allora, riceve le richieste dell'utente implementando la logica del gioco (gestendo opportunamente una rete neurale) e dialogando con un servizio esterno per la gestione dei dati.

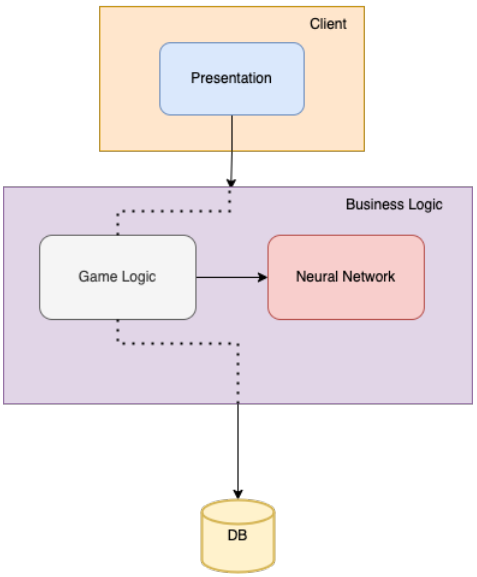

Figura 13: Schema logico dell'architettura "client-server".

Un modello di questo tipo, tipicamente, avrá almeno tre strati fisici differenti, con la possibilità di adottare uno schema *3 tier*, in cui i livelli sono client,server e database, gestendo però opportunamente la presenza di un livello logico aggiuntivo nel server, dovendo gestire l'interfacciamento tra la game logic e la rete neurale. Più avanti nella trattazione, sarà ulteriormente analizzato lo schema dell'architettura più adatto per il sistema software, sopratutto al fine di ottenere i requisiti di qualità chiave.

4.6 Stima dei costi

Per stimare i costi di sviluppo e, in generale, tenere traccia della dimensione del progetto, può essere necessario conoscere la produttività (basandosi su dati passati o approssimabili) ma anche la dimensione del team di sviluppo e l'ambiente di lavoro. La produttività, in particolare, può essere stimata adottando diversi criteri: in questo caso, é possibile considerare le *"Function-related measures"* basate su una stima della funzionalità del software rilasciato (Use-Case Point). Più complessi sono i casi d'uso, più tempo è necessario per la produzione. Essi dipendono da molteplici fattori tra i quali attori e ambiente in cui verrà sviluppato il software.

Prima di procedere con la stima vera e propria, si esaminano il numero di transazioni che interessa ogni singolo caso d'uso e si individua la tipologia dei relativi attori.

	Transazioni complessive	**Actor Type**
EffettuaRegistrazione	6	1
EffettuaLogin	4	1
Gioca	5	2
VisualizzaScoreboard	3	1
Disegna	6	3
TerminaGioco	4	1

4.6.1 Valutazione casi d'uso

Dalla stima della complessitá di ogni caso d'uso si puó ottenere un insieme di UUCW (*Unadjusted Use Case Weight*) considerando il numero di transazioni moltiplicate per il rispettivo peso assegnato:

$$\sum_{i=1}^{3} w_i n_i$$

UseCase complexity	**Weight**	**Number of use cases**	**Product**
Simple	5	3	15
Average	10	25	250
Complex	15	0	0
Total			265

4.6.2 Valutazione attori

Un fattore correttivo che puó essere considerato, riguarda la complessitá dell'interazione dell'attore in ciascuno dei casi d'uso.

$$\sum_{i=1}^{3} w_i n_i$$

Actor type	Weight	Number of Actors	Product
Simple	1	4	4
Average	2	1	2
Complex	3	1	3
Total			9

Ottenuto il fattore di UAW (*Unadjusted Actor Weight*) si possono ottenere gli UUCP (*Unadjusted Use Case Points*):

$$UUCP = UAW + UUCW$$

Unadjusted Use Case Weight - UUCW	**265**
Unadjusted Actor Weight - UAW	**9**
Unadjusted Use Case Points - UUCP	**274**

4.6.3 Valutazione Fattori di Complessità Tecnica

É necessario considerare dei fattori di complessità tecnica per la realizzazione del progetto (*Technical Correction Factor*). Il valore di TFactor si ottiene mediante la tabella tramite 13 fattori:

$$TFactor = \sum_{i=1}^{13} w_i v_i$$

Factor	Description	Weight	Assessment	Impact
T1	**Distributed system**	2	5	10
T2	**Performance objectives**	2	5	10
T3	**End-user efficiency**	1	3	3
T4	**Complex processing**	1	3	3
T5	**Reusable Code**	1	3	3
T6	**Easy to install**	0,5	5	2,5
T7	**Easy to use**	0,5	5	2,5
T8	**Portable**	2	5	10
T9	**Easy to change**	1	3	3
T10	**Concurrent use**	1	4	4
T11	**Security**	1	1	1
T12	**Access for third parties**	1	1	1
T13	**Training needs**	1	0	0
	Total			53

Assegnati i fattori ed ottenuto il totale ($TFactor$), si puó calcolare il fattore di correzione definitivo secondo la formula:

$$TCF = 0.6 + (0.01 * TFactor)$$

Technical Complexity Factor - TCF	**1,13**

4.6.4 Valutazione complessità dell'ambiente

Infine si considerano i fattori relativi all'ambiente di lavoro. In questo caso il totale EFactor si ottiene mediante 8 fattori:

$$EFactor = \sum_{i=1}^{8} w_i v_i$$

Factor	Description	Weight	Assessment	Impact
E1	**Familiar with the development process**	1,5	2	3
E2	**Application experience**	0,5	1	0,5
E3	**Object-oriented experience**	1	3	3
E4	**Lead analyst capability**	0,5	2	1
E5	**Motivation**	1	3	3
E6	**Stable requirements**	2	4	8
E7	**Part-time staff**	-1	1	-1
E8	**Difficult programming language**	-1	3	-3
Total				14,5

Assegnati i fattori ed ottenuto il totale (*EFactor*), si puó calcolare il fattore di correzione definitivo secondo la formula:

$$EF = 1.4 + (-0.03 * EFactor)$$

Environmental Factors - EF	**0,965**

A questo punto é possibile ottenere il valore finale degli *Use Case Points*:

$$UCP = UUCP * TCF * EF$$

Use Case Point - UCP	**298,7833**

5 Progettazione

5.1 Architettura

A partire dal system contex diagram e dalle analisi svolte sulla base dei requisiti funzionali e non funzionali, è necessario, nella fase di progettazione, intraprendere delle decisioni di design architetturale.

5.1.1 Separazione delle responsabilità

La progettazione tra client e server, analizzata nella sezione precedente, deve tuttavia prevedere una precisa separazione delle responsabilità. Il server non solo deve gestire l'autenticazione dell'utente, ma anche la logica di gioco. É possibile allora scomporre questi due servizi in due componenti distinti.

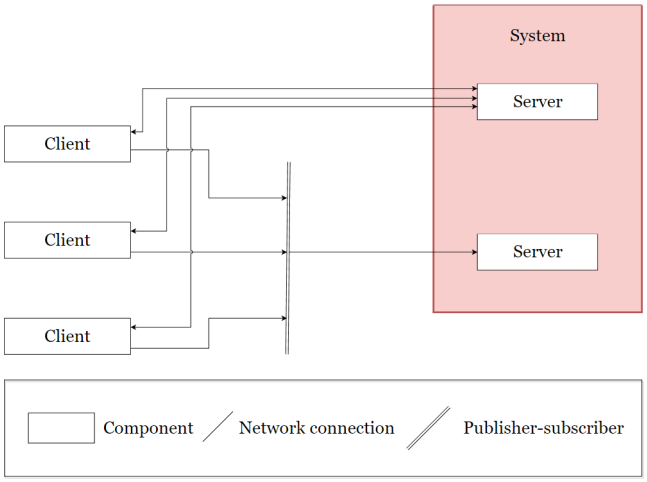

Figura 14: Schema logico di funzionamento del server.

Lo schema rappresenta come gli utenti possano accedere al server di autenticazione per effettuare il login ed iniziare una partita. I giocatori che entrano in partita diventano subscriber del gameserver, responsabile di comunicare gli aggiornamenti sulla partita in corso e gestire le dinamiche di gioco.

5.1.2 Scomposizione in microservizi

A partire dalle considerazioni precedenti é possibile determinare un possibile modello architetturale che garantisca le funzionalità necessarie con

particolare attenzione ai requisiti di qualità richiesti. É possibile introdurre un'**architettura a microservizi**, ossia costituita da un insieme di servizi indipendenti e specializzati. Un'architettura di questo tipo permette di ottenere:

- **Modularità** attraverso la scomposizione del sistema in sottosistemi indipendenti.
- **Scalabilità** dell'applicazione al crescere della domanda, avendo a disposizione la possibilità di istanziare più microservizi dello stesso tipo.
- **Performance** e bassa latenza, a patto di gestire opportunamente la comunicazione tra i servizi e di bilanciare il carico di lavoro.
- **Modificabilità** sfruttando l'alta coesione ed il basso accoppiamento tra i servizi.
- **Fault tolerance** migliore facendo uso di unità isolate.

Lo sviluppo di un sistema software service oriented, allora, può avvenire scomponendo il problema in un insieme di azioni da eseguire, assegnandole ad opportuni servizi. Bisogna tener conto che non é semplice realizzare da subito un workflow efficace che comprenda tutti i microservizi necessari, quindi un eventuale suddivisione iniziale può subire delle variazioni.

Una volta chiarita la suddivisione delle responsabilità, possiamo individuare che nel caso specifico di questa architettura, il lato server dell'applicazione é composto da un insieme di microservizi. Riportiamo di seguito il set di microservizi essenziali per il funzionamento dell'applicazione, e che derivano direttamente dalla fase di analisi. Il numero di microservizi, e la loro **successiva decomposizione** é riportata nella sezione successiva dedicata al diagramma dei componenti.

Il servizio di **Authentication** é specializzato nell'autenticazione dell'utente ed il servizio **Scoreboard** gestisce la visualizzazione dei dati globali dei giocatori. Authentication e Scoreboard sono due microservizi che implementano funzionalità esterne alla logica dei gioco. Il microservizio di **GameLogic** implementa la logica della partita, avvalendosi dei servizi di un **NeuralNetwork**. Al fine di ridurre il carico sui server di gioco, e per mettere in comunicazione i client con essi, é necessaria l'introduzione di un microservizio intermedio, il **GameProvider**. Questo microservizio ha

lo scopo di realizzare la funzionalità di matchmaking ma anche garantire che le partite vengano assegnate opportunamente ai game server disponibili. Il GameProvider implementa un modulo **Broker** che ha lo scopo di indirizzare gli utenti che sono pronti ad iniziare una nuova partita verso un game server sufficientemente libero. Per fornire le informazioni necessarie alla connessione, il GameProvider implementa un meccanismo a "Pagine bianche".

5.1.3 Diagramma dei componenti

Il diagramma dei componenti rappresenta le relazioni tra le singole strutture che compongono un sistema in una vista di progettazione statica. Nel farlo possono essere considerati aspetti di modellazione sia logici che fisici. Nel contesto UML, i *componenti* sono parti modulari di un sistema, che sono indipendenti e possono essere sostituiti da componenti equivalenti. Sono chiusi in sé e incapsulano tutte le strutture complesse desiderate. Per gli elementi incapsulati sono possibili contatti con altri componenti solo tramite interfaccia. I componenti possono mettere a disposizione le proprie interfacce, ma anche ricorrere a interfacce di altri componenti, per avere accesso ad esempio alle loro funzioni o servizi. Allo stesso tempo, in un diagramma dei componenti, le interfacce documentano le relazioni e interdipendenze di un'architettura software.

Il diagramma, nello specifico, presenta cinque componenti principali:

- **Client**: realizzato secondo il pattern Model-View-Controller. Offre delle interfacce per visualizzare la classifica, per effettuare registrazione e/o login e per giocare. Inoltre, usufruisce dei servizi messi a disposizione dai microservizi di Scoreboard e Authentication quali: *RequestScore*, *SendRegistration*, *SendLogin* e *Gamerequest*.
- **Authentication** e **Scoreboard**: forniscono tutti i casi d'uso non correlati al gioco come l'autenticazione e la consultazione della classifica (Scoreboard providing e Auth providing).
- **GameProvider:** ha il compito di gestire due importanti aspetti per la creazione delle partite:

  - esegue la logica di matchmaking attraverso un bus di eventi pub/sub
  - esegue il load balancing tra istanze multiple del servizio di *GameLogic*

  Inoltre ha il compito di interfacciarsi con il database per l'aggiornamento dello score.

- **GameLogic**: gestisce la logica di gioco. Permette ai giocatori di disegnare, inviare disegni e ricevere risultati basati sulla proposta della rete neurale. Si interfaccia col database per la selezione dei topic. Dovrà inoltre comunicare con la NeuralNetwork e al contempo accettare gli input provenient dai client.

- **NeuralNetwork**: Ha il compito di classificare le immagini dei disegni dei giocatori e proporre un'etichetta alla GameLogic.

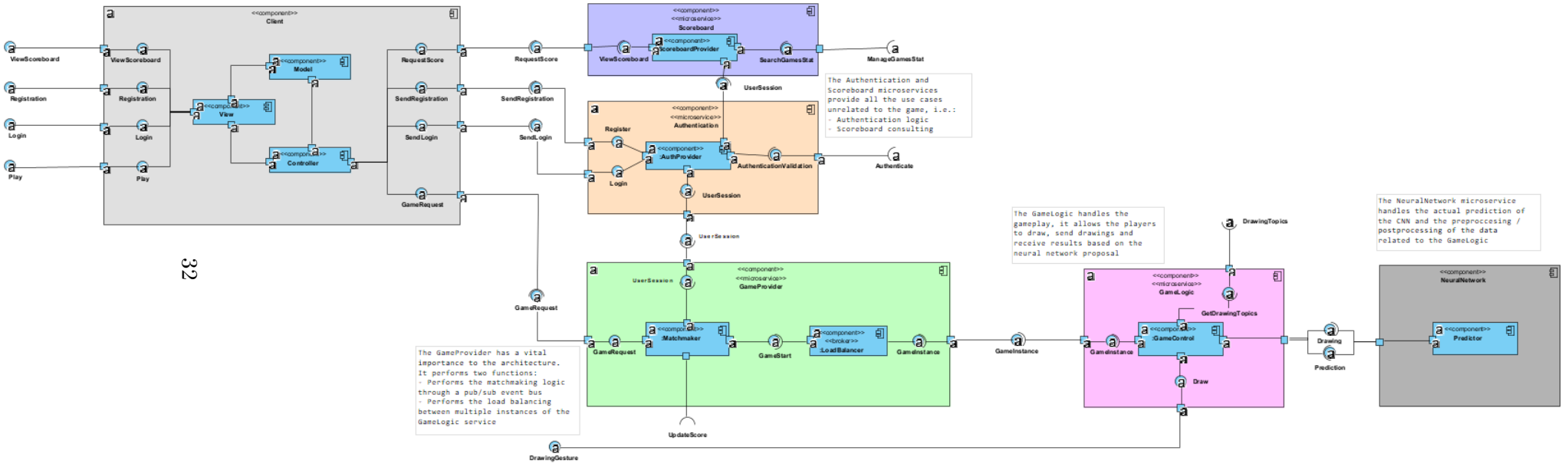

Figura 15: Diagramma dei componenti: architettura complessiva del sistema

5.2 Diagrammi di progettazione

5.2.1 Client

Si è scelto di utilizzare il pattern *Model-View-Controller* (MVC). L'utilizzo del pattern MVC favorisce la separazione delle responsabilità, favorendo la modificabilità del sistema ed il modo con cui è possibile interfacciarsi con l'utente. L'**MVC** è basato su tre diversi componenti software che interpretano tre ruoli principali:

- il **model** fornisce i metodi per accedere ai dati utili all'applicazione;
- la **view** visualizza i dati contenuti nel model e si occupa dell'interazione con untenti e agenti;
- il **controller** riceve i comandi dall'utente (in genere attraverso la *view*) e li attua modificando lo stato degli altri due componenti.

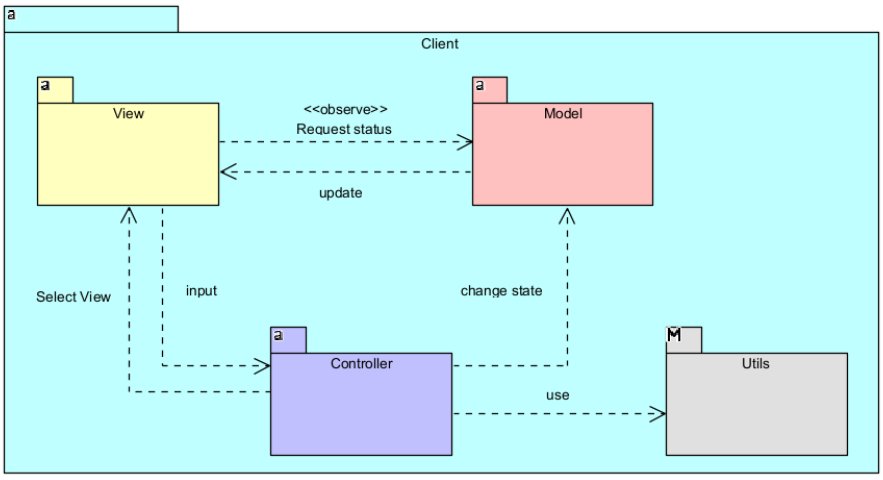

Figura 16: Diagramma dei package - Client

Nel caso specifico, il *Model* si compone delle classi precedentemente citate in fase di analisi con l'aggiunta della classe singleton "*Lobby*" e la classe "*Observable*". Tra le due classi vi è una relazione di *generalizzazione*. La lobby permette di tenere traccia del numero di giocatori in attesa di entrare in gioco.

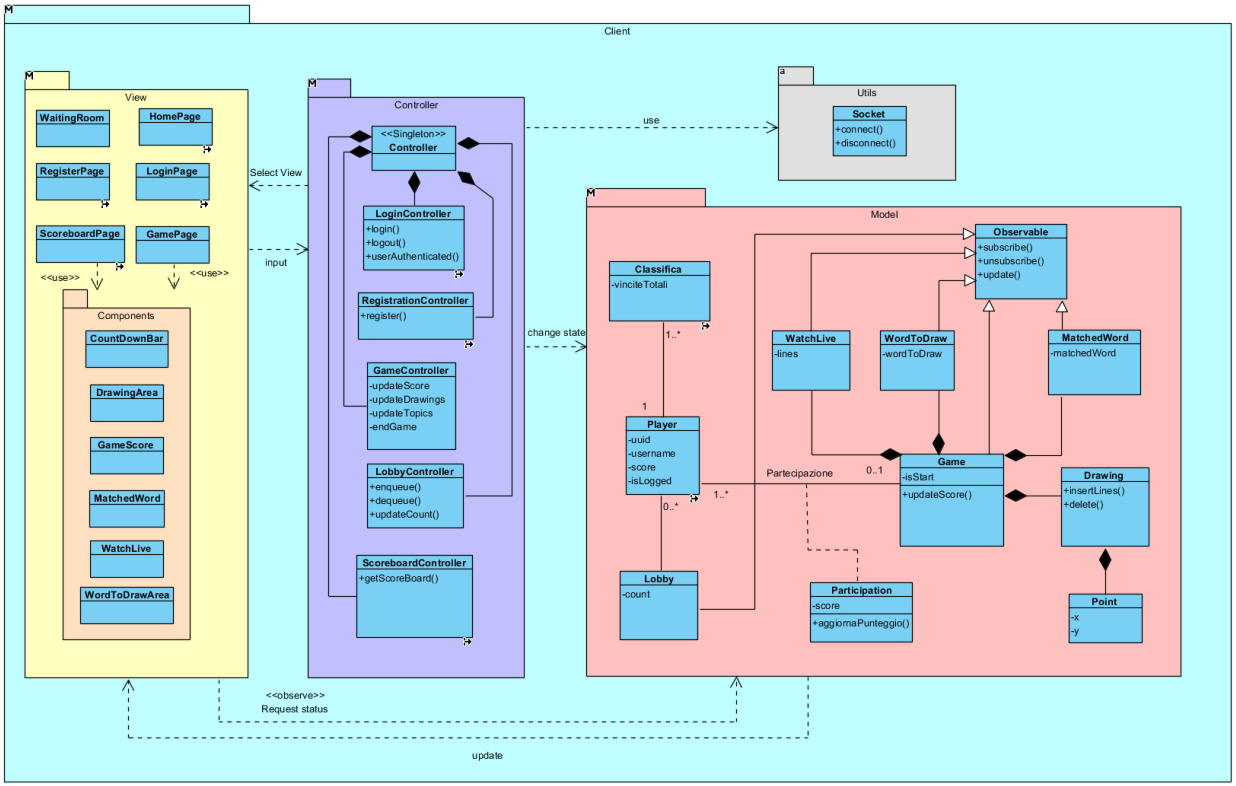

Figura 17: Diagramma delle classi - Progetto (Client)

- La *View* costituisce l'interfaccia e comprende:
  - WaitingRoom: l'utente in attesa di entrare in gioco, visualizza il numero di giocatori in attesa.
  - ScoreboardPage: l'utente visualizza la classifica.
  - RegisterPage: presenta i campi per poter effettuare la registrazione.
  - LoginPage: presenta i campi per poter effettuare il login.
  - HomePage: presenta dei bottoni che consentono all'utente di entrare in gioco oppure di visualizzare la classifica.
  - GamePage: presenta un'area di disegno dove il giocatore può rappresentare gli oggetti descritti e due tasti; uno per sottoporre il disegno a valutazione e l'altro per cancellare quanto prodotto. Inoltre, la GamePage permette ad ogni giocatore di visualizzare i disegni degli avversari.
- Il *Controller* presenta una classe singleton che istanzia tutte le altre.
  - LoginController: gestisce il login e logout.
  - RegistrationController: si occupa della registrazione dell'utente.
  - GameController: gestisce le dinamiche di gioco. Aggiorna il punteggio dei giocatori e permette di terminare il gioco.
  - LobbyController: aggiunge e rimuove i giocatori dalla lobby, gestendone il conteggio.
  - ScoreboardController: gestisce la visualizzazione della classifica.
- Il *Model* presenta le entità previste dal class diagram raffinato, conservando le medesime relazioni. In particolare:
  - Observable è la classe che realizza la logica dell' *Observer*.
  - Le entità WatchLive, WordToDraw e MatchedWord che ereditano dall'observer, vengono utilizzate come supporto alle operazioni di visualizzazione dei disegni degli utenti, le parole degli oggetti da disegnare e gli output della rete.

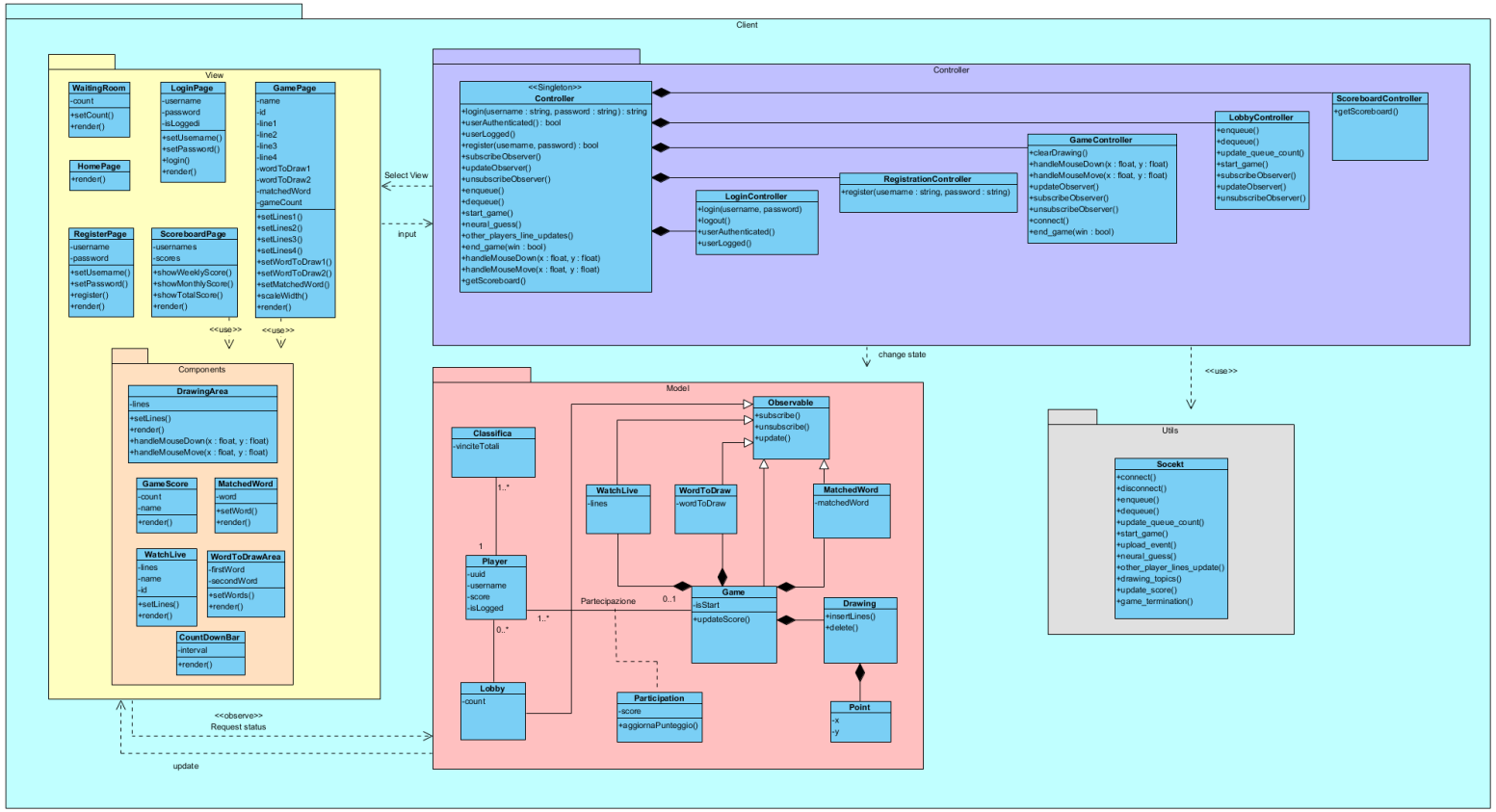

Figura 18: Diagramma delle classi revisionato - Progetto (Client)

5.2.2 Authentication e Scoreboard

- **Authentication**: è il microservizio che si occupa della autenticazione degli utenti. Esso ha il compito di realizzare l'operazione di registrazione, l'operazione di login e l'interfacciamento al database. Viene modellato con i componenti *Token, Registration, Login e dbInterfaceLogin*. Il componente di login utilizza il Token per l'autenticazione dell'utente. Il Token puó essere istanziato dall'opportuno metodo *createToken*.

- **Scoreboard**: é il microservizio che si occupa della visualizzazione della scoreboard quando richiesta dall'utente. Viene realizzato con i componenti *Scoreboard e dbInterfaceScore*.

  Entrambi i microservizi presentano una classe dbInterface che consente l'interfacciamento con il database. Tali classi, infatti, sono legati da una relazione di *usage* con gli altri componenti.

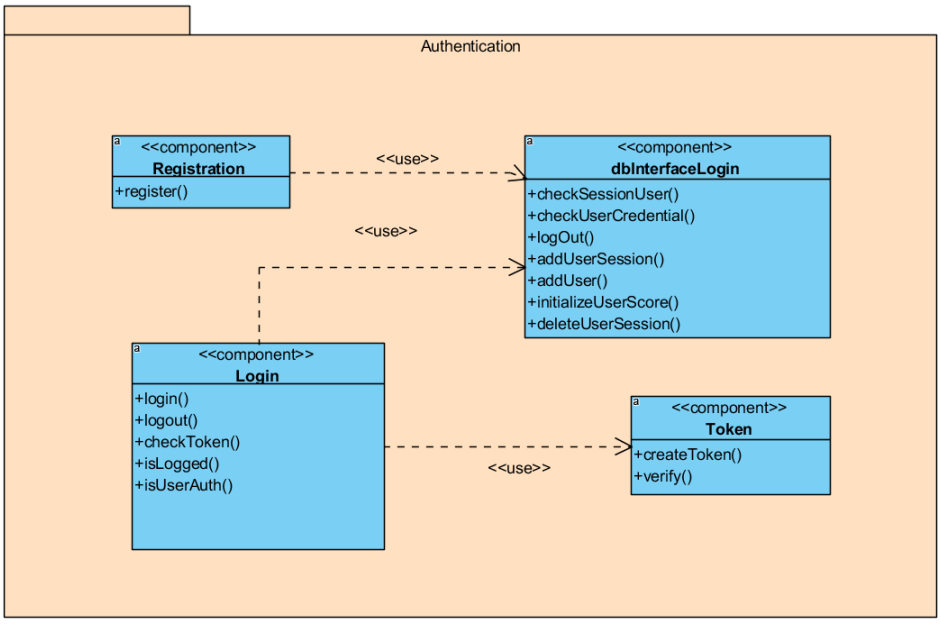

Figura 19: Diagramma delle classi - Authentication microservice

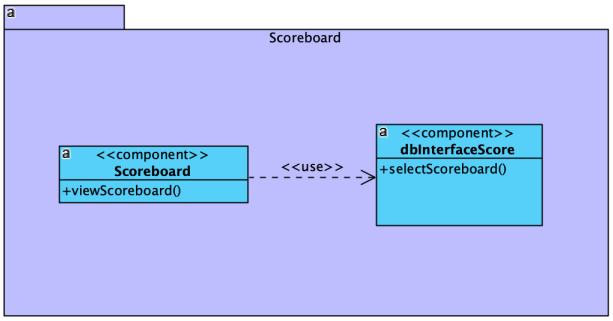

Figura 20: Diagramma delle classi - Scoreboard microservice

5.2.3 GameProvider

GameProvider ha il compito di gestire il matchmaking e assegnare i giocatori ad una partita garantendo il load balancing delle risorse computazionali a disposizione. Esso si interfaccia con le classi: Player,Game (Partecipation), MatchMaker e LoadBalancer.

- Il componente di MatchMaker è dotato dei metodi necessari alla gestione della coda dei giocatori: enqueue e dequeue, nonchè un metodo tryMatchmaking, utilizzato per provare ad avviare una partita ogni volta che un nuovo giocatore si unisce alla coda.
- Il componente di LoadBalancer ha la facoltà di avviare una nuova partita mediante il metodo spawn-game, nonché la funzione responsabile della comunicazione con la GameLogic, define-master-event.

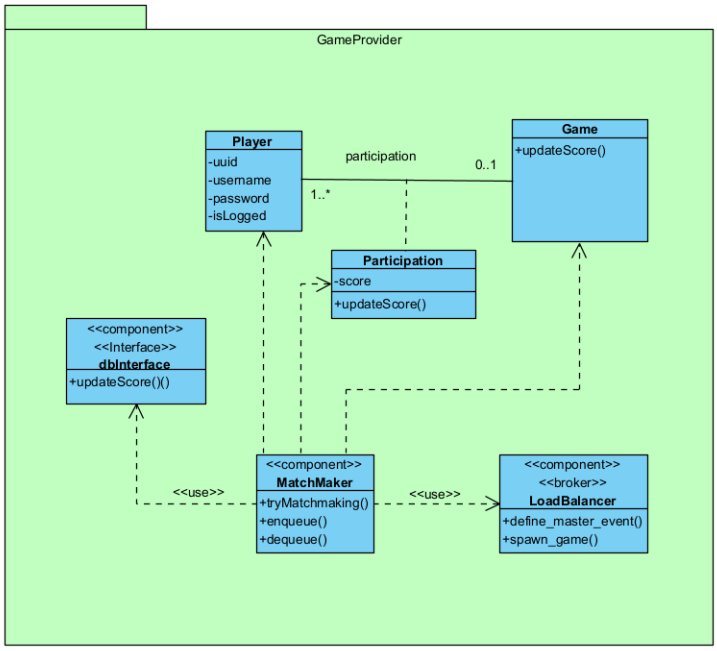

Figura 21: Diagramma delle classi -GameProvider microservice

5.2.4 GameLogic

Il GameLogic microservice ha il compito di gestire la logica di gioco. Il GameController ha il compito di gestire la logica della partita, permettere agli utenti di incominciare a giocare e gestisce l'aggiornamento dello score ed il gameover. Esso si interfaccia con le classi: Game, Partecipation e Player. In particolare, associa i giocatori ad una determinata partita e aggiorna il punteggio opportunamente.

La GameLogic comprende il componente GameControl:

- **Game Control:** gestisce tutte le dinamiche di gioco andando ad interfacciarsi con le classi: Game, Partecipation e Player. In particolare, associa i giocatori ad una determinata partita e aggiorna il punteggio opportunamente. La MessageQueue, invece, é l'entitá che gestisce l'interfacciamento con la rete neurale, fornendo operazioni di push per le immagini e pop per le predizioni.

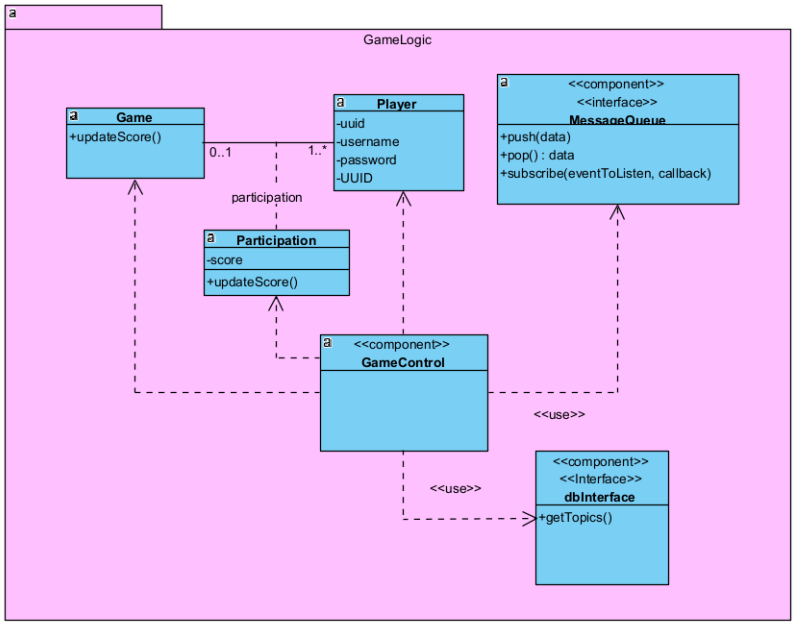

Figura 22: Diagramma delle classi - GameLogic microservice

5.2.5 Neural Network

Neural Network é il microservizio che ha il compito di classificare le immagini degli utenti e fornire un'etichetta del soggetto rappresentato.

La GameLogic comprende il componente Game Control:

- **Predictor:** Si occupa della vera e propria operazione di classificazione. Comprende le classi worker e image. La MessageQueue, invece, é l'entitá che gestisce l'interfacciamento con la Game Logic, fornendo operazioni di push per le predizioni e pop per le immagini.

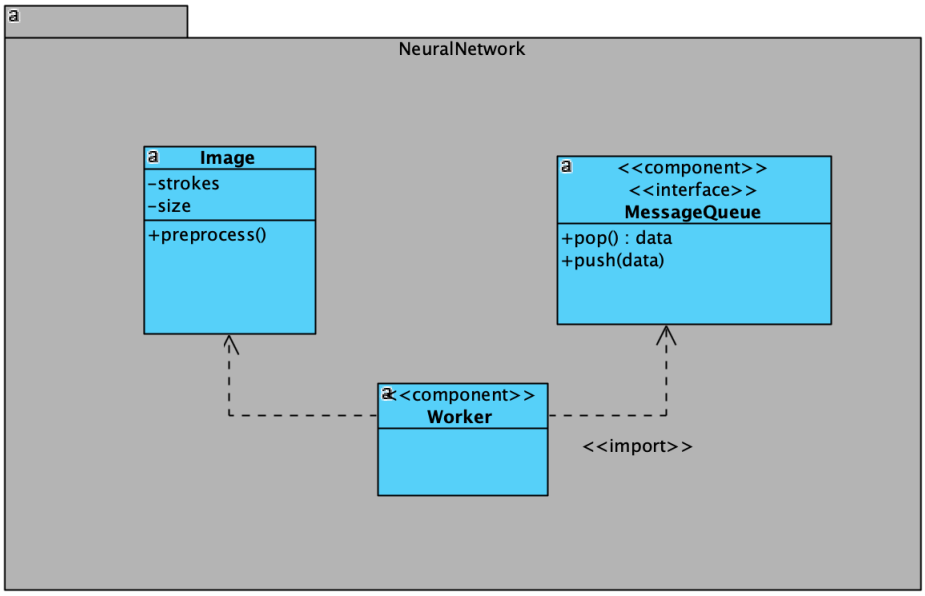

Figura 23: Diagramma delle classi - Neural Network microservice

5.3 Diagrammi di sequenza

Consiste in un gruppo di oggetti, rappresentati da lifeline, e nei messaggi che tali istanze si scambiano durante l'interazione. Analogamente ai diagrammi delle classi precedentemente presentati, si riportano separatamente i diagrammi inerenti al client e al server.

5.3.1 EffettuaLogin

Il *Player* manifesta la volontà di effettuare il login tramite l'apposita pagina (*LoginPage*) che invia una richiesta al *LoginController*. Questa richiesta viene inoltrata al *loginService* che riceve username e password.

Quest'ultimo fornisce come risposta l'autorizzazzione (*auth*) e il *token*:

- Se auth=false, la LoginPage restituisce al player un messaggio di errore.
- Se auth=true, il LoginController va ad aggiornare i dati del giocatore e notifica al PagesController di restituire la *HomePage*

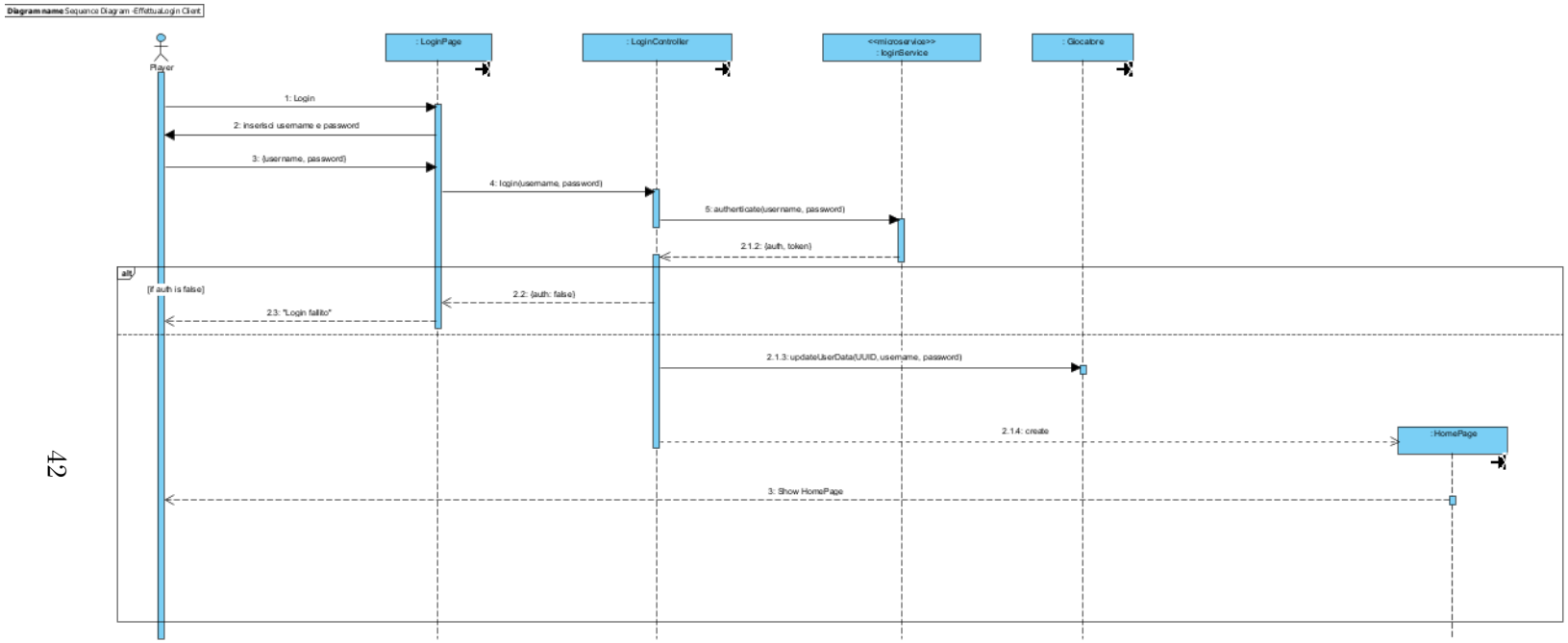

Figura 24: Diagramma di sequenza ”EffettuaLogin” - Client

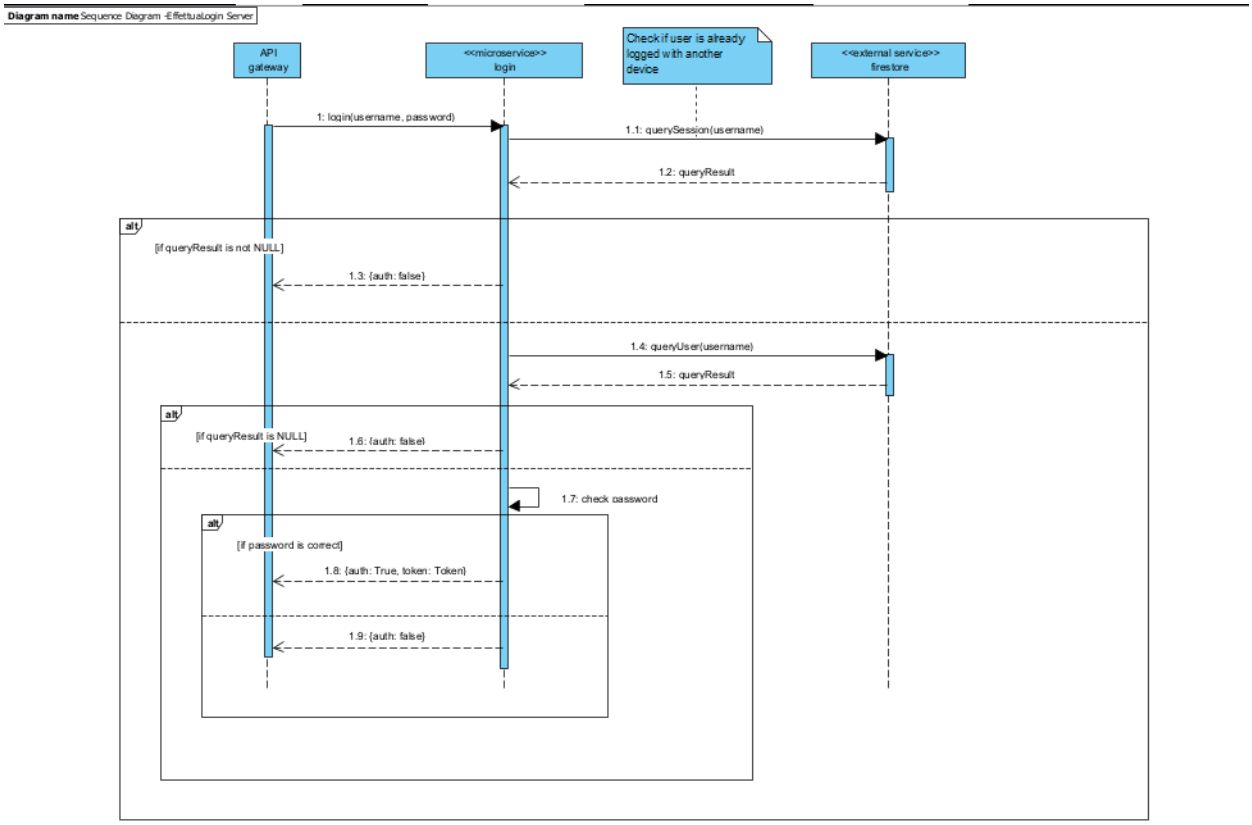

Figura 25: Diagramma di sequenza "EffettuaLogin" - Server

L'*API gateway* inoltra la richiesta di login all'apposito microservizio e quest'ultimo verifica, effettuando una query al database esterno sull'username relativo alla sessione, se l'utente già l'ha eseguito.

- Se la query restituisce risultato positivo, si nega l'autorizzazione.
- Se il risultato è negativo, si effettua una query sul campo username per verificare la presenza di quell'utente tra i registrati.
  - Se il risultato è negativo, si nega l'autorizzazione.
  - Se il risultato è positivo, il microservizio si occupa del controllo della password che, se corretta, permetterà all'API di ricevere l'autorizzazione e il Token.

5.3.2 EffettuaRegistrazione

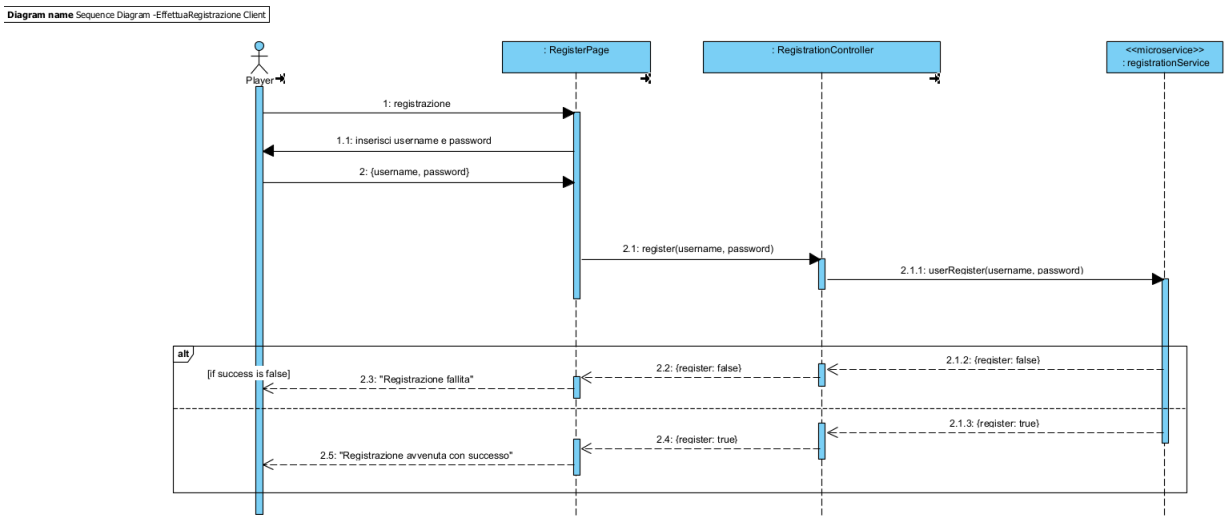

Figura 26: Diagramma di sequenza "EffettuaRegistrazione" - Client

Il *Player* manifesta la volontà di effettuare la registrazione tramite l'apposita pagina (*RegisterPage*) che invia una richiesta al *RegistrationController*. Questa richiesta viene inoltrata al *registrationService* che riceve username e password. Quest'ultimo fornisce una risposta e, a seconda del valore (*true* o *false*), viene inviato al *Player* un messaggio di registrazione avvenuta o di errore.

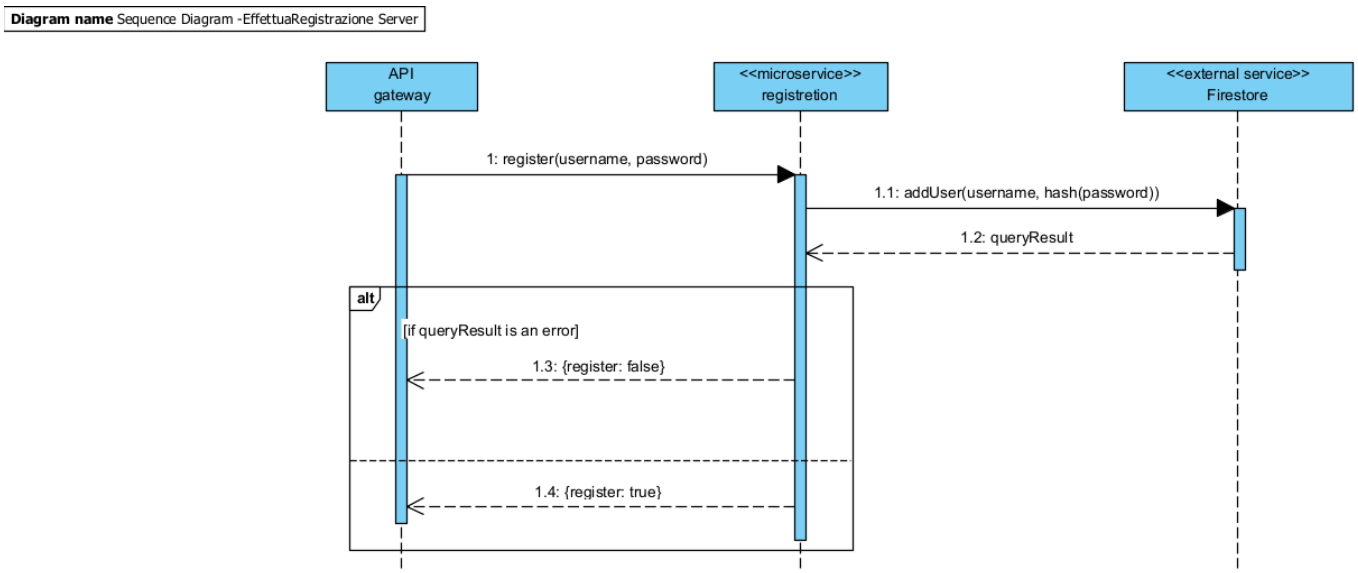

Figura 27: Diagramma di sequenza "EffettuaRegistrazione" - Server

L'*API gateway* inoltra la richiesta di registrazione all'apposito microservizio che aggiunge il nuovo utente al database esterno memorizzando username e password cifrata. Se la memorizzazione avviene con successo, l'API riceve *request=true*, altrimenti *request=false*, ovvero la richiesta effettuata non è andata a buon fine.

5.3.3 VisualizzaScoreboard

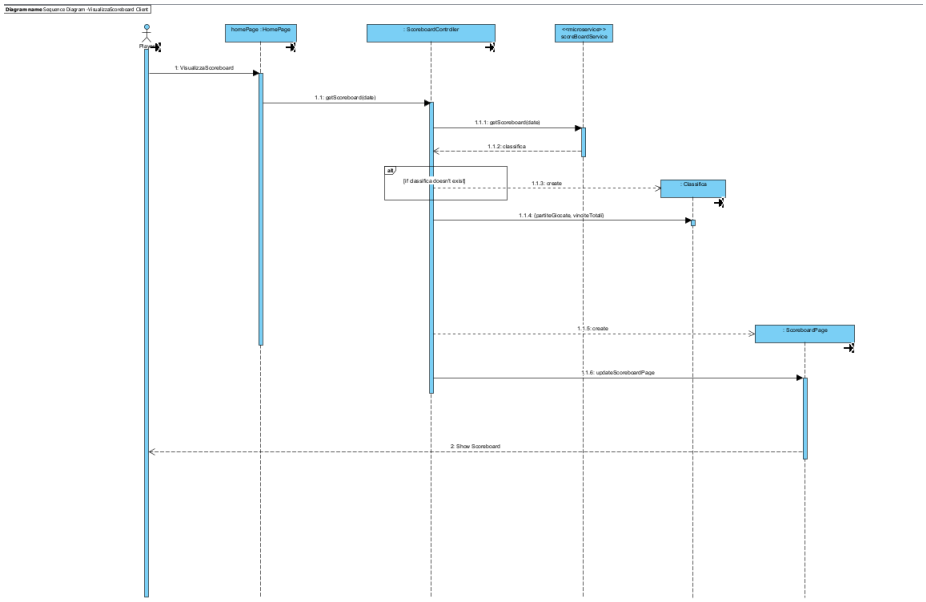

Figura 28: Diagramma di sequenza "VisualizzaScoreboard" - Client

Il *Player* manifesta la volontà di visualizzare la classifica tramite l'apposita pagina (*HomePage*) che invia una richiesta al *ScoreboardController*. Questa richiesta viene inoltrata al *scoreBoardService* che riceve la data. Questo restituisce la classifica.

- Se non esiste, viene creata.
- Se esiste, viene aggiornata la ScoreboardPage e lo ScoreboardController mostra la ScoreboardPage.

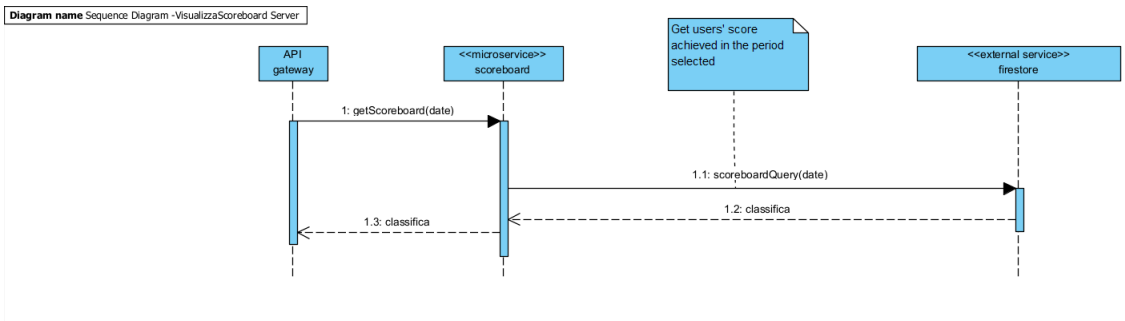

Figura 29: Diagramma di sequenza "VisualizzaScoreboard" - Server

L'*API gateway* inoltra la richiesta di visualizzazione della Scoreboard all'apposito microservizio che chiede al database esterno di restituire la classifica relativa ad un periodo selezionato.

5.4 Diagramma di attività

Il diagramma di attività modella un processo e organizza più entità in un insieme di azioni secondo un determinato flusso. Può essere usato per descrivere la logica con cui viene svolta una partita. Esso mette in evidenza il processo di accesso ad una partita tramite la lobby nonché lo svolgimento del gioco, il riconoscimento delle immagini tramite la rete ed il termine della partita allo scadere del timer.

Quando il giocatore clicca il bottone "*Play*" viene inserito nella coda di gioco. Se i giocatori sono inferiori a tre il gioco non inizia, altrimenti viene creata una stanza di gioco dal GameServer. Il gioco inizia per i giocatori selezionati e viene dato il via al timer di gioco. Quando il tempo prestabilito scade, il gioco termina. Quando il gioco inizia si entra in un loop. Vengono selezionate due possibili tipologie di disegno che vengono presentate all'utente. Il giocatore disegna quanto richiesto e viene sottoposto alla rete neurale:

- se la predizione è corretta, viene incrementato lo score del giocatore
- se la predizione è sbagliata, l'utente deve continuare a disegnare fino allo scadere del tempo.

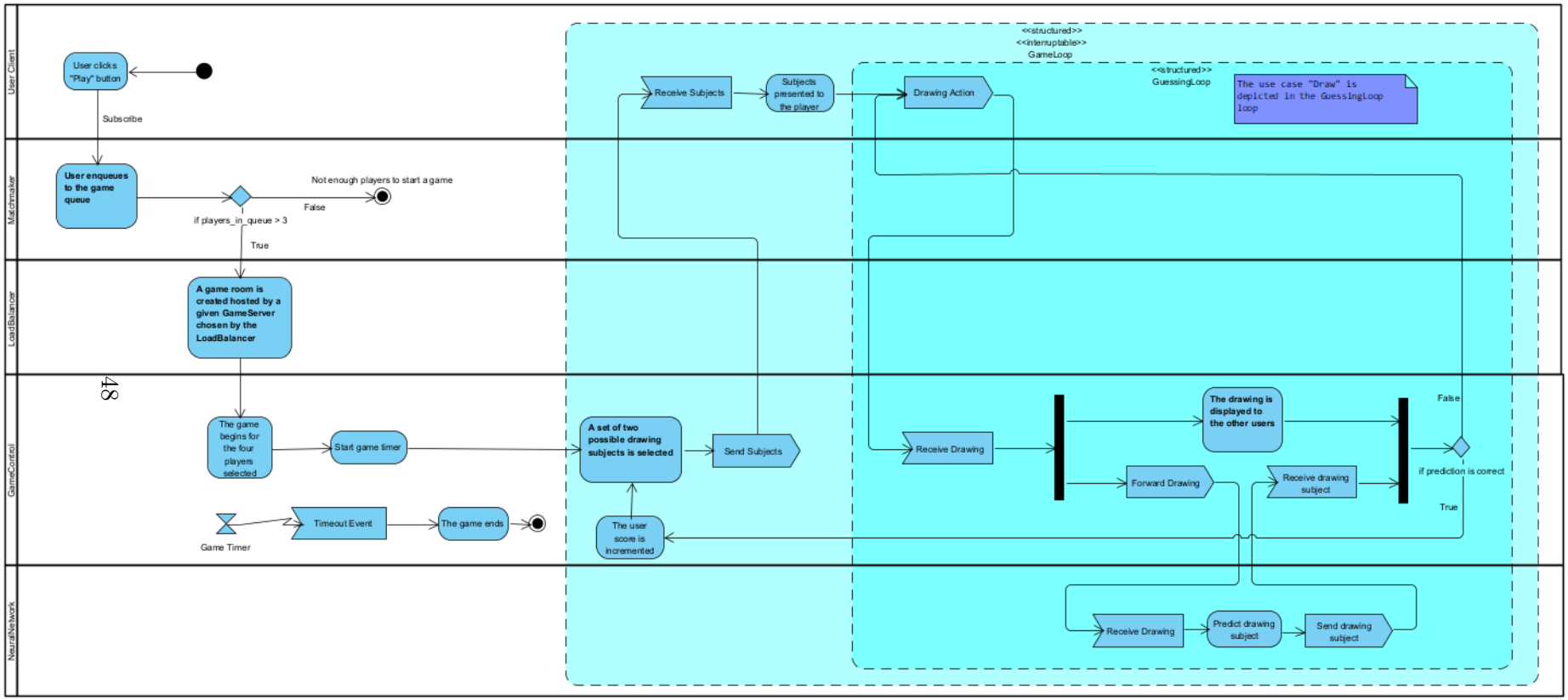

Figura 30: Diagramma di attività - Svolgimento Partita

6 Implementazione

6.1 Riconoscimento delle immagini

La logica del gioco prevede che un insieme di utenti si possano collegare ad una partita e che l'insieme dei disegni da loro prodotto sia classificabile. I giocatori hanno bisogno di un'entità che gestisca la partita in corso, implementando una game logic; tale logica deve opportunamente gestire i risultati della classificazione sui disegni prodotti. L'operazione di riconoscimento dei disegni prodotti e classificazione di essi, viene realizzata facendo utilizzo di una rete neurale artificiale. Tale rete, opportunamente addestrata su un set di immagini, è in grado di ricevere in input un immagine e restituire in output la classificazione del soggetto rappresentato. La rete utilizzata é *MobileNet*.

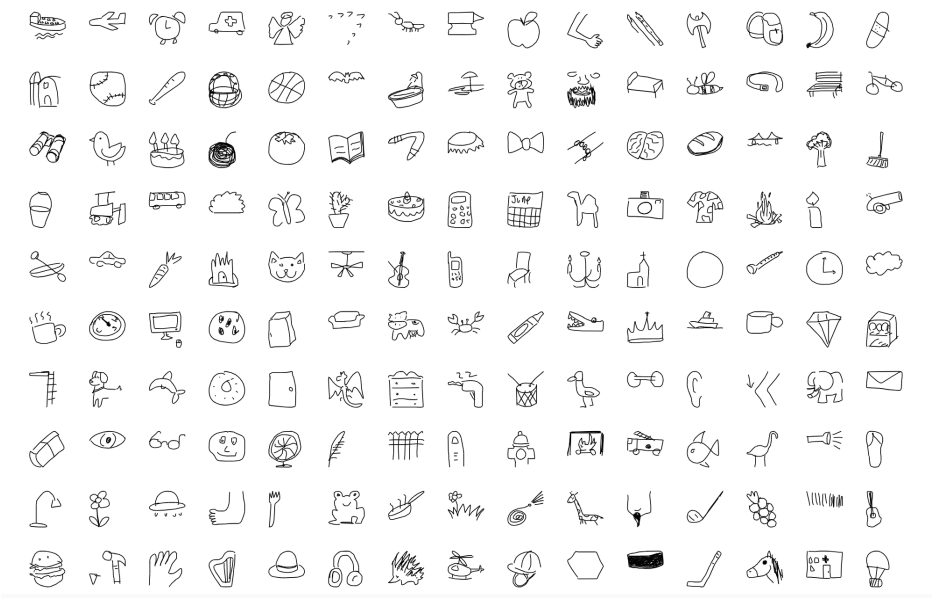

Figura 31: Esempi di disegni che costituiscono il dataset

Il dataset su cui la rete è addestrata è costituito da 10.000.000 di immagini ed un totale di 340 oggetti riconoscibili.

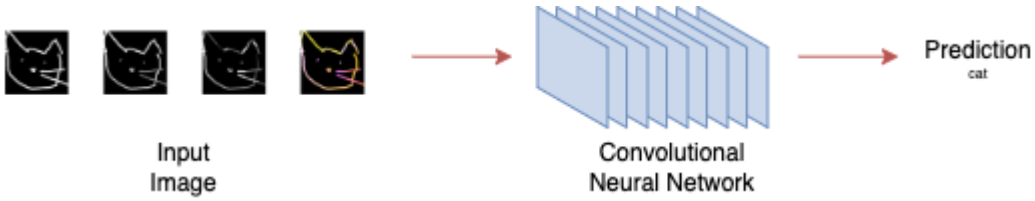

Figura 32: Esempio di classificazione di un disegno.

La rete, dovendo gestire un gran numero di possibili oggetti, deve garantire buoni risultati nella precisione delle classificazioni. La realizzazione di una logica di gioco coerente e corretta nei confronti del giocatore deve mantenere più bassa possibile la percentuale di errori di classificazione, evitando di sfavorire i giocatori.

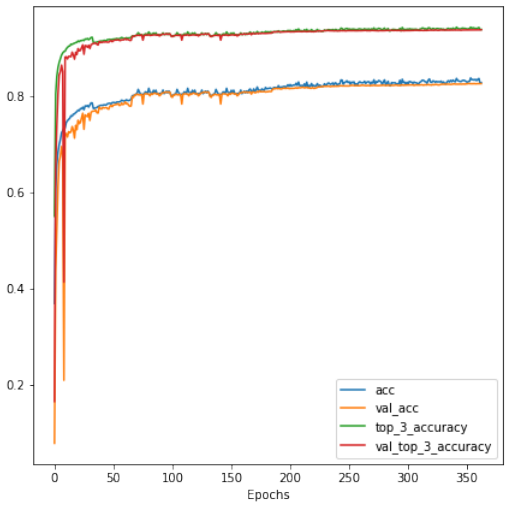

Figura 33: Percentuali di accuratezza della rete.

I risultati dell'addestramento della rete, evidenziano una percentuale di accuratezza di oltre il 98% per la top 3 delle classi e, in generale, una precisione media dell'80 %. Percentuali di questo tipo sono del tutto applicabili al sistema da realizzare.

6.2 Implementazione scelte progettuali

Lo sviluppo di un multiplayer game che permetta ad utenti di utilizzare i servizi offerti dal software da remoto, ha permesso di indirizzare alcune scelte implementative rivolgendole alla usabilità del sistema, alle performance ed in particolare alla scalabilità. Per lo sviluppo del client é stato utilizzato Nodejs/React, permettendo di ottenere interfacce grafiche semplici ed accattivanti per gli utenti che vi interagiscono; la scelta di React consente inoltre una elevata portabilità dell'applicazione ed una semplice configurazione. Anche il microservizio per l'autenticazione é stato implementato utilizzando Node.js, in particolare, esso si interfaccia con un servizio esterno, la piattaforma Firebase con la quale é possibile gestire i dati associati agli utenti con un *realtime database noSQL*. La GameLogic é stato implementato con Node.js ed é in comunicazione con il client; il microservizio per la classificazione delle immagini, invece, prevede l'implementazione di un modulo python per l'avvio della rete neurale. Per far fronte alla mole di dati scambiati tra GameControl e NeuralNetwork, é stato utilizzato Redis, allo scopo di fornire un archivio condiviso in cui fosse possibile effettuare operazioni di push e pop per le code di messaggi scambiate. Redis può consentire sia la comunicazione remota tra questi microservizi, sia quella locale. Per ridurre la latenza per la comunicazione con una rete neurale in esecuzione remota é stato ritenuto opportuno dislocare entrambi i microservizi di GameLogic e Classificazione sulla stessa macchina fisica. La descrizione specifica di ciascuno dei servizi utilizzati é fornita nella sezione successiva con particolare riferimento agli obiettivi di qualità raggiungibili.

6.2.1 Autenticazione con JWT

Il meccanismo di autenticazione implementato, prevede il rilascio di un token. **JWT** è l'acronimo di **JSON Web Token** ed è un sistema di cifratura e di contatto in formato JSON per lo scambio di informazioni tra i vari servizi di un server. Ogni volta che un utente, comunica al server le sue credenziali di accesso questo risponde rilasciando un token ci acceso che resta valido (per un certo tempo) per gli accessi dell'utente al sistema. Ogni token dispone, oltre di header e payload, anche di una firma, utilizzata per implementare un meccanismo di cifratura a chiave pubblica.

Il client comunica username e password all'autentication service. Se le credenziali di accesso sono corrette, il server invia in risposta un JWT. Il client potrà usufruire degli altri servizi del sistema, senza effettuare nuovamente l'accesso e sfruttando il token precedentemente assegnatogli.

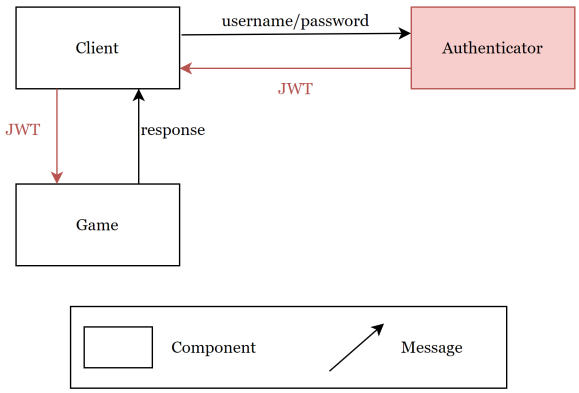

Figura 34: Logica di autenticazione.

6.2.2 Brokering e load balancing

Uno degli aspetti piú importanti per l'ottentimento di buone performance del sistema é la gestione delle risorse disponibili. L'architettura proposta si dispiega con un insieme di GameServer, pronti a gestire le partite degli utenti, classificando le immagini e rispondendo con il tempo di latenza piú basso possibile. É altrettanto importante, in un architettura a microservizi, l'esistenza di un entitá intermediaria per la comunicazione tra i microservizi disponibili. Il microservizio GameProvider in tal senso, ha proprio il compito di porsi come intermediario tra i giocatori in attesa di entrare in partita ed il GameServer stesso; questa intermediazione viene realizzata con un pattern di *brokering a "pagine bianche"*.

Il componente Load Balancer del GameProvider ha il compito di tenere traccia dei microservizi GameServer correntemente istanziati. Quando un gruppo di giocatori é pronto ad iniziare una nuova partita, il processo **master**, assegna per quella partita il GameServer meno utilizzato in quel momento, inviando un messaggio al processo **Daemon** corrispondente. il Load Balancer ha traccia di quante sono le partite in corso su ciascun GameServer sulla base dei messaggi precedentemente scambiati. Una volta ricevuto il messaggio di assegnazione, il Daemon effettua una fork, istanziando di fatto un processo **Worker** su cui entra in esecuzione la partita. Differenti messaggi allo stesso Daemon corrispondono a molteplici worker in esecuzione e quindi un maggior numero di partite in svolgimento parallelamente in riferimento alla stessa macchina fisica. Idealmente, ogni daemon puó istanziare un numero infinito di worker, dal punto di vista pratico, un numero di partite troppo elevato puó saturare la rete neurale. Per evitare

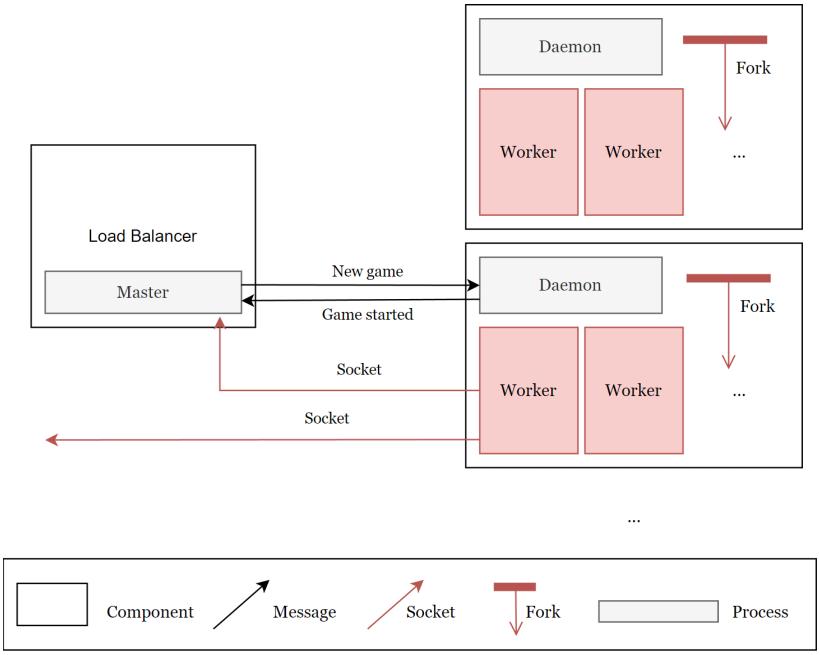

Figura 35: Logica brokering e load balancing.

a situazioni del genere, ed in previsione di un elevata affluenza di utenti rispetto alle previsioni, é possibile dispiegare un numero superiore di Game Server sull'architettura.

Ogni Worker istanziato comunica sia con il master, condividendo informazioni sulla partita in corso, ma anche con i client che sono stati assegnati a quel GameServer; Questo canale di comunicazione é rappresentato da due *socket*.

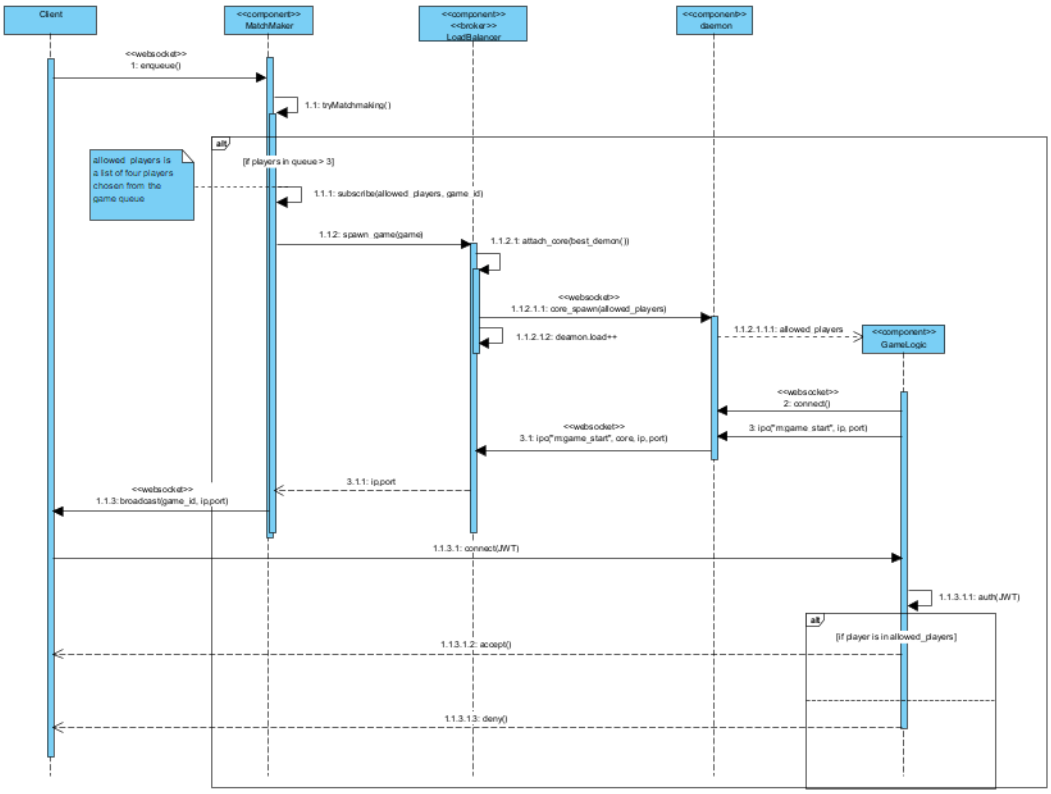

Figura 36: Diagramma di sequenza del load balancer

6.3 Framefork e servizi utilizzati

6.3.1 Node.js

Runtime system open source multipiattaforma orientato agli eventi per l'esecuzione di codice JavaScript. Consente di utilizzare JavaScript anche per scrivere codice da eseguire lato server, ad esempio per la produzione del contenuto delle pagine web dinamiche prima che la pagina venga inviata al browser dell'utente. Node.js in questo modo permette di implementare il cosiddetto paradigma "*JavaScript everywhere*", unificando lo sviluppo di applicazioni Web intorno ad un unico linguaggio di programmazione. Ha un'architettura orientata agli eventi che rende possibile l'I/O asincrono. Questo design punta ad ottimizzare il **throughput** e la **scalabilità** nelle applicazioni web con molte operazioni di input/output. É inoltre ottimo per

applicazioni web sistema real-time (programmi di comunicazione in tempo reale o browser game).

6.3.2 React

React rende semplice la creazione di UI interattive. Permette la progettazione di interfacce per ogni stato dell'applicazione. Ad ogni cambio di stato React aggiorna efficientemente solamente le parti della UI che dipendono da tali dati. La natura dichiarativa dell'UI rende il codice più prevedibile e facile da debuggare. Consente la creazione di componenti in isolamento, componibili per creare UI complesse. Dato che interazioni e logica per i componenti sono implementate in JavaScript si può facilmente passare ed accedere strutture dati complesse in vari punti dell'applicazione. Si è scelto di utilizzare questo web framework in quanto può effettuare rendering lato server con Node e in applicazioni mobile grazie a *React Native*.

6.3.3 Firebase

É una piattaforma serverless per lo sviluppo di applicazioni mobili e web. Open source ma supportata da Google, Firebase sfrutta l'infrastruttura di Google e il suo cloud per fornire una suite di strumenti per scrivere, analizzare e mantenere applicazioni cross-platform. Firebase infatti offre funzionalità come analisi, database (usando strutture noSQL), messaggistica e segnalazione di arresti anomali per la gestione di applicazioni web, iOS e Android.
In particolare, sono stati utilizzati i servizi legati ad *autenticazione* e *storage*.

6.3.4 Redis

Redis, acronimo di Remote Dictionary Server, è un archivio dati veloce, open source, in memoria e di tipo chiave-valore. Offre tempi di risposta inferiori al millisecondo, consentendo milioni di richieste al secondo per applicazioni in tempo reale in settori come *giochi*, tecnologia pubblicitaria, servizi finanziari, sanità e IoT. I principali vantaggi sono:

- **Prestazioni:** a differenza dei database tradizionali, gli archivi dati in memoria non richiedono un accesso al disco, riducendo la latenza. Per questo motivo, essi possono supportare una quantità di operazioni di ordine di grandezza superiore e tempi di risposta più rapidi. Il risultato sono prestazioni incredibilmente veloci con operazioni di lettura

e scrittura medie che richiedono meno di un millisecondo e supporto per milioni di operazioni al secondo.

- **Strutture dati flessibili:** dispone di un'ampia varietà di strutture dati per soddisfare le esigenze dell'applicazione.
  I tipi di dati Redis includono: String, List, Set, SortedSet, Hash, Bitmap, JSON.

- **Semplicità e intuitività:** si può utilizzare una semplice struttura di comandi rispetto ai linguaggi di query dei database tradizionali. Include strutture dati native e diverse opzioni che permettono di gestire e interagire con i dati. Sono disponibili più di cento client open source. Tra i linguaggi supportati sono presenti Java, Python, PHP, C, C++, C, JavaScript, Node.js e molti altri.

- **Replica e persistenza:** i dati vengono replicati su diversi server appositi. In questo modo è possibile ottenere *migliori prestazioni in lettura* (poiché le richieste vengono suddivise tra i diversi server) e *ripristino* in caso di interruzione del server principale. Per la persistenza, Redis supporta backup point-in-time (supporta il ripristino delle modifiche dei dati fino a un determinato momento).

- **Disponibilità e scalabilità elevate:** offre un'architettura primary-replica in un singolo nodo principale o una topologia a cluster. In questo modo è possibile creare soluzioni dotate di elevata disponibilità e prestazioni costanti e affidabili. Le dimensioni di un cluster possono essere modificate in vari modi, secondo modelli di scalabilità orizzontale e verticale.

- **Open source:** non esistono vincoli di fornitore o di tecnologia

6.3.5 Code di messaggi

L'utilizzo di Redis ha reso possibile lo scambio di messaggi tra Game Controller e rete neurale. Il Game Server ha l'esigenza di gestire la comunicazione tra questi due componenti, in particolare, il controller deve accodare le immagini ricevute dagli utenti in una coda e prelevare i risultati di predizione. Per realizzare questo tipo di logica, é stato deciso di realizzare due code di messaggi condivise: la coda delle immagini e quella delle predizioni.

- La **coda delle immagini** é la coda in cui vengono conservate le immagini che devono essere classificate dalla rete. GameControl puó effettuarvi il *push* e NeuralNetwork il *pop*.

- La **coda delle predizioni** contiene l'insieme delle classificazioni effettuate dalla rete. NeuralNetwork puó effettuarvi il *push* e GameControl puó effettuarvi il *pop*.

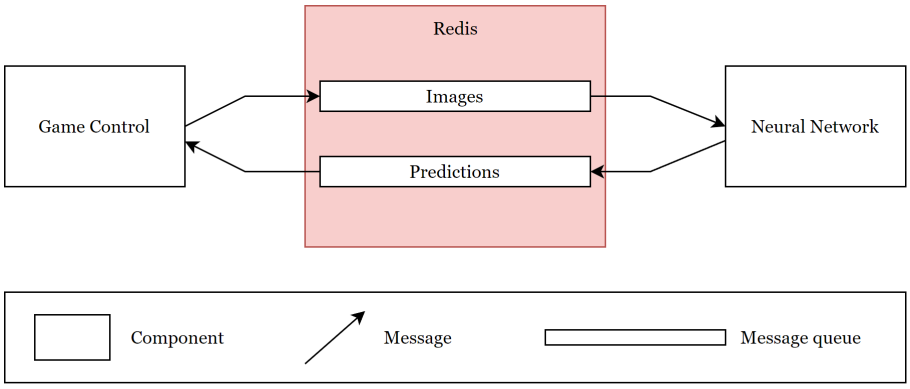

Figura 37: Logica di gestione dei messaggi.

6.3.6 Interfaccia

Più volte é stato sottolineato quanto é importante disporre di un folto bacino di utenti da cui attingere. Il successo di un multiplayer game si può misurare sopratutto in funzione dell'affluenza di giocatori nel tempo. La chiave per venire incontro alle esigenze di utenti diversi per età e abilità nell'utilizzo di dispositivi digitali, rende molto importante la cura dell'usabilità e della portabilità dell'applicativo. Se la scelta di React/Node garantisce un ottimo grado di portabilità, altrettanto importante é la gestione delle interfacce grafiche.

Figura 38: Logo app

Per rendere l'applicazione unica e riconoscibile é stato creato un logo che richiamasse l'azione del disegno.
L'applicazione presenta una pagina di login composta da form che consentono l'inserimento di username e password e, nell'eventualità di nuovi utenti, un link che ne consente la registrazione.

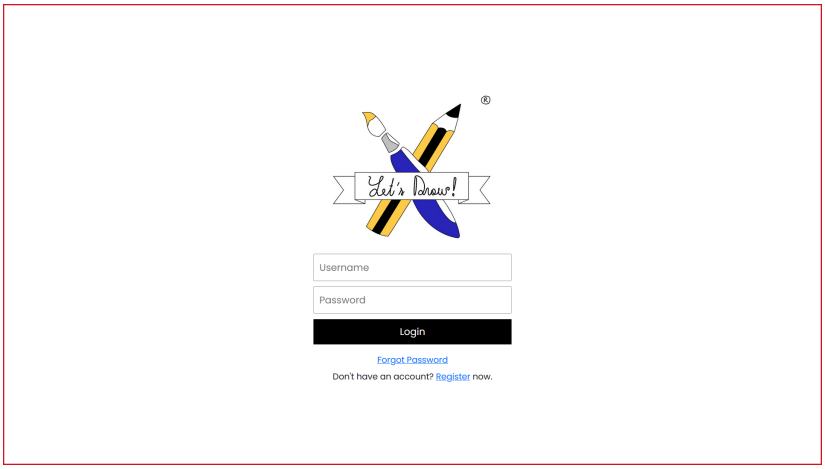

Figura 39: Login Page

La prima pagina che compare in seguito all'operazione di login presenta un design accattivante che descrive nell'immediato il topic del gioco. Predomina ovviamente il logo seguito dal nome del player e due pulsanti che consentono all'utente di iniziare una partita oppure visualizzare la classifica.

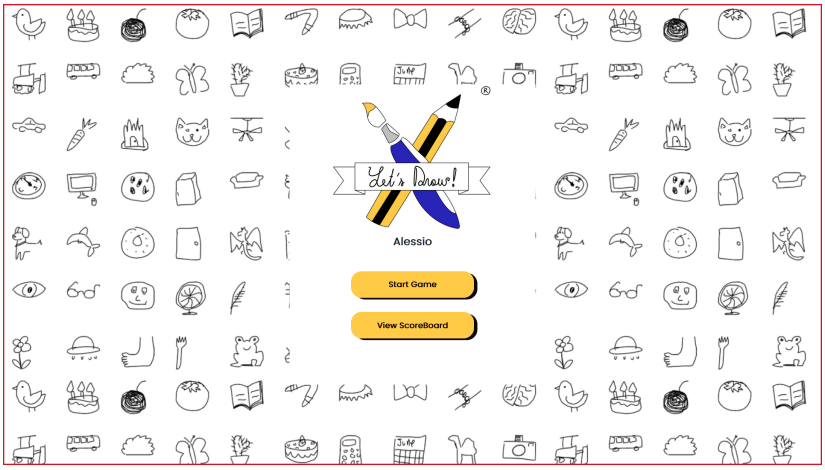

Figura 40: Home Page

Quando l'utente clicca il pusante "*Start Game*" viene posto in attesa di altri giocatori per poter iniziare la partita. Visualizzerà una gif e quanti giocatori sono in coda.

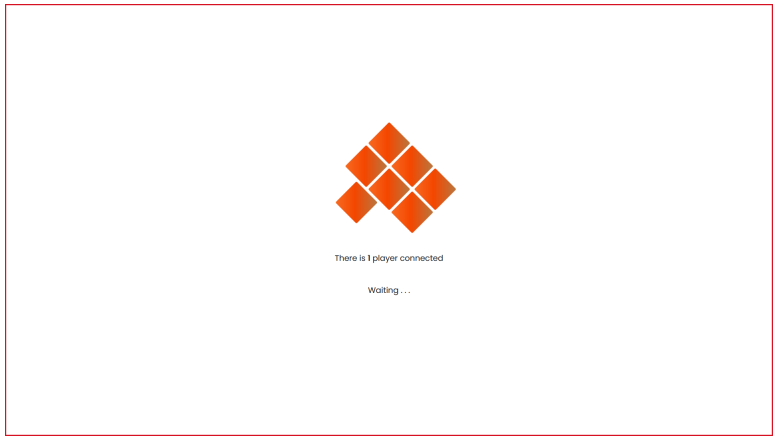

Figura 41: Waiting Room

6.4 Interfaccia fisica

I metodi esposti al client sono implementati tramite due interfacce fisiche:

- RESTful HTTP
- WebSocket

Metodo	URL	Request Body	Response Body

Post	/register	{
		”username”: <username>,
		”password”: <password>
		}
			”message”: <result>

Post	/login	{
		”username”: <username>,
		”password”: <password>
		}
			{
			”auth”: <result>,
			”token”: <token>,
			”uuid”: <uuid>,
			”message”: <message>,
			}

Get	/isLogged	{
		”token” : <token>
		}
			{
			”loggedin”: <result>,
			”message”: <message>
			}

Get	/isUserAuth	{
		”token” : <token>
		}
			{
			”auth”: <result>,
			”message”: <message>
			}

Get	/Logout	{
		”uuid” : <uuid>
		}
			{
			”logout”: <result>,
			”message”: <message>
			}

Post	/viewScoreboard	{
		”date” : <date>
		}
			”scoreBoard”: <score>

Type	Message

connect	{
	”token”: <JWT>
	}

onUpdateQueueCount	{
	”value”: <count>
	}

onStartGame	{
	”players” : <message>
	”game-server”: <ip>
	}

UploadEvent	”lines” : <lines>

onOtherPlayerLineUpdate	{
	”id” : <uuid>,
	”payload”: <lines>
	}

onDrawingTopics	{
	”topic1” : <topic1>,
	”topic2” : <topic2>
	}

onUpdateScore	”score” : <score>
onWinningEvent	{}
onLosingEvent	{}
disconnect	{}

6.5 Diagramma di deployment

Viene sfruttato per modellare l'hardware utilizzato per l'implementazione del sistema e i collegamenti tra i diversi dispositivi hardware. Si possono disporre i singoli componenti (file ottenuti dopo il building), così da mettere in evidenza la loro collocazione nei vari elaboratori.

Per il deployment dell'applicazione viene realizzato coerentemente alla progettazione dell'architettura a microservizi, con la possibilità di poter dispiegare ogni microservizio su una macchina fisica differente. Unica eccezione è rappresentata dai microservizi Game Control e Neural Network, i quali, per motivi di latenza e per ottimizzare le prestazioni generali, eseguono sulla stessa macchina, sfruttando Redis come canale di comunicazione. In caso di carenza di dispositivi la seguente configurazione può, in ogni caso, essere modificata con la possibilità di affiancare 2 microservizi sulla stessa macchina (ad esempio Scoreboard e Authentication microservice). Firebase, invece, in qualità di servizio esterno, non è gestito direttamente dalle macchine fisiche in nostro possesso.

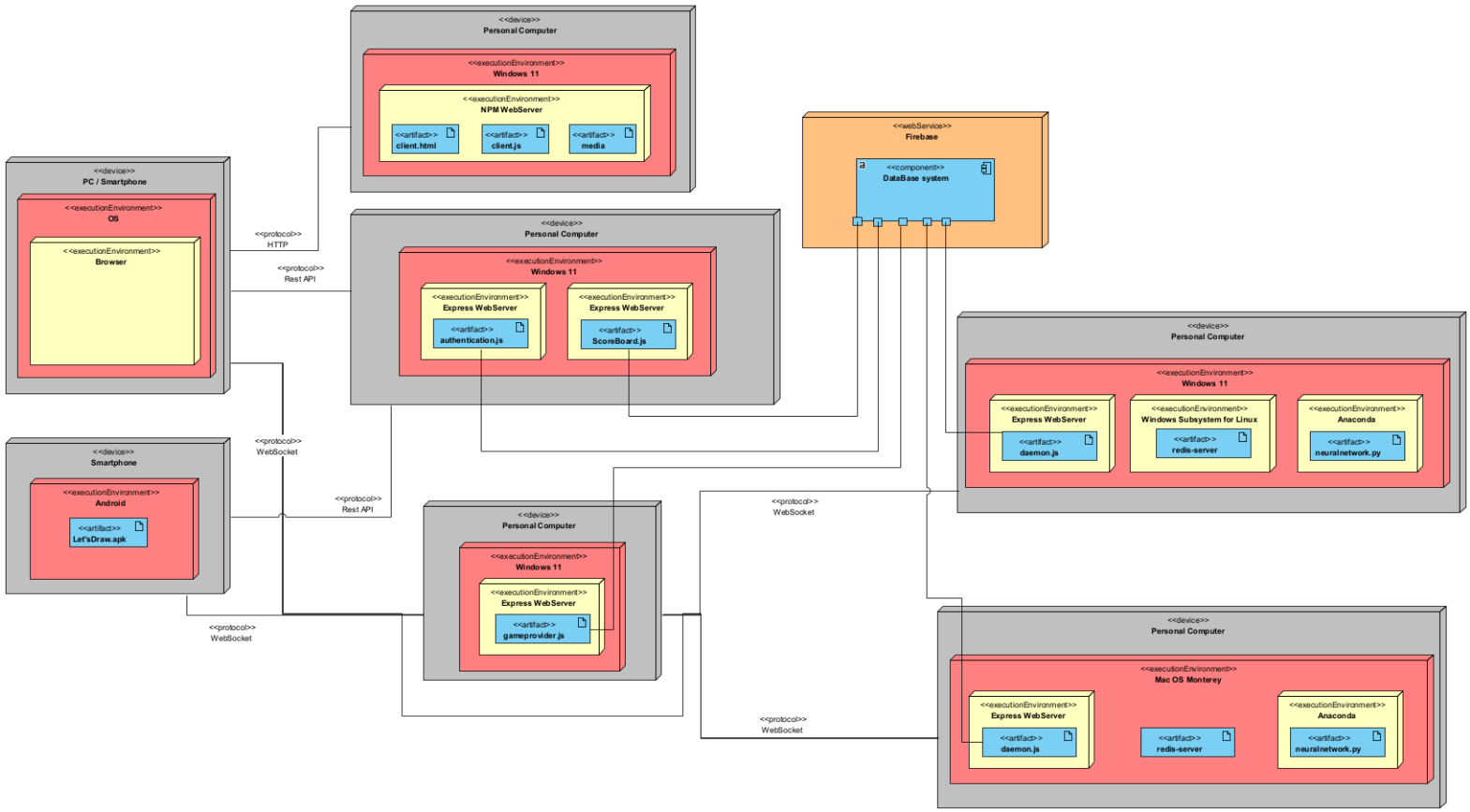

Figura 42: Deployment Diagram

7 Test

7.1 Unit Test

Serve per testare singole unità software, ovvero, piccoli componenti che hanno un funzionamento autonomo. Nelle applicazioni web permette anche di verificare che una determinata porzione di codice continui a funzionare anche quando le librerie o i tool che utilizza vengono aggiornati. Nel caso specifico sono stati effettuati test manuali in quanto le dipendenze dei singoli moduli sono difficilmente simulabili.

7.2 End-to-end Testing

Comprende il test di interfacce e delle dipendenze esterne, come l'ambiente in cui viene eseguito e il backend. Possono essere verificate non solo proprietà funzionali ma anche proprietà non funzionali e prestazioni. L'obiettivo dei test end-to-end è proprio quello di identificare le dipendenze nel sistema, assicurandosi che le informazioni scambiate tra i vari suoi componenti o con altri sistemi siano corrette. L'intera applicazione è quindi testata in uno scenario reale in cui interagisce con un utente o con un database. Esistono degli standard che permettono di automatizzare questa categoria di test come, ad esempio, Cypress.
Cypress testa qualsiasi cosa venga eseguita in un browser web. Tutta l'architettura di Cypress è costruita per gestire al meglio i moderni framework JavaScript. Funziona altrettanto bene anche su pagine o applicazioni renderizzate su server di vecchia generazione.
Per tutte le pagine previste dall'applicazione, si controlla che tutto ciò che compone l'interfaccia sia visibile. Inoltre, considerandole singolarmente:

- **Login Page:** controlla la correttezza dell'inserimento username e password, un'eventuale sessione già aperta dell'utente e login effettuato con successo.
- **Register Page:** controlla la corretta registrazione dell'utente con relativo popup.
- **Home Page:** controlla se al click dei bottoni viene settato l'URL corretto.

7.3 Testing delle API

Il test delle API determina se le API (Application Programming Interface) soddisfano le specifiche di funzionalità, coerenza, efficienza, usabilità, prestazioni e sicurezza. Inoltre, aiuta a scoprire bug, anomalie o discrepanze dal comportamento previsto di un'API. Durante lo sviluppo di un'API, cicli di feedback rapidi aiutano a garantire che funzioni nel modo desiderato e restituisca i dati previsti. Il testing è stato automatizzato utilizzando "Insomnia".

Insomnia è un potente client API REST multipiattaforma che è molto facile da usare e ha un'interfaccia utente ben congegnata e intuitiva. Contiene tutte le funzionalità di base per testare le API REST come la creazione di richieste HTTP con la possibilità di aggiungere intestazioni e autenticazione o acquisire la risposta e visualizzare lo stato, il corpo, le intestazioni e i cookie. I test vengono scritti selezionando una delle richieste dalla scheda Debug e quindi facendo asserzioni sui dati restituiti dal server. Si possono eseguire test singoli o un'intera suite di test.

Come si può vedere di seguito, sono stati scritti test per ciascuno degli endpoint API raccolti in tre suite di test: Login, Registration e ScoreBoard.

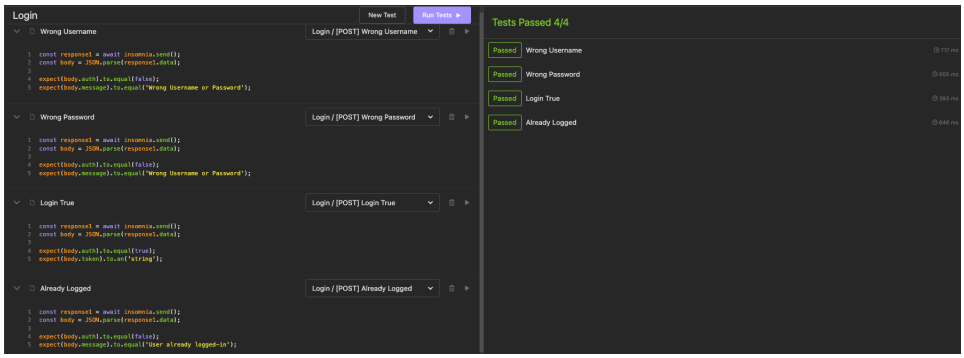

Figura 43: Test della pagina di login

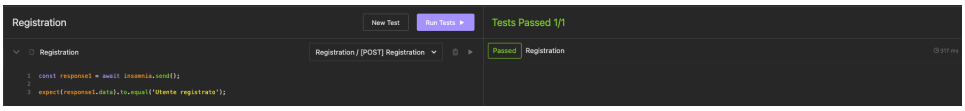

Figura 44: Test della pagina di registrazione

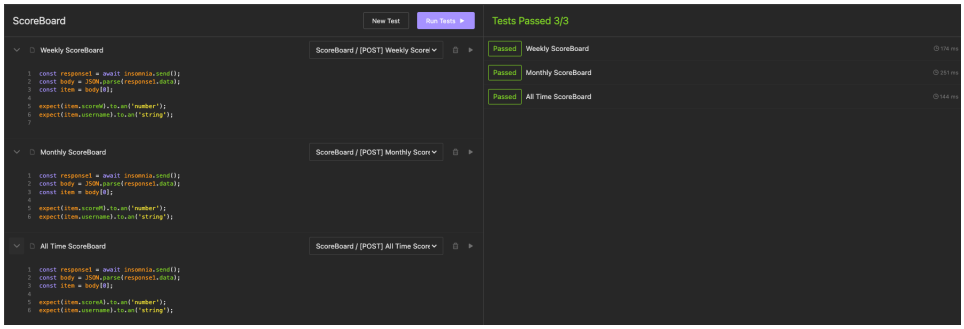

Figura 45: Test della pagina di visualizzazione della classifica

7.4 Testing della rete neurale

La parte dell'applicazione sicuramente più difficoltosa da testare è la rete neurale. Il testing di una rete neurale non può avvenire con le normali tecniche e ,naturalmente, il suo funzionamento è accompagnato da una aleatorietà dei risultati ottenuti. Può capitare che uno stesso input sia correttamente riconosciuto dalla rete che restituisce un risultato corretto, e in un secondo momento, un risultato del tutto scorretto.

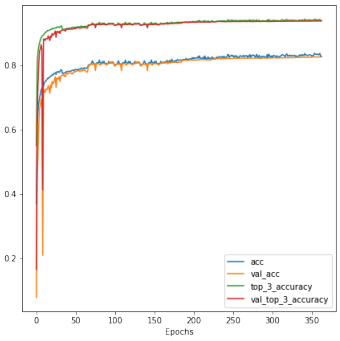

Figura 46: Percentuali di accuratezza della rete.

Si può ragionare sulla percentuale di errore al momento del training della rete, trattandola come una black box e individuando un set di immagini di test. Nella fase di implementazione è stato discusso il tasso di accuratezza (percentuale) della rete, che si assesta sul 98% per la top3 delle immagini e un 80% in media.

Un semplice test che é possibile effettuare consiste nel sottoporre alla rete un set di immagini allo scopo di individuare esempi ricorrenti di classificazioni scorrette.

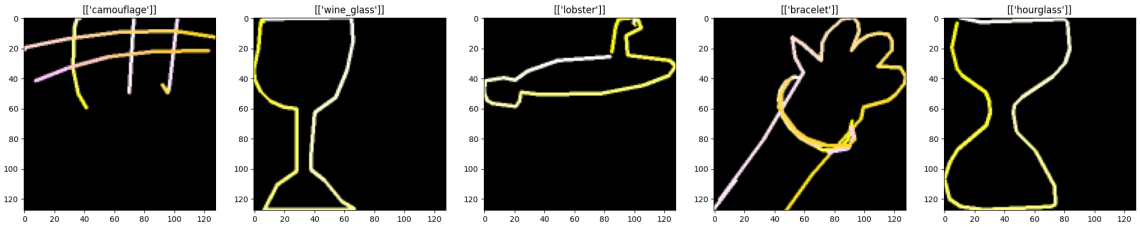

Figura 47: Testing con classificazioni corrette

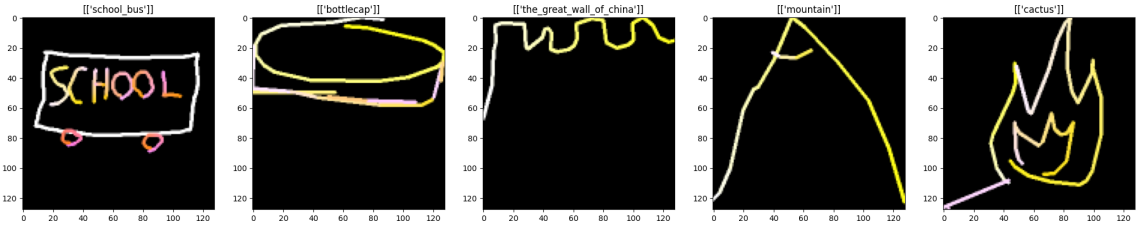

Figura 48: Testing con errore di classificazione