Gioco Multigiocatore a turni: Beccaccino

Alpi Davide
davide.alpi@studio.unibo.it

Parrinello Angelo
angelo.parrinello@studio.unibo.it

Penazzi Paolo
paolo.penazzi@studio.unibo.it

February 2023

Il progetto si pone come obiettivo lo sviluppo di una versione mobile multigiocatore del celebre gioco di carte "Beccacino". Le regole del gioco sono consultabili su WIkipedia.

"Beccaccino" è un famoso gioco romagnolo, che è stato protagonista in diversi progetti accademici passati. Per questo motivo, e per altri quali la semplice logica di gioco e la non-necessaria importanza grafica, abbiamo deciso di procedere alla realizzazione di una versione distribuita fruibile tramite dispositivo Android mobile che ci permettesse di focalizzarsi maggiormente su altri aspetti core dei Sistemi Distribuiti.

Nella versione da noi realizzata, un giocatore potrà accedere ad una partità già esistente o crearne una nuova. Ogni nuova partita avrà un codice che dovrà essere inserito manualmente dai giocatori tramite l'apposita interfaccia. Una volta raggiunto il numero massimo di giocatori (4), la partita potrà cominciare. Terminate le mani, verrà decretato il vincitore.

Contents

1 Obiettivi

1.1 Descrizione del progetto

Il progetto è una versione distribuita del gioco di carte *Beccacino*[3]. Il gioco avrà un'architettura client-server e sarà fruibile attraverso dei client mobile Android.

1.2 Learning Goals

- Prendere familiarità con la logica distribuita e gestire gli aspetti legati alle connessioni tra nodi non locali.
- Comprensione del message broker **RabbitMQ**[16] e del protocollo **AMPQ**[2].
- Gestione di database non relazionali come **MongoDB**[13].
- Controllo della fase di deployment e building con strumenti adeguati come **Docker**[5] e **Gradle**[9].
- Imparare a ragionare con dati strutturati de/serializzati attraverso **Protobuf**[10].

1.3 Deliverables

Gli artefatti da noi prodotti si articoleranno in:

- Un server che gestirà le partite di *Beccacino*.
- Un applicativo client mobile (Android) per giocare.

1.4 Scenario di utilizzo

Un nuovo giocatore quando vuole giocare a *Beccacino*, deve accedere al client mobile, inserendo il nickname alla prima apertura dell'app. In seguito, può creare una nuova partita o accedere ad una già esistente tramite l'inserimento di un codice univoco per ogni lobby di gioco.

Un giocatore che decide di creare una nuova stanza di gioco deve selezionare il pulsante "NUOVA PARTITA". Si aprirà quindi l'apposita schermata che mostra il codice univoco dell'incontro, nell'attesa che tutti i giocatori previsti (4) siano collegati al medesimo match. Una volta raggiunto il numero adeguato di persone, il creatore della stanza può avviare la partita: a questo punto verrà mostrata una nuova schermata contenente l'interfaccia di gioco (carte in mano, in tavola, carte e nomi degli altri giocatori).

Un giocatore che decide di entrare in una lobby esistente, deve selezionare il pulsante "CERCA PARTITA" e inserire il codice della stanza nell'apposito campo. Se il numero

risulta valido, il giocatore entra nella lobby, dove attenderà l'inizio della partita.

Una volta terminate tutte le mani di gioco, una schermata mostra la coppia vincitrice e il punteggio finale. I giocatori hanno la possibilità di uscire dalla partita.

2 Analisi

2.1 Requisiti Funzionali

- Creazione di una lobby.
- Unione ad una lobby esistente.
- Avvio della partita.
- Interfaccia grafica per la partita, che permetta di visualizzare le proprie carte e le carte sul tavolo.
- L'applicazione dovrà permettere agli utenti di giocare una partita di Beccacino.
- La partita dovrà terminare, permettendo ai giocatori di uscire dalla partita.
- Visualizzazione dei vincitori e del punteggio della partita.

2.2 Requisiti non Funzionali

- Il sistema dovrà gestire eventuali crash dei client partecipanti.
- Il sistema dovrà permettere scaling-out (opzionale).

2.3 Scelte architetturali

Per l'implementazione del sistema verranno sfruttati i seguenti paradigmi:

- Architettura **Client-Server**[4] per agevolare lo sviluppo e semplicità del sistema. Standard de facto per i videogiochi.
- Protocollo **AMQP** per lo scambio sicuro e affidabile dei messaggi.
- **RPC**, protocollo standard per applicazioni client-server.

2.4 Scelte tecnologiche

Le tecnologie selezionate sono:

- **Java** come linguaggio di programmazione sia lato client che lato server.
- **Gradle** per la build automation di ogni attore.
- **Docker** come tecnologia matura per la realizzazione del deployment del sistema (velocizzare quindi tale fase), effettuare una miglior gestione dei servizi (uno per container), e garantire la disponibilità di quest'ultimi.
- **Protobuf** come tecnologia per la de/serializzazione delle informazioni, trasparente lato piattaforma e linguaggio.
- **RabbitMQ** come Message Broker basato sul protocollo AMQP per la comunicazione tra client e server.
- **MongoDB** per ottenere persistenza delle informazioni, agevolare le operazioni CRUD e flessibilità nella struttura dati.

3 Tecnologie

3.1 Lato Server

3.1.1 MongoDB

MongoDB è un sistema di gestione di database NoSQL orientato ai documenti, che permette di salvare e recuperare dati in formato JSON-like.

A differenza dei database relazionali, non ha una struttura rigida di tabelle e relazioni, ma permette di creare documenti che possono contenere dati strutturati o non strutturati, conservati a loro volta all'interno di collezioni[12]. A differenza dei classici database relazionali, permette maggiore scalabilità orizzontale e flessibilità.

3.2 Middleware

3.2.1 ProtoBuf

Protocol Buffers (protobuf) è un formato di serializzazione dei dati sviluppato da Google, utilizzato per la comunicazione tra applicazioni distribuite e per la memorizzazione di dati in modo efficiente.

Consiste in un'interfaccia di definizione del linguaggio (IDL) che definisce la struttura dei dati in un formato standardizzato e leggibile dall'uomo, e un compilatore che genera codice sorgente per le applicazioni in vari linguaggi di programmazione.

Le principali caratteristiche di protobuf sono la compattezza, la scalabilità, la flessibilità e la compatibilità con numerose piattaforme e linguaggi di programmazione. Inoltre, grazie alla definizione esplicita della struttura dei dati tramite IDL, è possibile mantenere la retro-compatibilità tra le versioni dei dati serializzati.

3.2.2 RabbitMQ

RabbitMQ è un message broker open-source basato su protocollo AMQP (Advanced Message Queuing Protocol) che consente ai componenti di un'applicazione di comunicare tra di loro in maniera affidabile e scalabile. RabbitMQ funziona come un intermediario tra mittenti e destinatari, gestendo la distribuzione dei messaggi tra i nodi della rete. Ciò consente alle applicazioni di essere disaccoppiate, garantendo un'alta disponibilità, l'affidabilità e la scalabilità del sistema. RabbitMQ supporta diversi tipi di scambio di messaggi, tra cui i messaggi point-to-point e publish/subscribe. L'uso di questo broker consente di ridurre alminimo la consapevolezza reciproca che devono avere i servizi di un'applicazione per potersi scambiare messaggi, permettendo quindi, di implementare in modo efficace il disaccoppiamento. Basandosi sullo standard AMQP eredita le astrazioni di Channel, Queue ed Exchange:

- **Channel**. Un unico canale [14] costituisce una connessione logica al broker, evitando la necessità di aprire molteplici connessioni TCP più pesanti e dispendiose di risorse di sistema per la stessa applicazione. Il canale esiste solo all'interno di una connessione e quando questa viene chiusa, vengono chiusi anche tutti i canali ad essa associati.

- **Queue**. Una coda[17] è una struttura dati sequenziale con due operazioni principali: enqueued e dequeued, rispettivamente l'aggiunta di un elemento in coda e la consumazione dell'elemento in testa. Questa è l'astrazione principale che consente lo scambio di messaggio tra le entità connesse al broker, in particolare quest'ultime vengono solitamente definite Publisher se svolgono l'operazione di enqueue sulla coda e Consumer se invece eseguono quella di dequeu. RabbitMQ si rifrisce alle code attraverso un nome e ciascuna può avere diverse proprietà come ad esempio essere durevole, cioè persistere anche in caso di riavvio del broker o esclusiva, cioè utilizzabile solo attraverso una specifica connessione, la cui chiusura determina anche l'eliminazione della coda stessa

- **Exchange**. Gli exchanges [15] sono le entità alle quali vengono inviati i messaggi, preso un messaggio l'exchange lo instrada verso una o più code sulla base di regole dette Bindings, in seguito, il broker consegna i messaggi ai Consumer sottoscritti alle code oppure i Consumer stessi li recuperano su richiesta. La politica con cui i messaggi vengono instradati non dipende esclusivamente dalle Bindings ma anche dalla tipologia stessa dell'exchange:
  - *Direct exchange*: questo tipo di exchange consegna i messaggi alle code sulla base di routing key, nello specifico una coda si lega ad un exchange specificando la sua routing key, quando l'exchange riceve un messaggio, questo viene consegnato a tutte le code che hanno specificato la medesima routing key del messaggio.

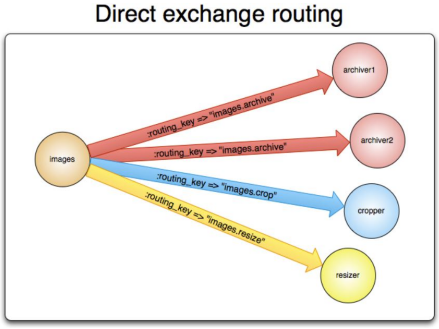

Figure 1: Rappresentazione grafica del Direct Exchange.

  - *Fanout exchange*: in questo caso l'exchange consegna i messaggi a tutte le code che sono a lui legate, ignorando eventuali routing key. Questo tipo è

quindi particolarmente adatto quando si vuole implementare un broadcast di messaggi.

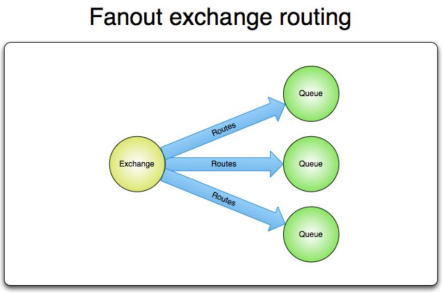

Figure 2: Rappresentazione grafica del Fanout Exchange.

- *Topic exchange*: questo tipo di exchange consegna il messaggio ricevuto a tutte quelle code che hanno specificato come routing key un'espressione regolare che ha una corrispondenza con la routing key associata al messaggio
- *Headers exchange*: in questo caso l'instradamento dei messaggi sulle code avviene sulla base degli headers dei messaggi stessi, le code quindi effettuano il binding all'exchange specificando a quali attributi dell'header sono interessate.

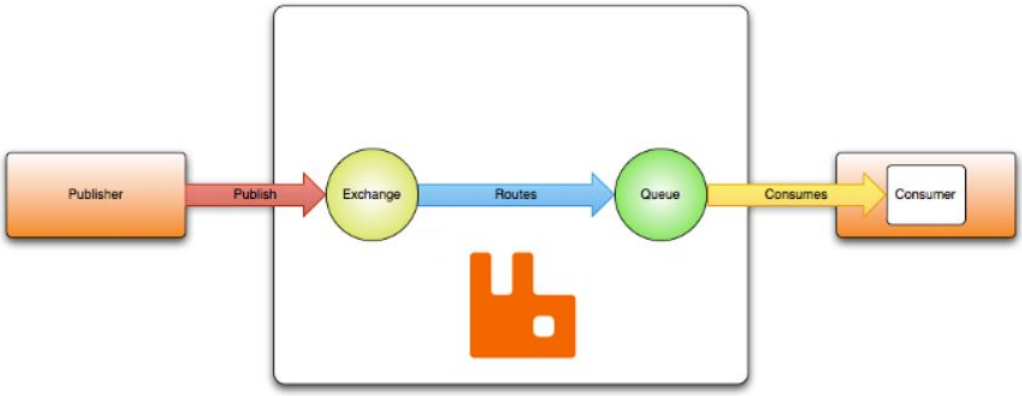

Figure 3: Rappresentazione grafica riassuntiva del routing in RabbitMQ.

3.3 Build Automation e Deployment

3.3.1 Gradle

Gradle è uno strumento di build automation open source che viene utilizzato per automatizzare il processo di compilazione, test e distribuzione di software. È stato progettato

per essere estremamente flessibile e adattabile a diverse esigenze di sviluppo, consentendo agli sviluppatori di definire e configurare il processo di build attraverso un linguaggio di scripting basato su Groovy o Kotlin. Grazie alla sua architettura basata su plugin, Gradle può essere esteso e configurato per soddisfare le esigenze specifiche di un progetto. Tra le caratteristiche principali di Gradle ci sono:

- Un sistema di caching intelligente per velocizzare i tempi di compilazione
- Il supporto per la parallelizzazione del processo di build
- La possibilità di eseguire test automatizzati durante il processo di build
- La compatibilità con diversi strumenti di integrazione continua, come Jenkins e Travis CI.

Un progetto Gradle ha una specifica struttura di cartelle predefinita. Inoltre, include uno script chiamato "gradlew" (Gradle Wrapper) [11] che serve a standardizzare la versione di Gradle utilizzata per il progetto e a garantire build più affidabili e robuste. Infine, include un file chiamato "build.gradle.kts" [8] che specifica le dipendenze e i task dell'applicazione.

3.3.2 Docker

Docker è un progetto open-source utilizzato principalmente per automatizzare il deployment di applicazioni all'interno di container. In particolare sfrutta l'isolamento delle risorse del kernel Linux per consentire a container indipendenti di coesistere sulla stessa istanza di Linux, evitando di dover creare e manutentare una macchina virtuale.

La macchina host che eseguirà l'applicazione dovrà essere dotata di un qualche Container Engine. I concetti fondamentali di Docker sono:

- Dockerfile: è un file di configurazione contenente tutti i comandi necessari per assemblare la nuova immagine [6].
- Docker Images: è un file eseguibile che rappresenta una descrizione statica di un contenitore (container) [19]. L'esecuzione dell'immagine determina l'istanziazione del container stesso. .
- Docker Container: è una macchina virtuale "leggera" che viene eseguita sul kernel della macchina host, condividendo lo stesso ambiente della macchina host e garantendo l'isolamento da altri container [19].
- Docker Compose: è uno strumento che consente la definizione di applicazioni multi-container composte da più servizi, nonché la gestione dell'intero ciclo di vita di tali applicazioni. Questo può essere fatto attraverso un file YAML, chiamato un *compose-file* [18].

4 Design

La prima decisione riguardo il design dell'applicazione è stata quella riguardante il design architetturale. Le opzioni prese in considerazione sono state peer-to-peer e client-server. L'architettura peer-to-peer applicata ai giochi multigiocatore, pur essendo tema possibile di ricerca, ha pochissime applicazioni reali per una lunga serie di motivi (sicurezza, latenza e performance, scalabilità, etc.).

In figura 4 una vista molto astratta del sistema e delle interazioni tra i componenti:

- Il Client mobile si connetterà al Server, tramite il quale potrà creare o unirsi ad una partita ed effettuare mosse.
- Il Beccacino server si occuperà di gestire le mosse effettuati dai giocatori e modificare di conseguenza lo stato delle partite.
- Il Database è la singola source-of-truth sullo stato delle partite (questo permette di avere più istanze del server per gestire più client).

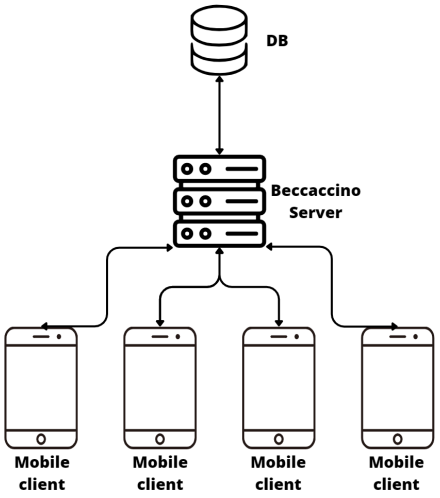

Figure 4: L'architettura ad alto livello.

4.1 Design Strutturale

Dopo aver analizzato ad alto livello la struttura del sistema, vediamone più nel dettaglio le parti principali come mostrato in figura 5.

- **Mobile Client**. Il client di gioco sarà l'interfaccia grafica con cui un utente potrà giocare. Il servizio client, che dovrà essere eseguibile sui dispositivi Android, comunicherà con il server attraverso RabbitMQ e dati standardizzati, conosciuti da ambo le parti.

- **RabbitMQ**. RabbitMQ è il middleware che ci permette di scambiare messaggi tra i vari componenti del sistema. Per farlo verrà predisposto un manager del servizio che si occuperà di creare le connessioni, i canali e le code su cui transiteranno le informazioni di gioco.
- **Server**. Il componente Stub fa da intermediario tra la logica del server e il broker di messaggi. Come spiegato precedentemente viene usato ProtoBuf per la serializzazione e deserializzazione. Queste informazioni saranno gli unici dati scambianti tra client e server. I dati riguarderanno informazioni sul giocatore, sulle carte, sulle lobby ecc. La Business Logic è la parte rimanente del server, che si occupa di gestire in maniera opportuna le richieste, e di modificare lo stato del db di conseguenza. Idealmente è possibile fare scaling-out con le istanze del server, trattandosi semplicemente di handler senza stato.
- **DB**. Il database è la memoria centrale di gioco, al suo interno verrà mantenuto lo stato di tutte le partite e di tutte le lobby.

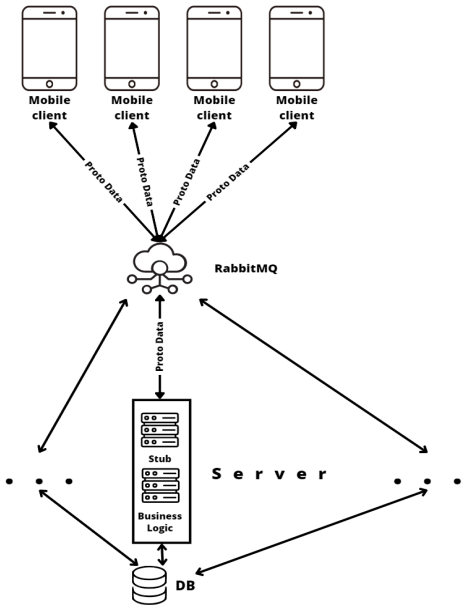

Figure 5: L'architettura del sistema.

4.1.1 Struttura Gestione Partite

Questa sezione descrive come sarà definita la gestione delle partite dalla sua esecuzione alla sua terminazione.

L'architettura legata alla gestione delle partite seguirà lo schema RPC (Remote Procedure Call) dove, in breve, il client invoca una richiesta che inoltrerà al server e quest'ultimo, sempre in ascolto, la compie per poi restituire un risultato. La gestione delle partite lato server avrà due componenti principali: il GameStub e il GameRequestHandler. Il primo ha il compito di interfacciarsi con la rete esterna e con il secondo componente, il GameRequestHandler, che gestirà nell'effettivo le richieste in arrivo. In fase realizzativa è stato poi necessario definire una classe di utility che gestisse in maniera più chiara le interazioni con il DB e tutte le operazioni che il server dovrà fare su di esso. Il tutto è spiegato tramite un diagramma delle classi nella figura 6.

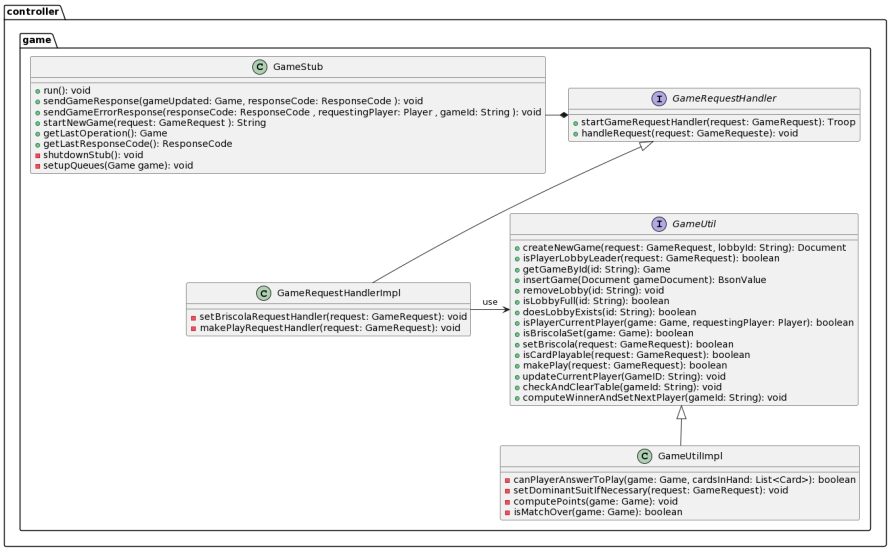

Figure 6: Diagramma delle classi riguardo la gestione delle partite.

4.1.2 Struttura Gestione Lobby

Questa sezione descrive come sarà definita la gestione delle lobbies in tutta la sua interezza.

Analogamente alla gestione delle partite, anche il design del governo delle lobbies segue un approccio RPC. Similmente, avremo due layer di decomposizione dei compiti: in un primo momento il LobbiesStub si occuperà di ricevere/inviare i messaggi di gestione della lobby e il secondo strato dato dal LobbyManager effettuerà le opportune operazioni che sono state richieste. Il diagramma delle classi sottostante 7 sintetizza ciò.

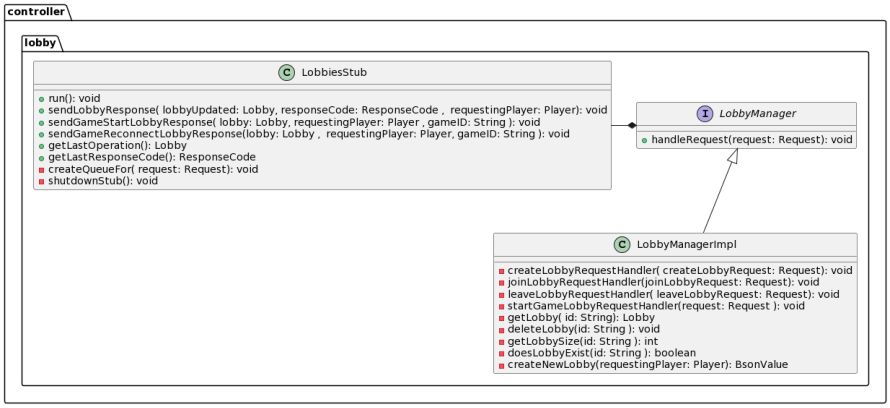

Figure 7: Diagramma delle classi riguardo la gestione delle lobby.

4.1.3 Struttura Model

Le due entità principali di gioco sono il Deck, che modella la pila di carte di una partita, e il BeccaccinoGame, che tiene conto delle informazioni di gioco di ogni giocatore, quali carte ha e il deck in gioco. Ogni giocatore, Player, è caratterizzato da un identificativo, utilizzato da MongoDB, e da un nickname. La carta di gioco, Card, è composta da un seme (Suit) e da un valore (Value). Di vitale importanza per l'avanzamento del gioco è la struttura del PrivateDate; il concetto ricalca il fatto che ogni giocatore ha una serie di carte che solo lui conosce e che potrà giocare nel corso di una mano. In figura 8 viene riportato il diagramma delle classi delle principali strutture dati di gioco.

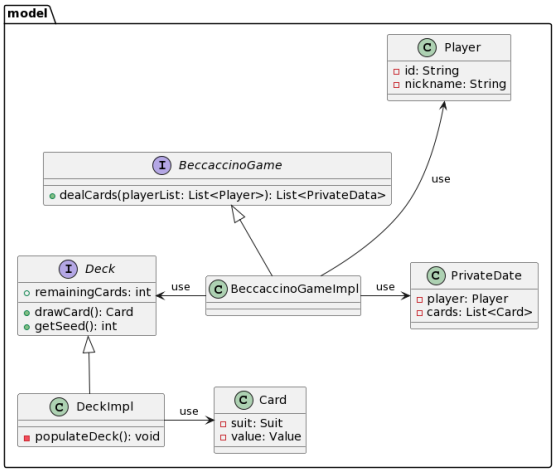

Figure 8: Diagramma delle classi incentrato sulla parte core del modello di gioco.

4.2 Design delle Interazioni

In questa sezione viene descritto come interagiscono i diversi componenti della nostra architettura durante le diverse fasi di gioco, corredando ogni scenario con il corrispondente diagramma di sequenza. In questi scenari proposti descriviamo solamente *l'happy path* ovvero i casi classici dove le sequenze vanno a buon fine ma citeremo eventuali controlli che vengono eseguiti in ciascuna operazione.

4.2.1 Creazione di una lobby

Come mostrato in figura 9, l'utente intenzionato a cominciare una partita lo fa attraverso l'app client Android. Quando viene creata una lobby, il player che ha richiesto la creazione è automaticamente al suo interno. La richiesta può essere fatta solamente dal menù iniziale. Il client invia una richiesta di creazione lobby (`createLobbyRequest`). Il messaggio di creazione della lobby è così ricevuto dal LobbiesStub che compie l'unmarshalling della richiesta e lo sottopone al LobbyManager. Quest'ultimo gestisce la creazione della lobby, creando un `roomID` univoco per la partita, e rispedisce il messaggio al LobbiesStub che si occupa poi di inviare il pacchetto dati (`LobbyResponse`) al client richiedente . A questo punto il client deve aspettare l'arrivo degli altri giocatori (mandandogli eventualmente tramite altri canali di comunicazione l'id della lobby visualizzato a schermo).

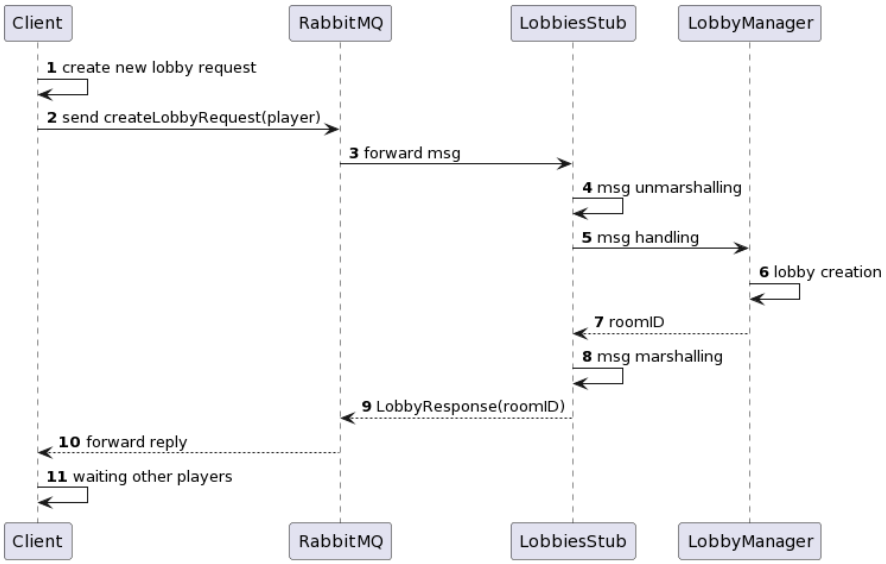

Figure 9: Diagramma di sequenza per la creazione di una partita.

4.2.2 Unione di un giocatore ad una lobby

Per unirsi ad una partita, un giocatore deve inserire l'id univoco (`roomID`) della partita già creata nella quale si vuole entrare. Questa azione, come la precedente, può essere fatta nel menù principale. La sequenza comincia quando l'utente, attraverso l'app client, inserisce l'id della partita. Il client costruisce così la richiesta di unione alla lobby (`searchLobbyRequest`). La richiesta viene inoltrata da RabbitMQ al LobbiesStub che si occupa di ridirezionare il messaggio al LobbyManager. Nel caso classico in cui la lobby esiste, il LobbyManager, confrontandosi sempre con il DB, restituisce al LobbiesStub la Lobby aggiornata. Esistono poi due alternative, che son state gestite ma non verranno trattate nella spiegazione, riguardo alle due casistiche principali di problemi legati all'unione ad una lobby, quali lobby piena o lobby inesistente (in ambo i casi il client viene informato dell'errore). Il LobbiesStub ancora una volta si occupa di impacchettare e inoltrare la richiesta (`LobbyResponse`) al client, depositando il messaggio nell'apposita coda di RabbitMQ. Il client riceve il messaggio e in quel momento viene portato nella schermata di pre-partita in attesa dell'inizio del gioco. In figura 10 viene riportato il diagramma di sequenza dell'operazione appena descritta.

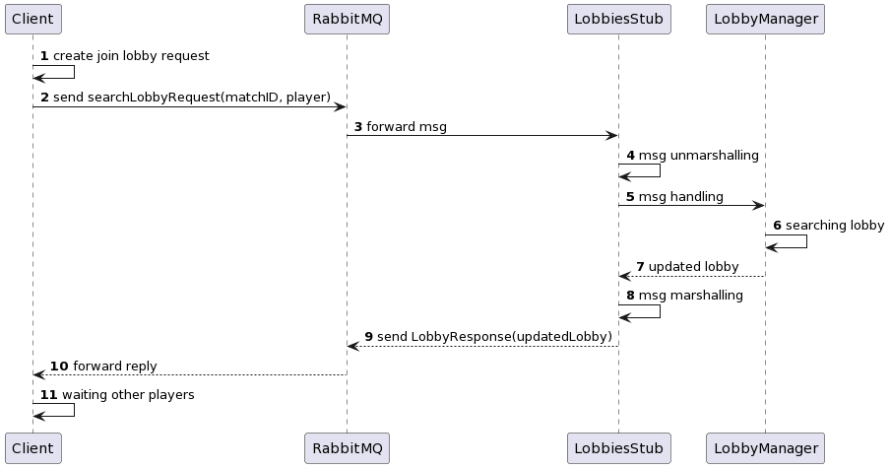

Figure 10: Diagramma di sequenza per l'unione di un giocatore ad una partita.

4.2.3 Inizio di una partita

Non appena la lobby raggiunge il numero adeguato di giocatori (4), l'host della partita, colui che ha creato la lobby, è in grado di cominciare il match. Il giocatore preme il bottone di avvio della partita ed istantaneamente viene creata una richiesta di inizio partita (`startGameRequest`). Come al solito, il client inoltra la richiesta a RabbitMQ il quale si interfaccia con il server, in particolare con il LobbiesStub. Ricevuto il pacchetto, il LobbiesStub lo elabora e lo fornisce al LobbyManager. Il LobbyManager si interfaccia con il GameStub per collaborare affinchè la partita cominci. A tale scopo, il LobbyManager chiede al GameStub di creare una partita per una determinata Lobby. Ricevuta la richiesta, il GameStub domanda a sua volta al GameRequestHanlder se è possibile cominciare la partita. Il GameRequestHandler esegue tutti i controlli del caso (esistenza della lobby, controllo della pienezza della lobby e controllo che il giocatore richiedente sia l'host). In seguito, il game viene creato, e il suo analogo documento salvato sul DB, e i dati della partita spediti al GameStub. Il GameStub crea i canali di comunicazione con i 4 client (code di RabbitMQ) e notifica direttamente i client su tali connessioni con i dati della partita (`gameResponse`): per tutta la durata della partita i client comunicheranno con il GameStub tramite questi canali. Contemporaneamente, il GameStub fornisce il response-code (`responseCode`, un intero che indica l'esito dell'operazione) al LobbyManager. Il LobbyManager si incaricherà di controllare il codice dell'operazione e richiederà al LobbiesStub di inoltrare la risposta al client. Ciascun client, una volta ricevuto il messaggio di inizio partita, controlla il codice dell'operazione e allaccia a sua volta la connessione con il GameStub. Una volta costruite le connesioni da ambo le parti, i client prelevano i dati, che il GameStub aveva precedentemente inviato, dalla coda che collega al GameStub; tali dati servono a cambiare schermata di gioco dove comincierà la partita. La figura 11 rappresenta ciò che è appena stato descritto a parole.

4.2.4 Gestione della partita

Il client intenzionato a giocare una carta deve cliccare su di essa nell'app mobile. Una volta fatto ciò il client prepara una richiesta di giocata (`playRequest`) e spedisce i dati della carta al server. Come al solito, RabbitMQ deposita il messaggio nella coda privata tra il client ed il server. Il messaggio, ricevuto dal GameStub, viene gestito dal GameRequestHandler; quest'ultimo verifica tutte le condizioni del caso (es. la carta che si vuole giocare è della stessa briscola scelta? è il turno del giocatore?), aggiorna lo stato della partita e chiede al GameStub di comunicare l'informazione a tutti i client in gioco (`gameResponse`). Una volta ricevuti i dati, le schermate di ciascun giocatore cambiano a seconda della nuova impostazione della partita. La sequenza di selezione della briscola ad inizio partita non verrà spiegata in quanto, salvo alcuni controlli ulteriori che devono essere fatti, è simile alla fase appena descritta per le giocate. La figura 12 illustra ciò che avviene tra client e server durante la una giocata.

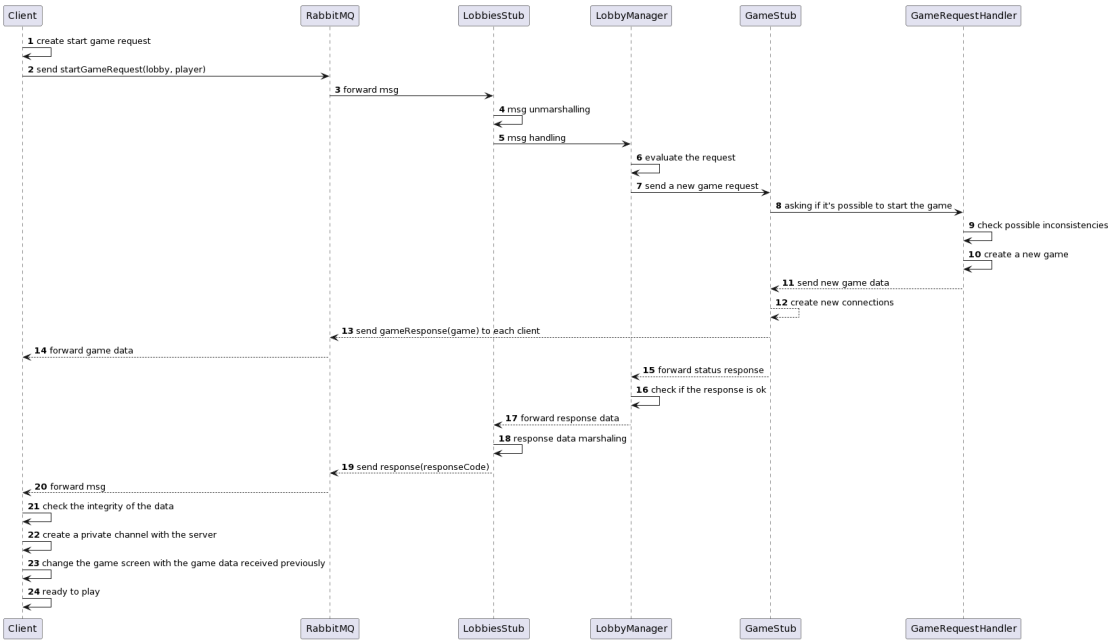

Figure 11: Diagramma di sequenza per l'inizio di una partita.

4.2.5 Gestione delle disconnessioni e riconnesioni degli utenti

Se un giocatore dovesse disconnettersi per qualunque motivo da una partita in corso, potrà riconnettersi alla lobby e verrà automaticamente reintrodotto alla schermata di gioco, recuperando al volo tutti i dati sullo stato della partita.

Un altro problema possibile è invece quello di un giocatore che va afk ("away from keyboard"), ed essendo il beccacino un gioco a turni questo evento bloccherebbe gli altri giocatori per un tempo indefinito. Per ovviare a questo problema implementiamo nel server un AfkGuard, che si occupa di verificare periodicamente il timestamp relativo all'ultima mossa effettuata in ciascuna partita in corso. Nel caso trovi una partita aggiornata per l'ultima volta più di un minuto fa, essa viene terminata e i giocatori riportati al menù principale.

4.3 Design Comportamentale

Nella seguente sezione verranno analizzati i comportamenti del LobbyManager e del GameRequestHandler, due componenti del sistema lato server che implementano la logica di gestione delle Lobby e dei Game.

4.3.1 Comportamento del LobbyManager

All'avvio del server viene istanziato un LobbiesStub il quale, attraverso il LobbyManager, potrà gestire i messaggi relativi alla manutenzione delle varie lobby. Vediamo assieme

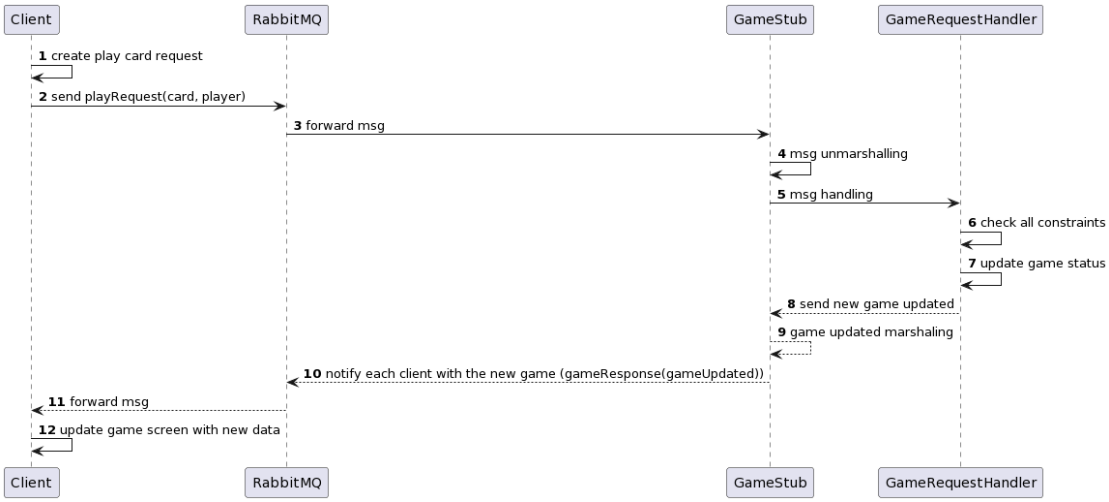

Figure 12: Diagramma di sequenza che descrive come una carta può essere effettivamente giocata.

il comportamento del LobbyManager in risposta ai vari messaggi che può ricevere. Si assume che i messaggi vengono prima deserializzati e poi serializzati dal LobbiesStub, che in un secondo momento spedirà al client. Si assuma inoltre che inizialmente lo stato della gestione delle lobby sia in WAITING, ovvero in attesa di messaggi in entrata. La figura seguente 13 riassume gli stati che può assumere il LobbyManager. Ora analizzeremo nel dettaglio ognuno di questi stati.

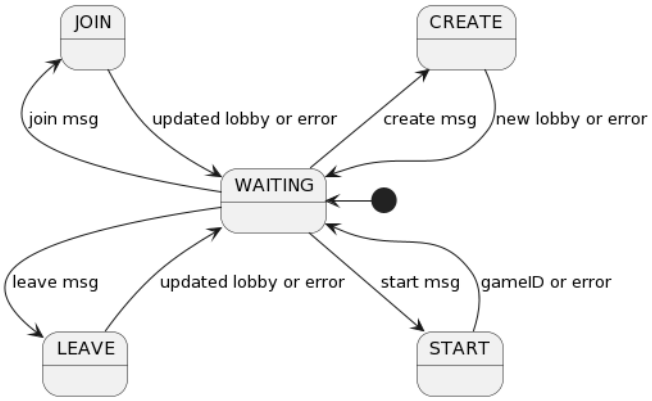

Figure 13: Diagramma degli stati del LobbyManager.

- **Richiesta di creazione Lobby**.
  Alla richiesta di creazione della Lobby, il LobbyManager controlla innanzitutto se il Player è già in partita. In caso di esito positivo, viene creata una nuova Lobby nel database e i dati della lobby vengono inoltrati al LobbiesStub. In caso di esito negativo il LobbiesStub viene notificato con un `CREATE_ERROR`. Rappresentazione grafica in figura 14.

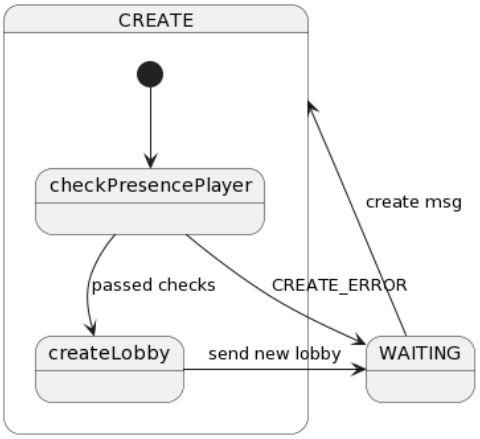

Figure 14: Analisi dello stato CREATE del LobbyManager.

- **Richiesta di unione ad una Lobby.**
  Alla richiesta di entrare in una stanza pre-partita (lobby), il LobbyManager controlla subito se il giocatore è già in un'altra lobby. In caso di errore viene notificato il LobbiesStub con un `JOIN_ERROR`. In caso di esito positivo, si controlla se la Lobby esiste e se non ha raggiunto già il numero massimo di giocatori al suo interno. Se tutte le condizioni citate fino ad ora sono state soddisfatte, il LobbyManager aggiorna la lobby precedentemente creata con le nuove informazioni sul player e i dati della lobby aggiornata vengono inviati al LobbiesStub, altrimenti il LobbiesStub riceverà un `JOIN_ERROR`. Per una rappresentazione grafica del comportamento appena descritto, si veda la figura 15.

- **Richiesta di lasciare una Lobby.**
  Alla richiesta di uscire da una lobby da parte di un giocatore, salvo errori, il LobbyManager invierà allo stub la risposta positiva con la lobby aggiornata. Un ulteriore controllo che questo deve poi fare è sulla cardinalità della stanza: in caso non ci fossero più giocatori nella lobby pre-partita, questa deve essere cancellata dal database. Si faccia fede alla figura 16 che rappresenta gli stati interni dello stato LEAVE del LobbyManager.

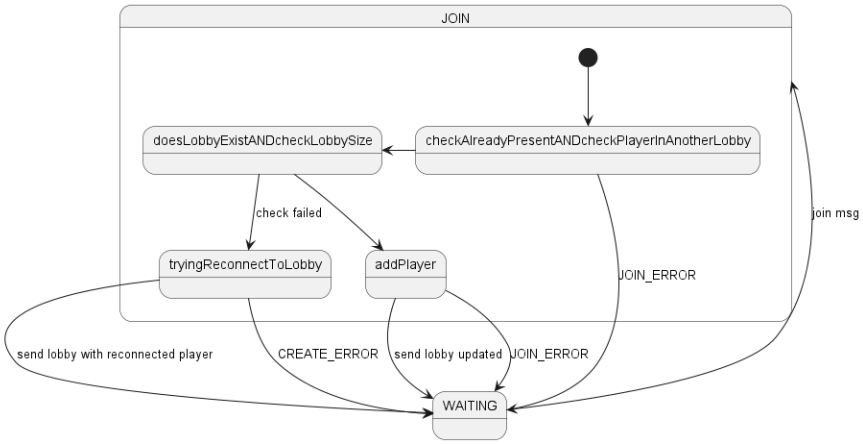

Figure 15: Analisi dello stato JOIN del LobbyManager.

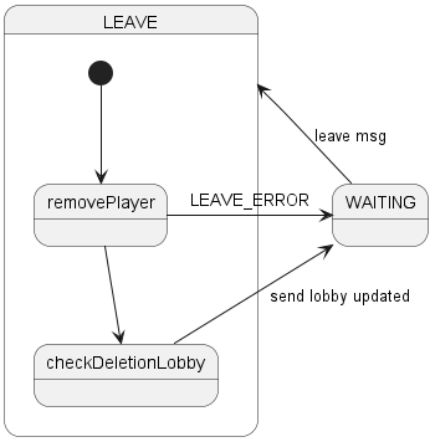

Figure 16: Analisi dello stato LEAVE del LobbyManager.

- **Richiesta di inizio partita.**
  Alla richiesta di cominciare una nuova partita, innanzitutto il LobbyManager si confronta con il GameStub per assicurarsi che questo è fattibile, essendo il GameStub responsabile della gestione delle partite. Il LobbyManager, a seguito di questa richiesta, riceverà un codice sullo status della richiesta; a seconda dello status reagirà diversamente. Un esempio di status, oltre al caso di esito positivo ovviamente, potrebbe essere il caso del permesso negato, nell'eventualità che il richiedente non è il lobby-leader ossia chi ha creato la lobby. Nella circostanza che la partita possa effettivamente cominciare, il LobbiesStub verrà avvertito con i dati necessari tra cui l'id della partita appena cominciata. Il diagramma degli stati in figura 17, esamina graficamente quanto detto sopra.

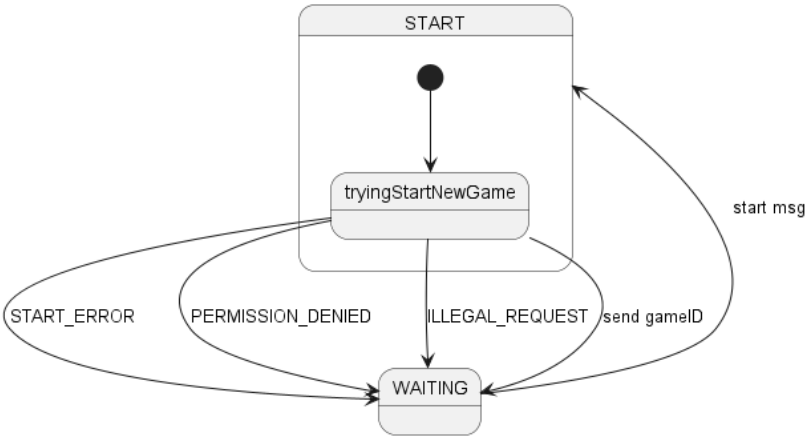

Figure 17: Analisi dello stato START del LobbyManager.

4.3.2 Comportamento del GameRequestHandler

All'avvio del server viene istanziato un GameStub il quale, attraverso il GameRequestHandler, potrà gestire i messaggi relativi alla manutenzione delle varie partite. Vediamo assieme il comportamento del GameRequestHandler in risposta ai vari messaggi che può ricevere. Si assume che i messaggi vengono prima deserializzati e poi serializzati dal GameStub. In figura 18 si può trovare una rappresentazione ad alto livello degli stati del GameRequestHandler.

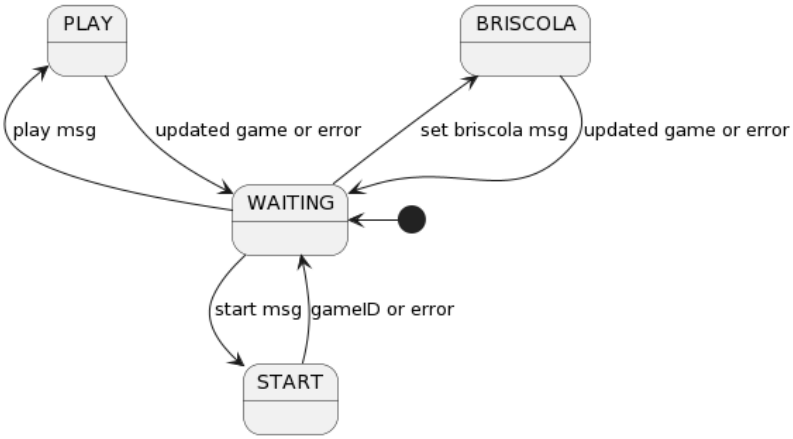

Figure 18: Diagramma degli stati del GameRequestHandler.

- **Richiesta di start di una partita.**
  La seguente richiesta viene gestita assieme al LobbyManager; il GameRequestHandler si occupa dell'effettiva realizzazione e autorizzazione della partita. Ricevuta la richiesta di cominciare una partita si avvieranno una serie di controlli:

– Controllo sull'esistenza. Si controlla che la stanza pre-partita esista, in caso contrario creerà un `illegal` status.
– Controllo sulla cardinalità della lobby. A seguire, si controlla che il numero dei giocatori nella lobby sia esattamente 4, altrimenti si genera un `error` status.
– Controllo sul player richiedente. Il successivo controllo si focalizza su chi produce la richiesta di start: se è il leader della lobby si prosegue, diversamente si segnala un `permission-denied` status.

Superati i seguenti controlli, l'istanza di gioco verrà creata e verrà inoltrato al GameStub l'id del game creato (in caso di qualunque errore in questa fase, verrà prodotto un generico `error` status). Nel diagramma 19 è rappresentato ciò spiegato poc'anzi.

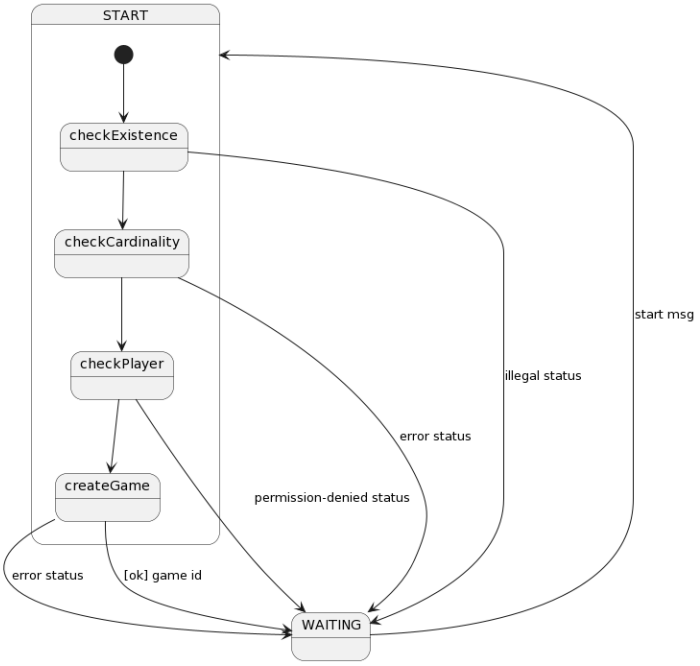

Figure 19: Analisi degli stati dello stato START del GameRequestHandler.

- **Richiesta di fissare una briscola.**
  Ricevuta la richiesta di definire la briscola di gioco, il sistema cercherà, prima di capire se chi manda il messaggio è effettivamente autorizzato (altrimenti l'operazioni si blocca e si recapita al GameStub un `PERMISSION_DENIED`) e poi se la briscola non è stata già settata (anche qui, in caso di errore il GameStub riceverà un messaggio di `ILLEGAL_REQUEST`). Se tutto prosegue come dovuto, la briscola della partita viene aggiornata e invieremo al GameStub la partita con i dati aggiornati; altrimenti un generico messaggio di `FAIL` arriverà al GameStub.

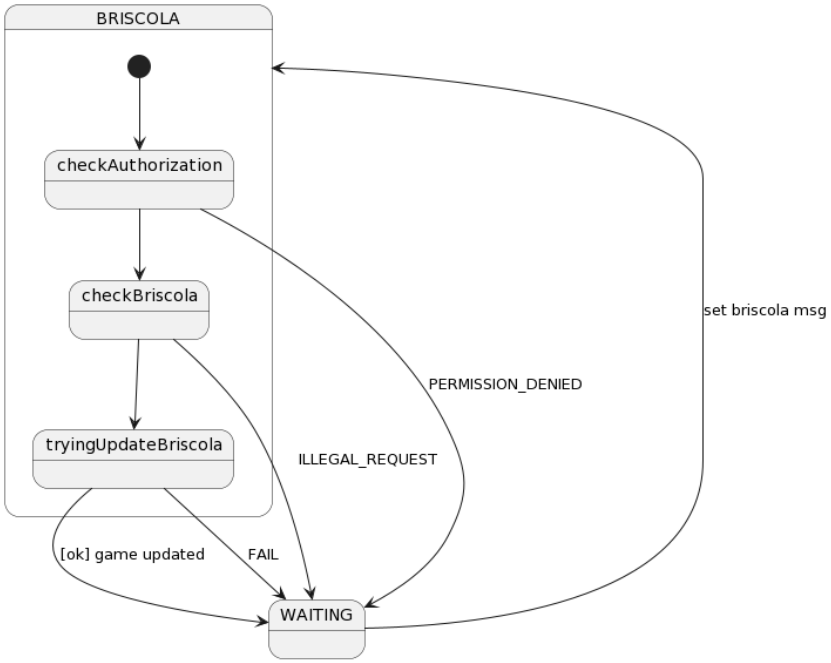

Figure 20: Analisi degli stati dello stato BRISCOLA del GameRequestHandler.

- **Richiesta di giocare una carta.**
  Ricevuta la richiesta di giocare una specifica carta da uno specifico giocatore, si attiva un'altra serie di controlli:
  - Controllo sul player corrente. Si controlla che il giocatore che chiede di giocare una carta ne abbia effettivamente il diritto, altrimenti viene generato un messaggio di `PERMISSION_DENIED`.
  - Controllo sulla briscola. Ci si accerta che la briscola sia già stata definita precedentemente, altrimenti si crea un messaggio di `ILLEGAL_REQUEST`.
  - Controllo sulla giocabilità della carta. Se le verifiche precedenti si concludo correttamente, bisogna verificare che la carta sia giocabile (es. la carta è dello stesso seme della briscola). In caso contrario, viene prodotto un messaggio di tipo `ILLEGAL_REQUEST`.

  A seguito di questi controlli, si aggiornano i dati della partita con la nuova carta, viene notificato il GameStub e si controlla se la partita è finita: se in questa fase venisse prodotto un errore, un messagio di `FAIL`, verrebbe generato. Infine, in figura 21 viene spiegato graficamente il tutto.

‘

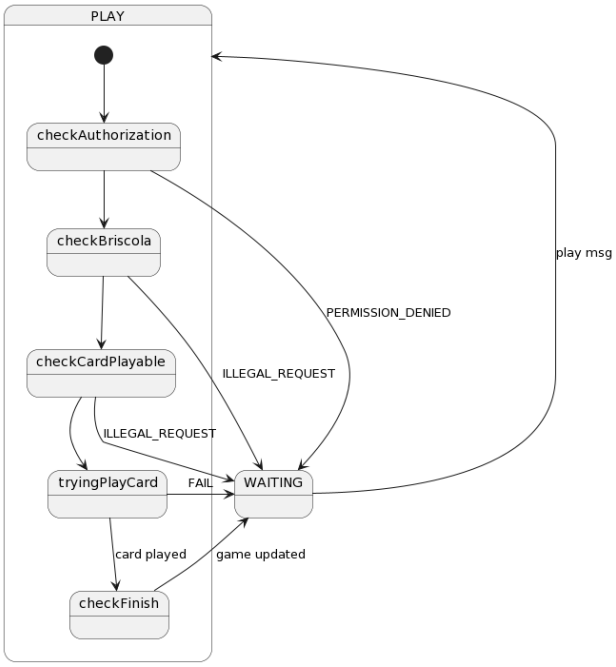

Figure 21: Analisi degli stati dello stato PLAY del GameRequestHandler.

5 Containerizzazione

La parte di containerizzazione è stata affidata a Docker. Le immagini di ogni parte del sistema sono:

- MongoDB e RabbitMQ hanno le loro immagini ufficiali, facilmente reperibili da Docker Hub[7].
- l'immagine del Server, realizzata tramite un suo `Dockerfile`, si basa su l'ultima versione di un'immagine Alpine Linux[1], sul quale verranno poi copiati i sorgenti e tutte le operazioni necessarie al suo funzionamento. È qui riportato 1:

```
# Alpine Linux , latest version
FROM alpine:latest
# Install OpenJDK 11
RUN apk update ; apk add openjdk16
# Create environment variable for container working dir path
ENV WORKINGDIR=/home/gradle/src
# Change working dir
WORKDIR ${WORKINGDIR}
# Create dir (if dosen't exist) and concede full rights
RUN mkdir -p ${WORKINGDIR} && chmod 666 ${WORKINGDIR}
# Copy the Gradle config, source code, and static analysis config
# into the build container. Without --chown gradle throws a permission error
COPY --chown=gradle:gradle . ${WORKINGDIR}
# Best solution we've found so far is to install glibc into our alpine image
# (protobuf-plugin-gradle, that contains protoc, artifact is compiled for glibc).
RUN apk update
RUN wget -q -O /etc/apk/keys/sgerrand.rsa.pub \
    https://alpine-pkgs.sgerrand.com/sgerrand.rsa.pub
RUN apk add gcompat
# Concede rights at the script
RUN chmod +x /home/gradle/src/script.sh
RUN chmod +x /home/gradle/src/gradlew
# Run the script that run server
CMD ./script.sh
```

Listing 1: File Dockerfile

Per configurare l'orchestrazione tra questi servizi Docker, è stato realizzato un `compose-file`, riportato di seguito 2

```yaml
version: '3'

services:

  mongo:
    container_name: mongo
    image: mongo
    restart: always
    ports:
      - "27017:27017"

  rabbitmq:
    container_name: rabbitmq
    image: rabbitmq:3-management
    restart: always
    environment:
      RABBITMQ_DEFAULT_USER: user
      RABBITMQ_DEFAULT_PASS: user
    ports:
      # address host:address container
      - "5672:5672"   # the port RabbitMQ will listen on.
      - "8081:15672"  # the port the dashboard web service will be available from

  server:
    container_name: server
    build:
      context: ./
      dockerfile: Dockerfile
    depends_on:
      - rabbitmq
      - mongo
    environment:
      - RABBIT_HOST=rabbitmq
      - MONGODB=mongodb://mongo
    volumes:
      - "/logs:/opt/WebSphere/HTTPServer/logs"
    restart: always
```

Listing 2: File docker-compose.yml

5.1 Dipendenza temporali

Per integrare i vari servizi, è stato necessario introdurre delle dipendenze temporali. Questo comportamento è dovuto alla composizione del server: se i servizi di RabbitMQ e MongoDB non sono attivi, è impossibile che il server interagisca con l'esterno. L'istruzione `depends_on` nel file `docker-compose` mostrato poco fa 2, risolve questa problematica.

5.2 Persistenza

Per aumentare la resilienza in caso di errore, è stato aggiunto l'attributo `restart: always` per ogni servizio nel file 2. Questo ordina a Docker di ricostruire il container in caso di malfunzionamento e di riavviare il motore Docker all'avvio del sistema. Ad esempio, se il server fallisce durante una partita, lo stato di gioco viene salvato nel database e Docker ricostruisce il container che si ricollega al database, garantendo un'esperienza utente senza interruzioni.

6 Validazione

Lo sviluppo del progetto ha cercato di seguire per quanto possibile la filosofia del TDD (Test-Driven-Development). Sono stati realizzati diversi unit test per verificare il comportamento dei vari componenti del sistema. Il sistema nel suo complesso è stato testato attraverso l'attivazione di 4 emulatori Android.

6.1 Unit Testing

6.1.1 Testing Controller-Gamer

Al fine di testare quanti più casi possibili, sono stati definiti tre file di test legati alla gestione delle partite:

- **MakePlayRequestTest**. Questo file controlla che il server risponda correttamente alle richieste di giocare una carta da parte di un utente e lo fa attraverso 6 test:
  - *testMakePlay*. Testa che il sistema registri correttamente una giocata.
  - *testMakePlayWithMessage*. Testa se il messaggio mandato quando viene fatta la giocata è correttamente registrato.
  - *testWrongPlayerMakePlay*. Testa il comportamento del sistema se la giocata è fatta da un player non autorizzato. Questo non dovrebbe accadere in una situazione normale, in quanto il client non dovrebbe permettere tale azione.
  - *testPlayNotInHandCard*. Testa il comportamento del sistema se la giocata fatta contiene una carta che non è nella mano del player autorizzato. Questo non dovrebbe accadere in una situazione normale, in quanto il client non dovrebbe permettere tale azione.
  - *testDominantSuitSetup*. Testa che il sistema salvi correttamente la briscola dominante per quel turno. La briscola dominante è il seme della prima carta giocata.
  - *testDominantSuitNotRespected*. Testa il comportamento del sistema se la giocata comprende una carta non autorizzata (non è dello stesso seme della briscola). Questo non dovrebbe accadere in una situazione normale, in quanto il client non dovrebbe permettere tale azione.

‘

Figure 22: Risultati della classe di test MakePlayRequestTest.

- **SetBriscolaRequestTest**. Questo file controlla che il server risponda correttamente alle richieste di settare una briscola all'inizio della partita da parte di un utente e lo fa attraverso 3 test:
  - *testSetBriscola*. Controlla che il sistema collezioni correttamente la briscola che un utente vuole settare.
  - *testWrongPlayerSetBriscola*. Controlla che un giocatore che non è il leader della partita, non riesca a definire la briscola di gioco.
  - *testSetBriscolaAlreadyDone*. Controlla che, se la briscola è già stata settata, non possa essere ri-definita successivamente.

‘

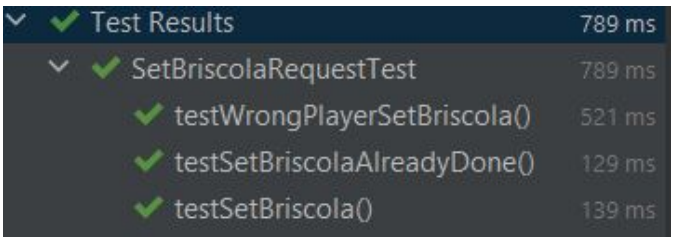

Figure 23: Risultati della classe di test SetBriscolaRequestTest.

- **StartGameRequestTest**. Questo file controlla che il server cominci correttamente una partita a seguito di una tale richiesta e lo fa attraverso 4 test:
  - *testStartGame*. Si assicura che una partita cominci correttamente.
  - *testGameAlreadyStarted*. Si assicura che se una partita è già cominciata, non possa ricominciare. Si intede una partita univocamente definita assieme ad una lobby.
  - *testStartGameWithLobbyNotFull*. Si assicura che se una lobby non è al completo (ovvero 4 giocatori al suo interno), la partita non possa cominciare.

– *testStartGameWithoutPermissions*. Si assicura che una partita non possa essere cominciata se non dal lobby leader.

‘

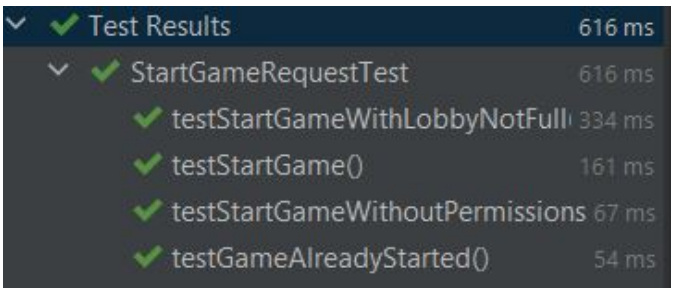

Figure 24: Risultati della classe di test StartGameRequestTest.

6.1.2 Testing Controller-Lobby

La gestione delle lobby è stata verificata con il supporto del file di testing **LobbyManagerTest**. che contiene:

- *testCreateLobby*. Controlla che la lobby venga correttamente creata.
- *testJoinLobby*. Controlla che, se un giocatore cerca di entrare in una lobby, il sistema lo permette.
- *testJoinLobbyFail*. Controlla che, se un quinto giocatore cerca di entrare in una lobby, il sistema non lo permette.
- *testLeaveLobby*. Controlla che, se un giocatore cerca di uscire da una lobby, il sistema lo permette.
- *testLastPlayerLeavesLobby*. Controlla che, se l'ultimo giocatore cerca di uscire dalla lobby, il sistema lo permette e in seguito cancella la partita.

‘

Figure 25: Risultati della classe di test LobbyManagerTest.

6.1.3 Testing Model

La sezione del model è stata meno testata perchè diverse strutture dati lato model, sono generate in maniera automatica da Protobuf. Assumenendo ciò, è stato testato il Deck di gioco (**DeckTest**), in particolare:

- *testRemainingCards*. Si assicura che, la funzione che ritorna il numero di carte rimaste nel deck, operi correttamente.
- *testGetSeed*. Si assicura che, la funzione che ritorna il seme di gioco attuale, operi correttamente.
- *testDrawCard*. Si assicura che, la funzione che fa in modo di pescare la carta in cima al deck, operi correttamente.

‘

Figure 26: Risultati della classe di test DeckTest.

6.1.4 Testing RabbitMQManager

È stato realizzato un Manager in grado di semplificare la gestione delle connessioni con RabbitMQ. La realizzazione di esso ha visto concretizzarsi test relativi alla creazione della connessione (*testCreateConnection*), test sull'istanziazione di code per ricevere (*testCreateQueueForReceive*) e mandare (*testCreateQueueForSend*) dati.

‘

Figure 27: Risultati della classe di test RabbitMQManagerTest.

6.1.5 Testing DBManager

Uno dei componenti più critici del sistema è il DB e per questo uno dei più testati. Prima di ogni test relativo al DB, il DBManager (colui che si occupa di interagire effettivamente con il DB) viene inizializzato e a seguito di ogni test, le due collezioni principali usate dal DB (`players` e `lobbies`) svuotate. Appreso ciò, sono stati fatti i seguenti test:

- *testCreateDBManager*. Test sulla creazione del database.
- *testInsertDocuments*. Test sull'inserimento di un documento nel DB.
- *testRetrieveAllDocuments*. Test sul recupero di tutti i documenti di una data collezione.
- *testRetrieveDocument*. Test sul recupero di un singolo documento di una data collezione. *testUpdateDocument*. Test sull'aggiornamento di un documento per una data collezione.

‘

Figure 28: Risultati della classe di test DBManagerTest.

6.2 Testing funzionale

Il testing funzionale del sistema viene svolto in locale, avviando il server con:

```
1 docker compose up
```

Successivamente, vengono avviati 4 emulatori android da Android Studio (o da Intellij, opportunamente equipaggiato con i vari plugin per lo sviluppo Android) e viene runnata

su ciascuno di essi l'applicazione client. Vengono verificati i seguenti comportamenti attesi:

- L'utente riesce a creare una lobby con successo.
- Possono essere create più lobby da utenti diversi.
- Gli utenti possono unirsi alle lobby e uscire da esse.
- La partita viene avviata correttamente e i giocatori vengono portati alla schermata di gioco.
- In ogni momento della partita, le informazioni a schermo sono consistenti e corrette per tutti e 4 i giocatori.
- Il calcolo di fine partita viene eseguito e mostrato correttamente.
- I messaggi di errore vengono visualizzati correttamente in tutti gli scenari di errore
- La riconnessione di un utente viene gestita correttamente.
- Una partita "stuck" da più di 60 secondi viene eliminata e i giocatori riportati al menù principale.

7 Deployment

In questo progetto si utilizza Docker per creare un'istanza del server Beccacino, che a sua volta sarà composto da più servizi (MongoDB, RabbitMQ). Il punto di partenza per la creazione di un container è il file 'docker-compose.yaml', che si occupa dell'inizializzazione dei vari servizi. MongoDB e RabbitMQ vengono inizializzati creando un'istanza dalla loro immagine Docker. Per quanto riguarda la dichiarazione del servizio server, ovvero la nostra applicazione, specifichiamo il Dockerfile che verrà utilizzato per creare un'istanza del server. Il Dockerfile creerà un container che ha come base l'immagine Alpine (una distribuzione Linux leggera). Eseguirà poi alcune operazioni di configurazione: copia dei file di progetto dentro al container, aggiornamento del jdk, setting dei permessi dei file all'interno della cartella di lavoro. Come ultima operazione verrà eseguito uno script che avvierà il server utilizzando Gradle.

Il client verrà costruito tramite Android Studio e verrà fatto generare automaticamente da Gradle un .apk in fase di build: qualora qualcuno fosse intenzionato a giocare a questo gioco di carte distribuito, dovrà installare il file .apk nel proprio dispositivo Android.

7.1 Guida al deployment

Tramite Docker, è possibile realizzare il deploy del sistema tramite il comando:

```
1 docker compose up
```

Listing 3: Comando per effettuare il deploy del sistema

A seguito di tale comando, Docker andrà a creare le immagini di ogni servizio. Infine, basandosi su ogni immagine, crea un container e segue i comandi contenuti nel docker-compose file.
Il sistema host che intende eseguire il sistema, dovrà aver installato Docker come pre-requisito.

Esiste poi la possibilità di generare in maniera agevolata il client sfruttando uno script contenuto nella cartella `client` del progetto. Al suo interno esiste uno script (`build-client.sh`) che genera il file .apk con cui lanciare il servizio client su un dispositivo Android. Affinchè il tutto funzioni correttamente, bisognerà recarsi all'interno della cartella `client` ed in seguito lanciare il seguente comando:

```
./build-client.sh
```

Listing 4: Comando per effettuare il build del file .apk del app client

8 Esempio d'uso

Proponiamo ora una serie di schermata che mostrano le varie fasi del gioco.

In figura 29 viene mostrata la schermata iniziale di gioco. Una volta inserito il nickname di gioco (che verrà chiesto solo la prima volta che il player apre l'app), l'applicazione porterà l'utente nella schermata iniziale dove potrà creare una partita o accedere ad una già esistente.

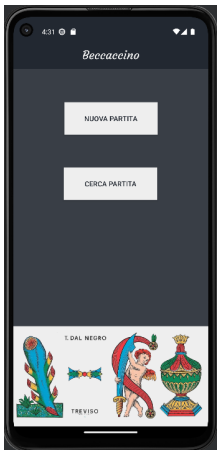

Figure 29: Schermata di gioco iniziale.

Una volta entrati in una lobby pre-partita, si potrà assistere ad una schermata simile a quella mostrata in figura 30. Qui è possibile vedere i nickname dei partecipanti e, nel caso in cui si è lobby leader, cominciare una nuova partita tramite l'apposito bottone.

Figure 30: Schermata della lobby pre-partita.

Comincia la partita. Scelta la briscola, il primo giocatore potrà effettuare delle chiamate (es. "striscio") per mandare "informazioni" relativamente a cosa giocare, al suo compagno di gioco. Questo, è raffigurato nella figura 31 seguente.

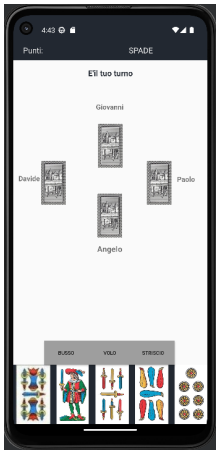

Figure 31: Schermata di inizio partita, con il primo giocatore che può effettuare delle "chiamate di gioco".

Ogni giocatore potrà giocare delle carte, scegliendole dalla propria mano. Le carte in mano si possono visualizzare scorrendole, come visualizzato in figura 32. Nella schermata 32, si mostra come è possibile visualizzare anche le carte giocate dagli altri utenti in quella mano.

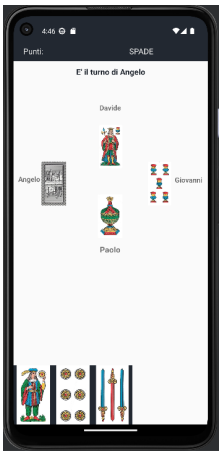

Figure 32: Schermata di metà partita.

Infine, come si evince dalla figura 33, alla conclusione della partita si potranno visualizzare i risultati di gioco, con la possibilità di uscire dall'app.

Figure 33: Schermata di fine partita.

9 Conclusioni

9.1 Autovalutazione individuale

9.1.1 Alpi Davide

Nel corso di questo progetto sono molto contento dell'aver studiato e appreso nuove tecnologie (mongodb, rabbitmq, protobuf, docker) e i paradigmi ad esse associati, nonostante il tempo richiesto.

Mi sono abituato a ragionare nel distribuito, comprendendo meglio in prima persona tutta una serie di problematiche ad esso associate.

Il lavoro è stato frammentato nel corso di un anno. Questo mi ha senza dubbio creato problemi, ma ha evidenziato ancora di più l'importanza di seguire certe pratiche di buona ingegneria. Inoltre, abituarci a lavorare con questo orizzonte temporale così ampio ha comunque avuto altri 2 vantaggi:

- similarità con orizzonti temporali di progetti "reali",
- visualizzazione del proprio miglioramento come ingegneri del software, paragonandosi ai se stessi di pochi mesi prima (con un bagaglio di più corsi affrontati alle spalle).

Una nota dolente sono state le risorse computazionali a mia disposizione, che non mi permettevano di runnare l'applicazione con i 4 emulatori per il testing. Ho lavorato con il pc dei miei colleghi, tramite teamviewer da remoto.

9.1.2 Parrinello Angelo

Mi ritengo soddisfatto del lavoro da me svolto. Era una totale novità lavorare in un contesto distribuito, quindi ragionare distribuito. Una delle difficoltà più grandi è stata senza dubbio, ragionare e modellare il sistema per container: soprattutto in fase implementativa si rischiava di ragionare come un unico blocco quando in realtà erano servizi esterni fra loro che comunicavano. Un aspetto che sicuramente poteva seguire una strada migliore, secondo il mio punto di vista, è la gestione del progetto a 360 gradi. Infatti lavorando molto a scaglioni (lavoriamo al progetto un mese, poi stacco di 6 poi riprendiamo e così via) si è perso molto tempo ogni qualvolta che si ripartiva, a capire codice scritto e ragionamenti fatti in precedenza. Questo è sicuramente dovuto al fatto che non è stata gestista in maniera adeguata la documentazione, che sì, veniva eseguita, ma superficialmente. Nessuno di noi si aspettava probabilmente che il progetto sarebbe stato così prolungato, forse.

Ad ogni modo, bisogna guardare il prodotto realizzato e cosa si è imparato nel mentre, malgrado qualche problema. In definitiva, esco da questo progetto molto più convinto e solido su temi estremamente importanti nel mondo della produzione del software e quindi ne sono contento.

9.1.3 Penazzi Paolo

Sono molto soddisfatto del progetto, per vari motivi.
Ho familiarizzato con varie tecnologie che non conoscevo e che non avrei approfondito senza questo progetto: in particolare RabbitMQ, Docker, Protobuf e MongoDB. Sviluppare un'applicazione distribuita è stato molto utile per imparare a gestire tutta una serie di situazioni che si presentano in questo contesto, e che fino a questo momento non avevo mai incontrato. Purtroppo lo sviluppo del progetto si è allungato più del previsto e questo ha portato con sè qualche problema, ma sono contento che come team siamo riusciti a gestirli. Mi sarebbe piaciuto utilizzare nel progetto alcune delle cose imparate in seguito come Scrum, tutta la parte di DevOps o altri linguaggi come Kotlin e Scala.

9.2 Considerazioni finali

Il progetto ha rappresentato un'esperienza fondamentale per comprendere l'importanza dei sistemi distribuiti e delle tecnologie utilizzate nella loro realizzazione. Attraverso l'analisi e lo sviluppo di un'applicazione distribuita, è stato possibile approfondire le conoscenze acquisite durante il corso e metterle in pratica. In particolare, l'uso di tecniche di containerizzazione, di comunicazione tra i nodi e di consistenza dei dati oltre che affrontare temi come la scalabilità e la tolleranza ai guasti sono state alcune delle sfide affrontate durante il progetto. Grazie a questo lavoro di gruppo, è stato possibile acquisire competenze importanti e utili sia in ambito accademico che professionale, non solo nel campo dei sistemi distribuiti, ma anche in termini di problem solving, comunicazione e collaborazione.

In particolare poi, la scelta di questo progetto ci ha permesso di poter realizzare una versione distribuita di un gioco di carte molto celebre nel nostro territorio il chè ci rende molto fieri. Inoltre, si sta valutando, a seguito di ulteriori miglioramenti, di rendere disponibile questa applicazione a tutti tramite l'app store e rendere il gioco davvero fruibile a tutti.

L'utilizzo di tecnologie molto mature, ha poi permesso al sistema di ottenere alcune caratteristiche tipiche dei sistemi distribuiti a "gratis". Questo ha poi permesso al gruppo di focalizzare i nostri sforzi su altre tematiche, come la gestione dei casi di disconnessione o la struttura dell'architettura.

In conclusione, si ritiene di aver raggiunto un ottimo risultato nella realizzazione di questo progetto. Se inoltre si considera anche la sfida affrontata nell'integrare molteplici nuove tecnologie e nuovi concetti teorici, il tutto dà ancora più soddisfazione. Sappiamo che il sistema è largamente migliorabile, ma ci riteniamo felici di questa prima versione realizzata.

9.3 Sviluppi futuri

Inizialmente il progetto puntava a sviluppare un'architettura comune per tutti i giochi di carte a turni, in modo tale che chi volesse cimentarsi nello sviluppo di un gioco di carte simile al nostro sarebbe stato molto più avvantaggiato. Questa idea non è stata poi portata avanti in quanto ci siamo trovati in difficoltà a definire fin da subito cosa era generalizzabile e cosa no, generando una difficoltà nell'analisi: i punti generalizzabili subito individuati non erano completi e giusti. Avevamo quindi deciso di procedere alla creazione del Beccaccino, i cui obbiettivi erano ben chiari, per poi poter definire meglio in seguito, con l'esperienza accumulata, il middleware. In corso d'opera però il progetto ha richiesto molte più ore del previsto e un'ulteriore modifica importante come quella del middleware comune, avrebbe fatto largamente sforare il monte ore preventivato. Detto ciò, la prima modifica futura che andrebbe sviluppata sarebbe un middleware comune per i giochi di carte a turni distribuiti, come il nostro.

In seguito, si potrebbe pensare di arricchire l'interfaccia grafica che, essendo fuori dallo scope del progetto, è stata un po' lasciata da parte.

Infine, si potrebbe pensare di aggiungere la possibilità di giocare più round e poi più partite di fila, in modo tale da aumentare l'usabilità dell'applicazione.

References

[1] Alpine dockerhub reference. https://hub.docker.com/_/alpine.

[2] Ampq-homepage. https://www.amqp.org/.

[3] Beccaccino-wikipedia. https://it.wikipedia.org/wiki/Beccaccino.

[4] Clientserver-model. https://en.wikipedia.org/wiki/Client%E2%80%93server_model.

[5] Docker-homepage. https://www.docker.com/.

[6] Dockerfile reference. https://docs.docker.com/engine/reference/builder/.

[7] Dockerhub reference. https://hub.docker.com/.

[8] File build.gradle. https://docs.gradle.org/current/userguide/what_is_gradle.html.

[9] Gradle-homepage. https://gradle.org/.

[10] Gradle-homepage. https://protobuf.dev/overview/.

[11] Gradle wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html.

[12] Mongodb-collections. https://www.mongodb.com/docs/manual/core/databases-and-collections/.

[13] Mongodb-homepage. https://www.mongodb.com/.

[14] Rabbitmq-channel. https://www.rabbitmq.com/channels.html.

[15] Rabbitmq-exchange. https://www.rabbitmq.com/tutorials/amqp-concepts.html.

[16] Rabbitmq-homepage. https://www.rabbitmq.com/.

[17] Rabbitmq-queue. https://www.rabbitmq.com/queues.html.

[18] Using docker compose. https://docs.docker.com/get-started/08_using_compose/.

[19] What is a docker container. https://docs.docker.com/get-started/#what-is-a-container.