CAHu CLONE - Report finale

Rossi Luca
luca.rossi57@studio.unibo.it

Schiaroli Davide
davide.schiaroli2@studio.unibo.it

Tronetti Elisa
elisa.tronetti@studio.unibo.it

Febbraio 2021

Il progetto ha come obiettivo quello di sviluppare una versione digitale e distribuita del gioco *Cards Against Humanity*. La versione ufficiale del gioco è reperibile al seguente link `https://cardsagainsthumanity.com`.

Lo scopo del gioco è quello di utilizzare le carte che si hanno in mano per completare una frase, cercando di rendere il completamento della stessa il più divertente possibile. A inizio partita infatti ogni giocatore riceve 10 carte bianche, chiamate di riempimento, le quali contengono delle frasi oppure delle parole.
Il leader del turno, che nel primo round sarà il creatore della partita, pescherà, oltre alle carte bianche, una carta nera, contenente una frase che avrà uno o più spazi vuoti da completare. Una volta che tutti i giocatori hanno selezionato una o più carte bianche per completare la frase, vengono mostrate anonimamente al leader, in modo che non si possa risalire a chi ha giocato un determinato completamento. Il compito del leader del round è quello di selezionare le carte bianche più divertenti, assegnando di conseguenza un punto al giocatore a cui appartengono. Il vincitore diventa leader del turno successivo e i giocatori pescano per tornare ad avere 10 carte bianche nella mano.
La partita continua in questo modo finché un giocatore raggiunge un determinato punteggio e vince quindi la partita.

Indice

1 Obiettivi del progetto

L'obiettivo del progetto è quello di permettere agli utenti di giocare a Card Against Humanity (CAHu Clone) in una versione digitale e distribuita, sviluppando lato client un applicativo Java che interagisce con un server, il quale permetterà ai diversi client di comunicare tra loro.

Si vuole quindi andare a progettare e implementare una particolare tipologia di architettura Client-Server, con il fine di rendere i client il più indipendenti possibile dal server; in questa architettura il client viene tipicamente denominato *thick client* o *fat client* [4], poiché tutte le informazioni principali della partita e del suo funzionamento sono salvate localmente.

Questo permette di avere un server molto più snello, veloce, meno soggetto a errori e che non deve necessariamente avere ingenti quantità di memoria o potenza computazionale.
Per questa ragione il server non salverà le informazioni relative alle partite ma farà riferimento alle lobby create e ai giocatori che fanno parte delle lobby, in modo da poter inviare i messaggi a tutti i client quando necessario.

Questo consente anche di avere una struttura più solida lato client che non porta a dover ricevere continue informazioni relative allo stato della partita e quindi anche in caso di instabilità di connessione, lo stato della partita non viene perso.
Infatti la logica del gioco, con la gestione di tutte le sue fasi sarà interamente gestita dal client; questo introduce una complessità notevole per quanto riguarda lo sviluppo dell'applicativo, poiché lo stato della partita non è presente in una struttura sincronizzata comune, ma ogni client deve gestire eventuali errori in modo consistente.

Ciò che il gruppo si impegna a consegnare è:

- Un applicativo Java per il client con il quale l'utente potrà interagire per creare o entrare in partite già esistenti, giocare e inviare messaggi agli altri giocatori.
- Un server implementato in Java che permetterà di gestire le lobby e di far scambiare messaggi ai client.

1.1 Scenarios

All'avvio dell'applicativo l'utente deve decidere se creare una nuova lobby e di conseguenza una nuova partita o se vuole entrare in una lobby già esistente.

Nel caso in cui l'utente crei una nuova lobby, sceglierà il nome che vuole utilizzare durante il gioco e dovrà selezionare il numero di punti necessari per poter vincere la partita. Fatto ciò, il server si occuperà di creare la nuova stanza e di fornire il codice

alfanumerico univoco per permettere ad altri utenti di accedervi.

Quando un utente vuole invece accedere a una lobby esistente, dovrà per prima cosa inserire il codice univoco di una stanza e il proprio username, e nel caso in cui esista e non sia già iniziata una partita, effettuerà l'accesso.
Nel momento in cui un nuovo utente entra nella lobby, tutti gli altri già presenti ricevono un avviso che li informa dell'evento, mentre l'utente che ha appena effettuato l'accesso riceve la lista dei giocatori già online nella lobby.

Oltre alla console che mostra queste informazioni, si aprirà all'accesso della lobby una finestra che permette agli utenti di scambiarsi messaggi di testo, la quale costituisce quindi la chat del gioco.

Perché il gioco si possa svolgere correttamente, in una partita devono esserci un minimo di 3 giocatori. Se il creatore della stanza prova a iniziare la partita prima che questo requisito sia soddisfatto, verrà avvisato che ancora non sono presenti 3 utenti nella lobby; in alternativa, il gioco avrà inizio.

L'utilizzo dell'applicativo è differente a seconda se l'utente è leader oppure no:

- un **giocatore** pescherà le 10 carte bianche che costituiranno la sua mano e attenderà di ricevere la carta nera pescata dal leader. Una volta ricevuta la carta da completare, il giocatore può scegliere il completamento che ritiene più divertente, digitando il numero della carta che vuole selezionare. A questo punto aspetterà che tutti gli altri partecipanti scelgano il completamento. Una volta che tutti hanno giocato, il giocatore riceve le carte che sono state giocate e quando il leader avrà deciso la carta vincente, tutti scopriranno il giocatore a cui appartiene, che sarà il leader del turno successivo.
- il **leader** invece, una volta pescata (in modo automatico) la carta nera a inizio round, dovrà attendere che tutti i giocatori selezionino la carta bianca che hanno intenzione di giocare. Quando riceverà tutte le carte giocate, potrà scegliere la sua preferita, per poi sapere quale giocatore l'ha selezionata. Al turno successivo il leader diventa un giocatore normale e quindi rientra nella casistica precedentemente descritta.

Alla fine di ogni turno tutti gli utenti vedranno la classifica aggiornata che mostra quanti punti ha ottenuto ogni giocatore.
Quando un giocatore raggiunge il punteggio prestabilito dall'utente durante la creazione della lobby, vince la partita e si troverà fuori dalla lobby dove potrà decidere se creare una nuova partita, entrare in una lobby già esistente o uscire dal gioco.

1.2 Self-assessment policy

Il gioco deve poter girare su qualsiasi client dotato di jvm con una versione di java maggiore o uguale alla 11.

Dovrebbe sempre essere possibile creare una partita o collegarsi ad una partita esistente (non ancora iniziata) di cui si conosce il codice.
Una volta entrati in partita deve essere possibile giocare giocare e chattare, anche contemporaneamente, e deve esserci coerenza e sincronia fra i giocatori di una stessa lobby: ad esempio i punteggi e i mazzi di carte devono essere consistenti fra i vari host, non presentando incongruenze o errori.

- Il software prodotto ha come obiettivi principali la massimizzazione della scalabilità e la fruibilità del gioco. È importante che il software prodotto giri in modo fluido, senza dare problemi di performance sui dispositivi ospitanti ed è importante che non si blocchi, facendo rimanere in stallo tutti i giocatori in partita.
  Il software prodotto deve soddisfare le più comuni caratteristiche di buona programmazione, in particolare è importante che il software sia leggibile e facilmente estendibile (almeno per modifiche previste e coerenti alla versione del gioco sviluppata, es: modificare le regole del gioco, estenderlo con nuove carte ecc).

- Il progetto è stato sviluppato per consentire un'esperienza piacevole e fluida, che consumi una piccola quantità di banda e resti veloce e funzionante a prescindere dalla quantità di giocatori online.
  Purtroppo ci è molto complesso testare sul campo la scalabilità del server e il suo comportamento sotto stress.
  Sono stati effettuati numerosi test con junit per simularne il corretto funzionamento ma non è stato possibile (sia per la scarsa visibilità del progetto, sia per la scadenza della prova gratuita di AWS che ci ha tolto la possibilità fare partite online) verificare il numero effettivo di connessioni che il sistema è in grado di gestire in contemporanea.

2 Analisi dei requisiti

Di seguito verranno elencate le analisi dei requisiti.

2.1 Requisiti impliciti

Per il corretto funzionamento del programma è necessario che ogni host riceva correttamente i messaggi che vengono inviati dagli altri. Per far sì che ciò accada è stata usata una connessione di tipo TCP e si assume che se un messaggio viene mandato da un host, lo stesso messaggio verrà ricevuto da ogni altro host nella lobby in cui si sta giocando la partita.

Si tiene a precisare che nel tipo di approccio scelto il compito del server è solo quello di far "rimbalzare" messaggi e indirizzarli agli host corretti. Si sarebbe potuto anche usare un approccio che sfruttasse il server per immagazzinare informazioni comuni a tutti gli host e lo storico dei messaggi ma per garantire la massima scalabilità possibile si è demandato questo lavoro ai client.

2.2 Ipotesi implicite

Il progetto, nella sua interezza, ha senso sotto le ipotesi di funzionamento attuale di internet, con scarsità di indirizzi IP, difficoltà di comunicazione fra host senza IP pubblici, come la maggior parte degli utenti, e utilizzo di NAT, sia a livello di rete domestica che, talvolta, di ISP.
Tutte queste motivazioni hanno portato allo sviluppo dell'attuale applicazione con logica **fat client** e fortemente orientato al peer-to-peer.

2.3 Requisiti non funzionali

I requisiti non funzionali del progetto sono, buona scalabilità e facilità d'uso nell'utilizzo del gioco. Come già preannunciato, tutta la logica del gioco viene gestita lato client, rendendo il server quasi completamente scarico di ogni informazione, in questo modo si garantisce la massima scalabilità possibile, permettendo un utilizzo dell'applicazione da un numero massiccio di utenti in contemporanea.

Per quanto riguarda la semplicità d'uso, essendo il software sprovvisto di GUI (se non per la chat), è stato necessario fare in modo che il suo utilizzo fosse il più semplice possibile, con poche istruzioni e pochi comandi a disposizione. Questo ha permesso una migliore esperienza di gioco e dovrebbe facilitare in futuro l'implementazione di una GUI da consegnare ad utenti non avvezzi alla linea di comando.
Infatti, allo stato attuale modificare le interazione dell'utente da linea di comando a un'effettiva interfaccia grafica richiederebbe un lavoro non troppo complesso grazie agli accorgimenti presi durante le fasi di progettazione e sviluppo.

2.4 Paradigmi e tecnologie: usati vs ideali

Idealmente l'applicazione non avrebbe bisogno di alcun tipo di server e sarebbe completamente autogestito, tuttavia per la gestione delle partite e in generale per il matchmaking non si è potuto fare a meno di un server che facesse da punto di sincronizzazione e di riferimento a cui collegarsi in fase di creazione della partita.
L'applicazione è stata pensata in un'ottica di estrema indipendenza in modo da aumentare al massimo la scalabilità, per questo motivo avremmo voluto utilizzare un'architettura peer-to-peer che dopo l'inizio della partita si distaccasse completamente dal server, rendendo il servizio utilizzabile contemporaneamente da un numero molto ampio di utenti.

La soluzione che abbiamo implementato si avvicina molto a quella ideale ma sfrutta un server per lo scambio di messaggi.
Questa scelta è motivata dall'impossibilità di avere in modo comodo e veloce un indirizzo IP pubblico da associare ad ogni utente e dalla volontà di rendere la nostra applicazione utilizzabile così com'è dall'utente finale.

La progettazione ha sempre avuto un occhio di riguardo per la possibile estendibilità del sistema, in ottica ad esempio di IPv6.
Nel caso diventasse facile la gestione degli IP, ad esempio avendo un IP pubblico di default, e non avendo NAT, basterebbe modificare qualche riga di codice per passare ad un modello peer-to-peer, che utilizzerebbe un server con l'unico scopo di fornire gli indirizzi degli utenti al fine di rendergli possibile comunicare tra loro.

3 Tecnologie

Fra le tecnologie sfruttate per il progetto abbiamo:

- Gradle
- Docker
- AWS
- Git
- VertX
- Java
- GSON
- Swing
- Junit
- Websocket

3.1 Gradle

Gradle è uno strumento di automazione della compilazione di applicativi Java, che permette di scaricare dipendenze, compilare e testare codice in modo veloce e richiedendo all'utente un'interazione minima.
Gradle permette infatti di utilizzare librerie open source senza doverne importare esplicitamente, ma semplicemente andando a specificare all'interno del file *build.gradle* la repository in cui tale codice è presente.

Nel nostro caso Gradle è stato utilizzato per dichiarare in modo semplice e immediato le librerie relative a Verxt e GSON, due tecnologie utilizzate nel progetto che discuteremo in seguito.
Come specificato sopra, l'uso di Gradle non solo garantisce un modo immediato per la gestione delle dipendenze, ma permette di compilare e testare il codice per mezzo di comandi utilizzabili tramite terminale.

Tipicamente un comando Gradle ha la seguente struttura:

```
$ gradle [taskName...] [--option-name...]
```

Dati i vantaggi appena descritti, l'utilizzo di Gradle è stato estremamente utile e fondamentale per lo svolgimento di questo progetto.

3.2 Docker

Avendo avuto la necessità di spostare la parte Server della nostra applicazione tra servizi di hosting, abbiamo pensato che la containerizzazione dell'applicazione avrebbe potuto essere d'aiuto.

Si è pensato subito come soluzione a Docker, sia perché già studiato a lezione, sia perché è senz'altro lo standard de facto per quello che riguarda la containerizzazione, avendo ormai superato come utilizzo e supporto LXC (Linux Container).
In particolare Docker utilizza il kernel di Linux, in particolare sue funzionalità come i namespaces e i cgroups, per costruire processi isolati in dei container.

3.3 AWS

Essendo il sistema distribuito ed essendo l'intento principale del gruppo quello di creare un gioco che fosse realmente accessibile dall'esterno abbiamo deciso di mettere online nel minor tempo possibile la versione prototipata del progetto in maniera da iniziare a giocare qualche partita e controllare che il risultato della partita fosse quello atteso.

In questo caso la scelta è ricaduta su Amazon Web Services, il più famoso e utilizzato sistema del suo tipo. Per lo scopo del progetto abbiamo utilizzato una delle versioni più basilari di server presenti nella piattaforma online, Amazon EC2.

3.4 Git

Git è un software di versionamento distribuito utilizzabile sia da interfaccia che da linea di comando. Viene tipicamente usato per coordinare il lavoro sullo stesso software tra più sviluppatori, facendo in modo che sia possibile mantenere il codice sempre aggiornato nonostante venga modificato contemporaneamente.

Dato che il progetto è stato sviluppato dai 3 membri del gruppo, l'utilizzo di un sistema di versionamento è stato necessario, dato che ci ha permesso di coordinarci più facilmente e in alcuni casi di consultare passate versioni del software. Inoltre è stato possibile vedere i vari cambiamenti in modo rapido ed efficace grazie ai commit fatti dai membri, che consistevano sempre di una descrizione molto significativa delle modifiche compiute.

3.5 VertX

VertX è una piattaforma su cui è possibile rilasciare ed eseguire le proprie applicazioni sia lato client che lato server. Questo framework viene utilizzato per la realizzazione di applicazioni reattive in Java; infatti VertX gira su una JVM ed è scalabile, concorrente, non bloccante, distribuito e poliglotta.
Offre un modello di programmazione event driven, quindi è progettato per lavorare principalmente in modo asincrono.

Un applicativo VertX è composto da uno o più componenti denominati *Verticle*. Ogni Verticle del sistema viene eseguito in maniera concorrente e autonoma rispetto agli altri.

Nel nostro caso, questo tipo di organizzazione del framework ha permesso la creazione di un Verticle indipendente per ogni partita, di modo da rendere il server estremamente scalabile.
Il framework VertX permette anche la creazione di client, che consentono di effettuare in modo semplificato lo scambio di messaggi HTTP tra client e un web server. Nel nostro caso VertX è infatti stato utilizzato anche lato client proprio per le agevolazioni che questa piattaforma mette a diSposizione.

3.6 Java v11

Data l'elevata complessità del client e la scelta di usare VertX, si è deciso di utilizzare il medesimo linguaggio anche per la scrittura di codice lato client.
È stata usata la versione 11 del linguaggio, la stessa vista durante il corso.
Nonostante javascript sia il linguaggio del web, sia orientato al front-end e lavori nativamente con i Json, abbiamo comunque deciso di utilizzare Java per motivi come: tipizzazione, maggiore fluenza da parte di tutti i componenti del gruppo e volontà di sperimentare alcuni dei concetti visti a lezione.

È stato fatto un uso abbastanza intensivo di Runnable ed ExecutorService permettendoci di compartimentalizzare bene diverse parti di codice e di tenere separati concettualmente input ricevuto e output visualizzato.
Inoltre abbiamo potuto sfruttare anche le *CompletableFuture*, che hanno consentito la gestione dello scambio di messaggi con il server in modo asincrono e non bloccante.

3.7 GSON

Il gioco sviluppato è la versione digitalizzata e distribuita di un reale gioco di carte molto famoso e conosciuto, online se ne trovano diverse versioni e formati. Quella che abbiamo deciso di utilizzare è scritta in Json e per poter maneggiare questo tipo di dati con facilità abbiamo usato una delle più note librerie in circolazione in questo ambito, ovvero GSON.
La libreria è molto conosciuta e sviluppata ed esistono strumenti online che permettono a partire da un Json di ricavare la schematizzazione conforme alle specifiche GSON da utilizzare nel programma.
Il tool online utilizzato è stato altrettanto utile per avere una più chiara definizione degli oggetti utilizzabili e ha permesso di evitare banali errori di trascrizione o di formattazione.

3.8 Swing

Essendo la grafica non valutata ed essendo poco pratico chattare in bash o equivalentemente da linea di comando, è stata scelta la più conosciuta e utilizzata libreria in

circolazione per costruire ambienti grafici in Java, ovvero Swing.
La libreria è stata utilizzata per la creazione di una chat molto basilare con una *text-area* con i messaggi arrivati, una con il messaggio da inviare e un bottone per mandarlo.
Questa scelta si è rivelata particolarmente giusta per la semplicità con cui si riesce a creare un pannello di base e successivamente modificarlo.

Ciò ci ha permesso di mettere in piedi in poco tempo un prototipo funzionante con cui scambiare messaggi al fine di controllare il corretto comportamento sia della chat che del server. È infatti spesso capitato che il server non inoltrasse i messaggi inviati agli altri utenti, e la presenza di un GUI ci ha permesso di identificare precocemente il problema e per questo siamo riusciti a indirizzare meglio la nostra attenzione e di debuggare più velocemente.

3.9 Junit

Junit è un framework per l'unit testing su JVM. Dalla versione 5 è diviso in tre componenti:

- Junit Platform - Layer di base che permette il lancio di test.
- Junit Jupiter - Fornisce un TestEngine per eseguire test per l'ultima versione di Junit (5)
- Junit Vintage - Fornisce un TestEngine per eseguire test di versioni precedenti di Junit (3,4)

L'utilizzo Junit ha consentito di testare in modo dettagliato e mirato le varie componenti di base che costituiscono i sistema, di modo che fossimo certi che ogni parte svolgesse il suo compito correttamente. Inoltre è stato possibile modificare durante lo sviluppo alcune parti del codice, avendo sempre la certezza che il resto continuasse a funzionare come previsto.

3.10 Websocket

Le Websocket sono una tecnologia che permette lo scambio di messaggi tra client e server HTTP utilizzando una connessione full-duplex.
Per questo progetto, abbiamo utilizzato l'implementazione delle Websocket di Vertx.

L'utilizzo delle Websocket ha semplificato lo scambio di messaggi realtime tra client e server, oltre a permettere il pooling, che oltre a velocizzare lo scambio di messaggi, sarebbe stato utile per costruire funzionalità interessanti come gli heartbeat.
D'altra parte, l'utilizzo di Websocket crea delle falle di sicurezza per quanto riguarda l'header "Origin", in quanto la mancata validazione dello stesso può portare ad attacchi di tipo XSRF, ovvero Cross-site request forgery.
Non trasmettendo dati sensibili nelle richieste HTTP, ed essendo questo un progetto non incentrato su tematiche di sicurezza delle reti, abbiamo pensato di non considerare questi aspetti.

4 Design

Di seguito si andranno ad elencare e motivare le scelte di design adottate.

4.1 Struttura

Il progetto è stato strutturato secondo il pattern architetturale **MVC**, che ci ha permesso di dividere il codice in blocchi distinti a seconda delle funzionalità [2].
MVC è un acronimo che sta per Model-View-Controller, e ognuna delle parti ha un ruolo ben preciso:

- **Model**: il Model costituisce la struttura del dominio applicativo in modo completamente indipendente dall'interfaccia grafica. Tutte le strutture dati e gli oggetti che costituiscono il core dell'applicativo sono incapsulati nel modello.
- **View**: con View si intende un qualsiasi tipo di rappresentazione in output, con il quale l'utente può eventualmente interagire e che mostra i risultati delle sue azioni.
- **Controller**: il Controller è il componente che riceve i comandi dell'utente attraverso la View e reagisce con operazioni che potrebbero andare a modificare il Model e conseguentemente aggiornare lo stato della View. Il Controller implementa la logica del programma, il motore che permette il funzionamento e l'interazione tra le altre due componenti.

La base del progetto può essere schematizzata tramite le seguenti classi, in Figura 4.1.

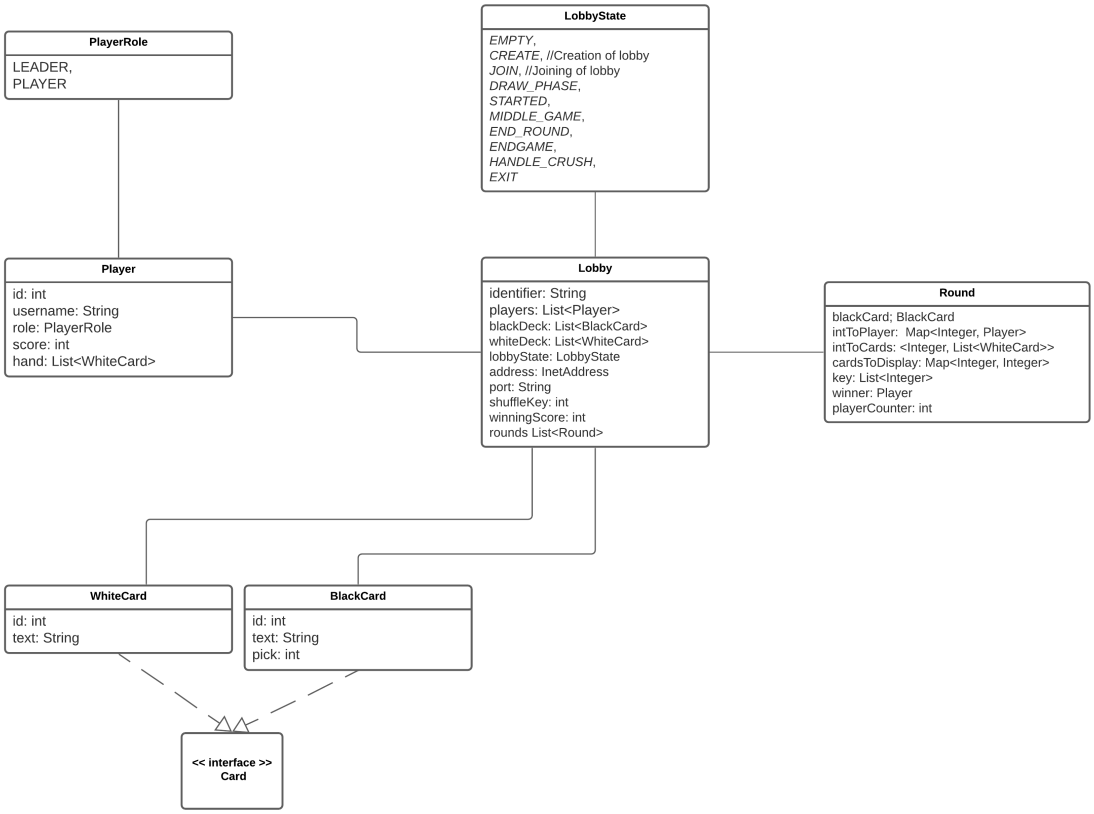

Figure 1: Diagramma delle classi più rappresentanti per il sistema lato client

Queste classi costituiscono il Model e gestiscono le strutture dati del gioco e tutti gli aspetti implementativi.
Sono stati modellati i giocatori con il nome di *Player* e il loro ruolo è stato separato in un Enum che permette di definire in modo chiaro il fatto che è possibile che un giocatore sia o il *Leader* o un semplice *Player*.
La *Lobby* è la struttura principale, la quale modella tutta una partita; anche in questo caso abbiamo separato lo stato della lobby, che costituisce la fase del gioco in cui la partita si trova, in un Enum separato che permette di farci riferimento in modo più esplicativo.
Una partita è stata suddivisa in *Round*, classe che permette di modellare chiaramente lo svolgimento di una sola parte del gioco, la quale si ripete più volte durante l'intero svolgimento di una partita.
Infine i due tipi di carte, le carte bianche e le carte nere, implementano la stessa interfaccia; l'unica importante distinzione è quella che la carta nera ha un campo che identifica il numero di carte bianche necessario per completarla.

Il flusso principale di comandi dietro la logica viene gestito tramite il Controller, che è anche incaricato di far iniziare e concludere la partita. Inoltre si occupa della suddivisione del gioco in varie fasi e per ognuna di esse è stato implementato il suo diverso funzionamento.

La View è stata trattata il più semplicemente possibile: avendo lavorato tramite console sono state semplicemente usate delle stampe, di modo da dare un feedback all'utente. Una componente aggiuntiva e a sé stante è invece la chat, che ha una GUI apposita per rendere più agevole il controllo e lo scambio dei messaggi rispetto che a un ambiente command line. Si è optato per un approccio di questo tipo in quanto sarebbe stato oltremodo scomodo leggere messaggi di sistema e inserire input, dovendo stare attenti a distinguere messaggi arrivati dalla chat da quelli dello stato della partita.

4.2 Behaviour

Le entità principali coinvolte sono tre:

- **Client**: il Client è l'entità principale, la quale inizia la comunicazione con il server. Quando l'utente vuole creare una nuova partita o vuole entrare in una partita già esistente, il server centrale viene contattato.
- **Server**: il server centrale ha il compito di stare in ascolto di eventuali richieste da parte dei client e di fornire le informazioni relative alla lobby appena creata o alla lobby alla quale l'utente vuole accedere.
- **LobbyVerticle**: il LobbyVerticle viene generato dal server e si occupa della gestione completa di una partita. Infatti una volta che il server centrale ha fornito all'utente le informazioni principali per contattare il verticle responsabile della lobby, il suo compito termina e l'utente non avrà più alcuna comunicazione diretta con esso. Il LobbyVerticle invece avrà le unicamente le informazioni dei giocatori, di modo da poter inviare i messaggi all'occorrenza.

In Figura 4.2 è presente l'*Activity Diagram* che mostra il flusso delle attività, nello specifico come avviene l'interazione tra le tre entità descritte in precedenza quando il client vuole creare una nuova partita.

Inizilmente il client invia al server centrale la richiesta di creazione di una nuova Lobby. Ricevuta questa richiesta il server genera il Verticle responsabile della partita e demanda la comunicazione con il client ad esso. Il Verticle invia i dati relativi al suo indirizzo e al codice utilizzabile per entrare nella lobby e il client, una volta ricevute tali informazioni, diventa il leader della partita.
Il Verticle genera anche una chiave che verrà utilizzata da tutti i giocatori della partita per mescolare il mazzo; in questo modo i mazzi saranno tutti mescolati allo stesso modo, evitato quindi incongruenze. Il client, mentre attende altri utenti, mescola quindi il mazzo basando sulla chiave generata e memorizzata dal Verticle.

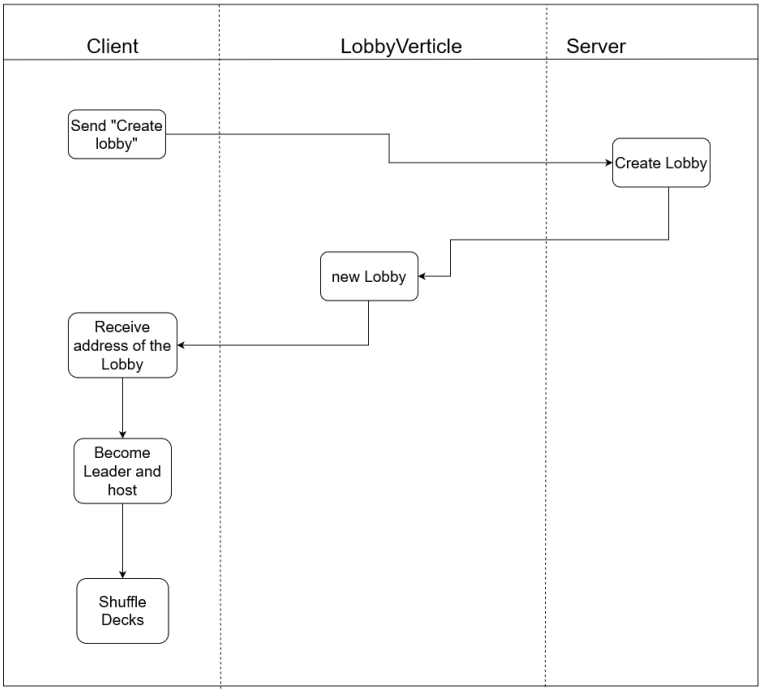

Figure 2: Activity Diagram che mostra l'interazione tra il giocatore leader e il server

In Figura 4.2 il diagramma delle attività rappresenta invece il flusso che si svolge nel caso il cui il client abbia intenzione di entrare in una Lobby già esistente e quindi entrare in comunicazione con un LobbyVerticle che è già stato generato in precedenza.

All'utente verrà richiesto di inserire il codice della Lobby alla quale vuole accedere, supponendo che gli venga fornito dal creatore della Lobby stessa. In questo modo il client si metterà in comunicazione con il server, il quale controllerà che la Lobby richiesta esista. Nel caso in cui il controllo avvenga con successo, il nuovo giocatore della partita riceve tutte le informazioni necessarie per entrare in comunicazione con il LobbyVerticle e tutti i dati relativi alla partita: riceve la chiave da utilizzare per mescolare il mazzo, i nomi dei giocatori presenti nella Lobby e il punteggio che decreterà il vincitore, che è stato precedentemente scelto dal creatore della partita.
Infine, il client mescola il mazzo, di modo che sia consistente con quello degli altri giocatori in partita e si mette in attesa che il leader dia inizio al gioco.

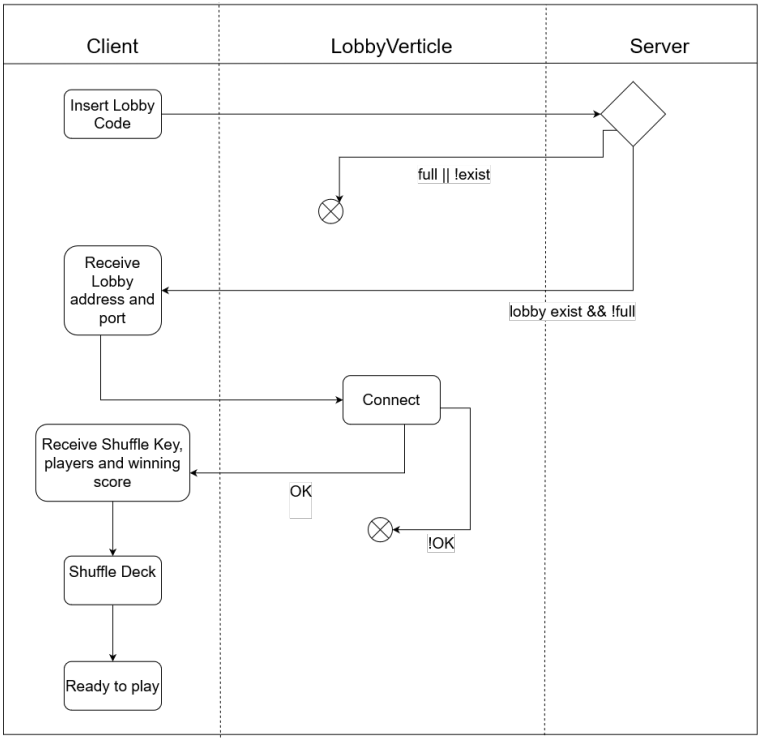

Figure 3: Activity Diagram che mostra l'interazione tra il giocatore che vuole entrare nella lobby e il server

L'Activity Diagram in Figura 4.2 mostra il flusso delle attività quando il leader avvia effettivamente una partita. Per semplicità di lettura del diagramma il numero di giocatori coinvolti è solo di due, anche se non è effettivamente possibile svolgere una partita con solo due giocatori.

È necessario notare il fatto che il server centrale non è più presente tra le entità che interagisco in questa fase, in quanto il suo compito è stato svolto e lo scambio di messaggi tra i giocatori di una partita è completamente gestito dal LobbyVerticle, garantendo quindi un'importante scalabilità.

Quando il giocatore che ha creato la stanza, indicato come leader, decide di iniziare la partita, invia un messaggio al LobbyVerticle comunicandogli la sua intenzione. Il verticle si occupa di informare tutti i giocatori che la partita ha inizio; a questo punto tutti i giocatori pescano le carte che andranno a costituire la loro mano. In questa fase, è importate che ogni giocatore segua una politica ben precisa per il pescaggio, di modo che non ci siano più giocatori con le stesse carte in mano.
Una volta eseguita questa prima fase di pescaggio, può avere inizio la partita effettiva.

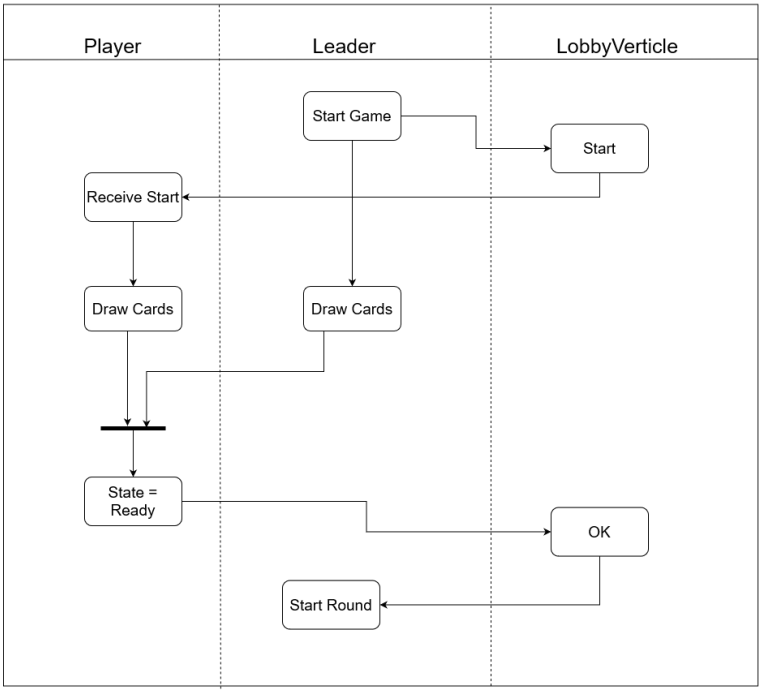

Figure 4: Activity Diagram che mostra l'interazione tra giocatori e server quando viene dato inizio alla partita

Infine quest'ultimo Activity Diagram, in Figura 4.2 va a descrivere il flusso delle attività che avviene durante un round completo, di modo da chiarire in che modo interagiscono tra di loro le entità durante la fase principale del gioco.

All'inizio del round il leader pesca la carta nera, ovvero la carta che gli altri giocatori dovranno completare con le carte bianche che compongono la loro mano. Il leader invia al LobbyVerticle l'identificatore della carta nera pescata, di modo che quest'ultimo mandi l'id a tutti gli altri giocatori. Il leader si mette quindi in attesa che tutti i giocatori selezionino una carta bianca da giocare.

Il server riceve gli id delle carte bianche giocate da ogni giocatore e le manda a tutti gli altri, finché non vengono inviate N-1 carte, con N uguale al numero di giocatori presenti nella partita.
Tutti i giocatori vedranno a questo punto le carte giocate e solo il leader avrà la possibilità di selezionare la combinazione più divertente.

Il giocatore vincitore viene comunicato a tutti gli utenti e localmente ogni client si occuperà di gestire la fine del round, senza ulteriori comunicazioni con il LobbyVerticle. Ogni client infatti aggiorna la classifica e controlla se un utente ha raggiunto il punteggio massimo che ne sancisce la vittoria: in caso affermativo, la partita termina e la socket di comuncazione con il LobbyVerticle viene chiusa; in alternativa, ogni client pesca le

carte, di modo da ritornare ad avere tutte le carte in mano. Infine il vincitore del turno corrente diventa leader e viene iniziato un nuovo round.

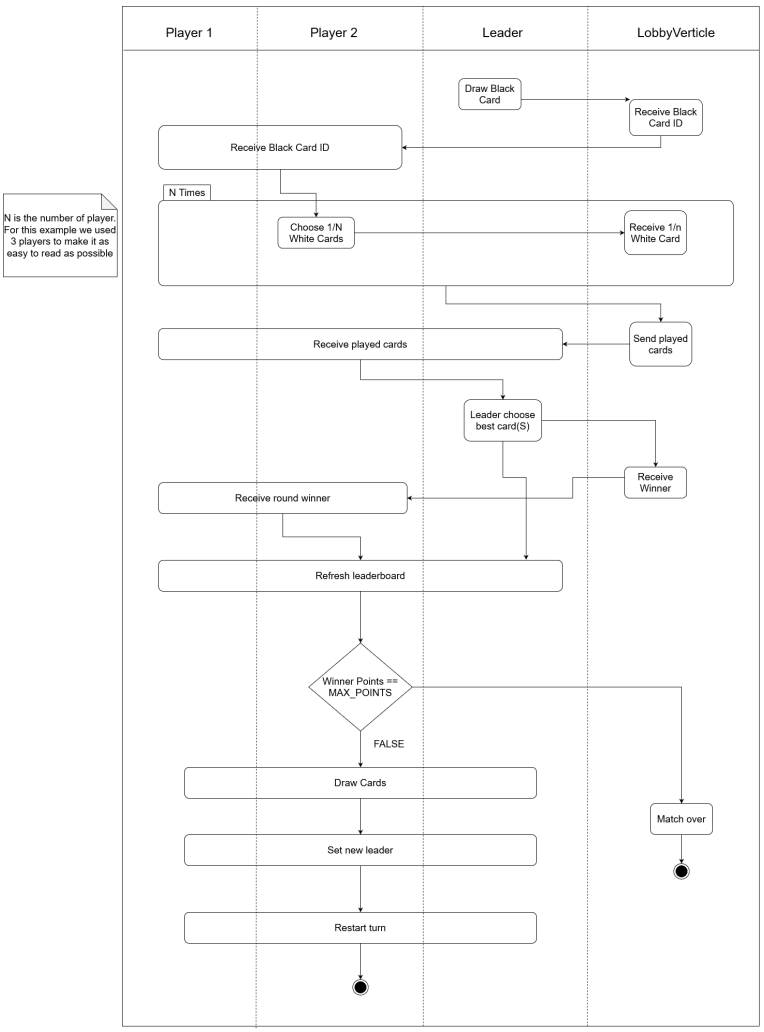

Figure 5: Activity Diagram che mostra l'interazione tra giocatori e server durante lo svolgimento di un round

4.3 Interaction

Per la prima interazione, il client manderà una richiesta HTTP Post al server, richiedendo di creare una nuova lobby, oppure di entrare in nuova lobby già esistente. Per entrambe le richiesta è stata scritta una specifica Swagger, visto che rispetta i principi Rest.
Dopo aver richiesto la creazione o l'accesso alla lobby, il server risponderà con il numero di porta della lobby corrispondente e qua termina lo scambio di dati tra client e server. Dopo aver ricevuto il numero di porta della lobby, ogni client creerà una websocket per scambiare messaggi con la lobby. Per scambiare informazioni si è deciso di scrivere i

messaggi utilizzando il formato Json e decidendo un modo formale per poterli leggere e scrivere.
In particolare, si è scelto di comunicare seguendo il seguente formato:

```
{
    type: "messageType",
    attribute1: "value1",
    attribute2: "value2",
    attribute3: "value3",
}
```

Il diagramma di sequenza in Figura 4.3 rappresenta in che modo comunicano Client, Server e LobbyVerticle quando un utente vuole creare una nuova Lobby. Come detto in precedenza, gli unici messaggi scambiati da Client e Server sono quelli che permettono la richiesta di creazione di un nuovo Verticle da parte del Client e le informazioni per aprire una WebSocket di comunicazione con il Verticle.

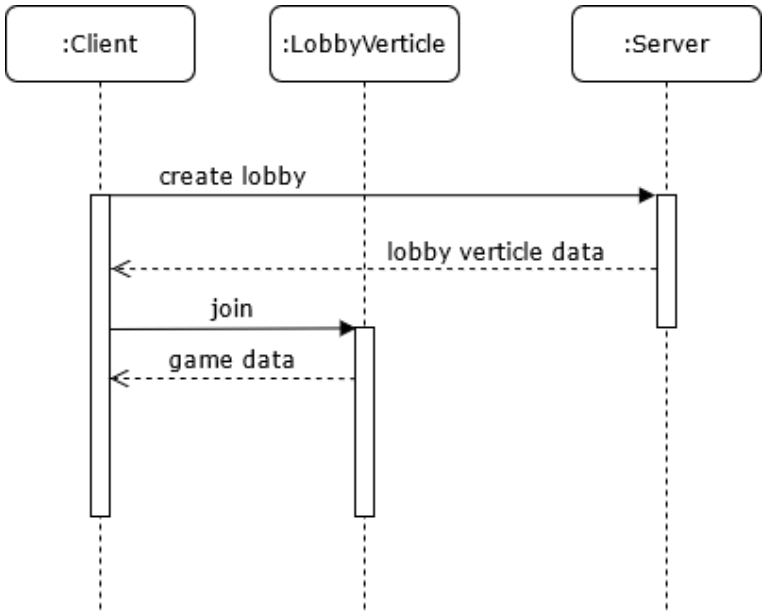

Figure 6: Diagramma di sequenza che mostra i messaggi scambiati tra un utente e il server durante la creazione della lobby

È stato deciso di trascurare il diagramma di sequenza che rappresenta lo scambio di messaggi durante la fase di accesso a una Lobby già esistente, poiché sarebbe del tutto equivalente a quello appena discusso; l'unica differenza è che il Server riceve il codice alfanumerico di una Lobby e ne ritorna i dati al Client.

Molto più di interesse è il diagramma di sequenza in Figura 4.3, che rappresenta lo scambio di messaggi tra i vari componenti durante lo svolgimento di un round completo.

Come detto in precedenza, l'obiettivo principale che abbiamo tenuto in considerazione per tutto lo svolgimento del progetto è quello di fare in modo che la gestione della logica del gioco avvenisse il più possibile lato client e che quindi il compito del server fosse solo quello di inoltrare i messaggi. Per questa ragione le comunicazioni con il server sono state ridotte il più possibile, dato che tutta l'elaborazione del gioco viene svolta localmente.
A inizio partita il leader invia al Verticle la sua intenzione di iniziare la partita e tutti gli altri giocatori vengono notificati.
A questo punto il leader pesca una carta nera e ne invia l'id al Verticle, il quale lo inoltra agli altri giocatori.
Ogni utente seleziona le carte che vuole utilizzare nel turno corrente e il server si occupa nuovamente al semplice invio di tali informazioni.
L'ultimo messaggio che viene inviato è quello della selezione da parte del leader del vincitore del turno.

Come si può quindi notare il server non modifica in alcun modo i dati che riceve, ma li manda semplicemente a tutti gli utenti tranne quello dal quale ha ricevuto il messaggio. Inoltre, nonostante le informazioni che il server riceve sarebbero sufficienti per mantenere lo stato della partita, dopo l'opportuna elaborazione di tali dati, ciò non viene fatto per mantenerlo veloce e reattivo.

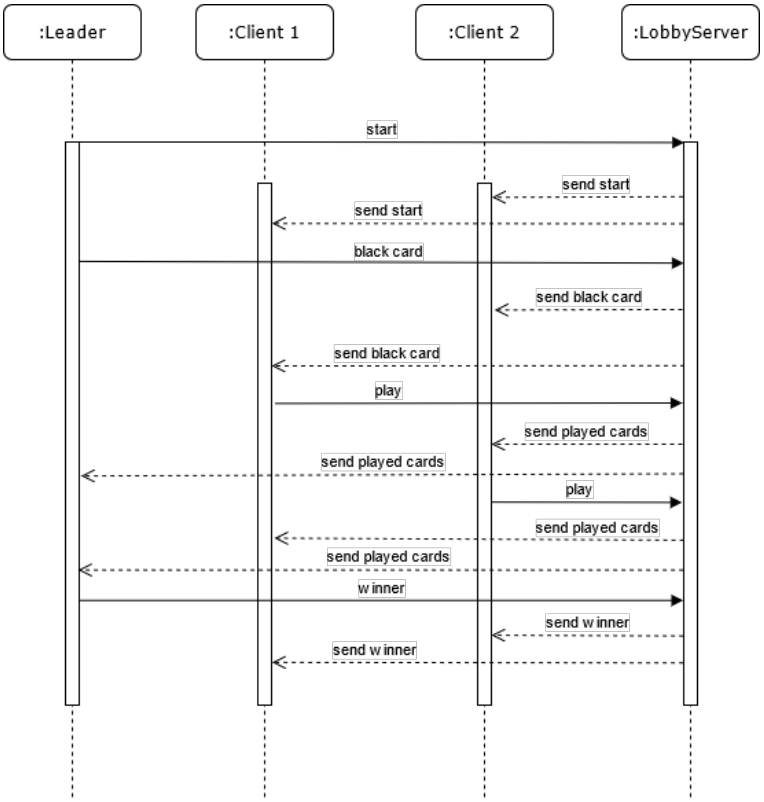

Figure 7: Sequence Diagram che mostra tutti i messaggi scambiati i giocatori e il lobby verticle durante lo svolgimento di un round

5 Implementation Details

Nello sviluppo dell'applicativo lato client una parte degna di nota è il vasto utilizzo di Runnable ed ExecutorService: andiamo a vedere le motivazioni dietro questa scelta.

Il framework utilizzato per lavorare lato server (VertX) mette a disposizione dei validi e utili strumenti, che ben si prestavano al nostro progetto; in particolare per la gestione della connessione sono state usate le websocket. Le socket hanno bisogno di continuo scambio di messaggi e rendono estremamente semplice la creazione di chat e stanze per giocare online.
La parte critica dietro il loro utilizzo sta però nel lato client. In molti punti del gioco ci si può ritrovare in attesa di ricevere messaggi (altri giocatori devono entrare in lobby, i giocatori devono giocare le proprie carte o il leader deve decretare la carta più divertente), in poche parole bisogna sempre restare in ascolto, ma in quei frangenti la console deve essere comunque reattiva, deve essere possibile giocare carte, dare comandi o banalmente mandare messaggi in chat.
Avendo un unico mezzo per gestire input e output (la console) soddisfare i requisiti appena scritti risulta difficile, se non impossibile. In particolare il problema emerge quando il giocatore deve poter inserire degli input mentre potrebbe ricevere degli output.

Per risolvere il problema si è deciso di mantenere separati vari concetti distinguendo:

- La chat, incaricata di mostrare i messaggi ricevuti nell'apposita TextArea e di mandare i messaggi agli altri utenti.
- L'output, con messaggi di sistema e stato della partita.
- L'input, questa parte è poi diventata un tantino più grande di quanto non si fosse inzialmente programmato ma se ne parlerà più in dettaglio a breve.

5.1 Chat

Per semplicità si è deciso di creare una chat utilizzando Swing, in tal modo entrando in partita viene creato un thread apposito, incaricato unicamente della gestione dei messaggi della chat, e della loro rappresentazione (ovvero la molto semplice interfaccia grafica a cui abbiamo già accennato).
Questa tecnica ci ha concesso una miglior gestione generale del codice scritto e una migliore separazione dei concetti.

5.2 Output

Questa parte è presa in carica dal Controller che si impegna a mostrare in output tutti i messaggi importanti per l'utente.
In particolare questa parte può essere vista come punto di sincronizzazione e filtraggio dei messaggi.

5.3 Input

Questa parte doveva essere inizialmente incaricata del solo input ma andando avanti con l'implementazione è stato più comodo gestire la fase di input con dei "sistemi a task" in cui ogni task viene racchiuso in un Runnable che mostrerà eventuali messaggi di input errato o le istruzioni da svolgere e una volta completato ridarà la parola al controller per passare alla fase successiva del gioco.

6 Validazione

Come metodo di valutazione del codice abbiamo sempre fatto riferimento ai test da noi scritti. In particolare, per alcuni parti del client, abbiamo sviluppato il codice attraverso la modalità del "Test Driven Development". Per quanto riguarda il server invece, visto la difficoltà nello scrivere test non banali per parti di codice che vengono eseguite in modo asincrono, abbiamo deciso di testare le API che il server metteva a disposizione.
Dopo ogni commit dove venivano apportate modifiche sostanziali, sopratutto nella parte server, il codice veniva analizzato con strumenti che l'IDE da noi utilizzato metteva a disposizione, come il Profiler Java, per identificare eventuali colli di bottiglia e SpotBugs e il Code Inspector default di Intellij Idea.

7 Testing

Per rendere la struttura dell'applicazione più solida abbiamo costruito dei test, utilizzando il framework Junit, progressivamente durante lo sviluppo del progetto.
Possiamo distinguere due tipologie di test: i test che riguardavano la logica del gioco, come ad esempio la non replicazione delle carte da gioco e i test che riguardavano l'interazione client e server in rete.
Dopo avere sviluppato le prime versioni del gioco che permettevano l'interazione tra giocatori, abbiamo aggiunto la possibilità di lanciare i test direttamente da gradle e poi anche durante il push su Gitlab, utilizzando la pipeline.

7.1 Testing logica gioco

Per il gioco lato client sono stati creati degli host simili a degli agenti, regolati da stati che simboleggiano lo stato della partita.
Nel testing ci siamo assicurati del corretto funzionamento dei comandi e del corretto passaggio da uno stato all'altro in seguito a eventi specifici, come la scelta delle carte o il settaggio di informazioni riguardanti la lobby. È stato inoltre testato il corretto funzionamento di un "round", ovvero del periodo che va dal rilevamento di una carta nera al decretamento di un vincitore e del suo conseguente diventare leader.
Sono stati fatti inoltre test riguardanti il numero di carte pescate e la coerenza del mazzo.
Ciò che non è stato possibile controllare è invece la coerenza di mazzi fra host diversi quando si gioca online, in quel caso le prove sono state fatte di persona, giocando più partite e assicurandosi una corretta gestione della partita.
Molti test portano con loro delle stampe, che sono collegate al funzionamento della partita e all'output del gioco.

7.2 Testing interazione client-server

Dopo aver testato la logica di gioco e che lo stato della partita fosse coerente tra i vari giocatori, si è passato a testare la corretta costruzione delle lobby da parte del server.
Per fare ciò abbiamo riscritto una versione più piccola in termine di linee di codice del server, eliminando parti non funzionali al solo testing. Dopo aver costruito questo "mockup" del Server, abbiamo testato con Junit che un client potesse correttamente richiedere l'accesso ad una lobby o crearne una nuova.
Per fare questa tipologia di test abbiamo utilizzato la libreria standard di Java per il protocollo HTTP, **HttpClient**, in modo da non essere vincolati a Vertx per quanto riguarda la creazione di semplici messaggi HTTP. Dopo aver costruito il client di testing, abbiamo testato che effettivamente il server rispondesse adeguatamente alle varie richieste, che le lobby venissero costruite correttamente e che non venissero utilizzati numeri di porta o chiavi ripetuti.

7.3 Testing con utenti

Oltre ai vari test eseguiti per controllare il corretto funzionamento dell'applicativo abbiamo anche giocato diverse partite. Per controllare il corretto funzionamento del gioco fare riferimento al seguente video su yt: `https://www.youtube.com/watch?v=kU-8GAHXokY`

8 Deployment

Per rendere più agile lo sviluppo e la distribuzione su Server della nostra applicazione, abbiamo dapprima elaborato una pipeline attraverso gli strumenti messi a disposizione da Gitlab e Gradle, tramite il file ".gitlab-cy.yml" e poi sviluppato un container **Docker** per pacchettizzare la nostra applicazione.

8.1 CI/CD

L'utilizzo di Gitlab ci ha permesso di utilizzare il metodologia della CI/CD, ovvero Continuous Integration and Continuous Development/Distribution, per velocizzare quelle che erano le fasi cruciali dello sviluppo del progetto.
In particolare, rilasciando continuamente nuovi aggiornamenti, aggiornare ogni volta il codice presente nel Server avrebbe richiesto molto tempo. Per ovviare questo problema, abbiamo cercato di utilizzare la pipeline Gitlab per aggiornare automaticamente nel Server il codice ad ogni rilascio di nuove funzionalità.

Nella pipeline abbiamo specificato tre fasi:

- Build, dove il progetto viene compilato
- Test, dove vengono eseguiti i test
- Deploy, dove viene fatto partire il server fisico dove risiede l'applicazione

In particolare, abbiamo deciso di utilizzare l'immagine di Alpine come immagine di base per la pipeline e Docker. In entrambi i casi avere una immagine leggera, veloce ed efficiente nell'utilizzo delle risorse è stato d'aiuto nel rendere davvero agili le operazioni di deployment.
Alpine è una distribuzione Linux nata come derivazione della distribuzione LEAF, ovvero un sistema pensato per essere eseguito solamente in apparecchi di rete (router, switch e firewall) e quindi con caratteristiche di hardware molto modeste se paragonate a un qualsiasi dispositivo a disposizione di chiunque.
Nel seguente frammento di bash si può notare come un semplice comando bash abbia tempi anche tre volte superiori se eseguito in un container con in esecuzione Ubuntu, rispetto ad Alpine.

```
$ time docker run --rm -it ubuntu sh -c "apt-get  update && apt-get install -y gcc"
...
real 0m 12.66s
user 0m 0.03s
sys 0m 0.01s

$ time docker run --rm -it alpine sh -c "apk add --no-cache gcc"
...
real 0m 4.68s
```

```
user 0m 0.03s
sys 0m 0.00s
```

Nello script di deployment, i passaggi più importanti risiedono nella terza fase, cioè quella di deploy. In questa fase lo script deve comunicare con la macchina remota attraverso SSH in modo da eseguire uno script su macchina remota che sostanzialmente si occupa di aggiornare i file presenti aggiornando la repository.

```
image: gradle:7.1.1-jdk11

build:
  stage: build
  script: gradle build

test:
  stage: test
  script:
    - gradle :Server:test
    - gradle :Client:test

deploy:
  stage: deploy
  image: registry.gitlab.com/gitlab-org/cloud-deploy/aws-base:latest
  script: aws ec2 start-instances --instance-ids i-0b71ec63209a0bcb1
```

8.2 Containerizzazione

Dopo avere costruito la pipeline su Gitlab, si è pensato allo sviluppo del container Docker dell'applicazione.
Per poter eseguire un'applicazione in un container Docker è necessario racchiudere i file di cui l'applicazione necessita, come librerie, file di configurazione ed eseguibili, in una immagine, attraverso un Dockerfile.
Un dockerfile è un file che contiene tutti i comandi utili ad assemblare un'immagine. Una volta completata la scrittura del Dockerfile, è necessario compilare l'immagine attraverso il seguente comando:

```
docker build -t "nomeImmagine" .
```

Questo comando specifica al Docker Engine di compilare l'immagine presente nel Dockerfile della cartella corrente "." e di etichettare l'immagine con il nome presente dopo l'opzione "-t".

Mentre per mandare in esecuzione il container corrispondente è necessario eseguire la seguente riga:

```
docker run -d --network host "nomeImmagine"
```

Ciò permette di eseguire un immagine, "nomeImmagine" in un container specificando che la stessa deve utilizzare lo stesso stack di rete del client sul quale è in esecuzione (–network host).
Questa opzione pone alcuni problemi in termini di sicurezza, poiché fornisce al container l'autorizzazione ad accedere a processi importanti del client e deve pertanto essere utilizzata con attenzione. Essendo però questo un progetto non esposto su Internet abbiamo valutato che le configurazioni aggiuntive avrebbero complicato il progetto e si è quindi deciso di non apportare queste modifiche di sicurezza.

L'opzione -d indica di eseguire il container in modalità detached, ovvero in background rispetto al terminale dal quale viene lanciato.

```
FROM openjdk:12-jdk-alpine

#INSTALL GIT AND UNZIP FOR GRADLE
RUN apk add --no-cache git && \
    apk add --no-cache unzip

#INSTALL AND CONFIGURE GRADLE
RUN wget https://services.gradle.org/distributions/gradle-6.7.1-bin.zip
RUN mkdir /opt/gradle
RUN unzip -d /opt/gradle gradle-6.7.1-bin.zip
ENV PATH "$PATH:/opt/gradle/gradle-6.7.1/bin"

#CLONE PROJECT REPO
RUN git clone https://davide.schiaroli2:"***"@gitlab.com/pika-lab/courses/ds/projects/ds-

#GO INSIDE SERVER FOLDER, BUILD AND RUN GRADLE PROJ
WORKDIR /ds-project-rossi-schiaroli-tronetti-ay2021/

RUN gradle build

CMD ["gradle" , ":server:run"]
```

9 Esempio di utilizzo

9.1 Requisiti

Per eseguire l'applicazione è necessario avere installato una versione del JDK Java ≥ 11 e il DVCS Git.(opzionalmente è possibile eseguire l'applicazione utilizzando Gradle 7).

9.2 Comandi

Per lanciare l'applicazione in ambiente Unix è necessario eseguire i seguenti comandi:

```
git clone {repository}
cd {repository}
gradle run
```

Si suppone che un server in ascolto sia già in esecuzione, se non lo fosse è possibile crearlo eseguendo:

```
cd {repository}
gradle :Server:run
```

10 Conclusioni

Il progetto si è rivelato un po' più dispendioso a livello di tempo del previsto. La maggior parte delle difficoltà è stata causata dalla scarsità di documentazione del framework utilizzato lato server, VertX. Siamo comunque riusciti ad ottenere un risultato più che soddisfacente, conforme alle aspettative, fluido e leggero.

Di seguito possiamo trovare le varie fasi di lavoro, seguite da delle brevi considerazioni su ciò che abbiamo imparato e sui lavori futuri.

Il lavoro svolto è stato caratterizzato da una prima parte di brainstorming sulla possibile logica del gioco e gestione delle risorse. Una volta scelto il paradigma da utilizzare e l'architettura su cui far girare l'intero progetto c'è stata una fase un po' più implementativa, mirata alla creazione del design dell'applicativo.
In prima battuta è stato sviluppato il diagramma di sequenza, che ci ha permesso di avere una più chiara idea sulle dinamiche dell'applicativo e di capire meglio quali componenti entravano in gioco, dandoci anche una prima idea sullo stack tecnologico che avremmo potuto utilizzare (è qui che abbiamo iniziato a pensare di poter sfruttare le potenzialità di VertX e le websocket).

Ultimata la questa modellazione si è pensato all'activity diagram, che ben si prestava ad esprimere più nello specifico il comportamento del programma per poi passare al diagramma delle classi.

Dopo questa prima fase svolta in gruppo si è passati all'implementazione vera e propria.
Ogni componente del gruppo ha portato avanti singolarmente una parte del progetto e più nello specifico il lavoro è stato distribuito come segue:

- Luca Rossi: sviluppo della logica di gioco lato client (gestione della lobby, round, player, input e output) e visualizzazione e modellazione della chat.
- Davide Schiaroli: sviluppo del server, creazione delle lobby, creazione pipeline Gradle e container Docker.
- Elisa Tronetti: sviluppo lato client dello scambio di messaggi con il server e implementazione in collaborazione con Luca Rossi del Controller.

Ovviamente la suddivisione sopra descritta è stata influenzata da scelte condivise e ci si è ritrovati spesso a lavorare insieme per parti che si intersecavano o avevano bisogno di relazionarsi col codice scritto dagli altri componenti del gruppo.

Una volta terminati i lavori e ottenuto un prototipo funzionante, già testato sotto i punti che ritenevamo fondamentali, sono state giocate alcune partite per cercare di trovare bug o falle nel sistema implementato, risolvendole ogni qualvolta venivano trovate.

10.1 Lavori futuri

Di seguito verranno elencati alcune tra le possibili migliorie e lavori futuri.

10.1.1 Interfaccia grafica

Per rendere l'applicativo effettivamente utilizzabile dall'utente medio il primo passo da eseguire sarebbe la creazione di una interfaccia grafica apposita dove poter giocare.

10.1.2 Aggiunta di funzionalità

Si potrebbero aggiungere nuovi set di regole per dare varietà al gioco o far scegliere dei subset di carte.

10.1.3 Peer to peer

Si potrebbe anche pensare di decentralizzare ancora di più il gioco rendendolo totalmente peer to peer.
Ciò aggiungerebbe più complessità lato client e pur essendo abbastanza semplice da convertire in quel formato, andrebbero aggiunti numerosi controlli e protocolli per garantire una stabilità accettabile del sistema che in questo momento è invece assicurata dal server centrale. Banalmente si dovrebbe decidere come mandare i messaggi, in che modo eleggere leader, come comportarsi in caso di fault ecc.

10.1.4 Orchestrazione container

Un problema noto della nostra applicazione è la sua mancanza di scalabilità e centralizzazione a livello di Server.
Se è vero che il server si occupa solo di rispondere alla prima richiesta di una applicazione, non c'è modo per lo stesso di rispondere a molte richieste di ingresso in lobby o creazione di una nuova lobby nello stesso momento. Questo può portare ad una congestione dello stesso, aumentando i tempi di attesa.

Una possibile soluzione a questo problema, sarebbe stata la creazione tramite strumenti come Kubernetes di cluster di container, che automaticamente e senza necessità di controllo manuale, avrebbero potuto scalare il server e renderlo disponibile ad un livello crescente di richieste senza rallentare il flusso di richieste.
Questa metodologia di lavoro, chiamata orchestrazione dei server, insieme al lavoro di un Load Balancer avrebbe reso il server ad alta disponibilità. Il lavoro però che richiede questo tipo di configurazione, unito al fatto che non esistano soluzione completamente gratuite ed adatte al tipo di progetto, ci hanno fatto desistere dalla sua attuazione.

10.2 Cosa abbiamo imparato

Il lavoro svolto ci ha aiutato a comprendere meglio le logiche, i problemi e i vantaggi di un'architettura distribuita.

Singolarmente ci ha anche permesso di migliorare nella programmazione e nell'utilizzo di tecniche che prima ci erano sconosciute, come ad esempio l'utilizzo di Runnable, ExecutorService, Promises, strutturazione del lavoro lato server.
Abbiamo anche capito quanto sia importante fare test e farli completi ed esaustivi; ci siamo infatti resi conto che molti aspetti, se testati in maniera più approfondita fin da subito ci avrebbero permesso di risparmiare molto tempo in fase di sviluppo.

References

[1] Class httpclient.

[2] Definizione di mvc.

[3] Docker Docs. Docker reference documentation, 2021.

[4] A. S. Gillis. Thick client (fat client), 2020.