Autonomous Distributed Blackjack

Sistemi Distribuiti

Lorenzo Simoncini - 0000931899
{lorenzo.simoncini2@studio.unibo.it}

Maichol Dadi - 0000921828
{maichol.dadi@studio.unibo.it}

Aprile 2022

Indice

Capitolo 1

Introduzione

Lo scopo del progetto è la realizzazione di una versione autonoma e distribuita del gioco di carte blackjack. Ogni giocatore parteciperà alla partita in maniera autonoma, scegliendo quali mosse effettuare nel proprio turno per vincere la partita.
Il sistema verrà realizzato con un'architettura client-server mediante l'utilizzo di WebSocket per la gestione della comunicazione tra le parti, con l'implementazione di un timeout che vada ad escludere eventuali giocatori che per qualsiasi motivo dovessero interrompere la comunicazione con il server.

1.1 Regole di gioco

Il Blackjack è un gioco che impiega le carte da gioco francesi; queste si compongono di quattro semi differenti (Cuori, Quadri, Picche e Fiori) e per ogni seme sono presenti tredici carte (Asso, i numeri dal 2 al 10, Jack, Queen e King).

Figura 1.1: Le carte da gioco utilizzate

Il gioco si svolge tra il banco/dealer e diversi giocatori; essi ricevono delle carte dal dealer e possono richiederne altre nel corso della partita, a turno. Lo scopo dei partecipanti è avere delle carte il cui valore complessivo è maggiore di quello del dealer ma minore o uguale a ventuno. In alternativa, possono vincere con un punteggio più basso di ventidue quando il valore della mano del dealer supera ventuno. Ogni carta numerata ha il relativo valore, le carte "figura" (Jack, Queen e King) valgono dieci e l'Asso può valere sia uno che undici (ma solo un Asso per mano può avere il secondo valore).
Prima di ogni mano, i giocatori puntano un certo numero di fiches scelto da loro che deciderà poi il guadagno di vittoria; qualora le carte di un giocatore superino il valore ventidue, egli

avrà "sballato", con conseguente perdita delle fiches puntate.
All'inizio di ogni mano il dealer consegna due carte ad ogni giocatore e due a sè stesso, di cui una nascosta ai partecipanti; all'inizio del proprio turno ogni giocatore può:

- richiedere una carta (**HIT**): il giocatore può eseguire questa azione fino a che il valore delle sue carte sia minore di ventuno;
- richiedere una carta e terminare il proprio turno (**DOUBLE**): il giocatore può eseguire questa azione una sola volta per turno se il valore delle sue carte è minore di ventuno; dopo aver preso una carta in questo modo dovrà automaticamente passare il turno. Questa azione inoltre raddoppia la puntata effettuata.
- passare il turno senza chiedere carte (**STAND**): in questo il giocatore non riceverà alcuna carta dal dealer e procederà al confronto con solo le due carte ricevute inizialmente.

Dopo che tutti i giocatori avranno eseguito il proprio turno giocherà il dealer, che dovrà pescare carte fino a che il loro valore non sia maggiore o uguale a diciassette. Una volta concluso il turno del dealer egli scoprirà le sue carte e si procederà al calcolo dei risultati, che può avere diversi esiti per ogni giocatore:

- valore delle carte superiore a ventuno con conseguente perdita della puntata;
- valore delle carte inferiore a ventuno ma minore di quello del dealer, che anche in questo caso comporta la perdita della puntata;
- valore delle carte inferiore a ventuno ma maggiore di quello del dealer, con guadagno pari alla puntata eseguita;
- valore delle carte uguale a quello del dealer, con riassegnamento delle fiches puntate inizialmente;
- valore delle carte pari a ventuno (**Blackjack**), che comporta un guadagno doppio rispetto alla puntata eseguito;

Finito il calcolo dei risultati, i giocatori che hanno ancora delle fiches procedono a giocare un nuovo turno.

1.2 Struttura client-server

Per la comunicazione tra le parti che comprendono il gioco (giocatori e dealer) si è scelto di realizzare un'architettura client-server tramite Vert.x; i client dovranno connettersi al server per entrare in partita e ricevere da esso tutte le informazioni necessarie allo svolgimento del gioco e alla corretta comunicazione e sincronizzazione tra le parti. La comunicazione farà uso di WebSocket per inviare e ricevere messaggi ai giocatori connessi.

1.3 Obiettivi e requisiti

Il progetto si pone quindi i seguenti obiettivi:

- Studio ed utilizzo di Vert.x per l'implementazione della struttura Client-Server
- Studio ed utilizzo di Vert.x per l'implementazione delle WebSocket

- Implementazione di API REST per la comunicazione client-server
- Semplici test per verificare il corretto funzionamento dell'architettura e delle logiche di gioco

Il sistema risultante dovrà quindi sincronizzare le operazioni dei client per garantire che il gioco si svolga in maniera corretta e i giocatori dovranno effettuare le loro azioni in maniera efficiente e logica.

1.4 Requisiti Opzionali

Oltre a garantire il funzionamento di base del gioco, opzionalmente, qualora ce ne fosse la possibilità si prevede di implementare i seguenti requisiti opzionali:

- Possibilità per uno o più giocatori di unirsi ad una partita già iniziata;
- Introduzione di fiches e meccanismi di puntate per una esperienza di gioco più realistica.

1.5 Piano di lavoro

Il lavoro verrà suddiviso nei seguenti punti:

- Analisi dettagliata degli obiettivi del sistema e definizione dei protocolli di comunicazione e delle API da implementare
- Progettazione ed implementazione delle logiche di gioco dei giocatori che si occuperanno di giocare le partite e dell'applicazione client
- Sviluppo del server e implementazione della logica del banco
- Test di funzionamento dei sistemi realizzati

L'analisi e i test verranno effettuati da entrambi i membri del gruppo in contemporanea, mentre la parte client/giocatori sarà svolta da Maichol e la parte server/banco da Lorenzo. La suddivisione del lavoro potrebbe subire variazioni in caso di necessità o imprevisti.

Capitolo 2

Design

L'architettura dell'applicazione è composta da un Dealer (Server) e uno o più Player (Client); questi comunicano attraverso lo scambio di messaggi inviati tramite websocket e diverse API.

2.1 Flusso di gioco

Inizialmente un nuovo giocatore dovrà "sedersi al tavolo", utilizzando l'apposita API "**init**", con cui riceverà uno username univoco dal Dealer per essere identificato durante tutta la sessione di gioco. In seguito al successo di questa procedura, il player invierà una richiesta al Server per creare la sua WebSocket, utile alle comunicazioni che avvengono durante la partita.
Quando almeno un giocatore ha completato la procedura di inizializzazione il Dealer può dare inizio alla partita, mettendosi in attesa che tutti i giocatori eseguano la propria puntata; questa avviene tramite l'API "**bet**". Dopo questo passaggio il Dealer distribuirà le carte ai giocatori e inciterà il giocatore di turno ad eseguire la propria mossa.
Un giocatore durante il proprio turno può eseguire una delle azioni viste nella sezione 1.1, comunicandola al Dealer attraverso l'API "**makePlay**"; in base all'azione eseguita il Dealer risponderà nei seguenti modi:

- **HIT:** il Dealer risponde al player inviadogli la carta ricevuta, e se il nuovo valore delle sue carte non supera 20 egli può effettuare una nuova mossa;
- **STAND:** il Dealer notifica la ricezione del messaggio e passa il turno al giocatore successivo;
- **DOUBLE:** il Dealer invia al player la carta pescata e passa il turno al giocatore seguente.

Una volta che tutti i giocatori in partita avranno eseguito il proprio turno il Dealer eseguirà il suo turno e invierà a tutti le sue carte (ora scoperte) e comunicherà ad ognuno il proprio risultato e le modifiche alle sue fiches. Ogni player risponderà al server comunicando la presa visione dei risultati tramite l'API "**endRound**"; ricevuto un messaggio di questo tipo da ogni giocatore il Dealer provvederà a resettare il tavolo e ad iniziare una nuova partita.

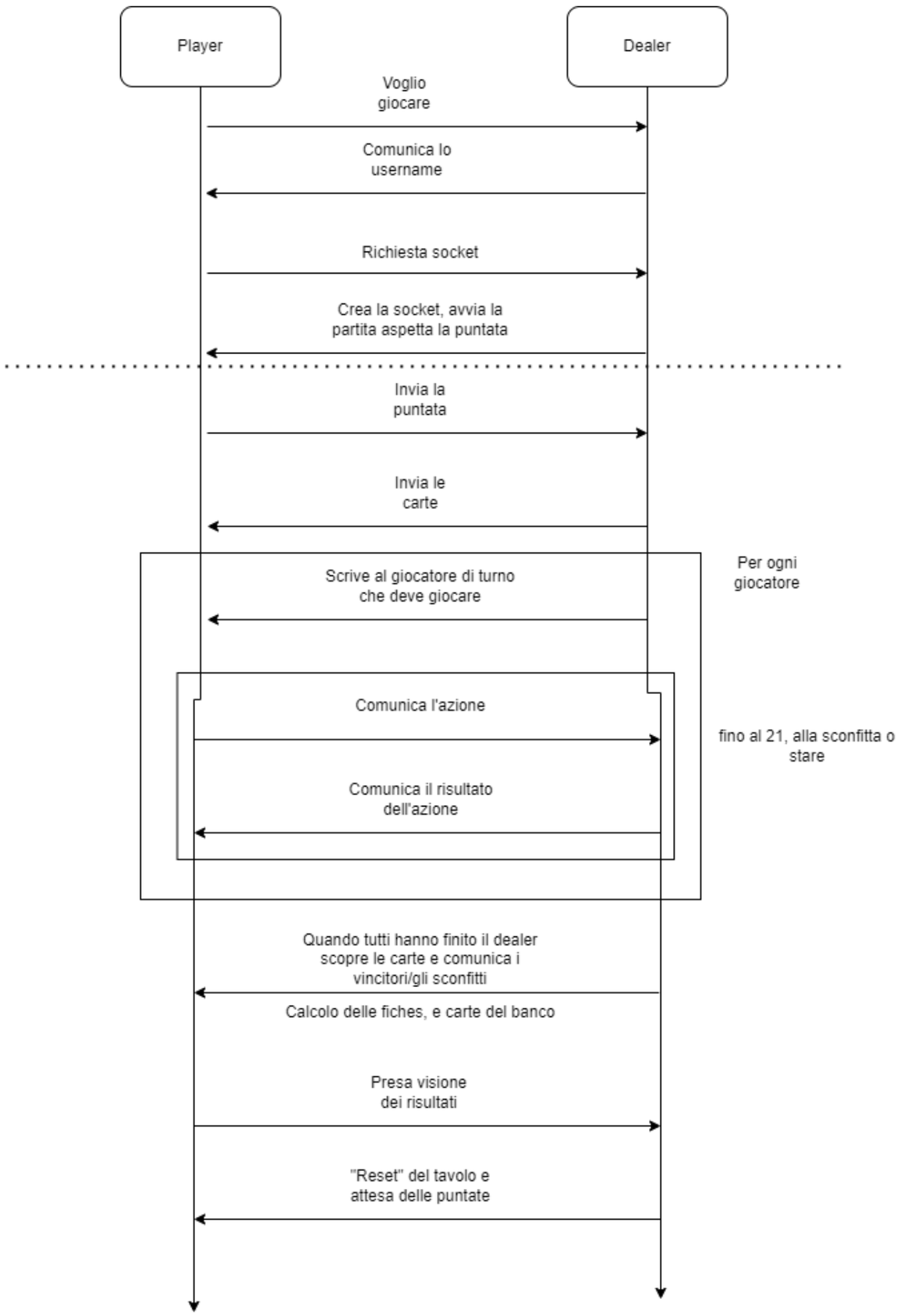

Figura 2.1: Lo scambio di messaggi che avviene tra un giocatore e il banco durante una partita

API		
Nome API	Tipo	Parametri
init	GET	---
bet	PUT	username bet
makePlay	PUT	username action
endRound	PUT	username

Figura 2.2: Le API progettate

2.2 Entrata di nuovi giocatori

Le fasi del gioco si dividono in:

- **Sessione di gioco:** inizia all'avvio del server e si conclude alla sua terminazione; si compone di diverse partite;
- **Partita:** fase di gioco che va dalla distribuzione delle carte al calcolo dei risultati; si compone di uno o più turni;
- **Turno:** il momento del gioco in cui un determinato giocatore può eseguire le sue azioni.

Nuovi giocatori possono unirsi ad una sessione di gioco in qualsiasi momento inviando una richiesta di partecipazione al server, per poi essere inseriti nella partita successiva a quella in corso. Per fare ciò, al momento della richiesta, se è già in corso una partita, il giocatore verrà inserito in una lista di attesa per poi essere aggiunto alla lista effettiva di giocatori al termine della partita corrente.

2.3 Uscita di un giocatore

Un giocatore può uscire da una sessione di gioco in diversi modi, che verranno poi gestiti di conseguenza:

- **Terminazione delle fiches:** la modalità di uscita attesa per la maggior parte dei giocatori; una volta terminate le fiches a disposizione, il giocatore verrà escluso dalla sessione;
- **Crash del client:** all'avvenire di questo evento il server dovrà escludere il giocatore dalla sessione, senza che ci siano ripercussioni sul corretto svolgimento della partita;
- **Mancata ricezione dei messaggi:** allo scattare di un timeout il messaggio relativo verrà considerato perso e il giocatore che lo ha inviato/che lo ha ricevuto verrà escluso dalla sessione.

2.4 Interfaccia grafica

Per mostrare a schermo le informazioni importanti si prevede una semplice stampa su console che mostri le carte possedute dai giocatori e le azioni che compiono (lato client) e le richieste ricevute e le carte del dealer (lato server).

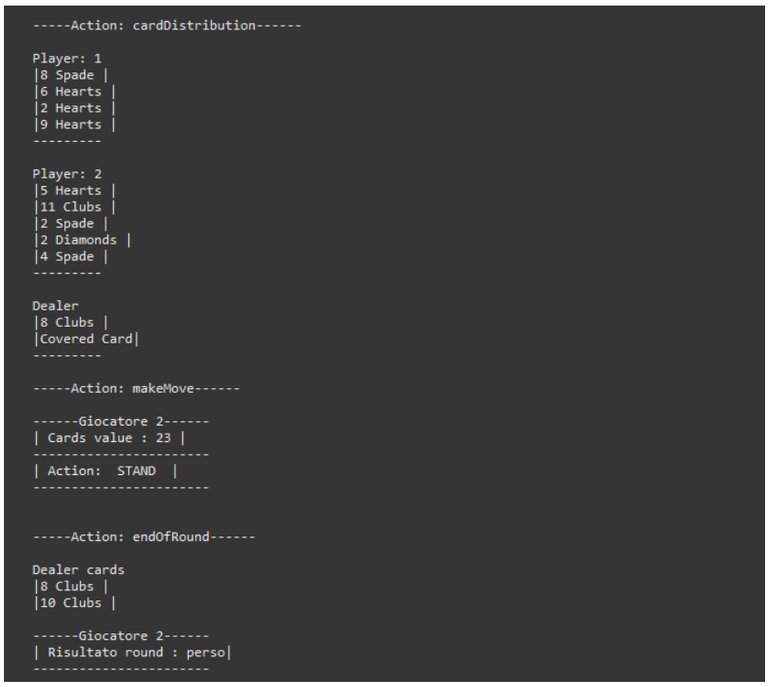

Figura 2.3: L'interfaccia a lato client

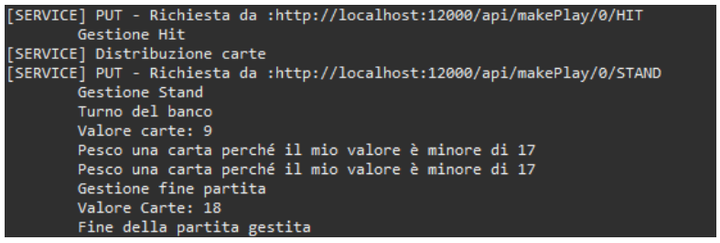

Figura 2.4: L'interfaccia a lato server

Capitolo 3

Sviluppo

3.1 Struttura del codice

Le classi implementate si dividono in tre package:

- **Server:** contiene le classi Server e Dealer; la prima si occupa di gestire la comunicazione con i vari client mentre la seconda incapsula la logica di gioco e le informazioni necessarie allo svolgimento delle partite;
- **Client:** contiene le classi App, ClientPlayer e RoutesManager; ClientPlayer rappresenta il giocatore e incapsula il suo comportamento mentre RoutesManager gestisce la comunicazione con il server.
- **Utils:** contiene varie classi ed enum utili a molte delle classi sia a lato server che a lato client;

3.2 Avvio sessione di gioco

Una volta avviato il server, la sessione di gioco avrà inizio dopo che il primo giocatore avrà inviato una richiesta tramite l'API "init".

3.2.1 Server

All'avvio del server viene istanziato un verticle che rimane in ascolto per eventuali richieste in arrivo da parte dei client; tramite il metodo setup() vengono definite le API alle quali dovrà rispondere e i metodi con i quali lo farà. Inoltre viene anche inizializzato un webSocketHandler che si occupa di gestire la comunicazione attravero le varie webSocket che verranno create.

```
1  /**
2   * Declaring routes and api handling methods
3   */
4  private void setup() {
5    // timeout for unresponding players
6    HttpServerOptions options = new HttpServerOptions().setIdleTimeout(60);
7
8    Router router = Router.router(vertx);
9    router.route().handler(BodyHandler.create());
10
11   router.get("/api/init").handler(this::handlerInit);
12   router.put("/api/bet/:username/:bet").handler(this::handlerBet);
13   router.put("/api/makePlay/:username/:action").handler(this::
       handlerPlayerAction);
14   router.put("/api/endRound/:username").handler(this::handlerEndRoundAck);
15
16   getVertx().createHttpServer().requestHandler(router).webSocketHandler(
       this::webSocketHandler).listen(localPort);
17   getVertx().createHttpServer(options).webSocketHandler(this::
       webSocketHandler).listen(8000, "localhost");
18 }
```

Ricevuta una richiesta "init" verrà chiamato il metodo handlerInit che assegnerà al giocatore uno username univoco e lo aggiungerà alla lista dei player; fatto ciò, gli invierà una risposta comunicandogli il suo username.

```
1  /**
2   * Handle a init request by a new player
3   *
4   * @param ctx, the routing context
5   */
6  private void handlerInit(RoutingContext ctx) {
7    log("GET - Richiesta da :" + ctx.request().absoluteURI());
8    String username = String.valueOf(playerList.size());
9    playerList.add(username);
10
11   HttpServerResponse response = ctx.response();
12   JsonObject obj = new JsonObject();
13   obj.put("username", username);
14
15   response.putHeader("content-type", "application/json").end(obj.
       encodePrettily());
16 }
```

Dopodichè il client invierà una richiesta di creazione della webSocket che verrà gestita tramite il metodo webSocketHandler(); dopo averla creata, il server agirà in modo differente a seconda dello stato della sessione: se non c'è nessuna partita in corso il giocatore verrà

aggiunto ad una mappa in cui sono contenuti i player attualmente in gioco e verrà inizializzata una nuova partita, altrimenti il giocatore verrà aggiunto ad una lista di attesa.

```
if (dealer == null || !dealer.isGameStarted()) {
  //if there is no game started
  webSocketMap.put(String.valueOf(playerIdCounter++), ws);
  dealer = new Dealer(playerList);
  handleStartGame();
} else {
  //else add new player to waiting list
  waitingList.add(ws);
  if (webSocketMap.isEmpty()) {
    //previous players finished playing
    ackCount = 0;
    int newPlayers = waitingList.size();
    int lastHigherId = playerIdCounter;
    waitingList.forEach((socket) -> {
      webSocketMap.put(String.valueOf(playerIdCounter++),socket);
    });
    waitingList.clear();
    dealer.addPlayers(newPlayers, lastHigherId);
    dealer.resetTurnPlayer();
    handleStartGame();
  }
}
```

3.2.2 Client

All'avvio del client egli invia una richiesta "init" tramite il metodo init; questo metodo crea una promise che viene poi passata al metodo sendInit di RoutesManager, che si occupa di inviare la richiesta al server. Al completamento della promise il client stamperà a schermo il suo username appena ricevuto.

```
public void init() {
  Promise<JsonObject> promise = Promise.promise();
  Future<JsonObject> future = promise.future();
  routesManager.sendInit(promise);
  future.onSuccess(body -> {
    username = body.getString("username");
    System.out.println(username);
    createSocket();
  });
}
```

```
public void sendInit(Promise<JsonObject> promise) {
  webClient.get(12000, "localhost", "/api/init").send(ar -> {
    if (ar.succeeded()) {
      HttpResponse<Buffer> response = ar.result();
      JsonObject body = response.bodyAsJsonObject();
      promise.complete(body);
    } else {
      System.out.println("Something went wrong"+ar.cause().getMessage());
      promise.fail("Something went wrong");
    }});}
```

Ricevuto lo username invierà una nuova richiesta al server per creare la websocket con il metodo createSocket in RoutesManager; viene specificato un timeout di 60 secondi per riconoscere eventuali mancanze di comunicazione.

```
public void createSocket(Promise<WebSocket> promise) {
  HttpClientOptions options = new HttpClientOptions().setIdleTimeout(60).
   setDefaultHost("localhost").setDefaultPort(8000);
  vertx.createHttpClient(options).webSocket("/api/createSocket", res -> {
    if (res.succeeded()) {
      System.out.println("connesso al socket!");
      webSocket = res.result();
      webSocket.closeHandler(Void -> {
        System.out.println("WebSocket closed");
        System.exit(0);
      });
      promise.complete(webSocket);
    } else {
      promise.fail("Something went wrong");
    }
  });
}
```

3.3 Avvio partita e puntate dei giocatori

3.3.1 Server

Una volta che almeno un giocatore è entrato nella sessione la partita ha inizio; il dealer inizializza le strutture necessarie al gioco e il server invia ad ogni giocatore connesso le proprie fiches iniziali in un messaggio di tipo ”startGame”, rimanendo in attesa delle puntate.

```
private void handleStartGame() {
  dealer.initGame();
  dealer.setBetCounter(0);

  JsonObject obj = new JsonObject();
  obj.put("msgType", "startGame");
  obj.put("startingFiches", Dealer.getStartingFiches());
  webSocketMap.forEach((id, webSocket) -> {
    log("Game started, waiting for bets");
    webSocket.writeTextMessage(obj.toString());
  });
}
```

Dopo aver ricevuto una puntata da ogni giocatore, il server può procedere alla distribuzione delle carte e all'avviso del giocatore di turno.

```
private void handlerBet(RoutingContext ctx) {
  log("PUT - Richiesta da :" + ctx.request().absoluteURI());

  String username = ctx.request().getParam("username");
  int bet = Integer.parseInt(ctx.request().getParam("bet"));
  //update fiches after bet
  dealer.updatePlayerFiches(username, bet);

  HttpServerResponse response = ctx.response();
  JsonObject obj = new JsonObject();
  obj.put("msg", "bet ricevuta");
  response.putHeader("content-type", "application/json").end(obj.
   encodePrettily());

  dealer.increaseBetCounter();
  if (dealer.getBetCounter() == webSocketMap.size()) {
    distributeCards();
    notifyTurnPlayer();
  }
}
```

3.3.2 Client

Quando il client riceve un messaggio sulla websocket per prima cosa ne controlla il "msgType", ossia una stringa di testo che specifica il tipo di messaggio e reagisce di conseguenza.

```
webSocket.frameHandler(frame -> {
  System.out.println("risposta sulla socket");
  JsonObject obj = new JsonObject(frame.textData());
  String msgType = obj.getString("msgType");

  switch (msgType) {
  case "startGame": {
    //faccio la mia puntata al server
    if (fiches == null)
      fiches = obj.getInteger("startingFiches");
    makeBet();
    break;
  }
  case "cardDistribution":
  ...
  ...
  ...
  ...
  default:
    throw new IllegalArgumentException("Unexpected value: " + msgType);
  }
});
```

Ricevuto un messaggio di tipo "startGame" il giocatore procede ad eseguire una puntata attraverso il metodo makeBet(), puntando un numero di fiches in base ad un moltiplicatore scelto casualmente tra uno e tre e stando attento a non superare le fiches a disposizione.

Una volta determinato il valore esso verrà inviato al server tramite il metodo sendBet() di RoutesManager.

```
1  private void makeBet() {
2    System.out.println("Fiches iniziali: "+ fiches);
3    Random random = new Random();
4    int betMultiplier = random.nextInt(1,4);
5    currentBet = MINIMUM_BET * betMultiplier;
6    if (currentBet > fiches)
7      currentBet = fiches;
8    fiches -= currentBet;
9    System.out.println("Punto "+ currentBet + " fiches - Rimanenti: "+
       fiches);
10
11   Promise<JsonObject> promise = Promise.promise();
12   Future<JsonObject> future = promise.future();
13   routesManager.sendBet(promise, username, currentBet);
14
15   future.onSuccess(body -> {});
16 }
```

3.4 Distribuzione delle carte, avvio dei turni e invio dell'azione

Dopo che tutti i giocatori hanno effettuato la propria puntata si procede distribuendo le carte e iniziando lo svolgimento dei turni.

3.4.1 Server

Il server attraverso il metodo distributeCards() prepara un messaggio di tipo "cardDistribution" da inviare a tutti i giocatori attualmente in partita; esso contiene le carte dei player e quelle del dealer, opportunamente nascoste (poiché i giocatori vedono solamente la prima carta del banco).

```
1  private void distributeCards() {
2    JsonObject obj = new JsonObject();
3    obj.put("msgType", "cardDistribution");
4    Map<String, Object> cardsMap = new HashMap<String, Object>();
5    //put dealer cards, true to hide cards
6    cardsMap.put("-1", convertCardList(dealer.getDealerCards(true)));
7
8    //put player cards
9    dealer.getPlayerCards().forEach((id, cards) -> {
10     List<String> cardsList = convertCardList(cards);
11     cardsMap.put(id, cardsList);
12   });
13   obj.put("cards", cardsMap);
14   webSocketMap.forEach((id, webSocket) -> {
15     log("do le carte");
16     webSocket.writeTextMessage(obj.toString());
17   });
18 }
```

Una volta distribuite le carte il giocatore di turno verrà notificato attraverso un messaggio di tipo ”makeMove”.

```
1  private void notifyTurnPlayer() {
2    JsonObject turnObj = new JsonObject();
3    turnObj.put("msgType", "makeMove");
4    Object key = webSocketMap.keySet().toArray()[dealer.
       getCurrentTurnPlayerID()];
5    webSocketMap.get(key).writeTextMessage(turnObj.toString());
6  }
```

3.4.2 Client

In seguito al ricevimento di un messaggio di tipo ”distributeCards” il client chiama il metodo updateCards(), che si occupa di aggiornare la propria lista di carte e di mostrare a schermo le mani di tutti i giocatori e del banco.

```
1  private void updateCards(JsonObject obj) {
2    playerCards.clear();
3    Map<String, Object> playerCardsMap = obj.getJsonObject("cards").getMap()
       ;
4    playerCardsMap.forEach((id, cards) -> {
5      System.out.println("id giocatore: "+ id);
6      List<String> cardsCast = (List<String>) cards;
7      if (id.equals(username)) {
8        cardsCast.forEach(card -> {
9          playerCards.add(Card.getCardFromString(card));
10       });
11       playerCards.forEach(card -> {
12         card.printFullCardName();
13       });
14     } else {
15       cardsCast.forEach(card -> {
16         Card.getCardFromString(card).printFullCardName();
17       });
18     }
19   });
20 }
```

Dopo aver ricevuto un messaggio di tipo ”makeMove” il giocatore sceglie un’azione da eseguire in base alle carte che possiede; questo comportamento è presente all’interno del metodo makePlay(). Più il valore delle sue carte si avvicina a ventuno (Blackjack) minore sarà la probabilità che il giocatore decida di prendere una carta aggiuntiva; inoltre, se il valore è compreso tra sette e quattordici ci sarà una probabilità del venticinque percento che egli decida di eseguire un’azione ”Double”.

```
1   private void makePlay() {
2     Promise<JsonObject> promise = Promise.promise();
3     Future<JsonObject> future = promise.future();
4     Actions action = Actions.STAND;
5     double decisionMultiplier = 21;
6     AtomicInteger cardValue = new AtomicInteger(0);
7     AtomicInteger acesFound = new AtomicInteger(0);
8     playerCards.forEach(card -> {
9       int value = card.getNumber();
10      if (card.getNumber() > 10)
11        value = 10;
12      if (card.getNumber() != 1)
13        cardValue.addAndGet(value);
14      else {
15        cardValue.addAndGet(11);
16        acesFound.addAndGet(1);
17      }
18    });
19    for (int i = 0; i < acesFound.get(); i++) {
20      if (cardValue.get() > 21)
21        cardValue.addAndGet(-10);
22    }
23    //probabilit  di NON prendere una carta
24    double decisionPercentage = (double) cardValue.get() / 21;
25    System.out.println("valore: "+ cardValue.get() + ", percentuale: "+
    decisionPercentage);
26    if (decisionPercentage <= 0.55) {
27      // hit, e possibile double
28      action =  calculateDouble(cardValue.get());
29      System.out.println("Faccio"+ action +"con valore "+ cardValue.get())
    ;
30    } else {
31      //usiamo quella probabilit  per calcolare lo stand
32      Random random = new Random();
33      if (decisionPercentage > random.nextDouble(0.56, 1)) {
34        action = Actions.STAND;
35        if (decisionPercentage > 1)
36          System.out.println("Ho pi  di 21, non posso fare niente");
37        else
38          System.out.println("Faccio STAND con valore "+ cardValue.get());
39      } else {
40        action = calculateDouble(cardValue.get());
41        System.out.println("Faccio "+ action +" (secondo if) con valore "+
    cardValue.get());
42      }
43    }
44    routesManager.sendAction(promise, username, action);
45    future.onSuccess(body -> {
46      System.out.println(body.getString("msg"));
47    });
48  }
```

```java
private Actions calculateDouble(int cardValue) {
  if (fiches >= currentBet*2) {
    if (cardValue > 6 && cardValue < 15) {
      Random random = new Random();
      if (random.nextInt(0,4) == 0) {
        fiches -= currentBet;
        currentBet = currentBet * 2;
        return Actions.DOUBLE;
      }
    }
  }
  return Actions.HIT;
}
```

Decisione della mossa			
Valore	Stand %	Hit %	Double %
1	0%	75%	25%
2	0%	75%	25%
3	0%	75%	25%
4	0%	75%	25%
5	0%	75%	25%
6	0%	75%	25%
7	0%	75%	25%
8	0%	75%	25%
9	0%	75%	25%
10	0%	75%	25%
11	0%	75%	25%
12	57%	32%	11%
13	62%	29%	10%
14	67%	25%	8%
15	71%	21%	7%
16	76%	18%	6%
17	81%	14%	5%
18	86%	11%	4%
19	90%	7%	2%
20	95%	4%	1%
21	100%	0%	0%

Figura 3.1: Le probabilità di effettuare ciascuna mossa in base alle carte possedute

3.5 Gestione delle mosse

La mossa inviata dal client viene ricevuta dal server, il quale la passerà al dealer per eseguire le dovute azioni di gioco.

3.5.1 Server

La mossa ricevuta tramite l'API makePlay viene gestita dal metodo handlerPlayerAction() nella classe Server; questo si occupa di chiamare il metodo handleAction() nella classe Dealer e in base al risultato dell'operazione invia un messaggio di notifica al giocatore.

```java
private void handlerPlayerAction(RoutingContext ctx) {
  log("PUT - Richiesta da :" + ctx.request().absoluteURI());

  String username = ctx.request().getParam("username");
  String action = ctx.request().getParam("Action");

  boolean hasGameEnded = false;
  ActionResults result = dealer.handleAction(username, action);

  switch (result) {
  case STAND: {
    //handling result of stand action
    System.out.println("Risultato di stand");
    HttpServerResponse response = ctx.response();
    JsonObject obj = new JsonObject();
    obj.put("msg", "Azione Stand eseguita, fine del turno");
    response.putHeader("content-type", "application/json").end(obj.
   encodePrettily());
    hasGameEnded = dealer.changeTurnPlayer();
    System.out.println("prima del break");
    break;
  }
  case PLAY: {
    //handling result of hit action
    System.out.println("Risultato di hit");
    HttpServerResponse response = ctx.response();
    JsonObject obj = new JsonObject();
    obj.put("msg", "Azione hit eseguita");
    response.putHeader("content-type", "application/json").end(obj.
   encodePrettily());
    distributeCards();
    break;
  }
  case DOUBLE: {
    //handling result of double
    System.out.println("Risultato di double");
    HttpServerResponse response = ctx.response();
    JsonObject obj = new JsonObject();
    obj.put("msg", "Azione double eseguita, fine del turno");
    response.putHeader("content-type", "application/json").end(obj.
   encodePrettily());
    distributeCards();
    hasGameEnded = dealer.changeTurnPlayer();
    break;
  }
  default:
    throw new IllegalArgumentException("Unexpected value: " + result);
  }
  handleTurnEnd(hasGameEnded);
}
```

Il metodo handleAction esegue diverse operazioni in base all'azione ricevuta:

- **HIT:** consegna una carta al giocatore; l'ActionResult è "PLAY" in quanto il giocatore può eseguire una nuova azione in base a questa mossa;
- **STAND:** restituisce al server l'ActionResult "STAND";

- **DOUBLE:** raddoppia la puntata corrente del player e gli consegna una carta; ritorna l'ActionResult "DOUBLE" poichè il giocatore riceve una carta ma non può eseguire nuove azioni.

```
1  public ActionResults handleAction(String username, String action) {
2    switch (Actions.valueOf(action)) {
3    case HIT: {
4      System.out.println("hit");
5      giveCardToPlayer(username);
6      return ActionResults.PLAY;
7    }
8    case STAND: {
9      System.out.println("stand");
10     return ActionResults.STAND;
11   }
12   case DOUBLE: {
13     System.out.println("double");
14     int currentBet = playerBets.get(username);
15     playerFiches.put(username, playerFiches.get(username) - currentBet);
16     playerBets.put(username, currentBet * 2);
17     giveCardToPlayer(username);
18     return ActionResults.DOUBLE;
19   }
20   default:
21     throw new IllegalArgumentException("Unexpected value: " + action);
22   }
23 }
```

3.6 Fine del turno

Dopo che il server ha comunicato l'esito dell'azione al giocatore, viene gestita la fine del turno:

- se il giocatore di turno può eseguire una nuova mossa oppure ci sono altri giocatori che non hanno effettuato un turno viene inviata una notifica a chi deve agire;
- se tutti i giocatori hanno concluso il loro turno viene gestita la fine della partita, con conseguente comunicazione ai player dei risultati e delle carte del banco.

3.6.1 Server

Una volta che tutti i giocatori hanno finito il proprio turno il dealer effettua la sua giocata, pescando carte fino a che il loro valore non è maggiore o uguale a diciassette; dopodiché, procede al calcolo dei risultati per ogni giocatore, inserendo poi i moltiplicatori delle puntate in una mappa che verrà poi inviata dal server a ciascun client.

```
1  private void endGame() {
2    playerMultiplier.clear();
3
4    // Sum of dealer's cards
5    System.out.println("Dealer value: " + currentCardValue);
6
7    // sum of players' cards
8    HashMap<String, Integer> playersValueMap = new HashMap<>();
9    AtomicInteger acesFound = new AtomicInteger(0);
10   playerCards.forEach((id, cards) -> {
11     playersValueMap.put(id, 0);
12     cards.forEach(card -> {
13       int cardValue = determineCardValue(card, acesFound);
14       playersValueMap.put(id, playersValueMap.get(id) + cardValue);
15     });
16     for (int i = 0; i < acesFound.get(); i++) {
17       if (playersValueMap.get(id) > 21)
18         playersValueMap.put(id, playersValueMap.get(id) - 10);
19     }
20     int playerValue = playersValueMap.get(id);
21
22     //calculate game results for each player
23     if (playerValue > 21 || (playerValue < currentCardValue &&
       currentCardValue <= 21)) {
24       System.out.println("Loss");
25       playerMultiplier.put(id, LOSS_MULTIPLIER);
26     } else if (playerValue == 21 && currentCardValue != 21) {
27       System.out.println("Blackjack");
28       playerMultiplier.put(id, BJ_MULTIPLIER);
29     } else if (playerValue > currentCardValue || currentCardValue > 21) {
30       System.out.println("Win");
31       playerMultiplier.put(id, WIN_MULTIPLIER);
32     } else {
33       System.out.println("Draw");
34       playerMultiplier.put(id, DRAW_MULTIPLIER);
35     }
36
37     int newFichesValue = playerFiches.get(id) + (playerBets.get(id) *
       playerMultiplier.get(id));
38     playerFiches.put(id, newFichesValue);
39     System.out.println("Player Value" + playerValue);
40     System.out.println("Player Fiches" + newFichesValue);
41   });
42 }
```

3.6.2 Client

Ricevuti i risultati con un messaggio di tipo "endOfRound", ogni giocatore stampa a schermo le carte del dealer (ora tutte scoperte) e il risultato della partita (vittoria, sconfitta, pareggio o blackjack).

```
1  private void endMatch(JsonObject obj) {
2    System.out.println(obj.getString("msg"));
3    System.out.println("Dealer cards: ");
4    JsonArray dealerCards = obj.getJsonArray("dealerCards");
5    dealerCards.forEach(card -> {
6      Card.getCardFromString((String) card).printFullCardName();
7    });
8
9    Map<String, Object> playersMultiplier = obj.getJsonObject("mult").getMap
       ();
10   int betMultiplier = (int) playersMultiplier.get(username);
11   showMatchResult(betMultiplier);
12   fiches += currentBet * betMultiplier;
13   System.out.println("Updated fiches value: " + fiches);
14
15   Thread.sleep(1900);
16   //dico al server che ho visto il risultato
17   Promise<JsonObject> promise = Promise.promise();
18   Future<JsonObject> future = promise.future();
19   routesManager.endRoundAck(promise, username);
20
21   future.onSuccess(body -> {
22
23     System.out.println(body.getString("msg"));
24   });
25 }
```

3.7 Sconfitta ed esclusione dei giocatori

Dopo aver comunicato ai giocatori i risultati della partita, il server rimane in attesa di un messaggio di ack da ciascuno di loro; una volta che tutti lo avranno inviato, può procedere ad iniziare una nuova partita, inserendo nella sessione i giocatori nella lista di attesa. Quando il server riceve un messaggio di questo tipo, chiama il metodo checkLoss() nella classe Dealer, che controlla se il giocatore in questione ha terminato le proprie fiches, e in caso affermativo lo rimuove dalla lista dei partecipanti e gli invia un messaggio in cui gli comunica la sconfitta, terminando il processo e chiudendo la socket, facendo così uscire il giocatore dal tavolo.

3.7.1 Server

Quando un client viene chiuso (per sconfitta o a causa di crash imprevisti) viene richiamato il closeHandler() del metodo webSocketHandler(), che si occupa di rimuovere il giocatore dalla mappa delle socket e di far sì che il gioco continui senza imprevisti; nel caso in cui un giocatore smetta di rispondere durante il proprio turno, questo metodo passa il controllo al prossimo player nella lista. Questa procedura viene eseguita anche in caso di timeout dei messaggi.

```
ws.closeHandler(Void -> {
  log("Socket closed");

  // handling player exit
  Optional<String> key = webSocketMap.entrySet()
          .stream()
          .filter(entry -> ws.equals(entry.getValue()))
          .map(Map.Entry::getKey).findFirst();
  if (key.isPresent()) {
    System.out.println("Rimuovo "+ key.get());
    int turnPlayer = dealer.getCurrentTurnPlayerID();
    if (turnPlayer >= webSocketMap.size())
      turnPlayer -= 1;
    ServerWebSocket w = (ServerWebSocket) webSocketMap.values().toArray()[
   turnPlayer];

    dealer.removePlayer(key.get());
    webSocketMap.values().remove(ws);
    if (webSocketMap.isEmpty()) {
      dealer.resetTurnPlayer();
    } else {
      if (w.equals(ws)) {
        System.out.println("   il giocatore di turno");
        boolean hasRoundEnded = dealer.changeTurnPlayer();
        handleTurnEnd(hasRoundEnded);
      } else
        System.out.println("Non    il giocatore di turno");
    }
  } else {
    System.out.println("Un player sconfitto ha chiuso la sua socket");
  }
});
```

3.8 Test

Sono stati realizzati quattro semplici test per verificare il corretto funzionamento delle logiche di gioco e della procedura di inizializzazione. Per simulare le azioni d'interesse sono state create alcune varianti "test" delle classi principali contenenti dei metodi appositi d'aiuto ai test.

3.8.1 Test 1: Init test

Questo test verifica la corretta assegnazione dello username ad un giocatore che vuole entrare in sessione; inviando la richiesta "init" ad un server appena creato (senza giocatori registrati) lo username assegnato dovrà essere "0".

3.8.2 Test 2: Distribution test

Questo test verifica che le carte distribuite dal server arrivino correttamente ai client; il giocatore effettuerà una richiesta all'API di test "testdistribution" rimanendo in attesa delle carte ricevute dal banco, che dovranno corrispondere a delle certe decise in fase di scrittura del test.

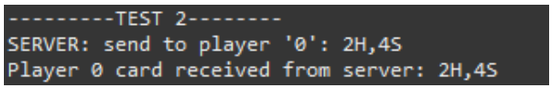

Figura 3.2: Esito del test 2

3.8.3 Test 3: Blackjack test

Questo test verifica l'assegnazione del corretto moltiplicatore punti in seguito ad un blackjack; effettuata una richiesta all'API di test "blackjacktest" il moltiplicatore ricevuto calcolato dal server dovrà essere pari a tre, in quanto la vincita dovrà essere pari al triplo della puntata eseguita.

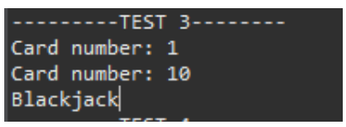

Figura 3.3: Esito del test 3

3.8.4 Test 4: Ace test

Questo test verifica il corretto calcolo del valore degli assi. Il server assegna al giocatore "0" tre carte: un asso, un otto e un dieci; il valore corretto da assegnare all'asso deve essere uno (poiché undici farebbe salire il valore totale sopra ventuno) per un valore totale di diciannove.

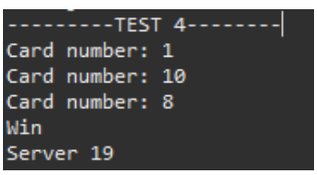

Figura 3.4: Esito del test 4

Capitolo 4

Conclusioni

Il progetto si è concluso portando a termine i requisiti decisi in fase di pianificazione, includendo anche i requisiti opzionali. Questo lavoro ci ha permesso di migliorare la comprensione dei modelli di comunicazione tra componenti distribuite, e del modo in cui strutturare le singole parti per evitare problemi di sincronizzazione. Avremmo voluto realizzare dei test strutturati più complessi e completi ma alcune limitazioni dell'integrazione di Junit con Vert.x lo hanno impedito, costringendoci a dei test più semplici riguardanti perlopiù le regole di gioco.