University of Nottingham
UK | CHINA | MALAYSIA

Dissertation Report

Multi-User Interactive Disconnection Tolerant Network

Rekop Poker

I hereby declare that this dissertation is all my own work, except as indicated in the text
Signature: J. W. Scully
Date: 13/12/2019

James Scully (14304469)
psyjs20@nottingham.ac.uk
G400 Computer Science

Project Supervisor: Milena Radenkovic

Contents

1 Introduction

Poker is a type of card game that is played and watched in tournaments by many around the world. Because of it's wide popularity, it has spawned multiple types, the most popular one being Texas Hold'em, which is the main type used in both research (A Gilpin, 2006) and is the main variant used in the World Series of Poker (WSOP, 2019). Other types include the similar yet lesser-known Omaha hold'em and Five-card draw; a simpler type of poker, which does not utilize the typical table seen in both hold'ems.

When one looks for a casual poker game that offers power to the actual end user, rather than attempting to fish money it can be quite difficult to find. From the authors own personal experience and observations with free and open software, it can be quite difficult to implement in some areas (such as games), however the ability to control a game how you would like and the ability to not have to rely on companies servers, trudge through in-application payments and ads is a unseen experience today. Some also do not allow for players with unreliable connections to continue; they are simply dropped from the game.

I believe that many of these are far too focused on the financial aspect of the game and neglect the small groups whom simply would like another medium to play poker. This is because lobbies are often not customizable and will drop players whom do not have a good enough network connection to the lobby, so that more players are able to join.

To remedy these issues, Rekop poker will be developed to handle disconnections or unstable connections by reserving their place in the lobby and the ability to join back when the player is available to. It will also not feature micro-transactions and provide the ability for players to host their own servers, with their own settings.

2 Motivation

Rekop poker aims to provide a new, alternative to other multi-player applications on the Google Play store, as many of these prevent user customization and introduce in-application purchases, with no ability for players to play locally on their own network.
Moreover, it provides a good vessel to learn more about and provide an implementation for a disconnect-tolerant network. This is because poker only allows for one player to make a choice at a given time, and if a player was to disconnect this could ruin the match for the rest of the game lobby.

3 Methodology

3.1 Test-Driven Development

Test-Driven Development has been essential in the development of the poker algorithm. This is primarily because many of the functions rely on each other, and thus modifying one can cause other functions to break. With unit tests, it was possible to run multiple cases that involve edge cases and general tests to ensure that changing code did not break the whole system, with near instantaneous feedback.
It soon became evident that whilst it was very effective at providing rapid feedback, writing tests themselves became cumbersome. For this, a factory-esque pattern of generating hands was created. Take this snippet from the testing whether a hand is a straight or not:

```
assertTrue(new TexasEvaluator("0D 0D 2D KD AD QD JD").isStraight());
assertTrue(new TexasEvaluator("AD KD QD JD 0D QD JD").isStraight());
/* Falses */
assertFalse(new TexasEvaluator("0D 0D 0D 0D AD QD JD").isStraight());
assertFalse(new TexasEvaluator("2D 5D 9D JD 0D QD JD").isStraight());
```

This allowed us to rapidly create test cases, but more importantly easily generate cases which could break the method itself via edge cases or worst case scenarios. For example, one issue that was resolved was the method would register duplicate cards as being a straight, i.e. only 4 sequential cards would be needed. Furthermore, as the overall evaluation function changed, we could run tests to ensure that all edge cases were tested as changes were made.

3.2 Agile (FINISH!)

Agile has also been the driving development cycle - primarily creating a very basic yet functional version of the software and accepting of changing requirements. For example, the back-end was created first so that the game mode itself was in an acceptable state. From here, it made it easier to visualize how the server would handle sending, receiving and processing the player's cards.

Though this helps develop a certain sector and thus the entire project succeed, as observed by the link of Agile/iterative incentive to success in (P Serrador, 2015), it can lead to over-development of one area and cause tunnel-vision, as was the case with not developing both the poker back-end and the server software at-least somewhat concurrently. For a time-constrained project like this, it is best to find a middle-ground.

4 Design

4.1 Overview

The project itself is broken down into three compartments - a backend that satisfies the core Texas Hold'em gamemode, the server which will provide the connections needed for players to play together on their own networks and the Android application, which links both of these together with the graphical interface.

The system as a whole is very server-sided, as this holds all the data instead of the clients. Appendix 7.3 shows the basic design of the system, whereby the client only receives its current hand, the ranking of their hand and the cards currently shown on the table.

For this project, we'll be developing on Android with use of standard Java libraries, though this is subject to change. Artwork will be designed with myself, with the addition of royalty-free music as needed. This will be credited.

4.2 Backend

The backend is written solely in Java as the language itself is platform agnostic - Android being reliant upon Java (bytecode). This also allows for the server to run on Linux, Windows and Mac, as per the requirements. It has been broken down in such a way that game modes other than Texas Hold'em can be added to it, enabling further development down the line.

For example, we've created data classes for Faces and Suits, each containing values for easy comparison. Furthermore, we've created a Deck class that allows for the user to pull a random card from a standard 52-card deck and ensure that it hasn't been pulled before. These building blocks allow for future development by providing the basics needed in a standard card game.

4.3 Networking

4.3.1 Introduction

The networking is based upon client-server architecture using a Thread-pool written in Java. Initially, we wished to use a Peer-to-Peer / UPnP approach, however disconnection-tolerance, whilst possible, can become difficult to implement; given that all clients would have to be constantly synchronized on the state of the host, then delegate a host if the original was to disconnect. In this case, disconnection tolerance means that a player who disconnects from the poker match will be able to reconnect at the end of the game. A new, unique player will not take their slot however, and we decided to allow the game to continue as normal, because otherwise the match could be on hold indefinitely waiting for the player to reconnect.

4.3.2 Server

For the server, we needed a way for each connection to exist independent of each other, meaning that if one was to disconnect, the others would not experience issues because of it. There were multiple options that we could have taken to design the server, including a single-threaded approach, a thread pool and a multi-process approach.
To clarify, a thread is a single line of execution - that is, a sequence of instructions separate to others - that is processed on a CPU at one time.
A process is essentially a separate program entity, meaning that instead of the program creating a separate line of execution for a client, it would ask the operating system to create and run a new process to handle the client.
With networking, it is typical that each connection is given its own thread. This is because networking code involves the retrieving and sending of data, the former blocks execution of code because generally instructions ahead rely on the said data. This means that a single-threaded approach, while would work, would not adequately show disconnection tolerance as we would only have one client; the purpose is to show that this network can cope with dropouts.
In regard to the multi-process approach, it would become cumbersome to create a new process for each connection that comes in. This is because synchronizing processes is a much more difficult task than synchronizing between threads - a process contains threads rather than vice versa. This would mean that we would need some external file or 'management' system that each process can read from.
This is how we arrived at a thread-pool approach, where we had a process containing a main thread, which accepts connections and pushes them into a worker pool.

4.3.3 Client

The client is written in Java for compatibility with the server counterpart, as they will transfer Java objects. However, it was designed such that it is not a mobile application itself but rather an interface for the application to use. This was because during development, it made unit testing the client much easier as it was ran locally on the machine. Beyond this, the author's home network was restricted, meaning that running the client and server on the same machine meant they could connect, and multiple instances of the client could be run.
In order to develop disconnection tolerance, we required that the server had some method of identifying what clients had connected and an anonymous token was generally the best way to go about this. During design, we investigated multiple avenues for this, including the server taking note of the connecting clients MAC addresses and storing them.

This wasn't suitable however, as the Java method of getting MAC addresses is difficult and not easily comparable. Another was to use the username of the client connecting, but this would lead to conflicts if another person with the same name was to join - the originally disconnected player wouldn't be allowed back in.
Instead, we decided to opt for an identity file approach, whereby upon installing the application the user has a generated key file that when connecting to servers, send a unique identifier. This way, conflicts do not occur (very unlikely, with a 32-character identifier being generated).

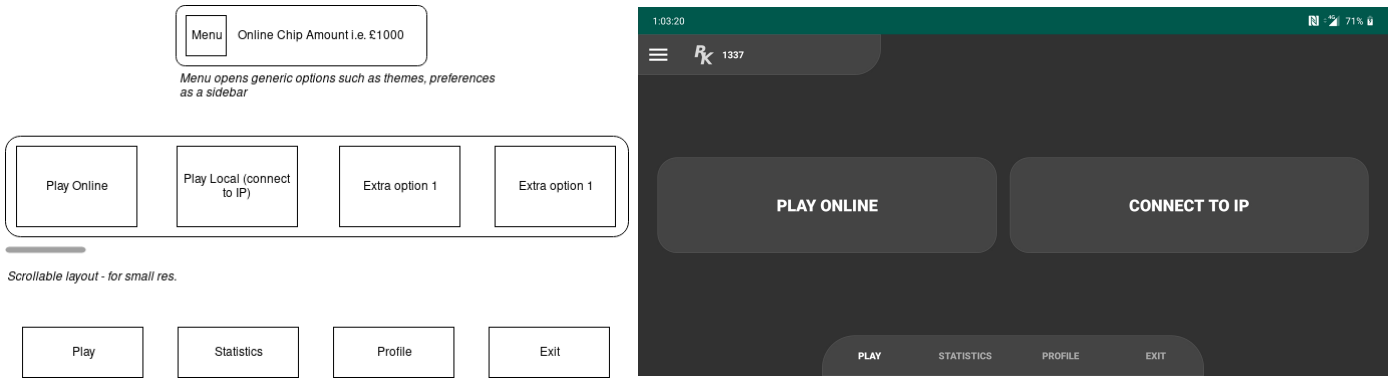
4.3.4 Communication / Diagram

In order to achieve a disconnect-tolerant network, the architecture of communication is a major factor in it being successful. What is meant by communication is that the order of sending and receiving messages from players is structured in such a way that if one was to disconnect, the server/network would not:
- wait on a player whom cannot respond - crash due to a broken socket / processing invalid data
In order to communicate with each client effectively, we needed to abstract the way that we handle connections. As previously mentioned, the architecture is important as an unexpected disconnection can cause a wide range of errors depending on the state of the current program.
This became evident during development when we kept object I/O streams within the TPokerThread class. Accessing these streams generated undefined behaviour. Whilst we have a thread pool to keep connections alive, we needed a method of flagging that a user has folded but not disconnected or vice versa, and a way to communicate certain procedures to them. In this case, an aptly-named class *Player* was created.
The Player class is handled and only used by the server, as it is the counter-part to the TPokerClient, which runs on a separate computer or device. It provides the server with necessary information such as where to write objects to or read them from, the ID of the player in order of creation and the players identity file, which acts as an authorization key, and the TPokerThread that it represents in the thread-pool.

4.4 User Interface / Android application

The most critical part and the binding element of this project is the actual mobile application itself. The user interface is to be kept simple, with the main menu being a simple strip of buttons that can be added to for extra gamemodes or features. There is a navigation bar that will allow the user to switch between playing, statistics, profile and to exit. The interface is required to be simple, as this allows for intuitive and good user experience with the application.

The following figure shows how we've changed our initial wireframe slightly. As we've implemented it into Android, we've attempted to make it appeal to certain conventions, such as using the Hamburger icon for the menu and moving it to the side of the screen.



(a) Wireframe

(b) Android sketch

Figure 1: A slightly altered production of wireframe

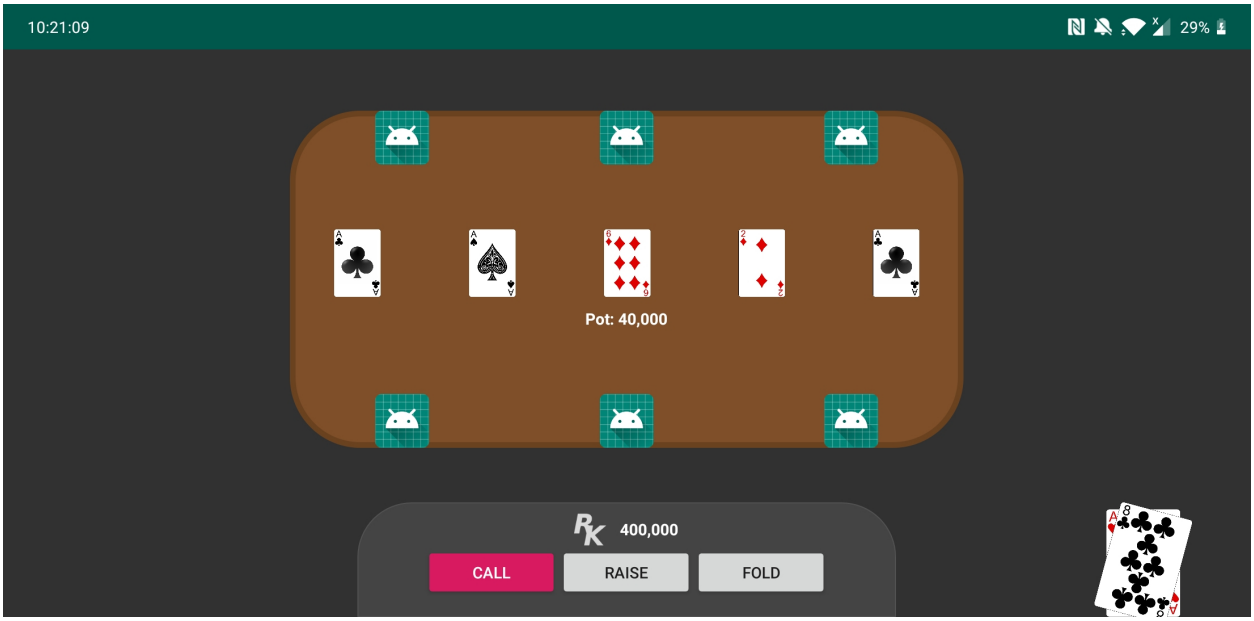


Figure 2: Android sketch of playing a match

4.5 Algorithms

Hand Strength - The algorithm for hand-strength is very basic and as such is inefficient, however this is not the main focus of the project. The base of our algorithm is in a TexasEvaluator class, whereby we test each potential result i.e. straight, flush in their own methods. We then sequentially run these methods in order of ranking, so that we can return a result.

Win evaluation - Win evaluation can be done by sorting each players hand by result, i.e. Royal Flush, Four of Kind, Straight. If each player has a unique result, then we can simply take the top result, in this case Royal Flush. If two players contest the highest result, e.g. a Straight, then we'll look at the highest card in the TResult object, which determines each players result (i.e. Three of a Kind) and the highest card.

If two players contest the highest result, but have the same highest card, then the pot is split as is the normal result in poker.

5 Implementation

5.1 Project Requirements

Our requirements for the project were not particularly specific, however some core ones were defined in our vision and scope, as well as project proposal.

5.1.1 Backend

Functional

- Programmed in Java (or Kotlin)
- Able to generate a result (i.e. 3 of kind) from the players hand and table
- Abstracts elements of card-based games (i.e. hands, cards, faces)

5.1.2 Server

Functional

- Runs on Linux and Windows
- Handles the evaluation of games internally

5.1.3 Application

Functional

- Runs on devices using Android KitKat and above
- Allows for connecting to servers via IP address
- Scalable interface for screens with low resolutions (1280 x 720)

Non-Functional

- Presents a simple, uncluttered interface for connecting/joining games
- Presents a simple game screen, with not too much clutter for game statistics
- Presents the users current chips for online matches
- Presents the users current chips within the game
- Presents a server browser for online matches

5.2 Client

5.2.1 Overview

Our initial idea for the client was to develop it as solely as an Android application. However, as we'll discuss, this made testing very difficult to do.

5.2.2 Development

5.3 Server

5.3.1 Overview

As the target of disconnection tolerance implies, the server was a crucial part of the project. It also however took a lot of effort to coordinate with the client, as any changes made to the client would need to be synchronized with the server, and so development of both had to be done somewhat in parallel.
We had multiple challenges to overcome when developing the server, including:

- Allowing multiple clients to connect
- Gracefully deal with disconnections
- Allow only those who'd previously lost connection back
- Find a suitable method for hosting the server

5.3.2 Development

The server at this moment in time is very basic and barebones - it only operates on String messages passed back between the server and the client thread and has not yet fully implemented the Texas Hold'em gamemode.

It operates upon a thread-pool, where we can have multiple connections on their own thread, allowing for multiple tasks - in this case, connections - to be processed (Oracle, 2019).

Figure 2 shows how the inner workings of the server is speculated to work. Each thread within the pool will communicate with a database or other medium of storage, as this allows for each action to be recorded.

So far, we can communicate between the client and each thread for certain actions, however the next step is to coordinate these requests between, as shown in the figure, a centralized database/storage medium, so that we can pass all actions to our backend evaluator.

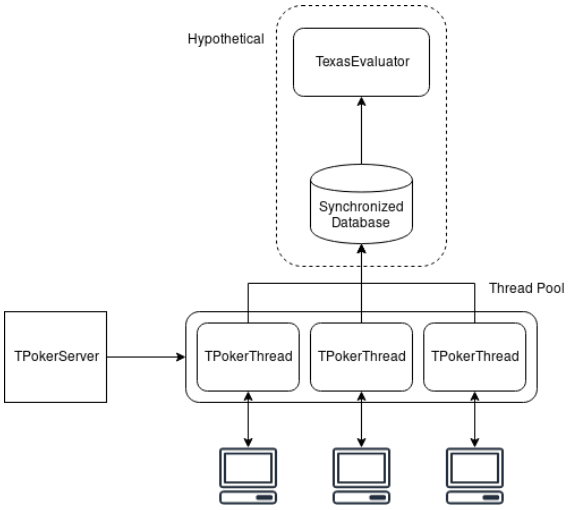
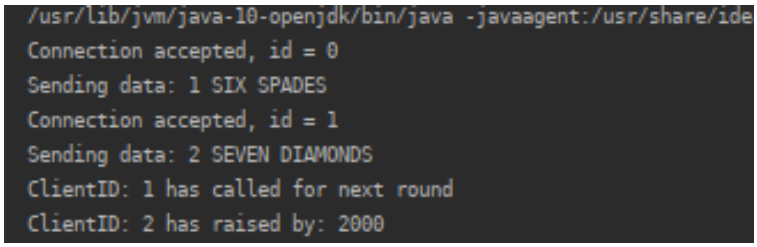
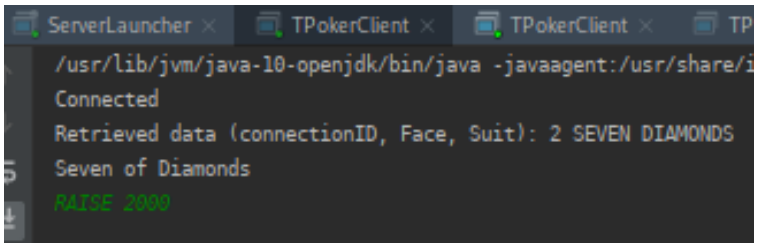


Figure 3: Current and future workings of inner server

Figure 3 shows our bi-directional communication between the client connection and its parent thread. Note that though these are simply text messages currently this is the foundation for future development, whereby we can execute necessary code on requests.
Appendix entries 7.5 and 7.6 show our methods currently for both our TPokerThread and TPoker-Client.



(a) Server receiving data from clients



(b) Client showing received enums and card, sending commands

Figure 4: Demonstration of bidirectional server-client communication

5.3.3 Challenges

The server was arguably the most challenging part of the entire project. This can be boiled down to a few reasons, the most notable being the difficulty to test the system with great accuracy or detail.
A problem that arose was synchronizing the input/output messages, that is that the client should be sending and the server should be waiting for input, the secondary being that the data type is correct i.e. we are expecting the same data type as the client is sending.

5.4 Data Classes

5.4.1 Overview

The data classes used in the project include cards comprised of faces and values, ranks including different outcomes of hands and more. When considering how we were going to design them, we needed to look at how they would be sent between the client and server, including using a object representation format such as JSON, XML or by using serializable classes between the client and server.
We decided to build up these objects from enumerated types because these are easily sent over a network if both the client and server use Java and the classes on both sides are built from the same source, using an ObjectStream.

5.4.2 Face / Suit

For the Face and Suit part of the Card class (Queen, King, Ace, etc) it was best to use an enumerated type. This is because we can easily initialize & create them, both initially on creation of pulling a new Card, but they are inherently much easier to transfer over a network due to the default and final values associated with them.

This was chosen because it simplifies reading the code, but they can also easily define other values in their constructor. Enums can easily be transferred over a network via their value name or id. The code following shows this, as we can set a custom display value, e.g. "Three" and a value for the card. Later on, we can use these values to sort hands and make determining straights and other results much easier.

```
public enum Face {

    // these can be accessed as Face.TWO, Face.ACE, etc.
    TWO   ("Two", 0),
    THREE ("Three", 1),
    FOUR  ("Four", 2),
    FIVE  ("Five", 3),
    ...
    JACK  ("Jack", 9),
    QUEEN ("Queen", 10),
    KING  ("King", 11),
    ACE   ("Ace", 12);

    private final int val;
    private final String str;

    Face(String display, int value) {
        this.str = display;
        this.val = value;
    }
}
```

Because the Card class - which will be transferred over the network most frequently - is built from enumerated classes, sending a Card class itself can easily be done by implementing Java's serializable interface. This reduced the amount of code necessary for the server-client architecture.

5.5 Evaluating Hands of Texas Hold'em

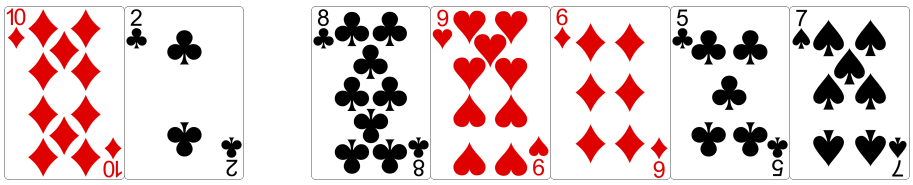
I decided it was best to have multiple methods for evaluating a hands rank as other methods of implementing this involved lookup tables or other more convoluted methods, such as representing cards as 52-bit numbers and performing bitwise operations on them. Though lookup tables are very quick to evaluate (L. Teofilo, 2013), I felt that for this project having clear and handwritten code was more suitable as this project is marked upon my own work. It also is much easier to read than bit-wise operations or complex lookup-table generation.

The evaluate method itself contains the following code:

```
public TResult evaluate() {
    // these conditions must be done in sequence, for order of rankings
    TResult kindOutput = getKinds();
    // this is required so that StraightFlushFlag is set
    TResult isStraight = isStraight();
    // variable to hold each test
    TResult result = null;
    // assignment in if statement is to remove calling method twice
    if( (result = isRoyalFlush()) != null)
        return result;
    if(StraightFlushFlag)
        return isStraight;
    // because kindOutput may be null, we need to ignore it to get past to straight
    try {
        if(kindOutput.rank == Rank.FOUR_OF_KIND)
            return kindOutput;
        if(kindOutput.rank == Rank.FULL_HOUSE)
            return kindOutput;
    } catch (NullPointerException ignored) { }
    if( (result = isFlush()) != null)
        return result;
    if(isStraight != null)
        return isStraight;
    if(kindOutput != null)
        return kindOutput;
    // the highest card will always be first as we use a sorted collection
    return new TResult(cards.get(0).face, Rank.HIGH_CARD);
}
```

5.6 Determining a Straight

Determining whether a hand is a straight would be difficult had we not assigned values to the Face values of our cards. Take the following table, whereby the first two cards are the players, the other 5 are on the table.



This hand's highest rank would be a straight, as we 10 - 5 in the hand; though they are not displayed as in order.

Take for example we start off with the first card, 10 of Diamonds. We would have to find any card on the table that is either a Jack or a 9. From here, we would then have to branch off and see if there is a card that is a Queen or an Eight, for either direction. This would have to be done for each card on the table, and optimizations would make the code more complex and introduce more areas where bugs or incorrect results can slip in.

However, since we have assigned each of the Faces values as seen previously in Section 5.2 we can use Java's Collections library to sort our cards from highest to lowest. This way, we only need to search in one direction, i.e. if the next card is lower than our current one; which is shown in Appendix 7.4. This will reduce code complexity and also the worst case scenario; we don't have to arbitrarily search the entire deck for an 'adjacent' number.

6 Progress

6.1 Management

To help me manage what needs to be done in the project and primarily drive development, GitLab issues tracker have been made for various tasks that need to be completed. These are easy to read via the use of labels and offer reminders through school email. In addition to this, branches are created for each issue to manage changes to code.

In the initial design stages, I focused on the vision and scope whereby outlining the ideal product but also the bare essentials for the project. Working under agile principles, I focused on getting a usable algorithm for evaluating poker hands as this gives us early ideas from where to go next or problems that would arise; once we had this to a good state, research for how the server will handle sending these objects or communicating became easier.

Appendix 7.1 and 7.2 show the old and the new Gantt charts. Upon developing the backend, I realised that no databases and thus database classes were needed in this part. Later on during the end stages of the servers development, databases will be used for online play, and can be removed at this stage. Therefore, server tasks aswell as implementing them into the Android application have been brought forward, as these are critical.

6.2 Contributions and Reflections

A key goal of all software is that it must be efficient - both in terms of code complexity and performance. Upon undertaking this project it was naively assumed that, given how easy it is to recognize the outcome of a poker round in reality, it wouldn't be too difficult to implement efficiently through code.

It is therefore that the algorithm currently used is very compartmentalized, and potentially slower than some implementations (such as look-up tables or doing bit-wise operations). For example, it takes between 50 - 100ms for one result to be calculated on a desktop computer. Since most servers will be running on one, it doesn't particularly matter about the performance as much as the other aspects of the program.

Reflecting upon how I assumed Test-Driven Development would take much of our time before and during up and that we did not have "too much time to write tests and develop a fully-functional program", this proved to be false. Having instant feedback on whether it passed, what the result was and the ability to debug into it was essential.

Although, initially I was under the assumption that each test case was going to have to be made up of newly created objects. Test cases wrote this way were lengthy and becoming a nuisance to write. Instead, I created a factory-esque pattern within the evaluator class itself to resolve, for example, "0D 0D 2D KD AD QD JD" into a testable hand of cards.

The example below shows how writing test cases becomes much easier through this method, rather than creating a new object for each test case.

```
TexasHand FLUSH_FOK = new TexasHand(
    new Card(Suit.CLUBS, Face.FOUR),
    new Card(Suit.CLUBS, Face.FOUR),
    new Card(Suit.CLUBS, Face.FOUR),
    new Card(Suit.CLUBS, Face.FOUR),
    new Card(Suit.CLUBS, Face.ACE)
);
...
public void isFlush() {
    assertTrue(FLUSH_FOK.isFlush());
```

```
assertTrue(new TexasEvaluator("4C 4C 4C 4
    C AC QS JC").isFlush());
```

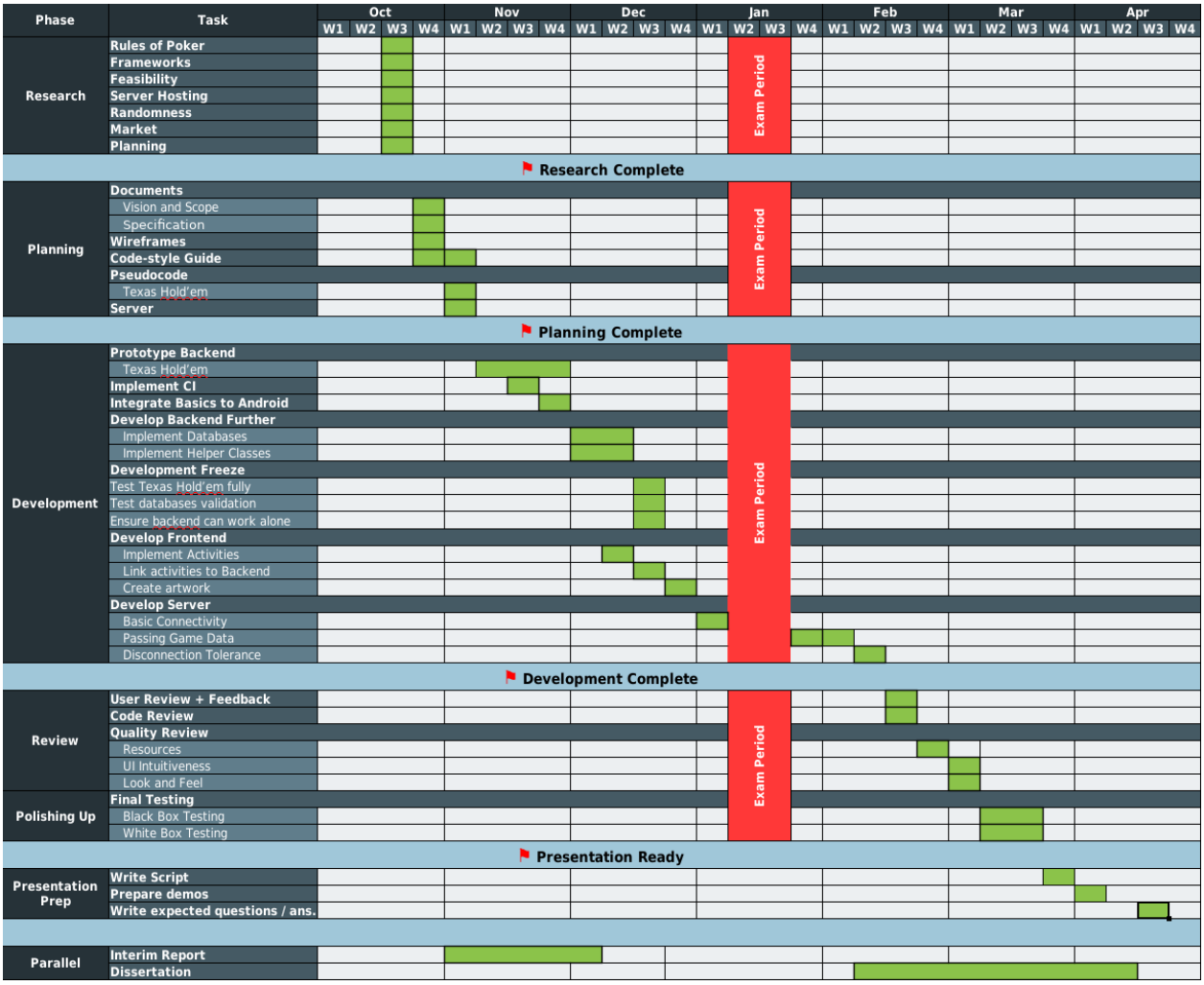
Conclusion

Currently, I am satisfied with the backend of the project as the main area of concern for this area was accuracy in determining the outcome of a game. The use of unit tests has taught me how to effectively break a problem down into small, testable blocks of code and then re-construct them into an algorithm.

However, I think that one area to work on would be time management and consequently gauging how much time tasks will take and overlooking key parts of this project; in particular the next step which is the server. This next step is not so easy to debug as it is not as easy as sequencing certain blocks of code together to create a system and most certainly not easy to debug. Therefore, I have pushed these tasks to the front of jobs to be done and allowed myself more time for these, as seen in the Gantt charts in the Appendix.

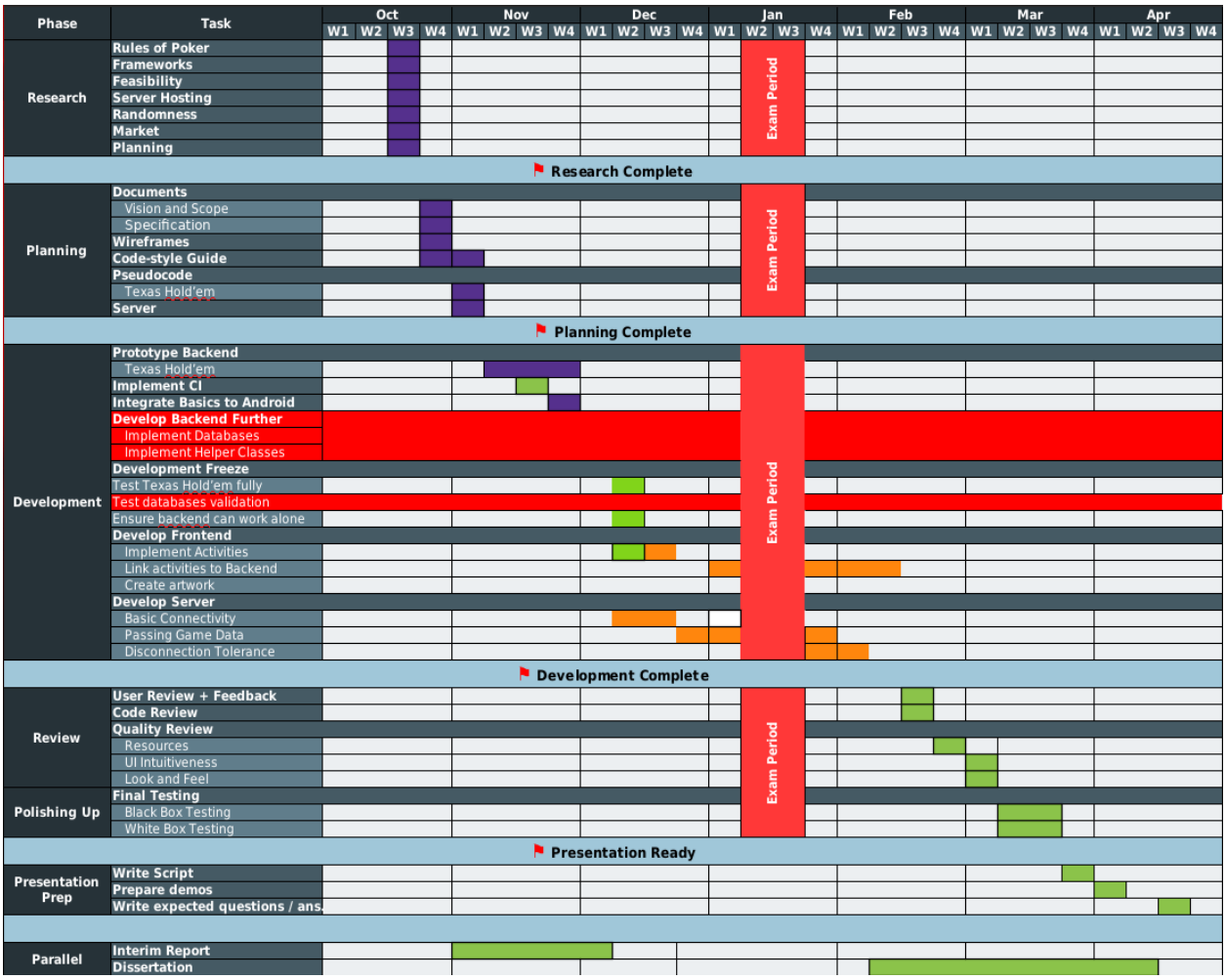
7 Appendix

7.1 Old Gantt Chart

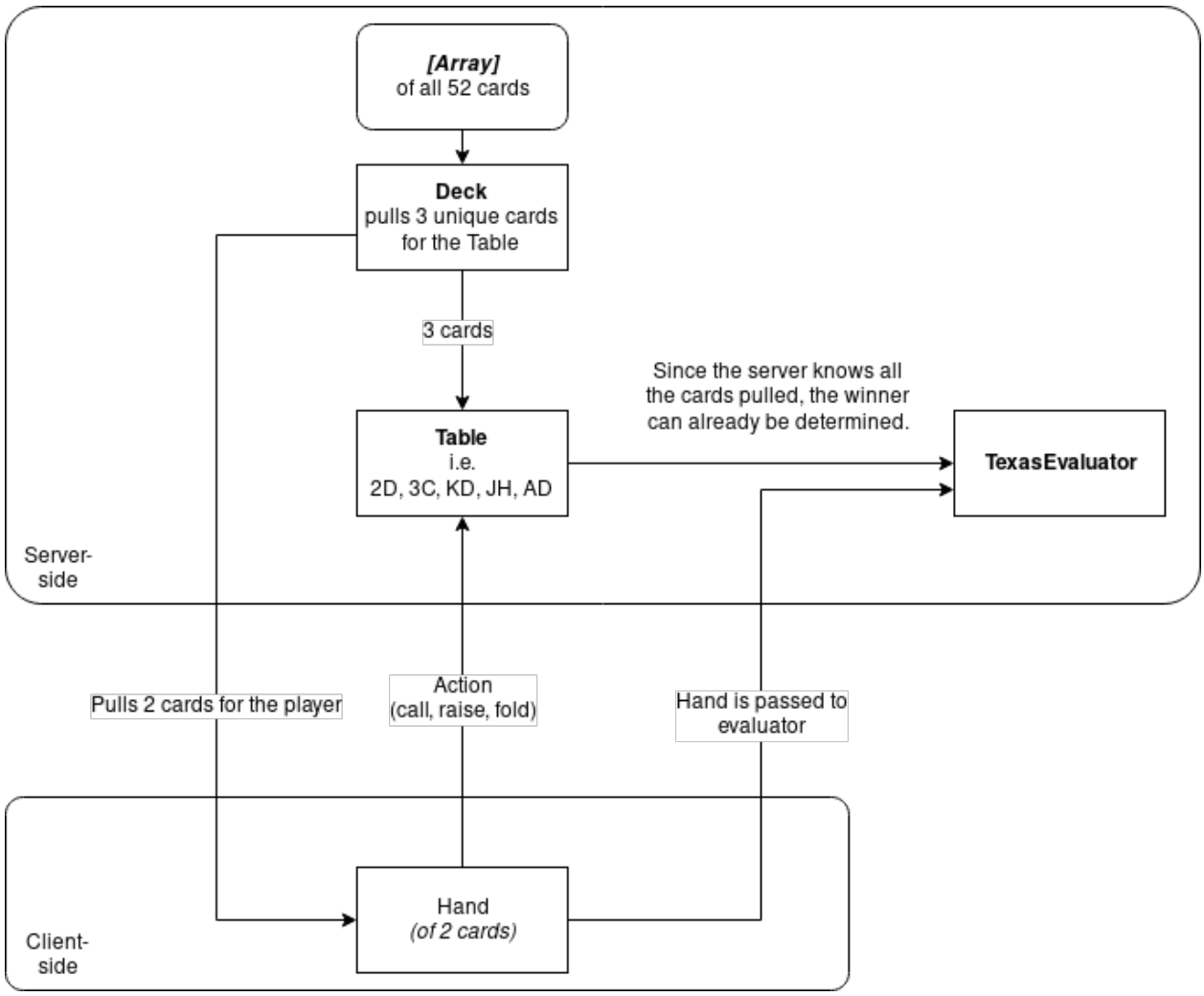


7.2 New Gantt Chart

Color chart:
Purple = completed
Green = to be completed
Orange = moved forward
Red = deleted



7.3 Abstract System Design



7.4 Straight method

All cards are sorted before this and other methods in the evaluator are called.

```
public TResult isStraight() {
    int valStreak = 0;
    int suitStreak = 0;
    int origin = 0;

    // our previous value going in should be the first in the sorted array
    // note: cards is a class member containing all cards in the evaluator,
    // sorted high to low.
    int previousVal = cards.get(0).getValue();
    Suit previousSuit   = cards.get(0).getSuit();

    for(int i = 1; i < 7; i++)  {
        // these are the attributes of card i
        Card card   = cards.get(i);
        Suit suit   = card.getSuit();
        int  value  = card.getValue();

        // if we have a previous card of same value, just skip over.
        if(previousVal == value)
            continue;

        // if the previous card was higher than the current, then add to streak
        // else, reset counter to 0.
        if(previousVal == value + 1) {
            valStreak++;

            if(suit == previousSuit)
                suitStreak++;
            else
                suitStreak = 0;

        } else {
            valStreak = 0;
            origin = i;
        }

        // if we've already managed a straight, then return true.
        // note that this should return the highest STRAIGHT, as we're descending down.
        if(valStreak == 4) {
            // this removes the need for ANOTHER function for Strt. Flushes.
            if(suitStreak == 4)
                StraightFlushFlag = true;

            Face high = cards.get(origin).face;
            Rank result = StraightFlushFlag ? Rank.STRAIGHT_FLUSH : Rank.STRAIGHT;

            return new TResult(high, result);
        }

        previousVal = value;
        previousSuit = suit;
    }
    return null;
}
```

7.5 TPokerThread run method

```
@Override
public void run() {
    try {

        DataInputStream in  = new DataInputStream(new BufferedInputStream(client.
    getInputStream()));
        ObjectOutputStream out = new ObjectOutputStream(client.getOutputStream());

// concatenate data (in this case ID), face and suit as a string
        String sendData = data + " " + face + " " + suit;
        System.out.println("Sending data: " + sendData);

// write the full data string to our client's input stream.
        out.writeUTF(sendData);
        out.flush();

        String line = "";

// read input from the user. If we receive DISCONNECT, then we close the connection
    .
// else we simply print their instruction. These requests can be forwarded to other
     components in the future.
        while(!line.equals("DISCONNECT")) {
            try {
                line = in.readUTF();

                if(line.equals("CALL")) {
                    System.out.printf("ClientID: %s has called for next round\n",
    data);
                } else if (line.equals("FOLD")) {
                    System.out.printf("ClientID: %s has folded\n", data);
                } else if (line.split("\\s+")[0].equals("RAISE")) {
                    System.out.printf("ClientID: %s has raised by: %s\n", data,
    line.split("\\s+")[1]);
                } else {
                    System.out.printf("ClientID: %s, Message: %s\n", data, line);
                }
            } catch (IOException e) {
                System.out.println(e);
            }
        }

        // close all IO connections
        client.close(); out.close(); in.close();

    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

7.6 TPokerClient main method

```
public static void main(String[] args) {
    Socket sock = null;

    DataOutputStream out = null;
    DataInputStream  in  = null;
    ObjectInputStream servOut = null;

    try {
// for now, connect to localhost (127.0.0.1) on port 1337.
// later, this will be replaced with the servers external IP address
        sock = new Socket("127.0.0.1", 1337);
        System.out.println("Connected");

        // in is our input stream, in this case command-line
        // out is the servers
        in = new DataInputStream(System.in);
        out = new DataOutputStream(sock.getOutputStream());

        // this is the connection to our TPokerThread
        servOut = new ObjectInputStream(sock.getInputStream());

        String line = (String) servOut.readUTF();

        // parse our clientID and initial card details - todo: pass two cards
        if(line == null || line.isEmpty()) {
            System.out.println("Line was null or empty");
        } else {
            System.out.println("Retrieved data (connectionID, Face, Suit): " + line
   );
            // re-build a Card object from Face and Suit enums
            getCardFromData(line);
        }

    } catch (IOException i) {
        System.out.println("Exception Caught");
        i.printStackTrace();
    }

    String line = "";

    // this will fail when we have a disconnect message, and close IO
    while(!line.equals("DISCONNECT")) {
        try {
            line = in.readLine();
            out.writeUTF(line);
        } catch (IOException e) { e.printStackTrace(); }
    }

    try {
        in.close(); out.close(); sock.close();
    } catch (IOException e) { e.printStackTrace(); }

}
```

8 Bibliography

Bibliography

[1] Pedro Serrador, Jeffrey K. Pinto
Does Agile work?—A quantitative analysis of agileproject success
Source in paragraph: *4. Results*
Retrieved from https://people.eecs.ku.edu/~saiedian/Teaching/Sp19/811/Papers/Agility/does-agile-work.pdf#s0055

[2] Luís Filipe Teófilo, Luís Paulo Reis, Henrique Lopes Cardoso
Speeding-Up Poker Game Abstraction Computation: Average Rank Strength
Source in paragraph: *2. Background*
Retrieved from https://www.aaai.org/ocs/index.php/WS/AAAIW13/paper/view/7083/6489

[3] Andrew Gilpin, Tuomas Sandholm
A Texas Hold'em poker player based on automated abstraction and real-time equilibrium computation
Source in paragraph: *2.1 Texas Hold'em*
Retrieved from https://www.cs.cmu.edu/~sandholm/texas_demo.aamas-06.pdf

[4] World Series of Poker
Champion Wins Main Event No-Limit Hold'em
Source in paragraph (uses Texas Hold'em terminology): *Claas Segebrecht Eliminated in 2nd Place*
Retrieved from https://www.wsop.com/tournaments/updates/?aid=4&grid=1628&tid=17580&dayof=8298&rr=5

[5] Oracle
Thread Pools Documentation
Retrieved from
https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html