



**University of
Nottingham**
UK | CHINA | MALAYSIA

Dissertation

Multi-User Interactive Delay-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: 4/5/2020

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

Project Supervisor: Milena Radenkovic

Contents

1	Introduction	4
2	Motivation	4
3	Methodology	5
3.1	Test-Driven Development	5
3.2	Agile (FINISH!)	5
4	Design	6
4.1	Overview	6
4.2	Backend	6
4.3	Networking	7
4.3.1	Introduction	7
4.3.2	Server	7
4.3.3	Client	7
4.3.4	Communication / Diagram	8
4.4	User Interface / Android application	9
4.5	Algorithms	10
5	Implementation	11
5.1	Project Requirements	11
5.1.1	Backend	11
5.1.2	Server	11
5.1.3	Application	11
5.2	Client	12
5.2.1	Development	12
5.3	Server	12
5.3.1	Overview	12
5.3.2	Development	12
5.4	Data Classes	15
5.4.1	Overview	15
5.4.2	Face / Suit	15
5.5	Build System	16
5.6	Evaluating Hands of Texas Hold'em	19
5.7	Determining a Straight	20
6	Evaluation	21
6.1	Unit Testing	21
6.2	Testing over a network !!	21
6.3	Testing network code !!	21
7	Summary and Reflections	21
7.1	Management	21
7.2	Contributions and Reflections	22

8	Appendix	23
8.1	Old Gantt Chart	23
8.2	New Gantt Chart	24
8.3	Abstract System Design	25
8.4	Straight method	26
8.5	TPokerThread run method	27
8.6	TPokerClient main method	28
8.7	Server-client sequence diagram	30
8.8	Poker Evaluation Algorithm	31
9	Bibliography	32

1 Introduction

Poker is a type of card game that is played and watched in tournaments by many around the world. Because of its 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.

A common problem encountered with most networks is that the client or user of a service can have a weak or 'spotty' internet connection, whereby they may lose connectivity occasionally or in certain bursts. This is especially prevalent with mobile connections, where rural or otherwise distanced areas can have trouble holding a steady connection. In these cases, it is vital that both the client is able to use the network or join as normal and that the server does not grind to a halt waiting on this person or break due to it, known as a *delay-tolerant network* (DTN).

Delay-tolerant networks have been especially important in the modern world since the advent of mobile devices and their surge in popularity. This is due to the fact that they do not have a regular stable internet connection like most desktop computers do and have the problem of losing signal due to physical movement or placement of the device. If a network cannot handle delay or dropouts from a mobile client gracefully this can have devastating impacts on data integrity or the overall function of the service.

Though the term mobile devices can insinuate the idea of mobile phones, mobile devices can range from not only a handheld phone but to vehicles, robotics, aeroplanes and satellites. Whilst data integrity to a mobile phone may not be incredibly important, data integrity is certainly incredibly important to an aeroplane or satellite, where a small error or wrong instruction can result in major failure.

The service must be able to realise that a device is either experiencing delay/dropouts and must respond accordingly; either by aiming to keep future data consistent with the previously sent data, or by accounting for the delay.

2 Motivation

Rekop poker aims to provide an open-sourced framework for the popular variant of poker, Texas Hold'em. Many existing games of poker often include microtransactions and no ability to customize the gamemode, host their own servers or modify the source code to implement other gamemodes or wanted features. Moreover, the game of poker can be used as a great vessel - especially considering mobile networks - to learn more about delay-tolerant networks, as this is becoming the most important feature of any given service on the internet as mobile devices rise in popularity.

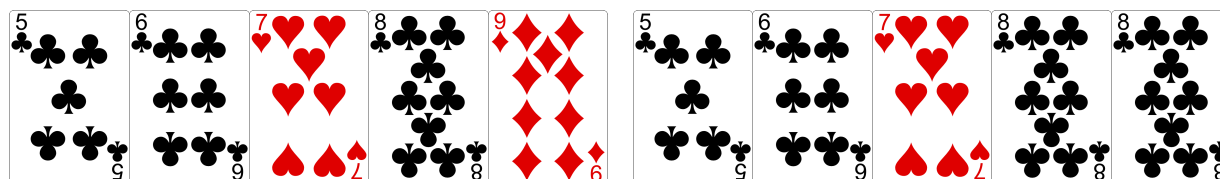
3 Methodology

3.1 Test-Driven Development

Test-Driven Development (TDD) drove the development of the poker evaluation algorithm because we can easily check what the highest suit of a hand is manually, such as if it is a straight or a flush. When developing these small functions, we could easily write test cases ahead of actual programming and then run them with each change. This had the added benefit of being able to easily write edge cases for each function to make sure that it was rigorous and we were not experiencing "bias" or writing tests to fit the current code when testing.

For example, the `isStraight` function at one point had a small case where a straight only needed 4 cards to be ascending/descending.

Note: Below show five cards, usually we use 7 (2 from player + 5 from table)



A typical straight

Two 8's - not a straight hand

Had we not taken a test-heavy approach, given that a straight itself is quite uncommon, this could have been undetected for quite some time.

However, a benefit of this that was only realised afterwards was that writing unit-testable code made the actual code-base much more modular. This has lead to the actual evaluator in turn becoming much higher quality code, as there is much less overlap between each function when determining the outcome of a hand. Appendix 8.8 shows this, as we can see that we only have functions to call rather than intertwining them.

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 two components, the back-end, which is the processing of poker hands and the representation of poker data classes, and the other component being the server, which is responsible for handling incoming connections and disconnections gracefully.

In order to handle re-connections and disconnections gracefully, it was decided to make the server handle most of the work, only sending small chunks to the clients. Appendix 7.7 shows this, with the client only receiving prompts to give their actions to the server, and receiving data on what is currently on the table / the outcome of the match.

The previously mentioned chunks are easily sent over the network as they are enumerated, meaning that they can easily be recreated on the client side with less code needed, and in the case of a delay, there is only one 'packet' to receive or send back. This was inspired by similar approaches to synchronizing clients in video games such as the original DOOM ([id Software, 1996](#)), albeit not continuous incremental changes; this means anyone joining back would be able to see all updates, not just new ones.

4.2 Backend

The backend - that is, the Texas Hold'em evaluator and other objects - is written solely in Java as the language itself is platform agnostic, which allows for the server to run on Linux, Windows and Mac, as per the requirements defined in the initial vision and scope document. This makes the client and server portable as the running platform needs only a Java 8 compatible virtual machine. It has been broken down in such a way that it is modular, by having the basics common to all other types of poker involved, such as cards, ranks, values and suits, as well as a 52-card deck with the ability to pull a guaranteed random, unique card regardless of where the code is executed (via a singleton pattern).

The backend was the most tedious to design, as it was very difficult to use traditional methods such as pseudocode for evaluation algorithms. However as we will discuss later on, unit testing drove the development through trial and error via set tests.

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 delay/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, delay 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

Threading 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. As noted by Kleiman^[7], the rise of client-server programming and multiple processors on a CPU drove the use of threads. 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. This also means that it would need to create new memory for this new client, thus a new addressing space and the inability to use variables seamlessly between processes.

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 delay 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. In initial tests, we found it very difficult to not implement a thread pool, as it was realised without it there would be difficulty in achieving delay-tolerance; single threaded approaches or ones which executed each thread were prone to breaking if connection was lost.

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.

The clients on their first launch create an identity file that is unique to them, through the generation of a random number. This identity file is used when connecting to the server as a way of determining if the player has joined before and gives the opportunity to block or prevent certain players from joining. However, it primarily is used to allow players whom have had a dropout and may have disconnected from the game to join back. In any case, certain identification of a connecting client can be used in many ways; verification, management of users, etc.

4.3.4 Communication / Diagram

In order to achieve a delay-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.

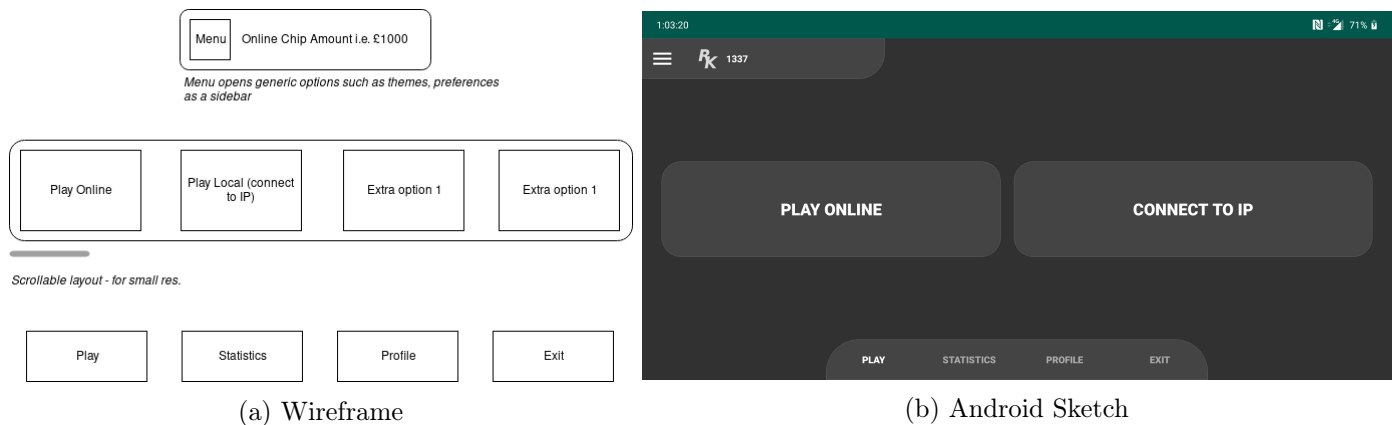
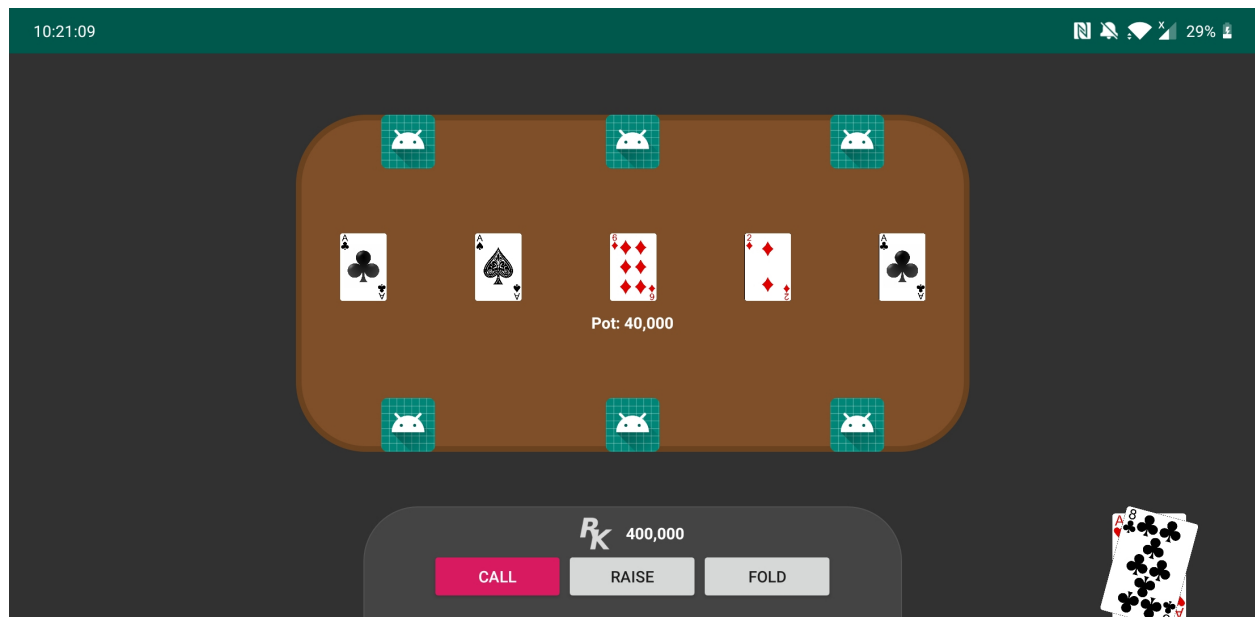


Figure 1: A slightly altered production of wireframe



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

Non-Functional

- Open-sourced
- Available for free download

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 Development

During development, we had to make multiple changes to the initial design of the client. Initially, we planned to make it solely an Android application, as this would provide a good opportunity to showcase drop outs. However, developing it as only an Android application made testing much more difficult. It was for this reason we opted to make it a terminal-based application that could be hooked into an application later on.

Because we made this choice, the ability to instantly get a client/server connection running on the development machine was trivial, as a modern computer is able to do this very quickly compared to having to compile an Android application and run it either on an emulator or even slower on a physical device, let alone having to develop a build system to compile both the server and an Android application.

Furthermore, we realised that the client would need some form of identification, because we didn't want players whom hadn't been in the game to join back; this would annoy users of the system if they were to experience a dropout and could not join. Therefore, we opted to make a file that exists locally to identify the user and is created upon first launching the program.

We arrived at this choice because the other approaches were not the most reliable or optimal. For example, we could have used the IP address of the user to identify them but using an IP address is extremely unreliable as they are very susceptible to change - especially after a dropout from the clients internet provider. We could have also used a media-access control (MAC) address to identify them - which is tied to the devices network interface card (NIC) - but this is quite difficult to retrieve from a Java perspective, some devices may or may not return one reliably.

Therefore, we decided that the most reliable option would be to use an approach that can be used on any device - be it a web browser, a mobile device or a desktop, and that would be using a file to identify the user. While many devices may not support some network functions, it is almost guaranteed that a file can be created and read anywhere Java is used.

5.3 Server

5.3.1 Overview

As the target of delay 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 is written in Java as this means we can use compatible, standard Java networking libraries to communicate with the client counterpart. Furthermore, this makes it simple to port to new platforms to host on, particularly any operating system with a Java Virtual Machine.

This platform interoperability particularly came in useful when we needed to test external devices being able to connect and thus the delay tolerance of the network, as local hosted servers are typ-

ically hard to simulate delay or dropouts. We ran into the problem of restrictive network policies at student accomodation not allowing us to host the server so that we could attempt to connect from the internet, and as such we had to find another place to host the server to test delay and unexpected dropouts.

As we were able to simply compile the server and its dependencies into a JAR file, we decided to host an Amazon Web Services EC2 instance - a Amazon-hosted Linux/Windows server - to test connections that originated over the internet. From here, we could simply push the file onto the server, configure the firewall to open a specified port, and test.

The server itself uses a thread pool to keep connections to clients alive, whereby each connection has its own thread and thus sequence of execution as previously discussed. A thread pool simply provides a way for each thread to get its time to execute (Oracle, 2019).

Furthermore, an array exists containing Player objects, which not only act as a "wrapper" for each of the connection threads but also provide connectivity data, such as if the player has unexpectedly disconnected or intentionally quit, aswell as game data such as how many chips they have and what action they last took, solving the synchronization problem we initially had.

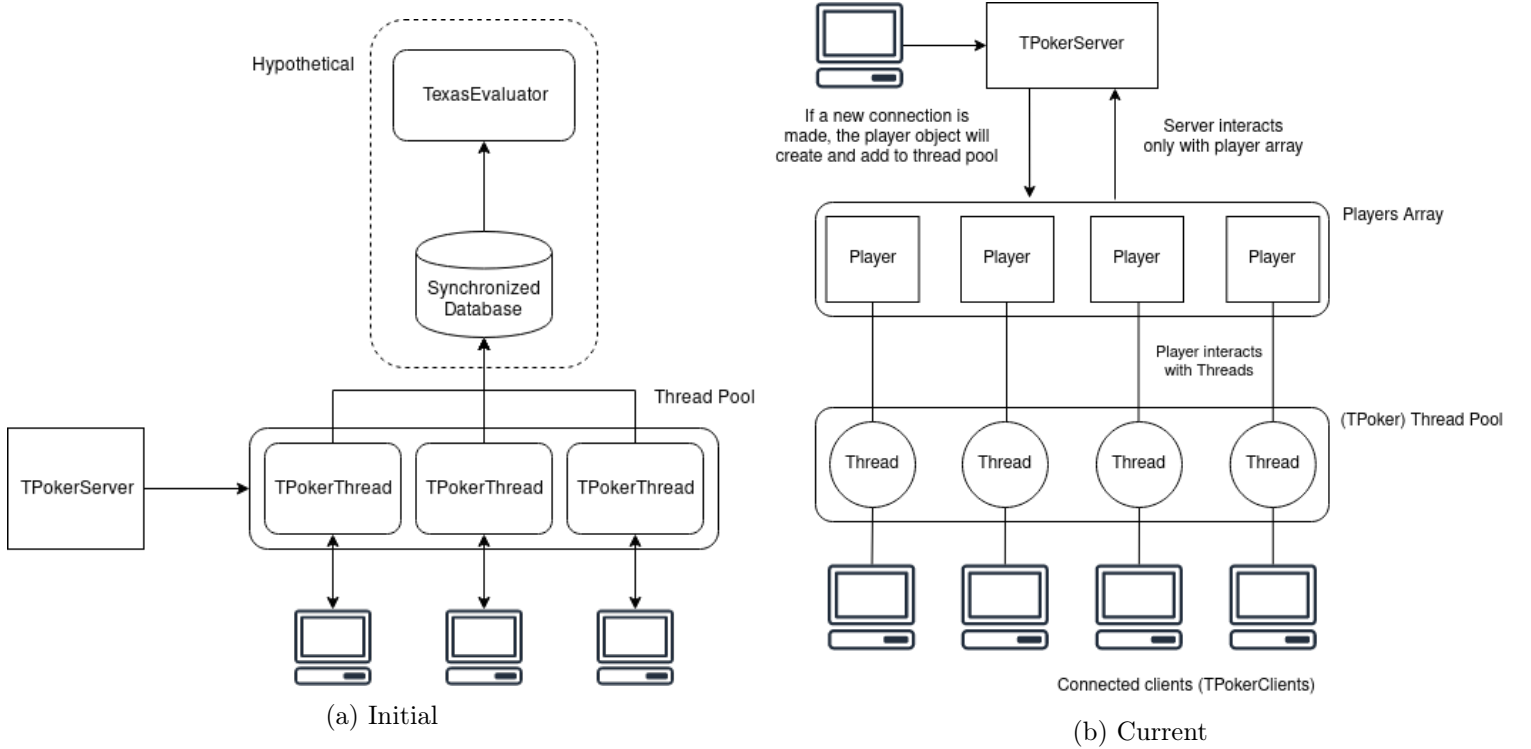
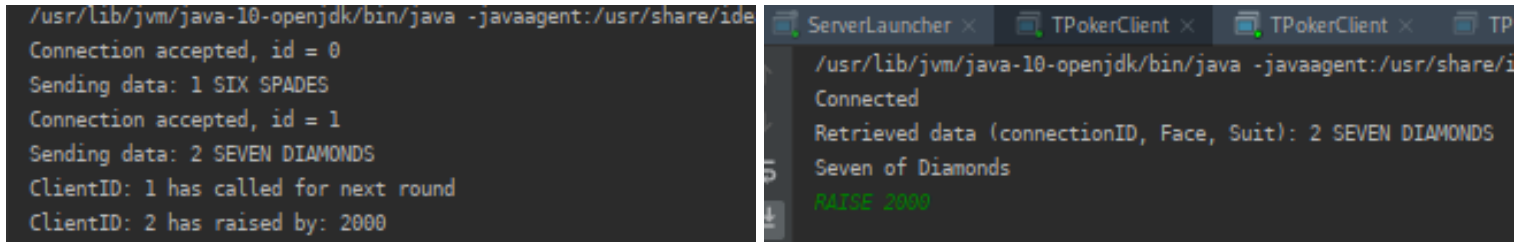


Figure 3: Synchronization of player data and threads

If we had proceeded with the initial synchronization method seen in Fig. 3(a) data integrity would also be kept, but there would still be the problem of sending messages to each thread/connection and communicating with them. Instead, the player class acts as an abstraction layer between the server and the client programs.

Furthermore, as discussed in the client section, when a player rejoins they have an identification token. If a player that has previously been in the game joins back, we can simply refresh the player object with the in/out streams and socket, then add back to the thread pool.



```
/usr/lib/jvm/java-10-openjdk/bin/java -javaagent:/usr/share/ide
Connection accepted, id = 0
Sending data: 1 SIX SPADES
Connection accepted, id = 1
Sending data: 2 SEVEN DIAMONDS
ClientID: 1 has called for next round
ClientID: 2 has raised by: 2000

ServerLauncher x TPokerClient x TPokerClient x TP
/usr/lib/jvm/java-10-openjdk/bin/java -javaagent:/usr/share/i
Connected
Retrieved data (connectionID, Face, Suit): 2 SEVEN DIAMONDS
Seven of Diamonds
RAISE 2000
```

(a) Server receiving data from clients

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

Figure 4: Demonstration of bidirectional server-client communication

Appendix entries 7.5 and 7.6 show our methods currently for both our TPokerThread and TPokerClient.

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 an 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 ObjectOutputStream.

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 Build System

During our development we initially planned to just use the built-in IntelliJ IDE build. However, we realised that the project requirements required that we open-sourced the code so that people could build upon it and make their own projects.

Therefore, we needed a build system that was reliable and would work on any system compatible with Java and the build system, as an IDE build configuration can be dependent on ones computer and preferences. This is where we decided to use Gradle as a build system, because it is cross-platform and it is somewhat a solid foundation for building projects; it's the preferred build system of popular frameworks such as Android.

The Gradle build system provided solutions and time-saving measures that we did not anticipate during the beginning. As we previously discussed in implementing the client, we needed an identity file for each client so that we could identify them. During development, we used a JAR of the client which would create its own file in whatever folder it was located, and so we used a file structure which contained players 1 - 8.

Gradle allowed us to JAR the client, and automatically deploy the JAR file into each of the environment folders.

```
// define the JAR's location
def jarToCopy = copySpec {
    from 'build/libs/texas-client-' + version + '.jar'
}

task moveToEnv {
    // for each player (p0 - 8) folder, copy the JAR to the players folder
    ['env/p1/', 'env/p2/', 'env/p3/', 'env/p4/',
     'env/p5/', 'env/p6/', 'env/p7/', 'env/p8/'].each { dest ->
        copy {
            with jarToCopy
            // this is a for-each loop: dest is 'env/pX/', where X=1-8
            into dest
        }
    }
}
```

We realised that the server would need to be in a JAR format so that we were able to distribute the software; we found that in gradle, it is possible to jar a certain class and its dependencies for use elsewhere, as shown below.

```
task servJar(type: Jar) {
    manifest {
        attributes 'Implementation-Title': 'TPoker Server',
                  'Main-Class': 'com.scully.server.ServerLauncher'
        // 'Main-Class': 'com.scully.server.TPokerClient' for packing the client
    }

    baseName = project.name + '-server'
    from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) } }
    with jar
}
```

As we will discuss later in the evaluation section, we required that the server is hosted elsewhere than the client's device, to test delay and disconnect tolerance properly in a more suitable environment. Unbeknownst to us, we found a Gradle plugin that supports us using Secure Shell (SSH) to

move certain files to a remote server.

```
remotes {
    withGroovyBuilder {
        // create our webserver connection
        "create"("webServer") {
            setProperty("host", "ec2-35-178-207-104.eu-west-2.compute.amazonaws.com")
            setProperty("user", "ubuntu")
            setProperty("identity", file('rekop.pem'))
        }
    }
}

// this is all the files we want to move, aka where the compiled JAR files are
FileTree myFileTree = fileTree(dir: 'build/libs/')

// example exec: $ ./gradlew deploy
task deploy {
    doLast {
        ssh.run {
            session(remotes.webServer) {
                // put these into the home folder of the server
                put from: myFileTree.asList(), into: '/home/ubuntu/'
                // restart the texas-server ON the server
                // this gives us the servers stdout too
                execute '~/restart_server.sh'
            }
        }
    }
}
```

This further saved time, because the workflow typically went:

1. Develop a change to server/client
2. Build from Gradle
3. Gradle will Jar the Client and Server
4. Gradle will push the new server JAR to the remote server
5. Gradle will execute a script on the AWS server to restart the server
6. Gradle will grab the terminal output of the server

Note that these can all be executed in sequence, so that we could press build gradle or a Bash script, and it would run the tests, build the necessary components, push them to the remote server, get that servers output (so we can see connections and what the server is doing) and then connect.

This was possibly the most insightful part of the project, as the power of time-consuming or complicated setup can make development and testing a breeze; on a custom Arch Linux install, we were able to launch multiple terminals spawning multiple clients, as shown below.

```
ROOT_DIR=$HOME/Dissertation/texas/env
TERM_SWITCHES="-e"
JAVA_EXEC="java -jar"
CLIENT_JAR_NAME="texas-client-1.2.1.jar"
SERVER_HOST="ec2-35-178-207-104.eu-west-2.compute.amazonaws.com"

# for p1-p8 in ROOT_DIR (env/),
for PLAYER_DIR in $(ls $ROOT_DIR)
```

```
do
    # TERMINAL = default terminal used, -e = execute command on launch
    # example cmd: alacrity -e java -jar env/p0/texas-client-1.2.1.jar 192.168.0.1
    $TERMINAL $TERM_SWITCHES $JAVA_EXEC $ROOT_DIR/$PLAYER_DIR/$CLIENT_JAR_NAME $SERVER_HOST
done

# setting exec perms (one-time necessary) + run script
$ chmod +x script.sh
$ ./script.sh
```

5.6 Evaluating Hands of Texas Hold'em

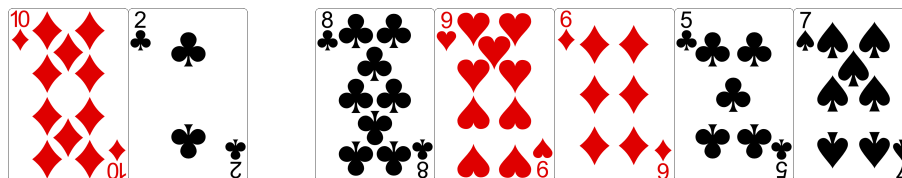
I decided it was best to have multiple methods for evaluating a hands rank as other methods of implementing this involved lookup tables or other more convoluted methods, such as representing cards as 52-bit numbers and performing bitwise operations on them. Though lookup tables are very quick to evaluate (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);
}
```

5.7 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 Evaluation

6.1 Unit Testing

Unit testing was crucial in the development of the backend, as writing and designing algorithms for this ahead of time is quite difficult due to the amount of edge cases that can crop up when there are many outcomes of a poker hand.

It also showed how testing can not just be used to verify that a program is working properly, but to drive its development forward. This constant feedback of cases that needed to be tested helped spot bugs and allowed each part of the evaluator to work together flawlessly. Because most edge cases had been ironed out in `isFlush` and `isStraight` functions, implementing whether or not a hand was a straight flush was a somewhat simple procedure.

Furthermore tests that would have been somewhat cumbersome to perform were suddenly much easier to; the Deck that we use to pull cards contains, as normal, 52 cards in an array. We needed to test that no two cards came out the same, and that pulling 52 cards resulted in an error being thrown.

6.2 Testing over a network !!

6.3 Testing network code !!

Testing network code is much more different than testing typical single-threaded, local code. Due to the blocking nature of network calls - that is, they halt execution until data has been received - when we hosted the server and client on the same computer, it became difficult even with a debugger to root down the cause of the issues as we didn't always know what the client or server was waiting for. For this,

7 Summary and Reflections

7.1 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 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.

7.2 Contributions and Reflections

Build system

Whilst not directly related to the project at hand, I believe that the build system was a massive help and achievement in its own right in driving development forward; it has shown that it is often overlooked in terms of its power and ability to save time.

It would be estimated that for each change, to manually move the server JAR file to the remote server would take a minute or two, and then to get multiple clients connected would take another. For each small change, this can add up to a lot of time wasted during setup; the addition of Gradle scripts made it almost instantaneous as to the setup and allowed us to rapidly test the network.

Amazon Web Services

Conclusion

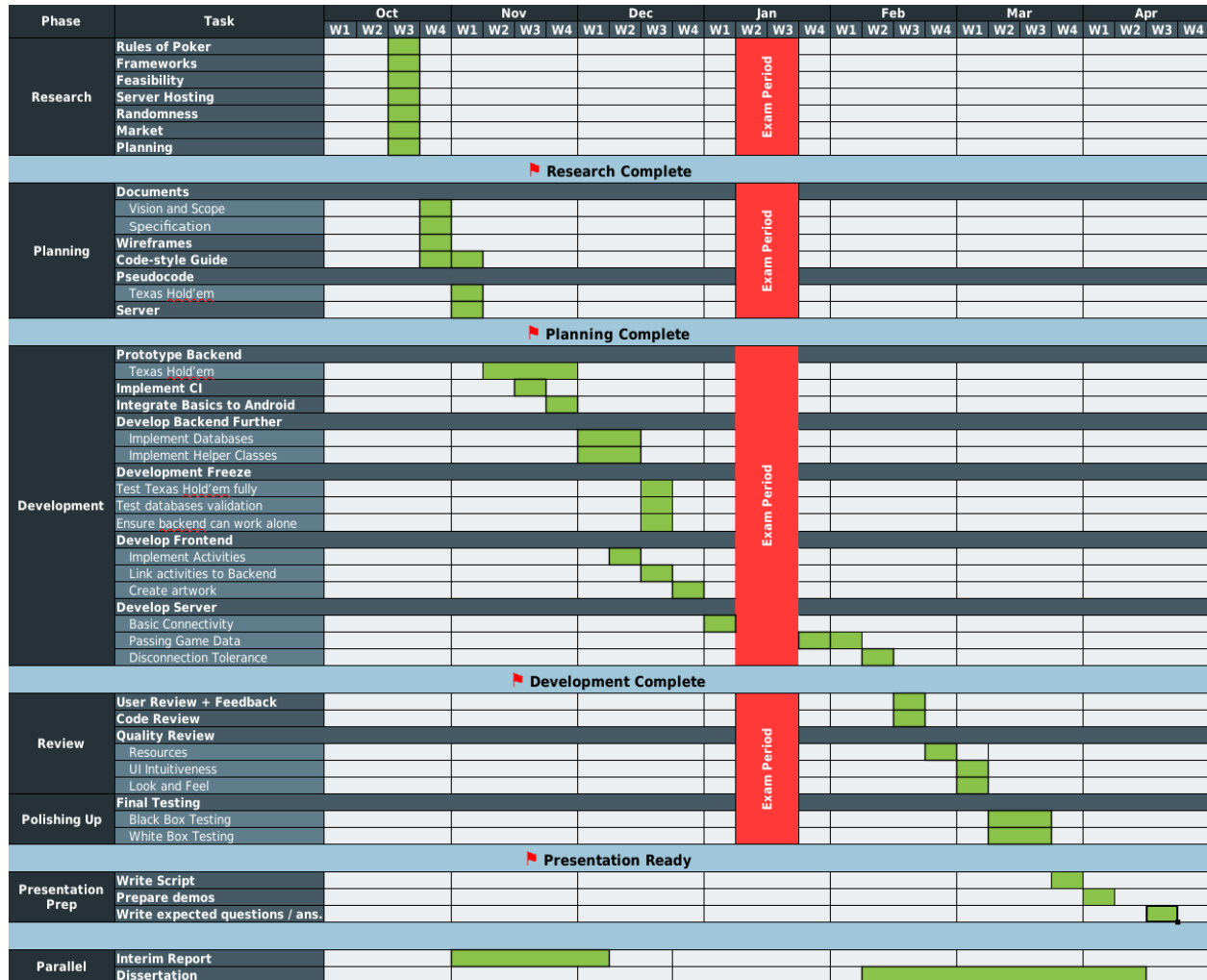
Overall, I'm quite satisfied with the personal gains made from this project, as I've learned much about not only creating a network and implementing tolerance but the development of a project overall. Prior to this undertaking I hadn't much work regarding networking or concurrency as a topic, which requires a different shift in thinking compared to sequential programming like most projects.

I think some areas could have been improved, namely implementing a suitable and fun user interface or front-end so that it could be presented to actual players but the framework to do so is somewhat there itself and could be picked up by anyone in the future. Moreover I think time management could have gone a lot better on this project, but the demands of other project-based modules on the course limited my undivided attention on this; there was a lot of context switching involved when transferring from one module back to this project.

I also think that more work could have been done on the network side to make it more robust, potentially adding in a lander thread or configuration options, so that we could set timeouts for disconnections and a method of reliably detecting that the user has disconnected rather than using a certain Java exception type to detect what has happened; which is quite flimsy.

8 Appendix

8.1 Old Gantt Chart

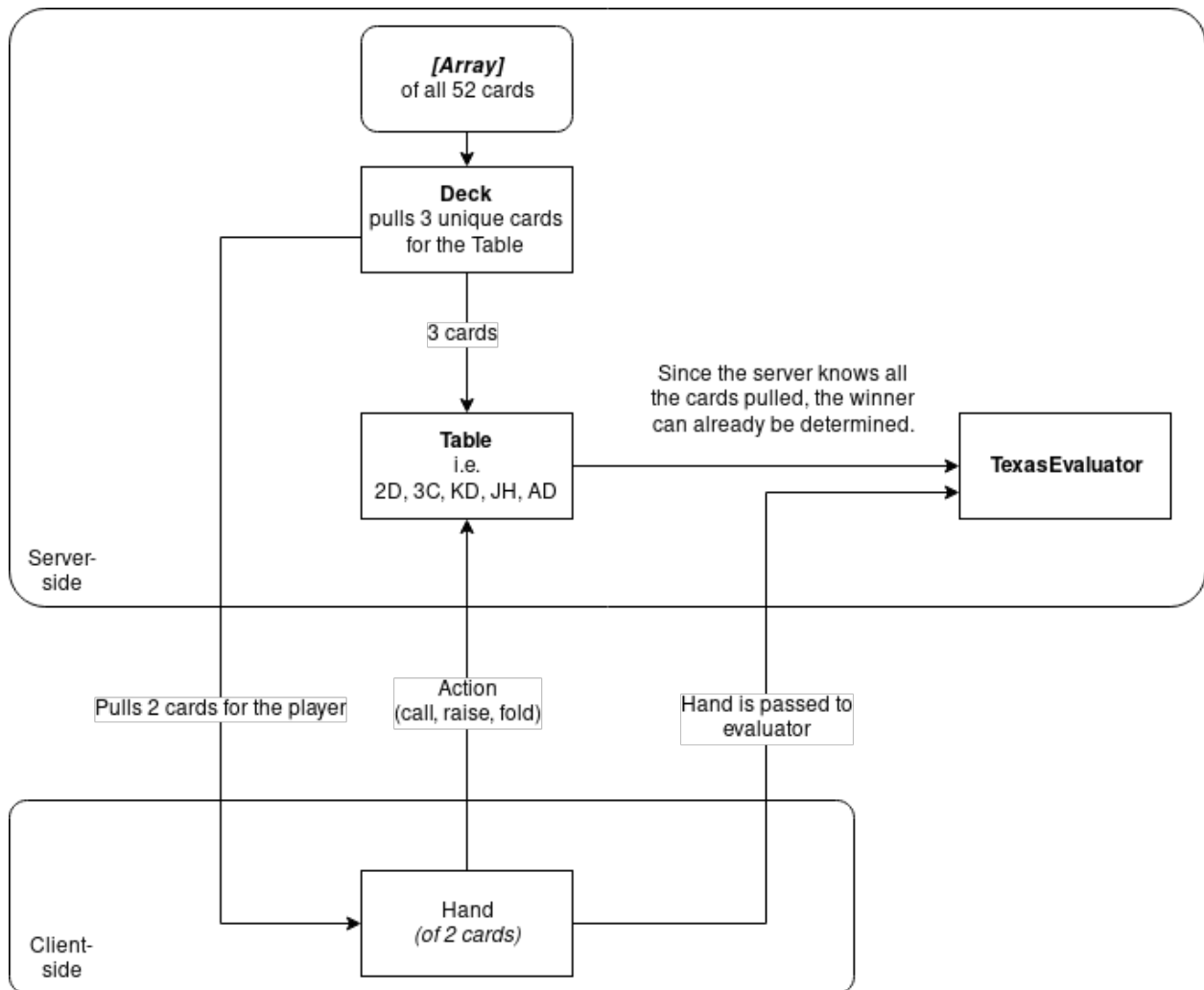


Color chart:
 Purple = completed
 Green = to be completed
 Orange = moved forward
 Red = deleted

Red = deleted



8.3 Abstract System Design



8.4 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;
}
```

8.5 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();
    }
}
```

8.6 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);
        }

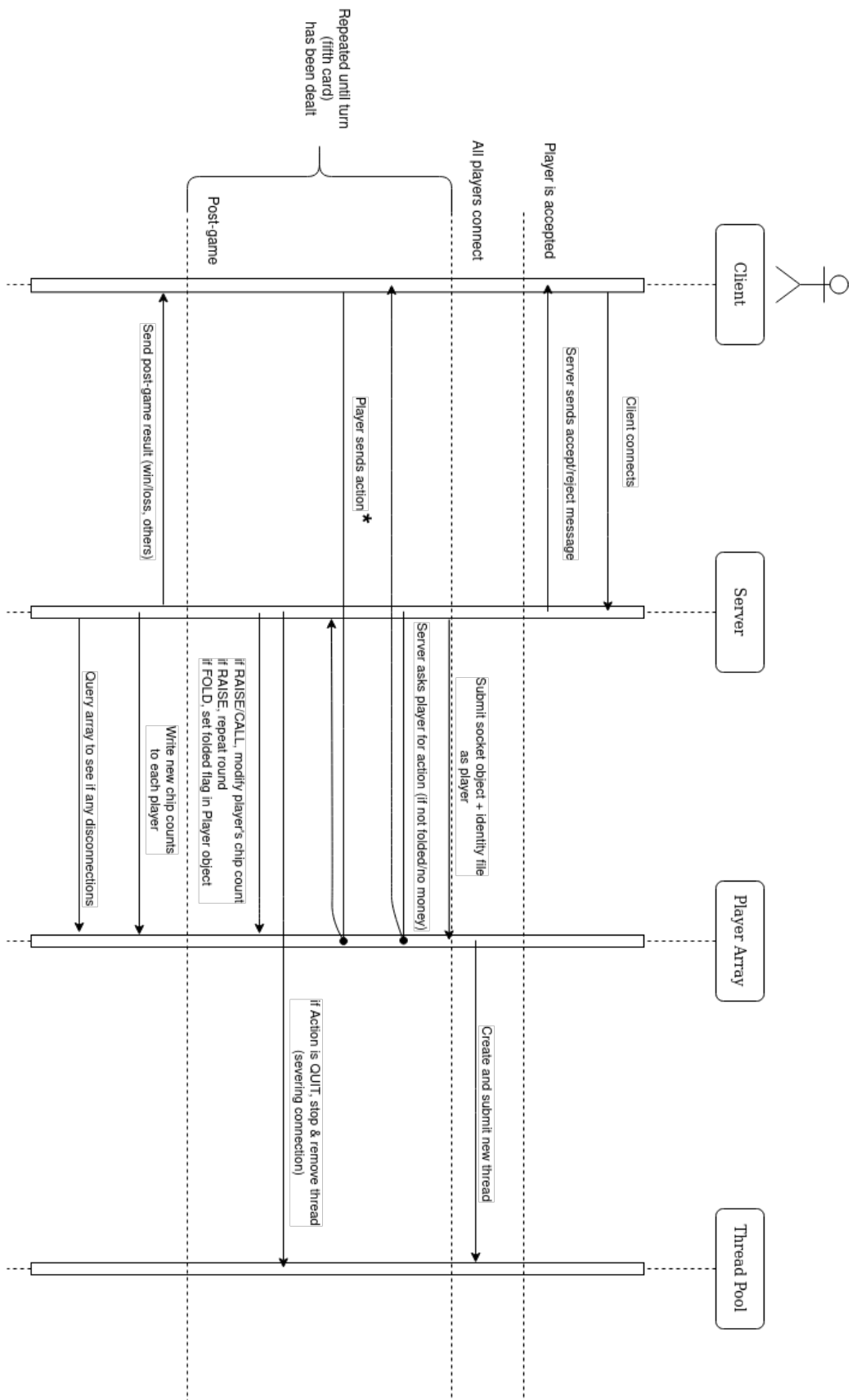
    } catch (IOException i) {
        System.out.println("Exception Caught");
        i.printStackTrace();
    }

    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(); }
}
```


8.7 Server-client sequence diagram



8.8 Poker Evaluation Algorithm

```
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);
}
```

9 Bibliography

Bibliography

- [1] Pedro Serrador, Jeffrey K. Pinto
Does Agile work?—A quantitative analysis of agile project success
Source in paragraph: 4. Results
Retrieved from <https://people.eecs.ku.edu/~saedian/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>
- [6] id Software
Doom 1993 source code: data sent in a game tick
Retrieved from https://github.com/id-Software/DOOM/blob/master/linuxdoom-1.10/d_ticcmd.h
- [7] Steve Kleiman, Devang Shah, Bart Smaalders
Programming with Threads, 1996
Retrieved from <http://www.cs.ioc.ee/yik/lib/2/Kleiman1pre.html>