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

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 [3] and is the main variant used in the World Series of Poker [4]. 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.

Figure 1 shows how the rotation of play in poker occurs. If the player with an unstable connection is unable to respond and the server is not able to proceed, then the match cannot occur as normal. Other players will not be able to take their turns, and the lost player upon reconnection may not be able to see choices taken or their chips.

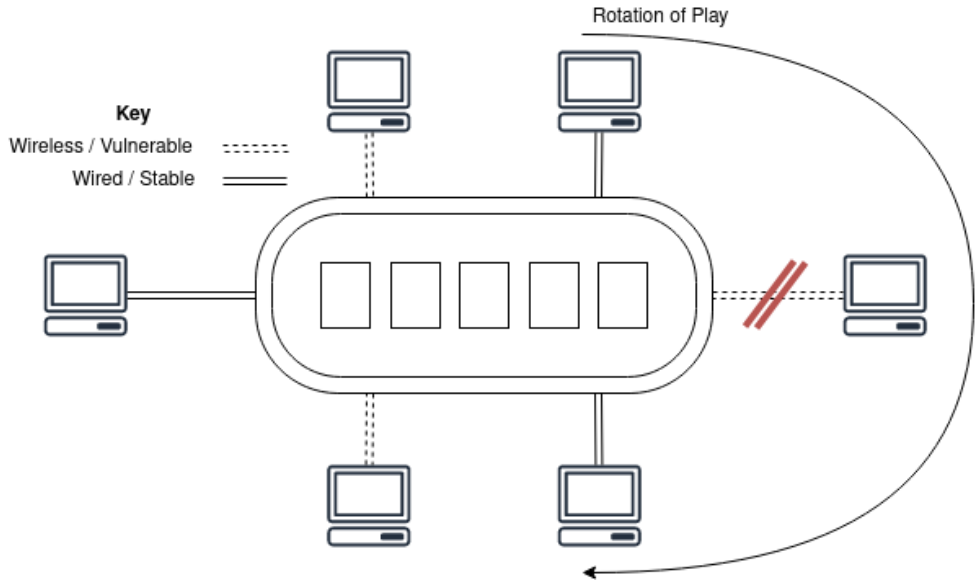


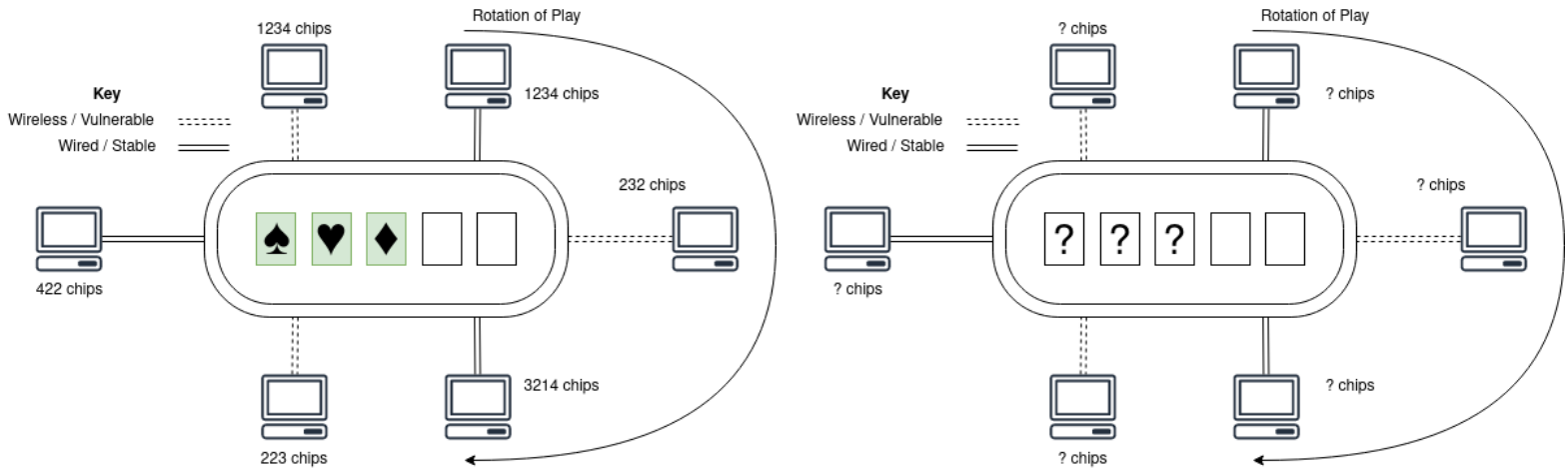
Figure 1: When not handled properly, the service can be halted

Though the term mobile devices can imply the use of mobile phones, mobile devices can range from not only a handheld phone but to vehicles, robotics, aeroplanes and satellites[12]. 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. In our case, we aim to send data stored on the server to each client in order to keep them up to date with the game.

This data 'overwrites' the clients data: we send packets to them that describe the exact state of the game at the current time. Figure 2 shows the difference between sending overwriting packets, or sending incremental, normal packets upon the end of the round.

2(a) shows when a player reconnects, they are sent the state of the game such that they know what has occurred whilst disconnected and allow them to plan ahead; 2(b) does not give any state on the game, any new packets sent would only reveal the cards not in view to anyone, and they would not see others or their own chip count. This approach was inspired by the store-and-forward approach that DTNs require[15], so that any lost or missed data is able to be reclaimed.



(a) Overwriting packets of data

(b) No 'overwriting' packets of data

Figure 2: Disconnected player joining back during the game, after the flop

2 Motivation

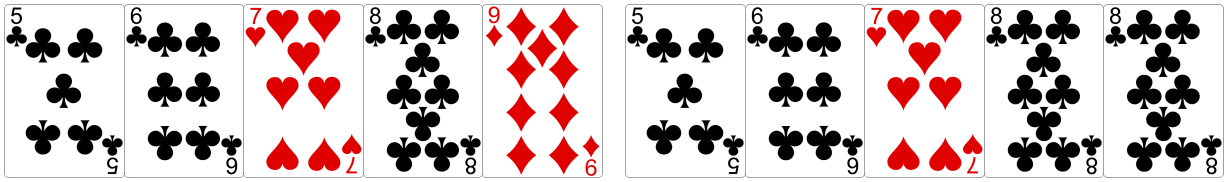
Rekop poker aims to provide an open-sourced framework for the popular variant of poker, Texas Hold'em. Many existing games of poker often include micro transactions and no ability to customize the gamemode, host their own servers or modify the source code to implement other gamemodes or wanted features. 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 as 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.



(a) A typical straight, 5 to 9 sequentially

(b) Edge case found, duplicate 8's counted

Figure 3: A valid straight versus the edge case we detected through unit testing

Had we not taken a test-heavy approach, given that a straight itself is quite uncommon [8], at around 0.392465% 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.

3.2 Agile

Agile is a relatively new methodology that can be described as very iterative, with a low design overhead. Its principles hold a focus on the self-sustenance of developers, rapid and continuous delivery of working software and continuous communication between each team member, as declared by it's manifesto[10]. It has been reported that the more the methodology is applied to a project, the more likely a project is to be considered successful.[1]

Whilst this was a one man project, the principles and ideas behind the Agile methodology have not been forgotten. This is primarily because of how well the rapid development cycle coincides with TDD; we were able to get instantaneous feedback and improve upon it. This especially occured in the backend of the project, whereby we could push better quality code and constantly improve with a concrete set of tests.

We were accepting to new requirements as it was realised that the initial idea of a grandiose mobile application may be a bit too much to handle within the given timeframe; we instead decided to move towards developing the core idea of the project: exploring creating a delay tolerant network.

When it came to developing the server and client connection, we realised that making an incredibly detailed design on a concept that was not particularly familiar with the author would more than likely end up in time wasted. Therefore, we decided to iteratively work on getting certain milestones completed: basic connection, starting with small ping messages. Next, we would work on implementing sending a single card between the client/server, then we would work on getting a hand of cards over.

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.

We decided to make the server handle most of the operations performed, with the client being responsible for sending their intentions (call, fold, raise) and receiving data about the current cards on the table. This is part of ensuring the network holds a form of 'store-and-forward', whereby data can be packaged and sent to a reconnecting client [15].

This can be seen in Figure 2, where the user is asked for their intention when it is their turn, and sent back. On the beginning of each round, we send the current table data to the players regardless of if they have folded or not; such that when a player is to reconnect, they are able to see the cards in play and actions taken, aswell as their current chips.

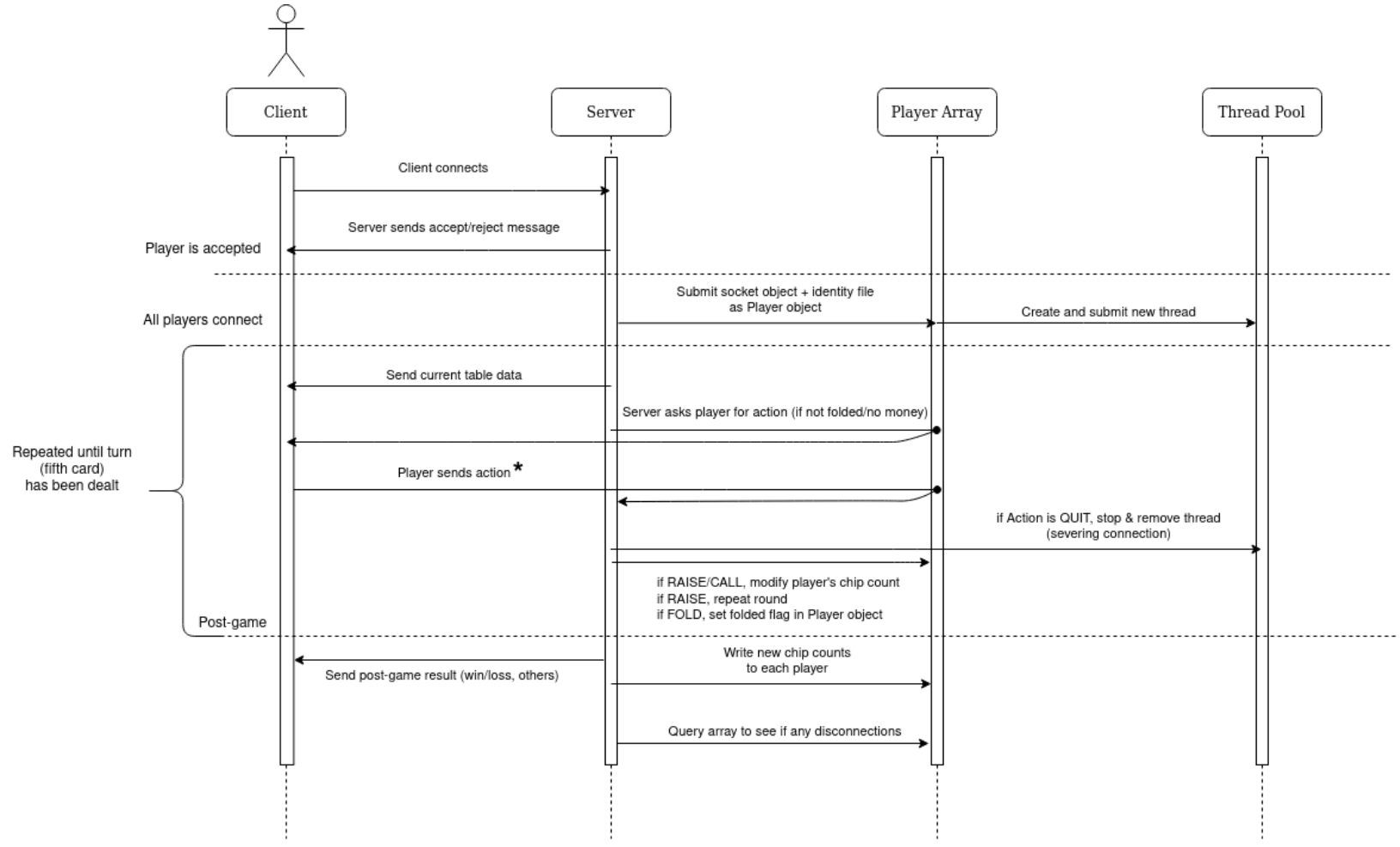


Figure 4: Communication between client and servers components

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.

We considered other languages such as C/C++, which would be magnitudes faster in performance however they are not platform agnostic by design, which is part of our project requirements - for example compilation issues can arise from using different compilers. Similarly, using a web-based language and framework such as JavaScript or PHP with libraries such as React/Angular would not fit into the system design; we need to host a server, and link the backend into it.

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, aswell 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).

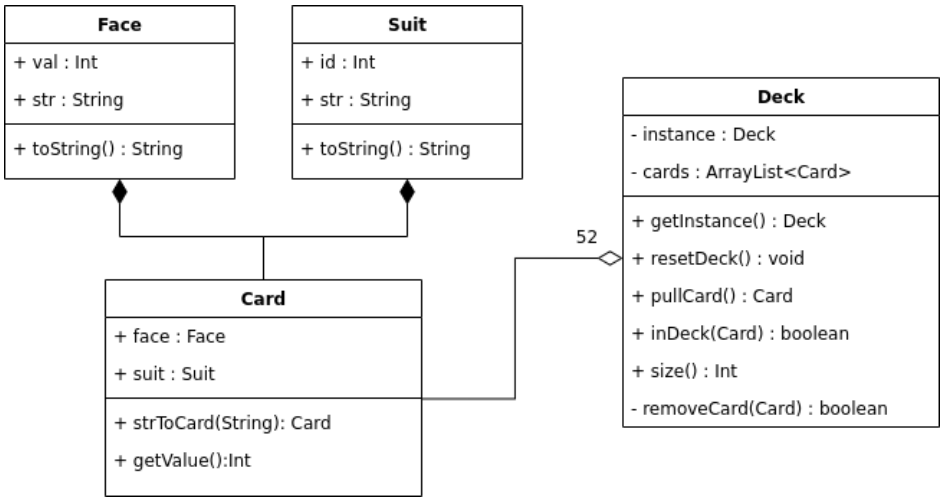


Figure 5: Class diagram of the base card objects

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 / Universal Plug-n-Play (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

For the server, we needed a way for each connection to be asynchronous, such that one connection which loses connectivity or experiences an error would not affect the others. 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.

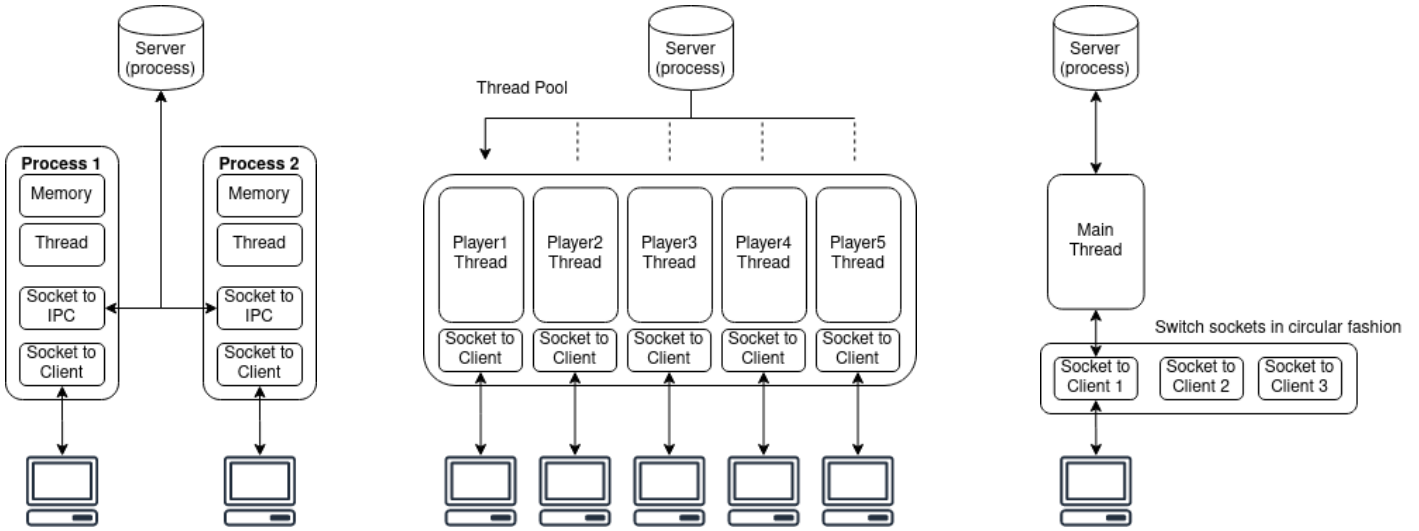
To clarify, a thread is a single line of execution - that is, a sequence of instructions separate to others - that is processed on a CPU at one time. As noted by Kleiman[7], the rise of client-server programming and multiple processors on a CPU drove the use of threads. A process is a separate program entity, meaning that it contains its own memory separate to that of others, and can host one or more threads of execution.

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 retrieval of data requires that the series of execution is halted; these functions are known as *blocking calls*[16], where the program cannot continue until data has been retrieved, hence why the connections from server must function asynchronously.

This means that a single-threaded approach is not feasible because if a connection is to be severed then the rest of the players will not be able to respond due to the thread finishing execution. This would eliminate any attempt at delay-tolerance.

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. Whilst inter-process communication offers shared memory[17] to combat the synchronization problem, it would be cumbersome to have to communicate with all processes from the server process.

Figure 5 illustrates the approaches we tried in greater detail. Fig 5(a) was notoriously complex to implement, as this involved having to create a socket not just for the client and process, but also from the process to the server to retrieve data.



(a) Process approach (b) Thread-pool approach (c) Single-threaded approach

Figure 6: Process, thread-pool and single threaded approaches illustration

Figure 5(c) shows the rotation of sockets between the threads, but this approach was flimsy and often if the player was to disconnect or an error occurred, the program experienced undefined behaviour and would throw a range of exceptions such as Thread.ConcurrentModificationExceptions, a simple SocketException or simply react in an unpredictable manner.

Whilst these are expected when working with networking and threads, the variation of exceptions thrown meant we could not rely on a single exception which we know is the cause of an unexpected or sudden disconnection. For example, in our final implementation we only rely on a SocketException - that typically is a broken pipe - in the network I/O functions.

We found that Figure 5(b), a thread pool, was the best design for implementing delay tolerance. This is primarily because we are able to give each player their own thread for connectivity rather than have to add in layers of complexity to communicate between processes, or the volatile nature of single-threaded, multiple-clients approach. If a player is to lose connection, their thread in the pool will finish execution, marked as 'inactive' and can simply be skipped over, as the figure shows.

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

In order to achieve a delay-tolerant network, the architecture of communication is a major factor in it being successful. For example, when a player is to disconnect there is a mechanism of "caching" or storing updates to send when a player reconnects. In our case, the data between initial disconnection and reconnection is not of importance, so we are able to send the current state of the game.

Communication between the client and server should utilize the Transmission Control Protocol (TCP). TCP is a protocol that sits atop of the network layer of the internet utilizing the Internet Protocol (IP)[18].
TCP uses the Internet Protocol to route data packets, but it describes the *format* of the datagrams. Furthermore, this architecture includes the use of the packets sequence number and a checksum to valid the data has not been corrupted. This is to ensure reliability[19], whereby the recipient must respond with an acknowledgement (ACK) packet to let the sender know they have received it. If no acknowledgement is received, the missed packet is resent.

In the case that our player is experiencing an unstable connection, there is the possibility packet loss will occur. TCP negates the affects of packet loss, namely by ensuring data integrity is kept.
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 when the player had lost connection. 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

The user interface is a simple command-line interface, whereby the player is able to view their cards and the total cards in play on the table, as well as the round. They are able to input their actions such as call, raise, fold or view their own chips.

Whilst this interface may seem quite limited, the underlying interface allows the UI to be transferred to other mediums, such as a mobile application. This is because the client reads in objects and formats them as text, and sends their actions back as a string; implementing a mobile application would simply require catching the objects and formatting them in a respective manner. Similarly, the new application would only have to send back the players actions as strings.

```
TPokerClient: Read player info
        ID   : 0
        Chips: 990
TPokerClient: Waiting for FLOP card(s)
TPokerClient: Retrieved
        SEVEN of Hearts
        ACE of Hearts
        ACE of Spades
Current cards in play:
        [QC 5H] 7H AH AS
TPokerClient: Current round = FLOP

What would you like to do? CALL | RAISE X | FOLD
RAISE 500
TPokerClient: TPokerClient: Wrote data: RAISE 500
```

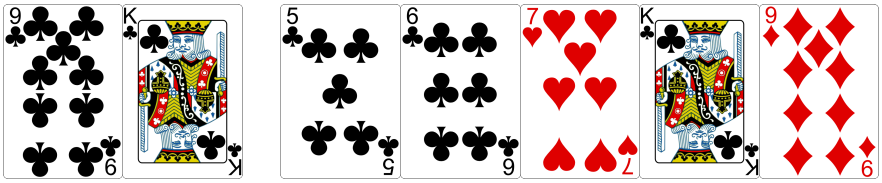
Figure 7: User interface

4.5 Algorithms

4.5.1 Evaluating Hand Strength

Evaluating a hand's strength is rather difficult and can be approached many ways. In our case, we opted to use a more human readable format by writing many helper functions that delegate certain checks for each type of hand, i.e. isStraight, isRoyalFlush, etc. This has the benefit of being readable and somewhat more unit testable; the only problem is that the speed of the algorithm can be quite slow, compared to other methods such as lookup tables have existed for some time.[2]

Whilst we knew using a computational approach like this would be slower, we attempted to bring down the speed of the algorithm by processing commonly accessed attributes ahead of time.



Two cards from the player, five from the flop, turn and river

From this array of unordered cards, we can use a TreeMap to store their faces (values) and suits which:

1. Does not accept duplicate entries
2. Orders its entries by key automatically
3. Provides an easy method of counting its entries

This means that our cards are already sorted from most powerful to lowest in terms of value as they are entered into the TreeMap. This means we can easily check if a flush exists by accessing the first value of the map - which will be accessed in a time of O(1) - and checking if it has a value of 5 or more. Similarly, if we need to check if any three/four-of-a-kind or pairs exists, we can iterate through the map and we will have the most powerful hand if one exists.

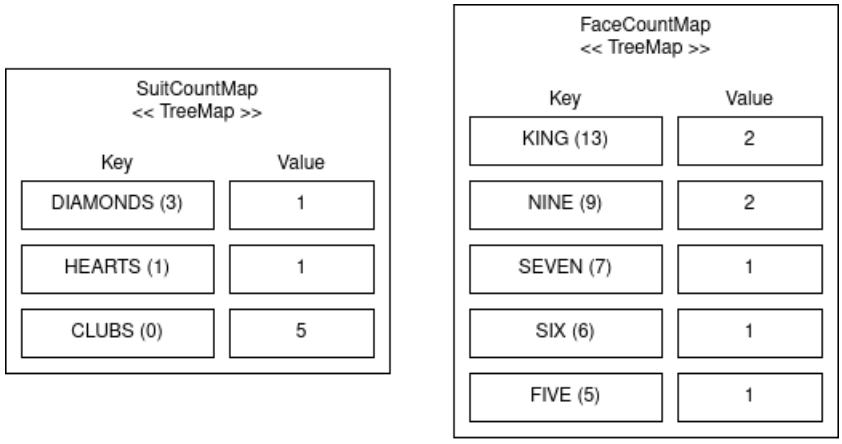


Figure 8: Diagram showing the ordering of key-values in TreeMaps

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

- Written in Java / 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, suits)

5.1.2 Server

Functional

- Runs on Linux and Windows
- Handles the evaluation of games internally
- Utilizes TCP/IP

Non-Functional

- Open-sourced
- Available for free download

5.1.3 Application

Functional

- Ability to run on any JVM-compatible device
- Allows for connecting to servers via IP address

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 drop out 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 drop out 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

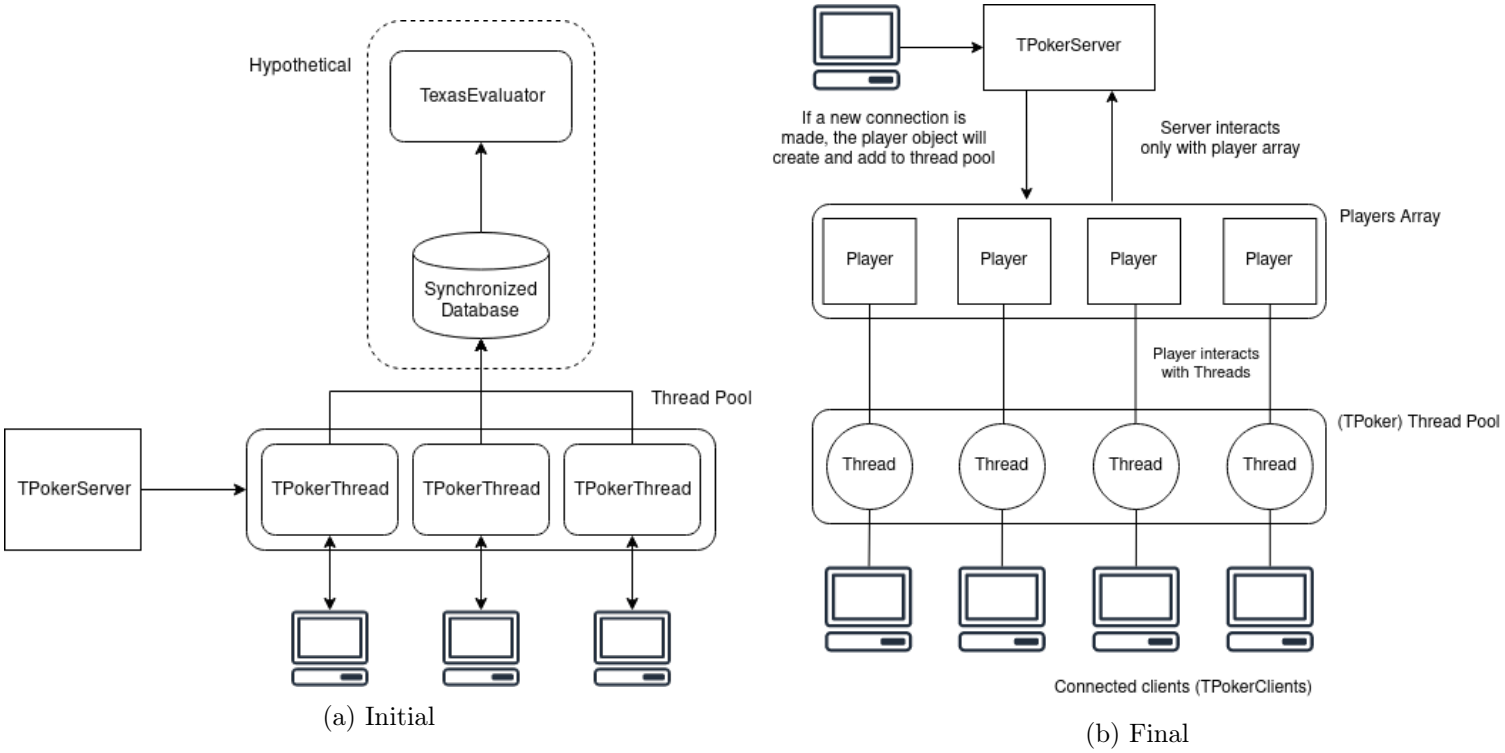
The server is written in Java as this means we can use compatible, standard Java networking libraries to communicate with the client counterpart. Furthermore, this makes it simple to port to new platforms to host on, particularly any operating system with a Java Virtual Machine.

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 typically hard to simulate delay or drop outs. We ran into the problem of restrictive network policies at student accommodation 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 drop outs.

As we were able to simply compile the server and its dependencies into a JAR file, we decided to host an Amazon Web Services (AWS) EC2 instance - a small, Amazon-hosted Linux 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[5].

Furthermore, we wrap these threads in a Player class, which not only keeps a reference to the connection thread in the pool, but also the data regarding the player: their identity, chip count, fold status and connectivity information; whether they have disconnected intentionally or a network error occurred, and their cards. These are then stored in an array, which solve a problem we encountered with our original design.



(a) Initial

(b) Final

Figure 9: Synchronization of player data and threads: initial vs final design

The flaw within Figure 8(a) is that we looked to store data in a database and initially kept some of this data within the threads themselves. However, upon disconnection during the round or before the ability to save to the database, the thread will finish execution and be destroyed; losing all of the players data.

We found the best method was to keep the thread independent of the data. The player class creates a new connection thread for the player, and passes it to the thread pool to be executed on first connection. In the case that the player loses connectivity, the thread will finish execution and be removed from the pool but the player's data remains in the object.

When the disconnected player rejoins, their identity is checked against those whom have disconnected: if they are rejoining, the thread within the player object can simply be 'swapped' out with a new instance and all network input/output can be regenerated. This removes the possibility of data been wiped in a drop out or significant delay, and the need for an external constantly updated database.

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 a built-in method.
We decided on using the built-in Serializable interface and the Object(Input/Output)Streams, as this did not require the need to built a serializer and deserializer for each object that we wanted to move over the network. Instead, we could build classes from enumerated and primitive (i.e. integers, strings) and then Java would serialize them for us. This does however have the impact that the server and client programs must have data classes built from the same source, otherwise there will be a version mismatch.

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 (enum) 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 integrated development environments (IDE) build functionality. 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.

Gradle[13] is a flexible, cross-platform build system with the ability to not only compile code but include test-case checks before completion. It is feature-rich, with the ability to add dependencies/libraries from common repositories such as Maven[14]. It also provides the ability for plug-ins, enabling further automation for the likes of transferring files to a server, and is the preferred build system for Android.[11]

Gradle provided solutions and time-saving measures that we did not anticipate during the beginning of the project. 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 identity file in the folder it was located, and so we used a file structure which contained players 1 - 8.

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

// for each player (p0 - 8) folder, copy the JAR above to the players folder
task moveToEnv {
    ['env/p1/', 'env/p2/', 'env/p3/', 'env/p4/',
     'env/p5/', 'env/p6/', 'env/p7/', 'env/p8/'].each { dest ->
        copy { with jarToCopy into dest }
    }
}
```

Figure 10: Copying the client JAR to 'virtual' player folders

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 with Gradle, it is possible to jar a certain class and its dependencies for use elsewhere, as shown below. This method also applies to the client JAR as referred to in Figure 9.

```
task servJar(type: Jar) {
    manifest {
        attributes  'Implementation-Title': 'TPoker Server',
                    'Main-Class': 'com.scully.server.ServerLauncher'
    }
    baseName = project.name + '-server'
    from { configurations.compile.collect { it.isDirectory() ? it : zipTree(it) } }
    with jar
}
```

Figure 11: Compiling the server package to a JAR file

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"("webServer") {
            setProperty("host", "ec2-35-178-207-104.eu-west-2.compute.amazonaws.com")
            setProperty("user", "ubuntu")
            setProperty("identity", file('rekop.pem'))
        }
    }
}

FileTree myFileTree = fileTree(dir: 'build/libs/') // JAR files location

// example exec: $ ./gradlew deploy
task deploy {
    doLast {
        ssh.run {
            session(remotes.webServer) {
                    // move JAR files to server
                    put from: myFileTree.asList(), into: '/home/ubuntu/'
                    // restart remote server; grab server output
                    execute '~/restart_server.sh'
            }
        }
    }
}
```

Figure 12: Declaring remote server address/credentials, transferring files and restarting

This further saved time, because the workflow typically went as follows:

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 execute a Gradle build script, and it would run the tests, ensure non are failing, 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 set up 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 # root directory to search in
T_SWITCHES="-e" # switches for terminal, -e = execute command
JAVA_EXEC="java -jar" # command to run
CLIENT_JAR_NAME="texas-client-1.2.1.jar" # file to launch with above
HOST="ec2-35-178-207-104.eu-west-2.compute.amazonaws.com" # server address

# for p1-p8 in ROOT_DIR (env/), -
for PLYR_DIR in $(ls $ROOT_DIR)
do
    # $TERMINAL = default terminal used
    # example cmd: alacritty -e java -jar env/p0/texas-client-1.2.1.jar 192.168.0.1
    $TERMINAL $T_SWITCHES $JAVA_EXEC $ROOT_DIR/$PLYR_DIR/$CLIENT_JAR_NAME $HOST
done

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

Figure 13: The Bash script used to launch multiple instances of the client to a server for testing

6 Evaluation

6.1 Unit Testing

Unit testing was crucial in the development of the back end, 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.

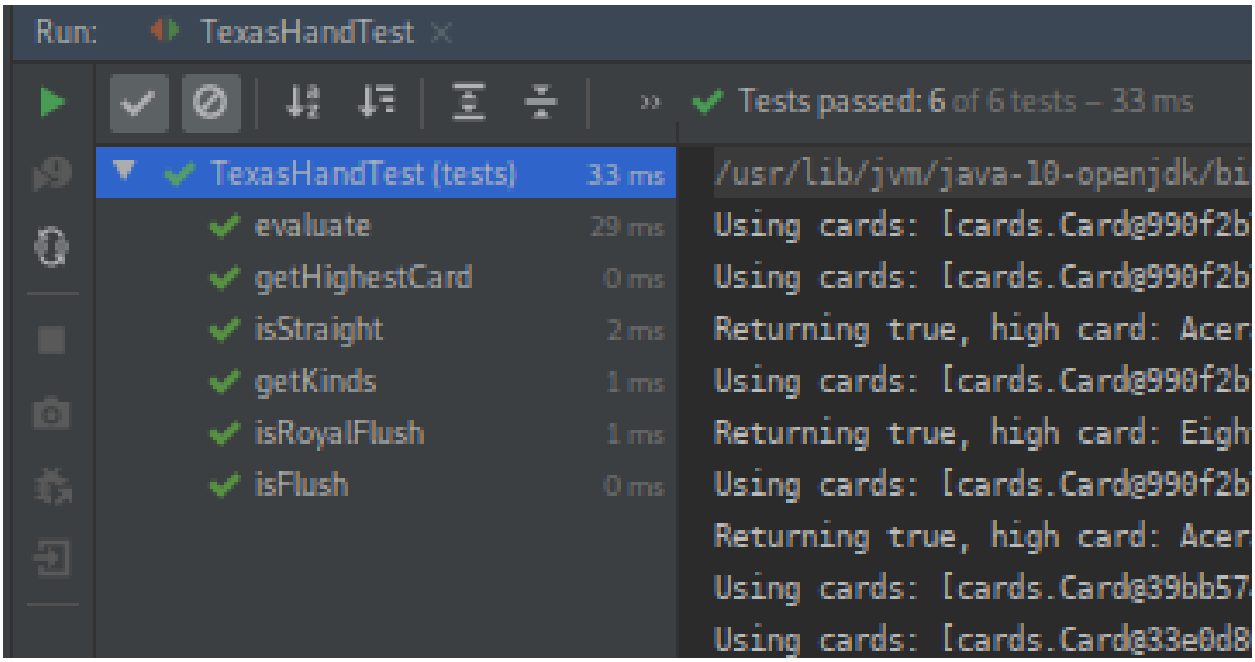


Figure 14: Unit test results

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 network code

Testing network code is much more different than testing typical single-threaded, local code. Due to the blocking nature of network calls when we hosted the server and client on the same computer, it became 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.

Whilst it is possible to unit-test network code, it can be tedious and complex to set-up, and not all errors can easily be detected. We opted to use a logging system, whereby we constantly printed what was being sent on both sides of the network to the console to see if there was delay, a block of execution or simply crashed. We found this to be the most effective way, as debuggers were often awkward to use when working with multiple threads and when a bug was unpredictable.

Figure 10 shows the server and client logs side by side. For example, when we are sending card information we know that if the client is to crash or otherwise encounter an error, then the client is attempting to cast the sent data to the wrong type, i.e. the client is expecting the data to be a PlayerInfo, instead of Card data.

```
ubuntu@ip-172-31-21-252:~$ TPokerServer: Server started
TPokerServer: Waiting for connections
TPokerServer Pool Stats: Active: 0 Complete: 0 Count: 0 Size: 0
TPokerServer: Connection accepted, id = 0
Player: Retrieved identity file with token: 2116063865
TPokerServer: Not a duplicate, adding to hashmap
TPokerServer Pool Stats: Active: 1 Complete: 0 Count: 1 Size: 1
TPokerServer: Connection accepted, id = 1
Player: Retrieved identity file with token: 671557375
TPokerServer: Not a duplicate, adding to hashmap
TPokerServer Pool Stats: Active: 2 Complete: 0 Count: 2 Size: 2
TPokerServer: All players connected; proceeding with game.
TPokerServer: Entered PlayStage
TPokerServer: Dealing round: PREFLOP
Dealing cards + QUEEN of Clubs FIVE of Hearts
        (QUEEN of Clubs)
        (FIVE of Hearts)
Dealing cards + SIX of Diamonds KING of Spades
        (SIX of Diamonds)
        (KING of Spades)
TPokerServer Pool Stats: Active: 2 Complete: 0 Count: 2 Size: 2
TPokerServeR: Writing PING request - player 0 should know to reply
TPokerServer: Waiting for action input
TPokerServer: Player 0 has taken action: CALL
TPokerServeR: Writing PING request - player 1 should know to reply
TPokerServer: Waiting for action input
TPokerServer: Player 1 has taken action: CALL
TPokerServer: Sending global message - NEXT
TPokerServer: Dealing round: FLOP
TPokerServer: Dealing FLOP card: SEVEN of Hearts
        (SEVEN of Hearts)
        (SEVEN of Hearts)
TPokerServer: Dealing FLOP card: ACE of Hearts
        (ACE of Hearts)
        (ACE of Hearts)
TPokerServer: Dealing FLOP card: ACE of Spades
        (ACE of Spades)
        (ACE of Spades)
TPokerServer: Server Current round = FLOP
TPokerServer Pool Stats: Active: 2 Complete: 0 Count: 2 Size: 2
TPokerServeR: Writing PING request - player 0 should know to reply
TPokerServer: Waiting for action input
```

(a) Server

```
[yames@yames p1]$ java -jar texas-client-1.2.1.jar ec2-5
TIdentity: Using identity token: 2116063865
TPokerClient: Connecting...
TPokerClient: Server accepted our connection
TPokerClient: Waiting for PREFLOP card(s)
TPokerClient: Retrieved
        QUEEN of Clubs
        FIVE of Hearts
Current cards in play:
        [QC 5H]
TPokerClient: Waiting for server to ask us for response.

What would you like to do? CALL | RAISE X | FOLD
CALL
TPokerClient: TPokerClient: Wrote data: CALL
TPokerClient: Read player info
        ID   : 0
        Chips: 990
TPokerClient: Waiting for FLOP card(s)
TPokerClient: Retrieved
        SEVEN of Hearts
        ACE of Hearts
        ACE of Spades
Current cards in play:
        [QC 5H] 7H AH AS
TPokerClient: Current round = FLOP

What would you like to do? CALL | RAISE X | FOLD
```

(b) Client (Player 0)

Figure 15: Output of both the server and client in synchronization

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 back-end, 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 as well as implementing them into the Android application have been brought forward, as these are critical.

7.2 Contributions and Reflections

7.2.1 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.

7.2.2 Data classes

An area that could have been improved would have been the data classes. Whilst the built-in option of packing and unpacking data, if we needed to make a small change to a data class even so much so as a tiny change in calculating something, we would have to recompile and deploy the server / client executables again. If we had taken the time to write our own packing/unpacking classes for the classes, we could have removed this limitation and sped up development further.

7.3 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.

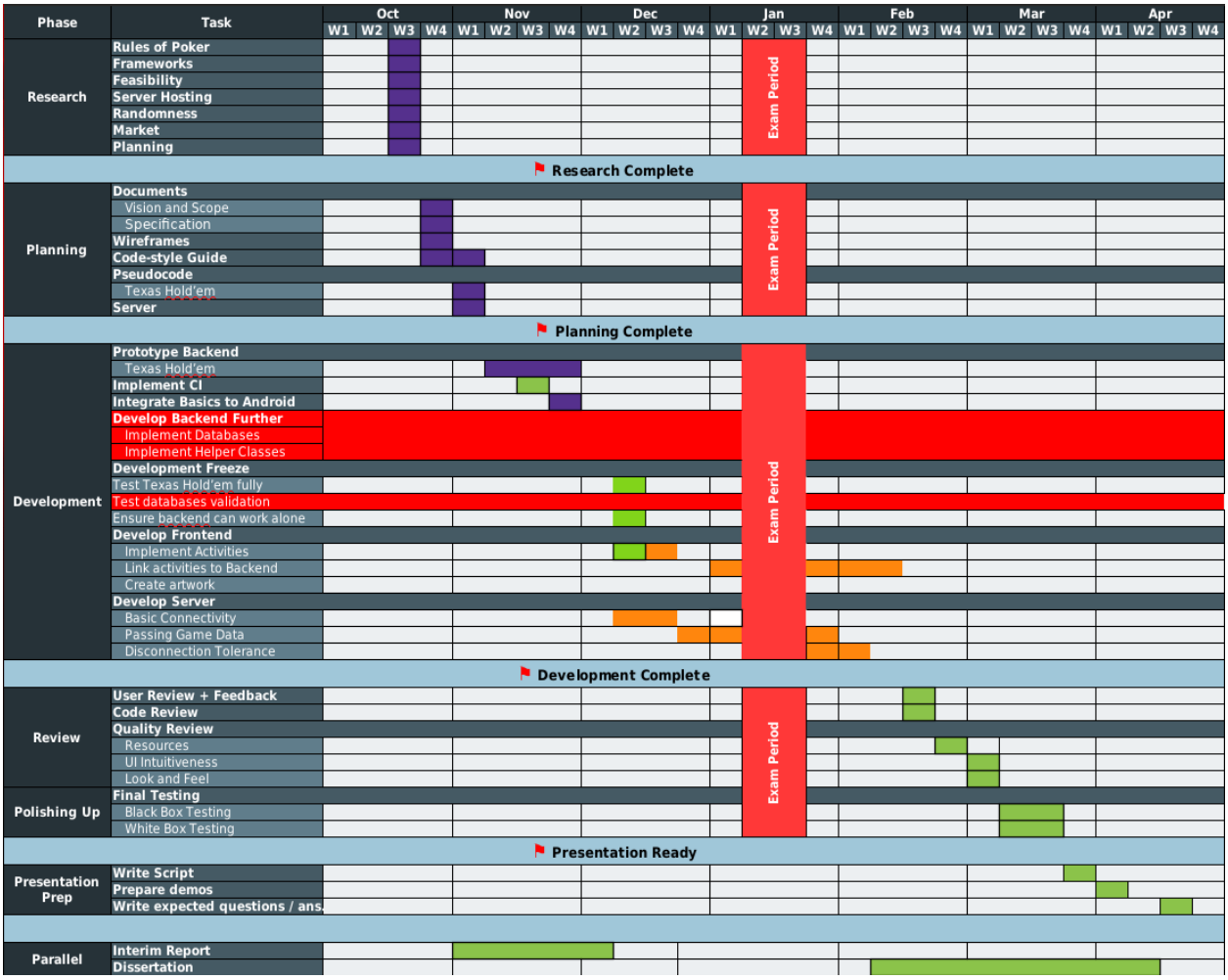
I think some areas could have been improved, namely implementing a suitable and fun user interface or front-end so that it could be presented to actual players but the framework to do so is somewhat there itself and could be picked up by anyone in the future. Moreover I think time management could have 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, where we continuously are looking for incoming connections, 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.

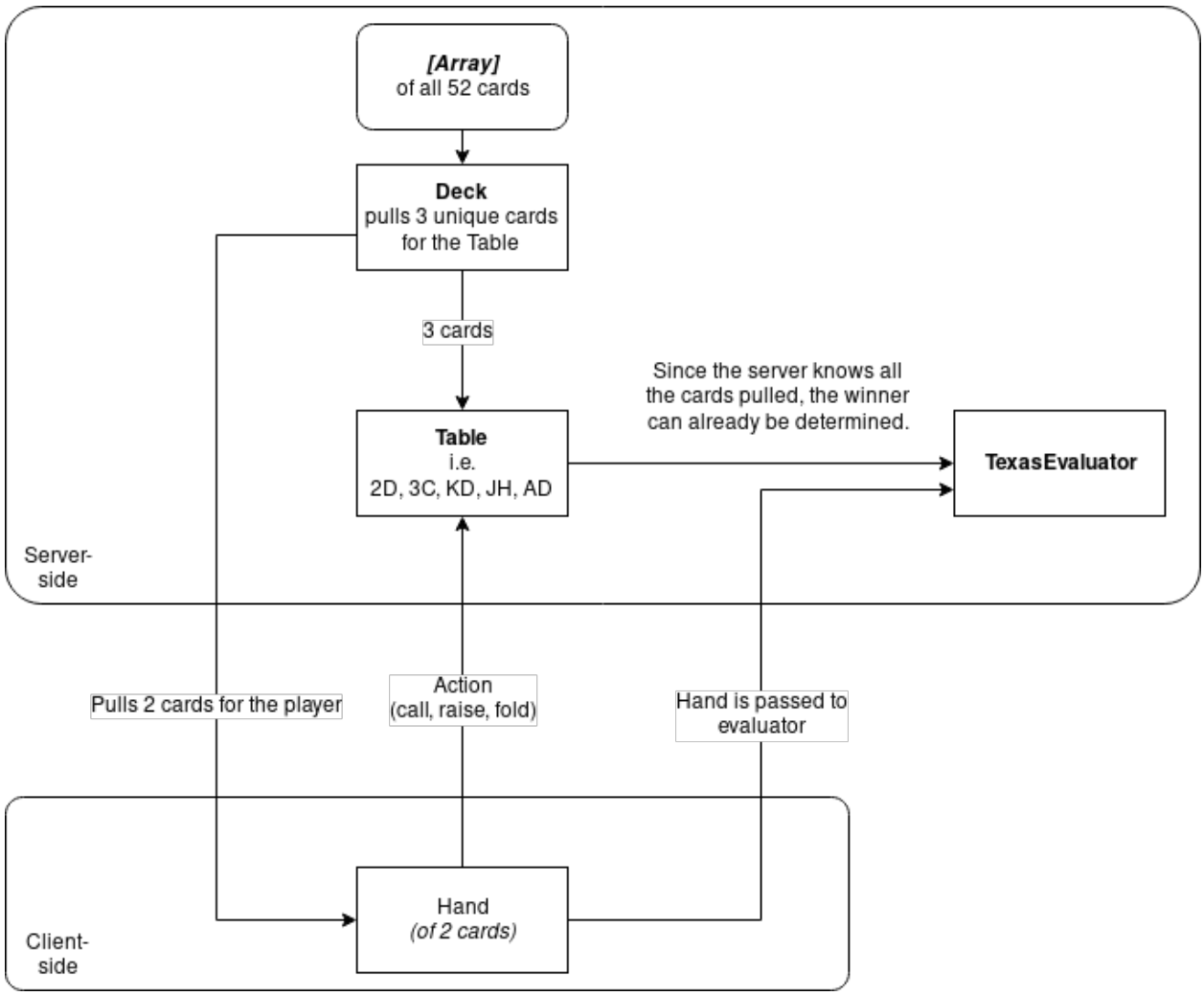
8 Appendix

8.1 Gantt Chart

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



8.2 Abstract System Design



8.3 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;
}
```

8.4 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();
    }
}
```

8.5 TPokerClient main method

```
public static void main(String[] args) {

    try {
        Socket sock = null;
        // by default, use localhost
        String HOST = "127.0.0.1";

        // if we've started with an argument, use it as the host
        if(args.length > 0) {
            HOST = args[0];
        }

        TIdentityFile identityFile = new TIdentityFile();

        System.out.println("TPokerClient: Connecting...");
        sock = new Socket(HOST, TPokerServer.PORT);

        // in is our input stream, in this case command-line
        // out is the servers
        out = new ObjectOutputStream(sock.getOutputStream());
        in = new ObjectInputStream (sock.getInputStream());
        stdIn = new Scanner(System.in);

        out.writeObject(identityFile);

        String status = "REJECT";

        status = getMessage();

        if(status.equals("REJECT")) {
            System.err.println("TPokerClient: Server rejected this connection")
;
            System.exit(1);
        } else {
            System.out.println("TPokerClient: Server accepted our connection");
        }

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

    while(!QUIT) {
        reset();
        mainLoop();
    }
}
```

8.6 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

[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

[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

[8] Tom Ramsey, University of Hawaii
5-Card Poker Hands
Retrieved from http://www.math.hawaii.edu/~ramsey/Probability/PokerHands.html

[9] Kent Beck
Test-driven Development: By Example, pg. ix - x
Mirror located at https://books.google.co.uk/books?id=CUlsAQAAQBAJ&lpg=PR7&ots=QBj-044PSZ&lr&pg=PR9#v=onepage&q

[10] Agile Manifesto (17 authors)
Retrieved from https://agilemanifesto.org/principles.html

[11] Google
Android Developers: Default build system
Retrieved from https://developer.android.com/studio/releases/gradle-plugin

[12] IEEE Distributed Systems Online, Greg Goth
Delay-Tolerant Network Technologies Coming Together
IEEE Distributed Systems Online, vol. 7, no. 8, 2006, art. no. 0608-o8002.
Retrieved from
https://www.computer.org/csdl/magazine/ds/2006/08/o8002/13rRUy08MzY

[13] Gradle
Gradle Build Tool: Build Anything, Automate Everything, Deliver Faster
Retrieved from https://gradle.org/

[14] Apache: Maven
Maven Central Repository
Retrieved from https://maven.apache.org/repository/index.html

[15] Institute of Electrical and Electronics Engineers (IEEE)
RFC 4838 - Delay-Tolerant Networking Architecture
Retrieved from https://tools.ietf.org/html/rfc4838#page-5

[16] IBM Knowledge Centre
Client/server socket programs: Blocking, nonblocking, and asynchronous socket calls
Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSLTBW_2.1.0/com.ibm.zos.v2r1.hala001/orgblockasyn.htm#orgblockasyn__sockpro

[17] GeeksforGeeks
Inter Process Communication (IPC)
Retrieved from https://www.geeksforgeeks.org/inter-process-communication-ipc/

[18] Information Sciences Institute, University of Southern California
Internet Protocol: DARPA Internet Program Protocol Specification p.7
Retrieved from https://tools.ietf.org/html/rfc791

[19] Information Sciences Institute, University of Southern California
Transmission Control Protocol: DARPA Internet Program Protocol Specification p.4
Retrieved from https://tools.ietf.org/html/rfc793