Yury Shukhrov

MMORPG Development for Android Platform

Author Title	Yury Shukhrov MMORPG Development for Android Platform
Number of Pages Date	68 pages 20 May 2014
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor	Olli Hämäläinen, Senior Lecturer

The goal of the project was to develop and deploy an Android platform architecture for a turn-based MMORPG. The task included the development of the architecture which should satisfy a client-server model, provide all necessary functionality to interact with a remote server machine and add interactivity through the design and styling of the graphical user interface. In addition to that, the development process involved the engineering of the game world model and the integration of the Observer pattern and the communication protocol.

The report describes the process of multiplayer game development, properties and features of the selected technologies, guides through the architectural structure of the project and reviews the implemented functionalities. The project was carried out using Android platform framework. An integrated development environment was Eclipse software. The main programming languages were Java and XML. The development process was divided into two parts: client side programming and server side programming. The report emphasizes the client-side development.

A multifunctional multiplayer Android game architecture, combined with social role-playing features was created. In its current state the game supports the authentication and registration of players, character selection procedure, character inventory and battle statistics, location-specific operations, duel request generation, fight instance and logic, active duels list and location players list. The game supports a large number of simultaneous connections and can withstand the load of hundreds of online players due to the features of NIO technology.

Keywords	multiplayer game, MVC, client-server, MMORPG

Helsinki
Metropolia
University of Applied Sciences

Contents

1   Introduction

The history of computer games counts for several decades. During this period, many types and genres of games were developed for different information technology systems and platforms. Initially, all games were single-player games, as it was not possible to connect multiple players together without the Internet. When the Internet gained popularity and a fast connection speed became a standard for the majority of people, a new category of games called online multiplayer games was born.

As the name suggests online multiplayer games enable multiple players to play the game in an asynchronous manner. Players can be located in different geological locations but as long as they all a have fast and stable Internet connection, they can interact with the game world and other players. This is possible due to a client-server model that allows intercommunication between the game world and connected clients or peers, controlled and supervised by the server.

The most popular online multiplayer games belong to a group of role-playing games (RPG). In general, the RPG involves a selection of fantasy characters that will delve into the game world and explore its content. The game character has a unique history and features based on the storyline of the game. Each character may belong to a particular race such as Elf, Orc, Undead or Human and plays a reserved role in the game world.

A combination of online multiplayer game and role-playing context results in a concept of a massive multiplayer online role-playing game (MMORPG) if a population of the game world counts a few hundred or more players. In this case, the architecture of the game server should be designed and developed in such a way that it would be feasible to support the hosting and asynchronous processing of a large number of online clients.

Developing MMORPG is considered a challenging task, mostly because there are many extra modules and components that are required in order to satisfy a client-server model. Compared to a single-player game, where the implementation is mainly focused on the gameplay realization, a multiplayer game raises the level of complexity by involving factors such as server architecture software development and maintenance, creation of a communication protocol, game world synchronization, game data persistence, socket

programming, connection and load management, security solutions deployment and many more.

Creating an online game of the MMORPG scale and complexity opens vast opportunities for improving professional programming skills, getting a deeper insight on a game development process and gaining experience and expertise in different software development areas.

The goal of the project is to design and develop a universal and flexible client-side architecture for a simple, sequential, two-dimensional (2D) multiplayer game. The architecture targets Android platform and satisfies the criteria of the client-server model. The development process is divided into two parts: client-side programming and server-side programming. The report describes the process of the client-side development. More detailed requirements and specifications of the project are revealed in chapter 2. Furthermore, chapter 2 describes the game idea. Chapter 3 provides a theoretical background to the key technical aspects of the project. Chapter 4 sheds light on the architectural structure of the application. Chapter 5 describes the implemented functionalities and features. Finally, chapter 6 discusses the process of the application testing and troubleshooting.

2   Project Objectives

This chapter describes the requirements and specifications of the project. In addition to that, the idea of the project is explained. The members of the project and their responsibilities are introduced. Furthermore, this chapter discusses the reasons behind this project.

2.1   Requirements and Specifications

The idea of the project originated in May 2013. At that time, I was looking for a task that could satisfy the criteria of the Innovation Project that was part of the Bachelor's Degree Programme. A friend of mine Artem Moskalev who was studying with me proposed the idea of making a game using the Java programming language.

Initially, we were considering the options of creating a simple three-dimensional (3D) game, but very soon, we realised that we did not have enough skills and experience to start working with 3D graphics. There was a solution to that problem – the usage of some framework that would simplify the development process. Unfortunately, the only viable framework for 3D games development was the Unity framework that required a very expensive licensing. Later we concluded that we should start with developing a 2D game. Since both of us did not have much expertise and experience in game development, it was logical and obvious to start with a project that would be less complicated than a full-scale 3D game. We decided that it should be a multiplayer game.

A multiplayer support has its pluses and minuses. It makes the game more interactive, adds a real-time factor, expands user base and provides social features such as players' communication. On the other hand, multiplayer support adds more complexities and increases the scale of the project. It requires to split the development process into two parts: client side programming and server side programming.

Fortunately, Artem had a solid background in network programming, and I was keen and enthusiastic on learning mobile programming. Thus, we decided to divide the development process into two parts: I working on the client side, and Artem working on the server side. We chose the Android platform because it used the Java language that we knew well and in addition to this it was the leading mobile open source operating system (OS) with a large community.

We were planning initially to target Android handset devices, as their market share was much larger compared to tablets and other Android devices. However, soon we realised that the limitations such as a small screen size and lack of user interface (UI) space would result in poor integration of visual features such as Android graphical user interface (GUI) widgets and game sprites. Thus, we decided to target all tablet devices ranging from seven to ten screen inch size.

The minimal requirements for the server-side programming included:

- Designing and developing server software architecture
- Establishing a point-to-point connection with peers
- Integrating the Non-Blocking Input/Output (NIO) technology
- Creating the game world instance
- Synchronizing the game world instance
- Validating and caching the game data
- Deploying, maintaining and testing the server machine.

The minimal requirements for the client-side programming included:

- Designing and developing the client-side architecture
- Developing a communication interface to interact with the server
- Choosing parser technology to parse server messages
- Implementing game listeners based on the Observer pattern
- Adopting the architecture to support the game messaging protocol
- Designing and developing the authentication module
- Designing and developing the character selection module
- Designing and developing the game world module.

The main goal of the project is to create a multifunctional and universal multiplayer game architecture aimed for Android OS tablet devices that would satisfy the criteria of a role-playing game. The game should be based on the client-server pattern and support asynchronous communication with the server. The client software architecture should utilize local infrastructure to communicate with the server. The objective of the client side developer is to create such infrastructure and synchronize its operability with the server.

Moreover, the client side developer should come up with a look and feel of game elements and objects. This task includes planning and designing the way GUI will look like and how it will be positioned on the screen. Furthermore, the goal of the project includes carrying out research about the features of Android framework libraries and selection of the technologies that would meet the requirements and specifications.

2.2   Game Idea

Before delving deeper into the details of the game world and its content, it is important to introduce the general game idea. The project is based on the development of MMORPG game for the Android platform, targeting all tablet devices with seven to ten inch screen. The game mimics most of the operations and actions that are common for a game of this type. Although the game is much less complex and advanced as, for example, Word of Warcraft is, most of the concepts introduced in the game shed light on many challenging topics that are typical for multiplayer games and give rise to research in different areas of software engineering.

The game does not deviate from the MMORPG concept. It combines two major modules that are common to any MMORPG: multiplayer support for a large number of online connections and role-playing social features. The first task is completed by using NIO server architecture, which allows processing hundreds of sockets with a single thread. The second task becomes feasible by developing and deploying communication channels such as chat and private messaging.

The game world is divided into several modules: locations, character details, duel requests, active fights and fight instances. The game is 2D with a turn-based mechanism. A turn-based mechanism implies a sequential and systematic flow of actions. This mechanism can be explained on a board game example such as chess. In chess, the events unfold in a systematic manner. Each player takes the actions only when the turn comes. The same strategy applies to the turn-base game.

The game idea is centered on social interactions and the role-playing context, which is realized through fights and duels. Online players can challenge each other and gain experience and rewards through battles. The fight mechanism is based on sequential actions. Each player has an opportunity to select a part of the enemy's body where he/she wants to perform an attack and a part of his/her own body, which he/she wants to protect.

After both players finalize their choices, the server calculates the result based on a mathematical expression. Whoever generates more damage will win the fight.

Fights can be organized as duels or team fights. A duel is a face-to-face challenge, in which players take actions in a turn-based manner. Typically, a player with a better gear and higher level has a significant advantage over his/her opponent. During team fights, players are separated into red and blue teams. Each team member can attack only one target at a time, but the targets change frequently, so there is no fixed target. A team that generates most damage or outnumbers the enemy team, as the fight is declared finished will be considered a winner.

Several factors affect the flow of the fight: character stats, experience and level. The more experience the character has, the higher chance of a successful attack. With each new level, character stats are increased. Higher-level characters are more powerful than lower level ones. As a character advances through a level hierarchy ladder, he/she becomes more experienced and mighty. Another factor that increases character stats is the gear. Each character can wear a particular category of armor: cloth, leather, mail and plate. In addition to that, different armor items have a different impact on a character's offensive and defensive abilities.

Character stats can be divided into two major subcategories: offensive stats and defensive stats. Offensive stats include all battle properties that define a chance of a successful attack, damage generation, a chance for a special attack such as a critical attack that inflicts double damage and a chance to break the enemy's defense. Defensive stats give the probability to avoid being hit or damaged. For example, a dodge chance determines the probability of dodging enemy's attack, a block chance increase the chance to block a hostile attack and an armor stat allows to mitigate or absorb a certain amount of damage.

The game world is divided into several zones called locations. Each location has its own properties, features and characteristics. Currently, the game supports only three locations: Training Room, Castle and Barracks. By default, all new players start in the Training Room. Each location displays a list of online players present in that location. In addition to that, each location provides a list of active duel requests that can be accepted by other players in that location.

Each fight instance is running with two parameters: timeout and duration. Both parameters are set when a duel request is posted by the player. Timeout defines the amount of time the player can be idle in an active fight. If this amount is exceeded, a penalty will be applied, which grants an opponent the right to kill instantly the idle player and win the fight. A duration parameter is used to countdown a life span of a duel request. When the duration time of a duel request runs out, the request instance is automatically removed by the server.

When the fight starts, it will be registered by the server and common information will be available for public view. A common information includes details of the fight such as players involved in that fight, the fight's type and current state of the fight. The information is used to keep track of all active fights and monitor their progress. Fight instances have unique tracking numbers, which are used for their identification. All active fights are displayed in a list, embedded into each location.

3   Theoretical Background

3.1   MVC Architecture

In general, games can be categorized as dynamic and static. In dynamic games, the main content is presented to the user via frame animation, which brings some elements to live by redrawing them frame by frame at a certain frequency. If the number of redrawn frames exceeds a certain limit, it will create an effect of alive element that can move and perform some actions. In fact, the effect of alive, dynamic object is achieved by updating the position of the sprite – artistic image that represents the game element, and redrawing it inside the game loop. The difference between previous and current loop interactions constitutes a concept of frames per second (FPS), which serves as a standard measure of graphical processing unit productivity. [1, 129-136]

Static games do not use the game loop pattern to update graphics. Instead, they rely on the specification of the chosen architecture to update graphics only when it is necessary. Most of the game elements are updated via the framework's default widget rendering mechanism. For example, in Android, custom-image elements can be replaced and updated by changing the reference to the image resource and calling *invalidate* method. In most cases setting a new image resource for the Android widget will be sufficient to trigger a widget redrawing routine. [1, 129-136]

As the project game does not require redrawing graphics too frequently and most of the game content is not aimed for fast rendering, it is obvious that it may be categorized as a static game. Instead of rendering graphics in the game loop, the game updates graphics via the Model-View-Controller (MVC) architectural pattern. Each game view element is hooked up to a custom listener. Whenever a server reports that some information has changed, the client will detect which view elements should be updated in order to display new information. An update signal issued by event dispatchers notifies all subscribed views to redraw themselves.

The MVC architecture is widely used in many areas of software engineering. The key idea behind this architecture is to provide a clear and accurate separation between three modules: model, view and controller. The biggest challenge developing large applications resides in the fact that the complexity of the application code management rises exponentially with the amount of new code written. The MVC architecture was created

to simplify code management and provide a logical relationship scheme for different application modules. [2]
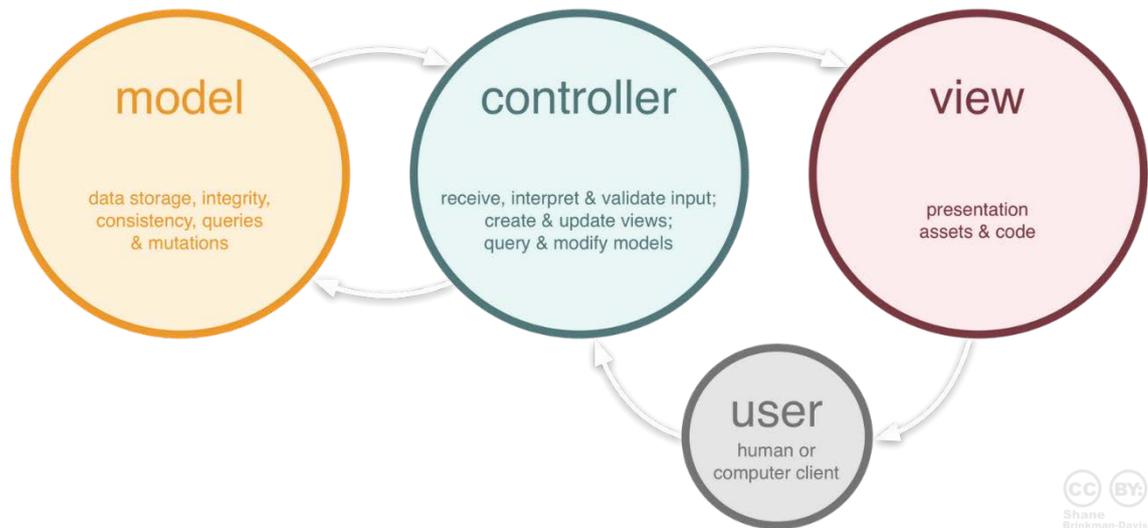

Figure 1. High-level MVC structure. Reprinted from Sagar Tahelyani blog. [3]

The model section displayed in Figure 1 is used to store Plain Old Java Objects (POJO). Following a good programming practice and Object-Oriented-Paradigm (OOP) all objects written in high level language such as Java should be designed and structured based on their properties, features and inherited characteristics. All objects have some unique properties and features that distinguish them from other objects. A model section is used to store information regarding object properties. In some way, it acts as a database, where POJOs act as tables. The information cached in model objects is later displayed by view objects that display this data in UI. [4, 31-39]

The view module shown on Figure 1 hosts GUI elements that consume and present the model data on the screen. Most of the view objects are responsible for detecting user interactions and notifying the controller that is responsible for keeping other elements alarmed in case of model's data change. The biggest difference between the Java language and the Android platform resides in the GUI part. Java uses Swing libraries to handle all GUI operations, while Android has its own built-in library that provides various widgets aimed to work with GUI. Nevertheless, most Java libraries are compatible with Android, providing a trouble-free transition to Java developers from Java to Android. [4, 31-39]

The controller module depicted in Figure 1 contains all objects and dispatchers that are involved in the process of coordination, data processing, action performing and event

triggering. The controller acts as a bridge between them. Whenever new data is available, it refers to the appropriate object model and stores it there. Any view objects can access model data to display it on the screen. As soon as new data is available, and some view needs to be redrawn, the controller sends a signal to the target view, ordering it to be updated. [4, 31-39]

The MVC architecture plays an important role in developing well-structured applications. All parts of the MVC work as a team. They communicate all the time, and each of them performs a particular task. While the model and view modules are performing monotonous and obvious tasks, the controller has a more multifunctional role. It acts as a team leader inside that team. It is up to a controller how to handle the data and which view to notify if the update is available.

3.2 Multithreading

Before multicore processors were available and publicly affordable, most computer applications and software were implemented by utilizing only one thread. By that time, this type of programming was called sequential programming, as all actions performed by a program followed a single flow of actions. A program executed a piece of code systematically from beginning to end. Although a large subset of programming problems could be solved using sequential programming, this approach was an inefficient exploitation of valuable and deficient computer resources, mostly because the throughput of a central processing unit (CPU) was low, as it stayed idle waiting for some operations to complete. [5, 1-5]

As computers and operating systems evolved, a more sophisticated and efficient approach for utilizing computer resources was introduced. It allowed more than one program to run simultaneously. Each program was allocated an "operating instance" called a process, which isolated it from other running programs and granted access to a common pool of system resources such as direct memory access (DMA), external storage utilities, input/output (I/O), and security credentials. In addition to that, a mechanism of processes priority was applied to allow the system to kill processes with low priorities in case there was a memory shortage or request from top priority processes. [5, 1-5]

A multithreaded program is the one that utilizes more than one thread inside its process instance. The integration of multiple threads divides a program into small modules, each

controlled by a sub-routine running independently from other parts but coexisting within the same process instance. Program threads allow performing multiple different operations at the same time. Each task can be assigned to a dedicated separate thread, which will utilize system pool resources such as memory and file handles. Each thread consumes resources allocated to a hosting process instance and has access to a separate program counter, stack, and local variables. [5, 1-5]

In modern days when most processors use more than one core, threads play an important and profound role. The structure and specification of multicore processors provide great benefits for multithreaded programs. Thus, a process, containing multiple threads that run different operations concurrently exploits a hardware parallelism of multicore processors. In other words, threads can run simultaneously on multicore computers. This approach is very efficient and productive, since CPU is kept busy most of the times and throughput is significantly higher compared to the sequential approach. [7, 797-801]
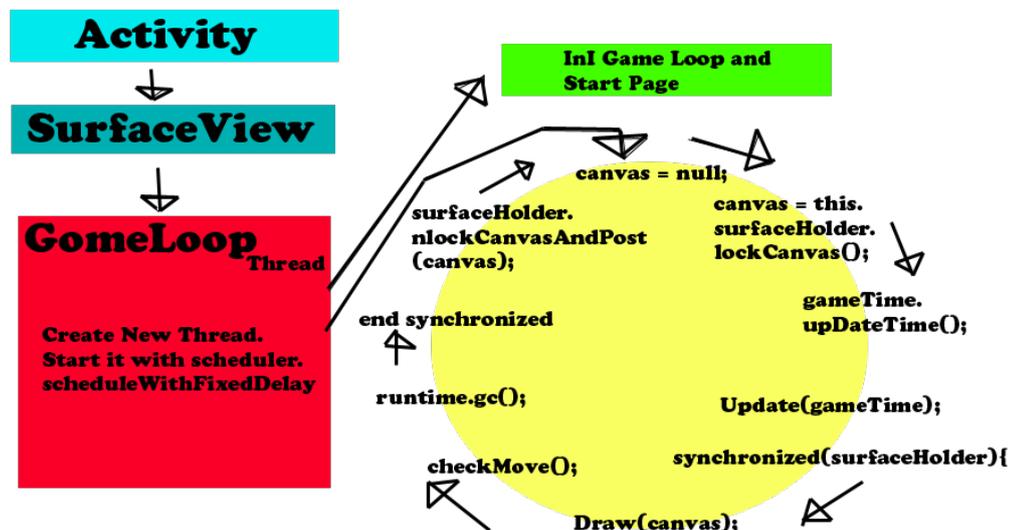

Figure 2. Game loop in Android. Reprinted from ZoZety Games. [6]

Concurrent programming is widely used in a large area of software engineering; game development is no exception. A primary area of utilizing extra threads in game application resides in game loop implementation. It is especially relevant for dynamic, graphic-intensive games. As Figure 2 shows, a separate thread is used to host the game loop that acts as the main tool for redrawing graphics and creating an animating effect. Android Activity holds a reference to the *SurfaceView* class, which provides more powerful functionalities for graphics rendering. *SurfaceView*, in turn, relies on a separate thread, which

manages and runs the game loop. The process of the game loop operation is illustrated as a yellow circle in Figure 2. [7, 797-801]

Extra threads are most commonly used when there is a need to perform some long running operation, which could otherwise block the main UI thread and cause an application crash or freeze its screen. For example, downloading a file from a remote server over the Internet or playing music in the background could be good reasons to use extra threads to perform extensive operations. In fact, it is a good practice to use extra threads if there is a need to run different operations concurrently, thus eliminating the risk of blocking the main UI thread and increasing the application's productivity. [7, 797-801]

For the part I was working on in the Android client the main application area of multi-threading approach touches the network module. By default, Android platform runs code in the main UI thread. All Android GUI widgets extend *View* class that is as a parent class for all widgets and base class for drawing GUI components. Android engineers equipped GUI widgets with a large set of callback operations that take place when a widget is created. It passes through multiple stages that define the look and feel by measuring its size, setting paint styles and colors, registering listener objects, applying attributes and finally drawing a widget and displaying it on the screen.

In order to establish a connection with the game server, the client should create a socket passing the address and the port number of the server. In the Java platform, this operation does not require to be performed in a worker thread. On the other hand, in Android most widgets and UI components undergo a drawing and measurement routines and are subjects to frequent change. In addition to that, most Android widgets are hooked up to specific listeners, which monitor the interaction process between a user and a target UI component.

For that reason, Android developers defined a maximum time gap, for which the UI thread can be blocked. Upon exceeding this threshold value the "Application Not Responding" (ANR) message pops up, and the user has to terminate the application in order to resolve this situation. Any network-dependent tasks are potentially risky and may cause UI thread blocking. In order to limit any risks, sockets are opened and closed in the background thread. [8]

When the connection is established, another thread whose role is to listen to server messages will be started. This thread receives a reference to a *Runnable* object that has an infinite loop running inside the *Run* method. In case a new message arrives, it will be detected at the entrance stage and passed to the parser module that deciphers the content from Extensible Markup Language (XML) format to Java and notifies subscribers through the controller class. A recipient is usually a GUI component or view such as an Android widget that upon notification updates information and redraws itself.

3.3 Networking

The main difference between a single-player and multiplayer game lies in the networking part. A single-player game does not require a network connection in order to expose its content to the user. On the other hand, it is impossible to design an online multiplayer game without utilizing a client-server model. In order to synchronize the game world data, control the game logic, monitor and handle the game world events, persist players' actions, and update the game world model, an external software is required.

The software is installed on a remote host or server, and its main objective is to process the incoming requests from the connected client programs and provide necessary data or take appropriate measures in a particular situation. The game server should record all changes, store valuable data in the database and apply security solutions in order to prevent unauthorized access and protect the integrity and confidentiality of the data. [21]

One of the major concerns in MMORPG is to ensure that game data will not be fabricated, stolen or replaced by unauthorized third-party software; hence, it is important to stress the confidentiality and integrity of game data during the process of server side programming. All major weaknesses and possible flaws should be taken into consideration. [21]

For example, let us consider the following scenario: During the communication process between the client and the server, there is a frequent flow of data packages transmitted from one direction to another. This data is private and should not be disclosed to an unauthorized person. If the communication channel is not protected, an intruder can easily intercept data, fabricate it or break its integrity; thus valuable data can be stolen if this case is not taken care of from the security point of view. [21]

Unfortunately, the scope of the project does not allow me to carry out a research and report on all possible security issues that could be discovered during the communication process between the client and the server. In fact, a special security treatment deserves the following modules:

- Authentication Module
- Communication Channel
- Game Items
- Server Database
- Communication Protocol.

The authentication module includes registration and login procedures. Players' credentials should be carefully stored in the server database and validated upon requests. It is recommended to store authentication credentials in an encrypted format. In order to secure data confidentiality and integrity during the communication process, a secure socket channel can be used. Game items should be tracked and validated on the server side. The server database should be guarded with the highest level of security. It should be obligatory to perform constant backups of the database and the data inside it should be encrypted with 256 bit or stronger keys. The communication protocol should be designed and implemented in such a way that it will prevent trial attacks aimed to guess some commands or against a malicious action aimed to flood the server with junk data commands. [21]

Any online multiplayer game is based on a client-server model. The client and the server are in a state of regular communication process. The data is transmitted in small data chunks that can belong either to Transport Control Protocol (TCP) or to User Datagram Protocol (UDP). The TCP is said to be a connection-oriented protocol, and it requires initiating a three-way handshake procedure before the connection can be established. The TCP adds a header to each chunk of client data to form a TCP segment. [9, 230-238]

The TCP header consists of header fields and a data field. The TCP header fields include the information about the origins and destination of the datagram, ordering and sequence, acknowledgement number, checksum and timing. Due to the header's overhead, the TCP header size is almost double the size of the UDP header. TCP offers a reliable data transfer service and guarantees datagram delivery. It ensures that data

packets arrive in order, and their integrity is not broken. The result is achieved by utilizing extra information that is present in the header of the TCP segment and the technology behind the transmission process. [9, 230-238]

The UDP is said to be a connectionless data transmission protocol. It does not perform a three-way handshake in order to establish a connection. It does not use the acknowledgement-signaling scheme either. UDP puts twice less information into the header of the datagram than TCP, which makes it lighter and faster to process. The specifications and structure of UDP do not imply any reporting or notification routines during the communication process. If the segment is lost during end-to-end transmission, the sender will not know it, since the recipient will not reply and acknowledge. [9, 198-204]

For a networking-oriented application, the UDP may sound like a more preferred choice, since it offers faster delivery speed. However, the speed benefit can be outweighed by the quality reduction. Although UDP transmission indeed generates lower latency, resulting in faster and smoother user experience, it is not guaranteed that the datagram will arrive safely in the correct order. The choice of the UDP and TCP depends on the nature of the transmitted material. The UDP is commonly used when the loss of several datagrams will not cause a noticeable quality degradation or data corruption and a smaller delay gap is desirable. The TCP transmission will be favored when a reliable data transfer is required. [9, 198-204]

It was decided to use TCP for the client-server messaging. There are several reasons in favor of this choice. First, small delays are not critical for a turn-based static game. Second, the quality is prioritized over speed and TCP guarantees packet delivery, ordering and integrity. Third, Java provides more complete and sophisticated tools for TCP socket programming.
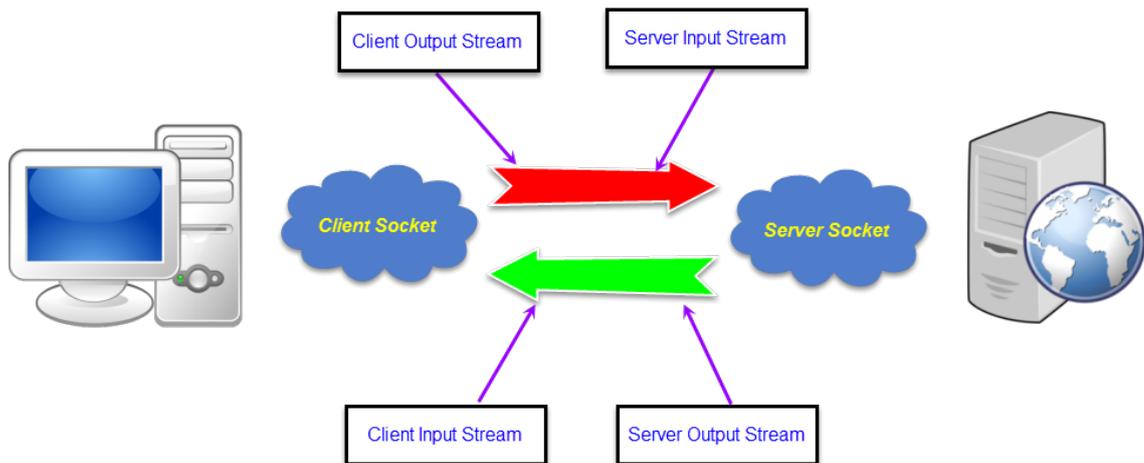
Figure 3. Client-server communication using sockets.

In order to create a point-to-point connection, an interface is required. Such an interface can be provided by the socket technology. Network communication using sockets is very similar to performing file IO. There are two types of sockets: client socket and server socket. An example of a full duplex communication using sockets is shown in Figure 3. The client socket provides a communication interface for the client application while the server socket does so for the server side. The server socket is bound to a reserved port, which is used to guide other clients in order to reach the correct target. When the client sends a request to the server, it will arrive at the main port, and then if the server accepts it, the new socket will be allocated with a different port number; meanwhile the main socket becomes unoccupied and is ready to accept other connections. [10, 346-352]

The client is using an output stream to transmit data to the server and input stream to read data from the server. The same routine applies to the server. The main difference between the client socket and the server socket is that the server has a well-known public IP address and port number, and it uses this port to listen to incoming connections. The server accepts each client connection and allocates another socket that will be unique and bound to a particular client. A unique socket is associated with a unique client's credentials such as an Internet Protocol (IP) address and local port. By using a new socket for each peer, the server shifts the load off the original listening socket reserved for establishing a new connection. [10, 346-352]

3.4   Android Framework

As the client-side developer of MMORPG, my main working environment was the An-
droid platform framework. Android is an open-source mobile OS running on a modified
version of Linux kernel. A company called Android Inc. originally created it and in 2005
it was purchased by Google. The Android framework offers numerous benefits to devel-
opers such as a complete and comprehensive start-up guide for beginner developers,
large community support, charge-free advanced integrated development environment
(IDE) software, a compact and ready-to-use standard development kit (SDK), portability
and compatibility with different hardware devices released by different companies, direct
access to the Google Play market, full Java language support and much more. [11, 2]

Android framework consists of four major layers divided into five sections: Linux kernel,
Libraries, Android runtime, Application framework and Applications. [14, 4]
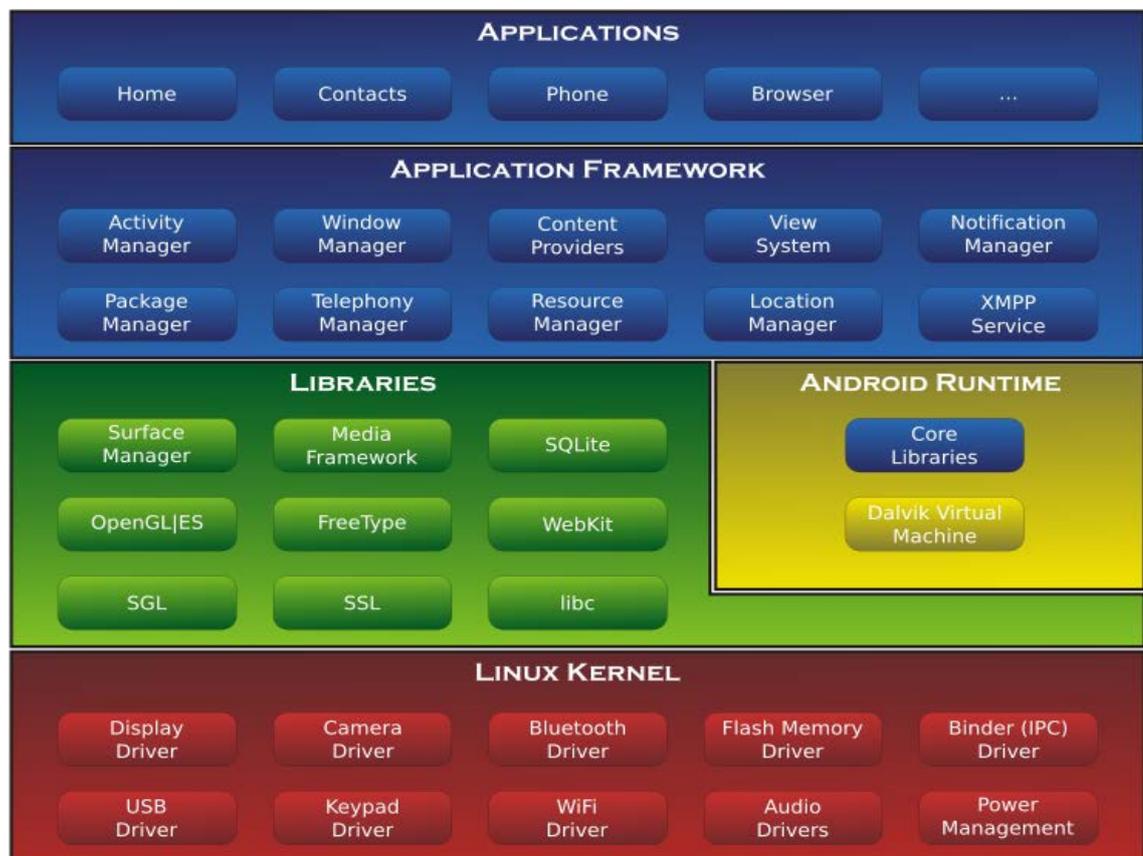

Figure 4. Android framework architecture. Reprinted from Edureka [12]

As Figure 4 illustrates, the bottom layer of the Android architecture covers the Linux
kernel. This part of the Android OS is written in an efficient, low-level and fast language

C, which is generally used for writing most low-level components, leading to the low memory footprint and promoting optimization possibilities due to explicit memory management in the form of Pointers. This layer contains all the low-level device drivers for various hardware components of an Android device. The driver is a middleware component or building block in the form of an adapter that creates an interface between the hardware and application layer to support the communication process between these layers and their components. [13]

The second bottom layer is divided into two sections: Libraries and Android runtime. Android libraries represent a collection of abstract and encapsulated classes, grouped together, serving the purpose of simplifying the development process and providing various tools for solving a wide range of problems. The main purpose of Android libraries is to provide a generic solution for common developing problems in order to eliminate the need of rewriting the code from scratch and emphasizing the reusability issue. Libraries include all the code that provides the main features of Android OS. The second part of the layer includes Android runtime. [14, 4]

Android runtime provides a set of core libraries that enable developers to write Android applications using the Java programming language. The major part of core libraries is inherited or adapted from the Java platform. For example, the *java.concurrent* package that belongs to the Java platform is available for Android developers providing various solutions for multithreading tasks. The Android code is interpreted by the Dalvik virtual machine during runtime. A new instance is created for every application that is run in its own process. The Dalvik virtual machine converts Java bytecode into a Dalvik executable file, stored and processed in *.dex* extension. This results in a lower memory footprint. Eventually, *.dex* file is converted into *.apk* file by the process of zipping and compression, which significantly reduces the application file size and converts it into executable form. Android *.apk* files can be run directly on Android devices in the same way *.exe* files are run on Windows OS. [14, 4]

The second top layer, the Application framework, contains the main Android utility helper services aimed to deal with telephony, messaging, data persistence, locations, notifications and visual data presentation. This layer is especially important for a developer as it provides the main tools for using the above-mentioned services and implementing solutions related to them.

The Application layer occupies the top layer of the Android architecture. It contains all applications that ship with an Android device, such as Phone, Browser, Contacts etc., as well as applications that are downloaded and installed from the Android Market. It also serves as a default location for all applications. [14, 4].

4   Architecture of the Application

4.1   Game Engine Concept

At the heart of any game architecture lies a game engine. The game engine is a software module that includes a bundle of libraries required to support the key elements of the game. Most game developers do not write a game engine from scratch; instead, they use a framework that provides ready-to-use solutions and simplifies the development process. The framework can be considered an optimized engine with a collection of implemented functionalities to support the game development.
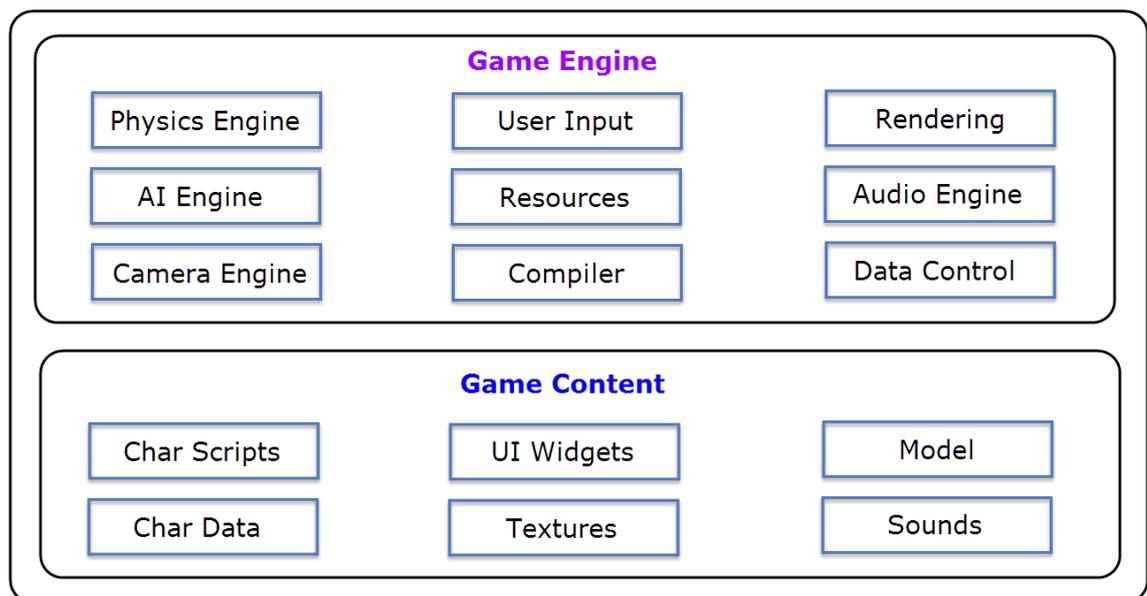

Figure 5. General structure of the game engine. Data gathered from The Game Design Initiative at Cornell University [15]

A typical game architecture consists of at least two modules: game engine and game content. Figure 5 shows the details of a general game structure. The game engine includes several modules, each aimed to provide utilities and support in developing a particular game part. The engine module is a collection of libraries bundled in a single framework. [15]

The physics engine block is responsible for working with game physics: movement, velocity, acceleration, matrices and vectors. The artificial intelligence engine provides a general solution for game logic such as collision detection and path finding. The camera

engine focuses on the camera work and defines which portions of the scene are visible at render time. [15]

The user input module provides the functionality to deal with user interactions such as mouse clicks, finger touches, tapping and swiping. Each game includes a solid portion of resources such as images, textures, sprites, sounds and fonts. The game engine should provide some functionality to manage and control these resources. A framework is written using a particular programming language. The language is interpreted and converted into a bytecode by a compiler. [15]

Games that require fast graphics rendering exploit the properties of the game loop. The game loop mechanism allows redrawing graphic frames in the background thread inside the loop and using delta time to define FPS. Most of the games use different sounds which are controlled by the audio engine. Finally, a game engine should provide the functionality or support classes to handle data manipulation processes such as data caching. [15]

Although the game can be developed without using most of the engine features, it is unfeasible to create a game without a game content. Game content is the face of the game, which determines the success and popularity of the game. Game content consists of the character data, maps the game world objects to model classes and provides the necessary functionality to display UI using framework widgets. It persists and consumes the application resources. [15]

Several factors affected the choice of the client-side architecture. Since the game belongs to the turn-based role-playing game category, it was obvious that the game does not require intensive graphics rendering. Moreover, it was decided that updating UI widgets with a default library *invalidate* method is sufficient for graphics rendering. In fact, the game does not use any physical properties such as velocity, acceleration and vectors. Considering these factors, it was decided that no commercial game engine framework is needed, and most of the planned features and functionalities could be implemented by utilizing a self-written game engine based on the MVC structure and Observer pattern. To control the game world resources and host the game content, an MVC architecture was implemented and deployed. A clear separation between the model, view and controller, powered by a multifunctional observer pattern allows applying game logic, displaying sprites and keeping the game content up to date.

4.2   Client-Side Architecture

The game is too complex to be described in detail, such as every aspect of the architectural style, structure and design. However, the architectural structure on the higher level will be covered and explained. Furthermore, the key modules of the application will be reviewed. The major steps of the client side development will be discussed, and the reader will get a sufficient amount of theoretical background to comprehend and grasp the idea behind the process of the construction of the different application blocks.
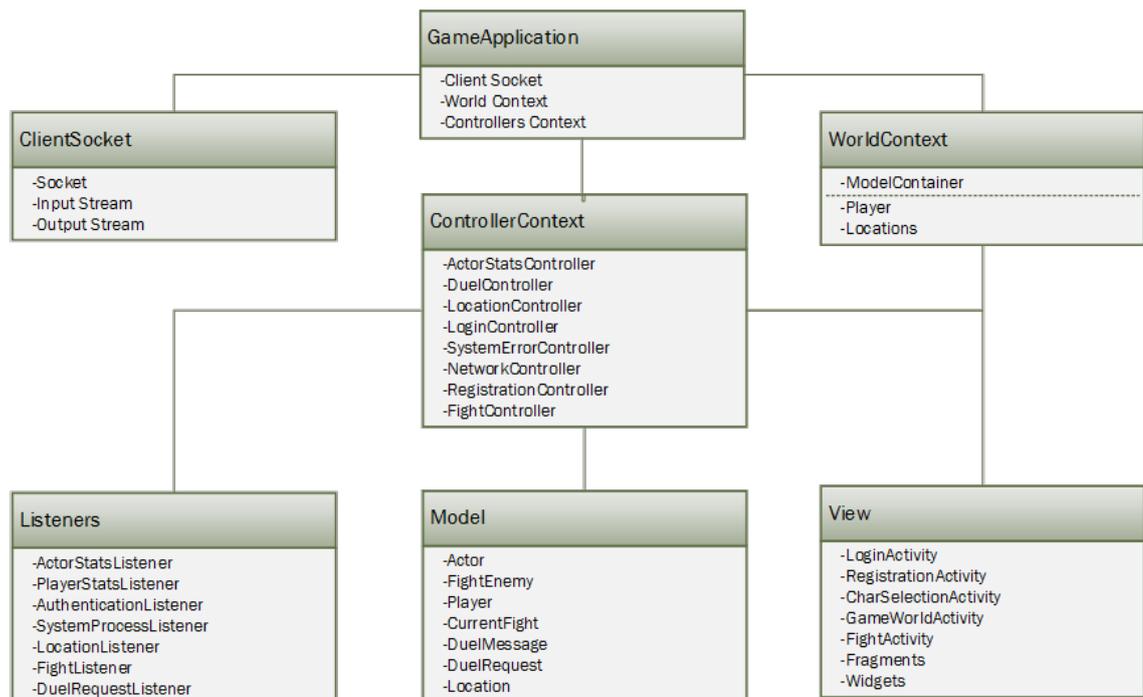

Figure 6. High-level game structure.

Figure 6 shows the client-side architecture on a higher level. Global data objects are stored in the Application scope realized through the implementation of the class *Game-Application*. This class contains a reference to the client socket, game world context and controllers' context. All objects from the class that extends Application are accessible from any part of the application if the reference to the application is obtained. The *Game-Application* class in instantiated automatically by the Android OS before it starts building any UI components, Activities or Fragments.

The client socket is the main networking interface of the application. It stores server information required to establish a point-to-point connection between the client and server.

Each time the application creates a new instance of the client socket, it generates a new local port number and assigns it to the socket. The client socket performs two major functions: provides the functionality to send requests to the server and receive messages from the server. In order to read the response from the server, an instance of the input stream should be created. Meanwhile to send any message to the server, an instance of the output stream should be created.

Client socket attributes are thread-safe. All class field variables are synchronized in order to avoid data mutability from different threads at the same time. The input stream is used to listen to the server messages. The reference to the input stream is passed to a background thread in the form of a message dispatcher object that starts an infinite loop and waits inside the loop for any server messages.
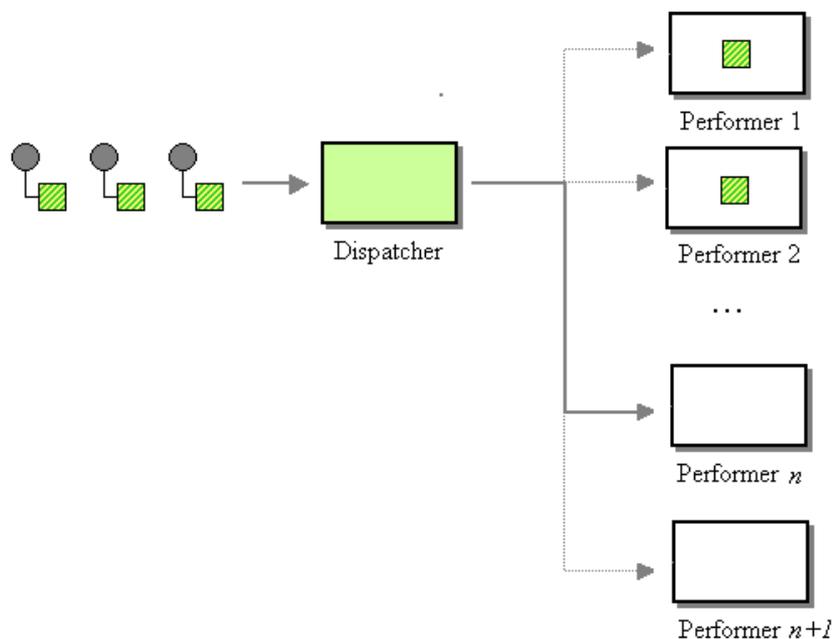

Figure 7. Message dispatcher diagram. Reprinted from the Enterprise Service Bus [16]

The server message listening loop performs the role of the dispatcher that fetches the message from the socket input stream and sends it to the performer. A performer is a local recipient object that expects data from the dispatcher. The process is illustrated in Figure 7.

The *WorldContext* object holds a reference to the game world data model. The game world data model is wrapped by the *ModelContainer* class to group and manage the

game world data model classes in a more organized way. The *ModelContainer* class includes direct references to the game world data objects such as *Player, Location* and *Actor*. The *Actor* class is an abstract base class for all game characters. *Player* class inherits from the *Actor* class.

The concept of polymorphism is used here. For example, we can create different classes that extend the *Actor* class and inherit common properties and features of it. In addition, these classes may have some unique properties, which make them actors but at the same time objects with some unique distinguishing features. A situation may arise when we need to store in a single array or collection different *Actor* objects. We may, of course, create several collections to store an object of a concrete type or we may just use the concept of polymorphism and cache all the objects of the type *Actor* in a single array. In order to retrieve different *Actor* objects and identify the correct subclass name we can apply the method of class casting. Applying the method of class casting, we can get an *Actor* from the array and cast, for instance, the *Player* class type on it, allowing to use both *Actor* and *Player* properties and methods.

Figure 8. Diagram of *Player* and *Actor* classes.

Figure 8 shows the class diagram of both *Player* and *Actor* classes. In addition to this, their relationship is marked. Private field variables are marked with red rectangles and methods are marked with green circles.

Location is another important game world data model class that is exposed by the *ModelContainer.* The game is designed in such a way that it has several locations. Each location has a unique look and hosts a list of players currently present in that location. Locations can be visualized as zones in the game world. Each zone has a particular

design, offers some tasks and hosts a group of players that are eligible to access this zone. Locations can be considered key elements in the game as they act as mini game worlds.
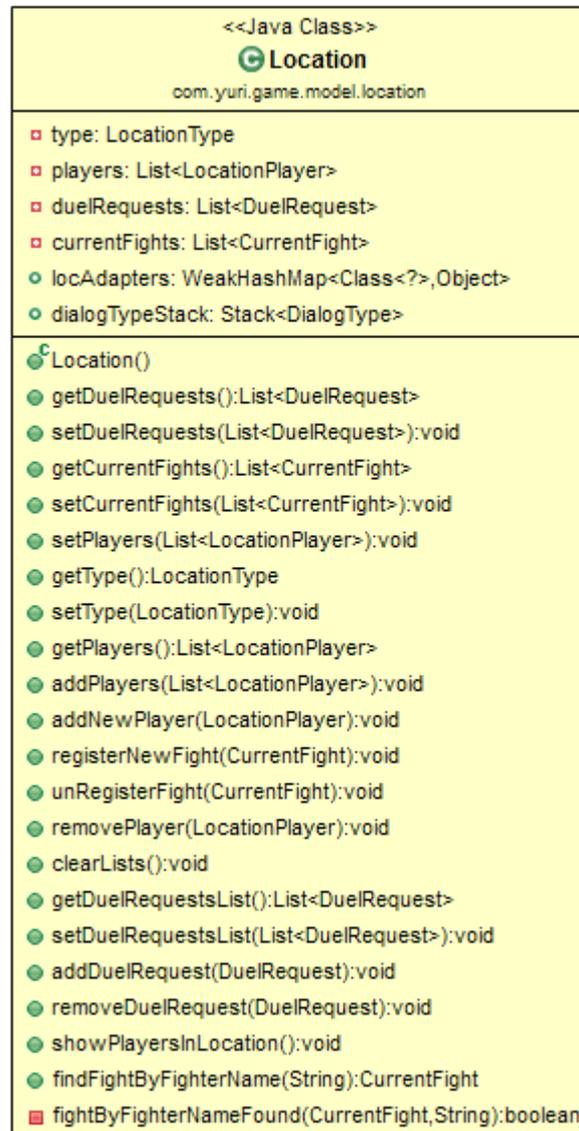

Figure 9. Location class diagram.

The content of the *Location* class is depicted in Figure 9. As a model class, it serves the purpose of storing the location-related information and keeping track of its state. There are different types of the *Location* object. The implemented ones are the Training Room, the Barracks and the Castle. Each location belongs to a particular type, stores the data about all location players and displays a list of duel requests and a list of active fights.

The *ControllerContext* is an application scope class that holds the references to a different type of game controllers. The main goal of the controller is to receive new information and based on the content and type of the information update the content of the affected model classes and notify the view classes to redraw themselves.
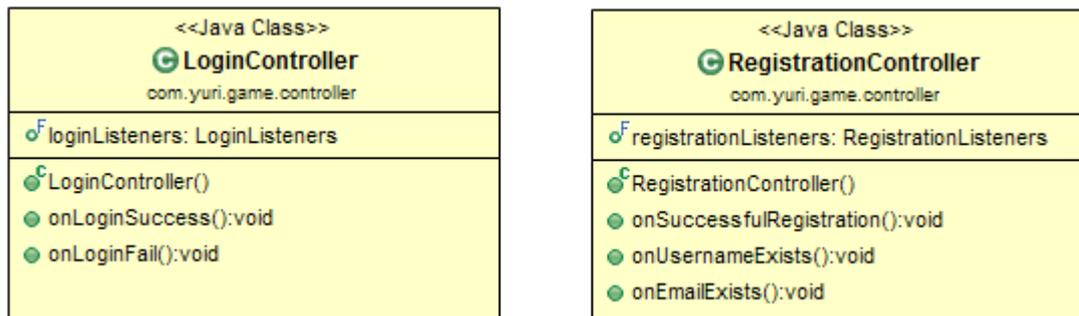

Figure 10. Class diagram of the *LoginController* and the *RegistrationController.*

Figure 10 provides a visual example of the class diagram of the controllers involved into the authentication process. Each type of the controller follows a similar architectural structure. The methods that belong to *LoginController* and *RegistrationController* are usually called from the parser class, after the content of the parsed server message becomes available. Based on the content of the received message the controller devices some game data model classes and views needs to be updated. To perform the update routine the controller relies on the local-scoped methods that trigger the update mechanism.

A question may arise how the views and the game world data model objects are updated by the controller. In other words, how the controller communicates with the model and the view. To answer this question I suggest examining Figure 10 diagrams carefully. Both controllers have access to the list of listeners of a certain type. The technology behind that will be discussed in the next subchapter. At the moment, it is important to say that this list of listeners holds references to all objects, which are subscribed for a certain events. When this event happens, all subscribers will be notified. In this way the controller can call back any object that is interested in the update.

4.3   Observer Pattern and Game Listeners

An Observer pattern is a popular paradigm that is frequently used in object oriented programming. It is especially useful when it is coupled with an MVC architecture. As the name suggests the main responsibility of an Observer pattern is to observe or monitor the changes of the observed objects. Observed objects can be any objects that are interested in getting a callback when a cetrain type of event takes place. In the project game, an Observer pattern is utilized to update the game world data and views as soon as a new game message from the server is received and parsed.
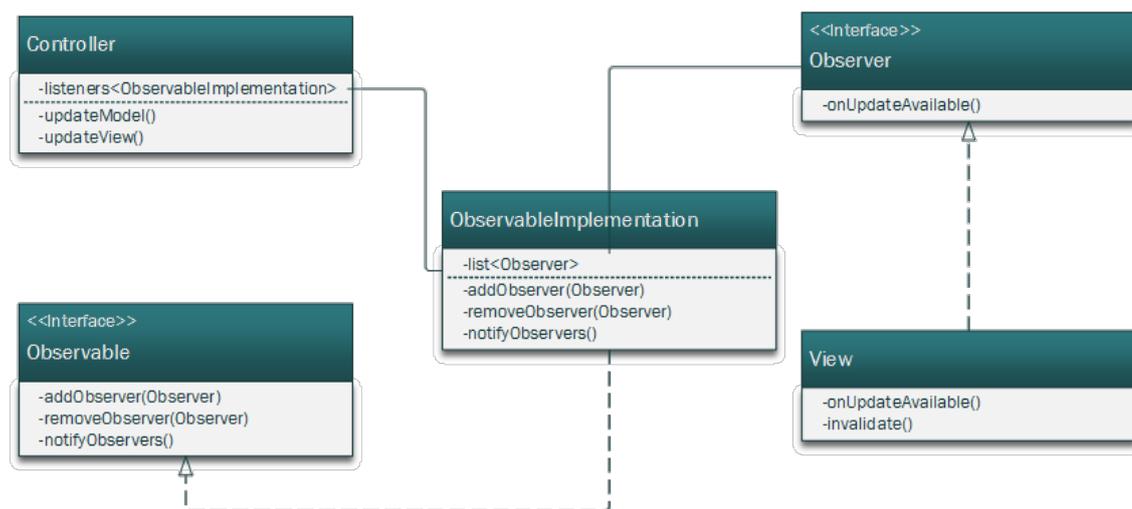

Figure 11. Observer pattern design diagram.

Figure 11 shows a diagram of the generic structure and design of the Observer pattern. In order to implement the Observer pattern, one needs an Observer interface that contains a group of methods that may be useful to invoke during update routine. The Observer interface should be implemented by all classes that wish to be notified once the event dispatcher fires. An object that has implemented the Observer interface becomes the Observer itself and should override the required methods.

The Observable interface contains a list of methods aimed to manage and control the state and population of the subscribed Observers. This interface should be implemented by a class that will store a list of registered Observers and override common functionality methods. The functionality should imply the possibility to register, delete and trigger a callback routine. In most cases, the reference to the implemented Observable class belongs to the objects that coordinate the update process. Usually this role is performed by controllers. [17]

Each controller type stores the reference to the public Observable object. The Observable object also belongs to a particular type as it stores a list of Observer objects that are of a particular type. In the game, an Observer interface is called a listener of some type. For example, a registration observer is called a *RegistrationListener* interface. An Observable class is called *RegistrationListeners*.

When the *RegistrationActivity* view needs to subscribe for registration related news and updates, it implements the *RegistrationListener* interface and overrides the required methods. Now a callback can be routed directly to this view when the server sends any registration scope data. Upon receiving a notification from the Observable object, a view can update the UI data and retrieve new data from the model class.

Figure 12. Architectural style and structure of the game's Observer pattern.

Figure 12 illustrates a shortened version of the class diagram of the game architecture's Observer pattern. The game uses a slightly optimized version of the Observer pattern. Instead of implementing an Observable interface, a generic class called *ListenersAdapter<T>* is extended by the Observable implementation class. In the example

illustrated in Figure 12, the Observable implementation class is *FightListeners*. The *ListenersAdapter<T>* uses encapsulated and implicit Observers notification routine performed by applying the concept of the Anonymous Interface. As can be seen from the diagram, the *ListenersAdapter<T>* contains an overloaded methods called *callAllListeners(…).* Each overloaded method expects to get a reference to the class that implements the Anonymous Interface and a particular number of arguments.

A skeleton of the anonymous interface content is protected and resides within the local scope of the *ListenersAdapter<T>* class. *FightListeners* inherit the properties from the *ListenersAdapter<T>* class and override the methods from the *FightListener* interface. Inside each method, instead of looping through the content of the Observers list a method *callAllListeners(…)* is invoked. It passes the reference to the anonymous interface object that was created within the same class instance and sends data as extra parameters.

```
protected final <Arg1> void callAllListeners(Function1<T, Arg1> e,
Arg1 arg) {
        for (int i = listeners.size()-1; i >= 0; --i) {
            T listener = listeners.get(i);
            if (listener == null) listeners.remove(i);
            else e.call(listener, arg);
        }
    }
```
Listing 1. Notifying the Observers.

Listing 1 shows an example of the overloaded method *callAllListeners.* When the controller calls a method *onFightMessage* and supplies *DuelMessage* object as a parameter, within the scope of the *FightListeners,* a method with the same name will be invoked. It has only one line of code as shown by Listing 2.

```
@Override
public void onFightMessage(DuelMessage message) {
        callAllListeners(this.onFightMessage, message);
    }
```
Listing 2. The implementation of the *onFightMessage* within the *FightListeners* scope.

As can be seen from Listing 2, when *onFightMessage* within the *FightListeners* scope is called by the controller, it will not directly trigger the notification process. Instead, it calls a protected method *callAllListeners* that is accessible and visible since *FightListeners* is a child class of the *ListenersAdapter<T>* parent.

The method *callAllListeners* iterates through the collection of Observer objects and using the anonymous interface instance performs a callback to the Observer. Compared to the general implementation of the Observer pattern, where the update dispatcher has an explicit relationship with the Observable implementation, the architectural structure of the Observer pattern used in the game applies an extra layer of visibility. This method involves an indirect Observers notification, through the encapsulation provided by a generic and universal solution in the face of *ListenersAdapter<T> class.*

4.4   Client-Server Communication

Networking operations are managed by the *NetworkController* object. The object acts as a request and response dispatcher and in addition to that provides a common functionality and utilities for performing network tasks.
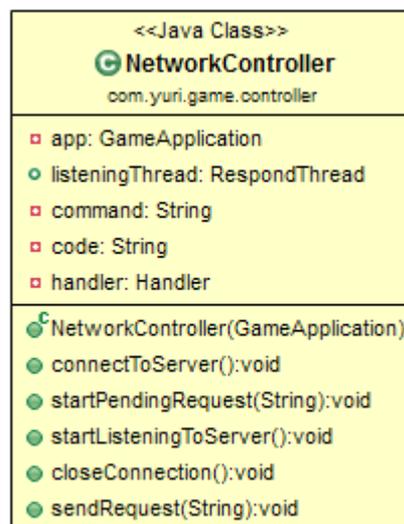
Figure 13. *NetworkController* class diagram.

The class diagram of the *NetworkController* object is depicted in Figure 13. The controller has access to the global application scope. It controls the flow of network operations and uses the input and output streams from the client socket to communicate with the server. A *Handler* object is used to provide an interface between a UI and background threads. The *Handler* allows to fetch the message from the messaging queue and sends it to the next recipient.

NetworkController works in contact with *Request* and *Respond* threads. As could be guessed, one is responsible for delivering a request message to the server while the

other acts as a local respond dispatcher whose role is to listen to the server messages. The *command* and *code* variables store the message content and code respectively. An access to the client socket is obtained through the *app* variable, which belongs to the application scope.
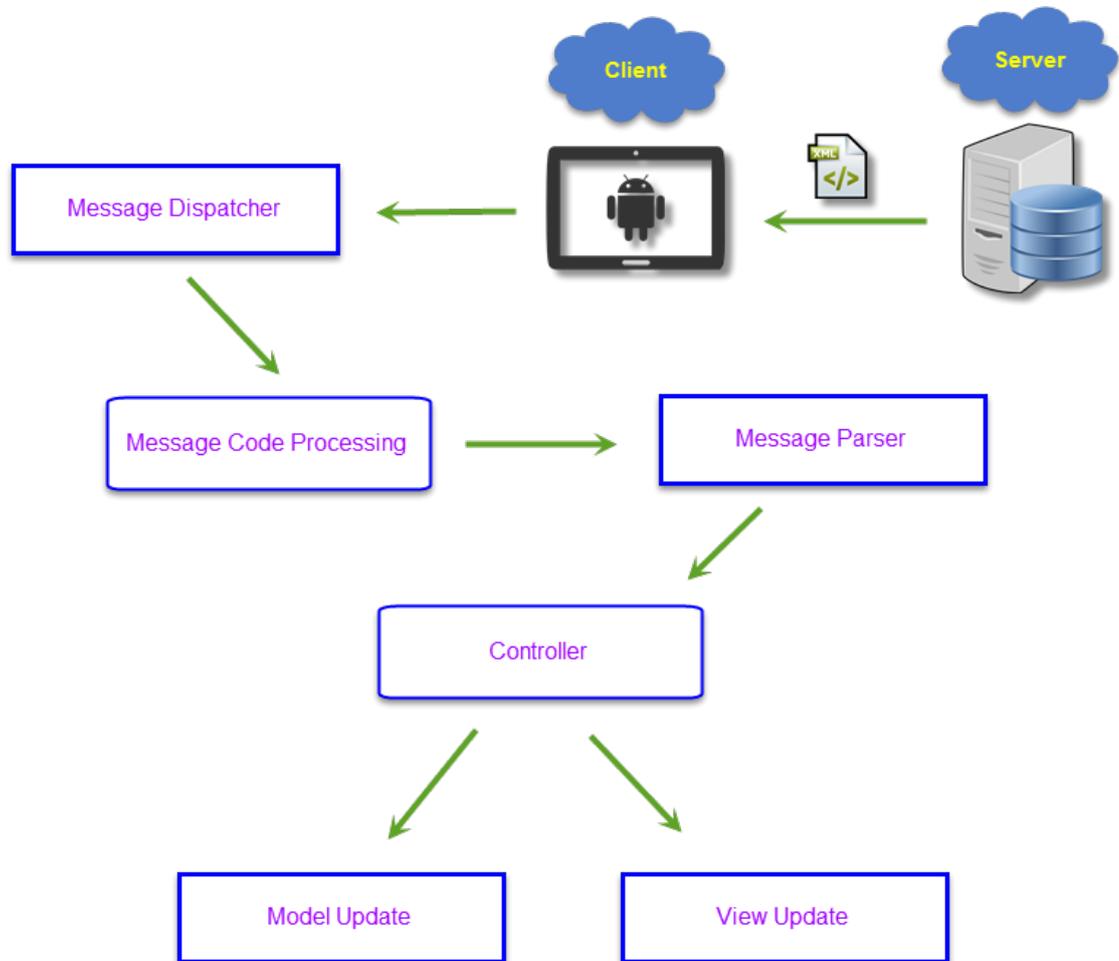

Figure 14. Processing a server respond message.

Figure 14 describes the flow of steps that the client takes when it receives a message from the server. The server message is detected inside the server listening loop. The message is sent through a callback implementation to the *NetworkController*, which in turn decides which Performer should get this message. The message is passed to the *ServerMessageHandler* class that matches the message code against a list of possible codes. When the nature of the message becomes identified, it will be sent to the parser that can parse this type of message.

After the parser deciphers the message and converts it from XML format into the Java language, it checks the code of the message and based on the value decides what type of controller should be notified. The parsed message is passed to the qualified controller. The controller takes further steps to determine the fate of the message. The mission of the controller at this stage is to update the game world data model and to notify Observers about the change.

4.5   The Game Messaging Protocol

A client cannot communicate with the server without a set of rules, agreement pattern and a common language that could be understood by both parties. For that purpose an XML communication protocol was developed. The choice of XML language was determined by a common proficiency and knowledge that we possessed about that language and the tools that were available for parsing XML data. As a matter of fact, XML is one of the preferred choices for data transmission over the Internet. It is highly readable, lightweight, scalable, and flexible with outstanding community support.

Before the client can understand the meaning of each message that arrives from the server, it should translate it into a familiar object-oriented language such as Java. Data conceived in an XML message is structured according to the XML language rules and standards. Each object present in an XML message is typically surrounded by tags that define the data range of a specific tag and enables the parser to identify its data scope. A general XML message structure follows a parent-child relationship and is interpreted by a parser running a recursive algorithm.

```java
public final class ServerRespondCodes {

// System Error Message Codes
public final static int SOCKET_TERMINATED = 0;
public final static int WRONG_MESSAGE_STRUCTURE = 1;
public final static int WRONG_COMMAND_ORDER = 2;
public final static int MISSING_PARAMETERS = 3;
public final static int PARAMETER_NOT_FOUND = 4;

// Registration Message Codes
public final static int SUCCESSFUL_REGISTRATION = 10;
public final static int USERNAME_EXISTS = 11;
public final static int EMAIL_EXISTS = 12;

// Login Message Codes
public final static int LOGIN_SUCCESSFUL = 13;
public final static int LOGIN_FAILED = 14;

// Client Closes Socket
public final static int SHUTDOWN = 19;

// Actor and Player Message Codes
public final static int PLAYER_STATS = 101;
public final static int ACTOR_STATS = 102;
public final static int CHARACTER_NOT_FOUND = 103;

// Location Related Message Codes
public final static int WRONG_LOCATION_COMMAND = 300;
public final static int PLAYERS_LIST_IN_LOCATION = 301;
public final static int NEW_PLAYER_IN_LOCATION = 302;
public final static int PLAYER_GONE_IN_LOCATION = 303;

// Duel Related Message Codes
public final static int DUEL_REQUESTS_LIST = 401;
public final static int NEW_DUEL_REQUEST_ADDED = 402;
public final static int DUEL_REQUEST_REMOVED = 403;
public final static int DUEL_APPLICATION_FULL = 404;
public final static int PLAYER_ACCEPTED_DUEL_REQUEST = 405;
public final static int PLAYER_REJECTED_DUEL_REQUEST = 406;
public final static int DUEL_TO_START_NOT_FOUND = 407;
public final static int DUEL_START_FAIL_SOMEONE_LEFT = 408;
public final static int DUEL_TO_REMOVE_NOT_FOUND = 409;
public final static int DUEL_TO_ADD_NEW_PLAYER_NOT_FOUND = 410;
public final static int DUEL_TO_REMOVE_PLAYER_FROM_NOT_FOUND = 411;

// Fight Codes
public final static int FIGHT_REGISTERED = 1001;
public final static int FIGHT_UNREGISTERED = 1002;
public final static int FIGHT_MESSAGE = 1003;
public final static int ATTACK_STATE_MESSAGE = 1004;
public final static int ENEMY_INIT_STATS_MESSAGE = 1005;
}
```

Listing 3. Code list of the communication protocol.

It was decided to equip each XML message with a numerical code value, which would describe the content of the message and imply the possible actions that should be taken upon receiving this message. A list of codes is presented in Listing 3 and is shared between the server and the client, and the client processes each message based on the code that is present in the list. For example, a message with code 10 means that the registration was successful. The server generates the message to the client as soon as it receives the message from the client, which contains some information. This information will be processed and validated. As a result, the server generates its own message, assigns a code to it and sends it back to the client.

There are two types of messages implemented and supported by a communication protocol: system messages and game messages. System messages carry information related to the system scope, and game messages contain information that belongs to the game scope. For example, during system events such as authentication or account registration, the client application will generate a request message attaching data filled in by the user.

Both system and game message requests include key tags: command and parameters. A command tag defines an action status or service that the client requests from the server. In other words, a command tag describes the intention of the client and what it expects the server to do. Each parameter is stored in a list of objects in the form of key-value pairs that describe the details of each command. Thus, for example, if the client asks the server to register a player, the client will send an XML message with *register* command and credentials in the form of key value pairs.

For the purpose of request generation, a utility helper class called *ServerRequestFormer* was written. The main task of this class is to convert data from the Java language into an XML format according to the communication protocol standard. This class contains dozens of static methods, which can be invoked without the need to instantiate the class itself. Each method accepts a number of arguments which are used to generate the XML message returned in a string format.

When the server sends a message, it will be received inside the server listening loop. The code is extracted from the message and checked against a set of commands that may signify the communication link termination. In case the code does not match any of such commands, the message will be passed to the class called *ServerMessageHandler*,

which identifies the meaning of it based on the supplied code. As soon as the nature of the message becomes evident, the program will pass it to the corresponding parser class, which will interpret it from the XML language into the Java language.

5   Implementation

5.1   Authentication

The authentication module consists of two parts: login and registration sections. The purpose of the login section is to provide the functionality of accepting the user's credentials and validating them. The client sends the user's credentials to the server that in turn validates them and provides a positive or negative reply. My task was to design and develop the client side functionality of these sections. The task involved building the UI by creating and inflating the layout, positioning Android widgets, setting up the resources and programming the logics.
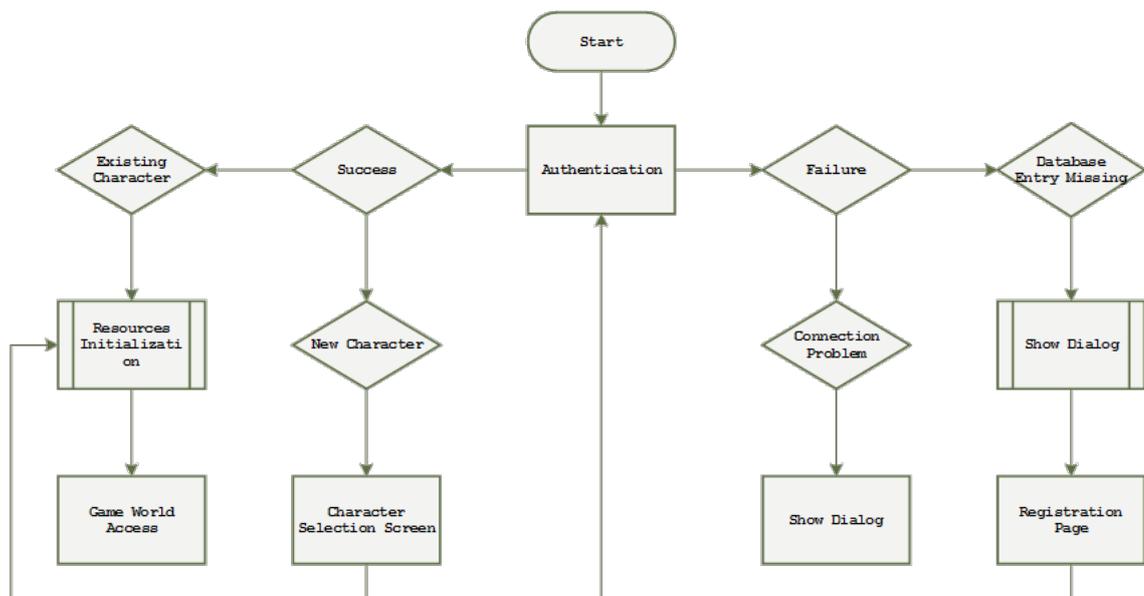
Figure 15. Authentication module flow chart.

The functionality of the authentication module is illustrated in Figure 15. The flow diagram states that two scenarios are possible during successful authentication. If a user has an existing account, he/she will be redirected to the game world instance after the server validated the supplied credentials and replied positively. Before the user can access the game world, the operation of the resources initialization takes place.

In case the user does not have an account, he/she will be redirected to the registration activity, which provides a GUI form with fields to be filled. Upon completing this procedure a new activity will be opened, and user will be asked to select a character. After the character is selected, the user can access the game world.

The failure implies two scenarios. If a connection problem is detected, the user will be informed via a popped up GUI dialog box. A failure may occur if the credentials do not match an entry in the database. In this case, the server will respond with a failure message and supply a code. Based on the communication protocol and code list the client informs the user about the problem. If user does not have an existing account, he/she will be redirected to the registration activity.

A login section was implemented using *LoginActivity*. It inflates XML layout that contains the following widgets: *TextView*, *EditText*, *Button* and *ViewGroups*. Two types of listeners were created and implemented by *LoginActivity*: *LoginListener* and *SystemErrorListener*. The login listener interface contains the following methods: *onLoginSuccess()* and *onLoginFail()*. The first method will be called if the server responds with a success code; the other one signifies a failure, which might be caused by wrong credentials.

A system error listener is an interface that contains the following methods: *onSocketTerminated(), onWrongRequestStructure(), onWrongCommandOrder(), onMissingParameters(), onParameterNotFound()*. If the connection between the client and the server is broken, *onSocketTerminated()* will be invoked. The rest of the four methods serve the purpose of protection against spam and flood requests, which contain some errors in their structure or some missing parts.
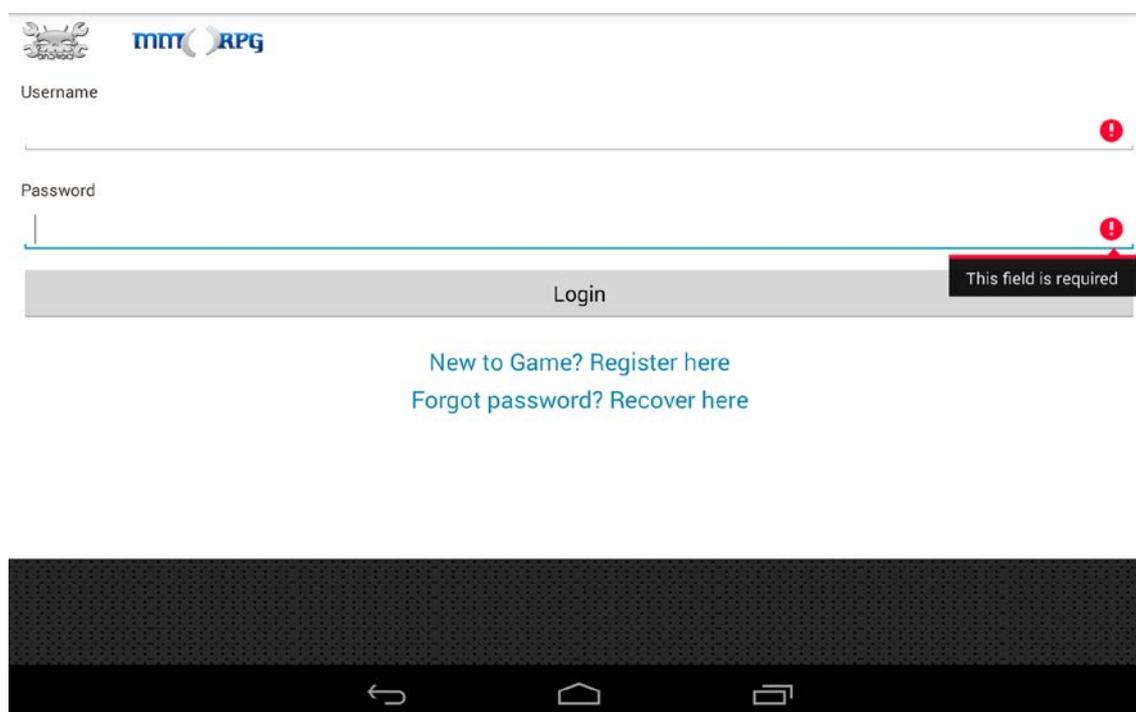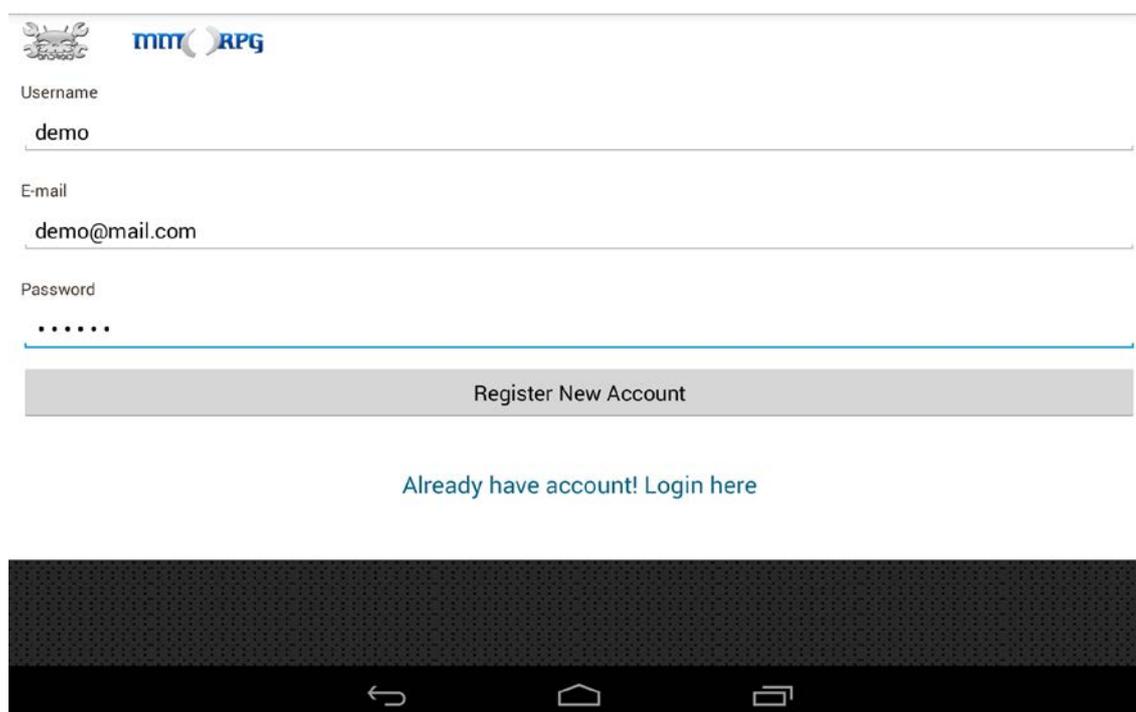

Figure 16. *LoginActivity* user interface and the field's requirement.

The *LoginActivity* implements both interfaces and overrides all necessary methods. Each callback method contains a sub-routine that executes a piece of programming code aimed to deal with a specific scenario. In order to set some restrictions on input field data, I used text and input length utilities to determine if the input was empty or if its length was smaller than the required value. The example of the *LoginActivity* user interface and the field's requirement filter is displayed in Figure 16.

If the input data meets the requirements, the program will check if the Internet connection is available. If the result is positive, a runtime logic will check if the client has an active connection with the server. If the client is connected, a helper class will generate a request and an XML string will be sent to the server. In case the connection is not established, the client will try to open the connection socket.

If the server responds with a success code, the player can access the game world. In other cases a dialog with a descriptive message will be displayed. If the user has no active game account, he/she can proceed to the registration activity by clicking text link at the bottom of the screen.

Figure 17. The UI of the registration activity.

Figure 17 shows the implemented registration activity and its user interface. The registration activity is built in the same style as the login activity. Instead of implementing the *LoginListener*, the registration activity implements the *RegistrationListener* interface, which includes the following methods: *onSucessfulRegistration(), onUsernameExists(), onEmailExists()*. If the registration was successful *onSucessfulRegistration()* will be called. In case a username exists, *onUsernameExists()* will be called, and *onEmailExists()* will be called if an email exists.

Overall, the rest of the operations are developed in a similar style as *LoginActivity*. User data input is checked against the empty field and length restrictions. If any of those restrictions are not met a descriptive *JQuery* style message will be displayed, as shown in Figure 18. In order to notify the user about some error or event, I developed a dialog box that is fired when required.

Custom listeners are registered inside *onResume()* method and unregistered inside *onPause()* method. It is important to carry out this routine because when some event fires the notification chain, only active listener methods should be called in order to avoid duplicate calls for registered targets that do not hold focus and lie in the background process scope. For example, if the user navigates from one activity to another and both activities implement the same type of listener and register for the same events, both will be notified although the previous activity is stopped and cannot operate at that moment. Hence, the risk of unpredictable behavior will be very high, if this situation is not treated properly.

When the user registers an account, the server will check if the local database does not contain an entry with user credentials. If so, it will notify the client that the registration was successful. On the client side *onSucessfulRegistration()* will be called, and the player will be redirected to the character selection activity.

5.2   Character Selection

Before the player can access the game world, he/she should choose a character that would represent him/her in the game world. There are four races available for selection. Currently, those races are Human, Elf, Orc and Undead. Each race has its own history, storyline, features and strong and weak sides. The players can choose male or female characters, and their visual properties alter according to the selected gender.

The overall look and feel of the character selection activity is based on *ViewPager* and *PagerTitleStrip* components provided by *android.support.v4.view* library. Each character race occupies a fragment, which inflates the XML layout. In total, there are four fragments managed and controlled by *FragmentPagerAdapter*. A combination of *ViewPager*, *Fragments* and *FragmentPagerAdapter* creates a visual effect of horizontal page swiping. The player can navigate between different character races by swiping left and right. A simple and straightforward logic enhances user experience and adds interactivity.

FragmentPagerAdapter is an adapter type available in the default Android library aimed to cache, group, manage and present fragments as a collection of views. It is useful in any situation which involves interaction with multiple fragments simultaneously. *FragmentPagerAdapter* can be used by creating a class which extends it and overrides the required methods. The basic methods are *getItem(), getPageTitle() and getCount().*

The method *getItem()* receives a position as a parameter and should return an instance of the created fragment. The runtime logic inside the switch block determines what type of fragment will be created and returned. In order to pass additional arguments, a bundle can be used. The method *getPageTitle(),* as its name suggests, returns a page title by the provided position and the method *getCount()* is used to determine the number of fragments that are cached and stored by an adapter.

When *FragmentPagerAdapter* is instantiated, its reference is passed to the *ViewPager* component. *ViewPager* is a complex and advanced widget, created by Google engineers and included in the public Android library. *ViewPager's* functionality is based on utilizing an adapter that manages data collection. Hence, both *FragmentPagerAdapter* and *ViewPager* operate as a team. The first one provides a proper fragment, keeps track of any changes and counts the available fragments while the second one animates the transition between fragments, measures the fragment view and draws it on the screen.
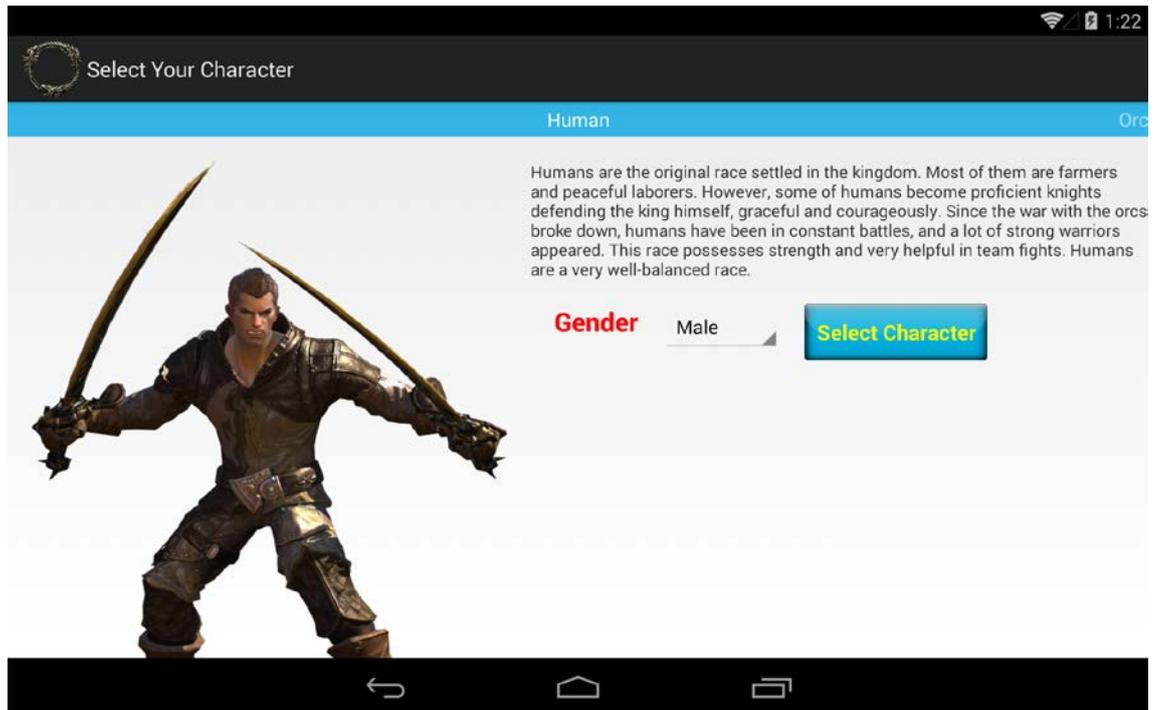
Figure 18. Character selection.

All four character races are displayed within the same XML layout consumed by the fragment class. Since the design of each character page does not deviate or distinguish from the rest, there is no point of writing four unique XML layouts. The fragment class communicates with the parent activity via *OnCharSelectedListener* interface implemented by parent activity, which signals when the user selected a character, by calling the overridden method. I wrote a few logic blocks of code that determine what gender and character were chosen. The implemented functionality of the character selection routine is illustrated in Figure 18.

A gender is displayed by the *Spinner* widget. Gender data is fetched from the string array XML resource. Gender *Spinner* is hooked up to the listener, which notifies it about the gender the user selected. Each time the gender is altered, a character image corresponding to the selected gender will be displayed. When the user submits data and clicks the button, this event will be detected by the click listener, after the race and gender are passed to the parent activity via the callback method. As soon as the parent activity will receive user data, it will generate and send a request message to the server.

If the request was successful, the server will examine the request structure and reply with a message that provides the initial game world information: character stats and de-

fault location with the current list of players within that location. By default, all newly created characters start in the Training Room location, and the initial stats are generated by the server. The program will not redirect the character into the game world until it receives all the required information in order to prepare presentation data for the next stage.

5.3 Character Details

From the design point of view, the game world interface is embedded in *MainActivity*, which uses the *ViewPager* technology. Each section of the game world is represented by the fragment consumed by *ViewPager*. A scrolling mechanism allows to switch from one section to another with just a few swipes. An Action Bar with tabs is positioned above each fragment. Each tab is equipped with a descriptive heading and icon, which helps the player to navigate through the game sections.

MainActivity serves the purpose of hosting the main components and objects, required for binding different views and widgets. It extends *FragmentActivity* in order to utilize *ViewPager* correctly and supports fragments manipulation. *MainActivity* implements *ActionBar.TabListener, LocationListener, ActivityCallback, DuelRequestListener, FightListener, OnPageChangeListener, ActorStatsListener* interfaces, which are required for controlling the flow of the game.

In order to create a connection between fragments controlled by the adapter and tabs managed by the action bar, I created a mechanism which allows to cache tabs that match the fragments. Each fragment is bound to the tab. In fact, this is not an explicit relationship but an implicit one, implemented by the static class *TabInfo*, which contains the information about the fragment and arguments. Whenever a new tab is created, *TabInfo* will be instantiated and bound to the tab as a tag.

A list of *TabInfo* objects exists inside *MainActivity* and is used to manipulate the state of tabs and fragments. *TabInfo* class contains the name of the Fragment class to be instantiated and arguments to be supplied. This approach allows to replace existing fragments by just changing the references inside *TabInfo* class and calling *notifyDataSetChanged()* inside the adapter object.
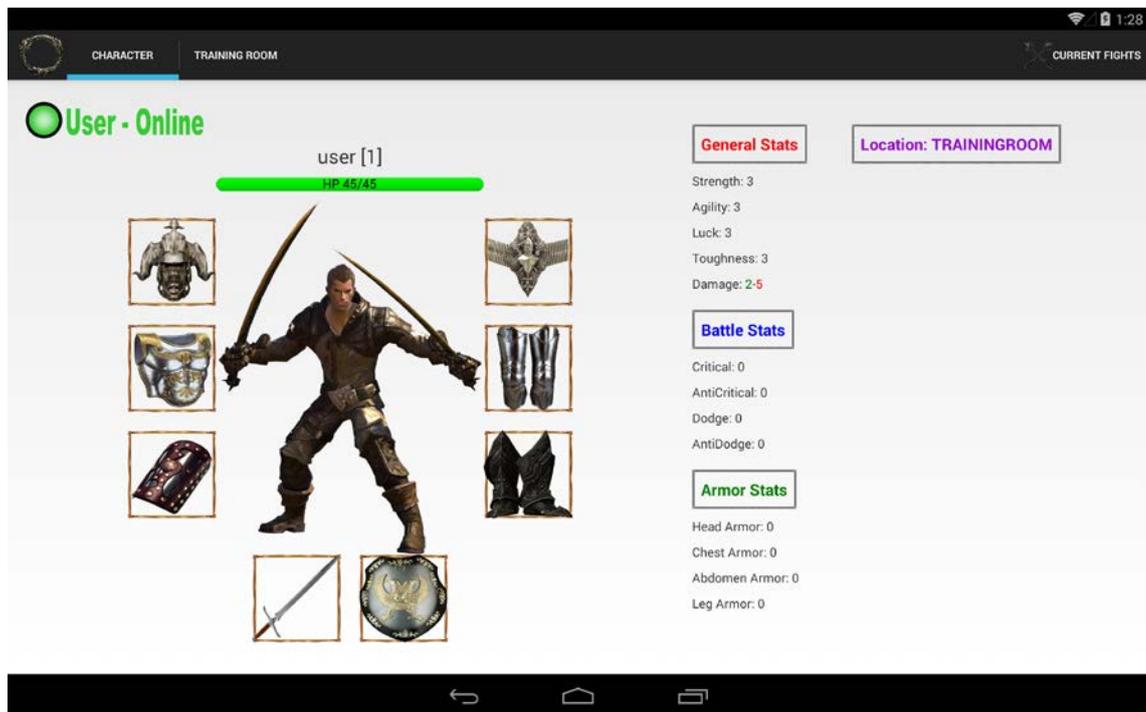
Figure 19. Character inventory, items and stats.

As the player enters the game world, the first section that he/she sees is the character details view, inflated and exposed by the *CharSelectFragment*. An example of this view is shown in Figure 19. A layout used to design this page is split up into two panels. The first panel provides a UI for character inventory, icon, health bar, name, level and online status. The second panel hosts a list of *TextView* widgets that describe the character stats and show current character location.

5.4 Locations

It was decided to create a list of locations inside the tab of player's current location. Thus, a player can see a list of available locations, scroll it down and move to another location by picking one from the list. Touching a location image, a dialog pops up. The purpose of the dialog is to simulate a transition. A progress bar and target location will be displayed to the player. By default, moving to another location will take 10 seconds.
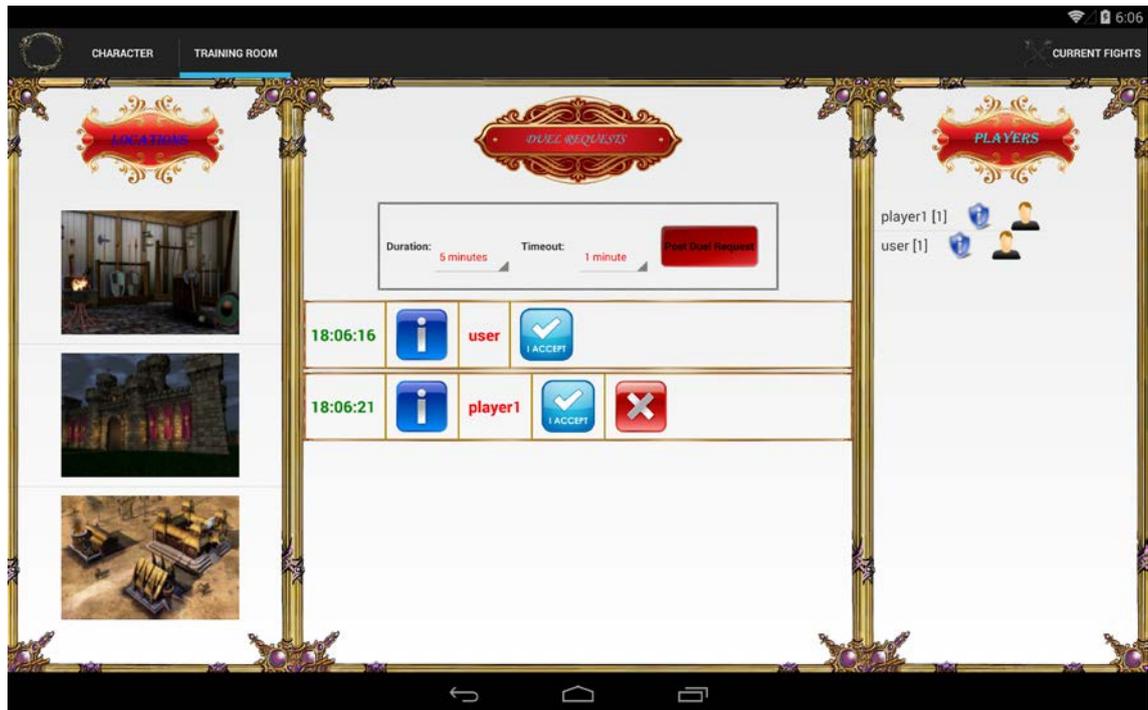
Figure 20. Locations: Training Room.

Locations, along with duel requests and location players list, are embedded in *Location-Fragment*. All three parts are divided into three UI view blocks as illustrated in Figure 20. The left block is occupied by a locations list, the middle block is implemented in the form of a fragment nested into the parent fragment. It hosts the UI for a duel request list and a fight configuration menu. A nested fragment is used because it provides the possibility to update a group of views with a button click. Thus, the middle UI block space can be utilized for showing a duel request list and fight configuration menu. Finally, the right UI block is reserved for a list of online players present in that location.

A locations list was implemented by using the *ListView* widget. An adapter was created and it extends the *Base Adapter*. This is the simplest form of a view adapter used when no data collection binding is required, and the total number of items is known beforehand. In our case, we had a fixed number of locations. Each view entry is designed as a row containing only one widget, *ImageView*, which hosts the image of the location.

The middle block is the most complex compared to adjacent blocks. The layout space that covers the middle block is used by *FrameLayout*, which serves as a container for a fragment. By default, a *DuelRequestFragment* is used. It contains the fights configuration block and *ListView*, which lists all active duel requests in the current location. *DuelRequestList* extends the *ArrayAdapter* of *DuelRequest* objects. A list is implemented in a

custom fashion, with all row items as separate objects registered for UI events such as touches and clicks. A player can have only one active duel request. He/she cannot accept his/her own request, but he/she can delete his/her request at any time. When a duel request is created, it will visible for other players. If the request is accepted, the duel request owner will be notified by a popped up dialog. If a duel owner agrees to fight a person who accepted his/her duel, a fight instance will be created, and both players will be moved to a fight activity.

DuelRequest ListView contains the following components: time of registration, owner name, current status, accept and delete buttons. The first parameter is time value, the exact time when the duel request was created and registered by the server. The information icon enables other players to get details about this duel request such as timeout, duration and owner information. The owner name means a person who created this request. The *Accept* button is used to challenge the owner and *Delete* button is available only to the owner of the duel request and it is used to remove the duel request from the list.

A menu button, located in the top right-hand corner, provides a mechanism of toggling the middle container fragment, replacing it with a current fight fragment. This fragment contains a *ListView* with active fights entries. Each entry is a view with a specific design. The list row contains the following elements: time, red and blue players, name and levels. The right-hand block represents the list of players in the current location. Each list entry contains the following elements: player's name and level, information icon and gender icon. By touching or clicking on the information icon, a dialog with the player's stats and inventory will show up. The players can inspect the gear and each other's achievements. The gender icon means if the player is a male or female.

The trickiest and the most challenging part was to come up with a solution that would allow to identify and get a reference to the *ListView* row that contains the event triggered widget. For example, when the user touches the information icon from the *ListView* row, this event is detected by the event dispatcher. In order to apply runtime logic and handle the event a reference to the view that hosts the event triggered widget should be obtained. However, the *ListView* row clicking listener is deactivated since the aim was to provide the row view widgets the functionality to handle and detect the events rather than creating a listener that would listen to the whole row interaction.

In order to solve this problem I used the *ViewHolder* pattern, containing references to all *ListView* objects and keeping track of their positions. Each time *ListView* adds a new *View*, a new *ViewHolder* object is created and attached to that View as a tag. When a player interacts with a specific widget in a certain list row, the listener will return a reference to the *View*. Unfortunately, this View may not match with the *View* the user was interacting. Retrieving a tag *View* and passing it to a recursive function returns a reference to the correct *View*.

5.5 Fight Instance

When a player accepts a duel request posted by another player, a duel request owner will be notified by the server in the form of a popup dialog. If he/she agrees to fight the challenger, the client will send a request to the server, which means that the duel application has been successfully created and awaits a transition to the fight instance. A positive response from the server will move both players into a fight instance.

A fight instance is implemented in form of Activity. Although a fight instance could be implemented as an extra dynamic tab within the Main Activity, it was decided to make it separate because hosting it in the same Activity would result in some limitations and drawbacks, for instance, assume we decided to implement fight instance by generating extra tab. From the navigation point of view the player would easily move from a fight instance to another game module by just swiping left or right. However, the *ViewPager* mechanism works in such a way that if the user swipes more than one tab left or right a fragment will be destroyed, and all data will be lost. Thus if the player will decide to move from an active fight tab to a character details tab, the fight data will be lost.

Of course, it is possible to invent a mechanism, which would cache the fight instance state and restore in from the last state. However, this approach would require much more effort and resources. In order to keep things simple, it was decided to use an extra Activity. A player may be notified if he/she tries to leave the Fight Activity. On the other hand, even this approach would require, at later stages, extra synchronization and fight state caching.

A Fight Activity is aimed to operate in the landscape mode in order to utilize the screen space more efficiently. The UI is divided into three blocks: left, middle and right. The left

block contains the player's character image, health bar, name and level. In order to customize the progress bar and convert it into a health bar with extra text drawn on top of it, I created my own widget. It is a class called *HpBar* that extends the Progress Bar. It supports custom attributes and utilizes them through a custom namespace.

The health bar is updated according to the data value obtained from the server message. During the fight players actions are interpreted by server messages. Each player should select one out of four body parts that he/she wants to attack and two body parts to protect. There are four body sections that may be targeted: head, chest, abdomen and legs. The server waits until both players finalize their actions. Then the server calculates the outcome using a mathematical expression and sends the response.

The middle UI block serves the purpose of providing an action control panel and displaying the outcome of fight events in the form of a list. An action control panel is implemented as a nested fragment. It hosts an attack icon, a group of body part attack targets and a group of body part defend targets. The player can choose only one body part to attack and two body parts to defend. By clicking the attack icon, he/she finalizes his/her decision and the client sends the data to the server.
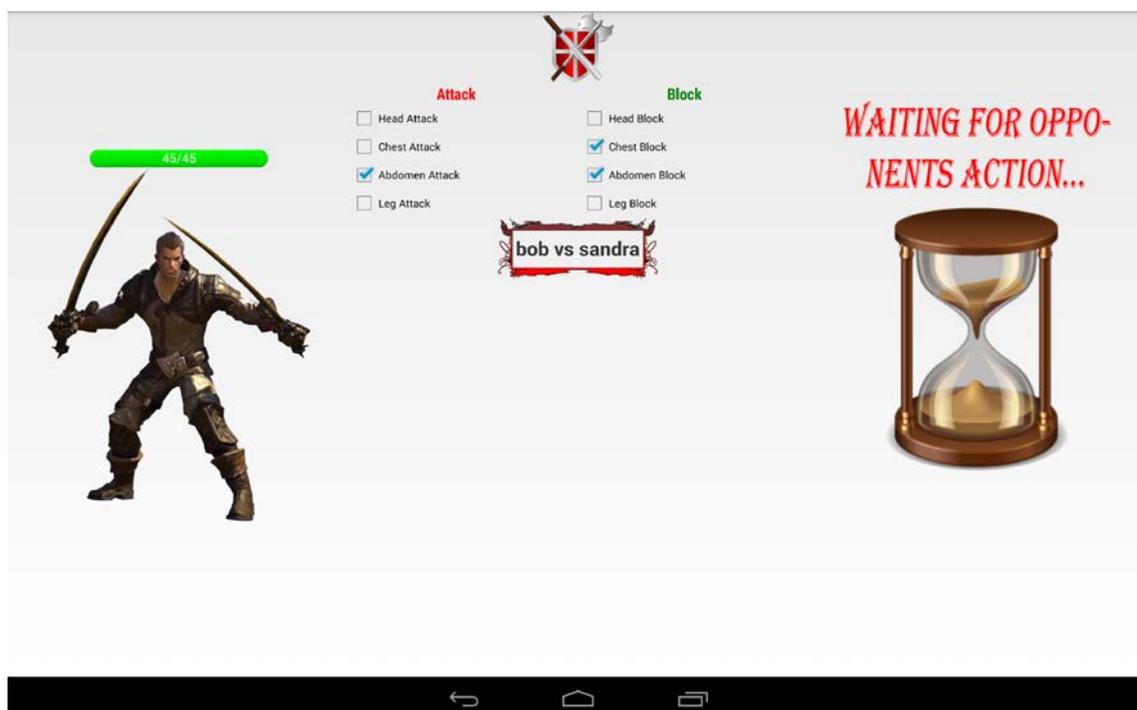

Figure 21. Fight in progress.

When the server receives a response from all clients located in the current fight instance, it will calculate the outcome and generates a new message that should be parsed and displayed by clients. The server launches a timeout countdown, shown in Figure 21, as soon as it receives the first fight message. If the server does not get a response from another player when timeout expires, it will generate a penalty message, which allows the first player to finish the fight immediately by killing the opponent instantly. This type of sanction or punishment will be applied only if the second player stays idle longer than a timeout duration.

Fight messages are processed by the Simple API for XML (SAX) parsers. After the message is interpreted from XML to the Java language, it will be cached into the *DuelMessage* object. It is a POJO class which contains fields for storing general information about the fight such as date, fight state, fight event and the first and the second attacks. A list of *DuelMessage* objects is used in a list adapter. All fight messages are displayed in a *ListView*, placed below the action control panel. It exposes data from the adapter whose reference it holds. The adapter is a simple class that extends *ArrayAdapter* of the *DuelMessage* type.

The right UI block mirrors the left block except that the data it contains is related to the enemy player. This block is implemented as a fragment container, because all UI components are changed during a fight progress. Thus, when a player submits information about his/her offensive and defensive choices, the content of the left block is swapped with a sand clock image, which means that the system is waiting for the enemy's player turn. Upon completing the fight, a victory or defeat fragment is displayed.

6   Testing and Troubleshooting

Testing and troubleshooting is an inevitable part of any serious application development process. Each developer should have expertise and skills in debugging routine. There are no perfect applications. All applications contain some vulnerabilities and are subjects to malfunction to some extent. It is a priority of a developer to discover and fix all problems to ensure proper functionality. A process of testing and troubleshooting the game was divided into three stages:

- Resolving coding problems and debugging
- Testing implemented functionalities
- Testing the game with real Android devices.

Before testing the whole application, each of us carried out particular testing routines aimed to discover any inconsistencies, problems and vulnerabilities within the scope of the client and server architectural structures. During the testing stage, a dummy data was used to simulate a particular situation and analyse the behaviour of the application. In addition to this, stress testing was applied to evaluate the execution flow under critical circumstances.

The server software architecture was tested to withstand the load of hundreds of simultaneous connections. Furthermore, the heap state was analysed to spot any memory leaks or excessive memory usage. The client software architecture was tested by imitating different conditional situations that may cause some ambiguous outcomes.

6.1   Android Debugging Tools

The main tools for resolving technical problems such as code errors and bugs were provided by Eclipse IDE and Android Debugging Toolkit (ADT). In order to get access to the Eclipse debugging features, the ADT plugin should be downloaded and installed. ADT plugin creates an interface between the Android platform and Eclipse framework. The Eclipse debugging functionality is split into two modules: Debug Perspective and DDMS Perspective.

The debug perspective provides a visual representation of UI blocks aimed to display and monitor the flow of the debugging process. Each Eclipse perspective in general represents a set of views that feature UI utilities for particular tasks. [18]
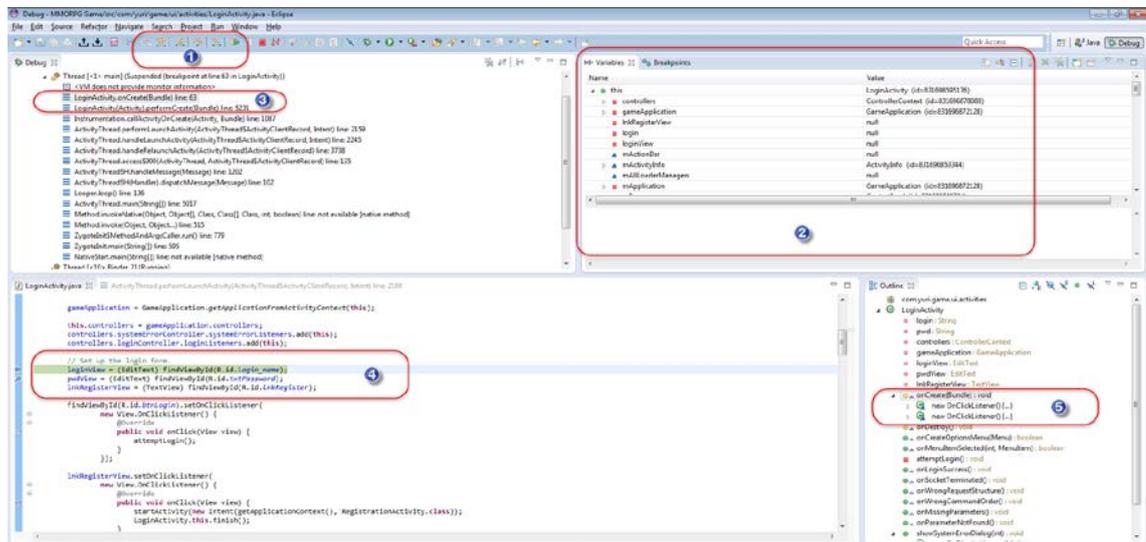

Figure 22. Eclipse debugging perspective.

Figure 22 shows the general look and feel of the Eclipse debugging perspective. The perspective is divided into five key areas. The areas enclosed in red rectangular boxes provide the most important information and functions of the debugging analysis. A group of action bar buttons enclosed in the box marked by number 1 represent the key action buttons that help to manipulate the flow of code execution. These buttons enable to jump from one code line to another, jump into a particular method or class, return to the previous position and resume the execution.

The right-hand upper view shows a list of objects and their values. Each Java class has local instance field variables. Since Java is a high-level programming language, it encapsulates the explicit memory management that is controlled by pointers. Hence, a Java developer cannot use the pointer syntaxes and explicitly apply the pointer arithmetic. Nevertheless, most of the class field variables are encapsulated pointers that store the memory address of the object they reference. A developer can monitor the state of objects in the area marked by number 2.

The left-hand upper view contains the a of the current application debugging threads. Each Android application has several default library threads that are running in the background. Marked with number 3 is a group of methods with set break points. A developer

can set a break point at any part of the code by double-clicking it. Break points serve the purpose of a starting point for the debugging analysis. Break points belong to a certain method and class. These methods and classes are displayed in the left-hand upper window.

The bottom left-hand view displays lines of codes that are being examined. The current break point is highlighted by the green colour. By using a *Step Over* command, the runtime environment executes precisely one line of code and shifts the control to that line. During this action, the state of some objects may change and hence new values can be shown under the variables tab. In other words, a *Step Over* command allows to monitor the application state change after the next line of code was executed. The variables that were affected during the code execution change their values.

The bottom right-hand view offers a visual representation of all methods that are included in the class that is currently debugged. By clicking on the method name, the control is passed to the location of that method. In this way it is easy to examine the content of each method and set a break point. A rectangular red box marked with number 5 represents the methods that are being currently debugged. These methods contain the latest break points, and the code execution is focused on that area.

Another way to verify the correctness of the application code or resolve any technical malfunctions and bugs is to use the Dalvik Debug Monitor Server (DDMS) perspective, which provides several functionalities for Android debugging. In the DDMS debugging mode, a developer can analyse the heap state and its size, examine the state of threads, carry out a networking analysis, read data logs from the *LogCat* utility, take emulator screenshots and much more. [19, 153-158]
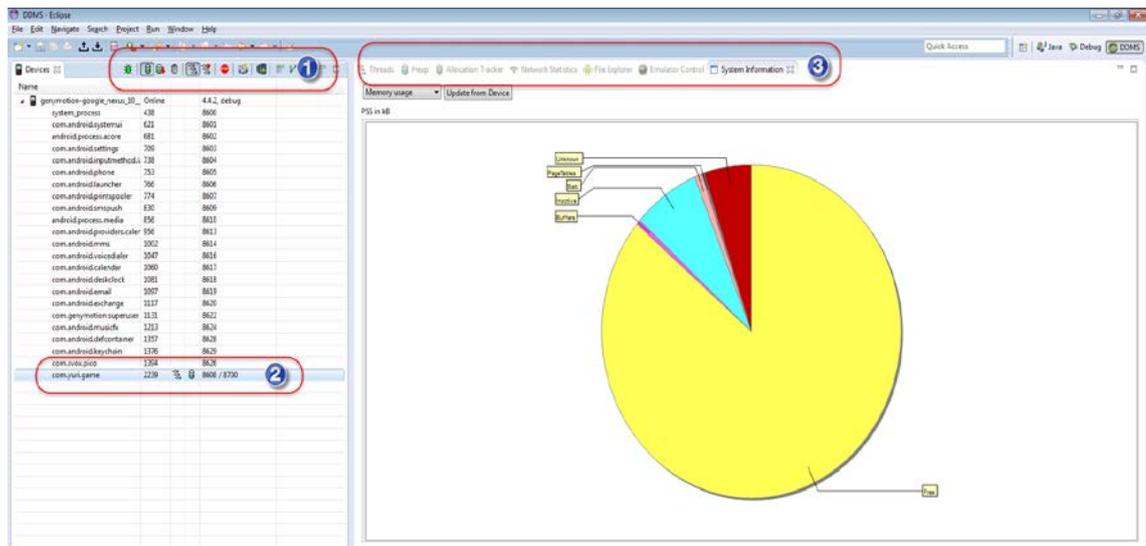
Figure 23. DDMS debugging perspective.

Figure 23 illustrates the key elements of the DDMS debugging perspective. The DDMS perspective separates the screen into two views. A set of control buttons and options is embedded in the action bar and tabs bar above each window view. The most important features of the DDMS module are enclosed in red rectangular boxes and marked numerically.

The left-hand view contains a list of processes running in the emulator. The process marked by number 2 and enclosed in a rectangular box represents the application we are interested in debugging in the DDMS perspective. A control panel marked by number 1 hosts a set of action buttons embedded in the action bar. The key functionalities of the control panel include a heap and threads update, various types of dumping and a screenshot option.

The right-hand view covers the majority of the screen space, and it is used to display the results of each form of the debugging routine. A tab bar positioned above the window lists the main categories of DDMS debugging. The Threads section shows a list of active threads that are spawned within the instance of the selected process. The Heap section shows the memory footprint consumed by the heap. Under the network tab, a developer can browse the current network activity and data transmission in the form of datagrams.

File Explorer displays process directories, folders and files. It is a useful tool when it comes to exploring data generated from input-output operations and tracking the resources hosted by the application process. The *Emulator Control* option enables to mimic

real device operations such as telephony and messaging. It is possible to route the call to the emulator's artificial phone and monitor the behaviour programmed by the developer. Finally, the last tab provides useful system information about the CPU load, memory usage and frame rendering time.

The last category of the Android debugging process involves the logging utilities, such as *LogCat*, and the UI widget, such as Toast. I personally find the logging debugging method the fastest and the most effective way to find and resolve a problem. The *LogCat* utility is a logging utility that reports about any problems during code execution. Whenever an exception is thrown or the application is crashed, a stack trace report will be displayed in the message window. A report lists all code lines that were affected by the crash.

LogCat can be configured to display the report only from one of the following categories: Verbose, Debug, Info, Warn, Error. In this way the *LogCat* message will carry a particular meaning based on the category of the message. Using a particular messaging category helps to get rid of the information that has no important meaning for the developer. Another way to optimize the performance of *LogCat* is to apply a message filter that will display debugging messages according to the filtering criteria.

LogCat debugging was widely used during the project testing phase. Especially it was very useful when detecting and fixing the errors related to the category of *NullPointerException*, which usually takes place when a variable is used without instantiation. The second most commonly used debugging tool was a *Toast*. The *Toast* is an Android GUI dialog that can be implemented by using only one line of code. It creates a sense of faster responsiveness as it is shown to the user during the application runtime without the need to read the *LogCat* logs.

6.2   Testing the Implemented Functionalities

After the implementation of the specified functionalities was completed, it was decided to troubleshoot the application in the hope of finding any problems. The testing process of the implemented functionalities was divided into two stages. The first stage involved the analysis of the communication operability between the client and the server. In addition to that, it was tested if the implemented functionalities behaved as intended. The second stage implied the usage of real Android devices to evaluate the look and feel of

the implemented functionalities, check the UI elements positioning based on a different screen size. Furthermore, the task included auditing the responsiveness of the application under different conditions. Finally, some time was dedicated to test and troubleshoot the operability of the networking module, in particular WI-FI utilization efficiency.

For testing the operability of the communication interface between the client and the server, Java utility software was created and used. The software allows to simulate a client connection and provides UI with the functionality to send requests to the server. The tester client software is written solely on the Java language. Its main goal is to mimic the connection between the client and the server in order to test request commands and monitor the responses. Furthermore, the tester client serves the purpose of testing the authentication commands and populating the game world with dummy players.
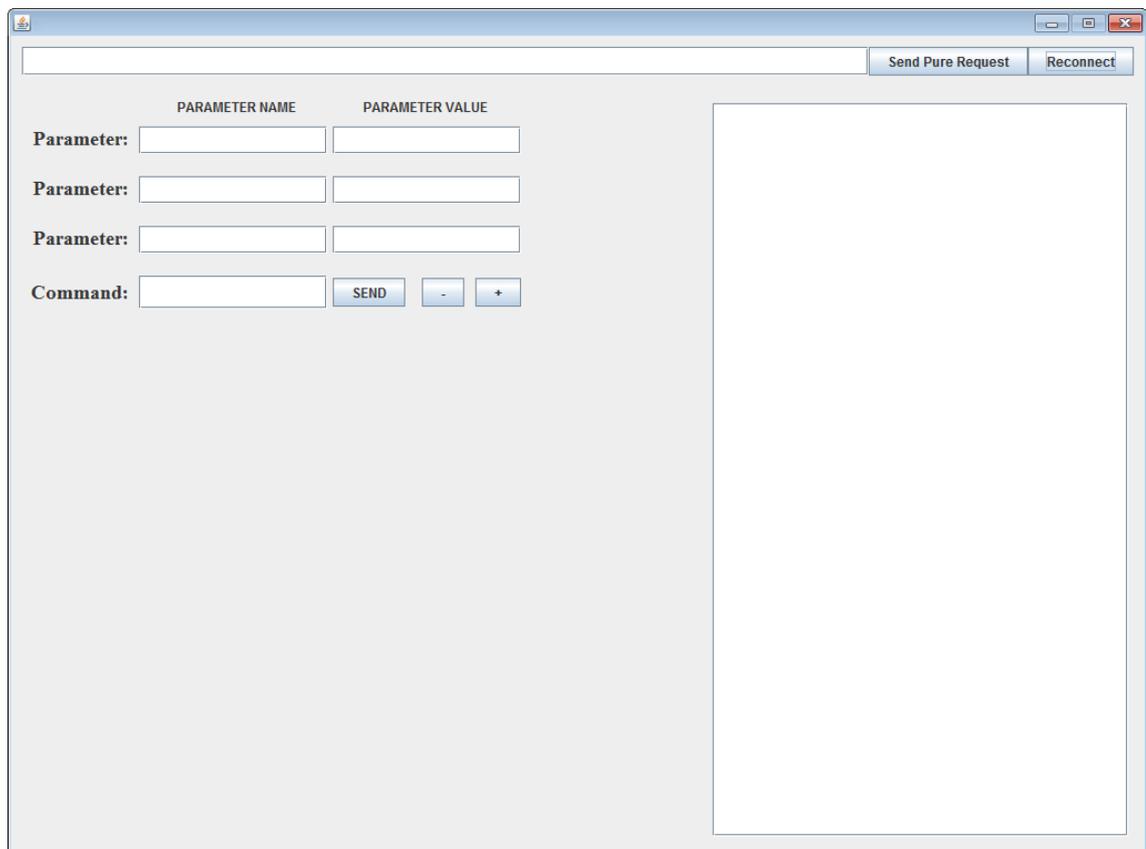

Figure 24. The main UI of the tester client.

The main UI of the tester client is shown in Figure 24. The game uses an XML communication protocol between the client and server. Each request and response are generated based on the communication protocol pattern. The structure of the message includes key-value tags as parameters. Each parameter holds the name and value that

corresponds to that name. A command defines the purpose of the message. The user can fill in the required data into the fields. If the number of fields is insufficient, new fields can be dynamically generated and placed below the row of the last field.

The plus button adds a new parameter field, the minus removes the last parameter field. By clicking the *Send* button, the filled in information is examined, converted into XML command message and sent to the server. The right-hand window acts as a log screen that displays the generated request and server response. The tester client is a useful tool when it comes to testing and analysing response details as well as the visual outcome traced from the game application.

Figure 25. Client tester in action.

Figure 25 shows how the client tester works. Suppose we would like to register a new player. For that purpose, we would need to generate a request command that would contain a list of parameters with key-value pairs. The data is read from the form fields and is converted into an XML language message using the implemented class *ServerRequestFormer.* This class has a set of methods used to generate an XML message for a particular request type. In this case, only two methods were use:

- *public String registerNewPlayer(String login, String pwd, String mail)*
- *public String createServerRequest(String command, HashMap<String, String> param).*

The first method accepts user credentials as a parameter list, wraps the data into a hash map and then calls the second method supplying the command name and a reference to the parameter's key-value pair in the form of a hash map. After sending the request to the server, we can see the response immediately. According to Figure 25 the server responded with a message that belonged to the system category scope. The code of the message tells the client about the purpose of this message and how it should be treated.

During the second stage, the game was tested on a real Android device. For testing purposes, two tablets with the most popular screen sizes were chosen: a 7-inch Google Nexus tablet and a 10-inch Samsung Galaxy Tab tablet. The attention was focused on the following aspects:

- UI's look and feel on large and extra-large screens
- Operability of the implemented functionalities
- General responsiveness of the application
- Connection response according to the zone change.

The primary concern was to develop the game that would look the same on screens with different sizes. It was decided to target the most popular tablets with screens ranging from 7 to 10 inches. The goal was to create different layouts that would define the positioning values based on the given space of the device screen. Running the game on Android Google Nexus 7 and Samsung Galaxy Tab 10 would determine if the game needs any fixes concerning the UI positioning and layout design.

Another goal of this testing process was to ensure that the operability of the implemented functionalities was exactly the same as observed from the emulator perspective. It is a known fact that the application looks slightly different on a real device than on an emulator. Furthermore running the application on a real device and observing its behaviour may help to spot some malfunctions and issues. Therefore, it was important to run the game on the real devices and ensure that all features were working as intended.

Since the game is heavily dependent on networking operations, it was logical to carry out a test that would evaluate the responsiveness of the implemented functionalities. This task included measuring the delay between the user's interaction and the application response. Most of the screen views are registered to listen to particular events. When interaction takes place, this even will be registered, and all listeners subscribed to this event will be notified. Before a view can be updated, a response from the server should be received and parsed. Only after the controller obtains updated data, the view will be updated. The difference between the interaction and the view's update is a delay that should not exceed a threshold value or otherwise the application may look sluggish and unresponsive.

Finally, one of the major concerns was to examine how the game networking functionality reacted if the network status changed. One point of interest was to test the game in situations when the WI-FI signal could be temporarily lost. For example, if the user is travelling in the metro and the train enters a tunnel, the signal can be lost, and the Internet connection will also be lost. In this case, the application should be ready to handle this situation accordingly.

7   Results and Discussion

At the current stage, the minimum requirements set for client-side development were met. In its current state the game supports authentication and registration of players, character selection, inventory and character statistics, locations, duel request list, location players list, fight instance and active fights list.

A multifunctional and universal client-side architecture was successfully designed and developed to satisfy the basic needs of the above-mentioned game modules and to provide a user-friendly and interactive UI functionality. The architecture was based on the MVC structure aimed to promote the maintainability and extensibility of the code, in addition to providing an organized structural pattern for application development. The implemented architecture does not require the integration of some commercial game engine in order to run the game.

A point-to-point connection between the client and the server machine was established. A multiplayer functionality was achieved by successful development and deployment of the network module. The network module provides an interface for client-server communication based on the socket technology. A point-to-point messaging protocol was developed. The client can successfully interact with the server, process and parse XML messages, update the game world model, trigger and handle the listener's events and generate XML requests.

The Observer pattern was successfully integrated into the architectural structure of the application. Different types of game listeners empowered by the generic architectural style of the Observer pattern were successfully embedded into the game environment. As a result, various game modules received a functionality to detect and react to the game changes.

As a sole developer of the client side architecture, I had to concentrate on the following tasks:

- Plan, design and develop the look and feel for game modules
- Adjust all client code to support the server message XML structure
- Synchronize all operations with the server
- Develop and implement XML layouts to support the target screen size
- Manage all game resources.

The hardest part was to design the general look and feel of UI components. Since neither of us had experience in game design, I had to use my basic Photoshop skills to play the role of a game artist. Unfortunately, it is very difficult to find a good game artist that would be interested in working for the game idea. We could not afford to hire a professional game artist.

The biggest problem was to create a game design that would follow a common, coherent style and fit nicely into the game genre. It is very difficult to complete this job if one does not have an artistic vision and decent designer skills. Thus, we had to stick to a more simple approach and realize the game design by trial and error in a blindfolded way.

To enhance the look and feel of UI widgets, the nine-patch technology was used. The nine-patch exploits the transparency of image files such as *png* files in order to achieve an advanced form of *nine-slice* or *scale9*. The nine-patch mechanism allows filling in bitmaps by stretching ordinary image files without quality loss, maintaining a uniform look. A system will scale up a nine-patch file to cover the desirable bitmap area. [20]

I have used the nine-patch mechanism to wrap a container into a frame, creating a sense of unique and decorated borders with custom shapes. The nine-patch was also used as a background material for several widgets such as buttons, text views and list views.
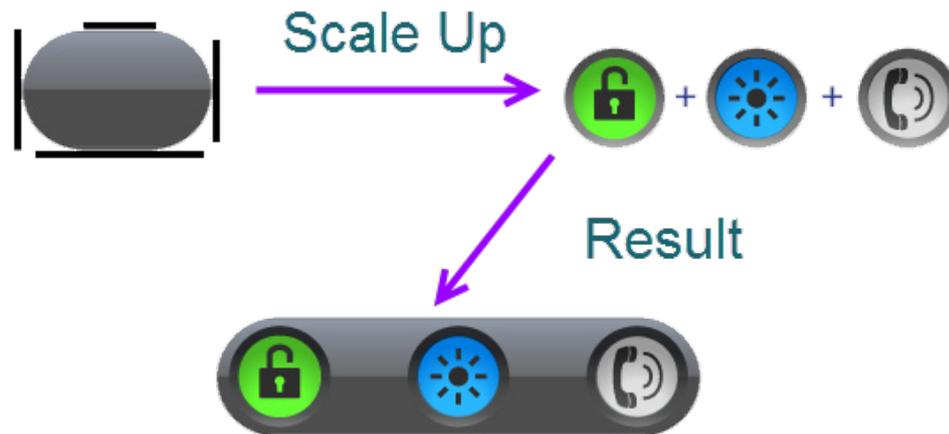
Figure 26. Nine-patch application. [20]

Figure 26 shows how the nine-patch approach could be used to create a decorated GUI element. In this case, a gradient rectangle with rounded edges was used to be stretched horizontally covering and including three custom-button icons. The outcome creates a sense of complex and advanced GUI widget drawn by a professional artist. With the nine-patch, it is easy to create a form with three-button icons engraved in the gradient ornament.

Developing MMORPG in Android requires smart and efficient use of system resources. All resources should be utilized in such a way that it would be relatively easy to manage, arrange, find and maintain them in the future. As the application grows, it will become harder to maintain it. Hence, all resources should be structured keeping in mind the issue of maintainability. Fortunately, Android helps to overcome this problem by defining the structural pattern of application resources.

A pattern implies a hierarchical resource path structure aimed for universal use of the XML language to simplify the utilization of Android resources. Following this pattern, it is becoming evident where to place a certain resource file. For example, for the different type of resources, the system implies allocation of a certain folder. During the application runtime the framework grants access to this folder, by invoking a certain method and getting a resource reference. Thus, image files can be stored in the *Drawable* folder, sound and video files can be stored in the *Raw* folder and so on.

Android resources are represented in the form of XML files. Learning the specification and application of Android resources is of vital importance. Many tasks can be achieved

and completed by correct and appropriate use of Android resources. During the project I learnt how to create Array type resources, how selectors work, how animation works, how XML layout is inflated into the View object and how image files are turned into bitmaps, how to create namespaces for custom views and pass attributes to them.

Unfortunately, the project has some bottlenecks and limitations. Not all desired features were implemented. In addition to the minimal requirements, it was planned to make the game more interactive by implementing extra game functionalities such as an in-game communication channel, full-scale game items support, in-game shop development, team fights, questing and rewards. The reasons for omitting these functionalities lie in the insufficient human resources and lack of time.

Some technical difficulties originated during the development phase involved the synchronization process between the client's and the server's technicalities. The lack of details in the documentation caused some malfunctions that took some time for investigating and resolving. Furthermore, the complexity and scope of the project appeared to be overwhelming and started to require extra labor resources on later development stages. Thus, it became too difficult to manage the resources with only two people involved into the development process.

Some difficulties were detected while dealing with a network module. In particular, the level of concurrency and its management during the connection establishment caused some problems. It was not very clear at which point to initiate a socket opening routine and how to synchronize these operations with the use of input and output streams. A sequential execution would solve this problem if only one thread was used. However, a multithreading approach made this straightforward operation much more complicated.

The synchronization of client and server operations caused some major problems during the development process. When new functionalities were implemented on the server side, client architecture was the subject of some significant changes. Due to lack of human resources, a massive and time-consuming modification of the client-side architecture was inevitable each time my partner deployed a new functionality on the server side. This process of synchronization of the client and server functionalities was tedious and cumbersome and could be improved in the future.

Nevertheless, the project can be considered successful as the minimum requirements were met. A client-side and server-side software architectures were developed and deployed to satisfy the client-server model. The game provides the basic functionalities that are common for a multiplayer game. A great scale of work was done and a new set of professional skills was acquired. Furthermore a structural analysis and research were conducted in various software engineering areas such as network programming, concurrent programming, user interface design, game development and Android platform programming.

8 Conclusion

The goal of the project was to develop a client side architecture for the MMORPG game. Overall, the goal was achieved and the minimum requirements set for the client side development were fulfilled. As a result, a multifunctional and universal client side architecture, based on the MVC structure and powered by the Observer pattern was created. The architecture provides multiplayer support by establishing a client-server communication interface based on the socket technology.

The game satisfies the criteria of MMORPG. It supports a large number of simultaneous connections, follows a self-made game messaging protocol, and offers a selection of different characters that play a specific role in the game. Several functionalities that are common for MMORPG were implemented. These functionalities include different locations, an inventory system, character statistics and a fight instance.

Participating in a project of such scale opens vast opportunities for the improvement of professional and communication skills and gives rise for a deep analysis and research in different computer science areas such as networking, Android framework, Java platform, concurrent programming, game design and development.

During the development process I learnt, for example, the structure of the Android framework, its tools and libraries available for game development, MVC structure, Observer pattern, socket programming, Android widgets and resources, custom views, multithreading, client-server model and XML parsing technics. I became familiar with the ideas behind the multiplayer game development and I learnt new concepts about the multiplayer game server architecture. In particular, I discovered the meaning of the NIO technology, learnt how to design and develop a client-server messaging protocol and how the game world instance could operate under the influence of the client and server.

The project can be further extended. Existing functionalities can be improved, the architecture can be modified and upgraded to meet the needs of new functionalities and security solutions can be applied to improve the confidentiality and data integrity. Furthermore new features aimed to enhance the interactivity of the game can be implemented. Finally, the recruitment of a professional game artist may solve the major problem – the lack of quality game arts, which would make the game much more user-friendly, interactive, fascinating and successful.

References

1   Harbour J. Android Game Programming in 24 Hours. Stoughton, Massachu-
     setts, USA: Pearson Education Inc.; 2013.

2   Java Code Geeks. Building Games Using the MVC Pattern [online].
     URL: http://www.javacodegeeks.com/2012/02/building-games-using-mvc-pat-
     tern.html.
     Accessed: 8 May 2014.

3   MVC Architecture in Android [online].
     URL: http://gkonandroid.blogspot.fi/2013/11/mvc-architecture-in-android.html.
     Accessed: 8 May 2014.

4   Phillips B, Hardy B. Android Programming: The Big Nerd Ranch Guide.
     1989 College Avenue, Atlanta, USA: Big Nerd Ranch Inc.; 2013.

5   Goetz B. Java Concurrency in Practice. Stoughton, Massachusetts, USA: Pear-
     son Education Inc.; 2010.

6   Zozety Games. Creating My Android Game Loop [online].
     URL: http://www.zozety.com/Java.aspx.
     Accessed: 8 May 2014.

7   Eckel B. Thinking in Java, Fourth Edition. Upper Saddle River, NJ, USA: Pear-
     son Education Inc.; 2006

8   Android Developers. Keeping Your App Responsive [online].
     URL: http://developer.android.com/training/articles/perf-anr.html.
     Accessed: 8 May 2014.

9   Kurose J, Ross K. Computer Networking: A Top-Down Approach, Sixth Edition.
     Harlow, Essex, England: Pearson Education Limited; 2013.

10   Buyya R, Selvi T, Chu X.Object Oriented Programming with Java: Essentials
     and Applications. Patel Negar, New Delhi, India: Tata McGraw Hill; 2009.

11 Meier R. Professional Android 4 Application Development. Indianapolis, Indiana, USA: Wiley; 2012.

12 Edureka. Android Architecture [online].
URL: http://www.edureka.in/blog/beginners-guide-android-architecture/.
Accessed: 8 May 2014.

13 Microsoft. What is a driver [online]?
URL: http://windows.microsoft.com/en-us/windows/what-is-driver#1TC=windows-7.
Accessed: 8 May 2014.

14 Lee WM. Beginning Android 4 Application Development. Indianapolis, Indiana, USA: Wiley; 2012.

15 The Game Design Initiative at Cornell University. Game Architecture [online].
URL: http://www.cs.cornell.edu/Courses/cs3152/2014sp/lectures/10-GameArchitecture.pdf.
Accessed: 8 May 2014.

16 Enterprise Service Bus. Message Dispatcher [online].
URL: https://docs.wso2.org/display/IntegrationPatterns/Message+Dispatcher#MessageDispatcher-IntroductiontoMessageDispatcher.
Accessed: 8 May 2014.

17 Object Oriented Design. Observer Pattern [online].
URL: http://www.oodesign.com/observer-pattern.html.
Accessed: 8 May 2014.

18 Android Developers. Debugging [online].
URL: http://developer.android.com/tools/debugging/index.html.
Accessed: 8 May 2014.

19 Komatineni S, MacLean D. Expert Android. California, USA: Apress Media; 2013.

20 Radley Marx Blog. A simple guide to 9-patch for Android UI [online].
    URL: http://radleymarx.com/blog/simple-guide-to-9-patch/.
    Accessed: 8 May 2014.

21 Stallings W. Cryptography and Network Security.5th edition. Upper Saddle
    River, NY: Prentice Hall; 2011.