Contents

List of Corrections

1 Introduction

Creating new games requires a lot of time and funds. Creating games similar in functionality to games that already exist require the same amount of time and funds as creating the original. Having to create n games that have similar functionality, requires n times as much time and funds if we keep following that logic - but creating some underlying structure when creating the first game cuts down the time and funds needed to create similar games.

A big problem for developers of a new game is the issue of having to create good usable framework for their specific needs. A framework is a basic set of functionality and structural guidelines for the game to be built upon. At the start of a project, game developers set out to create a framework that can handle all of the functions that the game need, without really having specified precisely how those functions are implemented. This means that if functions change or an entirely new functionality need comes up, the framework have to be adjusted to the specific need. An example of this could be how the game handles several users, maybe the game have different needs depending on some variating variable. The amount of workload a developer has to put into creating the framework to match exactly his demand is greatly increased. The different requirements and often change in the framework often makes code incomprehensible.

It requires a lot more work and overview from the small game developers to keep the framework completely up to date. Bigger firms almost always require multiple sections to use the same framework[1] - and therefor consistency (in form of little to no change) is a must. More often than not these firms create their own frameworks to design them exactly the way they need it, but smaller firms might have limited funds or time. This means that smaller developers often either skip on creating features they normally would or create sloppy and inefficient code as a side effect.

Creating a joint framework that could be used by different developers could save smaller and new game developers a lot of time, and create a standard that makes coding on top of it easy.

[1]FiXme Warning: kilde?

The subject is widely pursued in the form of game engines. A large amount of games being developed are developed on an already existing game engines. An example of a game engine is the ‘Adventure Game Studio’ and this is one of the biggest frameworks for creating games in third person pre-rendering. The developer ‘Chris Jones’ have created different functionality he deemed important for creating these games and published it in the form of an simple IDE for inexperienced game developers [3]. Another example of game engines is full blown engines with multiple purposes for a specific purpose. One of the most used engines for first person shooters is the ‘Unreal Engine’ - which is used for almost every major shooter game.

1.1 Chapter Description

The following is a brief description of each chapter in the report

Design and Implementation The design part of this section will describe the general idea behind the code - what small choices have been made along the way and why those were chosen. The implementation section will describe how the code for the project was then implemented.

Test

Future Work This section will describe the future work of the project. Ideas and unfinished functions will be described in a perfunctory fashion - giving the possibility for people to advance the project in the direction the original authors intended.

Conclusion A brief summary of the entirety of the project and an evaluation of whether the project upholds the goals of the project.

2 Problem Analysis

Development of an online multiplayer game serving many users can be split into two parts: Creating the game, and creating the underlying architecture. The focus of this project is on creating a flexible and reliable server architecture [1] allowing easy development of multiplayer games for Android smartphones. The aim is to make it possible for developers to focus on the game development without having to worry about managing the underlying architecture.

A problem with developing different games from scratch as opposed to having a standardized framework is the compatibility. Having different types of devices with different versions of various operating systems means that some devices might not be compatible with the desired program. What this means is that programs act differently on different devices because of the inconsistent code - and for this to be a non issue you would have to create code to match each type of device and operating system which is an extreme task. Having a framework that is already compatible with the different devices allows the coder to focus solely on the game.

2.1 Problem Statement

When creating games the amount of time and funds spent are very limited. To ensure a high amount of quality code, the developer has to spent a lot of time writing trivial code and it will thrive the cost up. This naturally raises the question:

How can we develop a independent framework for creating simple multiplayer games - limiting the amount of time developers have to spent on writing code

The idea is simple, we want to creating framework for creating simple coordinate-based multiplayer shooter games. In order for the problem statement to be fulfilled,

[1] FiXme Fatal: client-server? smth else?

we raised some additional goals to the question asked above:

- The framework has to be as flexible as possible.
- The framework has to account for multiple instances of games with multiple users.
- The framework has to be easy to use.

2.2 Game scenario

This section describes the game we have decided to focus on. This game will be implemented as a prototype for proof of concept. It implements different uses of the server functionality that we want to display. It was also created for the purpose of setting a limitation for the server functionality. Otherwise we could have kept creating functionality for multiple different purposes. The game we have decided to create a scenario for, is a game based on the real-life coordinates of the participants. The game should be playable among friends, and be interactive. The game is a prototype that could be implemented on top of our server and item balance, e.g. a certain weapon may be stronger than intended, is in no way to be considered final in the game.

The biggest advantage of creating a specific game scenario is a better insight into the general functionality needed on the server side. It furthermore provides a goal which helps with motivation and work culture, in the sense that we could more easily divide the work needed in the project. One of the drawbacks of using a scenario in this way is that, if one is not not careful, it could cause us to accidentally focus strictly on the scenario, and might exclude functionality that could be useful on the server side. Just because the functionality does not fit a certain scenario does not mean it is not useful for a general model. Furthermore, we might not get the flexibility we look for in a server as we don't have different types of games to test it with.

2.2.1 Game Rules

The following is a description of the game scenario.

Game winning criteria The idea is for one of the competing teams to obtain an ultimate relic. Once a team locates it, the game should end with a victory for the team with the relic. At the start of the game it is impossible to know where the relic is located, and you gradually locate it by finding clues around the map. Another way to win is to be the team with the largest amount of points if a time limit is set and reached.

Objectives There is a list of different objectives each with a unique functionality and meaning to the game.

- Point Objectives - Gives +1 team point, takes 30 seconds to capture and reveals your locations to nearby enemies for 120 seconds.

- Puzzle Objectives - Gives your team a clue, takes 300 seconds to capture and reveals your location for 600 seconds.
- Crates - Can contain weapons, instantly acquired and reveals your location for 120 seconds.
- Powerups - Can contain bonus for your stats, instantly acquired and reveals your location for 120 seconds.
- Shrines - Will give different bonuses depending on the shrine type, the shrine is active for 4 hours after activation, takes 300 seconds to capture and reveals your location for 600 seconds.
- Ultimate relic - Digging for the ultimate relic has a one hour cooldown, takes 300 seconds to capture and reveals your location for 600 seconds.
- Watchtowers - Gives increased range when within the tower.

Items There is a list of different items in the game as well. These items are obtainable through crates dropping.

- Start-pistol - Range: 25 meter, Damage: 20 HP, reveal-time: 60 seconds and cooldown: 20 seconds.
- Shotgun - Range: 15 meter, Damage: 60 HP, reveal-time: 90 seconds and cooldown: 30 seconds.
- Sniper - Range: 40-80 meter, Damage: 70 HP, reveal-time: 90 seconds and cooldown: 120 seconds.
- Ammunition - Pistol ammunition +16 bullets, Shotgun ammunition +4 shells and Sniper Ammunition +3 bullets.
- Medkit - gives +25 health points on pick up.

Powerups This is the list of what is obtainable as powerups.

- Armor - +15 Armor (Max 50 Armor)
- Vision - +2 meter (Max 50 Meter)
- Scan - +20 meter (Max 60 Meter)
- Range - +2% range (Max 6% Range)
- Shield - Blocks the next shot

Shrines The following is a list of the different shrines and functionality. Shrines expire after a given amount of hours.

- Power Plant - Provides 2 points every half an hour.
- Locater - Pinpoints the location of an object in range of scan every half an hour.
- Shield Generator - Gives the entire team a shield every hour.
- Factory - Provides each user on the team with 3 pistol bullets, 2 shotgun shells and 1 sniper bullet every half an hour.

2.2.2 Different game scenarios

Since we created a server with functionality completely independent of the game implemented on top [2], a lot of different games can be implemented. The game might not even focus on coordinates of the players but simply be an ordinary multiplayer game. One could imagine a multiplayer Pac-Man game, where some people play ghost and one person play Pac-Man - this could easily be built on our server. The server would receive actions performed by the user and adjust the game accordingly. Basically the server is simply an implementation of the multiplayer aspect of a game and hosting a game.

Another option entirely is that there is no reason why there should be teams. Co-op and all versus all games are a possibility as well. It is up to the creator of the game to decide what should be built on top of the server implementation.

2.3 Transmission Control Protocol

The Transmission Control Protocol (TCP) is a part of the Internet protocol suite. Together with the Internet Protocol (IP), TCP is so widely used that the entire Internet protocol suite, containing many protocols, is often just called TCP/IP.

In the following, the parts of TCP that are particularly important for this project are described. This chapter is based on [4]. [3]

2.3.1 Data Transfer

Because we are working with transfer of data to a device with an uncertain connection, it is relevant to consider how TCP handles transferring data. An important aspect of this is to ensure reliable transmission, i.e. to make sure all the data is transferred correctly. To ensure this, each byte of data gets a sequence number, allowing the destination host to reconstruct the data in case of for example packet loss. Additionally, when a packet is received, the receiver sends back an acknowledgment, and if such an acknowledgment is not received the packet will be sent again. To ensure correctness of the packet content, each packet has a checksum included which the packet's content can be compared to upon arrival at its destination.

[2] FiXme Note: reference til separate frameworks fra server shit

[3] FiXme Fatal: Find better source.

This possibility of error checking comes in handy in our project. When a map is transmitted to a client, it ensures that the map data is not corrupted. This is very desirable since corrupted map data could make the map unusable in the best case, and show wrong map data in the worst case.

2.3.2 Flow Control

It is possible for the server to sent more data than the receiver can process, and to accommodate this a flow control protocol is used. When data is sent, the receiver answers back with a *window size*, telling the sender how much more data it is able to process. If the window size reaches 0, a persist timer will be set to account for the possibility that the updated window size was simply lost. When the timer runs out the server will send a small packet, probing the receiver for an updated window size.

2.3.3 TCP/IP versus UDP

We consider two aspects of TCP/IP versus UDP. The first aspect is which protocol is cheapest in space usage when transferring from one unit to another. The second aspect is which protocol offers the best suited services for our application.

These questions are related to the problem we try to solve. We want to try to download data on an endangered connection or download within a limited time-span before the device goes offline. Therefore we investigate TCP/IP versus UDP further.

When transferring data from one unit to another, the size of data to send varies depending on the program. However, the header has a minimum or fixed size it always uses, without considering the size of the data the user wants to send. In TCP/IP, the size of the header is 21 octets[1]. In UDP, the size of the header is 8 octets[2]. In comparison, UDP uses 38% of what TCP/IP uses for each exchange.

The services provided by TCP/IP are reliability through error checking and delivery validation whereas UDP emphasizes low-overhead operation and reduced latency[4]. Both protocols provide valid services to be used when solving our problem. TCP/IP will ensure we have correct, consistent and working data. UDP could be used for transferring GPS-coordinates because they will be transferred extensively when the application is running. Losing an UDP transaction will not break the application. If a few data is lost or inconsistent, new data will be sent in the very near future.

3 Design

This chapter outlines the different design choices made for the project. In addition, possible alternatives are presented and discussed where applicable.

3.1 Synchronous vs. Asynchronous I/O

For the client/server socket communication, a choice between synchronous and asynchronous I/O has to be taken. Synchronous I/O can have a better performance than asynchronous, but can cause problems when using a threaded architecture that spawns a new thread for each client. This is particularly true when the server should be scalable in regards to its number of connected clients. There might be 5 and there might be 5000 or even more. Tests show that threads are very efficient when it comes to memory and context switching, but only when the threads are kept alive for the entire execution of an application. This is not the case for our application, which will likely have many connections of varying durations during its up-time. [1]

Asynchronous I/O is chosen for this project. It scales well when there are many clients, and the system should scale well with a potentially large number of clients. A notable advantage of asynchronous is that it limits the number of concurrent threads. The server asynchronously accepts a connection request from a client, and then starts an asynchronous worker thread to handle communication with the client. Meanwhile it continues to listen for new client connections.

[1] FiXme Fatal: cite

3.2 Architectural Pattern

3.3 Design Patterns

When considering how to design a software solution it can be a good idea to consider using some design patterns. Without those, code easily becomes messy later on in the project. Patterns can help by providing conventions and best practices to use as guidelines. With defined patterns it can be ensured that everyone working on the project has similar understandings of the code. It is worth to note that like everything, patterns can be misused; It is important to choose those that will fit the project, rather than try to force harmful patterns on to it.

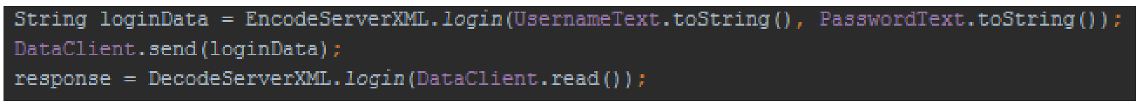
Below is a description of patterns found usable in this project. Patterns are split into two, one with patterns applicable in the client application, another with patterns applicable on the server program.

3.3.1 Patterns in the application

Lazy Initialization pattern is designed to save/delay use of resources by postponing steps for as long as possible (Until right before it is needed).The Android application in this project does not perform many heavy operations, but it should always be remembered that the application is running on a phone, and that it has a battery. As such, most initializations of objects has been pushed back to the point where they are required along with operations on data.

Adapter pattern is a simple idea, but can be used in many contexts. It stems from the problem of wanting two incompatible interfaces to work together. The pattern solution is to use adapters to facilitate a common ground for the two interfaces. In Android, adapters are used to construct lists, grids and more. The adapters help link lists of items together with a visualization, using a given layout to inflate and a set of items. It will then provide a view of each item, effectively providing a user-friendly way of using/viewing the data. For this project adapters have been used to create and display clickable lists.

Facade pattern describes the idea of hiding messy procedures by using a facade. The result is that whoever wants to use the procedures have access to simple, easy to use facades which acts as access points to the procedural steps. In this project it has been used mostly for abstracting away the server communication. As this is done by creating XML strings with relevant data, this would be tedious to do in every activity that needed server communication. The communication itself was also abstracted. Instead, this functionality has been written in separate classes and is accessed through simple method calls. An example can be seen in 3.1. This displays what happens when a user tries to login. The username and password is converted to XML and sent to the server, which in turn provide a responde. All the details of how this happens are however hidden.



```
String loginData = EncodeServerXML.login(UsernameText.toString(), PasswordText.toString());
DataClient.send(loginData);
response = DecodeServerXML.login(DataClient.read());
```

Figure 3.1: Abstraction of server communication

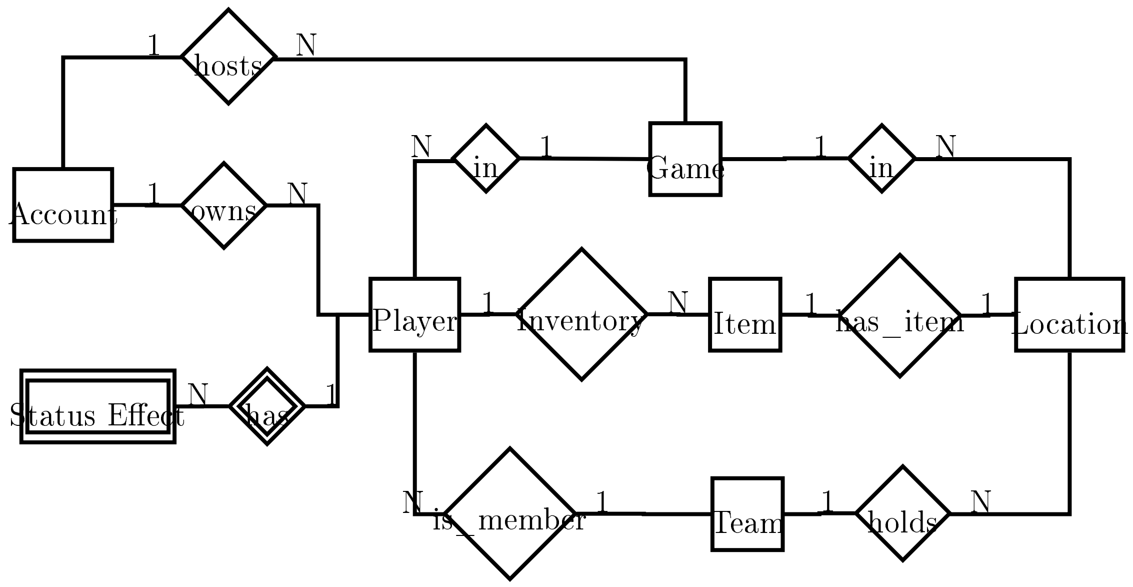


Figure 3.2: ER Diagram.

3.3.2 Patterns on server

3.4 Server Architecture

3.4.1 Database

In this project a database was needed to keep track of all information related to users and games they are playing. The database is designed with flexibility in mind which means that the logic behind a game is responsible for interpreting the data in the database. Figure 3.2[2] shows the entity relation diagram for the database without attributes. The database is structured as follows:

Account The Account entity represent the users in the system. An account can host games represented by the host relation and take part in games represented as owning a player in a particular game.

Player This entity represents a user playing in a game. A player can hold items, have status effects and be a member of a team represented by Inventory, has(Status Effect) and Team(is_member) respectively. The player entity also contains score and location etc. for a particular player in a game.

Game This entity represent games either *starting up*, *in progress* or *ended.*

Status Effect Effects on a player is represented by this entity. An effect could be *disabled until time* on a particular player. Status Effects have an effect type which the game logic is responsible for defining.

[2]FiXme Warning: export new ER diagram

Team Players can be members of teams in a game, though a player is not required to be on a team to allow free for all game modes.

Item Items represent any object in the inventory of players or on a location in the game world. Items can be anything from objects players can "pick up" to a capture-able area in the game world. The attributes and behavior of items are defined by game logic.

Location A location is an item in the game world that can belong to a team. To own a location, a team must take it first.

3.4.2 Game thread-pool

The game thread-pool is a collection of active game thread-instances, uniquely identified by a game-Id.

The game thread-pool is used as a handle to the game threads, and allows dynamic call to the desired game through method-parametrization.

3.4.3 Game thread

Starting a game A game thread will be initialized and started whenever a user requests to host a game. User-specified settings for hosting the game are sent as part of the request to host the game. These are then initialized and the game will be created in the database which assigns it a game-id be assigned a game-id for future identification.

When the server initializes each setting, it calls a method within the game thread-class. These methods calls the database-controller to change the state of the game in the database. These settings are:

- Game-privacy (Public or private game)
- Number of teams
- Game start-time
- Game end-time (If any)
- Game-boundary NorthWest GPS-coordinate
- Game-boundary SouthEast GPS-coordinate

Updating a game Updates to a game can be split into two groups. One group is specific changes to a game, like inviting a new player or firing a gun. Another group is updating a players position when moving around in the game.

Updating a players position is trivial. The game thread receives a game-id, player-id and a new position. It calls the database-controller to store the new position in the database.

When performing an action like firing a gun, the server will have to fetch the gunman's position and the victim's position. It will then calculate if the range of the fired weapon allows the victim to be hit. If the shot is successful it will return that to the player, if the shot is unsuccessful it will return that the victim is out of range.

All updates to change the state of a game will be in the game-thread class. This class will need to contain methods for all the in-game functionality.

Closing a game When a game ends, the call will come from the timer thread. This will ask the game thread to clean up what is has stored in the database, and return a status message. The timer thread will then continue to close the game thread.

3.5 Interfaces

We have decided that all information between the server and the client are shared in the form of XML. We have chosen to do so because of the diversity XML provides, and its interoperability with a wide variety of software. This fits well within the theme of creating a customizable framework for creating games. Furthermore XML is a widely used transport format for data, and thus many people know how to use it.

3.5.1 Client and Server correspondence

The idea behind the XML output is to create tags for each different kind of data. We have different kinds of data to sent depending on which activity we take on the client side. An example of this could be the login activity on the client side, this requires sending some username and a password for confirmation on the server side.

We create individual methods for each different scenario rather than creating a generic XML converter because we want to enforce the right data structures and types.

The information is structured around what the server needs for authentication - so what the client sends is of course built around what the server requires.

```
<Login>
    <Username> USERNAME </Username>
    <Password> PASSWORD </Password>
</Login>
```

For the same reasons mentioned above and for the sake of consistency the client expects XML on roughly the same form. For this example we require an ID for the specific user to load his specific settings/information. We also require a boolean in the case that it wasn't a valid login, this is then handled on the client side. The following code is what a client expects to receive.

```
<Login>
    <Id> IDENTIFICATION </Id>
    <Valid> BOOL </Valid>
</Login>
```

XML provides a convenient way to deserialize the input - making it an appropriate solution for small constant interactions like the ones we need. There are many different alternatives to XML, but we feel that it is a simple interaction without extraordinarily complicated needs - which makes the simplicity of XML a great trait for us.

3.6 Separation of Framework and Game

The framework and the game is split into two completely separate components. This enables creation of a new game on the same framework.

The reason for this choice, is that the framework can be reused for different games. Because of this future developers who would like to make a game can reuse the framework and do not have to touch any framework code, only the game code itself.
Developing the framework and game as two completely independent components has a number of good side effects. First of all, it makes it easier to avoid writing code which is intertwined (between the framework and the game).

If this was not avoided, it would be very hard to create an independent framework, since the game which has been developed on top of the framework would be tied into the framework and vice versa. Therefore, a clear code-wise distinction between framework and server produces cleaner, more independent overall program structure. On the other hand, if the framework and game are not separate components, it could be easier to develop a single game, since the strict separation between the two can cause a development overhead. The development overhead can occur for example because code has to be planned more carefully when written for the separated components.

Because the framework and game are separated components, the framework has some limitations. The framework essentially enables development of many kinds of networked multiplayer games. Therefore, the database's tables for example have to be quite “fixed”, since the developers of the games for the framework should not need to change the database and framework code. A database change will warrant some code in the framework to be rewritten - as the controller between the database and the server is invalid. If the developers have to start rewriting code in the framework to develop their game, the point of the framework is lost. Efficiency wise having a framework with the flexibility that our implementation has, you of course lose a little. Being able to fit your game perfectly to your server would create more efficient code for that single scenario, but never be as scalable as we would want. That being said, given the framework is primarily a setup for the multiplayer part of a game, which is needed under all circumstances, it might not be inefficient to create a framework.

4 Implementation

5 Test

This is where we describe the different tests we have chosen to implement and what the results of the given tests where. The test ideally test the program up against the goals set by the problem statement.

6 Conclusion

Here we will describe the result of the entire project, whether it upholds the study regulation. We will also analyse the test results and argue whether they answer the problem statement. We will evaluate the entire project and its functionality as a full program.

Bibliography

[1] Information Sciences Institute, U. o. S. C. (2014, October 23). *DARPA Internet Program Protocal Specification.* http://tools.ietf.org/pdf/rfc791.pdf.

[2] J. Postel, I. (2014, October 23). *User Datagram Protocol.* http://tools.ietf.org/rfc/rfc768.txt.

[3] Studios, A. G. (2014, December 2). *Adventure Game Studios.* URL. @MISCwiki-tcp, author = Adventure Game Studios, title = *Adventure Game Studios*, month = December 2, year = 2014, note = http://www.adventuregamestudio.co.uk/, owner = Johan .

[4] Wikipedia (2014, October 12). *Transmission Control Protocol.* http://en.wikipedia.org/wiki/Transmission_Control_Protocol.