Networked Systems and Services, Fall 2019: Multiplayer game

Daniel Michael Walder
daniel.walder@helsinki.fi

1 INTRODUCTION

This document is for the NSS, fall 2019 course at the University of Helsinki. It is a design model for a multiplayer game, which discusses ideas about the game itself, reliability and speed of it.

The first chapter is about the game itself. How is the game is supposed to work, how to win it, lose it or just simply play it. The second chapter is about the design of the game. Various design decisions are presented there and it also discusses a few problems with a focus on reliability and speed. Especially why I have taken those design decisions.
At the end follows a summary of the most important game design decisions.

2 THE GAME

A name for the game doesn't exist yet, but it is supposed to become some sort of Clash of Clans clone. Clash of Clans is a massively multiplayer online real-time strategy (MMORTS) game. Which means it is not round-based but actions will take real-time to get activated.

Such games are mostly browser-based and players interact with one another in a virtual world. This kind of world is called persistent world, because it starts from zero and develops over time, even without interacting players.

The main idea of the game is that each player starts with a fortress and then tries to get in command of all the other fortresses on the map. The last player left wins the game. If a player has no fortresses left, the player looses the game.

Players can join at any time. The later they join, the more far away they will be from the players which started. Which is a simple game balance decision, because otherwise the players which are in the game from the beginning, would defeat the new players immediately.

Fortresses can get developed gradually. The more it is developed, the easier it is to defend. To develop forts, the player has to mine resources. With which the player can also produce soldiers, to attack other forts.

Taking over a fort can be achieved through producing soldiers and then let them attack a fort. The way to the fort takes some time, so the more far away the desired fort is, the longer it takes. If the number of attackers outmatches the defenders, the fort switches the owner. The numbers of defenders will go down to zero and the attackers will also have to deal with loses. Ressources will also go down to zero.

It is a real-time strategy game, so the players can interact constantly with the game, but set actions will take time. For example, if a player wants to develop his fortress, it will take a few minutes till it is developed. How long everything will take, can be a very tricky thing. No specific numbers will be mentioned in this document, because it depends finally on the game balance, how long everything should take.

In modern MMORTS, players have a wide range of playing options. For the sake of simplicity, this game will only provide the basics of such games. Players can develop forts, take over forts, produce units and mine resources. More complex things like alliance systems, spy possibilities or trading resources will not be part of it. At least not in the first version of the game.

As mentioned earlier, the virtual world is persistent. So the map contains many fortresses, which are in the hand of so-called rebels. Bots, which will passively develop their forts. They will not participate actively. A smart adversary is hard to implement and not common in browser-based MMORTS. Even though the game is designed as a multiplayer game, the game can be played as single player as well, because of the persistent world.

3 DESIGN

3.1 Architecture

In Figure 1 we can see a sketch of a server-client model. It shows a rough interaction between a server and client via a protocol and both store their data after it in a critical data area.
The server consists of 4 main parts. A receiver socket, which listens all the time to incoming messages. A sender socket to distribute messages to all clients. The critical data area, which will be the server's memory to store data. It will be protected with semaphores or locks, to avoid race conditions. Last part is the server logic. Which will decide what will happen, with every message, the game itself and so on.

The clients look similar. The logic will take care of the critical data area and all outgoing and incoming messages. Plus it will also take care of the interaction with the player. So it has to display the map, keep the map up to date and await commands from the player.

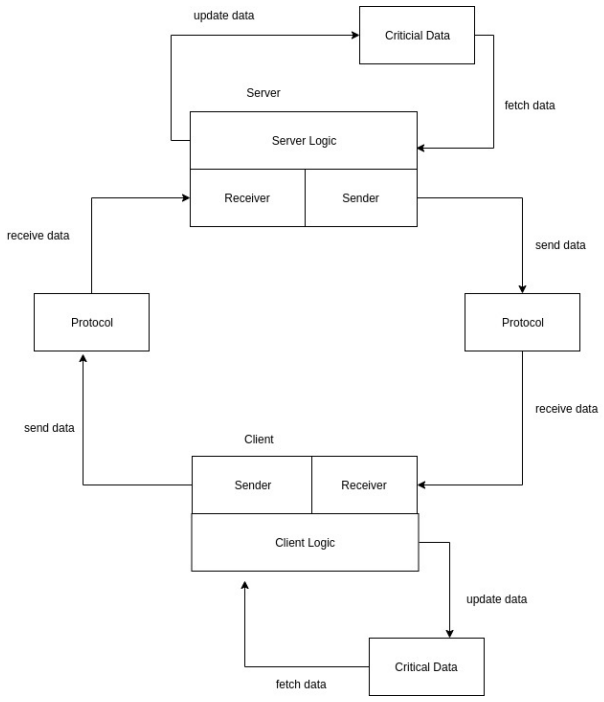

Figure 1: Server-Client Model

Message Name	Specification
ALIVE	Client sends Server a message to prove that he is still part of the game.
ACK	Server acknowledges messages from clients.
SPECIALSETTINGS	Clients informs server that he wants to set special settings.
UPDATEALL	Server sends all public information to the clients-
NEWPLAYER	Client informs Server that he joined the game.
INCREASEMINING	Client tells Server that he wants to increase mining.
INCREASEDEFENCE	Client tells Server that he wants to increase defence.
INCREASEATTACK	Client tells Server that he wants to produce soldiers.
ATTACKFORT	Client tells Server which fortress he wants to attack.
AFTERATTACK	Server tells both clients the outcome of the attack.

Table 1: Message Types

3.2 Communication

Server and client communicate via a self-made protocol over UDP. The protocol will contain around ten different kinds of messages. In Table 1. we can see all ten types of messages. This self-made protocol will consist of two parts. A header and the actual data. The header will hold information about message type, sequence number, player id and FEC method. Total length of the data part is going to be maximum of 512 bytes. Most of the messages won't need 512bytes, but if a player wants to use extra settings, then it will be needed.

As mentioned before, as transport protocol this game will use UDP. While TCP is known for reliable connections and convenient use, it also is known to slow down the speed. There is a lot of overhead which is used to ensure that every message gets delivered correctly. UDP is a lot faster because it abandons all the error correction overhead. It also gives more freedom to execute packets out-of-order.

When using UDP, packet loss is an issue. The system will use FEC to deal with that problem. As FEC method a simple triple redundancy method will be used to ensure that packets get delivered completely. With this method and a loss rate of e.g. 10% fluent and proper gameplay should be still possible. Despite that, ACK messages will be used to ensure that ALIVE messages will get delivered definitely. If messages about e.g. resources get lost, it is acceptable, because the server will send a similar message sooner or later again anyway.

To keep the total amount of messages low, various considerations have been taken. For example, when it comes to fortresses. Every single fortress contains several important game information like, the position of the fort, defence values, stored resources and which player is in charge of the fort. If there is a single change in a single fort, not every client needs to know about it immediately. The player in charge needs to know everything, but adversaries only the position and who is in charge. And these information don't change very often, or not at all. Information about the position is only necessary when a new player joins the game. Later on, when they attack, the can know the other values as well.
Check out Figure 2 too for this consideration. We can see that a server only distributes public information (position, owner) to all clients, while a single client gets all his personal information (resources, soldiers and so on).

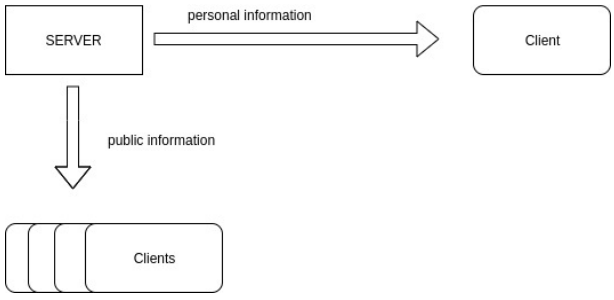

Figure 2: Message Distribution

Another consideration is to outsource as much of the exception handling to the clients. If a player, for example, wants to produce soldiers, the client architecture will check first if the player is even allowed to, before informing the server. Otherwise too many messages will be distributed, with the command of producing soldiers and then the neglection of the server. The goal is to keep as much locally as possible.

The game needs to perform many operations time-based. Also here the idea is to make the timing on the client-side. For example, a player wants to improve his fortress and that would take five minutes. The client won't send a message to the server before the time is up. This way the client will do the work and then informs the server, instead of the other way around. It reduces the number of messages a little bit, plus the server doesn't have all the work to do and is faster through this. Another advantage is that not so many different types of messages are needed.

3.3 Connection Problems

If a player disconnects, his forts will be taken over by rebels. The player's state will also not be stored. The disconnected person can rejoin but has to start over. This is simply because for the sake of simplicity. The development process, tests and most of the gaming will happen on a single machine. Distinguishing between players will get very complex. To verify if a player has not disconnected, they have to send regularly alive messages. If they stop and after a specific time has passed (timeout), they will be considered as disconnected.
If clients lose their connection to the server, they will try to reconnect several times, to avoid to get kicked out of the game.
A server crash would disconnect all clients, which would mean that everybody has to start over, as soon as the server is back online.

3.4 Number of Players

Another important game design decision is the number of players per map. The assignment says that the game is supposed to host an unspecific amount of players. In a MMORTS this is possible but in a slightly different way. A virtual MMORTS world can't host an infinite number of players. Boundaries are needed, to make the world observable/comprehensible for the actual players.
If too many players want to play, a second world will be created. If the second one is full, a third one will be created and so on.
How many players can participate in one world will be decided later. It depends on the size of the map, and how all the information (including the actual map) will finally be displayed.

3.5 Storage of Data

When it comes down to speed, a problem can become the critical data area of the server. The area is secured with locks to avoid race conditions. Reading and writing are only allowed to who has access. So it is always just one thread at a time. If the final design is wrong, this area can become a huge bottleneck for all the incoming and outgoing messages. Depending on the time a simple database would make more sense and is overall far more reliable than a mix of arrays, which are secured by locks. A database would also provide the opportunity to save the state of the game, so players wouldn't have to start over after a crash or a disconnection.

3.6 User Interface

The user interface will display a one-dimensional map of the world, the values of the player's fortresses and the operations the player can make. A graphical interface would make the appearance and handling a lot nicer than a text-based one, but this is very time-dependent. It is planned, and a GUI could be one of the special settings a player wants to set. So that the player can choose between text-based and a graphical user interface. If a GUI won't be part of the final version, there will be at least the option to download the graphics with textures.

3.7 Bots

To prove that the game is reliable and works fast, many clients will be needed to play this game. To make this happen, the rebels will be considered as clients. Instead of the server calculates their values directly, they will be implemented as simple clients, which have to connect to the game like normal human beings. Simple logic will assure that they participate in the game.

4 DESIGN SUMMARY

Overall there have been many considerations taken on how to develop this MMORTS game.

To not slow it down too much, exception handling and timing is supposed to reside on the client-side. Through that, the number of total messages will be less, and the server has less work to do.

UDP in connection with FEC will ensure speed, but also keep it reliable. A self-made protocol with several message types should ensure not just a proper communication between client and server, but also higher reliability.
Trying to reconnect, timeout time and ALIVE message will also be part of the game to make it safer.

There are also ideas, where it is time-depending if they will get implemented or not. For example, a GUI or a database. While a GUI would be for nicer gameplay, the database would make it possible to store game states.