DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF LEICESTER

INTERIM REPORT

Project MOBO

Christopher J. Cola

DECEMBER 2015

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).

Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).

I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Christopher J. Cola
Signed: Christopher J. Cola
Date: 07/12/2015

1 Table of Contents

2 Planning and timescales

2.1 Objective changes

- ❏ Familiarize with C#, the Monogame/XNA framework and the Lidgren.Network library.
- ❏ Develop a game client similar to XKobo that utilizes 2D graphics.
- ❏ Develop a server application that will enable multiplayer synchronization.
- ❏ Utilize the Lidgren.Network library to develop network protocols for multiplayer gameplay.
- ❏ Perform usability testing to gauge the quality of the user experience.

2.2 Objective progress

The objectives have been altered to better reflect the progress of the project and the goals now that some of the more nebulous aspects such as the choice of graphics and networking libraries are much more definite. The first objective is technically difficult to measure but it refers to knowing enough about the XNA framework and Lidgren.Network library to be able to implement the features they provide that fall within the scope of the project. I believe this objective has been completed as I now know enough about the XNA framework to create a basic game with graphics, input and logic as evidenced in the prototype and I have already used the Lidgren.Network library to structure and define a protocol of five messages that underpin the current multiplayer gameplay.

The second objective is in progress, but have seen some signification progress as of the prototype. The underlying structure of the game is now in place, and is it easy to add new screens and entities as I have programmed the game in an object-orientated fashion that allows relatively simple extension. The prototype provides a menu GUI interface, a settings screen where client-specific options can be configured and an in-game screen where a ship can be controlled and can shoot projectiles. The game features a dynamic camera that can track the local player's movement and a heads up display that details health, names, and a message log in the lower left hand corner for server and system messages.

The progress of the third and fourth objectives are similar to that of the second, in that the basic underlying structure of the server and client network messaging system is in place and will facilitate and trivialize the addition of new messages in the future. Currently, movement, the firing of projectiles, health changes are synchronized, and there are protocol specific messages for connection and disconnection to and from the server.

2.3 Future challenges

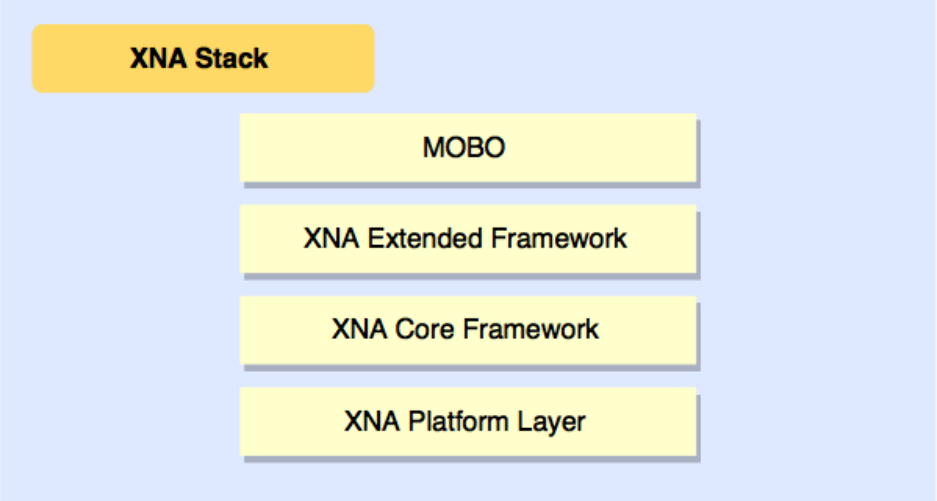
Obviously there is more to XKobo than basic movement and shooting. As the project moves into the second phase, I will be working on more gameplay focused aspects, and creating new message protocols for these new additions. Stations, or fortresses, are the main 'enemy' in XKobo that will be generated using a tree-like algorithm and based on a set size and choosing whether the logic for these enemies is calculated on server side or calculated on one client and sent to the rest is an area for experimentation. I also aim to improve the 'smoothness' of remote player movement using entity interpolation and input prediction, both of which have very helpful explanations on the Valve Developer Community wiki[1].

3 Prototype Software Architecture

3.1 Overview

The project employs a client-server architecture with MOBO, the C# application developed using the Monogame/XNA framework assuming the role of the client, and MOBOSERVER, a rather less complex C# command line program. MOBO is essentially the game itself, containing the presentation, application, and business logic required for one to play the game single player if the user so desires. For this reason it is a 'thick' client as it contains all the functionality bar multiplayer networking. MOBOSERVER acts a dedicated server that MOBO clients are able to connect to and is core to the ability for clients to communicate and synchronize with one-another.

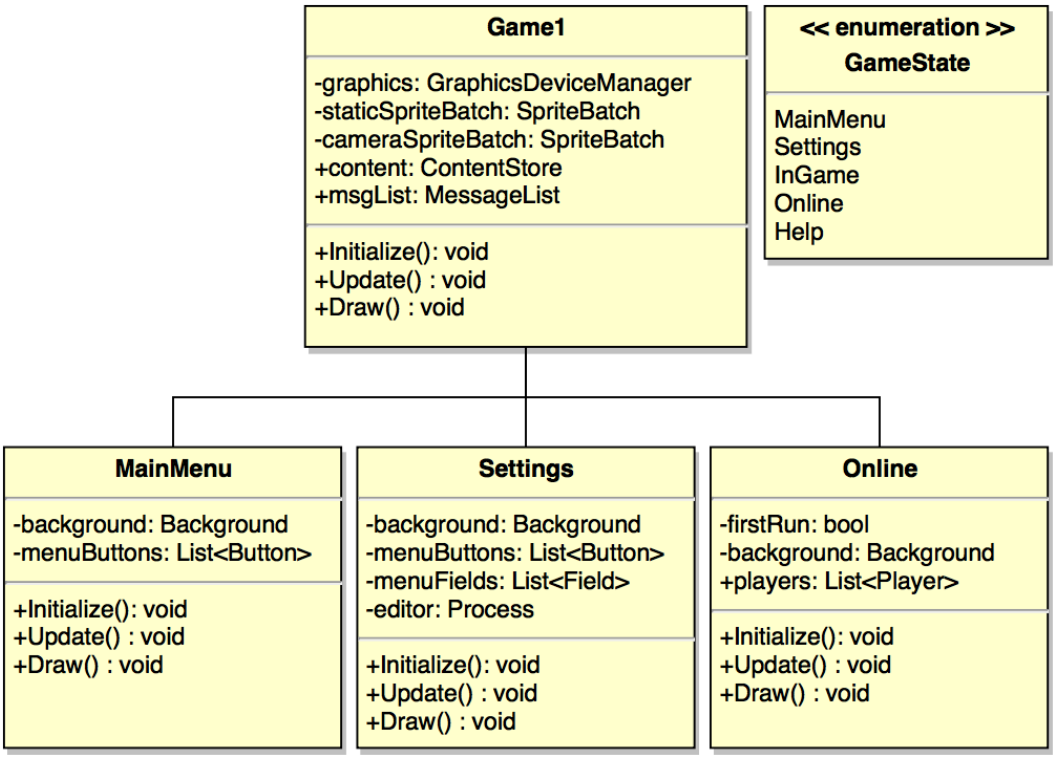
3.2 MoboClient



Architecturally, XNA framework applications follow a similar layered architecture. The application I have written runs atop the XNA extended framework which provides functionality to define an application model, that is, an application that is able to make use of XNA framework functionality. It is also this layer where the XNA content pipeline would normally process content such as graphics, fonts and sounds, but the Monogame SDK replaces the content pipeline with an external program used to build and compile content. The XNA core framework provides all the functionality for drawing graphics, playing sounds, capturing input from devices and more. Finally, the XNA platform level provides low-level APIs to render graphics such as Direct3D that are used in the core framework level.

The most basic of XNA applications define five essential core methods. They are, `Initialize()` , where game logic that needs to be present immediately at runtime can be initialized, `LoadContent()` , where content such as graphics, font and sounds are loaded in ready for drawing, `UnloadContent()`, where content can be destroyed from memory (not currently required in MOBO due to garbage collection), `Update()` , where game logic that updates at run time is placed, and finally, `Draw()`, where calls to the graphics renderer are placed to render graphical content on screen. During runtime, `Update()` and `Draw()` are called asynchronously, one after the other, so data updated by the update method can be drawn almost immediately.

XNA does not provide much more than methods to easily draw graphics and tools useful to games such as the Vector class. It does not automatically provide the ability to create menus, game states and take input from the user easily, since it is not an engine. Therefore it was necessary for me to construct my own classes and methods for transitioning between menus and changing game states. In the MOBO prototype, each 'screen' represents a different game state, such as MainMenu, SettingsMenu and Online. Each of these game states is an encapsulated class with its own `Update()` and `Draw()` methods that are called only when that particular game state is active.



Although simplified to reduce its complexity, the diagram above shows that depending on the game state active, the update and draw methods of Game1 (the main application class) will call the respective update and draw methods of the class corresponding to the current game state. In this way, each game state is essentially a smaller separate application and activity of game states that are inactive are suspended. This has allowed me to produce various 'screens' with different roles and appearances, and transitioning between these screens is easy and intuitive.

3.3 MoboServer

MOBOSERVER is a simple command line C# application as of the prototype. It is a simple collection of classes: Program is the application class which instantiates the network and holds business logic such as the tickrate of the server. Network is responsible for initializing the server, receiving messages from clients, parsing them, and sending messages back to clients. Player keeps a list of currently connected players and specifies variables that the server needs to know about each player, such as coordinates.

For the end product I would like to see MOBOSERVER as full-fledged Windows Forms application, with buttons for common server commands like restart and kick, ability to change the tickrate on the fly and text areas that show connected players and associated statistics.

4 Networking Technical Analysis

4.1 Overview

As mentioned previously, MOBO utilizes the Client-Server architectural pattern to enable network play. The dedicated server is a separate application that must be ran in parallel to one or more MOBO clients. In the current state of the project, MOBOSERVER follows the non-authoritative model[2] of networking servers which means the main role of the server is to propagate messages from individual clients to the rest as opposed to handling a full simulation of the current game and calculating game logic.

Currently, MOBOSERVER only stores a unique ID for each player (used to differentiate between players as opposed to their names), a player name, an x and y coordinate for positioning, a value to represent the current health of the player and a timeout variable used to disconnect clients that have lost/poor connection to the server. Note that this does not include player bounds for collision detection or data concerning projectiles, this logic in fact is handled client side (each client handles collisions to its own player object) and sent to the server for propagation. This is also similar for the movement of players, the server only retains the current position of players and each client is involved with handling the input from its own player.

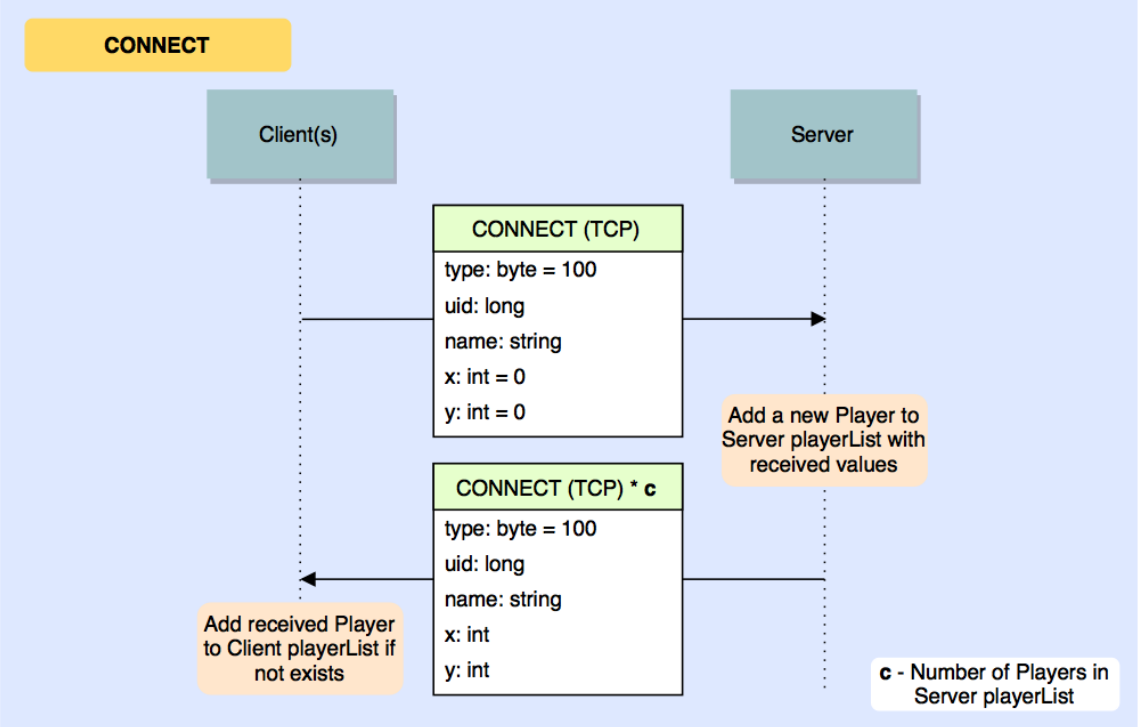
Messages are transmitted from client to server and vice versa using the Lidgren.Network library. The library provides functionality similar to Sockets in that it can be used as both a client that connects to a remote server and also as a server itself for aforementioned clients to connect to. The high-level message types that Lidgren.Network employs are two-fold: 'Library' messages and 'Data' messages. Library messages are automatically generated by the server to convey information pertaining to the current state of the server such as warnings or errors. Data messages are defined by the software developer using the library and are tailored to convey specific messages between the client and server.

For MOBO, I have made extensive use of Data messages to transmit client state between clients using the server as a means of propagation. These messages are then categorized by the contents of its message with a simple byte header and each message type is handled and parsed individually by the client and server depending on the functionality the message provides. Each of the following sections describes a key functionality to multiplayer networking in MOBO and the core message structure that underpins how it works. For reference, the message types and associated byte headers that distinguish them are as follows:

Message type	*Byte value*
CONNECT	100
MOVE	101
DISCONNECT	102
SHOOT	150
HEALTH	151

Lidgren.Network allows for messages to be sent with a variety of protocols[3]. For CONNECT, DISCONNECT, SHOOT and HEALTH, it is essential that these messages are successfully sent and received otherwise inconsistencies will arise between clients and thus are sent using TCP. MOVE messages are sent between the client and server very frequently and are sent using the UDP protocol as it is not catastrophic if an individual MOVE message is lost or duplicated.

4.2 Connecting to a Server

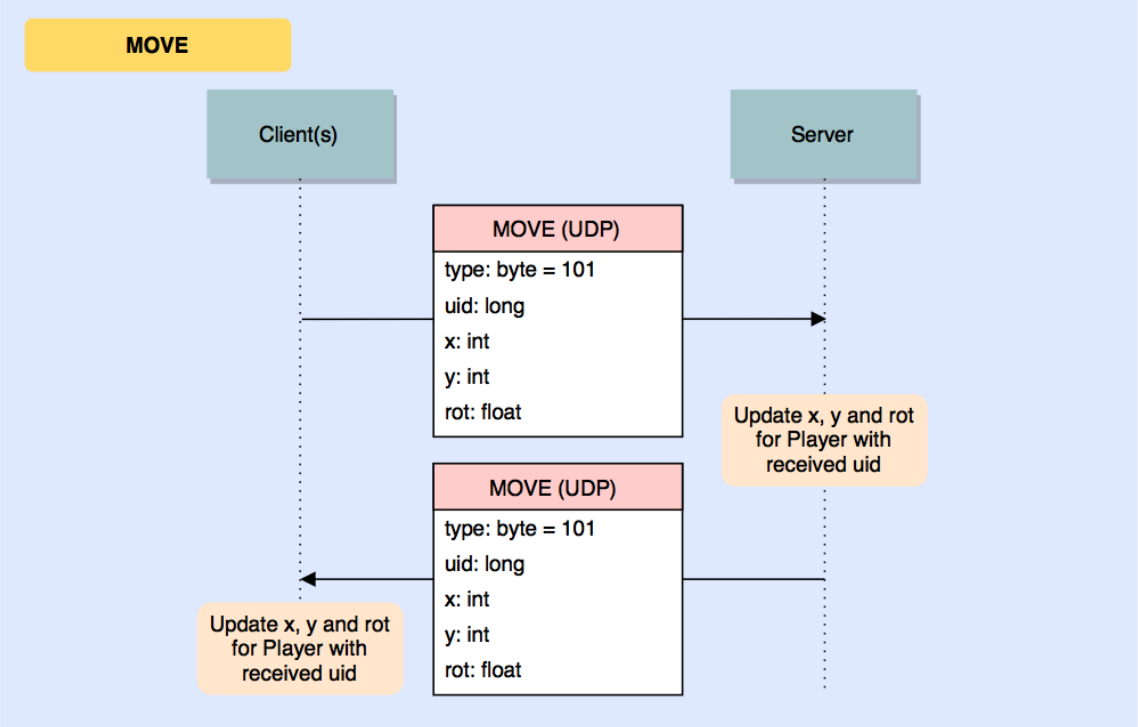


The process of connecting a MOBO client to a MOBOSERVER instance begins when a user selects 'Join Server' on the client main menu. A NetClient object is initialized using the Lidgren.Network library and will form the interface by which messages are created and sent. The NetClient then attempts to make a connection using the specified server IP address and network port specified in the MOBO configuration file. If the server is offline or otherwise unreachable the user is returned to the main menu.

As described in the diagram above, the client then sends an initial CONNECT message to the server. The message consists of the byte header that identifies the message as a CONNECT, a long (signed 64-bit integer) generated by the Lidgren.Network NetClient object that uniquely identifies the client, the name of the player obtained from the configuration file as a string and two ints (both always zero in the prototype) that defines the starting coordinates of the player. Post-prototype it may be wise to randomize player starting positions a little; players with the same coordinates overlap completely which hides all players but the player on top and can be jarring.

Once the server receives the CONNECT message, it reads in each variable separately before instantiating a Player object with the unique ID, player name, coordinates and a timeout variable, initially zero, that defines the number of ticks since the last message received from that player. The server then sends a CONNECT message to the clients with identical structure to the previous for every player. Each client then receives this sequence of CONNECT messages and decides whether the player defined in each message has not been added to the Client's Player list already, and if so the respective player is added. This ensures that whenever a player joins a server, its client receives all the current players and in turn, that player is added to all previously connected clients. This is essential for the 'Drop-In-Drop-Out' gameplay concept I wish to achieve with MOBO.

4.3 Movement Synchronization

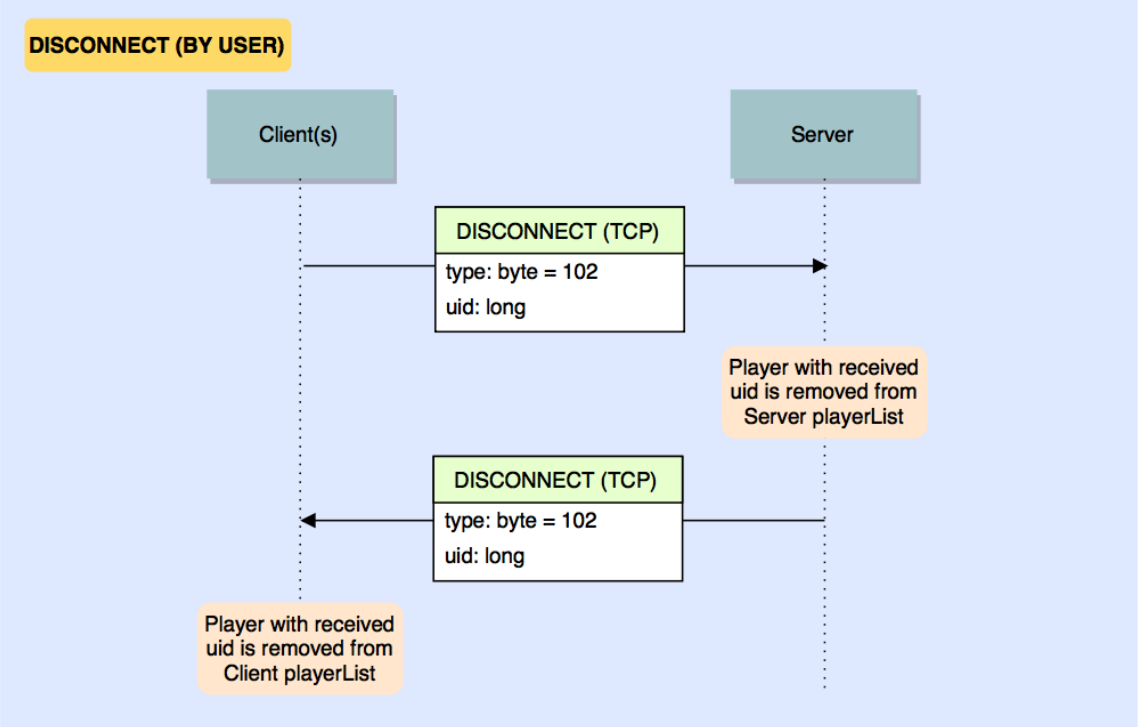


A key feature of the prototype is movement synchronization, i.e. being able to see players move about independently whilst transmitting the coordinates of the local player to other players. In MOBO, this is achieved by the server by receiving coordinate data from each player in a parallel fashion, the server updating its own records of current player position and finally the server sending a large batch of player coordinates to each client so that they can update the positions of other players themselves.

If a client has successfully connected to a MOBOSERVER instance and has joined a game, it immediately begins broadcasting MOVE messages pertaining to the local player to the server. These messages consist of the MOVE byte header, the unique client ID of the player for which the data belongs to, coordinates of the player and a rotation float value that defines the current direction the player is pointing in radians. On the client side, rotation of the local player is calculated using the inverse of the trigonometric function tangent using the velocity of the player (change in x and change in y each tick). Since only absolute coordinates are used to synchronize remote network players it is necessary therefore to include this rotation value in move messages as it cannot be calculated for network players. I have experimented with using relative velocity values instead of absolute coordinates - which would render sending a rotation value in MOVE messages obsolete - but after some time clients become desynchronized due in part to UDP packet loss and latency between client and server.

The server consolidates the coordinate data it receives and repeatedly sends batches of MOVE messages containing the coordinate data for all players to all connected clients. This does not occur for every move message the server receives like the diagram may suggest, but instead is tied to the pre-defined server tickrate which regulates how often per second the server updates. For the prototype this is roughly 60 times per second, or every 16ms.

4.4 Disconnection events



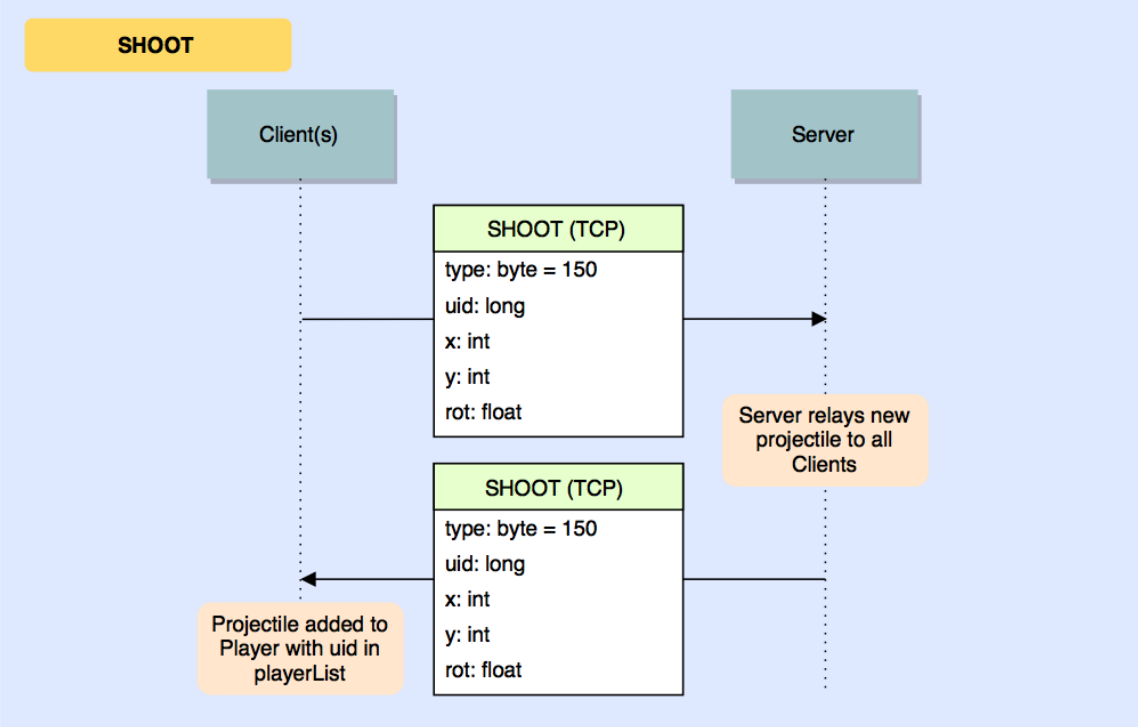
In any type of multiplayer networked game it is important to know when clients need to disconnect and how to act accordingly. In MOBO, we not only need to remove the player from the server and close the connection but also to tell all remaining connected clients that the player has disconnected. The MOBO prototype currently has two methods by which disconnection events can occur, and this will likely be extended by a few more for the final product.

The first disconnection event occurs when a player voluntarily leaves the server by closing the application. Just before `this.Exit()` is called to terminate the application process, the client checks whether it is connected to a server, and if so sends as DISCONNECT message to the server. The message consists of the byte header that identifies it as a DISCONNECT and the unique identifier of the player client that is disconnecting. The server then needs only to remove the player with the associated ID from its list of players, and propagate the same message to all remaining clients. The clients receiving the message then behave in an identical fashion by removing the player from their lists with the uid. This event only occurs when the application is closed normally and if the client crashes or otherwise loses connection to the server another event must occur to compensate otherwise the disconnected player will persist in the player lists of both the clients and server long after they actually disconnected.

To rectify this problem, a second disconnection event occurs when the server has not received a MOVE message from a client for 180 ticks (roughly three seconds). On each tick the server loops through the connected players and increments a timeout variable for each player. Successfully receiving a MOVE message from a player resets its timeout to zero.

A potential addition for a disconnection event will be ‘kicking’ players from the server. Kicking is a common feature of online games employed to remove unwanted players from a game. It is desirable that this feature will be built into the server as a runnable command for the final product.

4.5 Projectile Synchronization



Another feature of the MOBO prototype is the ability to shoot other players and decrease their health. The creation of projectiles is handled client-side when players press or hold the space key. A timer which initializes as soon as a new projectile is created ensures a cool down of at least 200ms between each shot. The movement of projectiles is also calculated client-side, with projectiles only storing a current position and current velocity (change in x and change in y vector). Projectiles are synchronized across clients by using SHOOT messages. Since projectiles in the MOBO prototype never change direction or speed, it is enough to send the initial starting rotation and position of a projectile being fired for its initial velocity can be calculated client-side in order for it to appear and travel in an almost exact path between clients.

When a player fires a projectile, a SHOOT message is sent to the server with a SHOOT byte header, the unique identifier of the player that shot the projectile, starting coordinates and the rotation of the player in radians. The server in this instance acts only as a message propagator, the server as of the prototype build does not store any data pertaining to projectiles. When the clients receive SHOOT messages, the projectiles are added to the projectile list of the player with the uid in the message so we are able to track who shot the projectile (to give them credit for successful hits potentially) and they appear on all clients.

In my opinion the projectile system is suitable for the prototype but not mature enough for the final product. Theoretically, high latency of the connection between the client and server can cause desynchronization of the projectiles. Essentially the local player sees their projectile being fired almost instantly, but due to a latency of 100ms for example, the projectile may appear a split second later on the other clients. A small difference for sure, but it may be the case that a player sees a successful hit on another player, but the hit isn't registered due to desynchronization. This will be a key area of research as the second phase of the project commences.

5 Career Plan

5.1 Where do I want to go after graduation?

In early 2015 I successfully applied for an internship at BAE SYSTEMS Applied Intelligence for which I went through a screening and multi-stage interview process. The internship lasted three months spanning the length of the Summer holidays and shortly after leaving I received a permanent job contract for Summer 2016 after my graduation. Having now accepted the contract for next year, I do not plan to apply to any other places as my graduate job is secure.

In the long term, after I have some level of financial security and experience, I wish to look for a job at a smaller start-up developing innovative technology and at the forefront of the industry. An example of a contemporary company like this would be Oculus, who have pioneered and popularized practical virtual reality technology that has the potential to dramatically change the face of how we experience media. As technology continues to exponentially improve and diversify on each passing year, I expect the face of the industry to be very different by the time I will be seeking this move, with many new opportunities available, and indeed is something that I am very excited for.

5.2 What will I do this academic year to get there?

Despite already having a graduate job I do not intend to rest on my laurels. By freeing up the time I would normally be writing cover letters and applying for jobs, I now have invaluable time to further develop my technical skills and improve my craft. In my own time I am currently learning the Drupal open-source content management system as it can be used effortlessly create web applications, from simple dynamic webpages to more complex systems. I have already produced a commissioned website in Drupal and I believe it is a web technology worth learning and showing I have experience in.

In my own time I continue to learn and enjoy developing for the Android operating system. Many companies now seek experience in mobile app development, so having experience in Android will no doubt prove valuable for my employability.

5.3 How does my project contribute to my career?

Tackling a project on this scale and the necessity to manage one's time independently and effectively has provided me with a unique opportunity to develop my time and project management skills. In particular I have learnt how to break a project down into manageable, atomic tasks and to how to prioritize tasks based upon the relevance to the objectives. In addition in creating such a user-centric product, I believe by the time I have completed the final product I will have gained valuable experience in designing interfaces, maximizing user experience and usability testing.

C#, the language which I am currently programming the system in, is not a language I have prior experience with, but is very intuitive and enjoyable for me. It is quickly becoming my preferred programming language and by the end of the project will be a language I will be able to add to my CV and show I have experience and knowledge in. Other useful technologies I have learned that will no doubt be useful to my career include XML parsing, defining message protocols for networking and, in general, designing a fairly complex client-server system in a correct and suitable fashion.

6 Bibliography

1. Source Multiplayer Networking. *Valve Developer Community Wiki*, Accessed 5/11/2015, https://developer.valvesoftware.com/wiki/Source_Multiplayer_Networking
2. High Level Networking Concepts. *Unity Manual*, Accessed 7/11/2015, http://docs.unity3d.com/Manual/net-HighLevelOverview.html
3. Lidgren Network Basics. *Lidgren Networking Library Google Code Wiki*, Accessed 7/11/2015, https://code.google.com/p/lidgren-network-gen3/wiki/Basics
4. Lecky-Thompson, Guy W. *Fundamentals of Network Game Development*. Herndon, VA, USA: Course Technology / Cengage Learning, 2008.
5. Rabin, Steve. *Introduction to Game Development (2nd Edition)*. Herndon, VA, USA: Course Technology / Cengage Learning, 2009.
6. Mount David. Varshney Amitabh. *Networking and Multiplayer Games*. Chapter 9 University of Maryland 2011.