DEGREE PROJECT

Implementation of Network Components for Game Projects

Gunnar Kollnitz

Computer Game Programming, bachelor's level
2018

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering

Abstract

Online multiplayer is the focus of gameplay for a majority of games released in this day and age, and at the core of multiplayer lies the network code. This means that bad networking code can be ruinous for an entire game. A Sweet Studio is a small game studio that both works as a consultant in the industry as well as develop their own games. But with limited time and money to spend on their own projects, it can be difficult to develop new games.

The purpose of this paper is to cover how a multiplayer base can be developed in Unity, iterated on, and how to build it for reusability in a way best, that implementing it in new projects is as simple as possible. The end result is a library with broad functionality and a simple layout.

Sammanfattning

I majoriteten av datorspel som släpps nuförtiden, är flerspelar gameplay ett av de största fokusen, och i kärnan av den funktionen finns nätverkskoden. Detta betyder att dålig nätverkskod kan förstöra upplevelsen helt och hållet. A Sweet Studio är en mindre spelstudio som både gör konsultarbete inom industrin, och utvecklar sina egna spel. Men med begränsad tid och begränsade pengar att spendera på sina egna projekt, kan det bli tufft att skapa nya spel.

Mening med den här rapporten är att gå över hur en flerspelarbas kan utvecklas i Unity, itereras över och hur den bäst byggs för återanvändning, så att det lätt går att implementeras i nya projekt. Slutresultatet är ett bibliotek med bred funktionalitet och en simpel layout.

Acknowledgments

I wish to thank everyone that helped me on this project. I huge thank you to my supervisor Patrik Holmlund, who has helped with and given valuable feedback on my drafts. I also wish to thank A Sweet Studio for giving me the opportunity to work with them on this project. And lastly, I am grateful to Luleå University of Technology' Campus Skellefteå and all the people I've met there who have helped me inside and outside of my education.

Contents

1 Introduction

Multiplayer games have been around for almost 60 years, but most of that time only as non-networked types. LAN *(Local Area Network)* games became popular in the early 90s, and not long after online games rose up with games such as *Everquest*, and later on *World of Warcraft* [10]. Looking at the number of singleplayer AAA titles released compared to those that include multiplayer, it becomes quite clear what the majority of players want. There were a few very successful singleplayer games released in the last year, e.g., *The Legend of Zelda: Breath of the Wild*, and the most recent *God of War*. These games, however, are only exceptions to the rule. You see games like the *Far Cry* series that early on only were singleplayer experiences, which are switching to a more co-op *(Cooperative gameplay)* style. Of the ten best selling games in the US according to GameSpot [11], only one of them is without any multiplayer features.

A Sweet Studio [1] is a small game studio located in Stockholm. They are founded by industry veterans, who have decided to step down from the more significant game industry to make their own games. Most of the work they do is consultant work, but they also develop their own games on the side. However, as a smaller studio, they have a hard time testing new concepts and game ideas. This is mostly because they have a limited workforce and to try out a new game idea might require the work of half the employees. To reduce the amount of work necessary to develop a proof of concept for a new game, reusing assets is the most efficient thing to do. Since multiplayer is a core mechanic of the games designed by A Sweet Studio, making a reusable framework for it would save a lot of time.

The purpose of the project is to create a multiplayer base that a user can implement with ease into any project, and with minimal setup be able to provide the game with the most standard multiplayer functionalities. The key here is to get a right balance of making it easy to implement, and making it easy to adapt for the user so that it fits the game they are making.

1.1 Method

1.1.2 Social, Ethical, and Environmental Considerations

The project depends primarily on a third party cloud service that handles all traffic between players and the game. This removes most if not all capability for the work to breach any privacy of the players. Since the main purpose of the project, in the long run, is to increase efficiency and reduce the workload, it benefits not only the company but also the employees and end users. Instead of having to reinvent the wheel each time they wish to try out a new game they can spend that time on being creative, which in turn reduces stress.

1.1.2 Unity

At A Sweet Studio, they use Unity [2] to develop their games, which therefore also became the go-to environment for the multiplayer base. Unity is a cross-platform game engine, with scripting in C#. It is one of the most popular game engines on the market, with over 5 billion downloads [7], and companies as big as Blizzard Entertainment using it [8].

Because of its popularity, there is a significant number of free add-ons and packages available that can enhance the engine as well as plenty of open projects and tutorials that provide the user with all the information needed to develop games and other projects in the engine. All this makes it a great game engine for fast game development which is what Sweet is known to be able to do.

1.1.3 Photon

Photon [3] is a multiplayer engine for cross-platform real-time multiplayer games. Photon provides a cloud for hosting games, as well as features such as player rooms, lobbies, customizable matchmaking, and cross-platform interaction. Photon is available on many other platforms and game engines as well, which can enable an easy porting of the multiplayer base too, e.g., Unreal Engine [9]. Photon's platform availability makes it a great option to Unity's networking functionality. The Photon Engine cloud is simply a large number of machines running their Photon Server [12] product, which means that there is always an option to use in-house servers instead. The Photon payment plan [13] is for the user to buy CCU (Concurrent Users) slots, with 20 slots given for free initially to properly be able to test the game.

1.1.4 Steamworks

The Steamworks SDK [4] provides a link between games and the steam distribution platform [5]. It offers a wide range of features to help ship games, such as friends lists, invitation management, matchmaking, out-of-game inventory, anti-cheat technology, and profiles. All elements which are standard in multiplayer games.

The main reason why the multiplayer base is not solely based on Steamworks is that it lacks support for the Xbox platform, which Photon does not.

1.2 Abbreviations and Terms

- **Game Engine** - A software framework designed for creation and development of video games
- **Unity** - A game engine
- **Steam -** A video game distribution platform for PC, Mac, and Linux
- **C# -** A object-oriented programming language, developed by Microsoft
- **SDK -** Software Development Kit
- **API -** Application Programming Interface

2 Design and Implementation

2.1 Research

Since A Sweet Studio [1] has several released games, many features of this project were present in their previous games. Although those implementations weren't all reusable, they were still of great help in planning out the different features. This included code snippets of more complicated functions, as well as structural information.

Unity [2] is one of the most popular game engines on the market at the moment which means that there are solutions for most if not all problems concerning Unity [2] that might appear. The Photon Engine [3] is also widely accessible, and there are plenty of informative sites that go over the SDK thoroughly, most specifically Photon's own documentation [14].

2.2 PUNBehaviour

Most Photon [3] calls and functions communicate with the cloud server. The communication is in the form of data packages, that take time to travel to the client and the server. This means that the implementation cannot be expected to run without intervals of delay. To control these intervals, there is a class called PUNBehaviour (Photon Unity Networking Behaviour). It contains callbacks/events which are called by the cloud as it executes.

For examples of callbacks see Table 1.

Table 1. Examples of Callbacks

OnConnectedToMaster()	*Called when the user connects to the master server.*
OnJoinedRoom()	*Called when the user has successfully joined a room*
OnReceivedRoomListUpdate()	*Called when the client has received an updated list of all potential rooms on the server.*

This means that when communicating with the cloud, it is more of a wait-and-listen type of workflow. Depending on what the server tells the modules, they need to act differently.

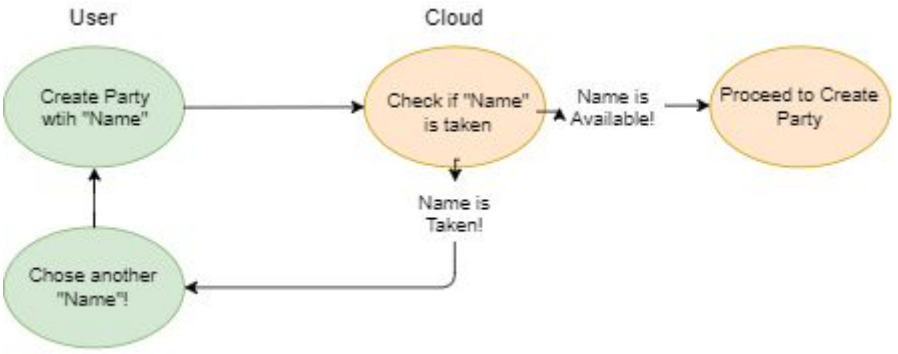


Figure 1: *Create Party communication flow*

For example, as shown in Figure 1, when creating a party, the party module needs to wait to see if the party name is available before it can create the room.

2.3 Modules

The network components are supposed to be split up into different modules, and each of them had to be as independent of each other as possible. This enables the user to pick and choose what network properties a game should have, and not force them to implement an entire framework. The defined modules decided upon by A Sweet Studio [1] for the project were, Authentication, Party, Matchmaking, Lobby, and Character Synchronization.

2.3.1 Authentication

The network was built with Steam in mind as the primary distributor. Therefore all aspects of the authentication of players could be done using the Steamworks API. This included signing in, accessing the nickname of the player, the avatar, and their Steam ID. Since this was to be the only required module for the network, many of the variables of the framework and settings were set up and defined here. This included values that are used by many other of the other modules.

However, the Steamworks SDK[4] doesn't support C# (which is the script language used in Unity [2]). To bypass this problem, Steamworks.NET [6], a C# Wrapper was downloaded and implemented in the project.

2.3.2 Party

The Party module was planned to be a core component for co-op games, where the forming of a party is an essential part. It was defined to have five core features; the player must be able to invite other players, kick players from the party, join another party, access the other members of the party, and set the party to either public or private. To enable invites over the Steam [5] platform, there needed to be an overhanging Steamworks [4] method for the otherwise purely Photon [2] based module.

2.3.3 Matchmaking

Matchmaking is a core component for all player versus player games, as well as other online co-op games. Most of the time a premade party will not fill all slots in a multiplayer game; instead, they automatically need to be paired with other users looking to play. This module had to be very moldable by the user, since matchmaking can work very differently depending on the game, with different amount of players, teams, game modes and depending on the game, even player rankings can change how you want it to work.

2.3.4 Lobby

After the players have successfully found a game by using the Matchmaking module, they join a lobby. This is where the teams are set, as well as where other pregame events take place. This module is the last step before the players load into their actual match.

2.3.5 Character Synchronization

The last and final module was Character Synchronization. The purpose of this module was to enable syncing of specific in-game objects across all players. This module was critical since a poor implementation of it could completely disrupt the player's experience. This module took the shape of a C# script, whichthe user could place on any Unity objects that they would like to be synchronized. Synchronizing characters was done using Photon [2] callbacks where all characters with the script, are always sending out the values as well as receive the information of other clients, through the server, that require synchronization.

2.4 Demo Menu

To better implement the modular framework, a testing ground was needed. Since the majority of the functionality of the modules only is present in the “pregame” state of the games, it made sense to develop a simple menu to test all the functionality. The menu was built to resemble the menu an actual game might have, to better get a feel of how to implement the modules.

2.4.1 Panels

The menu was built using different panels, a few seen in Figure 2, where each panel presented a different interface and module to the player. The various panels being:

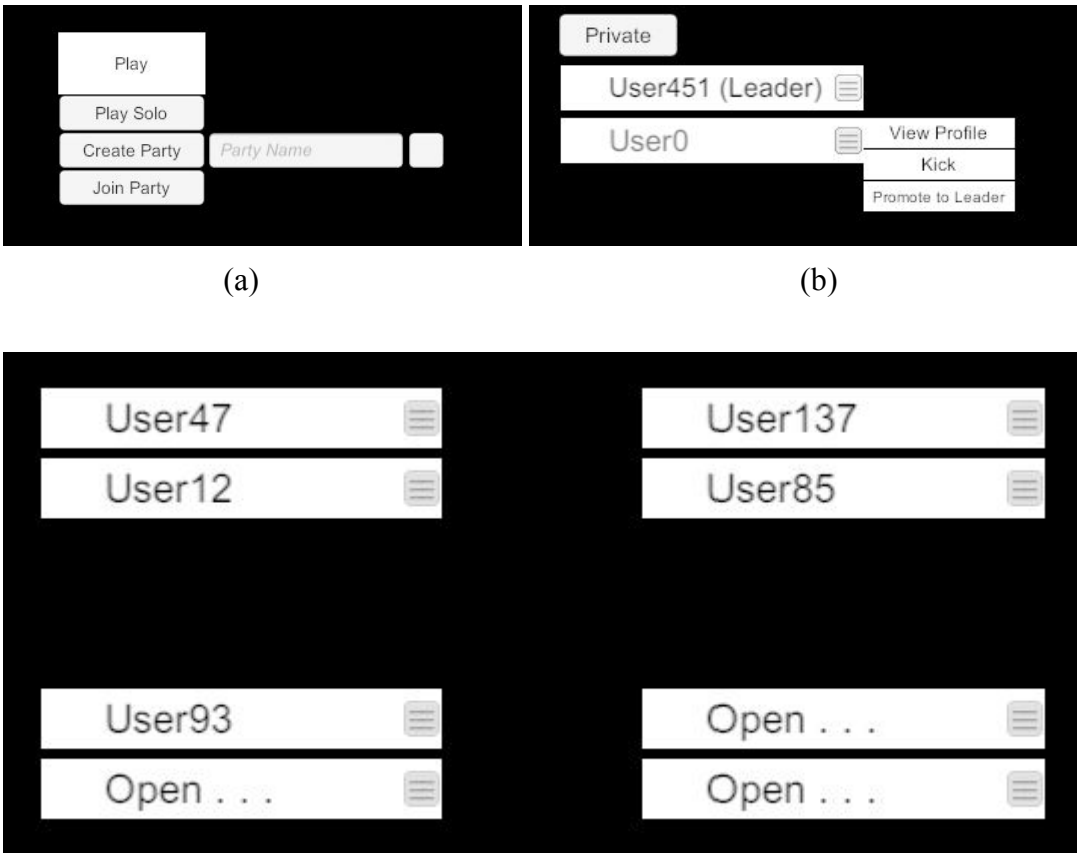
- StartPanel. The StartPanel contains a small menu, with three different options in how to continue through the interface.
    - Play Solo. Activates the matchmaking module for the player, taking them to the SearchingPanel.
    - Create Party. Lets the player specify a name for their party, and create it. This takes them to the PartyPanel.
    - Join party. Lets the party specify a name for the party they want to join. If successful, it takes them to the PartyPanel.

- PartyPanel. In the PartyPanel one of the players will be given the role of Leader, by default, the player who created the party. As players join and leave, the panel will update to display the correct members of the party. The panel contains a small menu anchored to each player with three options.
    - View Profile. This option is usable by all players in the party, and on activation will trigger the steam overlay, and bring the using player to the selected player’s steam profile.
    - Kick Player. This option is only usable by the leader of the party and on activation will remove the selected player from the party, and bring them back to the StartPanel.
    - Promote to Leader. This option is only usable by the leader of the party and on activation will change the leader of the party to the selected player.

Other than this, the leader will also have an option to change the party type from public to private. To only allow friends on steam to join the party manually. They also have the opportunity to start the search for a matchmade game. This will bring all the players of the party to the SearchingPanel.

- SearchingPanel. This panel serves no real purpose outside of being a waiting room to the LobbyPanel. It holds a text stating how many players that are currently in the same waiting room, out of the amount needed to start. When the waiting room is full, the players swap to the LobbyPanel.

- LobbyPanel. In the LobbyPanel, the teams and its players are visible for all to see. If the game is not team-based, the panel displays them in a single list. When all players have entered the LobbyPanel, a countdown will start signaling the start of the game. Like the PartyPanel, players can here also view the profiles of the other players.



(a) (b)

(c)

Figure 2: *Illustration of the different panels that make up most of the Demo Menu. a) shows the StartPanel, b) shows the PartyPanel, and c) shows the LobbyPanel*

The panels were at their core the same. This was so that it would become easier to manage them together with the network components. Their visibility could easily be toggled on and off to emphasize the properties of their backend better.

2.5 Demo Game

A simple 2D game was made to test the Character Synchronization module properly. To properly demonstrate the syncing between the movement of the characters across networks, this was crucial. The best way to do this was to create a simple 2D map and populate it with obstacles, as seen in Figure 3.

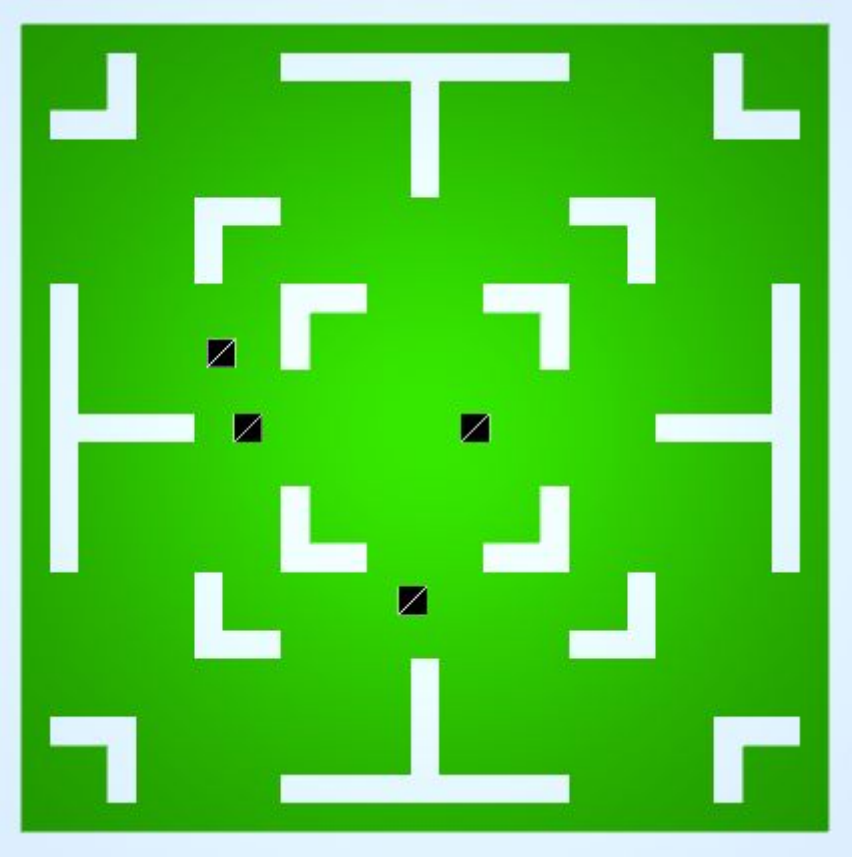


Figure 3. *An illustration of the Demo Game.*

This stage was first tested using multiple clients on the same computer, and when proved successful the project was exported to numerous machines and tried over the internet there as well. While the script component itself worked fine, it did become apparent that it would require custom made physics properties on the objects it inhabited.

2.6 Iterations

When creating a system, where core philosophy is reusability, it is much more critical to build a good foundation, than it would be if it were intended for a single project only. This way it can be changed over time, and iterated upon continuously.

2.6.1 Climb

After building the first functioning framework, it was moved from its own project to an ongoing game project, Climb. The best way to test the usability of the network components is to implement them in a real project.

Climb is a 2.5D game with fairly simple game modes such as deathmatch, and a collect-rings game mode. It draws inspiration from the Ice Climbers game, by Nintendo for the NES game console. Figure 4 shows an early version of the games main menu.

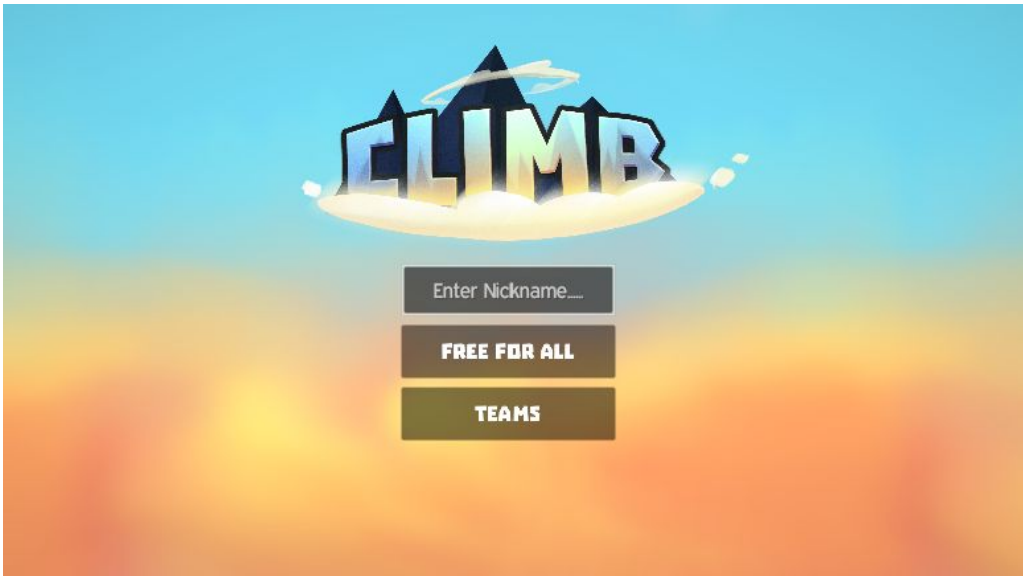


Figure 4. *Illustration of Climb start menu screen*

The transition was seamless except for a few unexpected errors in setting up teams in the lobby where for example the players of a premade parties were treated solo players, resulting in split parties. This was also a great chance to get direct feedback since many people were working on Climb already. The modules received further changes, which helped shape the framework to its final form.

3 Result

The result of the project was a Unity Package, containing a library, and a test menu. Even those who were not the most capable programmers could still utilize the end design of the library to its full potential. Developing it this way did, however, simplify and reduce the amount of customization available on the surface of the library.

3.1 Library

Six C# scripts make out the entire library, one for each module, and one to receive callbacks. Each script represents its own class with specific methods for the user to implement.

3.1.1 Authentication

The authentication module is closely connected to the Steamworks SDK [4] and is relying on the fact that the player has steam running in the background and a valid copy of the game. Table 2 displays the methods available to the user.

3.1.2 Party

The party module is relevant to all games that desire premade groups. Without the party module, players will only be able to go through matchmaking by themselves. Many of the party methods are usable and in fact very useful in other stages of the game, e.g. *ViewProfile()*, and *InviteToParty()* found in Table 3, that could be used in the lobby as well as in game.

Table 2. Authentication methods

Initialize()	*The framework always needs to call this method as it initializes all variables and values that need to be set for the framework to function properly.*
GetSmallAvatar() GetMediumAvatar() GetLargeAvatar()	*Returns the avatar of a player as either a 32x32, 64x64, or 184x184 size texture.*

Table 3. Party methods

CreateParty()	*Creates a party with a given name, which is private on creation which means it won't be visible to everyone.*
JoinParty()	*Attempts to join a party with a given name.*
PromoteToLeader()	*Changes the owner of the party to another member.*
KickFromParty()	*Removes a party member from the party.*
ViewProfile()	*Toggles the steam interface and directs the player to the steam profile of another player*
LeaveParty()	*Leaves the current party, and returned to the global lobby.*
InviteToParty()	*Invites a steam user to the current party. If accepted, the invited user will automatically start the game if not already running, and join the party.*
RejoinParty()	*If the end user wants the players to return to their party after a match/game, this will let all party members find their leader and join them.*

3.1.3 Matchmaking

The methods seen in Table 4, can be called with parameters that decide the matchmaking pool to search in. This allows for multiple game modes, search queues and even a competitive ranking system. The methods also set a party id depending on the players premade team or if they are playing solo, which makes it easier to later on group people up into teams, as their ID specifies if they are queueing alone or with others, or give other visual cues depending on premade groups.

Table 4. Matchmaking methods

MatchmakingFromParty()	*Starts the matchmaking process for the entire party. The leader finds a suitable game, and the other party members follow.*
MatchmakingFromSolo()	*Starts the matchmaking process for the player only.*

Table 5. Lobby methods

SetupTeams()	*Groups up the players currently in the lobby into teams, depending on the game mode and premade parties. Solo players get put in groups as well, as long as they have available slots and are not expecting more party members to join.*

3.1.4 Lobby

The lobby module only contains one method, *SetupTeams()*, as seen in Table 5. That does not, however, make it a trivial module. There is also functionality for a Ready-system, where if all players indicate that they are ready, the game will start. This is however very dependant on what type of game it is.

3.1.5 Character Synchronization

The Character Synchronization module works differently compared to the rest of the modules. While the other modules see use in most stages of the game, this script is placed on a game object in the game, and will afterward sync all its physical properties to the server. It has no methods that the end user can use but has a function that the Photon cloud calls for, *OnPhotonSerializeView()*. This is where the script communicates with the Photon cloud, sending information about the object which has the script, and receiving similar information.

3.2 Test Menu

The test menu is a cleaner version of the Demo Menu with a more straightforward UI, to make it easier to understand the methods and modules used to connect the different elements in the UI. It provides visual representations of how to implement different matchmaking types, as well as how parties work.

4 Discussion

The outcome was good, with a large bunk of methods and functionality for any upcoming projects. There are, however, a few things that could have been done better in the development of the task. The big thing is more in-depth planning when it comes to the code. There were many small band-aids put on over the implementation that is now heavily relied upon while not being as efficient or easy to use as they could have been, had they been planned out properly, though this was in most likelihood a symptom of lack of experience.

4.1 Limitations

While the project was in development, two philosophies affected the project the most. It had to be modular to make it easier for the user to modify it and chose what parts they wanted to use for their project, and it had to be easy to use. These two proved to be a challenge to balance in a way that would not make the end product a mess. Because it had to be easy to use, some functionality did not make it all the way, and the framework a bit more static in its customization. In the *Lobby module,* there is an excellent example of this, where all the teams have to be the same size since without rewriting the method it would gain an extra layer of difficulty to the use of the module.

4.2 Future Work

The framework developed will most likely continue to be used for projects at Sweet [1], and while most of it has a solid foundation and core, there is still room for it to be iterated upon even further. Specific features that would complement the package would be a complete premade ranking system for the matchmaking module. It could also benefit from redesigning the lobby module a bit, with allowing for nonsymmetrical team sizes, such as 1v4 or 1v3v3, and so on. Doing that would allow for many exciting games and game modes. While these could all be custom made for each game, being able to provide them together in the framework would be of great value. The reason for using Photon in this project is because it is compatible with the Xbox platform. But at the moment there is no support for Xbox implemented in the framework, which is something that definitely should be added in later iterations.

5 Conclusion

In conclusion, this project reached its goal of creating a modular framework of network components for Unity [2] driven game development. The implementation provides the tools and information needed to implement multiplayer functionality into most Unity [2] projects. It fulfills the goals set by Sweet [1], and while there is room for a lot more work, it is a stable foundation for a continued building process. Because of the modularity of the project, the methods used are broad enough not to restrict itself to Unity, but can instead see rebuilding in other environments and game engines with ease. Using this framework should enable a lot faster creations of proof-of-concept's which in turn increases the effectiveness of the development.

When comparing this project to some of the previous implementations of similar components like those observed in A Sweet Studio's previous projects, while not necessarily a faster or cheaper solution, it provides the context needed to understand it. For a project with only one person working on it, context will not be as important, but when working on a project where multiple people need to understand the solution, it can be crucial.

6 References

[1] A Sweet Studio. [2012]. Retrieved from http://www.asweetstudio.com/

[2] Unity. [2005]. Retrieved from https://unity3d.com/

[3] Photon. Retrieved from https://www.photonengine.com/en-US/Photon

[4] Steamworks SDK https://partner.steamgames.com/doc/sdk

[5] Steam. [2003]. Retrieved from http://store.steampowered.com/

[6] Steamworks Net. Retrieved from https://steamworks.github.io/

[7] Unity Public Relations. Retrieved from https://unity3d.com/public-relations

[8] Unity Card Life. Published: June 19, 2014. Retrieved from https://unity3d.com/showcase/case-stories/hearthstone

[9] Unreal Engine.[1998]. Retrieved from https://www.unrealengine.com/

[10] Multiplayer video game. In *Wikipedia*. Retrieved May 03, 2018, from https://en.wikipedia.org/wiki/Multiplayer_video_game

[11] Makuch, E. (2018, January 18). Top 10 Best-Selling Games Of 2017 Revealed In The US. *Gamespot*. Retrieved from https://www.gamespot.com/articles/top-10-best-selling-games-of-2017-revealed-in-the-/1100-6456205/

[12] Photon Server. Retrieved from https://www.photonengine.com/en/OnPremise

[13] Photon Payment Plan. Retrieved from https://www.photonengine.com/en/PUN/pricing#plan-20

[14] Photon PUN Documentation Retrieved from https://doc.photonengine.com/en-us/pun/current/getting-started/pun-intro