Final Report

Multiplayer First Person Shooter Game Framework

Chris Bachelor
Melodie Butz
Joseph Garvey
Alexis Korb
Wyatt Watkins

TABLE OF CONTENTS

1. Motivation and Application Overview

The Multiplayer Game Framework is a framework for the Unity game engine that allows developers to add efficient lobby-based multiplayer to their game. It is implemented as a Unity project and illustrated through a simple first-person shooter game. It is intended for primary use by Escality but can be adapted for many types of developers and games. Components of the framework include a Server, Client, and Master Server. Developers host a Master Server which connects to all dedicated servers running the game. These other Servers host games created by users, while the Master Server facilitates match-making and Steam friends list invites/joins. Servers hosting lobbies control an authoritative game state for each lobby, which is propagated to its users to ensure gameplay cohesiveness. To adapt the framework for a developer's particular game, they only need to specify which data must be synced by the server, which is highly specific to the particular game. Client-server communications are built upon the free-to-use UNet Transport Layer API, and Steam friends are collected through open-source Steamworks.NET.

While the game Server and Client both run using the Unity game engine, there must be a middleman to connect them together on the internet. This is done through a Master Server written in C++ for Windows machines. There must only be one Master Server running on the internet facilitate the game, but it must be on a machine with either a static IP address or with the port 8484 forwarded on the router. This is unavoidable due to NAT translation and firewalls, as there must be a single, canonical access point. Otherwise, each single dedicated server instance launched for this game must access forwarded ports or use a static IP.

2. Requirements

2.1 Client Requirements

Listed below are the requirements specified by our client, Escality. Since this should be a very scalable framework that should be adaptable to different games with different amounts of data to synchronize, it is hard to judge performance metrics.

Functional	Nonfunctional
● Dedicated server architecture ● Multiple players per lobby ● Multiple lobbies per server ● Player Interaction 　○ Synchronized movement ● First Person Shooter gameplay	● Limit lag, smooth player movements 　○ Should be able to host 16 or so players in a match at least ● Free to host (no paid software included) 　○ Open source Steamworks.NET and only using the low-level portion of UNet API

2.2 Requirements Changes

Our requirements have not changed much since the midterm period. Since the midterm, we have added true networking to the project, as without the Master Server communication could only happen over localhost. This made us have to account for packet loss in communication between server-client and between server/client-MasterServer. But the bulk of our goals and design remain the same. And as stated before, the amount of network traffic depends on the amount of data that Game Developers add to their game as synchronized data.

3. Features Description

3.1 Features Overview

To meet the requirements given by the stakeholders, the project contains two main products. The first component is the multiplayer game framework. The framework is a library that provides the backbones of implementing an authoritative multiplayer server in a client-server architecture. It is built around Unity, and has the capability to support real-time 3D multiplayer. The framework comes with thorough documentation, with the intent to facilitate a user to implement online multiplayer for their games.

The second feature in our product suite is a sample online multiplayer game. This game is a “proof of concept” for the framework in that it utilizes our library to create a multiplayer game in Unity.

3.2 Old Features

These are features that were implemented by the time of the midterm report.

3.2.1 Basic Matchmaking

We had basic matchmaking features implemented by the midterm demo. A player could create, view, and join match lobbies. Once in a match lobby, a player could start the game for all players in the lobby. However, our matchmaking was somewhat limited. We did not yet have the master server described later. Only one game could be played at a time, and the server and clients had to be reset after each game.

3.2.2 Basic Gameplay and Synchronization

By the midterm report, multiple players could play in the same game. Player movement was synchronized between clients. However, players could not interact with each other nor could they fire bullets at each other. Additionally, we were using an inefficient network messaging system and had no client-side prediction, leading to poor performance and a lack of scalability. We later replaced this system with our more efficient Netcode system.

3.2.3 Basic UI

Some of the UI elements were implemented, including many of the menu screens. However, more UI functionality was added after the midterm, including support for Steam invites and overhead player displays. Additionally, the UI was later fine-tuned for increased aesthetics and functionality. The UI was also integrated with our Netcode communication system we developed later.

3.3 New Features

These are features that were implemented after the midterm report.

3.3.1 Steam and Inviting Friends

One important new feature is the ability to invite friends from Steam to join a match created by a specific user. The external asset Steamworks.NET is used in order to interface with Valve’s Steamworks, since Steamworks does not offer native support for the Unity Game Engine. This allows the game to access information about the player’s steam account, like their username and their friends list. Their username on Steam is automatically set to be their player name in matches they create or join, although they may change it if they so choose.

Additionally, when creating a match, players have the option to invite friends to join their match. The game retrieves the list of their Steam friends and selects those friends who are set as Online on Steam and are currently playing the same game. It then allows the user to select a friend from the list and invite them to the game. Their friend will receive a popup message in-game with the information about the invitation (namely, who it is coming from), and the ability to accept or reject it. If they opt to accept the invitation, they are automatically added to the match lobby and can wait like other players for the game to start.

3.3.2 Drop Match

Once in the game, users have the ability to opt to drop out of their current match. They can do so by pressing Esc and then selecting the option to drop match from the menu that appears. They will then be returned to the main menu of the game, and other players in their previous match will no longer be able to see their player icon or interact with it, as they have left the match.

3.3.3 Interactive Gameplay

Users now have the ability to interact with each other. They can collide with other players and fire bullets which can harm other players. Bullets are fired from the front right side of the player in the direction the player is looking. Additionally, bullets and player health are synchronized among clients. Player health is indicated in an overhead display that has a heart for each life the player has. A player starts with five lives. However, a player is not kicked out upon losing all their lives. Instead, if a player takes damage while at no lives, a player will regain five lives. When a player is hit by a bullet, they lose a life and become immune from harm for one second. After this grace period, a player can again lose lives from other bullets. Bullets that hit a player become inactive, regardless of whether or not the player was in a grace period or was damaged by the bullet. Inactive bullets are no longer visible and cannot harm players. Like all bullets, they are destroyed after three seconds of in-game time.

3.3.4 Interfacing between the Client, Server, and Master Server

With the added master server, there are three lines of communications between the separate parts of the framework. The client and server will communicate similar to before, by updating each other with snapshot information, and match meta information such as starting a match and dropping a match.

Now, when the client wants to create a lobby, it will contact the master server. The master server will find an open lobby server to host the lobby, and act as the middleman in connecting the client and server together. The master server also acts as a method to invite Steam friends, since it has information on all the players online, and handles handles sending and accepting invites between clients.

3.3.5 Reliability in Sent Packets

When dealing with important information such as creating a match and joining a lobby, we do not want to lose those packets due to the unreliable nature of UDP. However, TCP requires too much overhead in a system that requires low latency. The framework implements a form of communication with UDP that ensures packet reliability. By repeatedly sending packets that require reliability and processing ACK packets, all critical information in the framework is handled reliably.

3.3.6 Performance and Client Prediction

The performance of the server improved and became more scalable with more efficient serialization and deserialization of packets. In the midterm demo, the positions and rotations of the players were sent as strings. Now, structs are efficiently copied into byte arrays, e.g. a 22 byte struct is sent as a 22 byte packet.

In modern multiplayer first-person shooters, client prediction is non-optional, so we knew this feature would make its way into the netcode eventually. With an authoritative game server architecture, the clients send the player input to the server, which processes the input. But for player input that controls movement and animations, the round-trip time delay of packets introduces intolerable latency, e.g. there would be delay between the player pressing W and the character actually moving. Client prediction solves this problem: It recognizes that, in order for the game to feel responsive, the client shouldn't wait for the server to ack data like position and rotation. The client "predicts" that its movement will be valid, and later reconciles it with the server. Client prediction was implemented with snapshots, a circular buffer of player state indexed by the sequence number.

3.4 Types of Users

Since the project consists of two distinct parts, there are naturally two types of users that will interact with the project. The developer user will simply extend and integrate the framework to assist in the development of their own multiplayer game, utilizing the documentation provided by the framework. More thorough user interaction for the project exists in the demo game, where the user can play the game as a "player". The player can input controls through the mouse and keyboard in-game, with familiar first person shooter controls. Playing the game as a player requires an internet connection, since multiplayer game state has to be synchronized through an online server. Additionally, players must have a Steam account as our project was integrated with the Steam friends system.

3.5 Use Cases

This section covers the interactions a player can make once they start the game on their local computer. A key thing to note with these use cases is that all network features and communication with the server are heavily abstracted, in that multiplayer functionality appears to happen locally despite the complications behind the scenes.

Use Case 1: Create a match

Summary: A player (a client) is able to create a new instance of the game (a match) hosted on a server that other players can join and play together.
Preconditions: Player starts the game and has Steam running
Basic Flow:

- Player specifies the name of the match to create and asks the master server to create a match.
- The master server allocates the match to an open server.
- The master server keeps track of the player lobby for this match and of which server is running each match.
- Upon request, the master server can later send a list of open matches to another player.

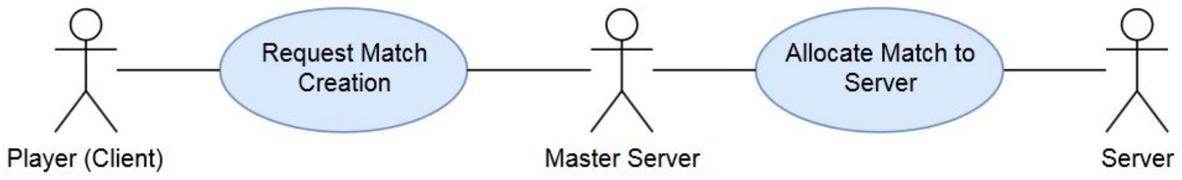


Figure 3.4.1: Use case 1 diagram for creating a match.

Use Case 2: Invite a friend to a match
Summary: A player who is in a match lobby can invite Steam friends to their match lobby
Preconditions: A player is in a match lobby
Basic Flow:

- Player can talk to Steam to get their list of Steam friends. Then, they can view their list of Steam friends in the match lobby.
- Player can invite a friend from their Steam friends list to their match lobby.
- The master server will receive the Steam invite and forward it to the Steam friend.
- The friend will be notified about the invite, choosing whether to accept or decline.
- If the friend accepts, they can get the match server IP from the master server and use this to join the match.

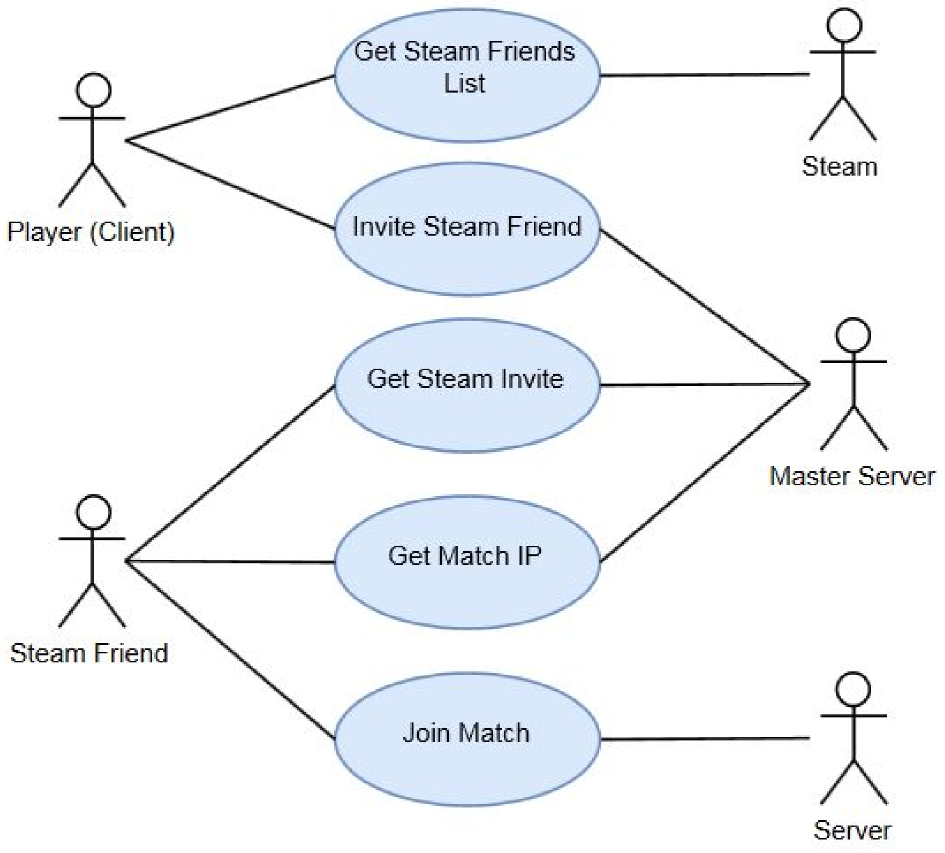


Figure 3.4.2: Use case 2 diagram for inviting a Steam friend to a match.

Use Case 3: Join a match

Summary: A player can join a match that another player has created, as long as the match has not started.

Preconditions: Player starts the game

Basic Flow:

- Player queries the master server to get a list of open matches.
- Player can choose a match to join and inform the master server of this request.
- Upon receiving a match request, the master server will send back the match IP address.
- Player can join the match by using the match IP.

Alternative Flow:

- If a player is invited by a friend on Steam, the player can join a match using the use case described earlier.

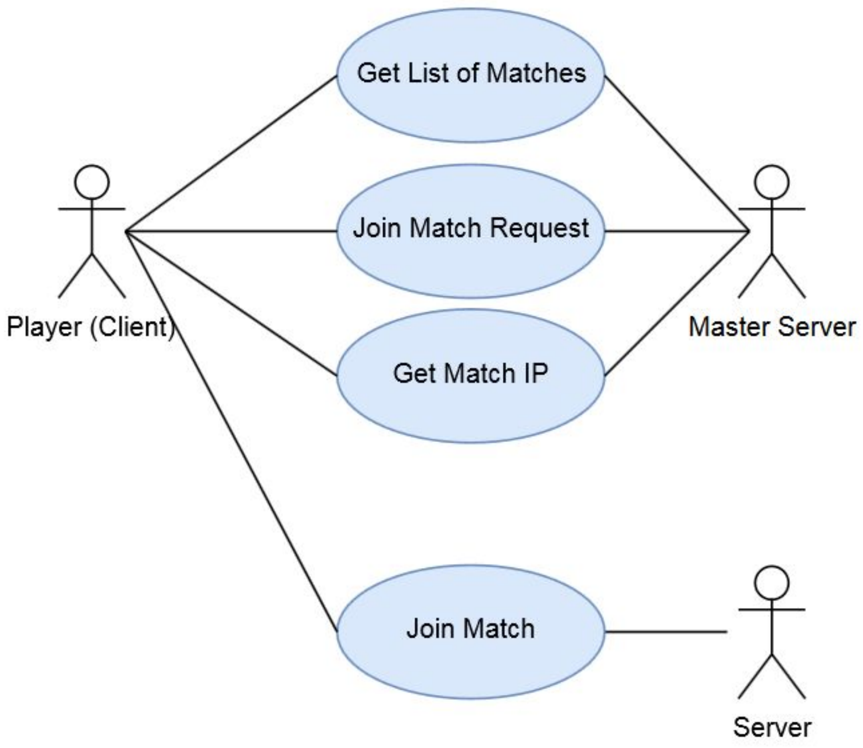


Figure 3.4.3: Use case 3 diagram for joining a match

Use Case 4: Playing a match
Summary: Players in a match can interact with other players over the network
Preconditions: Player is in a match
Basic Flow:

- Player in a match will automatically communicate with the server on updates about the state of the match.
- A user can provide keyboard and mouse input to the game to perform player actions.
- A player's actions will be reflected to the other players in a match, and vice versa.

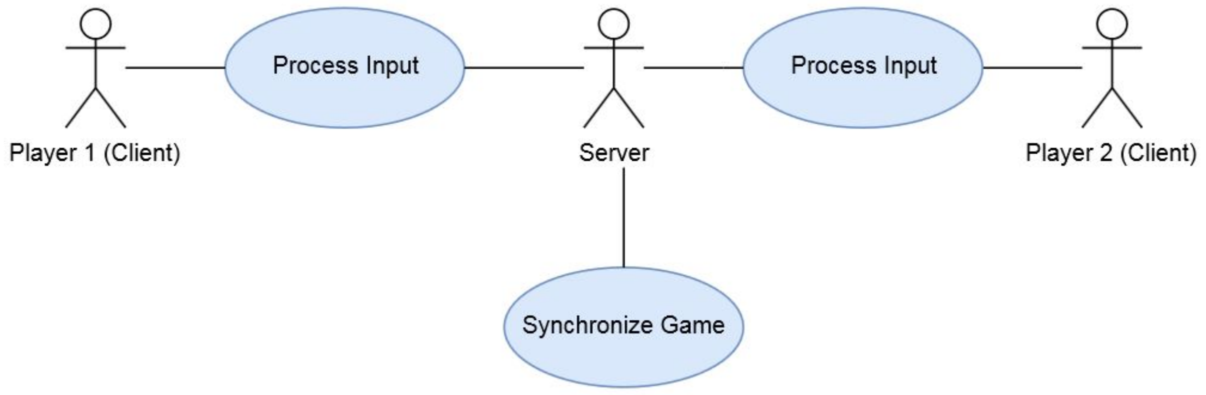


Figure 3.4.4: Use case 4 diagram for playing the game.

Use Case 5: Dropping a match
Summary: Players in a match can drop out of a match and return to the main menu
Preconditions: Player is in a match
Basic Flow:

- Player in a match can open a menu screen and click a button to drop out of the match
- Dropping out of a match will remove the player from the game. This will be visible to the server and all remaining players.
- A player who has dropped out of a match will be returned to the main menu screen.
- A player can choose to drop a match anytime.

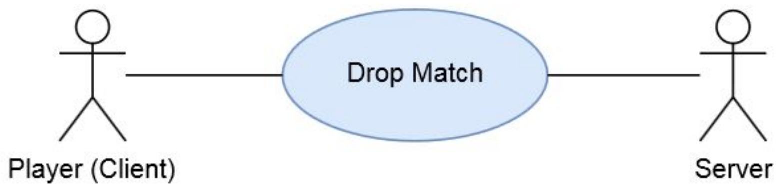


Figure 3.4.5: Use case 5 diagram for dropping a match.

4. Architecture Design

4.1 Overall Architecture

The overall architecture of the multiplayer game is shown in Figure 4.1. The line delineates between the game and the networking layer. In this way, the game can view the networking layer as a suite of services for sending and receiving messages across the network. This relationship is expressed in the UML diagram as composition; GameClient is composed of a Client object, and GameServer is composed of a Server object. Similarly, Client is composed of

GameClient and Server is composed of GameServer, so that the networking layer can alert the game to network events.

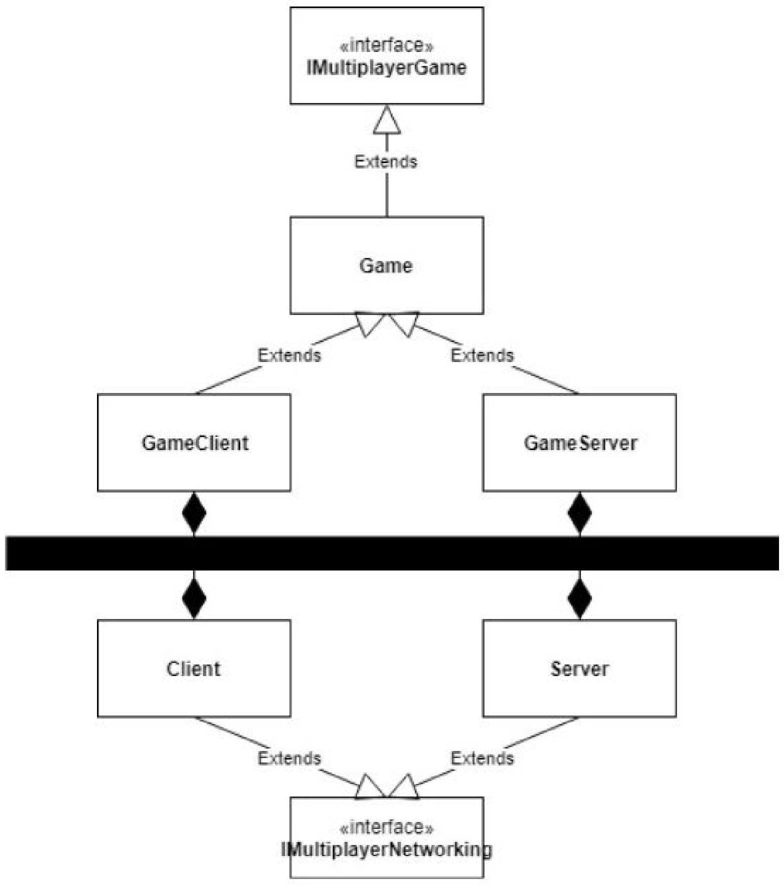


Figure 4.1.1 UML diagram of the overall server architecture

Client and Server need access to the same set of network functionality, such as initializing UDP and sending/receiving packets, so these functions were pushed into the base class IMultiplayerNetworking.

Game contains code that is common to the game logic on both the client side and the server side. Game is an abstract class itself, so it pushes the responsibility of implementing the IMultiplayerGame interface to GameClient and GameServer.

4.2 Game Architecture

The game architecture can be visualized with the UML diagram of Figure 4.2.1. Both GameClient and GameServer manage a set of NetworkedPlayer; additionally GameClient has a MainPlayer. This is consistent with the view that to the server every player is the same, while a main player is controlled by the client.

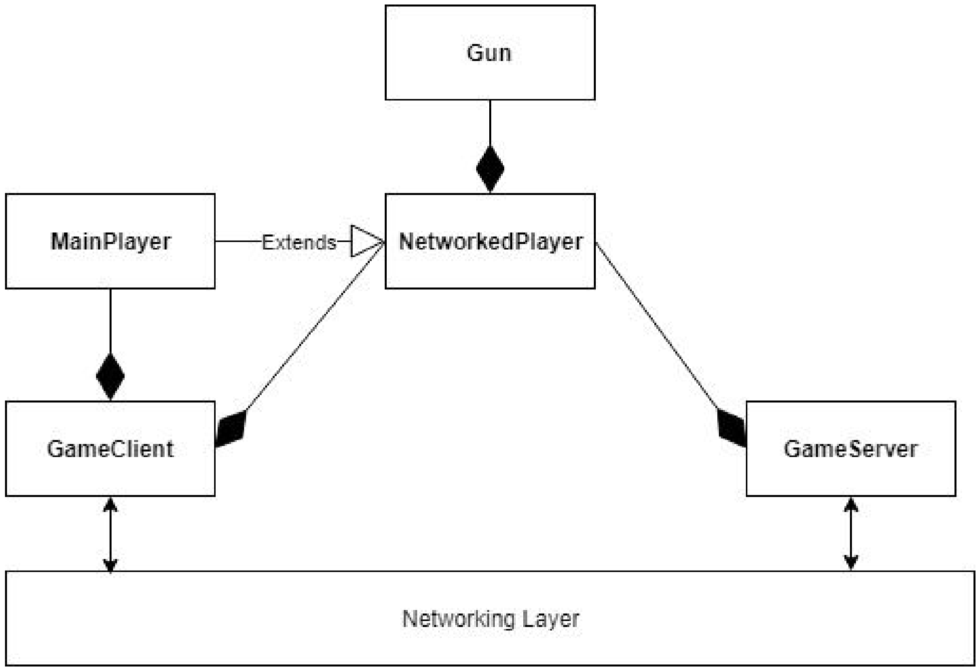


Figure 4.2.1 UML diagram of the overall game architecture

From this diagram, it is also clear that the client and server communicate through the networking layer. The client sends snapshots and player input; the server processes this input, simulates the effect of the input, and sends the state of its “simulation” to all the clients. Under this game architecture, the server is authoritative of the game state.

MainPlayer extends NetworkedPlayer so that there is a separation between player input and the state of the player. This is a common design pattern in multiplayer games because player input isn’t always applied locally; it is sent as a packet to the server which the server then processes. For example, in the proof-of-concept game, when a player presses left mouse, a packet is sent essentially saying “Player X fired weapon”. The server then calls TakeCommands on the NetworkedPlayer with server ID X, the argument to this function being the bitmask of the buttons pressed. The NetworkedPlayer then processes the input as if the buttons had been pressed on the server. The result of the input isn’t actually seen by the client until the server sends back the updated game state, but using UDP for the transport layer protocol does a decent job of hiding this latency.

4.3 Matchmaking Architecture

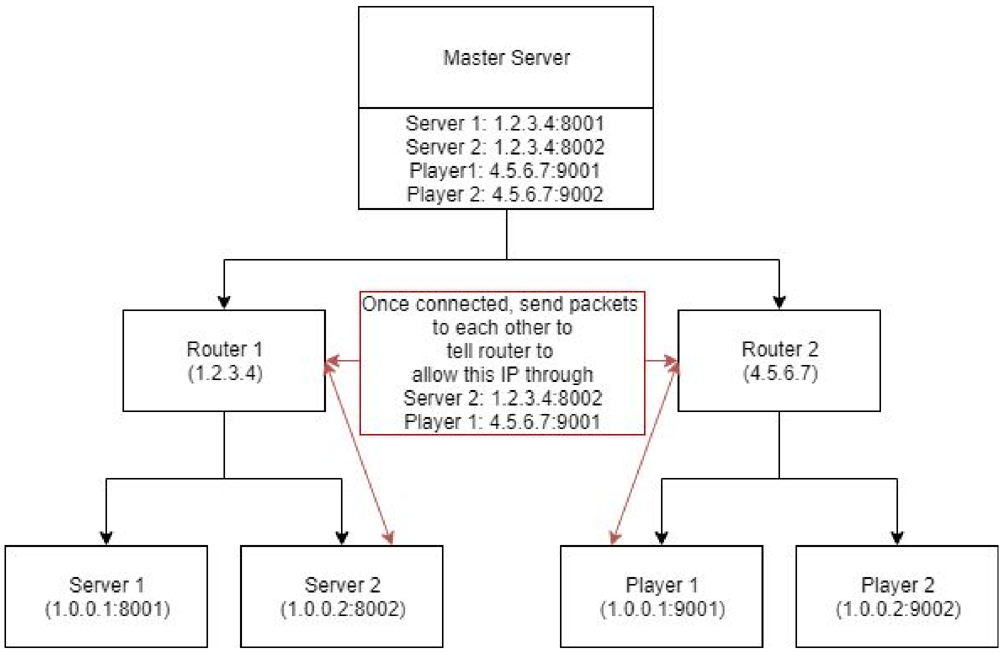


Figure 4.3.1 Diagram of the multiplayer network and matchmaking process, including NAT punchthrough.

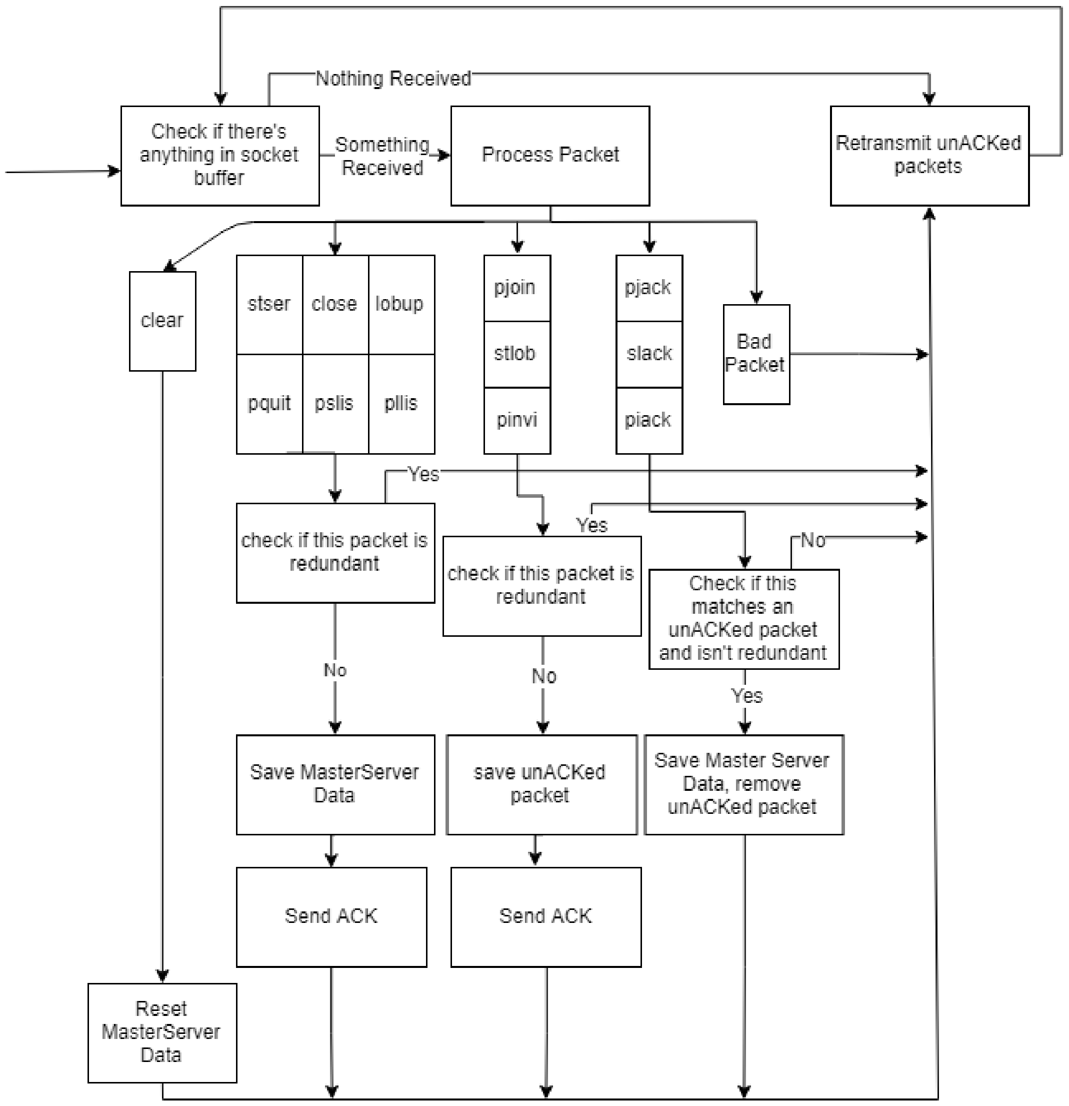
Previously, a game server and its player clients connected to each other directly to create a match and play. In order to have a more scaling and dynamic lobby creating system, we utilized a master server. The master server is the first point of connection for all servers and clients, and is used to connect these components together, even if they are hidden behind the private networks of their routers. The master server runs on a public static IP, and by connecting to it the master server can find out the public IP address that is randomly assigned to the client and server by their router. The master server shares this information with the client and server so that they can connect to each other. Every match handling function, from creating a lobby up to a player joining a lobby, is handled through the master server. Once a player/client is connected to the lobby/server, the communication between them will be not visible to the master server, except the server will notify the master whenever a player leaves its lobby. In addition, when a player desires to invite a Steam friend into the lobby it exists in, it will communicate with the master server to find where the friend is, and send a request with the lobby IP and port.

However, even with the middleman of the master server and knowing the public IP/port of a system a sender wants to connect to , direct connection between two private networks isn't possible. This is because the router of a client or server that exists in a private network will drop any packets from a public IP it hasn't established a connection with. To avoid this, the client and server that wishes to connect must first find out each others public IP using the master server, and send a initialization packet to each other. With the router now knowing that its private network has tried to communicate with the public address, it will accept packets from the public address. This process is called NAT punchthrough, and is key to making the matchmaking structure work.

As noted previously, since these network packets extend beyond a single computer, they may be dropped due to the nature of UDP. The matchmaking architecture uses a simple reliable UDP protocol to make sure that the important packets are all transmitted reliably.

In this way, a client can create a lobby for a match without knowing anything about where the server will be, and can create as many lobbies as servers that are connected to the master server. By having this central master server, all clients and servers only need one point of connection to communicate with everyone else.

The master server itself reliably transmits packets through a custom protocol where duplicate packets are not reprocessed and updates to the master server only process if they are expected.



Nothing Received
Check if there's anything in socket buffer
Something Received
Process Packet
Retransmit unACKed packets
clear
stser
close
lobup
pquit
pslis
pllis
pjoin
stlob
pinvi
pjack
slack
piack
Bad Packet
Yes
check if this packet is redundant
Yes
check if this packet is redundant
No
Check if this matches an unACKed packet and isn't redundant
No
No
Yes
Save MasterServer Data
save unACKed packet
Save Master Server Data, remove unACKed packet
Send ACK
Send ACK
Reset MasterServer Data

Figure 4.3.2 State diagram of the master server and it's network protocol.

5. UI Design and Snapshots

Our game framework is designed so that the UI system is modularized and can be easily swapped out or modified. We decided to separate the UI from the multiplayer framework and gameplay since the UI is likely to be tailored for each specific game. This is due to its dependence on the environment in which it is used and the game presentation and control desired. As such, our UI never directly controls or keeps track of game state. Instead, it merely informs game managers of user input and keeps track of the UI state. Communication and dependencies between the game managers and the UI are also minimized to increased modularity.

Our UI is game-specific and not designed for heavy extensibility since we expect that the UI in other games will be customized to those games. Additionally, Unity already provides a good UI framework; in fact, our UI utilizes much of Unity's built-in UI elements, including canvases, buttons, and input fields. However, we did implement some UI features applicable to multiplayer games such as scrollable open match lists, popup invite menus, and overhead player displays.

5.1 Menu UI

The first type of UI we have controls the menu screens and guides the player through matchmaking and match creation. Each major screen of the UI is controlled by a separate UI class. The MainMenuUI class coordinates the different screens and handles user input dealing with matchmaking.

5.1.1 Main Menu

When the game starts, it tries to communicate with Steam. If this Steam initialization is successful, then the game will show the main menu screen. From this screen, a player can choose to create or join a match.

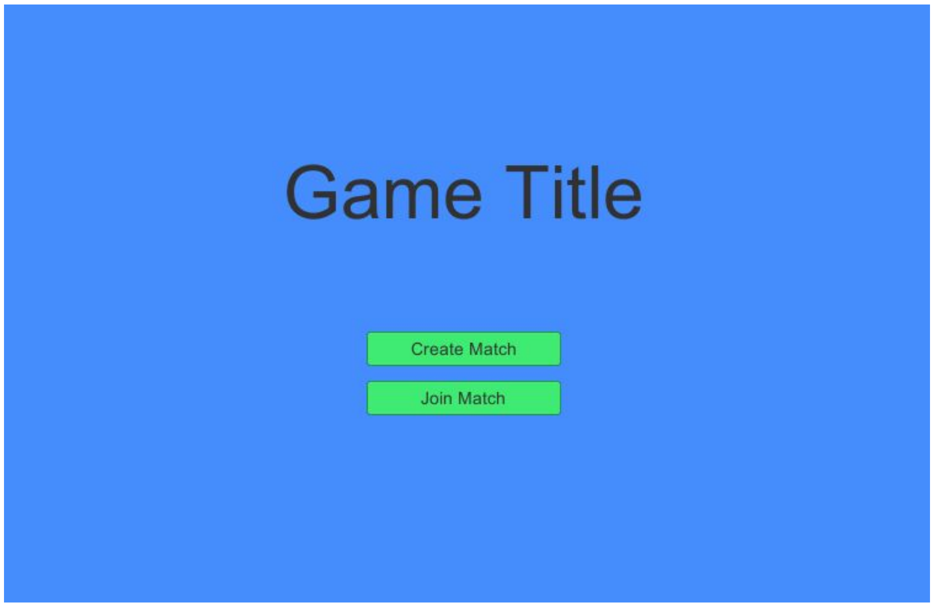


Figure 5.1.1.1: Main menu screen

If the Steam initialization fails, the game will display an error screen and prevent the user from performing other menu actions. To play the game, the user has to make sure that Steam is running and restart the application.

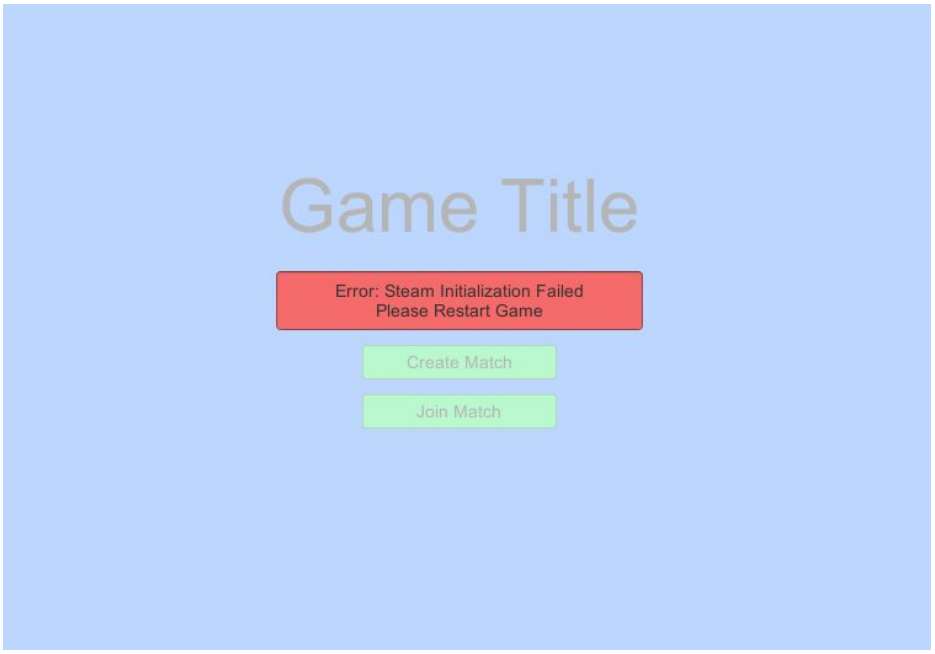


Figure 5.1.1.2: Steam initialization error screen

5.1.2 Create Match Menu

If the player chooses to create a match, they will go to the create match menu. In this menu, the player can input a match name and create a match. Once a match is created, the server will automatically add the match to the list of matches available for other players to join. The player name will be automatically set to their Steam name. Since the game will not allow users to progress to this menu without initializing Steam properly, this name should be initialized correctly.

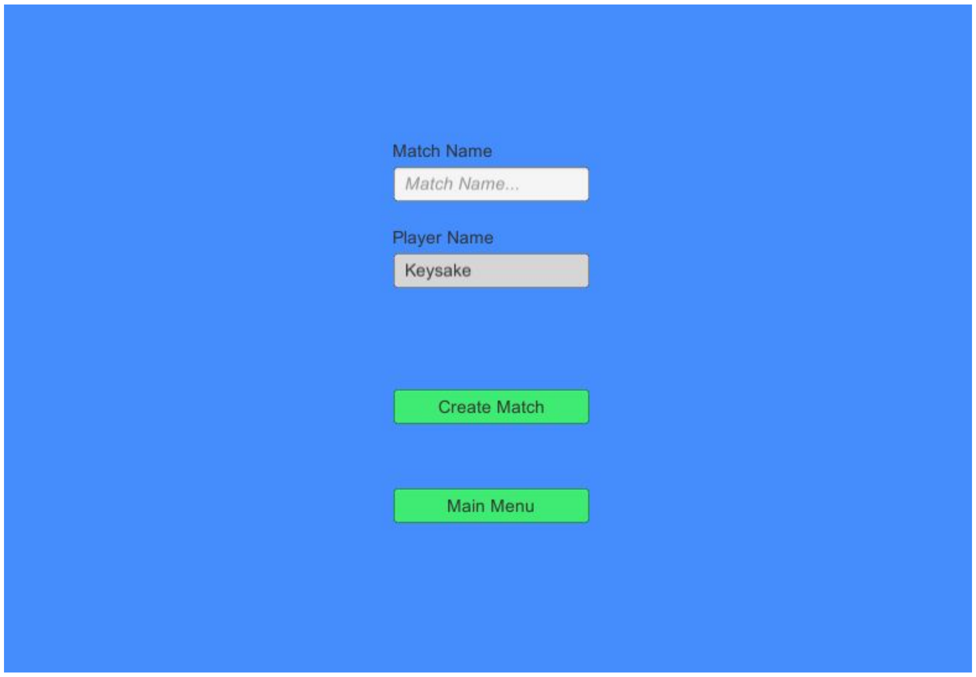


Figure 5.1.2.1: Create match menu

5.1.3 Join Match Menu

If the player chooses to join a match, they will go to the join match menu. The player can choose the match they would like to join from a scrollable menu listing all open matches currently hosted on the server. This menu can be updated by pressing the refresh button to request an update from the server. The player name will default to their Steam name, but a user can alternatively choose a different player name by typing it into the relevant input field. After selecting a match, the player can join the match which will send them to the start match menu.

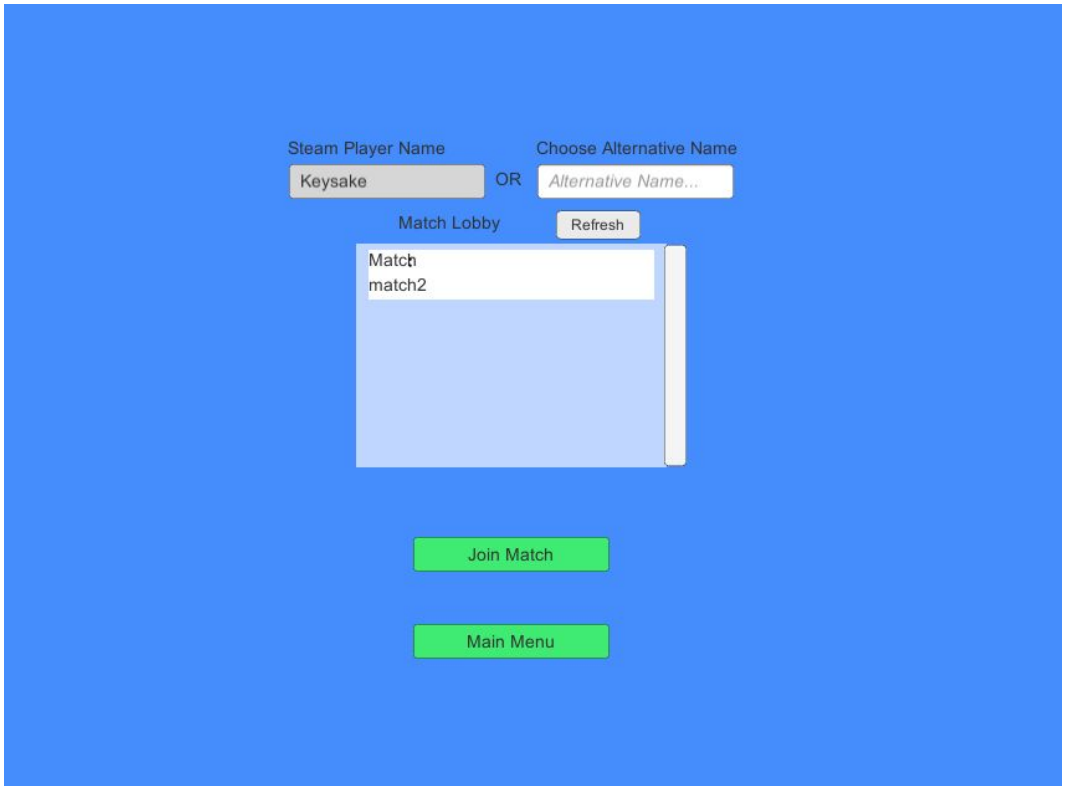


Figure 5.1.3.1: Join match menu

5.1.4 Start Match Menu

Once a players has created or joined a match, they will be placed in a match lobby and sent to the start match menu. From this menu, they can view a list of players in the same match lobby. They can also invite their Steam friends by clicking the "Invite Steam Friends" button. This will open up a window which displays a list of all Steam friends currently running the game. Clicking the invite button next to a friend name will send an invite to that friend. Pressing the start game button will cause all players currently in the match lobby to be loaded into the game. Once the game has started, no other players may join the game.

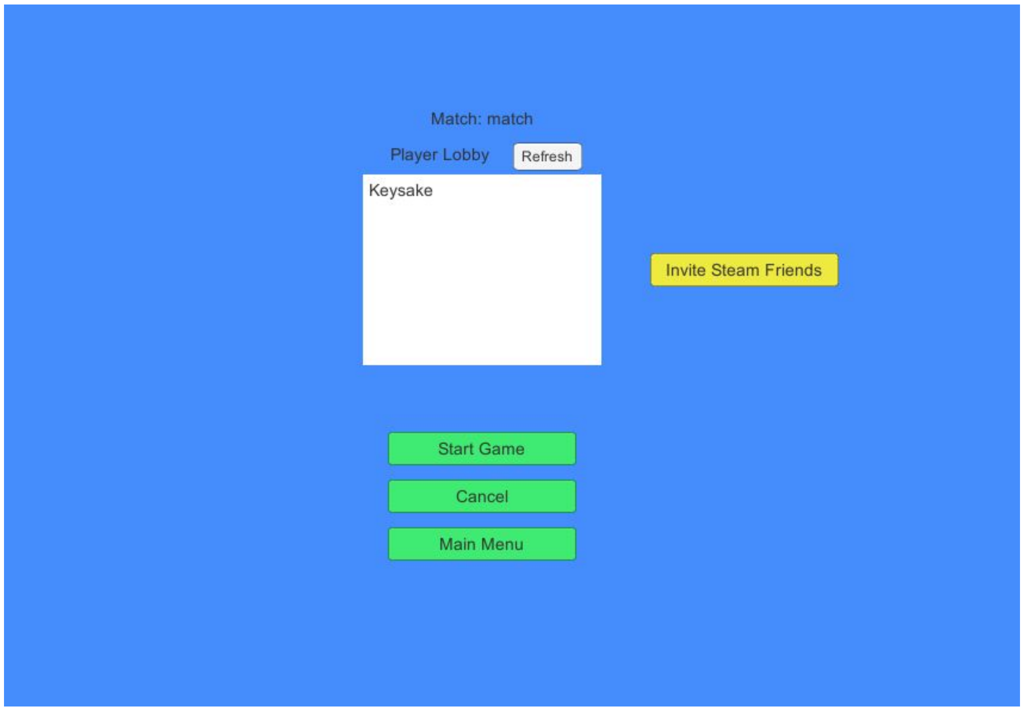


Figure 5.1.4.1: Start match menu with invite menu closed

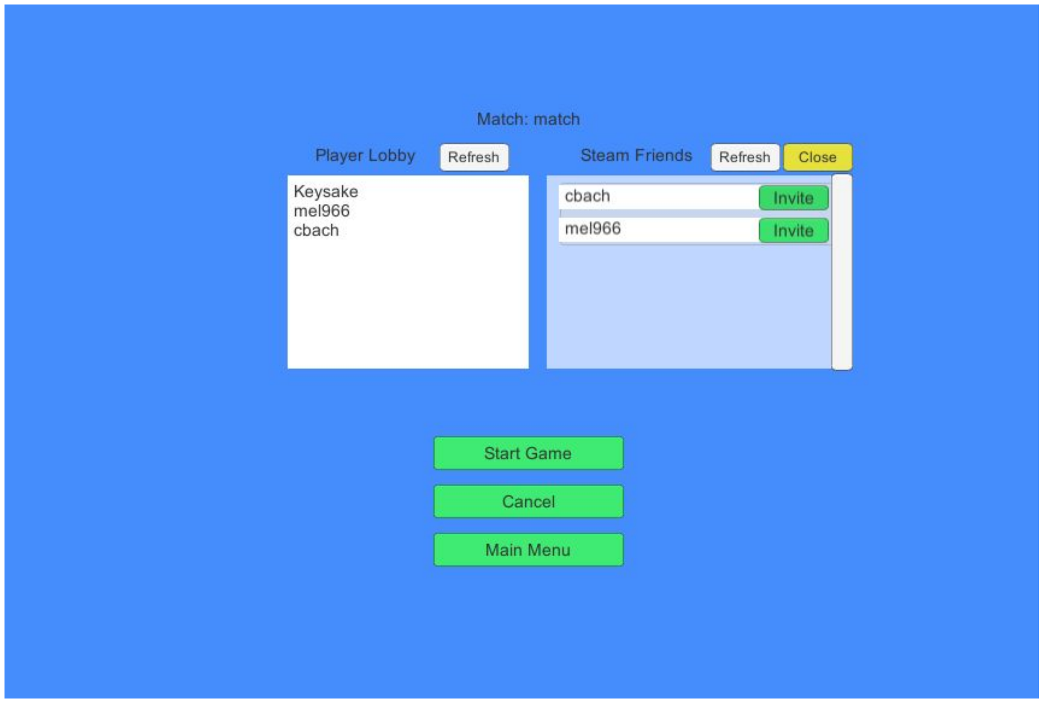


Figure 5.1.4.2: Start match menu with invite menu open

5.1.5 Steam Invite

When a player sends an invite to a Steam friend from the start match menu, the target friend will receive a popup informing them of the invite. They can then choose to either accept or decline this invite. If they accept, they will be moved into the match lobby of the friend who invited them. If they decline, they will remain in their current menu.

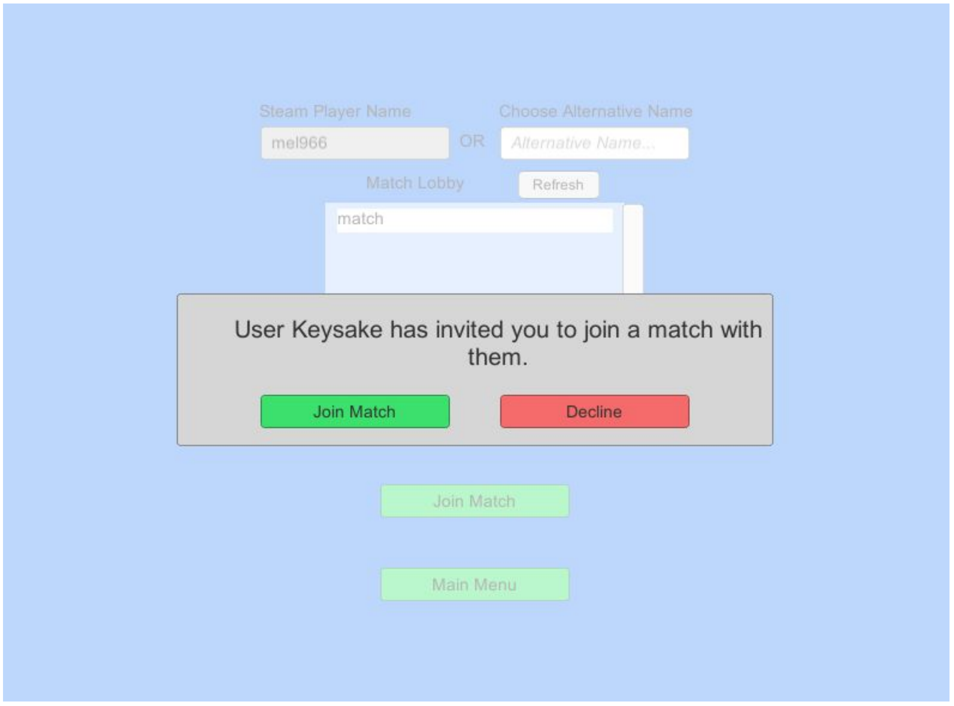


Figure 5.1.5.1: Steam invite popup

5.1.6 State Machine Diagram

The following state diagram summarizes the UI menu states and matchmaking actions a player can take.

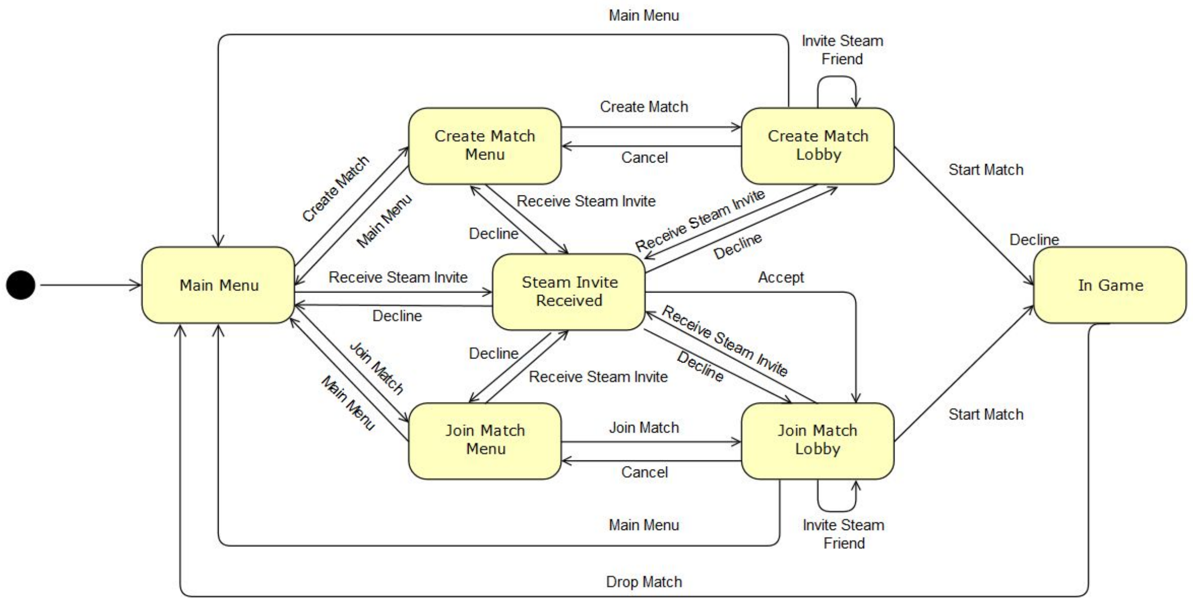


Figure 5.1.6.1: State machine diagram for matchmaking

A client user will start off at the main menu screen. From there, they can move to the create match menu or the join match menu. From the create match menu, a player can enter match information and create a match. This will move them to a create match lobby where they can view all the players in the match lobby. From the join match menu, a player can view open matches and enter player information. If they join a match, the player will be moved to the join match lobby where they can view all players in the match lobby. Any player in the match lobby, whether they created or joined the match can invite Steam friends or start the match. At any point a player not already in a game can receive an invite from a Steam friend. If they accept the invite, then they will go to the match lobby of their friend. If they decline, they will remain in their current menu. Choosing to start a match from a match lobby will move all players in the match to the in-game state where the game is played. A player can return to the main menu from the game by choosing to drop out of the match.

5.2 Game UI

The second type of UI we have is used once the player has started a game. This in-game UI includes an overhead player display and a menu screen.

5.2.1 Overhead Player Display

Shown above each player are zero to five hearts, representing the number of lives the player has. As each player loses or gains lives, the number of hearts above that player decreases or increases accordingly.

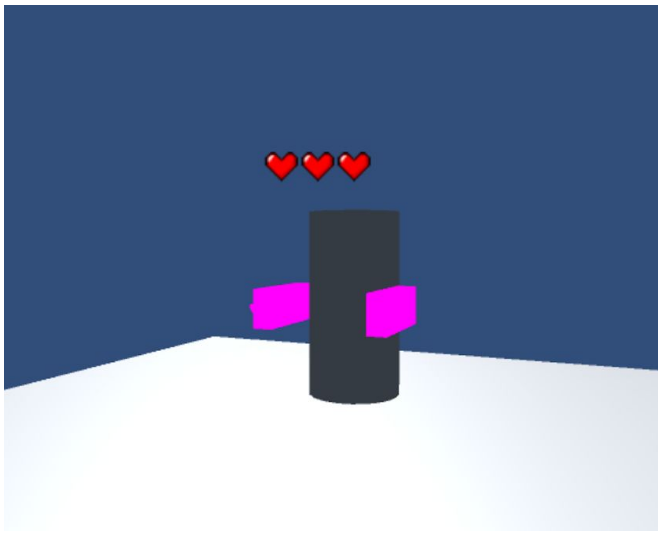


Figure 5.2.1.1: Overhead display of player life

5.2.2 Menu Screen

When the user presses Esc, a menu screen is shown. Pressing Esc again will close the screen. Alternatively, the player can press the resume game button to close the screen. Opening the screen does not pause the game nor does it stop player actions. However, it may obscure player vision. Pressing the drop match button will drop the player from the game and return them to the main menu.

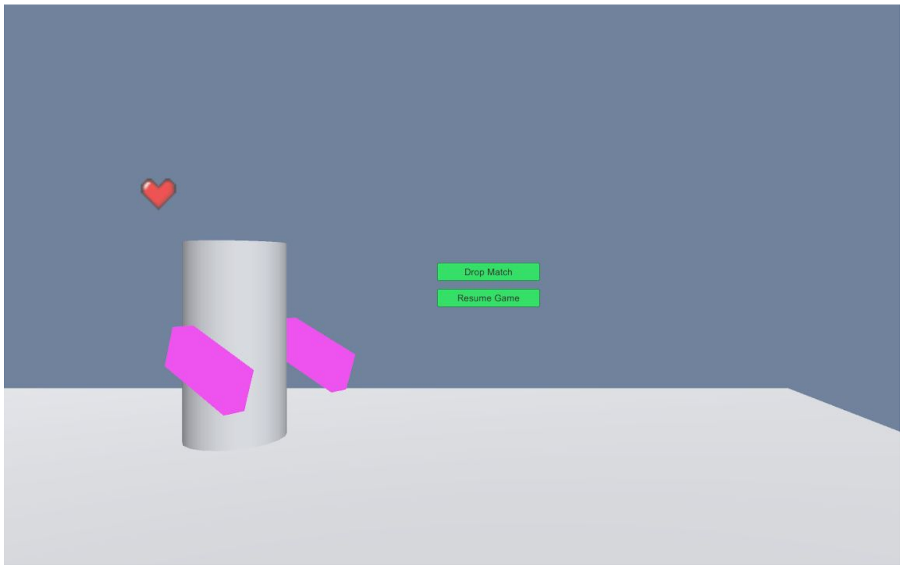
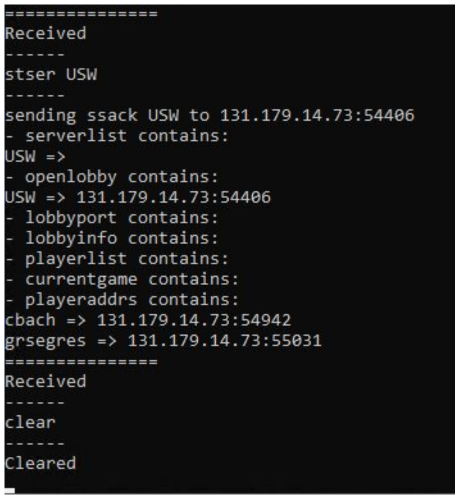


Figure 5.2.2.1: In-game menu screen

5.3 Master Server UI

If the print flag is set when compiling the Master Server, it will output its remembered player and server data to screen every time it receives a packet. In addition, it will print out the contents of packets that Master Server is still waiting to receive an ACK for, regardless of the print flag. This is so an administrator can tell if there are any network issues. Below is an illustration of Master Server's screen output.



```
===============
Received
------
stser USW
------
sending ssack USW to 131.179.14.73:54406
- serverlist contains:
USW =>
- openlobby contains:
USW => 131.179.14.73:54406
- lobbyport contains:
- lobbyinfo contains:
- playerlist contains:
- currentgame contains:
- playeraddrs contains:
cbach => 131.179.14.73:54942
grsegres => 131.179.14.73:55031
===============
Received
------
clear
------
Cleared
```

Figure 5.3.1: Master server screen. A server has been registered at 131.179.14.73:54406, and is waiting for a lobby to be started on it. Users "cbach" and "grsegres" are not in any lobby, and are mapped to 131.179.14.73:54942 and 131.179.14.73:55031 on the router they are connected to (131.179.14.73).

6. Testing

6.1 UI Testing

6.1.1 Create Match

To test this functionality, a client and master server/server instance must be run. From the client game, the main menu is loaded and this will be clear if it works since the main menu will display the Game Title and two buttons labeled “Create Match” and “Join Match.” Once this has correctly loaded, the “Create Match” button should be pressed.

This button will be known to work correctly if it loads a new menu where the user may type in the name of a match and also displays their Steam username as player name. If the user does not enter a match name or enters an incorrect match name (like one that uses symbols or other non-alphanumeric characters), the game should display an error and prevent the match from being created. To verify this, the “Create Match” button should be pressed while the match name is left empty, and while it is filled with some non-alphanumeric characters. An error message should appear right above the “Create Match” button. Additionally from this menu, the “Main Menu” button can be tested by pressing it and verifying that this returns the user to the original menu of the game.

Next the “Create Match” button should be tested with a valid match name. An alphanumeric match name should be entered, and then the button should be pressed. This should load a new menu that shows a player lobby with only the player who created the match. The player names in the lobby are retrieved from the server, so if the connection with the server is working then the user’s name will be displayed in the lobby. This also tests that the Netcode used by the client and server to communicate is able to correctly send and receive packets: the client here is sending a packet to the server to inform it of the new lobby and the player name who created it, and the server is replying with a packet to tell the client which players are in the lobby. If the player name is displayed, this process occurred correctly.

6.1.2 Join Match

To test this functionality, two clients must be running with a master server/server. One client must create a match and then wait in the player lobby without pressing Start Match. The second client must click the “Join Match” button from the main menu. This will load a second menu, where the user will be able to see a list of all matches that are available to join. They will be able to select a match from the list and click “Join Match”. They will also at this point be able to enter a player name other than their Steam username. This is explicitly for testing purposes, as otherwise there would be no way to test any other parts of the game or Netcode without having multiple Steam accounts and multiple computers in use for testing.

Once they have selected a match and clicked “Join Match,” the player lobby menu should be loaded. In the player lobby, the player will see the names of all the players currently in the lobby, including at the very least the player who created the match and their own player name. Again, this information is retrieved from the server, so the player’s own name being displayed in the lobby means that the client is correctly communicating with the server. Additionally, on the second client, the player lobby should be automatically updated to include the name of the player who just joined.

6.1.3 Invite Friends

After opting to create a match, users can opt to invite their friends to their match’s player lobby. To test this, two clients must be running with a master server/server and both clients must be logged in to their separate Steam accounts. Both clients must also be friends on Steam, and additionally have their state on Steam set to online. Then one client can Create a Match and then click “Invite Friends.” From there, they should see the Steam name of the second client appear on the list. This indicates that Steamworks.NET has been correctly set up, and that the steam_appid file is in the correct place and set to the correct value.

They can then click the name of their friend and then click invite. Before this, on the other client, they must either click join a game, and then refresh, or they must opt to create their own match and enter their name. This is so that the master server and server know their identity, since for testing purposes players can change their name from their Steam name. Once the invite is received from the master server, a popup window will appear on the second client which tells them that their friend has invited them to join their match and gives them the chance to accept or reject the invitation.

If they reject the invite, the menu must close. If they accept, then the player will be added to the match’s player lobby and they are loaded into the player lobby in the same way as if they had simply selected to join from the main menu. Again, this popup window and being able to join the player lobby indicates that the client and server are correctly communicating.

6.1.4 Start Match

From within the player lobby, the Start Match button can be pressed. Once it is pressed, all of the players in the lobby should be loaded into the game scene and spawned as players. This indicates that the clients and server are correctly communicating, as the other players are also spawned in to the game when one presses start match.

6.1.5 Drop Match

Once the player is the game, pressing Esc should open a menu that allows them to choose between dropping the match and resuming the match. If they press Esc a second time or choose resume, then the menu should close. If they select drop match, then they should be removed from the game and returned to the main menu. Additionally, the other clients should no longer be able to see or interact with their player object.

Having this occur additionally indicates that the server and clients are communicating correctly, as the removal of one player is propagated throughout all the clients.

6.2 Game Testing

6.2.1 Player Movement

In order to properly test player movement, a master server/server must be run with at least one client. Once within the game, the user should use the arrow keys to move around and observe that this movement is represented locally. With multiple clients, the movement of one client should also be reflected in the game scenes of the other players. This indicates that the snapshots used by Netcode are working and successfully allowing the server and clients to communicate game states with one another.

Additionally, the player should use their mouse to look around and rotate their avatar. This change should also be reflected in their view of the game field and the orientation of their avatar in the game scenes of the other players.

6.2.2 Bullets and Collisions

To test bullets and the collisions between both players and players as well as players and bullets, at least two clients must be run with a master server/server. The player should first test that clicking correctly generates a bullet object that travels in the direction their gun was pointed (the gun can be aimed by moving the mouse). Then they should verify that this bullet object also appears and travels in other clients' versions of the game. The bullets should disappear after a certain length of time, so this should also be tested by firing a bullet and observing that it eventually disappears.

Collisions between bullets and players should be testing by firing at another player. The bullet should disappear after contacting the other player and additionally the player should not be affected by the bullet's momentum. This is to say that the bullets should not affect the players' physics. Additionally, when a player is shot, the number of hearts above their head should decrease by one.

Finally, the collisions between players must be tested. To do this, one player should move around the game field and attempt to run into other players. The game should prevent the player from traveling through the other player. It may be slightly glitchy due to the perspective imposed by the first person camera, but should prevent the players from moving through each other.

6.3 Master Server Testing

Handling the connection between a public IP address such as the master server and a private network that the client and server existed on proved to be quite a challenge. First, packets can be easily dropped while using UDP, and so the important packets were sent using a form of reliable UDP, discussed previously.

For the master server to know the presence of all of its servers and clients, the server and clients send a reliable packet notifying they are online. To test match connection features, the master server outputs all of the servers it has currently connected but not assigned a lobby, all lobbies that are connected and assigned a lobby, and what lobby every client is connected to. It holds the public IP/port of every entity connected as well.

When a client creates a lobby, the master server should output a message that says it assigned a lobby to one of its connected open servers. When a client joins a lobby, it will assign a player to the lobby and displays the list of players that are in a lobby in the terminal. When a player leaves a lobby, they are unassigned from that lobby, which is displayed in the terminal. If an invalid packet is sent, the master server will notify the packet in the terminal.

All of the important match connection cases above are sent reliably. The master server will output in the terminal every time it has received and still needs an ACK for some reliable packet it sent. The client and server will also output to their logs any time a packet is added to the set of reliable packets, and must display that the packet is removed from the set of reliable packets once it receives the ACK.

The server will initially start with a black screen. When a server is started, it will try to connect to the master server. When it does, it will change scenes with a camera. To test that a server has successfully connected with the master server, it must be in this scene. When a match is started by one of the clients in its lobby, the server must change scenes again to a view of the match running. Here, it will see all of the synchronized movements between its players.

6.3.1 Packet Loss Testing

To test the master server in a more realistic environment when running the entire multiplayer framework on one local system, the 'clumsy' utility can be used (https://jagt.github.io/clumsy/). Enabling packet loss, packet delay, out of order packet delivery, etc., allows the user to test the reliable packet system of the,master server. Using a simple UDP echo client that retransmits until it received something back, we simulate creating a lobby, joining a game, and sending friend invites and ensured that repeat packets were not reprocessed and that the master server only saved player and lobby information at the correct points in our reliable protocol. For instance, the echo client sends "stlob UCLA:lobby1" to the master server, the master server receives the packet and forwards it to the correct server, and packet is lost sending to the server. Resending the "stob" packet again on the client shows that the master server does not reprocess this packet or add it to its list of un-acknowledged packets, and the master server instead keeps retransmitting the first packet to the server since it has received no response. Upon receiving the response from the server, it finally sends an ACK back to the client. If this packet is lost, the client will try to start the lobby again, but master server will know that they are supposed to already be in the lobby and simply send an ACK back to the client as if lobby creation was successful. The master server was also tested interacting with Unity clients and servers from different networks, including UCLA Wi-fi, private Wi-fi, and Ethernet. The protocol was never broken due to any network latency issues.

7. Future Plans

Given more time, implementing a reliable networking layer would be at the top of the todo list. Some messages need to be sent reliably, such as “Start Game” and “Disconnect”, otherwise unexpected behavior arises. This is especially crucial to the communication between the client and the master server, where using TCP makes sense. If a developer were to extend the netcode to have a reliable networking layer, the developer is encouraged to look into using the C# socket library for this (https://msdn.microsoft.com/en-us/library/system.net.sockets.socket(v=vs.110).aspx). As for integrating the reliable networking layer, the developer should put this functionality in the MultiplayerNetworking class (where UDP is encapsulated) and expose a method for sending packets using TCP.

In addition, the Master Server component of the system can be rewritten or replaced altogether by another similar implementation. There exist a few popular options for this that either require payment or licensing. First, there is RakNet (http://www.jenkinssoftware.com/) which is free for development of free games on PC only, but requires licensing and payment for other products and platforms. The RakNet library contains functions for MasterServer hosting, and can thus replace a lot of the current Master Server functionality. With this, though, you would still likely need to host a Master Server on a static IP address. Another option where you don’t need to interact with the Master Server code at all and can host on a cloud machine would be using the Steamworks API directly (https://partner.steamgames.com/doc/features/multiplayer/game_servers). This requires approval directly from Valve Software, but allows you to host your server processes on cloud machines and use the Steam servers as the Master Server for NAT punchthrough and translation, and more fully integrates Steam with the game itself.

8. Individual Contributions

8.1 Alexis Korb

- Designed and created all UI elements
- Implemented gameplay features, such as player health and bullets
- Helped design game architecture

- Created gameplay and network code for midterm demo

8.2 Melodie Butz

- Implemented Invite Friends using Steamworks.NET and Netcode (functions on client and servers to handle and create invitations to friends)
- Implemented Drop Match using Netcode (functions on client and servers to handle and drop players from matches)
- Helped integrate UI with client-server communication system (Netcode)
- Helped design game architecture

8.3 Chris Bachelor

- Research and prototyping of different game architectures
- Ported the midterm demo, which used a very basic and inefficient network interface, to the current snapshot system.
- Implemented all of the functions that handles match connection (creating match, joining, updating, etc.) between the client, server, and master server
- Integrated the match server and client model that existed in the midterm demo with the masterserver to allow matchmaking.

8.4 Joseph Garvey

- Implemented the Netcode.
- Created the Snapshot API.
- Created the API for extending the game functionality.
- Implemented the scalable authoritative game server.
- Implemented client prediction.

8.5 Wyatt Watkins

- Researched different multiplayer game server architectures
- Helped design game architecture
- Designed master server/matchmaking architecture
- Implemented master server
- Tested master server interactions with test programs, real servers, and real clients
- Integrated master server with existing architecture to provide a true multiplayer network

9. Project Github

Our project can be found and cloned here:

https://github.com/alexiskorb/CS-130-Project