Documentation

Multiplayer First Person Shooter Game Framework

Chris Bachelor
Melodie Butz
Joseph Garvey
Alexis Korb
Wyatt Watkins

TABLE OF CONTENTS

1. User

1.1 Introduction

This section is intended for users who want to play our game. It describes how to start, play, and quit our multiplayer game.

1.2 Starting the game

To build the game, enter the Unity build settings, and create two executables. One executable will be the server, which will require adding the scenes “ServerStartScene”, “ServerStandby”, and “ServerMainScene”, all under “Assets/Scenes/Server”. The other executable, client, will require the scenes “MainMenu”, “MainScene”, and “StartScene”, all under “Assets/Scenes/Client”.

To run the game, the game executable (.exe file) must be run from within a folder that additionally contains the folder client_Data and the file called UnityPlayer.dll. To be able to play the game, users must additionally have an account on Steam and be currently logged in to that account on their computer.

To play the game the setup requires a MasterServer instance running with a static public IP and port, or behind a router with the appropriate port forwarded. Once the IP/Port are determined, you must change the MasterServerIP and MasterServerPort fields in “Client” and “Server” objects in “StartScene” and “ServerStartScene”, respectively. With the MasterServer running, you must run the server executable, so that a match lobby can be hosted. The server will initially be a black screen however will change scenes once it establishes a connection with the server. Now, we can run as the client executable and create a lobby.

Once the executable is launched, the game will load to the main menu. From there, users can choose to either create their own match or join an already created match. If they opt to create a match, they will be allowed to select a match name and invite some of their friends from Steam to join their lobby. In order to be invited to a friend’s lobby, a user must be logged in to their Steam account and marked as online on Steam, and also have the game open on their own computer. If they opt to join a match, they will be presented with a list of available matches, and may select any to join.

Once the user is in a lobby, they can start the game by pressing start game, or waiting for other players to join and press start game.

1.3 Playing the game

Once in game, the player may move around by using their arrow keys. By moving their mouse, players can look around and move the camera as well as rotate their avatar. They can thus aim their gun at other players, and can shoot by clicking their mouse.

Each player has “lives” above their avatar that are decremented when they are struck with a bullet by another player.

1.4 Quitting the game

While in game, the player can press Esc on their keyboard in order to drop out of their current match. This will take them back to the main menu of the game, where they can start a new match or join a different match. They can also close the game executable to quit the game.

1.5 Summary of controls

Arrow keys (left, right, up, down) OR W, A, S, D (up, left, down, right)	Control player’s movement around the game
Mouse right click	Shoot gun, bullets
Esc	Pull up menu to quit game and return to main menu

2. Developer

2.1 Introduction

This section is intended for developers of multiplayer games. It describes our multiplayer framework and details how to extend it to create a new multiplayer game. It also provides information about other game specific elements we implemented.

2.2 Code Overview

This section gives an overview of each of the source code files and classes in our system.

2.2.1 Multiplayer Game Framework

Netcode.cs
Contains definitions for packets, as well as two utilities used by the netcode: `PeriodicFunction` and `Serializer`.

All packets should inherit from the `Packet` class. If a new packet is added, the type of packet should go in the `PacketType` enum, e.g. `START_GAME`, so that the packet can be serialized/deserialized. Every packet must have a default constructor and have the `[StructLayout(LayoutKind.Sequential, Pack = 1)]` class attribute.

Snapshots (i.e. game state) should implement the `ISnapshot` interface for the client and server to be able to automatically handle client prediction. In the `ISnapshot` interface, the abstract function `Equals` performs an equality test between the same type of packets. (C# is a managed language, so it can't compare classes by value unless this method is overridden, necessary because the netcode sometimes compares by value.) The abstract function `FromObject` simply initializes the snapshot from a Unity `GameObject`. The abstract function `Apply` simply applies the snapshot packet to the Unity `GameObject`, useful for the game to easily update objects from packets.

The `Serializer` class handles all serialization and deserialization of objects. It can serialize into a byte array any object with the `[StructLayout(LayoutKind.Sequential, Pack = 1)]` class attribute and a default constructor.

The `PeriodicFunction` class is used by the client and server tick functions. The tick function is the function that runs every tick, the period between sending packets.

MultiplayerNetworking.cs

`MultiplayerNetworking` is the interface implemented by the client and server. It contains the shared functionality between each -- for instance, initializing UDP, sending and receiving packets, and registering packet callbacks. With `RegisterPacket`, the client and server can tell `MultiplayerNetworking` to invoke a callback if it receives a particular packet. Packets not registered with the packet are automatically passed along to the game. (Some games might not want to register any packet callbacks and instead just let the game process every packet. This is a perfectly valid approach, and one that we found ourselves doing more of as the project progressed.) Classes which inherit from `MultiplayerNetworking` should call `InitNetworking` at startup to assign a port number to the process and begin listening for UDP packets, and `ProcessPacketsInQueue` from within their tick function to invoke the packet callbacks that were set by `RegisterPacket`.

SnapshotHistory.cs

The `SnapshotHistory` class maintains the client's history of snapshots for client-side prediction and delta compression. It's basically a wrapper around a circular buffer indexed by the sequence number of the snapshot packets. It has a generic constraint of `ISnapshot`, which means the generic argument must inherit from `ISnapshot`. `PutSnapshot` inserts the snapshot into the circular buffer. `Reconcile` can be called from the client to determine if it needs to be rolled back to the server. For the most part, developers need not concern themselves with `SnapshotHistory`, since `Client` and `Server`, described later, use it transparently to the game.

IMultiplayerGame.cs

`IMultiplayerGame` is the interface that multiplayer games should implement in order to be alerted to network events such as packets arriving. It also contains methods for queueing packets to be sent by the networking layer of the game.

Packets are sent differently depending on if they are queued from the game client or the game server. If the developer calls `QueuePacket` from the game logic on the client side, then the networking layer will send the packet to the server it is connected to. `QueuePacket` is overloaded so that you can specify the IP/Port to send it to.

One important thing to note is that there are two main methods of transmitting packets. One is called by passing packets in `QueuePacket`. This will use simple UDP with no reliability in the message being delivered. This is useful for packets that need to be sent fast with no large penalty for losing the packet, such as a player position on an old frame.

The other method is using a simple reliable UDP, with the function `AddReliablePacket`. This function requires as its parameter the packet itself, the location to send it to, and a key. Packets will be stored in a dictionary with the key, and periodically will send all packets in the dictionary, forever. To utilize this, you want to have the receiver respond to each packet it receives with some form of an ACK, which can even be the full packet. Once the sender receives the ack, it will remove the packet, since reliability is achieved. The key in this dictionary must be a value that can be derived from the ACK packet, since you want to uniquely identify that you got an ACK for the specific packet you wanted to send reliably. Use the combination of `WaitingForAck` and `RemoveReliablePacket` to check and remove the reliable packet based on the key.

The game on the client side should call `QueueInput` every frame to tell the networking layer to send a `PlayerInput` packet containing the bitmask of buttons pressed.

Client.cs

`Client` is responsible for performing client-side prediction and synchronization with the server. It has a field called `TICK_RATE` which tells the client how fast, in seconds, to send and receive updates to/from the server. For example, a tick rate of .01667 tells the client to send and receive updates every 0.01667 seconds. `RELIABLE_TICK_RATE` is used to specify how often packets should be resent for packets that require reliability

The port and IP address of the master server should be hardcoded in `MASTER_SERVER_IP` and `MASTER_SERVER_PORT`. After the master server has arranged the game, this changes to the port and IP of the server it is connected to.

The game logic on the client will typically want to own a reference to `Client`. When the player is spawned, `BeginSnapshots` needs to be called by the game to start snapshots (and conversely `EndSnapshots`).

How does `Client` know which type of snapshots to send? This is specified right above the class definition by the alias `MySnapshot`. The developer sets this alias to the type

of snapshot that the client should perform prediction and reconciliation on. In the test game, this is the position and rotation of the player character.

In short, `Client` performs three important functions every frame: It processes the incoming packets (sending to the game for further handling), runs the tick function, and sends outgoing packets the game requested it to send.

Server.cs

The `Server` class is the networking layer of the game server. It keeps a mapping between clients to snapshots and client addresses to server IDs. The main server loop runs every `TICK_RATE` and sends snapshots, game state, and packets requested by the authoritative game server.The game server will want to own a reference to `Server` to take advantage of its networking facilities.

The server will send a heartbeat packet to the MasterServer every `HEART_BEAT_RATE` seconds. This is to keep the connection with the MasterServer open, since routers can choose to drop packets from public IP addresses it doesn't communicate with first. More details about the networking process are described below.

The details of `Server` are similar to that of `Client`. It handles the network end of all of the packets coming in, and going out of the `GameServer` class. It identifies the types of packets coming in and delegates them to the appropriate handler for the `GameServer`.

GameServer.cs

The `GameServer` class contains a component of the `Server` class, and its main functionality is to process the packets that the `Server` receives and change the game state based on the packets. Depending on the game state, the `GameServer` will also call `Server` functions to send packets out. One instance of the `GameServer` is meant to handle all of the functions for running one lobby.

There are three main types of packets the `GameServer` handles. The first type is the packets between the players in a lobby and the `GameServer` itself, regarding match meta information. These packets are created from the `Packet` interface in `Netcode.cs`. These packets are used to communicate things like joining a lobby, leaving a lobby, and many other communication states that are separate from the "gameplay" in a match.

The second type of packets the `GameServer` handles are between the GameServer and the `MasterServer`. When a `GameServer` instance is started, it will notify the MasterServer that it is online. Then, if a player desires to create a match lobby, the MasterServer will assign the lobby to an open `GameServer` instance, and redirect any players that wish to join a specific lobby by acting as the “middle man” and sharing network information of the lobby. The `GameServer` and `MasterServer` will closely share all player information for the lobby that the specified `GameServer` is running, and once a match begins for the lobby, it will close connections with the `MasterServer`, until the match ends, in which case everything returns to the start where the `GameServer` notifies that it available to host a lobby. The communication with the `MasterServer` is done with string packets which contains the type of packet, and colon separated information specific to that packet type.

The third type of packets the `GameServer` handles are snapshot information between its players. The client player will repeatedly update the `GameServer` with information regarding its position and input commands, and the `GameServer` will process the snapshots between all the players in the match and simulate it in the `GameServer` itself. The `GameServer` will act as the authority of the game state, and updates all players with the current game state as the `GameServer` sees it.

The functions `MasterServerEvent`, and `NetEvent`, inherited from `IMultiPlayerGame`, are the key functions in the file that are used to process the different packets it receives, and sends out packets in response. For any type of packet that you add to the framework, have these functions handle the packet in the switch statement. The `MasterServerEvent` handles specific string packets between the MasterServer, and `NetEvent` will handle packets regarding clients.

GameClient.cs

The `GameClient` is very similar to the `GameServer`, in that it handles all three types of packets above, except with the perspective of the client. When it launches it will notify the MasterServer with its online presence, and interfaces with the UI to join and create match lobbies. It has the capabilities to know online players using Steam, and invite players to the lobby they are in through the MasterServer.

The `GameClient` will communicate with the `GameServer` once it is in a match regarding match meta information, and once a match has started the `GameClient` will read in keyboard input from a user and sends its gamestate as snapshots. It will receive snapshot updates from the `GameServer` to update information about other players. Similar to `GameServer`, the class contains many functions to handle and send the

different types of packets it communicates with between the `GameServer` and `MasterServer`, through the central handlers `MasterServerEvent` and `NetEvent`, which will be called as packets are received based on the type of packets. The `GameClient` also has many functions that are called by the UI, such as sending a create lobby packet to the MasterServer when a player clicks on the Create Lobby button.

2.2.3 Gameplay

Bullet.cs
The `Bullet` class is attached to each bullet object. It automatically destroys the bullet after its given `lifetime_` is up. It also tracks whether a bullet is active, allowing other classes to get or set the bullet state. An inactive bullet is invisible.

Gun.cs
The `Gun` class is attached to each gun object. When its `Fire` method is called, it fires a bullet from the front of the gun by instantiating a new bullet object and setting its initial velocity. The initial bullet speed is set by the variable `bulletSpeed_`.

NetworkedPlayer.cs
The `NetworkedPlayer` controls the state of the player. It processes the input from the input packets that have arrived from the server.

MainPlayer.cs
`MainPlayer` is the player game object on the client. It extends `NetworkedPlayer` to allow player movement using Unity keyboard and joystick input.

2.2.4 UI

The following scripts and classes control the UI both in the main menu and in-game. They inform the game manager of user input and may be called by the game manager when the display needs to be updated. However, they do not keep track of or directly control game state or network communication.

MainMenuUI.cs
The `MainMenuUI` class is the primary entity in charge of controlling and coordinating the menu screens in the `MainMenu` scene. It keeps track of menu state, including which menu screen is currently visible. It also informs the `GameClient` of user matchmaking requests, such as join match requests, triggered by user input.

CreateMatchUI.cs
The `CreateMatchUI` class controls UI elements specific to the Create Match menu screen which allows the user to create a match.

JoinMatchUI.cs
The `JoinMatchUI` class controls UI elements specific to the Join Match menu screen which allows the user to view and join existing matches. The class periodically polls GameClient for a list of open matches and then displays these to the user.

StartMatchUI.cs
The `StartMatchUI` class controls UI elements specific to the Start Match menu screen of a joined match. From this screen, players can view the list of players in the match lobby, view and invite Steam friends to the lobby, and start the match. This class calls the Steamworks.NET's GetFriendGamePlayed, and uses its SteamFriends class and its EPersonaState to retrieve the Steam friends of the current user and display them if they are online and currently in game.

SteamJoinMatchUI.cs
The `SteamJoinMatchUI` controls the join match popup that appears when a player is invited to join a match by a Steam friend. It informs the `GameClient` of whether the player chooses to accept or decline the invite.

GameUI.cs
The `GameUI` class controls the in-game pause menu which allows players to resume the match or drop the match.

PlayerDisplay.cs
The `PlayerDisplay` class controls the overhead player display showing a player's current number of lives. It may be called to change the number of lives displayed or set the maximum number of lives.

2.3 Links to Documentation of Third Party Applications

2.3.1 Steamworks SDK

This provides all of the interfacing with Steam. In order to use the SDK, developers must have a Steam account and sign up as a developer here:

https://partner.steamgames.com/. The documentation of the Steamworks SDK can be found here: https://partner.steamgames.com/.

2.3.2 Steamworks.NET

In order to develop with Unity and use Steamworks SDK, third party assets are required, since Steam doesn't natively support Unity. Steamworks.NET is open source and provides an API to allow developers to interface with Steam in Unity. The documentation and information about Steamworks.NET can be found here: https://steamworks.github.io/.

2.4 Getting Started

Developers interested in using the Multiplayer Framework to make a game will want to take the Game Objects they want to synchronize and create snapshots for them, (for more information on snapshots, see the documentation for *Netcode.cs*.) as well as write the callbacks for these packets. For player input they want to send to the server for processing, it works the same way except the developer only needs to add a new value to the `PlayerInput` enum and write the corresponding callback in `NetworkedPlayer`.

2.5 Our Game Specific Features

This section covers features implemented in our game that are not intended to be directly extended or used by developers creating a new multiplayer game. These features are more game specific and were created to showcase some of the elements of our multiplayer framework. Although they were not written for extensibility, they can still be used as a point of reference if you choose to provide similar features in your game.

2.5.1 Gameplay

Our game is a first person shooter which allows players to fire bullets from a gun and take damage from bullets. Each player is assigned a `NetworkedPlayer` script which keeps track of and controls player state, including player position, rotational orientation, and health. Since the player state required to be synchronized is game-specific, this class is also game-specific. However, `NetworkedPlayer` can be used as a general guideline for how to keep track of player state and integrate it with our multiplayer framework. Additionally, `MainPlayer` extends `NetworkedPlayer` to allow the user to directly control their player with user input.

The shooting system in our game is also very basic and somewhat game specific. Our game features guns which create bullet objects that may collide with players. However, other games may use choose to use a hitscan feature to determine if a player is hit rather than creating individual bullet objects. Furthermore, different and more elaborate guns may be used. However, the `Bullet` and `Gun` classes can still be used as a reference for creating basic shooting.

2.5.2 UI

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 as much as possible 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. Additionally, the only time the menu UI is called by other classes is when `GameClient` receives an acknowledgement of a user request and calls `MainMenuUI` to update the display accordingly.

Our UI relies heavily on the use of Unity's built-in UI elements, including canvases, buttons, and input fields. Thus, much of the functionality is encoded in the configurations of components of Unity objects, rather than in UI scripts. However, developers may still use our UI system as a guide for creating new UI systems. Features such as scrollable open match lists, popup invite menus, and overhead player displays may be applicable to other multiplayer games.

3. Important Networking Information

3.1 Public IP Addresses

There are many subtleties to the ways in which the framework handles connection between clients, servers, and MasterServer. Since client and servers usually exist in a private network and communicate to the internet through their router, their public IP address is unknown within the private network, and change with every connection. This is where the MasterServer comes in, in that by first connecting to this public IP source, the MasterServer will let all clients and servers know what public IP they must connect to, even if the destination is hidden behind a private network.

3.2 NAT Punchthrough

Even if the client and servers find out each other's public IP and port through the MasterServer, they will not be able to communicate with each other. The reason for this is that routers will drop any public IP addresses it doesn't send packets to first. Therefore, when the MasterServer notifies the connection information of a client and a server that want to connect, then they must each send a random packet to each other first, so that their routers will accept packets between them in the immediate future. In our code, GameServer and GameClient will send these initial packets in their respective ReceivePlayerJoin functions. In the GameClient's case, it is just repeatedly sending the packet to join GameServer's lobby, and the GameServer sends a random string.

This is also the reason why the Server sends heartbeat messages to MasterServer. After a certain time, a router may delete its port tables if connection to a public IP is not established. By sending random heartbeat messages to the MasterServer, it allows the MasterServer to contact the local lobby when need be.

3.3 Router Hairpinning

A very rare case that must be taken into account is router hairpinning. In the instance that you host a lobby in a server in the network as the MasterServer, the messages may not be sent between the server and client, since some routers don't traffic to leave and land back in the public end of the router. This is called router hairpinning, and in this case you may need to use an internal IP address to connect.

Many of these networking cases are covered in helpful online articles. We recommend https://keithjohnston.wordpress.com/2014/02/17/nat-punch-through-for-multiplayer-games/ For more information.

4. Master Server

4.1 Description

The Master Server is a component of the Multiplayer Game Framework that allows servers and players to be able to find each other on the Internet, implemented in a single C++ file (**masterserver.cpp**) and supporting Windows machines. Master Server uses UDP packets for transmission and exists for 3 primary reasons: to resolve NAT issues and provide UDP hole-punching, to collect the IP addresses of server instances in one place, and to allow users to send invites to one another by knowing which players are connected to the game. Since clients must learn the IP address of servers in order to connect to them, Master Server collects the IP addresses and ports of any instance of a server that begins execution. When servers and players each start up, they first connect to the Master Server in order to register their name and address/port. Players can request the names of each server that is open, join and quit servers, create lobbies, and invite other players to the game since Master Server knows all of the public IPs and ports. It features basic reliable transmission through a custom ACK protocol, where repeat requests are not processed and packet loss is accounted for.

One thing that Master Server provides that is necessary for making a multiplayer game across multiple networks is NAT punchthrough. Since NAT port mappings can be anything and may change, it is impossible for a player behind a NAT to directly connect to a server that is behind a NAT. By connecting first to a master server on a static IP, these programs can circumvent the NAT. Thus there are some limitations on where you host the masterserver: it must either A) have a static IP address or B) have the port (default 8484) masterserver uses forwarded on the router. Another symptom of this is that you cannot host a dedicated server on the same machine as the Master Server, though you may host a client.

Machines hosting players are identified on the Master Server by a unique username, while machines hosting lobbies are identified by a Region name used to partition servers and a Lobby name that is unique to a Region.

Server and player data is saved across seven cross-referential maps:

map<string, vector<string>> serverlist

- maps Region to a vector of lobbies that are waiting to be joined. This is what is sent to players when they ask for a list of servers.

map<string, vector<string>> openlobby

- maps Region to a vector of machines hosting servers waiting to be used. When lobbies are created, they are pulled from this list.

map<string, string> lobbyport

- maps a Region:Lobby to an IP address and port. This is updated anytime a new server registers with the masterserver.

map<string, vector<string>> lobbyinfo

- maps a Region:Lobby to a game-dependent vector of lobby information like maximum players, game mode, etc. This is periodically updated by servers with the “lobup” packet.

map<string, vector<string>> playerlist

- maps a Region:Lobby to a vector of users that have connected to that lobby. This is used to handle matchmaking and invites.

map<string, string> currentgame

- maps a User to the Region:Lobby of the lobby they are currently in. This is used to handle matchmaking and invites.

map<string, string> playeraddrs

- maps a User to the IP address and port that they last pinged the Master Server with. This is updated when players request a server list with “pslis”.

The general control flow of the program:

1) Receive packet from UDP socket if there is anything in the buffer, if not then 3)
2) Parse/process the packet, changing values in Master Servers maps and sending ACKs to the appropriate machine. Remember which packets you are waiting to receive ACKs for.
3) Retransmit any unACKed packets that have timed out according to RTO

These 3 values can be changed in the source code to tune and add stdout logging:

RTO(X), where X is time before retransmitting packets, in milliseconds

line: `chrono::duration<int, ratio<1, 1000>> RTO(250);`

timesToRetransmit, # of times to retransmit a packet

line: `int timesToRetransmit = 3`

flag, a flag to output map contents every time you receive a packet

line: `int flag = 1;`

4.2 Packet Formats

The following are the formats of the possible packets that Master Server can receive from players and servers. They consist of a 5-character command followed by a space, then a colon-delimited list of arguments :

stser [region]
Registers sender's IP:port as able to host a lobby for specified region

stlob [region]:[lobby]
Starts creating lobby in an open server in region

slack [region]:[lobby]
Finishes creating new lobby

close [region]:[lobby]
Close specified lobby, and region if it has no more lobbies

lobup
Heartbeat sent by servers to keep their NAT mappings

pslis [user]
Get serverlist (list of regions)

pllis [region]
Get lobbylist (list of lobbies/servers available in region)

pjoin [user]:[region]:[lobby]
Start adding specified user to specified lobby

pjack [user]:[playerIP]:[playerPort]:[region]:[lobby]
Finish adding specified user to specified lobby and send the lobby's IP:port back to the user

pquit [user]
Remove user from whatever game they are currently in

pinvi [fromUser]:[toUser]
Send an invite to toUser containing the lobby that fromUser is in

piack [fromUser]:[toUser]:[region]:[lobby]
ACK that toUser received the invite from fromUser

qplay [user]:[region]
send user the IP:port of an available lobby for the region

clear
clears Master Server's memory, essentially restarting the Master Server

4.3 Suggestions for Improvement/Replacement

To extend the current Master Server, you could add fields to lobbyinfo so that users can see more info about lobbies when they request lists. This is done by sending info in the "lobup" packets and changing Master Server to parse those additions and add them to the lobbyinfo[lobby] info vector. Then you can add them to the 'psack' packet that is sent to users that send Master Server 'pslis'.

To really get a more scalable or robust system, the better option would be to use the Steam Multiplayer API itself. We could not use it as you must apply for a license from Valve Software to access this API. Using Steam Multiplayer, the Steam servers act as a Master Server that also provide better authentication and a more robust set of features. Lobbies are hosted on Steam servers, and users send invites using the Steam API rather than the roundabout way that we do it.