TachyoNet

Andrew Cunningham, Keet Curtis

CSE 481S -- Winter 2019

[**Section 1. Executive Summary**](#_aadg695ukwtx) **4**

[**Section 2. Project Overview**](#_jvee4qnqnm51) **5**

[Section 2.1. Motivation and Use Cases](#_4n2utc6zgfm) 5

[Section 2.2 Related Works](#_698gzregracl) 6

[**Section 3. Requirements**](#_qg72nr9fvnhy) **6**

[Section 3.1. Brief System Summary](#_7jtjrhxun9ma) 6

[Section 3.2. Functionality Goals](#_xur6m2wmvdz8) 7

[Section 3.3. Functionality Non-goals](#_d002o6ko5oew) 7

[Section 3.4. Threat Model](#_a9pm2z75g4bt) 8

[Section 3.4.1. Actors](#_g9jo8zywqz1i) 8

[Section 3.4.2. Assets](#_dg0244uuzl19) 9

[Section 3.4.3. Security Goals](#_pholxf6pxfaf) 9

[Section 3.4.4. Security Non-goals](#_mw9cuopcjaev) 10

[**Section 4. Design**](#_s6xxc7h613s7) **11**

[Section 4.1. Server-Client Communication](#_ui3h4s4v7lh1) 11

[Section 4.1.1. Server-Client Functionality](#_t2w942xziiqk) 11

[Section 4.1.2. Server-Client Security](#_wl88q8s59tm3) 11

[Section 4.2. Server Map and Item Management](#_mkjynlnznsyj) 12

[Section 4.2.1. Map and Item Functionality](#_t0jd5bhvdgnr) 12

[Section 4.2.2. Map and Item Security](#_yk7b6f1h5v2k) 13

[Section 4.3. Player Interactions](#_p1onrms645nn) 13

[Section 4.3.1. Player Functionality](#_tzs1c29ff7yi) 13

[Section 4.3.2. Player Security](#_bgvsr9adppvt) 13

[Section 4.3.2. Server Action Verification and Accountability](#_6u1dc2khyv3j) 13

[Section 4.4. Known Issues](#_3ghq49qxoy8g) 14

[Section 4.4.1. Server Reliability](#_m4ist9myochn) 14

[Section 4.4.2. DDOS Protection](#_xvvq54g3h3i2) 14

[Section 4.4.3. SteamWorks](#_rdzotsoso5h7) 14

[Section 4.4.3. Possible Vulnerabilities in Telepathy’s TCP Protocols](#_bb2jkqv4be9o) 14

[**Section 5. Implementation**](#_h9u0iylsc8hn) **14**

[Section 5.1. Running Your Project](#_b1an6x2mwvfx) 15

[Section 5.2. Functional Implementations](#_ckexndd6tydr) 16

[Section 5.2.1. Server Hardware](#_apmfb5gzjs1z) 16

[Section 5.2.2. Server Communication](#_ymwomodrd6zp) 16

[Section 5.2.3. Server Update Interval and GameState](#_1gwxtpj7b089) 16

[Section 5.2.4. Client Input and Player State](#_7z0ddfwhkbwx) 18

[Section 5.2.5. Client’s Player Update and Server’s Game Update Communications](#_ladzyzgbpt12) 18

[Section 5.3. Security Implementations](#_33psogegyg61) 19

[Section 5.3.1. Authentication](#_cxrzfwvbea4) 19

[Section 5.3.2. Encrypted Communication](#_nsk0dvgxs73n) 20

[Section 5.3.3. Server Verification: VerifyPosition](#_pkod5gwy5j5s) 20

[Section 5.3.4. Server Verification: VerifyHit](#_u8neswzd3801) 21

[**Section 6. Security Evaluation**](#_o0ukkfeqlzq3) **21**

[Section 6.1. Security Evaluation from the Team](#_wq78nnqev51d) 22

[Section 6.1.1. Areas of Possible Weakness](#_d9pidm5bxxd3) 22

[Section 6.1.2. Areas Out of Scope](#_9likb1edk42f) 22

[Section 6.1.3. Within-team Peer Audits](#_yzzkepx67lmg) 23

[Section 6.1.3.1. Andrew’s Client Analysis](#_7t3cw48qatrn) 23

[Section 6.1.3.1. Keet’s Server Analysis](#_7x8rddqzp343) 23

[Section 6.2. Response and Revisions Following Peer Analysis](#_7p39fd3n5w8) 24

[Section 6.2.1: User input into game console](#_dcrqxxxzxxb3) 24

[Section 6.3. Summary](#_l27wds7z6tqj) 26

[**References**](#_xn1qwkdt6aod) **26**

[**Appendix A. User Centered Design**](#_y7ojetjln4vu) **27**

[Appendix A.1. Methodology](#_hik6huvpnhcn) 27

[Appendix A.1.1. General Theme](#_915eb27ghyjb) 27

[Appendix A.1.2. Location and Time](#_y6fuyl9ofp15) 27

[Appendix A.1.3. Expected Participants](#_rwwk1xe2tkie) 28

[Appendix A.1.4 Participant Interaction (Reddit Thread Conversation)](#_owlgtgn06bq6) 29

[Appendix A.1.5 Budget, Materials and Other Logistics](#_lpsis3wubh6n) 29

[Appendix A.2. Results](#_5d3t4mxdn7xz) 29

[Appendix A.2.1 Results: Participants’ Demographics](#_t9zr8bhdrd2p) 30

[Appendix A.2.2 Methodology changes](#_b0d0dokvdu3b) 30

[Appendix A.3. Analysis](#_cnndh0brpsqn) 30

[Appendix A.4. Project Implications](#_3mh82v3rjhky) 31

[Appendix A.5. Going Further](#_b2f9xbfymfqk) 32

[Appendix A.5.1. Security Vs Effort](#_95smj41z1s08) 32

[Appendix A.5.2. An Online game, PWN Adventure](#_yxqor4hjnxed) 33

[Appendix A.5.3. 3rd Party Clients](#_vmalzm5cuog0) 33

# 

# Section 1. Executive Summary

TachyoNet is an early prototype of a system of tools that integrates security and functionality into a game that has multiplayer elements. For this early prototype we demonstrate multiple balls rolling around a simple map and can shoot other balls, changing their color. The prototype involved creating a client to run on a Unity game and creating a server that is houses on an off site Linux machine.

The client is baked into Unity and runs as an asset in the game’s working environment. It utilizes the Unity engine by capturing gameplay as it happens and sending it off to the server so that the player’s character can be integrated into the server’s GameState. The server also receives these updated GameStates from the server and translates them into objects in the world.

The server manages connected clients, maintains the GameState, verifies, incoming updates, and sends the GameStates to the clients. Upon initial connection the server must authenticate the client to ensure it’s not someone trying to send malicious inputs through an invalid source, then create a new player for that client. After receiving updates from the clients, which have presumably been generated through the Unity game engine, the server must run some basic verification functions to ensure the update is within a tolerance of expected norms. After verifying, the server must then correctly apply the update to the GameState.

The security features broadly include:

* AES 256 encrypted communication
* Client authentication
* Position update verification
* Shoot update verification

TachyoNet seeks to achieve tremendous flexibility by closely adhering to rigid protocol. Every update sent too and from the server must fit into a specific, serializable, JSON object. Since this object breaks down into simple byte arrays tremendous flexibility is opened up for the clients. Nothing in the server needs to change to account for different client game engines or platforms (like Windows, MacOS, mobile, etc.)

The project itself is a proof of concept that will be applied on a much larger scale to an existing game produced by Mitchell Gentry. All objects and interactions in the game world are placeholders, and will be changed to fit in with more complex gameplay. The core functionality of TachyoNet, however, will remain largely the same.

Sincerely,

Andrew, Keet, and Mitchell

# Section 2. Project Overview

The TachyoNet client/server project has the objective of creating a series of secure, robust, tools that will be deployed in creation of several games created by Theine Dreams, a group of game developers currently split between Seattle and San Diego.

Every game franchise has a suite of unique tools that, when working together, give the games their unique playstyle, artistic impression, and “feel.” The aim of TachyoNet is to build the first set of tools that will give Theine Dreams its own feel, while focusing on the second, and equally important, task of laying a solid foundation of security. These tools will be applied to a test project codenamed ManaCraft, and any mention of TachyoNet thus forth is referring to the relationship between TachyoNet and the prototype game.

A game could be filled with unique storytelling and game play, but without proper security the game quickly is either ruined or broken. As such, security will be the main focus of the TachyoNet. Andrew and Keet will bake a foundation of security in every tool their create so that gameplay developers and artists can create captivating gameplay that isn’t immediately defeated by juvenile security vulnerabilities.

By the end of Winter quarter 2019 the prototype game, ManaCraft, will be in its early phase of prototyping critical client/server infrastructure.

## Section 2.1. Motivation and Use Cases

The main motivation for TachyoNet is to create a robust set of secure tools that can be applied towards creating a unique feel for several future games.

* **Use Case 1. Multiplayer arena-style shooter prototype**. The first application for this server will be towards prototyping a new genre of shooter that has players all starting with nothing in a procedurally generated map and are tasked with resource collection and item gathering to increase their strength as they accomplish objectives and fight their way to victory.
* **Use Case 2. Cooperative, story driven, role-playing-game (RPG)**. These tools will be applied towards a cooperative RPG where players will preserve progress but spawn into a procedurally generated world, gather resources, and work cooperatively with friends to claim victory through objectives, defeating non-playing characters, or defeating other players.
* **Use Case 3. Large scale battle-royale-style multiplayer game**. These tools could be applied on a larger scale to create a large, procedurally generated, open-world map that 100+ players spawn into and fight towards victory. This use-case is a larger-scale application of use-case 1.

## Section 2.2 Related Works

TachyoNet will seek inspiration from various existing game servers while bringing new ideas to the table. Some existing projects include:

* **Telepathy**. Telepathy is a free Github project that developers can use to quickly deploy a multiplayer server that utilizes the same techniques that large scale publishers use. Telepathy is simple, easy to use, and only about 400 lines of code. ManaCraft will seek some inspiration from Telepathy, but deviate quite a bit in regards to gameplay support and security. Telepathy has no security infrastructure in place: no authentication or encryption. TachyoNet will have both of these.
* **Fortnite.** Fortnite popularized the idea of a large scale action multiplayer game focused around resource gathering, with everyone starting off with nothing. ManaCraft hopes to expand on this basic idea by having players fight in a procedurally generated map.
* **Minecraft.** Minecraft supports multiplayer action in procedurally generated worlds. The capability of this feature has grown more complex with time, but the main model used is still for one person to host a customly configured server that other players join. ManaCraft will utilize world generation algorithms that seek inspiration from the algorithms popularized the MineCraft creators (and are now free to utilize) and integrate centrally hosted servers.
* **Call of Duty.** Every multiplayer shooter has to pay respects to games like Call of Duty that helped build this genre. Activision can be given a lot of the credit for popularizing the concept of fast-action multiplayer shooter games through the Call of Duty franchise, and games like these are the source of a lot of inspiration for ManaCraft

# Section 3. Requirements

## Section 3.1. Brief System Summary

The TachyoNet project, Winter 2019, will focus mostly on security. Gameplay elements will be included as part of this larger package, but most game development will happen in future projects. TachyoNet will include the server as well as the client processes running on player’s machines. Authenticity will be managed through the API provided by Valve Software, called SteamWorks. Client-server communication will be handled through the TCP protocols and will be secured using encryption and hashing. Initial gameplay will be sparse, but the framework will be laid for item and inventory management and resource spawning and gathering around the world. The server will be responsible for ensuring items acquired through honest means and handled properly.

## Section 3.2. Functionality Goals

Once the TachyoNet is complete, it will have the following functioning components

* **Decentralized, Remote, GameState.** Given that many clients will be playing one game that’s being hosted on the server, it’s important for the GameState to be updated in one, centralized, location. All design and implementation will keep this idea in mind as it preserves gameplay and security.
* **The Server.** A functioning server that users can connect to in order to play with each other in the in-game world. The server will be responsible for maintaining the GameState, that is, a centralized state that the game is in for all clients. The server will accept user input, update the GameState accordingly, and then push it to the clients. As explained further on, this means the server is responsible for player interactions, world management, and item management.
* **The Client.** The client is mostly responsible for secure transmission of packets. Each client has to know who they are and how to handle send/receive requests. The client has to know how to advance the game based on what the user does. For example, if one client shoots an object that can move, that user’s client might be allowed to let the object actually move but other clients have to listen to the server so that they all receive those same “move” instructions.
* **Player Interactions with their characters.** Player controls, movement, and animations are one of the most important elements for establishing a game identity. The TachyoNet project will use default Unity assets to provide a placeholder that can be sculpted to work with the server.
* **Player Interaction with World Assets.** Items and other assets will be populated into the world. Tools must be built to facilitate this interaction between all of the clients. The items themselves and player interactions are an objective of Mitchell Gentry, whereas the network communication is an objective of Andrew and Keet.
* **Player Interaction with Other Players.** Like above, player to player interaction is critical to gameplay advancement. Andrew and Keet will build the tools needed for this to happen.

## Section 3.3. Functionality Non-goals

* **Game Menus, Lobbies, and Private Sessions.** Any functional game on the market needs some notion of a main menu and means to connect to a game session. This is a “next step” implementation after the initial tools are established.
* **Completely unique player.** Developing a unique game identity for player movements and interactions will be built in the future. This project will use basic assets readily available from the Unity Asset Store.
* **System for long-term player advancement.** Each round will be a unique instance, and a system that carries progress between rounds will be implemented in the future.
* **Non-Playing characters.** TachyoNet will not have to create or manage the AI behind any NPCs.
* **Story Telling, Cutscenes, and Complex Animations.** These elements are all outside the scope of the TachyoNet project, but are still desired elements in the future.
* **Procedurally Generated World.** A long term goal is to have the world be procedurally generated using algorithms inspired by the mechanics that exist in MineCraft. After generation the world will be populated with items and players and given to each client. The world is one of the focuses of Mitchell Gentry, the lead game developer based in San Diego, CA.

## Section 3.4. Threat Model

### Section 3.4.1. Actors

To ensure that our TachyoNet is secure, we need to be aware of all the possible actors that could interact with the server. Many of these actors are people who we would expect to interact with the server while some of them are not. Any of these actors may be malicious, so we need to design the system with that in mind. The actors that are expected to interact with the TachyoNet are:

* **Players.** The players are people connecting to the server to play the game. The main goal is to prevent players from maliciously updating the GameState beyond the intended framework.
* **Game Developers.** The people designing, building, and maintaining the game built into the server.
* **Steam Administrators.** The people working at Valve who run the online game store, Steam. ManaCraft will be run on this platform.
* **Telecommunications Companies.** These companies provide the network services that the clients and server run on.
* **Programs and Users Running on the Server.** TachyoNet will be hosted on a clean version of Centos and no other living users will be on the machine. However it’s entirely possible for a user to infiltrate the machine and compromise its security.

The actors that are not expected to interact with the TachyoNet are

* **People on the Player’s Local Network.** Friends and family of the player that are using the same local network connection as the player.
* **People on the Broader Network.** These are people on the network that connects the server and clients together. If these people are malicious, they could attempt to intercept the packets that are exchanged by the server and the clients.
* **Shoulder Surfers.** These are people who may watch the player playing the game over their shoulder.

### Section 3.4.2. Assets

When securing our system, there are important assets that we need to protect. Some important assets that are relevant to ManaCraft are:

* **Steam Login Information.** These are the credentials that players use to login to their Steam account to play games they have purchased. An adversary who gains access to this information could buy unwanted games with the user’s credit card, make unwanted changes to the user’s friends list, etc. ManaCraft will be using SteamWorks for authentication and use the given token for client authentication. It seems unlikely that this token can be used maliciously, but care must be taken to protect it none-the-less.
* **Personal Player Information.** Player-player communication will not be directly supported in this project, but social engineering is possible through indirect channels. A malicious player could gather any personal information available and use other methods to contact players outside of the game. Care must be taken to ensure that PII is not collected. We don’t need to do anything with PII since SteamWorks is handling the authentication, so there’s no reason to collect it.
* **In-Game Items.** These are items that the player collects in the game. An adversary may try to delete these items and thus negate the player’s progress. Another possible way an adversary may exploit this asset is to acquire extra items using methods that were unintended by the game developers.
* **The World.** Since the world is randomly generated the server must run checks to ensure the clients are not manipulating it in any unauthorized way. If the server is programmed to blindly update the GameState based on player input then the way is paved for anyone to do anything.

### Section 3.4.3. Security Goals

Secure, reliable, communication is the overall goal. In more detail this includes:

* **Basic Authentication Measures.** It’s difficult to provide reliable authenticity for the server without the use of SteamWorks, but some basic measures must be enacted. If there’s no authentication then anyone with a terminal can send packets to the server and potentially update the state of any player as their please. A simple, although likely easily exploitable, authenticator will be implemented.
* **TCP Networking Protocols.** TCP enables the server and client to remain assured that they received the correct state. It greatly reduces the risk of compromising the game due to lost packets.
* **Strict JSON Object Adherence.** All network traffic will fit one, single, JSON object. The server will only guarantee a response if the format is correct. Wrong format, the server will not respond. It would be possible to make the server parse any update and look for valid information, but this opens up countless attack vectors. Also, by not responding if the packet isn’t in the correct format, an attacker can’t spam the server with small changes to the packet and take error feedback into account.
* **Encryption of Sensitive Networking Traffic.** Networking traffic like the position updates for the client and overall GameState updates from the server should be encrypted. Ideally this accomplishes two purposes. Encrypting the objects helps hide their format, thus making it difficult to replicate the required JSON formats. It also prevents third-party scripts from listening to the traffic which can be used, for example, to tell the player where everyone is on the map.
* **World Map and Item Management.** The server will be responsible for maintaining the world, players, and items. The server will have to communicate with the clients to ensure there are no discrepancies. The server will also be responsible for spawning items around the world that the players can acquire and use. Accountability measures will need to exist to prevent abuse such as item duplication, malicious spawning, or unauthorized item stat nerfing.

### Section 3.4.4. Security Non-goals

The following are some security considerations that are beyond the scope of the current project.

* **Steam Authentication.** Once published as an early-access title on Steam access to SteamWorks will be granted, allowing utilization of Valve’s current authentication system to establish secure initial communications. This will be a perfect opportunity to create unique player identities, public/private key combinations, and other necessary measures.
* **Server Status Reliability.** There are no plans to implement features that increase server reliability like a distributed system with fail-safes, expensive hardware, or software failsafes.
* **Social Engineering of Players.** There is no in-game multiplayer chat in the initial server plans, so there are no plans to prevent adversarial players from using social engineering to obtain sensitive personal information from other players.

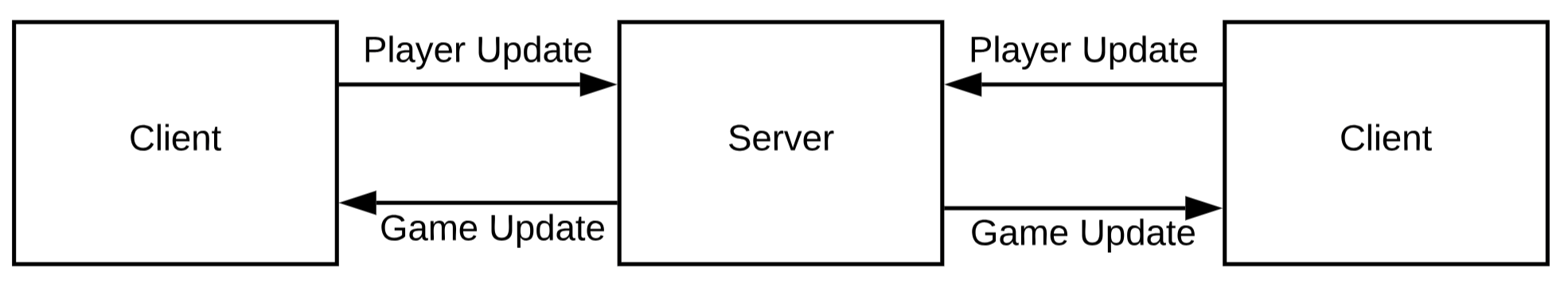
# Section 4. Design

The underline focus with all design decisions will be balancing speed and security. Low latency will allow quick, smooth, and detailed gameplay while adequate security prevents abuse. This will be a constant point of compromise because the most secure systems are also far too slow for gaming demands.

## Section 4.1. Server-Client Communication

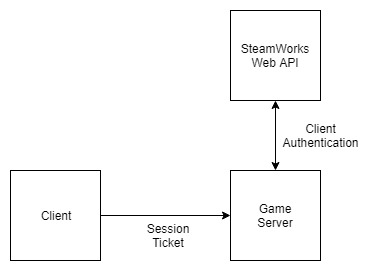
### Section 4.1.1. Server-Client Functionality

The most basic function of this server is to communicate quickly and reliably with the clients. We will be using basic network building techniques used all over the industry to build the foundations. We are currently looking at Telepathy as a large source of inspiration in designing a basic checklist of features. Clients will read the updates that happen inside the player’s Unity environment and send these updates to the Server. The Server must, then, perform some verification math to ensure that the difference in state is within a tolerance. The Server then returns the total picture to all the Clients. This is shown in the diagram below. We will be using TCP as it’s been shown to provide quick-enough communication that is also more reliable than UDP.



### Section 4.1.2. Server-Client Security

Before game communication between the clients and server can start, the client must be authenticated as a valid player that owns the game. This is done by the client first acquiring a session ticket from Steam on the user’s computer. The client then sends the session ticket to the game server. When the game server receives the session ticket, it sends the ticket to the SteamWorks web API to be authenticated. Once the server receives confirmation that the client has been verified, gameplay communication can begin. This process is shown in the figure below.



The communication between the server and clients has data that needs secured. A possible attack vector is the attacker tracking information about opponents weak spots (say, the x,y,z coordinates of everyone’s heads) then relaying to the server something along the lines of “Fire Weapon towards x, y, z’ by running a script that embeds the desired coordinates into memory. The developer needs to be given the freedom to determine what values are protected and what can exist as plaintext.

A possible solution is implementing objects that will easily allow developers to decide what variables need to be encrypted or randomized by other security methods like a stack canary, and what can traverse the network as plaintext.

After initial functionality is established, finding this line between speed and security will be a critical element in this project. We will explore several different methods of securing the sensitive data and then perform speed and CPU load tests to find an acceptable balance.

## Section 4.2. Server Map and Item Management

### Section 4.2.1. Map and Item Functionality

The server has to manage the map, players, and items that exist in the world. There are several Unity tools, such as SmoothSync, to make this process easier to implement. The trade-off is increased reliability on external assets and, therefore, more security unknowns, but it could prove sufficient for the prototype.

To accomplish this the server needs to have access to a database of items, players, and actions. The prototype will contain a GameInfo class that contains this information, but it won’t be communicating with the Unity environment. A future plan will be to make a developer script in Unity that sends the updated file to the Server. This is because the verification would fail if the developer updates something like jump height in Unity but doesn’t put it on the server.

### Section 4.2.2. Map and Item Security

Protecting the integrity of the map and items will be handled through smart use of tools like authenticity, hashing, and TCP. Since the GameState updates will happen server side it’s difficult to maliciously impact the GameState, but the client needs to be able to predict actions to give the user rapid feedback. This method is visible in many games. A common source of frustration in Call of Duty is when player1 delivers a fatal shot to player2, player1 observes the animation begin, but suddenly ends up dead and a point is awarded to player2. This is simply due to latency. Both players actually shot at roughly the same time but player2 made it into the queue before player1. This is a natural consequence of server latency and will never be perfect, but the client must be able to predict gameplay so that there isn’t hundreds of milliseconds of input lag.

## Section 4.3. Player Interactions

### Section 4.3.1. Player Functionality

For the early prototype we will demonstrate player movement, a limited amount of weapon usage, and interaction with items in the world and other players. There are many tools on Unity to make this relatively straightforward, and a “first-person player” asset will be utilized early on. As the prototype moves forward the player will be changed to give the game a unique feel.

### Section 4.3.2. Player Security

The early stages of development will rely heavily on Unity Assets, collected from the Unity Asset Store, having smartly-built features that help mitigate player abuse, but modifications will have to be made to account for case-by-case situations.

### Section 4.3.3. Server Action Verification and Accountability

Much like the server needs to authenticate connection, it also needs to be able to verify that the requested action is valid. We want to make the Unity client do the heavy lifting in regards to raycasting, but the server must have a primitive raycaster for authenticity. If player1 says that he shoots player2 in the head, the server must do a raycast to verify that action was possible. This is going to a very simple trigonometry calculation to extrapolate the action vector through space and see if it comes into contact with anything.

## Section 4.4. Known Issues

### Section 4.4.1. Server Reliability

There are no plans to prototype any hardware or software to increase server reliability. This server will likely be hosted on an old computer living on a home network which is prone to all the reliability vulnerabilities one would expect in a home-network device.

### Section 4.4.2. DDOS Protection

Distributed Denial of Service attacks are becoming increasingly problematic thanks to the massive increase in cheap, vulnerable, internet-connected gadgets. There are no plans to include measures to protect against these attacks.

### Section 4.4.3. SteamWorks

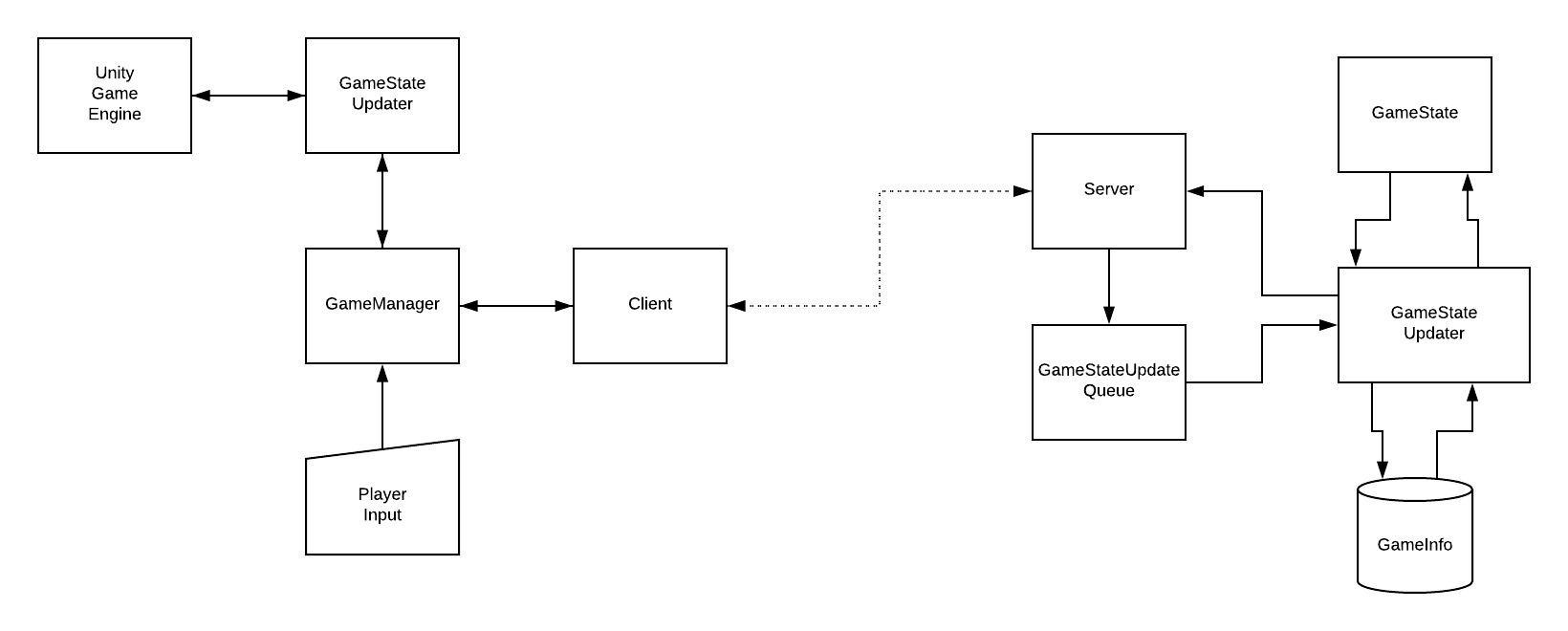
All of the authentication is taking place through Valve’s SteamWorks. The server will be vulnerable to weaknesses in SteamWorks. This is a concern that won’t be taken seriously at the time since there are liabilities far larger than this small server that SteamWorks is responsible for securing.

### Section 4.4.3. Possible Vulnerabilities in Telepathy’s TCP Protocols

Limited networking knowledge puts Andrew and Keet at the mercy of poor decisions made by the Telepathy developers. Given our lack of networking knowledge, it’s possible that we won’t be able to catch a subtle vulnerability in Telepathy.

# Section 5. Implementation

TachyoNet will have a number of moving parts that will be spread between scripts that are baked into the Unity game over to standalone executables running on the Linux powered Server.

  
This is a generalized diagram of the implementation plan for general gameplay.

On the client side GameManager is going to be the mediator, taking in data from the player and client, and both update the local Unity game and push that user input to the server.

The Server is responsible for collecting all users’ inputs and updating the GameState through a collection of update handlers. It will use several scripts to calculate the validity of player actions before storing that information on the GameState and then pushing the updated GameState back out to the clients.

## Section 5.1. Running Your Project

All files for this project will be stored in the GitLab repo for ease of instructor access. The repo will then be cloned to the developer’s local filesystem and split between the server and the Unity project.

* **Unity:** This is the client functions that are running in collaboration with Unity. They will be stored as scripts in a Unity project and synced between developers on the Unity cloud. It should only be necessary to push the client files into the GitLab repo when a deadline comes up and code submission is required to the course staff. These scripts will be compiled and run with Unity. So when the developers hit “play” from within the Unity IDE the client will automatically establish connection with the server.
* **Linux Server:** The server software will be edited with an SSH connection to the server which is currently sitting on Andrew’s entertainment center. Developers will edit, compile, and run the server which will listen constantly. When it’s time for code submission developers will git push these files to gitlab.

## Section 5.2. Functional Implementations

Functional implementations are based around achieving Unity integration for the client and standalone operations for the server.

### Section 5.2.1. Server Hardware

Andrew had an old desktop in storage with an Intel Core 2 Duo processor. It should offer plenty of processing power for the purposes of this project. The computer was given a fresh installation of Ubuntu and setup with an SSH connection and two ports forwarded on Andrew’s home network. One port is for an SSH connection and the other is game traffic.

If the network volume increases it may be pertinent to increase the number of open ports, but the early prototype will only see a few active connections.

When the game is scaled for larger deployment the server host will need to be greatly expanded.

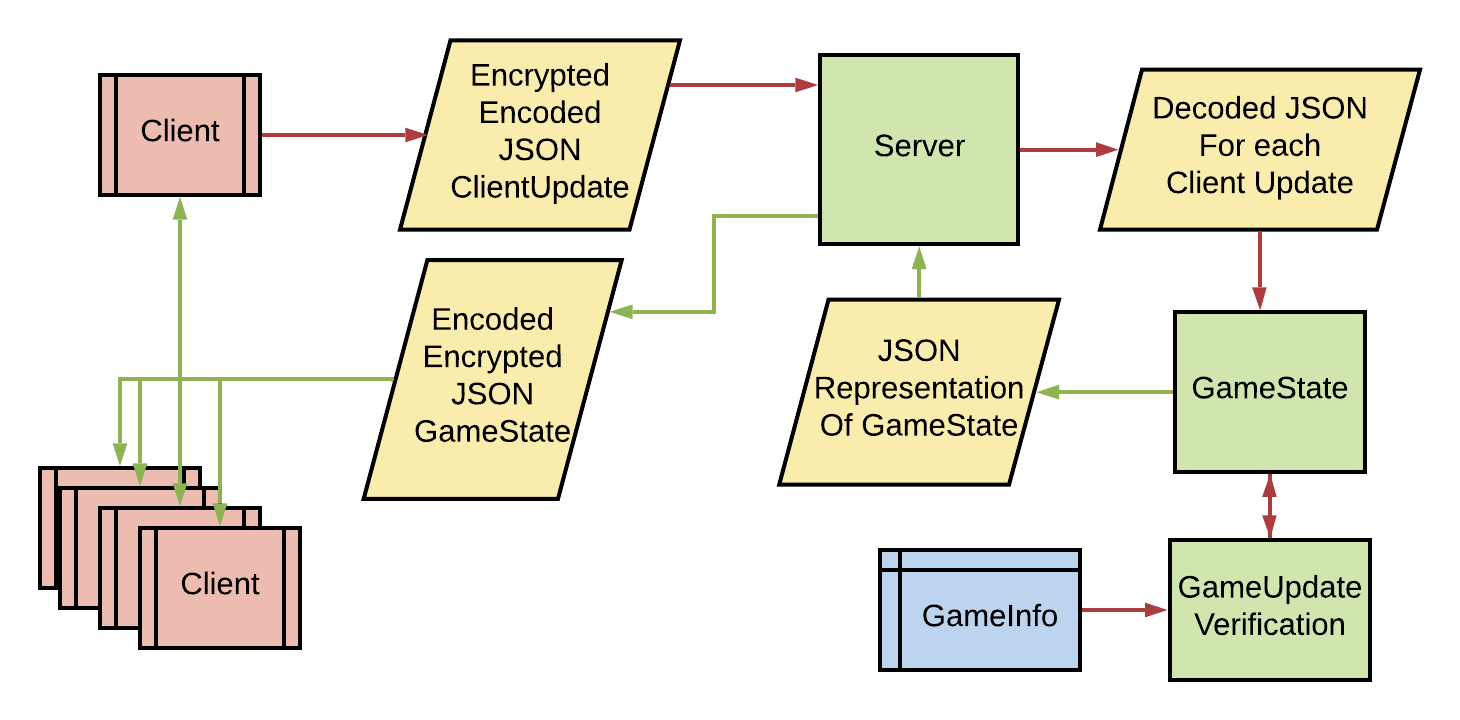
### Section 5.2.2. Server Communication

Telepathy was used as the base library for client-server communication. After connection is established all Telepathy traffic happens as simple arrays of bytes into a Telepathy Message that has the type of “message.” This concept had to be greatly expanded upon to meet all the needs of both the client and the server. An important security feature is that every message fits a singular JSON object, so it must be robust and checks through to ensure packets are handled correctly. The server only sends out a GameUpdate object that has fields for SessionToke, PlayerId, an array of PlayerStates, an array of Splats for weapon effects. This object is highly flexible in that we can add more fields to it later on to expand features in the game.

### Section 5.2.3. Server Update Interval and GameState

The first major software development is developing the update interval, or tick, and an object that represents GameState.

The tick is the interval that the server updates. Normally the server updates at a fixed 32 or 64 times a second. Implementation has to take into account the time to update the GameState, making careful use of timers. The server must constantly be logging packets into a queue and process them on a very specific time interval to ensure persistent continuity of gameplay. Updates cannot just “happen as they come” because this would offer too much of an advantage to clients with a faster connection speed. Limiting the update interval increases consistency and fairness.



*Diagram: Control flow of information on the server.*

Hand in hand with update interval is also GameState. This GameState, currently, stores a PlayerState object and an array of new weapon effect Splats that need to be pushed to all of the clients.

When a GameState processes an update from the player it checks to ensure that the player is active inside of it’s player list. All players are added or removed by the server directly, so the GameState simply ignores the update if the player is either not in the list of set to an invalid state. Note that the invalid state is a feature that exists, but is under-utilized given the scope of this project.

After checking for the validity the GameState calls the appropriate verification function for the type of update, then applies that update to the GameState.

Weapon shot effects are processed by the GameState, but they are simply copied into a list of “splats” that need to be pushed to the clients. It’s definitely an annoyance if players are able to get splats applied to the game that don’t belong because the weapons sounds and effects are a very important gameplay element, but having the server store and push them are a good baseline for future growth.

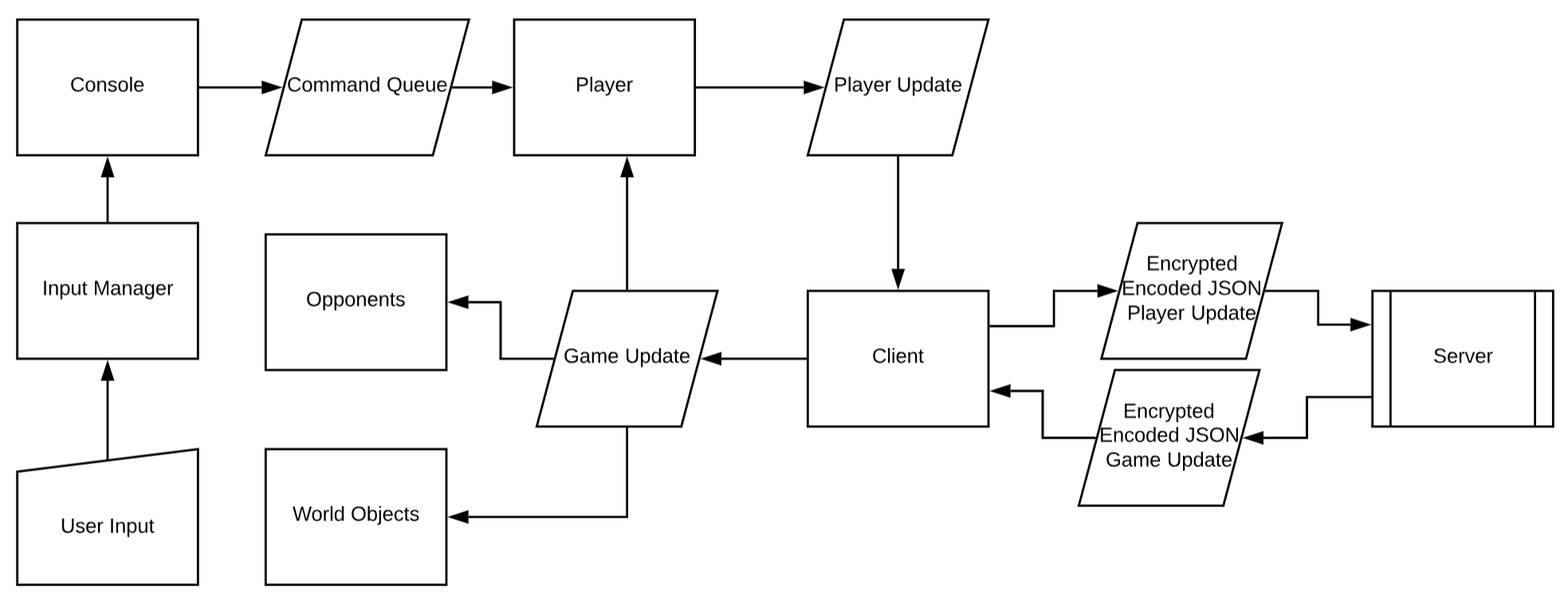
After processing all updates for the tick the Server calls a function that returns the GameState as an object that includes the list of players and a list of splats to be applied to the world. The server also calls a function that clears the list of splats.

### Section 5.2.4. Client Input and Player State

The client has a component called Input Manager that detects when keys are pressed and translates to the appropriate command. The console processes these and puts them in a command queue. Every update cycle these commands are executed on the player player object using the unity engine. The change in player state is recorded and sent to the client to be sent to the server.

Everything up through most of the Command Queue was created by our game developer, Mitchell Gentry. He gave us the tools needed to convert key presses into actions. He also coded player movement and set up the Unity RayCast for player shots.

We then added to these functions to send off updates to the Server where appropriate.



*Diagram: Control flow of information on client*

### Section 5.2.5. Client’s Player Update and Server’s Game Update Communications

Upon game startup the following actions take place:

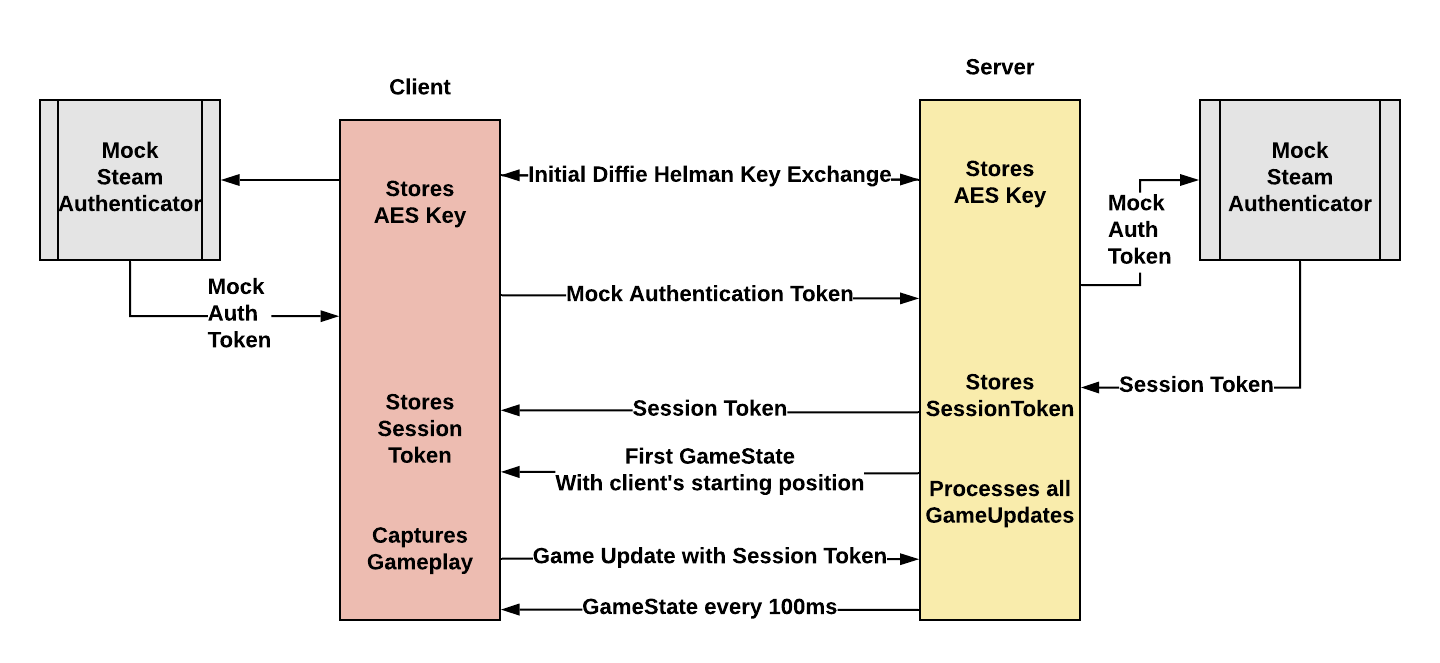
1. The Telepathy Client connects to the Telepathy Server
2. A Diffie Hellman key exchange creates the AES keys that are stored on the client and server.
3. The client sends the Steam Authorization to the Server
4. The server:
   1. Verifies this Authorization Token
   2. Generates a new Session Token
   3. Adds that player to the PlayerList in GameState
   4. Randomly Generates starting coordinates and a color for that player
   5. Sends that update to the client
5. The server then starts sending the client regular GameStates
6. The Client starts sending the Server PlayerUpdates

The goal of having all game related communications fitting one JSON object poses a significant development problem. As such, a number of switch statements are employed to account for each type of update, and the fields are populated appropriately based on which enum is found.

## Section 5.3. Security Implementations

The main theme of security comes down to two core concepts. The server must take no actions at face value and must establish well defined protocols. Trusting user input opens an attack vector for players manipulating the GameState and inconsistent protocols make holes in communication.

### Section 5.3.1. Authentication

****

*Diagram: Authentication Process*

**Achieved Security Properties**

The server will ignore updates from the client if they have not properly authenticated. Right now this token is hardcoded into the client and must be sent to the server after initial connection is established.

After sending the token the client will get back a SessionToken. All future updates must have this SessionToken attached or the server will ignore them

**Known Issues**

There is no real authentication. It’s just sending a integer hardcoded into the client’s code. The plan is to expand the authenticator to use Valve’s SteamWorks API.

### Section 5.3.2. Encrypted Communication

**Achieved Security Properties**

Plan: All GameState updates will be handled in private variables stored on the user stack. An attacker could see the object class if they decompile the Unity executable, but preventing that is beyond the scope of this project. We will encrypt each packet using AES 256 symmetric encryption. This was originally believed to not be fast enough, but ended up being sufficient. The keys used for the AES encryption are exchanged using Diffie-Hellman key exchange when a client connects to the server.

**Known Issues**

It may be possible to execute a man-in-the-middle attack on the server when exchanging keys.

### Section 5.3.3. Server Verification: VerifyPosition

**Achieved Security Properties**

Every time a player tries to update their position, this function is called. It takes in their current position, the timestamp of that current position, and their new position to determine if the change of position is within a tolerance.

Given that players move at a maximum velocity, even when falling, we can form a “perceived velocity” based off the change in position divided by the change in time. This perceived velocity is calculated on the 2d plane and separately for the vertical plane.

If the player violates the maximum 2d or y velocity they enter an invalid state. It can be observed that we have a flag for InvalidState and toBeKicked. This was done out of consideration for the fact that my verifier and Unity could be imperfect. Either a packet could be lost or Unity could send an improper update, and we want to set up a framework that doesn’t immediately punish the player. For now, however, both flags are set to false.

If the player is in an invalid state, their update is ignored and they are set to be kicked. We set it up so that the server itself is responsible for adding or removing players from the game, so the server, after calling the game updater, checks to see if any players are in an invalid state and removes them from the game.

This means that a player will never teleport to an unauthorized place in the game and be able to make any meaningful impact on the game. On the client they are still left in the game, so they can keep moving around, but they have been disconnected.

**Known Issues**

The client still moves the player regardless of how valid the input is. The server will immediately disconnect them if invalid, but they will get a snapshot of the map as it was before the game booted them. The game does stop updating and they can’t give any input, but the game needs to go back to the welcome screen on disconnect.

### Section 5.3.4. Server Verification: VerifyHit

We rely on Unity to do the raycast for the player interactions, but have a rudimentary check in place to make sure that action is performed within reason.

We know both player’s position and the shooting player’s orientation, so we can perform some math to see if the line extending from the player comes within contact with the cross sectional area of the other player’s ball.

The orientation, characterized by Euler angles, is converted into a unit directional vector. We then take the center of both players and subtract them to find the line extending from the shooter to the receiver. We then apply the line-point distance formula to work out the distance between the line extending from the player and the other player’s center.

**Known Issues**

Unfortunately our math in this calculation is imperfect, and testing led us to implement very large tolerances since valid shots were coming back with distances ranging from 0 to 4, which is well outside the expected max of 0.5, which is the player’s radius.

This verifier, then, is more a demonstration of intent and technology and will require substantially more testing and refinement before a strict tolerance can be set.

We also chose not to take action against the player for sending an invalid hit. The shot is simply ignored if it’s invalid.

# Section 6. Security Evaluation

This security evaluation took place before many of the security features were implemented.

The overall security objective is to encourage legitimate gameplay by making it inconveniently difficult to spoof the server into making an unauthorized update to the GameState. A secondary objective is to maximize reliability by limiting the ability to crash the server by input alone.

For security testing the developers programmed a back-door console into the Unity game. Anyone is able to open the in-game console and send strings to the server directly. This was done intentionally to bypass the trouble of making another program on the side to send messages to the server. Please note that this backdoor will be removed when the game deploys.

## Section 6.1. Security Evaluation from the Team

The first objective is met by strict adherence to a fixed protocol while encrypting packets to minimize the opportunities to learn this protocol. The server seeks to guarantee that it will only act on packets that match a very specific JSON format. All other packets will be ignored, not evening receiving a response from the server.

The rationale behind this this is security through simplicity. There is no reason for the client to send anything outside of the narrow definitions of this JSON object. And if the server tries to parse and respond to every packet then the floodgates are opened to an infinite wealth of attacks.

### Section 6.1.1. Areas of Possible Weakness

Right now there is no client authentication. It’s possible for anyone with a terminal to send the server packets with either their own generated client id or one that matches any id on the server. The server is happy to accept those packets and update the GameState given that the data fits the correct update object. This is a glaring problem that is discussed more in 6.1.2, since it is out of scope without Steam integration.

The server has not been rigorously tested for concurrency. It may be possible to break data structures.

The update verification software ranges from non-existent to its infancy. It should be extremely easy to spoof the GameState at the time of writing.

Since there is no reliable way to verify client authenticity, it’s very easy to prod the server using a range of ints for other client ID’s until the adversary finds the ID of another player through trial and error and then can freely send updates for that player. It would only take us, as developers, a few seconds to figure out how to do things like move opponents to our location on the map for an easy point.

Telepathy itself has not been tested for security. Testers are encouraged to prod possible weaknesses in the Telepathy TCP protocols. The ManaCraft developers have free access to the Telepathy code, so security holes can be patched.

### Section 6.1.2. Areas Out of Scope

This game has turned into an active workbench, rather than a game being staged for release. As such, it will never see the light of day on Steam. That means there is no reliable way to verify the authenticity of clients. Once the development key is worked into the current build it will become possible to simulate the task of cracking this key, but without Steam the solution is imperfect.

Attacks on the server hardware and software outside of ManaCraft are out of scope. Testers are encouraged to try crashing the server by direct attacks, but breaking into the network the server’s listening on through side channels is out of scope.

### Section 6.1.3. Within-team Peer Audits

#### Section 6.1.3.1. Andrew’s Client Analysis

At the point of review there is no testable prototype, so everything was done by reasoning through the code. Every mention here will be addressed in future updates.

The client is off to a great start. The foundation is in place for only having one way to send packets to and from the server, which provides a choke-point for logging verification. A big consensus within the security field is treating the client as if they have complete control over everything on their system, which means that only having one way to get into the client shouldn’t be the only security measure, but it’s still good design.

Right now there is no way for the client and server to exchange authentication tokens or player names. This is a huge vulnerability that must be addressed immediately. After connection the server needs to send the client a unique ID so that the client knows what updates belong to what objects.

The client is currently configured to send an update to the server on every game input. The input is currently being processed by offloading a buffer of commands into the update routine at a fixed interval. Sending an update to the server on every commands creates a significant amount of overhead in a very short amount of time and could potentially cause packets to get mixed up if the server receives them out of order. Instead, the client should track the game updates as the console is parsing instructions and send updates when the buffer is empty. This will be done by having an update for movement (position, velocity, and orientation), another update for jump, and a third update for “interact.” Each one being sent only if there’s an update to be made.

#### Section 6.1.3.2. Keet’s Server Analysis

We have made some good progress on our server. Currently, the server is able to accept changes in the player states from the clients and update it’s internal game state using these updates. The server does cannot yet verify that player updates are valid. This means that a client could change it’s position to teleport around the map if it wanted to. Player update validation will be implemented in the future using physics equations as a model to calculate approximately what state the player should be in.

Right now the server broadcasts the entire state of all players to every player. This could be a problem in the future since players may want some of their traits to be secret from other players. One such trait is player position. In a competitive game we do not want players to know where the other players are unless they are in the player’s view. We may look into this problem in the future if time allows.

Another issue along the same vein is that the server broadcasts the entire game state every tick. This is a lot of information to be constantly sending and may cause some latency in the game. Eventually we are planning to either only send info that has changed or only send info that is within a certain distance of the player.

## Section 6.2. Response and Revisions Following Peer Analysis

### Section 6.2.1: User input into game console

We originally had a console that allowed us to send commands to the server directly. This would allow us to test whether certain inputs are accepted by the server. However, we had no checks for permission or any sort of filtering. This could be a major security flaw if malicious users were to use it to send commands to the server. We decided that users do not have any legitimate reason to use the console other than to cheat, so we decided to remove the functionality for now. If we do decide to implement it again, we will only use it to debug. The console will not make it into the final product.

**Affected Code:**

LocalConsole.cs in Unity. We commented out the send function, so the user can currently type on the console, but nothing will be sent.

<https://gitlab.cs.washington.edu/andyham/manacraft-server/issues/4>

Section 6.2.2: JSON injection

This was the primary objective for this week’s development. The prior focus was to “just get something working” which resulted in a messy solution that was neither secure nor scalable. The client and server were just communicating “gamestate updates” and anything else was treated as a corner case. We realized that a game server has to do so much more than just update the game. Things like lobbies (playing with just your friends), security, and name changes were not possible.

The game update framework was greatly expanded to include an arbitrary amount of update types. This is the system we have in place now.

* When a player connects (controlled by our Telepathy tool) nothing happens, the server does nothing.
* The client then calls a method to simulate reaching out to the SteamWorks servers for an authorization token. Right now there is no SteamWorks integration, so a hard-coded integer is returned. The client then sends that to the server.
* The server ignores all packets from the client until they send an authorization token. Once received, the server simulates a verification method to confirm it’s valid, then makes a new player, establishes a new session token, and sends that session token back to the client.
* After the session token is sent to to the client, the client is added to a pool and will receive game state updates from the server

Here are known issues

* There is still no SteamWorks authentication, so someone reading the code can get this hard-coded test token and still spoof a client.

**Affected code:**

Almost every class in the server and client were impacted by this. We conducted almost a total rewrite.

<https://gitlab.cs.washington.edu/andyham/manacraft-server/issues/3>

Section 6.2.3: Player floats when moving off of an object higher than ground level

This bug occurred when the player attempted to jump on top of the red box. If they tried to move off of it, they would fly on the same plane as the top of the box. We do not want players to be able to cheat by flying, so this was an issue. The reason for this bug is that we were not applying gravity when the player move in the x and z directions. This was an easy fix as we just needed to apply gravity whenever the player moved.

**Affected Code:**

FreeLookControl.cs in Unity. ProcessGravity is now called every update cycle.

<https://gitlab.cs.washington.edu/andyham/manacraft-server/issues/2>

Section 6.2.4: Camera angle view is orientated based on the player, there is not any standardized view of the world.

This error is a result of mechanism used to attach the player’s camera to the player object. It has been in a fixed position and freely rotates as the player’s orientation rotates. This is a game design and development decision that doesn’t affect the networking infrastructure, as such, fixing it was not made a priority. Regardless, the issue was fixed and camera orientation locked.

**Affected Code:**

FreeLookControl.cs in Unity. Camera Angle has been Locked

<https://gitlab.cs.washington.edu/andyham/manacraft-server/issues/1>

Section 6.2.5: Lack of Server-Side GameUpdate Verification

At the time of review there was no verification. Now there are verifications for both the player position updates at the player shoot update. See the associated Security Implementation sections.

## Section 6.3. Summary

Here is a short list of security features

* All communication happens with AES 256 encryption
* Server ignores input from clients that are not in the game.
* In order to get in the game, the client must send an Authentication Token
* Communication happens through one JSON object
* Input is ignored if the object does not fit very specific formats
* In the event of a rejected object, the server does not respond
* All position updates are tested for reasonability
* Players are kicked if their perceived velocity is too high
* All shot commands are verified to ensure player1 is reasonably facing towards player2
* Invalid shot commands are ignored

# References

<https://unity.com/>

<https://github.com/vis2k/Telepathy>

<https://partner.steamgames.com/>

# Appendix A. User Centered Design

## Appendix A.1. Methodology

This activity was based around the Security Cards and took place on the internet. The idea was to tap into the game development and security community on Reddit to gain insight and understanding from people of all skill levels who share a similar passion for online games.

A survey that is inspired by the security cards was posted on two subreddits: r/asknetsec and r/gamedev. The survey consisted of four sections, one for each category of security card: Adversarial Motivation, Adversarial Methods, Adversarial Resources, and Human Impact. In each section, participants were asked to rate the relevance of each security card as it relates to a gaming server from 1 to 5 where 1 is least significant and 5 is most significant. Participants were also encouraged to share their thoughts about game server security in the comment section.

The objective was to find several different subreddits where these communities congregate and create several posts that ask thought-provoking questions. Passionate, experience, internet users are often anxious to share their views on the technology they use. We hope to utilize this to gain insight for both the gaming elements and security elements that make sense for this project.

### Appendix A.1.1. General Theme

The general theme of this activity was to gain perspective from different groups of people about what security considerations they find most relevant. Game developers and security-minded individuals typically hold strong opinions about the games that they make or play, and are eager to share that opinion on the internet. By creating a Reddit post with a survey inspired by the Security Cards, we were able to gain some more insight on important security considerations that we must look at while designing our game.

### Appendix A.1.2. Location and Time

This activity lasted for several days on two different subreddits.

* **Location/Time Pro 1.** Large, diverse, pool of participants. We utilized a platform that is highly trafficked by a diverse group of passionate individuals. It would be difficult to access this audience using any other means.
* **Location/Time Pro 1.** Time for discussion. People are frequently willing to spend a lot of time talking about their passions on the internet.
* **Location Con 1.** Credibility. There is no good way to identify credibility on Reddit. It’s likely the results will need more scrutiny than an invited focus group.
* **Location Con 2.** Anonymity. We did not know the demographics of the respondents and people on the internet often do not want to provide such information.
* **Location Con 3.** Uncertainty of response. There was the possibility that our posts would be largely ignored and we would not get many responses.
* **Time Con 1.** Scope of time. It’s probable that many people will post a response and vanish, eliminating the opportunity for further discussion.

### Appendix A.1.3. Expected Participants

These subreddits could be populated by everyone with all different backgrounds. But it’s not unreasonable to predict the category of participants based on the cultures of the targeted interest groups.

It is difficult to anticipate the number of participants, but, in general, Reddit is very popular.

* **Age group**. 20-40 for the game developer subreddit, 18-30 for the security students subreddit.
* **Gender.** Likely male.
* **Major/Occupation**. We wanted to include a wide range of majors and occupations. Everyone has unique insight regardless of background
* **Other common interest.** Gaming and security in general, not just online gaming.
* **Prior knowledge about your topic**. Everyone had experience interacting with the type of system that we will be building. They had their thoughts and opinions on the strengths and weaknesses on both the usability and on the security of the systems
* **Tech use.** The aim is to acquire insight from a wide range of people. Users who have little knowledge of the underlying tech will have opinions on the effectiveness of what is ultimately delivered to the end user, while other users will have
* **Other demographic information.** Again, the audience is widely diverse, which greatly enhances the quality of the results.

### 

### Appendix A.1.4 Participant Interaction (Reddit Thread Conversation)

The two subreddits focussed on were r/gamedev and r/asknetsec. For each subreddit, the survey based on the security cards was posted and users were encouraged to comment on their other thoughts.

**Schedule.**

* **Thread instantiation** (10 minutes). Create subreddit posts.
* **Wait for replies** (0 - 36 hours). We will engage in conversation for the next day and a half.
* **Conversation** (time appropriate). Ask questions to expand on the participants initial response to gather more insight.
* **Total time.** 37 hours.

**Props**. none.

### Appendix A.1.5 Budget, Materials and Other Logistics

**Materials**:

* Computers for Reddit usage.

**Logistical help:**

* **None.** Students are familiar with the mechanics and usage of Reddit.

**Budget:**

* **None.** Students have the resources they need

## Appendix A.2. Results

This section contains the the results of our online Reddit activity. This includes analysis of the data collected and how that will affect the project as a whole. The posts can be found using the following two urls

<https://www.reddit.com/r/gamedev/comments/anv5mx/security_survey_for_a_senior_computer_engineering/>

<https://www.reddit.com/r/AskNetsec/comments/anuvr6/security_survey_for_a_senior_computer_engineering/>

## Appendix A.2.1 Results: Participants’ Demographics

As was mentioned previously, it is difficult to collect the demographic information of people online. It must be asked for explicitly and there is no way of knowing whether it is accurate or not. We decided to not ask for demographic information so as not to scare off potential survey takers. We figured that since we cannot verify the information, it was no big loss.

Another problem that was encountered was the small number of responses. The total number of survey responses we received was 8. While we we knew it was possible that the number of responses would be low, we were hoping to get a large opinion pool to get a better sense of which security considerations were most important to the majority of people.

## Appendix A.2.2 Methodology changes

We originally were planning to post to a larger number of subreddits. Unfortunately, we ran into an unexpected roadblock. Many subreddits have specific rules on who can post and what can be posted. A common restriction was that users had to have above a certain amount of positive karma (points obtained from upvotes from other users) to post. r/security is an example of a subreddit we wanted to post to, but had this restriction. Since neither of us had ever posted anything on Reddit, this was a problem. Therefore, we had to find subreddits that were more lenient.

## Appendix A.3. Analysis

This section will analyze the responses to the security card inspired survey.

* **Adversarial Motivation** The general consensus among participants was that Access and Convenience, Desire of Obsession, and Malice or Revenge were the most likely motivations. This makes sense because a person who is most likely to attack a game server is someone who hates the game enough to have the desire to ruin the enjoyment for other people. This is even more likely if attacking the server is convenient for the adversary. The motivations that people thought were least likely were Self Promotion, Religion, Politics, Diplomacy or Warfare, and Self Defense. Most of these make sense because they are not really relevant to video games. Self Promotion and Politics are surprising though. I could see an attacker trying to take down a game server of a game that has politics they disagree with. An adversary could also try to attack a game in order to make a name for themselves as a hacker. The most contested categories were Money, World View, and Curiosity or Boredom.
* **Adversarial Methods** The agreed upon methods that were most likely were Technological Attacks, Processes, Multi-Phase Attacks, and Manipulation or Coercion. This makes sense since technological attacks, process attacks, and multi-phase attacks can all be used to attack the integrity of the game while users can be manipulated to give up personal information in games. Physical Attacks and Indirect Attacks were agreed upon to be the least likely adversarial methods. This is because the server hardware is often in a place that adversaries do not have access to. Attack Cover-up was the one contested category in this section. I think this is mainly because this category was not very clear what the method entailed.
* **Adversarial Resources** Impunity, Time, and Tools were the agreed upon to be the most likely adversarial resources. This is probably because of the stereotype that hackers are people with no life on their computer in their parents’ basement. It was also agreed upon that Inside Capabilities and Inside Knowledge were potential threats, but not the most likely. Power or Influence was agreed to be the least likely threat. This makes sense since attacking a video game does not provide much power. Money, Future Resources, and Expertise were the contested categories.
* **Human Impact** The most likely human impacts that were agreed upon were Emotional Well-being and Personal Data. These both make sense. Personal data can be vulnerable to attacks in video games (e.g. phishing). Emotional Well-being can be affected if people are not able to play the game they enjoy. Physical Wellbeing was agreed to be least likely to be affected since you cannot get hurt physically by a video game. Financial Well-being was the most contested category. This makes sense since some people may have been thinking that the game includes microtransactions. In this case, attacks such as stealing in-game items may hurt financially. This is not the case if the game does not contain microtransactions.

One aspect of security that was a common theme among the responses we received was in regard to the information contained in the server packets. These server packets contain information on the game state. Some users warned that if the client gained access to certain information contained in these packets, they could use it maliciously. For example, a player who has access to the positions of their enemies gains an unfair advantage. This is an issue that we must look into.

## Appendix A.4. Project Implications

* **Security Goal Impact 1:** We must be careful of the information that we send in the packets between the client and server. This goes back to the client-server communication security discussed in section 3.4.3. It is not only unauthorized clients we cannot trust. We also cannot trust authorized clients with certain game state information because they could use it to cheat.
* **Design Impact 1:** We must find a way to send only send the Client game state information that cannot be exploited. This is an addition to section 4.1.1. Relying on the Unity client to do the heavy lifting of things like physics calculations opens up an attack vector. “Player 1 performs this action to that object” has a lot of implications. It’s easy for malicious users to perform actions like speed hacks to trick the server into accepting out-of-bounds input.
* **Design Impact 2:** An interesting idea for limiting vulnerability is to limit client exposure to the GameState. An simple example is “does every client need to know the location of every player?” Valve’s game, CounterStrike: Global Offensive, goes to great lengths to only show players that are within line-of-sight of each client. It then is logical for us to limit GameState when pushing to clients. The extent of this limitation has to be explored further.
* **Security Impact 1:** Over the course of this discussion someone brought up the distinction between sound and sight. We need to decide when sound events are triggered, and that has to happen differently from sight. For example, a client could hear footsteps from around the corner but the exact coordinates are refrained to prevent the client from maliciously collecting that data before it’s drawn on the screen by the default client.

## Appendix A.5. Going Further

## Appendix A.5.1. Security Vs Effort

One response on Reddit brought up a very interesting idea. A perfect game server has to be future-proof. That is, it has to be immune from unknown tools of the future.

The whole idea of immunity-from-future-tools is a paradox that security professionals deal with on a daily basis. It’s impossible to perfectly predict the tools that adversaries can deploy in the future. Moore’s law implies that future tools will be exponentially more powerful than today’s tools, and history proves that.

So an effective game server cannot be implemented under the pretense that a quick-fix will be everlasting. There are permanent security professionals on staff at every game studio for a reason. This project has to think about the baseline, but acknowledge that the solution is far from permanent.

## Appendix A.5.2. An Online game, PWN Adventure

One user on Reddit suggested looking up PWN Adventure for inspiration on how not to write a game server. PWN Adventure is an MMORPG (Massively Multiplayer Online Role Playing Game) that is designed by security professionals to be purposefully vulnerable to a large number of common attacks. To get a better idea of possible attacks on our server, we could study this game to see what attacks its vulnerable to and why.

## Appendix A.5.3. 3rd Party Clients

It’s surprisingly common these days for enthusiasts to create 3rd party clients that replace the default client. These clients replace ALL of the server-client communication with a separate client that interacts with the user. The discussion on Reddit stressed the importance of persistence in protocols. We only, truely, have control over what the server does and what the server sends to the client. Everything the client does is up for manipulation. That being said, it’s important to promote healthy playing-habits by providing a secure client that works, but it highlights the importance of not trusting client to relay reliable, critical, information to the servers. The ultimate implications is, of course, limitless… There are game studios that will be fighting third party clients until online gaming no longer exists, but it emphasises one important point. There is an inherent distrust that must exist between the client and server, and that must transcend every design decision.

## 