# CloudRenderVR

**Client:**
Dr. Mehmet Belviranli

**Team:**
Andrew Von Bereghy
Briana Brooks
Andrew Depke
Allyssa Rued
Walden Ruemmele

**Date:**
June 16, 2021

## Table of Contents


Introduction 2

Requirements 3

- A List of Functional Requirements
- A List of the Non-Functional Requirements

System Architecture 4

- Client-Server Communication
- High Level Architecture

Technical Design 6

- Server Side
- Client Side

Quality Assurance 11

- Client-Server
- Server
- Client

Results 13

- Features Not Implemented
- Summary of Testing
- Results of Usability Tests
- Lessons

Future Work 15

Appendix 16

- Figures
- Terminology

## Introduction

Our client was Dr. Mehmet Belviranli. Currently, Dr. Belviranli is an Operating Systems professor at the Colorado School of Mines. He is researching using predictive motion to pre-render frames in the cloud. This would allow for untethered virtual reality streaming to work even with current network latency limitations.

Currently, there are two main forms of virtual reality (VR), these being tethered and untethered systems. Tethered systems are physically connected to a device, allowing for intensive graphics processing, but minimal mobility. Untethered systems typically use a mobile device in a VR headset, meaning that the processing power is limited due to mobile processing power but there are no mobility limitations. The goal of this project is to use graphics streaming to allow for untethered VR to run with a similar level of graphics quality to tethered VR, while retaining the benefit of unlimited mobility.

To do this, we created a graphics streaming system which allowed for an untethered system (client) to send inputs to a server and receive future frames. In order to avoid motion sickness in future VR applications, we would need to reach a frame rate greater than 90hz and a motion to photon latency of less than 25ms. For the basic implementation of our project we started with keyboard and mouse input rather than full head-mounted display (HMD) motion, which would have been pursued if time allowed.

# Requirements

**A List of the Functional Requirements**

- Users have the ability to move around in the Unreal Engine game(s).
- The stream of keyboard and mouse inputs from the client should be able to be processed by the server, which in turn should stream graphics back to the client.
- Implementation of Standard and Third-Party Application Programming Interfaces (APIs) should be compatible with ARM Linux.
- Make a consistent connection between server/client (correct interval from user input to display).
- Establish a connection between Unreal's Pixel Streaming plugin and the client software.

**A List of the Non-Functional Requirements**

- Software must have little performance demands, and be optimized for mobile platforms.
- Software must achieve >90hz frame rate minimum, <25ms Motion to Photon latency.
- Software must minimize API usage for compatibility.
- Both server and client sides of the project must be compatible for the encoding/decoding of H.264/FFMpeg video frames.
- Code must be modular and well documented (extensible for future work).

# System Architecture

### Client-Server Dependencies

Figure 1 below shows the system's technical client-server architecture. This diagram illustrates many of the lower level components such as the type of data transmission, APIs in use, etc. The diagram starts with the control logic of the keyboard and mouse. This begins on the client side that, via a transmission control protocol (TCP) socket connection to the server, sends encoded game input to the Unreal Game Engine. The server side handles the game input through the game engine, encodes rendered frames, and then returns to the client a stream of H.264 encoded frames. The cycle is complete once the client side is able to decode the frames and display to the user, creating a full circle gaming experience.

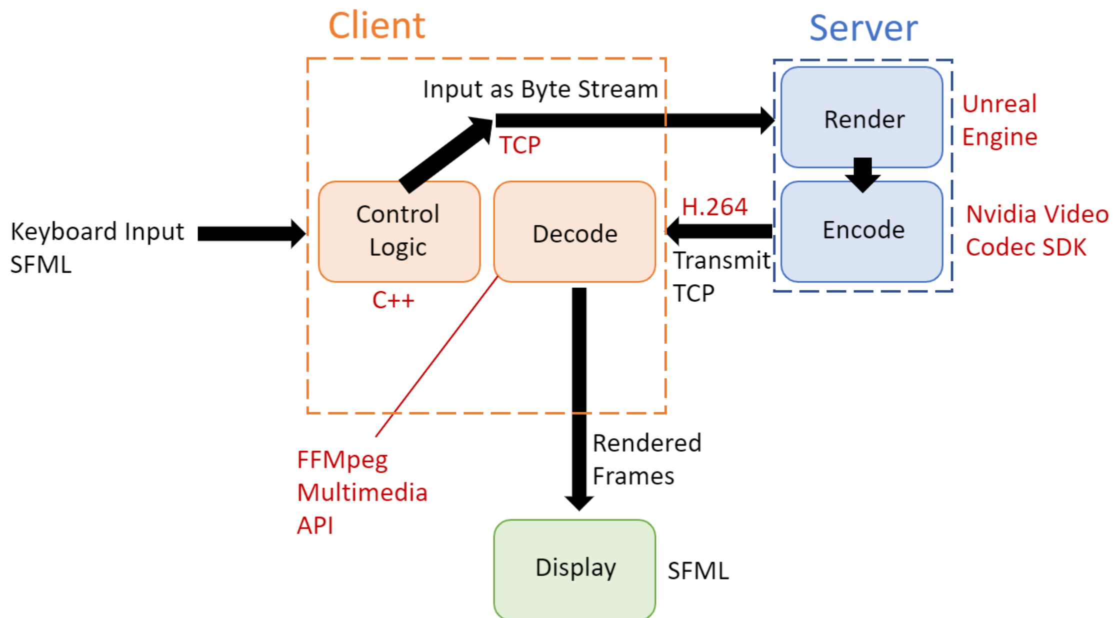



*Figure 1: Low Level Client-Server Dependencies*

**Client-Server Communication**

Figure 2 shows the design overview of how our team went about creating a system to meet the project requirements. By creating this early on in the project we were able to break the project into smaller tasks. Also, by including the future research team's goals as part of the design (predictive inputs/buffering) we ensured that our project would be easily extensible. This diagram was also created to plan how and when the server and client would communicate. For example, input was sent as it was recorded using a packet format, whereas frames were streamed in essentially a byte-stream format because they often overflowed into the next packet.

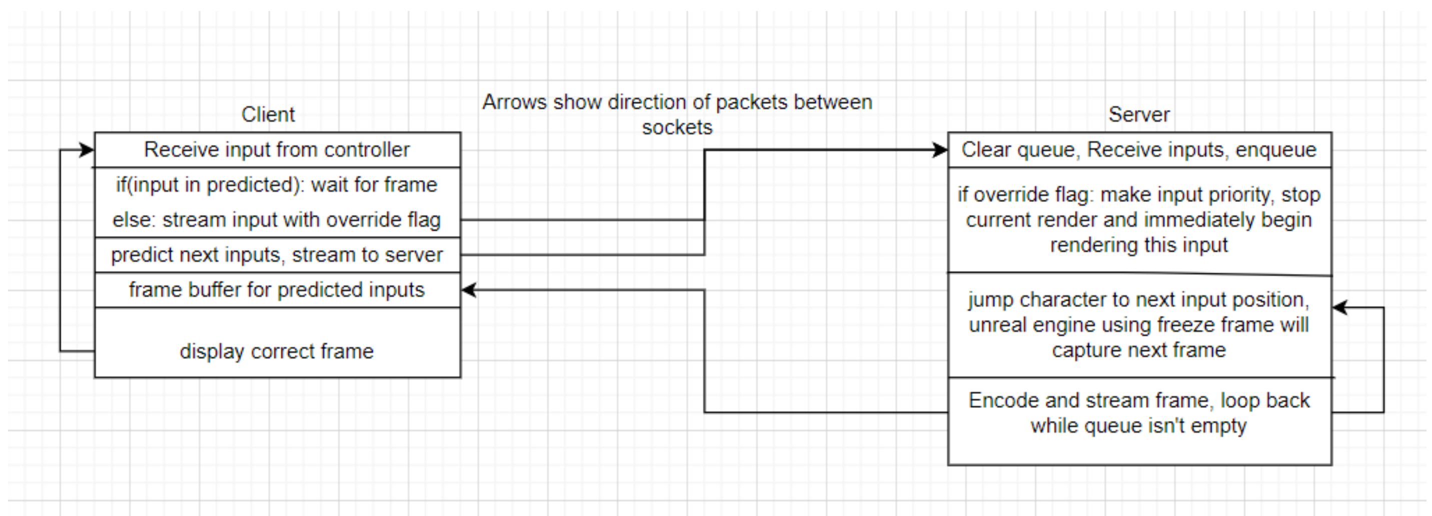



*Figure 2: Client-Server Communication*

# Technical Design

### Server Side

On the server, one of the most interesting technical aspects was the implementation of H.264 encoding, and how we used it to dramatically decrease the bandwidth requirements of our project's implementation. We initially implemented a compressed jpeg format to stream frames to the client from the server. In this version of the project each frame was being fragmented due to the relatively large size of the images, spanning anywhere from four to over ten packets. We had quickly found out that sending so many packets per frame was unacceptable, and we couldn't realistically decrease the size due to the maximum transmission unit size, so in order to meet the project's latency requirements we implemented H.264 encoding. H.264 is a lossy compression algorithm which relies on sending the differences between frames (delta encoded), rather than the entire frame. By implementing H.264, the server could not only fit an entire frame within a single packet, but it would often be able to merge multiple frames into a packet. This change dramatically improved the performance of the entire system.

However, implementing H.264 had its own unique challenges. Since it relies on delta encoding, the H.264 factory is required to wait for the next frame before it can encode anything more. This caused the speed of encoding to be dramatically affected by the framerate of the server, with a low frame rate leading to huge encoding times, for example 60 frames per second resulted in 33 millisecond (ms) average encoding time. This was a problem because the target goal of 25 ms roundtrip could not be achieved, especially if the encoding latency was greater than the roundtrip itself. To remedy this problem, the server team uncapped the framerate and optimized the Unreal Engine rendering path, turning the max framerate from ~110 fps to ~280 fps. This improvement caused the H.264 encoder latency to drop to ~5ms, an acceptable latency.

### Server Side Design Diagram

Figure 3 displays the overview of how the server side code worked. The majority of edits we made were based on the streaming functionality of Unreal engine. Because of this, our branches started off in the InitStreamer() function, provided by the PixelStreaming plugin. The left side of the flowchart shows how we captured rendered frames from the engine using hooks into Slate's rendering callbacks and the RHI ReadSurfaceData() function for extracting the color buffer. After capturing the frame it is sent to the H.264 encoder, and upon finishing is streamed to the client in a compressed H.264 format. The right side of the flowchart shows how we received user input data from the client and used this to update the game state within the engine. The two branches are related because after the game engine is updated, such as after an input from the player is made, a new frame must be sent to the client.

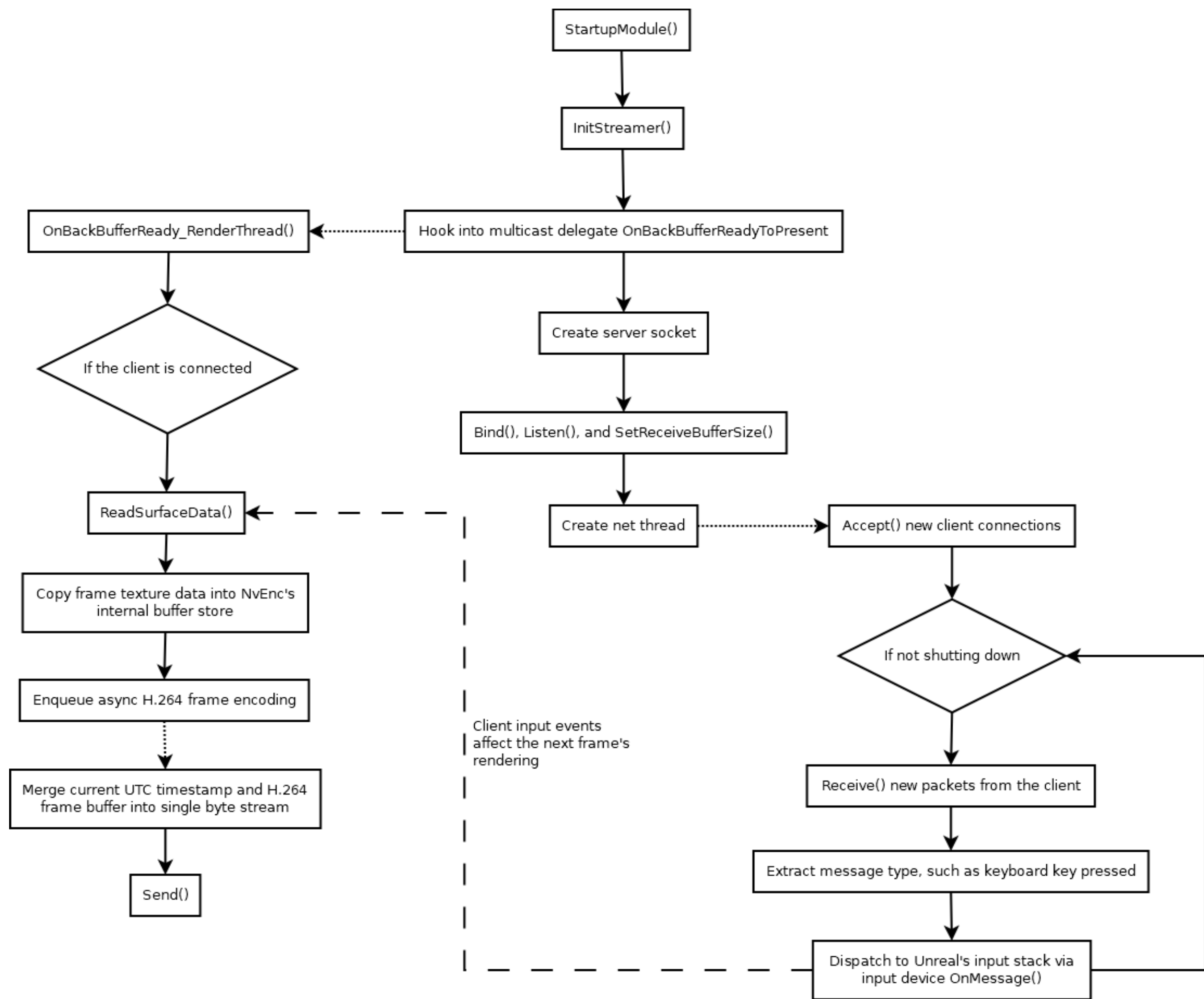



*Figure 3: Server Side Design Diagram*

**Client Side**

On the client side, one of the most interesting technical aspects of our design was parsing the byte stream data sent by the server and decoding that data using FFmpeg. We first determined that the server was planning to send H.264 encoded byte streams through packets sent via TCP socket. In order to parse these packets, we first started by creating a function in our Socket class that would parse the bytes themselves by passing the data into a buffer object, which was a vector of bytes. Then, we started by putting the header information such as the timestamp and duration of the frame into variables themselves. This information is needed to decode H.264 frames so it must be saved for later. Then, we had to determine how many frames were embedded in the received packet, making sure to feed every individual frame to our decoder to prevent visual desynchronization, or if the frame had been fragmented across multiple packets and needed reconstruction. In order to achieve minimal time spent waiting on packets to arrive over the network, we leveraged C++ futures to asynchronously send and receive packets, greatly reducing the overall client latency.

After implementing asynchronous networking with futures, the client needed to determine whether a packet contained one frame or multiple. Because each frame had its size embedded in the packet header this could be calculated. We compared the total size of the packet received over the network, without the header bytes, to the size of the image retrieved from the packet. If the image size was less than the packet size, then we knew that multiple frames were rendered and needed to be sent to the H.264 decoder separately.

This methodology was essential for the decoding purposes of our project. By using H.264 encoding on the server side, our team was able to utilize a tool called FFmpeg. FFmpeg is a collection of cross-platform libraries which allows users to decode and convert video frames. As suggested by our Client, we used the audio-visual (AV) libraries downloaded with FFmpeg and could decode the H.264 frames using the built in decoder that the AV libraries provided. The AV libraries provided functions for the software engineer to decode bytes into frames that could be used by display software. After spending a copious amount of time researching, we were finally able to use the parsed bytes directly sent to us along with the timestamp and duration of frame headings to decode the H.264 frames into YUV (a coloring encoding system) formatted frames provided by FFmpeg into BGR32 (a color encoding format using 32 bit RGB pixel values) frames using a formatting library also provided by FFmpeg.

Finally, the client team had all of the information to display the decoded frame using Simple and Fast Multimedia Library (SFML). SFML is an efficient and simple interface that the user can use to create, develop, and display multiple forms of media to be displayed and interacted with via a window. Through this, we were able to convert the BGR32 frame into a texture which SFML uses to display an object, such as a Sprite, to the Window.

**Client Side Design Diagram**

Figure 4 highlights the different components of the client side implementation. The Main function handled the SFML window, connection to the server, keyboard and mouse inputs, and rendered frame outputs of the specific functions and the three main class objects that were implemented. The inputs class held the different types of keyboard and mouse inputs which were then used by the SFML event handler in Main.

Once these inputs were received, they were sent to the server through the Socket object receive function. The Socket class would use the received packet data to parse the bytes that were encoded in H.264 frame information rendered by the server. The Frame object would use the parsed data from Socket to decode the data using the Frame object's decoder function. This used FFmpeg libraries to decode and transform the frame into decoded BGR data that could be used by SFML to display the frame into the window in Main.

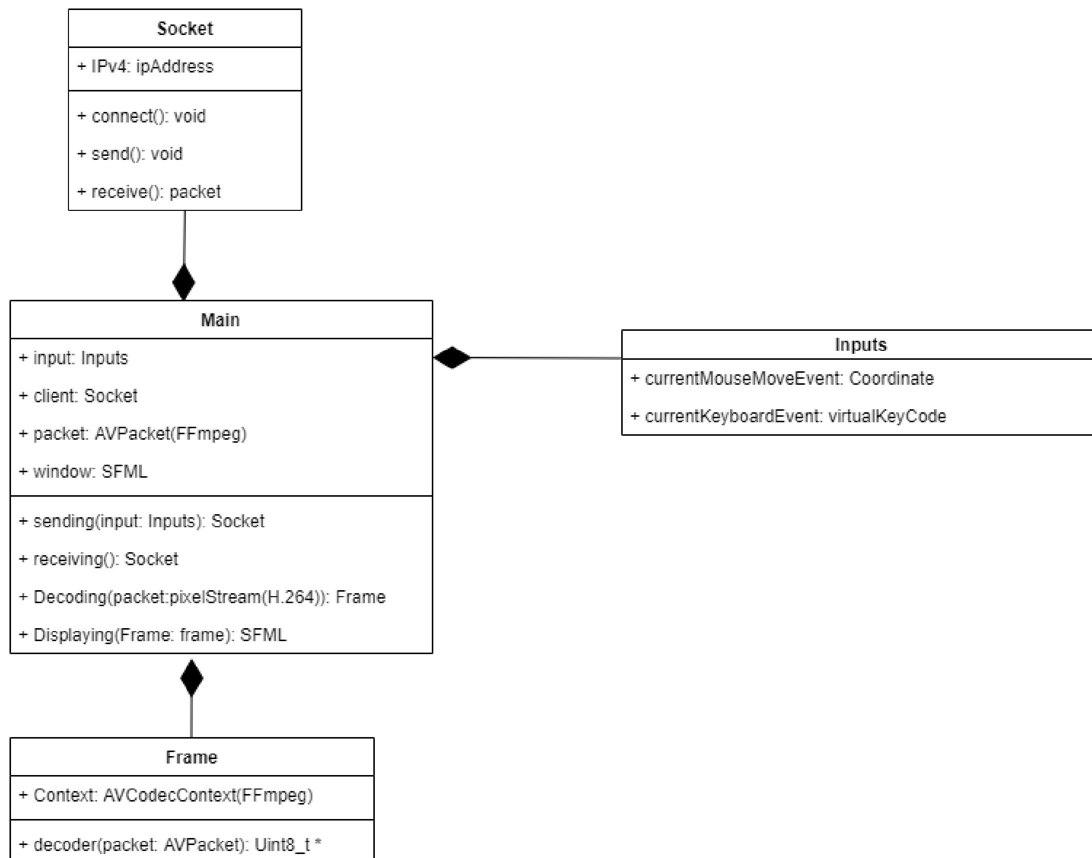



*Figure 4: Client Side UML Diagram*

**Client Side Design Flowchart**

Figure 5 outlines the client side flow of data. As the user played through the game, the input would be captured using keyboard and mouse inputs. This raw input was then converted into a virtual key code and then sent as a byte stream to the server. After the data had been encoded by the server, the client would receive this as a byte stream of an encoded H.264 frame. Once received, the client device decoded the stream, displayed the output, and continued to wait for more user input to send.

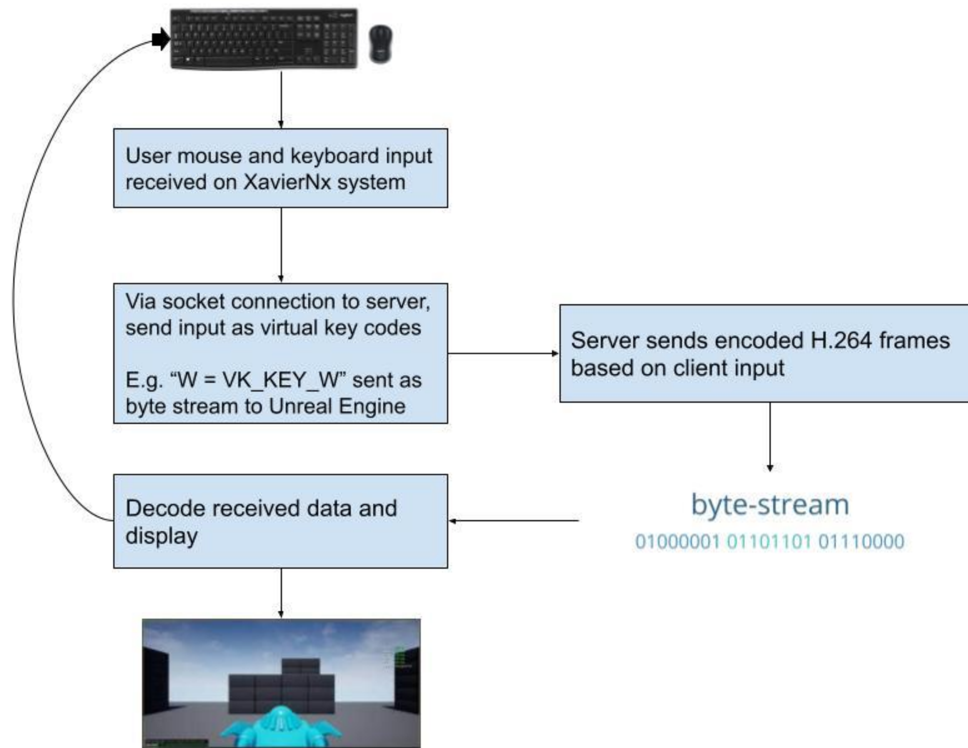



*Figure 5: Client Side Design Flowchart*

# Quality Assurance

### Client-Server

Both teams made use of pair/group programming. Pair programming helped the quality of our product by reducing the number of bugs encountered and keeping the coding process consistent throughout. It also helped us maintain a high level of coding standards which was significant for any future adaptations of our code. The client and server teams also had continued communication in order to preserve a compatible input/output protocol across both the server and the client.

### Server

Integration testing was essential to creating a working product. It was implemented by ensuring that user inputs were sent to the server and H.264 encoded frames were sent to the client. It was easily testable by running the program and checking the output logs created by print statements throughout our code. Integration testing fell under both defect detection and verification because it tested if the code worked and verified that our project requirements were met.

The server team implemented timing systems to test how fast parts of the code executed. This was an important part of the quality plan because fast response times were a key part of this project. By using many different timings, we were able to pinpoint which parts of our code executed the slowest, and then worked on making them faster. This addressed verification testing to ensure that our product met the requirements.

The server side also added various log messages to the provided logs in the Unreal Engine. This essentially replaced the unit testing implementation by printing messages for each unit rather than using assert statements. The server team found this to be useful because it meant our code was always being tested and we could easily trace back any issues that occurred during runtime. This addressed defect detection because it was easy to tell where an error arose.

Finally, by stress testing the server using JPEG images, the server team was able to determine that H.264 streaming would work as a more efficient output stream to the client.

### Client

The client team maintained documentation throughout the code to ensure extensibility for future modifications to the program. The documentation included comments in the code, design diagrams, and a requirements document. This documentation also allowed for easy debugging as we could see what we had already done and what would or would not work.

Integration testing was the main testing method of the client team. We ensured that as we developed more of the project, there would not be any bugs or damage to previous code implementations. The process included making a new segment of code, running and compiling the code, then examining the result. If the new code caused any bugs to occur, we ran print statements and logs during our debugging process to locate the bug and research what might be causing the issue. Once the bug was resolved, we continued on with that process until our code was complete. This process made sure that with each new implementation, our code continued to work with as minimal to no bugs as possible, while maintaining modularity through refactoring.

The client team also implemented a window display to verify that keyboard/mouse inputs were being processed correctly. We did this by utilizing screen share on Zoom, and viewing the server team's logs in real time to see if the packets that we sent were actually being received and parsed by the server.

Finally, the client team ensured that the libraries and APIs that we used in the project had been researched to meet compatibility/license requirements set by our Client. We also made sure that the version control was maintained for documentation and the delivery of our code to the Client would be seamless and easy by using GitHub.

# Results

## Features Not Implemented

On the server side we met all of the functional requirements, but have had issues with the non-functional latency requirements. Currently, the server side code executes in roughly 17 milliseconds (ms). However, the network latency pushed our round-trip, motion-to-photon latency above 25ms. We hoped that the future work of motion prediction would allow us to successfully reach under 25ms motion-to-photon latency.

On the client side, we did not have the time to implement the Android interface due to having trouble getting the Linux interface running correctly. Otherwise, there were no known bugs at the finish of the client side implementation.

## Performance Test

Using a combination of Tracy, Unreal Insights, and manual UTC synchronized timestamps, we were able to measure the average latency of various components in our end-to-end architecture as seen in Figure 6. Overall, we found that a significant portion of our latency was coming from frame encoding on the server side, largely due to H.264 not being able to finalize the current frame until the next was available in the queue. We also were able to measure our end-to-end latency being about 17 milliseconds, excluding the networking round trip time.

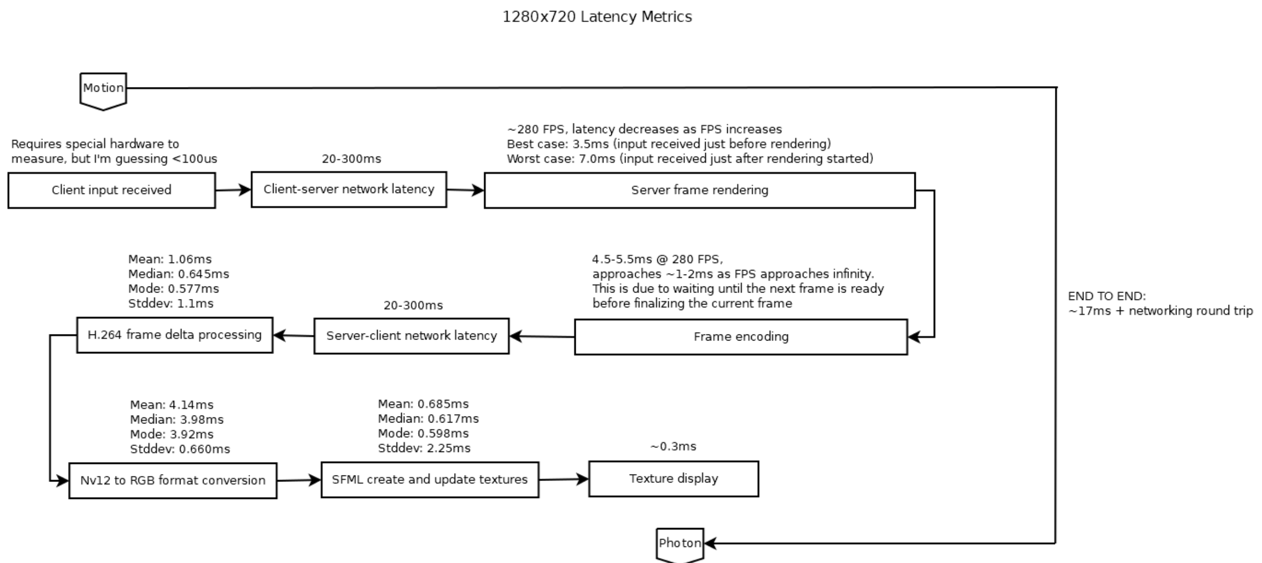



*Figure 6: Performance Metrics*

**Summary of Testing**

The existing solution we have implemented is very playable for desktop applications, but it will likely induce motion sickness in virtual reality rendering due to the high latency. To tackle this, we were currently looking at optimizing H.264 encoding on the server, and replacing the Nv12 to RGB format converter with something that could better leverage the hardware, ideally via GPU compute.

The client side to server connection is consistently working and the inputs, received packets, decoding, and displaying are all working properly. Therefore, we have completed the main functional requirements of the project by ensuring that on the local XavierNx device, the user would be able to play a game in real time rendered from the cloud. The XavierNx was the client side computation device which was considered to be a virtual machine that utilizes Ubuntu, which is another Operating System built off of Linux.

**Results of Usability Tests**

From what we could tell, our software is very usable for the intent in which it was made. Dr. Belviranli contributed a great amount of input and approved of our design decisions throughout the course of this field session. With this in mind, we believed that the future research team would be able to extend our code quite easily due to our design and documentations, especially considering that the future research team includes members of our field session team.

**Lessons**

During this field session we learned a lot about encoding, networking, device compatibility, and latency timings. Understanding how the Unreal Engine source code worked was the key to the server side implementation. Between using render-buffers and editing source plugins, the vast majority of code written on the server was within the Unreal Engine's code. One of our greatest struggles was dealing with the complications of practical networking, especially handling packet fragmentation and reassembly. Dealing with our own custom protocols taught us how to write high level protocols, while at the same time teaching us how a router breaks packets down further, and how to parse a byte stream upon reassembly.

The understanding of project builds and library usage also became very important for this session. Compilation errors rather than runtime errors were the root cause of delay for the client side of the team. After many library implementations, the client team had to make sure that any linker issues with any additional libraries were resolved using the Makefile. Through Makefile adjustments and various amounts of research on Linux builds, the team learned about the complexities of what a linking issue was, how to locate the specific libraries necessary to include in a Makefile to resolve it, and how to follow the necessary precautions when trying to reference libraries from Third-Party APIs.

Finally, the sending of data to a server side socket required a specific protocol to be used, one that all types of keyboard/mouse input require. Using a specific byte breakdown of the protocol, we learned to send packets containing more than just the specific key press or mouse button click. The packets also contained data pertaining to the physical behavior of the player, such as holding down keys or moving the mouse rapidly. This was a powerful socket application and was great for the CloudRenderVR project.

## Future Work

A large portion of our architecture's latency was due to the H.264 encoder waiting until the N+1 frame was fully rendered and enqueued, before finalizing the encoded output of frame N. This added the full frame rendering time latency on top of the encoder, in addition to the small time spent actually decoding the frame. In the future, we would look at using a transport stream layer on top of the H.264 codec to reduce this delay. Another approach we considered was using a more latency-optimized video encoding protocol instead of H.264. We also considered replacing our custom networking protocol for inputs and video streaming, with an industry standard solution, such as RTP or SRTP. This would allow us to leverage UDP sockets instead of TCP, further reducing the network latency.

The client team would work to include a packet system protocol for human motion input in order to eventually apply predictive movement to the project. As the socket connection was very strong, implementation of human VR movement would be a challenge on the software side of things, but would not be too difficult for the packets to handle. Unreal Engine contained VR plugins as well, meaning that the engine would be able to handle this input with minimal change.

Since we were unable to transition to the Android platform due to time constraints, future project extensions would be to include cross-platform compatibility reaching beyond the Linux-based XavierNx device. We intended to reach the Android platform for compatibility testing, however optimizing latency and game runtime on the original platform was more important. By incorporating developed, cross-platform APIs into the project, we believed that porting to Android (which was also Unix based) would not be a challenge for further development.

During our time working on this project we left extensive comments and created many artifacts, such as design documents, in order to help aid the future developers. We believe that these will make the project easily extensible for future developments.

# Appendix

## Terminology

Virtual Reality (VR): An immersive experience of a game or simulation that creates a virtual environment in which the user can interact.

Cloud Rendering: A client/server communication process where the user interacts directly with the client side, and then the server is able to render frames based on client inputs.

H.264 Encoding: H.264 is a stateful lossy compression algorithm which relies on sending the differences between frames (delta encoded), rather than the entire frame.

FFmpeg: An open source collection of libraries that is cross-platform and allows users to decode and convert video frames.

Simple and Fast Multimedia Library (SFML): Cross platform library designed to provide a simple API to various multimedia components in computers.

Delta Encode: Encoding is stateful and reconstructs frames based on differences between frames (deltas).

RHI: Render hardware interface, a high level layer above the actual rendering API, for accessing the renderer in an agnostic way.

TCP: Transmission control protocol, a reliable and ordered networking protocol for sending packets. As a result of the reliability, it often has significantly more latency than other protocols.

UDP: User datagram protocol, an unreliable and unordered networking protocol meant for lowest possible latency packet transmission.

Slate: Unreal Engine's native graphical user interface API.

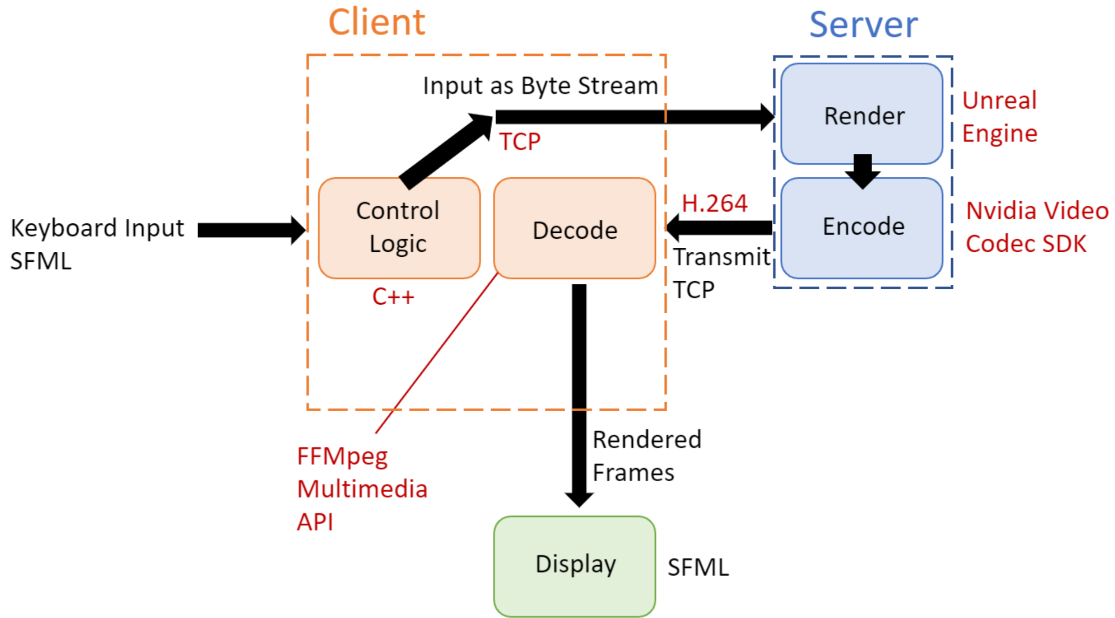
Tracy: A hybrid frame and sampling profiler for native applications, providing statistical timing measurements of the CPU and the GPU for the profiled client.

Unreal Engine: An open source game engine which we have used to implement our project.

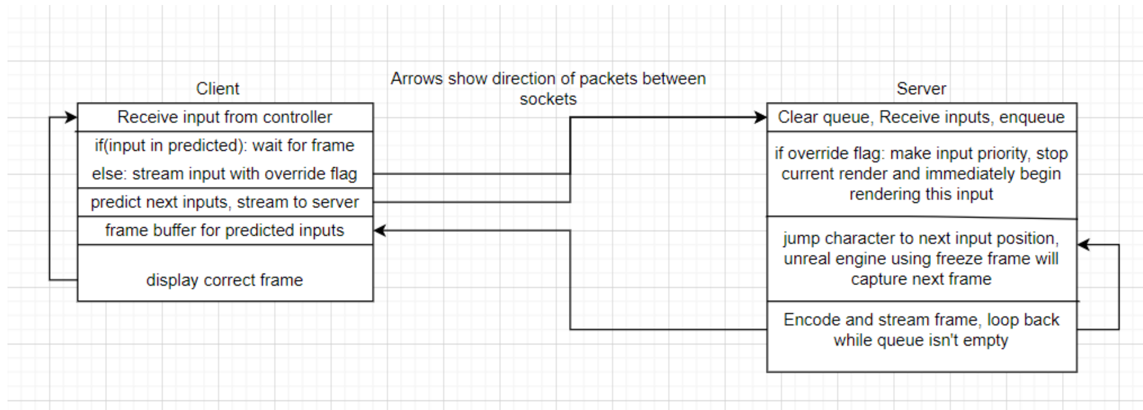
Unreal Insights: Unreal Engine's built-in profiling tools for CPU clock cycle measurements.

## Figures

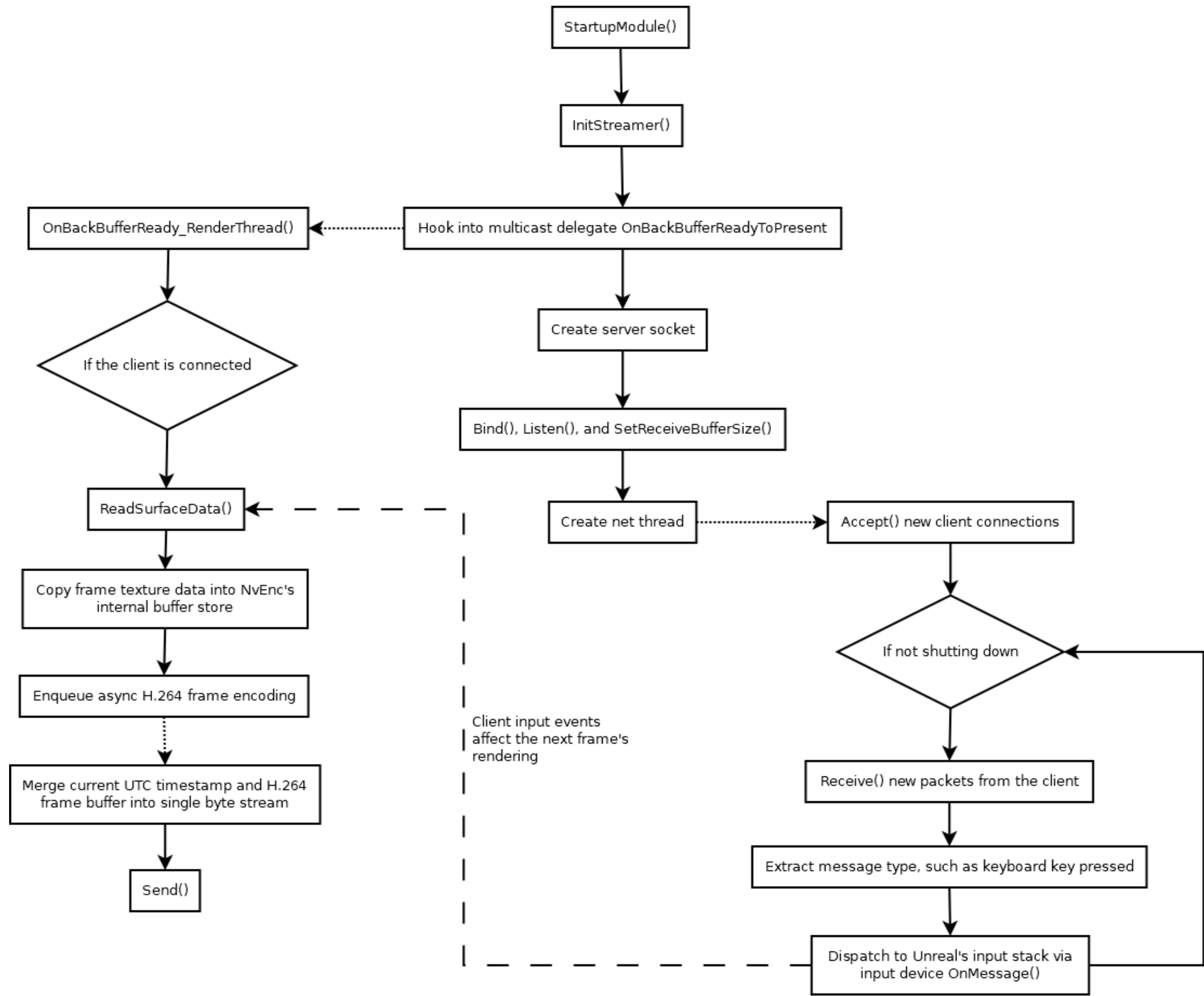
### Figure 1 - Client-Server Dependencies



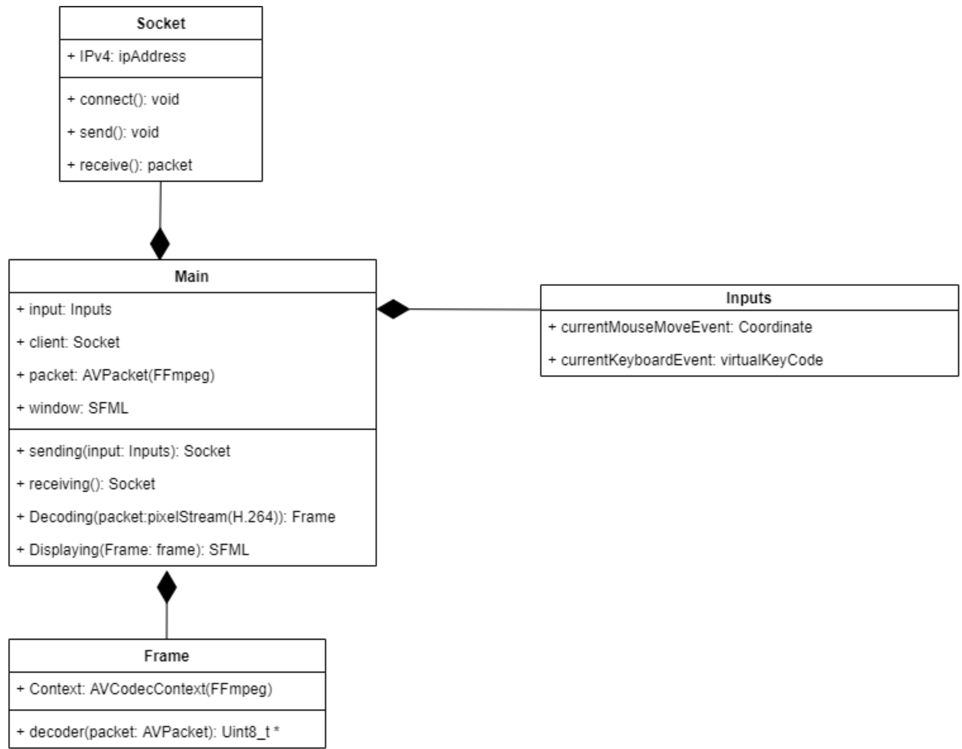

### Figure 2 - Client-Server Communication



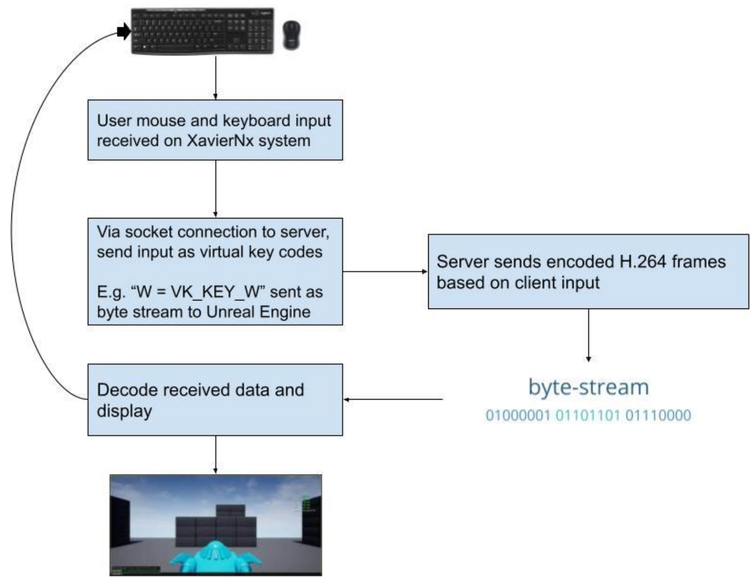
## Figure 3 - Server Side Design Diagram



## Figure 4 - Client Side UML Diagram




## Figure 5 - Client Side Design Flowchart



## Figure 6 - Performance Metrics