Low-Latency Remote Rendering and Streaming with Unreal Engine and FFmpeg: Architecture, Implementation, and Evaluation

1st Duc Long Tran
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
duc.long.tran@fokus.fraunhofer.de

2nd Louay Bassbouss
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
louay.bassbouss@fokus.fraunhofer.de

3rd Alexander Zoubarev
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
alexander.zoubarev@fokus.fraunhofer.de

4th Stefan Pham
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
stefan.pham@fokus.fraunhofer.de

5th André Paul
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
andre.paul@fokus.fraunhofer.de

6th Stephan Steglich
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
stephan.steglich@fokus.fraunhofer.de

7th Stefan Arbanowski
Future Applications and Media
Fraunhofer FOKUS
Berlin, Germany
stefan.arbanowski@fokus.fraunhofer.de

***Abstract*—This paper presents a modular, low-latency remote rendering and streaming pipeline integrating Unreal Engine 5 with FFmpeg to deliver graphics-intensive applications to low-power client devices. The system leverages headless rendering by containerizing the Unreal Engine application alongside FFmpeg using Docker, enabling scalable deployment across heterogeneous infrastructures equipped with compatible GPUs. The pipeline supports a wide range of video codecs, including H.264 (AVC), H.265 (HEVC), and AV1, as well as streaming protocols such as UDP, TCP, and SRT. This flexibility enables adaptation to different network conditions and quality-of-service requirements. We evaluate the performance and latency implications of these configurations, demonstrating how hardware-accelerated encoding, GPU memory sharing, and protocol selection affect the responsiveness and efficiency of the remote rendering experience. Our findings show that, with careful optimization, remote rendering can achieve near real-time interactivity, making it suitable for applications such as cloud gaming, virtual reality, and interactive simulations.**

***Index Terms*—Remote Rendering, Low-Latency Streaming, Cloud Gaming, Unreal Engine 5, FFmpeg, Video Codecs and Streaming Protocols, Dockerized Deployment**

I. Introduction

The growing complexity of interactive applications such as real-time simulations, virtual and augmented reality (VR, AR), and high-end gaming demands substantial computational resources [6]. Although desktop systems can meet these requirements, many emerging platforms, such as mobile devices, head-mounted displays, and smart glasses, are limited by physical and energy constraints, prohibiting high-quality on-device local rendering.

Remote rendering provides a promising solution by offloading the rendering workload to a remote server and transmitting the resulting video stream to the client device. This approach enables high-quality graphics on resource-constrained platforms and extends the reach of advanced visual experiences to a broader range of devices, especially where user comfort, mobility, and battery life are critical.

Despite its advantages, remote rendering introduces challenges related to latency and responsiveness. In interactive applications, even small delays between user input and visual feedback can degrade the experience. In VR environments, motion-to-photon latency, which is the time between a user's physical movement and the corresponding visual update, must typically remain below 15 milliseconds to avoid motion sickness and maintain immersion [1]. Achieving such low latency requires careful coordination between rendering, encoding, transmission, and decoding, as well as the selection of appropriate codecs and streaming protocols.

In this paper, we present a remote rendering pipeline using Unreal Engine 5 and FFmpeg as a configurable tool for comparing video streaming technologies. The paper first presents the state of the art, then presents the core concepts of the proposed solution. After that, it details possible implementations before evaluating them in section V, concluding with potential future works.

II. State of the Art and Related Works

A. 3D Game Engines

Game engines are the core frameworks for developing interactive 3D environments, handling rendering, input, audio, physics, and more [2]. While often associated with games, they find applications in many domains, such as filmmaking and simulations [4]. Physics calculation is especially important when considering the performance of an application, as higher scene complexity comes with higher processing demand, especially when aiming for realistic graphics.

Unreal Engine: Developed by Epic Games and first showcased in 1998, Unreal Engine went through many iterations, with the newest being Unreal Engine 5 (2022). Unreal Engine 5 supports a variety of platforms and is mainly known for its high-profile capabilities [4].

B. Cloud Gaming Services

Cloud gaming services run video games on a remote cloud server and stream the video and audio output back to the user's device. Doing so provides the user with high-performance resources on demand, enabling them to access games that they would normally not be able to on lower end hardware [7].

Providing a cloud gaming service requires significant datacenter infrastructure and low-latency networking between server and user [3], limiting current solutions to known names such as Nvidia (GeForce Now) and Microsoft (Xbox Cloud Gaming).

Pixel Streaming (Unreal Engine): Pixel Streaming is an extension provided by Epic Games that provides a remote rendering service in Unreal Engine 4 and 5 using WebRTC. By communicating with a web service, a connection can be established to the rendering unit to receive the rendered content.

C. Docker Containerization

Docker is a platform that provides operating system-level virtualization to deliver software in packages (containers). These containers host applications packaged with their dependencies, offering standalone software bundles that can be run on any system supporting the Docker Engine. Docker runs containers in virtual environments, guaranteeing their isolated execution [5].

D. FFmpeg Framework

FFmpeg is a multimedia framework for decoding, encoding and processing video and audio playback, including transcoding or transmission. It provides access to a large variety of protocols, codecs, formats, etc. whether designed by standards committees, communities, or corporations.

III. Concept and Methodology

In this section, we define the structure of the implemented remote rendering pipeline. In addition, we introduce the methodology for evaluating this pipeline to measure the impact of the pipeline on the original applications. To lead to the best user-experience, we measure the performance overhead of the system while incrementally lowering the latency.

A. Remote Rendering Pipeline

A **remote rendering pipeline** provides computing resources for rendering graphical processes at a remote location and delivers content in the form of a video stream that can be accessed through a network connection, enabling low cost devices to access complex 3D environments.

We separate the remote rendering pipeline into five key steps. We consider the **rendering** of the scene, the **encoding** of the generated frames, the **streaming** of the video over the network, the **decoding** of the video stream and the **playback** of the video

Since the rendering unit acts as a **server** providing a service to a **client** that plays the content, we refer to them as such throughout the rest of the paper.

B. Performance

In this work, we use the **framerate** as the holistic metric to evaluate the pipeline's performance.

We extract the framerate by using the delta time Δt (the time between the last frame and the current one), provided by the Tick-function inside the game thread. By calculating the reciprocal and averaging the results over a given time, we get the average framerate $\overline{f}$ [fps] of the application.

$\overline{f} = \sum_i^n \frac{1}{\Delta t_i \cdot n}$, n is the number of frame times captured.

C. Latency

The most notable difference between a live video stream and a remote-rendered application is the interactive nature of the application. K. Brunnström et al. show how in virtual reality, a disparity between one's head movement and the movement in the application can affect the quality of experience negatively [1].

In the following section, we investigate the input latency defined as the time difference between a physical action and its corresponding action displayed on the screen.

1) Measurement: We split the input latency of an application into the following sections:

1. The time for a physical action to be recognized by the operating system.
2. The time for the application to process the action and render the result to a frame.
3. The time for the monitor to display the frame.

To determine how a change in the pipeline affects the overall latency, we measure the latency for different setups and compare them with each other.

Video latency: Since we compare different pipeline setups, we isolate the second component by using synchronized timestamps displayed before and after rendering (Fig. 1). The difference yields the frame processing time. This avoids the need for an external capture device; a single screenshot suffices to measure the current video latency.

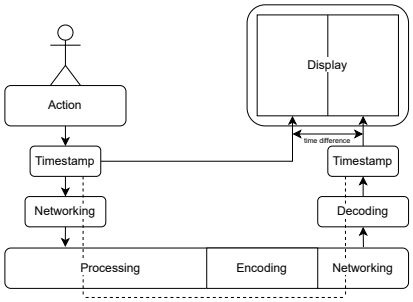


Fig. 1. Video latency with additional overhead from the remote rendering pipeline. Note that, the bottom unit represents the server while the rest represents the client.

IV. Proof of Concept

This chapter presents the implementation of the remote rendering pipeline in UE5 using FFmpeg for encoding and streaming, running on Ubuntu. We use UE5's Top-Down template and load the following modules/libraries via C++:

- SlateCore module for frame extraction through the SlateUI.
- VulkanRHI (and AVCodecCore) module and Libcuda libraries for optimized hardware encoding on Linux.
- Libavcodec and Libavformat libraries for encoding and streaming.
- JsonUtilities, Networking and Sockets for remote control.

A. Frame Extraction for video streaming

To receive the frame data from the rendering unit in UE5, we mirror the Pixel Streaming extension's approach by accessing the back buffer of the Slate application, which provides a platform-independent reference to the output texture. Note that the Slate application is responsible for rendering a UI, which is usually the last step of the rendering process. To do that we need to: 1) Access the renderer of the Slate application (FSlateApplication) 2) Define a function that processes the buffer reference (FTexture2DRHIRef) 3) Bind the function to the ready-to-present event of the Slate (OnBackBufferReadyToPresent)

The FTexture2DRHIRef is a texture reference that Unreal Engine's Render Hardware Interface (RHI) abstracts from the underlying graphics API for different platforms. This allows for operations on the graphics memory that support different graphics API such as Vulkan for Linux or DirectX for Windows.

In the following, we present three options of using Ffmpeg with Unreal Engine 5.

B. Using FFmpeg through inter-process communication

Using the FFmpeg command line tool, a process is started that takes in frame data and converts them to our specified output format. Unreal Engine 5's pipeline support then allows passing its frame data to FFmpeg. This approach requires: 1) Locating the FFmpeg process on the system 2) Specifying the encoding settings, video format and destination 3) Initializing the read- and write pipes 4) Starting FFmpeg as a subprocess using the selected settings

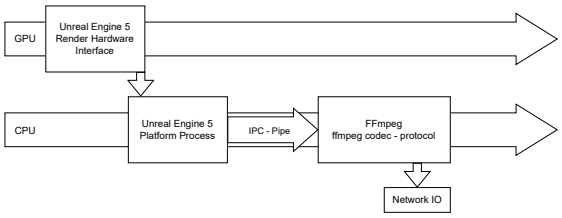


Fig. 2. Schematic for using inter-process communication between Unreal Engine 5 and FFmpeg.

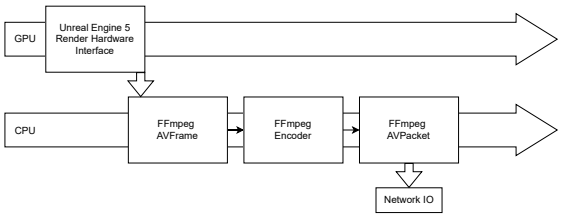


Fig. 3. Schematic for using Unreal Engine 5 with FFmpeg libraries.

To use the pipe between Unreal Engine 5 and the FFmpeg process, we first copy the GPU texture into CPU memory by using Unreal Engine's Render Hardware Interface. Afterwards we pass the texture data to FFmpeg using the initialized pipe for it to handle the encoding and streaming process.

C. Integrating FFmpeg libraries

By linking the FFmpeg library files to Unreal Engine, we access the encoding and streaming services directly without needing to communicate with an external process. This skips the need for a pipe that was used for inter-process communication and enables error handling during the encoding and streaming process (Fig. 3).

1) Frame data encoding: Once integrated, the libavcodec-library provides access to all the encoders compiled by the FFmpeg installation. Similarly to the command line tool, we use presets through a dictionary and configure the encoder based on the frames format, video framerate, and bitrate.

D. Encoding using hardware acceleration

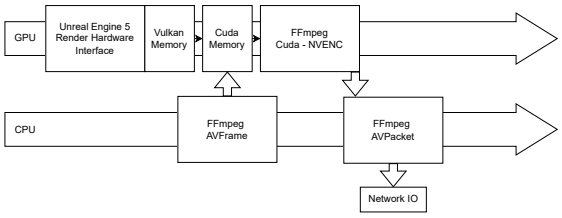


Fig. 4. Schematic for using Vulcan-CUDA-interoperability for GPU memory optimizations.

To use hardware-accelerated encoding we select an encoder provided by FFmpeg such as "h264_nvenc". As the name implies, this uses Nvidia NVENC to encode for an H.264 (AVC) stream. Note that this requires a compatible NVIDIA graphics card.

We register the texture generated by Unreal Engine's Render Hardware Interface to CUDA using Vulkan-CUDA-interoperability (Linux/Ubuntu) to provide FFmpeg with GPU memory. This avoids expensive CPU-readbacks that occur when extracting texture data to use for software encoding.

E. Dockerization of the Unreal Engine Application

Unreal Engine's documentation provides a guide on deploying runtime images using docker. This allows us to upload and run the same build on multiple devices with different computing power or network conditions. After creating our Docker container, ports and hardware devices need to be explicitly exposed upon deployment, since we are using graphics and networking components for rendering and streaming. We use **–net=host** to expose the host's network stack and **–gpus device=0** to expose a GPU to the docker container. Note that this requires NVIDIA Container Toolkit configured for Docker.

F. Evaluation using a Python remote controller

We create a **Python client** that communicates with the Unreal Engine application using a socket connection to connect, control and configure the application remotely. This allows us to measure the framerate and video latency of the application during runtime on a server.

1) Decoding and playback using ffplay: We use ffplay for decoding and playtback of the video stream. To ensure minimal latency, we run ffplay using the following command:

ffplay -fflags nobuffer -flags low_delay -strict experimental -framedrop udp:...

2) Video latency measurement using a timer: To measure the video latency, we implement a timer within a custom client as well as in the user interface of the rendering application. We send a command to synchronize the timers between renderer and client. By taking a screenshot of both the client and the video stream, it is now possible to calculate the video latency of the remote rendering pipeline by determining the difference between the timers.

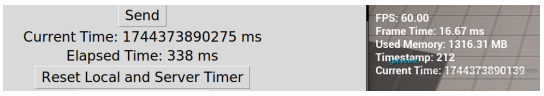


Fig. 5. Example measurement: video latency between the client (left), video stream (right). Remote client 338 ms, video 212 ms, video latency 126 ms

V. Measurements and Evaluation

In this chapter, we evaluate the implemented remote rendering pipeline by comparing different setups according to their resulting performance and latency. This allows us to determine the impact of each component presented in Chapter 4 on the quality of the streaming service.

The following results are measured using: **CPU**: 12th Gen Intel(R) Core(TM) i7-12800H (local), AMD EPYC 9124 16-Core (remote/server) **GPU**: GeForce RTX 3070 Ti Laptop (local), NVIDIA L40 (remote/server) **Codec**: Advanced Video Coding (AVC/H.264), Protocol: UDP

By streaming with a resolution of 1280x720 and disabling dynamic global illumination and anti-aliasing, we increase the performance significantly and thus reduce the error rate while capturing the frame rate.

A. FFmpeg Subprocess and Libraries

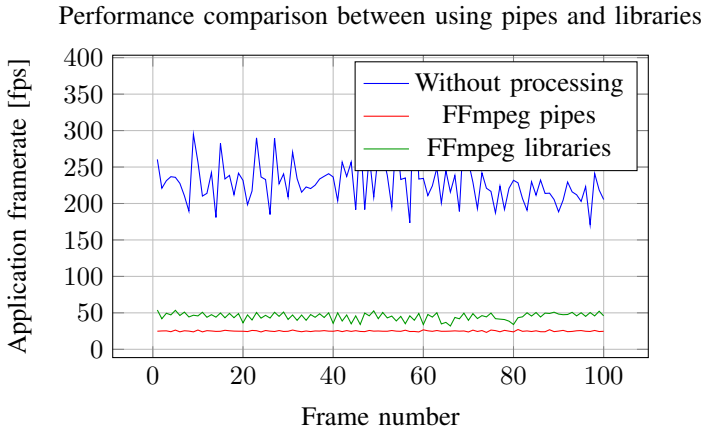


Fig. 6. Performance measured over 100 frames (higher is better).

We investigate the performance impact that results from the inter-process communication (IPC) between Unreal Engine 5 and FFmpeg using pipes (Fig. 6). Without processing, the application averaged 230 fps (4.4 ms). Using FFmpeg pipes reduced this to 25 fps (39.9 ms), while using FFmpeg libraries achieved 45 fps (22.2 ms).

Using the FFmpeg libraries outperforms using FFmpeg through pipes by around 17.7 ms (an increase of 80%). Since passing large payloads through pipes can take a long time, the size of the raw images has a significant effect on the performance. With an original average framerate of 230 fps and a total time overhead of 35.5 ms using the FFmpeg command line process and 17.8 ms using the FFmpeg libraries, we see a significant decrease in overall performance when passing frames to FFmpeg. This means that the application is running at a very low framerate considering that we already disabled the more performance-eating features of Unreal Engine 5.

B. Software and Hardware Encoding

We further optimize the application by introducing hardware-accelerated encoding on the FFmpeg side. This offloads the calculations to the GPU and improves overall performance. We measure the frame rate using the same settings as before (Fig. 7). Without processing, the baseline

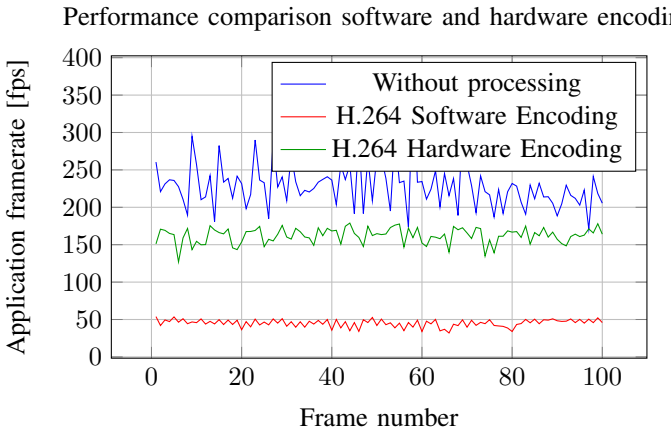


Fig. 7. Performance measured over 100 frames (higher is better).

was 230 fps (4.4 ms). Software encoding averaged 45 fps (22.2 ms), while hardware encoding achieved 162 fps (6.2 ms).

We outperform software encoding significantly using hardware encoding (with optimized memory operations) by 16 ms (an increase of 260%), with a total time overhead of 6.2 ms. The performance is significantly closer to the original performance without any frame processing in comparison to using software encoding. we achieve a time overhead of 1.8 ms for processing the frame, encoding and passing the package to the networking unit.

C. GPU Hardware

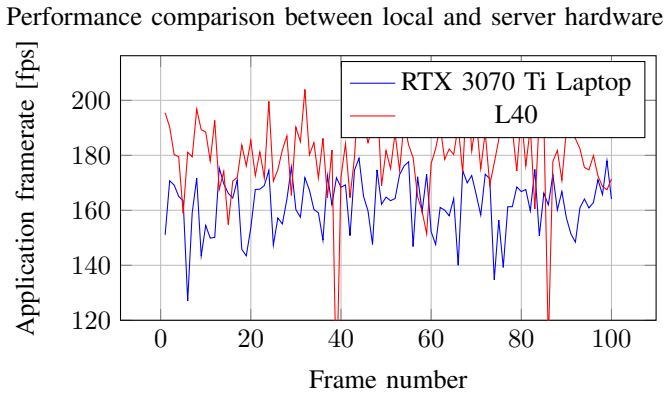
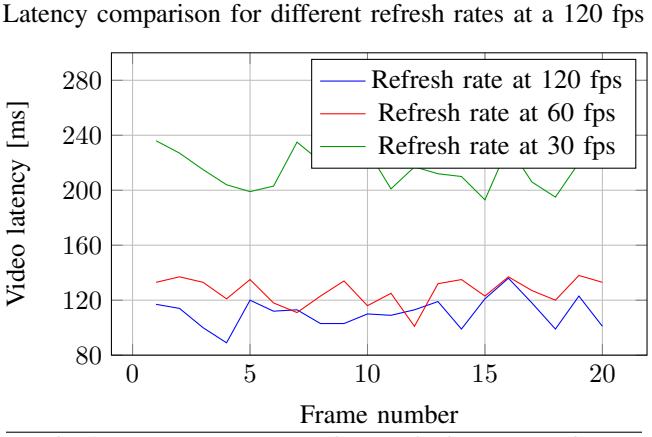


Fig. 8. Performance measured over 100 frames (higher is better).

Up to this point, we used high-end consumer hardware to run the remote rendering pipeline. By using hardware dedicated to graphical processing inside our servers, we have access to more powerful computing units while also being more reliable. We measure and compare the frame rate between the local device and the server using the optimized hardware-accelerated encoding (Fig. 8). The RTX 3070 Ti Laptop averaged 162 fps (6.2 ms), while the NVIDIA L40 achieved 179 fps (5.6 ms).

We see an improvement of 0.6 ms (a 10% increase in performance) using the dedicated server. Here we make full use of the added computational power of the NVIDIA L40 GPU to change image formats using compute shaders, allocate/transfer GPU memory, and encode frames using NVENC.

D. Refresh Rate



Refresh rate at	120 fps	60 fps	30 fps
Average latency [ms]	111.0	126.6	214.8
Mean absolute deviation [ms]	8.6	8.1	11.0
Refresh time [ms]	8.3	16.7	33.3

Fig. 9. Latency measured in ms over 20 separated frames (lower is better).

Considering that the user experience is reduced by the added latency of this service, we focus on and investigate the latency induced by the remote rendering pipeline. Consequently, we look into how the choice of refresh rate, video codec and networking protocols change the video latency of the service.

In the previously outlined measurements, we encode every frame the game engine produces. This means that the frame rate and refresh rate are equal in our application. This does not have to be the case, since we can limit the amount of frames that should be encoded by setting a timer. We investigate how changing the refresh rate influences the lag (in our case the latency) of the remote rendering service (Fig. 9).

The video latency is increased with a lower refresh rate. Switching from 120 fps to 60 fps increases video latency by 15.5 ms, which, considering the difference in refresh time of 8.3 ms, seems reasonable, since game threads can be ahead of up to two render threads. Switching from 60 to 30 fps increases video latency by an unexpectedly high value of 88.2 ms. This is likely due to an unoptimized implementation of the encoding and networking component inside the render thread.

E. Comparing Video Codecs

We compare the video latency using AVC (H.264) and its successor HEVC (H.265, better compression efficiency but higher processing time) at a framerate and refresh rate of 120 fps (Fig. 10). At 120 fps framerate and refresh rate,

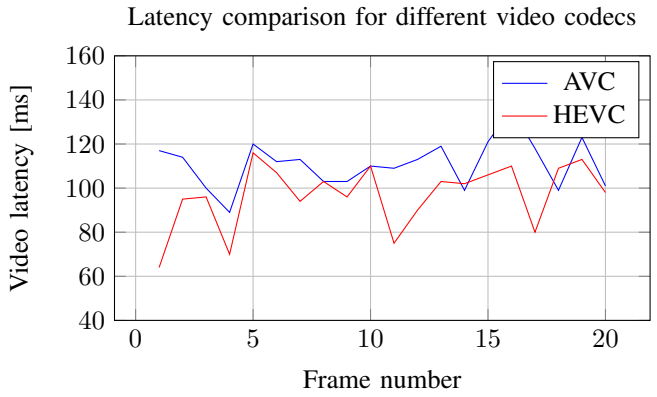


Fig. 10. Latency measured in ms over 20 separated frames (lower is better).

AVC averaged 111.0 ms latency (MAD 8.6 ms), while HEVC averaged 96.9 ms (MAD 11.2 ms).

Unexpectedly, we see a significant decrease in video latency by 14.1 ms. We conclude that the decrease in bitrate and therefore increase in download and playback speed of HEVC outweighs the performance overhead compared to AVC, especially when using powerful hardware such as the NVIDIA L40 GPUs.

F. Networking

We measure the potential overhead in input latency introduced by the complexity of the protocol (Fig. 11). UDP averaged 107.0 ms latency (MAD 6.5 ms), TCP averaged 103.8 ms (MAD 6.7 ms), and SRT averaged 225.5 ms (MAD 8.8 ms).

Considering that we are in a controlled network with almost no package loss, UDP and TCP perform very similarly. Even though we expect TCP to be slower than UDP, the improvement is probably significantly lower than the measurement error rate. SRT is mostly labeled as low latency, its main focus is to deliver the content reliably and with high quality given unstable network conditions. Therefore, an increase in latency of 120 ms is reasonable when using SRT.

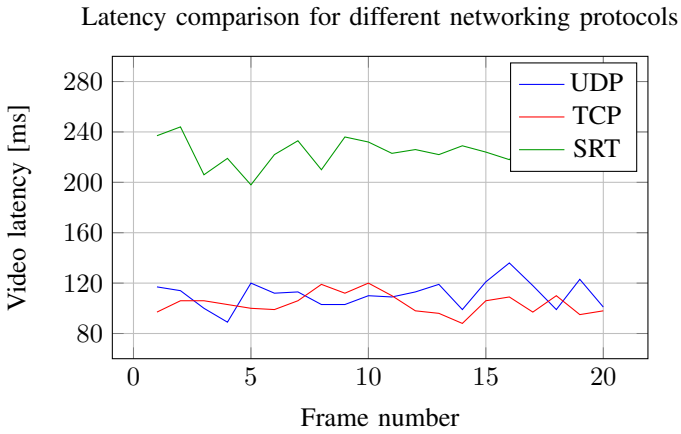


Fig. 11. Latency measured in ms over 20 separated frames (lower is better).

VI. Conclusion

We propose a remote rendering pipeline using Unreal Engine and FFmpeg's media processing capabilities. Integrating FFmpeg into Unreal Engine 5 led us to consider and appreciate various performance optimizations such as avoiding inter-process communication, GPU utilization for encoding and memory management. We also managed to make use of FFmpeg libraries that provide a variety of codecs and protocols, to easily integrate and switch between them. Doing so led us to a testing environment, that is easy to configure and expand.

Given the results of the performance and latency measurements, we achieved a high-performance remote rendering pipeline as an alternative to native execution. In addition, we showed how we can switch between frame rate, refresh rate, codecs, protocols and resolution as well as how to evaluate their effectiveness while also experiencing that some parameter changes may lead to unexpected results. Overall, we gained various insights on the rendering technologies of gaming engines and how they mesh with media processing.

A. Future Works

Given that all these measurements were done in the same local environment, it is of interest how different network setups can influence the latency. In addition, by using cloud services such as AWS, one can provide results using publicly available resources.

It is also of interest to compare this implementation with industry plugins such as PixelStreaming (WebRTC) or implement this pipeline in Unity, which also supports WebRTC.

Using this tool, it is also possible to perform tests for newer protocols such as Media over QUIC (MoQ), since we use FFmpeg to provide various media support.

References

[1] et al., K.B.: Latency impact on quality of experience in a virtual reality simulator for remote control of machines (2025), https://www.sciencedirect.com/science/article/pii/S0923596520301648
[2] arm: What is a gaming engine? (2025), https://www.arm.com/glossary/gaming-engines
[3] Engebretson, J.: Report: Google stadia cloud gamers poised to exceed internet data caps (2019), https://www.telecompetitor.com/report-google-stadia-cloud-gamers-poised-to-exceed-internet-data-caps/
[4] Games, E.: Unreal engine: The most powerful real-time 3d creation tool (2025), https://www.unrealengine.com/
[5] Inc., D.: What is a container? (2025), https://www.docker.com/resources/what-container/
[6] Mahmoud: The evolution of pc gaming: 25 years of system requirements (2023), https://public.tableau.com/app/profile/mahmoud1925/viz/TheEvolutionofPCGaming25YearsofSystemRequirements/Dashboard1
[7] Warren, T., Hollister, S.: Cloud gaming: Google stadia and microsoft xcloud explained (2019), https://www.theverge.com/2019/6/19/18683382/what-is-cloud-gaming-google-stadia-microsoft-xcloud-faq-explainer