

Remote Rendering for Virtual Reality: performance comparison of multimedia frameworks and protocols

Daniel Mejías[†], Inhar Yeregui[‡], Roberto Viola

Fundación Vicomtech

Basque Research and Technology Alliance

San Sebastián, Spain

Email: {damejias, iyeregui, rviola}@vicomtech.org

[†]PhD Candidate at UPV/EHU

Miguel Fernández*, Mario Montagud*[†]

*i2CAT Foundation, Barcelona (Spain)

[†]Universitat de València, València (Spain)

Email: {miguel.fernandez, mario.montagud}@i2cat.net

Abstract—The increasing complexity of Extended Reality (XR) applications demands substantial processing power and high bandwidth communications, often unavailable on lightweight devices. Remote rendering consists of offloading processing tasks to a remote node with a powerful GPU, delivering the rendered content to the end device. The delivery is usually performed through popular streaming protocols such as Web Real-Time Communications (WebRTC), offering a data channel for interactions, or Dynamic Adaptive Streaming over HTTP (DASH), better suitable for scalability. Moreover, new streaming protocols based on QUIC are emerging as potential replacements for WebRTC and DASH and offer benefits like connection migration, stream multiplexing and multipath delivery. This work describes the integration of the two most popular multimedia frameworks, GStreamer and FFmpeg, with a rendering engine acting as a Remote Renderer, and analyzes their performance when offering different protocols for delivering the rendered content to the end device over WIFI or 5G. This solution constitutes a beyond state-of-the-art testbed to conduct cutting-edge research in the XR field.

Index Terms—Extended Reality, Cloud/Edge Rendering, Multimedia streaming, Volumetric multimedia processing.

I. INTRODUCTION

In recent years, Extended Reality (XR) technology has grown in popularity as XR applications and use cases are increasing thanks to progress in network capability and computing hardware. This growth comes at the cost of increasing demand for processing powers, as XR applications execute complex processing tasks requiring Graphics Processing Units (GPUs) to perform them in real time.

Nevertheless, the processing power is not always available on the end devices. Virtual Reality (VR) headsets, smartphones and tablets, as well as lightweight laptops, are not provided with the necessary hardware acceleration to run XR applications. For such a reason, Remote Rendering solutions are being used as an alternative to the local processing on the end device. Heavy volumetric video rendering tasks are offloaded to a remote GPU-equipped computing node, which delivers only the processing result to the end device [1], [2]. The result can be a 2D directive or a 360° video stream, associated with an audio stream.

Current solutions for Remote Rendering deliver the content to the end device using Web Real-Time Communications (WebRTC) [3], while exploiting its data channel to exchange information concerning user interactions with the VR scene [4], [5]. WebRTC protocol is a perfect fit for its real time latency that allows the interaction information to be delivered as fast as possible. Nevertheless, it has an intrinsic limitation when high scalability is requested. Employing a Hypertext Transfer Protocol (HTTP) Adaptive Streaming (HAS) protocols, e.g., such as Dynamic Adaptive Streaming over HTTP (DASH) [6], enables the use of Content Delivery Networks (CDNs) to increase the number of connected users. HAS protocols constitute the best solutions when the scalability is more important than the interaction capabilities, as the latter are limited only to rotation within the VR scene and only when 360° video is employed [7].

Furthermore, the rise of QUIC [8] as the transport protocol for HTTP/3 [9] has the potential to be a game changer for future multimedia communications. Its native features such as session migration, stream multiplexing and multipath delivery make it suitable for user mobility or flexible network reconfiguration scenarios. Consequently, new streaming protocols, such as Real-time Transport Protocol (RTP) over QUIC (RoQ) [10] and Media over QUIC (MoQ) [11], are proposed and some initial implementations are already available and tested [12], [13]. If these protocols reach a mature development and become widely used, they could replace WebRTC and DASH, as they could ideally combine real time capabilities with scalability supported by CDN infrastructure.

The contributions of this work are as follows:

- A full-fledged Remote Rendering pipeline capable of offloading from headsets the processing tasks required for rendering VR experiences, generating a 360° video stream compatible with any lightweight client device, while maintaining responsive interactions.
- A performance comparison among two prominent open-source media streaming frameworks, GStreamer [14] and FFmpeg [15], including their advantages and disadvantages when integrated into XR services.
- An analysis of the latency achieved by standard

multimedia protocols—WebRTC, DASH, Low Latency DASH (LL-DASH) [16], QUIC and MoQ—when delivering remotely rendered content.

Even if Remote Rendering is relevant and applicable to any XR scenario, this work focuses primarily on VR to maximize clarity and depth. However, the contributions made can effectively improve a variety of immersive experiences in the broader XR landscape.

II. BACKGROUND AND RELATED WORK

Different authors have addressed the need for remote rendering in various applications, such as scientific simulation, XR, training, computer-aided design (CAD), and cloud gaming [17]. Studies have shown that executing rendering tasks remotely enables users to access complex virtual scene environments from resource-constrained devices [18]. However, this approach introduces challenges related to real-time data transmission. To mitigate these issues, researchers have explored the use of communication protocols optimized for multimedia data. For instance, the RTP protocol has been widely adopted in image capture and videoconferencing applications due to its ability to handle real-time streaming [19], [20]. Nevertheless, significant challenges remain, particularly in terms of adaptability to different types of content, scalability across various devices, and efficiency in low-bandwidth networks, underscoring the need for further research in this area [21], [22].

Modern frameworks for virtual scene development engines support remote communication using RTP-based protocols. WebRTC is one of the most versatile solutions for real-time communication and is integrated into some development engines. Unity and Unreal Engine are widely used engines for video games and interactive 3D applications, each with distinct characteristics, but both are highly versatile due to their integration with multiple platforms [23], [19], [24]. Unity offers Universal Render Streaming¹, which enables remote rendering with various graphical features, including encoding, dynamic bitrate adjustment, and communication via WebRTC. Similarly, Unreal Engine provides Pixel Streaming², which allows rendering and transmitting video output to the client through WebRTC communication.

However, in some cases, WebRTC may not be the most optimal choice, as firewall restrictions depending on network conditions may require specific port openings or the use of STUN/TURN servers, introducing additional latency. Additionally, the QUIC protocol may offer better performance in terms of reduced negotiation time and lower latency [23]. Currently, there is no QUIC-based development in rendering engines, although it is an emerging and rapidly growing protocol. Another important requirement is the ability to achieve scalability and video adaptability according to network conditions using adaptive bitrate protocols such as DASH. Therefore, the need for systems capable of supporting

multiple communication protocols becomes even more critical, as well as the development of protocols adapted to the continuous evolution of networks and applications. Integrating a virtual scene development engine with frameworks such as GStreamer[14] or FFmpeg[15] can enable the use of different protocols, allowing greater adaptability to user requirements.

At the user level, the renderer must be accessible from any end device while maintaining the same quality of experience and low-latency characteristics. For this reason, the most optimal implementation is the use of a web browser as the rendering client. Many real-time communication protocols support web-based player development, enabling seamless integration with remote rendering. Additionally, technologies such as WebXR³ facilitate 3D visualization on headsets and mobile devices.

III. MULTI-FRAMEWORK AND MULTI-PROTOCOL REMOTE RENDERING SOLUTION

Figure 1 presents the proposed and implemented architecture of the Remote Renderer for VR experiences that communicates with the video player on the end device using different standardized streaming protocols. The Remote Renderer consists of two main software components: a Unity main application, developed in C#, and a native rendering plugin, developed in C++ and dynamically linked to the main application.

The Unity main application acts as the central hub, coordinating the interactions between the different components. It is based on the Unity rendering engine and the Unity Render Streaming (URS) plugin [25]. The former uses NVIDIA GPU capabilities to process heterogeneous content integrated in a VR scene, including audio and volumetric video, and generate rendered raw audio and 360° video streams. The latter receives the raw streams, encodes them and generates a WebRTC stream to be delivered to the video player. NVIDIA encoder and *libwebrtc* library [26] are employed for GPU encoding and WebRTC communication, respectively. The audio is encoded with OPUS codec, while the video with H.264 codec.

The native rendering plugin [27] consists of a custom Unity plugin to bridge the Unity main application and the multimedia framework in charge of creating a protocol-specific multimedia pipeline. Since no framework supports all the streaming protocols, two plugin versions are developed to integrate GStreamer and FFmpeg. The differences in implementation and supported protocols are described in the specific subsections.

Overall, the combination of URS, GStreamer and FFmpeg enables to generate WebRTC, DASH, LL-DASH, QUIC and MoQ streams. These standardized protocols have the advantage of enabling the reception, decoding and visualization by compliant video players without modifications. Adding user interaction instead requires introducing mechanisms for capturing the interaction on the

¹<https://docs.unity3d.com/Packages/com.unity.renderstreaming>

²<https://www.streampixel.io/>

³<https://immersiveweb.dev/>

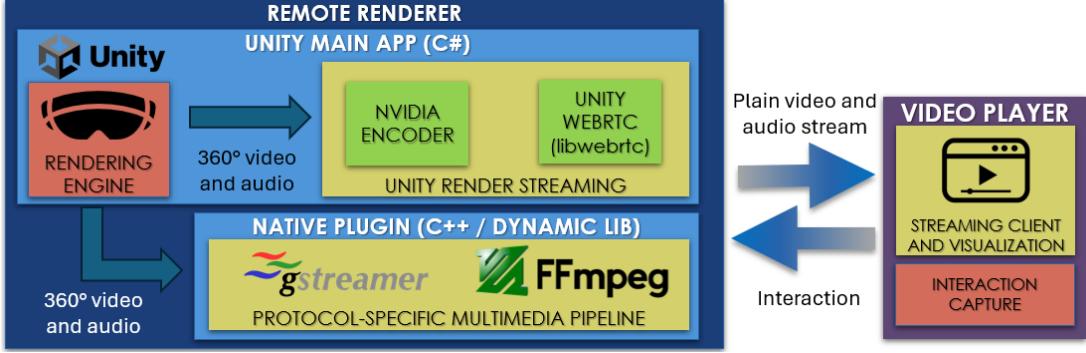


Fig. 1. General solution for multi-protocol Remote Renderer.



Fig. 2. Integration of GStreamer with the Remote Renderer.

end device, which are currently available only when using the URS to generate WebRTC.

In conjunction, this constitutes a beyond state-of-the-art testbed to conduct cutting-edge research in this field. It allows to find out pros and cons of different alternatives for media processing, streaming and interaction.

A. GStreamer

Figure 2 represents the integration of the GStreamer native plugin with the Unity main application and the operations performed by its internal multimedia pipelines. GStreamer is integrated by calling its native Application Programming Interfaces (APIs) to build and run the multimedia pipeline. Two GStreamer *appsrc* [28] elements at the beginning of the pipeline allow pushing separately the 360° video and audio buffers into the pipeline. Then, the buffers are encoded with the support of NVIDIA encoders. The video is always encoded with H.264 codec, while the audio could use either OPUS or AAC. Depending on the protocol, the last operation within the pipeline is different. Current supported protocols and operations are as follows:

- WebRTC: the encoded flows are packaged into a WebRTC stream. The resulting pipeline is an alternative to the URS WebRTC implementation and enables a direct comparison with it.
- DASH: the encoded flows are muxed and split into MP4 segments, while a DASH Media Presentation Description (MPD) is generated. This pipeline enables higher scalability by design, even if it comes at a cost of higher latency compared to the WebRTC pipeline.
- QUIC: the encoded video is packaged into RTP streams [29]. Differently from all the other protocols which belong to the application layer, this is simply a transport

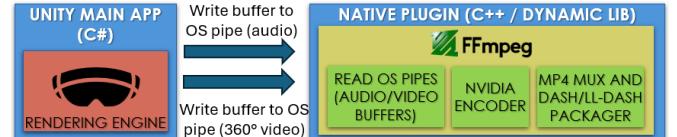


Fig. 3. Integration of FFmpeg with the Remote Renderer.

layer protocol. Since no application layer protocols based on QUIC are officially supported yet by GStreamer, this can be considered as a reference for future application layer implementations. On top of QUIC, a legacy RTP stream is employed. This must not be confused with RoQ which requires further adaptations of the RTP stream.

- MoQ: a fragmented MP4 (fMP4) muxer is employed for packaging the encoded stream. Then, an unofficial GStreamer element implements MoQ to deliver the fMP4 flow [30]. Since this element is not included in stable and maintained GStreamer releases, it is less mature than other alternatives and does not support audio yet. However, it constitutes the only QUIC-based application layer protocol.

B. FFmpeg

Figure 3 represents the integration of the FFmpeg native plugin with the Unity main application and the operations performed by its internal multimedia pipeline. Differently from GStreamer, FFmpeg is integrated by calling its Command Line Interface (CLI) and using two Operating Systems (OS) pipes [31] to separately write the 360° video and audio buffers from Unity and read them from the FFmpeg pipeline. Then, the buffers are encoded with the support of NVIDIA encoders. The video is encoded with H.264 codec, while the audio with AAC codec. For the last processing step, two alternatives are available depending on the protocol:

- DASH: the encoded flows are muxed and split into MP4 segments, while a DASH MPD is generated. This is similar and directly comparable with GStreamer DASH pipeline.
- LL-DASH: similar to the DASH pipeline, the segments and MPD are generated, but a fMP4 muxed is instead

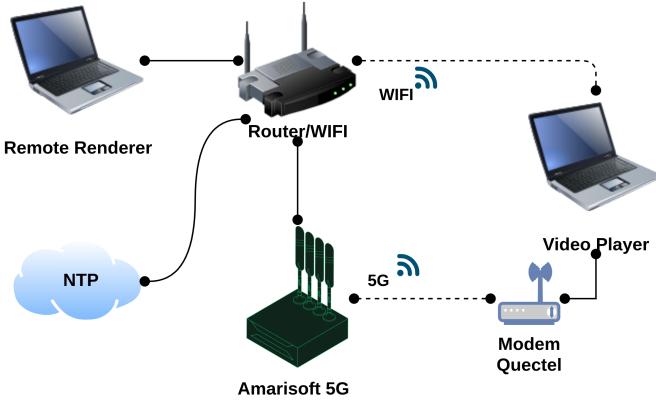


Fig. 4. Testbed for WIFI and 5G latency measurement.

employed. It allows the reduction of MP4 fragment duration within the segments to lower the latency.

IV. EXPERIMENTAL RESULTS

The testbed in Figure 4 is employed to compare the multimedia frameworks and protocols integrated into the Remote Rendering solution. Its specifications are as follows:

- Remote Renderer: a laptop acting as the rendering node with an Intel i7-13650HX Central Processor Unit (CPU), 16GB Double Data Rate 5 (DDR5) Random-Access Memory (RAM) and NVIDIA RTX4060 GPU.
- Router WIFI: a WIFI 6 compatible router employing the 5GHz band for communication between the Remote Renderer and the video player. Alternatively, this equipment can route the traffic to the 5G network instead of WIFI.
- AMARI Callbox Mini [32]: equipment with 5G Core and Radio Access Network (RAN) implementing the 3rd Generation Partnership Project (3GPP) Release 15 [33]. It is configured to use the N78 band, 20 MHz bandwidth, 30 MHz subcarrier spacing, max modulation of 256 Quadrature Amplitude Modulation (QAM), 2 antennas for downlink and 1 for uplink.
- Quectel 5G modem: Quectel RM500Q modem implementing 3GPP Release 15 and connected to the video player equipment via USB.
- Video Player (laptop): the video player is displayed from a web interface or an application (only for QUIC). Any computer with basic capabilities can be used. This laptop has an internal WIFI 6 adapter and the driver installed for the Quectel modem.
- Network Time Protocol (NTP): the router provides an Internet connection that allows the Remote Renderer and the Video Player to synchronize to a public NTP server for latency testing.

The development involves the rendering of a Unity-generated virtual scene into a plain video representation, to be later encoded and transmitted using different protocols. The scene consists of a test environment with a simple clock object

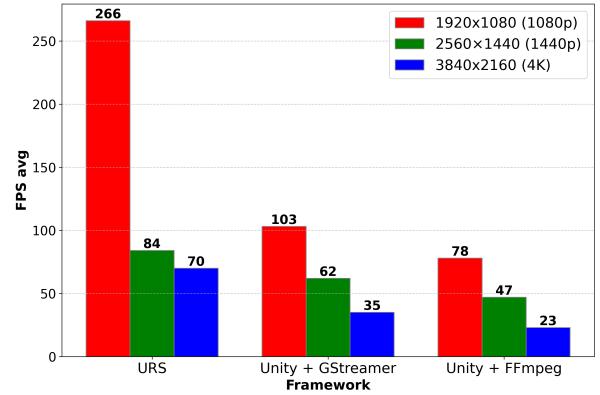


Fig. 5. Maximum fps for each framework.

TABLE I
LATENCY OVER WIFI WITH VIDEO STREAM AT 1080P/60FPS/10MBPS.

Protocol	Sender framework	Receiver framework	Latency _{avg} (ms)	Latency _{dev} (ms)
WebRTC	URS	URS player [25]	82	12
	Unity + GStreamer	gstwebrtc-api [34]	219	22
DASH	Unity + GStreamer	Dash.js [35]	9399	13
	Unity + FFmpeg	Dash.js [35]	6183	11
LL-DASH	Unity + FFmpeg	Dash.js [35]	4873	13
QUIC (RTP)	Unity + GStreamer	GStreamer pipeline	248	2
MoQ	Unity + GStreamer	MoQ player [30]	159	12

TABLE II
LATENCY OVER 5G WITH VIDEO STREAM AT 1080P/60FPS/10MBPS.

Protocol	Sender framework	Receiver framework	Latency _{avg} (ms)	Latency _{dev} (ms)
WebRTC	URS	URS player [25]	129	11
	Unity + GStreamer	gstwebrtc-api [34]	253	16
DASH	Unity + GStreamer	Dash.js [35]	10227	14
	Unity + FFmpeg	Dash.js [35]	6519	11
LL-DASH	Unity + FFmpeg	Dash.js [35]	5084	19
QUIC (RTP)	Unity + GStreamer	GStreamer pipeline	293	31
MoQ	Unity + GStreamer	MoQ player [30]	194	20

showing the timestamp. This clock is necessary to assess the exact time when each frame is generated. We execute two performance tests: the first consists of measuring the maximum framerate achieved in rendering task, and the second is an end-to-end video latency test.

In the first test, the results are presented in Figure 5, which shows the average maximum framerate (frames per second or fps) processed by each framework, independently of the considered protocol. These values are calculated by counting the frames after the GPU encoding and writing them directly into a file. URS is outperforming GStreamer and FFmpeg as it has the best integration with Unity, being part of its official plugins. When using GStreamer and FFMPEG, the raw frames are extracted from the GPU to be injected into their pipelines. This operation is inefficient as later both frameworks inject the frames back into the GPU for encoding. In any case, the usage of GStreamer native APIs provides better performance than FFmpeg CLI, as passing through OS pipes is not necessary. FFmpeg CLI is still a valid solution for fast prototyping due to its easier integration, even if the framerate at 4k might not guarantee a smooth visual experience.

In the second test, the results are presented in Table I and II present the latency achieved by each streaming protocol when delivering a 1080p/60fps video encoded at 10Mbps over WIFI and 5G, respectively. Since a high framerate allows to reduce the latency [36], 1080p is chosen for these tests as it is the only resolution that each framework can provide with at least 60fps. Except for QUIC which employs a GStreamer pipeline as a player (a QUIC stream is not playable within a browser without an application layer protocol), all the other protocols employ reference web players. For encoding, H.264 was used with an Nvidia GPU, configured with a Group of Pictures (GOP) of 5 frames, a encoding bitrate of 10 Mbps, and a frame rate of 60 FPS in URS, GStreamer and FFmpeg.

All the protocols show better results over WIFI than 5G, ranging from -36% with URS WebRTC to -4% with LL-DASH. WebRTC has the lowest latency when implemented with URS, while the GStreamer WebRTC alternative increases significantly by +167% over WIFI and +96% over 5G. Again, this can be explained by the inefficient process of extracting the frames from the GPU. The results of the QUIC transport protocol are quite promising for future multimedia communication. QUIC (RTP) shows a higher latency but close to GStreamer WebRTC, being the difference 29ms over WIFI and 40ms over 5G. MoQ implementation is instead already outperforming GStreamer WebRTC, giving the best results after URS WebRTC.

As expected, DASH and LL-DASH implemented with FFmpeg present the highest latency. Both of them are configured to generate segments of 2 seconds, but fMP4 allows LL-DASH to have 0.5 seconds fragments. The LL-DASH latency is 21% lower than DASH over WIFI and -22% over 5G. The generation of DASH content with GStreamer does not fully comply with the *DASH-IF* recommendations. It is worth noticing that the current GStreamer DASH implementation does not generate a stream compliant with DASH Industry Forum recommendations. As a result, we had to modify it to make it compatible with *Dash.js* and submitted a merge request to GStreamer⁴. However, it exhibits a latency increase

⁴https://gitlab.freedesktop.org/gstreamer/gstreamer/-/merge_requests/7886

TABLE III
CHARACTERISTICS OF STUDIED MULTIMEDIA FRAMEWORKS.

Framework	Prototype	Protocols	Tune	HW control
URS	built-in Unity plugin	single protocol	few parameters update on the fly	no control
GStreamer	modular integration	lots of protocols	most parameters update on the fly	GPU/CPU and threads
FFmpeg	command-line integration	lots of protocols	some parameters update on the fly	enable presets

of +156% compared to FFmpeg DASH, as the GStreamer implementation might not have the same maturity as FFmpeg.

Finally, Table III provides an overview of key features to consider when selecting any of the studied frameworks for producing and delivering rendered experiences as standard multimedia streams. Each framework has its advantages and disadvantages, and the choice depends on the specific requirements of the developers, such as support for new protocols, live adaptation of streaming parameters, or hardware (HW) control. It is useful to consider these parameters to make the best decision on which framework to choose.

V. CONCLUSION

This paper has described an architecture capable of offloading the computational load of the VR headset by remotely rendering experiences with capable systems and streaming the video with different protocols to lightweight client devices. In addition, a performance study of two leading open-source multimedia streaming frameworks, GStreamer and FFmpeg, has highlighted their respective strengths and weaknesses in XR service integration, comparing them to the built-in URS Unity plugin. Finally, the analysis of multimedia protocols, such as WebRTC, DASH, LL-DASH, QUIC and MoQ, has provided valuable insights into their latency performance in delivering remotely rendered content, facilitating efficient and seamless XR experiences.

The final contribution of this paper is to provide a comprehensive testbed implementation of a Remote Rendering solution to conduct cutting-edge research in this field, along with insightful results that reveal its usefulness.

ACKNOWLEDGMENT

This research was supported by the SNS-JU Horizon Europe Research and Innovation programme, under Grant Agreement 101096838 for 6G-XR project, the Spanish MINECO and the European Union – NextGeneration EU, in the framework of the PRTR Call UNICO I+D 5G 2021, under Grant TSI-063000-2021-4 for 6G-Openverso-Holo project. The work of Mario Montagud has been funded by MCIN/AEI/10.13039/501100011033 under Grant RYC2020-030679-I and by the ESF.

REFERENCES

- [1] G. Xiao, H. Li, C. Han, Y. Liu, Y. Li, and J. Liu, “Cloud rendering scheme for standalone virtual reality headset,” in 2020 International Conference on Virtual Reality and Visualization (ICVRV). IEEE, 2020, pp. 316–319.

- [2] I. Yeregui, D. Mejías, G. Pacho, R. Viola, J. Astorga, and M. Montagud, “Edge rendering architecture for multiuser xr experiences and e2e performance assessment,” in *2024 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*. IEEE, 2024, pp. 1–7.
- [3] “Real-time communication for the web,” [Online] Available: <https://webrtc.org/>, Accessed: 2024-10-10.
- [4] M. Casasnovas Bielsa, C. Michaelides, M. Carrascosa, and B. Bellalta, “Experimental evaluation of interactive edge/cloud virtual reality gaming over wi-fi using unity render streaming,” *Cloud Virtual Reality Gaming Over Wi-Fi Using Unity Render Streaming*, 2024.
- [5] H. Feng, H. Lu, F. Zhang, and Z. Li, “Application and research of high quality pixel streaming architecture based on unreal engine,” in *Proceedings of the 3rd International Conference on Computer, Artificial Intelligence and Control Engineering*, 2024, pp. 17–21.
- [6] *Information technology — Dynamic adaptive streaming over HTTP (DASH)*, International Organization for Standardization Std. ISO/IEC 23 009-1:2022, 2022.
- [7] D. Podborski, E. Thomas, M. Hannuksela, S. Oh, T. Stockhammer, and S. Pham, “Virtual reality and dash,” *Proc. Int. Broadcast. Conv.(IBC)*, pp. 1–11, 2017.
- [8] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” RFC 9000, May 2021. [Online]. Available: <https://www.rfc-editor.org/info/rfc9000>
- [9] M. Trevisan, D. Giordano, I. Drago, and A. S. Khatouni, “Measuring http/3: Adoption and performance,” in *2021 19th Mediterranean Communication and Computer Networking Conference (MedComNet)*. IEEE, 2021, pp. 1–8.
- [10] M. Engelbart, J. Ott, and S. Dawkins, “RTP over QUIC (RoQ),” Internet Engineering Task Force, Internet-Draft draft-ietf-avtcore-rtp-over-quic-11, Jul. 2024, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/draft-ietf-avtcore-rtp-over-quic/11/>
- [11] L. Curley, K. Pugin, S. Nandakumar, V. Vasiliev, and I. Swett, “Media over QUIC Transport,” Internet Engineering Task Force, Internet-Draft draft-ietf-moq-transport-06, Sep. 2024, work in Progress. [Online]. Available: <https://datatracker.ietf.org/doc/draft-ietf-moq-transport/06/>
- [12] M. Engelbart and J. Ott, “Congestion control for real-time media over quic,” in *Proceedings of the 2021 Workshop on Evolution, Performance and Interoperability of QUIC*, 2021, pp. 1–7.
- [13] Z. Gurel, T. Erkilic Civelek, A. Bodur, S. Bilgin, D. Yeniceri, and A. C. Begen, “Media over quic: Initial testing, findings and results,” in *Proceedings of the 14th Conference on ACM Multimedia Systems*, 2023, pp. 301–306.
- [14] “GStreamer,” [Online] Available: <https://gstreamer.freedesktop.org/>, Accessed: 2024-10-10.
- [15] “FFmpeg,” [Online] Available: <https://www.ffmpeg.org/>, Accessed: 2024-10-10.
- [16] “DASH-IF publishes Low-Latency DASH extensions and 2nd Community Review for DASH-IF Ad Insertion,” [Online] Available: <https://dashif.org/news/low-latency-dash/>, Accessed: 2024-10-10.
- [17] F. Lambert and A. Sanna, “A streaming-based solution for remote visualization of 3d graphics on mobile devices,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 247–260, 2007.
- [18] E. Lu, S. Bharadwaj, M. Dasari, C. Smith, S. Seshan, and A. Rowe, “Renderfusion: Balancing local and remote rendering for interactive 3d scenes,” in *2023 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2023, pp. 312–321.
- [19] L. Bassbouss, A. Neparidze, K. Kieslich, S. Steglich, S. Arbanowski, and P. Pogrzeba, “Metaverse remote rendering testbed,” in *2023 IEEE International Conference on Metaverse Computing, Networking and Applications (MetaCom)*, 2023, pp. 578–584.
- [20] Y. Li, S. Wang, Y. Li, A. Zhou, M. Xu, X. Ma, and Y. Liu, “Seamless cross-edge service migration for real-time rendering applications,” *IEEE Transactions on Mobile Computing*, vol. 23, no. 6, pp. 7084–7098, 2024.
- [21] S. Shi, M. Kamali, K. Nahrstedt, J. C. Hart, and R. H. Campbell, “A high-quality low-delay remote rendering system for 3d video,” in *Proceedings of the 18th ACM international conference on Multimedia*, 2010, pp. 601–610.
- [22] S. Shi, K. Nahrstedt, and R. Campbell, “A real-time remote rendering system for interactive mobile graphics,” *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 8, no. 3s, pp. 1–20, 2012.
- [23] A. Zoubarev, L. Bassbouss, D. L. Tran, S. Steglich, and S. Arbanowski, “A novel approach for remote rendering and streaming in xr,” in *2010 31477824 2nd International Conference on Intelligent Metaverse Technologies & Applications (iMETA)*, 2024, pp. 198–205.
- [24] I. Yeregui, D. Mejías, G. Pacho, R. Viola, J. Astorga, and M. Montagud, “Edge rendering architecture for multiuser xr experiences and e2e performance assessment,” in *2024 IEEE International Symposium on Broadband Multimedia Systems and Broadcasting (BMSB)*, 2024, pp. 1–7.
- [25] “Unity Renderer Streaming,” [Online] Available: <https://docs.unity3d.com/Packages/com.unity.renderstreaming@3.1/\manual/index.html>, Accessed: 2024-10-10.
- [26] “libwebrtc,” [Online] Available: <https://github.com/webrtc-sdk/libwebrtc>, Accessed: 2024-10-10.
- [27] “Unity Native plug-ins,” [Online] Available: <https://docs.unity3d.com/Manual/NativePlugins.html>, Accessed: 2024-10-10.
- [28] “GStreamer: appsrc,” [Online] Available: <https://gstreamer.freedesktop.org/documentation/app/appsrc.html?gi-language=c>, Accessed: 2024-10-10.
- [29] H. Schulzrinne, S. L. Casner, R. Frederick, and V. Jacobson, “RTP: A Transport Protocol for Real-Time Applications,” RFC 3550, Jul. 2003. [Online]. Available: <https://www.rfc-editor.org/info/rfc3550>
- [30] L. Curley, “Media over QUIC,” [Online] Available: <https://quic.video>, Accessed: 2024-10-10.
- [31] “pipe - overview of pipes and FIFOs,” [Online] Available: <https://manpages.ubuntu.com/manpages/jammy/en/man7/pipe.7.html>, Accessed: 2024-10-10.
- [32] “AMARI Callbox Mini,” [Online] Available: <https://www.amarisoft.com/test-and-measurement/device-testing/device-products/amari-callbox-mini>, Accessed: 2024-10-10.
- [33] “3GPP Release 15,” [Online] Available: <https://www.3gpp.org/specifications-technologies/releases/release-15>, Accessed: 2024-10-10.
- [34] “gstwebrtc-api,” [Online] Available: <https://gitlab.freedesktop.org/\gstreamer/gst-plugins-rs/-/tree/main/net/webrtc/gstwebrtc-api>, Accessed: 2024-10-10.
- [35] “Dash.js,” [Online] Available: <https://dashjs.org/>, Accessed: 2024-10-10.
- [36] A. Geris, B. Cukurbasi, M. Kilinc, and O. Teke, “Balancing performance and comfort in virtual reality: A study of fps, latency, and batch values,” *Software: Practice and Experience*, 2024.