Hindawi
International Journal of Computer Games Technology
Volume 2021, Article ID 6676644, 16 pages
https://doi.org/10.1155/2021/6676644

Research Article

Synchronous Remote Rendering for VR

Viktor Kelkkanen,[1] **Markus Fiedler**,[2] **and David Lindero**[3]

[1]*Department of Computer Science, Blekinge Institute of Technology, Karlskrona 37179, Sweden*
[2]*Department of Technology and Aesthetics, Blekinge Institute of Technology, Karlshamn 37435, Sweden*
[3]*Ericsson Research, Ericsson AB, Luleå 97753, Sweden*

Correspondence should be addressed to Viktor Kelkkanen; viktor.kelkkanen@bth.se

Received 5 November 2020; Revised 12 March 2021; Accepted 8 July 2021; Published 20 July 2021

Academic Editor: Michael J. Katchabaw

Remote rendering for VR is a technology that enables high-quality VR on low-powered devices. This is realized by offloading heavy computation and rendering to high-powered servers that stream VR as video to the clients. This article focuses on one specific issue in remote rendering when imperfect frame timing between client and server may cause recurring frame drops. We propose a system design that executes synchronously and eliminates the aforementioned problem. The design is presented, and an implementation is tested using various networks and hardware. The design cannot drop frames due to synchronization issues but may on the other hand stall if temporal disturbances occur, e.g., due to network delay spikes or loss. However, experiments confirm that such events can remain rare given an appropriate environment. For example, remote rendering on an intranet at 90 fps with a server located approximately 50 km away yielded just 0.002% stalled frames while rendering with extra latency corresponding to the duration of exactly one frame (11.1 ms at 90 fps). In a LAN without extra latency setting, i.e., with latency equal to locally rendered VR, 0.009% stalls were observed while using a wired Ethernet connection and 0.058% stalls when using 5 GHz wireless IEEE 802.11 ac.

1. Introduction

With increasing interest in virtual reality (VR) comes an opportunity to deploy VR functionality in phones or other thin devices that, due to cost or mobility, lack the computational power required to render high-quality VR on their own hardware [1]. By offloading game-logic and rendering to strong servers in the local network or at the edge, and streaming the live-rendered content as video, these thin devices would merely need strong hardware decoders and network connections to enable high quality VR. In the end, this can reduce the cost and increase the mobility of VR clients.

There are several categories of remote rendering for VR already available in the present market. On the inexpensive low end, there are apps designed to stream VR from home PCs to phones or other thin devices [2–4]. When a phone is used as client, it is placed in a VR gadget that can be used as Head-Mounted Display (HMD) by having the phone act as display [5–7]. This enables an inexpensive alternative of VR for users who already have high-powered PCs at home but no high-end headsets. On the high end of concurrent remote VR, there are for example the wireless VR adapters available to HTC Vive and Oculus Rift [8, 9] and USB-tethered remote rendering for Oculus Quest [10]. In this work, we present an architecture that can be used to minimize both latency and the amount of frame drops in remote rendering systems for VR.

In remote rendering, the latency from input to perceived effect on-screen must generally be low in order to provide a decent user experience. A popular application of remote rendering is cloud gaming. The latency requirements of cloud gaming depend on the type of game, but typically range between 60 and 120 ms [11]. With VR, however, this budget is defined as the Motion-To-Photon (MTP) budget and expected to range between 7 ms [12–14] and 20 ms [11, 15–18]. The 7 ms deadline was recently (2019) reached with Valve's VR headset Index, which has a 144 Hz mode

(6.94 ms budget). However, note that there may be additional delays other than the refresh interval (e.g., from input devices), while the latter is the lowest possible latency on the display in question. For example, framerates of up to 1800 Hz are expected to be required for a life-like experience, but a latency of 7 ms would still be unnoticeable on such displays according to previous research [19].

1.1. VR Remote Rendering. A multitude of steps are required for each frame to enable the remote rendering of VR:

(1) Client: get input data from input devices

(2) Client: send the input data to the server

(3) Server: run the game logic

(4) Server: render the virtual world individually per eye

(5) Server: encode the resulting images

(6) Server: send the encoded images to the client

(7) Client: decode the images

(8) Client: display the images in the HMD

Considering the number of steps and the tight MTP budget of VR in the order of 7 to 20 ms, remote rendering for this medium is a tall order.

Two of the most common VR headsets are the HTC Vive and Oculus Rift, both still in the top five of most common VR headsets on Steam as of September 2020 [20]. Both these HMDs use a 90 Hz refresh rate, which results in a frame time budget of 11.1 ms. However, even though the MTP-budget may be 20 ms, this does not mean that we obtain an additional 8.9 ms headroom that can be allocated to network latency. The MTP depends on the VR system in question, partly the devices and drivers that continuously provide the rotation and translation data of the headset. For example, in the HTC Vive, the rotation of the HMD is updated at 1000 Hz [21, 22], thus adding at least 1 ms to the MTP delay in the worst case. On the other hand, the Oculus Rift also samples tracking data at 1000 Hz, but sends two updates with each packet on the USB cable, resulting in a 500 Hz update rate or 2 ms worst-case delay [23]. Worse yet, supposedly, the update rate for translations may for example be as low as 250 Hz on the HTC Vive [21]. Additionally, there is the added latency caused by drivers, display, and scan-out. Scanning out the image from GPU to display requires a significant amount of time, typically one full frame time [24]. Thus, even on locally rendered VR, there is a significant amount of latency. However, by predicting future motions [25, 26] and utilizing other latency-mitigating technologies such as 2D- [26–29] and 3D-image warping [30–32], both these systems can achieve near-zero latency from the user perspective ([33], Chapter~12). Although the mentioned mitigation strategies can severely reduce the rotation and translation latency, it is always better to avoid a problem than to mitigate it [27]. Furthermore, interaction latency remains difficult to mitigate without moving the interactive component to the client [34], which complicates development and may defeat the purpose of using remote rendering.

1.1.1. Local Latency Mode. Ideally, the remote rendering solution completes all previously mentioned steps (1–8) and is able to present a new frame before the Vertical Synchronization (VSync) deadline for the display is hit (VSync is a technique that ensures that a new scan-out does not begin until the previous has finished. VSync avoids an issue where the new frame may overwrite parts of the old frame, commonly referred to as tearing [24]). If the solution achieves this, the VR experience has the potential to be no different in terms of latency to locally rendered VR, which we refer to as local latency mode. Naturally, this is difficult to achieve because the game logic and rendering itself may already consume most of the available frame time. Adding encoding, transmission and decoding may break the deadline in demanding games when using common codecs such as H.264, thus introducing delays or lowering the framerate.

1.1.2. Server-Client Synchronization Challenges. A naive remote rendering solution without special hardware typically consists of a server and client that execute in parallel without any synchronization. The server runs a game independently and is continuously fed input data which is used to render the following frame. The rendered frame is sent to the client that runs its own independent game loop and continuously presents its most recently received frame. Such a design causes issues with frame timing between the two programs. Ideally, new frames are decoded and ready to be displayed on the client a few milliseconds before the VSync deadline. This is difficult to achieve without some synchronization that for example decides when the server should start rendering [22].

1.1.3. Synchronous Design Motivation. We propose a remote VR solution that is synchronous and, in an ideal scenario, bufferless. In the latter case, it does not add latency and keeps the same MTP as if the HMD was used locally. However, due to the synchronization, it may reduce the framerate if the deadline of 11.1 ms (at 90 fps) cannot be met in the present environment (see Figure 1). In order to maintain the deadline though, the system may reduce the resolution of the images to reduce time consumption of the codec and other components. The prototype may also inject one frame delay to maintain 90 fps when local latency operation is not possible. When this is done, the system introduces latency that would not be present in local rendering and is thus no longer running in local latency mode.

In the proposed design, the remote rendering functionality is integrated into the game engine and acts as a part of the framerate budget. This restricts portability of the solution since it cannot run as a separate process, but must be written into the game. However, it enables a shorter MTP path because the rendering on the server waits for a new HMD pose before starting each frame. Likewise, the client waits for the resulting rendered images to be transmitted back from the server. The waiting ensures that the two stay in sync and

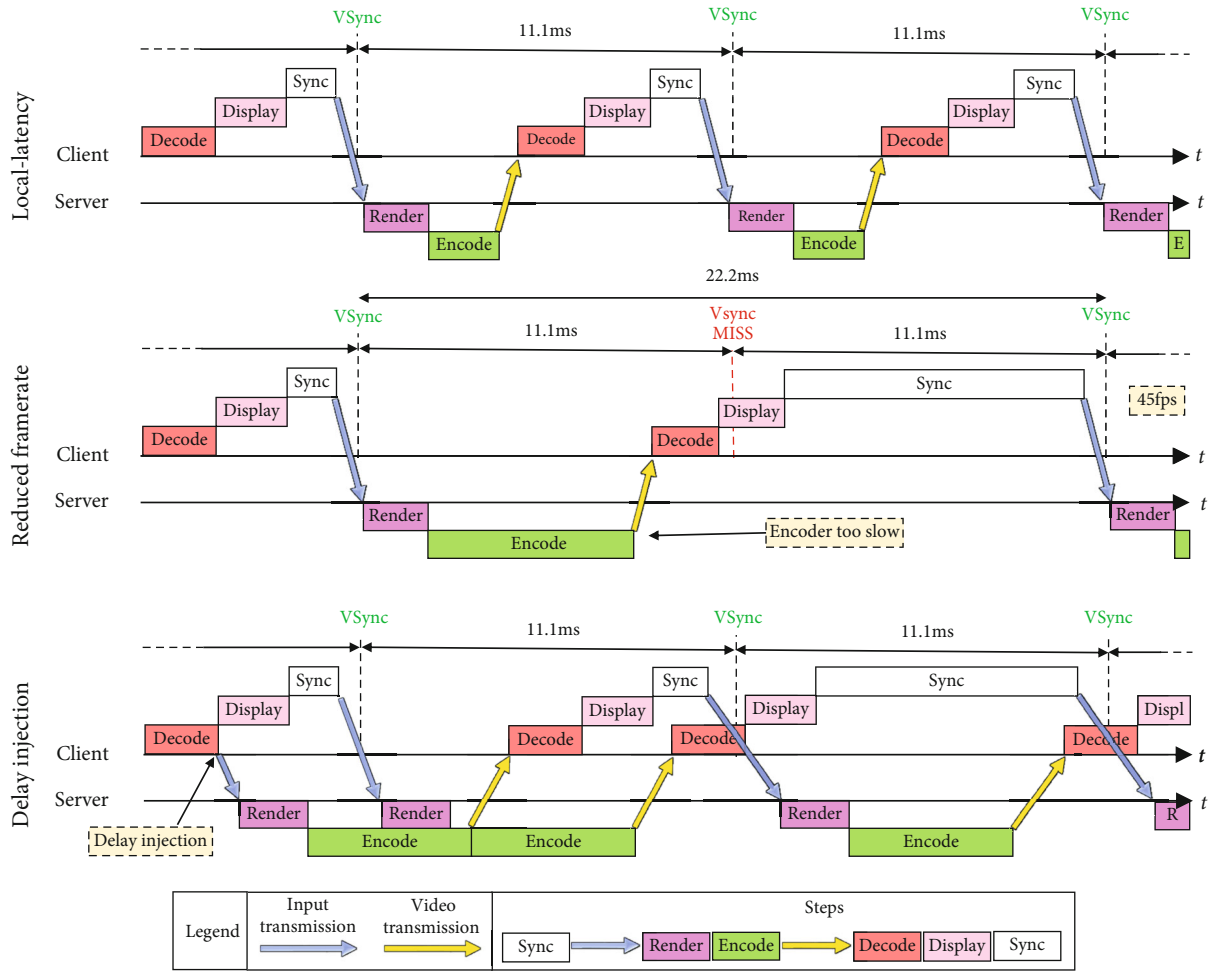

FIGURE 1: The three states of the synchronous system. (1) *Local latency:* all steps are complete within the refresh interval; latency is the same as local rendering. (2) *Reduced framerate:* the system cannot finish within the deadline, and fps will thus be reduced to half. (3) *Delay injection:* one frame delay is injected into the system. It is initialized by sending an extra input packet after decoding which will trigger parallel processing of two frames from then on, both delayed by one frame. The white Sync-bar is implementation-dependent and in this case refers mainly to the time consumption of the function WaitGetPoses() in the OpenVR API [75], which will return "running start" milliseconds before the next VSync deadline, typically 2 ms [76].

that operation continues as soon as possible upon arrival of the required data.

The design allows for local latency operation, will not suffer frame drops due to synchronization issues, and achieves minimal latency because each step in the chain begins as soon as possible. Furthermore, because server and client are in sync, every frame rendered at the server will be displayed on the client. This is because the server will not start rendering until it receives new input, and the client will not send input until it has presented the previous frame. Network-induced *frame drops* due to packet loss cannot occur in this system. *Frame stalls*, on the other hand, will occur when the refresh interval deadline cannot be kept, for example, due to losses which may cause retransmissions. The display may in such cases show the same frame multiple times which may be experienced as a playback stall by the user. Note that the effect of a missed deadline is implementation-dependent though and not necessarily a frozen image if the VR client utilizes image warping techniques which may warp the last image for a new orientation [29].

Due to the guarantee of arrival of all frames (guaranteed by the application layer protocol, e.g., by reliable UDP or TCP), the system begins streaming with one I-frame and runs entirely on P-frames thereafter. This is not possible in an asynchronous system which must recover from losses by sending a new I-frame that does not reference any lost data. Data delivery is guaranteed in this work with an in-house implementation of reliable UDP [35] imposing strict control of the retransmission timeout. Finally, the delay that may be injected into the system is constant and always the length of one full frame, e.g., 11.1 ms at 90 fps. This may make it easier for future motion prediction algorithms to make accurate predictions when compared to variable or unknown latencies. In an asynchronous system, the delay may vary between zero and nearly a full frame time, which may reduce prediction accuracy.

1.2. Contribution and Structure. The synchronous remote VR system design is presented, of which a prototype was implemented and studied in various networks using various system

parameters and hardware. The research is meant not only to pave way for future work in the field of synchronous remote rendering for VR but also to produce knowledge that may be of use for related work in any remote rendering scenario. Our main contribution is the *design and evaluation of the synchronous remote VR system*. The remainder of this article is structured as follows: In Section 2, we discuss the state of the art of remote rendering for VR. Section 3 presents the design of the system and details of its implementation. Section 4 shows how the system performed in a number of tests with various networks and hardware. Finally, Section 5 summarizes the most important conclusions of the work, its limitations, and potential future work.

2. Related Work

Remote rendering consists of many subcategories and has been both available and studied extensively for decades. Some notable samples from history are presented in the following list with release dates in parenthesis. The list provides an introduction to the field but is however by no means complete. For a more comprehensive literature review, we refer to the following publications: a literature review regarding interactive remote rendering systems [1], a review on existing (in 2016) web-based visualization applications [36], a survey on mobile cloud computing [37], and a survey on platforms for interactive cluster-based multiscreen rendering [38].

(i) (1984) X Window [39] is one of the earliest examples of a remote renderer. It is a windowing system that by design consists of a server and client and therefore easily can be used remotely. X Window has been used extensively in research, for example, in [40], the authors found that applications that generate small datagrams when using X Window create severe overhead and may only utilize 25% of the maximum bandwidth. The authors suggest that updates should not be flushed unless necessary as this will generate smaller datagrams; updates should instead be buffered and sent in bulk when possible. The caveat of this is of course that it increases latency

(ii) (1995) Virtual Network Computing (VNC) [41] is a protocol for remote access to graphical user interfaces that is independent of operating and windowing system. It has been described as an improvement over X Window [41]. The authors of [42] quantified the user experience while using VNC and found that a low latency network is the most important factor, more so than bandwidth

(iii) (2000) OpenGL Vizserver [43] is a remote rendering system developed by Silicon Graphics Inc. that utilizes hardware acceleration with OpenGL. In [44], the authors used both VNC and OpenGL Vizserver to show how distributed teams can simultaneously view and steer simulations

(iv) (2002) Paraview [45] is an open-source application that enables interactive remote visualization of datasets. It has been widely used in the scientific community and is targeted towards viewing extremely large datasets that can be rendered by supercomputers or server clusters. For example, one study used ParaView to provide insights into the evolution of the early universe [46]. The dataset used in that study consisted of several hundred time steps of point simulation data, with each time-step containing approximately two million point particles [46].

(v) (2007) StreamMyGame [47] is a cloud gaming service that enables streaming through LAN or the Internet to Windows or Linux PC and Playstation 3

(vi) (2009) RealityServer [48] is a GPU-based cloud computing environment that enables photorealistic rendering through the web

(vii) (2010) OnLive [49] was a provider of cloud virtualization technologies with a cloud gaming service. OnLive was discontinued and sold to Sony in 2015 [49]. A study was conducted in 2011 that measured the latency of OnLive in comparison to StreamMyGame and found processing delays ranging between 110 and 221 ms in tests with OnLive and between 356 and 471 ms in tests with StreamMyGame [50]. The processing delay was described as the difference between player input and presentation of a corresponding frame [50], i.e., MTP

(viii) (2014) GamingAnywhere [51] is an open source cloud gaming system published in 2014. The publication of GamingAnywhere included performance comparisons with the OnLive system and showed significant improvement [51]. Another study that involved both OnLive and GamingAnywhere found that the player performance decreases by 25% with each 100 milliseconds of latency when using these services [52]

(ix) (2014) Playstation Now [53] is a cloud gaming service provided by Sony Interactive Entertainment that was launched for the first time in 2014

(x) (2015) Trinus VR [2] enables remote rendering for VR by streaming a PC game session to a phone or similar thin client. Additionally, it supports the streaming of non-VR games to be viewed in VR on thin clients

(xi) (2015) VRidge [4] provides the same functionality as previously described for Trinus VR

(xii) (2017) ALVR [3] is an open source remote VR display that streams VR from PC to Gear VR or Oculus stand-alone HMDs

(xiii) (2017) TPCAST [8] enables wireless VR for HTC Vive and Oculus Rift through remote rendering by streaming with WirelessHD in the 60 GHz band

(xiv) (2018) Vive Wireless Adapter [9] enables wireless VR for Vive headsets through remote rendering by streaming with IEEE 802.11ad in the 60 GHz band

(xv) (2019) Stadia [54] is a cloud gaming service provided by Google. Authors of [55] conducted a study where Stadia and Geforce NOW were studied in terms of their effects on global CO_2 emissions. The authors' analyses show that game streaming will cause significant increases in the energy and carbon footprint of games [55].

(xvi) (2020) Geforce NOW [56] is a cloud gaming service provided by Nvidia. A beta version of the service was made available in 2015 and a full release to the general public was made in 2020 [57]. A publication from 2016 studied the Quality of Experience (QoE) of Geforce NOW under different network conditions, such as bitrate, latency, and packet loss. The range of subjects and tested parameters were limited, but results indicate that there were hardly any statistically significant differences in QoE depending on the parameters, with one exception. QoE was significantly reduced when latency was introduced after the session started. A possible explanation for this may be that the service perhaps only initially estimated the latency for use with prediction. Thus, if the latency changed after the estimation, the predicted view for the rendering would be incorrect throughout the remainder of the session

(xvii) (2020) Xbox Cloud Gaming [58] is a cloud gaming service provided by Microsoft

(xviii) (2020) Luna [59] is a cloud gaming service provided by Amazon

(xix) (2020) Oculus Link [10] is a VR remote rendering system that operates through USB cable for use with Oculus Quest or Quest 2

(xx) (2020) CloudXR [60] is a solution by Nvidia for streaming VR, Augmented Reality (AR), and Mixed Reality (MR) content from any OpenVR XR application in the cloud, data center, or edge to tethered or untethered clients running Windows or Android platforms. Examples of supported devices include Vive or Valve Index headsets connected to Windows 10 PCs, Microsoft HoloLens 2, Oculus Quest, and Oculus Quest 2 [60]. CloudXR is in a closed beta at the time of writing (spring 2021).

In the following subsections, we will go into more details surrounding remote rendering for VR and review relevant publications in the field.

2.1. 360° Panorama VR. Remote rendering for VR has been studied for some time. Most solutions involve rendering to a panorama that covers every angle; this technique is used in 360° videos and has been widely used in the research community, commonly in conjunction with the game engine unity [61–64]. In [64], authors studied remote rendering of 360° VR by using two laptops and reported an end-to-end latency of 30 ms with a bitrate of 30 Mbps for a stereo 1080p resolution at 30 fps. The rendered panorama was split into slices, and only the ones deemed necessary depending on the current view-direction were transmitted; this reduced the bitrate by around 50%. A similar study was conducted where focus lied on bandwidth savings by only streaming the field of view (FoV) of the panorama image [63]. The authors of that study were able to stream 8 K 360-VR through a 4G network from a PC to a Gear VR phone and saved more than 80% bandwidth compared to sending the entire panorama. The term motion-to-update [63] was introduced and reported to range between 67.4 and 559.6 ms depending on resolution, network, and codec. Our understanding is that the motion-to-update is used here to more accurately describe the MTP of 360-VR due to considering server updates. In [61], authors studied bandwidth reduction further and found that by storing rendered images on the server and reusing them for clients with similar positions and view directions, a 90% reduction in server computation burden and 95% reduction in bitrate could be achieved. Finally, foveated streaming of 360° videos was studied in [62]. Bandwidth was not addressed in that work though; the aim was instead to reduce cognitive load and visual discomfort.

By using a panorama, new viewports caused by rotations of the headset can instantly be rendered by selecting a different part of the already acquired panorama image. This makes panoramas ideal for video in which just three degrees of freedom are used (3-DOF), i.e., rotations around the X-, Y-, and Z-axis. However, panoramas require more memory and more time to render, which makes them difficult to use with high framerates in real-time. To create a 360° view live, one may render a cube-map, which requires six cameras and therefore six rendering passes [64–66]. While some culling can be done in this process [65], cube map rendering can never be as efficient as rendering just the viewport. They are furthermore not efficient for 6-DOF (with additional translations along X, Y, and Z) and/or interactive scenes. When the content of the scene changes due to user interaction or by other events that occur in the scene, a new image showing the change must be downloaded unless it can be rendered on the client. Due to these issues, we render just the default viewport for where the user is currently looking, which is a common approach in low-latency implementations and no different from local VR in terms of rendered pixels.

2.2. Split Rendering. There are also split- or collaborative-rendering solutions [34, 65, 67]. By rendering the most time-sensitive or low-fidelity objects on the client and more demanding ones at the server, one may maintain both responsiveness and visual quality. This technique has been applied to remote rendering for VR, for instance, with the

background consisting of panorama images prerendered at the server at every position in the game along a spaced grid [34]. Another approach renders a cube map live as background at the server while interactive objects are rendered on the client [65].

While split-rendering solutions can achieve low latency for interactive components even in poor network conditions, we question if they may be too complex and difficult to scale for industry development. The inherent issue of split-rendering is that developers must be concerned with developing an end-user application that consists of both a server and client part, which implies additional costs. Though less critical, there is also the added hassle for the customer that must download and keep updated the unique client executable for each game. Additionally, the point of remote rendering is to achieve high quality visuals on low-powered devices; therefore, it is not ideal to render any object on the client as the visual quality of any such object will be limited to what the low-powered device can output. However, judging by current talks in the industry, we note that there seems to be a push towards split rendering and a belief that this will solve the current generation of remote VR [68]. Still, some are sceptical [69], and only time will tell which technologies will be accepted by the market. We expect that split rendering may be a compromise to make remote VR work on present-day hardware, but it is hardly the ultimate solution for future generations.

With the design proposed in this article, the client can be seen as a video player that may run any game that the server provides. The server process is unique per game though as remote rendering functionality must be added to the game engine and does not run as a separate process.

2.3. Nonpanoramic VR. Not using a panorama saves time and bandwidth, but it also puts immense requirements of low latency and stability on the VR system. A new image must be downloaded at a rate equal to the refresh-rate of the HMD. And if this cannot be achieved on time, the VR experience may be ruined. Building such a system can be approached easiest by having an external process copy the rendered images of the VR game, encode them, and send them across the network for decoding on the client. The pros of this approach are that it is simple and that the remote functionality can be added to any existing game without modification, because it is essentially a screen recorder. It may be difficult to optimize such a general process though, especially ensuring an optimal frame synchronization.

2.3.1. Screen-Recorder Approach. The authors of [70] built a short-range (LAN) remote rendering VR system with commodity hardware. Their system is able to capture images from Steam VR games, encode them with JPEG using a GTX Titan X GPU, and transmit them across a WiGig network. To measure the latency as perceived by the user, they record two HMDs side-by-side with a 240 fps camera, one of which contains the local VR rendering, and the other the remote. By comparing the difference of the HMD images, they conclude that the added latency is in the range of 12 ± 2 ms. The presented solution is a typical example of a screen-recorder approach where remote rendering functionality can be added to any existing game since it is an external process. This approach is economical and easy to bring to the market but also difficult to optimize since there is no low-level control of the game, as is revealed in the latency.

2.3.2. Game Integration and VSync Estimation. In [22], authors report on another short-range remote rendering VR system that also utilizes a WiGig connection but is integrated into the game. Encoding is done by using the Nvidia Video Codec SDK [71], H.264, and a GTX Titan X. They utilize four parallel encoder threads as well as four parallel decoders. They save time by starting the encoding of the first eye immediately after its rendering is complete. This means that the encoding of the first eye is done in parallel with the rendering of the second. Furthermore, the authors of the work propose a technique that involves estimating when the server should optimally start rendering in order to minimize the risk of missing the VSync deadline on the client. Unless some form of synchronization is applied, the waiting time on the client to the next VSync signal will drift every frame until a frame is missed in a repetitive pattern [22]. According to their experiments, the VSync estimation results in missing between 0.1 and 0.2% of all deadlines, which is on average one frame drop every 5–11 seconds at 90 fps. This is a significant improvement over a naive implementation though, between 5.3 and 14.3% of frames were dropped without the synchronization estimation. Finally, an end-to-end latency of 16 ms in software is reported; our understanding is that this is the average of the most demanding scene. In our work, we use a similar approach but utilize the proposed synchronous design which solves the synchronization problem completely.

2.3.3. Example from Industry. The synchronization issue and time drift can also be observed in a developer blog from the creators of VRidge [72]. They show how their previous implementation caused time drift and how an update mitigates the issue. Although less recurrent after the update, the problem seems not entirely solved as frames are still occasionally dropped also in the improved version. There are no exact numbers reported, but judging from the graph in [72], it seems around 0.3% of frames are still dropped in the updated version.

2.3.4. Measurements. Measuring the performance of a remote rendering system is difficult if it operates asynchronously. For example, if image quality should be measured, one may save every rendered frame on the server and client and compare them by using an objective image quality estimator after the streaming session has ended. Saving every frame to disk, for example, may reduce performance though and may therefore increase the number of missed frames on the client, causing invalid measurement results. A method for addressing this issue is known as slow-motion benchmarking and was applied and studied with a number of thin clients including VNC [73]. Slow-motion benchmarking is an inherent feature of the proposed synchronous architecture which will slow down and utilize retransmissions if needed to deliver to

the client all frames rendered by the server. There are therefore no lost frames, which makes image quality estimation simple. The delta-time of the server can also be controlled by setting it fixed to the framerate of the headset during testing. This will make sure that any game content behaves as expected even if the system is running in slow motion due to saving every frame as PNG to disk for example. While the system will not drop frames, misses of deadlines set by the framerate can of course still occur. This is an important measure and is recorded herein by querying the VR driver for the number of physical frame presentations every time a frame is completed on the client (see Section 4.3.1 for details). In summary, we propose that measurements such as image quality estimation will be easier to conduct with the synchronous design, as slow-motion benchmarking is an attribute of the architecture, and the client is guaranteed to receive every rendered frame; an example of this functionality can be observed in [74].

3. Materials and Methods

Two Windows applications were developed in C++. One of them runs on the client and is connected to Steam VR; this application has no game logic nor geometric rendering functionality. The client polls the HMD pose matrix from the Steam VR driver and sends this matrix to the server, which handles the game logic and renders the scene for two eyes according to the received pose matrix (see Figure 1 for an overview). The server encodes the resulting images of both eyes and sends them back to the client, which in turn decodes the images and finally sends them to the VR driver as OpenGL textures to be displayed in the headset. Note that audio transmission is outside the scope of this work and was not implemented in the prototype.

To ensure that the design is a viable option, we test the prototype system in a range of network environments using various hardware. The system is finally soak-tested in select environments by rendering 100000 frames per session. The soak-tests determine the ratio of stalls, which are compared to those reported in related work. The latter will allow to judge the performance of the synchronous design as compared to state-of-the-art approaches.

3.1. Synchronization. The system contains one main synchronization point between client and display which occurs when polling the HMD pose matrix from the VR driver and decides the framerate of the system. This method is called WaitGetPoses(), is part of the Steam OpenVR API [75], and blocks until a few milliseconds before the start of the next frame, which is referred to as "running start" [76]. Without this sync-point, the system would render without VSync as fast as possible, which would cause tearing.

Synchronization also happens between client and server; the server will block execution until a new input matrix arrives and immediately start rendering upon its arrival. The client will block rendering until two new images, one per eye, have been decoded by the two decoder threads. On completion, the rendering thread will submit the images to the VR driver after resizing (since lower than native resolutions are supported). After submitting to the VR driver, the client calls WaitGetPoses() to get a new pose matrix, and the loop is thus complete. If the loop can be completed in less than 11.1 ms, it will operate with zero added latency compared to local VR in the HTC Vive at 90 fps. If the deadline is exceeded, WaitGetPoses() will block for longer and make the system run at 45 fps with a 22.2 ms deadline. In such cases, one might instead run with a delay injection of one frame or lower the resolution to speed up codec processing to try and hit the deadline.

3.2. Codec. The server prototype is designed for use with Nvidia GPUs that have two hardware encoders, e.g., GTX Titan X and GTX 1080 (please refer to [77] for a full list). With two encoder chips, two encoding processes can be utilized fully in parallel, each encoding their corresponding eye simultaneously.

Encoding and decoding are done in C++ with NVENC and NVDEC from the Nvidia Video Codec SDK [71] and CUDA Toolkit. Images are encoded using H.264 with NV12 as video surface format and YUV420 chroma subsampling. An infinite group-of-pictures is used where the frame interval is set to start with one I-frame and thereafter use only P-frames throughout the entire session. This saves bandwidth since the more demanding I-frames can be avoided, but it also makes the solution sensitive to loss, which is why a reliable application layer protocol must be used, for example, reliable UDP or TCP.

The framerate is set to the native rate of the HTC Vive, 90 fps. The VBV buffer size is set to the size of one average frame, as is recommended for low latency operation in the NVENC documentation [71]. Rate control mode is set to NV_ENC_PARAMS_RC_CBR_LOWDELAY_HQ. The size of the encoded image can be changed during execution (see Section 3.4 for details on image sizes). The average bitrate is set to 8 Mbps per encoder for a total of 16 Mbps. This bitrate was selected as it provided sufficient quality at maximum resolution according to SSIM measurements. It should be noted though that between 10 and 16 Mbps per eye should be used depending on expected levels of motion and available resources [74]. Thus, the bitrate used in this work is slightly below recommended levels. This has shown to have a negligible impact on time-consumption though, which is the main focus in this work. For a study relating bitrate requirements to image quality, the reader is referred to [74].

Other than the mentioned modifications, the encoder and decoder have the same configurations as the low latency encoder and decoder samples from the Nvidia Video Codec SDK [71].

3.3. Network. UDP is used to transmit video and input data; two encoder threads transmit video data in parallel from server to client. Input data is transmitted in one packet per frame sent from client to server. In order to guarantee delivery of all UDP packets, we implement our own version of reliable UDP (RUDP) [35]. The RUDP implementation is inspired by Valve's GameNetworkingSockets [78] and the ACK-vector of DCCP [79]. We chose to make our own implementation to make sure we have full control and are

able to optimize it as much as needed. The implementation is designed to quickly respond to loss by being highly sensitive to variations in delay. If a packet is expected and a higher delay than usual is detected, the system will estimate that the packet is lost and triggers a retransmission. In networks with little variation in delay, redundant retransmissions remain rare, and response time on loss remains low. However, redundant retransmissions may occur more often in networks with a large variation in delay.

A detailed presentation of the protocol is outside the scope of this work since the synchronous design does not rely on it per se. TCP with disabled Nagle's algorithm (Nagle's algorithm is a method for TCP to reduce the number of small packets and thus increase bandwidth efficiency. It does so by buffering data that may be intended to be sent immediately in separate packets, this introduces delays in TCP [80]) for example can be used as well or any other RUDP implementation. The effect of such a choice on delay distributions in remotely rendered VR is yet to be studied.

3.4. Dynamic Resolution. The server allocates five frame buffers of different sizes for testing the impact of resolution switches. The resolution levels and sizes are presented in Table 1. The resolutions were arbitrarily determined by subtracting the native resolution by 20% in width and height per level and additionally rounding up to the closest multiple of 16 to avoid cropping in the decoder.

3.5. Test Setups. Experiments were conducted using Windows PC on both server and client. Note that this is naturally not the main target device of future real-world scenarios, and most of the tested client devices could even run desktop VR on their own GPUs. However, they were used in order to gain more control of the testing environment and are sufficient for a proof-of-concept of the proposed method. Future work involves developing a client that can run on, e.g., a phone or stand-alone HMD equipped with hardware decoding capability.

The experiments were conducted in four different environments (see Table 2). The first is referred to as the *apartment LAN*, where both client and server were on the same LAN/WLAN. The purpose of the apartment experiments was to see how the remote VR solution can operate in a best-case scenario on strong hardware in short-range network conditions, i.e., fog rendering. In this use case, the consumer already has local access to a strong computer for example at home or at an Internet cafe and streams VR to a thin client for example to increase mobility or reduce cost. As previously mentioned, some examples of this from the current market are the low-end remote VR options that stream from PC to phone [2–4]. Another similar case is the Oculus Link [10].

The second testing environment was the *campus intranet* at the Blekinge Institute of Technology (BTH), where client and server were separated by approximately 50 km of optical fiber between two campuses with 1 Gbps bandwidth. These experiments give insight into how the solution may perform in larger networks with longer range but with good routing, i.e., edge rendering. The consumer would in such a scenario not have immediate access to the server, but be closely connected to it through a private intranet for example from home or at an Internet cafe while using a thin client. The intranet would then provide the remote VR service from a remote but optimally routed server with more stability and speed than through the public Internet. In this use-case, for example, a set-top-box could be connected with optimal routing to the remote server, and a VR headset could be plugged into the box or be connected wirelessly. Trials of such a case were conducted by HTC in 2017 [81].

Table 1: Resolution settings.

Level	0	1	2	3	4
Percent	100%	80%	60%	40%	20%
Pixels	1088×1200	880×960	656×720	448×480	224×240

Two experiments were conducted across the *public Internet* on fiber and 4G. In these tests, the client PC was located in the apartment and the server at the campus approximately 50 km away. These tests were included to reveal to which degree typical contemporary public networks may be able to support this type of remote rendering for VR. A 100/100 Mbps subscription was used for the fiber connection, and an Asus Zenfone 3 Zoom phone was tethered to the client by USB in the 4G testing. In the fiber connection, devices were separated by 18 network hops. This case tests public cloud rendering, which is not only the most flexible option but also most likely to fail due to the high Round Trip Time (RTT) of the public Internet.

Additionally, two experiments using 5G were conducted indoors at the Luleå University of Technology (LTU). The tests show the applicability of this type of remote rendering in an early test bed for the upcoming 5G technology. This use case refers to mobile edge rendering where the rendering server is closely connected to the mobile access point. The 5G experiments were run with an edge-server inside the LTU 5G network [82] connected to a local breakout core (see Figure 2 for details). Four hops were registered between the client and server. An early-release 5G USB modem provided the client laptop with connectivity to the cellular network.

No mobility was tested in any of the wireless networks. The client was only connected to one base station/WiFi-router and did not move to another, in any of the wireless conditions. Also, with the exception of 4G which was outside our control, the client device was also the only device connected to the serving base station/WiFi router during testing.

All experiments were conducted with an HTC Vive VR headset that lied perfectly still on a flat surface while connected to the client computer. To emulate user headset movements in a reproducible way, the HMD pose matrix was rotated in software with an angle that ranged between -180° and +180° at a speed of 90° per second.

4. Results and Discussion

The results are split up in three test cases. Two relatively short tests were conducted, in which the resolution was altered in run-time in local latency mode and while using

Table 2: Environment descriptions.

Network	Server device	Client device	Distance (bee-line)	Use case
Apartment LAN	Laptop GTX 1080	PC RTX 2070	2 m	Fog rendering
Campus intranet	PC GTX Titan X	Laptop GTX 1080	50 km	Edge rendering
Public Internet	PC GTX Titan X	PC RTX 2070	50 km	Cloud rendering
Indoor 5G	PC GTX 1080	Laptop Quadro M1200	100 m	Mobile edge rendering

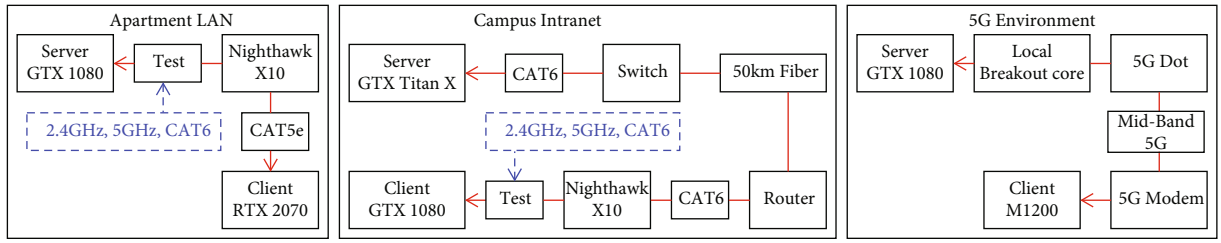

Figure 2: Layout of apartment, intranet, and 5G experiments.

delay injection, respectively. These tests reveal to which degree the codec time consumption affects the rendering loop, and additionally, what the impacts of live resolution switches are on the immediate time-consumption of the following frame. The two tests furthermore provide valuable insight into how the synchronous rendering loop performs in the various networks. This determines whether the networks in question are appropriate for this type of remote rendering. Finally, soak tests were conducted using appropriate settings and environments to reveal the resulting delay distributions and the ratio of stalls over time.

4.1. Resolution Tests in Local Latency Mode. The resolution tests were conducted in all environments and ran for a total of 2500 frames per physical medium. The resolution was reduced one level (20%) after each 500-frame interval.

Results show that the decoder time consumption is increased in the moment of a resolution switch, ranging from 1.3 ms to 10.3 ms extra depending on GPU. The encoder suffers no penalty from resolution switches though, but the bitrate is severely increased when such switches take place. Thus, the price to pay for a resolution switch is not only a temporary increase in decoder time consumption but potentially also in video network time consumption, depending on available bandwidth. In our experiments though, we see only negligible increases in video network time consumption at the moment of switches because the available bandwidth can handle the increase. Ordinary video frames generally required around 22 kB in these experiments; frames carrying resolution switches required around 50–60 kB.

Time plots of all resolution experiments are shown in Figure 3. This figure shows the time consumption of each component during execution. The total time consumed from input to display for each frame is shown with the white (Total) line in graphs and takes into account any missing part such as display synchronization and delays. A similar line (blue) shows the time spent between frame presentations on the client (*delta time*). This time controls the framerate and is the speed at which the game logic is updated. The shown delta time measurement is the average of three connected frames to reduce noise. Ultimately, it is the total time that accurately shows frame stalls, e.g., at spikes. In Figure 3, the total time and delta time are generally the same because no delay injection was used except for in the 5G case, which was included in that figure to save space. The difference between total time and delta time becomes clear in the following Figure 4 where the two are always separate due to delay injection.

4.1.1. LAN. Three experiments with resolution reductions were conducted in the apartment LAN environment, where the connection between server and router consisted of either 2.4 or 5 GHz WiFi or an Ethernet CAT6 cable. Results from 2.4 GHz show a stable 45 fps followed by a struggle to settle at a constant framerate (see the left-hand side of Figure 3). The 5 GHz experiment shows a relatively stable run starting at 45 fps at 100% resolution followed by 90 fps at lower resolutions. At 100% resolution though, there is an issue that we refer to as a *transition zone*, where the framerate should ideally stay at a stable 45 but keeps reaching 90 in recurring bursts. CAT6 provided a stable local latency experience at 90 fps and 100% resolution. The test with CAT6 contained a single frame stall that occurred in the moment of resolution switch from 100% to 80% due to the temporarily increased time consumption of the decoder.

4.1.2. Intranet. Results show that the wireless media in the intranet supported 45 fps with recurring frame stalls followed by a transition zone at 40% (5 GHz) and 20% resolution (both). The CAT6 connection supported 45 fps without stalls at full resolution (level 0), entered a transition zone at 80%, and ran with recurring frame stalls at the other levels.

The intranet was not able to support a stable local latency remote rendering at native framerate regardless of resolution and physical medium. A stable experience without frame stalls was achieved only at 45 fps with CAT6 and 5 GHz. Note in the graph for CAT6 at 20% resolution that several milliseconds are available before reaching 11.1 ms, but this was

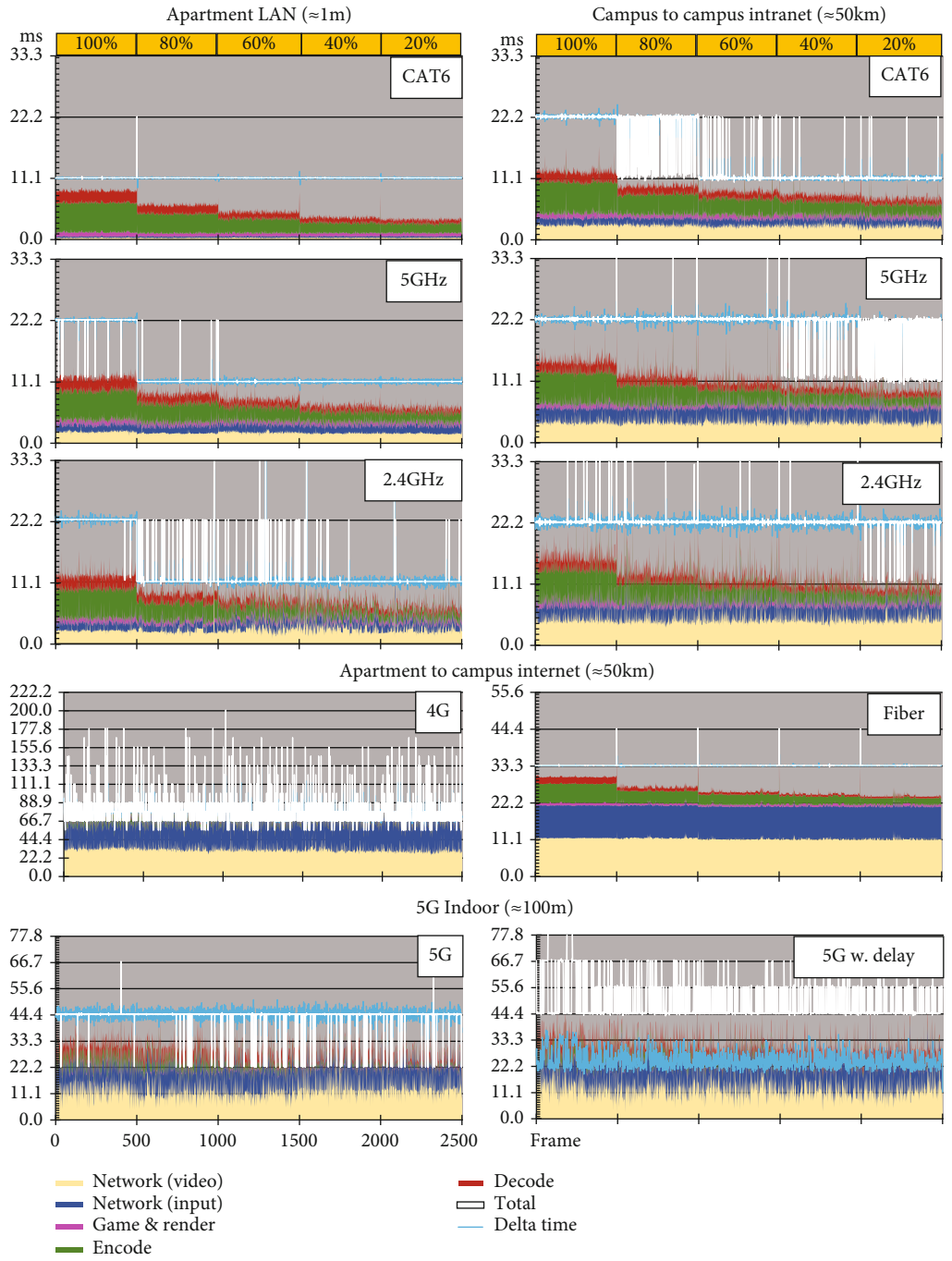

Figure 3: Examples of time consumption of the tasks performed each frame in the resolution experiments. Resolution was reduced one level at each 500-frame interval, see yellow bars and x-axis markers. "Total" shows the total time consumption in software of each frame from the client perspective; "delta time" shows time consumed between presenting two frames. The two overlap where there is no delay injection.

evidently not sufficient to compensate for the jitter that would occur occasionally in this network.

4.1.3. Internet. Two experiments with resolution reductions from the Internet environment are presented in the lower parts of Figure 3. Results show that resolution did not matter in these cases as the majority of time consumption was due to network latency and jitter. The 4G connection supported an unsteady average of around 10 fps. The fiber connection allowed for a stable connection with frame stalls only occurring at resolution switches, but could maintain only 30 fps due to the high latency of the public Internet. This shows the importance of using private infrastructure if one wants to enable synchronous remote rendering. Geographically, there is a few hundred meter difference in client position in the intranet and Internet tests, but the RTT was around 20 ms in the Internet connection and 2–3 ms on the intranet.

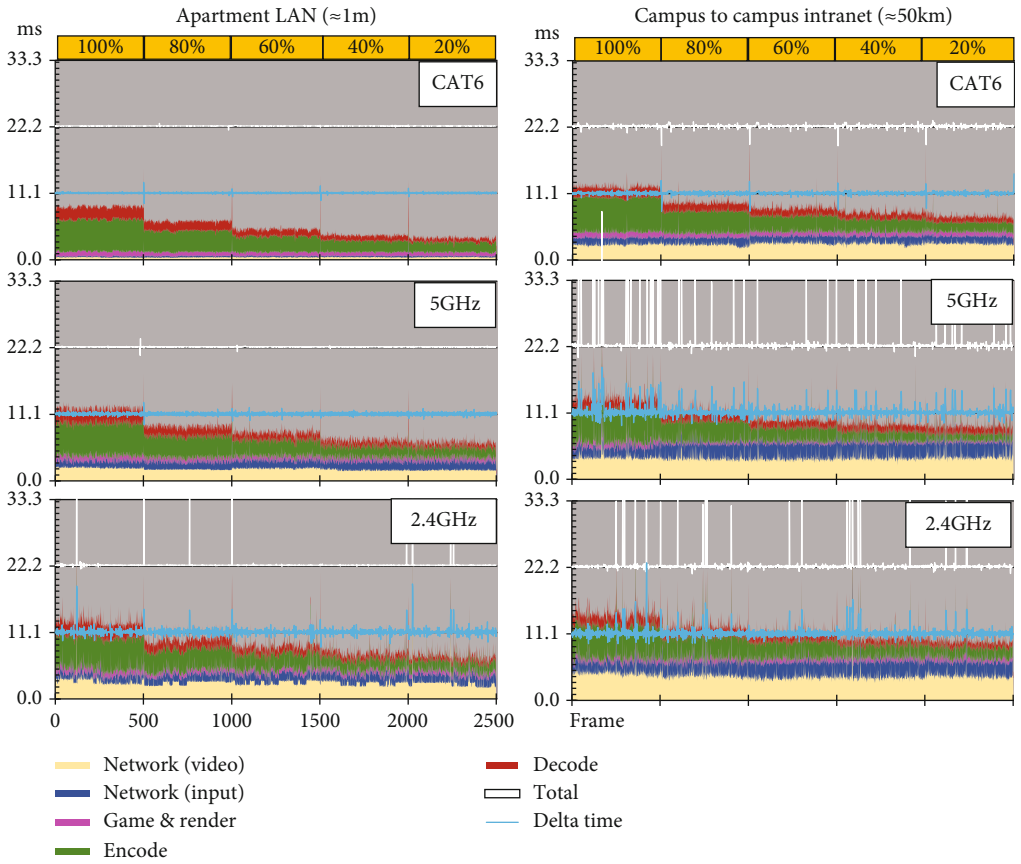

Figure 4: Examples of time consumption of the tasks performed each frame in the delay experiments. Note how total time and delta time are now separate due to a delay buffer.

4.1.4. 5G. In similarity with the Internet tests, the resolution reduction tests over the 5G connection are limited by the RTT. As seen in Figure 3, the majority of the time consumption comes from the network components, i.e., the 5G USB modem and the radio interface itself. Typically, the physical radio layer measures below an average of 14 ms RTT in real-world tests with similar midband Time Division Duplex (TDD) setups using a 3 : 1 downlink/uplink ratio. However, devices such as the USB-modem may add additional delays of around 5–6 ms per direction. This delay is expected to be reduced in upcoming, more optimized user devices. To achieve fiber-like behavior with submillisecond delay on each radio link Ultra-Reliable Low-Latency Communication (URLLC) [83] would have to be used, which is not yet available in this test network.

4.2. Resolution Tests with Delay Injection. The prototype supports injection of one-frame delay. Tests with this functionality were run in the apartment, intranet, and 5G settings with the same parameters as the previously described resolution tests. Except for the test on 5G which is shown in Figure 3 to save space, results from delay tests are shown in Figure 4. Adding a delay of one frame made the system run with little to no frame stalls in all media on LAN. 12 out of 2500 frames were stalled in case of 2.4 GHz but none in case of CAT6 and 5 GHz. Adding a delay in the intranet on CAT6 resulted in zero stalls at a constant 90 fps as well. However, recurring stalls still occurred in both wireless media on the intranet, note the white spikes in total time consumption in Figure 4. These spikes are likely a result of the congested 2.4 and 5 GHz bands in the campus area. Similarly, 5G was not improved significantly by a delay due to the high RTT and spiky network time consumption.

4.3. Soak Tests. Finally, soak tests were conducted in the apartment LAN and intranet environments. In these tests, 100000 frames per medium were recorded at the highest resolution that enabled 90 fps in the resolution tests of the respective medium. On LAN, these were CAT6 (100%), 5 GHz (60%), and 2.4 Ghz (20%). In the intranet, however, only CAT6 was soak-tested since the wireless media failed to maintain a stable framerate both with and without delay in this environment. Two soak tests were conducted with CAT6 on the intranet, one with 20% resolution and one with one frame delay at 100% resolution.

4.3.1. Soak Test Results. Figure 5 shows the Complementary Cumulative Distribution Function (CCDF) of time consumption of network components. The plots in the figure show the percentage of samples that exceed the time consumption shown on the abscissa, up to 20 ms. The CCDF reveals the necessary budgets for each component depending

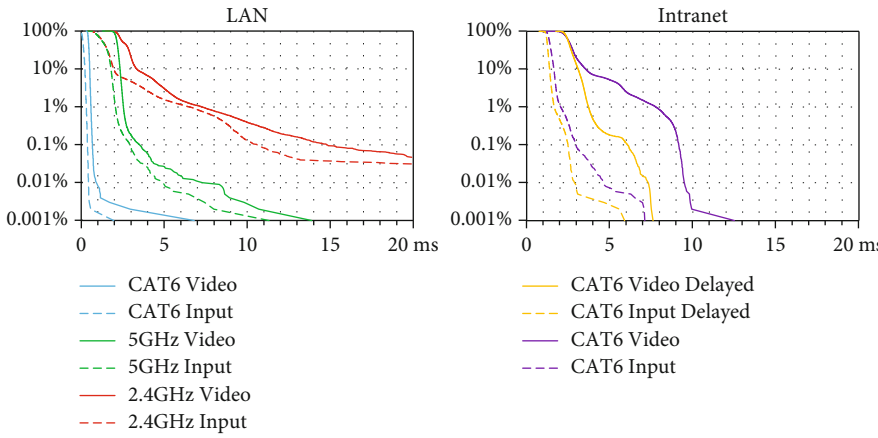

Figure 5: CCDF of network component time consumption in LAN. Each curve is based on 100000 samples.

Table 3: Soak test statistics (V: video; I: input).

	LAN						Intranet				Unit
Test	A		B		C		D		E		—
Medium	CAT6		5 GHz		2.4 GHz		CAT6		CAT6		—
Delay	No		No		No		No		Yes		—
Resolution	100%		60%		20%		20%		100%		—
Bitrate	16.2		16.3		15.0		15.0		16.0		Mbps
Framerate	89.52		89.46		86.61		87.34		89.53		fps
Total	11.188		11.195		11.563		11.497		22.369		ms
Stalls	0.009%		0.058%		3.220%		2.355%		0.002%		—
Overrun	V	I	V	I	V	I	V	I	V	I	
Max	6.8	2.0	13.9	11.3	105.5	118.7	12.5	7.1	7.6	5.9	ms
0.01%	0.9	0.4	7.3	4.9	38.1	117.7	9.5	4.6	7.3	2.9	ms
0.1%	0.7	0.4	3.5	2.9	14.8	10.6	9.2	3.0	6.1	2.5	ms
1%	0.6	0.3	2.5	2.1	7.3	6.5	7.8	2.1	3.7	1.7	ms
10%	0.5	0.2	2.3	1.8	3.3	2.0	3.6	1.6	3.1	1.4	ms
Min	0.3	0.0	1.3	0.4	1.5	0.1	0.9	0.7	1.8	0.7	ms

on configuration and acceptable ratio of budget overruns; the data is summarized in Table 3. The table provides a detailed overview by showing the budget requirement of both network components at each labeled percentage. For example on LAN with CAT6 (test A), a budget of 0.9 ms covers the requirement of the video component in all but 0.01% of frames. To maintain the same level of overruns in 5 GHz at 60% resolution (test B), a budget of 7.3 ms would be required. In addition to budgets, the table shows how many frame stalls occurred during the test, e.g., 0.009% of frames in CAT6 on LAN. To provide maximum accuracy, we used Steam VR to measure the number of stalls. They were determined by querying the VR compositor through the OpenVR API for the variable "Compositor_FrameTiming::m_nNumFramePresents" at each frame and adding its value to the total number of stalls. A number larger than one is regarded as one or more frame stalls, because the same frame was then presented multiple times.

The table further shows the average framerate, bitrate, and total time consumption per frame. Note that the true framerate of the HTC Vive is not exactly 90 fps but 89.53 fps [84]. Thus, the framerate of test A, 89.52 fps, is close to optimal but suffered nine frame stalls which resulted in a lower average framerate. Test E shows an optimal framerate of 89.53 fps though when rounded to two decimals.

The bitrate is also affected by the number of stalls, because when framerate is lowered by half, so is the effective bitrate. This can for example be seen in test C (2.4 GHz) where the average bitrate is 15.0 Mbps due to a relatively high number of frame stalls of 3.220%. The effective bitrate is also affected by the number of redundant UDP retransmissions, which is why it can reach above the set 16.0 Mbps.

The average total time consumption per frame is also shown, which should optimally be 11.169 ms due to the true framerate of 89.53 fps.

4.3.2. Soak Test Summary. The prototype achieved low frame-stall ratios on LAN at local latency, 0.009% with CAT6 at full resolution, and 0.058% with 5 GHz at 60% resolution. On the intranet, a CAT6 connection with a delay

of one frame was able to provide a stream with 0.002% stalls. These are improvements over previous work [22] which reported a frame missing rate of 0.1–0.2% as well as the industry example that showed 0.3% frame drops [72].

5. Conclusions and Outlook

We have shown one way to design a synchronous remote rendering VR system that solves the synchronization problem between client and server and thus eliminates a source of frame stalls. By testing an implementation of this design, we have quantified the ratio of stalls, identified, and shown some conditions under which it may be operated and what requirements it poses on supporting hardware. We conclude that the architecture can run well with a latency equal to local VR, i.e., in local latency mode. However, a stable network with low latency must be used in order to stream in that mode reliably. The campus intranet we used did not suffice, and neither did the public Internet nor the early 5G test network. Only on LAN could a stable local latency operation be achieved at the native resolution and framerate of the HTC Vive. On LAN/WLAN, the network was not the major issue, but the encoder and decoder time consumption. Indeed, apart from network delay spikes, the encoder time is generally the bottleneck of the system, especially at high resolutions. Decoding in our experiments generally took less than 2 ms while encoding could take up to 6 ms. Reducing the resolution to 20% of native size would in general reduce encoding to around 1.5 ms and decoding to around 1 ms, which was shown to be useful to meet the deadline in some scenarios, but will of course have a negative impact on video quality.

In case the local latency deadline cannot be kept, the architecture supports injection of full frames of latency, i.e., an added latency of 11.1 ms per frame at 90 fps. Delay injection proved to make streaming more reliable in environments where this was sufficient to compensate for the jitter of the network. We expect that the precise addition of latency may help when adding future motion prediction.

5.1. Limitations. We have not taken into account two parts that are outside our scope, but are otherwise important parts to consider in remote rendering for VR. They are the following:

Simple Game: The game used in our experiments is a simple 3D scene in which boxes surround the player in a small room. This scene and game logic are not complex and often required less than 1 ms to compute and render. In a real-world scenario, the game would likely require at least a few milliseconds on average, which would have to be taken into account when calculating budgets.

No Audio: We have ignored audio in the implementation of this system. In a real scenario, more data containing the audio would have to be sent from the server to client each frame.

Furthermore, to account for lens distortion in the HTC Vive, it is advisable to render at 1.4x the native resolution (1512×1680 instead of 1080×1200). However, to avoid transmitting such large images, at 100% resolution in tests. Rendering was conducted at 1.4x the size on the server but lens distortion was applied before encoding, thus reducing the size of the images and the amount of data processed by the codec and network. To do this though, one must also disable the lens distortion that is normally applied in the VR driver on the client. This will also disable any built-in image warping functionality, which exists to reduce the perceived rotation and/or translation latency. Thus, it would be better to just transmit non lens-distorted images at the native size and accept a loss in pixel density instead of disabling warping. Whether lens distortion is applied on server or client does not visibly affect the results presented in this work though. But future work may address this issue for example by using foveated video processing [85, 86] which is how it is addressed in the Oculus Link [10].

5.2. Future Work. Local latency mode, while difficult to support with a codec, may be perfectly viable if the time for encoding and decoding can be neglected, which may be the case if for example a very high bandwidth is available, requiring little to no compression. This is the case with the wireless VR adapters that are available on the market today, which typically utilize the 60 GHz band with for example 802.11ad [87].

Although 802.11ad in theory supports up to 8 Gbps [88], it does not support compression-less remote rendering for the resolutions and framerates of current generation HMDs, e.g., the Valve Index. To solve this issue, one may look forward to the next generation of 60 GHz WiFi known as 802.11ay [88]. 802.11ay is at the time of writing in development [89] and aims to support up to 100 Gbps [88]. The article that announces 802.11ay puts the VR use-case in focus, part of its abstract reads:

"While enabling multi-Gb/s wireless local communications was a significant achievement, throughput and reliability requirements of new applications, such as augmented reality (AR)/virtual reality (VR) and wireless backhauling, exceed what 802.11ad can offer." [88].

The prototype was so far tested with 802.11 ac (denoted as 5 GHz for short) and 802.11n (2.4 GHz). How will the architecture perform in more sophisticated wireless networks? Similar measurements could be conducted using for example 802.11ad, 802.11ay, 802.11ax, or 5G with URLLC enabled [90].

Applying lens distortion on the server is not optimal due to the interference with image warping techniques. Encoding and transmitting supersampled images is on the other hand time-consuming for the codec due to the data size. Smaller images may therefore be used in conjunction with superresolution [91] or foveated encoding [36], the latter of which is already applied in the Oculus Link [55]. How much speed-up can we get from these technologies and to what level of detriment to the image quality?

The presented work determined that the ratio of frame stalls could be reduced with a synchronous design. What ratio of frame stalls are noticeable to users though? Subjective QoE-studies may be conducted to answer this question.

Data Availability

Data and code are available at https://github.com/kelkka/RemoteRenderedVR. If the linked page should be unavailable, data and code will be made available on request through the email listed on the title page (viktor.kelkkanen@bth.se).

Conflicts of Interest

In the tests with a 5G connection, we have used a 5G test-site using equipment developed by Ericsson. This may be perceived as a conflict of interest since one of the authors is employed at this company.

Acknowledgments

Thanks to Mattias Schertell and Siamak Khatibi for letting us borrow their server with the GTX Titan X GPU. Thanks to Björn Mattsson for setting up the intranet network environment at BTH. Thanks to the anonymous reviewers for your feedback. Thanks to Jesper Gladh for running around at LTU and doing the 5G measurements. And finally, thanks to Hans-Jürgen Zepernick and our colleagues in the ViaTecH project for fruitful discussions and feedback. This work was supported in part by the KK Foundation, Sweden, through the project "ViaTecH" under contract number (20170056).

References

[1] S. Shi and C.-H. Hsu, "A survey of interactive remote rendering systems," *ACM Computing Surveys*, vol. 47, no. 4, article 57, pp. 1–29, 2015.

[2] S. L. Odd Sheep, "Trinus virtual reality," 2015, May 2020, https://www.trinusvirtualreality.com/.

[3] Polygraphene, "Air light VR," 2017, May 2020, https://github.com/polygraphene/ALVR.

[4] Riftcat, "VRidge-play PC VR on your cardboard," 2015, May 2020, https://riftcat.com/vridge.

[5] Google, "Cardboard," 2014, May 2020, https://arvr.google.com/cardboard/.

[6] Samsung, "Gear VR," 2015, May 2020, https://sv.wikipedia.org/wiki/Samsung_Gear_VR.

[7] Wikipedia, "Google Daydream," 2016, May 2020, https://en.wikipedia.org/wiki/Google_Daydream.

[8] TPCAST, "Wireless adapter for VIVE," 2017, April 2020, https://www.tpcast.cn/index.php?s=/Front/Goods/index/good/10228/l/en-us.

[9] Valve, "VIVEWireless adapter," 2018, June 2020, https://www.vive.com/us/wireless-adapter/.

[10] Oculus, "How Does Oculus Link Work? The Architecture, Pipeline and AADT Explained," 2019, February 2021, https://developer.oculus.com/blog/how-does-oculus-link-work-the-architecturepipeline-and-aadt-explained/.

[11] M. Abdallah, C. Griwodz, K.-T. Chen, G. Simon, P.-C. Wang, and C.-H. Hsu, "Delay-sensitive video computing in the Cloud," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 14, no. 3s, article 54, pp. 1–29, 2018.

[12] M. Abrash, "Latency - the Sine Qua Non of AR and VR. Valve," 2012, January 2020, http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/.

[13] A. Seam, A. Poll, R. Wright, J. Mueller, and F. Hoodbhoy, *Enabling Mobile Augmented and Virtual Reality with 5G Networks*, AT&T, 2018, January 2020, https://about.att.com/ecms/dam/snrdocs/Foundry%20ARVR%20Public%20Whitepaper%20.pdf.

[14] P. Jombík and V. Bahýl, "Short latency disconjugate vestibulo-ocular responses to transient stimuli in the audio frequency range," *Journal of Neurology, Neurosurgery & Psychiatry*, vol. 76, no. 10, pp. 1398–1402, 2005.

[15] 3GPP, *Virtual reality (VR) media services over 3GPP (Release 16)*, 2018, Technical Report, TR 26.918 V16.0.0.

[16] M. Abrash, *What VR Could, Should, and Almost Certainly Will Be within Two Years*, Valve, 2014, March 2020, https://media.steampowered.com/apps/steamdevdays/slides/vrshouldbe.pdf.

[17] G. S. M. A. Future Networks, "Cloud AR/VR Whitepaper," GSM Association, 2019, January 2020, https://www.gsma.com/futurenetworks/wiki/cloud-ar-vr-whitepaper/.

[18] S. Solotko, *The Instantaneous Cloud: Emerging Consumer Applications of 5G Wireless Networks*, TIRIAS, 2018, January 2020, https://www.tiriasresearch.com/downloads/the-instantaneouscloud-emerging-consumer-applications-of-5g-wireless-networks/.

[19] E. Cuervo, K. Chintalapudi, and M. Kotaru, "Creating the perfect illusion: what will it take to create life-like virtual reality headsets?," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications (HotMobile '18)*, pp. 7–12, Tempe, Arizona, USA, 2018.

[20] Valve, "Steam hardware & software survey: January 2020," 2020, February 2020, https://store.steampowered.com/hwsurvey/.

[21] O. Kreylos, "Lighthouse tracking examined," 2016, January 2020, http://doc-ok.org/?p=1478.

[22] L. Liu, R. Zhong, W. Zhang et al., "Cutting the cord: designing a high-quality untethered VR system with low latency remote rendering," in *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pp. 68–80, Munich, Germany, 2018.

[23] O. Kreylos, "Oculus Rift DK2's tracking update rate," 2016, January 2020, http://doc-ok.org/?p=1405.

[24] A. Hogge, *Controller to Display Latency in Call of Duty*, Activision Central Technology, 2019, August 2020, https://twvideo01.ubm-us.net/o1/vault/gdc2019/presentations/Hogge_Akimitsu_Controller_to_display.pdf.

[25] U. H. List, "Nonlinear Prediction of Head Movements for Helmet-Mounted Displays," Technical paper. Air Force Human Resources Lab Brooks AFB TX, San Fransisco, USA, 1983.

[26] M. Russell and I. I. Taylor, *Virtual Reality System Concepts Illustrated Using OSVR*, A K Peters/CRC Press, New York, USA, 2019.

[27] J. Carmack, "Latency mitigation strategies," 2013, September 2020, https://danluu.com/latency-mitigation/.

[28] R. H. Y. So and M. J. Griffin, "Compensating lags in head-coupled displays using head position prediction and image deflection," *Journal of Aircraft*, vol. 29, no. 6, pp. 1064–1068, 1992.

[29] J. M. P. van Waveren, "The asynchronous time warp for virtual reality on consumer hardware," in *Proceedings of the 22nd*

ACM Conference on Virtual Reality Software and Technology, pp. 37–46, Munich, Germany, 2016.

[30] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3D warping," in *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, p. 7, Providence, Rhode Island, USA, 1997.

[31] E. M. Peek, B. C. Wünsche, and C. Lutteroth, "Image warping for enhancing consumer applications of head-mounted displays," in *Proceedings of the Fifteenth Australasian User Interface Conference*, pp. 47–55, Auckland, New Zealand, 2014.

[32] F. A. Smit, R. van Liere, and B. Fröhlich, "The design and implementation of a VRArchitecture for smooth motion," in *Proceedings of the 2007 ACM Symposium on Virtual Reality Software and Technology*, pp. 153–156, Newport Beach, California, 2007.

[33] S. M. LaValle, *Virtual Reality*, Cambridge University Press, University of Oulu, Finland, 2019, http://vr.cs.uiuc.edu/.

[34] Z. Lai, Y. C. Hu, Y. Cui, L. Sun, N. Dai, and H.-S. Lee, "Furion: Engineering High-quality immersive virtual reality on today's mobile devices," *IEEE Transactions on Mobile Computing*, vol. 19, no. 7, pp. 1586–1602, 2020.

[35] T. Bova and T. Krivoruchka, *Reliable UDP Protocol*, IETF, 1999, August 2020, https://tools.ietf.org/html/draft-ietf-sigtran-reliable-udp-00.

[36] F. Mwalongo, M. Krone, G. Reina, and T. Ertl, "State-of-the-art report in webbased visualization," *Computer Graphics Forum*, vol. 35, no. 3, pp. 553–575, 2016.

[37] H. T. Dinh, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[38] O. G. Staadt, J. Walker, C. Nuber, and B. Hamann, "A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering," in *Proceedings of the Workshop on Virtual Environments*, pp. 261–270, Zurich, Switzerland, 2003.

[39] X Org Foundation, "X Window System," 1984, February 2021, https://en.wikipedia.org/wiki/X_Window_System.

[40] M. R. van der Werff, M. H. K. de Grijp, S. G. Vrind, and B. R. Haverkort, "The X Window System over ISDN-a performance study," in *Tenth UK Teletraffic Symposium, 10th. Performance Engineering in Telecommunications Network*, pp. 1–7, Martlesham Heath, UK, 1993.

[41] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33–38, 1998.

[42] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying interactive user experience on thin clients," *Computer*, vol. 39, no. 3, pp. 46–52, 2006.

[43] SGI, "OpenGL Vizserver," 2000, February 2021, https://web.archive.org/web/20000817094004/http://www.sgi.com/software/vizserver/overview.html.

[44] J. Brooke, T. Eickermann, and U. Woessner, "Application steering in a collaborative environment," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing - SC '03*, p. 61, Phoenix AZ USA, 2003.

[45] S. N. Laboratories and K. Inc, "ParaView," 2002, February 2021, https://gitlab.kitware.com/paraview/paraview.

[46] P. Navratil, J. Johnson, and V. Bromm, "Visualization of cosmological particle-based datasets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1712–1718, 2007.

[47] StreamMyGame, "About," 2007, March 2021, https://web.archive.org/web/20080927072853/http://www.streammygame.com/smg/modules.php?name=About.

[48] Nvidia, "Reality server," 2009, February 2021, https://web.archive.org/web/20091025074652/http://www.nvidia.com/object/realityserver.html.

[49] OnLive, "OnLive," 2010, February 2021, http://onlive.com/.

[50] K.-T. Chen, Y.-C. Chang, P.-H. Tseng, C.-Y. Huang, and C.-L. Lei, "Measuring the latency of cloud gaming systems," in *Proceedings of the 19th ACM International Conference on Multimedia (MM '11)*, pp. 1269–1272, Scottsdale, Arizona, USA, 2011.

[51] C.-Y. Huang, K.-T. Chen, D.-Y. Chen, H.-J. Hsu, and C.-H. Hsu, "GamingAnywhere," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 10, no. 1s, article 10, pp. 1–25, 2014.

[52] M. Claypool and D. Finkel, "The effects of latency on player performance in cloud-based games," in *2014 13th Annual Workshop on Network and Systems Support for Games*, pp. 1–6, Nagoya, Japan, 2014.

[53] Sony, "PlayStation Now," 2014, February 2021, https://www.playstation.com/en-us/ps-now/.

[54] Google, "Stadia," 2019, February 2021, https://stadia.google.com/.

[55] M. Marsden, M. Hazas, and M. Broadbent, "From one edge to the other: exploring gaming's rising presence on the network," in *Proceedings of the 7th International Conference on ICT for Sustainability*, pp. 247–254, Bristol, United Kingdom, 2020.

[56] Nvidia, "Geforce NOW," 2020, February 2021, https://www.nvidia.com/en-us/geforce-now/.

[57] Wikipedia, "Geforce NOW," 2015, February 2021, https://en.wikipedia.org/wiki/GeForce_Now.

[58] Microsoft, "Xbox cloud gaming," 2019, February 2021, https://www.xbox.com/en-US/xboxgame-pass/cloud-gaming.

[59] Amazon, "Luna," 2020, February 2021, https://www.amazon.com/luna.

[60] Nvidia, "CloudXR," 2020, March 2021, https://developer.nvidia.com/nvidia-cloudxr-sdk.

[61] Y. Li and W. Gao, "MUVR: supporting multi-user mobile virtual reality with resource constrained edge cloud," in *Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 1–16, Seattle, WA, USA, 2018.

[62] M. F. Romero-Rondón, L. Sassatelli, F. Precioso, and R. Aparicio-Pardo, "Foveated streaming of virtual reality videos," in *Proceedings of the 9th ACM Multimedia Systems Conference*, pp. 494–497, Amsterdam, Netherlands, 2018.

[63] S. Shi, V. Gupta, M. Hwang, and R. Jana, "Mobile VR on edge cloud: a latency-driven design," in *Proceedings of the 10th ACM Multimedia Systems Conference*, pp. 222–231, Amherst, Massachusetts, 2019.

[64] M. Viitanen, J. Vanne, T. D. Hämäläinen, and A. Kulmala, "Low latency edge rendering scheme for interactive 360 degree virtual reality gaming," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1557–1560, Vienna, Austria, 2018.

[65] T. Kämäräinen, M. Siekkinen, J. Eerikäinen, and A. Ylä-Jääski, "CloudVR: cloud accelerated interactive mobile virtual reality," in *Proceedings of the 26th ACM International Conference on Multimedia (MM '18)*, pp. 1181–1189, Seoul, Republic of Korea, 2018.

[66] M. Pitkänen, M. Viitanen, A. Mercat, and J. Vanne, "Remote VR gaming on mobile devices," in *Proceedings of the 27th ACM International Conference on Multimedia*, pp. 2191–2193, Nice, France, 2019.

[67] E. Cuervo, A. Wolman, L. P. Cox et al., "Kahawai: high-quality mobile gaming using GPU offload," in *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '15)*, pp. 121–135, Florence, Italy, 2015.

[68] Nvidia, "GTC silicon valley-2019 ID:S9914,Cloud XR: challenges and strategies in streaming XR over 5G," 2019, March 2021, https://developer.nvidia.com/gtc/2019/video/s9914/video.

[69] J. Carmack, "Day 2 Keynote | Oculus Connect 6," 2019, May 2020, https://www.youtube.com/watch?v=PMIDaomx0GA.

[70] R. Zhong, M. Wang, Z. Chen et al., "On building a programmable wireless high-quality virtual reality system using commodity hardware," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, p. 7, Mumbai, India, 2017.

[71] Nvidia, "Nvidia video codec SDK," 2013, January 2020, https://developer.nvidia.com/nvidia-video-codec-sdk.

[72] Riftcat, "Dev update #57-streaming enhanced release," 2020, August 2020, https://blog.riftcat.com/2020/04/dev-update-57-streaming-enhanced-release.html.

[73] J. Nieh, S. J. Yang, and N. Novik, "Measuring thin-client performance using slow-motion benchmarking," *ACM Transactions on Computer Systems*, vol. 21, no. 1, pp. 87–115, 2003.

[74] V. Kelkkanen, M. Fiedler, and D. Lindero, "Bitrate requirements of non-panoramic VR remote rendering," in *Proceedings of the 28th ACM International Conference on Multimedia (MM '20)*, pp. 3624–3631, Seattle, WA, USA, 2020.

[75] Valve, "OpenVR SDK," 2015, January 2020, https://github.com/ValveSoftware/openvr.

[76] A. Vlachos, *Advanced VR Rendering*, Valve, 2015, February 2020, http://alex.vlachos.com/graphics/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf.

[77] Nvidia, "Video encode and decode GPU support matrix," 2016, January 2020, https://developer.nvidia.com/video-encode-decode-gpu-support-matrix.

[78] V. Software, "GameNetworkingSockets," 2018, August 2020, https://github.com/ValveSoftware/GameNetworkingSockets.

[79] IETF, "Datagram congestion control protocol (DCCP) 11.4. ack vector options," IETF, 2006, August 2020, https://tools.ietf.org/html/rfc4340#section-11.4.

[80] Microsoft, "IPPROTO_TCP socket options," 2018, August 2020, https://docs.microsoft.com/en-us/windows/win32/winsock/ipproto-tcp-socket-options.

[81] Engadget, "HTC Vive ditches the PC thanks to China's cloud VR service," 2017, May 2020, https://www.engadget.com/2017-09-19-htc-vive-china-cloud-vr-service.html.

[82] K. Andersson, M. Nilsson, and A. Gylling, *5G Innovation Hub North*, Ericsson, Telia and Tieto, 2020, September 2020, https://www.5ginnovationhubnorth.se/.

[83] T.-K. Le, U. Salim, and F. Kaltenberger, "An overview of physical layer design for ultra-reliable low-latency communications in 3GPP release 15 and release 16," 2020, http://arxiv.org/abs/2002.03713 [eess.SP].

[84] V. Kelkkanen and M. Fiedler, "A test-bed for studies of temporal data delivery issues in a TPCAST wireless virtual reality setup," in *Proceedings of the 28th International Telecommunication Networks and Applications Conference (ITNAC)*, Sydney, Australia, 2018.

[85] A. Basu, A. Sullivan, and K. Wiebe, "Variable resolution teleconferencing," in *Proceedings of IEEE Systems Man and Cybernetics Conference-SMC*, vol. 4, pp. 170–175, Le Touquet, France, 1993.

[86] S. Lee and A. C. Bovik, "Fast algorithms for foveated video processing," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 2, pp. 149–162, 2003.

[87] C. J. Hansen, "WiGiG: multi-gigabit wireless communications in the 60 GHz band," *IEEE Wireless Communications*, vol. 18, no. 6, pp. 6-7, 2011.

[88] Y. Ghasempour, R. C. M. Claudio, C. C. da Silva, and E. W. Knightly, "IEEE 802.11ay: next-generation 60 GHz communication for 100 Gb/s Wi-Fi," *IEEE Communications Magazine*, vol. 55, no. 12, pp. 186–192, 2017.

[89] IEEE, *Status of project IEEE 802.11ay*, IEEE, 2020, June 2020, http://www.ieee802.org/11/Reports/tgay_update.htm.

[90] F. Alriksson, L. Boström, S. Joachim, Y.-P. Eric Wang, and A. Zaidi, *Critical IoT Connectivity*, Ericsson, 2020, September 2020, https://www.ericsson.com/49ba0b/assets/local/reports-papers/ericsson-technology-review/docs/2020/critical-iot-connectivity.pdf.

[91] X. Li, Y. Wu, W. Zhang, R. Wang, and F. Hou, "Deep learning methods in realtime image super-resolution: a survey," *Journal of Real-Time Image Processing*, vol. 17, no. 6, pp. 1885–1909, 2020.