On the Performance Comparisons of Native and Clientless Real-Time Screen Sharing Technologies

CHUN-YING HUANG*, YUN-CHEN CHENG, and GUAN-ZHANG HUANG, National Chiao Tung University, Taiwan
CHING-LING FAN and CHENG-HSIN HSU*, National Tsing Hua University, Taiwan

Real-time screen sharing provides users with ubiquitous access to remote applications, such as computer games, movie players, and desktop applications (apps), anywhere and anytime. In this paper, we study the performance of different screen sharing technologies, which can be classified into native and clientless ones. The native ones dictate that users install special-purpose software, while the clientless ones directly run in web browsers. In particular, we conduct extensive experiments in three steps. First, we identify a suite of the most representative native and clientless screen sharing technologies. Second, we propose a systematic measurement methodology for comparing screen sharing technologies under diverse and dynamic network conditions using different performance metrics. Last, we conduct extensive experiments and perform in-depth analysis to quantify the performance gap between clientless and native screen sharing technologies. We found that our WebRTC-based implementation achieves the best overall performance. More precisely, it consumes a maximum of 3 Mbps bandwidth while reaching a high decoding ratio and delivering good video quality. Moreover, it leads to a steadily high decoding ratio and video quality under dynamic network conditions. By presenting the very first rigorous comparisons of the native and clientless screen sharing technologies, this article will stimulate more exciting studies on the emerging clientless screen sharing technologies.

CCS Concepts: • **Information systems** → **Multimedia streaming**; **Web applications**;

Additional Key Words and Phrases: Live video streaming, real-time encoding, measurements, performance evaluations, performance optimization

ACM Reference Format:
Chun-Ying Huang, Yun-Chen Cheng, Guan-Zhang Huang, Ching-Ling Fan, and Cheng-Hsin Hsu. 2020. On the Performance Comparisons of Native and Clientless Real-Time Screen Sharing Technologies. *ACM Trans. Multimedia Comput. Commun. Appl.* 1, 1, Article 1 (January 2020), 26 pages. https://doi.org/10.1145/3437881

ACKNOWLEDGMENTS

This work was partially supported by the Ministry of Science and Technology of Taiwan (#107-2221-E-009-028-MY3 and #107-2221-E-007-091-MY3) and by a NOVATEK Fellowship.

*Chun-Ying Huang and Cheng-Hsin Hsu are co-corresponding authors.

Authors' addresses: Chun-Ying Huang, chuang@cs.nctu.edu.tw; Yun-Chen Cheng, emmas191@gmail.com; Guan-Zhang Huang, National Chiao Tung University, Computer Science, 1001 University Road, Hsinchu, 30010, Taiwan, johnhuang30834@gmail.com; Ching-Ling Fan, ch.ling.fan@gmail.com; Cheng-Hsin Hsu, National Tsing Hua University, No. 101 Sec. 2 Kuang-Fu Road, Hsin-Chu, 300, Taiwan, chsu@cs.nthu.edu.tw.

1551-6857/2020/1-ART1 $15.00
https://doi.org/10.1145/3437881

1 INTRODUCTION

Market research reports show the Internet's increasing penetration rate, e.g., there were 4.3 billion worldwide Internet users in March 2019 [27]. Besides, consumer-graded computers come in various forms, including workstations, desktops, laptops, tablets, smartphones, and smartwatches, which are connected to the Internet through heterogeneous networks. These computers have diverse resources in different aspects, such as computing power, storage space, network speed, battery capacity, and display dimensions. Oftentimes, users may want to access the resources of a remote *server* from a local *client* through the Internet, as illustrated in Fig. 1. Particularly, users can run their applications on relatively powerful servers, capture the screen and sound, and transfer the compressed video and audio streams to relatively thin clients. Upon receiving the streams, the clients decode and render the applications to the users. In addition, the users interact with the applications through different input devices, such as keyboards, mouses, and inertial sensors. The inputs are then streamed back to the servers and replayed to the applications. Through remotely sharing applications from the servers to the clients, users gain access to rich resources via the Internet. This is referred to as real-time *screen sharing*, which is also known as remote rendering [36], remote desktops [41], and screencast [1, 16].

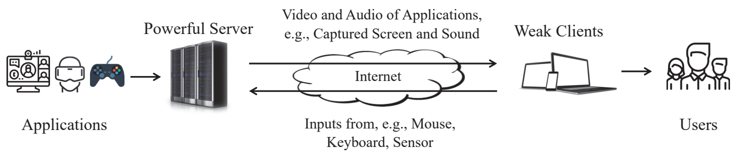

Powerful Server
Video and Audio of Applications, e.g., Captured Screen and Sound
Weak Clients
Internet
Applications
Inputs from, e.g., Mouse, Keyboard, Sensor
Users

Fig. 1. An overview of screen sharing technologies.

Screen sharing has been applied to many applications, including teleconferences, distance education, live audio/video streaming, and cloud gaming. The existing screen sharing technologies run the applications on desktop computers or cloud servers. For instance, on desktop computers, remote assistance [25] is offered by Windows OS, and Virtual Network Computing (VNC) [32] is available across many OSes. Another popular application is cloud gaming, e.g., PS4 Remote Play [37] and Steam's In-Home Streaming [39] allow users to play PS4 and PC games, respectively. More precisely, Google announced their cloud gaming platform called Stadia [14], while GeForce Now from NVidia allows OSX, Windows, and SHIELD users to play remote games [31]. In addition to the tremendous interest from the industry, there are also cloud gaming platforms, which have been developed in academia [5, 18, 42].

The quality of screen sharing may be quantified in several performance metrics, as users of different applications have diverse quality requirements. For example, users who use video players through screen sharing demand good video quality, but pay less attention to latency. In contrast, cloud gaming users have higher and more diverse (depending on the game genres) requirements regarding latency [6, 8], because users of fast-paced games (such as First Person Shooter, or *FPS* games) focus more on the quality of hand-eye coordination [7]. Therefore, depending on the target applications, we have to carefully choose the most suitable screen sharing technologies for a sweet spot in the tradeoff of user experience and resource consumption [36].

In this article, we carry out the very first measurement study on the performance comparisons of native and clientless screen sharing technologies. We achieve this in three steps. First, we survey representative screen sharing technologies, which can be roughly classified into two groups: (i) *native* and (ii) *clientless*. The native technologies require users to install dedicated software on the clients; representative examples include Remote Desktop Protocol (RDP) [24], VNC [32], and

GamingAnywhere (GA) [18]. The clientless technologies utilize web browsers in order to avoid software installation and achieve better portability; representative examples include FFmpeg [10], noVNC [24], and WebRTC [40]. Second, we design and build a measurement testbed to conduct real experiments for fair performance comparisons. The testbed consists of carefully designed tools for measuring key performance metrics, such as bandwidth consumption, latency, decoding ratio, and video quality [20, 30, 33]. Furthermore, our testbed allows users to adjust several key network parameters, including network bandwidth, delay, and packet loss rate. Third, we systematically conduct extensive experiments using our testbed to compare the performance of different screen sharing technologies. To the best of our knowledge: (i) the detailed performance measurements of clientless screen sharing technologies and (ii) the comparisons of clientless and native technologies have not been done in the literature.

Our measurement results reveal that:

- VNC, noVNC, and RDP are vulnerable to imperfect network conditions and result in longer latency, degraded video quality, and lower decoding ratios.
- GA suffers from lower video quality and decoding ratios when the packet loss rate is nontrivial.
- WebRTC is the most robust technology under bad network conditions, which can be attributed to its built-in error resilience mechanisms. Under the most challenging network conditions, WebRTC outperforms all other considered technologies: it achieves a lower bandwidth consumption (< 3 Mbps), higher video quality (> 27 dB in Peak Signal-to-Noise Ratio, or PSNR), and a higher decoding ratio (> 86%). Similar merits are also observed in the dynamic network conditions.

In summary, we found that WebRTC works as well as, if not better than, other screen sharing technologies, including the native technologies, most of the time. *Hence we recommend the clientless WebRTC [40] for the majority of applications and usage scenarios.* The only exceptions are when: (i) the network bandwidth is abundant, or (ii) extremely low latency is required. For the former case, existing WebRTC implementations in Chrome and Firefox are not aggressive enough in consuming more network bandwidth for even higher video quality, which may be addressed by augmenting their rate control algorithms. For the latter case, the additional latency due to web browsers was still too high at the time of writing. Thus, *we recommend GA [18], which is a native screen sharing technology tailored for cloud gaming and other highly interactive applications,when extremely low latency is a must.*

The rest of this article is organized as follows. In Section 2, we survey the literature. We discuss the architecture of screen sharing technologies in Section 3. Section 4 details the representative screen sharing technologies, which span over both native and clientless technologies. Section 5 presents our proposed measurement methodology, and Section 6 analyzes the measurement results. We conclude the article in Section 7. Appendix A provides some detailed analysis results that cannot be included in Section 6 due to space limitations.

2 RELATED WORK

Clientless Screen Sharing Technologies. Web browsers have been used as screen sharing clients in the literature. For example, Ganji et al. [11] investigate the possibility of using HTML5 virtualization to support screen sharing on resource-constrained mobile devices. They build a testbed that converts RDP content into HTML5 canvas, which is then streamed to mobile devices. The testbed runs several applications, including MS Word, Internet Explorer, and DICOM Viewer. Besides, multiple performance metrics, such as bandwidth consumption, CPU usage, and video quality, are considered. Mufali et al. [29] study how to allow disabled users to remotely access their specialized software tools installed and configured on cloud servers using public personal computers. This is an important problem, because installing and configuring such specialized software tools is time-consuming, tedious, and error-prone. They propose to employ the opensource noVNC [23] to allow disabled people to access their cloud servers via HTML5 in web browsers. Ueberheide et al. [38] present a real-time streaming framework for free-viewpoint rendering, which relies on open standards and software. To enable interactions via web browsers, user inputs are captured with JavaScript and sent via WebSockets. The rendered video is streamed with HTML5 video tag over HTML/TCP using FFmpeg [10]. Mochida et al. [28] propose a web-based remote collaboration system, which consists of three entities: a relay server, an overlay manager, and a timecode server. The real-time interactions are sent through the relay server using UDP packets for short latency. The overlay manager links the web browser with applications. The timecode server embeds timecodes in video packets, which allow web browsers to synchronize remote widget movements. Their system enables users to share texts, images, and screens. Users can remotely move mouse cursors, draw annotations on images, and control shared screens. *Similar to the abovementioned papers [11, 28, 29, 38], we also design, implement, and evaluate several clientless screen sharing technologies based on opensource projects, such as WebRTC [40] and FFmpeg [10]. Differing from the above works, we quantitatively measure and compare the performance of multiple clientless screen sharing technologies under the same controlled network conditions.*

Performance Measurement of Screen Sharing Technologies. The performance of some screen sharing technologies has been measured in the literature. For instance, Ammar et al. [2] employ Google Chrome's built-in tool to measure the performance of a video communication application built on WebRTC. They found that the built-in tool helps diagnose the inferior video quality (mainly due to video freezes). Although their work demonstrates the potential of the built-in tool, this tool is not applicable to native screen sharing technologies. Similarly, Garcia et al. [12] adopt the opensource Kurento [22], which is a high-level testing framework which measures the performance of WebRTC applications. Some measurement techniques of Kurento can be improved, e.g., Kurento embeds a new timecode every five seconds, while our testbed does that for every single frame to obtain finer-grained latency measurements. *These two studies [2, 12] only consider a single screen sharing technology: WebRTC.* The performance comparisons among different native screen sharing technologies have also been considered in the literature. For example, Lin et al. [21] evaluate bandwidth consumption and power consumption of the native RDP [24], VNC [32], and GA [18] over Wi-Fi and LTE networks. Their observations show that RDP and VNC consume fewer resources in more static scenes; GA trades higher bandwidth consumption for shorter latency. This is expected, as GA is designed for cloud gaming. *Compared with their work, our current article considers more performance metrics and more screen sharing technologies.* Last, our earlier work [17] compares the AirPlay, Chromecast, GA, Miracast, MirrorOp and Splashtop on different OSes under diverse network conditions. While a rich set of performance metrics is considered, the screen sharing technologies therein are native and mostly proprietary (except GA). *In contrast, the current article compares the performance of representative native and clientless screen sharing technologies, which has never been done before.*

3 ARCHITECTURE

Fig. 2 shows the common architecture of screen sharing technologies. Each server and client contains three components: a *capturer/renderer*, a *compressor/decompressor*, and a *sender/receiver*. The server first captures the screen and sound, the video/audio is compressed into bitstreams, which are then packetized and sent to the client. The client receives and reassembles the video/audio packets into compressed bitstreams, which are then decompressed into raw video/audio for rendering to the users. The key factors affecting the performance of screen sharing technologies are the designs of the compression (compressor/decompressor) and transmission (sender/receiver) components. Each

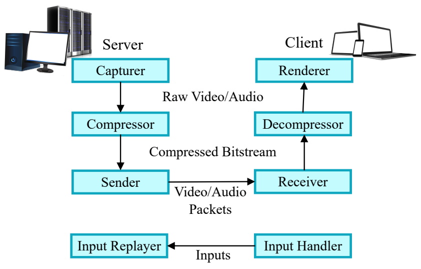


Fig. 2. Common architecture of screen sharing technologies.

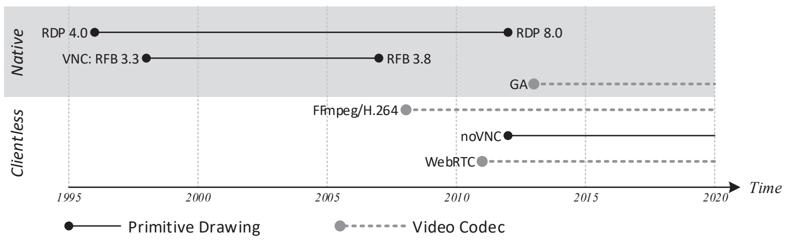


Fig. 3. The timelines and classifications of the representative screen sharing technologies.

screen sharing technology employs its own compression codec and network protocol to meet the design objectives of its target applications. Certain tradeoffs must be considered in the design phase, e.g., different video codecs achieve diverse tradeoffs between video quality and encoding speed. VNC client, for example, supports multiple codecs for users to opt for higher video quality or faster encoding speed. WebRTC clients and servers exchange a list of supported codecs using Session Description Protocol (SDP) at the initialization time. Its network protocols add temporal meta-data, such as timestamps and sequence numbers, to the packets to handle out-of-order delivery. Screen sharing technologies may measure live network conditions, such as network bandwidth, round-trip time, and packet loss rate, to better adapt to network dynamics. That is, under bad network conditions, the receiver notifies the sender to send lower-quality video, in order to avoid late and lost packets for smooth playout. For example, a VNC client periodically sends update requests for the next few frames. Therefore, the client can adjust the frequency of such requests based on the current network conditions. WebRTC also measures the network conditions and dynamically turns on/off Forward Error Correction (FEC) for better error resilience. Last, some screen sharing technologies send user inputs in reverse channels. The keystrokes, mouse movements, mouse clicks, inertial and other sensor readings are captured by *input handler* and sent to the *input replayer* at the server. The server replays these inputs to applications. The inputs typically have fairly low bitrates but need to be reliably delivered. Hence, inputs are mostly sent over TCP protocol, which may be different from video/audio packets.

4 SCREEN SHARING TECHNOLOGIES: OVERVIEW AND IMPLEMENTATIONS

In this section, we present an overview of the representative screen sharing technologies. We also introduce how they are implemented and configured on our testbed. Fig. 3 summarizes the timelines of these screen sharing technologies. It also classifies the technologies into two dimensions: (i) native versus clientless and (ii) primitive drawing versus video codec.

4.1 Native Screen Sharing Technologies

We first present three representative native technologies.

Remote Desktop Protocol (RDP) is a proprietary screen sharing protocol developed by Microsoft, which is used by several Windows tools, such as Remote Desktop Operation and Microsoft Windows Remote Assistance (MSRA). The captured screens are compressed using both intra- and inter-frame video coding before being transmitted. Our experiments employ MSRA [25] as the RDP implementation. Both the server and client run on Windows 10. We set up screen sharing sessions as follows. We first share a directory between the server and client. The server then launches MSRA, and saves an invitation file in the shared directory. After that, the server screen shows a password for the client to enter. The user at the client clicks the saved invitation file in the shared directory and enters the password to create a screen sharing session.

Virtual Network Computing (VNC) is a platform-independent screen sharing technology developed by Olivetti & Oracle Research Lab (ORL). VNC uses the Remote Frame-Buffer (RFB) protocol [32] to remotely access and control the screens of servers. Because the RFB protocol works at the framebuffer level, it is applicable to all Graphical User Interfaces (GUIs), including X11, Windows, and OSX. An RFB server runs a daemon process that maintains the framebuffer states. The RFB protocol defines a very primitive operation of drawing rectangles, which gives the VNC server flexibility to adaptively determine the granularity of the streamed screens under diverse network conditions and server/client computing power. VNC users may terminate VNC sessions at any time, and reconnect to the servers later so as to resume the sessions. Moreover, multiple VNC clients are allowed to connect to a VNC server at the same time. There are multiple VNC implementations, and we use tightVNC [13] in our experiments. tightVNC is free and provides several codec options, including a tight codec that is tailored for low network bandwidth. In the experiments, both the server and client run on Windows 10. First, the sever launches the tightVNC server and sets up the configuration such as the port number and the password. We choose tight codec with a compression quality and efficiency level of 6, and a screen polling cycle of 30 ms. The client launches tightVNC client with the IP address, port number, and password of the server. The screen sharing session is then created.

GamingAnywhere (GA) is a cloud gaming platform with high extensibility, portability, reconfigurability, and openness. It supports several video/audio codecs from the *libavcodec* library [18]. GA is designed to be modularized because the screen capturing and rendering APIs are platform-dependent, and the codecs and network protocols are platform-independent. GA supports Windows and Linux, and can be ported to other OSes like OSX and Android. GA users can change the configurable parameters, such as codecs and protocols, via configuration files. Each GA session consists of two network flows: data and control. The video/audio packets are sent via data flows using RTSP or RTP. The user inputs are sent via control flows. In our experiments, we use x264 as the codec, RTP as the data flow protocol, and TCP as the control flow protocol. The server runs the periodic mode, where the entire screens are streamed to the client. We run GA on Windows 10 using MinGW [26].

4.2 Clientless Screen Sharing Technologies

We next present three representative clientless technologies.

noVNC is an HTML5-based VNC client library and application [23], which allows a client to connect to a VNC server. The noVNC client is implemented with HTML5 WebSockets, canvas, and JavaScript, while the noVNC server is almost the same as a regular VNC server. However, regular VNC servers do not support WebSockets, and thus a noVNC server has to proxy the TCP sockets and WebSockets. Each client creates an RFB object at the server, and the RFB object is assigned to

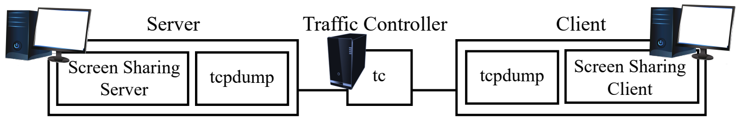


Fig. 4. The overview of our measurement testbed.

an HTML5 element, which is attached to a new canvas. Next, the screens are rendered at the client via commands of drawing pixels, while the actual drawing commands are determined by the chosen codec and network conditions. Because VNC is a pull-based protocol, a client may adjust the update request frequency to better match the current network condition. We run the server and client on Windows 10. For the server, we use the tightVNC server described earlier. The noVNC server also runs on the same machine serving as a proxy between TCP sockets and WebSockets. We employ Chrome to run our client, which supports HTML5 WebSockets and canvas in our experiments.

WebRTC is an open framework developed by Google [40], which aims for real-time communications among web browsers without plug-ins. It is supported on multiple web browsers: including Chrome, Opera, and Firefox. WebRTC supports high-quality video/audio communications over peer-to-peer networks. WebRTC APIs are implemented in JavaScript, which enables screen capturing, compression, and streaming for web browser applications. The APIs use SDP for session negotiation, but leave the implementation details to application developers for their usage scenarios. Moreover, WebRTC also integrates protocols, such as ICE, STU, and TURN to address the NAT and firewall traversal problems. In our experiments, we build a WebRTC application, where both the server and client run in web browsers. We test the WebRTC application with two browsers: Chrome (version 76) and Firefox (version 70), and denote them as WebRTC-C and WebRTC-F in tables and figures. Both Chrome and Firefox are 64-bit binaries running on Windows 10. We use VP8 as the video codec, and RTP/UDP as the network protocol. We note that the WebRTC implementations in Chrome and Firefox are different. First, Chrome retransmits lost packets using a channel specified by a different RTP Synchronization Source Identifier (SSRC), whereas Firefox retransmits lost packets using the existing channel. Second, Chrome supports FEC for better error resilience, while Firefox does not. More discussions on these subtle differences are presented in our experiment analysis.

FFmpeg can be leveraged to build a clientless screen sharing technology as follows. First, we set up a PHP server. The client opens a browser which supports HTML5 and connects to the server URL. Once the connection is up, FFmpeg APIs are used to capture server screens as a video stream and then stream it to the HTML5 video tag. In our experiments, FFmpeg APIs capture the screens into YUV420p videos, which are compressed with libx264 codecs at 30 fps. The resulting bitstreams are embedded into mp4 container files.

5 MEASUREMENT METHODOLOGY

In this section, we detail our experiment designs encompassing the testbed, environment setup, and interpretation procedure.

5.1 Testbed

We set up a testbed for our measurement study, as illustrated in Fig. 4. The testbed consists of three workstations: the *server* and *client* for running the screen sharing technologies, and the *traffic controller* for emulating the diverse and dynamic network conditions. The server and client are put on two different LANs connected by the traffic controller, which runs the *tc* utility to dynamically

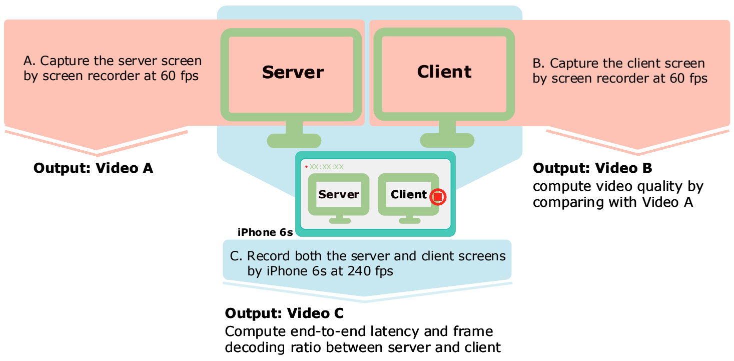

A. Capture the server screen by screen recorder at 60 fps
Server
Client
B. Capture the client screen by screen recorder at 60 fps
Output: Video A
Output: Video B
compute video quality by comparing with Video A
iPhone 6s
C. Record both the server and client screens by iPhone 6s at 240 fps
Output: Video C
Compute end-to-end latency and frame decoding ratio between server and client

Fig. 5. Video-related performance metrics require us to capture several videos using both HDMI video capturing boxes (videos A and B), and a high-speed camera (video C).

Table 1. Representative Network Conditions

Network Condition	Network Bandwidth	Delay	Packet Loss Rate
Ideal	Unlimited	0 ms	0%
Lossy	Unlimited	0 ms	2%
High Delay	Unlimited	200 ms	0%
Low Bandwidth	4 Mbps	0 ms	0%
Challenging	4 Mbps	200 ms	2%

Table 2. Considered Video Content

Application	Video	Content
Game	G1	First-Person Shooter
	G2	Car Racing
	G3	Real-Time Strategy
Movie	M1	Movie (Slow)
	M2	Movie (Fast)
	M3	Talk Show
Desktop Apps	A1	Google Map Street View
	A2	PowerPoint and Spread-sheet Editing
	A3	Web Browsing (Wikipedia and Amazon)

configure the Linux kernel packet scheduler, so as to throttle the network bandwidth, increase the round-trip delay, and inject packet losses. We run *tcpdump* on the server and client to capture packets. By comparing the captured packets at the server and client, we compute the bandwidth consumption among other network related metrics. For video-related metrics, we capture the videos from the testbed as illustrated in Fig. 5. More specifically, we use HDMI video capturing boxes to capture the server and client screens as a YUV420p video with a resolution of 1280x720 at 60 fps. The videos are then compared to calculate the video quality. We also use an iPhone 6s to shoot a video containing both the server and client screens at 240 fps. Through embedding frame numbers in the shared screens, we analyze the captured video to match the client frames to individual server frames. The analysis results lead to latency, frame decoding ratio, and other video-related metrics.

5.2 Setup and Procedure

We configure *tc* to emulate various network conditions. In particular, we chose the network parameters as below following the literature [15, 17][1].

- **Network bandwidth**: In addition to unlimited bandwidth, we also throttle the network bandwidth at 6 Mbps and 4 Mbps.
- **Delay**: We add a delay of 0 ms, 100 ms, and 200 ms from the server to client.
- **Packet loss rate**: We inject packet loss rate of 0%, 1%, 2%, and 5%.

The total number of emulated network conditions is therefore: $3 \times 3 \times 4 = 36$. Although we conduct a measurement study on all 36 conditions, we zoom into five more representative conditions when

[1]Our pilot tests indicate that expanding the ranges of our selected network parameters does not lead to significantly different observations.

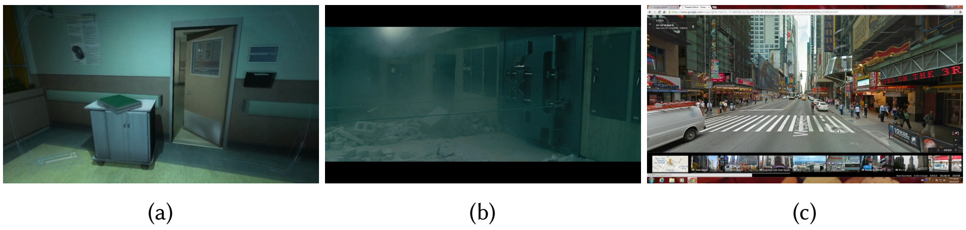


Fig. 6. Sample frames from the three considered applications: (a) game, (b) movie, and (c) desktop apps.

comparing the performance of different screen sharing technologies, as summarized in Table 1. The chosen conditions include: (1) ideal network, which has no limitation on the network, (2) lossy network, which injects some packet loss, (3) high delay network, which sets the delay at 200 ms, (4) low bandwidth network, which considers the network bandwidth at 4 Mbps, and (5) challenging network, which sets the bandwidth at 4 Mbps, the delay at 200 ms, and the packet loss rate at 2%.

We consider three representative applications: *game*, *movie*, and *desktop apps*, which are remotely accessed through the considered screen sharing technologies. For fair comparisons, we record three sample YUV420p videos for each application in 1280x720 at 30 fps at the server. Figure 6 shows sample frames from the considered applications. Table 2 summarizes the considered nine videos. Each video lasts for one minute. We concatenate all nine videos into a single test video to simplify the experiment procedure, and insert 3 seconds of white frames between any two videos. The resulting test video lasts for 566 seconds (or 16,991 video frames).

The test video is played once the client connects to the server. For every screen sharing technology, we play the test video under 36 different network conditions. The experiment procedure is summarized in the following:

(1) Launch *tcpdump* to record the network packets.
(2) Establish a session of a screen sharing technology.
(3) Start recording the screens into videos.
(4) Play the test video at the server.
(5) Stop collecting network packets and recording screens after the test video is finished.
(6) Change the network conditions and the screen sharing technology, and go back to (1).

We consider the following performance metrics.

- **Bandwidth consumption**: After each experiment, we analyze the pcap file saved by *tcpdump* and compute the bandwidth consumption.
- **Latency**: We embed the frame numbers in individual video frames (more details below)[2]. Since the frame rate is fixed, the latency between a frame sent by the server and rendered by the client is computed by analyzing the recorded screens of both the server and client.
- **Decoding ratio**: Some frames might be dropped due to network congestion or system failures. The decoding ratio is defined as the percentage of successfully decoded frames at the client. Frames that are significantly corrupted so that their frame numbers cannot be identified at the client are not counted toward the decoding ratio.
- **Video quality**: We consider two video quality metrics: PSNR and SSIM (Structural Similarity Index) [4]. PSNR is the ratio between the maximum power of a signal and the power of the

[2]Another possible way to measure the latency is to insert the timestamps or frame numbers into the video packets at the server and collect the receiving timestamps or frame numbers at the client. However, doing so might suffer from less accurate latency measurements because the clocks at the server and client may not be perfectly synchronized (say, at millisecond level).

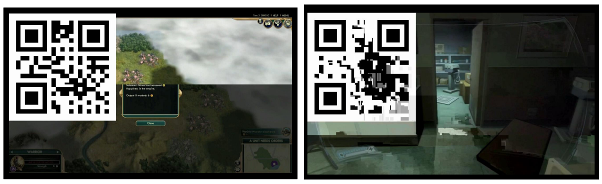

(a) (b)

Fig. 7. Sample problematic QR codes because of: (a) half-refreshed frame from VNC and (b) distorted frame from GA.

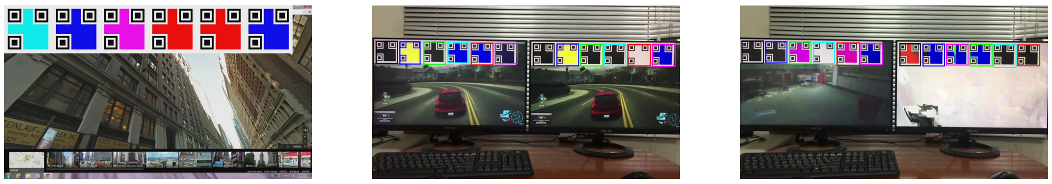

(a) (b) (c)

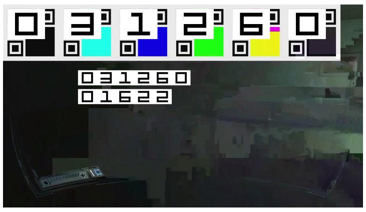
Fig. 8. Sample colorcodes proposed by us: (a) a frame with its embedded frame number, (b) localized colorcodes, and (c) localized colorcodes in a distorted frame.

corrupting noise, which is defined as a function of Mean Squared Error (MSE) in the decibel (dB) scale. In particular, the luminance value of the pixel pair from the two corresponding frames from the client and the server are compared to calculate the PSNR value. The SSIM is designed as a more comprehensive objective quality metric than PSNR. SSIM approximates the Human Vision System (HVS) and considers not only pixel values but also the contrast and structure. SSIM value varies between 0 and 1. Higher PSNR and SSIM values generally mean lower distortion and higher video quality.

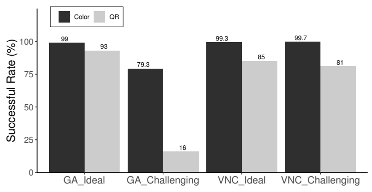
5.3 Embedded Frame Numbers

We embed frame numbers into the video before streaming it, in order to match the frames captured at the client to the frames captured at the server. We first consider the existing solutions: data matrix [19] and QR (Quick Response) code [9]. Decoding frame numbers from frames is generally done in two steps: (i) localizing the embedded patterns and (ii) converting the patterns into the frame numbers. We conduct some pilot tests to study the efficiency and robustness of the data matrix and QR code. We make two key observations. First, locating and decoding data matrices is computationally intensive: it takes *several seconds* to locate and decode each code in a high-resolution image. Hence, we do not consider the data matrix in the rest of this article. Second, we notice that QR codes have a complex structure and may be sensitive to distortion caused by network congestion. Fig. 7 shows two sample frames with problematic QR codes. Fig. 7a shows a half-refreshed frame. Since the top and bottom halves of the QR code come from two different frames, the decoded frame number would be *wrong*. Fig. 7b shows a distorted frame due to a couple of lost packets. In this case, the QR code is undecodable. Fig. 7 demonstrates the limitations of using QR codes to embed frame numbers, and thus we propose our own *colorcode* in the following.

Our proposed colorcode borrows the localization approach of the QR code, but introduces high redundancy by only encoding very few information bits. More specifically, each pattern is filled with a single 8-bit color, where each RGB component encodes two possible values: minimum (0) or

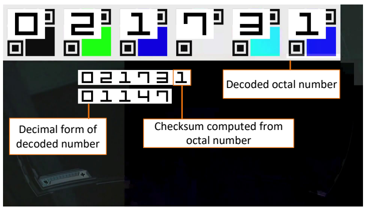


(a)

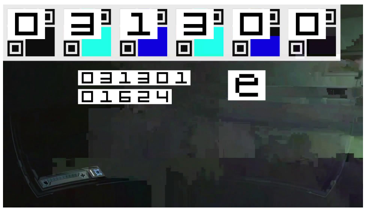


(b)

Fig. 9. Performance comparisons: (a) colorcodes in a distorted frame can still be decoded and (b) our proposed colorcodes are more robust than the QR code.


021731
01147
Decoded octal number
Decimal form of decoded number
Checksum computed from octal number

(a)



(b)

Fig. 10. Sample outputs of our colorcode decoder: (a) successfully obtained frame number and (b) checksum error found in a frame.

maximum (255). Therefore, each colorcode can be one of the $2^3 = 8$ possible values. Considering the length of our test video, we decide to employ five colorcodes to represent the frame numbers. To further increase the robustness, we add a sixth colorcode to encode the checksum. Fig. 8a shows a sample video frame with the octal frame number represented by six colorcodes. We note that about 26% of the frame is occupied by colorcodes.

To decode the colorcodes in captured screens, we first employ *quirc* [3] to localize the codes. Fig. 8b and Fig. 8c show sample localized colorcodes without and with distortion caused by packet loss. We define a Region-of-Interest (RoI) around the center of each colorcode, and convert the most occurring color in the RoI into an octal digit. We also validate the checksum for better robustness. Next, we compare the *successful decoding rate* of our colorcode against the QR code. For fair comparisons, we increase the QR code size, so that it also occupies about 26% of each frame. We consider two network conditions: (i) *ideal*, where *tc* is disabled and (ii) *challenging network*, where *tc* is configured for a network bandwidth of 4 Mbps, a delay of 200 ms, and a packet loss rate of 2%. Fig. 9a shows that our colorcode still works with GA under some distortion due to packet loss. Fig. 9b reports the overall results, which reveals that our colorcode achieves higher successful decoding rates than the QR code. For example, for GA in the challenging network, switching from the QR code to our colorcode increases the success rate by 22.4%. Last, we present two sample outputs of the colorcode decoder in Fig. 10, including a successfully decoded frame and a frame with a checksum error.

6 COMPARATIVE ANALYSIS

In this section, we analyze the measurement results. Table 3 summarizes the considered screen sharing technologies. We report the average results with error bars indicating the standard deviation

Table 3. Considered Screen Sharing Technologies

Technology	**RDP**	**VNC**	**GA**	**FFmpeg**	**noVNC**	**WebRTC-C&F**
Type	Native	Native	Native	Clientless	Clientless	Clientless
Components						
Capturer	OS-Specific	OS-Specific	OS-Specific	FFmpeg APIs	OS-Specific	WebRTC APIs
Codec	RDP	tight	H.264	H.264	tight	VP8
Transmission	RDP/TCP	VNC/TCP	RTP/UDP	HTTP/TCP	VNC/TCP	RTP/UDP
Renderer	RDP	VNC	GA	HTML5 Video Tag	HTML5 Canvas	HTML5 Video Tag

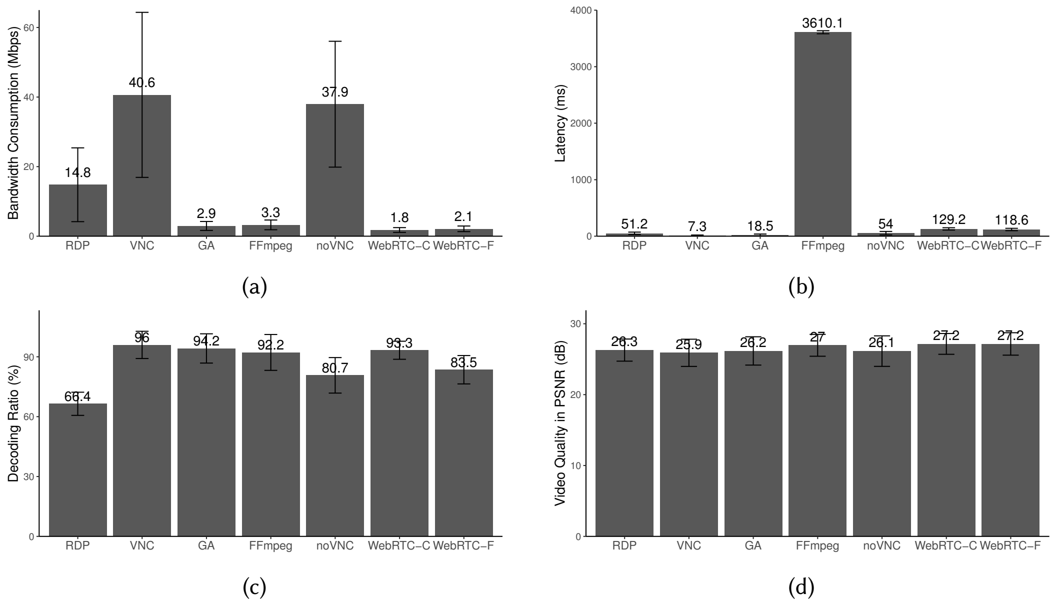
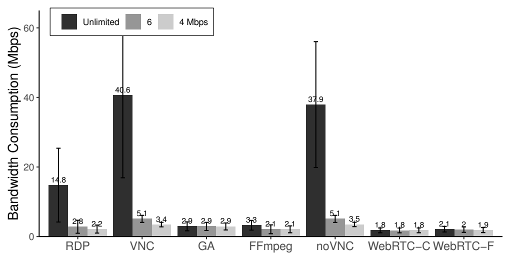


Fig. 11. Performance comparisons under the ideal network condition: (a) bandwidth consumption, (b) latency, (c) decoding rate, and (d) video quality in PSNR.

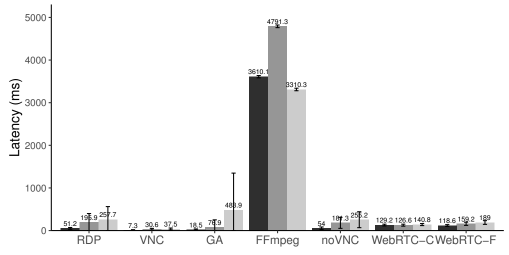
whenever possible. We note that due to space limitations, some detailed analysis results are given in Appendix A.

6.1 Performance Comparisons Under The Ideal Network Condition

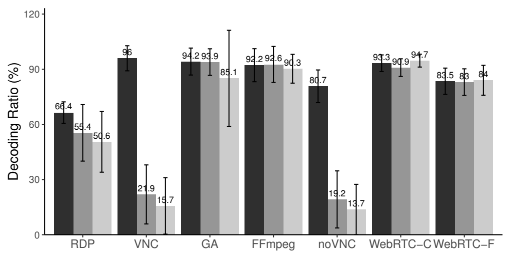
We first report the results under the ideal network condition (see Table 1) in Fig. 11. We make a few observations. First, Fig. 11a shows that VNC and noVNC incur high bandwidth consumption: 40.6 and 37.9 Mbps, respectively. These are more than 2.5 times the bandwidth consumption of RDP at 14.8 Mbps. All other considered screen sharing technologies only consume about 3 Mbps network bandwidth. A closer look indicates that the difference can be attributed to the codecs adopted by different screen sharing technologies. In particular, VNC and noVNC use the tight encoding algorithm, and RDP employs proprietary compression algorithms. All these three algorithms draw screens with some *graphics primitives*, such as rectangles with given width and height. In contrast, other screen sharing technologies employ the commodity video codecs. For example, FFmpeg and GA use H.264, and WebRTC-C and WebRTC-F use VP8 as the default codecs. Because video codecs are highly optimized, nontrivial difference on bandwidth consumption is observed.



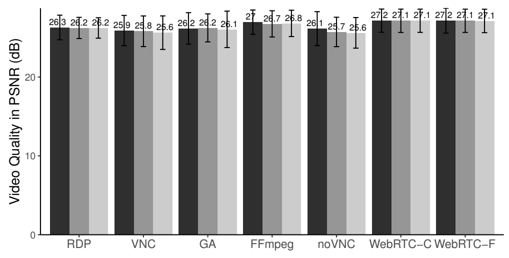
(a)



(b)



(c)



(d)

Fig. 12. Performance comparisons under different network bandwidth with no extra delay and packet loss rate: (a) bandwidth consumption, (b) latency, (c) decoding ratio, and (d) video quality in PSNR.

Next, Fig. 11b reveals that VNC and GA achieve very low latency: 7 ms and 18 ms, respectively. RDP and noVNC have slightly longer latency at 51 ms and 54 ms. These are trailed by WebRTC-C and WebRTC-F at 129 ms and 118 ms. FFmpeg suffers from an extremely long latency of nearly 3.5 seconds. Such a long latency may be due to its design: it adopts the FFmpeg APIs to capture the screens and an HTTP server to stream the captured screens. The HTTP server, unfortunately, incurs too much extra buffering delay, which results in long latency. Fig. 11c gives the decoding ratio. Among all screen sharing technologies, VNC achieves the highest decoding ratio at 96.2%. This is followed by GA at 94.4%. WebRTC-C and FFmpeg have decoding ratios of 93.5% and 92.4%, respectively. WebRTC-F and noVNC achieve around 80%. The worst one is the outdated RDP at 66.6%. Last, Fig. 11d shows that the screen sharing technologies achieve very similar video quality. Particularly, WebRTC-C, WebRTC-F and FFmpeg achieve the highest PSNR values at about 27 dB. RDP and GA achieve PSNR values at about 26 dB, while VNC and noVNC achieve PSNR values that are slightly lower than 26 dB. We note that the results from SSIM show similar trends, and thus are not plotted due to space limitations[3].

In summary, under the ideal network condition, VNC achieves the best overall performance. It achieves a low latency at 7.3 ms, a high decoding ratio at 96.2%, and fairly good video quality. However, we notice that VNC incurs a high bandwidth consumption of about 40 Mbps, which may be unrealistic in real-life scenarios. Hence, we discuss the performance of different screen sharing technologies under different network bandwidths in the next section. Last, because FFmpeg incurs an extremely long latency, we no longer consider it in the rest of this article.

[3]We only report sample PSNR results in the rest of this article, because the same trend is observed under other network conditions.

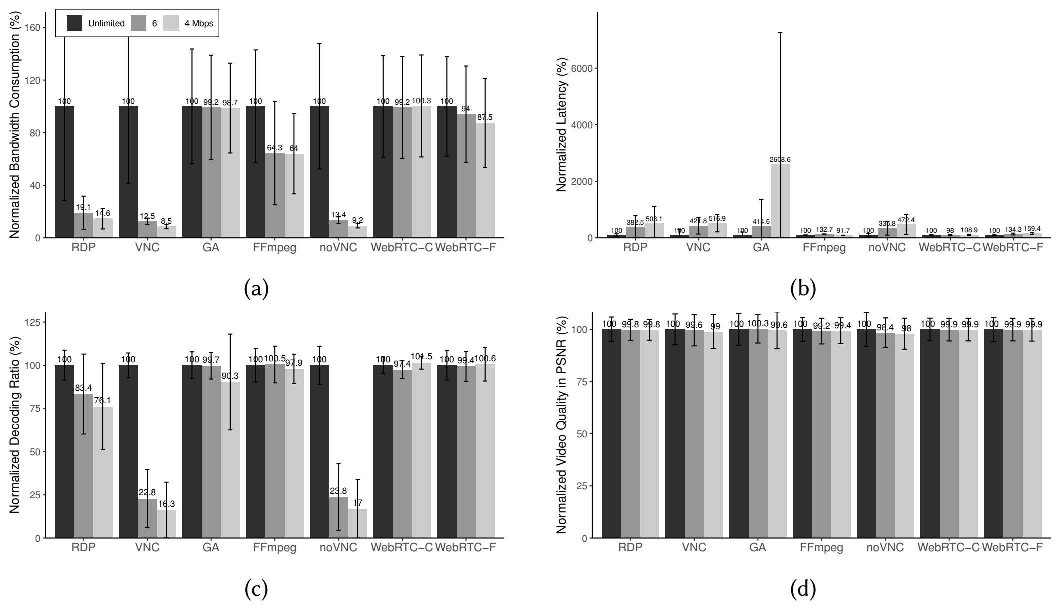


Fig. 13. Normalized performance comparisons under different network bandwidth with no extra delay and packet loss rate: (a) bandwidth consumption, (b) latency, (c) decoding ratio, and (d) video quality in PSNR. All bars are normalized to the ideal network condition with unlimited network bandwidth.

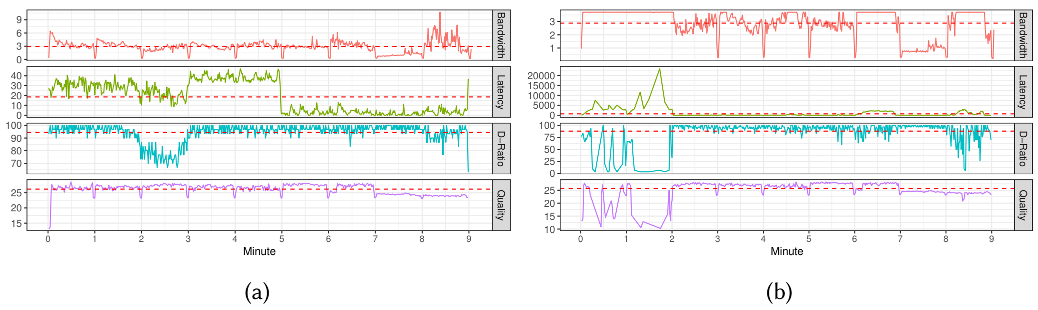


Fig. 14. The per-second performance results from GA: (a) no bandwidth limitation and (b) 4 Mbps bandwidth limitation. Dashed lines represent mean values.

6.2 Performance Implications of Bandwidth Limitations

We next analyze the implications of lower network bandwidth on the performance of different screen sharing technologies. Fig. 12 plots the results. First, Fig. 12a shows that the bandwidth consumption of noVNC, VNC, and RDP is indeed throttled at 4 (or 6) Mbps, which is quite different from those in Fig. 11a. Hence, they are expected to suffer from performance degradation under bandwidth limitations. Indeed, we observe their performance degradation in Fig. 12b, Fig. 12c, and Fig. 12d. Based on their performance, we can classify the screen sharing technologies into three groups: (1) WebRTC, (2) noVNC, VNC and RDP and (3) GA. For WebRTC, the bandwidth limitation has little, if any, influence on other performance metrics. As for noVNC, VNC and RDP, the deficiency of network bandwidth induces dramatic decreases on the decoding ratio. We also

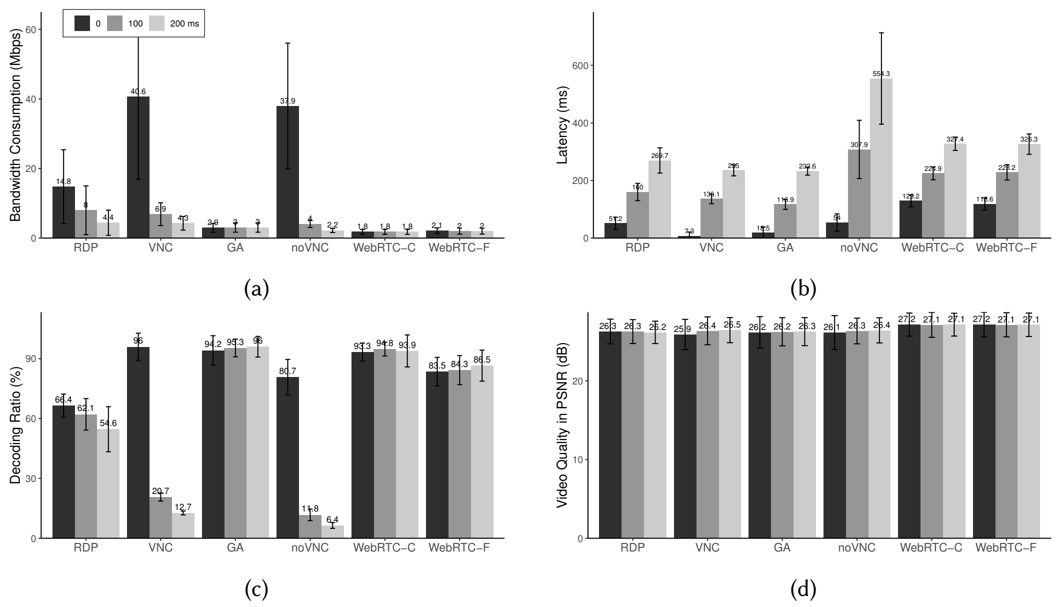


Fig. 15. Performance comparisons under different delays with no bandwidth limitation and packet loss rate: (a) bandwidth consumption, (b) latency, (c) decoding ratio, and (d) video quality in PSNR.

normalize the measurement results to those from the ideal network condition (unlimited network bandwidth), and plot the normalized results in Fig. 13. The figure further confirms the above observations.

Fig. 14 compares the performance results of GA under unlimited bandwidth and 4 Mbps, where the dashed lines indicate the mean values. Fig. 14a shows that GA may occasionally consume more bandwidth than 4 (or 6) Mbps, although its codec is configured for an average rate of 3 Mbps. This is because Average BitRate (ABR) instead of Constant BitRate (CBR) is employed by the default video codec library of GA. Fig. 14b gives the performance results of GA under 4 Mbps bandwidth limitation. In terms of bandwidth consumption, the curve is flat within applications 1, 2, and 9, which shows that GA needs more network bandwidth (than 4 Mbps) for some applications. For these applications, the corresponding latency increases significantly. In terms of video quality, it is observed that during applications 1 and 2, GA suffers from some degradation. This can be attributed to insufficient network bandwidth, which causes the frames to be delayed or distorted.

In summary, we observe that when the network bandwidth is reduced, the performance of VNC suffers, especially on the decoding ratio. GA performs better, as it achieves 18 ms latency, consumes less than 3 Mbps network bandwidth, and has a decoding ratio of 94.4%. Another good choice is WebRTC-C, which achieves a high decoding ratio of 93.5%, a high PSNR of 27.2 dB, and only consumes a low bandwidth of 2 Mbps.

Table 4. Measured TCP Throughput (Mbps) Under Different Network Conditions with *iperf3*

Unlimited	0%	1%	2%
0 ms	930	290	93
100 ms	15.5	2.94	2.06
200 ms	7.77	1.74	1.01

6 Mbps	0%	1%	2%
0 ms	5.8	5.79	5.61
100 ms	5.78	2.47	1.85
200 ms	5.55	1.58	1.03

4 Mbps	0%	1%	2%
0 ms	3.87	3.86	3.84
100 ms	3.86	2.79	1.71
200 ms	3.84	1.6	0.93

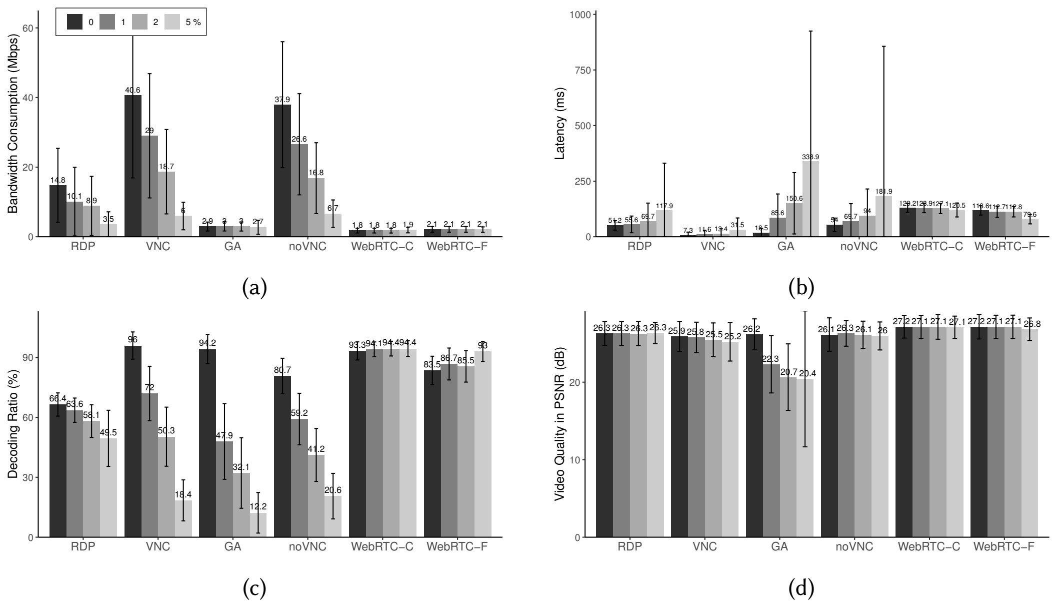


Fig. 16. Performance comparisons under different packet loss rates with no bandwidth limitation and delay: (a) bandwidth consumption, (b) latency, (c) decoding ratio, and (d) video quality in PSNR.

6.3 Performance Implications of Delay

Fig. 15 compares the performance under different delays at 0 (ideal network condition), 100, and 200 ms. We observe that with longer delays, VNC and noVNC suffer from larger decoding ratio drops (Fig. 15c). Moreover, the measured latency of noVNC is increased from 54 ms to 307 ms and 554 ms (Fig. 15b), which are significantly higher than the incurred extra delay. In fact, larger performance drops are seen with the screen sharing technologies that employ TCP transport protocol, which can be due to the TCP congestion control. This is validated by our *iperf3* experiments summarized in Table 4, which is collected from our testbed. We also observe that the UDP-based WebRTC and GA are rather resilient to higher delay (Fig. 15b): the only impact is the higher latency that is proportional to the incurred delay.

6.4 Performance Implications of Packet Loss Rate

Fig. 16 compares the performance at the different packet loss rates of 0%, 1%, 2%, and 5%. Differing from other screen sharing technologies, when the packet loss rate is increased from 0% to 5%, GA results in much higher latency (more than 15 times as shown in Fig. 16b) and nontrivial video quality drops (from 26 to 20 dB as shown in Fig. 16d). This is because GA adopts UDP transport protocol and does not implement the error resilience mechanism. Besides, Fig. 16c shows that, with most screen sharing technologies, the decoding ratio drops when the packet loss rate is increased. However, the decoding ratio of WebRTC-C/WebRTC-F is not affected by the packet loss rate. A closer look indicates that although WebRTC-C/WebRTC-F employ UDP protocol, they also enable FEC and NACK-based retransmission. Therefore, they are more resilient to packet loss. We study the WebRTC's error resilience mechanisms in Section 6.6.

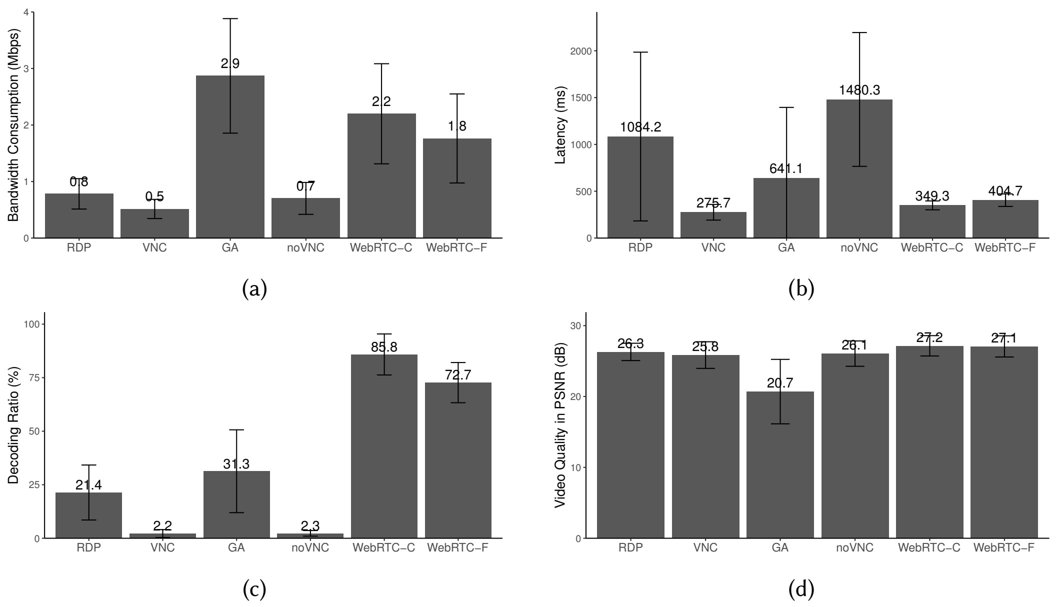


Fig. 17. Performance comparisons under the challenging network condition: (a) bandwidth consumption, (b) latency, (c) decoding rate, and (d) video quality in PSNR.

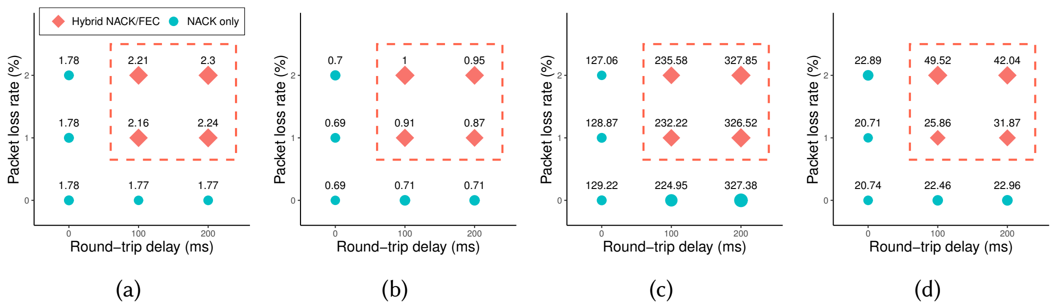
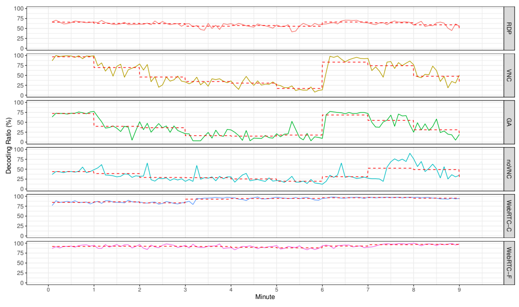


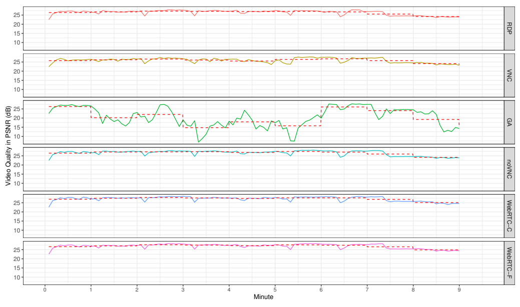
Fig. 18. WebRTC-C with unlimited bandwidth, latency varying among 0, 100 to 200 ms, and packet loss rate varying among 0, 1 to 2%: (a) mean bandwidth consumption, (b) standard deviation of bandwidth consumption, (c) mean latency, and (d) standard deviation of latency. The marker size represents the magnitude of the value. Dashed boxes indicate the network conditions where FEC is enabled.

6.5 Performance Comparisons Under The Challenging Network Condition

Next, we report the results from the challenging network condition (see Table 1) in Fig. 17. We observe that WebRTC-C retains a high decoding ratio of 86% and good video quality of 27 dB; while WebRTC-F also performs well. The TCP-based screen sharing technologies, including RDP, VNC, and noVNC suffer from low decoding ratio due to the TCP congestion control. Last, the less ideal decoding ratio of GA can be attributed to its lack of error resilience mechanism as discussed above. *In summary, WebRTC outperforms other screen sharing technologies under challenging network conditions due to the combination of UDP transport protocol and error resilience mechanisms.*



(a)



(b)

Fig. 19. Performance when packet loss rate is changed once every minute: (a) decoding ratio and (b) video quality in PSNR. Dashed lines represent mean values within each packet loss rate setting (roughly one minute).

6.6 Error Resilience Mechanism in WebRTC

Our measurement results presented above show that WebRTC (especially WebRTC-C, which runs in Chrome) is robust against diverse network conditions. For example, WebRTC-C achieves at least 86% decoding ratio, less than 350 ms latency, and at least 27 dB video quality in PSNR, while only consuming less than 3 Mbps bandwidth. This is made possible by several *error resilience* mechanisms, as we describe below based on analyzing network traffic and tracing their source code. First, we note that the WebRTC implementation in Firefox and Chrome both realize NACK to cope with packet loss. However, their actual implementations are slightly different. WebRTC-F retransmits the lost packets within the original channel along with the regular media stream, while WebRTC-C employs a different channel for retransmission. Hence, WebRTC-F has to treat the retransmitted packets as *out-of-order* packets, since they are streamed along with regular packets.

In addition to NACK, WebRTC-C also enables its Forward Error Correction (FEC) mechanism once the network delay and packet loss become non-trivial. When the FEC is enabled, the FEC codes are appended to the payloads of individual packets. More precisely, WebRTC-C switches between two modes: NACK and NACK/FEC. Fig. 18 reports the performance of WebRTC-C under different network conditions. From the bandwidth consumption given in Fig. 18a, we reverse engineer the logic of its mode switching: NACK/FEC is adopted when the packet loss rate exceeds 1% and the delay exceeds 100 ms. A closer look into the source code confirms the above experiment results: in fact, the NACK/FEC is enabled once the packet loss rate is nonzero and the delay is longer than 20 ms. We emphasize that the switching logic is only true for WebRTC-C (Chrome); WebRTC-F (Firefox) only implements NACK for error resilience.

6.7 Implications of Dynamic Network Conditions

Networks are dynamic in various aspects. Due to the space limitations, we focus on the two most critical aspects: the packet loss rate and link failure events, which may lead to a catastrophic drop in user experience. We first compare the performance of screen sharing technologies under a changing packet loss rate as follows. When the experiment starts, we set the traffic controller to incur no packet loss. We then increase the packet loss rate by 1% every minute until it reaches 5%. One minute after that, we reset the packet loss rate to 0% again. Each experiment lasts for 9 minutes. We plot the performance of different screen sharing technologies in Fig. 19. Fig. 19a shows that only VNC, GA, and noVNC suffer from fluctuating decoding ratios under dynamic packet loss rates.

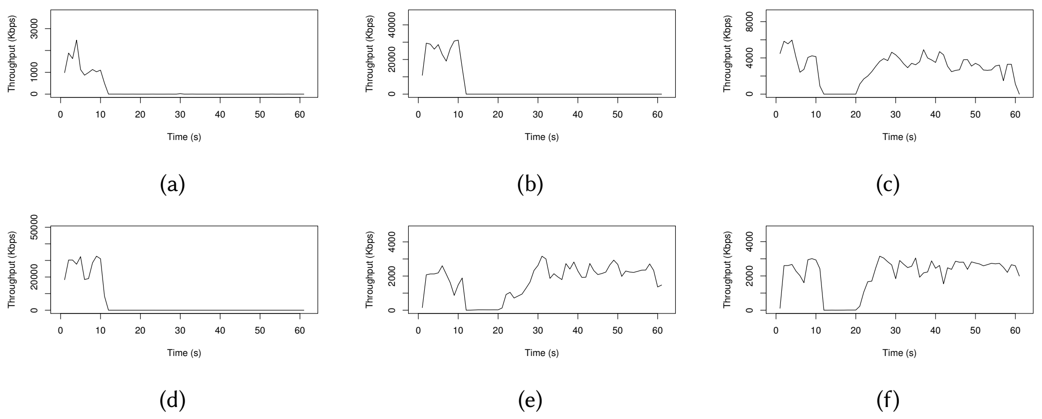


Fig. 20. Client throughput over time after a 10-sec link failure (∈ [10, 20]): (a) RDP, (b) VNC, (c), GA, (d) noVNC, (e) WebRTC-C, and (f) WebRTC-F.

Nonetheless, after 6 minutes, VNC and GA quickly recover from an inferior (about 25%) decoding ratio. Fig. 19b reveals that only GA suffers from fluctuating video quality under dynamic packet loss rates. GA, however, recovers after 6 minutes. In summary, Fig. 19 shows that WebRTC-C and WebRTC-F perform the best under dynamic packet loss rates, which is followed by RDP.

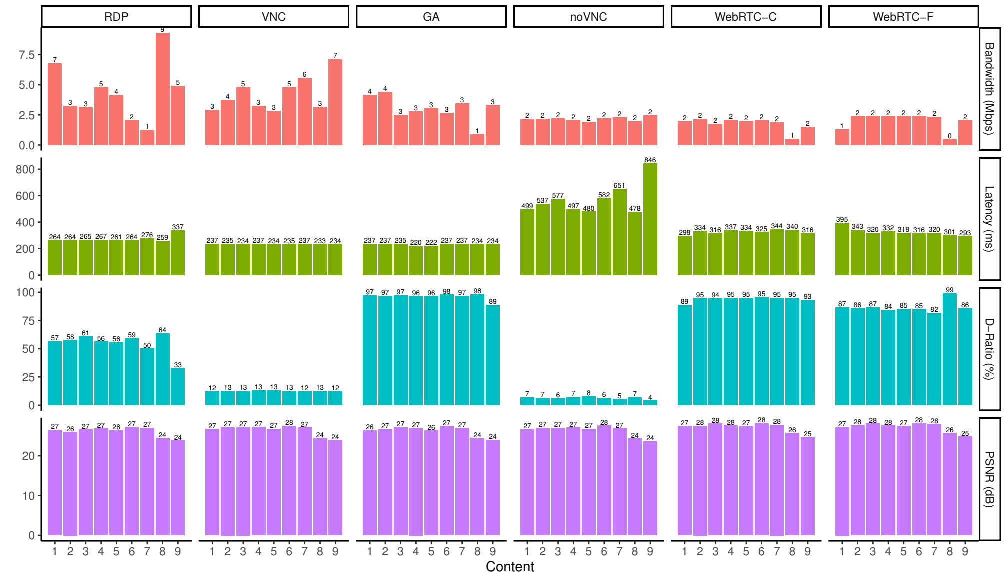
Next, we study the recovery speed of different screen sharing technologies after link failures. We start screen sharing sessions and manually introduce link failure between the 10- and 20-th sec using *iptables*[4]. Then, we observe whether the client recovers from the link failure. We plot the throughput at the client side in Fig. 20. This figure reveals that: (i) the UDP-based GA and WebRTC-C/WebRTC-F screen sharing technologies recover from the link failure soon after the 20-th sec, and (ii) the TCP-based VNC, RDP, and noVNC never recover from the link failure[5]. We like to mention that the throughput of FFmpeg recovers 10-sec after the link recovery (figure omitted for brevity), although it is based on TCP. This may be attributed to the fact that FFmpeg is rendered in HTML video tag. However, with FFmpeg, the client suffers from video freezes even after the throughput is recovered, which indicates that the received data are undecodable.

Overall, the TCP-based screen sharing technologies react more slowly than the UDP-based one under dynamic network conditions. Among the UDP-based screen sharing technologies, some mechanisms of monitoring the network conditions are needed to mitigate poor network quality or even terminate the network connections whenever necessary.

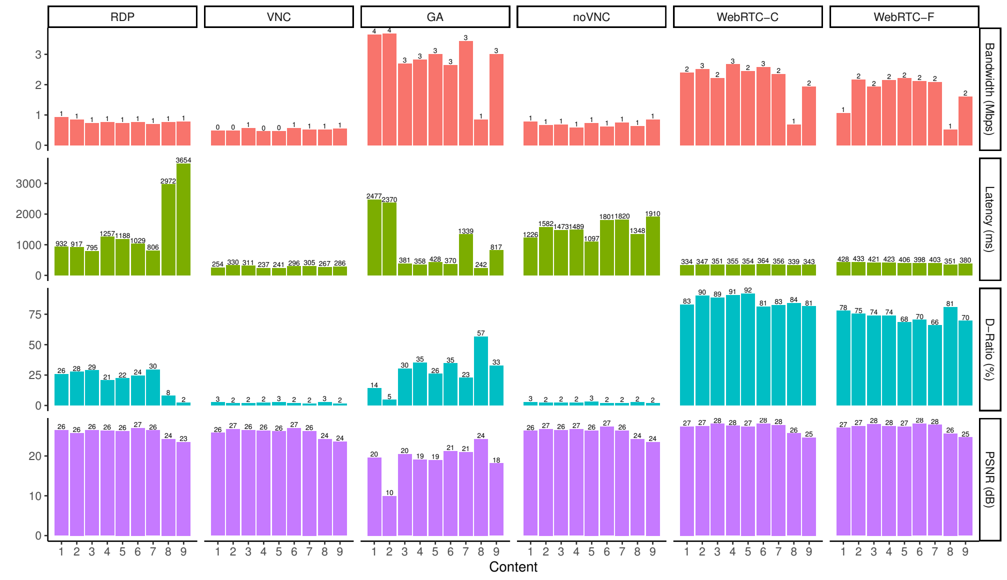
6.8 Summary and Recommendations

Following the measurement results, we classify the screen sharing technologies into three classes: (1) WebRTC, (2) GA, and (3) VNC, noVNC, and RDP. While both WebRTC and GA use RTP/UDP protocols, WebRTC is implemented with error resilience mechanisms, and thus is robust against packet loss and dynamic network conditions. In contrast, GA suffers from degraded decoding ratio under nontrivial packet loss rates, but achieves lower latency in general. Because WebRTC and GA both adopt UDP protocol, longer delay and higher packet loss rate do not affect their bandwidth consumption compared to the TCP protocol. Furthermore, the average bandwidth consumptions

[4]We chose to use *iptables* instead of removing the cable for more challenging situations. This is because removing the cable would lead to link-down events from Ethernet ports.
[5]We plot this figure for 60 sec for clarity, although we wait for much longer for the screen sharing technologies to recover.



(a)



(b)

Fig. 21. Performance summary for each class of applications under different network conditions. Sample results from: (a) high delay and (b) challenging network conditions are shown.

of WebRTC and GA are no higher than 4 Mbps. The codecs of VNC, noVNC, and RDP are quite bandwidth hungry. In the ideal network condition, these screen sharing technologies provide lower latency than WebRTC. However, network resources for most users are limited. Because these three technologies adopt TCP protocol, their performance drops once the delay and packet loss rate become nontrivial.

Last, we look into the performance of different application classes and give two sample results in Fig. 21. This figure reveals that the performance of individual screen sharing technologies differ greatly for diverse applications. For example, GA performs better with games and movies, but slightly worse with web browsing. Specifically, Fig. 21a shows the sample results under the high delay network condition. The figure shows that GA has a good balance among video quality, decoding ratio, and latency. It is therefore good for playing games, watching movies, and working with most desktop apps. We note that the results from the ideal network are similar, except for the shorter latency (about 200 ms less compared to the high delay network condition). Fig. 21b gives the sample results from the challenging network condition. It is clear that GA's performance suffers under nontrivial packet loss rate and insufficient network bandwidth, due to its lack of error resilience mechanism. In contrast, WebRTC performs fairly stably and provides good video quality and acceptable decoding ratio in the challenging network condition. Based on the aforementioned observations, we provide the recommended screen sharing technologies for different application classes in Table 5. In short, if short latency is required and the network condition is not bad, we recommend that users adopt the (native) GA [18]. Otherwise, we recommend that users adopt the (clientless) WebRTC [40].

Table 5. Recommended Screen Sharing Technologies for Different Applications and Network Conditions

Network Condition	Game	Movie	Desktop Apps
Ideal	GA	GA/WebRTC	GA
Lossy	WebRTC	WebRTC	WebRTC
High Delay	GA	GA/WebRTC	GA
Low Bandwidth	WebRTC	WebRTC	WebRTC
Challenging	WebRTC	WebRTC	WebRTC

7 CONCLUSION

From the extensive experiment results, we conclude that WebRTC is a promising screen sharing technology. Except for a higher latency of about 120 ms, WebRTC outperforms other screen sharing technologies. It consumes low bandwidth which is no higher than 3 Mbps, yet delivers high video quality and decoding ratio. WebRTC also has the best ability to adapt to changing network conditions and recover from link failure. Besides, WebRTC enables clientless screen sharing using web browsers, which relieve users from installing software. With WebRTC APIs, developers can focus on the protocol design and codec selection without worrying about the OS- and device-specific details. Our proposed measurement methodology can be seen as a contribution in its own right. The captured packets and videos are programmatically analyzed into performance metrics, including bandwidth consumption, latency, decoding ratio, and video quality. The procedure is applicable to any screen sharing technologies, including both the native and clientless screen sharing technologies.

Last, we remark that the *gap* between the native and clientless screen sharing technologies has continued to *shrink* over the past few years as WebRTC APIs are widely implemented in mainstream web browsers. We believe that the WebRTC has become mature enough to support screen sharing:

(1) in most network conditions except the *ideal network condition*, in which technologies like VNC trades high bandwidth consumption for better video quality;
(2) for most applications excluding *fast-paced games* such as first-person shooter games, in which extremely low latency is a must for acceptable user experience.

Several future research directions may address the above two limitations. For example, more adaptive codecs can be used in WebRTC, in order to *capitalize* the available bandwidth in the ideal network condition for better screen sharing performance. Besides, various *latency reduction mechanisms* proposed for cloud gaming, such as image-based warping [35] and the zero-buffering mechanism [18], can be integrated into fast-paced games and WebRTC APIs to achieve extremely low latency. Furthermore, more comprehensive measurement experiments can be designed to investigate the intelligent adaptation algorithms. The experiment results may provide more insights to further optimize the screen sharing technologies for different applications under diverse network conditions and heterogeneous clients (including desktops and mobile devices). In addition to video, the performance of audio streaming is also important and worth to be evaluated for some particular applications, such as games and movies. This is one of our future directions. We firmly believe that this article will stimulate many exciting works, which in turn make clientless screen sharing technologies applicable to more interactive applications under more diverse network conditions.

REFERENCES

[1] Maha Abdallah, Carsten Griwodz, Kuan-Ta Chen, Gwendal Simon, Pin-Chun Wang, and Cheng-Hsin Hsu. 2018. Delay-Sensitive Video Computing in the Cloud: A Survey. *ACM Transactions on Multimedia Computing, Communications, and Applications* 14, 3s (2018), 54:1–54:29.

[2] Doreid. Ammar, Katrien De Moor, Min Xie, Markus Fiedler, and Poul Heegaard. 2016. Video QoE Killer and Performance Statistics in WebRTC-Based Video Communication. In *Proc. of IEEE International Conference on Communications and Electronics (ICCE'16)*. 429–436.

[3] Daniel Beer. 2019. QR Decoder Library. (2019). https://github.com/dlbeer/quirc

[4] Sumohana S. Channappayya, Alan C. Bovik, Constantine Caramanis, and Robert W. Heath. 2008. SSIM-Optimal Linear Image Restoration. In *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'08)*. Las Vegas, USA, 765–768.

[5] Hao Chen, Xu Zhang, Yiling Xu, Ju Ren, Jingtao Fan, Zhan Ma, and Wenjun Zhang. 2019. T-gaming: A cost-efficient cloud gaming system at scale. *IEEE Transactions on Parallel and Distributed Systems* 30, 12 (2019), 2849–2865.

[6] Mark Claypool and Kajal Claypool. 2010. Latency Can Kill: Precision and Deadline in Online Games. In *Proc. of ACM SIGMM Conference on Multimedia Systems (MMSys'10) (MMSys '10)*. Phoenix, AZ, 215–222.

[7] Mark Claypool and David Finkel. 2014. The Effects of Latency on Player Performance in Cloud-Based Games. In *Proc. of ACM Workshop on Network and Systems Support for Games (NetGames'14)*. 1–6.

[8] Mark Claypool, Tianhe Wang, and McIntyre Watts. 2015. A Taxonomy for Player Actions with Latency in Network Games. In *Proc. of ACM Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'15)*. Portland, Oregon, 67–72.

[9] DENSO WAVE INCORPORATED. 2019. QRcode.com | DENSO WAVE. (2019). https://www.qrcode.com/

[10] FFmpeg Team. 2019. FFmpeg. (2019). http://ffmpeg.org/

[11] Rama Rao Ganji, Mihai Mitrea, Dancho Panovski, and Bojan Joveski. 2016. Improving the RDP Based Applications by Using HTML5 Content Representation. *Electronic Imaging* 2016 (February 2016), 1–7.

[12] Boni García, Luis López-Fernández, Micael Gallego, and Francisco Gortázar. 2016. Testing Framework for WebRTC Services. In *Proc. of EAI International Conference on Mobile Multimedia Communications (MobiMedia'16)*. Xi'an, China, 40–47.

[13] GlavSoft LLC. 2019. TightVNC: VNC-Compatible Free Remote Control / Remote Desktop Software. (June 2019). https://www.tightvnc.com/

[14] Google. 2019. Stadia. (2019). https://stadia.dev/

[15] Chih-Fan Hsu, De-Yu Chen, Chun-Ying Huang, Cheng-Hsin Hsu, and Kuan-Ta Chen. 2014. Screencast in the wild: performance and limitations. In *Proc. of ACM International Conference on Multimedia (MM'14)*. 813–816.

[16] Chih-Fan Hsu, Ching-Ling Fan, Tsung-Han Tsai, Chun-Ying Huang, Cheng-Hsin Hsu, and Kuan-Ta Chen. 2016. Toward an Adaptive Screencast Platform: Measurement and Optimization. *ACM Transactions on Multimedia Computing, Communications, and Applications* 12, 5s (2016), 79:1–79:23.

[17] Chih-Fan Hsu, Tsung-Han Tsai, Chun-Ying Huang, Cheng-Hsin Hsu, and Kuan-Ta Chen. 2015. Screencast Dissected: Performance Measurements and Design Considerations. In *Proc. of the ACM SIGMM Conference on Multimedia Systems (MMsys'15)*. Portland, Oregon, 177–188.

[18] Chun-Ying Huang, Kuan-Ta Chen, De-Yu Chen, Hwai-Jung Hsu, and Cheng-Hsin Hsu. 2014. GamingAnywhere: The First Open Source Cloud Gaming System. *ACM Transactions on Multimedia Computing Communications and*

Applications 10, 1s (January 2014), 10:1–10:25.

[19] ISO/IEC 16022:2006(E) 2006. *Information Technology — Automatic Identification and Data Capture Techniques — Data Matrix Bar Code Symbology Specification.* Standard. International Organization for Standardization/International Electrotechnical Commission.

[20] Benjamin F. Janzen and Robert J. Teather. 2014. Is 60 FPS Better Than 30?: The Impact of Frame Rate and Latency on Moving Target Selection. In *Proc. of the Extended Abstracts of ACM Conference on Human Factors in Computing Systems (CHI EA'14).* Toronto, Ontario, Canada, 1477–1482.

[21] Youming Lin, Teemu Kämäräinen, Mario Di Francesco, and Antti Ylä-Jääski. 2015. Performance Evaluation of Remote Display Access for Mobile Cloud Computing. *Computer Communications* 72 (2015), 17 – 25.

[22] Luis López, Miguel París, Santiago Carot, Boni García, Micael Gallego, Francisco Gortázar, Raul Benítez, Jose A. Santos, David Fernández, Radu Tom Vlad, Iván Gracia, and Francisco Javier López. 2016. Kurento: the WebRTC Modular Media Server. In *Proc. of ACM International Conference on Multimedia (MM'16).* Amsterdam, The Netherlands, 1187–1191.

[23] Joel Martin. 2019. noVNC. (April 2019). https://novnc.com/info.html

[24] Microsoft Corp. 2018. Remote Desktop Protocol. (May 2018). https://docs.microsoft.com/en-us/windows/desktop/termserv/remote-desktop-protocol

[25] Microsoft Corp. 2019. Use Remote Assistance to Let Someone Fix Your PC. (January 2019). https://support.microsoft.com/en-us/help/4026516/windows-use-remote-assistance-to-let-someone-fix-your-pc

[26] MinGW.org. 2019. MinGW | Minimalist GNU for Windows. (July 2019). http://www.mingw.org/

[27] Miniwatts Marketing Group. 2019. World Internet Users Statistics and 2019 World Population Stats. (2019). www.internetworldstats.com/stats.htm

[28] Yasuhiro Mochida, Daisuke Shirai, and Tatsuya Fujii. 2016. Novel Web-based Remote Collaboration System Architecture for Wider Use Cases. In *Proc. of International Conference on Supporting Group Work (GROUP'16).* Sanibel Island, Florida, 437–440.

[29] Davide Mulfari, Antonio Celesti, Massimo Villari, and Antonio Puliafito. 2014. Using Virtualization and noVNC to Support Assistive Technology in Cloud Computing. In *Proc. of IEEE Symposium on Network Cloud Computing and Applications (NCCA'14).* 125–132.

[30] Hyunwoo Nam, Kyung-Hwa Kim, and Henning Schulzrinne. 2016. QoE Matters More Than QoS: Why People Stop Watching Cat Videos. In *Proc. of IEEE International Conference on Computer Communications (INFOCOM'16).* San Francisco, CA, 1–9.

[31] NVIDIA. 2019. Game Anywhere on Your Mac, Windows PC, or SHIELD Device with NVIDIA's Cloud Gaming Service. (2019). https://www.nvidia.com/en-us/geforce/products/geforce-now/

[32] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. 1998. Virtual Network Computing. *IEEE Internet Computing* 2 (February 1998), 33–38.

[33] Ron Sharp. 2012. Latency in Cloud-Based Interactive Streaming Content. *Bell Labs Technical Journal* 17, 2 (September 2012), 67–80.

[34] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. 2003. The Effect of Latency on User Performance in Warcraft III. In *Proc. of ACM Workshop on Network and System Support for Games (NetGames'03).* Redwood City, CA, 3–14.

[35] Shu Shi, Cheng Hsu, Klara Nahrstedt, and Roy Campbell. 2011. Using Graphics Rendering Contexts to Enhance the Real-Time Video Coding for Mobile Cloud Gaming. *Proc. of ACM International Conference on Multimedia (MM'11)* (November 2011), 103–112.

[36] Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *Comput. Surveys* 47, 4 (2015), 57:1–57:29.

[37] Sony Interactive Entertainment. 2019. PS4 Remote Play Windows PC/Mac. (2019). https://remoteplay.dl.playstation.net/remoteplay/lang/en/index.html

[38] Matthias Ueberheide, Felix Klose, Tilak Varisetty, Markus Fidler, and Marcus Magnor. 2015. Web-Based Interactive Free-Viewpoint Streaming: A Framework for High Quality Interactive Free Viewpoint Navigation. In *Proc. of ACM International Conference on Multimedia (MM'15).* Brisbane, Australia, 1031–1034.

[39] Valve Corporation. 2019. Steam In-Home Streaming. (2019). https://store.steampowered.com/streaming/

[40] WebRTC. 2019. WebRTC Home | WebRTC. (June 2019). https://webrtc.org/

[41] Li Yan. 2011. Development and Application of Desktop Virtualization Technology. In *Proc. of IEEE International Conference on Communication Software and Networks (ICCCN'11).* Maui, HI, 326–329.

[42] Youhui Zhang, Peng Qu, Jiang Cihang, and Weimin Zheng. 2015. A cloud gaming system based on user-level virtualization and its resource scheduling. *IEEE Transactions on Parallel and Distributed Systems* 27, 5 (2015), 1239–1252.