GamingAnywhere: An Open Cloud Gaming System

Chun-Ying Huang[1], Cheng-Hsin Hsu[2], Yu-Chun Chang[3,4], and Kuan-Ta Chen[3]

[1]Department of Computer Science, National Taiwan Ocean University
[2]Department of Computer Science, National Tsing Hua University
[3]Institute of Information Science, Academia Sinica
[4]Department of Electrical Engineering, National Taiwan University

chuang@ntou.edu.tw, chsu@cs.nthu.edu.tw, congo@iis.sinica.edu.tw, ktchen@iis.sinica.edu.tw

ABSTRACT

Cloud gaming is a promising application of the rapidly expanding cloud computing infrastructure. Existing cloud gaming systems, however, are closed-source with proprietary protocols, which raises the bars to setting up testbeds for experiencing cloud games. In this paper, we present a complete cloud gaming system, called GamingAnywhere, which is to the best of our knowledge the first open cloud gaming system. In addition to its openness, we design GamingAnywhere for high extensibility, portability, and reconfigurability. We implement GamingAnywhere on Windows, Linux, and OS X, while its client can be readily ported to other OS's, including iOS and Android. We conduct extensive experiments to evaluate the performance of GamingAnywhere, and compare it against two well-known cloud gaming systems: OnLive and StreamMyGame. Our experimental results indicate that GamingAnywhere is efficient and provides high responsiveness and video quality. For example, GamingAnywhere yields a per-frame processing delay of 34 ms, which is 3+ and 10+ times shorter than OnLive and StreamMyGame, respectively. Our experiments also reveal that all these performance gains are achieved without the expense of higher network loads; in fact, GamingAnywhere incurs less network traffic. The proposed GamingAnywhere can be employed by the researchers, game developers, service providers, and end users for setting up cloud gaming testbeds, which, we believe, will stimulate more research innovations on cloud gaming systems.

Categories and Subject Descriptors: H.5[Information Systems Applications]: Multimedia Information Systems

General Terms: Design, Measurement

1. INTRODUCTION

The video game market has been an important sector of both the software and entertainment industries, e.g., the global market of video games is expected to grow from 66 billion US dollars in 2010 to 81 billion in 2016 [27]. Another market research [11] further breaks down the market growth into three categories: boxed-games, online-sold games, and *cloud games*. Cloud gaming systems render the game scenes on cloud servers and stream the encoded game scenes to thin clients over broadband networks. The control events, from mice, keyboards, joysticks, and touchscreens are transmitted from the thin clients back to the cloud servers. Among the three categories, it is the cloud games market that is expected to expand the most: nine times over the period of 2011 to 2017, at which time it is forecast to reach 8 billion US dollars [11].

Cloud gaming systems attract both users and game developers for many reasons. In particular, cloud gaming systems: (i) free users from upgrading their hardware for the latest games, (ii) allow users to play the same games on different platforms, including PCs, laptops, tablets, and smartphones, and (iii) enable users to play more games by reducing the hardware/software costs. Cloud gaming systems also allow game developers to: (i) support more platforms, (ii) ease hardware/software incompatibility issues, (iii) reduce the production costs, and (iv) increase the net revenues. In fact, the market potential of cloud gaming is tremendous and well recognized, as evidenced by Sony's recent acquisition [9] of Gaikai [15], which is a cloud gaming service provider.

However, providing a good user experience in cloud gaming systems is not an easy task, because users expect both high-quality videos *and* low response delays. The response delay refers to the time difference between a user input received by the thin client and the resulting in-game action appearing on the thin client's screen. Higher-quality videos, such as 720p (1280x720) at 50 fps (frame-per-second), inherently result in higher bit rates, which render the cloud gaming systems vulnerable to higher network latency, and thus longer response delay. Since a long response delay results in degraded user experience, it may turn the users away from the cloud gaming systems. User studies, for example, show that networked games require short response delay, even as low as 100 ms, e.g., for first-person shooter games [8]. Given that each game scene has to go through the real-time video streaming *pipeline* of rendering, encoding, transmission, decoding, and displaying, designing a cloud gaming system to meet the stringent response delay requirements, while still achieving high video quality is very challenging.

One simple approach to support cloud gaming is to employ generic desktop streaming thin clients, such as LogMeIn [24], TeamViewer [36], and UltraVNC [38]. However, a measurement study [4] reveals that generic thin clients achieve low frame rates, on average 13.5 fps, which clearly

MMSys'13, February 26-March 1, 2013, Oslo, Norway.

lead to sluggish game plays. A better user experience is possible with thin clients specifically designed for cloud gaming, e.g., Gaikai [15], OnLive [29], and StreamMyGame [34]. Nevertheless, another measurement study [5] demonstrates that these cloud gaming systems also suffer from non-trivial response time. Even under a strong assumption of *zero* network latency, at least 134 ms and 375 ms average response delays are observed in OnLive and StreamMyGame, respectively. The measurement results [4, 5] indicate that the problem of designing cloud gaming systems for high video quality and fast responsiveness remains open.

In this work, we design, implement, and evaluate a cloud gaming system called *GamingAnywhere*. GamingAnywhere follows three design goals. First, in contrast to OnLive and StreamMyGame, GamingAnywhere is an *open* system, in the sense that a component of the video streaming pipeline can be easily replaced by another component implementing a different algorithm, standard, or protocol. For example, GamingAnywhere by default employs `x264` [44], a highly-optimized H.264/AVC encoder, to encode captured raw videos. To expand GamingAnywhere for stereoscopic games, an H.264/MVC encoder may be plugged into it without significant changes. Since GamingAnywhere is open, various algorithms, standards, protocols, and system parameters can be rigorously evaluated using real experiments, which is impossible on proprietary cloud gaming systems. Second, GamingAnywhere is *cross-platform*, and is currently available on Windows, Linux, and OS X. This is made possible largely due to the modularized design of GamingAnywhere. Third, GamingAnywhere has been designed to be *efficient*, as can be seen, for example, in its minimizing of time and space overhead by using shared circular buffers to reduce the number of memory copy operations. These optimizations allow GamingAnywhere to provide a high-quality gaming experience with short response delay. In particular, on a commodity Intel i7 server, GamingAnywhere delivers real-time 720p videos at $\geq$ 35 fps, which is equivalent to less than 28.6 ms of processing time for each video frame, with a video quality significantly higher than that of existing cloud gaming systems. In particular, GamingAnywhere achieves a video quality 3 dB and 19 dB higher than that of OnLive and StreamMyGame, in terms of average Peak Signal-to-Noise Ratio (PSNR). PSNR is popular video quality metric, which is inversely related to mean-squared error [40, p. 29].

This paper makes two main contributions.

- We propose an open cloud gaming system, GamingAnywhere, which can be used by cloud gaming developers, cloud service providers, and system researchers for setting up a complete cloud gaming testbed. To the best of our knowledge, this is the first open cloud gaming testbed in the literature.
- We conduct extensive experiments using GamingAnywhere to quantify its performance and overhead. We also derive the optimal setups of system parameters, which in turn allow users to install and try out GamingAnywhere on their own servers.

The rest of this paper is organized as follows. Section 2 surveys related work in the literature. Section 3 presents the design goals and Section 4 depicts the system architecture. This is followed by the detailed implementations in Section 5. Section 6 gives the performance evaluation results. We conclude the paper in Section 7.

2. RELATED WORK

In this section, we survey the existing cloud gaming systems and the proposals for measuring their performance.

2.1 Cloud Gaming Systems

Cloud gaming systems, or more generally real-time remote rendering systems, have been studied in the literature. We classify these systems into three categories: (i) 3D graphics streaming [12, 19], (ii) video streaming [18, 42], and (iii) video streaming with post-rendering operations [16, 33]. These three approaches differ from one another in how they divide the workload between the cloud servers and clients.

With the 3D graphics streaming approach [12, 19], the cloud servers intercept the graphics commands, compress the commands, and stream them to the clients. The clients then render the game scenes using its graphics chips based on graphics command sets such as OpenGL and Direct3D. The clients' graphics chips must be not only compatible with the streamed graphics commands but also powerful enough to render the game scenes in high quality and real time. 3D graphics streaming approach does not use the graphics chips on the cloud servers, thereby allowing each cloud server to concurrently support multiple clients. However, as this approach imposes more workload on the clients, it is less suitable for resource-constrained devices, such as mobile devices and set-top boxes.

In contrast, with the video streaming approach [18, 42] the cloud servers render the 3D graphics commands into 2D videos, compress the videos, and stream them to the clients. The clients then decode and display the video streams. The decoding can be done using low-cost video decoder chips massively produced for consumer electronics. This approach relieves the clients from computationally-intensive 3D graphics rendering and is ideal for thin clients on resource-constrained devices. Since the video streaming approach does not rely on specific 3D chips, the same thin clients can be readily ported to different platforms, which are potentially GPU-less.

The approach of video streaming with post-rendering operations [16, 33] is somewhere between the 3D graphics streaming and video streaming. While the 3D graphics rendering is performed at the cloud servers, some post-rendering operations are optionally done on the thin clients for augmenting motions, lighting, and textures [6]. These post-rendering operations have low computational complexity and run in real time without GPUs.

Similar to the proprietary cloud gaming systems [15, 29, 34], the proposed GamingAnywhere employs the video streaming approach for lower loads on the thin clients. Differing from other systems [18, 42] in the literature, GamingAnywhere is open, modularized, cross-platform, and efficient. To the best of our knowledge, GamingAnywhere is the first complete system of its kind, and is of interests for researchers, cloud gaming service providers, game developers, and end users. Last, GamingAnywhere is flexible and can be extended to evaluate the potentials and performance impact of post-rendering operations. This is one of our future tasks.

2.2 Measuring the Performance of Cloud Gaming Systems

Measuring the performance of general-purpose thin client systems has been considered in the literature [20, 26, 30,

37, 43]. The slow-motion benchmarking [20, 26] runs a slow-motion version of an application on the server, and collects network packet traces between the server and thin client. It then analyzes the traces for the performance of the thin client system. However, slow-motion benchmarking augments the execution speed of applications, and is thus less suitable to real-time applications, including cloud games. The performances of different thin clients are investigated, including X Window [30], Windows NT Terminal Service [43], and VNC (Virtual Network Computing) [37]. Packard and Gettys [30] analyze the network traces between the X Window server and client, under diverse network conditions. The traces are used to compare the compression ratios of different compression mechanisms, and to quantify the effects of network impairments. Wong and Seltzer [43] measure the performance of the Windows NT Terminal Service, in terms of process, memory, and network bandwidth. The Windows NT Terminal Service is found to be generally efficient with multi-user access, but the response delay is degraded when the system load is high. Tolia et al. [37] quantify the performance of several applications running on a VNC server, which is connected to a VNC thin client via a network with diverse round-trip times (RTT). It is determined that the response delay of these applications highly depends on the degree of the application's interactivity and network RTT. However, because the aforementioned techniques [20, 26, 30, 37, 43] are designed for general-purpose thin clients, the performance metrics they consider are not applicable to cloud gaming systems, which impose stringent time constraints.

Recently, the performance and potentials of thin client gaming [4, 5, 7, 22] has been the subject of research. Chang et al.'s [4] methodology to study the performance of games on general-purpose thin clients has been employed to evaluate several popular thin clients, including LogMeIn [24], TeamViewer [36], and UltraVNC [38]. Chang et al. establish that player performance and Quality-of-Experience (QoE) depend on video quality and frame rates. It is observed that the general-purpose thin clients cannot support cloud games given that the achieved frame rate is as low as 9.7 fps [4]. Chen et al. [5] propose another methodology to quantify the response delay, which is even more critical to cloud games [8, 17, 46]. Two proprietary cloud gaming systems, OnLive [29] and StreamMyGame [34], are evaluated using this methodology. Their evaluation results reveal that StreamMyGame suffers from a high response delay, while OnLive achieves reasonable response delay. Chen et al. partially attribute the performance edge of OnLive to its customized hardware platform, which however entails a high infrastructure cost [28]. In addition, Lee et al. [22] evaluate whether computer games are equally suitable to the cloud gaming setting and find that some games are more "compatible" with cloud gaming than others. Meanwhile, Choy et al. [7] evaluate whether a wide-scale cloud gaming infrastructure is feasible on the current Internet and propose a smart-edge solution to mitigate user-perceived delays when playing on the cloud.

In light of the literature review, the current paper tackles the following question: *Can we do better than OnLive using commodity desktops?* We employs the measurement methodologies proposed in [5] to compare the proposed GamingAnywhere against the well-known cloud gaming systems of OnLive [29] and StreamMyGame [34].

3. DESIGN OBJECTIVES

GamingAnywhere aims to provide an open platform for researchers to develop and study real-time multimedia streaming applications in the cloud. The design objectives of GamingAnywhere include:

1. *Extensibility*: GamingAnywhere adopts a modularized design. Both platform-dependent components such as audio and video capturing and platform-independent components such as codecs and network protocols can be easily modified or replaced. Developers should be able to follow the programming interfaces of modules in GamingAnywhere to extend the capabilities of the system. It is not limited only to games, and any real-time multimedia streaming application such as live casting can be done using the same system architecture.
2. *Portability*: In addition to desktops, mobile devices are now becoming one of the most potential clients of cloud services as wireless networks are getting increasingly more popular. For this reason, we maintain the principle of portability when designing and implementing GamingAnywhere. Currently the server supports Windows and Linux, while the client supports Windows, Linux, and OS X. New platforms can be easily included by replacing platform-dependent components in GamingAnywhere. Besides the easily replaceable modules, the external components leveraged by GamingAnywhere are highly portable as well. This also makes GamingAnywhere easier to be ported to mobile devices. For these details please refer to Section 5.
3. *Configurability*: A system researcher may conduct experiments for real-time multimedia streaming applications with diverse system parameters. A large number of built-in audio and video codecs are supported by GamingAnywhere. In addition, GamingAnywhere exports all available configurations to users so that it is possible to try out the best combinations of parameters by simply editing a text-based configuration file and fitting the system into a customized usage scenario.
4. *Openness*: GamingAnywhere is publicly available at `http://gaminganywhere.org/`. Use of GamingAnywhere in academic research is free of charge but researchers and developers should follow the license terms claimed in the binary and source packages.

4. SYSTEM ARCHITECTURE

The deployment scenario of GamingAnywhere is shown in Figure 1. A user first logs into the system via a portal server, which provides a list of available games to the user. The user then selects a preferred game and requests to play the game. Upon receipt of the request, the portal server finds an available game server, launches the selected game on the server, and returns the game server's URL to the user. Finally, the user connects to the game server and starts to play. There is not too much to discuss for the portal server, which is just like most Web-based services and provides only a simple login and game selection user interface. If login and game selection requests are sent from a customized client, it does not even need a user interface. Actions can be sent as REST-like [10, 14] requests via standard HTTP or HTTPS

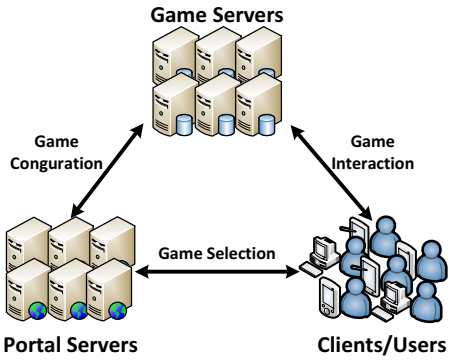

Figure 1: The deployment scenario of GamingAnywhere.

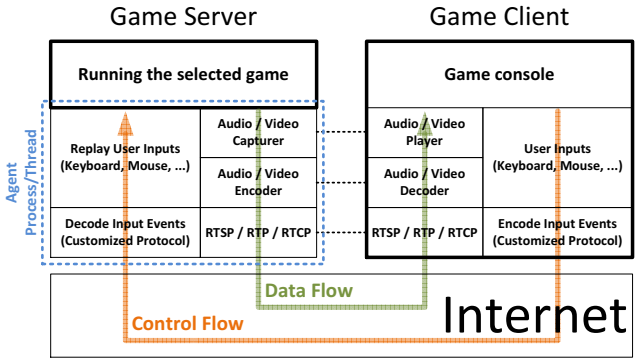

Figure 2: A modular view of GamingAnywhere server and client.

protocols. Therefore, in this section we only focus on the game server and the game client of GamingAnywhere.

Figure 2 shows the architecture of the game server and the game client of GamingAnywhere. We define two types of network flows in the architecture, the *data flow* and the *control flow*. Whereas the data flow is used to stream audio and video (A/V) frames from the server to the client, the control flow runs in a reverse direction, being used to send the user's actions from the client to the server. The system architecture of GamingAnywhere allows it to support any types of games, including PC-based and Web-based games. The game selected by a user runs on a game server. There is an agent running along with the selected game on the same server. The agent can be a stand-alone process or a thread injected into the selected game. The choice depends on the type of the game and how the game is implemented. The agent has two major tasks. The first task is to capture the A/V frames of the game, encode the frames using the chosen codecs, and then deliver the encoded frames to the client via the data flow. The second task of the agent is to interact with the game. On receipt of the user's actions from the client, it must behave as the user and play with the game by re-playing the received keyboard, mouse, joysticks, and even gesture events. However, as there exist no standard protocols for delivering users' actions, we chose to design and implement the transport protocol for user actions by ourselves.

The client is basically a customized game console implemented by combining an RTSP/RTP multimedia player and a keyboard/mouse logger. The system architecture of GamingAnywhere allows *observers*[1] by nature because the server delivers encoded A/V frames using the standard RTSP and RTP protocols. In this way, an observer can watch a game play by simply accessing the corresponding game URL with full-featured multimedia players, such as the VLC media player [39], which are available on almost all OS's and platforms.

5. IMPLEMENTATION

Presently, the implementation of GamingAnywhere includes the server and the client, each of which contains a number of modules whose details of each module are elaborated in this section. The implementation of GamingAnywhere depends on several external libraries including *libavcodec/libavformat* [13], *live555* [23], and *SDL library* [21]. The *libavcodec/libavformat* library is part of the *ffmpeg* project, which is a package to record, convert, and stream audio and video. We use this library to encode and decode the A/V frames on both the server and the client. In addition, it is also used to handle the RTP protocol at the server. The *live555* library is a set of C++ libraries for multimedia streaming using open standard protocols (RTSP, RTP, RTCP, and SIP). We use this library to handle RTSP/RTP protocols [31, 32] at the client. The *Simple DirectMedia Layer (SDL)* library is a cross-platform library designed to provide low-level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL and a 2D video frame buffer. We use this library to render audio and video at the client. All the above libraries have been ported to a number of platforms, including Windows, Linux, OS X, iOS, and Android.

5.1 GamingAnywhere Server

The relationships among server modules are shown in Figure 3. Some of the modules are implemented in separate threads. When an agent is launched, its four modules, i.e., the RTSP server, audio source, video source, and input replayer are launched as well. The RTSP server and the input replayer modules are immediately started to wait for incoming clients (starting from the path $1n$ and $1i$ in the figure). The audio source and the video source modules are kept idle after initialization. When a client is connected to the RTSP server, the encoder threads are launched and an encoder must notify the corresponding source module that it is ready to encode the captured frames. The source modules then start to capture audio and video frames when one or more encoders are ready to work. Encoded audio and video frames are generated concurrently in real time. The data flows of audio and video frame generations are depicted as the paths from $1a$ to $5a$ and from $1v$ to $5v$, respectively. The details of each module are explained respectively in the following subsections.

5.1.1 RTSP, RTP, and RTCP Server

The RTSP server thread is the first thread launched in the agent. It accepts RTSP commands from a client, launches encoders, and setups data flows for delivering en-

[1]In addition to playing a game themselves, hobbyists may also like to watch how other gamers play the same game. An observer can only watch how a game is played but cannot be involved in the game.

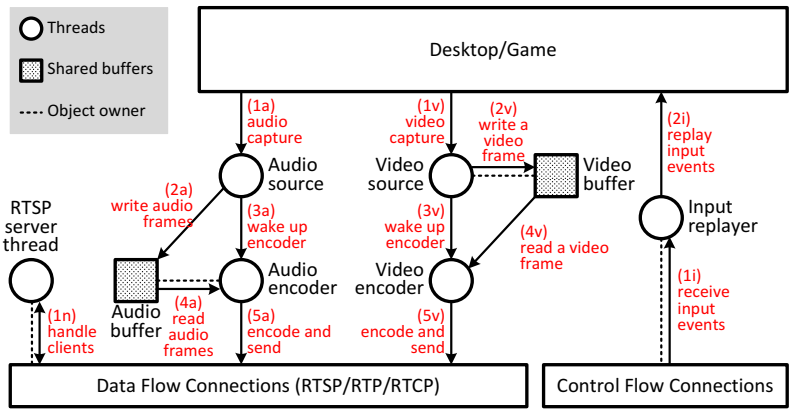

Figure 3: The relationships among server modules, shared buffers, and network connections.

coded frames. The data flows can be conveyed by a single network connection or multiple network connections depending on the preferred transport layer protocol, i.e., TCP or UDP. In the case of TCP, encoded frames are delivered as interleaved binary data in RTSP [32], hence necessitating only one data flow network connection. Both RTSP commands and RTP/RTCP packets are sent via the RTSP over TCP connection established with a client. In the case of UDP, encoded frames are delivered based on the RTP over UDP protocol. Three network flows are thus required to accomplish the same task: In addition to the RTSP over TCP connection, two RTP over UDP flows are used to deliver encoded audio and video frames, respectively.

We implement the mechanisms for handling RTSP commands and delivering interleaved binary data by ourselves, while using the *libavformat* library to do the packetization of RTP and RTCP packets. If encoded frames are delivered as interleaved binary data, a raw RTP/RTCP packet can be obtained by allocating a dynamic packet buffer and then be sent as interleaved binary data. On the other hand, if encoded frames are delivered via RTP over UDP, they are sent directly to the client using *libavformat*.

The RTSP server thread exports a programming interface for encoders to send encoded frames. When an encoder generates an encoded frame, it can send out the frame to the client via the interface without knowing the details about the underlying network connections.

5.1.2 Video Source

Capturing of game screens (frames) is platform-dependent. We currently provide two implementations of the video source module to capture the game screens in real time. One implementation is called the *desktop capture module*, which captures the entire desktop screen at a specified rate, and extracts the desired region when necessary. Another implementation is called the *API intercept module*, which intercepts a game's graphics drawing function calls and captures the screen directly from the game's back buffer [25] immediately whenever the rendering of a new game screen is completed.

Given a desired frame rate (commonly expressed in frame-per-second, fps), the two implementations of the video source module work in different ways. The desktop capture module is triggered in a polling manner; that is, it actively takes a screenshot of the desktop at a specified frequency. For example, if the desired frame rate is 24 fps, the capture interval will be 1/24 sec ($\approx$ 41.7 ms). By using a high-resolution timer, we can keep the rate of screen captures approximately equal to the desired frame rate. On the other hand, the API intercept module works in an *event-driven* manner. Whenever a game completes the rendering of an updated screen in the back buffer, the API intercept module will have an opportunity to capture the screen for streaming. Because this module captures screens in an opportunistic manner, we use a token bucket rate controller [35] to decide whether the module should capture a screen in order to achieve the desired streaming frame rate. For example, assuming a game updates its screen 100 times per second and the desired frame rate is 50 fps, the API intercept module will only capture one game screen for every two screen updates. In contrast, if the game's frame rate is lower than the desired rate, the module will re-use the last-captured game screens to meet the desired streaming frame rate.

Each captured frame is associated with a timestamp, which is a zero-based sequence number. Captured frames along with their timestamps are stored in a shared buffer owned by the video source module and shared with video encoders. The video source module serves as the only buffer writer, while the video encoders are all buffer readers. Therefore, a reader-writer lock must be acquired every time before accessing the shared buffer. Note that although only one video encoder is illustrated in Figure 3, it is possible to run multiple video encoders simultaneously depending on the usage scenario. We discuss this design choice between a single encoder and multiple encoders in Section 5.1.4.

At present, the desktop capture module is implemented in Linux and Windows. We use the MIT-SHM extension for the X Window system to capture the desktop on Linux and use GDI to capture the desktop graphics on Windows. As for the API intercept module, it currently supports DirectDraw and Direct3D games by hooking DirectX APIs on Windows. Both modules support captured frames of pixel formats in RGBA, BGRA, and YUV420P, with a high extensibility to incorporate other pixel formats for future needs.

5.1.3 Audio Source

Capturing of audio frames is platform-dependent as well. In our implementation, we use the ALSA library and Windows audio session API (WASAPI) to capture sound on Linux and Windows, respectively. The audio source module

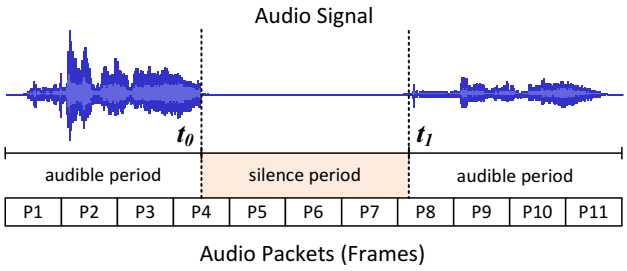

Figure 4: Sample audio signals that may cause the frame discontinuity problem.

regularly captures audio frames (also called audio packets) from an audio device (normally the default waveform output device). The captured frames are copied by the audio source module to a buffer shared with the encoder. The encoder will be awakened each time an audio frame is generated to encode the new frame. To simplify the programming interface of GamingAnywhere, we require each sample of audio frames to be stored as a 32-bit signed integer.

One issue that an audio source module must handle is the *frame discontinuity problem*. When there is no application generating any sound, the audio read function may return either 1) an audio frame with all zeros, or 2) an error code indicating that no frames are currently available. If the second case, an audio source module needs to still emit silence audio frames to the encoder because encoders normally expect continuous audio frames no matter whether audible sound is present or not. Therefore, an audio source module must emit silence audio frames in the second case to resolve the frame discontinuity problem. We observed that modern Windows games often play audio using WASAPI, which suffers from the frame discontinuity problem. Our WASAPI-based audio source module has overcome the problem by carefully estimating the duration of silence periods and generating silence frames accordingly, as illustrated in Figure 4. From the figure, the length of the silence frame should ideally be $t_1 - t_0$; however, the estimated silence duration may be slightly longer or shorter if the timer accuracy is not sufficiently high.

5.1.4 Frame Encoding

Audio and video frames are encoded by two different encoder modules, which are launched when there is at least one client connected to the game server. GamingAnywhere currently supports two encoding modes: 1) *one-encoder-for-all* and 2) *one-encoder-each-client* to support different usage scenarios. In the *one-encoder-for-all* mode, the frames generated by a frame source are only read and encoded by one encoder *regardless of the number of observers*[2]. Therefore, a total of two encoders, one for video frames and another for audio frames, are in charge of encoding tasks. The benefit of this mode is better efficiency as the CPU usage does not increase when there are more observers. All the video and audio frames are encoded only once and the encoded frames are delivered to the corresponding clients in a unicast manner.

On the other hand, the *one-encoder-each-client* mode allocates a dedicated encoder for each client, either a player or an observer. The benefit is that it is therefore possible to use different encoding configurations, such as bit rate, resolution, and quality parameters, for different clients. However, the consumed CPU resources would increase proportionally with the number of encoders. For example, in our study, each `x264` encoder with 1280x720 resolution and 24 fps increases the CPU utilization by nearly 10% on an Intel 2.66 GHz i5 CPU. In this way, a game server can only tolerate 10 observers at most when only one game instance is running. Therefore, the tradeoff between the *one-encoder-for-all* mode[3] and *one-encoder-each-client* mode needs to be seriously considered as it may have large performance impacts on the system.

Presently, both the video and audio encoder modules are implemented using the *libavcodec* library, which is part of the *ffmpeg* project. The *libavcodec* library supports various audio and video codecs and is completely written in the C language. Therefore, GamingAnywhere can use any codec supported by *libavcodec*. In addition, since the *libavcodec* library is highly extensible, researchers can easily integrate their own code into GamingAnywhere to evaluate its performance in cloud gaming.

5.1.5 Input Handling

The input handling module is implemented as a separate thread. This module has two major tasks: 1) to capture input events on the client, and 2) to replay the events occurring at the client on the game server.

Unlike audio and video frames, input events are delivered via a separated connection, which can be TCP or UDP. Although it is possible to reuse the RTSP connection for sending input events from the client to the server, we decided not to adopt this strategy for three reasons: 1) The delivery of input events may be delayed due to other messages, such as RTCP packets, sent via the same RTSP connection. 2) Data delivery via RTSP connections incurs slightly longer delays because RTSP is text-based and parsing text is relatively time-consuming. 3) There is no such standard of embedding input events in an RTSP connection. This means that we will need to modify the RTSP library and inevitably make the system more difficult to maintain.

The implementation of the input handling module is intrinsically platform-dependent because the input event structure is OS- and library-dependent. Currently GamingAnywhere supports the three input formats of Windows, X Window, and SDL. Upon the receipt of an input event[4], the input handling module first converts the received event into the format required by the server and sends the event structure to the server. GamingAnywhere replays input events using the SendInput function on Windows and the XTEST extension on Linux. While the above replay functions work quite well for most desktop and game applications, some games adopt different approaches for capturing user inputs. For example, the SendInput function on Windows does not work for Batman and Limbo, which are two popular action adventure games. In this case, GamingAnywhere can be configured to use other input replay methods, such as hook-

[2]In the current design, there can be one player and unlimited observers simultaneously in a game session.

[3]It is also possible to provide differential streaming quality for different clients in the *one-encoder-for-all* mode by adopting scalable video codecs such as H.264/SVC.

[4]The capturing of input events on clients will be elaborated in Section 5.2.3.

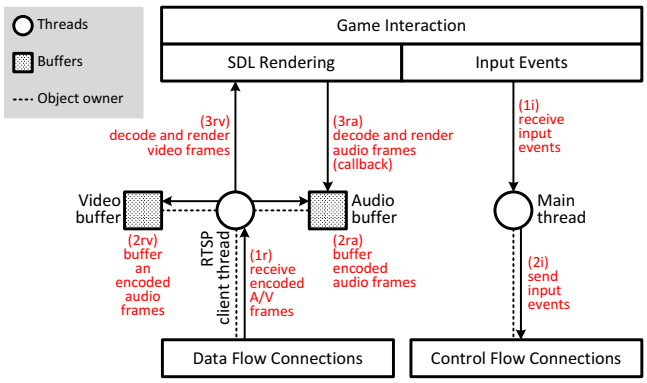

Figure 5: The relationships among client modules, shared buffers, and network connections.

ing the GetRawInputData function on Windows to "feed" input events whenever the function is called by the games.

5.2 GamingAnywhere Client

The client is basically a remote desktop client that displays real-time game screens which are captured at the server and delivered in the form of encoded audio and video frames. The relationships among client modules are shown in Figure 5. The GamingAnywhere client contains two worker threads: one is used to handle user inputs (starting from path $1i$) and the other is used to render audio and video frames (starting from path $1r$). In this section, we divide the discussion on the client design into three parts, i.e., the network protocols, the decoders, and input handling.

5.2.1 RTSP, RTP, and RTCP Clients

In the GamingAnywhere client, we use the *live555* library to handle the network communications. The *live555* library is entirely written in C++ with an event-driven design. We take advantage of the class framework of *live555* and derive from the `RTSPClient` and `MediaSink` classes to register callback functions that handle network events. Once the RTSP client has successfully set up audio and video sessions, we create two sink classes to respectively handle the encoded audio and video frames that are received from the server. Both sink classes are inherited from the `MediaSink` class and the implemented `continuePlaying` virtual function is called when the RTSP client issues the PLAY command. The `continuePlaying` function attempts to receive an encoded frame from the server. When a frame is received successfully, the function triggers a callback function that puts the frame in a buffer and decodes the video frame if possible. The `continuePlaying` function will then be called again to receive the next frame.

5.2.2 Frame Buffering and Decoding

To provide better gaming experience in terms of latency, the video decoder currently *does not buffer video frames* at all. In other words, the video buffer component in Figure 5 is simply used to buffer packets that are associated with the latest video frame. Because *live555* provides us with packet payloads without an RTP header, we detect whether consecutive packets correspond to the same video frame based on the marker bit [31] in each packet. That is, if a newly received packet has a zero marker bit (indicating that it is *not* the last packet associative with a video frame), it will be appended into the buffer; otherwise, the decoder will decode a video frame based on all the packets currently in the buffer, empty the buffer, and place the newly arrived packet in the buffer. Although this zero-buffering strategy may lead to inconsistency in video playback rate when network delays are unstable [45], it reduces the input-response latency due to video playout to a minimum level. We believe that this design tradeoff can yield a overall better cloud gaming experience.

The way GamingAnywhere handles audio frames is different from its handling of video frames. Upon the receipt of audio frames, the RTSP client thread does not decode the frames, but instead simply places all the received frames in a shared buffer (implemented as a FIFO queue). This is because the audio rendering of *SDL* is implemented using an on-demand approach. That is, to play audio in *SDL*, a callback function needs to be registered and it is called whenever *SDL* requires audio frames for playback. The memory address m to fill audio frames and the number of required audio frames n are passed as arguments to the callback function. The callback function retrieves the audio packets from the shared buffer, decodes the packets, and fills the decoded audio frames into the designated memory address m. Note that the callback function must fill exactly n audio frames into the specified memory address as requested. This should not be a problem if the number of decoded frames is more than requested. If not, the function must wait until there are sufficient frames. We implement the waiting mechanism for sufficient frames using a mutual exclusive lock (mutex). If the RTSP client thread has received new audio frames, it will append the frames to the buffer and also trigger the callback function to read more frames.

5.2.3 Input Handling

The input handling module on the client has two major tasks. One is to capture input events made by game players, and the other is to send captured events to the server. When an input event is captured, the event structure is sent to the server directly. Nevertheless, the client still has to tell the server the format and the length of a captured input event.

At present, GamingAnywhere supports the mechanism for cross-platform SDL event capturing. In addition, on certain platforms, such as Windows, we provide more sophisticated input capture mechanisms to cover games with special input mechanisms and devices. Specifically, we use the SetWindowsHookEx function with `WH_KEYBOARD_LL` and `WH_MOUSE_LL` hooks to intercept low-level keyboard and mouse events. By so doing we can perfectly mimic every move of the players' inputs on the game server.

6. PERFORMANCE EVALUATION

In this section, we evaluate GamingAnywhere via extensive experiments, and compare its performance against two well-known cloud gaming systems.

6.1 Setup

We have set up a GamingAnywhere testbed in our lab. We conduct the experiments using Windows 7 desktops with Intel 2.67 GHz i7 processors if not otherwise specified. For evaluation purposes, we compare the performance of GamingAnywhere against OnLive [29] and StreamMyGame

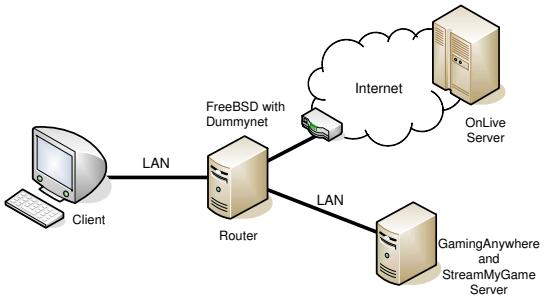

Figure 6: The network topology of our experiments.

(SMG) [34]. Figure 6 illustrates the experimental setup, which consists of a server, a client, and a router. The OnLive server resides in OnLive's data centers, while the GamingAnywhere and SMG servers are installed on our own PCs. More specifically, the OnLive client connects to the OnLive server over the Internet, while the GamingAnywhere and SMG clients connect to their servers via a LAN. To evaluate the performance of the cloud gaming systems under diverse network conditions, we add a FreeBSD router between the client and server, and run `dummynet` on it to inject constraints of delays, packet losses, and network bandwidths.

Because the OnLive server is outside our LAN, the quality of the network path between our OnLive client and the server might affect our evaluations. However, according to our observations, the quality of the path was consistently good throughout the experiments. The network delay of the path was around 130 ms with few fluctuations. Furthermore, the packet loss rates were measured to be less than 10^{-6} when receiving OnLive streams at the recommended 5 Mbps. Therefore, the path between the OnLive server and our client can be considered as a communication channel with sufficient bandwidth, zero packet loss rate, and a constant 130 ms latency.

Since the performance of cloud gaming systems may be game-dependent, we consider games from three popular categories: action adventure, first-person shooter, and real-time strategy. We pick a representative game from each category, and briefly introduce them in the following.

- *LEGO Batman: The Videogame* (Batman) [2] is an action-adventure game, created by Traveller's Tales in 2008. All the interactive objects in this game are made of Lego bricks. In this game, players control the characters to fight enemies and solve puzzles from a third-person perspective.
- *F.E.A.R. 2: Project Origin* (FEAR) [1] is a first-person shooter game, developed by Monolith Productions in 2009. The combat scenes are designed to be as close to those in real life as possible. In this game, players have great freedom to interact with the environments, e.g., they can flip over a desk to take cover.
- *Warhammer 40,000: Dawn of War II* (DOW) [3] is a real-time strategy game developed by Relic Entertainment in 2009. In the campaign mode, players control squads to fight against enemies and destroy the buildings. In the multiplayer mode, up to 8 players play matches on the same map to complete a mission, such as holding specific positions.

Modern video encoders strive to achieve the highest video quality with the smallest bit rate by applying complex coding techniques. However, overly-complex coding techniques are not feasible for real-time videos given their lengthy encoding time. As such, we empirically study the tradeoff among the bit rate, video quality, and frame complexity using `x264`. More specifically, we apply the real-time encoding parameters summarized in Appendix A, and exercise a wide spectrum of other encoding parameters. We then analyze the resulting video quality and encoding time. Based on our analysis, we recommend the following `x264` encoding parameters:

```
--profile main --preset faster --tune zerolatency
--bitrate $r --ref 1 --me dia --merange 16
--intra-refresh --keyint 48 --sliced-threads
--slices 4 --threads 4 --input-res 1280x720,
```

where `$r` is the encoding rate.

We configure the GamingAnywhere server to use the above-mentioned encoding parameters, and we set the encoding bit rate to be 3 Mbps. For a fair comparison, all games are streamed at a resolution of 720p. Whereas we configure GamingAnywhere and OnLive to stream at 50 fps, StreamMyGame only supports streaming at 25 fps. We design the experiments to evaluate the three gaming systems from two critical aspects: *responsiveness* and *video quality*. We also conduct experiments to quantify the network loads incurred by different cloud gaming systems. The details of the experimental designs and results are given in the rest of this section.

6.2 Responsiveness

We define response delay (RD) to be the time difference between a user submitting a command and the corresponding in-game action appearing on the screen. Studies [8, 17, 46] report that players of various game categories can tolerate different degrees of RD; for example, it was observed that first-person shooter game players demand for less than 100 ms RD [8]. However, since measuring RD in cloud gaming systems is not an easy task (as discussed in Section 2.2), we adopt the RD measurement procedure proposed in [5], in which the RD is divided into three components:

- *Processing delay (PD)* is the time required for the server to receive and process a player's command, and to encode and transmit the corresponding frame to that client.
- *Playout delay (OD)* is the time required for the client to receive, decode, and render a frame on the display.
- *Network delay (ND)* is the time required for a round of data exchange between the server and client. ND is also known as round-trip time (RTT).

Therefore, we have $RD = PD + OD + ND$.

ND can be measured using probing packets, e.g., in `ICMP` protocol, and is not controllable by cloud gaming systems. Moreover, ND in a LAN is much smaller than that in the Internet. Therefore, for a fair comparison among the cloud gaming systems, we exclude ND from RD measurements in the rest of this paper. Measuring PD (at the server) and OD (at the client) is much more challenging, because they occur internally in the cloud gaming systems, which may be closed and proprietary. The procedure detailed in [5] measures the

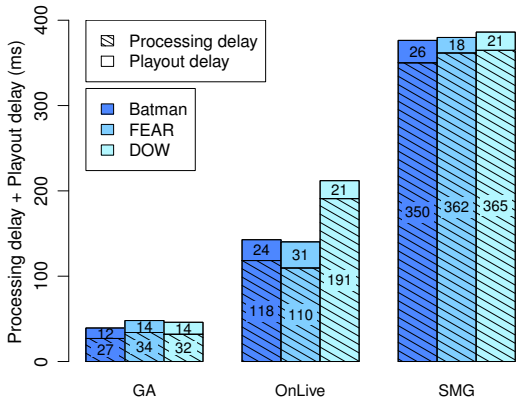

Figure 7: Response delays of GamingAnywhere, OnLive, and StreamMyGame.

PD and OD using external probes only, and thus works for all the considered cloud gaming systems.

For GamingAnywhere, we further divide the PD and OD into subcomponents by instrumenting the server and client. More specifically, PD is divided into: (i) *memory copy*, which is the time for copying a raw image out of the games, (ii) *format conversion*, which is the time for color-space conversion, (iii) *video encoding*, which is the time for video compression, and (iv) *packetization*, which is the time for segmenting each frame into one or multiple packets. OD is divided into: (i) *frame buffering*, which is the time for receiving all the packets belonging to the current frame (ii) *video decoding*, which is the time for video decompression, and (iii) *screen rendering*, which is the time for displaying the decoded frame.

Results. Figure 7 reports the average PD (server) and OD (client) achieved by the considered cloud gaming systems. From this figure, we make several observations. First, the OD is small, ≤ 31 ms, for all cloud gaming systems and considered games. This reveals that all the decoders are efficient, and the decoding time of different games does not fluctuate too much. Second, GamingAnywhere achieves a much smaller PD, at most 34 ms, than OnLive and SMG, which are observed to be as high as 191 and 365 ms, respectively. This demonstrates the efficiency of the proposed GamingAnywhere: the PDs of OnLive and SMG are 3+ and 10+ times longer than that of GamingAnywhere. Last, among the three systems, only GamingAnywhere achieves sub-100 ms RD, and may satisfy the stringent delay requirements of networked games [8].

Figure 8 presents the decomposed delay subcomponents of PD and OD. This figure reveals that the GamingAnywhere server and client are well-tuned, in the sense that all the steps in the pipeline are fairly efficient. Even for the most time-consuming video encoding (at the server) and video rendering (at the client), each frame is finished in at most 16 and 7 ms on average. Such a low delay contributes to the superior RD of GamingAnywhere, compared to the other well-known cloud gaming systems.

6.3 Network Loads

We next quantify the network loads incurred by GamingAnywhere. In particular, we recruit an experienced gamer, and ask him to play each game using different cloud gaming systems. Every game session lasts for 10 minutes, and the

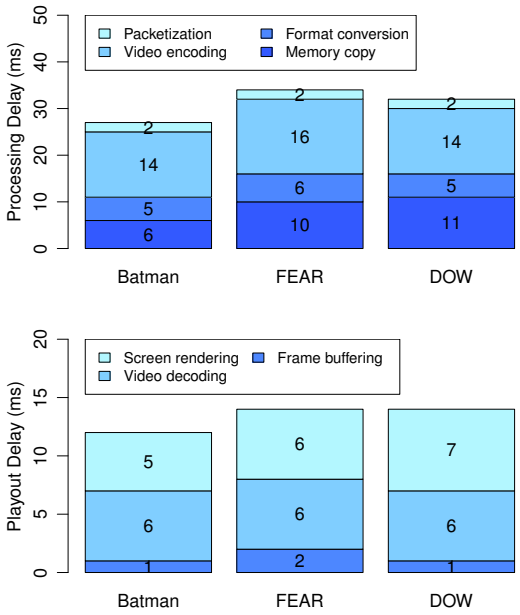

Figure 8: Delay decomposition of GamingAnywhere.

network packets are captured by `Wireshark`. For a fair comparison, the player is asked to follow two guidelines. First, he shall visit as many areas as possible and fight the opponents as in normal game plays. Second, he shall repeat his actions and trajectories as much as possible.

Results. Figure 9 plots the uplink and downlink traffic characteristics, including bit rate, packet rate, and payload length. The bar charts show the average values with 95% confidence intervals. Figures 9(a)–9(c) reveal that the proposed GamingAnywhere incurs a much lower uplink traffic loads, compared to OnLive and SMG. The only exception is that, with Batman, SMG incurs lower uplink packet rate (Figure 9(b)). However, SMG also produces a larger uplink payload size (Figure 9(c)), which leads to a higher uplink bit rate than that of GamingAnywhere (Fig 9(a)). Figures 9(d)–9(f) reveal that the downlink bit rates of OnLive are between 3–5 Mbps, while those of SMG are between 10–13 Mbps. This finding indicates that the compression algorithm employed by OnLive achieves up to a 4.33 times higher compression rate, compared to that of SMG.

We can make another observation on Figure 9(d): GamingAnywhere incurs a download bit rate ≤ 3 Mbps, which is also much lower than that of SMG. However, given that we set the encoding bit rate at 3 Mbps, the download bit rate should *never* be smaller than that. We took a closer look and found that, with GamingAnywhere, only Batman achieves 50 fps; FEAR and DOW only achieve 35–42 fps, which leads to lower download bit rate and may result in irregular playouts. Our in-depth analysis shows that, unlike Batman, both FEAR and DOW use *multisampling* surfaces, which cannot be locked for memory copy operations. More specifically, an additional *non-multisampling* surface and an extra copy operation are required for FEAR and DOW, which in turn hurts the achieved frame rates. As one of our future tasks, we will optimize the multi-threaded design of the GamingAnywhere server, so as to minimize the synchronization overhead.

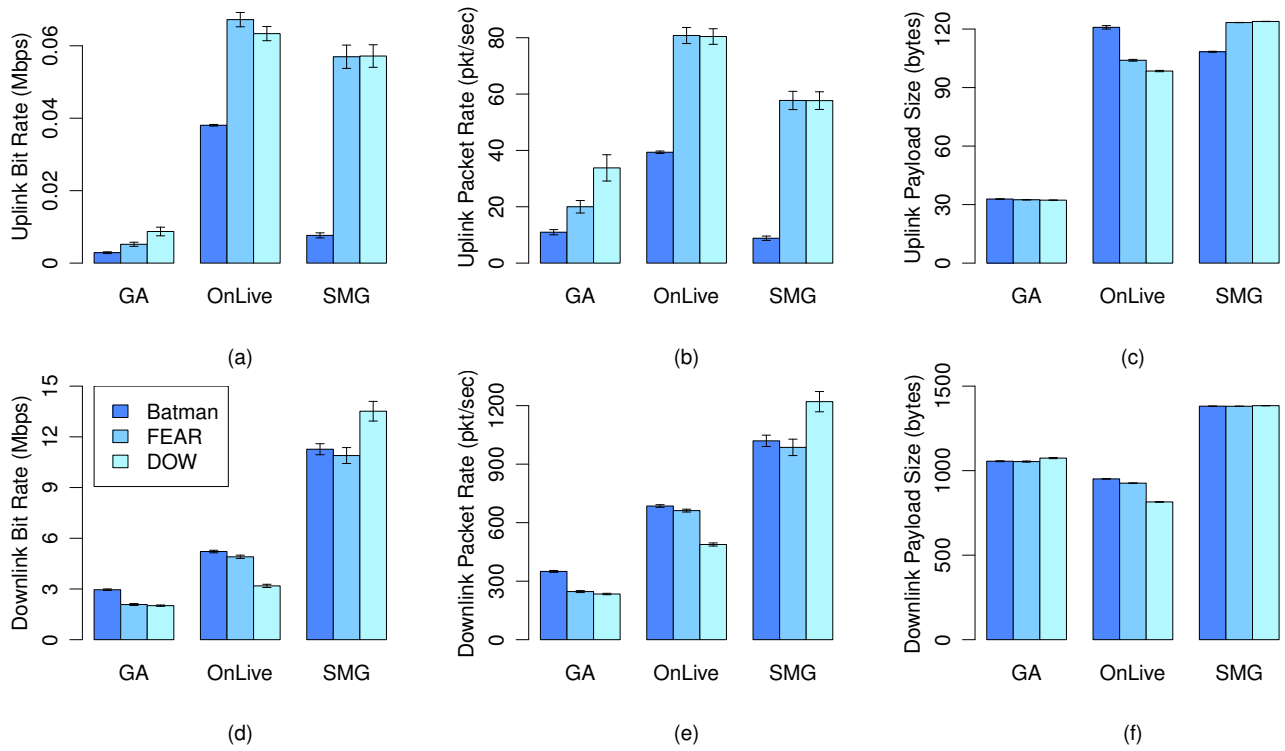

Figure 9: Network loads incurred by the considered cloud gaming systems.

In summary, we have shown that GamingAnywhere incurs much lower network traffic loads. Even though the current GamingAnywhere implementation only achieves 35–42 fps for games using multisampling surfaces, such a frame rate is still much higher than the 25 fps of SMG. On the other hand the slightly lower achieved frame rate may affect the fairness of video quality comparisons between GamingAnywhere and OnLive. Therefore, in the rest of this section, we only report results from Batman.

6.4 Video Quality

Video streaming quality directly affects gaming experience, and network conditions are the keys for high-quality streaming. In this light, we use `dummynet` to control three network condition metrics: network delay (ND), packet loss rate, and network bandwidth. We vary ND between 0–600 ms, packet loss rate between 0–10%, and bandwidth 1–6 Mbps in our experiments. We also include experiments with *unlimited* bandwidth. For OnLive, the ND in the Internet is already 130 ms and thus we cannot report the results from zero ND. Two video quality metrics, PSNR [40, p. 29] and Structural Similarity (SSIM) [41], are adopted. We report the average PSNR and SSIM values of the Y-component.

Results. Figures 10 and 11 present the PSNR and SSIM values, respectively. We make four observations on these two figures. First, ND does not affect the video quality too much (Figures 10(a) and 11(a)). Second, GamingAnywhere achieves much higher video quality than OnLive and SMG: up to 3 dB and 0.03, and 19 dB and 0.15 gaps are observed, respectively. Third, GamingAnywhere suffers from quality drops when packet loss rate is nontrivial, as illustrated in Figures 10(b) and 11(b). This can be attributed to the missing error resilience mechanism in GamingAnywhere. Nevertheless, high packet loss rates are less common in modern networks. Last, Figures 10(c) and 11(c) show that the video quality of GamingAnywhere suddenly drops when the bandwidth is smaller than the encoding bit rate of 3 Mbps. A potential future work to address this is to add a *rate adaptation* heuristic to dynamically adjust the encoding bit rate, in order to utilize all the available bandwidth without overloading the networks.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we presented GamingAnywhere, which is the first open cloud gaming system designed to be open, extensible, portable, and fully configurable. Through extensive experiments, we have shown that GamingAnywhere significantly outperforms two well-known, commercial, cloud gaming systems: OnLive and StreamMyGame. Compared to GamingAnywhere, for example, OnLive and StreamMyGame suffer from up to 3 and 10 times higher processing delays, as well as 3 dB and 19 dB lower video quality, respectively. GamingAnywhere is also efficient: it incurs lower network loads in both uplink and downlink directions. Given that GamingAnywhere is open, cloud game developers, cloud service providers, system researchers, and individual users may use it to set up a complete cloud gaming testbed. GamingAnywhere in publicly available at `http://gaminganywhere.org`. We hope that the release of GamingAnywhere will stimulate more research innovations on cloud gaming systems, or multimedia streaming applications in general.

We are actively enhancing GamingAnywhere in several directions. First, we strive to further reduce the delay at the GamingAnywhere server by minimizing the synchronization

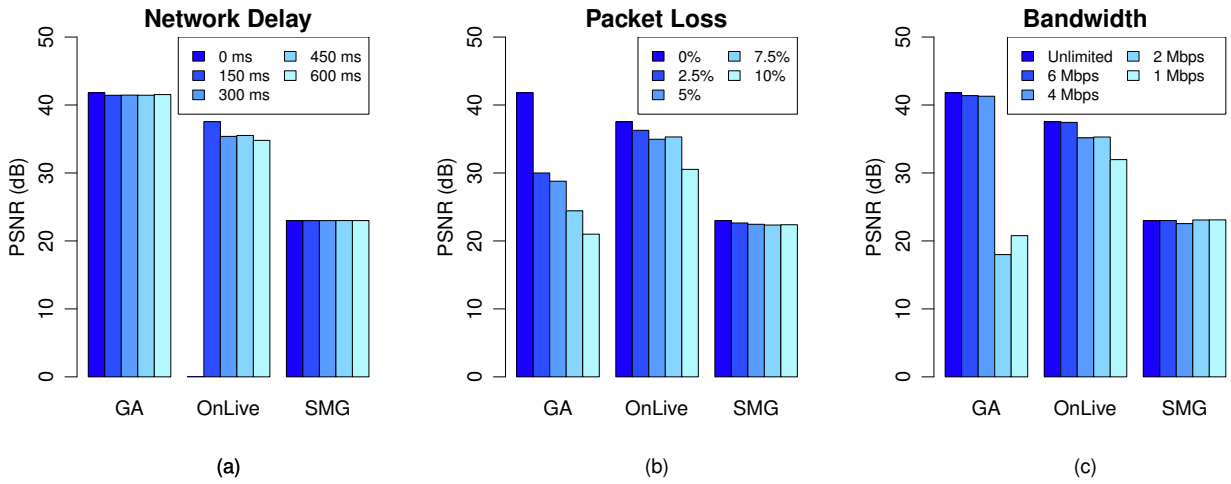

Figure 10: Achieved video quality in PSNR under different network conditions.

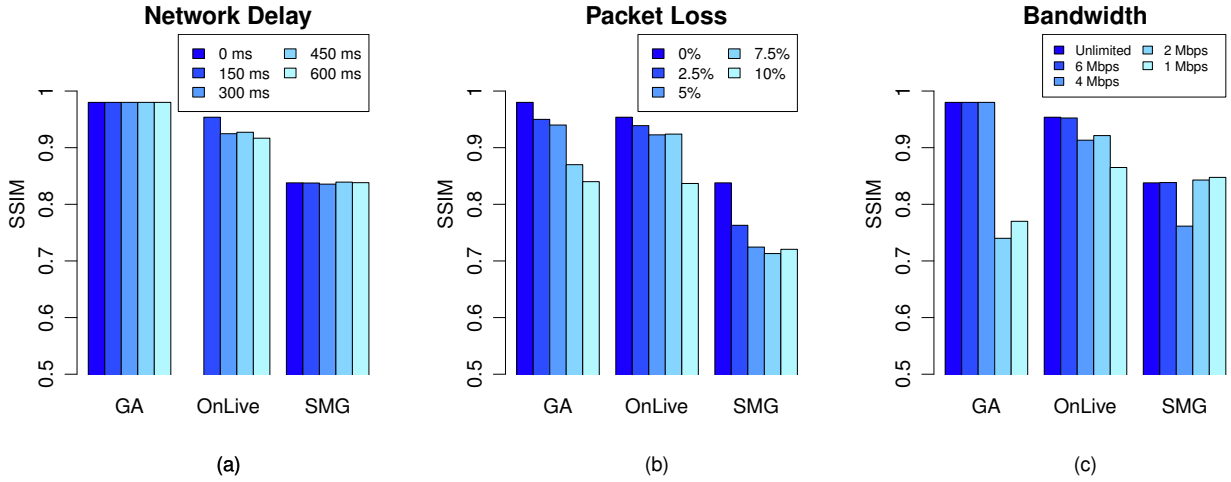

Figure 11: Achieved video quality in SSIM under different network conditions.

overhead. This will allow us to increase the achieved frame rate. Second, we are designing a practical rate control algorithm for GamingAnywhere, which may not be very useful in resourceful LANs, but is critical for remote players. Third, we are considering adding error resilience mechanisms to GamingAnywhere, in order to cope with packet loss due to, e.g., network congestion, hardware failures, and misconfigured routers.

References

[1] F.E.A.R. 2: Project Origin, November 2012. `http://www.whatisfear.com/`.

[2] LEGO Batman: The Videogame, November 2012. `http://games.kidswb.com/official-site/lego-batman/`.

[3] Warhammer 40,000: Dawn of War II, November 2012. `http://www.dawnofwar2.com/`.

[4] Y.-C. Chang, P.-H. Tseng, K.-T. Chen, and C.-L. Lei. Understanding the performance of thin-client gaming. In *Proceedings of IEEE CQR 2011*, May 2011.

[5] 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 ACM Multimedia 2011*, Nov 2011.

[6] Y. Chen, C. Chang, and W. Ma. Asynchronous rendering. In *Proc. of ACM SIGGRAPH symposium on Interactive 3D Graphics and Games (I3D'10)*, Washington, DC, February 2010.

[7] S. Choy, B. Wong, G. Simon, and C. Rosenberg. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Proceedings of IEEE/ACM NetGames 2012*, Oct 2012.

[8] M. Claypool and K. Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, November 2006.

[9] Cloud gaming adoption is accelerating ... and fast!, July 2012. `http://www.nttcom.tv/2012/07/09/cloud-gaming-adoption-is-acceleratingand-fast/`.

[10] R. L. Costello. Building web services the rest way. xFront - Tutorial and Articles on XML and Web Technologies, 2007. `http://www.xfront.com/REST-Web-Services.html`.

[11] Distribution and monetization strategies to increase revenues from cloud gaming, July 2012. `http://www.cgconfusa.com/report/documents/Content-5minCloudGamingReportHighlights.pdf`.

[12] P. Eisert and P. Fechteler. Low delay streaming of computer graphics. In *Proc. IEEE ICIP 2008*, October 2008.

[13] FFmpeg project. ffmpeg. `http://ffmpeg.org/`.

[14] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, Univer-

sity of California, Irvine, 2000.
[15] Gaikai web page, July 2012. http://www.gaikai.com/.
[16] F. Giesen, R. Schnabel, and R. Klein. Augmented compression for server-side rendering. In *Proc. of International Fall Workshop on Vision, Modeling, and Visualization (VMV'08)*, October 2008.
[17] T. Henderson. *The Effects of Relative Delay in Networked Games*. PhD thesis, Department of Computer Science, University of London, February 2003.
[18] O. Holthe, O. Mogstad, and L. Ronningen. Geelix LiveGames: Remote playing of video games. In *Proc. of IEEE Consumer Communications and Networking Conference (CCNC'09)*, Las Vegas, NV, January 2009.
[19] A. Jurgelionis, P. Fechteler, P. Eisert, F. Bellotti, H. David, J. Laulajainen, R. Carmichael, V. Poulopoulos, A. Laikari, P. Perala, A. Gloria, and C. Bouras. Platform for distributed 3D gaming. *International Journal of Computer Games Technology*, 2009:1:1–1:15, January 2009.
[20] A. Lai and J. Nieh. On the performance of wide-area thin-client computing. *ACM Transactions on Computer Systems*, 24(2):175–209, May 2006.
[21] S. Lantinga. Simple DirectMedia Layer. http://www.libsdl.org/.
[22] Y.-T. Lee, K.-T. Chen, H.-I. Su, and C.-L. Lei. Are all games equally cloud-gaming-friendly? an electromyographic approach. In *Proceedings of IEEE/ACM NetGames 2012*, Oct 2012.
[23] I. Live Networks. LIVE555 streaming media. http://live555.com/liveMedia/.
[24] LogMeIn web page, July 2012. https://secure.logmein.com/.
[25] Microsoft. Flipping surfaces (Direct3D 9). Windows Dev Center - Desktop, September 2012. http://msdn.microsoft.com/en-us/library/windows/desktop/bb173393%28v=vs.85%29.aspx.
[26] J. Nieh, S. Yang, and N. Novik. Measuring thin-client performance using slow-motion benchmarking. *ACM Transactions on Computer Systems*, 21(1):87–115, February 2003.
[27] Online sales expected to pass retail software sales in 2013, September 2011. http://www.dfcint.com/wp/?p=311.
[28] OnLive crushed by high infrastructure bills, August 2012. http://www.computerworld.com/s/article/9230376/OnLive_crushed_by_high_infrastructure_bills.
[29] Onlive web page, July 2012. http://www.onlive.com/.
[30] K. Packard and J. Gettys. X window system network performance. In *Proc. of USENIX Annual Technical Conference (ATC'03)*, pages 206–218, San Antonio, TX, June 2003.
[31] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. Rtp: A transport protocol for real-time applications. RFC 3550 (Standard), July 2003. http://www.ietf.org/rfc/rfc3550.txt.
[32] H. Schulzrinne, A. Rao, and R. Lanphier. Real time streaming protocol (rtsp). RFC 2326 (Proposed Standard), April 1998. http://www.ietf.org/rfc/rfc2326.txt.
[33] S. Shi, C. Hsu, K. Nahrstedt, and R. Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proc. of ACM Multimedia'11*, pages 103–112, November 2011.
[34] Streammygame web page, July 2012. http://streammygame.com/.
[35] A. S. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.
[36] TeamViewer web page, July 2012. http://www.teamviewer.com.
[37] N. Tolia, D. Andersen, and M. Satyanarayanan. Quantifying interactive user experience on thin clients. *IEEE Computer*, 39(3):46–52, March 2006.
[38] UltraVNC web page, July 2012. http://www.uvnc.com/.
[39] VideoLAN. VLC media player. Official page for VLC media player, the Open Source video framework! http://www.videolan.org/vlc/.
[40] Y. Wang, J. Ostermann, and Y. Zhang. *Video Processing and Communications*. Prentice Hall, 2001.
[41] Z. Wang, L. Lu, and A. Bovik. Video quality assessment based on structural distortion measurement. *Signal Processing: Image Communication*, 19(2):121–132, February 2004.
[42] D. Winter, P. Simoens, L. Deboosere, F. Turck, J. Moreau, B. Dhoedt, and P. Demeester. A hybrid thin-client protocol for multimedia streaming and interactive gaming applications. In *Proc. of ACM NOSSDAV 2006*, Newport, RI, May 2006.
[43] A. Wong and M. Seltzer. Evaluating Windows NT terminal server performance. In *Proc. of USENIX Windows NT Symposium (WINSYM'99)*, pages 145–154, Seattle, WA, July 1999.
[44] x264 web page, July 2012. http://www.videolan.org/developers/x264.html.
[45] Y. Xu, C. Yu, J. Li, and Y. Liu. Video telephony for end-consumers: Measurement study of Google+, iChat, and Skype. In *Proceedings of Internet Measurement Conference (IMC 2012)*, Nov 2012.
[46] S. Zander, I. Leeder, and G. Armitage. Achieving fairness in multiplayer network games through automated latency balancing. In *Proc. of ACM SIGCHI ACE 2005*, pages 117–124, Valencia, Spain, June 2005.

APPENDIX

A. REAL-TIME ENCODING PARAMETERS FOR X264

Real-time video encoding requires some coding tools, but prohibits some others. We briefly present the most critical real-time encoding parameters dictated by `x264` in the following. First, we do not have the luxury to use bi-directional (B) frames, which lead to dependency on future frames, and may result in additional latency. We also need to turn off the `x264` lookahead buffers, which are used to determine the frame type (I, P, or B) and facilitate frame-level multi-threading. These can be done by a convenient flag `--tune zerolatency` of `x264`.

Second, because frame-level multi-threading inherently results in additional latency, we need to enable *slice-level* multi-threading, in order to leverage the computation power provided by multi-core CPUs. Slices are essentially disjoint regions extracted from each video frame. Slice-level multi-threading cuts each frame into multiple slices, and allocates a thread to encode each slice. Regarding `x264`, it supports: (i) `--sliced-threads` to enable slice-level multi-threading, (ii) `--slices` to specify the number of slices, and (iii) `--threads` to control the number of threads.

Third, like many video streaming systems, GamingAnywhere is sensitive to packet losses. Packet losses not only result in video quality degradation in the current frames, but the imperfect reconstruction also leads to error propagation. One way to limit error propagation is to choose a small group-of-picture (GoP) size. This approach has two drawbacks: (i) rate fluctuations caused by the larger size of I frames and (ii) noticeable artifacts when losing an I frame. A better solution is to employ *intra refresh*, which distributes intra-coded macroblocks over multiple frames in a GoP. More specifically, each frame consists of a column of intra-coded macroblocks, and this intra-coded column moves along the time. Intra refresh is allowed by `x264` via the flag `--intra-refresh`, and the GoP size is set by `--keyint`.