GamingAnywhere—The First Open Source Cloud Gaming System

CHUN-YING HUANG, National Taiwan Ocean University, Taiwan
KUAN-TA CHEN, Academia Sinica, Taiwan
DE-YU CHEN, Academia Sinica, Taiwan
HWAI-JUNG HSU, Academia Sinica, Taiwan
CHENG-HSIN HSU, National Tsing Hua University, Taiwan

We present the first open source cloud gaming system, called GamingAnywhere. In addition to its openness, we design GamingAnywhere for high extensibility, portability, and reconfigurability. We implement GamingAnywhere on Windows, Linux, OS X, and Android. We conduct extensive experiments to evaluate the performance of GamingAnywhere. Our experimental results indicate that GamingAnywhere is efficient, scalable, adaptable to network conditions, and achieves high responsiveness and streaming quality. 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 and applications.

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

General Terms: Design, Measurement

Additional Key Words and Phrases: Cloud games, remote rendering, live video streaming, real-time encoding, performance evaluation, performance optimization

ACM Reference Format:

C.-Y. Huang, C. 2013.GamingAnywhere—The First Open Source Cloud Gaming System *ACM Trans. Multimedia Comput. Commun. Appl.* 2, 3, Article 1 (May 2010), 20 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Cloud gaming systems render the game scenes on cloud servers and stream the encoded game scenes to clients over the Internet. The clients send the user inputs, from joysticks, keyboards, and mice, to the cloud servers. With cloud gaming systems, users can: (i) avoid upgrading their computers, for the latest games, (ii) play the same games using the same clients on different platforms, such as PCs,

This work was supported in part by the National Science Council of Taiwan under the grants NSC100-2628-E-001-002-MY3, NSC102-2219-E-019-001, and NSC102-2221-E-007-062-MY3.
Author's address: C.-Y. Huang, 2 Pei-Ning Road Keelung, Taiwan 20224; email: chuang@ntou.edu.tw; K.-T. Chen, D.-Y. Chen, H.-J. Hsu, 128 Academia Road, Section 2, Nankang, Taipei 11574; email: swc@iis.sinica.edu.tw, r96922083@ntu.edu.tw, hjhsu@iis.sinica.edu.tw; C.-H. Hsu, No. 101, Section 2, Kuang-Fu Road, Hsinchu, Taiwan 30013; email: chsu@cs.nthu.edu.tw
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2010 ACM 1551-6857/2010/05-ART1 $15.00
DOI:http://dx.doi.org/10.1145/0000000.0000000

laptops, tablets, and smartphones, and (iii) play more games due to reduced hardware/software cost, while game developers may: (i) support more platforms, (ii) avoid hardware/software incompatibility issues, and (iii) increase net revenues. Therefore, cloud gaming systems have attracted attentions from users, game developers, and service providers. In fact, the market potential of cloud gaming is well recognized as evidenced by the recent acquisitions of cloud gaming startups, such as GaiKai [Sony 2012] and ESN [EA 2012].

For commercially-successful cloud gaming services, the cloud gaming systems must deliver high-quality videos with low response delay, which is difficult in the best-effort Internet. The response delay refers to the time difference between the client receiving a user input and the client displaying the game frame reflecting that user input. Higher quality videos, such as 720p (1280x720) at 60 fps (frame-per-second), inherently lead to higher bit rate, which render cloud gaming systems more vulnerable to network congestion, and thus longer response delay. Longer response delay results in worse user experiences, and may turn the users away from cloud gaming services. In fact, user studies reveal that users may *quit* playing networked games if the response delay is longer than a genre-dependent threshold, as low as 100 ms [Claypool and Claypool 2006; Chen et al. 2009]. Considering that game scenes have to go through a *pipeline of rendering, capturing, encoding, packetization, transmitting, decoding, and displaying*, it is very challenging to design and implement a cloud gaming system for both high video quality and low response delay.

Remote desktop software packages, such as LogMeIn [LogMeIn 2012], TeamViewer [TeamViewer 2012], and UltraVNC [UltraVNC 2012], have been popular for some time, but were not designed for highly interactive applications, and thus do not meet the strict requirements of cloud gaming [Chang et al. 2011]. Although there exist commercial cloud gaming services, e.g., GaiKai [GaiKai 2012], OnLive [OnLive 2012], and StreamMyGame [StreamMyGame 2012], a recent measurement study [Chen et al. 2011] reports that these cloud gaming systems still suffer from high response delay, among other limitations. For example, assuming a small, negligible network latency, 134 ms and 375 ms average response delay are measured on the OnLive and StreamMyGame platforms respectively. Hence, the problem of developing cloud gaming systems for high video quality and low response delay remains open. We consider a major cause of the inferior performance of existing cloud gaming systems to be the lack of an *open source* cloud gaming system, which will enable research groups to readily implement and evaluate their new ideas for better cloud gaming experiences.

In this article, we present our efforts on designing, implementing, and evaluating *GamingAnywhere*, which is to the best of our knowledge, the first open source cloud gaming system. GamingAnywhere has three main advantages over other existing systems.

(1) 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 uses `x264` [x264 2012] and `vpxenc` [WebM 2013] to encode captured raw videos. To expand GamingAnywhere for stereoscopic games, an H.264/MVC encoder may be plugged into it without significant changes. More generally, various algorithms, standards, protocols, and system parameters can be rigorously evaluated using real experiments, which is impossible on proprietary cloud gaming systems.
(2) GamingAnywhere is cross-platform, and is currently available on Windows, Linux, OS X, and Android. This is made possible largely due to the modularized design of GamingAnywhere.
(3) 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 ≥ 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 outperforms OnLive by 5 dB in Peak Signal-to-Noise Ratio (PSNR).

This article makes two main contributions. First, we develop 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. GamingAnywhere is the first open cloud gaming testbed in the literature. Second, 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.

1.1 Design Objectives

GamingAnywhere aims to provide an open platform for researchers to develop and study real-time multimedia streaming applications in the cloud. Its objectives are as follows.

(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 can follow the programming interfaces of modules in GamingAnywhere to extend the capabilities of the system. GamingAnywhere 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 desktop computers, mobile devices are now becoming one of the most potential clients of cloud services because of the widespread of wireless access and the limited resources available on mobile devices. For this reason, we maintain the principle of portability when designing and implementing GamingAnywhere. Currently the server supports Windows, Linux, and OS X, while the client supports Windows, Linux, OS X, and Android. 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 4.
(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 exposes 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.

1.2 Paper Organization

The rest of this paper is organized as follows. Section 2 discusses the related work in the literature. Section 3 depicts the system architecture. This is followed by the detailed elaborations of implementation issues in Section 4. Section 5 gives the experimental results. We conclude the paper in Section 6. Last, we empirically determine the best encoding parameters in Appendix A.

2. RELATED WORK

We first survey the existing cloud gaming systems. Then, we review prior works on quantifying the performance of cloud gaming systems.

2.1 Cloud Gaming Systems

We classify the cloud gaming systems into three genres: (i) 3D graphics streaming [Eisert and Fechteler 2008; Jurgelionis et al. 2009], (ii) video streaming [Winter et al. 2006; Holthe et al. 2009], and (iii) video streaming with post-rendering operations [Shi et al. 2011; Giesen et al. 2008]. 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 [Eisert and Fechteler 2008; Jurgelionis et al. 2009], 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. Because 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 [Winter et al. 2006; Holthe et al. 2009] 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 streaming clients on resource-constrained devices. Since the video streaming approach does not rely on specific 3D chips, the same clients can be readily ported to different platforms, which are potentially GPU-less.

The approach of video streaming with post-rendering operations [Shi et al. 2011; Giesen et al. 2008] 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 clients for augmenting motions, lighting, and materials using local resources [Chen et al. 2010]. These post-rendering operations have low computational complexity and run in real time even without GPUs.

Similar to the proprietary cloud gaming systems, the proposed GamingAnywhere employs the video streaming approach for lower loads on the clients. Differing from other systems [Winter et al. 2006; Holthe et al. 2009] in the literature, GamingAnywhere is open, modularized, cross-platform, and efficient. In fact, it is the first complete system of its kind, and is of interests for researchers, game developers, service providers, and end users. The initial version of GamingAnywhere [Huang et al. 2013] was introduced to the public in February 2013. Since then we have improved the system from several aspects: We have revised the mechanisms for video frame capture and user control event interception to improve the overall system performance. The architecture is now more neutral to platform-specific implementations so that GamingAnywhere supports exactly the same functionality on all the supported platforms. Further, we have integrated a number of new video encoders with GamingAnywhere and provide measurement studies based on the popular VP8 video encoder. GamingAnywhere is getting more matured and now researchers and developers are able to integrate the proposed system with their preferred flavors — as an standalone application, as as a static library, or as a dynamically-linked shared object.

2.2 Measuring the Performance of Cloud Gaming Systems

Measuring the performance of desktop streaming systems has long been considered in the literature [Lai and Nieh 2006; Nieh et al. 2003; Wong and Seltzer 1999; Packard and Gettys 2003; Tolia et al. 2006]. The slow-motion benchmarking [Lai and Nieh 2006; Nieh et al. 2003] runs a slow-motion

version of an application on the server, and analyzes network packets between the server and client. Slow-motion benchmarking augments the execution speed of applications, and is thus less suitable to real-time applications, including cloud games. Packard and Gettys [Packard and Gettys 2003] 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 [Wong and Seltzer 1999] 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 increases when the system load is high. Tolia et al. [Tolia et al. 2006] quantify the performance of several applications running on a VNC server, which is connected to a VNC client via a network with diverse round-trip time (RTT). It is determined that the response delay of these applications highly depends on the degree of the application's interactivity and network RTT. The performance metrics considered in [Lai and Nieh 2006; Nieh et al. 2003; Wong and Seltzer 1999; Packard and Gettys 2003; Tolia et al. 2006] are only suitable to desktop streaming systems, which do not impose strict real-time requirements.

More recently, a few studies measure the performance of remote desktop streaming and cloud gaming systems. Chang et al.'s [Chang et al. 2011] methodology to study the performance of games on desktop streaming systems has been employed to evaluate several implementations, including LogMeIn [LogMeIn 2012], TeamViewer [TeamViewer 2012], and UltraVNC [UltraVNC 2012]. Chang et al. establish that player performance and Quality-of-Experience (QoE) depend on video quality and frame rates. It is observed that the desktop streaming systems cannot support cloud games given that the achieved frame rate is as low as 9.7 fps. Chen et al. [Chen et al. 2011] propose another methodology to quantify the response delay, which is even more critical to cloud games [Claypool and Claypool 2006; Henderson 2003; Zander et al. 2005]. Two proprietary cloud gaming systems, OnLive [OnLive 2012] and StreamMyGame [StreamMyGame 2012], 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. [Chen et al. 2011] point out that the performance edge of OnLive can be attributed to its customized hardware platform and optimized game software. In addition, Lee et al. [Lee et al. 2012] 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. [Choy et al. 2012] evaluate whether a large-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 and unmodified game software?* We employ the measurement methodologies proposed in [Chen et al. 2011] to compare the proposed GamingAnywhere against OnLive.

3. SYSTEM ARCHITECTURE

Figure 1 presents the high-level deployment scenario of GamingAnywhere. 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. The portal server is a web service providing login and game-selection user interface. If login and game-selection requests are sent from a customized client, the portal server does not even need a fancy user interface. Actions can be

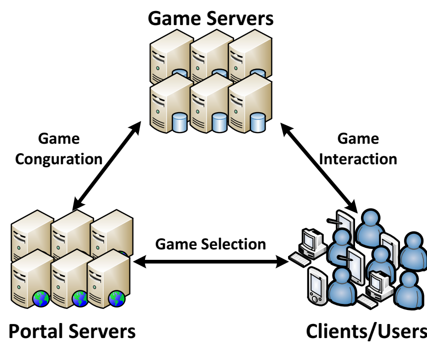

Fig. 1. The deployment scenario of GamingAnywhere.

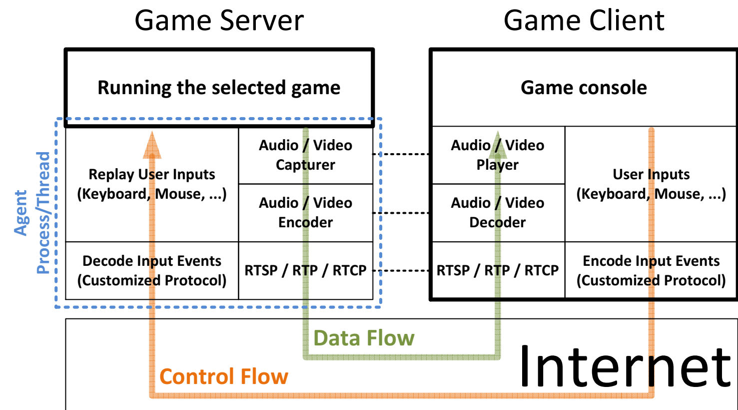

Fig. 2. The server and the client architecture of GamingAnywhere.

sent as REST-like [Fielding 2000; Costello 2007] requests via standard HTTP or HTTPS protocols. The design of the portal server is out of the scope of this article.

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 both PC- 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 module (in the form of shared object or DLL) 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 design and implement the transport protocol of user actions by ourselves.

The client is 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 [VideoLAN 2013], which are available on almost all OS's and platforms.

4. IMPLEMENTATION

The implementation of GamingAnywhere includes server and client, each of which contains a number of modules whose details are elaborated in this section. GamingAnywhere leverages several external libraries including *libavcodec*/*libavformat* [FFmpeg project 2013], *live555* [Live Networks 2013], and

[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.

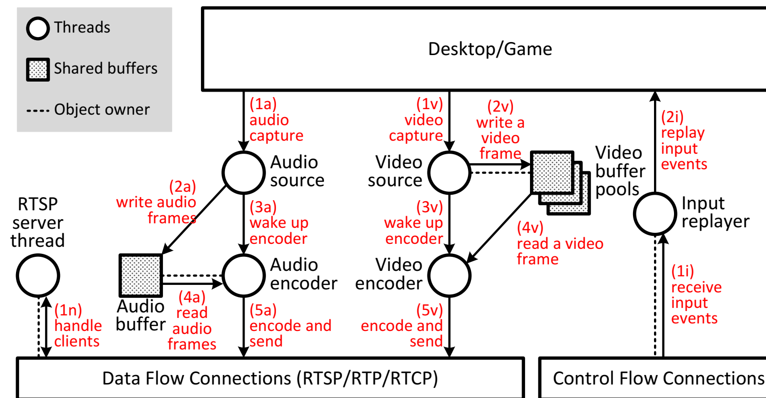

Fig. 3. The relationships among server modules, shared buffers, and network connections.

SDL library [Lantinga 2013]. 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 [Schulzrinne et al. 1998; Schulzrinne et al. 2003] at the client. The *Simple DirectMedia Layer (SDL)* library is a cross-platform multimedia 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.

4.1 GamingAnywhere Server

The relationships among server modules are depicted 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.

4.1.1 *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 encoded 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 [Schulzrinne et al. 1998], 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

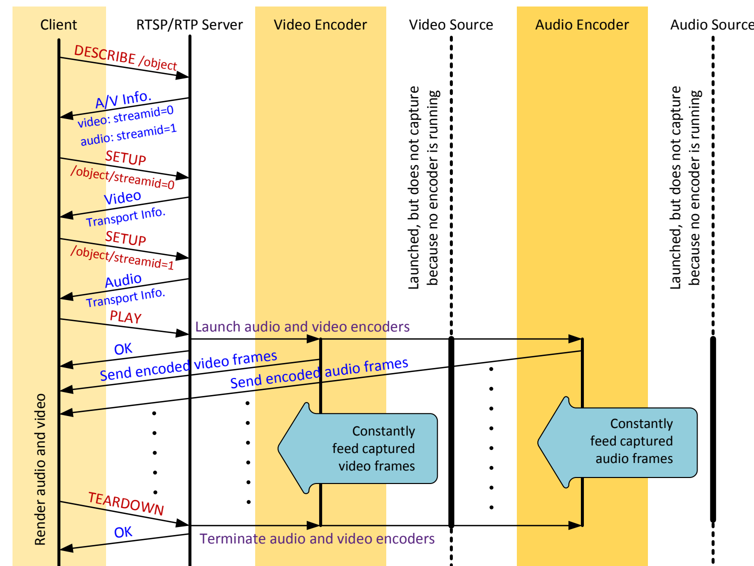

Fig. 4. The UML diagram for the RTSP/RTP protocol and the involved GamingAnywhere server components.

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 packets. If encoded frames are delivered as interleaved binary data, the frames are filled into a raw RTP packet and sent using the existing RTSP stream. On the other hand, if encoded frames are delivered via RTP over UDP, they are sent directly to the client via *libavformat* calls.

The RTSP server thread exports a programming interface for encoders to send encoded frames. Therefore, whenever 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. The UML diagram of the RTSP/RTP protocol and involved server components is shown in Figure 4. A client connects to the RTSP server, requests for the codec and channel information with the `DESCRIBE` command, sets up transport layer protocols for delivery of audio and video data with the `SETUP` command, and then starts receiving audio and video data after the `PLAY` command. The audio and video encoders are launched by the RTSP server on receipt of the `PLAY` command from the client. Meanwhile, the audio and the video sources also start to capture and feed audio and video frames to the corresponding encoders, which are responsible to encode and send encoded data to the client. The audio and video encoders are terminated when the client tears down the session or is disconnected.

4.1.2 *Video Source.* Capturing of game screens (frames) is a platform-dependent task. 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 on demand, and extracts the desired region if 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 [Microsoft 2012] 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 (≈ 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 [Tanenbaum 2002] to decide whether our module should capture the 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. However, if the game's frame rate is lower than the desired rate, the module simply captures game screens at the same rate of the game renderer. 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 a video encoder. The video source module serves as the only buffer writer, while the video encoder is the buffer reader. Therefore, a reader-writer lock must be acquired every time before accessing the shared buffer.

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, Direct3D, and SDL games on Windows and SDL games on Linux. 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.

4.1.3 *Audio Source.* Capturing of audio frames is a platform-dependent task 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 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 generate silence audio frames in the second case to resolve the frame discontinuity problem. We observe 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 5. 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.

4.1.4 *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

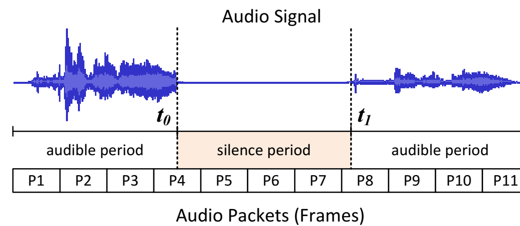

Fig. 5. Sample audio signals that may cause the frame discontinuity problem.

currently supports an *one-encoder-for-all* mode. In this 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. In case that a demand on providing different stream qualities for different observers in the *one-encoder-for-all* mode, we suggest to adopt a multimedia transcoder sit in-between the game server and the observers.

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 C. 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.

4.1.5 *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 formatting / 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 four input formats of Windows, X Window, Mac OS X, and SDL. Upon the receipt of an input event[3], 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, the XTEST extension on Linux, and the CGPostEvent function on Mac OS X. Additionally, with API interception techniques, GamingAnywhere also attempts to replay control events within a game process to shorten the replay latency incurred by the event processing mechanisms of operation systems. 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` func-

[2]In the current design, there can be one player and unlimited observers simultaneously in a game session.
[3]The capturing of input events on clients will be elaborated in Section 4.2.3.

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

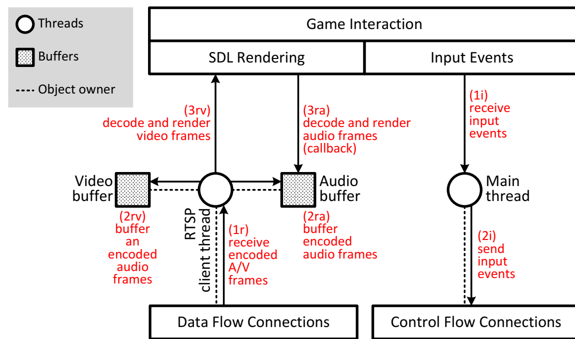

Fig. 6. The relationships among client modules, shared buffers, and network connections.

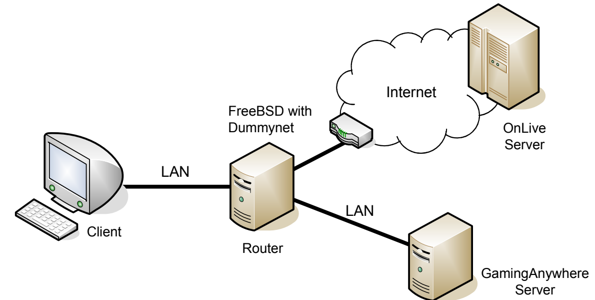

Fig. 7. The network topology of our experiments.

4.2 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 encoded audio and video frames. The relationships among client modules are shown in Figure 6. 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.

4.2.1 *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.

4.2.2 *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 6 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 [Schulzrinne et al. 2003] 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, it reduces the input-response latency due to video playout to a minimum level. We believe that this design tradeoff yields an 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 will wait until there are sufficient number of frames. We implement this waiting mechanism 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. To produce smooth audio outputs, the audio decoder starts to play when it has buffered at least n audio frames, which is by default 1024 frames per channel (approximately 23 ms for CD-quality stereo audio).

4.2.3 *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 doing so we perfectly mimic every move of the players' inputs on the game server.

5. EXPERIMENT STUDIES

In this section, we conduct extensive experiments to evaluate GamingAnywhere.

5.1 Setup

In our lab, we set up a testbed consisting of Windows 7 desktops with Intel 2.67 GHz i7 processors. Figure 7 illustrates the experimental setup, which consists of a server, a client, and a router. We compare the performance of GamingAnywhere against OnLive [OnLive 2012]. The OnLive client connects to the OnLive server over the Internet, while the GamingAnywhere client connects to its server via a LAN. To impose 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. Although the OnLive server is outside our LAN, we observed that the quality of the path was consistently good throughout the experiments. In particular, its network delay was around 130 ms and the packet loss rate is less than 10^{-6}. Hence, the path between the OnLive server and our client is essentially a communication channel with sufficient bandwidth, zero packet loss rate, and a constant 130 ms latency.

We configure the GamingAnywhere server to use x264 and the encoding parameters recommended in Appendix A. GamingAnywhere runs on UDP by default. We consider three games from three popular genres: action adventure (Batman), first-person shooter (FEAR), and real-time strategy (DOW). Detailed descriptions on the games are given in [Huang et al. 2013]. We set the encoding bit rate to be 3 Mbps. The games are streamed at a resolution of 720p with 50 fps (frames per second) unless otherwise specified. The detailed experimental designs and results are given in the rest of this section.

5.2 Responsiveness

We define response delay (RD) as the time difference between a user submitting a command and the corresponding in-game action appearing on the screen. Studies [Claypool and Claypool 2006; Henderson 2003; Zander et al. 2005] report that players of various game genres can tolerate different degrees of RD; for example, first-person shooter game players demand for less than 100 ms RD [Claypool and Claypool 2006]. We adopt the RD measurement procedure proposed in [Chen et al. 2011], which divides the RD into:

—*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 RTT.

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

ND can be measured using probing packets, e.g., in ICMP protocol, and is not controlled by cloud gaming systems. Thus, for a fair comparison between OnLive and GamingAnywhere, we compare only the processing delay on the server (PD) and the playout delay on the client (OD) of the two systems. Measuring PD and OD is much more challenging than measuring RD, because they occur internally in the cloud gaming systems, which may be closed and proprietary. The procedure detailed in [Chen et al. 2011] measures the PD and OD using external probes only, and thus works for OnLive even though we do not have access to their game servers in the cloud.

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 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 8 reports the average PD (server) and OD (client) achieved by the considered cloud gaming systems. We make several observations. First, the OD is small, ≤ 31 ms in all cases. 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, which is observed to be as high as 191 ms. This demonstrates the efficiency of the proposed GamingAnywhere: the PDs of OnLive are $3+$ times longer than that of GamingAnywhere. Last, only GamingAnywhere achieves sub-100 ms RD.

Figure 9 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 decoding

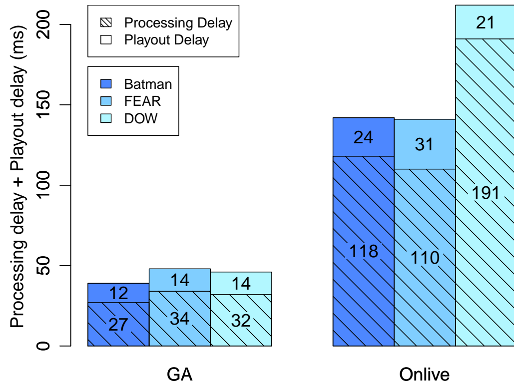

Fig. 8. Response delays of different systems.

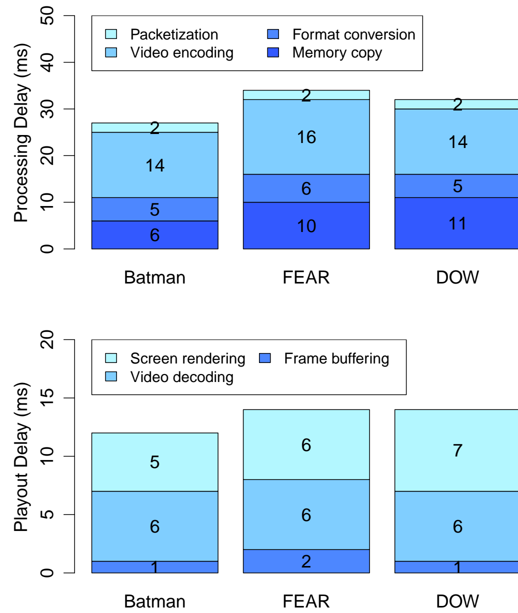

Fig. 9. Delay decomposition of GamingAnywhere.

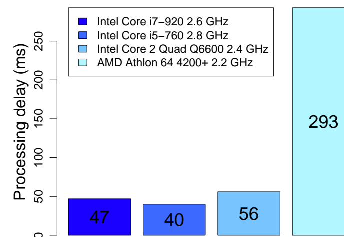

Fig. 10. Implication of CPU on responsiveness. Sample results from FEAR.

(at the client) operations, each frame is finished in at most 16 and 7 ms on average. Such a short delay contributes to the lower RD of GamingAnywhere.

Figure 10 reports how different CPUs affect the PD of GamingAnywhere. We do not consider OnLive servers as they are managed by OnLive Inc. This figure shows that GamingAnywhere achieves a PD of ≤ 56 ms on Intel C2D and better CPUs. Moreover, GamingAnywhere suffers from higher processing delays on AMD Althon 64 CPUs, which may be attributed to the SSSE3-enabled x264 binary, as the SSSE3 instruction set is not available on some AMD Althon 64 CPUs.

5.3 Network Loads

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 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

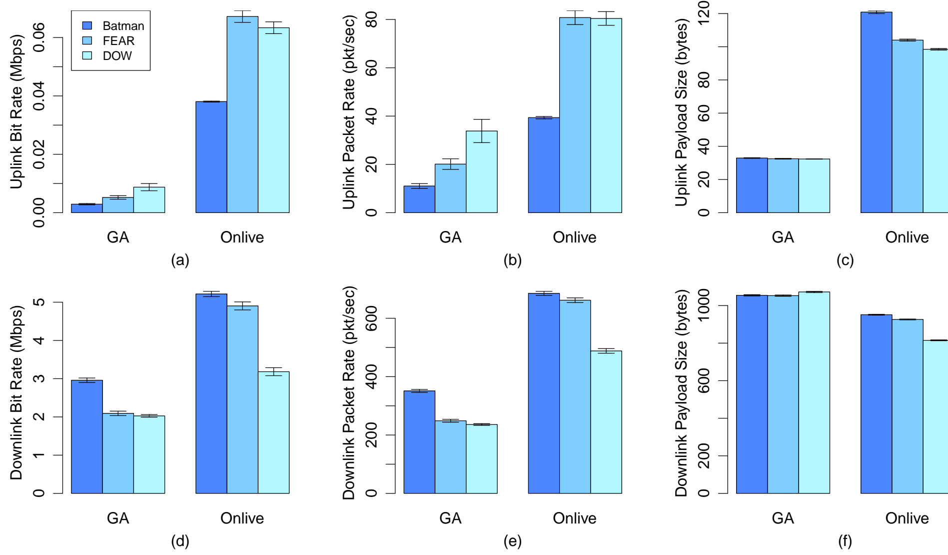

Fig. 11. Network loads incurred by the considered cloud gaming systems.

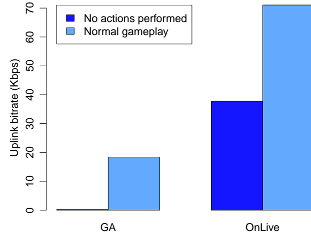

Fig. 12. The average uplink traffic rate when DOW is being actively played and kept idle.

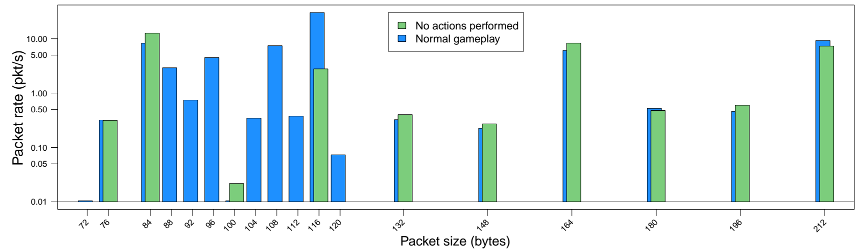

Fig. 13. The uplink packet sizes generated by the OnLive client when DOW is being played and kept idle (without players' actions).

possible and fight the opponents as in normal game plays. Second, he shall repeat his actions and trajectories as much as possible.

Results. Figure 11 plots the uplink and downlink traffic characteristics with 95% confidence intervals. Figures 11(a)–11(c) reveal that GamingAnywhere incurs a much lower uplink traffic loads compared to OnLive. Figures 11(d)–11(f) reveal that the downlink bit rates of GamingAnywhere are 2–3 Mbps, where those of OnLive are 3–5 Mbps. We notice that OnLive does not support user-configurable video encoding rates and parameters, which prevents them from being an experiment testbed like GamingAnywhere. Last, as we will see in Figures 16 and 17, even when we set the encoding bit rate of GamingAnywhere to be 3 Mbps, GamingAnywhere still outperforms OnLive in terms of video quality. The gap will be even larger if we increase GA's encoding bit rate.

We make another observation on Figure 11(d): 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 rates and may result in irregular playouts. Our in-depth analysis shows that, unlike Batman, both FEAR and DOW use Direct3D *multisampling* surfaces, which cannot be locked for memory copies. More specifically, an additional *non-multisampling* surface and an extra copy operation are required for FEAR and DOW, which in turn slightly affects the achieved frame rates.

Another observation that seems counter-intuitive in the first glance is Figure 11(a): The OnLive client sends out much more traffic to the server compared with GamingAnywhere. Since the uplink traffic should comprise only players' control actions, it seems unreasonable to have such a huge difference in uplink traffic between OnLive and GamingAnywhere. Our further analysis reveals that OnLive generates uplink traffic even when no gameplay actions are performed. Taking DOW as an example, we ask a gamer to play the game normally for 3 minutes and leave the game idle for another 3 minutes. We plot the average uplink traffic rate in Figure 12. The graph shows that OnLive generates around 40 kbps uplink traffic even when no gameplay actions are performed. Moreover, the differences between the idle and normal gameplay periods are 20 kbps and 30 kbps on GamingAnywhere and OnLive respectively, which indicate the rate of the traffic corresponding to players' actions. We further dissect the uplink packet sizes of OnLive. As shown in Figure 13, the OnLive client keeps sending packets with sizes 84, 164, 212, 116, and so on no matter whether a player is performing gameplay actions or not. More specifically, even when no gameplay actions are issued, the client sends out 40 packets per second on average with particular packet sizes. We believe that these packets are employed for path quality estimation and/or application-level acknowledgement purposes.

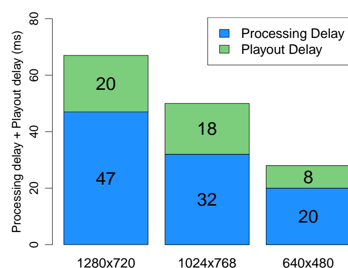

Fig. 14. Implication of resolutions on responsiveness, sample results from FEAR.

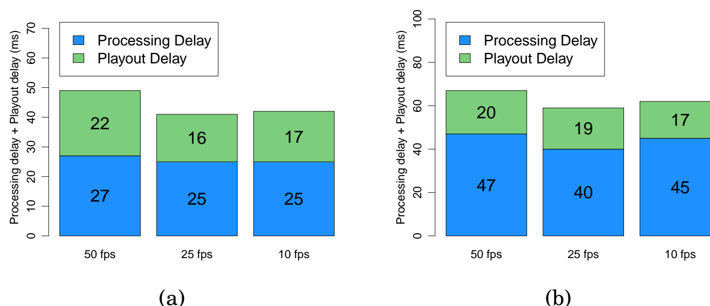

Fig. 15. Implication of frame capture rates on responsiveness: (a) Batman and (b) FEAR.

5.4 Impact of Screen Setting

We next vary the screen resolution and frame capture rate. We consider three resolutions: 1280x720, 1024x768, 640x480, and three frame capture rates: 50, 25, 10 fps. Since OnLive does not support different resolutions, we do not consider it.

Results. Figure 14 gives the impact of resolutions. This figure shows that GamingAnywhere scales well with screen resolutions, in terms of PD and OD, while Figure 15 presents the PD and OD, which shows that GamingAnywhere achieves stable PD and OD under different frame capture rates.

5.5 Streaming Quality under Different Network Conditions

The network conditions are the keys for high-quality video streaming, and we use `dummynet` to vary ND between 0–600 ms, packet loss rate between 0–10%, and bandwidth between 1–6 Mbps. We also include experiments with *unlimited* bandwidth. For OnLive, the ND in the Internet is already 130 ms

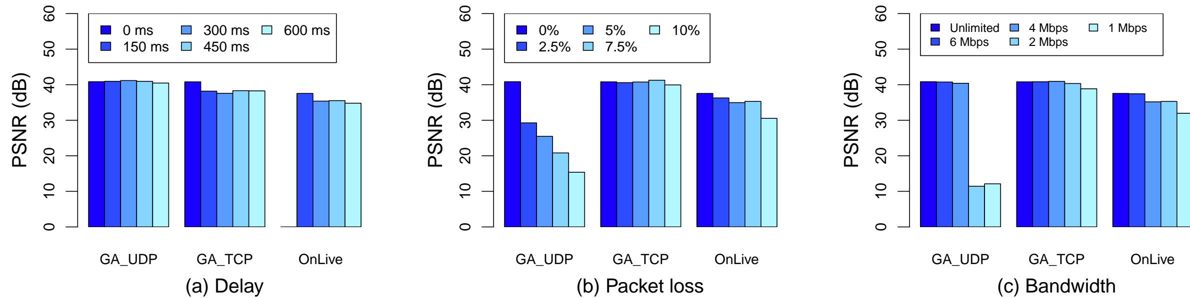

Fig. 16. Achieved video quality in PSNR under different network conditions.

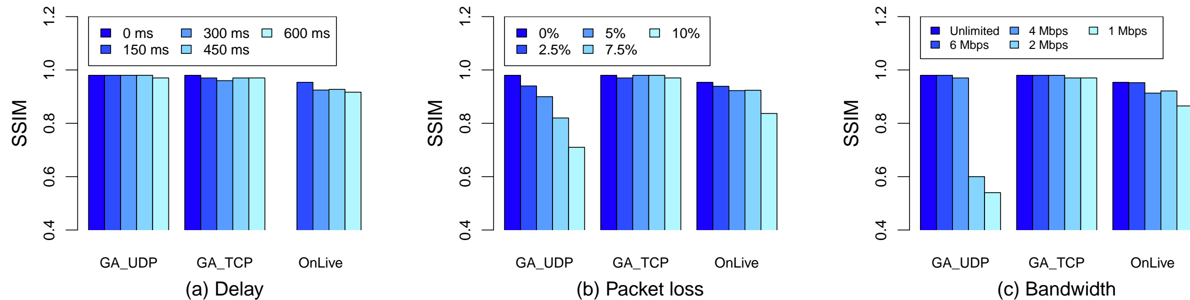

Fig. 17. Achieved video quality in SSIM under different network conditions.

and thus we cannot report the results of zero ND. Two video quality metrics, PSNR [Wang et al. 2001, p. 29] and Structural Similarity (SSIM) [Wang et al. 2004], are adopted. We run GamingAnywhere on both UDP and TCP in this experiment.

Results. Figures 16 and 17 present the PSNR and SSIM values, respectively. We make four observations. First, ND does not affect the video quality too much (Figures 16(a) and 17(a)). Second, GamingAnywhere achieves much higher video quality than OnLive: up to 5 dB (PSNR) and 0.05 (SSIM) gaps are observed. Third, GamingAnywhere over UDP suffers from quality drops when packet loss rate is nontrivial, as illustrated in Figures 16(b) and 17(b). This can be attributed to the missing error resilience mechanism in GamingAnywhere. This issue can be coped with running GamingAnywhere over TCP, which leads to stable video quality even under nontrivial packet loss rate, as shown in Figures 16(b) and 17(b). Last, Figures 16(c) and 17(c) show that the video quality of GamingAnywhere suddenly drops when the bandwidth is smaller than the encoding bit rate of 3 Mbps. In summary, GamingAnywhere performs well under diverse network conditions, except: (i) when the available bandwidth is lower than the specified video encoding bit rate and (ii) when the packet loss rate is high while GamingAnywhere runs on UDP. These advantages of GamingAnywhere are achieved under a rather low video coding rate at 3 Mbps (see Figure 11(d)).

5.6 Performance Profiling

We also profile the performance of GamingAnywhere server to identify possible performance bottlenecks. We use `oprofile` [Levon 2013] and `lttng` [Desnoyers et al. 2013] to conduct the measurements on Ubuntu Linux with the game Cube 2: Sauerbraten [van Oortmerssen 2013], which is a 3D fps game based on SDL and OpenGL. We use `oprofile` to retrieve performance counters at the function level

without modifying and recompiling the source code; meanwhile, we use `lttng` to obtain the usage of kernel functions and CPU counters. To observe performance overheads of the GamingAnywhere server, we play the selected game with two different setups. In the first setup, we play the game locally without the involvement of any cloud gaming technologies. In the second setup, we launch the game with the GamingAnywhere event-driven server and play the game remotely. In each game session, we choose the single player private "Run N' Gun Part I" campaign, kill five monsters, and then quit the game. We play each game setup three times and the average game play length is approximately 1 minute.

Results. The detailed profiling results are given in Appendix B due to the space limitations. We only summarize our observations in this section. When playing with GamingAnywhere, almost all the performance counters raise significantly. This is because many GamingAnywhere server operations, such as video frame capture (via `libdricore.so`), video encoding (in `libx264.so`), and color space conversion (in `libga.so`), require significant memory accesses and CPU time. Even so, the profiling results indicate that these operations are performed efficiently in that the branch miss rate and dTLB store miss rate with GamingAnywhere are even better (i.e., lower) than those without GamingAnywhere. The results also reveal that GamingAnywhere does not incur much overhead on the system kernel; actually, the number of system calls with GamingAnywhere is even less than that without GamingAnywhere. A closer look shows that the local game session invokes a large number of `ioctl` system calls, which should be used to handle inputs from control devices and outputs to the audio and graphics device. While a remote game session with GamingAnywhere still requires input handling, it does not need to output audio and graphics to local devices thus the `ioctl` system calls are largely eliminated. We also identify that GamingAnywhere makes a lot of `futex` system calls for synchronizing mutexes among threads. This is because GamingAnywhere adopts a multi-threaded architecture to manage the game screen capture, encoding, and packetization pipeline, and some of the threads inevitably require accesses to shared buffers, thus mutexes are used to ensure the synchronization among threads.

5.7 Multiple GamingAnywhere Instances

So far we are running a single instance of GamingAnywhere on a game server. Since GamingAnywhere does not exclusively use any input/output device, it is possible to run multiple instances of GamingAnywhere on the same server so that the server can serve multiple players at the same time. Using the same testbed (Windows 7 desktops with Intel 2.67 GHz i7 processors), we run 1, 2, 3, and 4 instances of GamingAnywhere on the game Cube 2: Sauerbraten with each instance serving one client, and measure the average processing delays of the instance(s).

Results. Our experiments reveal that the processing delays of the GamingAnywhere server increase linearly along with the increasing number of simultaneous instances. We plot the figure in Appendix B due to the space limitations. This indicates that GamingAnywhere scales well in terms of multiple instances and that GamingAnywhere is capable to be the cloud gaming solution hosting large-scale remote gaming services.

6. CONCLUSION

We presented GamingAnywhere—the first open cloud gaming system, which is designed to be open, extensible, portable, and fully configurable. Via extensive experiments, we have shown that GamingAnywhere significantly outperforms a well-known, commercial, cloud gaming system: OnLive. Compared to GamingAnywhere, for example, OnLive suffers from up to 3 times higher processing delays, and 5 dB lower video quality. GamingAnywhere: (i) incurs lower network loads in both uplink and downlink directions, (ii) scales well with screen resolutions and frame capture rates, and (iii) adapt to diverse network conditions (except some boundary cases). We expect that cloud game developers, cloud service providers, system researchers, and individual users will use GamingAnywhere to set up complete cloud

gaming testbeds for different purposes. In fact, a few weeks after making GamingAnywhere online at `http://gaminganywhere.org` in April 2013, we have received many inquiries. We firmly believe that the release of GamingAnywhere will stimulate more research innovations on cloud gaming systems, or multimedia streaming applications in general.

For example, this article does not address the deployment problem of GamingAnywhere, in which cloud game hosting companies have to find the best tradeoff between the hardware and bandwidth investment and achieved gaming experience. Techniques for cloud management, such as resource allocation and Virtual Machine (VM) migration, are critical to the success of commercial deployments. These cloud management techniques need to be optimized for cloud games, e.g., the VM placement decisions need to be aware of gaming experience. Tiered cloud platforms may also be used in real deployment, e.g., relatively delay-tolerant games may be served by servers in further-away large data centers, while delay-sensitive games may be served by servers in edge clouds for high responsiveness.

On another direction, GamingAnywhere attempts to transparently support cloud games, i.e., without any code customization, and thus works with almost all computer games. A design alternative is to provide APIs for game developers to call, which may enable more optimization opportunities and reduce overhead. For example, if game developers adopt the APIs, a cloud gaming platform like GamingAnywhere will no longer need to hook into functions in the operating systems, which results in lower overhead. Another more aggressive, although complex, optimization strategy is to divide the game engine into multiple components and dynamically distribute these components on multiple cloud servers based on the demanded and available resources of the games and servers. While such optimization is out of the scope of this article, GamingAnywhere is an enabler of designing, implementing, and evaluating such design issues in comprehensive cloud gaming platforms.

REFERENCES

Yu-Chun Chang, Po-Han Tseng, Kuan-Ta Chen, and Chin-Laung Lei. 2011. Understanding The Performance of Thin-Client Gaming. In *Proceedings of IEEE CQR 2011*.

Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. 2011. Measuring The Latency of Cloud Gaming Systems. In *Proceedings of ACM Multimedia 2011*.

Kuan-Ta Chen, Polly Huang, and Chin-Laung Lei. 2009. Effect of Network Quality on Player Departure Behavior in Online Games. *IEEE Transactions on Parallel and Distributed Systems* 20, 5 (May 2009), 593–606.

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

Sharon Choy, Bernard Wong, Gwendal Simon, and Catherine Rosenberg. 2012. The Brewing Storm in Cloud Gaming: A Measurement Study on Cloud to End-User Latency. In *Proceedings of IEEE/ACM NetGames 2012*.

M. Claypool and K. Claypool. 2006. Latency and Player Actions in Online Games. *Commun. ACM* 49, 11 (November 2006), 40–45.

Roger L. Costello. 2007. Building web services the rest way. xFront - Tutorial and Articles on XML and Web Technologies. (2007). http://www.xfront.com/REST-Web-Services.html.

Mathieu Desnoyers, Julien Desfossez, and David Goulet. 2013. Linux Trace Toolkit - next generation. LTTng Project. (2013). https://lttng.org/.

EA 2012. Electronic Arts Buys Online Gaming Studio ESN, The Developers Behind Battlefield's Battlelog Online Social Network. (September 2012). http://techcrunch.com/2012/09/26/electronic-arts.

P. Eisert and P. Fechteler. 2008. Low Delay Streaming of Computer Graphics. In *Proc. of IEEE International Conference on Image Processing (ICIP'08)*. San Diego, CA, 2704–2707.

FFmpeg project. 2013. ffmpeg. (2013). http://ffmpeg.org/.

Roy Thomas Fielding. 2000. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Dissertation. University of California, Irvine.

GaiKai 2012. GaiKai Web Page. (July 2012). http://www.gaikai.com/.

F. Giesen, R. Schnabel, and R. Klein. 2008. Augmented Compression for Server-Side Rendering. In *Proc. of International Fall Workshop on Vision, Modeling, and Visualization (VMV'08)*. Konstanz, Germany.

T. Henderson. 2003. *The Effects of Relative Delay in Networked Games*. Ph.D. Dissertation. Department of Computer Science, University of London.

O. Holthe, O. Mogstad, and L. Ronningen. 2009. Geelix LiveGames: Remote Playing of Video Games. In *Proc. of IEEE Consumer Communications and Networking Conference (CCNC'09)*. Las Vegas, NV.

Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. 2013. GamingAnywhere: An Open Cloud Gaming System. In *Proc. of ACM MMSys 2013*.

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. 2009. Platform for distributed 3D Gaming. *International Journal of Computer Games Technology* 2009 (January 2009), 1:1–1:15.

A. Lai and J. Nieh. 2006. On the Performance of Wide-Area Thin-Client Computing. *ACM Transactions on Computer Systems* 24, 2 (May 2006), 175–209.

Sam Lantinga. 2013. Simple DirectMedia Layer. (2013). http://www.libsdl.org/.

Yeng-Ting Lee, Kuan-Ta Chen, Han-I Su, and Chin-Laung Lei. 2012. Are All Games Equally Cloud-Gaming-Friendly? An Electromyographic Approach. In *Proceedings of IEEE/ACM NetGames 2012*.

John Levon. 2013. OProfile - A System Profiler for Linux. (2013). http://oprofile.sourceforge.net/.

Inc. Live Networks. 2013. LIVE555 Streaming Media. (2013). http://live555.com/liveMedia/.

LogMeIn 2012. LogMeIn Web Page. (July 2012). https://secure.logmein.com/.

Microsoft. 2012. 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.

J. Nieh, S. Yang, and N. Novik. 2003. Measuring Thin-Client Performance Using Slow-Motion Benchmarking. *ACM Transactions on Computer Systems* 21, 1 (February 2003), 87–115.

OnLive 2012. OnLive Web Page. (July 2012). http://www.onlive.com/.

K. Packard and J. Gettys. 2003. X Window System Network Performance. In *Proc. of USENIX Annual Technical Conference (ATC'03)*. San Antonio, TX, 206–218.

H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. 2003. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard). (July 2003). http://www.ietf.org/rfc/rfc3550.txt.

H. Schulzrinne, A. Rao, and R. Lanphier. 1998. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard). (April 1998). http://www.ietf.org/rfc/rfc2326.txt.

S. Shi, C. Hsu, K. Nahrstedt, and R. Campbell. 2011. Using Graphics Rendering Contexts to Enhance the Real-Time Video Coding for Mobile Cloud Gaming. In *Proc. of ACM Multimedia'11*. Scottsdale, AZ, 103–112.

Sony 2012. Cloud Gaming Adoption is Accelerating ... and Fast! (July 2012). http://www.nttcom.tv/2012/07/09/cloud-gaming-adoption-is-acceleratingand-fast/.

StreamMyGame 2012. StreamMyGame Web Page. (July 2012). http://streammygame.com/.

Andrew Stuart Tanenbaum. 2002. *Computer Networks* (4th ed.). Prentice Hall Professional Technical Reference.

TeamViewer 2012. TeamViewer Web Page. (July 2012). http://www.teamviewer.com.

N. Tolia, D. Andersen, and M. Satyanarayanan. 2006. Quantifying Interactive User Experience on Thin Clients. *IEEE Computer* 39, 3 (March 2006), 46–52.

UltraVNC 2012. UltraVNC Web Page. (July 2012). http://www.uvnc.com/.

Wouter van Oortmerssen. 2013. Cube 2: Sauerbraten. (2013). http://sauerbraten.org/.

VideoLAN. 2013. VLC media player. Official page for VLC media player, the Open Source video framework!. (2013). http://www.videolan.org/vlc/.

Y. Wang, J. Ostermann, and Y. Zhang. 2001. *Video Processing and Communications*. Prentice Hall.

Z. Wang, L. Lu, and A. Bovik. 2004. Video Quality Assessment Based on Structural Distortion Measurement. *Signal Processing: Image Communication* 19, 2 (February 2004), 121–132.

WebM 2013. The WebM Project Web Page. (April 2013). http://www.webmproject.org.

D. Winter, P. Simoens, L. Deboosere, F. Turck, J. Moreau, B. Dhoedt, and P. Demeester. 2006. A Hybrid Thin-Client Protocol for Multimedia Streaming and Interactive Gaming Applications. In *Proc. of ACM NOSSDAV 2006*. Newport, RI.

A. Wong and M. Seltzer. 1999. Evaluating Windows NT Terminal Server Performance. In *Proc. of USENIX Windows NT Symposium (WINSYM'99)*. Seattle, WA, 145–154.

x264 2012. x264 Web Page. (July 2012). http://www.videolan.org/developers/x264.html.

S. Zander, I. Leeder, and G. Armitage. 2005. Achieving Fairness in Multiplayer Network Games through Automated Latency Balancing. In *Proc. of ACM SIGCHI ACE 2005*. Valencia, Spain, 117–124.

Received May 2013; revised September 2013; accepted October 2013

Online Appendix to: GamingAnywhere—The First Open Source Cloud Gaming System

CHUN-YING HUANG, National Taiwan Ocean University, Taiwan
KUAN-TA CHEN, Academia Sinica, Taiwan
DE-YU CHEN, Academia Sinica, Taiwan
HWAI-JUNG HSU, Academia Sinica, Taiwan
CHENG-HSIN HSU, National Tsing Hua University, Taiwan

A. REAL-TIME VIDEO ENCODING PARAMETERS

GamingAnywhere supports `x264` [x264 2012] and `vpxenc` [WebM 2013] encoders for H.264/AVC and VP8 video encoding. These encoders are fairly comprehensive and provide many parameters to users for trading off the bit rate, video quality and encoding complexity. In this section, we conduct extensive experiments on an Intel i7 PC to find the best tradeoff settings for `x264` and `vpxenc`. We use PSNR as the video quality metric, and fps as the computational complexity metric. Typically, higher PSNR (higher video quality) comes with lower fps (higher computational complexity). Our goal is to achieve $\sim$60 fps at 720p (1020x720) *and* the highest rate-distortion (R-D) performance. We record 10-min game plays in YUV format from three games: Batman, FEAR, and DOW, which are chosen from three different game genres (see Section 5.1). We then encode the raw videos with various parameters and report our observations.

A.1 x264 Encoding Parameters

Mandatory parameters for real-time encoding. Several parameters are required for real time `x264` encoding. First, we need to disable the bi-directional (B) frames. We also need to disable the lookahead buffers, which are used for frame type (I, P, or B) decision and frame-level multi-threading. Disabling these two coding tools can be done by a convenience flag `--tune zerolatency`. Second, we need to enable *slice-level* multi-threading to leverage multi-core CPUs without incurring additional delay. Slices are essentially disjoint regions extracted from each video frame. Slice-level multi-threading cuts each frame into t slices, and allocates a thread to encode each slice. `x264` 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, `x264` supports *intra refresh* to control error propagation due to packet losses. With intra refresh, each frame consists of a column of intra-coded macroblocks, and this intra-coded column moves along the time. `x264` allows intra refresh via the flag `--intra-refresh`.

Implications of other parameters. We exercise several other parameters and present their implications on video quality and computational complexity. `--preset` is used to select a predefined complexity level, ranging from ultrafast to very slow; `--bitrate` is used to set the target bit rate; `--me` is used to select the motion estimation algorithms, which can be diamond, hexagonal, multi-hexagonal, exhaustive, and Hadamard exhaustive search, from low to high complexity; and lastly, `--merange` specifies the motion vector search range, from 4 to 64 pixels. Moreover, we consider the number of threads

© 2010 ACM 1551-6857/2010/05-ART1 $15.00
DOI: http://dx.doi.org/10.1145/0000000.0000000

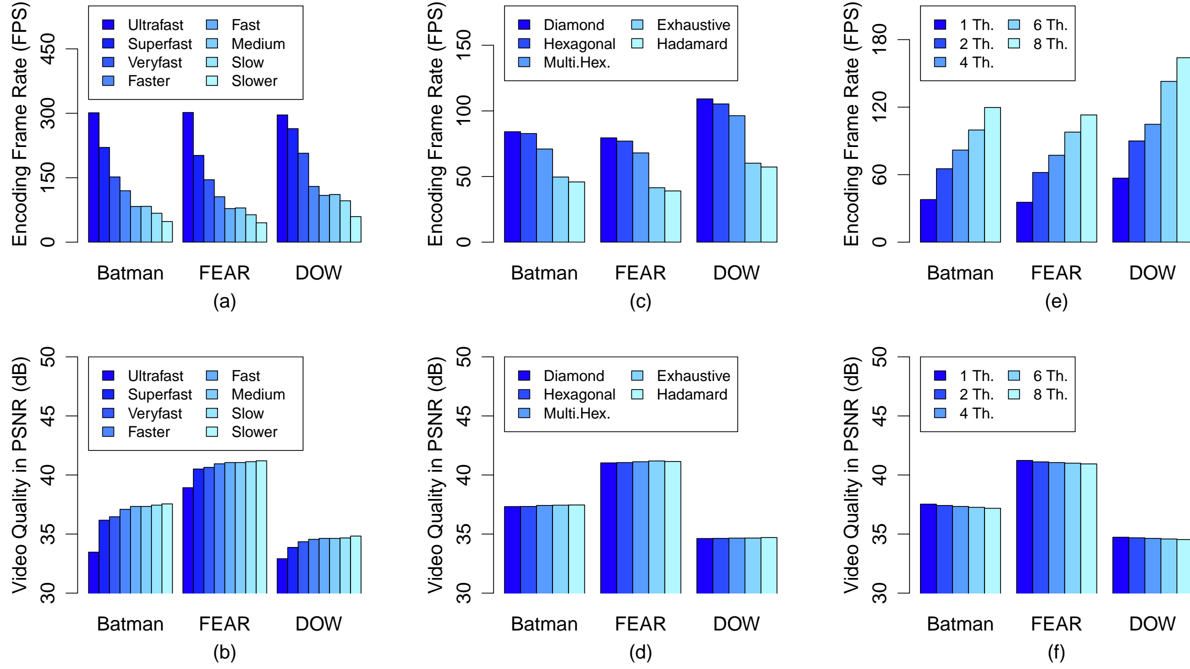

Fig. 18. Results from x264. Different presets: (a) encoding frame rates and (b) achieved video quality. Different motion estimation algorithms: (c) encoding frame rates and (d) achieved video quality. Implications of sliced-level threading: (e) encoding frame rates and (f) achieved video quality.

$t \in \{1, 2, 4, 6, 8\}$, and GoP size $g \in \{12, 24, 48, 96, 192, 384\}$. The GoP size is set by `--keyint`. If not otherwise specified, we employ `--preset fast`, `--bitrate 1000`, `--me hex`, `--merange 16`, $t = 4$, and $g = 384$.

We plot the results from different presets in Figures 18(a) and 18(b). We find that *very fast* preset leads to 100+ fps in all considered games. Moreover, moving from *very fast* to *fast* only results in up to 0.46 dB quality improvement, at the cost of at least 60 fps loss. Based on this observation, we recommend using *very fast*. We present the results from different motion estimation algorithms in Figures 18(c) and 18(d). Figure 18(d) shows that different algorithms lead to almost the same video quality. Figure 18(c) reveals that the exhaustive search algorithms may reduce the encoding frame rate by about 50%, for virtually no quality improvement. Hence, we recommend the diamond search algorithm (*dia*). We also find that increasing the search range (16 pixels by default) results in negligible impact on video quality and encoding complexity. We give the results from different number of threads in Figures 18(e) and 18(f). Figure 18(e) reveals that more slice-level threads result in higher encoding frame rates. Figure 18(f) shows that slice-level multi-threading leads to insignificant quality degradation. For example, increasing the number of threads from 1 to 4 results in a small video quality degradation of at most 0.19 dB, while the encoding frame rate is almost doubled. Hence, we recommend to use 4 threads. We plot the video quality with different GoP size in Figure 19, which shows that Batman and FEAR are less sensitive to the GoP size. DOW is more vulnerable to smaller GoP sizes, because the background of Real-Time Strategy (RTS) games does not move rapidly, and thus offers more inter-frame redundancy. We find that the GoP size does not affect the computational

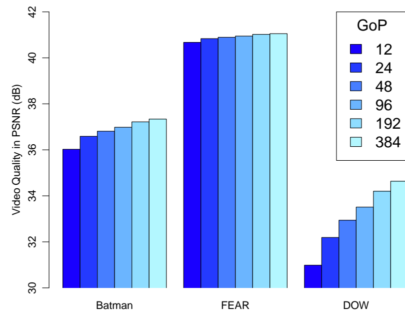

Fig. 19. Results from x264. Quality degradation due to smaller GoP size.

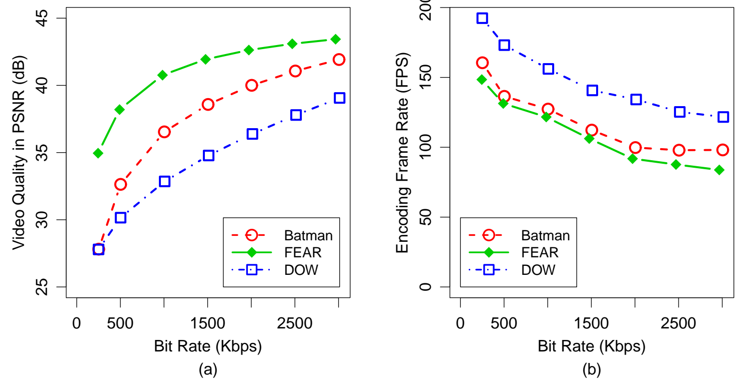

Fig. 20. Tradeoffs between rate and: (a) quality and (b) complexity. Results from x264.

complexity of x264. The best GoP size depends on the network condition, and we chose a medium GoP size of 48.

Performance of x264. We report the complexity-rate-distortion relation under the recommended 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 vary $r between 250 and 3000 kbps and plot the rate-complexity and rate-quality curves in Figure 20. Figure 20(a) reveals that, for Batman, FEAR, and DOW, we achieve a good video quality of 35 dB at respective bit rates of only about 250, 800, and 1500 kbps. For an excellent video quality of 40 dB [Wang et al. 2001, p. 29], Batman and FEAR require bit rates of about 800 and 2000 kbps, while DOW demands slightly over 3000 kbps. These bit rates are widely available in modern access networks. Last, Figure 20(b) reports the encoding frame rates under different bit rates. This figure reveals that for a video quality of 35 dB, the encoding frame rates are 160+, 130+, and 140+, which are much higher than the rendering frame rates of most games.

A.2 vpxenc Encoding Parameters

Mandatory parameters for real-time encoding. We present the required parameters for real-time `vpxenc` encoding in the following. First, we need to enable one-pass encoding (instead of two-pass encoding) using `--passes=1`. Second, we set `--end-usage=cbr` to use CBR (constant bit rate) encoding, which reduces the rate fluctuations and thus is more suitable to real-time streaming. Third, real-time encoding dictates *zero buffering*, which is achieved by `--buf-initial-sz=0`, `--buf-optimal-sz=0`, `--buf-sz=0`. Fourth, we enable multi-threading by (i) `--threads` to specify the number of threads and (ii) `--token-parts` to specify the number of *partitions*, where each partition may be encoded by different entropy encoders (in different threads). We set the number of partitions to its maximal value 8, and vary the number of threads.

Implications of other parameters. We study how the parameters affect video quality and computational complexity. `vpxenc` exposes fewer encoding parameters, compared to `x264`. `vpxenc` supports 3 modes and 23 levels that lead to different tradeoffs of video quality and computational complexity. The three modes are: `--best`, `--good`, and `--rt`, where good and rt have 6 and 16 levels, respectively. The 16 levels of mode rt specify a target CPU usage (of vpxenc), from 0 to 100%. We consider

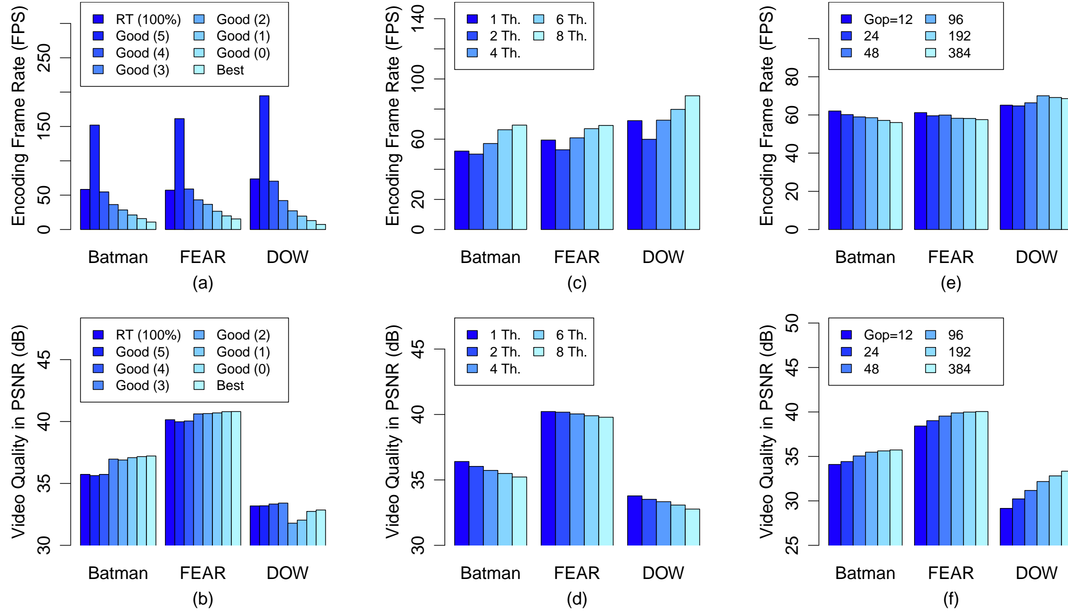

Fig. 21. Results from vpxenc. Different presets: (a) encoding frame rates and (b) achieved video quality. Different motion estimation algorithms: (c) encoding frame rates and (d) achieved video quality. Implications of sliced-level threading: (e) encoding frame rates and (f) achieved video quality.

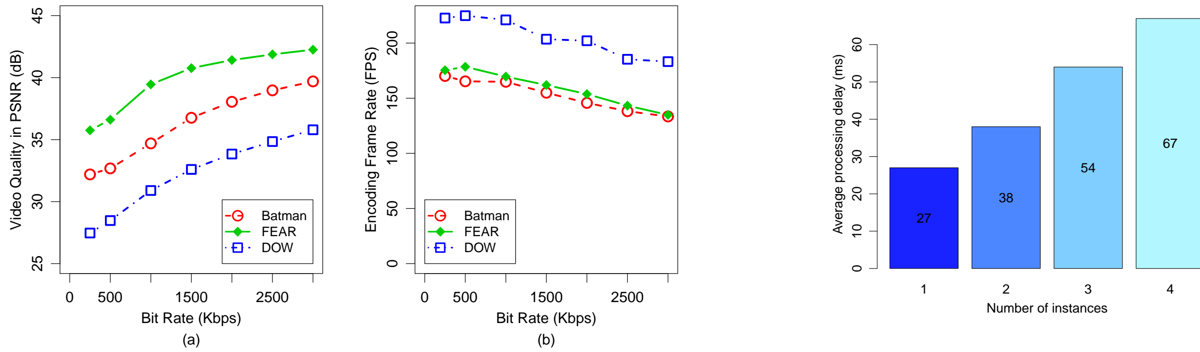

Fig. 22. Tradeoffs between rate and: (a) quality and (b) complexity. Results from vpxenc.

Fig. 23. Processing delays of multiple GamingAnywhere instances running the game Cube 2: Sauerbraten.

7 encoding modes/levels in total, best, good(0), good(1), . . . , good(5), and rt(100%), from high to low complexity. `--target-bitrate` sets the target rate, and `--kf-max-dist` sets the maximal GoP size. If not otherwise specified, we employ `--good`, `--cpu-used=4`, `--target-bitrate=100`, `--threads 4`, `--kf-max-dist 384`.

We plot the results from different models/levels in Figures 21(a) and 21(b). We find that vpxenc only achieves ≤ 50 fps, except with good(5) (`--good` and `--cpu-used=5`). Since the video quality drop from good(4) to good(5) is negligible, we recommend to use good(5). We present the results from different

Table I. Profiling summary for Cube 2: Sauerbraten — Without and with GamingAnywhere

	Local gameplay (without GA)		Remote gameplay (with GA)	
CPU cycles	66,263,040,386 (100%)		613,822,039,953 (100%)	
consumed by	22.80%	libdricore.so	47.99%	libdricore.so
	16.02%	sauerbraten†	20.48%	libx264.so
	13.07%	[kernel]	12.27%	libga.so
	12.65%	i965_dri.so	4.92%	libc.so
	8.84%	libc.so	3.54%	[kernel]
	5.61%	libjpeg.so	2.42%	sauerbraten
Branches	9,816,197,930		59,332,965,017	
Branch misses	274,832,612		1,047,463,987	
Branch miss rate	2.79%		1.76%	
Cache misses	57,157,352		1,018,626,894	
dTLB loads	22,944,538,803		195,569,272,710	
dTLB stores	12,663,900,914		90,938,730,599	
dTLB store misses	11,571,948		50,717,095	
dTLB store miss rate	0.09%		0.05%	
Invoked system calls	1,368,700 (100%)		804,627 (100%)	
contributed by	1,083,856 (79%)	ioctl	491,002 (61%)	ioctl
	142,145 (10%)	clock_gettime	68,274 (8%)	clock_gettime
	38,403 (3%)	nanosleep	65,466 (8%)	futex
	24,471 (2%)	socketcall	51,969 (6%)	nanosleep
	19,152 (1%)	read	31,817 (4%)	socketcall

† sauerbraten is the process name of the game Cube 2: Sauerbraten.

number of threads in Figures 21(c) and 21(d). Figure 21(d) shows that the video quality drops when there are more threads. This can be attributed to the design of multiple partitions and entropy coders—more partitions mean less redundancy. Figure 21(c) reveals that 6 threads are required for 60 fps and thus we recommend to use 6 threads. The impact of GoP size is shown in Figures 21(e) and 21(f). Although we observe a tradeoff between video quality and computational complexity, the deviation is moderate, and the best tradeoff depends on the network condition. We chose a medium GoP size of 48.

Performance of vpxenc. We report the complexity-rate-distortion relation under the recommended encoding parameters:

```
--i420 -w 1280 -h 720 -p 1 -t 6 --token-parts=3 --good --cpu-used=5 --end-usage=cbr
--target-bitrate=$r --fps=30000/1000 --buf-sz=0 --buf-initial-sz=0 --buf-optimal-sz=0
--kf-max-dist=48.
```

We vary $r between 250 and 3000 kbps, and plot the rate-complexity and rate-quality curves in Figure 22. Figure 22(a) shows that vpxenc achieves 35 dB at bit rates of about 250, 1000, and 3000 kbps for Batman, FEAR, and DOW, respectively. Figure 22(b) reveals that the corresponding encoding frame rates are 170+ in all considered games. Compared to x264 (Figure 20), we found that vpxenc achieves a slightly higher frame rates (up to 30 fps) at the expense of lower coding efficiency. Nonetheless, the resulting bit rates are available in most access networks nowadays.

B. ADDITIONAL EXPERIMENTAL RESULTS

Table I presents the detailed profiling results. Corresponding descriptions are given in Section 5.6. Figure 23 gives the results of running multiple GamingAnywhere instances, and the detailed discussions are given in Section 5.7.