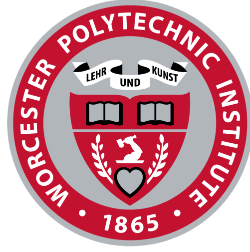


Improvement to Quality of Experience in Cloud-Based Game Streaming with Jitter Buffer Management

MS Thesis Report

By

Sudipta Biswas

Advisor

Mark Claypool

Reader

Craig E Wills

Computer Science Department

Worcester Polytechnic Institute

100 Institute Road

Worcester MA 01609

Date: April 2025

Abstract

Cloud-based game streaming has the potential to deliver high-quality gaming experiences anywhere by streaming game frames as video from servers to clients. However, frame jitter from bandwidth and delay variability can substantially degrade the user experience. While traditional streaming systems use playout buffer to smooth over frame variability, playout buffer algorithm in cloud-based game streaming remains underexplored. This thesis investigates the effectiveness of playout buffer algorithms for cloud gaming through the implementation of a custom game streaming platform and an automated Flappy Bird-style game. We evaluated two buffer strategies, the E-Policy and the Queue Monitoring (QM) under controlled jitter conditions. By applying existing Quality of Experience (QoE) models, we measure how the critical metrics of delay and interrupts influence users' perceived gaming quality. Our results suggest that the E-Policy reduces playback interruptions but causes higher delay, which lowers overall QoE, especially when network jitter is high. On the other hand, the QM algorithm with higher decay values keeps a better balance between delay and playback smoothness, leading to higher combined QoE across different network conditions.

Contents

1 Introduction **1**

2 Background **3**

2.1 Cloud-based game Streaming Architecture . . . 3

2.2 Latency . . . 4

2.3 Frame Jitter . . . 4

2.4 Playout Buffer . . . 4

2.4.1 E-Policy . . . 5

2.4.2 Queue Monitoring . . . 6

3 Related Work **7**

3.1 Overview of Cloud-based Game Streaming Services . . . 7

3.2 Effects of Latency . . . 7

3.3 Jitter Buffer Algorithms . . . 8

3.4 Jitter Buffer Algorithms in Cloud-based Game Streaming . . . 8

4 Methodology **10**

4.1 Design and Development of Streaming Game System . . . 11

4.1.1 Game Server . . . 11

4.1.2 Client . . . 12

4.2 Implement Jitter Buffer Algorithms . . . 12

4.2.1 E-Policy . . . 12

4.2.2 Queue Monitor . . . 14

4.3 Setup Testbed . . . 16

4.4 Add Jitter . . . 17

4.5 Design Experiment . . . 17

4.6 Analyze data . . . 19

5 Analysis **20**

5.1 Interrupt Magnitude and Delay . . . 20

5.2 Quality of Experience . . . 20

5.2.1 QoE on Interrupt Magnitude and Delay . . . 21

5.2.2 Combined QoE . . . 22

5.2.3 Evaluation of Combined QoE Under Baseline Delay . . . 23

5.3 Summary . . . 23

6 Future Work **25**

7 Conclusion **25**

Appendices **27**

A Combined QoE Using the Minimum of Interrupt Magnitude and Delay **27**

References **28**

List of Tables

1 Summary of Experimental Settings 18

List of Figures

1 Client-Server Communication in a Cloud-based Game Streaming 3
2 Latency 4
3 Inconsistent Playout Times of the Frames (Frame Jitter) 5
4 Playout Buffer 5
5 E-Policy Buffer Adaptation (Stone and Jeffay (2002) [1]). 6
6 Comparison of Interrupt Magnitude (IM) and Delay across jitter magnitudes for E-Policy and Queue Monitoring. 10
7 Comparison of QoE predictions based on Interrupt Magnitude (IM) and Delay for E-Policy and Queue Monitoring 11
8 Streaming Game System Architecture 12
9 Screenshot of the Game 13
10 Testbed 17
11 Frame Jitter - Interrupt Magnitude 17
12 Interrupt Magnitude (IM) and Delay for Different Policies. 20
13 Predicted QoE for Interrupt Magnitude (IM) and Delay 22
14 Combined QoE Among Delay and Interrupt Magnitude. 23
15 Combined QoE Among Delay and Interrupt Magnitude with Added Base Delay. 24
16 Combined QoE Among Delay and Interrupt Magnitude. 27

1 Introduction

Cloud-based game streaming offers a new way to play video games over the Internet. It allows people to play games on their computers, smartphones, or game consoles without buying expensive gaming equipment because the heavy-weight game processing is done in the cloud. According to recent data, 33% of committed gaming consumers and 10% of casual users [2] have used cloud-based game streaming services. It's estimated that there are 23.7 million gamers who play games in the cloud, and the number of paying cloud gamers is projected to reach 86.9 million by the end of 2025 [3].

Cloud-based game streaming services run games on remote computers and stream video and sound to the player's device over the Internet. The game frames are continuously captured and transmitted across the network, requiring a good Internet connection for low latency and high frame rates. But smoothing out a player's in-game visuals is key for a quality user experience, making the game feel like it's being played on a console or high-end PC.

Latency refers to the delay between a game frame being sent and displayed on the screen. In cloud-based game streaming, the delay is large compared to traditional setups due to data traveling between the player's device and remote servers. High latency can make games feel sluggish, especially in genres like shooters, arcade-style or racing games. In addition, fluctuations in network delay and bandwidth can interfere with the consistent and smooth delivery of game frames, resulting in frame jitter.

Frame jitter is when the game's pictures are not displayed at a consistent rate, causing jumps or pauses that make the game look less smooth. Unlike network jitter, which affects how smoothly game actions happen, frame jitter changes how smoothly the game looks on the screen. This can affect the Quality of Experience (QoE) of the player, making the game appear glitchy, breaking the feeling of immersion and making it hard to play games that need quick reactions or careful timing. In cloud-based game streaming, frame jitter can be from both the local system and the network.

To address frame jitter, streaming media players use playout buffers to mitigate the effects of delay and frame jitter. A playout buffer acts as a temporary storage mechanism, buffering incoming frames before smoothly releasing them for display on the player's device. While playout buffers have been used for video streaming such as YouTube and video conferences like Zoom, they have not been thoroughly evaluated for cloud-based game streaming. Most existing studies have focused on traditional media or interactive audio/video, where buffering-induced delay and playback interruptions are known to affect user experience [4, 5, 6]. However, cloud-based game streaming has unique QoE needs, where both quick response and smooth frame delivery

are both important. This means buffer policies that have been successful for traditional streaming media need to be revaluated for cloud-based game streaming.

In this research, we implement and explore the effects of jitter buffer algorithms under different network conditions on cloud-based game streaming QoE by evaluating key performance metrics like delay and interrupts. We design and develop a custom cloud-based game streaming platform using Node.js on the server and Unity on the client, with the game rendered and captured on the server and transmitted as individual frames to the client in real time. We developed a simple Flappy Bird-style 2D side-scrolling game for this system. An automated system was developed to enable repeatable testing without the need for player input.

We implement two jitter buffer algorithms on the client. The E-Policy [1] dequeues frames at a fixed interval after buffer is filled. This helps handle delays when there is frame jitter. The Queue Monitoring algorithm [1] adjusts playback based on buffer size. It discards frames when needed to reduce delay with a balance of fewer playback interrupts to maintain responsiveness. To simulate realistic network conditions, we introduce artificial jitter using a Raspberry Pi running NetEm. The Pi acts as a network bridge between the server and client. We run automated experiments where the game is streamed for 60 seconds per round under different jitter magnitudes and algorithm settings. In total, we collect 875 traces. For each configuration, we log important metrics such as average queue size as delay, interrupts magnitude across multiple runs to compare performance. We examine the E-policy and the Queue Monitoring performance using QoE models derived by prior researches that help to compare the trade-offs between delay and frame jitter.

The results show jitter buffer algorithms impact visual smoothness and delay in cloud-based game streaming. They also highlight the importance of adaptive buffer management for improving player Quality of Experience (QoE) under varying network conditions. Our analysis shows that while the E-Policy reduces frame jitter significantly, it increases delay. In contrast, the Queue Monitoring provides a better balance by adapting to network conditions and reducing interruptions with lower delay. These insights can guide system designers in choosing suitable and optimal buffering strategies for different gameplay needs.

The remainder of the paper is organized as follows: Chapter 2 provides background on cloud-based game streaming and jitter buffer algorithms. Chapter 3 presents prior research and their limitations. Chapter 4 describes the design of the jitter buffer algorithms and the experimental setup. Chapter 5 presents the analysis of performance metrics and examines how different buffer policies impact QoE under varying network conditions and concludes with a summary of key findings. Chapter 6 discusses current limitations and suggests directions for future work.

2 Background

The chapter provides background on cloud-based game streaming, beginning with its architecture, followed by a discussion of its advantages and limitations. The chapter then introduces key performance challenges such as latency and frame jitter, and concludes with an overview of the two jitter buffer algorithms used in this work.

2.1 Cloud-based game Streaming Architecture

Cloud-based game streaming is like playing video games on your computer or TV (Client), but instead of needing a powerful machine at home, the game runs on a powerful computer (Server) in the cloud, and the game frames are streamed over the Internet. Figure 1 shows how the client communicates with the remote server and plays games. The client sends actions such as key events to the server. The server sends frames to the client. The client receives the frames and displays them.

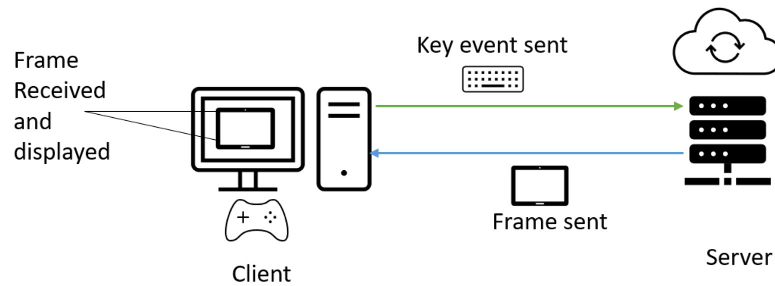


Figure 1: Client-Server Communication in a Cloud-based Game Streaming

One key advantage of cloud-based game streaming is that it removes the need for high-end hardware on the client side. Since the game is rendered in the cloud, users can play graphics-intensive games on low-powered devices such as smartphones, tablets, or older computers. It also allows access to games without long installation times or frequent updates. However, this model heavily depends on network quality. High latency, packet loss, or jitter can negatively impact the gameplay experience. Unlike local gaming, any disruption in the network can lead to input delays, lower frame quality, or stuttering visuals, which can be especially problematic for fast-paced or competitive games.

2.2 Latency

Latency is the delay experienced by the player between sending input to the server and the server sending a frame after processing the input. Aline et al. in a study, found that latency up to 300ms does not impact the players' experience if it is constant. When delay jitter was added to a latency of 200ms, however, the delay was noticed by participants more often, hindered players' ability to improve with practice, increased how often they failed to reach the goal of the game, and reduced the perceived motion quality of the character [7]. Figure 2 depicts the round-trip process of a client sending an input to change a circle's color and receiving the updated frame from the server. The time between the input and the visual update represents the response time or delay. Variable network latency can lead to frame jitter in cloud-based game streaming.

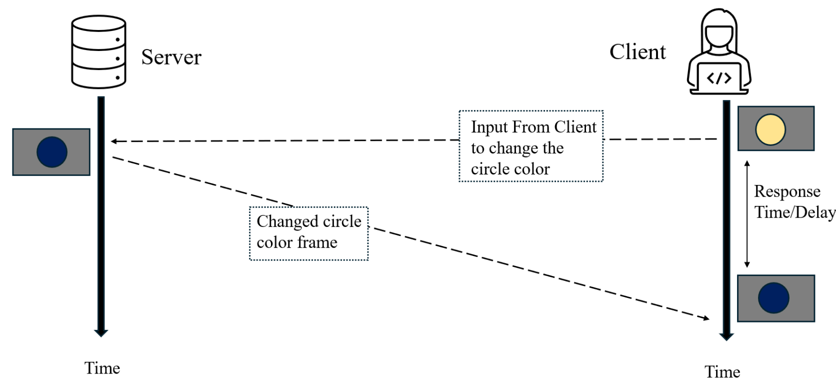


Figure 2: Latency

2.3 Frame Jitter

Frame jitter refers to the inconsistency in the playout timing of frames in a video on the client. It affects how smoothly the game renders on the player's device. Figure 3 illustrates a sequence of four frames, labeled A, B, C, and D, being transmitted with varying intervals between them, indicating the timing differences in their playout. Frame A is followed by Frame B, with a 1 millisecond (ms) gap between them. Frame B is followed by Frame C, also with a 1 ms gap. However, there is a longer gap of 4 ms between Frame C and Frame D. The larger gap before Frame D indicates an inconsistency in frame playout times at the client.

2.4 Playout Buffer

A playout buffer can smooth out frame delivery to reduce frame jitter. Frames are stored temporarily before being released at regular intervals, despite variations in arrival times. Figure 4 shows a playout buffer on a

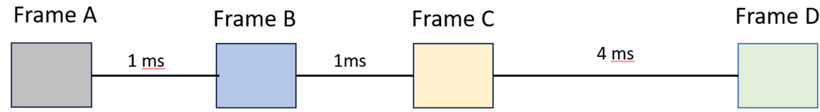


Figure 3: Inconsistent Playout Times of the Frames (Frame Jitter)

client machine. As frames arrive at the client, they enter a playout buffer. Once in the buffer, the frames are released at consistent intervals to the display. Various jitter buffer algorithms have been introduced to reduce frame jitter [1].

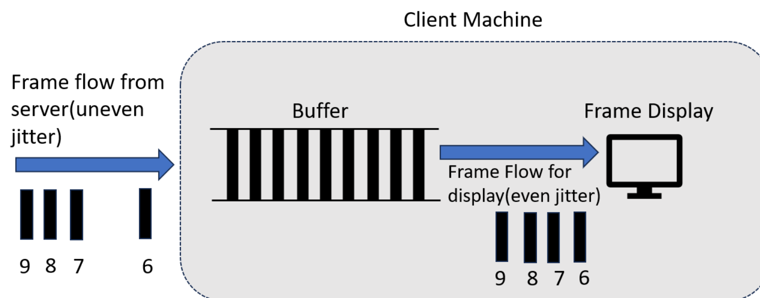
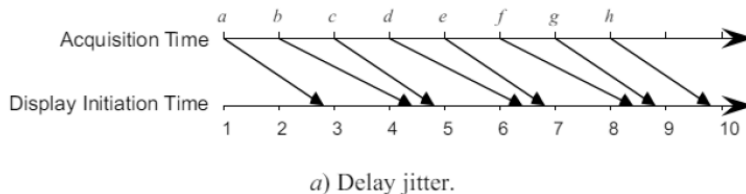


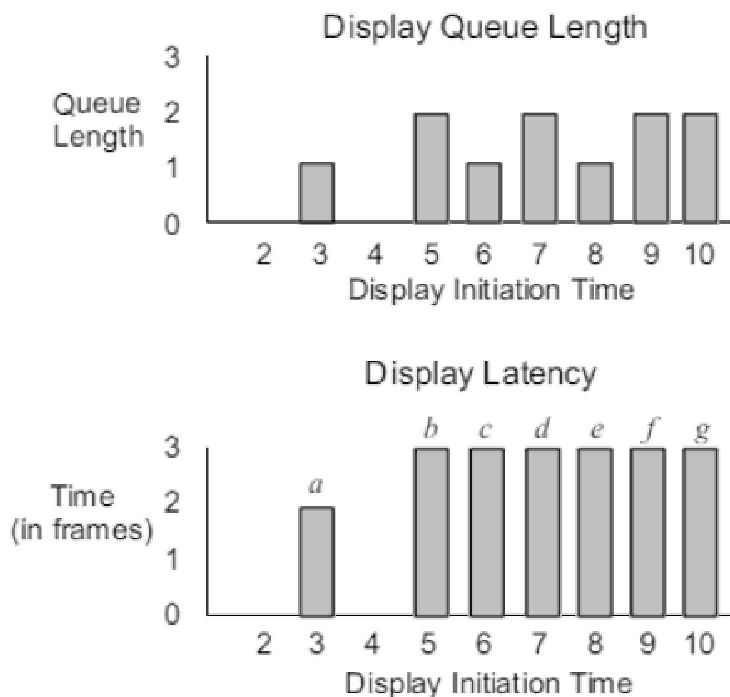
Figure 4: Playout Buffer

2.4.1 E-Policy

The E-Policy begins with a low initial display delay and increases the buffer size whenever a frame arrives late [1]. As frames are played at a steady rate, this gradually builds up a buffer to handle late frames. For example, in Figure 5 (a), frame c is delayed by more than two frame intervals, prompting the buffer to grow to three frame times for future frames. This approach avoids discarding frames and reduces playback interruptions as the buffer adjusts to network jitter. However, a key limitation is that the buffer continues to grow with each larger arrival and does not shrink, potentially leading to high buffer delay over time even if average jitter is small.



a) Delay jitter.



b) *E-Policy*

Figure 5: E-Policy Buffer Adaptation (Stone and Jeffay (2002) [1]).

2.4.2 Queue Monitoring

The Queue Monitoring is a playout buffer strategy that adjusts display latency based on the number of frames in the queue [1]. When the queue exceeds a defined threshold, the system drops the oldest frame and displays the next one to reduce delay. In our design, we play the frame faster instead of discarding. This method balances the delay and frame jitter. Its main drawback is that it only reacts when frames arrive late.

3 Related Work

This chapter reviews prior research across four major topics: Overview of cloud-based game streaming services (Section 3.1), effects of latency (Section 3.2), jitter buffer algorithms in conventional streaming (Section 3.3) and in cloud-based game streaming (Section 3.4).

3.1 Overview of Cloud-based Game Streaming Services

Shea, Liu, Ngai, et al. explore advancements in cloud gaming, emphasizing a systematic analysis of leading platforms [8]. They delve into their framework designs and conduct real-world performance measurements, unveiling challenges that hinder the widespread adoption of cloud-based game streaming.

Anmol Gupta et al. focus on cross-system gaming and issues of incompatibility [9]. They discuss the architectures and Quality of Service (QOS) for good response times and bandwidth efficiency.

These related works lay the foundation for this thesis by highlighting the challenges in cloud-based game streaming services.

3.2 Effects of Latency

Claypool and Claypool explore the impact of latency on online game performance, introducing a classification based on player control and camera view [10]. Experiments with controlled precision, deadline, and latency provide insights into the effects of latency. They also show different game actions are affected by latency differently.

Kuan-Ta Chen et al. investigate the effectiveness of cloud-based game streaming platforms with a focus on response latency [11]. Their study compares the platforms OnLive and StreamMyGame, finding that OnLive provides lower latency.

Halbhuber et al. examine how latency affects video game performance and experience across different in-game perspectives such as First-Person, Third Person, and Bird's-Eye [12]. Their research, involving 36 participants playing a shooting game under varying latency, found that latency negatively affects performance and experience regardless of perspective, with Bayesian analysis showing no significant perspective-latency interaction. This suggests other factors may be more important in understanding a game's sensitivity to latency.

This thesis explore how jitter buffer management can improve the quality of experience of a player in cloud-based game streaming by mitigating the effects of latency.

3.3 Jitter Buffer Algorithms

Yusuf Ciner et al. propose a jitter buffer management algorithm for Voice over IP in WebRTC. They detail WebRTC's core concepts, investigate behavior under network conditions with packet bursts, and propose an alternative approach to the default algorithm. The proposed strategy minimizes packet discards during packet bursts, enhancing voice quality for users, as objectively evaluated using ITU-T Rec. P.863 [13].

Stone et al. examine strategies for managing delay jitter in computer-based conferencing to ensure smooth audio and video playback [1]. They introduce Queue Monitoring, which dynamically adjusts display latency to balance quick display with gap avoidance. Tested against other policies, Queue Monitoring is often better, especially in challenging network conditions, suggesting it could enhance video conference quality.

Zhang et al.'s paper examines jitter management algorithms for real-time applications, particularly those using prediction based on past packet arrivals to optimize buffer settings and handle variable network traffic [14]. The study uses simulations to evaluate these algorithms, comparing them with other playout buffer algorithms, focusing on the balance between latency, gap probability, and packet discard rates to maintain quality of service.

Although previous works have applied jitter buffer algorithms to real-time gaming communication, these algorithms have not yet been applied to cloud-based game streaming.

3.4 Jitter Buffer Algorithms in Cloud-based Game Streaming

Suznjevic, Slivar, and Skorin-Kapov conducted a detailed study on the adaptation behavior of NVIDIA GeForce NOW under different network conditions [15]. Their work analyzed how the platform adjusts resolution, framerate, and bitrate in response to latency, jitter, packet loss, and bandwidth shaping. They combined network measurements with a subjective user study to evaluate how these adaptations affect player Quality of Experience (QoE). The findings showed that while the system adapts aggressively to latency changes, it does not respond effectively to packet loss, and recovery behaviors vary depending on the video content. This work highlights the importance of tuning adaptation strategies based on both gameplay context and measured network conditions, providing a baseline for evaluating commercial solutions.

Rossi et al. investigated the effects of network impairments on mobile cloud gaming (MCG) by conducting a

subjective QoE assessment using CS:GO streamed to smartphones [16]. They explored how round-trip time (RTT), packet loss, and different patterns of jitter (bursty and random) influence user ratings across 29 test conditions. Their study found that high RTT (greater than 100 ms) and bursty jitter have a strong negative impact on user-perceived quality, while random jitter had minimal effect. This work is particularly relevant as it addresses cloud gaming over mobile networks and offers insights into how specific network degradations affect gameplay perception.

This research focused specifically on how playout buffers should be designed for cloud gaming. Much of the existing work applies ideas from video streaming without considering the unique timing needs of interactive games. There is a need to evaluate how buffering policies perform when both low latency and low interrupts are required.

Our study addresses this by using an experiment-driven analysis to test different jitter buffer algorithms under controlled network conditions. We use cloud-based game streaming QoE metrics to measure how each policy handles the trade-off between delay and frame jitter.

4 Methodology

This chapter presents the hypothesis and pipeline of our methodology used to conduct our experiment. To enhance the quality of experience in cloud-based game streaming, we implemented jitter buffer algorithms to balance the trade-off between increased latency and reduced frame jitter. We developed a simple 2D game and a client-server pipeline to stream game video frames over a network in real time. To simulate different network conditions, we introduced controlled jitter via a Raspberry Pi router between the client and the server. On the client side, we applied two jitter buffer algorithms to reduce frame jitter. Finally, an automated framework was scripted to run the game client with different settings while collecting performance metrics.

Hypothesis 1

The E-Policy will result in lower interrupt magnitude compared to Queue Monitoring under high jitter. But Queue Monitoring will result in lower delay compared to the E-Policy across varying jitter conditions.

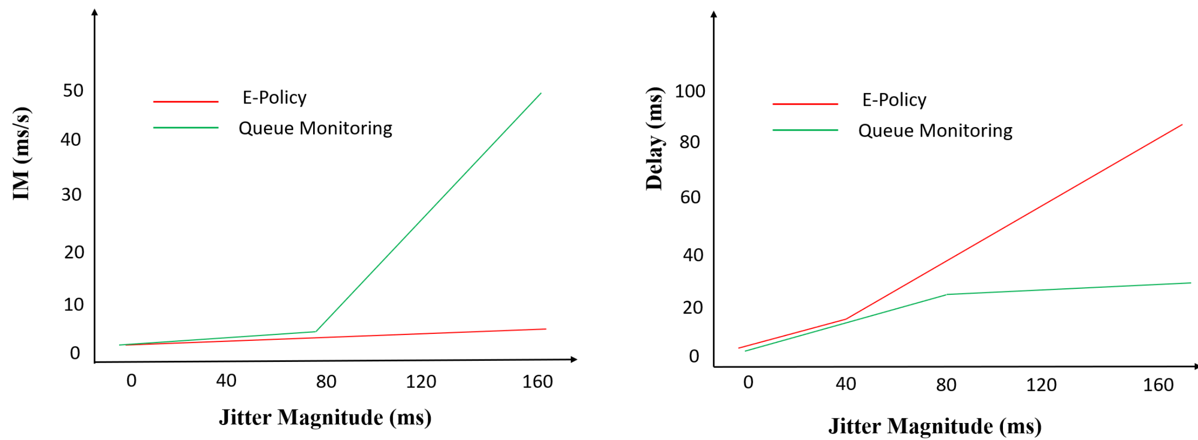


Figure 6: Comparison of Interrupt Magnitude (IM) and Delay across jitter magnitudes for E-Policy and Queue Monitoring.

Figure 6 depicts the hypothesis that under high jitter (left), the E-Policy can reduce playback interruptions compared to Queue Monitoring. However, the figure on the right shows that Queue Monitoring keeps delay lower across all jitter levels.

Hypothesis 2

The E-Policy keeps QoE for interrupt magnitude (IM) stable and high, while Queue Monitoring drops as jitter increases. On the other hand, E-Policy shows lower QoE for delay than Queue Monitoring.

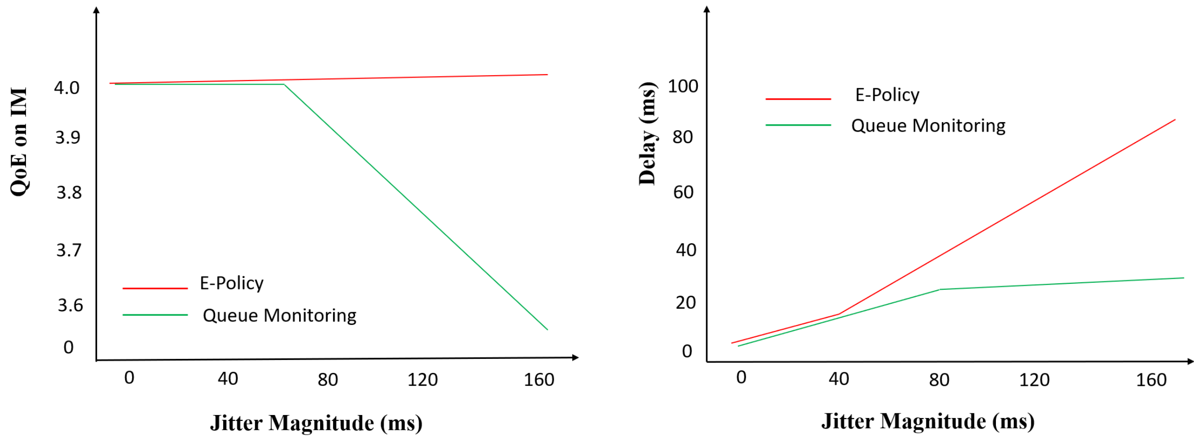


Figure 7: Comparison of QoE predictions based on Interrupt Magnitude (IM) and Delay for E-Policy and Queue Monitoring

Figure 7 suggests the hypothesis that E-Policy maintains more stable and higher QoE when evaluated using Interrupt Magnitude (left), even as jitter increases. However, the figure on the right shows that Queue Monitoring achieves better QoE for delay under medium to high jitter levels.

4.1 Design and Development of Streaming Game System

This section describes the cloud game streaming server and client setup, the real-time frame streaming process, and how the system was built to test jitter buffer algorithms.

4.1.1 Game Server

Figure 8 shows how the cloud game streaming system is structured. The game server was built using A (Node.js) and the B (Phaser3) framework [17]. Phaser.js is a lightweight, 2D game framework commonly used to create HTML5 games for desktop Pcs and mobile devices. It supports fast, real-time rendering, which is essential for maintaining a high frame rate (60 f/s) during streaming. The server hosted a simplified version of Flappy Bird designed specifically for automated testing. The bird continuously floated without collisions or scoring to run automatically without user input. The game ran in a browser, with a canvas size of 800x600 pixels. To capture the game frame, the server took a screenshot every 16.666 ms (60 f/s) and sent the frame to the C (client) in real time via Socket.io that uses WebSockets in the background. The screenshots were saved as JPEG images and sent to a Node.JS server from Phaser with an API. The images were converted to Base64 URLs to facilitate smooth, real-time transmission.

WebSockets were used for real-time communication between the client and the server because they support fast, two-way data transfer, which helps keep frame streaming smooth. TCP is used as networking protocol for socket communication between the server and client. While cloud-based game streaming systems often prefer UDP, our setup ran over a local area network (LAN) with no packet loss. Under these conditions, TCP and UDP behave similarly in terms of performance making TCP a suitable and simpler choice for this experiment.

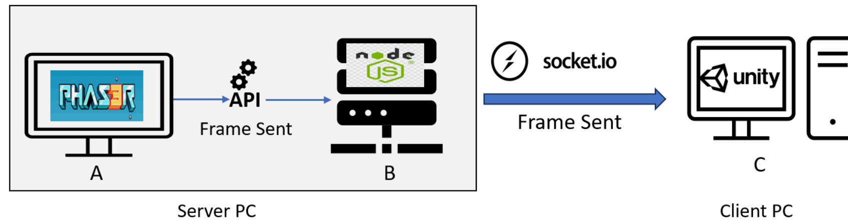


Figure 8: Streaming Game System Architecture

4.1.2 Client

The client was developed using Unity, which supports real-time communication through the Socket.io library. The server sends frames to the Unity client in the form of a Base64 URL. Upon receiving each frame, the client converts the Base64 URL back into an image and displays it using Unity's main camera.

To maintain a consistent frame rate of 60 f/s, the client displays each frame at a fixed interval of 16.667 ms. Upon receiving a frame, the client calculates the running average of the interval between consecutive frame arrivals. This approach helps maintain smooth playback despite variations in frame arrival times.

4.2 Implement Jitter Buffer Algorithms

Two jitter buffer algorithms were implemented in the client side, the E-Policy or Expansion Policy and Queue Monitoring [1]. The E-Policy was developed and tested first to establish a baseline.

4.2.1 E-Policy

First the E-Policy is designed to expand the buffer to accommodate jitter, even at the cost of increased latency. The E-Policy works by storing frames in a buffer before sending them for display as shown in Algorithm 1. Frames are dequeued at a regular interval based on a running average of interframe times.

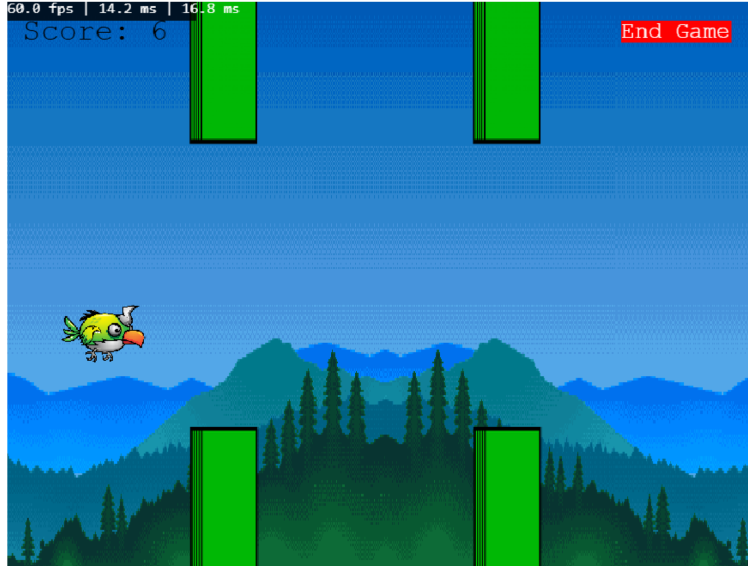


Figure 9: Screenshot of the Game

This average is calculated using the total interframe time divided by the number of frames, updated after receiving each frame. To ensure time accuracy, interframe times are recorded in nanoseconds.

The average is adjusted after each run by comparing the expected sleep time with the actual sleep time shown in Algorithm 2. Before displaying a frame, the system checks if the buffer is ready to dequeue. If the buffer is still filling up, it waits (line 16); otherwise, it proceeds to dequeue and display frames. The difference between the start and end time of the sleep is calculated and used to update the next sleep interval. This correction is applied after each frame is rendered to account for any additional overhead introduced by the rendering engine. This way, any extra delay from the previous frame is accounted for, helping maintain a consistent frame rate.

In Unity, extra work like rendering and engine updates happens after main logic of each game loop. To measure full frame time, including this extra cost, sleep adjustment is done at the end in the LateUpdate() function. This gives more accurate timing by including both dequeue, display, and Unity's internal overhead.

Algorithm 1 Frame Enqueue

```
1: while receiving frames do
2:     arrival_time = now()
3:     interframe = arrival_time - last_frame_time
4:     running_average = interframe_sum / frame_count
5:     last_frame_time = arrival_time
6:     buffer.enqueue(frame)
7: end while
```

Algorithm 2 Frame Dequeue with Post-Display Time Adjustment

```
1: while true do
2:     if buffer is ready then
3:         dequeue and display frame
4:         dequeue_time = time_since(lastDequeue_timestamp)
5:         to_sleep = running_average - dequeue_time - sleep_difference
6:         if to_sleep > 0 then
7:             start_sleep = current_time()
8:             sleep(to_sleep)
9:             actual_sleep = time_since(start_sleep)
10:            sleep_difference = actual_sleep - to_sleep
11:        else
12:            sleep_difference = 0
13:        end if
14:        last_timestamp = current_time()
15:    else
16:        wait for buffer to fill
17:    end if
18: end while
```

4.2.2 Queue Monitor

Next, the Queue Monitoring (QM) policy was implemented. Based on the observed network conditions, QM adjusts the buffer size to strike a balance between latency and frame jitter. The QM policy is a called "clawback" mechanism. This policy uses a dynamic approach where the buffer size grows when frames arrive

late, but can be reduced if the size remains large for too long. Each buffer position has a threshold, measured in frame times, which determines when to decrease the buffer size. The threshold is set using a base value for a small buffer as base length (like size 2) and a decay factor that makes the threshold smaller as the buffer size increases. This means that larger buffers are more likely to drop frames to avoid high delay, while a smaller buffer is kept to ensure smooth playback.

The enqueue frame algorithm is similar to that in the E-Policy. However, the dequeue mechanism varies from the E-Policy. In the QM, before dequeue and display frame QM does a thresholding operation.To implement this, the system keeps an array of counters that track the duration each frame stays in the buffer as outlined in Algorithm 3. If the buffer size changes (specifically increases), we dynamically add new counters to the array along with the corresponding threshold values to accommodate the new size. These counters increase with each frame. When any counter goes beyond its threshold, all counters are reset, and the oldest frame is discarded [2]. In our case, we dequeue and display the frame faster. To prevent excessive speeding up, we ensure that only one frame is played faster at a time when a threshold is exceeded. This adaptive method helps maintain the buffer within optimal limits, reacting to changes in network conditions. For our tests, we set the decay factor to 1.5, 2, 3 ,5 and used thresholds of 600 frame times.

The enqueue frame algorithm is similar to that in the E-Policy. However, the dequeue mechanism varies from the E-Policy. In the QM, before dequeue and display frame QM does a thresholding operation.To implement this, the system keeps an array of counters where each buffer position has a value that tracks how long a frame has been stored, as outlined in Algorithm 3. If the buffer size changes (specifically increases), we dynamically add new counters to the array along with the corresponding threshold values to accommodate the new size. These counters increase with each frame. Traditionally, when any counter goes beyond its threshold, all counters are reset, and the oldest frame is discarded. In our case, we do not discard rather dequeue and display the frame faster. To prevent excessive speeding up, we ensure that only one frame is played faster at a time when a threshold is exceeded. This adaptive method helps maintain a buffer that reacts to changes in network conditions. For our tests, we set the decay factor to 1.5, 2, 3, 5 and used thresholds of 600 frame times.

Algorithm 3 Queue Monitoring: Thrersholding Operation

```
1: function Thresholding(current_buffer.size)
2: ensure counter and threshold lists match current_buffer.size
3: for each i from 2 to current_buffer.size - 1 do
4:     counters[i] += 1
5: end for
6: for each i not in [2, current_buffer.size - 1] do
7:     counters[i] = 0
8: end for
9: for each i from base_length to current_buffer.size - 1 do
10:    if counters[i] is greater than thresholds[i] then
11:        reset all counters
12:        display(buffer.dequeue()) {Play one frame faster}
13:        return
14:    end if
15: end for
16: end function
```

4.3 Setup Testbed

We conducted our experiments in Fuller Lab 316 to maintain controlled jitter conditions. The lab had high-performance computers to make sure system processing power did not cause frame jitter. This helped us focus on network-induced jitter. The setup of testbed is shown in Figure 10. The game client was an Intel Core i7 8700 CPUs, NVIDIA GTX 1080 GPUs and 64 GB of RAM displaying on a 60 Hz Dell monitor with a resolution of 1920x1080. The server PC had the same specs and sent game frames to the client. Both machines were connected on the same LAN using a gigabit switch. A Raspberry Pi 4 was placed between the client and server and configured to act as a router [18]. The Pi had a 1.5 GHz quad-core CPU, 8 GB RAM, and ran Ubuntu 20.04 with Linux kernel 5.4 using the NetEm tool to apply jitter to the network. For the experiment, we added four levels of network jitter conditions: low, medium, high and very high.

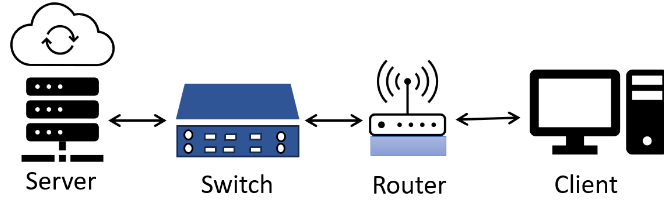


Figure 10: Testbed

4.4 Add Jitter

To create interruptions in a smooth gameplay, we introduced controlled jitter into the system.These interruptions are defined by how often they happen (frequency) and how long they last (magnitude). Figure 11 shows one such example in a game running at 60 f/s. On the graph, time is shown on the x-axis and frame display interval on the y-axis. In a perfect case, every frame would appear exactly 16.7 ms apart. However, the spikes in the plot show magnitude ranging from 50 to 150 ms.

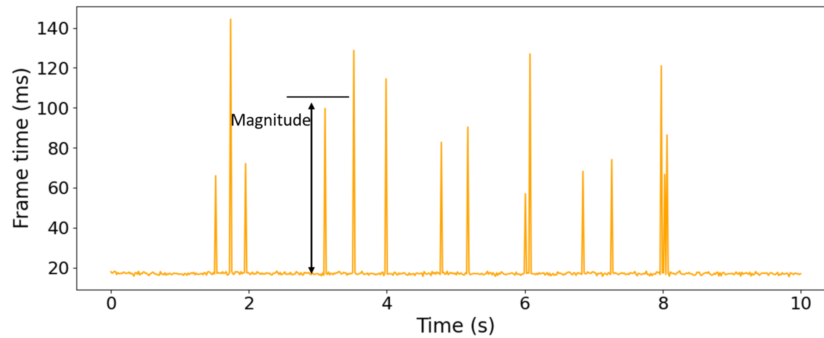


Figure 11: Frame Jitter - Interrupt Magnitude

4.5 Design Experiment

We automated the experiment using a script on the client PC. The script started the server remotely via SSH, set the desired jitter on the Raspberry Pi using NetEm, and then launched the Unity client. It also set up the round-specific configurations, including jitter level, buffer size, decay factor, and run count.Each test run lasted 60 seconds, and we performed 35 runs for each setting to ensure reliable results.

For the E-Policy, we used a static buffer size of 1, and tested 5 different jitter settings. For the Queue Monitoring algorithm, we set the threshold to 600 ms and tested 4 decay factors: 1.5, 2, 3, and 5. This created 20 combinations in total.

In total, we tested 25 different settings (5 from E-Policy and 20 from Queue Monitoring), each run 35 times, resulting in 875 test rounds. The jitter levels applied using NetEm included: No jitter (baseline), 40 ms (low), 80 ms (medium), 120 ms (high) and 160 ms (very high). This setup allowed us to systematically test how each buffering method performed under different jitter conditions. Table 1 describes the experiment settings.

Table 1: Summary of Experimental Settings

Policy	**Buffer Size**	**Decay Factor**	**Jitter Magnitude (ms)**	**Runs per Setting**
E-Policy	1	N/A	No Jitter (0) Low (40) Medium (80) High (120) Very High (160)	35
Queue Monitoring	0	1.5	No Jitter (0) Low (40) Medium (80) High (120) Very High (160)	35
	0	2	same as above	35
	0	3	same as above	35
	0	5	same as above	35
Total Settings				25 settings × 35 runs = 875

During each test run, we recorded several key metrics to analyze the impact of buffering strategies under different jitter conditions. The following metrics were logged for every round:

Average Queue Size (Delay): This metric represents the average number of frames buffered during playout. We calculate it by averaging the queue length (in frames) observed at each dequeue event. The resulting average is then multiplied by the frame time (16.67 ms) to convert it to delay in milliseconds. This helps measure how much delay the buffering introduced.

$$\text{Queue Delay} = \text{Average Queue Size} \times 16.67 \text{ ms}$$

Interrupt Magnitude (IM) (ms per second): To measures the total duration of interruptions normalized

over time, we captured IM. This indicates how long the game was stalled relative to the total playtime. We define IM as the total accumulated time (in ms) during which no frames were displayed (i.e., interruptions), divided by the total duration of the round in seconds.

$$\text{IM} = \frac{\text{Total stall time (ms)}}{\text{Total round duration (s)}}$$

Interrupt Frequency (interrupts per second): This metric captures how often frame playout was interrupted. Interrupt frequency is calculated by counting the number of times the display pipeline was stalled (i.e., no frame available when expected), and dividing that count by the total round duration in seconds. A higher frequency indicates more frequent disruptions in the streaming experience.

$$\text{IF} = \frac{\text{Number of interruptions}}{\text{Total round duration (s)}}$$

Average Frame Time (ms): This measures the average time between consecutive frame displays. This is the mean time difference between consecutive displayed frames. We compute it by logging timestamps for each displayed frame and averaging the differences between them. It helps evaluate how consistently frames are being rendered during playout.

$$\text{Average Frame Time} = \frac{1}{N-1}\sum_{i=2}^{N}(t_i - t_{i-1})$$

Where t_i is the timestamp of the i-th displayed frame, and N is the total number of displayed frames.

We include Average Frame Time and Interrupt Frequency for completeness, but they are not critical metrics for our QoE-based study.

4.6 Analyze data

A study by Xu and Claypool[19] found that interrupt frequency (IF) does not have much effect on the quality of experience (QoE) in cloud-based game streaming. Our data analysis uses objective performance metrics including playout delay and interrupt magnitude to evaluate QoE. These metrics serve as inputs to established QoE models, allowing us to assess the impact of different jitter buffer algorithms on cloud based game streaming qaulity.

5 Analysis

This chapter analyzes the interrupt magnitude and delay for each algorithm as well as the predicted quality of experience based on these factors.

5.1 Interrupt Magnitude and Delay

Figure 12 shows the Interrupt Magnitude(IM) (in milliseconds/second) on the left for the E-Policy and Queue Monitoring algorithms on the Y-axis, and the jitter magnitude (in milliseconds) on the X-axis. The values are means with 95% confidence intervals. The E-Policy performs similarly across all jitter magnitudes with minimal interrupt magnitude. The queue monitor has low interrupt magnitudes for jitter levels up to 80 ms, but the interrupt magnitude increases for higher jitter values. This increase is pronounced for the algorithm with more decay since the more agressive discard results in more interrupts.

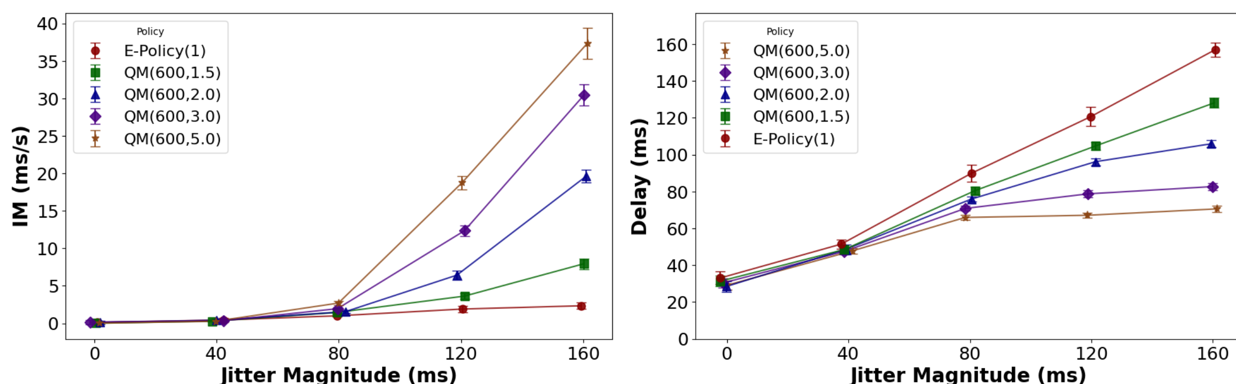


Figure 12: Interrupt Magnitude (IM) and Delay for Different Policies.

Figure 12 on the right shows the same graph as IM, but with delay in milliseconds on the Y-axis instead of IM. The E-Policy shows higher delay compared to all Queue Monitoring algorithms, with approximately linear progression from 0 ms to 160 ms jitter magnitude. Among the Queue Monitoring algorithms, the one with the highest decay shows the lowest delay compared to the others, particularly for jitter magnitudes of 80 ms and above.

5.2 Quality of Experience

This subsection shows the predicted Quality of Experience (QoE) for interrupt magnitude and delay, along with the minimum QoE derived from both.

5.2.1 QoE on Interrupt Magnitude and Delay

To predict QoE for interrupt magnitude (IM), Xu and Claypool[19] developed a linear model for QoE based on a series of user studies with various game types and network environments. The model is defined as:

$$Q_{IM} = -0.004 \times IM + 4 \quad (1)$$

where Q_{IM} denotes the QoE score (on a scale from low 1 to high 5) as a function of IM (milliseconds/second). For our analysis, we use their model along with our data on IM (milliseconds/second) to predict QoE for our experiment.

A prior study by Liu et al.investigated how delay impacts quality of experience (QoE) in competitive first-person shooter games [20]. They found that QoE decreases linearly as delay increases. Using data from their combined results shown in Figure 13, we identified approximate (x, y) pairs representing delay (in milliseconds) and corresponding QoE values. Based on these points, we extracted the slope as -0.0148 and intercept as 4.76 from the linear model The resulting model is:

$$Q_{Delay} = -0.0148 \times Delay + 4.76 \quad (2)$$

For our analysis, we use that model along with our data on delay to predict QoE for our experiment where Q_{Delay} denotes the QoE score (on a scale from low 1 to high 5) as a function of Delay.

Figure 13 shows the QoE for IM on the Y-axis for the E-Policy and Queue Monitoring algorithms on the left side, with the jitter magnitude (in milliseconds) on the X-axis. The values are means with 95% confidence intervals. The E-Policy maintains relatively stable QoE across all jitter magnitudes and outperforms the Queue Monitoring algorithms in terms of QoE. In contrast, the Queue Monitoring algorithm shows a higher QoE for jitter magnitudes up to 80 ms but shows a decline as jitter increases. This drop in QoE is more pronounced for the variant with more decay.

Figure 13 shows the same graph on the right side, but with QoE for Delay on the Y-axis instead of interrupt magnitude. Here, E-Policy shows lower QoE compared to all Queue Monitoring algorithms, following a nearly linear trend from 0 ms to 160 ms jitter magnitude. Among the Queue Monitoring algorithms, the one with the highest decay shows a higher QoE than the others, especially for jitter magnitudes of 80 ms and above.

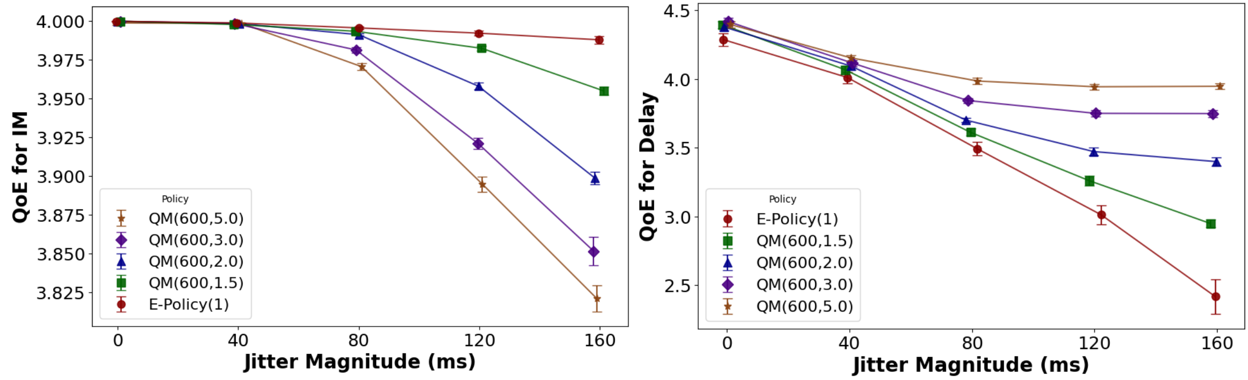


Figure 13: Predicted QoE for Interrupt Magnitude (IM) and Delay

5.2.2 Combined QoE

To calculate the combined QoE (Q_{combined}) that predicts QoE based on both delay and interrupts, for each configuration, we use the average of the QoE values derived from interrupt magnitude (Q_{IM}) and delay (Q_{Delay}):

$$Q_{\text{Combined}} = \frac{Q_{\text{IM}} + Q_{\text{Delay}}}{2} \tag{3}$$

The average value of QoE from interrupt magnitude and QoE from delay are taken as the combined QoE.

Figure 14 shows the QoE using both the interrupt magnitude and delay metrics on the Y-axis, with jitter magnitude (in milliseconds) on the X-axis. Here, Queue Monitoring with higher decay especially for the decay factor 3 and 5 demonstrates better QoE for jitter magnitudes of 80 ms and above compared to the E-Policy. The E-Policy shows better QoE at lower jitter levels, it's QoE declines at higher jitter values.

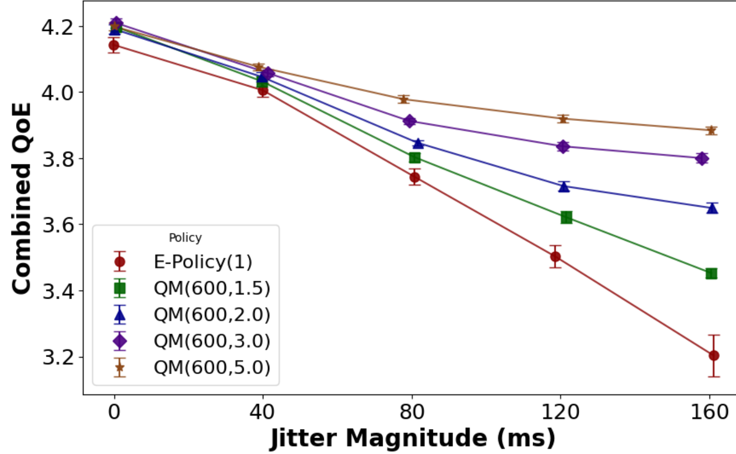


Figure 14: Combined QoE Among Delay and Interrupt Magnitude.

As discussed in appendix A, the minimum QoE, along with the average QoE, provides a more comprehensive evaluation of the system's performance.

5.2.3 Evaluation of Combined QoE Under Baseline Delay

To understand the effect of inherent network delay, we introduced a fixed base delay of 50 ms to each configuration and re-evaluated the average QoE values. This simulates scenarios where a constant delay exists in the system regardless of jitter conditions. Figure 15 depicts the results with the axes of data analysis as for Figure 14. As presented in the Figure 15, adding a base delay results in a noticeable drop in the combined QoE across all policies. The degradation is more significant for the E-Policy, which already suffers from higher delay under increased jitter. In contrast, the Queue Monitoring configurations, especially those with higher decay values, show greater resilience, maintaining relatively higher QoE under the same conditions. This highlights the importance of considering baseline delay while choosing jitter buffer algorithms in cloud-based game streaming.

5.3 Summary

In summary, the Queue Monitoring (QM) policy, especially with higher decay values, compared to the E-Policy improves QoE for cloud-based game streaming across varying jitter conditions. This improvement is mainly due to the adaptive nature of the QM algorithm. Its design helps reduce delay while mitigating playback interruptions, making it well-suited for real-time gameplay. A higher decay value like 5.0 may make the buffer more sensitive to delay growth relative to lower decay values like 2.0. In contrast, the E-Policy

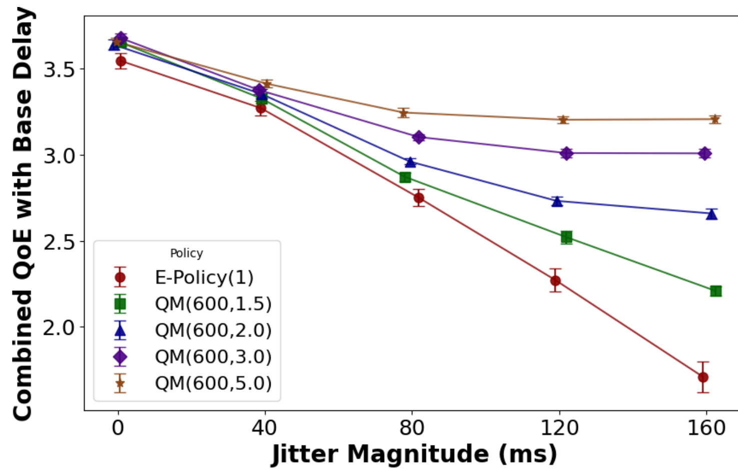


Figure 15: Combined QoE Among Delay and Interrupt Magnitude with Added Base Delay.

introduces higher average delays, which lowers QoE particularly under high jitter conditions, as observed in our simulation results. This suggests that higher decay in QM leads to better QoE in cloud-based game streaming, particularly under high jitter conditions.

6 Future Work

This work does not include user feedback on Quality of Experience (QoE) under different network conditions. An immediate extension would be to run a user study where participants play the streaming game under varying network conditions, and rate their experience after each round. This would help identify how different types of network impairments affect perceived smoothness, responsiveness, and overall enjoyment. To carry this out, the game can be modified to include post-round surveys, and participants can be recruited to assess a range of scenarios. This data would provide insight into users' subjective experience of when delay and interrupts start to negatively affect gameplay.

This research focuses only on a single game and does not examine how different genres may respond differently to delay and frame jitter. In particular, fast-paced games such as shooters or racing games are more sensitive to network impairments, where not only QoE but also player performance metrics are important. Previous studies indicate that different game genres have different levels of sensitivity to delay and jitter [21, 22]. A medium-term goal would be to create and test games from various genres such as racing, real-time strategy, and first-person shooters. For each genre, both subjective QoE (on delay and frame jitter) and objective performance measures (such as player score and input latency) would be collected. Building new gameplay modules suited to each genre and experimenting with the same jitter buffer policies would help reveal genre-specific sensitivities.

Furthermore, our analysis only evaluates existing jitter buffer algorithms and does not propose new solutions tailored for cloud-based game streaming. Future work could investigate designing a novel jitter buffer algorithm specifically for interactive streaming environments. This new approach could address the limitations observed in the Expansion Policy (E-Policy) and the Queue Monitoring (QM) such as trade-offs between delay and smoothness by dynamic adaption to the user's gameplay patterns and network feedback. Expanding the system to support additional jitter buffer policies could also enable a detailed guidance for optimizing cloud-based game streaming Quality of Experience (QoE). This goal would require a research phase, could use simulation-based evaluation, and eventual integration into the current streaming platform for final assessment.

7 Conclusion

Cloud-based game streaming offers an accessible alternative to traditional gaming hardware by delivering gameplay over the Internet. However, it faces challenges like delay and frame jitter that affect Quality of Ex-

perience (QoE). While playout buffers are common in video streaming, their role in interactive cloud gaming remains less explored, motivating this study on jitter buffer algorithms under varying network conditions.

To address this problem, a dedicated cloud-based game streaming platform was developed, capable of simulating frame jitter and delay. Two jitter buffer strategies, the Expansion Policy (E-Policy) and Queue Monitoring (QM) were implemented on the client side to smooth out frame delivery. The automated testbed was set up using two machines, one as the server and one as the client, with jitter added to the network connection ranging from low (40 ms) to very high (160 ms). The evaluation used metrics such as delay and interrupt magnitude to predict QoE using established models.

The evaluation of the system shows that buffering strategies impact gameplay differently based on network conditions. Under low jitter conditions (40 ms and below), both buffer strategies perform similarly with no significant difference in perceived QoE. However, once jitter exceeds 40 ms, the performance of the two policies begins to vary. In these high-jitter scenarios, the Queue Monitoring shows better performance, maintaining smoother gameplay with less perceived interruption. The E-Policy helps maintain smoothness by prioritizing frame completeness but adds delay, while the QM adjusts dynamically to balance delay and smoothness. Managing these trade-offs is important for preserving responsiveness without excessive latency in cloud-based game streaming. The QM with a decay factor 5 performed better than the lower decay values under network conditions with high jitter. Combining QoE insights from both policies shows that QM with higher decay value performs better than the E-Policy.

Appendices

A Combined QoE Using the Minimum of Interrupt Magnitude and Delay

To calculate the combined QoE (Q_{combined}) that predicts QoE based on both delay and interrupts, for each configuration, we use the minimum of the QoE values derived from interrupt magnitude (Q_{IM}) and delay (Q_{Delay}):

$$Q_{Combined} = \min(Q_{IM},\ Q_{Delay}) \tag{4}$$

This approach is based on the worst-case impact principle, where the factor with the greater limitation either interrupt magnitude or delay dictates the user's perceived QoE. If interrupt magnitude is high and delay is low, QoE will be restricted by interrupt magnitude, and if delay is high, it will be restricted by delay.

Figure 16 shows the QoE using both the interrupt magnitude and delay metrics on the Y-axis, with jitter magnitude (in milliseconds) on the X-axis. Here, Queue Monitoring with higher decay especially for the decay factor 3 and 5 demonstrates better QoE for jitter magnitudes of 80 ms and above compared to the E-Policy. The E-Policy shows better QoE at lower jitter levels, it's QoE declines at higher jitter values.

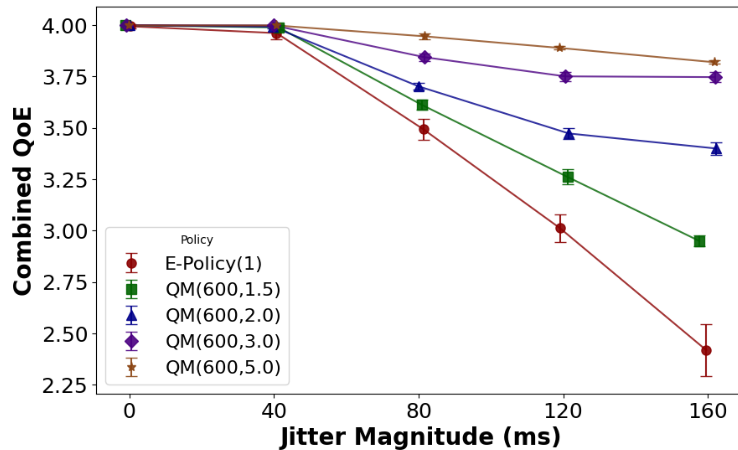


Figure 16: Combined QoE Among Delay and Interrupt Magnitude.

References

[1] D. L. Stone and K. Jeffay, "An empirical study of delay jitter management policies," *Multimedia Systems*, vol. 2, pp. 267–279, 1995.

[2] L. Rees, "Is netflix set to dominate the cloud gaming market?." https://www.pocketgamer.biz/is-netflix-set-to-dominate-the-cloud-gaming-market/#:~:text=The%20cloud%20gaming%20market%20is,service%20again%20in%20the%20future., Sept. 2023. Published by PocketGamer.biz, Last Accessed: 30th March 2025.

[3] E. McDonald, "Cloud gaming revenues to hit $2.4 billion in 2022, up +74% year-on-year; revenues will triple by 2025." https://newzoo.com/resources/blog/cloud-gaming-revenues-to-hit-2-4-billion-in-2022-up-74-year-on-year-revenues-will-triple-by-2025, Oct. 2022. Published by NewZoo, Last Accessed: 30th March 2025.

[4] M. Claypool and J. Tanner, "The effects of jitter on the perceptual quality of video," in *Proceedings of the Seventh ACM International Conference on Multimedia (Part 2) (MULTIMEDIA '99)*, (Orlando, Florida, USA), pp. 115–118, Association for Computing Machinery, 1999.

[5] L. Zhang, L. Zheng, and K. S. Ngee, "Effect of delay and delay jitter on voice/video over ip," *Computer Communications*, vol. 25, no. 9, pp. 863–873, 2002.

[6] A. Tatematsu, Y. Ishibashi, N. Fukushima, and S. Sugawara, "Qoe assessment in haptic media, sound and video transmission: Influences of network latency," in *Proceedings of the IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR 2010)*, (Vancouver, British Columbia, Canada), pp. 1–6, IEEE, 2010.

[7] A. Normoyle, G. Guerrero, and S. Jörg, "Player perception of delays and jitter in character responsiveness," in *Proceedings of the ACM Symposium on Applied Perception (SAP '14)*, (New York, NY, USA), pp. 117–124, Association for Computing Machinery, 2014.

[8] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: Architecture and performance," *IEEE Network*, vol. 27, no. 4, pp. 16–21, 2013.

[9] A. Gupta and K. Dutta, "Cloud gaming: Architecture and quality of service," *CPUH-Research Journal*, vol. 1, no. 2, pp. 19–22, 2015.

[10] M. Claypool and K. Claypool, "Latency can kill: Precision and deadline in online games," in *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems (MMSys'10)*, (New York, NY, USA), pp. 215–222, Association for Computing Machinery, 2010.

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

[12] D. Halbhuber, P. Schauhuber, V. Schwind, and N. Henze, "The effects of latency and in-game perspective on player performance and game experience," *Proceedings of the ACM on Human-Computer Interaction (HCI)*, vol. 7, no. CHI PLAY, pp. Article 424, 22 pages, 2023.

[13] Y. Cinar, P. Pocta, D. Chambers, and H. Melvin, "Improved jitter buffer management for webrtc," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 17, no. 1, p. Article 30, 2021.

[14] F. P. Zhang, O. W. W. Yang, and B. Cheng, "Performance evaluation of jitter management algorithms," in *Proceedings of the Canadian Conference on Electrical and Computer Engineering*, vol. 2, pp. 1011–1016, 2001.

[15] M. Suznjevic, I. Slivar, and L. Skorin-Kapov, “Analysis and qoe evaluation of cloud gaming service adaptation under different network conditions: The case of nvidia geforce now,” in *Proceedings of the Eighth International Conference on Quality of Multimedia Experience (QoMEX)*, (Lisbon, Portugal), pp. 1–6, IEEE, 2016.

[16] H. S. Rossi, N. Ögren, K. Mitra, I. Cotanis, C. Åhlund, and P. Johansson, “Subjective quality of experience assessment in mobile cloud games,” in *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, (Rio de Janeiro, Brazil), pp. 1918–1923, IEEE, 2022.

[17] Wikipedia contributors, “Phaser (game framework) — wikipedia, the free encyclopedia.” https://en.wikipedia.org/wiki/Phaser_(game_framework), 2024. Accessed: 2025-04-15.

[18] Wikipedia contributors, “Network emulation,” 2025. Accessed: 2025-04-10.

[19] X. Xu and M. Claypool, “User study-based models of game player quality of experience with frame display time variation,” in *Proceedings of the ACM Conference*, pp. 210–220, 2024.

[20] S. Liu, M. Claypool, A. Kuwahara, J. Sherman, and J. J. Scovell, “Lower is better? the effects of local latencies on competitive first-person shooter game players,” CHI ’21, (New York, NY, USA), Association for Computing Machinery, 2021.

[21] M. Claypool and K. Claypool, “Latency and player actions in online games,” *Communications of the ACM*, vol. 49, no. 11, pp. 40–45, 2006.

[22] S. Schmidt, S. Zadtootaghaj, and S. Möller, “Towards the delay sensitivity of games: There is more than genres,” in *2017 Ninth International Conference on Quality of Multimedia Experience (QoMEX)*, pp. 1–6, IEEE, 2017.