LAG COMPENSATION IN CLOUD GAMING

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ALI OSMAN KUTLUHAN BOSTAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2022
Approval of the thesis:

LAG COMPENSATION IN CLOUD GAMING

submitted by ALİ OSMAN KUTLUHAN BOSTAN in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University by,

Prof. Dr. Halil Kalpçilar
Dean, Graduate School of Natural and Applied Sciences

Prof. Dr. İ lkay Ulusoy
Head of Department, Electrical and Electronics Engineering

Prof. Dr. Gözde Bozdağ Akar
Supervisor, Electrical and Electronics Engineering, METU

Examining Committee Members:

Prof. Dr. Ece Güran Schmidt
Electrical and Electronics Engineering, METU

Prof. Dr. Gözde Bozdağ Akar
Electrical and Electronics Engineering, METU

Prof. Dr. Cüneyt F. Bazlamaççı
Computer Engineering, Izmir Institute of Technology

Assist. Prof. Dr. Pelin Angın
Computer Engineering, METU

Assist. Prof. Dr. Serkan Sarıtaş
Electrical and Electronics Engineering, METU

Date: 11.02.2022
I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Ali Osman Kutluhan Bostan

Signature: 

iv
ABSTRACT

LAG COMPENSATION IN CLOUD GAMING

Bostan, Ali Osman Kutluhan
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Gözde Bozdağ Akar

February 2022, 76 pages

In cloud gaming, computation intensive game process runs at cloud server, then rendered scenes are encoded and sent to a tiny client as a video stream. The tiny client then decodes and presents game scene to the player and sends player input to the cloud server. As cloud gaming introduces network latency from cloud server to the tiny client, this latency should be compensated to satisfy player’s Quality of Experience (QoE). In traditional multiplayer gaming setting, round-trip-time (RTT) latency from server to client is mitigated with a technique called lag compensation by using a server that, upon receiving the input, rolls back game world by RTT and makes decision over that instant. But when a player is connected to a game via cloud server, since game instance runs in cloud server, and game server behaves it as player’s input device, traditional lag compensation approach does not work. In this thesis, a method is proposed to correct that behavior by mitigating added latency from cloud server to tiny client by firstly exposing tiny client to cloud server latency to the game server and secondly sending tiny client shooting events to both game server and cloud server simultaneously. To evaluate the performance of our method, experiments are conducted with 20 players and their mean opinion scores (MOS) and hit accuracies are used. Proposed method shows significant accuracy and MOS improvement com-
pared to traditional approach.

Source codes and experiments can be accessed from:

https://drive.google.com/drive/folders/1ur4am1SnY8HJpYUpeYg6yMswctNpYS0Y?usp=sharing

Keywords: cloud gaming, latency compensation, multiplayer gaming, quality of experience
ÖZ

BULUT OYUNLARINDA GECİKME TELAFİSİ

Bostan, Ali Osman Kutluhan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Gözde Bozdagı Akar

Şubat 2022, 76 sayfa

Bulut oyunlarında, yoğun hesaplama gerektiren oyun işlemi bulut sunucusunda çalıştırılır, oluşturulunan sahneler kodlanır ve video akışı olarak bir küçük istemciye gönderilir, küçük istemci ise video akışını çözerek oyun sahnesini oyuncuya sunar ve oyuncu girdisini bulut sunucusuna aktarır. Bulut oyunları küçük istemci ve bulut sunucusu arasında ek bir ağ gecikmesi oluşturduğundan, oyuncu deneyimi kalitesini sağlamak için bu gecikme telafi edilmelidir. Geleneksel çok oyunculu oyun ortamında, sunucudan istemciye gidiş-dönüş süresi gecikmesi, girdiyi aldıktan sonra oyun dünyasını gidiş-dönüş süresi kadar geri alan ve geriye alınmış oyun dünyası üzerinde karar veren bir sunucu kullanılarak, gecikme telafisi adı verilen bir teknikle azaltılır. Bir oyuncu bulut sunucusu aracılığıyla bir oyna bağlandığında ise, oyun işlemi bulut sunucusunda çalıştığından ve oyun sunucusu bulut sunucusuna oyncunun giriş aygıtı olarak davranışından, geleneksel gecikme telafisi yaklaşımı başarısız olmaktadır. Bu çalışmada, küçük istemciden bulut sunucusuna olan gecikmeyi oyun sunucusuna aktararak ve küçük istemci girdilerini hem oyun sunucusuna hem de bulut sunucusuna aynı anda göndererek, bulut oyunlarında gecikme telafisi sağlanmıştır. Önerilen yöntem, oyuncuların ortalama görüş puanları ve isabetli vuruş oranları ile yapılan
analizlerde geleneksel yaklaşımı göre kaydadeğer bir isabet ve ortalama görüş puanı artışı göstermektedir.

Kaynak kodları ve deneylere bu adresten ulaşılabilir:
https://drive.google.com/drive/folders/1ur4am1SnY8HJpYUpeYg6yMswctNpYS0Y?usp=sharing

Anahtar Kelimeler: bulut oyun, gecikme telafisi, çok oyunculu oyun, kullanıcı deneyimi kalitesi
To my family
ACKNOWLEDGMENTS

Firstly, I would like to express my gratitude for my supervisor Prof. Dr. Gözde Bozdağ Akar, without whom this thesis would not have been possible, for her tremendous support and guidance during my thesis. I sincerely feel that I have been fortunate to have the pleasure of working with her.

I would like to thank my family and friends for their belief in me and their support. Special thanks to my friend, Can Deniz Turna, for his moral support during my thesis work.

I would like to thank committee members for their valuable time and feedback.

Finally, I also want to thank all experiment participants.
LIST OF TABLES

TABLES

Table 2.1 Interaction Delay Thresholds for Different Game Genres [7] . . . . 9
Table 5.1 Experiment sessions . . . . . . . . . . . . . . . . . . . . . . . . 43
Table 5.2 Comparison pairs for results . . . . . . . . . . . . . . . . . . . 45
Table 5.3 Statistical results, rejecting $H_0$ in favor of $H_1$ . . . . . . . . 49
Table A.1 Experiment results: Players 1-5 . . . . . . . . . . . . . . . . . . . 58
Table A.2 Experiment results: Players 6-10 . . . . . . . . . . . . . . . . . . . 59
Table A.3 Experiment results: Players 11-15 . . . . . . . . . . . . . . . . . . . 60
Table A.4 Experiment results: Players 15-20 . . . . . . . . . . . . . . . . . . . 61
# LIST OF FIGURES

## FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Traditional server rollback behavior with cloud server connection</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Two players that are connected to same game server, one via</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>local client and another with tiny client.</td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>Latencies between tiny client, cloud server and game server. Latencies are</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>assumed to be same in both directions.</td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>General architecture of a cloud gaming system.</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Basic Multiplayer Client-Server Architecture.</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Server rollback behaviour</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Experiment results in [6]</td>
<td>16</td>
</tr>
<tr>
<td>3.2</td>
<td>Different Actions and Avatar viewpoint deadline and precision requirements</td>
<td>17</td>
</tr>
<tr>
<td>3.3</td>
<td>MOS of users in a fast and a slow paced game with different</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>network conditions.</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>MOS variation with delay (with same amount in both paths of</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>connection).</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>Results of table tennis game performance.</td>
<td>20</td>
</tr>
<tr>
<td>3.6</td>
<td>Results of experiments with 400ms delay, with and without game</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>modifications.</td>
<td></td>
</tr>
</tbody>
</table>
Figure 3.7  MOS of players on Game Fairness with different network delays, with and without lag compensation in [20].

Figure 3.8  Suggested implementation point for late-warp in cloud gaming model in [16].

Figure 4.1  Illustration of inaccurate rollback. Image taken from [38].

Figure 4.2  Cloud-gaming aware server rollback behavior.

Figure 4.3  Basic approach behavior with $2t_{tc-cs} + 2t_{cs-gs} > \text{threshold}$.

Figure 4.4  Tiny client is connected to both servers.

Figure 4.5  Location and rotation of spawnPoint1.

Figure 4.6  Location and rotation of spawnPoint2.

Figure 5.1  Physical setting of test bed.

Figure 5.2  Experiment map (second player is not visible since it is obstructed by white box in this snapshot).

Figure 5.3  Player perspective in experiments.

Figure 5.4  Shot trajectories with single and constant fire.

Figure 5.5  Network delay between entities in test bed.

Figure 5.6  Players’ answers for their FPS Experience.

Figure 5.7  Players’ Mean Opinion Scores in Perceived Fairness.

Figure 5.8  Players’ Mean Opinion Scores in Perceived Accuracy.

Figure 5.9  Players’ Mean Opinion Scores in Perceived Responsiveness.

Figure 5.10  Players’ Mean Real Accuracy.

Figure A.1  Perceived fairness of players with and without CLC, with outliers.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>Cloud Gaming</td>
</tr>
<tr>
<td>fps</td>
<td>frames per second</td>
</tr>
<tr>
<td>FPS</td>
<td>First Person Shooter</td>
</tr>
<tr>
<td>GaaS</td>
<td>Gaming as a Service</td>
</tr>
<tr>
<td>MMORPG</td>
<td>Massively multiplayer online role-playing game</td>
</tr>
<tr>
<td>MOS</td>
<td>Mean Opinion Score</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>RPG</td>
<td>Role-playing game</td>
</tr>
<tr>
<td>RTT</td>
<td>Round trip time</td>
</tr>
<tr>
<td>$t_{tc-cs}$</td>
<td>Tiny client-cloud server latency</td>
</tr>
<tr>
<td>$t_{cs-gs}$</td>
<td>Cloud server-game server latency</td>
</tr>
<tr>
<td>$t_{tc-gs}$</td>
<td>Tiny client-game server latency</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

In the recent years Cloud Gaming or Gaming as a Service (GaaS) has gained a great popularity and interest as multiple key players in the industry either launching or working on their platforms. To name a few, Google Stadia [10], Nvidia GeForce Now [27] and Sony PlayStation Now [34] are already launched platforms. Upcoming ones are Amazon’s Luna [2] (currently in early access stage) and Microsoft’s Xbox Cloud Gaming [24].

In cloud gaming, game instances are run and scenes are rendered in cloud server. Then, encoded scenes are sent to tiny client as a video stream. On the opposite end of the connection, a tiny client decodes and presents the game scene to the player, then handles player input and sends it to cloud server. This presents an immense convenience to users since they do not need to purchase high-end hardware to play the computation intensive games. Users only require a hardware that is capable of decoding video streams and now, nearly all devices are capable of video decoding in our multimedia intensive world.

As cloud gaming introduces network latency from cloud server to the tiny client, from the beginning to now, tight latency constraints, that is inherent in the nature of gaming, has been an obstacle for a successful cloud gaming platform [33]. Many of the cloud gaming platforms in the beginning of 2010’s, such as OnLive, has gone out of business because of sub-optimal user experience, one of the prevalent reasons being high latency. Therefore latency compensation is essential for success for a cloud gaming provider.

Compared to other genres, player enjoyment and performance is crucially affected
by increasing latency in fast paced First-Person-Shooter (FPS) games. It is found that players can perceive latency if RTT is 100ms, and above 500ms of RTT, the game becomes unplayable [7]. Because of that, FPS games require tighter latency constraints, compared to other game genres, for better player quality of experience.

To mitigate effects of latency in FPS games, in traditional multiplayer gaming setting, round-trip-time (RTT) latency from server to client is mitigated with a technique called lag compensation [3]. In lag compensation an authoritative server records a snapshot of entities in a game world buffer. Upon receiving the shooting event from a client, it then rolls back game world by RTT, makes decision over that instant, and informs all clients accordingly.

In cloud gaming, game client runs in cloud server and the player input device is tiny client. However, traditional game servers consider game client running in cloud server as a local game client and therefore behaves cloud server as it is player’s input device. So, server only considers game server to cloud server network latency for a rollback. Therefore traditional lag compensation approach does not work as it does not consider added network latency from tiny client to the cloud server. This presents a disadvantage to the players that are connected to game server via a cloud server. An example of that behavior is given at Figure 1.1. For simplicity, processing latency is ignored, all connections are considered to have 50ms of end-to-end latency, and player at local client is changing its position with (x=1,y=0)/50ms velocity. In the figure, A is playing on a traditional local client and B is playing on a tiny client. B is missing the shot because of incorrect rollback. Connections are visualized in Figure 1.2.

Literature on latency compensation in cloud gaming is mostly divided to either world alteration ([5], [30]) and visual warping/speculation ([18], [16]). However, they are mainly focused on single player gaming. Closest work to ours is [20], but in this work, authors are aiming to achieve fairness between players by equalizing latencies of players via adding artificial delay to players with lesser latency. In this thesis, a penalty of such is not applied for low latency players, and latency compensation for cloud gaming is achieved by modifying server side lag compensation and rollback.

In this thesis, we first implemented a basic approach that also exposes tiny client-
cloud server latency \((t_{tc-cs})\) to the game server. It does so by making cloud server to report cloud server-game server latency \((t_{cs-gs})\) with \(t_{tc-cs}\) added to it. Therefore, traditional server can predict tiny client-game server latency \((t_{tc-gs})\) correctly. Latencies are shown in Figure 1.3.

Secondly, since game server decision is favoring the shooter, if the shooter’s latency is high enough, lag compensation causes a shot decision to be made, even if aimed player has taken cover (referred as shot-around-corner (SAC)). Therefore in some games, after a latency threshold, lag compensation is not applied to client events to reduce SAC inconsistencies [15]. Since we report real \(t_{tc-gs}\), if \(t_{cs-gs} \cdot t_{tc-cs}\) exceeds SAC threshold, the basic approach may backfire with the player not being lag compensated at all. Therefore in our implementation we follow basic approach until RTT latency exceeds 250ms (latency limit that is found to be beneficial to mitigate SAC [15]). After that, we investigate sending client events to both game server and cloud server simultaneously. In this approach, upon receiving shooting event tiny client sends event updates to both cloud and game server at the same time. With that, \(t_{cs-gs}\) at event latching to game server is eliminated.

### 1.1 Contributions and Novelties

Our contributions are as follows:

- While the literature on latency compensation in traditional multiplayer servers is rich, to our knowledge our study is the first one that considers cloud gaming in multiplayer gaming lag compensation with rollback mechanism.

- In our study we have proposed a method to decrease the response time of the multiplayer server via sending shooting events to both cloud and multiplayer server simultaneously.
Figure 1.1: Traditional server rollback behavior with cloud server connection
1.2 The Outline of the Thesis

In Chapter 2, a background information on cloud gaming and traditional server-client multiplayer game architecture is given. This chapter discusses architecture of cloud gaming platforms, its advantages and disadvantages compared to traditional gaming, and effect of latency on cloud gaming experience. Finally latency compensation techniques in traditional server-client multiplayer game architecture is given.

In Chapter 3, a literature review regarding adverse effects of latency on user Quality of Experience and latency mitigation techniques are given. In the first part of the chapter effects of latency on Quality of Experience is discussed. In the second part of
the chapter, mitigation techniques in both traditional multiplayer gaming and cloud gaming are given.

In Chapter 4, our proposed method and its implementation is given in detail. This chapter also contains discussion on our decisions on which Open Source Software are chosen for modification in implementation of our proposed method.

In Chapter 5, our experiments are given. Chapter starts with experimental setting and questionnaire for participants. Experiment results for both traditional server-client architecture and our method is shared. Comparisons between the results are then investigated.

In Chapter 6, conclusions and possible future work for lag compensation in cloud gaming is given.
CHAPTER 2

BACKGROUND INFORMATION

2.1 Cloud Gaming

Cloud gaming or Gaming-as-a-Service (GaaS) is a new way of game delivery. General architecture of a cloud gaming system can be seen in Figure 2.1. In a cloud gaming system, computationally demanding game instance is running at cloud server, a streamer program captures rendered scenes and encodes it as a video stream. Video stream is sent to client via best-effort Internet. Since computationally expensive rendering is already done in cloud server, at the other end of the connection, only a device that is capable of video decoding and input handling is needed. In cloud gaming platforms, this terminal device is termed as tiny client. Tiny client presents decoded game scene to the player, and handles input from the player. User input is sent from tiny client to game server, and then applied to the game instance.

Cloud gaming introduces many advantages that can be summarized as [4]:

- For players:
  - Access to games anywhere from any device, without a need for installation in local device.
  - Ability to play with low-power low-spec devices that are only needed to be capable of video decoding.
  - Platform independence, whether it is a console game or a PC game for various operating system, end user is agnostic to the platform.

- For game developers:
Figure 2.1: General architecture of a cloud gaming system. [4]

- Consideration of only one platform for game development.
- Avoiding piracy, since software is not downloaded to client computers.
- For multiplayer games, avoiding cheating, since a gaming session starts a brand new instance that cannot be altered in cloud server.

• For cloud service providers:
  
  - Leads to new business models where cloud provider acts also a middleman in product.
  - Demand on cloud resources that are already deployed increases, and also cost-effectiveness of new localized server deployments increase.

Like live streaming platforms, raw scene to be encoded is not readily present for
encoding in cloud gaming. So unlike video-on-demand (VoD) platforms that employ HTTP Adaptive Streaming (HAS) solutions with readily available streams, cloud gaming platforms need to encode live video. Furthermore while live streaming platforms can exploit tiny buffers for scenes that are to be encoded, in cloud gaming, since the scenes that are rendered in one game clock ought to be presented with minimal delay, rendered scenes are encoded at the time they are ready. [4]

Also since at client side buffering is not applicable, and the media is less forgiving for re-buffering events, in cloud gaming platforms, UDP packets are used for data delivery over Real-time Transport Protocol (RTP) streams and RTP Control Protocol (RTCP) for session control.

A cloud gaming system must register input from player, send it to the cloud server, replay the input in cloud server, send encoded scene back, and decode it in player’s end. For an interactive experience, all of this processing and delivery should be in milliseconds. Also, interaction delay threshold changes significantly depending on the game type as it can be seen in Table 2.1. Especially in fast twitch action FPS games, where who shot first would introduce significant game result change, interaction delay of above 100ms found to be introducing noticeable disadvantages to the player. Although interaction delay for a game genre is the same for traditional gaming and cloud gaming, in traditional gaming it only presents an issue in case of multiplayer gaming. Even so, since game program runs locally in traditional gaming, delays can be hidden with client side prediction (see: Section 2.2). Whereas in cloud gaming client side prediction is not applicable since game instance is running at cloud server [33].

Table 2.1: Interaction Delay Thresholds for Different Game Genres [7]

<table>
<thead>
<tr>
<th>Game Type</th>
<th>Perspective</th>
<th>Interaction Delay Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Person Shooter (FPS)</td>
<td>First Person</td>
<td>100ms</td>
</tr>
<tr>
<td>Role Playing Game</td>
<td>Third Person</td>
<td>500ms</td>
</tr>
<tr>
<td>Real-time Strategy</td>
<td>Omnipresent</td>
<td>1000ms</td>
</tr>
</tbody>
</table>

9
2.2 Multiplayer Gaming

In the most basic multiplayer client-server architecture, game logic runs at an authoritative server and client acts as a dumb terminal. In this scenario, client samples player actions and forwards it to the server. At server side, player action is reflected to game logic and executed. Resulting from player action, list of game objects is then forwarded to client to be rendered. Architecture can be seen in Figure 2.2.

![Figure 2.2: Basic Multiplayer Client-Server Architecture. [3]](image)

**Algorithm 1** Client side process in Basic Architecture [3]

```plaintext
while TRUE do
    Sample clock to find starting time
    Sample inputs from the player
    Package inputs with simulation time and send it to the server
    Receive object list from the server
    Render objects
    Sample clock to find end time
    Set simulation time to end time - starting time
end while
```

Client and server processes can be seen respectively in Algorithm 1 and 2. The main problem with this simple architecture is, since client waits for server response to render the objects and present over the screen, if the latency is high, say about 300ms, then client cannot render any objects for 300ms. As expected, this kind of behaviour is not desirable for the player.
Algorithm 2 Server side process in Basic Architecture [3]

```plaintext
while TRUE do
    Sample clock to find starting time
    Read inputs from connected players from the network
    Execute input messages
    Simulate object using simulation time
    For each client, package and send world state and visible objects
    Sample clock to find end time
    Set simulation time to end time - starting time
end while
```

This behaviour is corrected with a technique called client side prediction, where player’s actions are performed at client side as if server accepts it all without consideration. But an authoritative server still exists to manage a shared game world and whenever a server game world packet arrives, if mispredictions occur, client jumps its world state to the server game world state. The downside with this approach is that, since a correction would occur after one round-trip-time of latency from client to server, still if latency is high, a perceptible jump of object positions and world state would occur.

Prediction in client bases the last known server state at the start of each prediction. Still as same as basic architecture player actions are packaged and sent to server. But also, until a new world state arrives from server, client keeps a record of inputs and executes them in game logic. There are two problems in this execution. First is that since a client does not know the actions of other clients, other players’ avatars (in game character) in the game is changed via interpolation, but twitch actions, such as weapon fire is hard to predict. Secondly if latency is high, already applied inputs from the player might arrive again with server updates, resulting on them being replayed, so client marks and only plays the commands that are not predicted yet.

Still since we are using an authoritative server, player actions are registered after one RTT. Therefore server receives an action that is a response last game world change update after one RTT, so the player action is still lagging server game world by one round-trip-time. So for an accurate shot, players have to lead their shots according to
their instincts and awareness on RTT and this is not optimal for player’s experience. To mitigate this latency a technique called *Lag compensation* is employed. In simple terms lag compensation is a technique that normalizes inputs of connected players to the server game world by considering the inputs in a previous state of the game world that input is generated as a response. The algorithm is summarised in Algorithm 3

Algorithm 3 Lag Compensation at server side. [3]

```
while TRUE do
    for Each connected client do
        Compute latency for the client
        Search server game world history record for the client (current server time - latency)
        At found game world entry, allow player command to execute
        if Command is a weapon fire action then
            Run ray cast test to find any entity that matches in game world entry
        end if
        Move all players to their current positions
    end for
end while
```

With lag compensation, players can aim at other players without a need for leading their shots to successfully hit the target. Lag compensation allows players to hit wherever they aimed at directly. Though it is a huge improvement regarding the shooting player, there is also a complication for the player that is being shot at. Especially if the shooting player has a significant latency, then the player that is shot receives a penalty for an action/movement that he/she done noticeably before. At Figure 2.3 server rollback behaviour is given. Player latencies to the server are given next to player name. It is assumed that B has 150ms and A has 50ms of latency to game server on both ends. For simplicity A is given to be changing its position with (x=1, y=0)/50ms velocity and computational delay is ignored. Since B has a high latency, A receives a hit at 400ms for a position that he/she was at 0ms.
Figure 2.3: Server rollback behaviour
In general FPS game play, this late hit registration is generally not noticeable for players. But in cases that the player has just taken cover and expecting not to get hit, the player might get a feeling of being "shot around corners". So for that purpose game servers generally apply a latency threshold and do not apply latency compensation to the clients with a latency above it to provide better experience for low latency players.
CHAPTER 3

LITERATURE REVIEW

In this chapter we present a comprehensive review of effects of latency on player quality of experience, and latency mitigation techniques for both cloud gaming and multiplayer gaming.

3.1 Effects of Latency on Player Quality of Experience

In [6], authors investigated effects of network quality on game playing time in a MMORPG (massive multiplayer online role playing game) game. MMORPG genre is chosen since there is a continuous interaction flow without a fixed end of game session so a user can continue gaming as long as he/she quits. Authors also indicate that MMORPG games do not have split-second decision making that is needed in FPS games, effects that are observed in their experiments will be more resounding on FPS genre and their results can be thought as a baseline for such games. Their experimental results (see: Figure 3.1) shows a clear correlation with game play time with network latency (especially after 150ms), and network delay variation.

In [8], authors had classified games by avatar (player controlled character) viewpoints, and available actions. Classification is done according to the precision required to complete an action, and deadline of which an action must be completed. According to their classification, actions with high precision and tight deadlines are most sensitive to network latencies, whereas actions with low precision and loose deadlines are forgiving for even high latencies. In their experiments, it is found that games with First Person viewpoints require stricter deadlines and higher precision, which are in turn most sensitive to the network latency (See Figure 3.2).
Authors in [14], performed a subjective test in single player cloud gaming setting on 58 participants using scenarios of differing network delays and packet losses, with three different type of games that matches taxonomy presented at [7], for omnipresent perspective (slow-pace game play) they have used Pro Evolution Soccer (a soccer game), for third person perspective (medium-pace game play) they have used Final Fantasy XIII (a role playing game), and for 1st person perspective (fast-pace game play) they have used Gran Turismo HD Concept (a racing game). Their results, as it can be seen in Figure 3.3, presented player tendency to choose the scenario with higher packet loss than the higher delay in fast-paced game, and opposite of that in medium-paced game. By conclusion, they have presented that network delay appears as a decisive factor in fast-paced games, since fast reaction times contribute to success in game play in these type of games. Also in their experiments it is indicated that mean opinion scores of players are affected crucially with network delay in fast-paced games in cloud gaming (See Figure 3.4).

3.2 Latency Mitigation Techniques

In [31], authors investigated the relationship between delay, performance and QoE. In their first experiment they had 27 participants playing Need For Speed (a third-person
Figure 3.2: Different Actions and Avatar viewpoint deadline and precision requirements.

view fast-paced racing game), and a Table Tennis game (third-person view). Performance in racing game is defined as number of laps that are completed in the given time. In table tennis game, performance is defined as the difference between player’s points and opponent’s points. They found that performance has a considerable influence on QoE aspects. The effects of player performance in QoE for table tennis game (performance definition is score difference) for higher performance players is given in Figure 3.5. In a different experiment they found that delay has direct effect on both player performance and QoE and also has a indirect effect on QoE from its effect on performance. By that result authors conducted another experiment with 25 participants investigating latency compensation with improved player performance. For experiments they assessed QoE with in-game Game Experience Questionnaire (iGEQ) [29] for 0ms, 200ms and 400ms of delay and they used an open source game named Somi (a shooter game) and adapted it for better performance with precision change by modifying size of the objects and deadline change by modifying game pace. Their results showed significant increase on player performance and in turn, player QoE (See Figure 3.6).

In [20] authors investigated lag compensation in multiplayer cloud gaming with equalizing down and uplink delays for all players to the most lagged one. They used an open-source cloud gaming platform called GamingAnywhere [13] for their experi-
Figure 3.3: MOS of users in a fast and a slow paced game with different network conditions. [14]

ments. In their experiments, they had 10 participants which are grouped into two 5 player groups. Every five player than plays the game Assault Cube (an FPS game) on a tiny client running a GamingAnywhere client connected to different a GamingAnywhere server. They have modified GamingAnywhere server to get most lagged player’s latency and apply it to both down and uplink traffic of other players, thereby automatically equalizing delays in all clients. In their experiments they applied 0 to 80ms latencies to the 5 client, and got player QoE according to players’ mean opinion score in game fairness. Since it equalizes playing field with added artificial latency to less lagged players, authors set a threshold of 100ms for latency equalization. Results are shown at [3.7]

In [16] authors applied a technique that is used in virtual reality (VR) applications named late-warp (also named as Timewarp) to mitigate latency in cloud gaming. Ba-
sically it is an use of image based rendering to re-project a view to another. In their experiments they have investigated the benefits late-warp on user accuracy in FPS gameplay, using a simple FPS game they have implemented in [35]. In their implementation, they simulate latency from cloud server by presenting rendered images with 80ms delay. They warp an image received in tiny client with the most recent inputs that are applied but not rendered at cloud server yet (see Figure 3.8). Their results show significant improvement in aiming task completion time of players.

In [19], authors proposed a novel lag compensation technique for traditional multiplayer shooter games they called "Advanced lag compensation (ALC)" to address shot-behind-cover (SBC) inconsistencies in traditional lag compensation. In ALC, after a hit decision is sent from the game server to the player that is shot at, local game client confirms or denies by whether it detects a SBC event or not. After client response three scenarios happens in server. First, if hit is confirmed in game server, ALC acts as traditional lag compensation. If hit is denied, game server checks if a SBC event occurs locally in server with the location update that is sent by confirmation message. If hit is denied correctly, then server verifies denial message. If hit is denied incorrectly, then server sends hit event to both players. They indicate that their method is advantageous over conditional lag compensation (after a latency threshold lag compensation is not applied). They conducted their experiments using
Figure 3.5: Results of table tennis game performance.[31]
(a) Player performance with 400ms delay, before and after game modifications

(b) Player QoE with 400ms delay, before and after game modifications

Figure 3.6: Results of experiments with 400ms delay, with and without game modifications.\cite{31}
Figure 3.7: MOS of players on Game Fairness with different network delays, with and without lag compensation in [20].

an FPS game they have developed and invited 12 participants for game play sessions with differing artificial delays. Their results show that ALC has provided nearly same QoE as traditional lag compensation (TLC) without penalizing less lagged players.
Figure 3.8: Suggested implementation point for late-warp in cloud gaming model in [16].
CHAPTER 4

PROPOSED METHOD

4.1 Basic Approach

Currently in multiplayer cloud gaming, cloud server and game server are separate entities that are not deployed together. Since game servers are traditionally deployed to serve local clients, they do not address latency from tiny client to cloud server in lag compensation. Therefore performing inaccurate rollback in game world for shooting events (See Figure 1.1). An illustration of inaccurate rollback in game server is given at Figure 4.1 In this figure hit boxes in red and blue correspond to last seen position of opponent player on tiny client and rendered model is wrong rollback position of opponent player in game server.

In our basic approach, we do not change any logic in game server, only changing game client and streaming server program behavior on cloud server. In streaming server program, $t_{tc-cs}$ is already calculated with RTCP traffic from streaming server program in cloud server to streaming client program in tiny client. From streaming server program we send $t_{tc-cs}$ to game client over an UDP socket in loop-back. In each game loop, current $t_{tc-cs}$ is received and inputs that are sent to the server are timestamped with current game time - $2t_{tc-cs}$. At server when the input is received, server rollbacks timestamped input by $2t_{cs-gs}$, thereby rolling back tiny client input by $2t_{tc-cs} + 2t_{cs-gs}$ and effectively performs correct behavior in Figure 4.2. We do not modify game server in this method, therefore our basic approach can be deployed on already existing game server, and correct lag compensation for multiplayer cloud gaming.
4.2 Combined Sending of Tiny Client Events to Both Cloud and Multiplayer Server

Many games employ a latency threshold for lag compensation (to not negatively effect less lagged players), and do not apply rollback above it. Although we corrected game server rollback behavior with our basic approach, since now we are reporting real RTT latency, if $t_{tc-cs}$ is high enough our basic approach may backfire with player on tiny client not being lag compensated at all (See Figure 4.3). To mitigate this, we change streaming client program in tiny client and game server program in cloud server.

In streaming client, we keep game time locally and increment it with each game loop. To achieve synchronization with the server, we also take server game time and game time-step regularly from an UDP socket and update local game time regularly to match it. If a shooting event is observed at tiny client, we send a packet to the game server containing game time and cloud server IP address. Cloud server IP address is already available from the stream connection that we have between streaming server and streaming client.

In game server, we only listen streaming client connection, if total latency ($2t_{tc-cs} + 2t_{cs-gs}$) exceeds lag compensation threshold. When a packet is received from tiny client, game server performs rollback according to game time from received packet. Cloud server IP address from the packet is used to identify the corresponding game
Figure 4.2: Cloud-gaming aware server rollback behavior
client of that shooting event. Since we cannot receive orientation and viewpoint of player avatar from the packet (since game world is not present in tiny client), we keep last orientation and viewpoint that is received from cloud server and perform hit collusion test on that. Although still not an exact behavior, we think that it is a resounding mitigation since now we perform rollback on $2t_{tc-cs} + t_{cs-gs}$ which gives us more flexibility until we hit lag compensation threshold.

Not to repeat tiny client input in game server, next shooting input from cloud server is ignored. In Figure 4.4 combined sending is illustrated.

Figure 4.3: Basic approach behavior with $2t_{tc-cs} + 2t_{cs-gs} > \text{threshold}$
4.3 Open Source Software Used in Implementation

For our implementation we needed one streamer program, one stream receiver program and a multiplayer game with rollback mechanism. Since we need to modify behavior in all of them for our implementations, we needed to find suitable candidates with a source code that is available to the public. In our implementation we have chosen stream receiver program as Moonlight [25], streamer program as Sunshine [1], and multiplayer game as FPS Sample [37]. Although in most of the literature Gaming Anywhere [13] is used as streamer/stream receiver, we found it to be not performing well, so we decided on Sunshine-Moonlight pair for our implementation.

Moonlight is a third-party open source implementation for NVIDIA Gamestream [26] client that provides remote game play with a connection to the NVIDIA Gamestream server. Because of that, it needs an NVIDIA graphics card and NVIDIA Gamestream server installed. The source code is available at [25].

Sunshine is a streaming server side implementation for Moonlight solving Moonlight’s dependency on NVIDIA products. The source code is available at [1].

FPS Sample is a multiplayer only game that is developed by Unity to show-cast their technology available at the time of Unity 2018.3 and to provide a reference for possible game implementations on it. Game does not have a single player mode and cannot be played without a connection to a server. It is a traditional FPS game with interactions that are intuitive to players that are familiar with FPS games. The source code is available at [37].

Game has two different game modes that are:

- Assault: One team is attacking and one team is defending. Attacking team wins if they capture all capture points until time runs out, defending team wins by preventing that.
- Death-match: Two teams compete for most kills. The team with the most kill score wins.

It has two different maps that are:
• Level_00: Map for development usage, has small assets and size.
• Level_01: Main map, has bigger assets and size.

For implementation, Level_00 is used as map since it has smaller assets and the computational load is smaller, and death-match is used as game mode since the investigation is done on shooting events in our experiments.

4.3.1 Why GamingAnywhere is not used

GamingAnywhere is first published in 2014. Last commit in its official repository at [12] is in Sep 2019. Open issues are not handled. Since platform core is published in 2014, dependency packages (e.g. ffmpeg) are nearly a decade old. These can be uplifted to current versions but still some methods that are used are deprecated or not efficient. After handing the dependency problems and building GA successfully, the performance was still lacking in our experiments. Firstly it is highly affected by network inconsistencies and with introducing artificial inconsistency using Clumsy [36] (a network inconsistency simulator), a ripple effect in video stream is observed at client. It has no support for hardware encoding in official repository, although some forks of it introduced it, and its software encoding was not performing well. Also issues in audio were observed frequently.

4.4 Modifications on Existing Code

4.4.1 Basic Approach

In basic approach we only need to change the streaming program and the game client program. Therefore the changes are done at Sunshine and FPS Sample client.

In Sunshine, stream control is managed via an ENetPeer [9] struct, connected client (in our case Moonlight) statistics are available in stream module of Sunshine. Sockets are managed using boost::asio::ip::udp [17]. Implementation follows Algorithm [4].

In FPS Sample client we listen gameClientPort and keep and update tiny client to
Algorithm 4 Sending Tiny Client-Cloud Server RTT latency in Sunshine

Create rttEndpoint = udp::endpoint
Set rttEndpoint.address = 127.0.0.1 and rttEndpoint.port = gameClientPort
Create rttSocket = udp::socket
Open a socket with rttSocket.open and bind it to an unused port

while !streamEnded do
    iterate control events every 150 msec
    if eventOccured then
        Get streamPeer from event
        Send streamPeer.roundTripTime via rttSocket
    end if
end while

Close rttSocket

cloud server RTT. In each game loop we calculate how many game ticks RTT corresponds to by dividing it with tickrate (1/frames-per-second) then subtract that value from packet currentTick to send FPS Sample server. Also to change appliance of our method in runtime, a variable that can be set from command line on the fly is defined as cloudRtt. Implementation follows Algorithm 5.

Algorithm 5 Receiving Tiny Client-Cloud Server RTT latency in FPS Sample Client

Open RTTReceive socket
Start RTTReceiveThread and listen gameClientPort

while gameState == playingState do
    Update client game world every client tickrate
    if cloudRtt == 1 then
        Set rtt to latest received value from RTTReceiveThread
        Calculate rttTick = rtt/1000 * tickrate
        Set playerCommand.tick = playerCommand.tick - rttTick
        Send playerCommand to FPS Sample server
    end if
end while
4.4.2 Combined sending of Tiny Client events

In combined sending we need to change the Tiny Client program and the game server program. Therefore the changes are done at Moonlight and FPS Sample server.

In Moonlight, two structs are used to contain mouse event and game time: One is gsMouseButtonPacket containing streamServerAddress and timeStream, other is gsTimeContext containing gsTick and gsTickRate. Implementation follows Algorithm 6. GSTimeReceiveThread is used inside Moonlight for server game time updates (see: Algorithm 7) and GSEventSendThread is used inside Moonlight for event sending (see: Algorithm 8).

**Algorithm 6: Sending shooting events to FPS Sample Server in Moonlight**

- Set remoteGameServer.address to game server address
- Set remoteGameServer.port to mouseButtonPort
- Open gameTimeSocket and bind to gameTimePort
- Start GSTimeReceiveThread
- Start GSEventSendThread
- **while** !streamingServerConnectionClosed **do**
  - **if** mouseButtonEventOccurred **then**
    - **if** button == leftMouseButton **then**
      - Set buttonDown to !buttonDown
    - **if** gsTimeContext.gsTick == 0 **then**
      - Send empty input to remoteGameServer to make it start sending game time updates
  - **end if**
  - **end if**
- **end if**
- **end while**
- Join and close GSTimeReceiveThread
- Join and close GSEventSendThread

In FPS Sample server, current game time and shooting events are sent to/received from Moonlight respectively. Also to change appliance of our method in runtime a variable that can be set from command line on the fly is defined as cloudRtt. The
Algorithm 7 GSTimeReceiveThread in Moonlight

while !threadIsInterrupted do
    if Packet is received at gameTimeSocket then
        Lock readMutex
        Set gsTimeContext.gsTickRate to value received from the packet
        Set gsTimeContext.gsTick to value received from the packet
        Set serverLastTime to current time
        Unlock readMutex
    end if
end while

Algorithm 8 GSEventSendThread in Moonlight

while !threadIsInterrupted do
    if buttonDown then
        Lock readMutex
        Set tickrate to gsTimeContext.gsTickRate
        Set timestamp to gsTimeContext.gsTick
        Set prevServerTime to serverLastTime
        Unlock readMutex
        Set now to current time
        Set extraTicks to (now - prevServerTime) / 1000.0 * tickrate to calculate ticks that has passed since last server update
        Set timestamp to timestamp + extraTicks
        if timeStamp - lastTimeStamp > 1 (At least one tick has passed from last sent packet) then
            Set gsMouseButtonDownPacket.streamServerAddress to already known streamingServerAddress (we have RTP connection with it.)
            Set gsMouseButtonDownPacket.timeStamp to timestamp
            Send gsMouseButtonDownPacket to remoteGameServer
            Set lastTimeStamp to timestamp
        end if
    end if
end if
threshold for lag compensation rollback, rollbackThreshold, is set to 250. To send shooting events from Tiny Client since we do not have information about player look pitch and yaw in game world, we got it from last client command so they are one game tick behind. Since we already apply shooting events from Tiny Client directly, to not replicate shooting, we ignore command packets from FPS Sample client if command has PrimaryFire button activated. Implementation follows Algorithm 9.

### 4.4.3 Experimental changes

For experiments we needed to keep player hits and misses to calculate accuracy. Also we needed to spawn players in same area and keep the area small with added walls to the map. This changes are implemented in FPS Sample server. To do this first we expanded existing PlayerState class that holds player information with hitCount, hitReceivedCount, shotCount, and missCount. Implementation of accuracy calculation follows Algorithm 10.

Implementation of spawning players at same area follows Algorithm 11. Rotation and location of spawnPoint1 and spawnPoint2 is given at Figure 4.5 and Figure 4.6 respectively.
Algorithm 9 Receiving shooting events from Moonlight in FPS Sample Server

Open EventReceive socket
Start EventReceiveThread and listen mouseButtonPort

while gameState == gameActive do
  if lastTinyClientTick != tinyClientTick then
    Get client and clientConnection from cloudClientId
    Get rtt from clientConnection.rtt
    if cloudRtt == 1 and rtt > rollbackThreshold then
      Set userCommand.command to client.latestCommand
      Set userCommand.command.buttons to Button.PrimaryFire
      Set userCommand.command.tick to tinyClientTick
      Update client.controlledEntity with userCommand.command
    end if
  end if

  if Command packet received then
    if client.id == cloudClientId and cloudRtt == 1 and rtt > rollbackThreshold and userCommand.command.button != button.PrimaryFire then
      Update client.controlledEntity with userCommand.command
    end if
  end if

  if cloudRtt == 1 then
    if shootingEventReceived at mouseButtonPort then
      Set tinyClientEndpoint to event sender
      Set tinyClientTimeStamp to gsMouseButtonPacket.timeStamp and
      Set streamServerIp to gsMouseButtonPacket.streamServerAddress
      Set cloudClientId using streamServerIp
      Every third tick send gsTickRate and gsTick to tinyClientEndpoint
    end if
  end if
end while
Algorithm 10 Accuracy calculation for experiment sessions in FPS Sample Server

\[
\text{while} \quad \text{SessionState} \neq \text{Session.Ended} \quad \text{do}
\]

\[
\text{if} \quad \text{Shooting command is received} \quad \text{then}
\]

\[
\text{Get shootingPlayer from command.owner}
\]

\[
\text{Increment shootingPlayer.shotCount by one}
\]

\[
\text{if} \quad \text{hitCollisionTest resulted in hit} \quad \text{then}
\]

\[
\text{if} \quad \text{Collided entity is a player} \quad \text{then}
\]

\[
\text{Get shotReceivingPlayer from hitCollusionOwner}
\]

\[
\text{Increment shootingPlayer.hitCount by one}
\]

\[
\text{Increment shotReceivingPlayer.hitReceivedCount by one}
\]

\[
\text{else}
\]

\[
\text{Increment shootingPlayer.missCount by one}
\]

\[
\text{end if}
\]

\[
\text{else}
\]

\[
\text{Increment shootingPlayer.missCount by one}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{end while}
\]

\[
\text{for all} \quad \text{player : players} \quad \text{do}
\]

\[
\text{Calculate accuracy by player.hitCount / player.shotCount} \times 100
\]

\[
\text{Write player.playerName, player.playerId, player.shotCount, player.hitCount,}
\]

\[
\text{player.missCount, accuracy to output file}
\]

\[
\text{end for}
\]

Algorithm 11 Player respawn for experiment sessions in FPS Sample Server

\[
\text{while} \quad \text{SessionState} \neq \text{Session.Ended} \quad \text{do}
\]

\[
\text{if} \quad \text{Player is dead} \quad \text{then}
\]

\[
\text{if} \quad \text{Player.teamIndex} == 0 \quad \text{then}
\]

\[
\text{Spawn at spawnPoint1}
\]

\[
\text{else}
\]

\[
\text{Spawn at spawnPoint2}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{end while}
\]
Figure 4.4: Tiny client is connected to both servers.
Figure 4.5: Location and rotation of spawnPoint1

Figure 4.6: Location and rotation of spawnPoint2
5.1 Experimental Setting

Test bed is composed of four PCs with Windows 10 OS, connected together with a network switch. Physical connections can be seen in Figure 5.1.

A snapshot of game world can be seen in Figure 5.2. Scores of each players and the remaining time is available for the players on the top the screen. Remaining health and ammunition is shown at bottom left and bottom right of the screen respectively. Player perspective is given at Figure 5.3. Further modifications on FPS Sample is done for experiment purposes as such:

- Players are spawned in same area with their backs turned to each other.
• A game round takes 150 seconds.

• Only one character is available (also cannot be changed inside the game) for fairness.

• Only primary weapon with point contact ammunition is available for the players, since area contact weapons produced complexity in implementation.

• After death, player re-spawns in two seconds.

Already existing behavior of scattering shots with constant fire in chosen weapon is left as is to discourage players from constant fire (See Figure 5.4).

Figure 5.2: Experiment map (second player is not visible since it is obstructed by white box in this snapshot).

5.1.1 Experiment Questions

Before the experiment only one question is asked to the players:


After the experiment three questions are asked to the players:
1. In your opinion how fair was the game? Opinion score, 1: Totally unfair - 5: Totally fair. (The same question that was asked to players in [20]).

2. How accurate do you think your shots were registered? Opinion score, 1: Totally inaccurate - 5: Totally accurate. (The same question that was asked to players in [19]).

3. How responsive do you think your shots were registered? Opinion score, 1: Unacceptably late - 5: Instantly. (The same question that was asked to players in [19]).

5.1.2 Experiment Sessions

Experiments are done with adding artificial delay in latencies shown at Figure 5.5. Each session takes 150 seconds. Players are also given a warm-up session that also takes 150 seconds. At each experiment both players are assigned as Player 1 (Tiny Client) and Player 2 (Local Client). Only Player 1 is considered in experiments. Players start with warm-up session and then play all of the session settings at Table 5.1. Sequence of sessions are randomized so that players cannot expect which delay
(a) Shots are straight with single fire.

(b) Shots are scattered with constant fire.

Figure 5.4: Shot trajectories with single and constant fire.
Table 5.1: Experiment sessions

<table>
<thead>
<tr>
<th>Session</th>
<th>( t_{tc-cs} )</th>
<th>( t_{cs-gs} )</th>
<th>( t_{lc-gs} )</th>
<th>( t_{tc-gs} )</th>
<th>Cloud Lag Compensation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30ms</td>
<td>30ms</td>
<td>30ms</td>
<td>N/A</td>
<td>off</td>
</tr>
<tr>
<td>2</td>
<td>30ms</td>
<td>30ms</td>
<td>30ms</td>
<td>N/A</td>
<td>on</td>
</tr>
<tr>
<td>3</td>
<td>60ms</td>
<td>30ms</td>
<td>30ms</td>
<td>N/A</td>
<td>off</td>
</tr>
<tr>
<td>4</td>
<td>60ms</td>
<td>30ms</td>
<td>30ms</td>
<td>N/A</td>
<td>on</td>
</tr>
<tr>
<td>5</td>
<td>100ms</td>
<td>30ms</td>
<td>30ms</td>
<td>N/A</td>
<td>off</td>
</tr>
<tr>
<td>6</td>
<td>100ms</td>
<td>30ms</td>
<td>30ms</td>
<td>30ms</td>
<td>on</td>
</tr>
</tbody>
</table>

condition they are to be tested with in the following session. In sessions only \( t_{tc-cs} \) delay is changed, \( t_{cs-gs} \), \( t_{lc-gs} \) and \( t_{tc-gs} \) are kept constant. Since modified FPS Client server starts to listen tiny client port after RTT of 250ms (latency limit that is found to be beneficial to mitigate shot-around-corner events [15]) \( t_{tc-gs} \) only becomes relevant in Session 6 where shooting events are received directly from tiny client in FPS Client server and Cloud Lag Compensation (CLC) is applied. In sessions 2 and 4, only Basic Approach (see: Section 4.1) is running in CLC since RTT latency is 120ms and 180ms respectively. In session 6 since RTT latency is 260ms Combined Sending (see: Section 4.2) becomes activated in CLC.

Figure 5.5: Network delay between entities in test bed.

After each session, after experiment questions in 5.1.1 are asked to the participants. Also at the end of each session, player hit accuracy is recorded in game server.
5.2 Results

Experiments are conducted with 20 participants (19 male one female) with an age range of 20-35. Players' subjective ratings for FPS Experience is given at Figure 5.6. Before starting, players are given instructions about the game play. Players are also instructed to score as much as possible, as it is done in real game play. Also, information about recording their real accuracy is not shared with them, since it might tamper with the results. Experiment took nearly 30 minutes for one participant in total, including 150 seconds breaks between all the sessions.

![Figure 5.6: Players’ answers for their FPS Experience.](image)

Sessions with same network latency are compared against each other. For each latency odd numbered session is without Cloud Lag Compensation (CLC), and even numbered session is with CLC. In total there is three comparisons that are given at Table 5.2.

44
Table 5.2: Comparison pairs for results

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Session Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison 1 (30ms latency)</td>
<td>Session 1 (without CLC) - Session 2 (with CLC)</td>
</tr>
<tr>
<td>Comparison 2 (60ms latency)</td>
<td>Session 3 (without CLC) - Session 4 (with CLC)</td>
</tr>
<tr>
<td>Comparison 3 (100ms latency)</td>
<td>Session 5 (without CLC) - Session 6 (with CLC)</td>
</tr>
</tbody>
</table>

For all comparisons four parameters are investigated:

- Perceived Fairness (After experiment question 1)
- Perceived Accuracy (After experiment question 2)
- Perceived Responsiveness (After experiment question 3)
- Real Accuracy

To preprocess data, outliers are indicated and eliminated. Outliers are detected as the data points which are 1.5 interquartile ranges above the upper quartile or below the lower quartile using Matlab method `isoutlier` [21]. Figures for raw data, outliers, and data without outliers are given at Appendix A.

Players’ mean opinion scores for perceived fairness are given in Figure 5.7. In all comparisons an increase in perceived fairness score is observed.

There is a similar result in perceived accuracy (see: Figure 5.8) for Comparison 1 and 2 where a clear increase in MOS is observed. However in Comparison 3 (where Combined Sending is activated in CLC), there is no clear opinion score increase.

In perceived responsiveness an increase in players’ mean opinion scores are observed for all comparisons (see: Figure 5.9).

In real accuracy a flat increase in players’ mean accuracy is observed for all comparisons (see: Figure 5.10).

For statistical results similar procedure used in [12] is also followed for results. The procedure is:
Firstly a Shapiro-Wilk [32] test is applied to understand if the data can be considered as normally distributed or not. Since Matlab did not provide Shapiro-Wilk test by default, an implementation given at [28] is used for testing.

• If data is normally distributed then a paired-samples t-test is used (Matlab function \texttt{ttest} [22]) to investigate whether the hypothesis is correct.

• If data is not normally distributed then a Wilcoxon signed-rank test is used (Matlab function \texttt{signrank} [23]) to investigate whether the hypothesis is correct.

For statistical tests following two hypotheses are used. Null hypothesis should be rejected and test hypothesis should be accepted with $p < 0.5$ for a statistically significant result:

**Null hypothesis** ($H_0$): In session pairs there is no significant increase in sessions with CLC (even sessions) compared to sessions without CLC (odd sessions).

**Test hypothesis** ($H_1$): In session pairs there is a significant increase in sessions with
Figure 5.8: Players’ Mean Opinion Scores in Perceived Accuracy

CLC (even sessions) compared to sessions without CLC (odd sessions).

The results of statistical tests are shown at Table 5.3.

5.3 Discussions

It is found that our method significantly increased real accuracy of players in all comparisons ($p < .05$). In Comparison 2, Perceived fairness and accuracy increase is statistically significant and responsiveness is close with $p$ value of 0.0625. In Comparison 1, perceived accuracy increase is statistically significant with $p$ value of 0.047.

There is a clear increase in mean opinion scores of players’ in all perceived parameters for Comparison 1 and 2. In Comparison 2, since delay is higher, the difference between CLC and traditional lag compensation could be observed by players more clearly and thus opinion score difference is more significant.

In Comparison 3, an overall increase in perceived scores are observed but it is not as
clear as in Comparison 1 and 2. Since in Comparison 3 combined sending is activated in CLC, and with that shooting event registering is decoupled in FPS Sample Client and Server. This results in visual information of shooting to be late latching to the tiny client from the FPS Sample Client, even though correct action is already applied in FPS Sample Server. In Chapter 6, possible future work for visual delay compensation is discussed.

In experiments it is found that CLC increased player QoE in perceived accuracy, fairness and responsiveness. Also since our method significantly increases player accuracy, and there are many literature that points out the relationship between accuracy and player’s quality of experience ([31], [5], [30]), application of CLC in Multiplayer Cloud Gaming will benefit player’s QoE. Moreover, most of the multiplayer FPS games have a ranking system [11], and our method will also contribute to fairness in ranking for players connecting via a cloud gaming provider, with significantly increased accuracy.
Figure 5.10: Players’ Mean Real Accuracy

Table 5.3: Statistical results, rejecting $H_0$ in favor of $H_1$

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Used Test</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceived Fairness-S2 &gt; Perceived Fairness-S1</td>
<td>Wilcoxon</td>
<td>0.21582</td>
</tr>
<tr>
<td>Perceived Fairness-S4 &gt; Perceived Fairness-S3</td>
<td>Wilcoxon</td>
<td>0.04590</td>
</tr>
<tr>
<td>Perceived Fairness-S6 &gt; Perceived Fairness-S5</td>
<td>Wilcoxon</td>
<td>0.37244</td>
</tr>
<tr>
<td>Perceived Accuracy-S2 &gt; Perceived Accuracy-S1</td>
<td>Wilcoxon</td>
<td>0.04688</td>
</tr>
<tr>
<td>Perceived Accuracy-S4 &gt; Perceived Accuracy-S3</td>
<td>Wilcoxon</td>
<td>0.01480</td>
</tr>
<tr>
<td>Perceived Accuracy-S6 &gt; Perceived Accuracy-S5</td>
<td>T-test</td>
<td>0.50000</td>
</tr>
<tr>
<td>Per. Responsiveness-S2 &gt; Per. Responsiveness-S1</td>
<td>Wilcoxon</td>
<td>0.31396</td>
</tr>
<tr>
<td>Per. Responsiveness-S4 &gt; Per. Responsiveness-S3</td>
<td>Wilcoxon</td>
<td>0.06250</td>
</tr>
<tr>
<td>Per. Responsiveness-S6 &gt; Per. Responsiveness-S5</td>
<td>Wilcoxon</td>
<td>0.14514</td>
</tr>
<tr>
<td>Accuracy-S2 &gt; Accuracy-S1</td>
<td>T-test</td>
<td>0.00004</td>
</tr>
<tr>
<td>Accuracy-S4 &gt; Accuracy-S3</td>
<td>T-test</td>
<td>0.00059</td>
</tr>
<tr>
<td>Accuracy-S6 &gt; Accuracy-S5</td>
<td>T-test</td>
<td>0.01579</td>
</tr>
</tbody>
</table>
CHAPTER 6

CONCLUSIONS

In this thesis lag compensation in cloud gaming is implemented by firstly correcting server rollback with exposing tiny client to cloud server RTT latency to game server. Then since rollback is not applied after a RTT threshold in many games, after 250ms of RTT latency a combined sending of shooting events to both cloud server and game server is considered. In all comparisons our method shown statistically significant increase in real player accuracy with $p << 0.05$. Also in opinion scores of players in fairness, accuracy and responsiveness, it is found that there is a clear increase in players’ MOS with CLC compared to traditional lag compensation. Therefore, implementing CLC in multiplayer cloud gaming will boost player’s performance and QoE, and also improve fairness in in-game ranking of players playing via a tiny client.

Appliance of CLC in multiplayer cloud gaming will benefit the player QoE without a significant network load increase since only loopback packets introduced for rollback correction. In combined sending case, again only a small UDP packet containing game world time tick and cloud server IP is introduced in Tiny Client side to send the shooting event to game server, and a small UDP packet containing game world time tick and game world time step size is introduced in game server side for synchronization in Tiny Client. Packet transmission frequencies can also be optimized according to individual game.

In future work a combined "netcode"\footnote{Term in game development for the code base that is responsible for network events.} library will be developed for indicated limitations:

- Our implementation is limited to one tiny client for now, the library will handle
all connections from multiple tiny clients and match them with their respective game clients in server side.

- In Session 6, where RTT latency exceeds 250ms, and combined sending of shooting events are applied (see: Section 4.2), since our tiny client is synchronized only with game server, in small number of shooting events, a mismatch of one shooting event is perceived. The reason behind that is client side prediction, since in client side prediction, game client applies inputs to its game logic until a game world update is received from the server, in particularly small number of events, visually either one shot rollback (remaining rounds first decreases and increases by one) or one more shot applied (visually remaining rounds decrease by one) is observed with server correction. Since game client is also synchronized regularly with the game server this visual behavior is not frequently observed. The library will synchronize time for all three entities.

- Although the logic behind the lag compensation in multiplayer servers are the same, the implementations differ from one game developer to another. A library that could be shared would minimize the integration effort on different games.

In future work image-warping techniques such as [16] will also be applied to the Tiny Client side to compensate for visual latency. Also another connection will be used to send player position and view angle from Game Client to Tiny Client, in combined sending inputs are then could be sent to Game Server by applying transformations to position and view angle with the latest input in Tiny Client side.
REFERENCES


A.1 Player answers and accuracies

Figure A.1: Perceived fairness of players with and without CLC, with outliers.
### Table A.1: Experiment results: Players 1-5

<table>
<thead>
<tr>
<th>Player</th>
<th>Experience</th>
<th>Session</th>
<th>Fairness (Q1)</th>
<th>Accurateness (Q2)</th>
<th>Responsiveness (Q3)</th>
<th>Accuracy (over 100 percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>15.17</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>16.13</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>7.04</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>12.69</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>9.43</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8.15</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>12.63</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>17.32</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>10.83</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>15.17</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>9.09</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10.27</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>12.39</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5.74</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>7.58</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5.24</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>6.42</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>16.08</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>18.66</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>11.11</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>15.27</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6.89</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>7.54</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>5</td>
<td>12.88</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>16.12</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>9.52</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>12.24</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>7.53</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>10.22</td>
</tr>
</tbody>
</table>
### Table A.2: Experiment results: Players 6-10

<table>
<thead>
<tr>
<th>Player</th>
<th>Experience</th>
<th>Session</th>
<th>Fairness (Q1)</th>
<th>Accurateness (Q2)</th>
<th>Responsiveness (Q3)</th>
<th>Accuracy (over 100 percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>11.74</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>12.22</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>10.07</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>9.52</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5.45</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>5.96</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>6.12</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>10.2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>6.51</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8.09</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>5.88</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>6.68</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>10.09</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>12.27</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>17.16</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>22.09</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>9.93</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>6</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>16.71</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>8.78</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>10.09</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>12.26</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>8.56</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>5.66</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>10.62</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>14.02</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>15.73</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>9.29</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>10.87</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7.68</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>7.84</td>
</tr>
</tbody>
</table>
Table A.3: Experiment results: Players 11-15

<table>
<thead>
<tr>
<th>Player</th>
<th>Experience</th>
<th>Session</th>
<th>Fairness (Q1)</th>
<th>Accurateness (Q2)</th>
<th>Responsiveness (Q3)</th>
<th>Accuracy (over 100 percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>10</td>
<td>11.47</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>15.33</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>12.24</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>16.19</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>13.02</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>10.74</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>12.21</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>13.48</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>9.12</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>12.23</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>4.93</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>10</td>
<td>10.34</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>5.44</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>7.78</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>8.31</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>9.1</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>5.23</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>8.38</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>10</td>
<td>9.88</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>9.4</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>4.42</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>10</td>
<td>7.55</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>6.42</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>6.17</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>7.97</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>10</td>
<td>8.52</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>7.34</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>4.55</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>4.02</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>8.38</td>
</tr>
</tbody>
</table>
Table A.4: Experiment results: Players 15-20

<table>
<thead>
<tr>
<th>Player</th>
<th>Experience</th>
<th>Session</th>
<th>Fairness (Q1)</th>
<th>Accurateness (Q2)</th>
<th>Responsiveness (Q3)</th>
<th>Accuracy (over 100 percent)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>12.53</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>10.28</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>9.33</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>11.08</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>5.88</td>
</tr>
<tr>
<td>16</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>7.06</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>11.11</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>5</td>
<td>16.88</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>10.71</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>12.84</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>8.7</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5.82</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>12.99</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>14.37</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>12.32</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>17.36</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9.9</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10.05</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6.81</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>11.64</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7.1</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>4</td>
<td>12.15</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>4.95</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5.3</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>8.74</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>10.82</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>6.89</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>8.9</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>7.56</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5.93</td>
</tr>
</tbody>
</table>
Figure A.2: Perceived accuracy of players with and without CLC, with outliers.
Figure A.3: Perceived responsiveness of players with and without CLC, with outliers.
Figure A.4: Real accuracy of players with and without CLC, with outliers.
Figure A.5: Outliers for perceived fairness of players in session 1 to 6.
Figure A.6: Outliers for perceived accuracy of players in session 1 to 6.
Figure A.7: Outliers for perceived responsiveness of players in session 1 to 6.
Figure A.8: Outliers for real accuracy of players in session 1 to 6.
Figure A.9: Perceived fairness of players with and without CLC, without outliers.
Figure A.10: Perceived accuracy of players with and without CLC, without outliers.
Figure A.11: Perceived responsiveness of players with and without CLC, without outliers.
Figure A.12: Real accuracy of players with and without CLC, without outliers.
Figure A.13: Players’ perceived fairness histograms for all sessions.
Figure A.14: Players’ perceived accuracy histograms for all sessions.
Figure A.15: Players’ perceived responsiveness histograms for all sessions.
Figure A.16: Players’ real accuracy histograms for all sessions.