Worcester Polytechnic Institute

GameBeam: A Decentralized Framework for P2P Multiplayer Game Streaming

by

Cumhur Onat

A Thesis Submitted to the Faculty of the Worcester Polytechnic Institute

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Interactive Media & Game Development

Approved:

Advisor: _____

Professor Mark Claypool

Reader: _____

Professor Gillian Smith

April 2025

Abstract

Traditional multiplayer gaming faces challenges like hardware requirements and installation barriers with developers needing to synchronize state across computers, while centralized cloud gaming introduces infrastructure costs and potential latency. This thesis presents GameBeam, a novel, fully decentralized peer-to-peer (P2P) game streaming framework designed to overcome these limitations. GameBeam enables installation-free browser access for guest players and simplifies multiplayer development by streaming the host's game instance. This research details the design, implementation (including a Unity SDK), and quantitative evaluation of the GameBeam architecture. Performance analysis facilitated by a custom automated testing framework characterizes GameBeam's capabilities under various configurations. The results demonstrate the feasibility of the decentralized approach, achieving low mean end-to-end latencies (around 50-51 ms in baseline tests) without dedicated servers. The evaluation quantifies host resource utilization (highlighting GPU and hardware encoding impacts), network bandwidth consumption, and scalability limitations, while confirming high streaming quality. Collectively, these findings indicate that GameBeam is a viable alternative for specific scenarios, with contributions including the framework design, empirical performance data, and open-source tools, providing a foundation for future research in decentralized real-time applications.

Contents

1

Introduction

Multiplayer gaming forms a cornerstone of the modern interactive entertainment landscape, yet established approaches face significant challenges related to accessibility, cost, and development complexity. Conventional multiplayer gaming typically mandates that each participant installs the full game client, often requiring substantial download times and powerful local hardware that meets specific minimum requirements [1, 2]. This dependency on local installations and capable hardware creates a barrier to entry, hindering spontaneous gameplay and excluding players with less powerful devices or limited storage.

Cloud gaming platforms, such as Google Stadia (now defunct), NVIDIA GeForce NOW [3], and Xbox Cloud Gaming [4], have emerged as a popular alternative, addressing the installation and hardware barriers by rendering games on powerful remote servers and streaming the output to players' devices [5, 6, 7]. This model grants quick and easy access across a wide range of client devices. However, cloud gaming introduces its own challenges, most notably the significant operational costs associated with maintaining large-scale, geographically distributed data centers [8]. Furthermore, the reliance

on centralized servers can lead to perceptible network latency, particularly for players distant from server locations, potentially degrading the player experience, especially in fast-paced games [9, 10, 11].

Beyond the player-facing challenges, developing multiplayer games presents considerable technical challenges for developers. Traditional multiplayer architectures often necessitate complex state synchronization mechanisms, prediction algorithms, and rollback techniques to maintain a consistent and fair experience across all connected players, especially when dealing with network lag or intricate physics interactions [12, 13]. This complexity significantly increases development time and resource requirements, creating a steep learning curve and potentially limiting the scope or feasibility of multiplayer features, particularly for smaller studios or independent creators.

Peer-to-peer (P2P) technologies, particularly WebRTC [14] known for its success in real-time video conferencing [15], offer a potential avenue to mitigate the infrastructure costs associated with centralized servers by enabling direct connections between players [16]. However, integrating P2P networking into games has historically faced challenges related to implementation difficulty, NAT traversal, security, and the lack of standardized frameworks tailored for the specific demands of game streaming.

1.1 Proposed Solution: GameBeam

This thesis introduces GameBeam, a novel framework designed to address the limitations inherent in conventional, cloud, and traditional P2P gaming approaches by providing a *fully decentralized* peer-to-peer architecture specifically for real-time game streaming. GameBeam aims to combine the accessibility benefits of cloud gaming (no installation

required for joining players) with the cost-efficiency potential of P2P networking, while significantly simplifying the development process for multiplayer integration.

The core GameBeam system leverages WebRTC to establish low-latency, direct P2P connections between a host player (running the game natively) and one or more guest players who join via a standard web browser. Crucially, guest players do not need to install the game; they receive a real-time video stream from the host and send their inputs back through the WebRTC connection. This model mirrors the simplicity of traditional "local" or "couch" multiplayer development, where the game logic runs in a single instance (on the host), and inputs from different players are simply fed into that instance. This eliminates the need for developers to implement complex network synchronization or state management code.

To achieve a fully decentralized architecture, GameBeam integrates two key technologies:

- **IPFS (InterPlanetary File System) [17]:** The browser-based client application code is hosted on and distributed via IPFS. When a guest joins, their browser fetches the necessary HTML, CSS, and JavaScript directly from the decentralized IPFS network, eliminating the need for a dedicated web server for client distribution.
- **WebTorrent Trackers [18]:** Initial peer discovery and the WebRTC signaling handshake (offer/answer exchange) are facilitated using standard, publicly accessible WebTorrent trackers. The host registers the game session with the tracker, and guests use a unique session identifier (shared via an invitation link) to find and connect to the host via the tracker, enabling the direct P2P WebRTC connection without a central matchmaking server.

GameBeam provides an open-source Software Development Kit (SDK) designed for easy integration into games built with the Unity engine [19], abstracting the complexities of WebRTC, IPFS hosting, and tracker communication.

1.2 Research Goals and Objectives

The primary goal of this research was to design, implement, and quantitatively evaluate the performance of the GameBeam framework as a feasible alternative for multiplayer gaming. The specific objectives were:

1. Design and implement the GameBeam architecture, encompassing:
    - An open-source Unity SDK to facilitate easy integration for game developers [20].
    - A browser-based client enabling installation-free access for guest players [21].
    - Integration with IPFS for decentralized client code distribution.
    - Utilization of public WebTorrent trackers for decentralized signaling and peer discovery.
2. Develop a dedicated, automated Performance Analysis Framework capable of systematically executing GameBeam test scenarios and collecting detailed performance metrics.
3. Conduct a comprehensive quantitative performance evaluation of the fully decentralized GameBeam system under various configurations (e.g., varying client count, resolution, frame rate, encoding methods).
4. Analyze key performance characteristics, including end-to-end latency, host resource

utilization (CPU, GPU, memory), network bandwidth consumption, streaming quality (packet loss, jitter), and scalability.

5. Assess the viability and characterize the performance trade-offs of this fully decentralized P2P game streaming approach based on the empirical data.
6. Release the core components, including the GameBeam Unity SDK and the Performance Analysis Framework, as open-source contributions to benefit the research and developer community.

While initial plans considered a developer usability study, the research focus shifted to prioritize a rigorous quantitative performance analysis due to time constraints and the importance of characterizing the novel decentralized architecture's core capabilities.

1.3 Contributions

This thesis makes the following primary contributions:

1. **GameBeam Framework:** The design, implementation, and demonstration of GameBeam, a novel, fully decentralized, installation-free P2P game streaming framework leveraging WebRTC, IPFS, and public WebTorrent trackers. GameBeam simplifies multiplayer development by adopting a local-multiplayer paradigm over a network stream.
2. **Quantitative Performance Characterization:** A comprehensive performance evaluation of the decentralized GameBeam architecture, providing empirical evidence of its capabilities. This includes detailed analysis of end-to-end latency, host resource demands, network footprint, scalability, and streaming quality under various oper-

ational conditions. Notably, the evaluation demonstrates the feasibility of achieving low end-to-end latencies (median around 50-51 ms under baseline 1080p/30fps conditions; see Table 6.2) without reliance on centralized servers for streaming or signaling.

3. **Open-Source Performance Analysis Framework:** The development and release of a reusable, automated framework [22] specifically designed for evaluating the performance of P2P game streaming systems, facilitating reproducible research in this area.
4. **Feasibility Demonstration:** Empirical validation that a purely decentralized approach, using readily available P2P technologies like WebRTC, IPFS, and public trackers, can provide a functional and performant low-latency game streaming experience.
5. **Open-Source Artifacts:** The release of the GameBeam Unity SDK and the Performance Analysis Framework as open-source tools, fostering further research, development, and adoption within the gaming and P2P technology communities.

By demonstrating the viability and characterizing the performance of this decentralized model, this work aims to lower the barriers for developers seeking to implement cost-effective, accessible multiplayer experiences and contributes to the understanding of decentralized systems in the context of real-time interactive applications.

The rest of this thesis is organized as follows: Chapter 2 describes related work, Chapter 3 details the GameBeam system design, Chapter 4 covers the implementation, Chapter 5 outlines the performance evaluation methodology, Chapter 6 presents the results, and Chapter 7 provides the conclusion and discusses future work.

2

Background and Related Work

GameBeam builds upon concepts from peer-to-peer networking, real-time web communication, and decentralized technologies, while positioning itself as an alternative to existing multiplayer gaming solutions. This chapter provides background on these areas and reviews related work in P2P game streaming and cloud gaming.

2.1 Peer-to-Peer Networking Principles

Peer-to-Peer (P2P) networking represents a paradigm shift from traditional client-server architectures. In a P2P network, participants (peers) communicate directly with each other without necessarily relying on a central intermediary server for data exchange. This decentralization can offer benefits such as increased scalability, resilience (no single point of failure), and potentially lower infrastructure costs.

However, establishing direct P2P connections over the public Internet faces significant challenges, primarily due to Network Address Translators (NATs) and firewalls [23]. NAT devices, commonly found in home routers, allow multiple devices on a private net-

work to share a single public IP address. While essential for conserving IPv4 addresses, NATs obscure the internal IP addresses and ports of devices, making it difficult for external peers to initiate direct connections. Overcoming NAT traversal is a fundamental problem that P2P systems must address to enable direct communication between peers located behind different NATs [24]. Techniques like UDP hole punching, relaying (e.g., using TURN servers, discussed later), and signaling mechanisms are commonly employed to facilitate P2P connectivity.

2.2 WebRTC for Real-Time Communication

Web Real-Time Communication (WebRTC) is an open-source project and W3C standard that enables real-time communication (RTC) capabilities directly within web browsers and mobile applications via simple JavaScript APIs [14]. It facilitates the exchange of audio, video, and generic data directly between peers, forming the technological backbone for GameBeam's streaming functionality.

WebRTC encompasses several fundamental components and protocols:

- **Media Capture and Streams:** APIs ('getUserMedia') to access camera, microphone, or screen capture streams.
- **P2P Connection ('RTCPeerConnection'):** The core API for establishing and managing the direct connection between peers, handling media and data transmission.
- **Signaling:** WebRTC does not define a specific signaling protocol. Developers must use an external mechanism (like WebSockets, or in GameBeam's case, WebTorrent trackers) to exchange control messages needed to set up the connection. This includes exchanging session descriptions (using Session Description Protocol - SDP)

detailing media capabilities and network information.

- **NAT Traversal (ICE, STUN, TURN):** WebRTC incorporates the Interactive Connectivity Establishment (ICE) framework to find the best possible path for connecting peers. ICE utilizes STUN (Session Traversal Utilities for NAT) servers to help peers discover their public IP address and port, and TURN (Traversal Using Relays around NAT) servers to act as relays when a direct P2P connection cannot be established.
- **Data Channels ('RTCDataChannel'):** Allows for the transmission of arbitrary application data directly between peers, used by GameBeam for sending player inputs from the browser client to the host.

By integrating these capabilities, WebRTC provides a powerful, standardized foundation for building low-latency P2P communication applications, making it ideal for use cases like video conferencing, file sharing, and, as explored in this thesis, real-time game streaming.

2.3 Decentralized Technologies in GameBeam

GameBeam leverages specific decentralized technologies beyond the core P2P communication enabled by WebRTC to achieve its fully decentralized architecture for signaling and client distribution.

2.3.1 IPFS (InterPlanetary File System)

IPFS [17] is a peer-to-peer hypermedia protocol designed to make the web faster, safer, and more open. Instead of addressing content by its location (like HTTP URLs), IPFS uses content addressing. Files are identified by a unique cryptographic hash of their con-

tent, known as a Content Identifier (CID). When a peer requests content by its CID, the IPFS network locates peers storing that content and retrieves it directly from them. This approach offers benefits like data deduplication, persistence (content remains accessible as long as at least one peer hosts it), and censorship resistance [25, 26].

In GameBeam, IPFS is utilized to host and distribute the browser-based client application (HTML, CSS, JavaScript). The host game uploads the client build to IPFS, obtaining a unique CID. This CID is embedded in the session invitation link. When a guest clicks the link, their browser's IPFS client (e.g., via a gateway or extension) fetches the application code directly from the IPFS network, eliminating the need for GameBeam to rely on a centralized web server for client distribution.

2.3.2 WebTorrent Trackers

BitTorrent and WebTorrent protocols rely on trackers (or alternative mechanisms like DHT) for peer discovery [27]. A tracker is essentially a server that keeps track of which peers are interested in a particular torrent (identified by its info-hash). Peers periodically announce their presence to the tracker for a specific info-hash and receive a list of other peers participating in the same swarm. This enables peers to find and connect to each other to exchange data. WebTorrent adapts this concept for web browsers, often using WebSocket-based tracker protocols.

GameBeam repurposes WebTorrent trackers for use as a decentralized signaling mechanism for establishing WebRTC connections. Instead of a dedicated matchmaking server, the GameBeam host generates a unique identifier for the game session (acting somewhat like an info-hash) and announces itself to a public WebTorrent tracker using this identifier.

The guest client, receiving the identifier via the invitation link, queries the same tracker to find the host's connection information (IP address, port, and initial signaling data like the WebRTC offer). The tracker facilitates this initial discovery and rendezvous, after which the peers can proceed with the standard WebRTC handshake directly or via further messages relayed through the tracker, ultimately establishing the direct P2P data and media streams.

2.4 Existing Multiplayer Gaming Solutions

GameBeam positions itself relative to two dominant paradigms in multiplayer gaming: conventional architectures and cloud gaming platforms.

2.4.1 Conventional Multiplayer Architectures

The traditional approach requires each player to purchase and install a full copy of the game client on their local machine (PC or console). Gameplay relies on the processing power of the local hardware, demanding players meet minimum system requirements. Game installations can be large and time-consuming, hindering quick or spontaneous play sessions. While offering potentially high fidelity and low latency (depending on network conditions and architecture), this model faces accessibility challenges due to hardware and installation prerequisites.

From a development perspective, conventional multiplayer games often require significant effort to implement robust networking solutions [28]. Developers must tackle challenges like synchronizing game state across distributed clients, handling network latency through prediction and reconciliation techniques, managing complex physics inter-

actions in a networked environment, and preventing cheating. This complexity presents a substantial barrier, especially for smaller development teams.

2.4.2 Cloud Gaming Platforms

Cloud gaming services (e.g., NVIDIA GeForce NOW [3], Xbox Cloud Gaming [4]) address the installation and hardware barriers of conventional gaming. Games run on powerful servers in data centers, and the rendered audio/video output is streamed to the player's thin client device (e.g., PC, smartphone, smart TV) over the Internet. Player inputs are sent back to the server to control the game. This model allows access to demanding games on low-spec devices.

However, cloud gaming introduces significant operational costs for service providers due to the need for extensive server infrastructure. For players, performance is heavily dependent on the quality and latency of their Internet connection to the nearest data center. High latency can negatively impact the user experience, particularly in fast-paced genres [29]. While edge computing approaches aim to mitigate latency by moving servers closer to users, the fundamental reliance on centralized, provider-managed infrastructure remains. Open-source frameworks like Gaming Anywhere [30] exist but share the centralized server model.

2.5 Related P2P and Streaming Frameworks

Several academic and open-source projects have explored P2P technologies for gaming and streaming, representing relevant related work.

Peer-to-peer architectures have been explored for massively multiplayer online games

(MMOGs) to eliminate server costs. The **PartyPeer** [31] project used a P2P protocol for low-latency streaming in MMOGs, focusing on scalability and consistency in a decentralized setting. Similarly, Ahmad et al. proposed P2P live streaming for MMOGs to reduce host upload bandwidth, though both relied on some central overlay components and typically required full game installation by all participants [32].

WebRTC and HTML5 have been leveraged for P2P media streaming, demonstrating the browser's capability as a P2P node. In the gaming domain, **RenderLink** [33] used WebRTC to allow peers with powerful GPUs to render games for others. While innovative, it relied on Kubernetes for backend management and faced challenges in incentivizing resource sharing. The **Proton** framework utilized WebRTC for online multiplayer in Unity, offering a hybrid P2P approach where some critical state remained centralized for consistency [34].

Other popular proprietary streaming solutions like **SteamLink** [35], **Sunshine** / **Moonlight** [36], and **Parsec** [37] primarily focus on streaming from a user's own powerful host machine to other devices, either locally or over the I nternet. While offering low latency, they depend on the user possessing capable host hardware and often use centralized services for connection establishment.

2.6 Positioning GameBeam

GameBeam distinguishes itself from the aforementioned solutions through its unique combination of features, aiming to synthesize the benefits while mitigating the drawbacks of existing approaches:

- **Fully Decentralized Architecture:** Unlike cloud gaming platforms or hybrid P2P

frameworks like Proton and RenderLink that retain some centralized components for signaling, management, or state, GameBeam operates in a fully decentralized manner for core streaming and session setup. It relies only on existing, widely available decentralized infrastructure (IPFS for distribution, public WebTorrent trackers for signaling), minimizing specific infrastructure costs and dependencies.

- **Installation-Free Access for Guests:** Similar to cloud gaming, GameBeam allows guests to join and play instantly via a web browser without installing the full game client. This contrasts with conventional multiplayer and many P2P frameworks like PartyPeer that require local installation.
- **Simplified Development Model:** GameBeam abstracts the complexities of network synchronization by adopting a model akin to local multiplayer development. The host runs the authoritative game instance, and the GameBeam SDK handles the streaming of inputs/outputs via WebRTC. This significantly reduces the development effort compared to implementing traditional networked multiplayer logic.
- **Host-Owned Computation:** Unlike cloud gaming where computation occurs on provider servers, the game execution in GameBeam happens on the host player's machine. This avoids the operational costs of cloud infrastructure but requires the host to have sufficient resources to run the game and encode the stream(s).
- **Demonstrated Low Latency:** The performance evaluation (detailed in Chapter 6) confirms that GameBeam can achieve low end-to-end latency (around 50ms median in baseline tests) comparable to or better than some cloud gaming scenarios, without dedicated streaming servers.

By integrating WebRTC, IPFS, and WebTorrent trackers, GameBeam presents a novel frame-

work that prioritizes decentralization, accessibility, and developer ease-of-use, offering a distinct alternative in the landscape of multiplayer gaming technologies.

3

GameBeam System Design

This chapter details the architecture and core components of the GameBeam framework. It outlines the overall system structure, describes the roles of the key components, details the interaction flow for establishing a streaming session, and discusses the design rationale behind the chosen fully decentralized approach.

3.1 Overall Architecture

GameBeam implements a peer-to-peer (P2P) game streaming architecture designed for decentralization and ease of use. The system consists of two primary actors: the **Host** and one or more **Guests**.

The Host runs a native application, typically a game developed in Unity, integrated with the **GameBeam Unity SDK**. This SDK is responsible for capturing the game's video and audio output, managing P2P connections, and handling data exchange.

Guests interact with the system via a standard web browser, without needing to install the game locally. They load a lightweight **Browser Client** application, which is dynami-

cally fetched from the InterPlanetary File System (IPFS), typically via a public gateway.

Communication between the Host and Guest(s) occurs primarily over direct P2P connections established using **WebRTC**. The Host streams encoded video and audio to the Guest(s), while the Guest(s) send their input events (keyboard, mouse, gamepad) back to the Host via WebRTC data channels.

Crucially, the bootstrapping process—discovering peers and negotiating the WebRTC connection parameters (signaling)—is handled through a decentralized mechanism utilizing **WebTorrent Trackers**, eliminating the need for a dedicated central matchmaking or signaling server. The Browser Client itself is also distributed decentrally via **IPFS**. This overall architecture prioritizes minimizing reliance on centralized infrastructure, aligning with the goals of cost reduction and increased accessibility. A conceptual view involves the host handling game logic and streaming media/accepting inputs from browser-based guests via WebRTC.

3.2 Core Components

The GameBeam system comprises three main components working in concert: the Unity SDK integrated into the host game, the Browser Client run by guests, and the decentralized Signaling Mechanism leveraging IPFS and WebTorrent trackers.

3.2.1 GameBeam Unity SDK

The GameBeam Unity SDK is a library integrated directly into the host's Unity game. Its primary responsibilities include:

- **Session Management:** Providing an API for the host game logic to initiate ('Create

Session') and manage GameBeam streaming sessions.

- **Media Capture and Encoding:** Capturing rendered video frames and game audio directly within Unity. It handles the encoding of the video stream, offering options for hardware-accelerated encoding (e.g., NVENC) where available to reduce CPU load.
- **Client Distribution via IPFS:** Packaging the necessary Browser Client files (HTML, CSS, JS) and uploading them to the IPFS network to obtain a unique Content Identifier (CID).
- **WebRTC Peer Connection Management:** Creating and managing 'RTCPeerConnection' objects for each connected guest. This includes configuring media tracks (video, audio) and data channels (for input events), handling the SDP offer/answer exchange, and managing ICE candidate gathering and exchange via the signaling mechanism.
- **Signaling Coordination:** Interacting with a list of WebTorrent trackers (including the project's tracker at `http://tracker.gamebeam.org/` and other public trackers) to announce the session's availability and exchange signaling messages with potential guests, using a fast fallback method for reliability.
- **Invitation Link Generation:** Creating a unique URL containing the necessary information for guests to join: the list of WebTorrent tracker URLs, the unique session identifier, and the IPFS CID for the Browser Client code.
- **Input Handling:** Receiving input events from guests via the WebRTC data channel and providing mechanisms for the host game logic to interpret and apply these inputs to the corresponding guest player representations within the game scene.

3.2.2 Browser Client

The Browser Client is a web application executed in the guest's browser. It is designed to be lightweight and require no installation beyond a modern web browser supporting WebRTC. Its key functions are:

- **Loading via IPFS Gateway:** The client code is typically fetched from the IPFS network via a public IPFS gateway, using the CID provided in the invitation link. This approach enhances accessibility for guests, as they do not need to run a local IPFS node.
- **Signaling Coordination:** Parsing the invitation link to identify the tracker list and session ID, then interacting with the WebTorrent trackers (using the fast fallback strategy) to find the host and exchange the necessary WebRTC signaling messages.
- **WebRTC Peer Connection Management:** Establishing and managing its end of the 'RTCPeerConnection' with the host, including processing SDP offers/answers and ICE candidates.
- **Media Reception and Rendering:** Receiving the video and audio streams from the host via WebRTC, decoding them, and rendering the video output (typically to an HTML '<video>' element) while playing the audio.
- **Input Capture and Transmission:** Capturing user input events (e.g., key presses, mouse movements, gamepad states) within the browser and sending them reliably to the host over the WebRTC data channel.

3.2.3 Signaling Mechanism (Decentralized)

GameBeam replaces the need for a dedicated central matchmaking or signaling server by employing a two-part decentralized mechanism:

- **IPFS for Client Distribution:** As described, IPFS serves as the decentralized content delivery network for the Browser Client code. The host uploads the client, and guests download it, typically via a public gateway for ease of use, using the content's hash (CID). This removes reliance on a central web server for hosting the client files.
- **WebTorrent Trackers for Peer Discovery and Signaling:** A list of WebTorrent trackers, including the project-hosted `http://tracker.gamebeam.org/` and other public trackers, are used to facilitate the initial rendezvous and signaling exchange. The system employs a fast fallback mechanism, attempting to connect to trackers in the list sequentially until one responds successfully. The host announces the session under a unique ID to the tracker list. Guests use this ID (from the invitation link) to query the trackers. The responsive tracker acts as a temporary message broker, allowing the host and guest to exchange the initial WebRTC signaling messages needed to bootstrap the direct P2P connection. Once the direct connection is established, the tracker is typically no longer needed for that peer pair's communication. This multi-tracker approach enhances the reliability of the signaling process compared to relying on a single tracker.

3.3 Interaction Flow

The process of initiating a GameBeam session and connecting a guest involves the following conceptual steps, as illustrated in Figure 3.1:

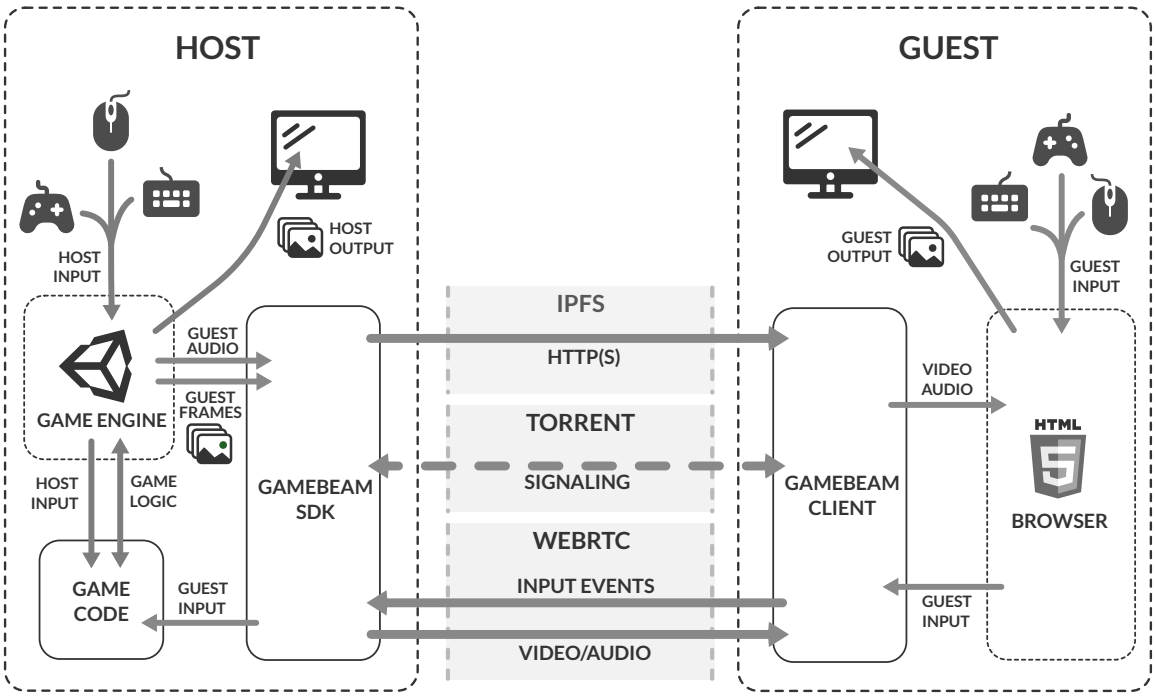

Figure 3.1: *System architecture overview of GameBeam, showing the key components and their interactions.*

1. **Host Initiates Session:** The player hosting the game interacts with the Unity application to start a new GameBeam session.
2. **SDK Prepares Client:** The GameBeam SDK packages the Browser Client web application files.
3. **SDK Uploads Client to IPFS:** The SDK uploads the packaged client files to the IPFS network, receiving a CID.
4. **SDK Generates Session ID:** The SDK creates a unique identifier for this specific game session.

5. **SDK Announces to Trackers:** The SDK connects to its configured list of WebTorrent trackers (e.g., starting with `http://tracker.gamebeam.org/`) using the fast fallback method. It announces the session identifier and its initial signaling information (e.g., WebRTC offer) to the first responsive tracker.
6. **SDK Generates Invitation Link:** The SDK constructs a shareable invitation URL containing the list of tracker URLs, the unique session ID, and the IPFS CID.
7. **Host Shares Link:** The host player copies this link and shares it with potential guests (out-of-band).
8. **Guest Opens Link:** The guest player clicks or opens the invitation link in their web browser.
9. **Browser Fetches Client via IPFS Gateway:** The browser uses the CID from the URL to request and download the GameBeam client application code from the IPFS network, typically via a public IPFS gateway.
10. **Client Contacts Trackers:** The loaded client application parses the session ID and tracker list. It attempts to connect to the trackers in the list (using fast fallback) and queries for peers associated with the session ID.
11. **Signaling Exchange via Tracker:** The first responsive tracker facilitates the WebRTC signaling handshake between the guest and the host, relaying messages like the SDP answer and ICE candidates.
12. **WebRTC Handshake:** The host SDK and the guest Browser Client use the exchanged signaling information to perform ICE connectivity checks and establish a secure DTLS connection directly.
13. **P2P Connection Established:** A direct, encrypted P2P WebRTC connection is suc-

cessfully established.

14. **Streaming Commences:** The Host SDK begins streaming video/audio; the Guest client begins sending inputs. Gameplay proceeds.

3.4 Design Rationale

The design of GameBeam was driven by several key principles aimed at addressing the limitations of existing multiplayer gaming solutions:

- **Emphasis on Decentralization:** A primary goal was to minimize reliance on centralized server infrastructure. Using IPFS and WebTorrent trackers eliminates the need for dedicated web and matchmaking servers, potentially reducing costs, improving scalability and resilience, and enhancing censorship resistance.
- **Prioritizing Accessibility (Installation-Free):** Leveraging a browser-based client ensures instant join capability for guests, significantly lowering the barrier to entry. Using public IPFS gateways further simplifies guest access currently.
- **Simplifying Developer Experience:** The architecture intentionally mimics local multiplayer development, aiming to abstract away network synchronization complexities via the SDK.
- **Leveraging Web Standards (WebRTC):** Choosing WebRTC provides a standardized, browser-native solution for low-latency P2P communication.
- **Utilizing Existing Decentralized Technologies (IPFS, Trackers):** GameBeam leverages established P2P technologies rather than inventing new protocols, benefiting from their existing ecosystems and robustness. The multi-tracker fallback strategy enhances signaling reliability.

However, this fully decentralized design also introduces certain **trade-offs and considerations**:

- **Reliance on Public/External Infrastructure:** The system depends on the availability and performance of the chosen WebTorrent tracker(s) (including `http://tracker.gamebeam.org/`) and the IPFS network (including the public gateways currently used by clients). These are external dependencies. While the multi-tracker approach mitigates tracker failure, widespread outages could still pose issues.
- **Gateway Centralization Point (Current Implementation):** While IPFS itself is decentralized, the current reliance on public IPFS gateways for client fetching introduces a potential point of centralization or failure for guests. This is a configurable aspect and clients could potentially use local IPFS nodes if available.
- **Initial Connection Variability:** Fetching client code via gateways and establishing connections via trackers can be more variable than connecting to dedicated central servers, depending on network conditions.
- **NAT Traversal Limitations:** While WebRTC's ICE helps, direct P2P connection is not always guaranteed. Reliance on TURN relays (not explicitly part of the core test setup) might sometimes be needed, introducing a degree of centralization.
- **Host Resource Requirements:** The host machine must have sufficient resources (CPU, GPU, bandwidth) for gameplay and streaming.

Despite these trade-offs, the design rationale prioritized the benefits of decentralization, accessibility, and developer simplicity, aiming to create a viable and novel alternative for specific multiplayer gaming scenarios.

4

Implementation

This chapter details the practical implementation of the GameBeam framework, turning the system design outlined in Chapter 3 into the functional components evaluated in Chapters 5 and 6. The core objective was to build a fully decentralized peer-to-peer game streaming system, reducing dependence on central servers while making access easier and simplifying the multiplayer development process for Unity game creators.

The implementation scope consists of three primary areas, directly addressing the research objectives:

1. **GameBeam Unity SDK:** An open-source SDK was developed to allow easy integration into Unity games. This involved creating modules for managing P2P connections, capturing and encoding media streams, handling input synchronization, and interfacing with decentralized services.
2. **Browser-Based Guest Client:** A lightweight web client was built using standard web technologies, enabling guests to join game sessions instantly without installation. This required implementing WebRTC connectivity, media rendering, input capture, and interaction with signaling and distribution mechanisms.

3. **Decentralized Infrastructure Integration:** The system was designed to leverage existing decentralized technologies for core functions. IPFS was integrated for distributing the browser client code, and public WebTorrent trackers were utilized for peer discovery and WebRTC signaling, fulfilling the goal of a fully decentralized architecture.

Furthermore, the implementation included specific mechanisms, such as precise timestamp embedding using WebRTC Insertable Streams, to facilitate the quantitative performance evaluation detailed later in Chapters 5. This chapter provides insight into the technical choices, structure, and challenges involved in realizing the GameBeam system. Figure 4.1 illustrates the high-level runtime interaction flow between these components.

4.1 Technology Stack

The development of GameBeam utilized a combination of modern web technologies and game development tools. The primary components of the technology stack were:

- **Unity Engine:** Version 6000.0.33f1 of the Unity Engine [19] was used for host application development and testing.
- **C#:** The language used for implementing the GameBeam Unity SDK.
- **WebRTC (Unity & Browser):** The 'com.unity.webrtc' package provided bindings in Unity, while the browser client used native browser WebRTC APIs [14].
- **Typescript:** Typescript [38] was used for the development of the browser client and bundled into single HTML file containing JavaScript during the build process.
- **HTML5 & CSS3:** Used for the structure [39] and styling [40] of the browser client.

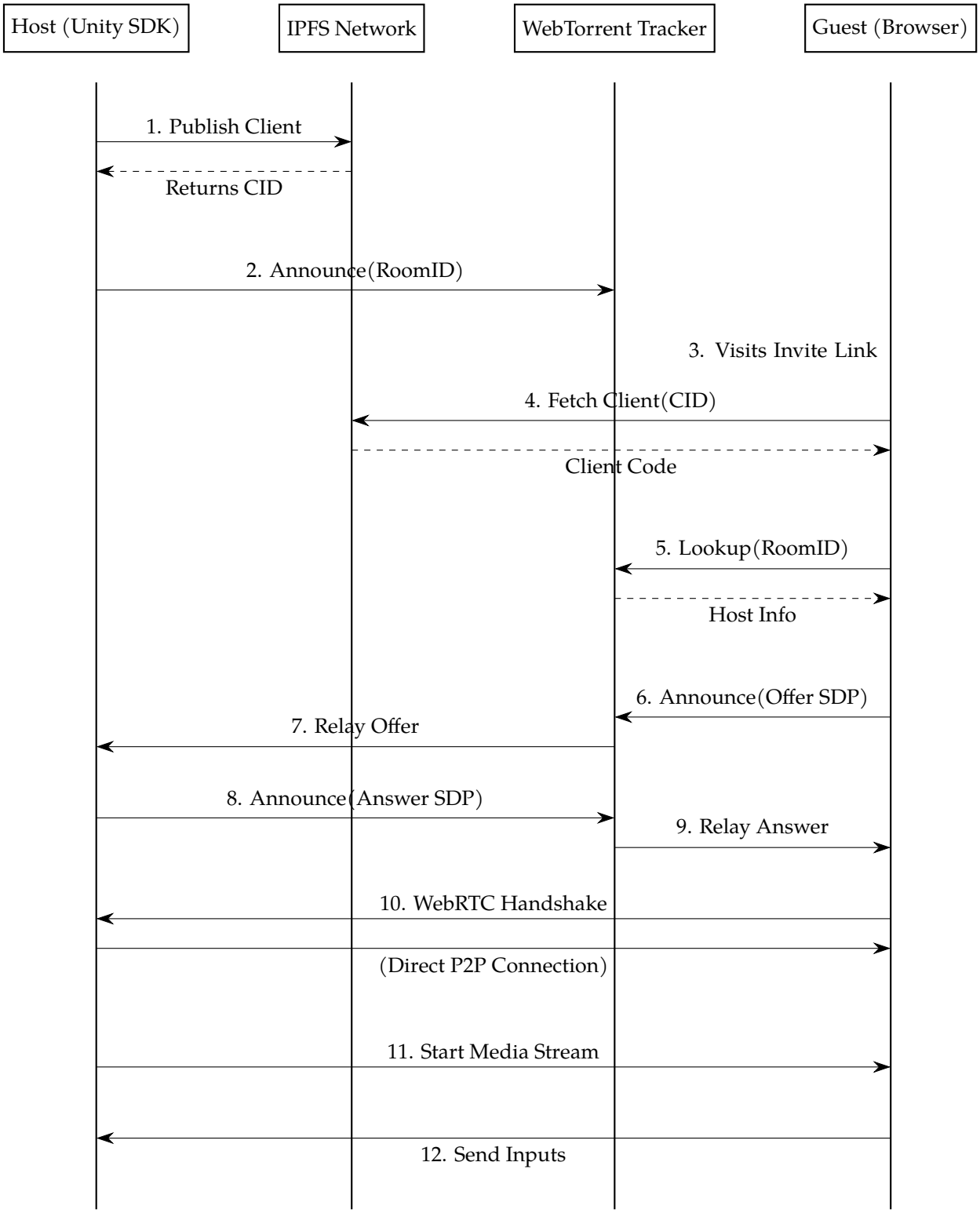

Figure 4.1: *Simplified Sequence Diagram of GameBeam Session Establishment.*

- **WebSockets:** Employed for signaling communication with the tracker (using browser APIs based on the WebSocket protocol [41] and the WebSocketSharp [42] library in Unity).
- **WebTorrent Trackers:** The WebTorrent tracker protocol [18] facilitated decentralized signaling, primarily using a custom instance at `wss://tracker.gamebeam.org/`.
- **IPFS:** The Go implementation (Kubo) of IPFS [17] was used via the 'IPFSHostingService.cs' for decentralized client distribution.
- **STUN/ICE:** Standard WebRTC mechanisms incorporating STUN [43] and ICE [44], using `stun:stun.l.google.com:19302`, facilitated NAT traversal.

4.2 Host SDK Implementation

The GameBeam Unity SDK ('org.gamebeam.unity') encapsulates the host-side logic, providing APIs for game integration.

4.2.1 Lifecycle and Initialization

The core 'GameBeam.cs' singleton manages the session lifecycle and is the entry point for developer integration. Initialization involves:

- Generating unique 'clientID' and 'roomID' strings using SHA256.
- Starting a hosting service ('IHostingService') to manage client distribution. The 'IPFSHostingService' implementation handles bundling the IPFS executable, running the daemon, adding the client 'index.html' (packaged as a Unity resource), retrieving its CID, and constructing the final invitation link (e.g., `https://ipfs.io/ipfs/<CID>/#<RoomID>`).

- Initializing and connecting the signaling service (‘WebtorrentSignaling.cs‘) to the tracker.
- Starting necessary WebRTC background processes.

The SDK provides a convenience property ‘InviteLink‘ for sharing the invitation link.

4.2.2 Media Capture and Encoding

The ‘WebRtcClient.cs‘ class handles media for each guest connection:

- **Video:** A ‘RenderTexture‘ is created and assigned as the target for a Unity Camera. A ‘VideoStreamTrack‘ using this texture is added to the ‘RTCPeerConnection‘. The target frame rate (default 30) is configured on the track sender and synchronized with ‘Application.targetFrameRate‘.
- **Audio:** Standard Unity architecture limits scenes to a single active ‘AudioListener‘, making it impossible to provide unique audio perspectives for multiple networked players within the same scene out-of-the-box. To overcome this limitation and provide each guest with audio spatially positioned relative to their corresponding representation in the host’s game world, GameBeam implements a custom audio capture system. This system involves three key scripts:
  - **‘MultiAudioSource.cs‘:** This component is attached to any GameObject in the scene that contains one or more standard Unity ‘AudioSource‘ components that should be audible to guests. It automatically detects these sources. For every active ‘MultiAudioListener‘ (representing a guest), ‘MultiAudioSource‘ dynamically creates a child GameObject. This child GameObject contains a duplicate ‘AudioSource‘ (configured with ‘spatialBlend = 0f‘ to bypass Unity’s default

spatialization for this duplicate) and an ‘AudioSourceFilter‘ component. It continuously monitors the original ‘AudioSource‘ for changes (play state, clip, loop, pitch, volume, etc.) and mirrors these changes onto all its corresponding child duplicate sources.

– **‘MultiAudioListener.cs‘:** An instance of this component is created for each connected guest ‘WebRtcClient‘. It represents the ”ears” of the guest within the host’s scene and is typically attached to the guest’s player character GameObject or a similar proxy. It holds a reference to the ‘AudioStreamTrack‘ that sends audio data to the specific guest via WebRTC. It maintains internal buffers and uses Unity’s ‘OnAudioFilterRead‘ callback purely for timing. Its core function is to mix the audio data it receives from various ‘AudioSourceFilter‘ instances.

– **‘AudioSourceFilter.cs‘:** This component resides on the child GameObjects created by ‘MultiAudioSource‘. Each instance links one specific original ‘AudioSource‘ to one specific ‘MultiAudioListener‘ ‘AudioSourceFilter.cs‘. In its ‘Update‘ method, it calculates the distance between the actual sound source (‘MultiAudioSource‘ transform) and its assigned listener (‘MultiAudioListener‘ transform). Based on this distance and the 3D audio settings of the *original* ‘AudioSource‘ (rolloff mode, min/max distance, spatial blend factor), it manually calculates the appropriate volume attenuation, effectively replicating Unity’s spatial audio falloff logic ‘AudioSourceFilter.cs‘. In the ‘OnAudioFilterRead‘ audio processing callback, it takes the audio samples generated by its duplicate ‘AudioSource‘, applies the calculated distance-based attenuation factor to these samples, and then sends the resulting attenuated audio data to its associated ‘MultiAudioLis-

tener''s mixing buffer via 'listener.AddAudioData()'. Crucially, it then zeroes out the audio data buffer provided by 'OnAudioFilterRead' to prevent this duplicate source from being audible in the host's main audio output.

- **Encoding:** Hardware-accelerated encoding (NVENC [45]) is used by default. If disabled or not supported by host, non-H264 codecs are preferred via SDP modification and 'SetCodecPreferences' to force software encoding.

4.2.3 Input Processing and Game Integration

Guest inputs are received via the WebRTC data channel:

- Messages are deserialized by 'WebRtcClient.cs' according to the 'InputProtocol' definition (Listing 4.1).
- Deserialized events populate a 'GameBeamInput' component, which exposes standard input polling methods ('GetKey', 'GetMouseButton') and events ('OnKeyDown', 'OnMouseMove', etc.).
- Game integration requires implementing 'IClientHandler' (as in 'GameController.cs'). The 'ClientJoined' method receives the 'WebRtcClient' instance, allowing the game to associate it with a guest entity, configure rendering/audio listeners, and access the 'InputHandler'. Input mapping can then be implemented, for example, using helper classes like 'JoystickHelper.cs' and 'GameBeamUIInput.cs' provided by the SDK or custom implementation.

Listing 4.1: *InputProtocol Message Types (C#).*

```
public enum InputMessageType : byte
{
    KeyDown = 1,
    KeyUp = 2,
    MouseMove = 3,
    MouseDown = 4,
    MouseUp = 5
}

public static class InputProtocol
{
    public static (string key, bool isDown) DeserializeKeyEvent(byte[] data)
    {
        int keyLength = data[1];
        var key = Encoding.UTF8.GetString(data, 2, keyLength);
        return (key, data[0] == (byte)InputMessageType.KeyDown);
    }

    public static Vector2 DeserializeMouseMove(byte[] data)
    {
        return new Vector2(
            BitConverter.ToSingle(data, 1),
            BitConverter.ToSingle(data, 5)
        );
    }

    public static (int button, Vector2 position) DeserializeMouseButton(byte[] data)
    {
        return (
            data[1],
            new Vector2(
                BitConverter.ToSingle(data, 2),
                BitConverter.ToSingle(data, 6)
            )
        );
    }
}
```

4.3 Guest Client Implementation

The browser client provides the installation-free guest experience using standard web technologies.

4.3.1 Initialization and Rendering

Loading ‘index.html‘ triggers ‘app.js‘, which initializes the client:

- The ‘roomId‘ is read from the URL hash.
- A ‘Renderer‘ (‘renderer.js‘) creates and manages the ‘<video>‘ element. CSS styles ensure it fills the screen and applies a vertical flip (‘transform: scaleY(-1)‘) to match the expected orientation from the Unity render texture. This is necessary because the browser client receives the video stream from the host in a vertically inverted orientation. This process was offloaded to the browser client to reduce the host’s processing load.
- A ‘UI‘ component (‘ui.js‘) adds a fullscreen toggle button.
- A ‘Game‘ component (‘game.js‘) sets up input listeners.
- A ‘PeerConnectionHandler‘ (‘webrtc.js‘) manages WebRTC and signaling setup.

4.3.2 Input Capture and Serialization

The ‘Game‘ class (‘game.js‘) captures user input:

- Standard DOM event listeners (‘keydown‘, ‘keyup‘, ‘mousemove‘, ‘mousedown‘, ‘mouseup‘) are used.
- Mouse coordinates are normalized relative to the video element’s bounds.

- The ‘InputProtocol‘ class (‘inputProtocol.js‘, see Listing 4.2) provides static methods to serialize captured events into ‘ArrayBuffer‘s matching the C# protocol definition.
- Serialized buffers are sent via the ‘RTCDataChannel‘ managed by ‘PeerConnectionHandler‘. Mouse move events are throttled.

Listing 4.2: *InputProtocol Serialization Signature*

```
export class InputProtocol {
    static serializeKeyEvent(type, key) { /* ... */ }
    static serializeMouseMove(position) { /* ... */ }
    static serializeMouseButton(type, button, position) { /* ... */ }
}
```

4.4 Cross-Cutting Mechanisms

Several mechanisms operate across both host and client components to enable the P2P streaming.

4.4.1 Decentralized Signaling

Peer discovery and WebRTC handshake negotiation occur via WebTorrent trackers:

- **Host (‘WebtorrentSignaling.cs‘):** Connects to the tracker infrastructure, periodically sends ‘announce‘ messages with its ‘peer_id‘ and the session ‘info_hash‘ (roomID) (Structure shown in Listing 4.3). Listens for messages containing offers from guests. Sends targeted ‘announce‘ messages containing the SDP answer back to specific guests via the tracker.
- **Client (‘signaling.js‘, ‘webrtc.js‘):** Generates an SDP offer (with ICE candidates bundled, as ‘canTrickleIceCandidates‘ is set to false to complete signaling in single

step). Connects to the tracker infrastructure, sends an 'announce' message containing the offer and a unique 'offer_id' (Structure shown in Listing 4.4). Retries sending the offer periodically until an answer is received. Listens for tracker messages containing the host's SDP answer.

- **Reliability (Fast Fallback):** To improve robustness against individual tracker failures, a multi-tracker "fast fallback" mechanism was implemented. Both the host SDK and client maintain a configured list of tracker URLs (including the primary `wss://tracker.gamebeam.org/` and other public trackers). When initiating a connection or sending an announce message, the system attempts to connect to the first tracker in the list. If this connection fails (due to timeout, network error, or tracker unavailability), it automatically closes the attempt and tries the next tracker in the sequence. This process repeats until a connection is successfully established with a responsive tracker, significantly enhancing the reliability of the signaling process compared to relying on a single instance.

Listing 4.3: *JSON Payload for Host Announce Structure*

```
{
  "action": "announce", "info_hash": roomId,
  "peer_id": clientId, "numwant": 30
}
```

4.4.2 NAT Traversal

Basic NAT traversal is handled using WebRTC's built-in ICE framework, utilizing a public STUN server (`stun:stun.l.google.com:19302`) configured on both host and client 'RTCPeerConnection's. This allows peers to discover their public IP addresses and attempt

Listing 4.4: *JSON Payload for Client Announce/Offer Structure*

```
{
  action: "announce", info_hash: this.roomId,
  peer_id: this.clientId,
  offers: [ { offer: { type: "offer", sdp: { /* SDP Payload */} },
              offer_id: this.offerID } ],
  numwant: 30
}
```

direct connections. TURN servers were not configured or utilized in this implementation, meaning connections may fail in complex network scenarios (e.g., symmetric NATs).

4.4.3 Timestamping for Latency Measurement

To enable accurate end-to-end latency measurement, a timestamping mechanism using WebRTC Insertable Streams [14] was implemented:

- **Host (‘EmbedTimeTransform.cs‘):** An ‘RTCRtpScriptTransform‘ appends the current UTC millisecond timestamp to the end of each encoded video frame’s data payload before network transmission.
- **Client (‘worker.js‘):** A Web Worker uses ‘RTCRtpReceiver.createEncodedStreams()‘ to access the raw encoded frames. It extracts the appended timestamp from each frame’s data, compares it with the current time upon arrival (after jitter buffering), and calculates the transit plus buffering delay ($t_6 - t_3$) 5.1. These delay values are collected for analysis.

4.5 Security and Privacy Considerations

The decentralized nature and technologies used in GameBeam introduce specific security and privacy considerations:

- **IPFS Daemon Permissions:** The ‘IPFSHostingService‘ in the Unity SDK launches a local IPFS daemon process. This process requires network and disk access permissions to connect to the IPFS network. Running local daemons introduces attack surface related to the IPFS implementation itself.
- **WebTorrent Tracker Exposure:** Using public WebTorrent trackers for signaling means the session’s ‘info_hash‘ (roomID) and participants’ ‘peer_id‘s are announced publicly or semi-publicly, depending on the tracker’s implementation. While peer IPs might not be directly listed in standard tracker responses (peers connect directly after discovery), interaction with the tracker reveals the host’s and guests’ IP addresses to the tracker operator. There is also potential for malicious actors on the tracker to observe active sessions or potentially attempt to interfere with signaling messages, although WebRTC connection establishment itself is secured. Hardening the tracker connection process and implementing additional authentication measures is a topic for future work.
- **Direct P2P Connections (No TURN):** Relying solely on STUN means direct P2P connections are attempted. While WebRTC connections (DTLS for data, SRTP for media) are encrypted end-to-end between authenticated peers, establishing the direct connection necessarily reveals the public IP addresses of the host and guest(s) to each other. The absence of TURN server usage prevents connections in some

network topologies but also avoids routing potentially sensitive game stream data through a third-party relay server and additional latency.

- **Data Channel Security:** Inputs are sent over an encrypted WebRTC data channel. However, the host application implicitly trusts the inputs received from connected guests. There are no specific measures within the SDK itself to validate or sanitize inputs against malicious data injection beyond what the game's logic might implement.
- **Client Code Integrity:** While IPFS [17] provides content addressing (ensuring the fetched client code matches the CID), relying on public gateways means users must trust the gateway not to tamper with the code during retrieval (though HTTPS (HTTP over TLS [46]) helps mitigate this for the gateway connection itself).

These factors highlight trade-offs between decentralization, accessibility, and security inherent in the current design.

4.6 Extensibility

The GameBeam Unity SDK was designed with some basic extensibility points for developers:

- **Event Handling (‘IClientHandler‘):** The primary integration point is the ‘IClientHandler‘ interface, allowing developers to react to ‘ClientJoined‘ and ‘ClientLeft‘ events to manage game state and player representations.
- **Input System (‘GameBeamInput‘):** Developers can subscribe to various input events (‘OnKeyDown‘, ‘OnMouseMove‘, etc.), use polling methods (‘GetKey‘, ‘GetMouseButton‘) provided by the ‘GameBeamInput‘ component associated with each ‘We-

bRtcClient‘ or use the helper classes provided by the SDK. This allows flexible mapping of guest inputs to game actions.

- **Hosting Service (‘IHostingService‘):** The hosting service mechanism allows for different client distribution strategies (IPFS, local, custom web server) by implementing the ‘IHostingService‘ interface.
- **Signaling Service (‘ISignaling‘):** While ‘WebtorrentSignaling‘ is the default, the ‘ISignaling‘ interface could potentially allow integration with alternative signaling mechanisms in the future. Public MQTT [47] brokers could also be utilized as an alternative signaling service.
- **Configuration Overrides:** Key parameters like resolution, frame rate, audio enablement, and hardware encoding usage can be overridden. Custom STUN/TURN servers are not directly exposed via configuration in this version but could be modified within ‘WebRtcClient.cs‘. Encoder parameters beyond hardware enablement are not exposed for fine-tuning.

4.7 Third-Party Licenses

The GameBeam implementation utilizes several third-party components, whose licenses are acknowledged here:

- **Unity WebRTC Package (‘com.unity.webrtc‘):** Licensed under the Apache License 2.0.
- **IPFS (Kubo / Go IPFS):** Dual-licensed under the MIT License and Apache License 2.0.
- **WebSocketSharp (Used by Unity Host):** Licensed under the MIT License.

The core GameBeam SDK and client code developed for this project are intended to be released under an MIT license as stated in the contributions.

4.8 Implementation Challenges

Developing GameBeam presented several technical challenges:

- **NAT Traversal:** Establishing direct P2P connections using only STUN proved insufficient for all network types, limiting connectivity compared to scenarios where TURN relays are available.
- **Signaling Reliability:** Dependence on the availability and performance of external WebTorrent trackers remained a concern. While the implemented fast fallback mechanism mitigates the impact of a single tracker failure, widespread tracker outages or poor performance across multiple trackers could still hinder session establishment.
- **IPFS Hosting Nuances:** Slow initial client load times via public gateways and the complexity of managing the host IPFS daemon affected user experience and setup.
- **Browser Compatibility:** Variations in WebRTC support, particularly iOS Safari's fullscreen limitations, restricted platform compatibility [48].
- **Host Resource Management:** Optimizing CPU/GPU usage for game execution and concurrent media encoding was crucial for performance and scalability.
- **Debugging Complexity:** Diagnosing issues in the distributed P2P system across diverse networks proved difficult.
- **Latency Optimization:** Achieving minimal end-to-end latency required careful implementation, including the timestamping mechanism, and balancing various system trade-offs.

- **Unity Integration:** Interfacing with Unity's audio system required custom solutions like the 'MultiAudioListener'.

5

Performance Evaluation Methodology

This chapter details the methodology employed to quantitatively evaluate the performance of the GameBeam framework. The primary goal of this evaluation is to characterize the performance of P2P game streaming using GameBeam under various conditions. Key performance aspects, including end-to-end latency, streaming quality, resource utilization, and scalability, were measured using a purpose-built automated testing framework.

While a key motivation for GameBeam is simplifying multiplayer development, a formal evaluation of developer usability was beyond the scope of this research phase due to time constraints and the prioritization of core performance analysis.

5.1 GameBeam Performance Analysis Framework

To ensure consistent, reproducible, and comprehensive performance testing, a dedicated **GameBeam Performance Analysis Framework**[22] was developed. This framework automates the process of running GameBeam tests, collecting diverse metrics, and storing results for analysis.

5.1.1 Purpose and Goals

The framework was designed with the following objectives:

- Automate the execution of GameBeam test runs with varying configurations (game, resolution, client count, frame rate, video encoding method).
- Systematically collect performance metrics from the host system (CPU, Memory, GPU, Network), the client-side WebRTC[14] metrics (jitter, packet loss, RTT), and end-to-end input latency.
- Store all collected configuration parameters and time-series metric data efficiently in a structured PostgreSQL database for subsequent detailed analysis.

5.1.2 Architecture and Components

The framework, implemented in TypeScript[38] using Node.js[49] (v22.14.0), consists of several core modules (referencing filenames in the `src/` directory of the analysis repository):

- **Test Runners** (`commands/batch.ts`, `commands/variant.ts`): Orchestrate the overall test execution based on JSON configuration files (`batch-config-combinations.json`, `variant-config.json`). They manage the sequence of parameter combinations, handle test repetitions (`runsPerConfig`), and implement retry logic (`maxRetries`) for failed runs. Two modes were used: 'batch' for exploring parameter combinations and 'variant' for comparing single parameter changes against a baseline with higher statistical power.
- **Unity Commander** (`unityCommander.ts`): Manages the lifecycle of the GameBeam host application. It launches the specified game executable (`game1` or `game2`) with

command-line arguments corresponding to the test configuration (e.g., resolution, frame rate, hardware encoding, audio enable/disable). It communicates with the running game via a WebSocket to receive status updates, including the crucial session connection link generated by the GameBeam SDK after initializing with IPFS[17] and the WebTorrent[18] tracker.

- **Game Performance Monitor** (`gamePerformanceMonitor.ts`): Monitors the host system resources while the Unity application is running. It utilizes Windows Management Instrumentation (WMI) via `typeperf`[50] for general system metrics (CPU Usage, Private Working Set Memory, Network I/O Bytes/Packets per second, basic GPU Utilization) and leverages `nvidia-smi`[45] for detailed NVIDIA GPU statistics (SM Utilization, Memory Utilization, Encoder/Decoder Utilization, Power Draw, Temperature, Clock Speeds) on the host's GPU. Data is collected at 1-second intervals.
- **Client Commander** (`clientCommander.ts`): Manages remote browser-based clients. It uses `puppeteer-core`[51] to connect to and automate Chrome instances hosted on the Testable.io cloud platform[52] (in the AWS `us-east-1` region). For each test, it navigates the required number of client browsers to the GameBeam session URL provided by the `UnityCommander`. It handles client-side interactions (e.g., initiating fullscreen) and orchestrates metric collection. This includes executing a predefined input sequence (`raceRecording.ts`) to simulate gameplay actions, downloading the `chrome://webrtc-internals` statistics dump at the end of each run, and retrieving end-to-end latency measurements recorded by custom JavaScript instrumentation.

- **WebRTC Dump Parser** (`webrtcDumpParser.ts`): Processes the JSON data dump obtained from `chrome://webrtc-internals`. It extracts detailed time-series metrics related to the video, audio, and data channels for the active, successful P2P connection (identified via the succeeded candidate pair).
- **Database Manager** (`db.ts`, `db/schema/schema.ts`, `db/schema/relations.ts`): Uses the Drizzle[53] to interact with a PostgreSQL[54] (v17) database. It defines the database schema and handles the insertion of run configuration details and all collected time-series metrics into appropriately indexed tables.
- **Logger** (`logger.ts`): Provides structured console output for monitoring the framework's execution progress and debugging.

5.1.3 Automated Test Execution Flow

A typical automated test run for a single configuration within a batch proceeds as follows:

1. The Test Runner selects the next configuration (e.g., game1, 2 clients, 1440p, 60fps, hardware encoding enabled, audio enabled).
2. `UnityCommander` launches the designated game executable with corresponding command line arguments.
3. `GamePerformanceMonitor` starts collecting host system and GPU metrics.
4. The Unity application initializes, integrates with the GameBeam SDK, hosts the client web application code on IPFS, registers with the WebTorrent tracker (`tracker.gamebeam.org`), generates a unique session link, and communicates this link back to `UnityCommander`.
5. ClientCommander initiates the specified number of client browser instances via

Testable.io, directing each to the session link.

6. Each client browser fetches the application code from IPFS, connects to the WebTorrent tracker to discover the host, and establishes a direct WebRTC P2P connection. The framework monitors for connection success within a timeout period (120 seconds).
7. Once all clients are connected and ready, `UnityCommander` signals the host game to begin, and `ClientCommander` starts replaying the predefined input sequence (`raceRecording.ts`) on each client. A 'gameStartedAt' timestamp is recorded.
8. During the test 'duration' (typically 60 seconds), host metrics are logged continuously by `GamePerformanceMonitor`, and client-side latency measurements are captured periodically by the instrumented JavaScript. Analysis typically excludes the first 5 and last 5 seconds to focus on steady-state performance.
9. Upon completion of the duration `ClientCommander` triggers the download of the `chrome://webrtc-internals` dump from each client and additional latency related metrics.
10. `UnityCommander` terminates the host game process; `GamePerformanceMonitor` stops data collection.
11. The `WebRTCDumpParser` processes the statistics dumps.
12. All collected metrics and configuration details are written to the PostgreSQL database, linked by a unique run ID.
13. The framework pauses briefly before proceeding to the next configuration or initiating a retry if the run failed (up to 'maxRetries').

5.2 Experimental Setup

All performance tests were conducted using the following hardware, software, and network configurations:

5.2.1 Hardware

- **Host Machine:**
  - CPU: AMD Ryzen 7 7800X3D (8-core, 16-thread)
  - RAM: 64 GB DDR5
  - GPU: NVIDIA GeForce RTX 4090 (24GB GDDR6X)
  - Storage: NVMe SSD
  - OS: Microsoft Windows 11 Pro (Version 24H2, 64-bit)
- **Client Machines:**
  - Provider: Testable.io via Amazon AWS
  - Region: `us-east-1` (N. Virginia)
  - CPU: 2 vCPUs (Intel Haswell E5-2676 v3 equivalent)
  - RAM: 4 GB
  - OS: Linux
  - Browser: Google Chrome (latest stable version available via Testable.io during testing)

5.2.2 Software

- Game Engine: Unity[19]6000.0.33f1

- GameBeam SDK.
- Game1: A Racing Game developed with Unity using Built-in Render Pipeline (Lightweight graphics, relatively less resource demanding).
- Game2: A Racing Game developed with Unity using Universal Render Pipeline (More resource intensive).
- Analysis Framework: Node.js v22.14.0.
- Database: PostgreSQL v17.

5.2.3 Network Conditions

- The host machine was connected via a residential 1 Gbps symmetric fiber optical network connection. Clients resided within the AWS `us-east-1` datacenter infrastructure.
- **Time Synchronization:** To enable accurate calculation of one-way and end-to-end latencies, high-precision time synchronization was implemented. The host machine synchronized with a local Stratum 1 NTP server using GPS as a reference clock, achieving sub-millisecond accuracy relative to UTC. The client virtual machines utilized the Amazon Time Sync Service, providing microsecond-level accuracy to UTC within the AWS environment.
- **Signaling:** WebRTC signaling for peer discovery and session negotiation relied exclusively on a custom public WebTorrent tracker deployed at `https://tracker.gamebeam.org/`. No significant tracker reliability issues were observed during the testing periods.
- No artificial network impairments (e.g., packet loss injection, added latency, band-

width throttling) were applied; performance was measured over the real-world internet path between the host and clients.

5.2.4 Games Used for Testing

- Two distinct Unity applications, referred to as **game1** and **game2**, were used. Both are 3D racing games developed using assets and templates from the Unity Asset Store, modified to integrate the GameBeam SDK for multiplayer streaming.
- **game2** was intentionally designed or selected to be more graphically demanding than **game1**, resulting in higher baseline CPU and GPU utilization on the host.
- *Rationale:* Using these applications provided a realistic testbed with representative graphical workloads. They also served as a practical case study for integrating GameBeam into existing single-player game templates, informing the understanding of potential performance implications.

5.3 Test Configuration Parameters

The performance evaluation systematically varied several parameters to isolate their impact on the system. The rationale for selecting the range or options for each parameter is as follows:

- **Client Count:** $[0, 1, 2, 3]$. *Rationale:* 0 clients established a baseline for host resource usage without streaming. 1-3 clients represented common small-group multiplayer sizes. Testing indicated that for these graphically intensive games, host CPU/GPU resources, particularly encoding capacity, became the primary bottleneck beyond 3 clients, making this a practical upper limit for evaluation within the scope of non-

MMO P2P streaming.

- **Resolution:** [1280x720 (720p), 1920x1080 (1080p), 2560x1440 (1440p)]. *Rationale:* Covered standard display resolutions commonly targeted by games, directly influencing host rendering load and the bitrate requirements for video streaming.
- **Frame Rate:** [30, 60 FPS]. *Rationale:* Represented common performance targets for cloud-based games, impacting host rendering/encoding demands and influencing the perceived smoothness and latency of the stream.
- **Duration:** 60 seconds (steady-state analysis window within each run). *Rationale:* Preliminary analysis showed system metrics (CPU, GPU, network throughput) generally stabilized within the first few seconds. A 60-second measurement window (excluding 5s start/end buffers) provided sufficient data for stable average/median calculation while keeping total batch test times manageable. Figure 8.1 in Appendix presents plots illustrating the trend of run-to-run variability for key metrics over the 60-second run duration. These plots demonstrate that variability generally stabilizes after the initial 5-second period, supporting the exclusion of start/end buffers and validating the chosen measurement window.
- **Audio Streaming:** [Enabled, Disabled]. *Rationale:* Allowed for quantification of the bandwidth and processing overhead specifically attributable to the audio stream transmission and processing.
- **Hardware Video Encoding (Host):** [Enabled (NVENC), Disabled (Software/CPU-based)]. *Rationale:* Comparison to evaluate the significant performance benefits (reduced CPU load, potentially lower latency) of using the dedicated NVENC hardware encoder on the host NVIDIA GPU versus relying on CPU-based encoding.

- **Runs Per Configuration:** 30 runs in 'variant' mode, 6 runs in 'batch' mode. *Rationale:* Multiple runs were essential to mitigate the impact of transient network or system variations. The higher count (30) in 'variant' mode provided increased statistical confidence when analyzing the effect of changing a single parameter relative to the baseline.
- **Input Simulation:** A fixed, pre-recorded sequence of keyboard inputs (`raceRecording.ts`) simulating driving actions in the racing games was used consistently across all clients and runs. *Rationale:* Ensured identical and repeatable gameplay actions, allowing for fair comparison between configurations by minimizing performance variations caused by differing user input patterns.

5.4 Measured Performance Metrics

The analysis framework collected a diverse set of metrics, stored in the PostgreSQL database, enabling a comprehensive performance assessment. Key metric categories and their rationale are:

- **Host Resource Utilization:** Measured by `GamePerformanceMonitor`. *Rationale:* To quantify the computational and network load imposed on the host machine by the game engine and the GameBeam streaming processes. Key metrics include host CPU usage, GPU utilization, GPU power consumption, application private memory usage, and total network traffic. These metrics help identify potential host-side bottlenecks.
- **WebRTC Streaming Quality:** Extracted by `WebRTCDumpParser` from client-side `chrome://webrtc-internals` dumps. *Rationale:* To evaluate the quality, stability, and ef-

ficiency of the P2P audio-video stream delivery as perceived by the client. Important indicators include video/audio packets lost, video frames dropped, video frames received rate, jitter buffer delay, video decode delay, and candidate pair round-trip time.

- **Network Bandwidth Consumption:** Measured at both the host and aggregated across clients. *Rationale:* To understand the network footprint of the GameBeam stream under different configurations, particularly the impact of resolution, frame rate, and client count on bandwidth requirements.
- **End-to-End Latency:** Calculated by summing the durations of key pipeline stages, identified by timestamps t_1 through t_7 in Figure 5.1. *Rationale:* This is a critical metric for user experience, representing the total time from a frame completing rendering on the host (at t_1) to that frame being displayed on the client (approximated by decode completion at t_7). The framework calculates this total latency by summing the following component intervals, derived using synchronized clocks and WebRTC statistics:

  1. **Host Processing Interval ($t_3 - t_1$):** This interval covers the time from the frame finishing rendering in the game engine (t_1) through its submission to the video encoder (step 1, occurring between t_1 and t_2) and the completion of the encoding process (step 2, finishing at t_3). Timings related to rendering, queuing, and encoding within this interval are typically derived from WebRTC statistics, such as those provided by the `googTimingFrameInfo` extension.
  2. **Transit and Buffering Interval ($t_6 - t_3$):** This crucial interval measures the time from the frame finishing encoding on the host (t_3) until it is ready for

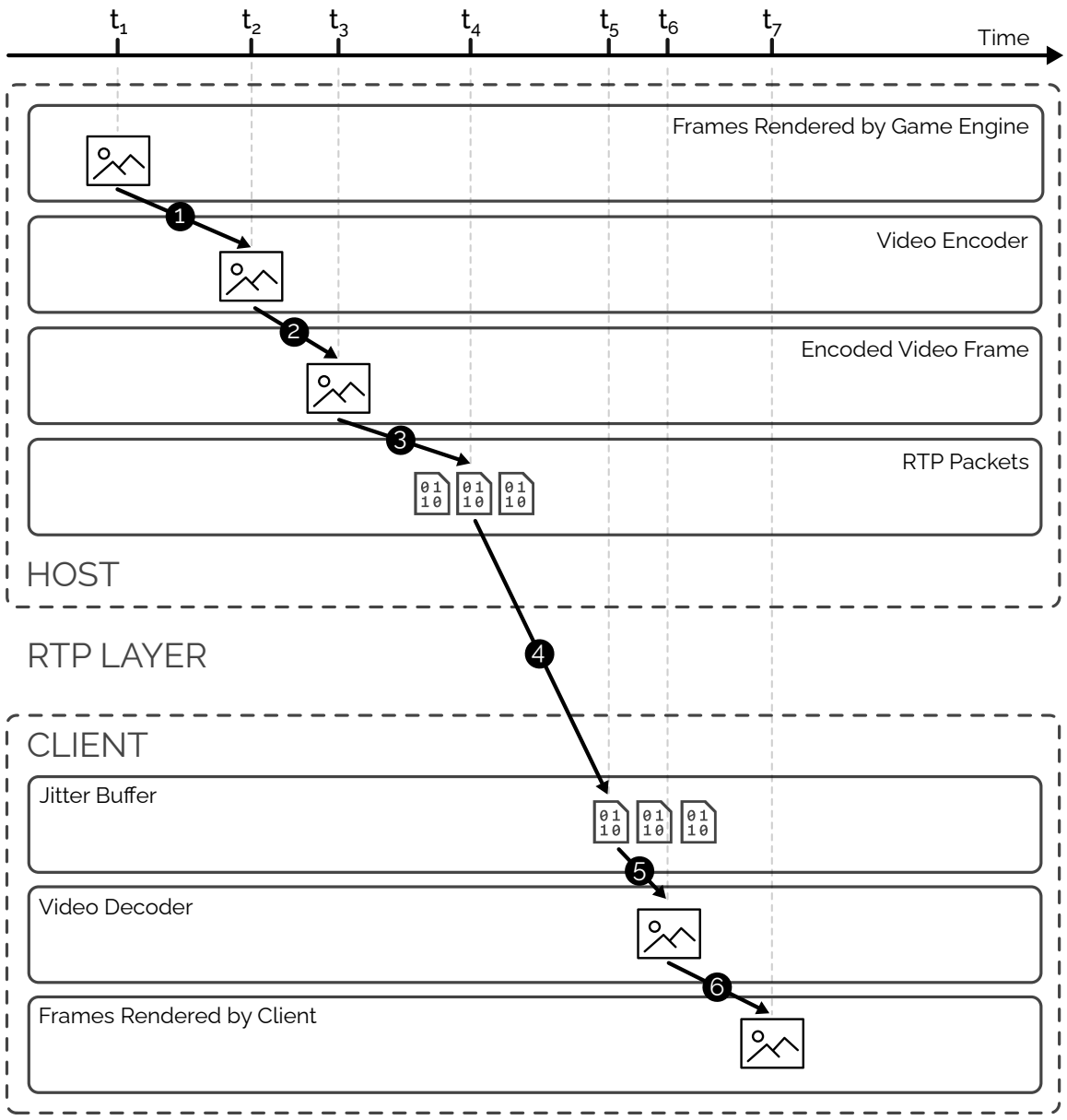

Figure 5.1: *Conceptual breakdown of the video frame pipeline and associated timestamps* $(t_1..t_7)$ *used for End-to-End Latency calculation. The total latency* $(t_7 - t_1)$ *is calculated by summing the durations of key intervals: host processing* $(t_3 - t_1)$*, transit and buffering* $(t_6 - t_3)$*, and client decoding/presentation* $(t_7 - t_6)$*.*

decoding on the client (t_6). It encompasses host-side packetization (step 3), network transmission time across the RTP layer (step 4, roughly $t_5 - t_3$), and client-side jitter buffering (step 5, duration $t_6 - t_5$). To accurately measure this interval despite clock differences between machines, GameBeam utilizes high-precision UTC time synchronization (GPS Stratum 1 NTP on host, Amazon Time Sync on clients) combined with the WebRTC Encoded Transform API. The host embeds the absolute UTC timestamp of encode completion (t_3) into the encoded frame's metadata. The client extracts this timestamp when the frame emerges from the jitter buffer ready for decoding (t_6) and calculates the difference $t_6 - t_3$. This directly measured duration is stored as the `delay` value in the `delayMeasurements` table.

3. **Client Decode and Presentation Interval ($t_7 - t_6$):** This interval represents the time spent by the client's video decoder (step 6) plus any subsequent delay until the frame is rendered or presented on the client's display (approximated as completed at t_7). Timings related to decoding and presentation within this interval are also typically derived from WebRTC statistics like `googTimingFrameInfo`.

The total end-to-end video latency is the sum of these distinct intervals:

$$\Delta t_{\text{end\_to\_end}} = (t_3 - t_1) + (t_6 - t_3) + (t_7 - t_6) = t_7 - t_1$$

The specific values contributing to the $(t_3 - t_1)$ and $(t_7 - t_6)$ intervals are extracted from detailed WebRTC statistics, while the $(t_6 - t_3)$ interval is the custom, directly measured value using synchronized clocks.

5.5 Adjustments and Decentralization Impact

Compared to the initial proposal, the most significant architectural evolution evaluated was the shift to a **fully decentralized model**, impacting the methodology:

- **Client Distribution via IPFS:** The browser-based client application was hosted on the InterPlanetary File System (IPFS). Clients fetched the application code directly from the IPFS network, eliminating reliance on a centralized web server for distribution.
- **Signaling via Public WebTorrent Tracker:** Initial peer discovery and WebRTC signaling relied entirely on a publicly accessible, WebTorrent tracker (`tracker.gameb eam.org`), replacing the originally proposed dedicated matchmaking server.

These architectural changes introduced factors inherent to decentralized systems into the evaluation scope:

- **Initial Connection Variability:** The time and success rate of establishing the initial P2P connection became subject to the performance of the IPFS network (for code fetching) and the responsiveness/availability of the public WebTorrent tracker. While not a primary focus of the steady-state analysis, the framework's connection timeout mechanism captured instances where this phase failed.
- **Reliance on Public Infrastructure:** The performance metrics obtained reflect the system operating over public, uncontrolled infrastructure (Internet path, IPFS network, public tracker), providing a realistic assessment of this decentralized approach but also incorporating external variability.

This detailed methodology, leveraging the custom automated framework, precise time synchronization, and accounting for the fully decentralized architecture, provides a robust foundation for the performance results presented and discussed in Chapter 6.

6

Results

This chapter presents the quantitative results obtained from the performance evaluation of the fully decentralized GameBeam framework. The data was collected using the automated **GameBeam Performance Analysis Framework** described in Chapter 5. The results focus on characterizing the system's performance across various configurations, specifically examining end-to-end latency, host resource utilization, network bandwidth consumption, and WebRTC streaming quality metrics. The findings presented here are based on the analysis of data stored in the PostgreSQL database, aggregated primarily using mean values and 95% confidence intervals across multiple runs per configuration to ensure robustness against outliers. Discussion of these results will follow in Chapter 7.

6.1 Local Single-Player Performance Baseline

Before evaluating the performance of the GameBeam streaming system, baseline metrics were collected for the two tech demo applications, `game1` and `game2`, running locally on the host machine without any streaming or networking components active. These met-

rics, collected using the host monitoring tools from the analysis framework (`typeperf`, `nvidia-smi`), represent the inherent resource demands of the games themselves on the test hardware. Table 6.1 summarizes these findings. This provides a crucial reference point for quantifying the overhead introduced by the GameBeam framework in subsequent sections.

Table 6.1: *Summary of Local Single-Player Performance Metrics.*

Metric	Game 1 (Mean ± 95% CI)	Game 2 (Mean ± 95% CI)
Host CPU Usage (%)	0.58 ± 0.02	2.57 ± 0.05
Host GPU Usage (% SM)	2.15 ± 0.04	10.48 ± 0.03
Host Memory (MB)	574.60 ± 0.08	993.19 ± 0.09
Host GPU Power (W)	84.48 ± 0.11	98.35 ± 0.09

6.2 Streaming Performance Baseline (1 Client)

To establish a reference point *for the streaming system*, a baseline configuration involving streaming to a single client was analyzed in detail. This configuration used: 1 client, 1080p resolution, 30 FPS target, hardware encoding, and audio enabled. Tests were conducted using both `game1` and `game2`.

Table 6.2 summarizes the key performance metrics (mean values $\pm$ 95% CI over 30 runs) for this 1-client streaming baseline configuration for both game demos. Comparing these values to the local baseline in Table 6.1 highlights the overhead introduced by the streaming process itself.

Figure 6.1 illustrates the distribution of end-to-end latency measurements observed during the baseline runs, providing insight into the variability of this critical metric. Similar stability was observed for other key metrics like frame rate and resource usage over the

Table 6.2: *Summary of Streaming Performance Metrics (1 Client, 1080p, 30 FPS, HW Enc, Audio On).*

Metric	Game 1 (Mean ± 95% CI)	Game 2 (Mean ± 95% CI)
Host CPU Usage (%)	1.22 ± 0.03	4.94 ± 0.07
Host GPU Usage (% SM)	4.01 ± 0.02	18.09 ± 0.03
Host Memory (MB)	620.15 ± 0.25	1074.99 ± 0.24
Host GPU Power (W)	85.83 ± 0.13	103.75 ± 0.15
Host Sent Bandwidth (KB/s)	326.81 ± 2.73	337.59 ± 0.32
End-to-End Latency (ms)	47.21 ± 0.58	52.47 ± 0.50
Client RTT (ms)	14.99 ± 0.08	14.88 ± 0.07
Video Frames Received/s	29.99 ± 0.02	29.99 ± 0.01
Video Frames Dropped (Total)	0.00	0.00
Video Packet Loss (Total)	0.00	0.00
Video Jitter Buffer Delay (ms)	42.75 ± 0.52	38.46 ± 0.54
Video Inter-Frame Delay (ms)	33.35 ± 0.01	33.34 ± 0.01
Video Inter-Frame Delay (St. Dev.)	4.39 ± 0.23	3.94 ± 0.24

60-second measurement window, as exemplified by the time-series analysis in Figure 6.2.

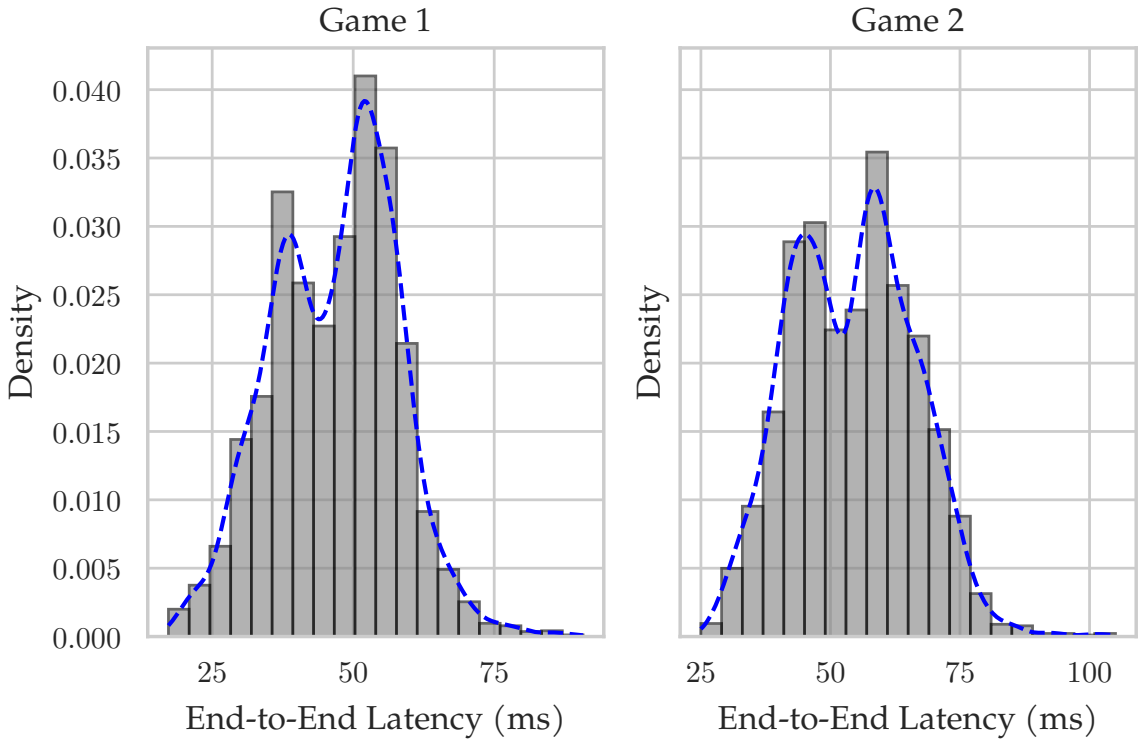

Figure 6.1: *Distribution of End-to-End Latency for Baseline Configuration (1 Client, 1080p, 30 FPS, HW Enc, Audio On).*

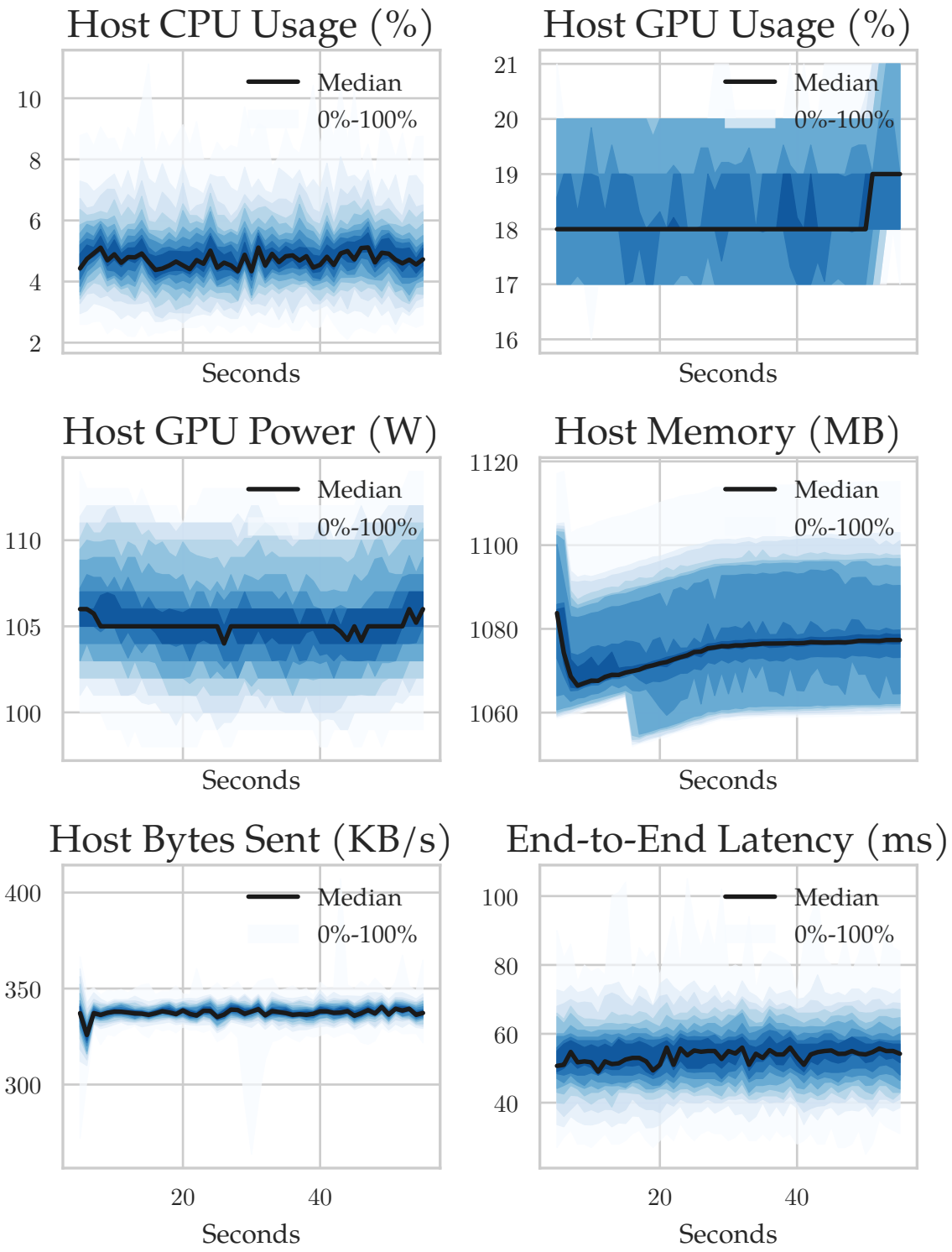

Figure 6.2: *Time-Series representation of key metrics during baseline runs (Game 2, 1 Client, 1080p, 30 FPS, HW Enc, Audio On). The solid black line indicates the median value across all runs at each second. The shaded blue regions represent percentile bands (0%–100% for the lightest band, converging inwards to the median) of the data distribution across these runs, illustrating the stability and run-to-run variability of key performance indicators over the 60-second measurement window.*

6.3 Impact of Client Count (Scalability)

The framework was tested with 0, 1, 2, and 3 simultaneous clients connecting to the host to assess scalability. The configuration was otherwise kept at baseline values (1080p, 30 FPS, HW Enc, Audio On).

Figure 6.3 shows the impact of increasing client count on end-to-end latency. End-to-end latency stayed relatively stable with the increasing count of clients showcasing the scalability performance of the architecture.

Figure 6.4 illustrates the total outgoing bandwidth from the host per connected client count. Host bandwidth scaled linearly with the number of clients, while per-client bandwidth remained relatively consistent.

6.4 Impact of Resolution

Performance was evaluated at 720p, 1080p, and 1440p resolutions, keeping other parameters at baseline (1 client, 30 FPS, HW Enc, Audio On).

Increasing resolution significantly impacted host GPU utilization and, to a lesser extent, CPU usage, as shown in Table 6.3. Network bandwidth consumption was not affected due to video stream being encoded with constant bitrate. End-to-end latency showed a notable increase at higher resolutions for Game 2, potentially due to increased GPU utilization of the game variant, illustrated in Figure 6.5.

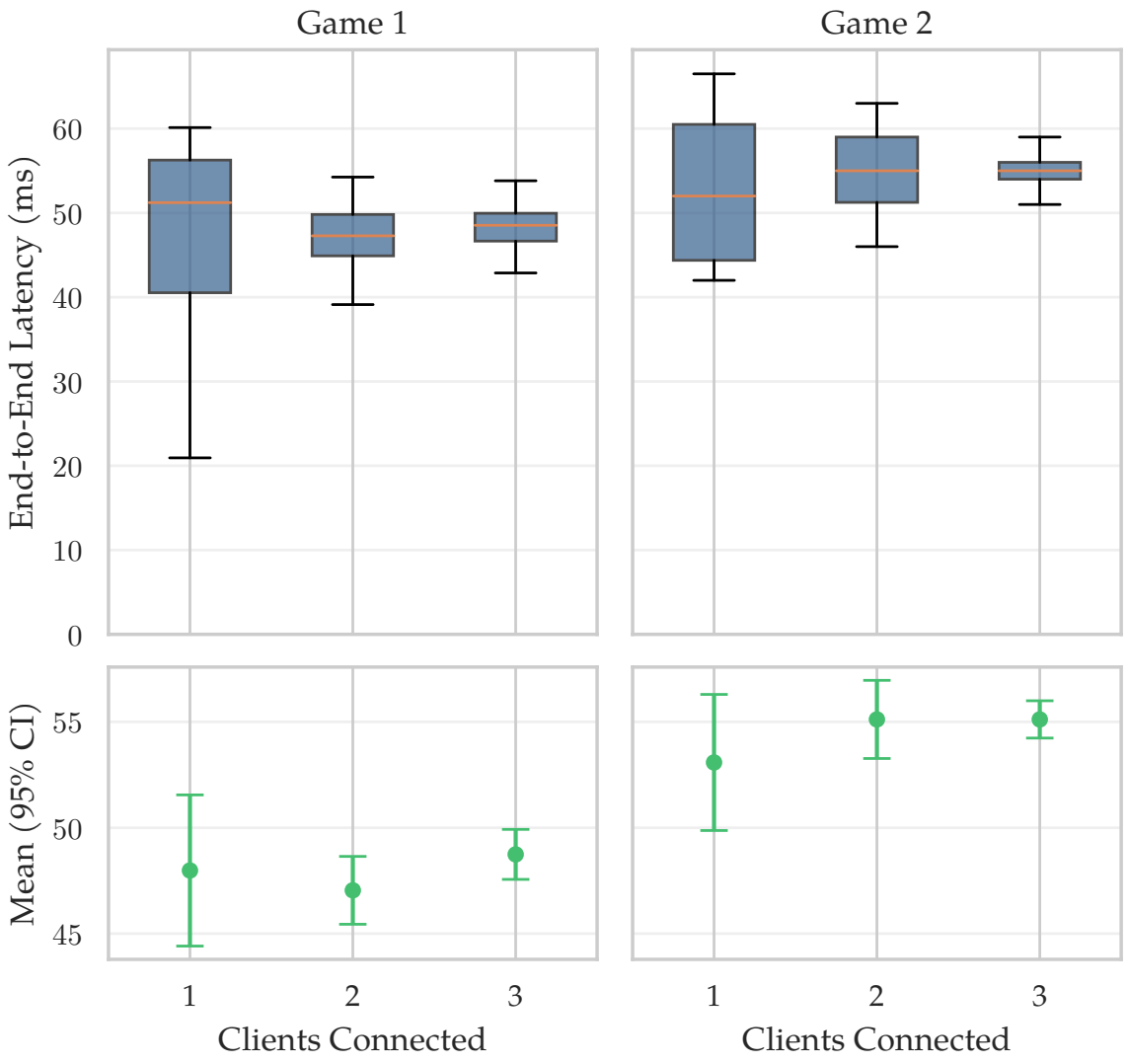

Figure 6.3: *End-to-End Latency vs. Number of Connected Clients, comparing Game 1 and Game 2. The top row shows box plots visualizing the distribution (median, IQR, range excluding outliers) of per-run latency medians. The bottom row shows error bar plots visualizing the mean latency and its 95% confidence interval for each group. (Configuration: 1080p, 30 FPS, HW Enc, Audio On).*

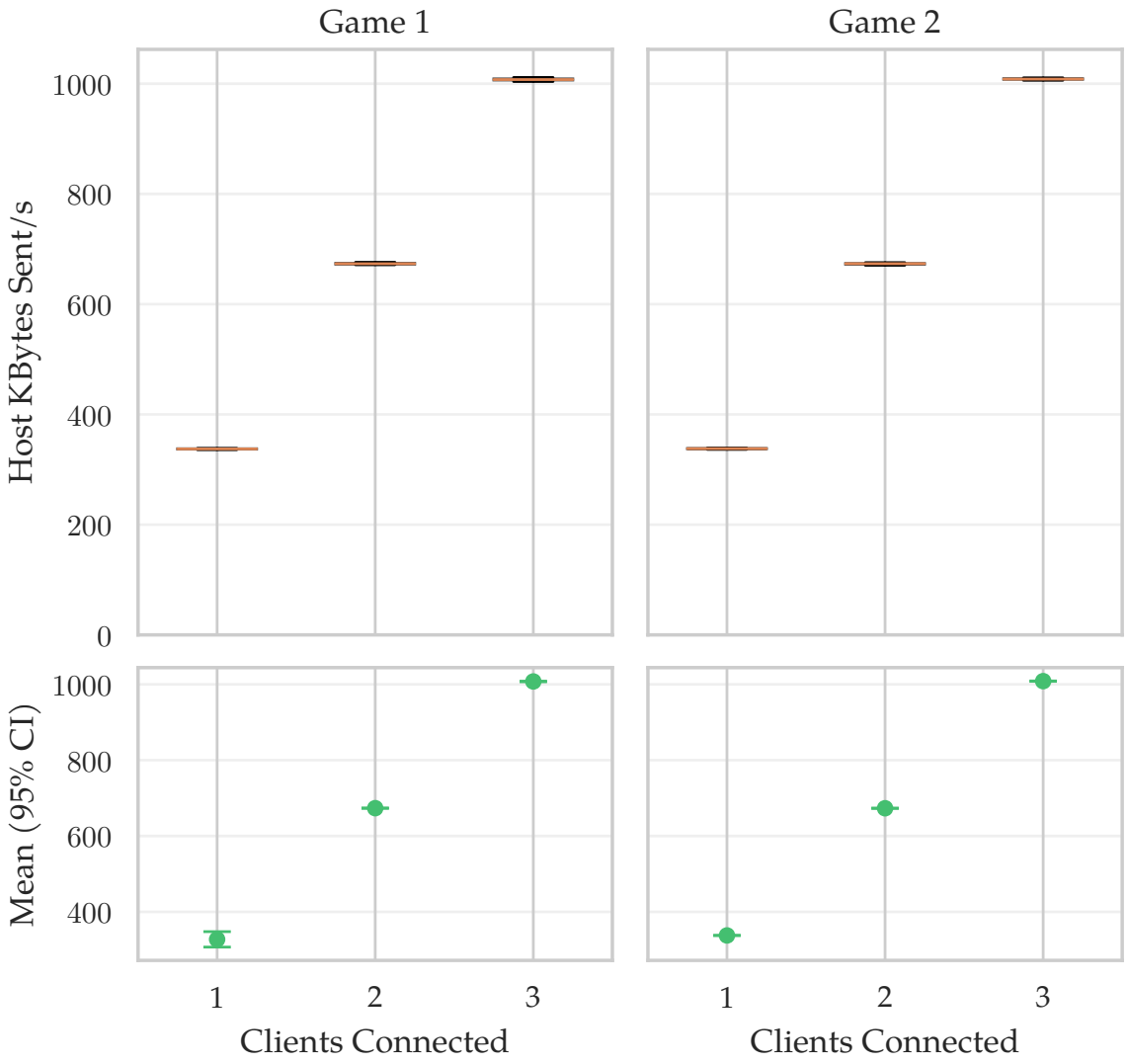

Figure 6.4: *Host Total Sent Bandwidth vs. Number of Connected Clients (1080p, 30 FPS, HW Enc, Audio On)*

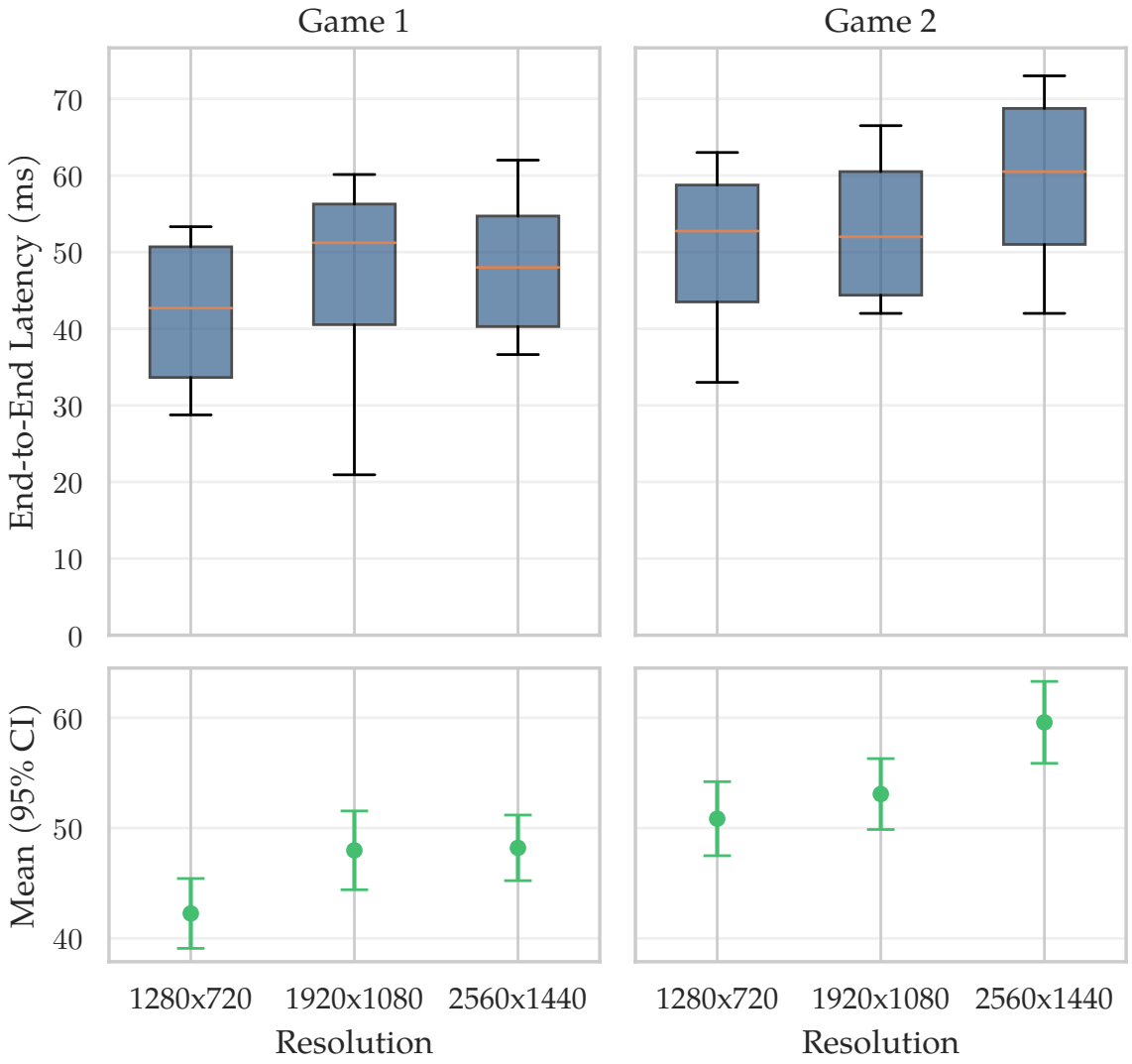

Figure 6.5: *End-to-End Latency vs. Streaming Resolution (1 Client, 30 FPS, HW Enc, Audio On) for Game 2.*

Table 6.3: *Impact of Resolution (1 Client, 30 FPS, HW Enc, Audio On, Game 2).*

Metric	720p (Mean ± 95% CI)	1080p (Mean ± 95% CI)	1440p (Mean ± 95% CI)
Host CPU Usage (%)	4.87 ± 0.07	4.94 ± 0.07	4.96 ± 0.07
Host GPU Usage (% SM)	17.15 ± 0.02	18.09 ± 0.03	20.01 ± 0.02
Host Memory (MB)	1060.43 ± 0.17	1074.99 ± 0.24	1096.87 ± 0.33
Host GPU Power (W)	103.44 ± 0.11	103.75 ± 0.15	109.20 ± 0.14
Host Sent Bandwidth (KB/s)	337.68 ± 0.32	337.59 ± 0.32	337.64 ± 0.29
End-to-End Latency (ms)	49.62 ± 0.53	52.47 ± 0.50	60.37 ± 0.63
Client RTT (ms)	14.93 ± 0.08	14.88 ± 0.07	15.35 ± 0.09
Video Frames Received/s	30.00 ± 0.01	29.99 ± 0.01	30.00 ± 0.02
Video Frames Dropped (Total)	0.00	0.00	0.00
Video Packet Loss (Total)	0.00	0.00	0.00
Video Jitter Buffer Delay (ms)	41.95 ± 0.54	38.46 ± 0.54	44.90 ± 0.49
Video Inter-Frame Delay (ms)	33.33 ± 0.01	33.34 ± 0.01	33.33 ± 0.01
Video Inter-Frame Delay (St. Dev.)	2.70 ± 0.21	3.94 ± 0.24	4.56 ± 0.21

6.5 Impact of Frame Rate

The target frame rate was tested at 30 FPS and 60 FPS (1 client, 1080p, HW Enc, Audio On).

Increasing the frame rate to 60 FPS resulted in higher host CPU and GPU utilization compared to 30 FPS, as expected due to the increased rendering and encoding demands (Table 6.4). Network bandwidth usage remained unchanged, as constant bitrate encoding was used. Notably, however, the end-to-end latency was significantly lower at 60 FPS than at 30 FPS, as illustrated in Figure 6.6. This reduction can be attributed to the shorter frame intervals at higher frame rates, which reduce the time between input capture and frame transmission, thereby improving overall responsiveness. This difference was statistically significant for both game demos, as confirmed by t-tests ($p < 0.000117$).

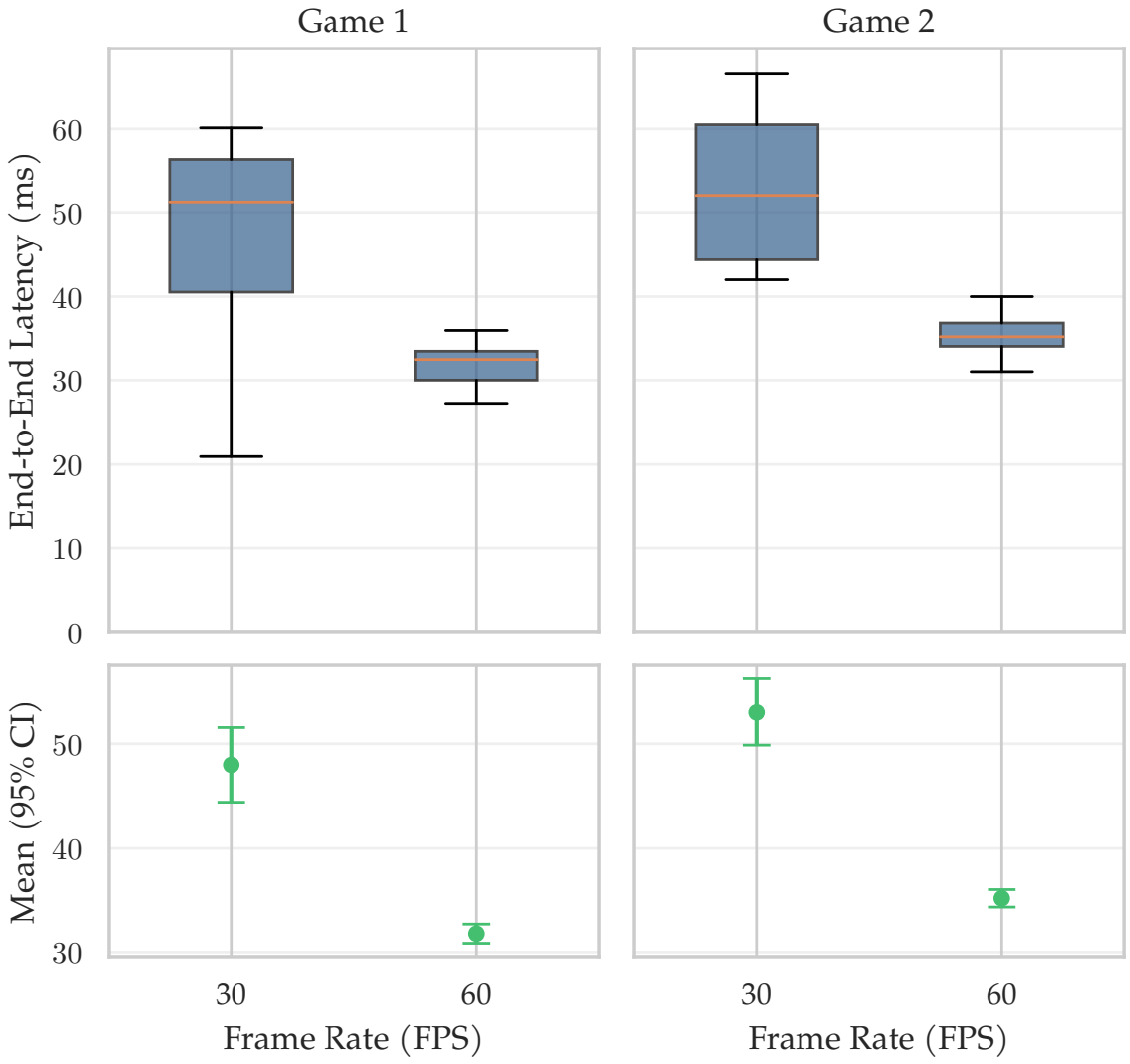

Figure 6.6: *End-to-End Latency vs. Target Frame Rate (1 Client, 1080p, HW Enc, Audio On)*

Table 6.4: *Impact of Frame Rate (1 Client, 1080p, HW Enc, Audio On, Game 2).*

Metric	30 FPS (Mean ± 95% CI)	60 FPS (Mean ± 95% CI)
Host CPU Usage (%)	4.94 ± 0.07	9.25 ± 0.10
Host GPU Usage (% SM)	18.09 ± 0.03	37.06 ± 0.05
Host Memory (MB)	1074.99 ± 0.24	1091.76 ± 0.26
Host GPU Power (W)	103.75 ± 0.15	132.87 ± 0.15
Host Sent Bandwidth (KB/s)	337.59 ± 0.32	337.44 ± 0.41
End-to-End Latency (ms)	52.47 ± 0.50	35.56 ± 0.31
Client RTT (ms)	14.88 ± 0.07	15.06 ± 0.08
Video Frames Received/s	29.99 ± 0.01	59.98 ± 0.02
Video Frames Dropped (Total)	0.00	0.00
Video Packet Loss (Total)	0.00	0.00
Video Jitter Buffer Delay (ms)	38.46 ± 0.54	21.15 ± 0.42
Video Inter-Frame Delay (ms)	33.34 ± 0.01	16.67 ± 0.00
Video Inter-Frame Delay (St. Dev.)	3.94 ± 0.24	4.80 ± 0.13

6.6 Impact of Hardware Encoding

The performance impact of using hardware-accelerated video encoding (NVIDIA NVENC) versus software (CPU-based) encoding was evaluated (1 client, 1080p, 30 FPS, Audio On).

As shown in Figure 6.7, enabling hardware encoding significantly reduced host CPU utilization while GPU utilization stayed the same as the GPU utilizes dedicated hardware for video encoding. Hardware encoding also resulted in lower end-to-end latency compared to software encoding (Figure 6.8), a difference confirmed as statistically significant ($p < 0.001$). Detailed results comparing hardware/software encoding are available in Appendix 8.1.

6.7 Impact of Audio Streaming

Tests were conducted with audio streaming enabled and disabled (1 client, 1080p, 30 FPS, HW Enc). As shown in Table 6.5 disabling audio resulted in a marginal decrease in to-

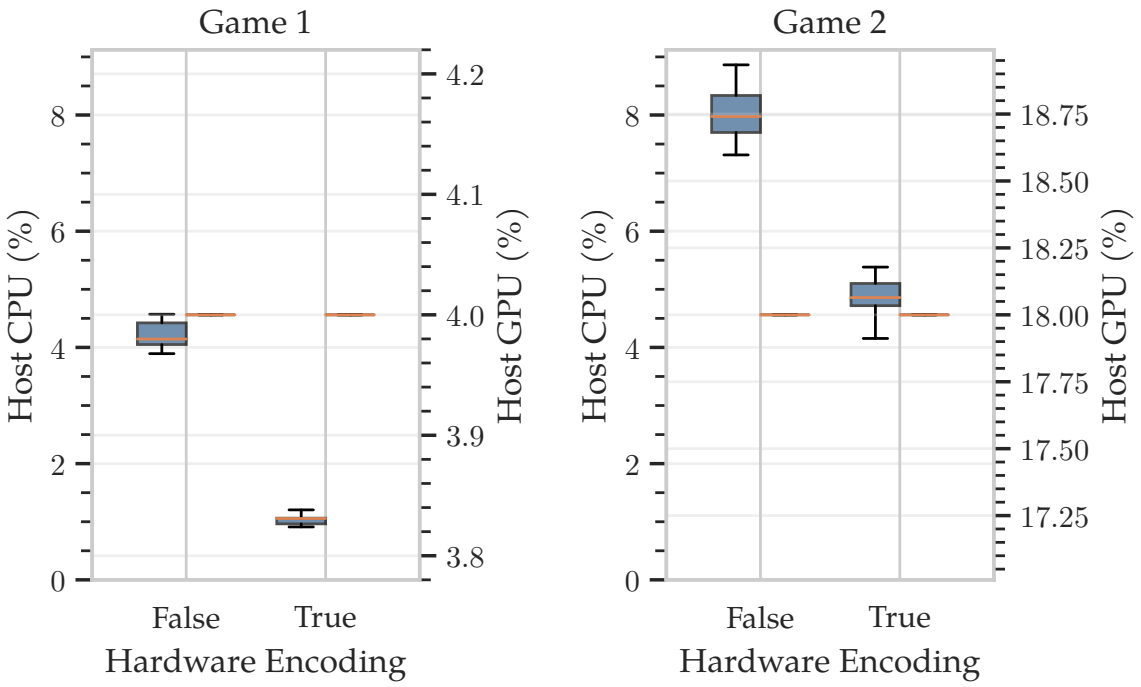

Figure 6.7: *Host CPU and GPU Utilization with Hardware vs. Software Encoding (1 Client, 1080p, 30 FPS, Audio On)*

tal network bandwidth consumption (approximately 8 KB/s). Minor impact was also observed on host CPU utilization.

Table 6.5: *Impact of Audio Streaming (1 Client, 1080p, 30 FPS, HW Enc, Game 2).*

Metric	Audio Off (Mean ± 95% CI)	Audio On (Mean ± 95% CI)
Host CPU Usage (%)	4.71 ± 0.07	4.94 ± 0.07
Host GPU Usage (% SM)	18.28 ± 0.02	18.09 ± 0.03
Host Memory (MB)	1070.80 ± 0.24	1074.99 ± 0.24
Host GPU Power (W)	105.72 ± 0.07	103.75 ± 0.15
Host Sent Bandwidth (KB/s)	329.63 ± 0.34	337.59 ± 0.32
End-to-End Latency (ms)	52.90 ± 0.45	52.47 ± 0.50
Client RTT (ms)	15.16 ± 0.07	14.88 ± 0.07
Video Frames Received/s	30.00 ± 0.02	29.99 ± 0.01
Video Frames Dropped (Total)	0.00	0.00
Video Packet Loss (Total)	0.00	0.00
Video Jitter Buffer Delay (ms)	38.66 ± 0.48	38.46 ± 0.54
Video Inter-Frame Delay (ms)	33.33 ± 0.01	33.34 ± 0.01
Video Inter-Frame Delay (St. Dev.)	3.75 ± 0.23	3.94 ± 0.24

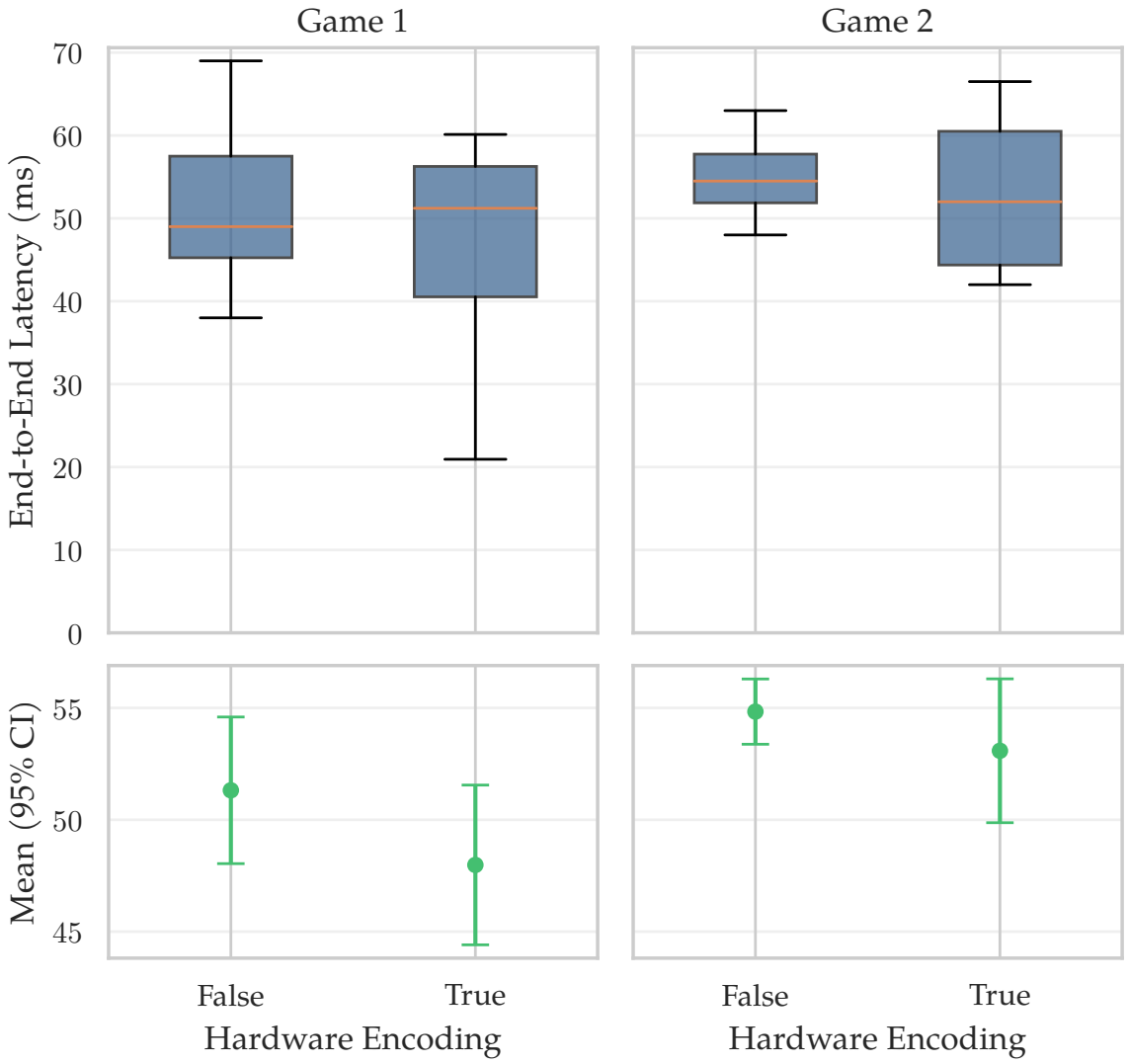

Figure 6.8: *End-to-End Latency with Hardware vs. Software Encoding (1 Client, 1080p, 30 FPS, Audio On)*

6.8 Effect Size Analysis

While the previous sections examined the impact of individual parameters on performance metrics, it is also useful to compare the *relative magnitude* of these impacts on a standardized scale. To achieve this, Cohen's d effect size was calculated for key parameters compared against baseline conditions, quantifying the standardized difference between means.

Cohen's d was calculated by comparing the following conditions, generally holding other parameters at the baseline established in Section 6.2 (1 Client, 1080p, 30 FPS, HW Enc, Audio On, using Game 2 data):

- **Clients:** 3 Clients vs. 1 Client.
- **Resolution:** 1440p vs. 720p (comparing extremes).
- **Frame Rate:** 60 FPS vs. 30 FPS.
- **Hardware Enc:** Software Encoding (False) vs. Hardware Encoding (True).
- **Audio:** Audio Disabled (False) vs. Audio Enabled (True).

The absolute values of the calculated Cohen's d are presented as a heatmap in Figure 6.9. The color intensity corresponds to the magnitude of the effect size, with brighter/yellower colors indicating a larger impact of the parameter change on the specific performance metric. Conventionally, Cohen's d values around 0.2 are considered small, 0.5 medium, and above 0.8 large effects.

Observing the effect sizes (absolute Cohen's d) presented in Figure 6.9, several key patterns emerge regarding the relative influence of the tested parameters on core performance metrics under the specified comparison conditions:

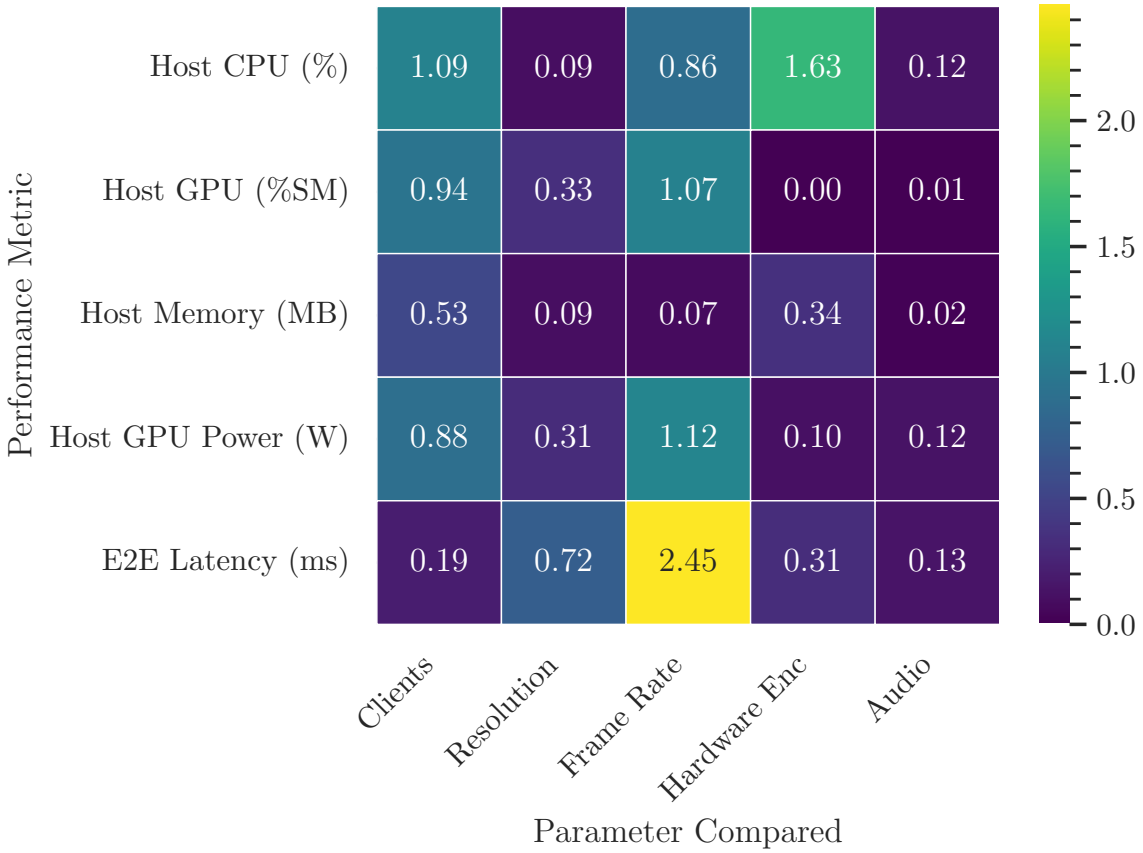

Figure 6.9: *Heatmap of Effect Sizes (Absolute Cohen's d) Showing the Relative Impact of Configuration Parameters on Key Performance Metrics. Comparisons made against baseline conditions.*

- **Frame Rate** exhibits the largest effect size ($d = 2.45$) on End-to-End Latency among all tested parameters. The next most influential parameter on latency is **Resolution** ($d = 0.72$), followed by Hardware Encoding ($d = 0.31$). The number of Clients and Audio setting showed smaller effects on latency in these comparisons ($d \leq 0.19$).
- **Hardware Encoding** has the most substantial effect ($d = 1.63$) on Host CPU Usage, significantly reducing it compared to software encoding, as expected. The number of **Clients** ($d = 1.09$) and **Frame Rate** ($d = 0.86$) also showed large effects on CPU usage.
- For Host GPU Usage (%SM) and Host GPU Power (W), **Frame Rate** shows the largest effect sizes ($d = 1.07$ and $d = 1.12$, respectively). The number of **Clients** also has a notable impact on both GPU Usage ($d = 0.94$) and Power ($d = 0.88$). **Resolution** showed only a small-to-moderate effect ($d \approx 0.3$) on these GPU metrics in this comparison. Hardware Encoding showed a negligible effect ($d = 0.00$) on GPU SM utilization itself, though it utilizes the dedicated encoder block.
- The number of **Clients** has a moderate effect ($d = 0.53$) on Host Memory usage, while other parameters like Resolution, Frame Rate, and Audio showed minimal effects ($d \leq 0.09$). Hardware Encoding had a small-to-moderate effect ($d = 0.34$) on memory.
- **Audio** streaming generally exhibits minimal or negligible effect sizes ($d \leq 0.13$) on these core host resource and latency metrics compared to the impact of varying clients, frame rate, resolution, or encoding method.

6.9 Streaming Quality Analysis

The quality and temporal consistency of the video stream received by clients were assessed using metrics derived from WebRTC statistics, including packet loss, frame drops, jitter buffer delay, and inter-frame delay characteristics. Table 6.6 summarizes these metrics as a function of the number of connected clients under baseline streaming conditions (Game 2, 1080p, 30 FPS, HW Enc, Audio On).

Under these specific test conditions, mean total dropped video frames were measured at zero for all client counts. Mean video packet loss was also zero for 1 and 2 clients, with a negligible mean value of 0.01% packets lost over the measurement period observed for the 3-client configuration. This suggests reliable packet transport and adequate client-side processing capacity within this baseline scenario.

The mean video jitter buffer delay exhibited a minor increase as the number of clients grew, rising from `38.46` ms for one client to `44.01` ms for three clients. This indicates a slightly longer buffering period was necessary, likely to compensate for potentially increased network timing variations associated with managing multiple concurrent streams.

Inter-frame delay metrics offer insight into the perceived smoothness of the video playback. The WebRTC jitter buffer aims to mitigate variations in network packet arrival times to ensure a steady output to the decoder. The results align with this function: the mean average video inter-frame delay was consistently measured at approximately `33.33` ms, closely matching the expected interval for the 30 FPS target frame rate. Furthermore, the standard deviation of the inter-frame delay, which quantifies the jitter or inconsistency in frame presentation timing, remained low across the tested client counts, with mean val-

ues ranging narrowly between 3.73 ms and 4.05 ms. These low standard deviation values suggest the jitter buffer successfully smoothed out packet arrival variations, resulting in consistent frame pacing after the buffering stage for these tests.

Table 6.6: *Impact of Client Count* (*Game 2, 1080p, 30 FPS, HW Enc, Audio On*).

Metric	1 Client (Mean ± 95% CI)	2 Clients (Mean ± 95% CI)	3 Clients (Mean ± 95% CI)
End-to-End Latency (ms)	52.47 ± 0.50	54.54 ± 0.37	54.98 ± 0.30
Client RTT (ms)	14.88 ± 0.07	15.00 ± 0.05	14.97 ± 0.05
Video Frames Received/s	29.99 ± 0.01	29.99 ± 0.01	29.99 ± 0.01
Video Frames Dropped (Total)	0.00	0.00	0.00
Video Packet Loss (Total)	0.00	0.00	0.01 ± 0.00
Video Jitter Buffer Delay (ms)	38.46 ± 0.54	41.84 ± 0.33	44.01 ± 0.32
Video Inter-Frame Delay (ms)	33.34 ± 0.01	33.34 ± 0.01	33.35 ± 0.01
Video Inter-Frame Delay (St. Dev.)	3.94 ± 0.24	4.05 ± 0.17	3.73 ± 0.12

6.10 Summary of Results

The quantitative evaluation demonstrated the performance characteristics of the decentralized GameBeam framework. Key observations include:

- End-to-end latency under baseline conditions (1 client, 1080p, 30fps, HW enc) was measured at a mean of 47.21 ms for game1 and 52.47 ms for game2 (Table 6.2).
- Host resource utilization (CPU, GPU) scaled with the number of clients, resolution, and frame rate, with hardware encoding significantly reducing CPU load. GPU utilization (SM %) was the primary limiting factor for scalability with multiple clients in graphically demanding scenarios.
- Network bandwidth scaled linearly with client count and was not influenced by resolution and frame rate choices. Audio streaming added a marginal overhead.
- Increasing frame rate from 30 to 60 FPS led to slightly lower end-to-end latency,

despite increased resource load.

- Streaming quality remained generally high, with effectively zero packet loss (mean $\leq 0.01\%$ total packets lost) and zero frame drops under the tested conditions (Table 6.6).
- Effect size analysis (Cohen's d) identified the target frame rate as the parameter with the largest impact on end-to-end latency ($d = 2.45$) and host GPU utilization ($d \approx 1.1$) among the tested configurations (Figure 6.9).
- Hardware encoding provided lower end-to-end latency and demonstrated the most substantial positive effect on host CPU utilization ($d = 1.63$), dramatically reducing CPU load compared to software encoding, while having a negligible direct impact on GPU compute unit (%SM) usage ($d = 0.00$).(Figure 6.9)

These findings provide a quantitative basis for discussing the feasibility, performance trade-offs, and potential applications of the GameBeam framework in the following chapter.

Okay, based on the provided chapters and the requested outline, here is a draft for Chapter 7. I have focused on maintaining the formal, technical, and objective tone established in the previous sections, summarizing the key findings and contributions while realistically assessing feasibility and outlining specific, grounded future work directions.

7

Conclusion and Future Work

This thesis introduced GameBeam, a novel framework designed to explore the potential of fully decentralized peer-to-peer (P2P) game streaming. Motivated by the desire to overcome the limitations of conventional installation-based multiplayer and the operational costs and potential latency issues of centralized cloud gaming platforms, GameBeam leverages existing P2P technologies—WebRTC, IPFS, and public WebTorrent trackers—to enable installation-free access for guest players and simplify the multiplayer development process. This concluding chapter summarizes the research undertaken, offers concluding remarks on the feasibility and potential of this approach, and outlines promising avenues for future work.

7.1 Summary of the Research

The primary goal of this research was to design, implement, and quantitatively evaluate a fully decentralized P2P game streaming framework. This objective was achieved through several key steps and resulted in tangible contributions:

First, the GameBeam architecture was designed and implemented, comprising a Unity SDK for host game integration and a browser-based client for guests. Crucially, this implementation embraced a fully decentralized model: the browser client code was distributed via IPFS, eliminating the need for a central web server, and peer discovery and WebRTC signaling were facilitated using public WebTorrent trackers, bypassing the requirement for a dedicated matchmaking server. The framework simplifies multiplayer development by adopting a local-multiplayer paradigm, where the host manages the game state and streams audio/video output while receiving inputs from guests via WebRTC data channels.

Second, to enable rigorous performance analysis, a dedicated, automated Performance Analysis Framework was developed. This framework facilitated the systematic execution of test scenarios with varying configurations (client count, resolution, frame rate, encoding methods) and collected detailed performance metrics from both the host (resource utilization) and clients (WebRTC statistics, end-to-end latency). This framework itself represents a reusable contribution to the research community, enabling reproducible evaluation of P2P streaming systems.

Third, a comprehensive quantitative performance evaluation was conducted using the analysis framework. This evaluation characterized the performance of the fully decentralized GameBeam system under controlled conditions, using two distinct Unity game applications. Key performance indicators were measured, including end-to-end latency, host resource utilization (CPU, GPU, memory), network bandwidth consumption, scalability with multiple clients, and WebRTC streaming quality metrics (packet loss, jitter).

The empirical results demonstrated the functional viability of the proposed decentral-

ized approach. Notably, low end-to-end latencies were achieved, with median values around 50-51 ms measured under baseline conditions (1 client, 1080p, 30 FPS, hardware encoding) between a residential host and cloud-based clients (Section 6.2, Table 6.2). The evaluation quantified the impact of various parameters: scalability was primarily limited by host GPU resources (particularly encoding load for multiple clients); network bandwidth scaled linearly with client count; hardware encoding significantly reduced host CPU load and improved latency compared to software encoding; and higher frame rates (60 FPS vs 30 FPS) correlated with slightly lower end-to-end latency despite increased resource demands. Streaming quality metrics indicated reliable delivery under the tested network conditions.

Finally, the core components, including the GameBeam Unity SDK [20] and the Performance Analysis Framework [22], have been prepared for release as open-source artifacts, fostering further research, development, and potential adoption.

7.2 Concluding Remarks on Feasibility and Potential

The findings of this research provide valuable insights into the feasibility and potential of fully decentralized P2P game streaming as implemented by GameBeam. The empirical data confirms that leveraging standard technologies like WebRTC, IPFS, and WebTorrent trackers can indeed create a functional, low-latency game streaming experience without reliance on centralized server infrastructure for the core streaming or signaling pathways. The achieved median end-to-end latency of approximately 50 ms under baseline conditions is promising and falls within acceptable ranges for many game genres, demonstrating the technical viability of the core concept.

The potential benefits of such an approach are significant. By eliminating the need for dedicated streaming servers, operational costs can be substantially reduced compared to traditional cloud gaming models. The installation-free nature, allowing guests to join instantly via a web browser, lowers the barrier to entry for players, mirroring a key advantage of cloud gaming. Furthermore, the architectural model, which abstracts network synchronization complexities behind a streaming interface, offers a potentially simpler development paradigm for creating multiplayer experiences, particularly appealing for developers accustomed to local multiplayer logic or those with limited networking expertise.

However, the evaluation also underscores critical practical limitations and trade-offs in the current fully decentralized model. The most significant constraint identified is the host resource requirement. Running the game, capturing audio/video, encoding multiple streams (especially without hardware acceleration), and managing P2P connections imposes a substantial load on the host machine's CPU, GPU, and network bandwidth. As demonstrated, scalability is directly bound by the host's capacity, limiting the practical number of concurrent guests, particularly for graphically demanding games (Section 6.3).

Furthermore, the reliance on public, decentralized infrastructure introduces variability and dependencies. The performance and availability of public WebTorrent trackers (used for signaling) and IPFS gateways (currently used for client fetching ease-of-use) directly impact the initial connection reliability and speed. While the multi-tracker fallback strategy mitigates single points of failure for signaling (Section 4.4), widespread infrastructure issues remain a potential concern. Similarly, NAT traversal remains a challenge; the reliance on STUN without TURN servers means direct P2P connections may not be

possible between all peers, potentially hindering connectivity in certain network environments (Section 4.5). Security and privacy aspects, such as IP address exposure between peers and the public nature of tracker announcements, also require careful consideration (Section 4.5).

In conclusion, while fully decentralized P2P game streaming via GameBeam shows considerable promise and demonstrated technical feasibility for achieving low-latency experiences, it is not presented as a universal replacement for existing multiplayer or cloud gaming architectures. Its potential is perhaps most pronounced in scenarios where the host possesses sufficient resources, the number of concurrent players is relatively small (e.g., cooperative games, small private sessions), and the benefits of zero-cost infrastructure and instant accessibility outweigh the challenges of host load, potential connection variability, and P2P security considerations. GameBeam serves as a valuable proof-of-concept and a foundation for future exploration in the domain of decentralized real-time interactive applications.

7.3 Future Work

The research presented in this thesis opens up several avenues for future investigation and development to enhance the capabilities, robustness, and usability of the GameBeam framework and the broader concept of decentralized P2P game streaming.

7.3.1 Investigating Alternative/Robust Decentralized Signaling Methods

The current reliance on public WebTorrent trackers for signaling, while demonstrating feasibility, presents potential bottlenecks related to reliability, scalability, and privacy (as

discussed in Section 4.5). Future work should investigate more robust and potentially more private decentralized signaling mechanisms. Exploring alternatives such as:

- Distributed Hash Tables (DHTs), similar to those used in BitTorrent for trackerless peer discovery (e.g., Kademlia [55]).
- Gossip protocols or other epidemic algorithms for broadcasting session information across a P2P network.
- Leveraging existing large-scale P2P networks or potentially federated signaling servers maintained by communities or developers.
- Implementing stronger authentication and encryption layers for signaling communication itself.

These investigations could lead to signaling solutions that offer better resilience against single points of failure and potentially enhanced privacy guarantees compared to public trackers.

7.3.2 Enhancing Security and Privacy Aspects

As highlighted in Section 4.5, the current implementation has several security and privacy considerations inherent to its P2P nature. Future work should prioritize enhancements in this area:

- Implementing peer authentication mechanisms before establishing the full WebRTC connection to prevent unauthorized join attempts.
- Investigating the selective use of TURN relays (part of the ICE framework [44]) not just for NAT traversal but also as a mechanism for IP address obfuscation between peers, acknowledging the trade-off of introducing a relay point.

- Developing standardized input validation and sanitization layers within the SDK to help developers mitigate potential risks associated with trusting client inputs over the data channel.
- Exploring methods to secure communication with signaling infrastructure (trackers or alternatives) beyond standard transport encryption.
- Providing clearer guidance to developers and users regarding the inherent privacy trade-offs (e.g., IP address visibility to peers).

7.3.3 Improving Latency via Data Channel Video Streaming

The quantitative analysis identified several components contributing to end-to-end latency (Figure 5.1), including encoding, network transit, jitter buffering, and decoding. While the measured median jitter buffer delay was around 40ms in baseline tests (Table 6.6), it represents a significant portion of the total latency budget. An important area for future work is to explore bypassing the standard WebRTC media pipeline's jitter buffer for video by transmitting encoded video frames directly over the 'RTCDataChannel' [14]. The motivation is to gain finer control over buffering strategy and potentially reduce the inherent delay introduced by the browser's default jitter buffer implementation. This approach, however, presents significant challenges:

- Implementing custom jitter buffering logic in JavaScript on the client-side.
- Handling packet loss and reordering, which are managed automatically by the RTP/SRTP [56] stack in the standard media pipeline but would need explicit handling over the data channel (potentially using SCTP's [57] features or application-level mechanisms).

- Potentially increased implementation complexity compared to using the built-in video channel.

Successfully implementing this could offer a pathway to further minimize latency, particularly in low-latency network conditions where large jitter buffers may be unnecessary.

7.3.4 Broader Testing Across Game Genres and Network Conditions

The current evaluation focused on two racing games under relatively stable network conditions between a residential host and cloud clients. To gain a more comprehensive understanding of GameBeam's applicability and robustness, future testing should encompass:

- **Diverse Game Genres:** Evaluating performance with genres highly sensitive to latency and input precision (e.g., first-person shooters, fighting games).
- **Varying Network Conditions:** Systematically testing under adverse network conditions, including simulated or real-world scenarios with higher packet loss, increased jitter, higher round-trip times, asymmetric bandwidth, and connections over wireless or mobile networks (e.g., LTE, 5G).
- **Different Hardware:** Evaluating performance on hosts with less powerful hardware to better understand resource constraints and minimum viable specifications.

This broader testing would help define the operational boundaries and identify specific scenarios where the decentralized P2P approach excels or struggles.

7.3.5 Developing Tooling/Guidelines for Developers

To facilitate adoption and ease the integration process for game developers, future work should focus on creating better tooling and comprehensive guidelines:

- **Debugging Tools:** Developing tools specifically aimed at diagnosing P2P connection issues (e.g., NAT traversal problems, signaling failures), which are notoriously difficult to debug in decentralized systems.
- **Performance Profiling:** Integrating performance monitoring overlays or APIs within the SDK to help developers understand the resource impact of GameBeam on the host and correlate it with stream quality.
- **Best Practice Guides:** Creating detailed documentation and guidelines on topics such as integrating the input system effectively ('GameBeamInput'), optimizing host game performance for streaming, configuring the custom audio system ('MultiAudioListener'), and understanding security implications.

7.3.6 Optimizing Streaming Quality and Latency Further

Beyond exploring data channel video streaming, further optimizations can be pursued within the existing framework:

- **Encoder Tuning:** Exposing more granular control over video encoder parameters (e.g., bitrate control modes, quantization parameters, codec profiles) to allow developers to fine-tune the trade-off between quality, bandwidth, and latency based on game content and network conditions.
- **Codec Exploration:** Investigating the integration and performance of newer, potentially more efficient video codecs like AV1 [58], considering browser and hardware support constraints.
- **Audio Pipeline Optimization:** Further profiling and optimizing the custom 'MultiAudioListener' system (Section 4.2) to minimize its CPU overhead while maintain-

ing accurate spatialization.

- **Client-Side Optimization:** Analyzing and potentially reducing client-side delays related to video decoding and rendering ($t_7 - t_6$ in Figure 5.1).

By addressing these areas, the GameBeam framework can be made more robust, performant, secure, and easier to use, further advancing the understanding and application of decentralized technologies for real-time interactive experiences. This thesis provides a solid foundation and empirical validation, encouraging continued exploration in this promising field.

8

Appendix

8.1 Additional Performance Data

Table 8.1: *Impact of Hardware Encoding (1 Client, 30 FPS, 1080p, Audio On, Game 1).*

Metric	Software (Mean ± 95% CI)	Hardware (Mean ± 95% CI)
Host CPU Usage (%)	4.29 ± 0.04	1.22 ± 0.03
Host GPU Usage (% SM)	3.80 ± 0.03	4.01 ± 0.02
Host Memory (MB)	697.53 ± 0.24	620.15 ± 0.25
Host GPU Power (W)	84.68 ± 0.13	85.83 ± 0.13
Host Sent Bandwidth (KB/s)	335.21 ± 1.77	326.81 ± 2.73
End-to-End Latency (ms)	52.71 ± 0.59	47.21 ± 0.58
Client RTT (ms)	15.32 ± 0.07	14.99 ± 0.08
Video Frames Received/s	29.99 ± 0.01	29.99 ± 0.02
Video Frames Dropped (Total)	0.00	0.00
Video Packet Loss (Total)	0.00	0.00
Video Jitter Buffer Delay (ms)	69.57 ± 0.60	42.75 ± 0.52
Video Inter-Frame Delay (ms)	33.35 ± 0.01	33.35 ± 0.01
Video Inter-Frame Delay (St. Dev.)	2.15 ± 0.09	4.39 ± 0.23

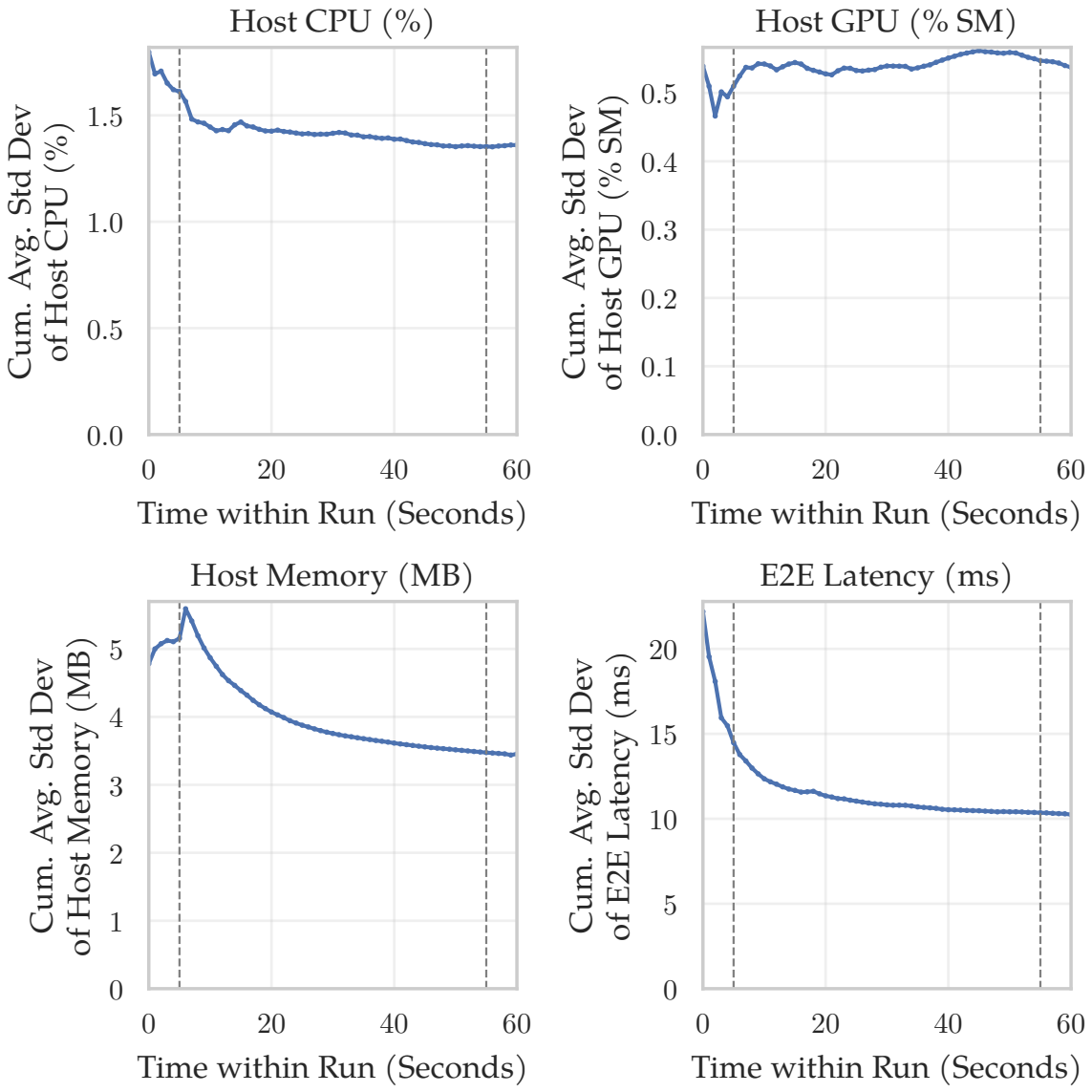

Figure 8.1: *Convergence of the cumulative average standard deviation for key performance metrics over the 60-second run duration. Dashed lines indicate the 5s start and end buffers excluded from steady-state analysis.*

References

[1] Kieren Mayers, Jonathan Koomey, Rebecca Hall, Maria Bauer, Chris France, and Amanda Webb. "The Carbon Footprint of Games Distribution". In: *Journal of Industrial Ecology* 19.3 (2015), pp. 402–415.

[2] Danielle Rourke and Ray Pastore. "Esports Equipment and Infrastructure". In: *Routledge Handbook of Esports*. Ed. by Seth E. Jenny, Nicolas Besombes, Tom Brock, Amanda C. Cote, and Tobias M. Scholz. Routledge, 2025.

[3] NVIDIA Corporation. *GeForce NOW*. 2024. URL: `https://www.nvidia.com/en-us/geforce-now/` (visited on 11/18/2024).

[4] Microsoft Corporation. *Xbox Cloud Gaming*. 2024. URL: `https://www.xbox.com/en-US/xbox-game-pass/cloud-gaming` (visited on 11/18/2024).

[5] Wei Cai, Ryan Shea, Chun-Ying Huang, Kuan-Ta Chen, Jing Liu, Victor C. M. Leung, and Chung-Heng Hsu. "A Survey on Cloud Gaming: Future of Computer Games". In: *IEEE Access* 4 (2016), pp. 7605–7620.

[6] Rajiv Tulsyan. "Introduction to Cloud Gaming". In: *International Journal for Research in Applied Science and Engineering Technology* (2024). IJRASET.

[7] Ryan Shea, Jiangchuan Liu, Edith C.-H. Ngai, and Yong Cui. "Cloud Gaming: Architecture and Performance". In: *IEEE Network* 27.4 (2013), pp. 16–21.

[8] Sawsan Ali Hamid, Yassine Boujelben, and Faouzi Zarai. "Enhancing Cloud Gaming Experience through Optimized Virtual Machine Placement: A Comprehensive Review". In: *Journal of Network and Systems Management* 32.4 (2024). Springer.

[9] Hanlin Sun. "Research on Latency Problems and Solutions in Cloud Game". In: *Journal of Physics: Conference Series*. Vol. 1314. 1. IOP Publishing. 2019, p. 012211.

[10] Henrique Souza Rossi, Niclas Ögren, Karan Mitra, Irina Cotanis, Christer Åhlund, and Per Johansson. "Subjective Quality of Experience Assessment in Mobile Cloud Games". In: *Proceedings of the 2022 IEEE Global Communications Conference*. 2022, pp. 1918–1923.

[11] Mark Claypool and Kajal Claypool. "Latency and player actions in online games". In: *Commun. ACM* 49.11 (Nov. 2006), pp. 40–45. ISSN: 0001-0782. DOI: 10.1145/1167838.1167860.

[12] Joshua Glazer and Sanjay Madhav. *Multiplayer Game Programming: Architecting Networked Games*. Addison-Wesley Professional, 2015.

[13] Glenn Fiedler. *Networked Physics (GDC 2011)*. Presentation Slides. Accessed: 2025-04-01. 2011. URL: https://www.slideshare.net/BarrelRoll/gdc-networked-physics-2011.

[14] Adam Bergkvist, Daniel C. Burnett, Cullen Jennings, and Anant Narayanan. *WebRTC 1.0: Real-time Communication Between Browsers*. W3C Recommendation. W3C, Jan. 2021. URL: https://www.w3.org/TR/webrtc/ (visited on 03/31/2025).

[15] Shyam Sunder Saini and Lalit Sen Sharma. "Performance Evaluation of WebRTC-Based Video Conferencing: A Comprehensive Analysis". In: *Journal of Advanced Zoology* 44.S7 (2023), pp. 322–330.

[16] Diomidis Spinellis and Stephanos Androutsellis-Theotokis. "Peer-to-peer systems". In: *Communications of the ACM* 47.1 (2004), pp. 31–36.

[17] Protocol Labs. *InterPlanetary File System (IPFS)*. 2015. URL: https://ipfs.tech/ (visited on 03/31/2025).

[18] Feross Aboukhadijeh and WebTorrent Community. *WebTorrent*. 2013. URL: https://webtorrent.io/ (visited on 03/31/2025).

[19] Unity Technologies. *Unity Engine*. 2005. URL: https://unity.com/ (visited on 03/31/2025).

[20] Cumhur Onat. *GameBeam SDK*. 2025. URL: https://github.com/cumhuronat/gamebeam_sdk (visited on 03/26/2025).

[21] Cumhur Onat. *GameBeam Client*. 2025. URL: https://github.com/cumhuronat/gamebeam_client (visited on 03/26/2025).

[22] Cumhur Onat. *GameBeam Performance Analysis Framework*. 2025. URL: https://github.com/cumhuronat/gamebeam_analysis (visited on 03/26/2025).

[23] Bryan Ford, Pyda Srisuresh, and Dan Kegel. "Peer-to-Peer Communication Across Network Address Translators". In: *ArXiv* abs/cs/0603074 (2005).

[24] Huynh Cong Phuoc, Ray Hunt, and Andrew McKenzie. "NAT traversal techniques in peer-to-peer networks". In: *Proceedings of the New Zealand Computer Science Research Student Conference (NZCSRSC)*. 2008.

[25] Juan Benet. "Ipfs-content addressed, versioned, p2p file system". In: *arXiv preprint arXiv:1407.3561* (2014).

[26] Dennis Trautwein, Aravindh Raman, Gareth Tyson, Ignacio Castro, Will Scott, Moritz Schubotz, Bela Gipp, and Yiannis Psaras. "Design and evaluation of IPFS: a storage layer for the decentralized web". In: *Proceedings of the ACM SIGCOMM Conference*. SIGCOMM '22. Association for Computing Machinery, 2022, pp. 739–752. ISBN: 9781450394208. DOI: 10.1145/3544216.3544232.

[27] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. "The Bittorrent p2p file-sharing system: Measurements and analysis". In: *Peer-to-Peer Systems IV: 4th International*

Workshop, IPTPS 2005, Ithaca, NY, USA, February 24-25, 2005. Revised Selected Papers 4. Springer. 2005, pp. 205–216.

[28] Graham Morgan. "Challenges of Online Game Development: A Review". In: *Simulation & Gaming* 40.5 (2009), pp. 688–710. DOI: 10.1177/1046878109340295.

[29] Mark Claypool and David Finkel. "The effects of latency on player performance in cloud-based games". In: *Proceedings of the 13th Annual Workshop on Network and Systems Support for Games*. 2014, pp. 1–6. DOI: 10.1109/NetGames.2014.7008964.

[30] Chun-Ying Huang, Cheng-Hsin Hsu, Yu-Chun Chang, and Kuan-Ta Chen. "GamingAnywhere: an open cloud gaming system". In: *Proceedings of the 4th ACM Multimedia Systems Conference*. MMSys '13. Oslo, Norway: Association for Computing Machinery, 2013, pp. 36–47. ISBN: 9781450318945. DOI: 10.1145/2483977.2483981. URL: https://doi.org/10.1145/2483977.2483981.

[31] Leslie S. Liu, Roger Zimmermann, Baoxuan Xiao, and Jon Christen. "PartyPeer: a P2P massively multiplayer online game". In: *Proceedings of the 14th ACM International Conference on Multimedia*. MM '06. Santa Barbara, CA, USA: Association for Computing Machinery, 2006, pp. 507–508. ISBN: 1595934472. DOI: 10.1145/1180639.1180748. URL: https://doi.org/10.1145/1180639.1180748.

[32] Shakeel Ahmad, Christos Bouras, Eliya Buyukkaya, Raouf Hamzaoui, Andreas Papazois, Alex Shani, Gwendal Simon, and Fen Zhou. "Peer-to-peer live streaming for massively multiplayer online games". In: *Proceedings of the 12th International Conference on Peer-to-Peer Computing (P2P)*. 2012, pp. 67–68. DOI: 10.1109/P2P.2012.6335814.

[33] Bogdan-Ioan Oros and Victor Ioan Bâcu. "RenderLink remote rendering platform for computer games: a WebRTC solution for streaming computer games". In: *Proceedings of the 16th International Conference on Intelligent Computer Communication and Processing (ICCP)*. 2020, pp. 555–561. DOI: 10.1109/ICCP51029.2020.9266231.

[34] Rodrigo Campos Borges, Marcelo de Gomensoro Malheiros, Cleo Zanella Billa, Marcelo Rita Pias, and Alessandro de Lima Bicho. "An open-source framework using WebRTC for online multiplayer gaming". In: *Proceedings of the 22nd Brazilian Symposium on Games and Digital Entertainment*. SBGames '23. Rio Grande (RS), Brazil: Association for Computing Machinery, 2024, pp. 143–150. ISBN: 9798400716270. DOI: 10.1145/3631085.3631238. URL: https://doi.org/10.1145/3631085.3631238.

[35] Valve Corporation. *SteamLink*. 2015. URL: https://store.steampowered.com/steamlink/ (visited on 11/24/2024).

[36] Open-Source Community. *Sunshine*. 2020. URL: https://github.com/loki-47-6F-64/sunshine (visited on 11/24/2024).

[37] Parsec Gaming. *Parsec*. 2016. URL: https://parsec.app/ (visited on 11/24/2024).

[38] Microsoft. *TypeScript*. 2012. URL: https://github.com/microsoft/TypeScript (visited on 03/26/2025).

[39] Ian Hickson, Robin Berjon, Steve Faulkner, Travis Leithead, Edward O'Connor Doyle Navara, Silvia Pfeiffer, and Anne van Kesteren Sicking. *HTML5*. W3C Recommendation. W3C, Oct. 2014. URL: https://www.w3.org/TR/html5/ (visited on 04/01/2025).

[40] W3C CSS Working Group. *Cascading Style Sheets Level 3 (CSS3) - Overview*. W3C Current Work. W3C, 2024. URL: https://www.w3.org/Style/CSS/current-work (visited on 04/03/2025).

[41] I. Fette and A. Melnikov. *The WebSocket Protocol*. RFC 6455. Internet Engineering Task Force (IETF), Dec. 2011. URL: https://tools.ietf.org/html/rfc6455 (visited on 04/03/2025).

[42] sta.blockhead. *WebSocketSharp*. 2012. URL: https://github.com/sta/websocket-sharp (visited on 12/22/2024).

[43] M. Petit-Huguenin, G. Salgueiro, M. Jones P. andISTRY, and M. Lepinski. *Session Traversal Utilities for NAT (STUN)*. RFC 8489. Internet Engineering Task Force (IETF), Feb. 2020. URL: https://tools.ietf.org/html/rfc8489 (visited on 02/07/2025).

[44] A. Keranen, C. Holmberg, and J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. Internet Engineering Task Force (IETF), Oct. 2018. URL: https://tools.ietf.org/html/rfc8445 (visited on 02/03/2025).

[45] NVIDIA Corporation. *NVIDIA Linux Driver / Windows Driver and NVIDIA System Management Interface (nvidia-smi)*. URL: https://developer.nvidia.com/nvidia-system-management-interface (visited on 03/31/2025).

[46] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. Internet Engineering Task Force (IETF), Aug. 2018. URL: https://tools.ietf.org/html/rfc8446 (visited on 01/19/2025).

[47] OASIS Message Queuing Telemetry Transport (MQTT) Technical Committee. *MQTT Version 5.0*. Tech. rep. OASIS Standard, Mar. 2019. URL: https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html (visited on 01/19/2025).

[48] Mozilla. *Fullscreen API*. API Specification. Mozilla, 2025. URL: https://developer.mozilla.org/en-US/docs/Web/API/Fullscreen_API#api.document.fullscreenelement (visited on 02/08/2025).

[49] Open-Source Community. *Node.js*. 2009. URL: https://github.com/nodejs/node (visited on 03/26/2025).

[50] Microsoft Corporation. *Windows Performance Monitor (perfmon.exe) and typeperf.exe*. Core components of Microsoft Windows Operating System. URL: https://learn.microsoft.

com/en-us/windows-server/administration/windows-commands/typeperf (visited on 03/31/2025).

[51] Google Chrome Team. *Puppeteer*. 2017. URL: https://pptr.dev/ (visited on 03/28/2025).

[52] Testable Inc. *Testable.io*. 2015. URL: https://testable.io/ (visited on 03/28/2025).

[53] Drizzle Team. *Drizzle ORM*. 2022. URL: https://orm.drizzle.team/ (visited on 03/31/2025).

[54] The PostgreSQL Global Development Group. *PostgreSQL*. 1996. URL: https://www.postgresql.org/ (visited on 03/30/2025).

[55] Petar Maymounkov and David Mazières. "Kademlia: A Peer-to-peer Information System Based on the XOR Metric". In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. Springer-Verlag, 2002, pp. 53–65. ISBN: 3-540-44179-4.

[56] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. *The Secure Real-time Transport Protocol (SRTP)*. RFC 3711. Internet Engineering Task Force (IETF), Mar. 2004. URL: https://tools.ietf.org/html/rfc3711 (visited on 03/11/2025).

[57] R. Stewart, M. Tüxen, and E. Nilsen-Nygaard. *Stream Control Transmission Protocol (SCTP)*. RFC 9260. Internet Engineering Task Force (IETF), June 2022. URL: https://tools.ietf.org/html/rfc9260 (visited on 02/21/2025).

[58] Alliance for Open Media. *AV1 Bitstream & Decoding Process Specification*. Web Document. 2024. URL: https://aomediacodec.github.io/av1-spec/av1-spec.pdf (visited on 03/10/2025).